Quick reference for manual unpacking

2012-04-01

Abhishek Singh

FireEye, USA
Editor: Helen Martin
Abstract

By packing their malicious executables, malware authors can be sure that when they are opened in a disassembler they will not show the correct sequence of instructions, thus making malware analysis a more lengthy and difficult process. Abhishek Singh provides a quick reference guide for unpacking malware from some of the most commonly used packers.

Table of contents

Malware authors utilize packers to make it difficult for their malware to be reversed – the packers encode the original instructions. By packing a malicious executable, its author can be sure that when it is opened in a disassembler it will not show the correct sequence of instructions. Packers add some instructions at the top of the binary to unpack the executable. The process of decryption is performed in memory at run time, and the state of the application is restored. Since packers work on a compiled executable, the unpacking module must be independent of the original application.

One of the methods that can be used to locate the original entry point (OEP) of the file is to apply break points on the following APIs:

```
GetLoadLibraryA
GetVersionExA
GetEnvironmentA
LoadLibraryA
GetProcAddress
IniHeap
```

These APIs are called by the packers' start-up routines in order to set up the execution environment. When a breakpoint is applied to these routines, we are close to the OEP. When the break point triggers, we can use step-by-step tracing to locate the initialization of the stack frame. The start of the function can be recognized by the initialization of the stack frame.

```
push ebp
mov  ebp, esp
```

The instructions shown above denote the start of the stack frame. Once these instructions are located, the debugged process can be dumped to obtain the unpacked version of the file.

In the following sections we describe some common packers and the assembly instructions that can be used to locate the OEP.

Manual unpacking

The purpose of this section is to provide a quick reference guide that will assist malware analysts in the unpacking of malware and reduce the response time for malware analysis – the full technical details of each packer have therefore been omitted.

ASPack

ASPack is an advanced *Windows 32* executable compressor capable of reducing the file size of 32-bit *Windows* (*95*/*98*/*ME*/*NT*/*2000*/*XP*/*2003*/*Vista*/*7*) programs by as much as 70%. It is also used by some hackers to protect their programs.

To unpack ASPack, follow the first jmp, and follow JMP EAX. Later in the code you will find the following instructions:

```
mov  eax,1
retn 0C
push 0
retn
```

Once these instructions have been identified, as shown in **Figure 1**, a break point should be put on RETN. When the break point triggers, we are at the OEP. The process can be dumped at this stage, leaving us with the unpacked executable.

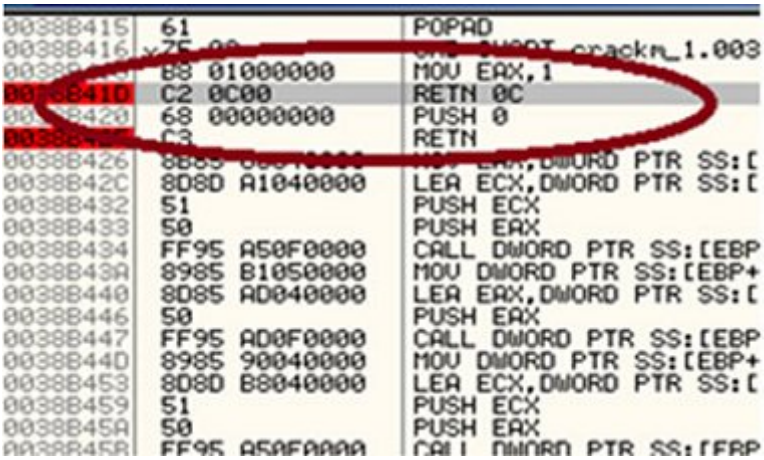

Figure 1. Instructions before the code is unpacked.

OllyScript code for the automatic unpacking of ASPack is shown in **FIgure 2**. The instruction 'findop eip, #6800000000#' locates the PUSH 0 instruction in a debugged process packed with ASPack. Once this instruction is located, the debugger steps once to reach the RETN instruction. The debugger then steps again to reach the OEP instruction. Once the OEP instruction is located the debugger steps once more to reach the OEP. The debugged process can now be dumped to get the unpacked version of the file.

```
findop eip, #6800000000#
go $RESULT
sti
sti
msg "OEP is found for ASPack "
run
```

Figure 2. OllyScript code used to locate the OEP for ASPack.

