Linnéuniversitetet
Institutionen för datavetenskap, fysik och matematik

Bachelor project

General Unpacking
Overview and Techniques

Author: Danut Niculae
Supervisor: Jesper Andersson
Date: 2015-05-18
Course code: 2DV00E, 15 credits
Level: Bachelor
Subject: Computer Science

Abstract

Over the years, packed malware have started to appear at a more rapid pace. Hackers are modifying the source code of popular packers to create new types of compressors which can fool the Anti Virus software. Due to the sheer volume of packer variations, creating unpacking scripts based on the packer's signature has become a tedious task. In this paper we will analyse generic unpacking techniques and apply them on ten popular compression software. The techniques prove to be successful in nine out of ten cases, providing an easy and accessible way to unpack the provided samples

Keywords: Protections, Reverse Engineering, Unpacking

Contents

List of Figures

1 Introduction

Since the beginning of software development, programmers have searched for a way to protect their work against intellectual theft. From simple code obfuscation to complex multi-layered protections, various methods have been used to not only shrink the files to a more manageable size, but also to keep people from peaking into the code itself. However, with the appearance of anti virus (AV) solutions, hackers have been using protections to hide their malicious code and remain undetected while subjugated to malware scans or manual analysis. In order for the samples to be ready for analysis, they will have to go through a process called "unpacking". At this stage, all protections are removed and the original code is extracted for further processing. However, in order to save time and eliminate the process repetition, automated tools are built to aid the unpacking process. Such tools can take multiple files at the same time and complete the tasks in matter of minutes. Still, these tools heavily rely on scripts written for each individual protection or compressor. As such, a certain amount of time is still being spent on reversing the protection and writing an automated unpacking script which can be used by an unpacking engine. The major problem with these scripts is that they are designed for a specific type of protection (UPX[17], FSG[5], etc.) and in some cases, a specific version of that packer. This means that a large database of scripts needs to be implemented and updated regularly, in order to keep up with the high volume of protection variants. To solve this issue, generic unpacking solutions are becoming more popular amongst researchers and AV companies. Most of these solutions are based on virtualization techniques and sandbox methods, which includes complex algorithms that monitors the malware sample and detects the memory page modifications. However, in this paper we will focus on a more simple way of unpacking different compressors, without the need of scripts.

1.1 Ethics

As previously mentioned, protections have appeared on the market with the sole purpose of providing a way for developers to protect their work against intellectual theft (this however excludes compressors, as their main focus is to shrink the binary file for an easier distribution). Even though some software anti-piracy techniques follow simple yet effective methodologies[25]:

- **Time Based Licensing**: This method will register your software for a certain amount of time, implementing encrypted registry entries and time based checks.

- **Product Activation**: This method will generate a serial key based on your hardware or a mathematical algorithm (including popular encryption routines).
- **Tamper Proofing**: CRC/MD5 checks which will validate the software's integrity, thus ensuring that the software is not compromised or modified in any way
- **Component Based Licensing**: Serial keys which are generated for a specific client.
- **Software Locking**: The serial key is bind by a client's hardware.

Protections such as Software Passport's Armadillo implements these licensing schemes directly into the protection. However, most of the techniques described in Mumtaz's paper[25] can be bypassed through Reverse Engineering. Furthermore, if unpacking knowledge is used on protections, such as Armadillo, the software considers itself registered, as the licensing scheme is in the protection. For this reason and more, all of the methods described above are addressed only to compressors and not actual protections. The information gathered in this paper, should not be used for any other purposes other than malware unpacking.

1.2 Research Question

As time is spent creating new unpacking scripts and creating new virtualization techniques which are able to detect the time where the sample target has reached the unpacking routine, could a more simple approach be used? Could an unpacking engine be built, which offers generic unpacking without relaying on scripts or pattern based detection? This paper aims to apply two generic unpacking methods and test their success rate against some of the known packers. Furthermore, the paper will also offer an insight into how these methods work and provide an overview on the necessity of generic unpacking.

1.3 Paper structure

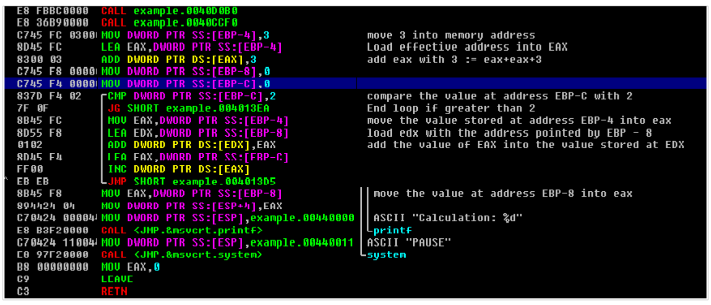
The paper aims to introduce the reader into the world of reverse engineering and packers, present two popular generic unpacking methods and test them on 10 popular packers and gather the results to verify if these two methods can be successful in unpacking the targets. The first chapters will focus on offering background knowledge on reverse engineering and how packers work. Next the paper will present the generic unpacking techniques and start gathering the data by using them on our packed executables. Afterwords, the results will be analysed and verify the success rate of the used methods. Last, the paper will offer an insight into how a generic unpacking engine could be built using Titan Engine SDK[7] and proceed with the conclusion.

2 Background Knowledge

In order for the reader to understand the concept of packed sample and generic unpacking methods, we must first understand two things: Reverse Engineering and protections. Both topics are important to understand the unpacking process and the methods used. In the following subsections, we will discuss how both work and how we can use these to achieve generic unpacking.

2.1 Reverse Engineering

Software Reverse Engineering is the process of taking a system, or parts of a system, and analyse their functionality and design[1]. As software engineers, we are used to this type of procedure as a way to analyse and reconstruct parts of a system at a higher level of abstraction. Be it our own system or a competitor's new technology, we read and understand the code and integrate it within our software.



```
E8 FBBC0000    CALL example.0040D0B0
E8 36B90000    CALL example.0040CCF0
C745 FC 0300(  MOV DWORD PTR SS:[EBP-4],3                    move 3 into memory address
8D45 FC        LEA EAX,DWORD PTR SS:[EBP-4]                  Load effective address into EAX
8300 03        ADD DWORD PTR DS:[EAX],3                      add eax with 3 := eax+eax+3
C745 F8 0000(  MOV DWORD PTR SS:[EBP-8],0
C745 F4 0000(  MOV DWORD PTR SS:[EBP-C],0
837D F4 02    ┌CMP DWORD PTR SS:[EBP-C],2                    compare the value at address EBP-C with 2
7F 0F         │JG SHORT example.004013EA                     End loop if greater than 2
8B45 FC       │MOV EAX,DWORD PTR SS:[EBP-4]                  move the value stored at address EBP-4 into eax
8D55 F8       │LEA EDX,DWORD PTR SS:[EBP-8]                  load edx with the address pointed by EBP - 8
0102          │ADD DWORD PTR DS:[EDX],EAX                    add the value of EAX into the value stored at EDX
8D45 F4       │LEA EAX,DWORD PTR SS:[EBP-C]
FF00          │INC DWORD PTR DS:[EAX]
EB EB         └JMP SHORT example.004013D5
8B45 F8        MOV EAX,DWORD PTR SS:[EBP-8]                 ││move the value at address EBP-8 into eax
894424 04      MOV DWORD PTR SS:[ESP+4],EAX                 ││
C70424 0000(   MOV DWORD PTR SS:[ESP],example.00440000      ││ASCII "Calculation: %d"
E8 B3F20000    CALL <JMP.&msvcrt.printf>                    │└printf
C70424 1100(   MOV DWORD PTR SS:[ESP],example.00440011      │ASCII "PAUSE"
E8 97F20000    CALL <JMP.&msvcrt.system>                    └system
B8 00000000    MOV EAX,0
C9             LEAVE
C3             RETN
```

