Malware Analysis

DSO National Laboratories
Information Division
Computer Security Lab (CSL)

Chua Xinhui Sarah
Joyce Yeo Shuhui
Under the supervision of: Ong Chee Eng

Contents

1 Introduction

Utilising static and dynamic analysis methods allow malware analysts to understand the function and inner workings of malware. This information can be useful for malware detection, classification, mitigation or countermeasure development, damage assessment and attribution. However, malware authors do not want their malware to be easily understood so that their criminal operations can run longer undisturbed. As a result, they employ many techniques which make their binaries harder to understand, such as anti-debugging, packing, altering the structure of the binaries, and running anti-VM checks.

2 Objectives

This project aims to study the techniques commonly employed by malware authors to make their binaries harder to analyse, as well as analyse and develop some tools and procedures to overcome them.

3 Approach

Firstly, we conducted a literature survey (Section 4) to find out the current state of static and dynamic analysis, and their respective anti-analysis and anti-forensic techniques. We then selected interesting directions to work on, namely:

1. Packer Detection Techniques (Section 5.1)
2. Tools and Procedures for Unpacking Binaries (Section 5.2)
3. PE Malformation (Section 5.3)
4. Anti-Debugging (Section 5.4)
5. Anti-Virtual Environment (Section 5.5)

4 Literature Survey

Malware analysts often obtain malware samples from post-security breach events or by setting up honeypots at gateways. These samples are then subjected to various analytic techniques. We break down the analysis techniques to two classes: static analysis and dynamic analysis; and further look at how malware authors employ anti-analysis techniques.

Anti-forensic techniques make it difficult for analysts to obtain malware samples. On the contrary, anti-analysis techniques make it difficult for analysts to understand the behaviour of malware samples. We focus on the anti-analysis component in the report.

4.1 Static Analysis

Analysts use static analysis to examine the suspicious binary without executing it, negating the risks associated with running the file. Static techniques can give quick snapshots of the malware as a prelude to performing more in-depth dynamic analysis.

- Detecting signatures: Anti-virus software detect signatures in the malware body and match these signatures to known classes of virus. These signatures usually appear as byte sequences or sets of byte sequences. Most of these signature matching can be done against online databases like VirusTotal and HybridAnalysis. However, when faced with confidential malware that should not be publicly uploaded, malware analysts can instead calculate hashes like the MD5 hash of the malware, and match these hashes.

  Anti-virus signature detectors are not foolproof because some zero-day malware have unknown signatures. Additionally, benign executables can also be labelled as malicious if these executables use packers, a ubiquitous anti-analysis technique used by malware. However, packers also have legitimate purposes, such as being used to reduce the disk space that legitimate software take up.

- Displaying strings: The Sysinternals Strings module can be used to look up embedded substrings. Strings like URLs can be useful in conveying information about the malware's originating C2. Besides Sysinternals Strings, strings can also be displayed in other tools like Binary Ninja and IDA Pro.

- Decompiling/disassembly: Most malware samples are binaries (eg. Windows uses the .exe, .dll, .sys extensions; Linux uses the Executable and Linkable Format (ELF) instead). These binary samples do not have any source code, making it difficult to examine them statically. A simple translation to assembly code can be done using disassemblers like IDA Pro (The Interactive Disassembler) or Ghidra (an open-source RE tool released by the NSA in March 2019). IDA and Ghidra also offer the option to decompile the assembly code to source code in C/ C++. However, decompilation is a difficult task that is often incomplete and sometimes unreliable. Furthermore, static disassemblers like Binary Ninja or IDA Pro will not expose data generated by the malware at runtime (for instance, indirect calls.)

- Decompiling to intermediate representation (IR): Binaries can be converted to intermediate representation (IR) which is a representation of the binary in the form of either a graph, tuple-based code or stack code which exists below the target language (could be assembly or a higher level language). Decompiling to IR can be done with, for example, the LLVM compiler. By converting the machine code to IR, control flow analysis or data flow analysis can be made easier.

- Import tables: The malware's import information can be found by loading the binary into parsers such as IDA Pro and Ghidra. The import table provides information on the dependencies of the malware, such as the functions used and their DLLs.

4.2 Anti-Static Analysis

Most malware authors recognise the common static analysis techniques used in malware analysis. Correspondingly, there is a host of techniques to thwart static analysis attempts. These attempts include:

- Fileless Malware: Fileless malware are memory-based artefacts (RAM) which do no exist on the disk, though they may store their payloads on-disk such as encrypted within the Windows Registry. As the malware which executes the payload itself is not on-disk, it will be automatically erased once the attack is complete, hindering evidence retrieval.

  The loading into memory is done using VirtualAllocEx function (from memoryapi.h). Fileless malware use in-built Windows tools, most notably Powershell, for malicious activity. Some common offensive PowerShell frameworks include Empire, PowerSploit, Metasploit and CobaltStrike. These frameworks allow attackers to quickly create attack payloads. A common delivery method for fileless malware is by inducing network activity with HTTP requests.

  Fileless malware executing living-off-the-land attacks are dangerous because antivirus software often whitelist the activity from trusted Windows applications like Powershell and WMI.

  Injected code in fileless malware may not even have a correct MZ header format.

- Packing: Packers use algorithms to manipulate the format of the executables, while still adhering to the PE format. There are numerous known packers like UPX and Armadillo. Packers employ some unique techniques in their algorithm, but there are also some commonalities. Packing compresses the code using lossless data compression algorithms and also destroys the IAT (Import Address Table) of the executable, obfuscating data as well as preventing software such as IDA Pro from decompiling it.

  LoadLibraryA and GetProcAddress are widely used in packing algorithms, because after unpacking the original code and reading the original import information, the unpacking stub uses them to load the DLLs needed and point to the correct functions respectively. Packers can also tweak the PE files by moving the functional code to the data sections and renaming sections.

  In turn, malware analysts can use tools like PEiD to check for packers. Instead of directly detecting specific instances of packers, it would make more sense to attempt to detect classes of packers. That way, small changes in the code (which drastically change the hashing signatures) can still be detected. Christodorescu et al used templates to identify malware based on their unpacking mechanisms, such as decryption loops. Unfortunately, this method still requires the use of a template.

With proprietary packers, it is harder to extract the packing signatures. In these scenarios, the analyst can execute the malware and periodically use process dumps, with the aim of breaking when the file is unpacked. Of course, malware authors also design techniques to make process dumps difficult (Section 5.4.5).

- Encryption: The body of the malware can be encrypted, but this means the malware must be paired with a decrypter. Anti-virus software can then attempt to trace patterns in the decrypters instead of in the malware. Some malware carry multiple decrypters, so anti-virus detectors can either store a database of the possible decrypters, or perform dynamic decryption by emulating the viral code decryption process. Besides changing the decrypters, malware can also change the viral body in each generation, making pattern detection challenging. However, having the malware change the viral body is a computationally expensive and uncommon technique.

- Domain Generation Algorithm (DGA): Malware can use a DGA to create the address of the C2, instead of directly writing the IP address or URL. DGAs use generators to create variant paths of Top-Level Domains (TLDs).

- Slack spaces: Files are usually stored in clusters in the NTFS file system. Malware can store data in the slack space of an existing file, whose existing data does not fully occupy the full cluster space. The malicious data cannot be overwritten unless the file in which it is stored is overwritten.

- Alternate Data Streams (ADS): ADS allows malware authors to hide their payload in a manner which is not easily detected. This is because adding data to the ADS will not increase file size or change its functionality, making it hard to detect. ADS applies only to the Windows NT File System (NTFS). An ADS may be created by appending the name of the data stream you want to create with a colon (':') to the back of the file name. Although ADS are hard to detect, analysts can use the Streams tool from Sysinternals to check for the presence of ADS.

4.3 Dynamic analysis

Sometimes, malware generate artefacts and traces only when they are run. Dynamic analysis allows for the execution of suspicious binaies in a controlled environment, allowing analysts to get valuable information on their behaviour and function.

- Sandboxing software: Using sandbox software such as Cuckoo Sandbox, the malware be analysed in isolated environments, protecting the host system from being infected. Sandboxes run the malware in virtual machines (VMs), then produce log files and reports detailing changes observed in the virtual environment. New processes run, changes to the files and registry settings and network activity can be captured. Furthermore, sandboxes can keep snapshots of the virtual environments which makes rolling back to a clean state simple.

- Log Analysis: Tools like Task Manager capture changes to the file system, registry, network and processes.

  Process Explorer, another tool used by analysts, can display the current processes and the DLLs they run; for instance, the presence of ws2_32.dll indicates the use of network processes. Process Explorer also provides a "Verify Option" to check if a binary is signed by MS.

  Process Monitor can also be used to identify the system calls made by the suspicious binaries. Common system calls made by malware include "WriteFile" and "RegSetValue". The analyst can view the log of activities relevant to files, processes, registry and the network. It is paramount to use effective filters to capture salient information sets, since Process Monitor captures all the activity.

- Debugging: Analysts can use debuggers to insert breakpoints and debug the behaviour of malware, looking at the registers involved and control flow invoked. Debuggers are useful as they allow analysts to inspect step-by-step when the malware is executing. Instructions can also be dynamically modified to change the control flow of the malware - changing the EIP instruction pointer or replacing instructions with NOP (no operation).

  Common debuggers include IDA Pro and OllyDbg (x86 only).

- Dumping. There are generally two forms of dumping that could be used by analysts.

  Memory dumping. Tools like Process Dump from Sysinternals can be used to dump the process memory to understand crashes.

  Executable dumping. Tools like OllyDump can be used to dump snapshots of the executable at runtime.

4.4 Anti-Dynamic Analysis

Anti-dynamic analysis methods generally includes anti-debugging and the ability of malware to detect virtual environments.

- Anti-debugging: When using the debugger with a packed malware file, the analyst uses process dumps to retrieve the unpacked file. However, there are several methods used by malware to make the debugging process difficult.

  1. Checking for the presence of debuggers using IsDebuggerPresent, CheckRemoteDebugger, FindWinow. When the malware detects a debugger present, it can either delay its execution or self-destroy

  2. CsrGetProcessId returns the process ID of csrss.exe, a system process. A process that can load csrss.exe is loaded by a debugger and has the SeDebugPrivilege privilege enabled in the access token.

  3: Self-debugging is used to prevent another debugger from being attached to the parent process. It relies on the simple fact that only one debugger can be attached to a process at a time. Self-debugging works by creating a process, which then attaches to itself by acting as a debugger. Armadillo, which is a software protection system, performs this technique using its nanomite feature. This prevents the software analyst from being able to use a debugger to perform dynamic analysis on the executable.

  4: Malware developers can change the value of the SizeOfImage variable stored within the PEB (Process Environment Block), preventing debuggers to be attached to the process and preventing memory dumpers from working correctly. This techniques relies on specific knowledge of the debuggers using methods like VirtualAlloc and ReadProcessMemory.

  5: Latency comparison: When a process is executed in a debugger, there is a larger latency between two instructions. This latency measured to detect the existence of breakpoints set by debuggers which slow the overall program execution time.

- Requiring interactive user input. This technique prevents malware from running independently in the sandbox, therefore making analysis difficult.

- Detecting virtual environments: By testing for virtual environments such as KVM, VMWare, VirtualBox, etc, certain malware will delay their payload so analysts are unable to observe its behaviour. For example, malware can test for the VMWare environment using these methods:

  1.MAC Address check: The MAC address of the virtual network adapter can reveal if the malware is in a VM. VMWare has MAC addresses starting with 00-05-69, 00-0c-29, 00-1c-14 or 00-50-56. VirtualBox also uses known Organisationally Unique Identifiers (OUIs) registered with the IEEE. These addresses can be queried from the registry at:

  - HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\
    Control\Class\4D36E972-E325-11CE-BFC1-08002BE10318\
    0000\NetworkAddress

2. Registry check: certain registry keys which are unique to virtual systems. These keys existed to allows users to customise their virtual machines. An example registry entry that indicates the virtualisation of the environment is:

- HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\
  Control \VirtualDeviceDrivers

3. Helper tools: VMWare environments contain the VMWare tools installed by default.

4. Spawned background processes: VMWare processes and tools like VMwareService.exe, VMwareTray.exe are commonly found in VMWare environments. These processes can be detected by using the CreateToolHelp32Snapshot API to get a snapshot of the current running processes, and then listing each process of the snapshot with the Process32First and Process32Next APIs.

5. Check for communication ports and behaviour: Communication check via IN (input from port) is a privileged instruction. VMWare does not directly raise the exception, but rather escalates the error to the host. The string "VMXh" is returned to the *ebx* register, and can be used to detect VMWare environments.

6. Execute special CPU instructions and compare the results with expected results. For instance, the malware can execute the CPUID instruction with $eax = 1$ as input, which will return the processor features. Return value in the *ecx* register has 0 as the 31st bit, the malware knows it is on a physical machine. If the 31st bit is 1, it indicates that the malware is running within a virtual environment.

7. Check the location of system structures such as descriptor tables. In a VM, the Interrupt Descriptor Table (IDT) is typically located at 0xffXXXXXX whereas, in a physical machine, this is located somewhat lower than that typically around 0x80ffffff. Other memory structures like the Global and Local Descriptor Tables can also indicate the presence of virtualisation technologies. Scoopy is a free suite of VM-detection tools running these checks on the memory structures.

8. Other hardware check: Various hardware parameters are specific to VMWare. Malware will query attributes such as SerialNo, SocketDesignation, and Caption to get values of Motherboard, Processor, and SCSI Controller respectively. VMWare will have different values for these parameters from the physical system.

VM designers are looking into countering these malware checks on the system. These techniques include searching for string instances of "VMXh" and changing these references to an alternate value. However, modifying the VM registry entries means that all programs are unable to detect the virtualised environment, including legitimate programs supporting drag-and-drop, shared folders and time synchronisation between the host and guest.

5 Findings

After conducting our literature survey and developing a better understanding of the topic at hand, we selected some areas of interest to analyse further. In these domains, we experimented heavily and gained a more practical grasp of the technicalities involved.

1. Packer Detection Techniques (Section 5.1)

   How do packer detectors like PEiD identify binaries as being packed, and by which packers?

   This section is done by Joyce.

2. Tools and Procedures for Unpacking Binaries (Section 5.2)

   In particular, we examine how to unpack UPX-packed binaries and develop a generalised understanding of the unpacking procedure.

   This section is done by Sarah.

3. PE Malformation (Section 5.3)

   How can malware exploit the PE file format to make analysing them harder?

   This section is done by Sarah.

4. Anti-Debugging (Section 5.4)

   How can malware thwart efforts by analysts to debug them?

   This section is done by Joyce.

5. Anti-Virtual Environment (Section 5.5)

   How can malware thwart efforts by analysts to run them in virtualised environments?

   This section is done by Joyce.

These areas will be further elaborated on below.

5.1 Packer Detection Techniques

Runtime packers are self-extracting archives - the executables unpack themselves in memory when they are executed. Packing is now very common among malware, because disassemblers are unable to parse the imports, strings and malicious payload in the programs. Numerous researchers have studied ways to identify if a file is packed, and also what the file is packed with. The former task is more straightforward, but the latter is very challenging.

5.1.1 Packer tool: Ultimate Packer for eXecutables (UPX)

We focus on the UPX packer in our project because it is commonly used, simple to use, and open-source. While most malware authors make tweaks to the packing algorithm, the underlying principles of the packing algorithms remain constant.

Generally, UPX works by destroying the import table, then changing the entry point to its unpacking code. In the unpacking code, the import table is reconstructed at runtime. As a result, UPX has a distinctive unpacking code that makes it easily detectable.

5.1.2 Packer Detection Tool: PE iDentifier (PEiD)

Analysts often use PEiD, which is able to detect most common packers, cryptors and compilers for PE files. PEiD is effective, but unfortunately not open source which could prove difficult for in-depth analysis.

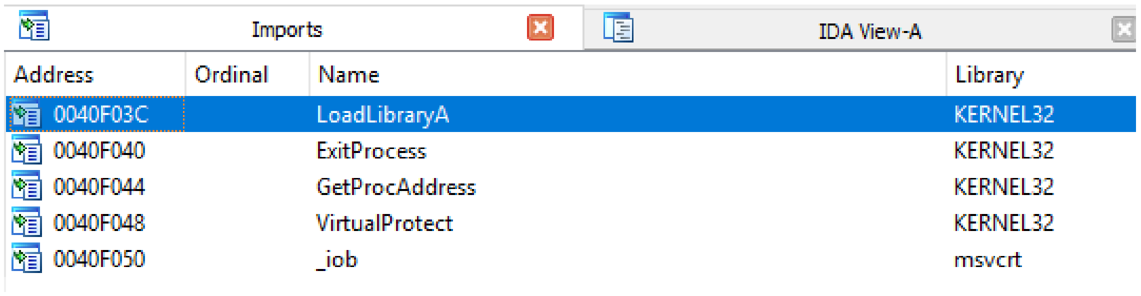
PEiD has an internal database of packer signatures that it uses to detect packers. PEiD does not publish its database on how it detects files as being UPX-packed. However, PEiD has the option to enable an external database where users can add their own entries. The external database only supports byte string matching with wildcards. It is not known if the internal database only relies on byte string matching, or other techniques.

5.1.3 Identifying if a File is Packed

Packed files often have telltale signs which make them recognisable. Their import tables are often destroyed, and they have high entropy.

5.1.3.1 Import Table

Often, the import table reveals a lot about the behaviour of the malware. Most packers like UPX destroy the import table and only restore it at runtime, as we will see later (Section 5.2). LoadLibraryA and GetProcAddress are widely used in packing algorithms (Figure 1), because they are used to load the DLLs needed and point to the correct functions at runtime. **Tools recommended: IDA Pro, PE Explorer**.