KKrunchy

KKrunchy [**1**] is a small executable packer intended for 64k intros. It does not try to pack DLLs and cannot handle exports or TLS. It performs a transform on the input code to allow it to compress better. It will fill uninitialized data sections with zeros and then pack them together with the rest of the code. KKrunchy is often used by malware authors to prevent AV analysts from reversing their code.

In order to unpack KKrunchy, put a break point on LoadLibraryA. When the break point triggers, step the debugger and search for the initialization of the stack frame. Once the stack frame initialization is complete, dump the debugged process. The dumped process is the unpacked version of the executable.

PECompact v2.x

PECompact [**2**] is fully compatible with DEP and code signing, and provides support for *Windows 7* and *Windows 2008*. It provides a good compression ratio compared to other compressors such as ASPack. The PECompact [**3**] loader consists of three components. The first is the SEH entry, which transfers control to the second component, the loader decoder. The loader decoder decodes the code and invokes the third component, the primary loader. The loader decoder is stored in the last section (or the second-to-last section if relocations have been preserved). The primary loader exists in uncompressed form at runtime in dynamically allocated memory. To hide the transfer of control, an SEH frame is set up and there is an exception. The exception handler then modifies the code at the exception address to a JMP and continues execution.

Figure 3 shows PECompact's exception handler. The instruction sequence 'PUSH EAX, PUSH DWORD PTR FS:[0], MOV DWORD PTR FS:[0], ESP' sets up the SEH frame. The instruction 'XOR EAX, EAX' sets the value in EAX to zero. The instruction 'MOV DWORD PTR DS:[EAX], ECX' triggers the exception.

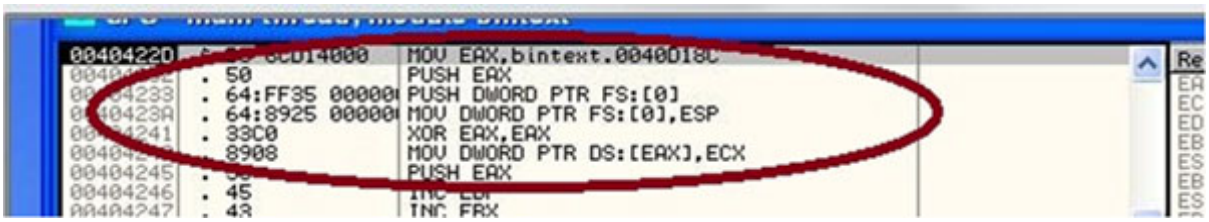

Figure 3. The PECompact exception handler.

To unpack PECompact, follow the exception and step through the code until the instructions shown in **Figure 4** are observed. JMP EAX is the jump to the OEP.

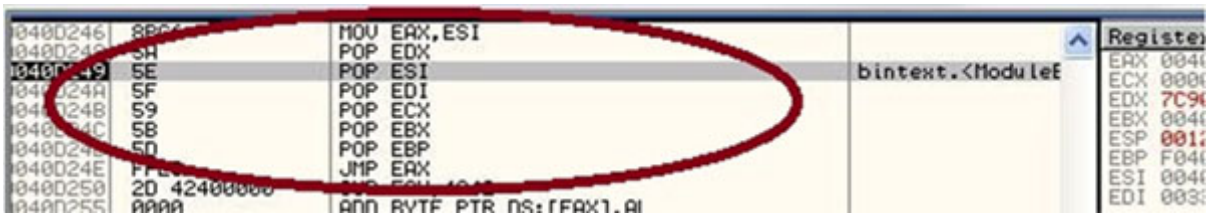

Figure 4. PECompact instructions before unpacking.

Set a break point on JMP EAX, step once, and observe the initialization of the stack frame as shown in **Figure 5**. Dump the process. The dumped process will be the unpacked executable.