Figure 2.1: Screenshot of a simple subroutine debugged in OllyDbg

However, reverse engineering is used for more than that. Let us assume that a new type of malware is released into the wild. Without the source code, we must rely on reversing skills to analyse and understand what the new threat does. This is why we rely on Reverse Engineering to analyze the malware's contents, understand how it works, so that a counter measure can be implemented, but also to protect intellectual property against malicious individuals.[1]. The process may vary from simply opening the malware file into a debugger and checking its internal structure, to more complex procedures (VM snapshots comparisons, network monitoring, etc.). The reverser must be able to understand Assembly code, be well versed in the Operating System's internal calls and API's, as well as have knowledge of different programming languages and their structure when disassembled. With the help of a debugger/disassembler (OllyDbg[11], IDA Pro[14], W32Dasm[15], etc.), a reverser can inspect the functionality and subroutines of a file, without having access to the source code. In order to demonstrate this process, I have built a small executable file which does a simple addition with 2 integers. Now, let us assume that we no longer have access to the source code, but we need to understand what the application does (in some cases even recreate the functionality). As such, the executable file has been loaded into a debugger (Immunity Debugger[6]) and is ready for analysis. Figure 2.1 shows what the executable does: places number 3 into a memory address a registry, add 3 again to that number, push 0 to another memory address, enter a loop and add the above mentioned registry with the number at the new address. The loop goes around 3 times. After the loop ends, print the number on screen. From here on, it is easy to recreate the source code of that file:

```
int a=3;
int b=0;
a+=3;
for(int i=0;i ≤ 2 ;i++) b+=a;
printf("Number is: "%d",b);
```

We could use the source code to repurpose it into our own software and recreate the test executable file from scratch. This is just a small example of how RE can work, as on a bigger scale, this could be used to reconstruct whole encryption routines and use it's mechanisms as an exploit point for both white hats and black hats.

2.2 Compressors and Protections

As we have seen so far, reverse engineering is used to analyse and reconstruct subroutines of a system. However, this process can also be used to modify files without the author's consent. Removing license checks, injecting code, modifying memory addresses, these are all possible with the help of just a debugger. Furthermore, a black hat hacker can infect other system files by injecting malicious code into them. However, AV companies have managed to detect such behaviour by scanning the files on your system (hard disk, boot sector, running processes) for certain patterns or code snippets which tags the file as malware[3]. As such, malware creators have started to utilize software protections (aka packers) to hide the payload from AV scans. To answer this, AV companies have started to implement unpacking engines into their software (providing updates with specific scripts for each packer)[2] and in some cases, certain packers are flagged as malware. By blacklisting certain packers based on their signature, some other variations of the same packer may fall under the blacklist. The reason for this is the ease with which hacker can modify source codes of different protections to create variations which can fool the AV. Unfortunately, the number of packers and their variations are increasing dramatically. According to a 2006 Black Hat Briefing, over 92% malware found in the wild are packed[20]. This means that AV software has to update their unpacking signatures more frequently then ever. With so many variants of popular compressors appearing each day, whole families of compressors are declared blacklisted and false-positive warnings are triggered even on legitimate software, which happens to use a modified version of FSG (Fast Small Good).

2.3 Pattern Based Detection

As we have seen so far, when a new malware appears in the wild there is a high chance that it is packed with some form of protection. Hundreds of samples are being sent each day to laboratories around the world for analysis[2]. To unpack all of these samples one by one would take too long and the backlog of samples would become too large to handle. In order to speed up the unpacking process, unpacking engines have been created. These engines will scan a file and compare the packers signature against a database of signatures and once a match has been found, the unpacking algorithm is selected and the procedure begins. In order to generate a database, engineers analyse and understands how the protection behaves and most importantly, finds repeating patterns within packed file. For the purpose of this thesis, we will take UPX (Ultimate Packer for eXecutables) and FSG (Fast Small Good) as packer examples. The selection was made as both packers are easy to understand and offer a simple way of pattern detection.

2.3.1 UPX

After packing a simple c++ executable file with UPX version 3.91 we open the file in OllyDbg. Checking the memory section reveals that there exists 3 sections called UPX. At first glance it may seem that it would be enough for an unpacker to start the unpacking routine for UPX. However, most packers can rename the memory sections to just about anything, thus relaying on the name itself, would not yield concrete results. The way UPX works is that after it has compressed a binary file, the output will contain 3 sections. The first section is a placeholder for the decompressed data, the second section contains both the compressed data and the unpacking code, while the 3rd section contains the resource data[21]. We will not go into more details about how UPX works, as we will focus more on the unpacking code and finding the Original Entry Point. Most files that are packed by an unmodified version of UPX, tend to share a similar pattern: decrypt the code, resolve imports and jump to OEP (Original Entry Point). The decryption routine may differ from file to file, thus finding a pattern there may pose some challenges. With this in mind, we can observe from the code bellow that UPX does 2 calls, a call to LoadLibraryA and a call to GetProcAddress. In our packed example we find the same behaviour, as shown in the following code:

```
00407676   .  50              PUSH EAX
00407677   .  83C7 08         ADD EDI,8
0040767A   .  FF96 3C700000 CALL DWORD PTR DS:[ESI+703C]   ;  LoadLibraryA call
\noindent
00407680   .  95              XCHG EAX,EBP
00407681   .  8A07            MOV AL,BYTE PTR DS:[EDI]
00407683   .  47              INC EDI
00407684   .  08C0            OR AL,AL
00407686   . 74 DC           JE SHORT test.00407664
\noindent
00407688   .  89F9            MOV ECX,EDI
0040768A   .  57              PUSH EDI
0040768B   .  48              DEC EAX
0040768C   .  F2:AE           REPNE SCAS BYTE PTR ES:[EDI]
0040768E   .  55              PUSH EBP
0040768F   .  FF96 40700000 CALL DWORD PTR DS:[ESI+7040]
00407695   .  09C0            OR EAX,EAX
00407697   .  74 07           JE SHORT test.004076A0
00407699   .  8903            MOV DWORD PTR DS:[EBX],EAX
0040769B   .  83C3 04         ADD EBX,4
0040769E   .  EB E1           JMP SHORT test.00407681
004076A0   .  FF96 50700000 CALL DWORD PTR DS:[ESI+7050]   ;  GetProcAddress call
004076A6   .  8BAE 44700000 MOV EBP,DWORD PTR DS:[ESI+7044]
004076AC   .  8DBE 00F0FFFF LEA EDI,DWORD PTR DS:[ESI-1000]
004076B2   .  BB 00100000    MOV EBX,1000
004076B7   .  50              PUSH EAX
004076B8   .  54              PUSH ESP
004076B9   .  6A 04           PUSH 4
004076BB   .  53              PUSH EBX
004076BC   .  57              PUSH EDI
004076BD   .  FFD5            CALL EBP    ; VirtualProtect call
```

