A Websense® White Paper

Win32 Portable Executable Packing Uncovered

ESSENTIAL INFORMATION PROTECTION™

Table of Contents

websense
ESSENTIAL INFORMATION PROTECTION™

Introduction

This paper introduces Win32 Portable Executable (PE) packing from a technical perspective. This includes PE file manipulation, compression, obfuscation, anti-dumping, import protection, and more. The paper describes various protection techniques, and presents a brief history of packers. Note that the most advanced techniques are found in commercial protection systems, and therefore are not presented here.

This paper provides enough information to understand the inner workings of executable packers: most packers are based on what is described here. Almost all custom packers (which means real packers, not loaders) seen in malware are based on the packing theory presented in this document.

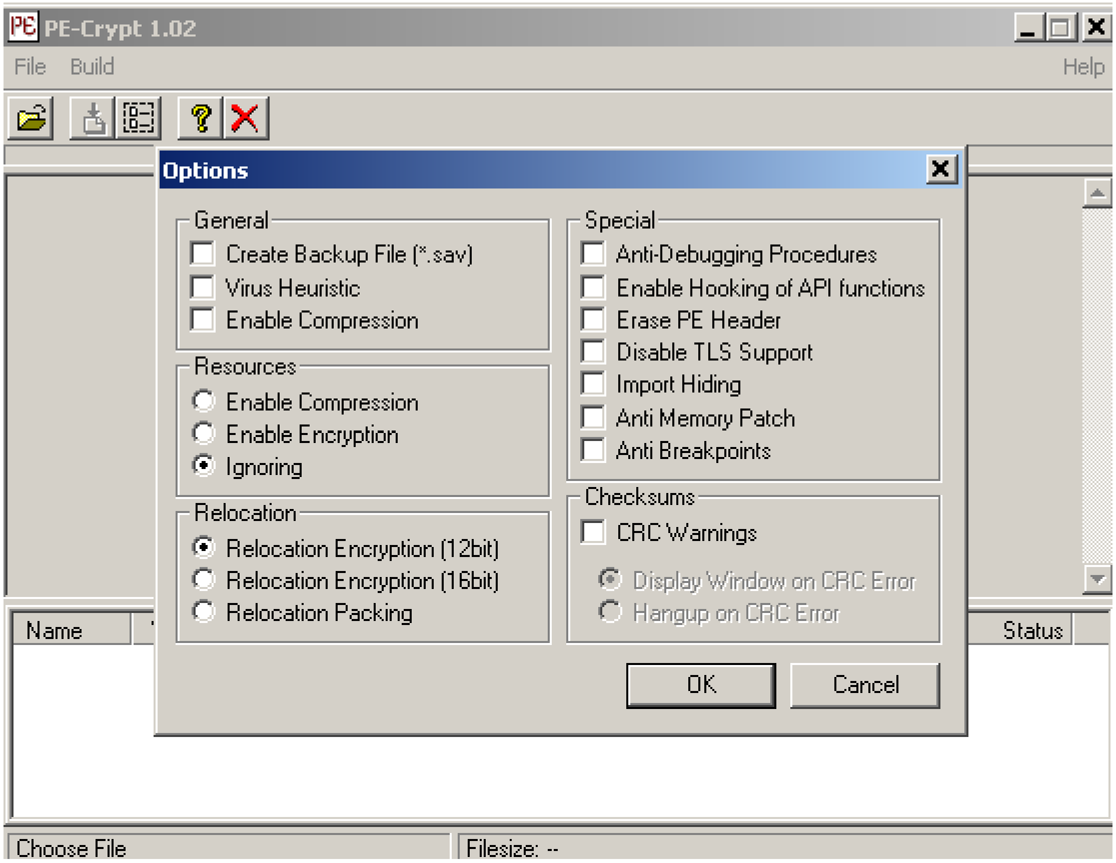
Packer History

The history below is by no means authoritative. It has been compiled from the author's personal recollections, and from the *.nfo* files found with the packers. Nor is it an exhaustive list of packers: only big names are listed here, with release dates where available.

The first public PE packer was introduced on December 23 1997 by Stone, and was named Stone PE Crypter 1.0. It was a very basic packer supporting both Windows® 95 and NT. DLLs were not supported until January 1998.

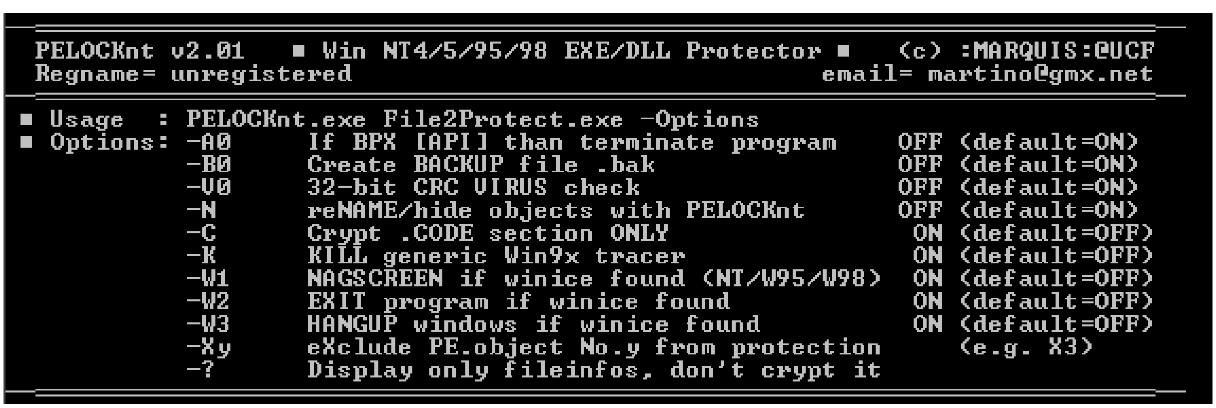
PECRYPT32 1.01 was published on January 22 1998. This seems to be the second packer publicly available, but considering the amount of features, it was probably created before Stone's. Here is a copy from the original .nfo file:

> *Code, Data, Resource, Relocation, Import Encryption.*
> *Code, Data, Resource, Relocation, Import Compression.*
> *Enhanced Relocation Loader.*
> *Anti-Debugging compatible with WINDOWS NT , WINDOWS 95 and WINDOWS 98.*
> *Dynamic Link Library support (DLL files).*
> *Routines against memory patches / loaders.*
> *Anti-API-Breakpoint routines.*
> *A (hopefully) working Import-Loader.*
> *Anti-Unpacking procedures.*
> *Lame Heuristic Virus File Check.*

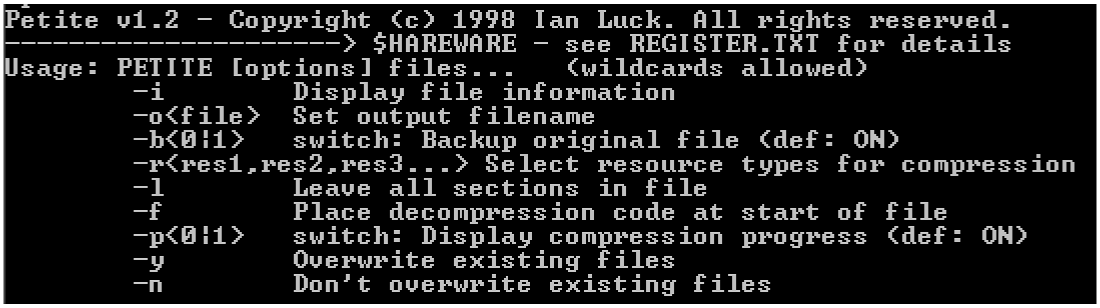


Although this was one of the very first packers, it was already able to encrypt and compress code, data , resources, relocations, and even imports. It was already using anti-debugging techniques working on both 9x and NT, detecting breakpoints on API functions, anti-memory patches, and CRC. It also used a graphical user interface (GUI).