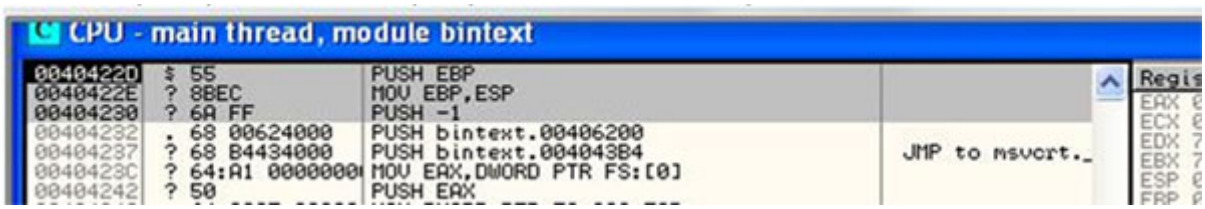

Figure 5. Initialization of the stack frame.

The logic shown in **Figure 4** can be converted into script such as that shown in **Figure 6** (the script is available from Open RCE [**4**]).

```
sto
sto
sto
sto
sto
sto
sto
esto
find eip, #8BC65A5E5F595B5DFFE0#
add $RESULT,08
bp $RESULT
run
bc $RESULT
sto
cmt eip,"This is a OEP!"
msg "OEP found, Dumped and fix IAT now!"
ret
```

Figure 6. OllyScript for PECompact.

The instruction 'find eip, #8BC65A5E5F595B5DFFE0#' locates the instructions 'MOV EAX ESI, POP EDX, POP ESI, POP EDI, POP ECX, POP EBX, POP EBP, JMP EAX'. Once these are located, the script steps once at the JMP instruction and the debugger is at the OEP. The debugged process now can be dumped to obtain the unpacked version of the file.

NSPack

NSPack [**5**] is capable of compressing EXE, DLL, OCX and SCR files. It also has the ability to compress 64-bit executables. It provides support to compress files packed by other packers such as UPX, ASPack and PECompact. It supports direct compression of directories or multiple files. This packer is quite commonly used by malware authors.

As shown in **Figure 7**, the packer starts with the instructions PUSHFD, PUSHAD.

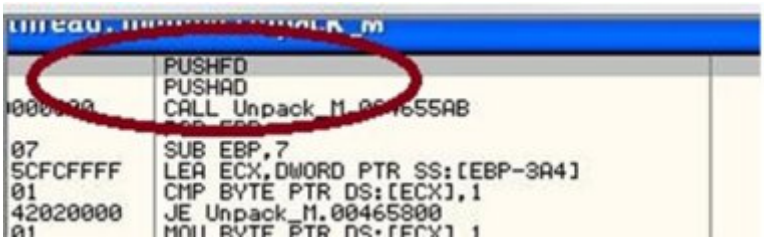

Figure 7. The starting instructions for NSPack.

Check for equivalent POPAD and POPFD instructions, as shown in **Figure 8**. The JMP instruction follows. Put a break point on the JMP instruction. When the break point triggers, step once and dump the process to obtain the unpacked file.

Figure 8. NSPack instructions before unpacking.

The abovementioned logic can be converted into the OllyScript shown in **Figure 9**. The instruction 'find eip, #619DE9#' locates the instruction POPAD, followed by POPFD, followed by a JMP instruction. Once these are located, the code is debugged, step by step, until the JMP instruction is executed – the debugger has then reached the OEP instruction. By using a plug-in like OllyDump, the process can be dumped to obtain the unpacked version of the file.

```
sti
find eip, #619DE9#
go $RESULT
sto
sto
sto
msg "OEP for NSPack "
```

Figure 9. OllyScript used to locate the OEP for NSPack.

FSG 1.33

FSG stands for Fast Small and Good, and is currently used to pack various malware. It was originally created to pack assembly demos. Since it has a small loader, it is one of the most desirable packers for small executables.

In order to obtain the unpacked executable file for FSG 1.33, put a break point on the LoadLibraryA function, as shown in **Figure 10**.