We can observe here that two calls are made each time to LoadLibraryA and GetProcAddress. The reason why these calls are made, is to restore the IAT (Import Address Table) table. Since we know that each time UPX will call these two functions and in the end it will call VirtualProtect, we can extract a pattern of bytes. Using Titan Engine SDK, we can start creating our patterns:

```
int dtPattern1Size = 5;
BYTE dtPattern1[5] = {0x50,0x83,0xC7,0x08,0xFF};
void* dtPattern1CallBack = &cbLoadLibrary;
void* dtPattern1BPXAddress = NULL;

int dtPattern2Size = 7;
BYTE dtPattern2[7] = {0x50,0x47,0x00,0x57,0x48,0xF2,0xAE};
void* dtPattern2CallBack = &cbGetProcAddress;
void* dtPattern2BPXAddress = NULL;

int dtPattern3Size = 6;
BYTE dtPattern3[6] = {0x57,0x48,0xF2,0xAE,0x00,0xFF};
```

```
void* dtPattern3CallBack = &cbGetProcAddress;
void* dtPattern3BPXAddress = NULL;

int dtPattern4Size = 8;
BYTE dtPattern4[8] = {0x89,0xF9,0x57,0x48,0xF2,0xAE,0x52,0xFF};
void* dtPattern4CallBack = &cbGetProcAddress;
void* dtPattern4BPXAddress = NULL;
```

Later on when we will create the unpacking algorithm, we will use these patterns to determine whether UPX has been used for compression, or rather another variation of it was used. As we can see, different types of patterns can be inserted into an unpacker. Usually an engineer will spend up to 6 hours developing the unpacking procedure[21] and more if necessary for harder protections.

2.3.2 FSG

FSG behaves somewhat similar to UPX, as it follows the same unpacking procedure: decrypt, call LoadLibraryA and GetProcAddress and jump to OEP. Same as before, we will not focus on the decryption routine, but rather on the last phases of the unpacking process. The same C++ file which was packed with UPX, is now packed with FSG. By opening the file in OllyDbg we find the following code:

```
004001C1    5E          POP ESI
004001C2    AD          LODS DWORD PTR DS:[ESI]
004001C3    97          XCHG EAX,EDI
004001C4    AD          LODS DWORD PTR DS:[ESI]
004001C5    50          PUSH EAX                          ; push library into memory
004001C6    FF53 10     CALL NEAR DWORD PTR DS:[EBX+10]   ; call LoadLibraryA
004001C9    95          XCHG EAX,EBP
004001CA    8B07        MOV EAX,DWORD PTR DS:[EDI]
004001CC    40          INC EAX
004001CD    78 F3       JS SHORT test_FSG.004001C2
004001CF    75 03       JNZ SHORT test_FSG.004001D4
004001D1    FF63 0C     JMP NEAR DWORD PTR DS:[EBX+C]     ; Jump to OEP
004001D4    50          PUSH EAX                          ; push name of function
004001D5    55          PUSH EBP
004001D6    FF53 14     CALL NEAR DWORD PTR DS:[EBX+14]   ; call GetProcAddress
004001D9    AB          STOS DWORD PTR ES:[EDI]
004001DA    EB EE       JMP SHORT test_FSG.004001CA
```

We can now spot a reoccurring pattern, the jump to the OEP is set between the two calls: LoadLibraryA and GetProcAddress. The way this code works is that after the last function has been resolved, the decryption routine ends with a jump to the address of the original entry point. By knowing this, we can set up a pattern into our unpacking engine:

```
int dtPattern1Size = 3;
BYTE dtPattern1[3] = {0x50,0xFF,0x53};
void* dtPattern1CallBack = &cbLoadLibrary;
void* dtPattern1BPXAddress = NULL;

int dtPattern2Size = 4;
BYTE dtPattern2[4] = {0x50,0x55,0xFF,0x53};
void* dtPattern2CallBack = &cbGetProcAddress;
void* dtPattern2BPXAddress = NULL;
```

By now, the engine can differentiate between UPX and FSG, based solely on the patterns. In case of AV software, the patterns for detection are much more complex and tools such as PEiD[12] have quite a large database of signatures.

2.4 Patterns and More Patterns

The biggest issues with pattern detection is the speed with which new versions of packers appear on the market. Hackers have access to source codes of popular compression programs such as FSG, UPX and Yodacrypt[22][18]. By having access to the source code, malwares are often compressed with modified versions of the packers previously mentioned. This means that pattern detection may often run into false-positive and in some cases, overlook them entirely[22][20]. Dedicated tools such PEiD often yields faulty detection if the packed sample has an obfuscated entry point for simple protections such as UPX. Furthermore, with each new version of a packer, the decompression routine may change, forcing the AV software to adapt as well. The unpacking engines are being updated frequently to accommodate these changes, but in most cases this is still not enough[20].

3 Generic Unpacking

As we have seen so far, pattern based detection is useful when it comes to detecting certain protections and developing unpacking routines based on the packer's signature. However, the issue with pattern based approach is still time and effort. The engine needs to be updated constantly with new signatures and more importantly, even with a such a database in effect, it is still rather easy to hide a common packer by just scrambling the entry point. In order to avoid such problems, reverse engineers have started to take a more generic approach. Unlike pattern based methods, general unpacking doesn't need to determine which type of packer was used, the main goal is to explore the binary contents after the decryption routine is at an end. New tools such as OmniUnpack[23] and PolyUnpack[24] provides alternative means of unpacking. Both tools are designed around the generic approach of unpacking. Modern generic unpackers are using virtual technologies and emulators to track and monitor the file's behaviour[23]. Thus the decrypted section of the file is detected when the sample's payload starts to activate. In the case OmniUnpack, it looks out for changes in the memory page, while also tracking any suspicious system calls. Once the file has done either of these, the process is paused and a step-by-step tracing is started[23]. However, the focus of this paper is the actual techniques implemented into unpacking engines to find OEP and dump a fully working unpacked file.

3.1 The paper

As previously mentioned, in the following sections we will discuss two most popular unpacking techniques which could be implemented into an unpacking engine (without the need to rely on scripts). The aim is to draw attention on how simple techniques could save researchers time and effort. As such, we will test our techniques on certain compressors used by malware creators to hide their work from an AV software. Unlike unpacking scripts, the methods described bellow are not dependent on the packer's signature and they do not follow different routines for each variation/version of the packer, but rather apply the same technique on all of our examples. These methods have been derived after unpacking hundreds of samples by both myself and reverser's around the world. Unfortunately, most of the relevant information regarding unpacking procedures which do not involve virtualization techniques, are extracted from different forums and websites where reverser's share their knowledge (ex: Tuts4You[16]). As the paper's focus is not the unpacking engines or the scripts themselves, but rather methods which could be implemented and used quick and easy.

3.2 Preparations

In order to properly test the unpacking methods, we will set up a list of 10 packers (UPX, FSG, nPack, Mew[8], nsPack, AsPack[4], Yoda's Protector[19], Morphine[10], Petite[13] and Mpress[9]). The testing file will be a simple "Hello World!" C++ executable. The executable will be packed with the 10 protections mentioned above and unpacked using a debugger (Immunity-Debugger). The procedure will be separated into two parts: the standard approach and the generic approach. While presenting these methods, we will also be using an unpacking engine, utilizing the Titan Engine SDK. The computer used for testing runs on a Windows 7 32 bit operating system.