PELOCKnt 2.01 was released on April 7 1998 with anti-hooking, anti-generic unpacker code, anti-trace, anti-dump and more.



Petite 1.0 was introduced on May 22 1998 , but was much more basic in terms of features: it was just a simple PE packer.



```
Petite v1.2 - Copyright (c) 1998 Ian Luck. All rights reserved.
---------------------> $HAREWARE - see REGISTER.TXT for details
Usage: PETITE [options] files...    (wildcards allowed)
        -i          Display file information
        -o<file>    Set output filename
        -b<0|1>     switch: Backup original file (def: ON)
        -r<res1,res2,res3...> Select resource types for compression
        -l          Leave all sections in file
        -f          Place decompression code at start of file
        -p<0|1>     switch: Display compression progress (def: ON)
        -y          Overwrite existing files
        -n          Don't overwrite existing files
```

Below is a summary of packer history from this point up to 2003.

1998
BJFNT 1.2rc – May 1998 Neolite 1.01 – September 5 1998 VGCrypt PE Encryptor 0.40 – November, 26 1998 v0.40. PE Prot – December 17 1998 UPX 0.50 – January 3 1999 (The first version of UPX-supporting PE files was released 1 year after the first public PE packer).

1999
Armadillo 1.0 – January 15 1999 PE Diminisher v0.1 – Crappy PE Packer, (C) 1999 Teraphy LameCrypt 1999 – June 27 1999 PECompact v0.91 beta Asprotect

PEX 0.99 by bart – August 10 2000 Krypton 2000 by Yado Armadillo 2 – June 11 2001 FSG 1.0 by Dulek – January 14 2002 Armadillo 3 – April 4 2003.

As stated earlier, this is not an exhaustive list of packers. Many more packers were created during these years. It is interesting to note that one of the very first packers already had advanced features and that the packing and protection of PE files was already mastered from the very beginning.

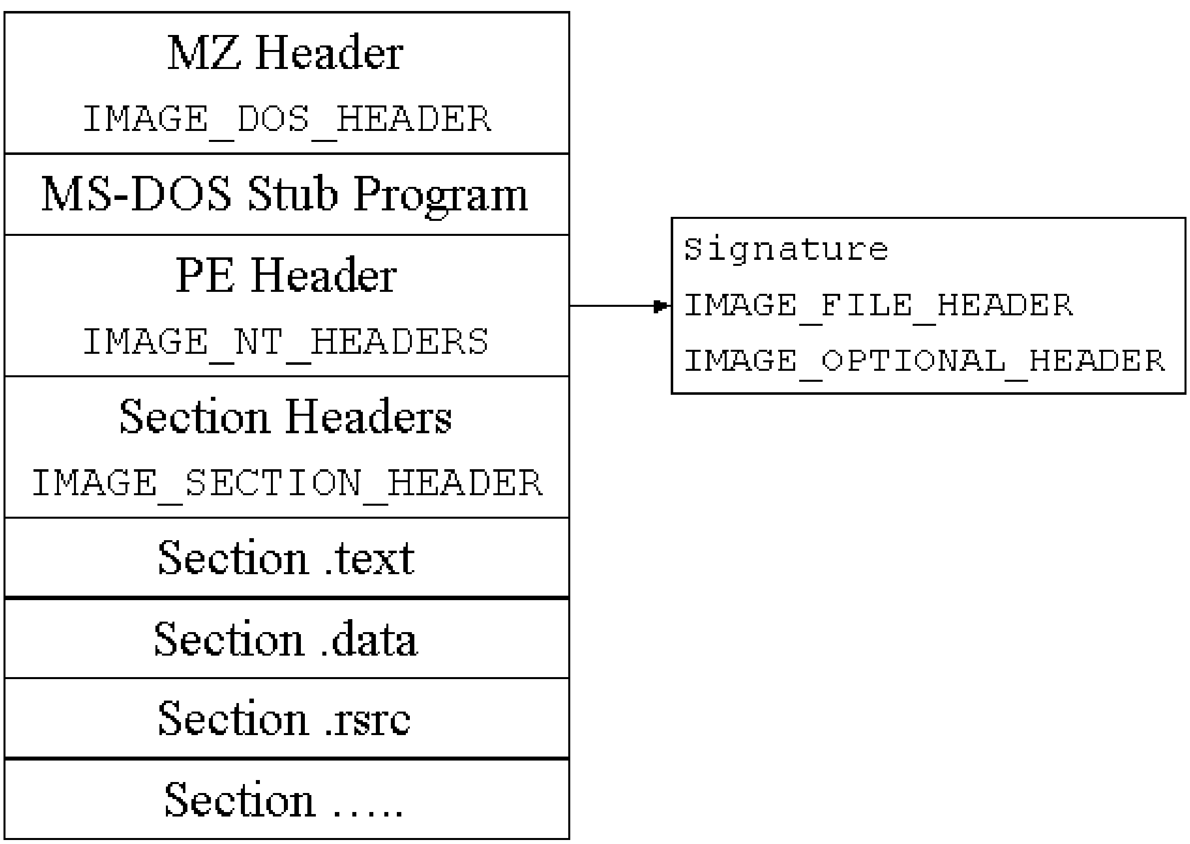
How Packers Work

A good understanding of the Portable Executable (PE) file format is required to follow the details in this paper: this is described in this section.

Portable Executable Format

Portable Executable (PE) format is the file format of executables and DLLs used in 32-bit and 64-bit (PE32+ or PE+) versions of the Microsoft Windows operating system. The term “portable” indicates that the format can be used on numerous architectures, such as x86, IA-32, ARM, ALPHA, and others.

PE files consist of a number of headers and sections that tell the Windows Loader how to map the file into memory. Every section in a PE file is mapped into a different region of memory (and therefore must be page-aligned; this is the Section Alignment in the PE header) with different permissions. To create smaller files, the sections are aligned differently on disk (File Alignment). Windows uses this information to load the sections appropriately.