Address	Ordinal	Name	Library
0040F03C		LoadLibraryA	KERNEL32
0040F040		ExitProcess	KERNEL32
0040F044		GetProcAddress	KERNEL32
0040F048		VirtualProtect	KERNEL32
0040F050		_iob	msvcrt

Figure 1: Displaying the Imports of a packed file in IDA Pro

5.1.3.2 Entropy

Entropy is a measure of how random the bytes in a file are. Packed files tend to have a high entropy with few coherent strings. On the other hand, unpacked files may have more null bytes or a concentration of a few bytes. The tables below show the distribution of the most common and least common bytes in a packed and unpacked InfiniteProgram.exe file. **Tool recommended: Detect It Easy**.

Unpacked file		Packed file	
Byte	Percentage (%)	Byte	Percentage (%)
0x00	45.27	0x00	37.91
0x5f	3.95	0x5f	3.46
0x74	2.33	0x74	2.71
0x61	2.10	0x61	2.60
0x01	1.62	0x03	2.16
0x03	1.59	0x01	1.90
0x69	1.53	0x2e	1.86
0x2e	1.49	0x69	1.80

Table 1: Distribution of the top 8 bytes in an unpacked and packed file

Unpacked file		Packed file	
Byte	Percentage (%)	Byte	Percentage (%)
0x99	0	0xe7	0.005
0x9b	0	0x97	0.02
0x9f	0	0xea	0.02
0xad	0	0xa3	0.02
0xcb	0	0xa7	0.02
0xe7	0	0xf1	0.02
0x8f	0.003	0xf5	0.02
0xa5	0.003	0x92	0.03

Table 2: Distribution of the bottom 8 bytes in an unpacked and packed file

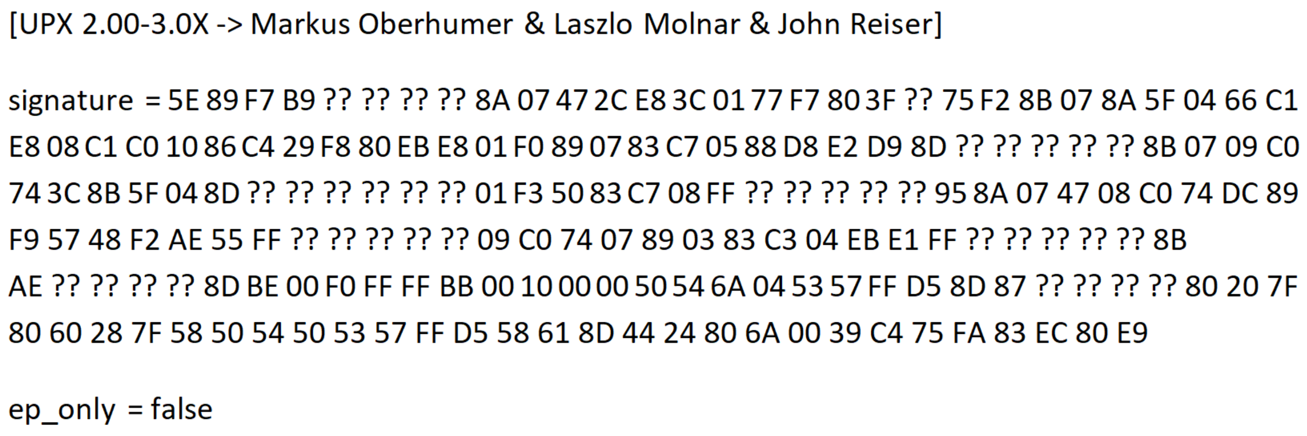
5.1.4 Identifying Specific Packers

While it is simple to detect if a file is packed, the task to identify what a file is packed with is very challenging. There are many packers in the market, and some packing algorithms are also proprietary.

Currently, there are two key methods to identify the specific packer, to look for patterns in the unpacking code or attempting to unpack the executable.

5.1.4.1 Patterns in the unpacking code

Files packed with the same algorithm tend to correspondingly have the same unpacking process. PEiD is not open source, but there are numerous attempts to piece together a comprehensive external database. One such entry is shown below.



```
[UPX 2.00-3.0X -> Markus Oberhumer & Laszlo Molnar & John Reiser]

signature = 5E 89 F7 B9 ?? ?? ?? ?? 8A 07 47 2C E8 3C 01 77 F7 80 3F ?? 75 F2 8B 07 8A 5F 04 66 C1
E8 08 C1 C0 10 86 C4 29 F8 80 EB E8 01 F0 89 07 83 C7 05 88 D8 E2 D9 8D ?? ?? ?? ?? ?? 8B 07 09 C0
74 3C 8B 5F 04 8D ?? ?? ?? ?? ?? ?? 01 F3 50 83 C7 08 FF ?? ?? ?? ?? ?? 95 8A 07 47 08 C0 74 DC 89
F9 57 48 F2 AE 55 FF ?? ?? ?? ?? ?? 09 C0 74 07 89 03 83 C3 04 EB E1 FF ?? ?? ?? ?? ?? 8B
AE ?? ?? ?? ?? 8D BE 00 F0 FF FF BB 00 10 00 00 50 54 6A 04 53 57 FF D5 8D 87 ?? ?? ?? ?? 80 20 7F
80 60 28 7F 58 50 54 50 53 57 FF D5 58 61 8D 44 24 80 6A 00 39 C4 75 FA 83 EC 80 E9

ep_only = false
```

Figure 2: External database entry on PEiD

The "?" symbols represent wildcard byte searches that PEiD supports. The string can be found in UPX-packed files, such as the packed InfiniteProgram.exe. A linear disassembly of the string in the packed InfiniteProgram is shown below.

The wildcard bytes tend to be constants. Often, malware authors make small modifications to the packing algorithms to evade packer detection. With wildcard bytes, changes to constants in the packing algorithm will not affect PEiD's ability to label files as being UPX-packed.

```
5e                       pop             esi {var_24}
89f7                     mov             edi, esi
b947000000               mov             ecx, 0x47

8a07                     mov             al, byte [edi]
47                       inc             edi
2ce8                     sub             al, 0xe8

3c01                     cmp             al, 0x1
77f7                     ja              0x40e46a

803f00                   cmp             byte [edi], 0x0
75f2                     jne             0x40e46a

8b07                     mov             eax, dword [edi]
8a5f04                   mov             bl, byte [edi+0x4]
66c1e808                 shr             ax, 0x8
c1c010                   rol             eax, 0x10
86c4                     xchg            ah, al
29f8                     sub             eax, edi
80ebe8                   sub             bl, 0xe8
01f0                     add             eax, esi
8907                     mov             dword [edi], eax
83c705                   add             edi, 0x5
88d8                     mov             al, bl
e2d9                     loop            0x40e46f

8dbe00c00000             lea             edi, [esi+0xc000]

8b07                     mov             eax, dword [edi]
09c0                     or              eax, eax
743c                     je              0x40e4de

8b5f04                   mov             ebx, dword [edi+0x4]
8d843000e00000           lea             eax, [eax+esi+0xe000]
01f3                     add             ebx, esi
50                       push            eax {var_24}
83c708                   add             edi, 0x8
ff963ce00000             call            dword [esi+0xe03c]
```

Listing 1: Linear Diassembly of byte string associated with UPX in Binary Ninja

Given such a byte string disassembly, it is possible to tweak the assembly instructions to perform equivalent operations that have different expressions. There are two types of changes that can be made:

- Re-ordering instructions which have no data dependencies
- Replacing instructions with different opcodes but have the same effect

A list of possible changes to the listing above is shown in Table 3.

No.	Possible change
1	Re-order lines 2 and 3
2	Re-order lines 6 and 7
3	Re-order lines 15 and 16
4*	Re-order lines 20 and 21
5*	Re-order lines 21 and 22
6	Re-order lines 24 and 25
7	Replace line 31 with $85c0 \quad test\ eax,\ eax$
8	Re-order lines 36, 37 and 38

Table 3: Possible changes to byte string
$*$ changes cannot be applied together

Unfortunately, after applying these changes individually and together, PEiD was still able to identify the executable as being UPX-packed. As mentioned earlier, PEiD is not open-source so it is unclear what byte strings are used to detect UPX-packed files. It is also possible that the PEiD internal database stores permutations of the changes being made, highlighting the arms race between packing evasion and detection. For every detection technique, a new evasion technique will develop, which then prompts a more comprehensive detection technique.

Nevertheless, detecting packers using byte strings still relies on the principle of keeping a finite database of entries. New packing algorithms would be able to evade packing detectors easily, but these novel algorithms are challenging to develop and do not appear frequently.

5.1.4.2 Attempting to unpack the file

Packers usually come with their own unpacking algorithms. In UPX, the `-d` flag can be passed in to denote decompression. As such, one way that packing detectors could possibly detect packed files is to attempt unpacking the files. It is not known if PEiD employs such a technique.

Attempting to unpack the files is not an effective method, since malware authors often make small tweaks to the packing algorithm to prevent analysts from unpacking the files. For instance, the UPX packing algorithm results in two sections named UPX0 and UPX1.

Section names do not affect the execution of the file, so renaming the sections to TPX0 and TPX1 still result in valid executable files. However, this act of renaming the sections corrupts the UPX decompression process. The UPX program will be unable to unpack the file, throwing a NotCompressibleException.

5.2 Tools and Procedures for Unpacking Binaries

Some malware authors pack their malware in order to obfuscate data, hinder static analysis and reduce the amount of space the malware takes up on disk (for non-fileless malware). To gain access to the inner workings of the binary, the malware analyst must first identify the packer used, and then apply the appropriate unpacking procedures to manually unpack the executable.

Packing a file destroys its import address table (IAT), which is an array of function pointers to statically-imported DLL function addresses, which will be populated at runtime. This provides key information on what kind of functions the binary uses, so as to allow the analyst a more accurate guess as to what the suspicious binary does. Destruction of the IAT prevents analysts from viewing the complete imports list of the binary which can be crucial to analysis activities.

Besides compressing the executable, some packers also integrate anti-debugging, anti-reversing tricks, self-modifying code techniques and encryption. These packers are also referred to as protectors (eg. Armadillo, Themida) or crypters (Yodas Crypter). Such techniques undermine the effectiveness of the procedures and tools described. Hence, the tools and procedures described in this section are meant for non-VM based packers such as UPX, which is one of the more popular packers used to pack malware.

5.2.1 Unpacking 32-bit packed binaries

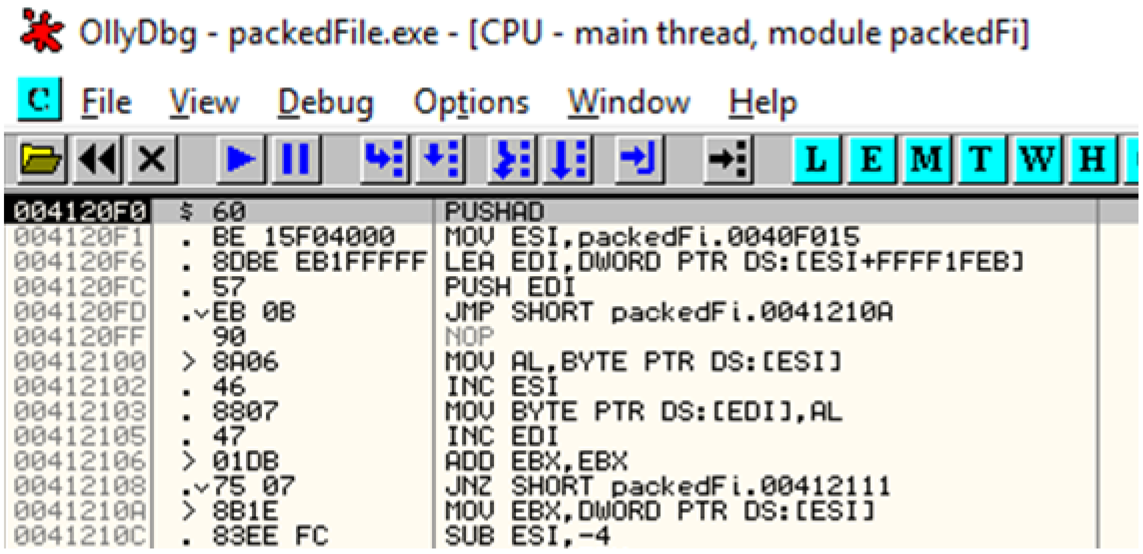
I started off reading guides on how to manually unpack a 32-bit UPX-packed file, and afterward tried to unpack it myself. I repeated this learning process with different packers, and while doing so realized that the guides for approximately 2 or 3 popular packers had more or less the same steps. I identified the general technique and tried to apply it to more packers, including the less popular ones. So far, I can verify that this technique works for UPX, AsPack, Mew11, BEP, and NSPack packers.

The standard tools required for this procedure are:

1. Debugger: I tried the procedure with both OllyDBG and Immunity Debugger for unpacking 32-bit files, and X64DBG for unpacking 64-bit files (covered later on).
2. Dumping tool: I used the OllyDumpEx plugin, which works for both debuggers mentioned previously.
3. Import reconstructor tool: I tested this procedure using both Scylla and ImpRec.

Because different packers use different algorithms for unpacking, there is no common unpacking technique. Hence, this procedure lets the packers' respective unpacking stubs run and unpack the file, stopping the program execution just before the unpacking stub finishes and transfers control over to the main program. Afterward, the program needs to be dumped out at that point in memory, and the dumped executable needs to be fixed by rebuilding the IAT. This provides an easy and efficient method to unpack files, without needing to understand the exact packing algorithm.

1. Open the file with the debugger, which will cause the debugger will stop at the entry point of the file, which is at the unpacking stub.

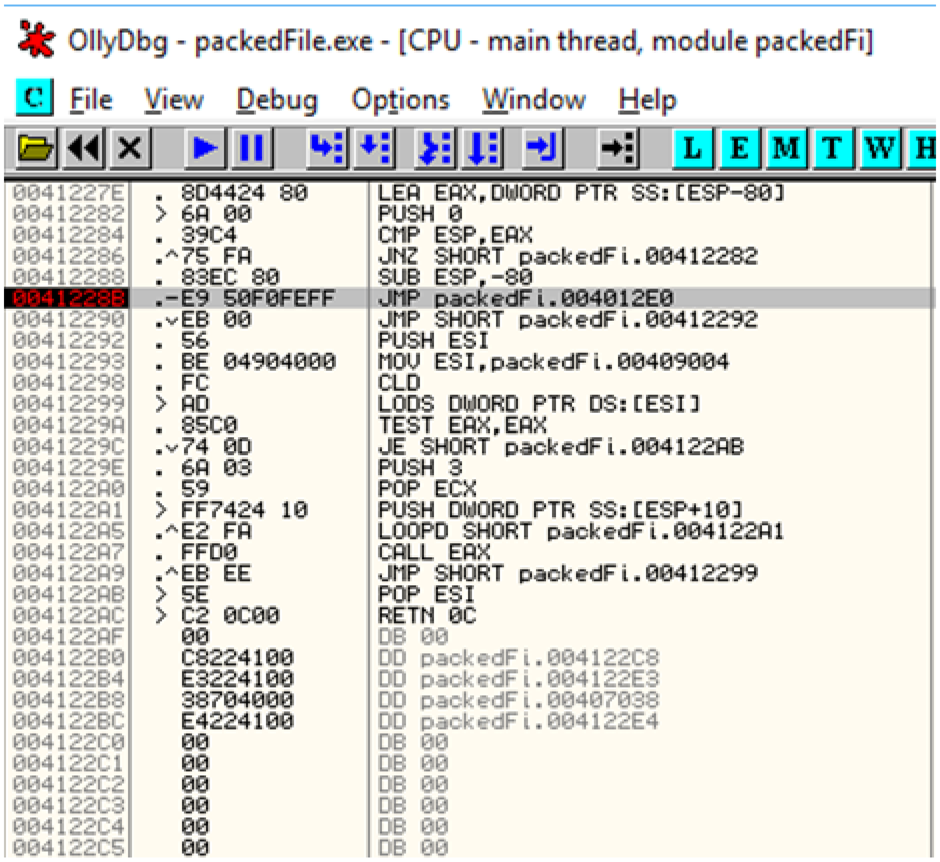

OllyDbg - packedFile.exe - [CPU - main thread, module packedFi]
File View Debug Options Window Help
L E M T W H

```
004120F0  $ 60            PUSHAD
004120F1  . BE 15F04000   MOV ESI,packedFi.0040F015
004120F6  . 8DBE EB1FFFFF LEA EDI,DWORD PTR DS:[ESI+FFFF1FEB]
004120FC  . 57            PUSH EDI
004120FD  .vEB 0B         JMP SHORT packedFi.0041210A
004120FF    90            NOP
00412100  > 8A06          MOV AL,BYTE PTR DS:[ESI]
00412102  . 46            INC ESI
00412103  . 8807          MOV BYTE PTR DS:[EDI],AL
00412105  . 47            INC EDI
00412106  > 01DB          ADD EBX,EBX
00412108  .v75 07         JNZ SHORT packedFi.00412111
0041210A  > 8B1E          MOV EBX,DWORD PTR DS:[ESI]
0041210C  . 83EE FC       SUB ESI,-4
```

Figure 3: Opening up the packed file in OllyDBG brings us to the start of the unpacking stub

Normally, the instruction is a PUSHAD instruction, which saves the state of the general-purpose registers onto the stack before the unpacking stub executes; this is the case for most of the unpackers I experimented with. However, some packers I have experimented with do not use PUSHAD to mark the start of the unpacking process. For instance, Mew11's file entry point is a JMP instruction which leads to the beginning of the unpacking stub. Mew11's unpacking stub itself also does not start with the PUSHAD instruction.

2. Scroll through the unpacking stub to find the OEP (original entry point) jump, where the unpacking stub transfers control to the original code.