3.3 Limitations and scope

All of the techniques used in this paper are designed for packers and not protectors. As such, we will not deal with anti-debugging techniques or other anti-reverse tricks. Protection software such as SoftwarePassport's Armadillo or Themida are beyond the scope of this paper, as these protections requires special tools and knowledge to defeat the protection. Furthermore, this paper will focus on the techniques used for unpacking and not the technologies themselves. Emulators and kernel drivers are also beyond the scope of this paper. With this in mind, this paper will offer an insight into two known methods of generic unpacking. We will test out and see if it is possible for an unpacking engine to be build without the need to rely on scripts for each individual protection, as specified by their pattern. There exists free engine on the market which provides solutions for unpacking different kinds of compressors and protections, but they need unpacking routines and signatures with each protection.

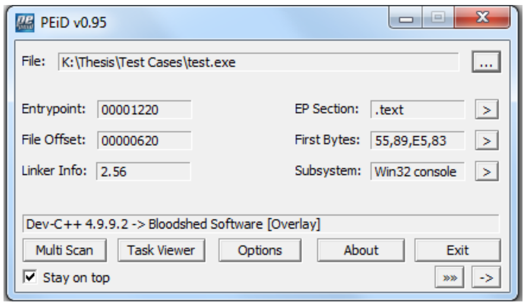
4 Unpacking Techniques

In this paper, we will discuss generic unpacking methods and the API's that can be used to find the end of decryption routine and the OEP. With this in mind, we will have a look at how certain Windows functions can be useful to us. We now know that after the decryption routine ends, the packer will resolve and restore the import table. For this reason, the LoadLibraryA (in some cases GetModuleHandle) and GetProcAddress functions are called at the end of the routine. As such, setting breakpoints on these functions can prove to be quite useful. The normal process for this method would go as follows: - Set a breakpoint on LoadLibraryA - Run the sample until it stops at the breakpoint - Run until all libraries are loaded - Trace the code back to the packed section of memory - Trace through the code until the last jump/return Setting breakpoints on LoadLibraryA may sometimes prove a bit tricky, as it may not always return to the end of the unpacking routine. However, in order to obtain a more refined result, we need to search for the last function loaded, which means setting the breakpoint on GetProcAddress. As the table is restored and the unpacked sections of the memory are available for access, the program will now run in the computer's memory.

In cases when breaking on the above mentioned function fails, reverser's make use of what is known as "The ESP Trick". Most packers includes the instruction "PUSHAD" near the entry point in order to place all of the registers into the stack. The reason for this is that when the decryption routine starts, the registers can utilized even if values change during the process.It creates a backup of the registers and utilizes them for decryption, returning them to their original value once the process has ended. As such, when a memory section is written, the registry stack still holds the initial values utilized by the packer. The ESP register is used as an indirect memory operand that points to the top of the stack. When a "PUSHAD" instruction is executed, ESP is modified to point at the address containing the stack. Once the decryption is done, the packer pop's all the registers from the memory and the application can utilize them for normal runtime. Knowing this, if we set a hardware breakpoint when ESP is modified, we can find the exact location when the registers are popped from the stack and the returning code to the original entry point is executed.

5 Data Collection

In the following section, we will unpack our 10 packers using the methods described above. We know from the test file we have coded that the Original Entry Point is at 00401220. Scanning the file with PEiD shows us the compiler used and the EP. The file contains no graphical elements and it was coded as a standard Win32 console application.



PEiD v0.95

File: K:\Thesis\Test Cases\test.exe

Entrypoint: 00001220
EP Section: .text
File Offset: 00000620
First Bytes: 55,89,E5,83
Linker Info: 2.56
Subsystem: Win32 console

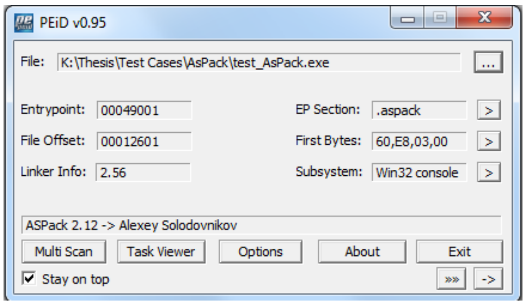
Dev-C++ 4.9.9.2 -> Bloodshed Software [Overlay]

Figure 5.1: Scan of the original test file

Please note that due to the discontinuation of nPack and nsPack, the test file could not be encrypted using these two packers. However, two "unpackmes" were found on http://tuts4you.com and they will be used as test files for this paper. Also, we will use the same website for a sample of Morphine, due to untrusted download sources for the packer.

5.1 AsPack

We begin by downloading a trial version of the packer and compress our test file. Initial results shows that file is indeed protected now with AsPack 2.12.



PEiD v0.95

File: K:\Thesis\Test Cases\AsPack\test_AsPack.exe

Entrypoint: 00049001
EP Section: .aspack
File Offset: 00012601
First Bytes: 60,E8,03,00
Linker Info: 2.56
Subsystem: Win32 console

ASPack 2.12 -> Alexey Solodovnikov

Figure 5.2: Scan of the AsPack file

Opening the file in Immunity we reach the entry point of the protection:

```
00449001    60              PUSHAD                      ; push registeres into the stack
00449002    E8 03000000     CALL test_AsP.0044900A
00449007    E9 EB045D45     JMP 45A194F7
0044900C    55              PUSH EBP
0044900D    C3              RETN
0044900E    E8 01000000     CALL test_AsP.00449014
00449013    EB 5D           JMP SHORT test_AsP.00449072
```

As we have learned from the ESP trick, we can set a breakpoint on the ESP register as soon as the PUSHAD command has been executed. As we set our breakpoint on the register, we need just one break to reach the end of the decryption stub:

```
0044940F    8985 0E040000   MOV DWORD PTR SS:[EBP+40E],EAX
00449415    61              POPAD
00449416    75 08           JNZ SHORT test_AsP.00449420
00449418    B8 01000000     MOV EAX,1
0044941D    C2 0C00         RETN 0C
00449420    68 20124000     PUSH test_AsP.00401220      ; Push OEP address into the stack
00449425    C3              RETN                        ; Return to OEP
00449426    8B85 8C040000   MOV EAX,DWORD PTR SS:[EBP+48C]
```

As we can see, after the decompression has ended, the register's are popped from the stack, the OEP address is pushed and the program returns to the memory location of the original code. Setting a breakpoint on GetProcAddress will also lead us to a close location of the end routine. The last function loaded in the case of our program is msvcrt.ungetc. After we exit the kernel's memory section and return to the application's memory page, we encounter the following code:

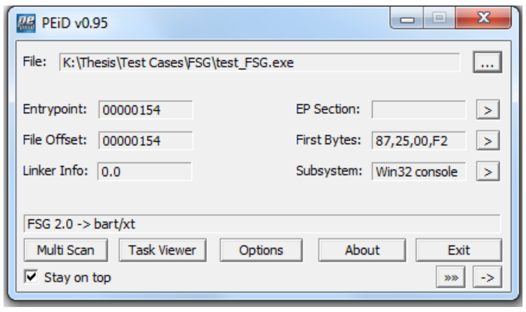
```
004492DA    53              PUSH EBX
004492DB    81E3 FFFFFF7F   AND EBX,7FFFFFFF
004492E1    53              PUSH EBX
004492E2    FFB5 A9050000   PUSH DWORD PTR SS:[EBP+5A9]
004492E8    FF95 A50F0000   CALL NEAR DWORD PTR SS:[EBP+FA5]    ; call to GetProcAddress
004492EE    85C0            TEST EAX,EAX                ; msvcrt.ungetc
004492F0    5B              POP EBX
004492F1    75 72           JNZ SHORT test_AsP.00449365
```

If we scroll a bit further down, we will encounter the end code as before. However, in the case of this packer, the call to GetProcAddress is too far away from end stub, which means that the unpacking engine has to step through multiple lines of code until it reaches the return code. As such, the ESP trick proves to be faster and easier to manage.