The MZ header, and most importantly the MS-DOS stub program, are there for backward compatibility with MS-DOS. If you run a Win32 executable under DOS, the MS-DOS stub program is executed, displaying “This program cannot be run in DOS mode” on screen.

In the IMAGE_DOS_HEADER, only two members of the structure are important here:

- *e_magic*: Contains the “MZ” letters.
- *e_lfanew*: Contains the offset of the PE Header.

The PE header follows; more precisely, the *IMAGE_NT_HEADERS* structure:

```
IMAGE_NT_HEADERS STRUCT
    Signature dd ?              // PE\00
    FileHeader IMAGE_FILE_HEADER <>
    OptionalHeader IMAGE_OPTIONAL_HEADER32 <>
IMAGE_NT_HEADERS ENDS
```

The *IMAGE_NT_HEADERS* structure holds important information such as the PE\00 signature (an executable without this value will never be executed) as well as other interesting structures, as described below.

IMAGE_FILE_HEADER structure

```
IMAGE_FILE_HEADER STRUCT
    Machine WORD              // Architecture the file was made for.
    NumberOfSections WORD     // Number of sections in the PE file.
    TimeDateStamp dd          // Compilation time (can be null or fake)
    PointerToSymbolTable dd   // Reserved
    NumberOfSymbols dd        // Reserved
    SizeOfOptionalHeader WORD // Size of Optional Header (important)
    Characteristics WORD      // Information on the file (DLL, EXE etc)
IMAGE_FILE_HEADER ENDS
```

Here is a dump of the Notepad executable (on a French Windows XP machine):

```
->File Header
   Machine:               0x014C  (I386)
   NumberOfSections:      0x0003
   TimeDateStamp:         0x3B7D840D  (GMT: Fri Aug 17 20:52:29 2001)
   PointerToSymbolTable:  0x00000000
   NumberOfSymbols:       0x00000000
   SizeOfOptionalHeader:  0x00E0
   Characteristics:       0x010F
                          (RELOCS_STRIPPED)
                          (EXECUTABLE_IMAGE)
                          (LINE_NUMS_STRIPPED)
                          (LOCAL_SYMS_STRIPPED)
                          (32BIT_MACHINE)
```

IMAGE_OPTIONAL_HEADER32 structure

This is the most important structure, because it holds a great deal of useful information for packing Windows executables. As it is quite large, only the most important fields are presented as background for understanding this paper. Before enumerating the structure, readers not familiar with the PE format and the PE Loader should review two important notions:

- **ImageBase:** Address where the PE image will be mapped in memory (unless there is a relocation).
- **Relative Virtual Address (RVA):** This is an address relative to the ImageBase. It is like an offset relative to the ImageBase. It is not a file offsets, which is relative to the start of the file on disk.

In order to compute a Virtual Address (VA) when its RVA is known, you simply add the ImageBase to it.
VA = RVA + ImageBase.

IMAGE_OPTIONAL_HEADER32:

Field	Description
AddressOfEntryPoint	RVA of the entry point
ImageBase	Where to map the PE image. This is usually 0x400000 in Windows executables.
SectionAlignment	Sections in memory are page-aligned, and therefore the RVA of each section must be a multiple of this value. Padding is used for alignment.
FileAlignment	The sections on disk must be aligned to the FileAlignment value. Padding is used for alignment. This is usually smaller than the SectionAlignment, unless the file has been dumped from memory.
SizeOfImage	Size of the PE image in memory. (all sections + headers + padding)
SizeOfHeaders	Size of all headers. Includes every byte from the first header until the start of the first section on disk. It can be used as the first section raw offset.
Subsystem	Gives information about the subsystem, for example Windows GUI, Windows Console, Windows CE, XBOX.

DataDirectory	Array of IMAGE_DATA_DIRECTORY structures. Each structure has the RVA of important structures, and their size. From there, you can get the RVA of the Import Table, Export Table, Relocation Table, and so on.

Partial dump of the IMAGE_OPTIONAL_HEADER (Example from Notepad.exe):

```
->Optional Header
   AddressOfEntryPoint:          0x00006AE0
   ImageBase:                    0x01000000
   SectionAlignment:             0x00001000
   FileAlignment:                0x00000200
   SizeOfImage:                  0x00013000
   SizeOfHeaders:                0x00000400
   Subsystem:                    0x0002  (WINDOWS_GUI)

   DataDirectory (16)            RVA        Size
   -------------                 ---------- ----------
   ExportTable                   0x00000000 0x00000000
   ImportTable                   0x00006D20 0x000000C8  (".text")
   Resource                      0x0000A000 0x00008E14  (".rsrc")
   Exception                     0x00000000 0x00000000
   Security                      0x00000000 0x00000000
   Relocation                    0x00000000 0x00000000
   Debug                         0x00001340 0x0000001C  (".text")
   Copyright                     0x00000000 0x00000000
   GlobalPtr                     0x00000000 0x00000000
   TLSTable                      0x00000000 0x00000000
   LoadConfig                    0x00000000 0x00000000
   BoundImport                   0x00000258 0x000000D0
   IAT                           0x00001000 0x00000324  (".text")
   DelayImport                   0x00000000 0x00000000
   COM                           0x00000000 0x00000000
   Reserved                      0x00000000 0x00000000
```

IMAGE_SECTION_HEADER Structure

To conclude this quick summary of the PE file format, after the PE header comes the IMAGE_SECTION_HEADER structure. Every section of a PE file is defined by this structure. You can learn where the section starts in memory and on disk, and the memory permission on each section. Only the most important fields are listed below.

Field	Description
Name	Name of the section. 8 characters maximum. Note that this is not a C-String, therefore there is no null byte at the end of the section name.
Virtual Size	Size of the section in memory, padded to the value of *SectionAlignment* as described earlier.
Virtual Address	RVA of the start of the section (memory)
SizeofRawData	Size of the section on disk
PointerToRawData	Offset of the start of the section (on disk)
Characteristics	Characteristics of the section: code, data, uninitialized data, rights (write, read, execute) and so on.

Dump of an IMAGE_SECTION_HEADER structure:

```
->Section Header Table
   1. item:
    Name:                   .text
    VirtualSize:            0x00017830
    VirtualAddress:         0x00001000
    SizeOfRawData:          0x00017A00
    PointerToRawData:       0x00000400
    PointerToRelocations:   0x00000000
    PointerToLinenumbers:   0x00000000
    NumberOfRelocations:    0x0000
    NumberOfLinenumbers:    0x0000
    Characteristics:        0x60000020
    (CODE, EXECUTE, READ)
```

To comprehend how executable protections work, it is critical to have a full understanding of the PE file format. For the full documentation, see [PE-DOC] in the References section.

Common Modifications of PE images Done by Packers

While protecting a Windows executable, packers perform various modifications on PE files, such as adding a new IMAGE_SECTION_HEADER in the SECTION_HEADER_TABLE (in other words, adding a new section to the file with the appropriate characteristics), updating the Entry Point RVA, and updating the SizeOfImage.