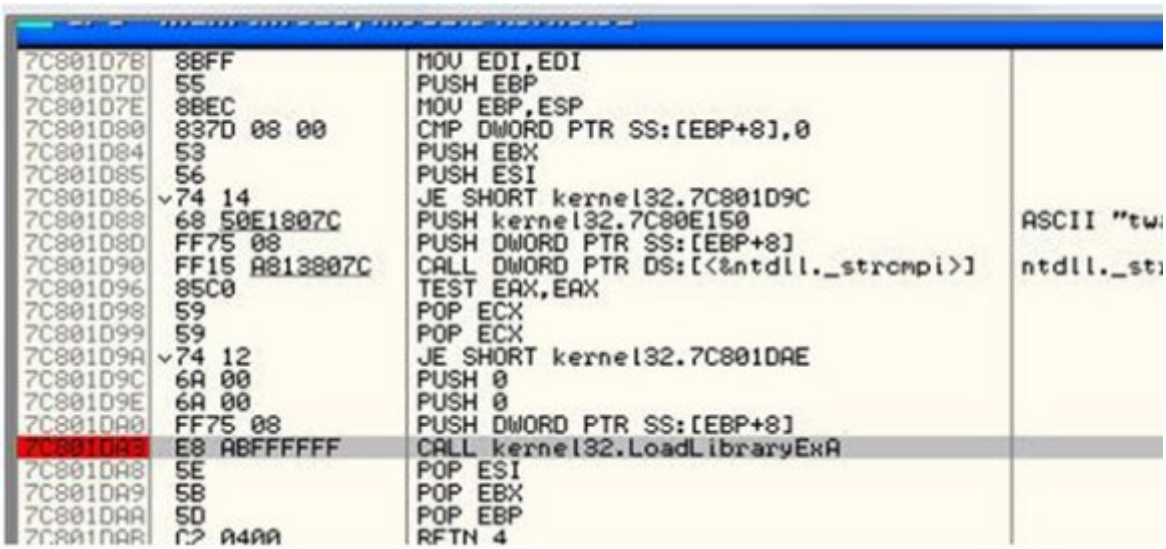

Figure 10. The LoadLibraryA function in FSG 1.33.

When the break point triggers, step a few instructions below until the following instructions are seen:

```
dec  byte ptr [esi]
jz   xxxxxxxx
PUSH ESI
PUSH EDP
CALL DWORD PIR D5:[EBX=4]
```

When JE Address triggers, we can observe the initialization of the stack frame. We are at the OEP, so dump the process to get the unpacked version of the file.

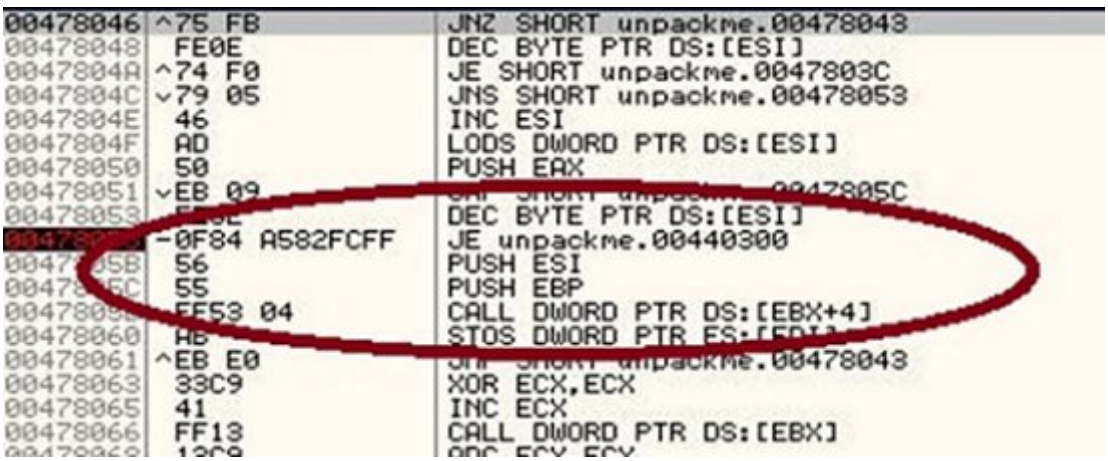

Figure 11. Instructions denoting the end of FSG.