5.2 FSG

Unlike AsPack before, FSG is a free compression tool, which can easily be downloaded and used to compress small binary files. It provides no actual protection outside of it's compression algorithm and it is very easy to detect using any scan tool. We compress our test file using FSG v2.0 and start the unpacking process.

Just like UPX, FSG is a simple compressor following the same simple technique: decrypt, restore imports and return to original point. As such, we will set a breakpoint on GetProcAddress. Once all the imports are restored, the code will jump to the memory address containing the OEP.



File: K:\Thesis\Test Cases\FSG\test_FSG.exe
Entrypoint: 00000154
EP Section:
File Offset: 00000154
First Bytes: 87,25,00,F2
Linker Info: 0.0
Subsystem: Win32 console
FSG 2.0 -> bart/xt

Figure 5.3: Scan of the FSG file

```
004001C4     AD              LODS DWORD PTR DS:[ESI]
004001C5     50              PUSH EAX
004001C6     FF53 10         CALL NEAR DWORD PTR DS:[EBX+10]    ; LoadLibraryA
004001C9     95              XCHG EAX,EBP
004001CA     8B07            MOV EAX,DWORD PTR DS:[EDI]
004001CC     40              INC EAX
004001CD     78 F3           JS SHORT test_FSG.004001C2
004001CF     75 03           JNZ SHORT test_FSG.004001D4
004001D1     FF63 0C         JMP NEAR DWORD PTR DS:[EBX+C]       ; OEP Jump
004001D4     50              PUSH EAX
004001D5     55              PUSH EBP
004001D6     FF53 14         CALL NEAR DWORD PTR DS:[EBX+14]     ; GetProcAddress
004001D9     AB              STOS DWORD PTR ES:[EDI]
```

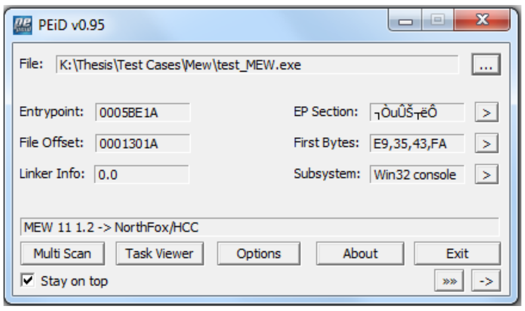
In this case, the ESP trick would work as well, as the EP starts off using an exchange register command. However, the setting a breakpoint on LoadLibrary or GetProcAddress would work faster.

5.3 Mew

Mew is another case of a free compressor that is used to hide malware from AV software. However, unlike UPX or FSG, Mew offers more options when it comes to packing. It provides LZMA algorithm for encryption, slice or delete unimportant resources and, if the program was written in Delphi, it offers the option to strip the resources which are specific to the language itself. We begin by packing our test file with Mew v11.1.2 and open our target in Immunity.

We again set a breakpoint on the GetProcAddress and after returning to the decryption stub, we encounter the following code:

```
004001DC   ^ 75 FB           JNZ SHORT test_MEW.004001D9
004001DE     FF53 F0         CALL NEAR DWORD PTR DS:[EBX-10]     ; LoadLibraryA
004001E1     95              XCHG EAX,EBP
004001E2     56              PUSH ESI
004001E3     AD              LODS DWORD PTR DS:[ESI]
004001E4     0FC8            BSWAP EAX
.......................................................
004001F1     91              XCHG EAX,ECX
004001F2     40              INC EAX
004001F3     50              PUSH EAX
004001F4     55              PUSH EBP
004001F5     FF53 F4         CALL NEAR DWORD PTR DS:[EBX-C]      ; GetProcAddress
004001F8     AB              STOS DWORD PTR ES:[EDI]
004001F9     85C0            TEST EAX,EAX
004001FB   ^ 75 E5           JNZ SHORT test_MEW.004001E2         ; Return to OEP
```


PEiD v0.95

File: K:\Thesis\Test Cases\Mew\test_MEW.exe

Entrypoint: 0005BE1A
EP Section: ┐ÒuÛŠ┬ëÔ
File Offset: 0001301A
First Bytes: E9,35,43,FA
Linker Info: 0.0
Subsystem: Win32 console

MEW 11 1.2 -> NorthFox/HCC

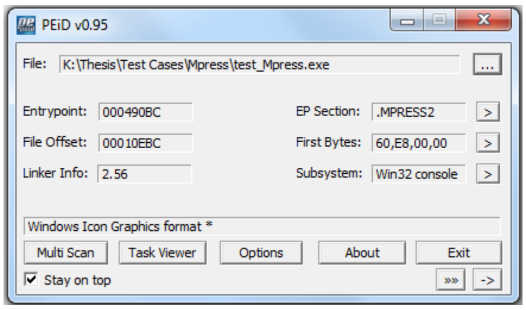
[✓] Stay on top

Figure 5.4: Scan of the Mew file

As we have seen so far, the packer follows the standard routine of decryption, restoration and returning to OEP. So far, most of these compressors are easy to unpack by a reverse engineer and in cases like this, all you need to do is scroll down through the code until a return code is seen. However, in cases of automatic unpacking tools, it is important to follow and understand patterns of end execution. Since we know that GetProcAddress is the last call that will be processed, every sign of a return or non-conditional jump may lead to the original entry point.

5.4 Mpress

In this paper, we have mentioned the fact that in some cases, certain packers will not be detected by scanning software. As it is the case with Mpress, the scanning software used yields non-conclusive results. Even though the


PEiD v0.95

File: K:\Thesis\Test Cases\Mpress\test_Mpress.exe

Entrypoint: 000490BC
EP Section: .MPRESS2
File Offset: 00010EBC
First Bytes: 60,E8,00,00
Linker Info: 2.56
Subsystem: Win32 console

Windows Icon Graphics format *

[✓] Stay on top

Figure 5.5: Scan of the Mpress file

memory section shows the Mpress name, it is easy to ”fake” the name of the section. However, as we have seen before PUSHAD command indicates that the ESP trick would be the best choice of approach. As we set the breakpoint on esp, we get to the return code rather quickly. Looking at the big picture, we see these codes:

```
0043ECF9      74 3D              JE SHORT test_Mpr.0043ED38
0043ECFB      03F8               ADD EDI,EAX
0043ECFD      56                 PUSH ESI
0043ECFE      E8 54000000        CALL <JMP.&KERNEL32.GetModuleHandleA>
0043ED03      8BD8               MOV EBX,EAX
```

Near the end of the decryption routine, the compressor loads the name of the module used in the original program (ex: kernel32.dll). Afterwards, it loads the function names using GetProcAddress:

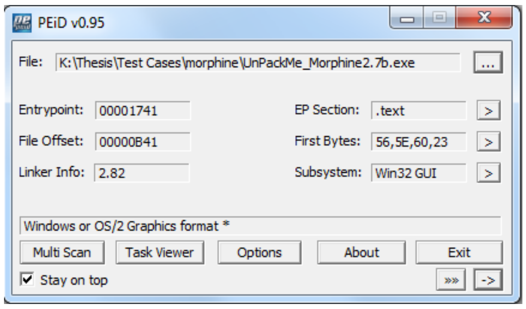
```
0043ED20      4E                 DEC ESI
0043ED21      C606 00            MOV BYTE PTR DS:[ESI],0
0043ED24      50                 PUSH EAX
0043ED25      53                 PUSH EBX
0043ED26      E8 32000000        CALL <JMP.&KERNEL32.GetProcAddress>
0043ED2B      AB                 STOS DWORD PTR ES:[EDI]
0043ED2C      32C0               XOR AL,AL
0043ED2E      8846 FF            MOV BYTE PTR DS:[ESI-1],AL
0043ED31      AC                 LODS BYTE PTR DS:[ESI]
0043ED32      0AC0               OR AL,AL
```

Finally after everything is loaded, all the registers are popped out of the stack and a jump to the OEP is made:

```
0043ED47      B8 1E010000        MOV EAX,11E
0043ED4C      AB                 STOS DWORD PTR ES:[EDI]
0043ED4D      61                 POPAD
0043ED4E      E9 CD24FCFF        JMP test_Mpr.00401220
```

5.5 Morphine

Morphine is a case of a polymorphic packer. Due to its nature, every time a file is compressed with morphine, the code changes and we will never obtain 2 similar compression routines. For this reason, obtaining a signature for the packer can prove to be quite difficult. After analysing the downloaded sample packed with Morphine, PEiD gives false results. After opening the


PEiD v0.95

File: K:\Thesis\Test Cases\morphine\UnPackMe_Morphine2.7b.exe

Entrypoint: 00001741
EP Section: .text
File Offset: 00000B41
First Bytes: 56,5E,60,23
Linker Info: 2.82
Subsystem: Win32 GUI

Windows or OS/2 Graphics format *

Figure 5.6: Scan of the Morphine file

file in Immunity, we try to set a breakpoint on GetProcAddress. However, after the imports are loaded, the return to the code section brings us quite far away from the OEP jump. However, we can still apply the ESP trick.

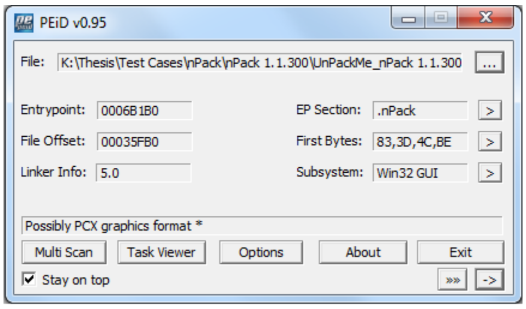
```
00521741 >    56                 PUSH ESI
00521742      5E                 POP ESI
00521743      60                 PUSHAD
00521744      23C0               AND EAX,EAX
00521746      F9                 STC
```

As we notice, the PUSHAD command will set all the registers into the stack. By following ESP and setting a hardware breakpoint on the first 4 bytes, we break near the OEP jump:

```
005211D5    5E              POP ESI
005211D6    5D              POP EBP
005211D7    83C4 04         ADD ESP,4
005211DA    5B              POP EBX
005211DB    5A              POP EDX
005211DC    83C4 08         ADD ESP,8
005211DF    894C24 04       MOV DWORD PTR SS:[ESP+4],ECX
005211E3    FFE0            JMP NEAR EAX        ; Jump to OEP
```

5.6 nPack and nsPack

These two packers are another example of commercial compressors. Developed by North Star, it offers more functionality and promises a more robust protection against reverse engineering. When opened in PEiD for scanning, nPack cannot be detected, while nsPack is shown immediately.



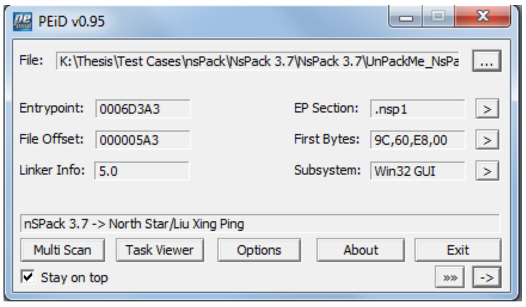
PEiD v0.95

File: K:\Thesis\Test Cases\nPack\nPack 1.1.300\UnPackMe_nPack 1.1.300

Entrypoint: 0006B1B0 — EP Section: .nPack
File Offset: 00035FB0 — First Bytes: 83,3D,4C,BE
Linker Info: 5.0 — Subsystem: Win32 GUI

Possibly PCX graphics format *

Figure 5.7: Scan of the nPack file



PEiD v0.95

File: K:\Thesis\Test Cases\nsPack\NsPack 3.7\NsPack 3.7\UnPackMe_NsPa

Entrypoint: 0006D3A3 — EP Section: .nsp1
File Offset: 000005A3 — First Bytes: 9C,60,E8,00
Linker Info: 5.0 — Subsystem: Win32 GUI

nSPack 3.7 -> North Star/Liu Xing Ping

Figure 5.8: Scan of the nsPack file

When opened in Immunity, both files are subjected to the ESP trick and soon enough in both cases we reach the OEP. What follows are extracts of code from both files:

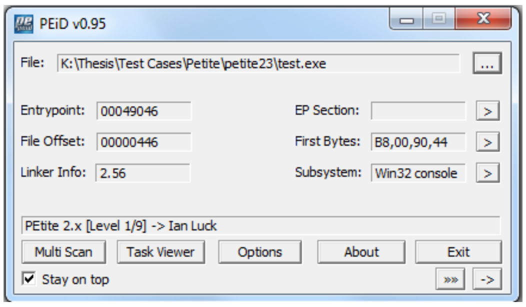
```
0046B1E3     E8  DD060000      CALL  UnPackMe.0046B8C5        ; nPack
0046B1E8     E8  2C060000      CALL  UnPackMe.0046B819
0046B1ED     A1  48BE4600      MOV EAX,DWORD PTR DS:[46BE48]
0046B1F2     C705  4CBE4600  0 MOV DWORD PTR DS:[46BE4C],1
0046B1FC     0105  00BE4600    ADD DWORD PTR DS:[46BE00],EAX
0046B202     FF35  00BE4600    PUSH DWORD PTR DS:[46BE00]
0046B208     C3                RETN                          ; Return to OEP
0046B209     C3                RETN
0046B20A     56                PUSH ESI                    ; Once we reach here set BP on ESP

0046D3A3 >   9C                PUSHFD                         ;nsPack
0046D3A4     60                PUSHAD
0046D3A5     E8  00000000      CALL  UnPackMe.0046D3AA

0046D614     9D                POPFD
0046D615   - E9  969BFBFF      JMP  UnPackMe.004271B0         ;jump to OEP
```

5.7 Petite

Petite is yet another compressors that manages to offer some extra options via the command line interface. Unlike previous compressors, Petite offers the option to check if the file is infected, which could be useful if a virus is packed using this compressor. We start out by packing our test file with Petite v2.3 and scan it with PEiD. Once the file is opened using Immunity


PEiD v0.95

File: K:\Thesis\Test Cases\Petite\petite23\test.exe

Entrypoint: 00049046
EP Section:
File Offset: 00000446
First Bytes: B8,00,90,44
Linker Info: 2.56
Subsystem: Win32 console