Section Addition

The vast majority of protectors and packers add one or more sections to the software they are protecting or packing. The new section holds the *loader* of the protector/packer, in charge of the decompression and decryption of the sections. It also performs some tasks usually performed by the Windows PE Loader, such as handling the import table of the protected executable. The addition of a new section is accomplished in two steps:

1. First, the PE header is modified, the NumberOfSections field is incremented, and a new IMAGE_SECTION_HEADER is added to the SECTION HEADER TABLE.
2. This structure is then filled with various information, such as the RVA of the new section, its virtual size, the raw offset, the size of the section on disk, and its characteristics.

Because the loader is going to be executed, the section characteristics are usually set to Executable, Readable, and Writable. Indeed, many protectors and packers update and decrypt themselves, and thus require write access.

Once the headers are modified, a packer increases the size of the file. Starting from the raw offset of the section (the end of the file if we add a section), an amount of bytes matching the raw size of the new section is inserted into the PE file. The section is now created, and is ready to hold the packer/protector loader.

A packer must also modify the SizeOfImage field in the PE header. The file grew up on disk, but in order to exist in memory, the headers must be modified accordingly. To do that, the virtual size of the section is added to the old SizeOfImage to compute the new size of the PE image in memory.

Entry Point modification

In the IMAGE_OPTIONAL_HEADER, the *EntryPoint* field holds the RVA of the entry point of a PE executable. The entry point is the address of the first instruction to execute when an executable is run. (Note that in some cases, such as TLS, it is possible to execute code before the entry point.) When protecting an executable, packers first save the RVA of the entry point, and then modify it to the start of the loader, in the packer section. This is why the added section must have the Execute characteristic. The packed application will start with the loader and will eventually execute the original entry point. The next section of this paper goes into more depth about the loader.

All of these modifications on the PE file format are necessary in order to inject code into any PE image, yet keep it executable by the operating system. This section does not cover every possible or necessary modification; only the most important ones are included to aid comprehension of executable packing.

The Loader

Every packer/protector injects a loader inside the file it is wrapping. The loader's role is to uncompress and decrypt the executable in memory, and to load the imports of the original application (mimicking the Windows PE Loader, because the original import table has been compressed,encrypted or destroyed).

Packers usually add a replacement import table to the packed executable. It is usually small and typically imports only a few specific functions, so it will run on every version of the operating system. Depending on the Windows version, you need certain conditions, such as specific, imported DLLs, in order to have a valid Windows executable. The most common imported functions are LoadLibraryA and GetProcAddress. These two functions are used by the packers to mimic the PE Loader, when it needs to resolve the application import table.

However, it is not necessary to use them, and many packers avoid using these two API functions, in order to be a little less obvious to analysts. Using Windows API functions as little as possible is always a good thing, because they offer an entry point into the loader, for an attacker trying to unpack the protection. A few of the tricks used are presented later in this paper.

Often, packers/protectors have a self-decrypting loader, and some of them can have many layers. Think of it as a set of nested Russian dolls. The first layer decrypts the second one, which decrypts the third one, and so on. Eventually, the loader is totally decrypted, and it starts doing its job.

Each major task can be wrapped by self-decrypting layers, and erased upon execution. Usually, anti-debugging techniques are used in the loader to prevent, or at least slow down, the analysis.[FERRIE09] By self-encrypting, the packer prevents easy attack and tries to hide the jump to the original entry point.

It is important to make the analysis of the loader as hard as possible, to slow down reverse engineering. It is in the loader that the majority of the protections are implemented. Some of the tricks used are described in the next part of this paper. When analyzing a packer, you usually have to locate the Import Table handling, and the jump to the original entry point. This is enough for most packers, but protectors do have other tricks available.

Usually, the loader is written in pure assembly language, because of its small size, and also for the infinite code obfuscation possibilities.

Here is a list of tasks executed by the loader. (This list obviously depends on the packer, but most of them have similar behavior.)

- Self decryption of the loader
- Decompression and decryption of the sections in memory
- Relocation handling for DLLs
- Import table handling (the part of the loader that mimics the Windows PE Loader and fills the Import Address Table)
- Jump to the original entry point (saved at packing time)

Compression

The vast majority of packers use the aPLib library for compression. [APLIB] This section explains the most common way used by packers to handle the compression. This is yet another PE file modification that needs to be done by the packer while it is packing an executable.

Many packers change the *RAWSIZE* of each packed section to 0. The size in memory remains unchanged, because the program still has to execute normally and be unpacked at its original location. If the *RAWSIZE* is null, it means the section is non-existent on disk.

Packers usually compress the contents of the section before they delete it from the file. The compressed section is usually appended at the end of the loader, or somewhere in the loader, for runtime unpacking. Once complete, the original sections are completely deleted on disk, and are present only in their packed form in the packed program.

Compressed sections usually have the *UNINITIALIZED DATA* flag enabled (because of the null size on disk). The loader takes the compressed sections and unpacks them to their original memory locations.

Example: The UPX Packer

UPX0 Section dump:

```
Name:                   UPX0
VirtualSize:            0x0001D000
VirtualAddress:         0x00001000
SizeOfRawData:          0x00000000
PointerToRawData:       0x00000400
PointerToRelocations:   0x00000000
PointerToLinenumbers:   0x00000000
NumberOfRelocations:    0x0000
NumberOfLinenumbers:    0x0000
Characteristics:        0xE0000080
(UNINITIALIZED_DATA, EXECUTE, READ, WRITE)
```

In the example above, you can see the UNINITIALIZED_DATA in the Characteristics, as well as a SizeOfRawData (RAWSIZE) of 0. This means that the section takes 0 bytes on disk; that is, the section does not exist in the file.

Interestingly, the PointerToRawData (Offset on disk) is 0x400, which is also the start of the following section on disk, as you can see with the next dump:

UPX1 Section dump:

```
Name:                   UPX1
VirtualSize:            0x00016000
VirtualAddress:         0x0001E000
SizeOfRawData:          0x00015600
PointerToRawData:       0x00000400
PointerToRelocations:   0x00000000
PointerToLinenumbers:   0x00000000
NumberOfRelocations:    0x0000
NumberOfLinenumbers:    0x0000
Characteristics:        0xE0000040
(INITIALIZED_DATA, EXECUTE, READ, WRITE)
```

In the dump above, you can see that the UPX1 section starts at offset 0x400, but its *RAWSIZE* is not null, meaning that the section really does exist on disk. The first one is therefore a compressed section.

Protection Techniques

To pretect against reverse engineering, packers and protectors try to slow down attackers as long as possible. Here is a non-exhaustive summary of techniques you may encounter.

Encryption Layers

To protect applications against analysis, packers and protectors often use encryption layers. Usually, in a manner similar to viruses, polymorphic engines are employed to generate a different crypt/decrypt algorithm for each protected application.