OllyDbg - packedFile.exe - [CPU - main thread, module packedFi]
C File View Debug Options Window Help
L E M T W H

```
0041227E  . 8D4424 80     LEA EAX,DWORD PTR SS:[ESP-80]
00412282  > 6A 00         PUSH 0
00412284  . 39C4          CMP ESP,EAX
00412286  .^75 FA         JNZ SHORT packedFi.00412282
00412288  . 83EC 80       SUB ESP,-80
0041228B  .-E9 50F0FEFF   JMP packedFi.004012E0
00412290  .vEB 00         JMP SHORT packedFi.00412292
00412292  . 56            PUSH ESI
00412293  . BE 04904000   MOV ESI,packedFi.00409004
00412298  . FC            CLD
00412299  > AD            LODS DWORD PTR DS:[ESI]
0041229A  . 85C0          TEST EAX,EAX
0041229C  .v74 0D         JE SHORT packedFi.004122AB
0041229E  . 6A 03         PUSH 3
004122A0  . 59            POP ECX
004122A1  > FF7424 10     PUSH DWORD PTR SS:[ESP+10]
004122A5  .^E2 FA         LOOPD SHORT packedFi.004122A1
004122A7  . FFD0          CALL EAX
004122A9  .^EB EE         JMP SHORT packedFi.00412299
004122AB  > 5E            POP ESI
004122AC  > C2 0C00       RETN 0C
004122AF    00            DB 00
004122B0    C8224100      DD packedFi.004122C8
004122B4    E3224100      DD packedFi.004122E3
004122B8    38704000      DD packedFi.00407038
004122BC    E4224100      DD packedFi.004122E4
004122C0    00            DB 00
004122C1    00            DB 00
004122C2    00            DB 00
004122C3    00            DB 00
004122C4    00            DB 00
004122C5    00            DB 00
```

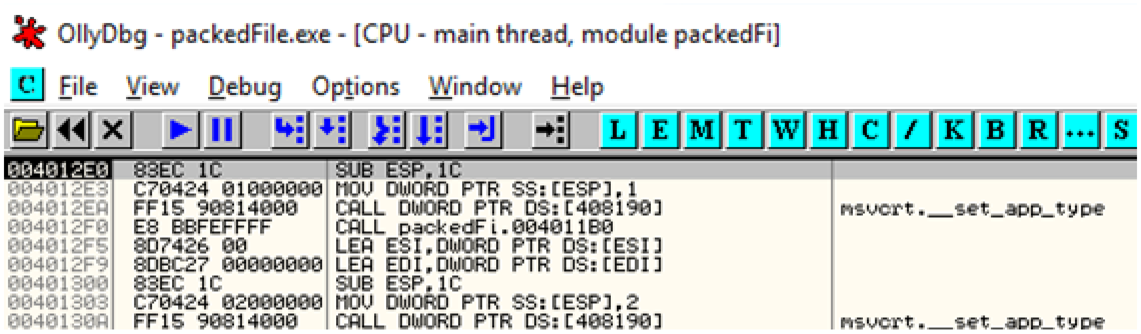
Figure 4: The OEP jump, located toward the end of the unpacking stub. A breakpoint has been set on the jump, as seen from the memory address of the jump highlighted in black.

The transfer of control needs not be strictly a JMP instruction, but rather something that transfers control to the OEP. For instance, instead of using a JMP instruction, AsPack uses a PUSH instruction to push the OEP address onto the stack, followed immediately by a RETN instruction to execute the jump to the OEP location.

There are certain telling characteristics that a JMP/PUSH + RETN instruction is a transfer of control back to the original code, such as being located just before a bunch of null/garbage bytes, or jumping to a faraway location (as compared to JMP SHORT instructions).

After identifying the OEP jump, set a breakpoint there and resume the program so that the debugger will stop the program at the jump.

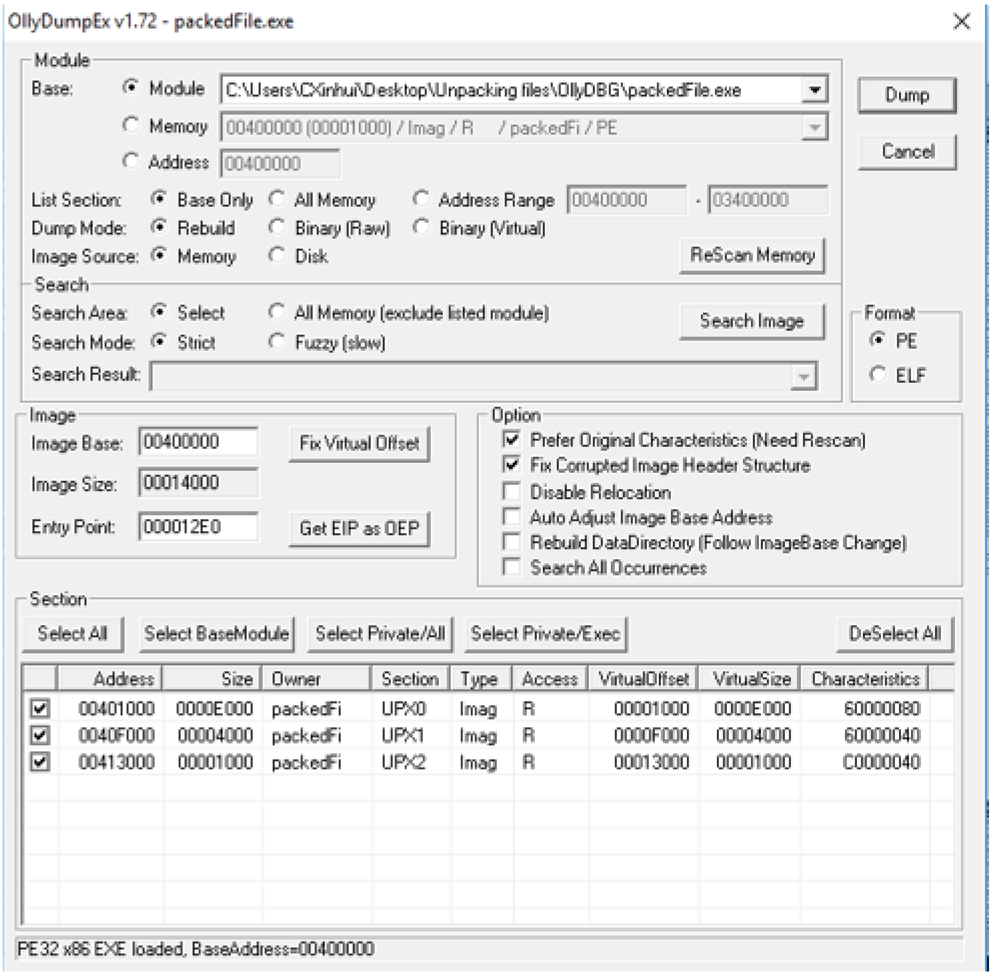
3. Once at the OEP jump instruction, single step through to land the program at the original entry point.


OllyDbg - packedFile.exe - [CPU - main thread, module packedFi]
File View Debug Options Window Help
L E M T W H C / K B R ... S

```
004012E0  83EC 1C           SUB ESP,1C
004012E3  C70424 01000000   MOV DWORD PTR SS:[ESP],1
004012EA  FF15 90814000     CALL DWORD PTR DS:[408190]        msvcrt.__set_app_type
004012F0  E8 BBFEFFFF       CALL packedFi.004011B0
004012F5  8D7426 00         LEA ESI,DWORD PTR DS:[ESI]
004012F9  8DBC27 00000000   LEA EDI,DWORD PTR DS:[EDI]
00401300  83EC 1C           SUB ESP,1C
00401303  C70424 02000000   MOV DWORD PTR SS:[ESP],2
0040130A  FF15 90814000     CALL DWORD PTR DS:[408190]        msvcrt.__set_app_type
```

Figure 5: The original entry point, reached after single-stepping through the tail jump as shown in Figure 4.

4. Dump the program using a dumping tool. At this stage it would be pertinent to note that there are 2 different types of dumping tools, one which takes a snapshot of the executable at that stage in memory (and saves the dump as an executable file) and one which dumps out the entire process memory (saving the dump as a .dmp file.) I am using the former.



Module
Base: [✓ Module] C:\Users\CXinhui\Desktop\Unpacking files\OllyDBG\packedFile.exe
Memory 00400000 (00001000) / Imag / R / packedFi / PE
Address 00400000
List Section: [✓ Base Only] All Memory Address Range 00400000 - 03400000
Dump Mode: [✓ Rebuild] Binary (Raw) Binary (Virtual)
Image Source: [✓ Memory] Disk

Search
Search Area: [✓ Select] All Memory (exclude listed module)
Search Mode: [✓ Strict] Fuzzy (slow)
Search Result:

Format
[✓ PE] ELF

Image
Image Base: 00400000
Image Size: 00014000
Entry Point: 000012E0

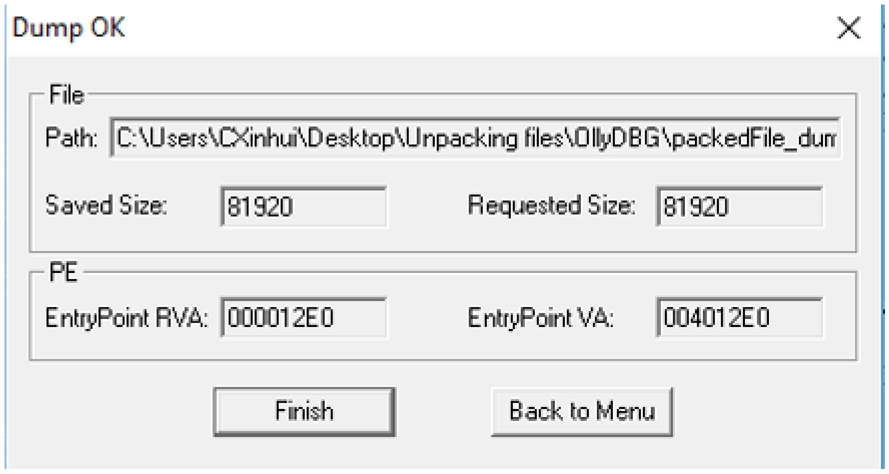
Option
☑ Prefer Original Characteristics (Need Rescan)
☑ Fix Corrupted Image Header Structure
☐ Disable Relocation
☐ Auto Adjust Image Base Address
☐ Rebuild DataDirectory (Follow ImageBase Change)
☐ Search All Occurences

Section

	Address	Size	Owner	Section	Type	Access	VirtualOffset	VirtualSize	Characteristics
☑	00401000	0000E000	packedFi	UPX0	Imag	R	00001000	0000E000	60000080
☑	0040F000	00004000	packedFi	UPX1	Imag	R	0000F000	00004000	60000040
☑	00413000	00001000	packedFi	UPX2	Imag	R	00013000	00001000	C0000040

PE32 x86 EXE loaded, BaseAddress=00400000

Figure 6: When using OllyDumpEx to dump the executable, click "Get EIP as OEP" button before dumping.


Dump OK

File

Path: C:\Users\CXinhui\Desktop\Unpacking files\OllyDBG\packedFile_dum

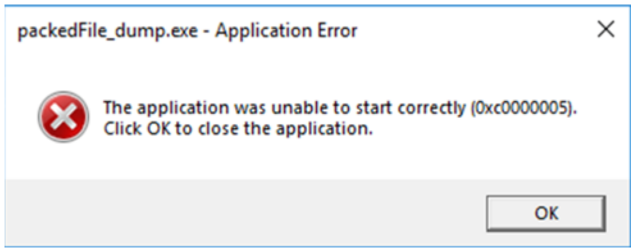
Saved Size: 81920 Requested Size: 81920

PE

EntryPoint RVA: 000012E0 EntryPoint VA: 004012E0

Figure 7: OllyDumpEx will notify the user when the dumping is successful, and generates the dump as a separate executable file.

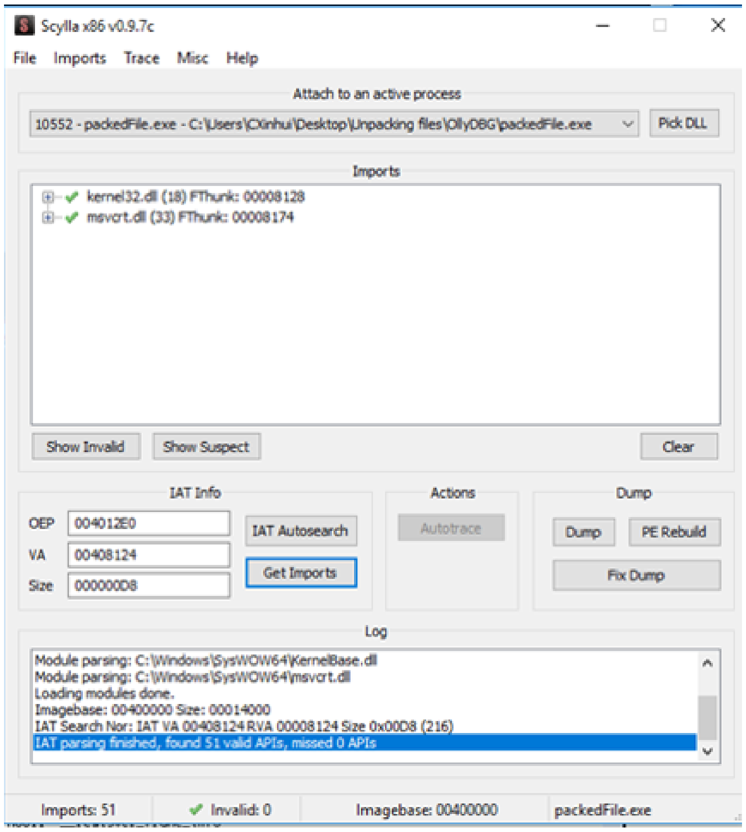
At this point, the Windows loader will not be able to run the dumped executable because the IAT is still destroyed.


packedFile_dump.exe - Application Error

The application was unable to start correctly (0xc0000005). Click OK to close the application.

Figure 8: Error message displayed when I try to execute the dumped file.

5. Open the import reconstructing tool, Scylla. After manually filling in the OEP of the program, use Scylla to retrieve the imports of the executable.



```
10552 - packedFile.exe - C:\Users\CXinhui\Desktop\Unpacking files\OllyDBG\packedFile.exe

kernel32.dll (18) FThunk: 00008128
msvcrt.dll (33) FThunk: 00008174

OEP  004012E0
VA   00408124
Size 000000D8

Module parsing: C:\Windows\SysWOW64\KernelBase.dll
Module parsing: C:\Windows\SysWOW64\msvcrt.dll
Loading modules done.
Imagebase: 00400000 Size: 00014000
IAT Search Nor: IAT VA 00408124 RVA 00008124 Size 0x00D8 (216)
IAT parsing finished, found 51 valid APIs, missed 0 APIs

Imports: 51    Invalid: 0    Imagebase: 00400000    packedFile.exe
```

Figure 9: After loading the active process (the packed file) into Scylla, manually fill in the OEP location which Scylla will use to retrieve import information.

Scylla applies a heuristic-based approach to find possible pointers to the imports of the packed executable. For instance, Scylla only chooses jump and unconditional branch instructions as possible IAT pointers, and checks if the instructions have a flag indicating that they use 64-bit RIP-relative memory indirection addressing. After identifying possible IAT pointers, Scylla checks the pointers against a list of module handles, retrieved using the Windows API function enumProcessModules, and generates a list of valid and invalid thunks.

After deleting invalid thunks, Scylla can be used to restore the IAT of the dumped executable such that the executable will be able to run. This resulting binary is the unpacked executable.

5.2.1.1 Determining when an executable has been unpacked successfully without an original file as basis

During my experiments, I had an original file as a basis to compare the unpacked file to, so I could determine if I had unpacked the file correctly. This could be done by comparing the entry point of the unpacked file with the original file; if they matched, the file was unpacked correctly. However, in the context of unpacking malware samples, it will not be realistic to expect a basis file to compare the unpacked malware to. However, some criteria can be used to check if the malware has been unpacked correctly, although there is no concrete acid test:

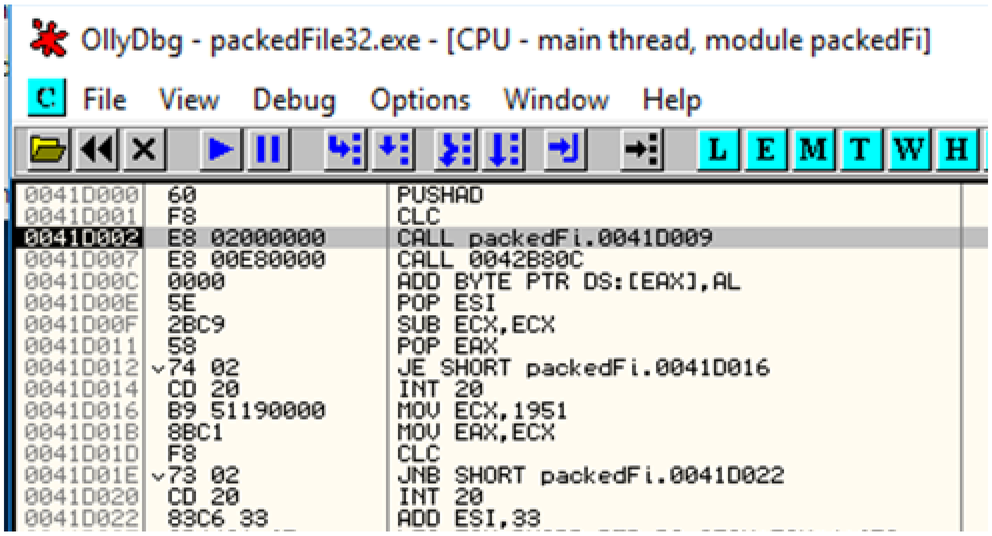
1. Ability to view strings and imports in parsers like IDA Pro and Binary Ninja after unpacking the file
2. File is executable after the IAT is fixed by tools like ImpRec and Scylla, and execution should also be similar to the packed file. For instance, if the packed malware changed certain registry keys, the unpacked malware, when executed, should also be able to change the same registry keys.
3. Ability to decompile the unpacked, disassembled instructions back to C/C++

5.2.1.2 Remarks on other tools and packers I have tested

I have tested other import reconstruction tools such as CHimpRec and Imports Fixer. However, because both these tools are outdated and no longer in development, they are not compatible with my Windows 10 OS.