PEtite 2.x [Level 1/9] -> Ian Luck

Figure 5.9: Scan of the Petite file

Debugger, we try to set a breakpoint on the GetProcAddress API. However, interestingly enough, in our test case it never breaks. However, as we look closely to the entry point, we see that a few addresses down a PUSHAD command is encountered:

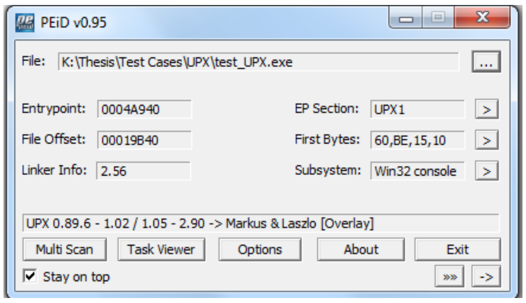
```
00449046     B8  00904400      MOV EAX, test.00449000
0044904B     68  90F14300      PUSH  test.0043F190
00449050     64:FF35  0000000>PUSH DWORD PTR FS:[0]
00449057     64:8925  0000000>MOV DWORD PTR FS:[0],ESP
0044905E     66:9C             PUSHFW
00449060     60                PUSHAD
```

As before, we follow ESP and set hardware breakpoint on access on the first 4 bytes. This trick allows us to reach the OEP jump after just a few breaks:

```
00449041     66:9D             POPFW
00449043     83C4  08          ADD ESP,8
00449046     E9  D581FBFF      JMP  test.00401220        ;jump to OEP
```

5.8 UPX

UPX is one of the most popular compression programs. Due to the fact that it is open source, it makes it easier for hackers to modify the algorithm and hide the signature of the packer, or in some cases, incorporate other packers within. In most cases when dealing with double or triple packed samples, UPX will be in one of the layers, due to its high versatility and compression algorithm. We pack our test file and after scanning it with PEiD we immediately get a result. The OEP jump lies after the call to



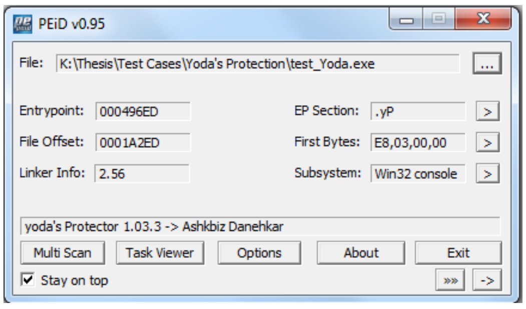
```
File: K:\Thesis\Test Cases\UPX\test_UPX.exe
Entrypoint: 0004A940        EP Section: UPX1
File Offset: 00019B40       First Bytes: 60,BE,15,10
Linker Info: 2.56           Subsystem: Win32 console
UPX 0.89.6 - 1.02 / 1.05 - 2.90 -> Markus & Laszlo [Overlay]
```

Figure 5.10: Scan of the UPX file

GetProcAddress API. As such, if we set a breakpoint on the function, we will reach the end jump after a few lines of code. The ESP trick would also work in this case, as the entry point of the compressor is a PUSHAD. Due to the fact that UPX code has already been explained in the previous section, we will not show the example code of UPX.

5.9 Yoda's Protector

Yoda's Protector is another example of an open source compressor. Unlike the previous packers, Yoda's Protector has an anti-debugging trick implemented: BlockInput. This function can pose problems when the application is debugged, as if the program is debugged, Yoda will check to see if the application is debugged and if it is, the function will block input from keyboard and mouse.


PEiD v0.95

File: K:\Thesis\Test Cases\Yoda's Protection\test_Yoda.exe

Entrypoint: 000496ED
File Offset: 0001A2ED
Linker Info: 2.56

EP Section: .yP
First Bytes: E8,03,00,00
Subsystem: Win32 console

yoda's Protector 1.03.3 -> Ashkbiz Danehkar

Figure 5.11: Scan of the Yoda file

Due to the nature of this protectors, the methods presented so far can be a bit more tricky to implement. Setting a breakpoint on GetProcAddress proves to be a bit useless, as it returns to different sections of the memory, without reaching any closer to the OEP. As such, a more complex algorithm needs to be developed in order to differentiate between sections of memory and setting breakpoints on the pages themselves.

6 Test Results

As we have seen from our tests, both methods of unpacking have worked on all 10 of our packers. Weather a variation of one of the packers presented will be used for future malware, the generic methods should still apply, as the base will follow similar rules. As such, pattern based detection is no longer required to detect and select the appropriate decryption routine. Hundreds of scripts are being written for each packer and their individual versions to not only detect them, but to lookout for end patterns and certain weak spots which leads to the OEP. However, by implementing such generic methods as the ones presented in this paper, we no longer need to load multiple scripts for each protection. Reverse Engineering Labs released their Titan Mist platform to aid researchers in unpacking malware samples. With that being said, their framework still relies heavily on scripts. Unless the framework has access to the script which describes how to deal with the problematic packer, even relatively easy compressors such as FSG cannot be unpacked. By using Titan Engine SDK, we are able to write our own unpacking engine which can incorporate our methods of unpacking and avoid the use of scripts.

6.1 Method Overview

Both methods have yielded concrete results and as we have seen, both methods have been successful in unpacking 9 out of our 10 samples. The table bellow shows what method was used for each packer and if the unpacking was successful:

Packers Name	ESP Trick	Breakpoint On API	Successful
AsPack	Yes	No	Yes
FSG	No	Yes	Yes
MEW	No	Yes	Yes
MPress	No	Yes	Yes
Morphine	Yes	No	Yes
nPack	Yes	No	Yes
nsPack	Yes	No	Yes
Petite	Yes	No	Yes
UPX	No	Yes	Yes
Yoda's Protection	Yes	Yes	No

Of course the unpacking engine should contain both methods and select each method for separate cases. However, in some cases the packer may need manual analysis in order to understand and develop other methods, as it was the case of Yoda's Protector. In the case of Yoda, a removal of the BlockInput check should first be inserted before further analysis begins. Also, other the engine needs to monitor any further anti-debugging techniques and remove them as it is necessary.

6.2 Implementation

The implementation of the engine is done utilizing C++ as primary programming language. The code is very straight forward and utilizes simple yet powerful commands which are meant to emulate a real debugger's step's. After the program is loaded into our engine, a check is made to see if the first 3 lines includes a command which pushes the registry's into the stack. If such a command exists, the ESP trick will be deployed against the packer. Otherwise, the engine sets a breakpoint on GetProcAddress and through a callback method, it will check to see if the return code contains known commands which signals the end of the decryption stub. Such commands can vary from popping all the registers from the stack to non-conditional jumps to addresses in the .text section. Titan Engine SDK comes prepared with a selection of commands which can emulate the steps taken in a debugger. The documentation provides explanations for functions such as : SetAPIBreakpoint. With this, we are able to create a breakpoint at the beginning or the end of a certain API and backtrack from there into the code section of the application's memory. The result code looks similar to this:

```
static void endOfAPI()
{
        long long rip = GetContextData(UE_EIP);
        unsigned char pattern[2] = {0xC9, 0xC2};    //patter of bytes which
                                                    //represents the return code
        BYTE wildcard = 0;
        long long found = Find((void*)rip, 0x1000, pattern, 2, &wildcard);
}
static void cbCallBack()
{
        long long rip = GetContextData(UE_EIP);
        StepInto((void*)endOfAPI);
        log("Context: %x", rip);
}
static void cbStepOver()
{
        long long eip = GetContextData(UE_EIP);
        eip -=11;
        SetBPX(eip,UE_BREAKPOINT,(void*)cbCallBack);
}
static void cbPutAPIBP()
{
        while(SetAPIBreakPoint("KERNEL32.DLL","GetProcAddress",
                                        UE_BREAKPOINT,UE_APISTART,
                                (void*)cbStepOver))
        {
                //Implementation of handling API calls
        };
}
```