Two different kinds of encryption are usually observed:

Loader encryption

The protection code resides in the loader. To protect against static analysis and modifications of the underlying code and protections, the loader is encrypted, usually many times. Therefore, it is not possible to directly patch the code underneath.

The loader can be split into many parts, each of them encrypted by many layers.

Application encryption

Like the loader, the application is also encrypted to prevent disassembly and modifications.

Although the application can be encrypted with many layers, most of the time it has only one or two layers. On the other hand, the loader may vary from a couple of layers to a few hundred. After parts of the loader have been executed, they can be re-encrypted or destroyed, so that a fully decrypted loader is never in memory at any time.

Example of a loader layout:

```
Loader Start:
Layer 1 Decryption
Layer 2 Decryption
        Start of decrypted loader
Layer 3 Decryption
        Suite du loader
Layer 4 Decryption
        Application Decryption 1
Layer 5 Decryption
Layer 6 Decryption
        Application Decryption 2

     … And so on …
```

Similar to stacking Russian dolls, every decryption routine is wrapped underneath another. Analyzing the full loader, requires the analysis of every layer, and going through the repetitive process of checking each encryption layer. To make things more tedious, those layers use obfuscation.

Obfuscation Techniques

One of the first tricks that appeared in packers was code obfuscation, designed to slow down analysis. Techniques are used to scramble the code, making it hard to read, follow, and debug. This section describes some of the most common techniques, especially those used since the beginning of software packing.

Bogus bytes between instructions

This technique began with bogus bytes inserted after jumps and calls, in order to fool “dumb” disassembly engines.

Example:

```
jmp over_thrash

       Db 0E8h   ; Bogus byte. This is never executed, but 0xE8 is the start byte of a CALL
                ; Some disassemblers will assemble this bogus byte to call, and the
                 ;  disassembly shows up as invalid in the disassembler.

over_thrash:

       call sub_function  ; Real code
```

Back in the old days, Soft ICE would constantly change the disassembly as an analyst single-stepped through the sort of thrash code shown above. It was tiresome to follow the real code, because it kept moving under the analyst’s eyes. Nowadays those basic techniques are useless, because modern reverse-engineering tools are not tricked by such simple devices.

Macros

The next step for obfuscation and packers was macros (note that the first obfuscations above could be made with macros, but that would not make much sense, since the obfuscations were quite short). By creating special macros that did nothing, yet confused disassembly engines, and by using them in between real instructions, it was possible to scramble the code totally, making it unreadable in a debugger or disassembler without user interaction.

They were typically used like this:

```
Macro
  Real code
Macro
  Real code
Macro
Macro
  Real code
```

```
Macro
  Real code
Macro


And so on
```

The macros scrambled the code and confused the tools and/or attackers. Usually, an analyst would see an invalid disassembly.

Here is an example of such a macro in action, written by the author about 5 years ago:

```
CODE:00401000
CODE:00401000                 public start
CODE:00401000 start:
CODE:00401000                 test    eax, eax
CODE:00401002
CODE:00401002 loc_401002:                             ; CODE XREF: CODE:00401040↓j
CODE:00401002                 jo      short loc_401011
CODE:00401004                 rep jnz short loc_401033
CODE:00401007
CODE:00401007 loc_401007:                             ; CODE XREF: sub_40103C:loc_40103
CODE:00401007                 push    ebx
CODE:00401008                 call    sub_401038
CODE:00401008 ; ---------------------------------------------------------------------------
CODE:0040100D                 db 0C7h
CODE:0040100E ; ---------------------------------------------------------------------------
CODE:0040100E ; START OF FUNCTION CHUNK FOR sub_40103C
CODE:0040100E
CODE:0040100E loc_40100E:                             ; CODE XREF: sub_40103C+1↓j
CODE:0040100E                 pop     ebx
CODE:0040100F                 push    ebp
CODE:00401010                 pushf
CODE:00401011
CODE:00401011 loc_401011:                             ; CODE XREF: CODE:loc_401002↑j
CODE:00401011                 call    $+5
CODE:00401016                 pop     ebp
CODE:00401017                 add     ebp, 2Ch
CODE:0040101A                 stc
CODE:0040101B                 ja      short near ptr loc_401023+1
CODE:0040101D                 jbe     short loc_401021
CODE:0040101D ; END OF FUNCTION CHUNK FOR sub_40103C
CODE:0040101D ; ---------------------------------------------------------------------------
CODE:0040101F                 db 0C7h
CODE:00401020                 db 0E9h
CODE:00401021 ; ---------------------------------------------------------------------------
CODE:00401021 ; START OF FUNCTION CHUNK FOR sub_40103C
CODE:00401021
CODE:00401021 loc_401021:                             ; CODE XREF: sub_40103C-1F↑j
CODE:00401021                 jbe     short near ptr loc_401023+1
```

As you can see, it does not look very friendly. In the middle of the macros, you can put the real code.

Obviously, any program using the same macros over and over could be bypassed easily by analysts. So, the next step was to write a macro generator that would generate random macros, ready to be used in a loader. All those macros would therefore be different, and a search and replace could not be done.

Now, depending on the macro generator, it is quite possible to find a pattern, and write plug-ins for the reverse engineering Tools to remove the macros. IDA Pro plug-ins, or even IDC script can do the job. You simply need a weakness allowing identification of the macros' start and end.

Finally, the last step forward (and one used by some commercial protection systems) is on-the-fly obfuscation generation. These complex packers have built-in assemblers that allow them to generate specific obfuscation routines and then insert them between lines of real code, to make it harder to identify and remove them. The power of such engines is that they can write obfuscations that work on specific registers only, in specific cases, making the obfuscation dependent on the real code, in the way it changes the registers, data, and so forth.

Program Flow Obfuscation

Another sort of obfuscation technique works on the program flow. It can be coupled with the obfuscation techniques described above, making the analysis tedious without special tools. Usually, application code is executed from top to bottom following program conditions (tests, comparisons, conditional jumps, and similar).

Program flow obfuscation allows the dispatch of the instructions in a random order in the program. Therefore the first instruction you see could be the last one executed, or might be executed in the middle of the routine. What you see in your disassembler is thus not the order of execution.

Chunks of code can be placed in random order and then called using a special dispatching routine. Such a routine can use an index in an array and execute the chunks of code in the correct order, even though they are in a totally random sequence in the application.

Static analysis becomes very tedious work, and depending on how obfuscated the program flow is, you need special tools or plug-ins to be able to understand the logic. Interactive disassemblers such as IDA Pro are again a very good weapon, especially if you couple them with a plug-in.