I have also tested these procedures on packers other than the ones mentioned above. However, some packers were outdated, such as PESpin, NeoLite, 32Lite, and AndPakk2; these are only compatible with the older versions of Windows, namely Windows 7, XP, and Vista. Similarly, packers like kkrunchy and MPress are also very old and hence do not support executables with TLS callbacks. There existed some packers which were both OS compatible and supported TLS callbacks which I have tested, but I was unable to unpack them manually. For instance, I was unable to locate the OEP jump for FSG and Exe32Pack. I also tried WinUPack, but the WinUPack unpacking method does not follow the technique of running the unpacking stub as described in Section 5.2?7.1 but rather, requires navigation through the library modules of the program (such as kernel32 and the ntdll modules) to arrive at the OEP.

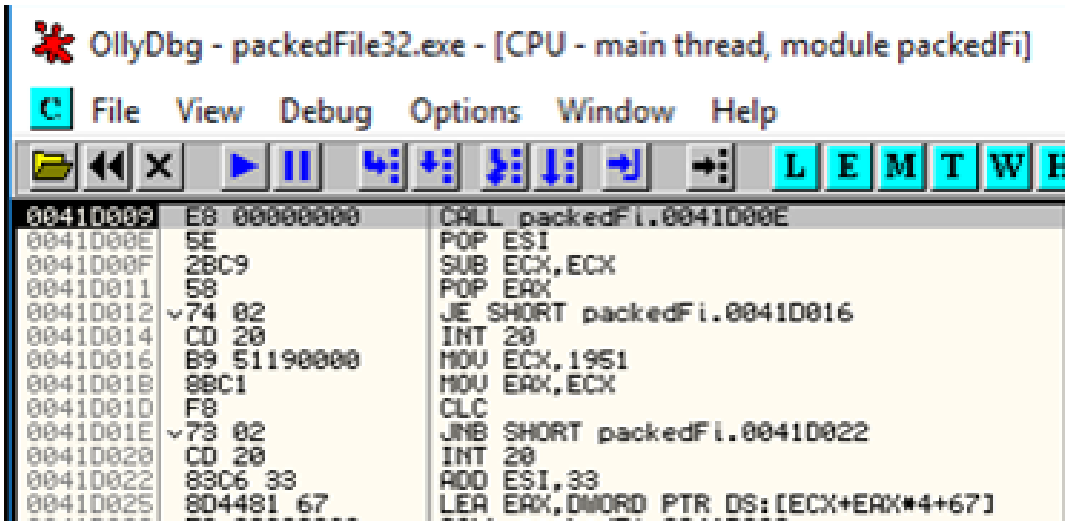
TeLock was one such packer that I tried to unpack, but was unsuccessful due to the presence of anti-analysis tricks. When I opened the code up in OllyDBG, there were incorrect instructions as well as software interrupts, which are usually not seen in user code, which further hints that the code may be wrong.


OllyDbg - packedFile32.exe - [CPU - main thread, module packedFi]
C File View Debug Options Window Help
L E M T W H

```
0041D000   60              PUSHAD
0041D001   F8              CLC
0041D002   E8 02000000     CALL packedFi.0041D009
0041D007   E8 00E80000     CALL 0042B80C
0041D00C   0000            ADD BYTE PTR DS:[EAX],AL
0041D00E   5E              POP ESI
0041D00F   2BC9            SUB ECX,ECX
0041D011   58              POP EAX
0041D012  v74 02           JE SHORT packedFi.0041D016
0041D014   CD 20           INT 20
0041D016   B9 51190000     MOV ECX,1951
0041D01B   8BC1            MOV EAX,ECX
0041D01D   F8              CLC
0041D01E  v73 02           JNB SHORT packedFi.0041D022
0041D020   CD 20           INT 20
0041D022   83C6 33         ADD ESI,33
```

Figure 10: Wrong instructions such as the one at memory address 0041D007 and interrupts, such as the instruction at memory address 0041D020, suggest that the disassembly is incorrect.

The correctly-formatted CALL instruction at memory address 0041D002 as seen in Figure 10 is jumping into the middle of the wrong instruction at location 0041D007. This causes the wrong instruction to resolve itself to become a correct instruction, when I single-step through the program, as seen in the figure below.


OllyDbg - packedFile32.exe - [CPU - main thread, module packedFi]
C File View Debug Options Window Help
L E M T W

```
0041D009   E8 00000000     CALL packedFi.0041D00E
0041D00E   5E              POP ESI
0041D00F   2BC9            SUB ECX,ECX
0041D011   58              POP EAX
0041D012  v74 02           JE SHORT packedFi.0041D016
0041D014   CD 20           INT 20
0041D016   B9 51190000     MOV ECX,1951
0041D01B   8BC1            MOV EAX,ECX
0041D01D   F8              CLC
0041D01E  v73 02           JNB SHORT packedFi.0041D022
0041D020   CD 20           INT 20
0041D022   83C6 33         ADD ESI,33
0041D025   8D4481 67       LEA EAX,DWORD PTR DS:[ECX+EAX*4+67]
```

Figure 11: After single-stepping, OllyDBG then evaluates the wrongly-formatted instruction to be correct.

Furthermore, the INT instructions are never actually executed because of the presence of short jumps just before the instructions, causing the program execution to skip the INTs so that software interrupts are never generated. Hence, OllyDBG has actually disassembled the code correctly. This is because OllyDBG is a recursive descent parser, which does not assume that all code is aligned sequentially, but instead disassembles code recursively by following branches. This prevents it from being confused by tweaked instructions such as the ones found in TeLock's unpacking stub. In contrast, linear disassemblers such as WinDBG will not be able to disassemble the instructions correctly.

After understanding how TeLock's code can still work even with the presence of interrupts and incorrect instructions, it will be worth spending more time exploring how to unpack such packers which employ such anti-analysis tricks.

5.2.2 Unpacking 64-bit packed binaries

The technique described in section 7.1 can also be applied to unpack 64-bit binaries. I have tried applying the procedure to unpack a 64-bit UPX-packed executable. However, different tools have to be used as OllyDBG, Immunity Debugger and OllyDumpEx all do not have x64 support. I used x64DBG, which comes with a built-in Scylla plugin. The Scylla plugin can be used in place of OllyDumpEx to dump the file, and also rebuild the IAT of the dumped executable.

5.3 PE Malformation

PE malformations are modifications made to the file format data or layout of the PE file in an attempt to achieve undesired behaviour by the programs parsing the file. The result of this modification is a binary which does not adhere to the boundaries stated in the Windows PE file documentation, but are still able to be read by the Windows loader and can thus be executed. Some PE malformations are for the purpose of only executing in some Windows environments but not others. PE malformations confuse and crash poorly-implemented parsers, thus hindering static analysis.

Though there are numerous complex types of PE file manipulations, the project focuses on replicating two specific forms:

- 1. Shifting the PE header to the overlay such that the PE header will not be loaded into memory

  This technique aims to confuse parsers which parse the file from memory instead from disk. Since the PE header is not loaded into memory, they will be unable to display the PE header.

- 2. Shuffling sections within the PE file to determine if swapped sections have any impact on analysis

In order to directly manipulate attributes of the PE file, I used a Python module called 'pefile'.

5.3.1 PE Header in Overlay

The PE header is part of a structure called the Image_NT_Headers, which also contains the optional header and the 16 data directories of the PE file. This implies that it is not possible to shift just the PE header itself into the overlay without shifting the other members of the Image_NT_Headers structure. In addition, the Microsoft documentation states that the "location of the section table is determined by calculating the location of the first byte after the headers". As a result, the section table also needs to be shifted into the overlay, along with the Image_NT_Headers.

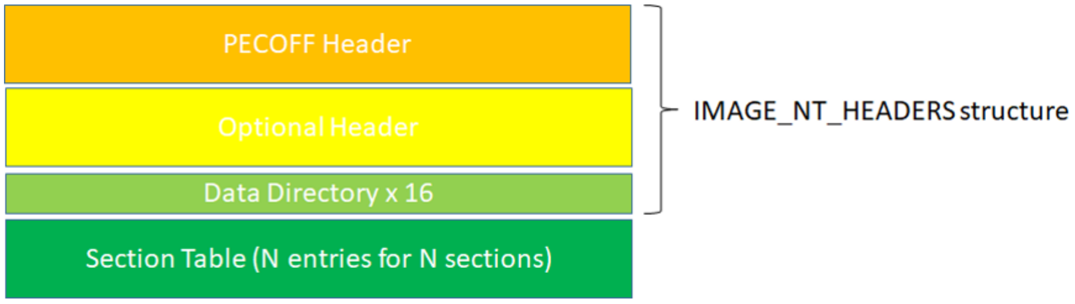

PECOFF Header
Optional Header
Data Directory x 16
Section Table (N entries for N sections)
IMAGE_NT_HEADERS structure

Figure 12: Graphical representation of IMAGE_NT_HEADERS. The bytes of these structures were copied and pasted into the overlay.

Afterward, the pointer to the PE header, which is the e_lfanew field in the DOS header, needs to be updated to point to the new location of the PE header in the overlay.

However, so far my attempts to get a working executable have not been successful. Trying to run the file returns an error message that the application is unable to run on the PC.

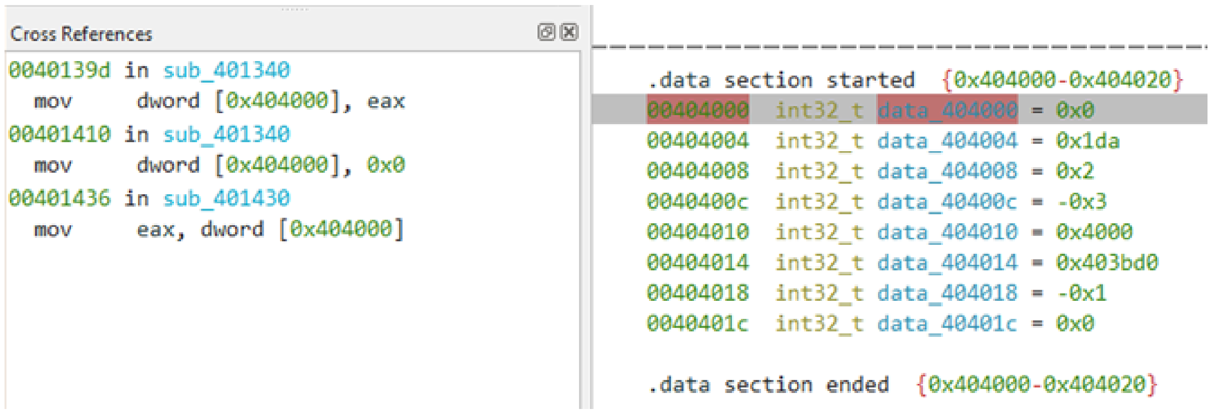
5.3.1.1 Troubleshooting

To double check whether or not Binary Ninja was indeed reading the PE header from the overlay, I replaced the original PE header with null bytes and decremented the NumberOfSections attribute in the overlay PE header by 1, and then reloaded the binary into the parser. If Binary Ninja loaded one less section, then it would definitely be reading from the PE header in the overlay. If Binary Ninja did not load one less section, then perhaps it had saved the original PE header and was still reading from that. As it turned out, Binary Ninja did indeed load one less section, which meant that it was reading from the overlay's PE header, so locating the PE header within the overlay was not the problem.

5.3.2 Section Shuffling

Online literature cites section shuffling as a common form of PE malformation, whereby the order of sections are swapped, making sections found at unexpected locations and claiming that this might crash poorly implemented parsers. Hence, I want to verify whether shuffling the order of sections has any impact on the tools commonly used for binary analysis, such as IDA Pro and Binary Ninja.

In order to swap sections and still result in an executable file, the virtual addresses of the sections must be swapped, the section table updated, and all pointers within the sections involved (if any) must be resolved. Binary Ninja provides a handy function for determining variables: the cross-reference window. When clicking on a data variable, the cross reference window will show the exact memory addresses where the variable is being referenced.

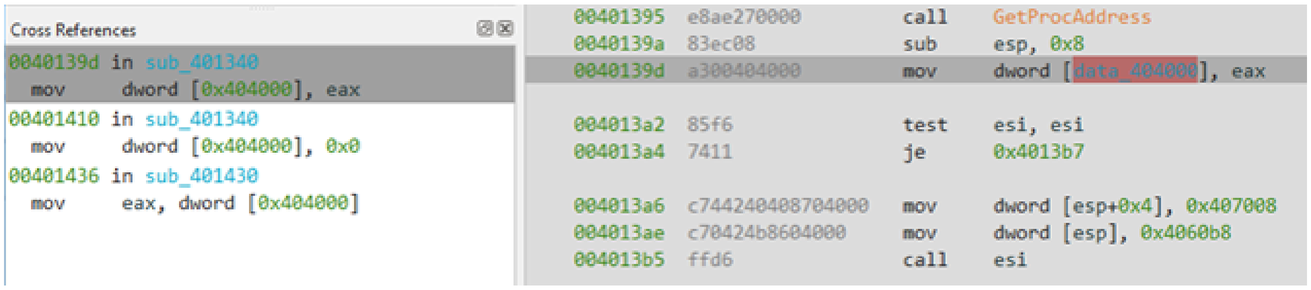


```
0040139d in sub_401340
  mov     dword [0x404000], eax
00401410 in sub_401340
  mov     dword [0x404000], 0x0
00401436 in sub_401430
  mov     eax, dword [0x404000]
```

```
 .data section started  {0x404000-0x404020}
 00404000  int32_t data_404000 = 0x0
 00404004  int32_t data_404004 = 0x1da
 00404008  int32_t data_404008 = 0x2
 0040400c  int32_t data_40400c = -0x3
 00404010  int32_t data_404010 = 0x4000
 00404014  int32_t data_404014 = 0x403bd0
 00404018  int32_t data_404018 = -0x1
 0040401c  int32_t data_40401c = 0x0

 .data section ended  {0x404000-0x404020}
```

Figure 13: By clicking on the variable data_404000, the memory addresses in which data_404000 is referenced appear in the cross references window.

I used this function to identify where references needed to be changed. For instance, selecting the first function in Fig. 13 will take me to the exact instruction which references variable data_404000 as shown below.

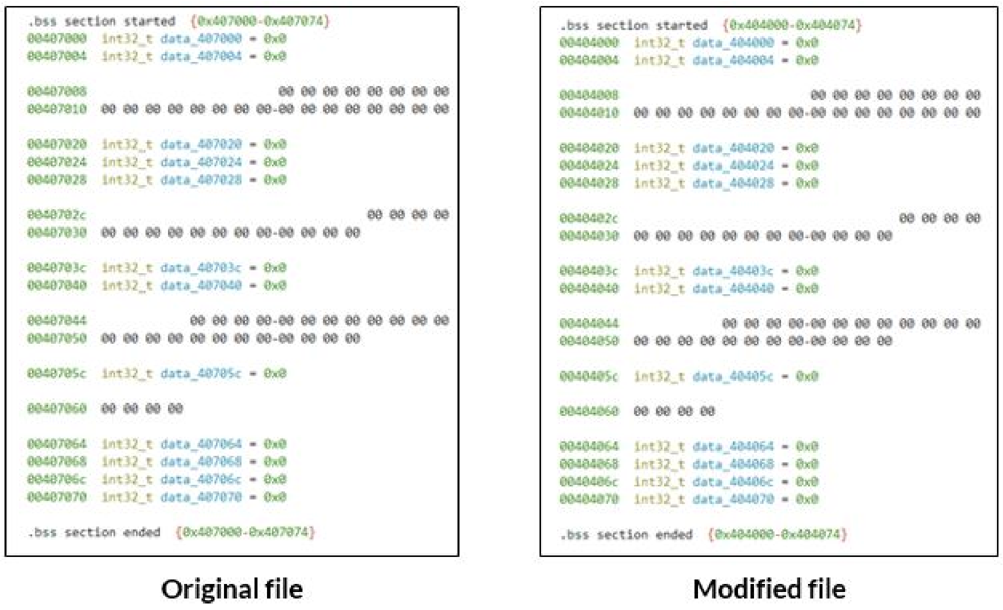


```
0040139d in sub_401340
  mov      dword [0x404000], eax
00401410 in sub_401340
  mov      dword [0x404000], 0x0
00401436 in sub_401430
  mov      eax, dword [0x404000]
```

```
00401395  e8ae270000          call    GetProcAddress
0040139a  83ec08              sub     esp, 0x8
0040139d  a300404000          mov     dword [data_404000], eax

004013a2  85f6                test    esi, esi
004013a4  7411                je      0x4013b7

004013a6  c744240408704000    mov     dword [esp+0x4], 0x407008
004013ae  c70424b8604000      mov     dword [esp], 0x4060b8
004013b5  ffd6                call    esi
```

Figure 14: Binary Ninja not only displays the disassembled instruction, but the exact bytes corresponding to the instruction. The bytes '404000' appears in little endian in the byte sequence, referencing the variable data_404000. By changing the bytes, another variable in another location can be referenced.