FSG 2.0

For version 2.0 of the FSG packer, the instructions that indicate the end of the FSG stub are as follows:

```
move eax (edi)
inc  eax
js   address
jnz  address
jmp  dword ptr [ebx+0Ch]
```

In order to manually unpack a file packed with FSG 2.0, put a break point on LoadLibraryA and execute the compressed file. When it breaks, clear the break point and execute until return (Ctrl -f9). Step through the debugged application until the instructions shown in **Figure 12** are reached.

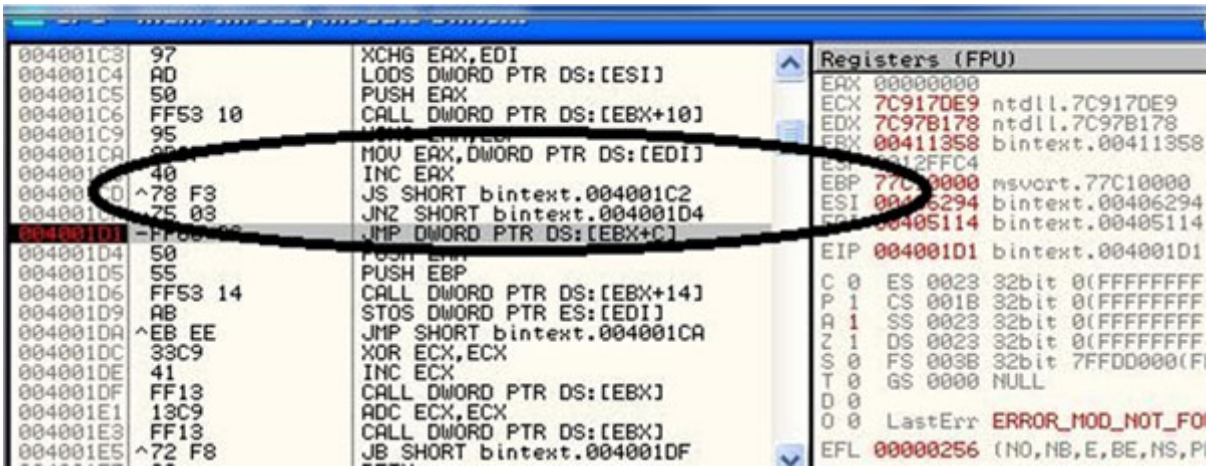

Figure 12. Instructions reached before unpacking FSG 2.0.

Here, 'JMP DWORD PTR DS: PTR [ebx+0Ch]' is the jump to OEP. Once the JMP instruction is executed, dump the process to get the unpacked version of the file.

UPX

UPX [**6**] stands for Ultimate Packer for eXecutables. It offers an excellent compression ratio which is better than WinZip, Zip and GZIP. It also maintains a checksum for both compressed and uncompressed files. It uses compression algorithms like UCL [**7**]. UCL has the inherent advantage that the decompressor can be implemented in a few hundred bytes of code. Many malware families such as Qakbot are packed using UPX. It offers very fast compression and decompression speeds: ~10MB/s on a *Pentium* 133. It also offers support for LZMA compression and has support for BSD. LZMA decompression is disabled on the 16-bit platform due to the slow decompression speed on older platforms. It also provides support for two types of decompression routines. The first is the in-place technique, which decompresses the executable in memory. In-place decompression is possible only for some platforms. The extraction of a temporary file, even though it uses extra overhead, allows any executable file format to be packed.

In order to unpack UPX using a manual approach, the end of the UPX routine must be identified. The end of the UPX routine can be identified by the instructions CALL, POPAD and JMP, as shown in **Figure 13**. Put a break point on the JMP instruction. The JMP instruction will lead to initialization of the stack frame. After the JMP instruction has executed, dump the process by using a plug-in such as OllyDump, and the program is unpacked.