Here is a basic example of program flow obfuscation that we can find in the loader of a packer. Note that the example is kept simple on purpose, to assist with understanding the concept.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Instruction Dispatcher
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        mov edi, 3       ; initialize index
loop_:
        ; get address to execute with the index:
        mov ebx, dword [edi*4+ array_obfuscation]
        call ebx   ; execute it
        dec  edi   ; decrement index
        jnz loop_ ; till not zero, keep looping


        push    0
        call  [ExitProcess]


code3:
; third instruction or chunk of instruction here
ret

code2:
; second instruction or chunk of instruction here
ret

code1:
; first instruction or chunk of instruction here
ret


array_obfuscation dd    0, code3, code2, code1
```

This short piece of assembly code executes code1, code2, and code3, in the correct order, even though in the disassembly, from top to bottom, there is code3, code2, and code1. This is obviously a very small example. On a larger scale, this technique can make static analysis a very painful task, because what you see is not what you get at runtime. This example is based on an obfuscation routine made by the author in a packer for a security challenge about 5 years ago.

It is certainly possible to use more than one array, and use various mathematical manipulations to calculate the final chunk address, such as using a matrix. It is possible to reorder the addresses in the arrays, and use a mathematical expression to generate the correct index. The only limitation is the coder's creativity.

In this category of obfuscation, there are also programming tricks used to obfuscate the program flow. It is possible to emulate a jump for instance, using the code below:

```
push    (jump_destination + 754841h)
sub     [esp],754841h
ret
```

A normal jump would make cross-references:

```
                jmp     short direct_cross_reference
; ---------------------------------------------------------------
                call    loc_401018      ; never executed
; ---------------------------------------------------------------

direct_cross_reference:                 ; CODE XREF: start↑j
                mov     edi, 3
```

On the other hand, this sort of flow obfuscation avoids cross-references:

```
                public start
start           proc near

var_4           = dword ptr -4

                push    402349h
                sub     [esp+4+var_4], 1337h
                retn                    ; on ret we execute loc_401012 but there is
start           endp                    ; no XREF.

; ---------------------------------------------------------------------------
                call    sub_401023      ; this line is never executed
; ---------------------------------------------------------------------------
                mov     edi, 3
```

Again, this is obviously a simple example. IDA does not make any cross-reference with this obfuscation by default, but since it is interactive, it is possible to either manually create cross-references, or write a plug-in to handle this case.

A simple IDC script would do the trick in such an easy case, and it would be possible to follow it statically.

Anti Debugging Techniques

Using a debugger, it is possible to single-step through applications, and inspect their code in real time. This is obviously a problem for packers and protectors, since it enables an analyst to reverse-engineer them. To counteract this, anti-debugging tricks are used. This section presents the most common techniques used in the last decade, but for a more complete anti-debugging reference, please read Peter Ferrie's *Anti-Unpacker* masterpiece. [FERRIE09]

IsDebuggerPresent

Despite being inefficient, the IsDebuggerPresent API function was very common in the first packers and protectors, and some of them are still using it as a first-stage check. This function uses the PEB [PEB] to detect a userland debugger. It only takes one change in the BeingDebugged flag (from 1 to 0) to bypass this check.

BreakPoint Detection

Another common technique, introduced more than a decade ago by packers and protectors, is the detection of software breakpoint. This technique is listed in the documentation of the first public packers, back in 1997.

Quite often, all packers and protectors were using something like this:

```
cmp     byte ptr [eax], 0CCh
jz      short breakpoint_detected
```

0xCC is the opcode for the *INT 3* instruction, which is what the debugger uses for software breakpoints. If EAX is pointing to an API function address and a breakpoint is set there, this piece of code detects the breakpoint.

Setting a breakpoint on the second instruction would bypass the detection. Therefore, some packers use a range scan. Here is a code snippet written by the author for a challenge [SOTM33] in 2004, to detect a software breakpoint in an obfuscated way. (This piece of code has been ripped in the past by a commercial protection system, byte to byte.)

```
DATA:00DE3274                  mov     eax, offset printf
DATA:00DE3279
DATA:00DE327A
DATA:00DE3558                  mov     eax, [eax+2]
DATA:00DE355B                  mov     eax, [eax]
DATA:00DE355D
DATA:00DE355E
DATA:00DE37F1                  mov     edi, eax
DATA:00DE37F3
DATA:00DE37F4
DATA:00DE3A90                  mov     ecx, 4
DATA:00DE3A95                  mov     eax, 660h
DATA:00DE3A9A                  shr     eax, 3
DATA:00DE3A9D
DATA:00DE3A9E
DATA:00DE3D2F                  repne scasb
DATA:00DE3D31                  test    ecx, ecx
DATA:00DE3D33                  jz      short no_bpx
DATA:00DE3D33
DATA:00DE3D35                  rdtsc
DATA:00DE3D37                  push    eax
DATA:00DE3D38                  retn
```

In order to make it a little less obvious to an unskilled reverser, the *INT* 3 opcode value is obfuscated using a "SHR" (Shift Right) instruction: 0x660 shr 3 = 0xCC. The program then checks four bytes at the API function entry point, looking for a breakpoint. If a breakpoint is found, RDTSC generates a pseudo random number and pushes it onto the stack. The RET instruction transfers to a random memory address, crashing the application. If no breakpoints are detected, the application continues its execution.

Soft ICE detections

When packers and protectors started to surface, Soft ICE was **the** debugger used by reverse engineers. There were no real alternatives at the time. As it is a kernel debugger, it hooks into the operating system and uses some of the drivers. There were a few tricks that most protectors were using at that time.

MeltIce:

This technique was once very famous, and uses the *CreateFileA* function to detect Soft ICE.

CreateFileA on *\\.\NTICE* and *\\.\SOFTICE* were two common checks. When *CreateFileA* returned a handle, packers knew Soft ICE was present in memory. However, since Driver Studio 2.7, this technique no longer works. Soft ICE is no longer sold, so these techniques are disappearing.

INT 1 DPL Trick:

When Soft ICE is installed on a machine, it hooks into the operating system. Some properties are changed and it allows detection. On a Soft ICE-free machine, whenever a user land program executes an *INT 1* trigger, it gets a "EXCEPTION_ACCESS_VIOLATION" (0xC0000005) exception, because the INT 1 has a DPL (Descriptor Privilege Level) of 0.

On the other hand, when Soft ICE is on the machine, it changes the INT 1 DPL to 3. When a user land application executes INT 1, it gets "EXCEPTION_SINGLE_STEP" (0x80000004).

Protectors usually set a SEH (Structure Exception Handler), and execute an INT 1. Depending on the EXCEPTION_CODE, they know whether Soft ICE is in memory.

This detection only works for NT-based operating systems, and not on the 9X version of Soft ICE.

Note: Apparently, the newer debugger SYSER might be detected by this technique as well. However, the author of this paper has not tested it.

There are many more tricks to detect Soft ICE, but those two were quite common. Armadillo, Asprotect and other protectors were all using them.

SEH - Structured Exception Handling

Some packers and protectors abuse Windows exception handling as a way to protect their code against analysis. This allows the packer to access the context structure of the current application and, therefore, access privileged registers such as debug registers. These registers are used by hardware breakpoints (BPM). If you can access them, you can also erase the hardware breakpoints and defeat debugging.

Here is a partial dump of the CONTEXT structure:

```
typedef struct _CONTEXT {

    DWORD   Dr0; // Debug Register 0  +4
    DWORD   Dr1; // Debug Register 1  +8
    DWORD   Dr2; // Debug Register 2  +0Ch
    DWORD   Dr3; // Debug Register 3  +10h
    DWORD   Dr6; // Debug Register 6  +14h
    DWORD   Dr7; // Debug Register 7  +18h
```

```
        DWORD   SegGs;        // GS +8Ch
        DWORD   SegFs;        // FS +90h
        DWORD   SegEs;        // ES +94h
        DWORD   SegDs;        // DS +98h

        DWORD   Edi; // EDI +9Ch
        DWORD   Esi; // ESI +0A0h
        DWORD   Ebx; // EBX +0A4h
        DWORD   Edx; // EDX +0A8h
        DWORD   Ecx; // ECX +0ACh
        DWORD   Eax; // EAX +0B0h

        DWORD   Ebp; // EBP +0B4h
        DWORD   Eip; // EIP +0B8h

        DWORD   SegCs;        // CS +0BCh
        DWORD   EFlags;       // EFLAGS +0C0h
        DWORD   Esp; // ESP +0C4h
        DWORD   SegSs;        // SS +0C8h

    } CONTEXT;
```

This structure holds all the information about the current context. When an exception occurs, the context is filled by Windows. A packer can then access the debug registers value, and check for hardware breakpoints.

Debug Registers

The debug registers are used by hardware breakpoints. Unlike software breakpoints (INT 3), the debugged program is not modified by hardware breakpoints.

Breakpoint Registers: DR0, DR1, DR2, DR3

Four registers are used for hardware breakpoints. Therefore, you cannot put more than 4 hardware breakpoints per context (at least, without the use of hacks). The registers are 32 bit (on x86 processors), and they hold the breakpoint addresses.

State Register: DR6

The DR6 register is used jointly with the INT1 Handler. When it triggers, DR6 is used to identify the cause of the interruption.

Control Register: DR7

DR7 is used to define the sort of hardware breakpoint we want to use. Certain bits of the register define the size of the breakpoint. It is possible to work on a byte, a word, or a double word. Other bits define the breakpoint condition:

Read (R), Write (W), Read-Write (RW), or Execution (X). Additional bits are available, but those are used for debug-register protection.

For more information, see the documentation provided by Intel.

Usage of the Debug Registers in Packers/Protectors

As seen previously, some protections will erase the debug registers with the help of structured exception handling (SEH). This way, hardware breakpoints are erased and the debugger will not stop. Some packers will also use the debug registers to store a decryption key or hardcoded value used to compute a decryption key. Whenever a hardware breakpoint is set, the value is modified and decryption can no longer be done while the software is debugged; the application simply crashes.

There are some tools to protect debug registers against erasing (for example, SuperBPM). However, it is quite easy to detect them. A protector can set some values in the debug registers. If, upon re-reading those values, they are not the same, then they are protected.

A better approach is to hook NtKiUserExceptionDispatcher and implement a fake debug register mechanism to be able to set hardware breakpoints while still providing a copy of their modification to the protection system.

Anti-Dump Techniques

Anti-dump refers to protections preventing process dumping or techniques used to render the dumped executable unusable. Such protection is done either at runtime or protection time.

Import Address Table Redirection

One of the first commercial products using Import Address Table (IAT) redirection was introduced by Macrovision/C-Dilla SafeDisc and was quite popular in 1999. Some custom protections might have used IAT redirection before, but not on a large scale.

The basic idea of IAT redirection is to "hook" the IAT entries of the Windows API functions used by the application.

The IAT is filled with protection pointers. The protected program no longer calls the protection directly. A protection stub is first called which redirects to a good API function address. This essentially acts as a proxy for function calls.

Schematically, here is what is occurring:

```
                    calls
normal program -----------> Windows API Function


                      calls                        calls
protected program -----------> Protection stub -----------> Windows API Function
```

The main advantage of such redirection is that when someone dumps the protected process to disk, all the IAT pointers are no longer valid. They are valid in memory only, and point to protection code, which in the dumped file is no longer valid.

With an unprotected program, the API function addresses would be available and it would be easy to reconstruct or repair the IAT.

The next section discusses various techniques used by protectors to redirect API function calls.

Simple Redirection

In an unprotected program, functions are usually called like this:

```
FF15D4B05300        CALL      [KERNEL32!GetVersionExA]   ;  CALL DWORD PTR [53B0D4]
```

This function call uses the IAT. In this case, the address at 0x053B0D4 is being used.

Now, if we look at a dump of the IAT, we see:

```
01AF:0053B0D4 0B 16 80 7C 88 43 80 7C-50 E1 80 7C C6 20 80 7C
01AF:0053B0E4 B1 EE 80 7C 08 2D 80 7C-B2 B9 80 7C 8D B9 80 7C
01AF:0053B0F4 BD C8 80 7C 8E 5A 80 7C-32 60 80 7C DA C5 80 7C
01AF:0053B104 FA AB 80 7C B1 42 80 7C-AE 79 80 7C D5 79 80 7C
01AF:0053B114 F8 D4 80 7C B1 6F 80 7C-6B 51 80 7C B7 EE 80 7C
01AF:0053B124 EC 13 80 7C 54 74 80 7C-9F 7D 80 7C 3C C6 80 7C
01AF:0053B134 9F FA 80 7C 22 0B 80 7C-2A 0A 80 7C 18 13 80 7C
```

The IAT is an array of API function pointers imported by the application, and it is filled by the Windows Loader at runtime. 0x7c80160B is the address of GetVersionExA.

A protection overwrites all Windows function pointers with pointers to the protector code. (All functions are not necessarily redirected; it depends which DLL the functions are imported from.)

Here is an example of simple redirection:

```
FF1512846000  CALL [00608412]

IAT DUMP: 01AF:00608412 75 20 85 00 41 52 85 00 11 74 85 00 73 98 85 00
```

The protected application calls 0x852075:

```
00852075 PUSH 7c80160B
0085207A RET
```

The protector pushes the API function address onto the stack, and then uses a RET instruction, which basically emulates a jump. This is the simplest redirection possible.

In order to fix this redirection, the IAT would need to be updated, replacing 0x0852075, at 0x00608412, with 7c80160B.

Doing this for all IAT slots defeats the import protection.

Function Entry Emulation Redirection

The idea of this technique is to emulate instructions from the redirected API function. Here is an example:

```
01AF:00442BAA  FF15D4504600     CALL    [004650D4]
```

IAT Dump:

```
01AF:004650D4 14 20 EE 00  28 20 EE 00  -34 20 EE 00 40 20 EE 00
01AF:004650E4 4C 20 EE 00  58 20 EE 00  -6C 20 EE 00 78 20 EE 00
01AF:004650F4 84 20 EE 00  98 20 EE 00  -AC 20 EE 00 10 C9 EC 00
01AF:00465104 B8 20 EE 00  C4 20 EE 00 -D4 20 EE 00 F0 20 EE 00
01AF:00465114 08 21 EE 00  14 21 EE 00  -2C 21 EE 00 38 21 EE 00
```

The hooked function calls the address: **EE2014**.

Here is a disassembly of the function:

```
01AF:00EE2014  55                        PUSH      EBP
01AF:00EE2015  8BEC                      MOV       EBP,ESP
01AF:00EE2017  83EC0C                    SUB       ESP,0C
01AF:00EE201A  56                        PUSH      ESI
01AF:00EE201B  57                        PUSH      EDI
01AF:00EE201C  E9F2F50ABF       JMP        7C801613 <= Calls the API Function
```

This uses the SoftICE command “:what 7C801613”. The value 7C801613 is (a) KERNEL32!GetVersionExA+0008

Using SoftICE (or any debugger), we find that the redirected function has a little stub, which eventually jumps to a Windows function address plus an offset.

Instead of jumping to the start of the API function, the protector makes a copy of the function entry. A few instructions are copied inside the protection buffer, and then a "JMP" is assembled to skip the copied instructions in the real function.

In this way, the initial instructions inside the DLL are skipped – they are executed inside the protection code instead – and the program then jumps to the API function plus number of bytes skipped.

Protectors usually have an Length Disassembler Engine (LDE) to determine the size of the instructions they emulate.

When this technique was first introduced, it defeated all import reconstruction tools available at the time. Modern tools, however, are not fooled by this technique. One benefit, though, is that breakpoints on Windows API functions are useless if placed at the API function entry once the protection has finished redirecting all pointers.

Better redirection techniques are used by commercial protection systems and are far more complex than those presented here.

API Emulation

As time went by, IAT tracers got better and better, and IAT redirections became easier to bypass with automated tools.

In order to block such tracers, some protectors started to "emulate" a few API function calls. Some functions always return the same value during the execution of a process (for example, GetProcessID, GetTempPath, GetWindowsDirectory). Protectors started calling these functions inside their loaders and saving the results. While redirecting the IAT, they would look for such easy to emulate functions, and update their pointers with a simple stub, returning the previously saved return value.

Example: GetVersion Emulation

```
001B:016D1408  6A00                      PUSH     00
001B:016D140A  E8513DFFFF         CALL       KERNEL32!GetModuleHandleA
001B:016D140F  FF35E86C6D01       PUSH       DWORD PTR [016D6CE8]
001B:016D1415  58                        POP      EAX
001B:016D1416  8B05F86C6D01       MOV        EAX,[016D6CF8]
001B:016D141C  C3                        RET
```

This emulation first calls GetModuleHandleA, but this is a fake call to trick IAT tracers. We know that Windows functions use the EAX register for their return values. That function actually updates EAX using a DWORD at 0x016D6CF8 and returns. The protector saves the return value of GetVersion at 0x016D6CF8.

Now, whenever the protected application calls this function, it will return the good value into the EAX register, yet it will not execute the API function at all.

Tracers have no idea of what to do since they would never find a real API call. Some of them would return GetModuleHandleA, but this is not the correct function and the rebuilt application would crash.

However, in many cases, it is possible to guess the emulated function. For example, if the emulated function is GetTempPath, the EAX register should point to the temp path string.

Some emulated functions are not so obvious. It is possible to use hardware breakpoints to stop whenever the variable holding the emulated information is updated at loading time. Usually, the function call is right above it. Using pattern matching, it is quite easy to write generic tools for a given protection.

Code Mangling

Code mangling is a protection technique which involves the modification of the executable code section prior to encrypting or compressing it. The modification is done at protection time, and therefore is permanent. Because of this, an analyst knows that the original code is gone forever, and a dump of the decrypted or decompressed section is not enough to bypass the protection.

The application becomes dependent on the protection and cannot run without it, unless it is fixed using custom tools.

Example of Code Mangling: NANOMITES

The Armadillo protection system introduced NANOMITES. Basically, parts of a protected program would be scanned for conditional and unconditional jumps, and then replaced by INT 3 instructions at protection time.

When such applications are executed, the INT 3 triggers exceptions, and the protector emulates the jumps (that are no longer there) using context modification.

The EIP register is updated according to the eflags to emulate jumps. Obviously, there is an internal table with all the information necessary for emulation, but there are also fake entries to fool "dumb" rebuilding tools. A custom tool is required to fix the code mangling in order to get a working executable again.

Entry Point Elimination

Some protectors make a copy of an APIfunction entry point before destroying it. Some make byte-to-byte copies, whereas others mutate the entry point in order to obfuscate it and have it inside the protection stub. It is no longer possible to simply copy and paste the original bytes from the protectors back to the entry point address.

It is possible, however, to make a new section; dump the mutated entry point there; and change the entry point address to point to the location of the new section. This way the application still executes even though the original entry points have not been reconstructed. The stack can also be used to recover mutated instructions, since all high-level compilers have a specific structure at their entry point.

Some protections also translate the entry point instructions into an intermediate language and use a virtual machine to emulate the instructions. However, it is still possible to guess the missing instructions depending on the number of ripped instructions (provided it is known what compiler is being used in the protected application).

SizeOfImage Modification

Some protectors will change the SizeOfImage in memory which results in invalid dumps. Many process dumpers use the SizeOfImage to compute the size of a process. The invalid image size results in an invalid process dump, or may even block the dump completely. LordPE, a famous process dumper and PE editor, has an option to fix the SizeOfImage before dumping the process.

PAGE NO ACCESS

Another trick is to set pages in the middle of a process with PAGE_NO_ACCESS rights.

Typically, a few pages of the protected application are never used by the protector nor the protected application. But there is still useful information after those pages, such as the loader or part of it.

By setting this memory area to PAGE_NO_ACCESS, process dumpers fail to read that region and the whole process cannot be dumped.

Conclusion

The paper has aimed to explain how packers work internally. As said in the introduction, the most advanced techniques were left out on purpose, because they are used in commercial protection systems. Most custom packers found in malware are usually quite simple, and rely heavily on the techniques presented here. Sometimes, malware is protected using what people tend to call a packer, when they are actually just loaders (an executable is embedded in the "packed" malware, and executed in memory without being dropped on disk). Since they are not packers *per se*, they were not included in this paper.

For further information about anti-debugging techniques, see the references below.

References

[PE-DOC] - http://spiff.tripnet.se/~iczelion/files/pe1.zip

[FERRIE09] - http://pferrie.tripod.com/ (Anti-Unpacker Tricks 2, parts 1 to 7)

[PEB] - http://undocumented.ntinternals.net/UserMode/Undocumented%20Functions/NT%20Objects/Process/PEB.html

[SOTM33] - http://old.honeynet.org/scans/scan33/

[APLIB] - http://www.ibsensoftware.com/products_aPLib.html