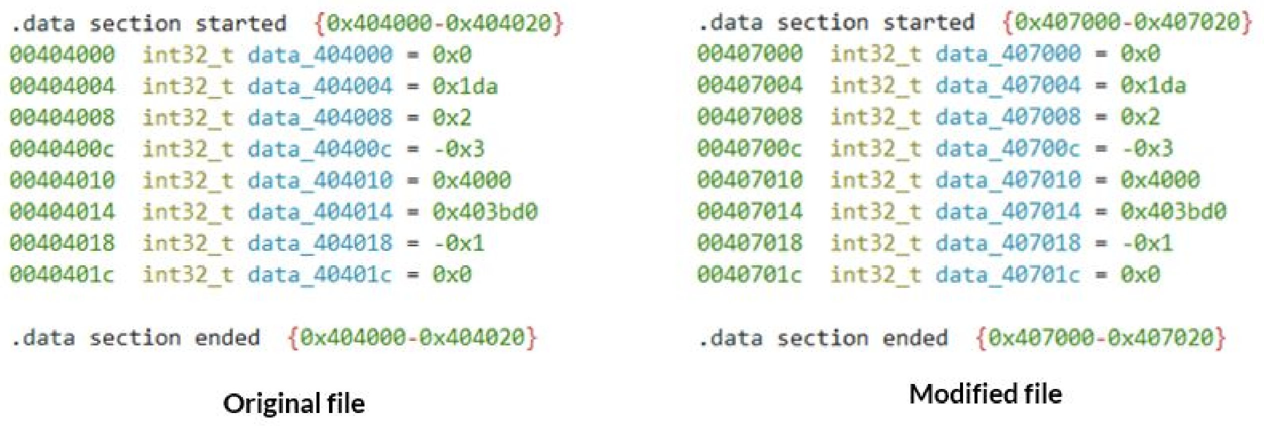
I manually changed the references for all variables in Binary Ninja using the hex editor function. For instance, the data_404000 variable in the original .data section was changed to data_407000, which is the first variable in the .bss section, and vice versa for data07000. After which, the swapped variables look like:



```
.bss section started  {0x407000-0x407074}
00407000  int32_t data_407000 = 0x0
00407004  int32_t data_407004 = 0x0

00407008                           00 00 00 00 00 00 00 00
00407010  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00

00407020  int32_t data_407020 = 0x0
00407024  int32_t data_407024 = 0x0
00407028  int32_t data_407028 = 0x0

0040702c                                       00 00 00 00
00407030  00 00 00 00 00 00 00 00-00 00 00 00

0040703c  int32_t data_40703c = 0x0
00407040  int32_t data_407040 = 0x0

00407044              00 00 00 00-00 00 00 00 00 00 00 00
00407050  00 00 00 00 00 00 00 00-00 00 00 00

0040705c  int32_t data_40705c = 0x0

00407060  00 00 00 00

00407064  int32_t data_407064 = 0x0
00407068  int32_t data_407068 = 0x0
0040706c  int32_t data_40706c = 0x0
00407070  int32_t data_407070 = 0x0

.bss section ended  {0x407000-0x407074}
```

```
.bss section started  {0x404000-0x404074}
00404000  int32_t data_404000 = 0x0
00404004  int32_t data_404004 = 0x0

00404008                           00 00 00 00 00 00 00 00
00404010  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00

00404020  int32_t data_404020 = 0x0
00404024  int32_t data_404024 = 0x0
00404028  int32_t data_404028 = 0x0

0040402c                                       00 00 00 00
00404030  00 00 00 00 00 00 00 00-00 00 00 00

0040403c  int32_t data_40403c = 0x0
00404040  int32_t data_404040 = 0x0

00404044              00 00 00 00-00 00 00 00 00 00 00 00
00404050  00 00 00 00 00 00 00 00-00 00 00 00

0040405c  int32_t data_40405c = 0x0

00404060  00 00 00 00

00404064  int32_t data_404064 = 0x0
00404068  int32_t data_404068 = 0x0
0040406c  int32_t data_40406c = 0x0
00404070  int32_t data_404070 = 0x0

.bss section ended  {0x404000-0x404074}
```

Figure 15: The BSS section of the original file (left) and after swapping the sections (right). Only the variable locations have changed.



```
.data section started  {0x404000-0x404020}
00404000  int32_t data_404000 = 0x0
00404004  int32_t data_404004 = 0x1da
00404008  int32_t data_404008 = 0x2
0040400c  int32_t data_40400c = -0x3
00404010  int32_t data_404010 = 0x4000
00404014  int32_t data_404014 = 0x403bd0
00404018  int32_t data_404018 = -0x1
0040401c  int32_t data_40401c = 0x0

.data section ended  {0x404000-0x404020}
```

```
.data section started  {0x407000-0x407020}
00407000  int32_t data_407000 = 0x0
00407004  int32_t data_407004 = 0x1da
00407008  int32_t data_407008 = 0x2
0040700c  int32_t data_40700c = -0x3
00407010  int32_t data_407010 = 0x4000
00407014  int32_t data_407014 = 0x403bd0
00407018  int32_t data_407018 = -0x1
0040701c  int32_t data_40701c = 0x0

.data section ended  {0x407000-0x407020}
```

Figure 16: The .data section of the original file (left) and after swapping the sections (right). Only the variable locations have changed.

The result is an executable which works, and which can still be viewed in both Binary Ninja and IDA Pro. Before the references were resolved, the file was not executable, implying that sections cannot just be swapped without updating the dependencies. It is likely that packers which try to compress the sections will preserve their existing order, to prevent the unpacking stub from being too complicated.

Parsers which use recursive descent parsing would not be easily confused by malformed files with shuffled sections. IDA Pro and Binary Ninja are both recursive descent parsers, meaning that they identify the headers and sections of the PE file first, before recursively descending through the code to disassemble, rather than simply assuming all code is aligned sequentially from the entry point. This makes recursive descent parsers more reliable to investigate malformed files, in contrast to parsers which make assumptions about section order.

5.4 Anti-Debugging

Anti-debugging techniques make it difficult for malware analysts to understand what is going on in the attack payload. There are three main ways of anti-debugging, the first involving a check on whether a debugger is attached to the malware (Sections 5.4.1, 5.4.2 and 5.4.3). When debuggers are detected, the malware will exit and delete its traces. That way, the malware can only execute the attack payload when no debugger is present. The second method is to prevent debuggers from attaching to the process (Section 5.4.4), and the third method is to prevent debuggers from creating memory dumps of the process (Section 5.4.5).

5.4.1 API-based Debugger Detection

IsDebuggerPresent and FindWindow are part of the Windows API, and allow malware to check if debuggers are present. Below are two sample programs implementing these methods minimally.

5.4.1.1 IsDebuggerPresent

```c
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

int main() {
    while (1) {
        if (IsDebuggerPresent()) {
            printf("there is a debugger");
        }
    }

    return 0;
}
```

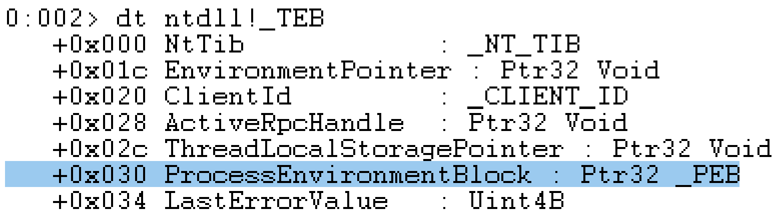
Listing 2: DebuggerCheck.c

IsDebuggerPresent detects invasive forms of debugging. Invasive debugging entails the debugger attaching to the process and being able to modify the control flow. The IsDebuggerPresent routine checks for the BeingDebugged flag in the Process Environment Block (PEB), and moves the flag value to the *eax* register.

BeingDebugged flag in PEB

The PEB can be found in the Thread Environment Block (TEB) of the running thread. The TEB can be displayed in WinDbg (Figure 17):

```
dt ntdll!_TEB
```

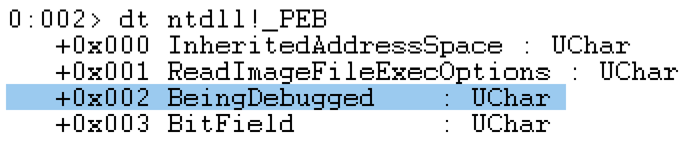


```
0:002> dt ntdll!_TEB
   +0x000 NtTib            : _NT_TIB
   +0x01c EnvironmentPointer : Ptr32 Void
   +0x020 ClientId         : _CLIENT_ID
   +0x028 ActiveRpcHandle  : Ptr32 Void
   +0x02c ThreadLocalStoragePointer : Ptr32 Void
   +0x030 ProcessEnvironmentBlock : Ptr32 _PEB
   +0x034 LastErrorValue   : Uint4B
```

Figure 17: Displaying the TEB in WinDbg

It can be seen that in 32-bit programs, the PEB is at an offset of 0x30 in the TEB. We can view the details of the PEB in WinDbg, and see that the BeingDebugged flag is located at offset 2 of the PEB (Figure 18):

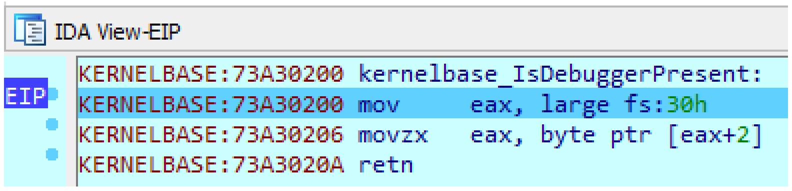
```
dt ntdll!_PEB
```



```
0:002> dt ntdll!_PEB
   +0x000 InheritedAddressSpace : UChar
   +0x001 ReadImageFileExecOptions : UChar
   +0x002 BeingDebugged     : UChar
   +0x003 BitField          : UChar
```

Figure 18: Displaying the PEB in WinDbg

In the assembly disassembly of IsDebuggerPresent, we can then see the routine moving the PEB information, and then the BeingDebugged flag to the *eax* register (Figure 19).



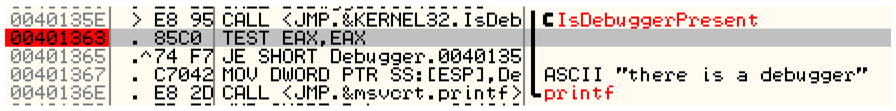
```
KERNELBASE:73A30200 kernelbase_IsDebuggerPresent:
KERNELBASE:73A30200 mov     eax, large fs:30h
KERNELBASE:73A30206 movzx   eax, byte ptr [eax+2]
KERNELBASE:73A3020A retn
```

Figure 19: Assembly routine of IsDebuggerPresent

To achieve a more obfuscated code, the malware can mimic the assembly code and directly pull the value from the BeingDebugged flag in the PEB instead of making the API call through IsDebuggerPresent.

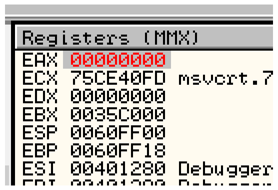
Patching IsDebuggerPresent

When the malware analyst identifies the use of IsDebuggerPresent, it is then a simple task to patch the return value to the *eax* register and bypass the check. First, a breakpoint is inserted just after the IsDebuggerPresent routine returns the value to the *eax* register (Figure 20). With a debugger present, the value in the *eax* register will be set to 1. During the breakpoint, the malware analyst can patch the value of the register to 0 (Figure 21). The breakpoint and patching menus can be accessed by right-clicking the respective command and register.



```
0040135E  > E8 95 CALL <JMP.&KERNEL32.IsDeb  IsDebuggerPresent
00401363  . 85C0  TEST EAX,EAX
00401365  .^74 F7 JE SHORT Debugger.0040135
00401367  . C7042 MOV DWORD PTR SS:[ESP],De  ASCII "there is a debugger"
0040136E  . E8 2D CALL <JMP.&msvcrt.printf>  printf
```

Figure 20: Inserting a breakpoint before test eax, eax



```
Registers (MMX)
EAX 00000000
ECX 75CE40FD msvcrt.7
EDX 00000000
EBX 0035C000
ESP 0060FF00
EBP 0060FF18
ESI 00401280 Debugger
```

Figure 21: Setting *eax* register to 0

5.4.1.2 FindWindow

FindWindow works in a similar manner as IsDebuggerPresent, by using an API call to search for the debugger window.

```c
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

int main() {
    while(1) {
        HANDLE hOlly = FindWindow(TEXT("OLLYDBG"), NULL);

        if(hOlly) {
            printf("OllyDbg is open!");
            ExitProcess(0);
        }
    }
}
```

Listing 3: WindowFinder.c

FindWindow is a more specific method than IsDebuggerPresent, as the malware author needs to know which windows to look out for. Some common x86 debuggers include OllyDbg, IDA Pro and WinDbg, but this list is not exhaustive.

Like IsDebuggerPresent, FindWindow also uses an API call, making it vulnerable to patching by malware analysts. After the call to FindWindowA, the handle value is stored in the *eax* register and should be zeroed.

Nevertheless, compared to IsDebuggerPresent, FindWindow is less overt as an anti-analysis API call. Legitimate, benign programs are more likely to use FindWindow.

5.4.2 Exception-based Debugger Detection

In using API calls, malware become susceptible to simple runtime patching. To increase the anti-analysis complexity, malware authors have used exception-based techniques to detect debuggers. The core idea behind exception-based debugger detection is to use debugger-specific functions that will raise exceptions when no debuggers are present.

5.4.2.1 OutputDebugString

OutputDebugString is a function used to send logging and debugging information to debuggers. When a debugger is present, the string is successfully sent to the debugger and no exceptions are raised. Without debuggers attached to the program, OutputDebugString raises an exception.

```c
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

void main() {
    DWORD userValue = 12345;    // user-defined value
    SetLastError(userValue);
    OutputDebugString("this string is for the debugger");
    if( GetLastError() == userValue ) {
        // no additional error raised
        printf("Debugger detected");
    } else {
        printf("No debugger detected");
        printf("\n Last error: %d", GetLastError());
    }

}
```

Listing 4: CheckOutputDebugStringLastError.c

The OutputDebugString program was run in Windows 10, in the presence and absence of debuggers (OllyDbg). However, both scenarios resulted in ”debugger detected”. This output suggests that even without debuggers present, the LastError value was not modified.

As the literature being referenced could be outdated, we looked at the Windows OutputDebugString documentation.

> If the application has no debugger, the system debugger displays the string if the filter mask allows it. (Note that this function calls the DbgPrint function to display the string. For details on how the filter mask controls what the system debugger displays, see the DbgPrint function in the Windows Driver Kit (WDK) on MSDN.) **If the application has no debugger and the system debugger is not active, OutputDebugString does nothing. Prior to Windows Vista: The system debugger does not filter content**

The documentation, contrary to most literature, states that nothing happens when no debuggers are present. Nevertheless, there is an exception listed for systems prior to Windows Vista, suggesting that the implementation of OutputDebugString changed with the introduction of Windows Vista.

We set out to test the OutputDebugString on various operating systems.

Operating System	Debugger Present	Result
Windows 10	✓	Debugger detected
	✗	**Debugger detected**
Windows 7 (VMWare)	✓	Debugger detected
	✗	**Debugger detected**
Windows XP (VMWare)	✓	Debugger detected
	✗	No debugger detected

Table 4: CheckOutputDebugStringLastError results

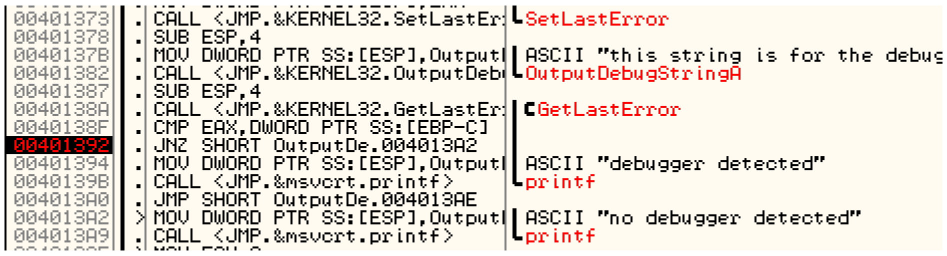
As shown, OutputDebugString works only on Windows XP, due to a change in implementation. On Windows XP, without a debugger, error code 2 is raised. In all, OutputDebugString is a more subtle way to detect debuggers than API calls like IsDebuggerPresent or FindWindow, but it is outdated. A large majority of the user base has already moved away from Windows XP systems.

Patching OutputDebugString

For completeness, a demonstration of how to patch OutputDebugString in Windows XP is shown. OutputDebugString works by reading the error value, but it is not possible to patch this value in OllyDbg. Instead, the call can be patched by modifying the jump instruction at the if-else statement.

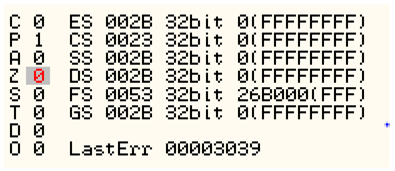
This if-else patching only works when the if-else block is found in the malware. The technique is also applicable to the other anti-analysis methods which rely on if-else checking.

The first method is to change the flag values, similar to patching register values (Figures 22 and 23).



```
00401373 . CALL <JMP.&KERNEL32.SetLastEr  └SetLastError
00401378 . SUB ESP,4
0040137B . MOV DWORD PTR SS:[ESP],Outputl ┌ASCII "this string is for the debug
00401382 . CALL <JMP.&KERNEL32.OutputDeb  └OutputDebugStringA
00401387 . SUB ESP,4
0040138A . CALL <JMP.&KERNEL32.GetLastEr  [GetLastError
0040138F . CMP EAX,DWORD PTR SS:[EBP-C]
00401392 . JNZ SHORT OutputDe.004013A2
00401394 . MOV DWORD PTR SS:[ESP],Outputl ┌ASCII "debugger detected"
0040139B . CALL <JMP.&msvcrt.printf>      └printf
004013A0 . JMP SHORT OutputDe.004013AE
004013A2 > MOV DWORD PTR SS:[ESP],Outputl ┌ASCII "no debugger detected"
004013A9 . CALL <JMP.&msvcrt.printf>      └printf
```

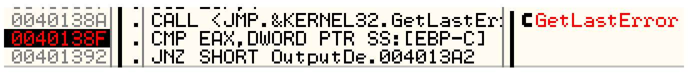
Figure 22: Inserting a breakpoint at the JNZ instruction



```
C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 0  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 26B000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr 00003039
```

Figure 23: Resetting the Z flag to 0

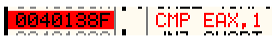
The second method is to change the instruction being executed. The cmp instruction (Figure 24) checks if the LastError value is the same as the user-defined value.



```
0040138A  .  CALL <JMP.&KERNEL32.GetLastEr  GetLastError
0040138F  .  CMP EAX,DWORD PTR SS:[EBP-C]
00401392  .  JNZ SHORT OutputDe.004013A2
```

Figure 24: cmp instruction before patching

To change the result, we make the cmp instruction compare the LastError value with another value (Figure 25) .



```
0040138F  .  CMP EAX,1
```

Figure 25: cmp instruction after patching

Here, we demonstrate the benefits of setting the LastError value at the start of the program. Firstly, we obfuscate the cmp instruction. By default, the LastError value is set to 0, so without setting a user-defined value, the cmp instruction merely compares with a 0 value. Secondly, we do not make assumptions about the default LastError value being 0. The malware program can then run other routines that may modify the LastError value.