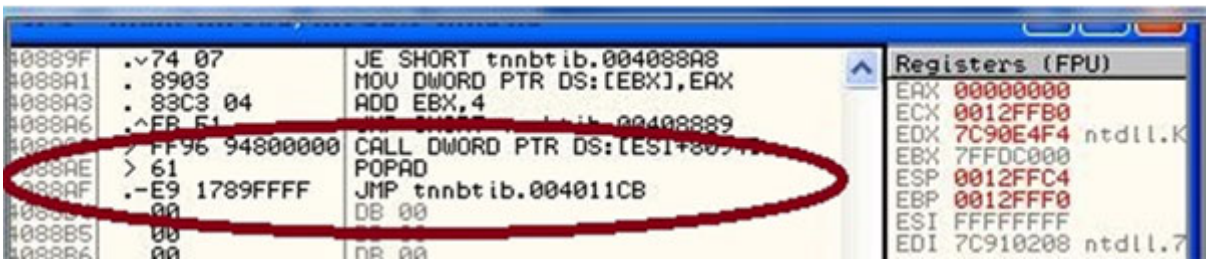

Figure 13. UPX end of routine instructions.

```
var BPforPOPAD
var BPforJMP
findop eip, #61#
mov BPforPOPAD, $RESULT
bp BPforPOPAD
run
findop eip, #E9????????#
mov BPforJMP, $RESULT
bp BPforJMP
run
sti
msg "OEP for UPX. Dump the Process"
```

Figure 14. The OllyScript used to unpack UPX.

The script shown in **Figure 14** is the implementation of the logic used to locate the OEP. The instruction 'findop eip, #61#' locates the assembly instruction POPAD, sets a break point on it, and then executes the code packed with UPX. Once the break point is triggered, the instruction 'findop eip, #E9????????#' locates the JMP instruction and sets a break point on it. When the break point triggers, the debugger steps once in the code and is at the OEP. The debugged process can be dumped to get the unpacked version of the file.

PEDiminisher

PEDiminisher is a simple PE packer. It uses the aplib compression/decompression library. Many AV engines have the ability to unpack files packed with PEDiminisher to check for malicious content.

The end routine for PEDiminisher is shown below:

```
pop EBP
POP EDI
POP ESI
POP EDX
POP ECX
POP EBX
JMP EAX
```

For unpacking, the end instructions must first be located in the packed file (as shown in **Figure 15**). JMP EAX is the jump to the OEP. Set a break point at the JMP instruction, step once and then dump the process to get the unpacked version of the file.