Through this code, we are waiting for all the imports to be restored and initialize a callback to the end of the API. As we have seen from our tests, the jump to the OEP lies next to the call to GetProcAddress. Of course this is not always the case, some protections or compressors may process the packed binary file in other ways, or deploy anti-debugging tricks, which throws the jump to OEP far hidden into the code. The other method can also be achieved through the Titan's SDK. By monitoring the change of ESP register, we can set hardware breakpoints on the stack and implement the callback to search and step into end routines. Patterns of bytes can be established to detect procedures such as jump to hard coded locations, return to addresses in the stack and so forth. A simple pattern implementation may look something like this:

```
int dtPattern5Size = 2;
BYTE dtPattern5[2] = {0x61,0xE9};
void* dtPattern5CallBack = &cbEntryPoint;
void* dtPattern5BPXAddress = NULL;

int dtPattern51Size = 4;
BYTE dtPattern51[4] = {0x83,0xEC,0x00,0xE9};
void* dtPattern51CallBack = &cbEntryPoint;
void* dtPattern51BPXAddress = NULL;
```

7 Discussion

What we can gather from our findings is that it is indeed possible to implement a rather simple yet effective generic unpacker to deal with common compressors. As previously mentioned in section 3, there have been attempts to build such tools which can handle a multitude of compressors and packers, however the engine behind them lacked the necessary features to accomplish this task. Furthermore, these tools are mainly built by developers outside of the "research scope", who's main purpose is to defeat commercial protections and aid in the creation of pirated material. With these things in mind, one also has to note that such occurrences are unavoidable, though a step needs to be taken by researchers to develop tools which aid in the unpacking process. However, there are protections which cannot be defeated by simple generic means and further investigations needs to be done before an unpacking subroutine can be implemented. As such, the need for scripts still exists, which should be made for custom protections and multi-layered packers. Generic unpacking can be achieved through higher means of actualization and memory surveillance, however we are not yet there in terms of reliable tools.

8 Conclusion

The importance of generic unpacking has never been more obvious. As the numbers of custom packers grows each day, it is hard to keep up with them and update existing scripts. Many of the free alternatives for generic unpacking still relies on scripts, while the commercial tools have started to implement sophisticated methods to obtain a clear binary file for analysis. We have seen from this paper that it is possible to implement an engine which is not dependent on scripts or patterns for decryption. As more advances are made in this field, tools such as OmniUnpack promises a new type of tool which can unpack binary files by following tracking their behaviour and system calls, signalling activity from the main program and not the packer's code. However, while these tools are yet to become widely available and easy to use, it is good to know that basic generic unpacking techniques can be incorporated into simple engines that not only are easy to use, but rather fast in their process. The methods presented in the paper can be used to quickly go through multiple packers and obtain a clean binary dump in a matter of seconds. For future work, a custom unpacking engine can be developed, using Titan Engine SDK, which could further prove the usability of these methods. As it stands now, only a short version of the prototype has been built, while most of the code snippets where done as an example to how the engine could be built and how the implementation may look like. As such, with a little more research, the engine could provide an alternative for malware analysts to use when dealing with packed binary files.

References

[1] Teodoro Cipresso, *Software reverse engineering education* M.S. thesis, Dept. CS, San Jose State Univ. San Jose, CA 2009.

[2] Gaith Taha, *Counterattacking the packer* McAfee Avert Labs, Alyesbury, UK, 2007.

[3] Sha Sha Chu et al(2000), *Virus: A Retrospective*[Online] Available HTTP: cs.stanford.edu/people/eroberts/cs181/projects/2000-01/viruses/index.html

[4] AsPack 2.12 [Online]. Available: http://www.aspack.com/. [Accessed: 09- Mar- 2015].

[5] Fast Small Good [Online]. Available: http://fsg.soft112.com/. [Accessed: 09- Mar- 2015].

[6] Immunity Debugger [Online]. Available: http://debugger.immunityinc.com/. [Accessed: 09- Mar- 2015].

[7] ReversingLabs [Online]. Available: http://reversinglabs.com/open-source/titanengine.html. [Accessed: 02- May- 2015].

[8] Mew 11 [Online]. Available:
http://www.softpedia.com/get/Programming/Packers-Crypters-Protectors/MEW-SE.shtml . [Accessed: 09- Mar- 2015].

[9] MPRESS 2.19 [Online]. Available: http://www.matcode.com/mpress.htm . [Accessed: 09- Mar- 2015].

[10] Morphine 2.7 [Online]. Available:
http://www.delphibasics.info/home/delphibasicscounterstrikewireleases/polymorphiccrypter-morphine27byholyfather . [Accessed: 09- Mar- 2015].

[11] Ollydbg [Online]. Available: http://www.ollydbg.de/. [Accessed: 09- Mar- 2015].

[12] PEiD [Online]. Available: http://www.aldeid.com/wiki/PEiD. [Accessed: 09- Mar- 2015].

[13] Petite 2.3 [Online]. Available: http://www.un4seen.com/petite/. [Accessed: 09- Mar- 2015].

[14] IDA Pro [Online]. Available: https://www.hex-rays.com/products/ida/ . [Accessed: 09- Mar- 2015].

[15] W32Dasm [Online]. Available: http://www.softpedia.com/get/Programming/Debuggers-Decompilers-Dissasemblers/WDASM.shtml . [Accessed: 09- Mar- 2015].

[16] Tuts4You [Online]. Available: http://www.tuts4you.com . [Accessed: 09- Mar- 2015].

[17] Ultimate Packer for eXecutables [Online]. Available: http://upx.sourceforge.net/. [Accessed: 09- Mar- 2015].

[18] Yoda's Crypter [Online]. Available: http://sourceforge.net/projects/yodap/files/Yoda%20Crypter/1.3/. [Accessed: 09- Mar- 2015].

[19] Yoda's Protector [Online]. Available: http://sourceforge.net/projects/yodap/. [Accessed: 09- Mar- 2015].

[20] Tom Brosch and Maik Morgenstern, *Runtime Packers: The Hidden Problem?*[Online], Available HTTP: https://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Morgenstern.pdf

[21] Fanglu Guo et al, *A Study of the Packer Problem and Its Solutions*[Online], Avaliable HTTP: http://www.ecsl.cs.sunysb.edu/tr/TR237.pdf

[22] Adrian Stepan, *Improving proactive detection of packed malware*[Online] https://www.virusbtn.com/virusbulletin/archive/2006/03/vb200603-packed.dkb March, 2006

[23] Mihai Christodorescu et al, *OmniUnpack: Fast, Generic, and Safe Unpacking of Malware*, presented at *Computer Security Applications Conference, Miami Beach, FL, 2007, pp. 431 - 441*

[24] Paul Royal et al, PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware *presented at Computer Security Applications Conference, Miami Beach, FL, 2006, pp. 289 - 300*

[25] Mumtaz, S. et al, Development of a Methodology for Piracy Protection of Software Installations *presented at 9th International Multitopic Conference, Karachi, Pakistan, 2005, pp. 1 - 7*