5.4.3 Privilege-checking-based Debugger Detection

Windows programs can run with different privileges. A normal user-mode program will not have privileges like SeDebugPrivilege enabled. SeDebugPrivilege allows debuggers and the debugged processes to inspect and adjust the memory of other processes. However, when processes are being debugged, they inherit the privileges of the debugger, which has SeDebugPrivilege enabled. It is then possible to detect if a process is being debugged by checking if the process has the SeDebugPrivilege enabled.

5.4.3.1 Opening csrss.exe

Csrss (client/ Server Runtime Subsystem) is a privileged process. In Windows XP and Windows 7, csrss could only be opened by processes with SeDebugPrivilege enabled. The following program attempts to open csrss.exe and checks if ERROR_ACCESS_DENIED was thrown, indirectly checking for the SeDebugPrivilege.

```c
#include <stdio.h>
#include <windows.h>

typedef HANDLE (*_CsrGetProcessId)();

void main() {
    HMODULE nt = GetModuleHandle("ntdll.dll");
    _CsrGetProcessId CsrGetProcessId =
        (_CsrGetProcessId) GetProcAddress(nt,"CsrGetProcessId");

    HANDLE pid = CsrGetProcessId();
    HANDLE process = OpenProcess(PROCESS_ALL_ACCESS, FALSE, (DWORD) pid);

    if (GetLastError() == ERROR_ACCESS_DENIED) {
        printf("no debugger detected");
    } else {
        printf("debugger detected");
    }

}
```

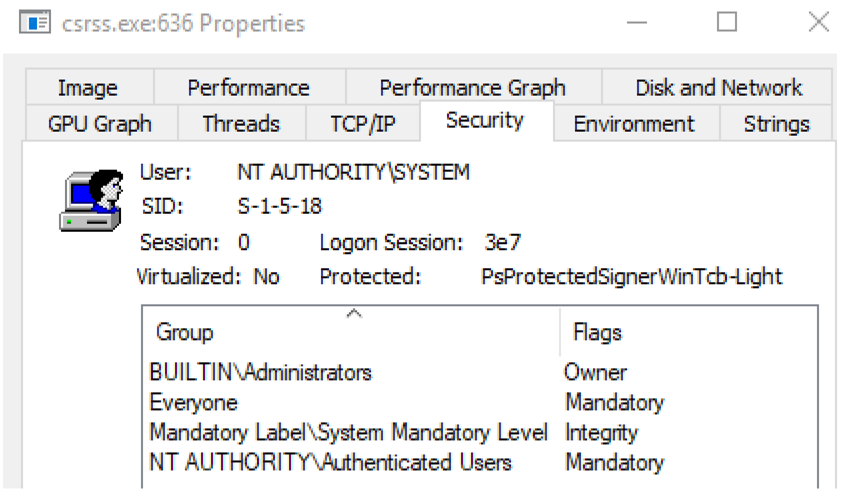
Listing 5: OpenCsrss.c

A summary of the results in different operating systems is shown below (Table 5). As mentioned earlier, opening csrss.exe shows the expected results in Windows XP systems (tested with VirtualBox and VMWare).

Operating System	Debugger Present	Result
Windows 10	✓	**No debugger detected**
	✗	No debugger detected
Windows 7 (VMWare)	✓*	Debugger detected
	✗	No debugger detected
Windows XP (VMWare)	✓	Debugger detected
	✗	No debugger detected

Table 5: OpenCsrss results
∗ The debugger must have elevated privileges

Mandatory Integrity Control (MIC), an integrity level-based privilege control system, was introduced in **Windows Vista**. In the MIC system, there are four integrity levels (LOW, MEDIUM, HIGH, SYSTEM), with processes having MEDIUM integrity by default and elevated processes having HIGH integrity. In **Windows 7** systems, granting SeDebugPrivilege to processes with HIGH integrity allows these processes to access SYSTEM integrity processes. However, in **Windows 10**, processes with SeDebugPrivilege were not able to open Csrss.exe because of a modified protection level system introduced. Csrss has a protection at the Windows Trusted Computing Base (WinTcb) level (Figure 26).



User: NT AUTHORITY\SYSTEM
SID: S-1-5-18
Session: 0 Logon Session: 3e7
Virtualized: No Protected: PsProtectedSignerWinTcb-Light

Group	Flags
BUILTIN\Administrators	Owner
Everyone	Mandatory
Mandatory Label\System Mandatory Level	Integrity
NT AUTHORITY\Authenticated Users	Mandatory

Figure 26: Csrss.exe properties in ProcessExplorer (Windows 10)

Previously in OutputDebugString, the program always detected the presence of a debugger (false positive). However, in OpenCsrss, the program was always unable to open csrss and hence always detected the absence of a debugger (false negative). In the context of debugger detection, false positives would limit the effectiveness of the malware spread as the program would self-destruct on almost every system. On the contrary, false negatives would mean that the malware continues executing without being aware of the presence of debuggers, exposing its internal processes to malware analysts.

5.4.3.2 Reading SeDebugPrivilege value

Instead of implicitly testing for SeDebugPrivilege, we can directly read the process token to check if SeDebugPrivilege is enabled.

```c
#include <stdio.h>
#include <windows.h>

BOOL CheckSeDebugPrivilege() {

    LUID luid;
    PRIVILEGE_SET privs;
    HANDLE hProcess;
    HANDLE hToken;
    hProcess = GetCurrentProcess();

    if (!OpenProcessToken(hProcess, TOKEN_QUERY, &hToken)) {
        return FALSE;
    }

    if (!LookupPrivilegeValue(NULL, SE_DEBUG_NAME, &luid)) {
        return FALSE;
    }

    privs.PrivilegeCount = 1;
    privs.Control = PRIVILEGE_SET_ALL_NECESSARY;
    privs.Privilege[0].Luid = luid;
    privs.Privilege[0].Attributes = SE_PRIVILEGE_ENABLED;

    BOOL result;
    PrivilegeCheck(hToken, &privs, &result);
    CloseHandle(hToken);

    return result;
}

void main() {
    int result = CheckSeDebugPrivilege();
    DWORD pid = GetCurrentProcessId();

    printf("Process %d, SeDebugPrivilege: %x\n", pid, result);

    if (result == 1) {
        printf("debugger detected");
    } else {
        printf("no debugger detected");
    }
}
```

Listing 6: ReadSeDebugPrivilegeValue.c

ReadSeDebugPrivilegeValue was tested on different operating systems, and the results are shown below.

Operating System	Debugger Present	Result
Windows 10	✓	Debugger detected
	✗	No debugger detected
Windows 7 (VMWare)	✓*	Debugger detected
	✗	No debugger detected
Windows XP (VMWare)	✓	Debugger detected
	✗	No debugger detected

Table 6: ReadSeDebugPrivilegeValue results
$*$ The debugger must have elevated privileges

5.4.4 Anti-Attaching Techniques

Earlier in the section, numerous ways of detecting debuggers were analysed. However, anti-debugging methods can be more complex than just simply detecting a debugger. These anti-attaching techniques focus on preventing external debuggers like WinDbg from invasively attaching to the malicious process. Invasive debuggers have the ability to suspend all the threads in the debuggee, and access the debuggee's memory and registers.

5.4.4.1 Self-Debugging

Each process can only have one debugger attached to itself. Hence, one such anti-attaching technique is for the malware to debug itself. In self-debugging, the malware process spawns and debugs a child process, and in turn this child process is debugging the parent process. Spawning another process is necessary because a process cannot debug itself (which involves suspending its own threads).

```c
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <windows.h>

int successfulDebugging = -1;

// Method for parent to spawn and debug child process
void spawnAndDebugChild() {
    HANDLE hProcess = NULL;
    PROCESS_INFORMATION pi;
    STARTUPINFO si;
    ZeroMemory(&pi, sizeof(PROCESS_INFORMATION));
    ZeroMemory(&si, sizeof(STARTUPINFO));

    GetStartupInfo(&si);
    char commandArgs[1024];
    sprintf(commandArgs,"%s %d",GetCommandLine(), (int) GetCurrentProcessId());

    CreateProcess(NULL, commandArgs, NULL, NULL, FALSE,
        DEBUG_PROCESS, NULL, NULL, &si, &pi);
}

// Parent program
void executeParentProgram() {
    spawnAndDebugChild();
    DEBUG_EVENT DbgEvent;

    while(1) {
        WaitForDebugEvent(&DbgEvent, INFINITE);

        if (DbgEvent.dwDebugEventCode == EXIT_PROCESS_DEBUG_EVENT) {
            ExitProcess(-1);
        } else if (DbgEvent.dwDebugEventCode == OUTPUT_DEBUG_STRING_EVENT) {
            printf("this is the malicious payload");
            ExitProcess(0);
        }

        ContinueDebugEvent(DbgEvent.dwProcessId, DbgEvent.dwThreadId,
            DBG_CONTINUE);
    }
}

// Method for child to debug parent
void *debugParent (void *pid_void_ptr) {
    DWORD *pid_ptr = (DWORD *)pid_void_ptr;
    successfulDebugging = DebugActiveProcess(*pid_ptr);

    DEBUG_EVENT DbgEvent;

    while(1) {
        WaitForDebugEvent(&DbgEvent, INFINITE);
        ContinueDebugEvent(DbgEvent.dwProcessId, DbgEvent.dwThreadId,
            DBG_CONTINUE);
    }
}

// Child program
void executeChildProgram(DWORD pid) {
    pthread_t debugParentThread;
    pthread_create(&debugParentThread, NULL, debugParent, &pid);

    // Update debugging status for parent thread
    while (successfulDebugging == -1) {
        if (successfulDebugging == 0) {
            ExitProcess(-1);
        } else {
            OutputDebugString("success");
        }
    }

    pthread_join(debugParentThread, NULL);
}

void main(int argc, char *argv[]) {
    if (argc > 1) {
        executeChildProgram((DWORD) atoi(argv[1]));
    } else {
        executeParentProgram();
    }
}
```

Listing 7: SelfDebugging.c

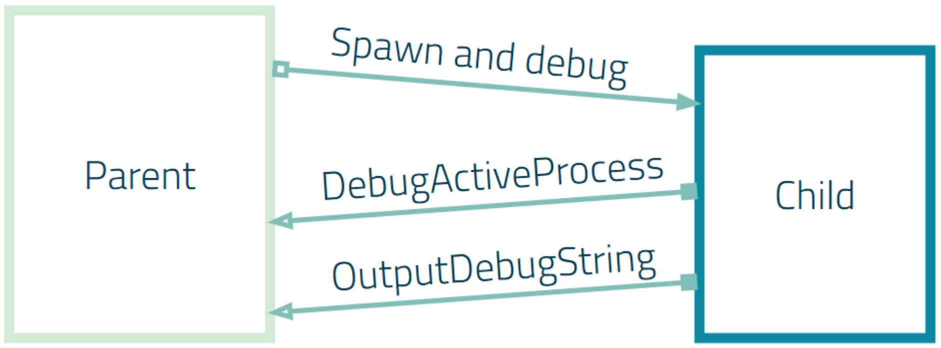


Figure 27: Self-debugging Diagram

The self-debugging workflow is as follows:

1. The executable is run by the user. The parent process is started.
2. The parent process spawns and debug the child process. Debugging is achieved via the DEBUG_PROCESS process creation flag.
3. Another instance of the executable is spawned, with the process ID of the parent process passed in as a system argument.
4. Since there is an argument passed into the execution, then the child process is started.
5. The child process spawns a thread that attempts to debug the parent process. A new thread is needed because the current thread is used to prevent a deadlock later on. Since the child and parent processes are listening to debug events and handling breaks from each other, we would be unable to send a debug signal on single-threaded processes.
6. The DebugActiveProcess call in the new thread updates the $successfulDebugging$ global flag, so that the other thread in the child process can read its value.
7. The initial value of the $successfulDebugging$ flag is set to a neutral value, so the child debugger only sends a signal after the DebugActiveProcess routine has been completed.
8. The child debugger checks if the DebugActiveProcess was successful. If successful, OutputDebugString is called, sending an OUTPUT_DEBUG_STRING_EVENT signal to the parent process. Else if debugging the parent process was not successful, ExitProcess is called, sending an EXIT_PROCESS_DEBUG_EVENT to the parent process.
9. The parent process is actively listening for debug events. If the parent process detects that the child process has exited, it will also exit.
10. The parent waits until the child debugger has sent an OUTPUT_DEBUG_STRING_EVENT before executing the malicious payload.
11. When the malicious payload is done executing, the parent process exits. Since the parent process was debugging the child process, the child process also exits.

As the malicious payload is executing, external debuggers like WinDbg will be unable to attach to the program. This technique can be demonstrated by executing an infinite while loop as the malicious program, and attempting to attach to the process with WinDbg.

5.4.4.2 Debugger Detachment

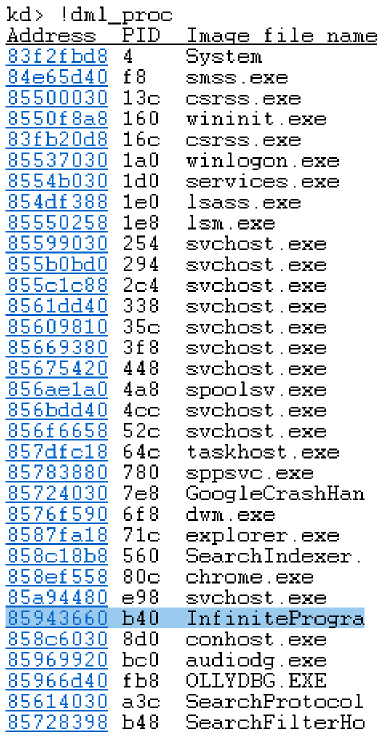
However, self-debugging programs can be defeated by manually detaching the debugger. The malware analyst can run the self-debugging program in a virtual environment, then access the EPROCESS structure of program. The EPROCESS structure is a kernel-mode representation of a process, containing information like the PEB which was mentioned earlier (Section 5.4.1.1). While the PEB could be accessed in user-mode, most other information in the EPROCESS can only be accessed via kernel-mode debugging.

The following experiments were done using a Windows 10 host and a Windows 7 Guest (VMWare), connected via a serial port.

DebugPort value in EPROCESS

The EPROCESS address can be retrieved by running the extension:

```
!dml_proc
```



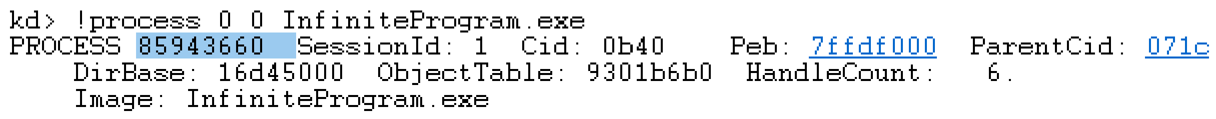
```
kd> !dml_proc
Address  PID  Image file name
83f2fbd8 4    System
84e65d40 f8   smss.exe
85500030 13c  csrss.exe
8550f8a8 160  wininit.exe
83fb20d8 16c  csrss.exe
85537030 1a0  winlogon.exe
8554b030 1d0  services.exe
854df388 1e0  lsass.exe
85550258 1e8  lsm.exe
85599030 254  svchost.exe
855b0bd0 294  svchost.exe
855c1c88 2c4  svchost.exe
8561dd40 338  svchost.exe
85609810 35c  svchost.exe
85669380 3f8  svchost.exe
85675420 448  svchost.exe
856ae1a0 4a8  spoolsv.exe
856bdd40 4cc  svchost.exe
856f6658 52c  svchost.exe
857dfc18 64c  taskhost.exe
85783880 780  sppsvc.exe
85724030 7e8  GoogleCrashHan
8576f590 6f8  dwm.exe
8587fa18 71c  explorer.exe
858c18b8 560  SearchIndexer.
858ef558 80c  chrome.exe
85a94480 e98  svchost.exe
85943660 b40  InfinitePrgra
858c6030 8d0  conhost.exe
85969920 bc0  audiodg.exe
85966d40 fb8  OLLYDBG.EXE
85614030 a3c  SearchProtocol
85728398 b48  SearchFilterHo
```

Figure 28: Displaying the list of active processes in WinDbg

Alternatively, the program name can be looked up:

```
!process 0 0 InfiniteProgram.exe
```

The parameters 0, 0 present a shortened version of information to display. Information on the EPROCESS address and PEB address are shown.



```
kd> !process 0 0 InfiniteProgram.exe
PROCESS 85943660  SessionId: 1  Cid: 0b40    Peb: 7ffdf000  ParentCid: 071c
    DirBase: 16d45000  ObjectTable: 9301b6b0  HandleCount:   6.
    Image: InfiniteProgram.exe
```

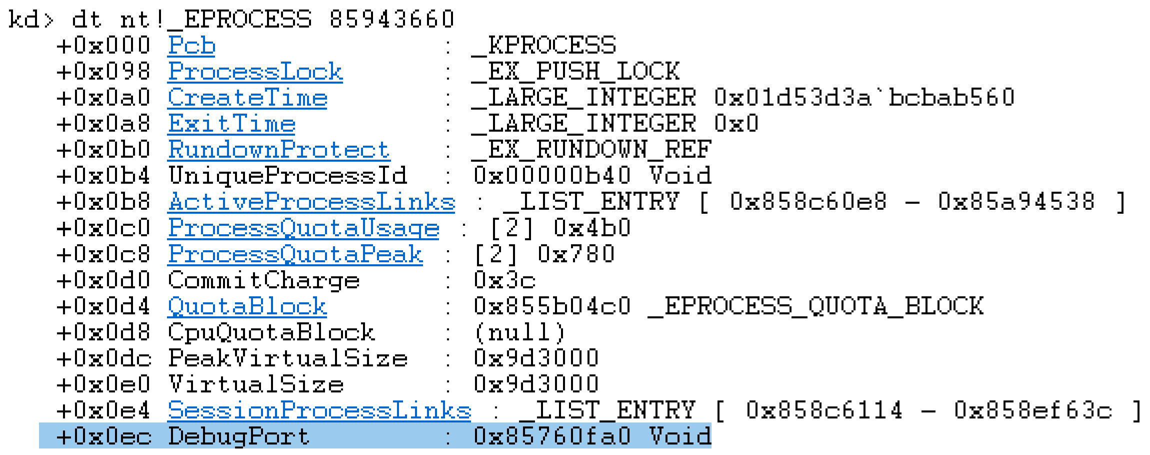
Figure 29: Displaying process information in WinDbg

To display the EPROCESS structure, the following command is used:

```
dt nt!_EPROCESS <EPROCESS address>
```

<EPROCESS address> is an optional argument. When supplied, the fields of the EPROCESS structure are populated.