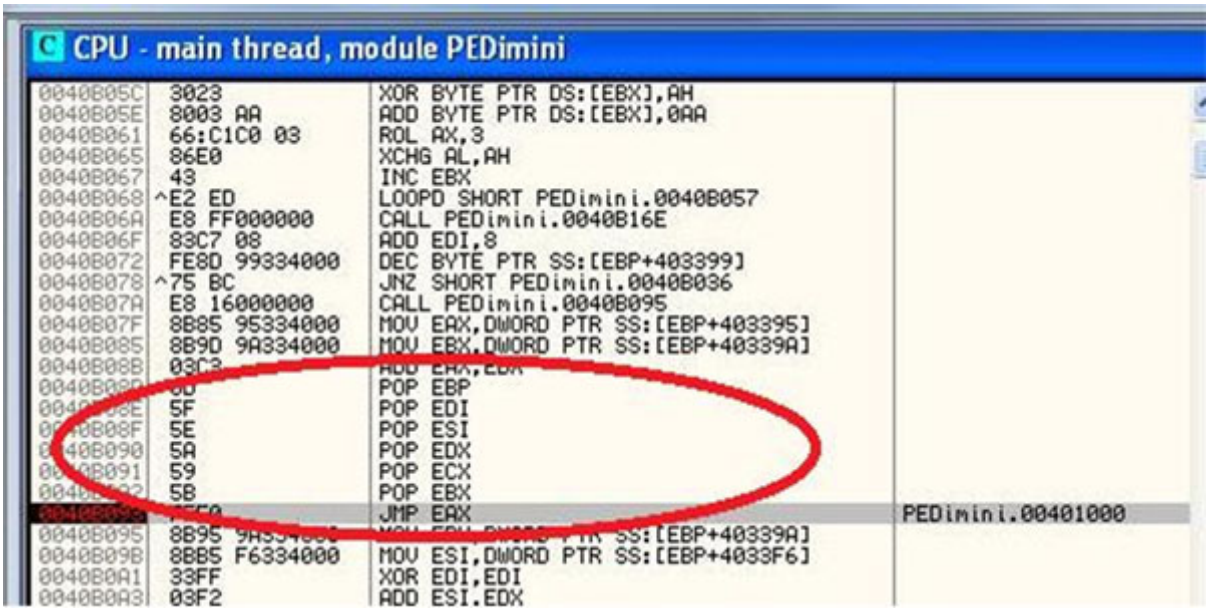

Figure 15. The end instruction for PEDiminisher.

```
find eip, #5D5F5E5A595BFFE0#
bp $RESULT
run
sti
sti
sti
sti
sti
sti
sti
sti
msg "OEP found for PEDiminisher, Dump now!"
```

Figure 16. The OllyScript used to unpack PEDiminisher.

The instruction ‘find eip, #5D5F5E5A95BFFE0#’ locates the instructions ‘POP EBP, POP EDI, POP ESI, POP EDX, POP ECX, POP EBX, JMP EAX’. The script then steps through the debugger until it reaches JMP EAX. Once it is at JMP EAX, the code steps once and is at the OEP. The OllyDump plug-in can be used to dump the process and we are left with the unpacked version of the executable file.

MEW

MEW [**8**] is an executable tool which was designed to handle small files. It works on 32-bit workstations and uses the LZMA algorithm. It strips reloc tables, Delphi resources, and unused resources. Even though it was designed to handle small files, it can compress large files as well.

The last instruction in the MEW stub, as shown in **Figure 17**, is RETN. After this instruction a jump to the OEP takes place. Set a break point on the RETN instruction. When the break point is triggered, as shown in **Figure 17**, step once and then dump the process to get the unpacked version of the file.

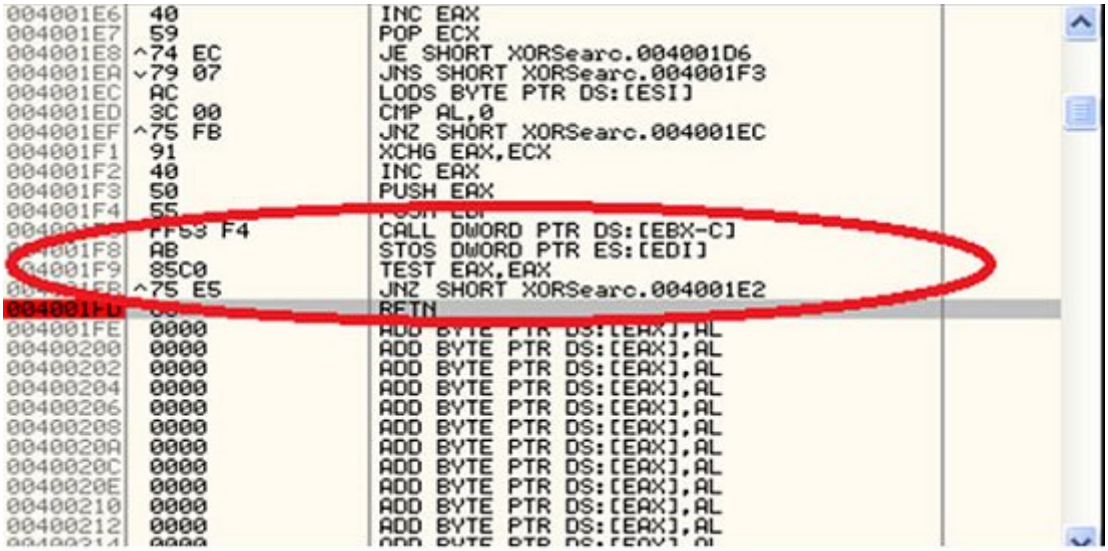

Figure 17. The last instructions for the MEW packer.