In particular, we are interested in the DebugPort of the EPROCESS. In 32-bit Windows 7, the DebugPort is at offset 0x0ec. When a process is being debugged, the DebugPort contains the address of the debug object handle (Figure 30).



```
kd> dt nt!_EPROCESS 85943660
   +0x000 Pcb              : _KPROCESS
   +0x098 ProcessLock      : _EX_PUSH_LOCK
   +0x0a0 CreateTime       : _LARGE_INTEGER 0x01d53d3a`bcbab560
   +0x0a8 ExitTime         : _LARGE_INTEGER 0x0
   +0x0b0 RundownProtect   : _EX_RUNDOWN_REF
   +0x0b4 UniqueProcessId  : 0x00000b40 Void
   +0x0b8 ActiveProcessLinks : _LIST_ENTRY [ 0x858c60e8 - 0x85a94538 ]
   +0x0c0 ProcessQuotaUsage : [2] 0x4b0
   +0x0c8 ProcessQuotaPeak : [2] 0x780
   +0x0d0 CommitCharge     : 0x3c
   +0x0d4 QuotaBlock       : 0x855b04c0 _EPROCESS_QUOTA_BLOCK
   +0x0d8 CpuQuotaBlock    : (null)
   +0x0dc PeakVirtualSize  : 0x9d3000
   +0x0e0 VirtualSize      : 0x9d3000
   +0x0e4 SessionProcessLinks : _LIST_ENTRY [ 0x858c6114 - 0x858ef63c ]
   +0x0ec DebugPort        : 0x85760fa0 Void
```

Figure 30: DebugPort of a process being debugged

We can retrieve more information on the debug handle by running the extension:

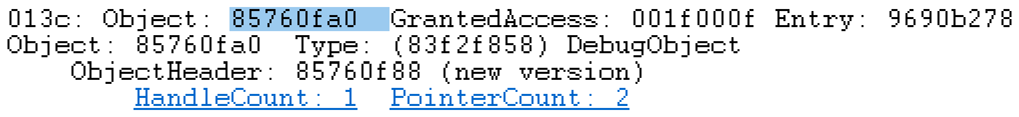
```
!handle
```

Note that if the process context is set using:

```
.process <EPROCESS address>
```

then the EPROCESS address should be the EPROCESS address of the debugger process and not the debuggee process.

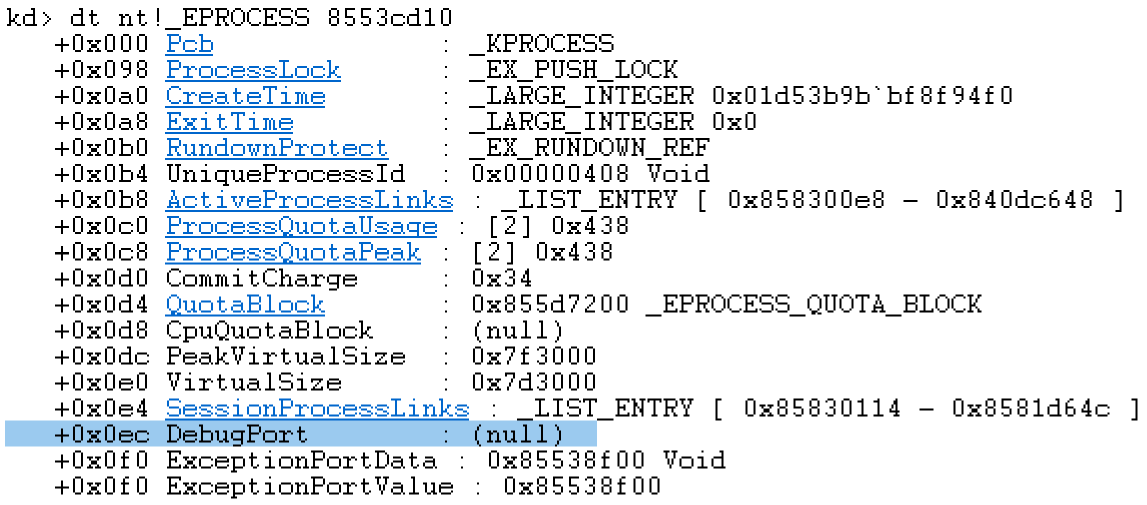
The DebugObject handle can be seen, and the address of the object is the value stored in the DebugPort.



```
013c: Object: 85760fa0  GrantedAccess: 001f000f Entry: 9690b278
Object: 85760fa0  Type: (83f2f858) DebugObject
    ObjectHeader: 85760f88 (new version)
        HandleCount: 1  PointerCount: 2
```

Figure 31: DebugObject handle address

On the contrary, when a process is not being debugged, the DebugPort would be 0 (Figure 32).



```
kd> dt nt!_EPROCESS 8553cd10
   +0x000 Pcb              : _KPROCESS
   +0x098 ProcessLock      : _EX_PUSH_LOCK
   +0x0a0 CreateTime       : _LARGE_INTEGER 0x01d53b9b`bf8f94f0
   +0x0a8 ExitTime         : _LARGE_INTEGER 0x0
   +0x0b0 RundownProtect   : _EX_RUNDOWN_REF
   +0x0b4 UniqueProcessId  : 0x00000408 Void
   +0x0b8 ActiveProcessLinks : _LIST_ENTRY [ 0x858300e8 - 0x840dc648 ]
   +0x0c0 ProcessQuotaUsage : [2] 0x438
   +0x0c8 ProcessQuotaPeak : [2] 0x438
   +0x0d0 CommitCharge     : 0x34
   +0x0d4 QuotaBlock       : 0x855d7200 _EPROCESS_QUOTA_BLOCK
   +0x0d8 CpuQuotaBlock    : (null)
   +0x0dc PeakVirtualSize  : 0x7f3000
   +0x0e0 VirtualSize      : 0x7d3000
   +0x0e4 SessionProcessLinks : _LIST_ENTRY [ 0x85830114 - 0x8581d64c ]
   +0x0ec DebugPort        : (null)
   +0x0f0 ExceptionPortData : 0x85538f00 Void
   +0x0f0 ExceptionPortValue : 0x85538f00
```

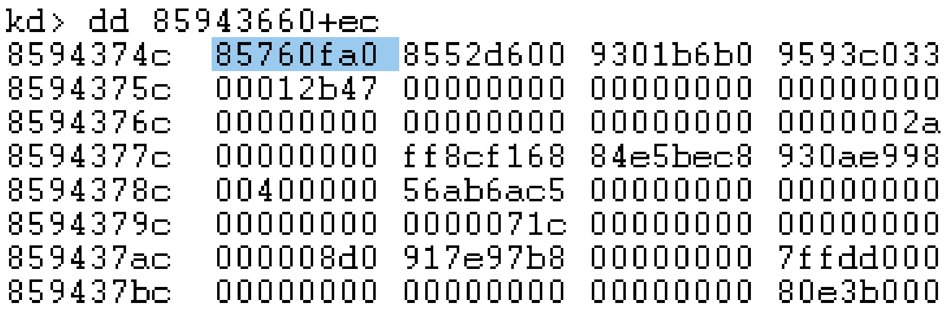
Figure 32: DebugPort of a process not debugged

BeingDebugged flag in PEB

Another indicator of a program being debugged is the BeingDebugged flag in the PEB. Earlier, we referenced this value for the call to the IsDebuggerPresent method. The key difference between the BeingDebugged flag in the PEB and the DebugPort in the EPROCESS is that the BeingDebugged flag is only informative. On the contrary, the DebugPort value affects the debugging status of the process.

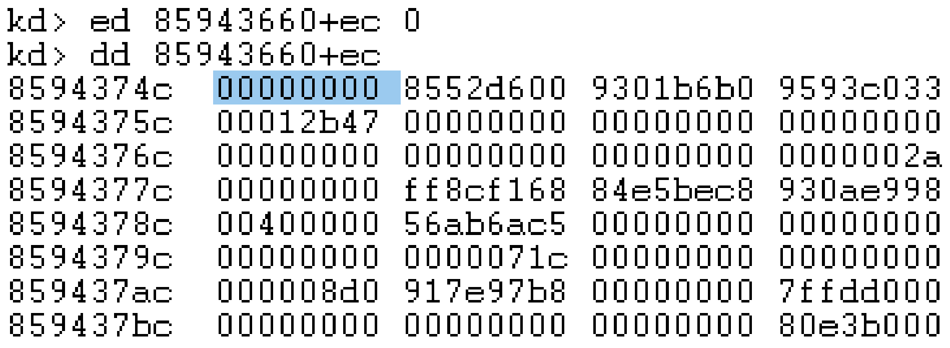
Setting DebugPort value in EPROCESS to 0

When the DebugPort of a process being debugged is set to 0 (Figures 33 and 34), the debugger is detached from the process.



```
kd> dd 85943660+ec
8594374c  85760fa0 8552d600 9301b6b0 9593c033
8594375c  00012b47 00000000 00000000 00000000
8594376c  00000000 00000000 00000000 0000002a
8594377c  00000000 ff8cf168 84e5bec8 930ae998
8594378c  00400000 56ab6ac5 00000000 00000000
8594379c  00000000 0000071c 00000000 00000000
859437ac  000008d0 917e97b8 00000000 7ffdd000
859437bc  00000000 00000000 00000000 80e3b000
```

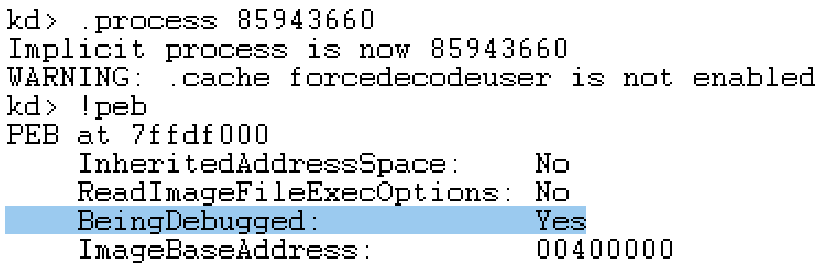
Figure 33: DebugPort value before being set to 0



```
kd> ed 85943660+ec 0
kd> dd 85943660+ec
8594374c  00000000 8552d600 9301b6b0 9593c033
8594375c  00012b47 00000000 00000000 00000000
8594376c  00000000 00000000 00000000 0000002a
8594377c  00000000 ff8cf168 84e5bec8 930ae998
8594378c  00400000 56ab6ac5 00000000 00000000
8594379c  00000000 0000071c 00000000 00000000
859437ac  000008d0 917e97b8 00000000 7ffdd000
859437bc  00000000 00000000 00000000 80e3b000
```

Figure 34: DebugPort value after being set to 0

The debuggee is now unable to send debugging messages to the debugger via the communication port (DebugPort). As such, the debugger has been detached and another debugger can attach to the process. However, the BeingDebugged flag is still set at this point (Figure 35), since the BeingDebugged flag is only loaded at the time of execution. This flag has no effect on debugger communication since it is only informative.



```
kd> .process 85943660
Implicit process is now 85943660
WARNING: .cache forcedecodeuser is not enabled
kd> !peb
PEB at 7ffdf000
    InheritedAddressSpace:    No
    ReadImageFileExecOptions: No
    BeingDebugged:            Yes
    ImageBaseAddress:         00400000
```

Figure 35: BeingDebugged flag after DebugPort is set to 0

When another debugger attaches to the process, a new handle value at the DebugPort is set.

Setting BeingDebugged flag in PEB to 0

Since the BeingDebugged flag is only informative, setting the BeingDebugged flag to 0 does not detach the debugger. While resetting the flag will fool the IsDebuggerPresent API call (Section 5.4.1.1), it is more efficient to simply patch the return value of the API call.

5.4.5 Anti-Dumping Techniques

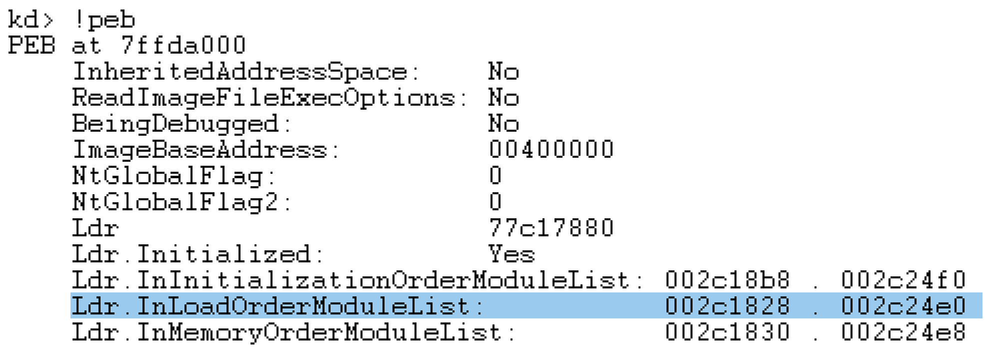
Dumpers like OllyDumpEx have the ability to take snapshots of the executable as it runs. As shown earlier, dumping is useful for analysts to recover and analyse the unpacked state of files. OllyDumpEx is a very common tool that relies on routines like ReadProcessMemory and VirtualAlloc. It is also known that OllyDumpEx uses the SizeOfImage from the PEB as an argument for the size parameter of these functions.

A popular anti-dumping technique is then to modify the value of SizeOfImage in the PEB (at runtime) to a very large value. This way, the dumping would not work. Most literature do not specify the expected behaviour of anti-dumping techniques, so we work to explore this question in this section.

The following experiments are done using a 32-bit InfiniteProgram, with OllyDumpEx as the dumper. Modifying the PEB is done via kernel debugging from a Windows 10 host to a Windows 7 guest (VMWare).

5.4.5.1 Modifying SizeOfImage in PEB

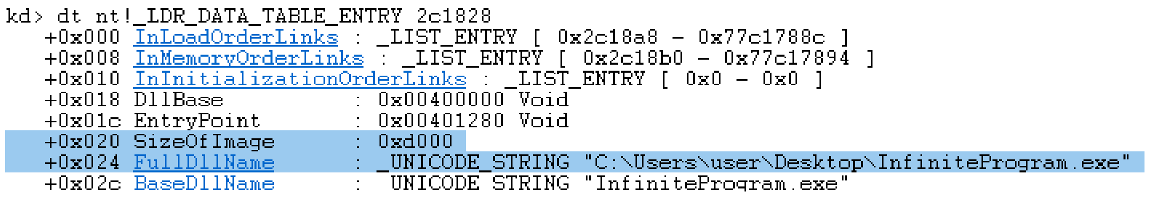
Every module loaded into memory has an image size that is stored in the PEB (Figure 36).



```
kd> !peb
PEB at 7ffda000
    InheritedAddressSpace:    No
    ReadImageFileExecOptions: No
    BeingDebugged:            No
    ImageBaseAddress:         00400000
    NtGlobalFlag:             0
    NtGlobalFlag2:            0
    Ldr                       77c17880
    Ldr.Initialized:          Yes
    Ldr.InInitializationOrderModuleList: 002c18b8 . 002c24f0
    Ldr.InLoadOrderModuleList:           002c1828 . 002c24e0
    Ldr.InMemoryOrderModuleList:         002c1830 . 002c24e8
```

Figure 36: InLoadOrderModuleList in PEB

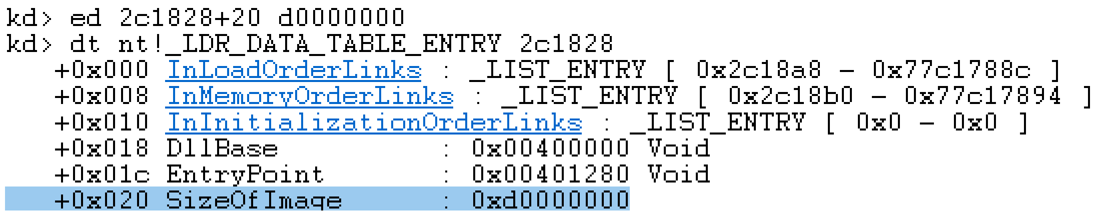
InLoadOrderModuleList is a doubly linked list containing one entry for every module loaded. Every entry points to a LDR_MODULE structure for the module loaded. Since the modules are shown in load order, the first module loaded is the executable itself (e.g. InfiniteProgram.exe).



```
kd> dt nt!_LDR_DATA_TABLE_ENTRY 2c1828
   +0x000 InLoadOrderLinks : _LIST_ENTRY [ 0x2c18a8 - 0x77c1788c ]
   +0x008 InMemoryOrderLinks : _LIST_ENTRY [ 0x2c18b0 - 0x77c17894 ]
   +0x010 InInitializationOrderLinks : _LIST_ENTRY [ 0x0 - 0x0 ]
   +0x018 DllBase          : 0x00400000 Void
   +0x01c EntryPoint       : 0x00401280 Void
   +0x020 SizeOfImage      : 0xd000
   +0x024 FullDllName      : _UNICODE_STRING "C:\Users\user\Desktop\InfiniteProgram.exe"
   +0x02c BaseDllName      : _UNICODE_STRING "InfiniteProgram.exe"
```

Figure 37: InLoadOrderModuleList in PEB

I edited the SizeOfImage value in the PEB, as shown below (Figure 38).



```
kd> ed 2c1828+20 d0000000
kd> dt nt!_LDR_DATA_TABLE_ENTRY 2c1828
   +0x000 InLoadOrderLinks : _LIST_ENTRY [ 0x2c18a8 - 0x77c1788c ]
   +0x008 InMemoryOrderLinks : _LIST_ENTRY [ 0x2c18b0 - 0x77c17894 ]
   +0x010 InInitializationOrderLinks : _LIST_ENTRY [ 0x0 - 0x0 ]
   +0x018 DllBase          : 0x00400000 Void
   +0x01c EntryPoint       : 0x00401280 Void
   +0x020 SizeOfImage      : 0xd0000000
```

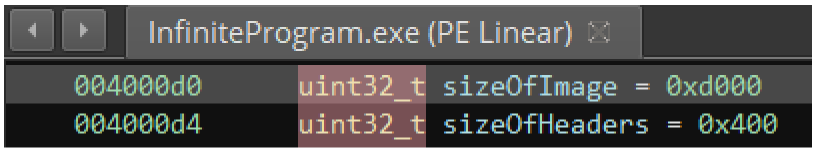
Figure 38: Modifying SizeOfImage in LDR_MODULE

However, OllyDumpEx was still able to successfully dump the file. The dumped executable could be run. We then tried variants of modifying the size of image, because OllyDumpEx could have other methods of calculating the image size.

Instead of reading the size of image value from the PEB, OllyDumpEx could have retrieved the value from the PE header. We first examined modifying the SizeOfImage value in the PE header in memory. Other than directly reading a field value for the image size, OllyDumpEx could have summed up the sizes of the sections. Hence, we examined modifying the size of the last section in memory. It is not possible to change these values on disk because the executable loading will raise an error and the file cannot be run.

5.4.5.2 Modifying SizeOfImage in PE header in memory

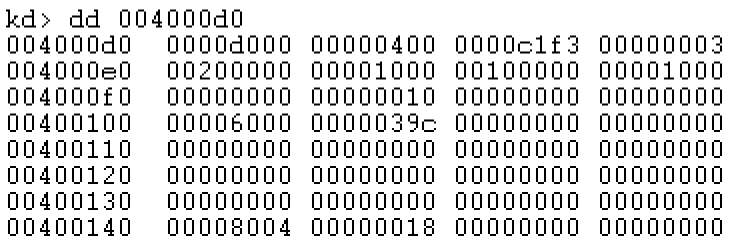
The memory address of the SizeOfImage value can be found using Binary Ninja.



```
004000d0    uint32_t sizeOfImage = 0xd000
004000d4    uint32_t sizeOfHeaders = 0x400
```

Figure 39: Address of SizeOfImage in PE header in memory

In WinDbg, the process context can be set to the current process, then the editing of the SizeOfImage value in the PE header in memory can be done.



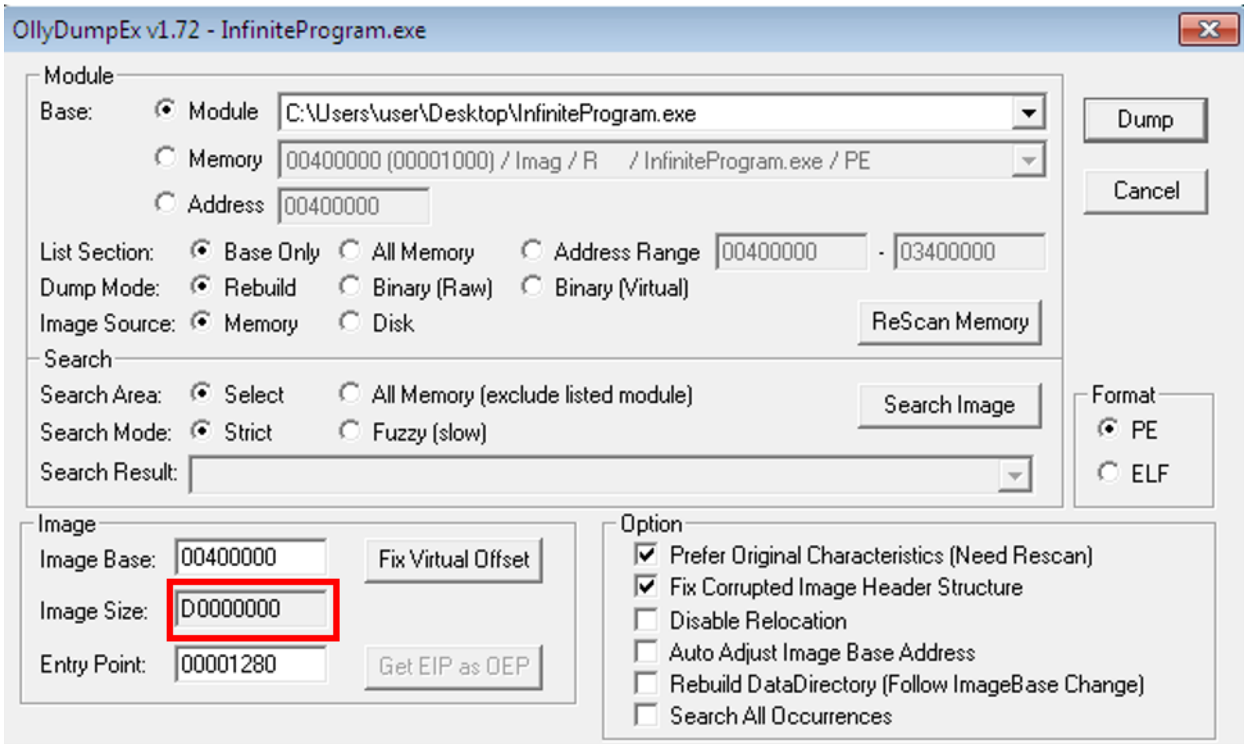
```
kd> dd 004000d0
004000d0  0000d000 00000400 0000c1f3 00000003
004000e0  00200000 00001000 00100000 00001000
004000f0  00000000 00000010 00000000 00000000
00400100  00006000 0000039c 00000000 00000000
00400110  00000000 00000000 00000000 00000000
00400120  00000000 00000000 00000000 00000000
00400130  00000000 00000000 00000000 00000000
00400140  00008004 00000018 00000000 00000000
```

Figure 40: SizeOfImage value before being set to a large number

```
kd> ed 004000d0 d0000000
kd> dd 004000d0
004000d0  d0000000 00000400 0000c1f3 00000003
004000e0  00200000 00001000 00100000 00001000
004000f0  00000000 00000010 00000000 00000000
00400100  00006000 0000039c 00000000 00000000
00400110  00000000 00000000 00000000 00000000
00400120  00000000 00000000 00000000 00000000
00400130  00000000 00000000 00000000 00000000
00400140  00008004 00000018 00000000 00000000
```

Figure 41: SizeOfImage value after being set to a large number

As a result of this change, OllyDumpEx will incorrectly read the SizeOfImage as the increased value.


OllyDumpEx v1.72 - InfiniteProgram.exe

Module
Base: [✓ Module] C:\Users\user\Desktop\InfiniteProgram.exe
Memory 00400000 (00001000) / Imag / R / InfiniteProgram.exe / PE
Address 00400000
List Section: [✓ Base Only] All Memory Address Range 00400000 - 03400000
Dump Mode: [✓ Rebuild] Binary (Raw) Binary (Virtual)
Image Source: [✓ Memory] Disk

Search
Search Area: [✓ Select] All Memory (exclude listed module)
Search Mode: [✓ Strict] Fuzzy (slow)
Search Result:

Format: [✓ PE] ELF

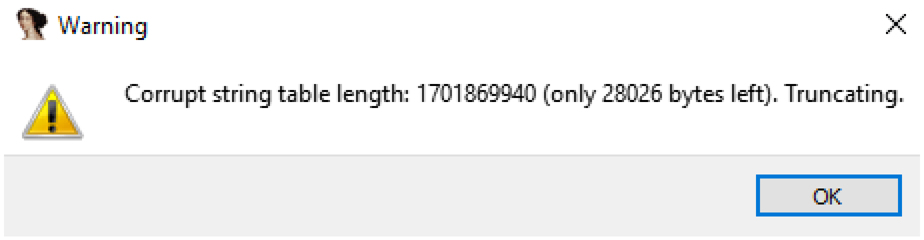
Image
Image Base: 00400000
Image Size: D0000000
Entry Point: 00001280

Option
- [✓] Prefer Original Characteristics (Need Rescan)
- [✓] Fix Corrupted Image Header Structure
- [ ] Disable Relocation
- [ ] Auto Adjust Image Base Address
- [ ] Rebuild DataDirectory (Follow ImageBase Change)
- [ ] Search All Occurrences

Figure 42: OllyDumpEx incorrectly reading the SizeOfImage value

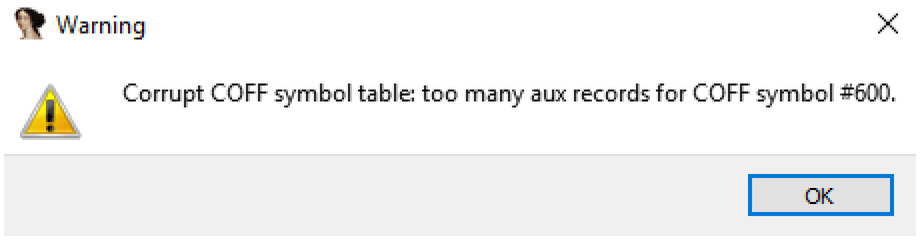
However, OllyDumpEx could still successfully dump the file. We then analysed if this dumped executable had lost any information.

When opening the dumped executable in IDA Pro, warnings were issued about a corrupt string table length (Figure 43) and COFF symbol table (Figure 44).


Warning

Corrupt string table length: 1701869940 (only 28026 bytes left). Truncating.

Figure 43: Corrupt string table length warning


Warning

Corrupt COFF symbol table: too many aux records for COFF symbol #600.

Figure 44: Corrupt COFF symbol table warning

However, other than these warnings, there were no visible differences between the dumped executable and the original executable. The dumped executable had its SizeOfImage value restored to the original value. The strings and import table were identical. Additionally, the disassembly result of the main function was identical up to naming of functions and variables.

This change to the file proved the most promising in terms of its anti-dumping functionality. More investigation can be done into the intended effects of anti-dumping. Furthermore, editing the SizeOfImage exploits knowledge about the implementation of OllyDumpEx. Over the years, OllyDumpEx could have implemented improved checks for corrupt file handling, as indicated in their v1.00 bug fix on 12 March 2013:

> Improve: PE Header parse and modify more carefully (corrupt PE handling)

5.4.5.3 Modifying size of last section in PE header in memory

Perhaps, the size of sections were used as checksums in OllyDumpEx's program. The address of the VirtualSize field of the last section can be located using Binary Ninja. Then, the VirtualSize was increased to a large value (from 0xd1 to 0xd00000d1). At this point, OllyDumpEx would still read the image size as 0xd000.

Using the technique in Section 5.4.5.2, the SizeOfImage in the PE header was concurrently edited. OllyDumpEx now reads the image size as 0xd0000000, but can still successfully dump the executable.

The anti-dumping tricks were not successfully replicated, so this is an area that future work can focus on.

5.5 Anti-Virtual Environment

The purpose of detecting virtual machines parallels that of why malware would want to detect the presence of debuggers. Execution in virtual environments signals to the malware that an analyst is attempting to capture and scrutinise its behaviour. Virtual environments aim to replicate physical systems as closely as possibly, but the tools provided to make virtualisation powerful also leak information that the machine is virtualised. We look at a simple way to detect virtual environments - by querying the registry of a system.

5.5.1 Registry Query

```cpp
#include <windows.h>
#include <iostream>
using namespace std;

int main() {
    HKEY hkey;
    int n = RegOpenKeyEx(HKEY_LOCAL_MACHINE,
        TEXT("SYSTEM\\CurrentControlSet\\Control\\VirtualDeviceDrivers"),
        0, KEY_QUERY_VALUE, &hkey);

    if ( n == ERROR_SUCCESS ) {
        cout << "VM detected" << endl;
    } else {
        cout << "VM not detected" << endl;
    }

    return 0;
}
```

Listing 8: RegQuery.cpp

Virtual Machine	Operating System	Result
None	Windows 10	VM not detected
VMWare	Windows XP	VM detected
	Windows 7	VM detected
VirtualBox	Windows XP	VM detected
	Windows 7	VM detected

Table 7: RegQuery results

While the registry entry may not appear in every virtualised environment, its existence in two popular virtualisation technologies (VMWare and VirtualBox) makes this VM-detection technique effective. A more comprehensive study can be done on other registry entries that are unique to virtual environments. Another registry entry that can be queried is the Network Address registry, to lookup the machine's MAC address. VMWare and VirtualBox have known OUIs, and VMWare machines must adhere to the 00:50:56:XX:YY:ZZ format to function correctly.

6 Specification

Throughout the document, we have made references to different tools, programs and operating systems. Their full specification is shown here.

- Virtualisation Technologies
  - VMWare Workstation 15 Player
  - VirtualBox 6.0.8
- Operating Systems
  - Windows XP SP3 Home Edition (32-bit)
  - 7 Home Premium (32-bit)
  - Windows 10 Enterprise N (64-bit)
- Tools
  - ProcMon 3.52
  - UPX 3.95w
  - PEiD 0.95
  - Detect It Easy 2.04
  - BinaryNinja 1.1.1689 demo (Build ID 856ac082)
- Compilers
  - gcc/g++ 5.1.0 (tdm-1)
- Debuggers
  - OllyDbg 1.10
  - WinDbg 10.0.18362.1 (x86)

7 Conclusion

We have covered ground on how malware employ anti-analysis techniques to make it difficult to analyse their behaviour. These anti-analysis techniques work against different types of static and dynamic analysis, which are key pillars of malware analysis.

8 Future Work

In each area of our analysis, we have identified future directions to expand the project on.

1. Packer Detection Techniques
   - Create tools to automatically generate permutations of changing byte strings
   - Reverse engineer PEiD to examine and evade its packer detection techniques
2. Tools and Procedures for Unpacking Binaries
   - Automate the unpacking process by inserting instrumentation code to perform checks to verify at which point the unpacking stub has already finished decompressing code and transferred control to the OEP (for instance, checking if the location the tail jump jumps to is a location which the unpacking stub has already written to before).
   - Exploring how to unpack packers with anti-debugging tricks
3. PE Malformation
   - Successfully shifting the PE header into the overlay
   - Documenting the requirements of the Windows loader
   - Try other forms of PE malformation and document ways of replicating and bypassing them
4. Anti-Debugging
   - Explore anti-dumping functionality by understanding how the SizeOfImage and VirtualSize come together
   - Develop a more complicated self-debugging architecture, where part of the attack payload is executed in the parent process and part of the attack payload is executed in the child process
   - Develop a detection and patching tool for anti-debugging techniques
5. Anti-Virtual Environment
   - Location of IDTs as a VM-detecting capability

9 References

1. OllyDumpEx. Retrieved from https://low-priority.appspot.com/ollydumpex/

2. Detect It Easy. Retrieved from http://ntinfo.biz/index.html

3. Christodorescu, M. (2005). Semantics-aware malware detection. *IEEE Explore.* Retrieved from https://ieeexplore.ieee.org/abstract/document/1425057

4. PEiD External Database. Retrieved from https://raw.githubusercontent.com/ynadji/peid/master/userdb.txt

5. Joshua T. (2008). An Anti-Reverse Engineering Guide. Retrieved from https://www.codeproject.com/Articles/30815/An-Anti-Reverse-Engineering-Guide

6. Tyler S. (2011). Anti-Debugging A Developers View. *Veracode Inc., USA.* Retrieved from https://www.secnews.pl/wp-content/uploads/2011/05/whitepaper_antidebugging.pdf

7. SeDebugPrivilege and Integrity Level. Retrieved from http://windbg.xyz/article/view/108-SeDebugPrivilege-and-Integrity-Level

8. (2013). Self Debugging code. Retrieved from https://www.unknowncheats.me/forum/c-and-c-/128293-self-debugging-code.html

9. walied. (2011). Debuggers Anti-Attaching Techniques - Part 1. Retrieved from http://waleedassar.blogspot.com/2011/12/debuggers-anti-attaching-techniques.html

10. (2000). Undocumented functions of NTDLL - LDR_MODULE. Retrieved from http://undocumented.ntinternals.net/index.html?page=UserMode%2FStructures%2FLDR_MODULE.html

11. al-khaser. Retrieved from https://github.com/LordNoteworthy/al-khaser

12. VMWare Worksation (2019). Maintaining and Changing the MAC Address of a Virtual Machine. Retrieved from https://www.vmware.com/support/ws5/doc/ws_net_advanced_mac_address.html