The logic used to locate the OEP for MEW is shown in **Figure 18**. The code 'findop eip, #C3#' locates the RETN instruction in the debugged process packed with the MEW packer. Once the RETN instruction is located, the debugger steps once and is at the OEP. The OllyDump plug-in can be used to dump the process and we are left with the unpacked version of the executable file.

```
sti
findop eip, #C3#
go $RESULT
sto
sto
msg "OEP found for MEW"
```

Figure 18. The OllyScript used to unpack MEW.

Conclusion

Reducing the time it takes to perform malware analysis is very important. For static analysis of malware it is important that the malware is unpacked. There are many approaches to unpacking a piece of malware – for example, it can be executed in a virtual environment and then we can capture a memory snapshot of the executing malware. Once we get the snapshot, we can dump the unpacked malware directly from memory. However, it is possible that not all of the code of the unpacked malware will be in memory, so dumping a process from memory might not be an effective unpacking method. Loading a packed malicious executable and executing step by step instructions in a debugger is one of the best ways to locate the OEP and execute the malware. In this article we have provided assembly instructions for the most commonly used packers which can be used to quickly unpack malware. We have also provided OllyScripts for the logic to manually unpack the malware. This can further aid in reducing response time for malware analysis.

Bibliography

[1] **http://www.farbrausch.de/~fg/kkrunchy/ (http://www.farbrausch.de/~fg/kkrunchy/)**.

[2] **http://pecompact.com/pecompact.php (http://pecompact.com/pecompact.php)**.

[3] **http://www.bitsum.com/pec2av.htm (http://www.bitsum.com/pec2av.htm)**.

[4] **http://www.openrce.org/downloads/details/156/PECompact_v.2.40_-_OEP_finder (http://www.openrce.org/downloads/details/156/PECompact_v.2.40_-_OEP_finder)**.

[5] **http://nspack.download-230-13103.programsbase.com/ (http://nspack.download-230-13103.programsbase.com/)**.

[6] **http://upx.sourceforge.net/ (http://upx.sourceforge.net/)**.

[7] **http://www.oberhumer.com/opensource/ucl/ (http://www.oberhumer.com/opensource/ucl/)**.

[8] **http://www.softpedia.com/get/Programming/Packers-Crypters-Protectors/MEW-SE.shtml (http://www.softpedia.com/get/Programming/Packers-Crypters-Protectors/MEW-SE.shtml)**.

Latest articles:

Throwback Thursday: Olympic Games (/virusbulletin/2016/august/throwback-thursday-olympic-games/)

In 1994, along with the Olympic Games came an Olympic virus, from a group of Swedish virus authors calling themselves 'Immortal Riot'. Mikko Hyppönen had the details.

Throwback Thursday: Holding the Bady (/virusbulletin/2016/07/throwback-thursday-holding-bady/)

In 2001, 'Code Red' caused White House administrators to change the IP address of the official White House website, and even penetrated the mighty Microsoft's own IIS servers. In August 2001, Costin Raiu analysed the Win32/Bady.worm,

The Journey of Evasion Enters Behavioural Phase (/virusbulletin/2016/07/journey-evasion-enters-behavioural-phase/)

No malware author wants their piece of code to be easy to detect. Over time, several different approaches have been put into action to detect malware, and in response, malware authors have put into action different methods of evading them. This paper…

Throwback Thursday: You Are the Weakest Link, Goodbye! - Passwords, Malware and You (/virusbulletin/2016/07/throwback-thursday-you-are-weakest-link-goodbye-passwords-malware-and-you/)

Have you heard the one about the computer user who used their pet's name as their password? Just like jokes, it seems the old ones and the obvious ones are considered the best when it comes to users

selecting their passwords. Martin Overton looks at…

New Keylogger on the Block (/virusbulletin/2016/07/new-keylogger-block/)

This paper provides an overview of the KeyBase trojan, both the keylogger itself and the server-side management component. Additionally, we will look at an example of when this trojan was used.