04 - Code Analysis & Windows Malware Analysis & Static Analysis Blocking

CYS5120 - Malware Analysis

Bahcesehir University

Cyber Security Msc Program

Dr. Ferhat Ozgur Catak [1] Mehmet Can Doslu [2]

[1]ozgur.catak@tubitak.gov.tr

[2]mehmetcan.doslu@tubitak.gov.tr

2017-2018 Fall

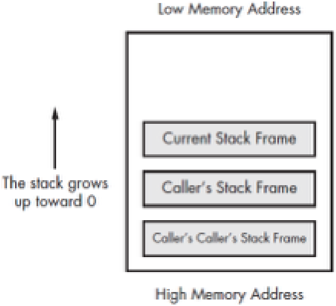
Table of Contents

Table of Contents

Stack Operations I

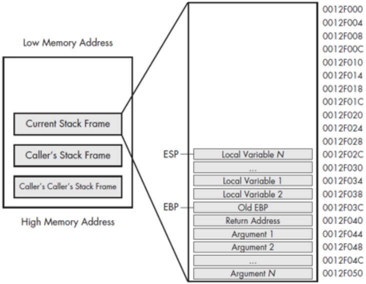
Stack Operations

- **NOP** does not take any action. Also, the command **xhcg eax, eax** does not take any action because it replaces **eax** with **eax**.
    - These types of commands are called **No Operation** or **NOP** for short. In some **buffer overflow** attacks, such cases can be observed to **exhaust code memory**.
- Stack functions are structures that contain local variables and flow control. LIFO data input and output operations are performed.



Stack Operations II

- A new stack frame is created for each new function call.
- Each function contains the return address as well as the parameters it uses in its stack frame.



Stack Operations III

There are two sets of instructions that are used as the basis for writing or reading data in the stack.

```
push 0xdeadbeef ; Put value to stack
pop eax         ; EAX value 0xdeadbeef
```

PUSH

- Write a value to the ESP register.
- Decrease the address value in the ESP register by the size of the operand. (The process is going forward)

POP

- Take the value at the top of the stack and copy the corresponding register.
- Increase the address value in the ESP register by the size of the processed data. (The process is going back)

Disassembler & Debugger I

Disassembler

- It is a program that **converts the machine code into assembly language**. They are usually written in high-level languages.
- It is the **most important tool** of reverse engineering applications. They help make machine code readable.

Debugger

- It is a program that helps to find and reduce bugs in an applications.
- They allow testing and debugging to be performed.
- An instruction set runs the code on the simulator and the functioning of the program can be understood. They are also used in reverse engineering applications.

Disassembler & Debugger II

Debugger Tools

- OllyDbg
- Radare2
- GNU Debugger
- WinDbg
- IDA Pro

Disassembler Tools

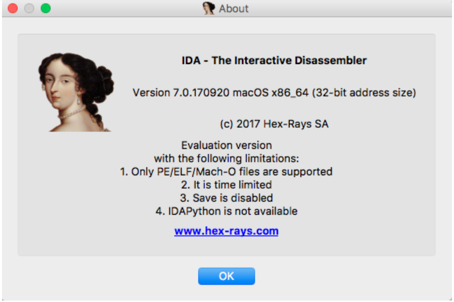
- ODA (OnlineDisassembler)
- IDA Pro
- OllyDbg
- Objdump, hexdump
- PE Explorer

IDA Pro I

IDA Pro

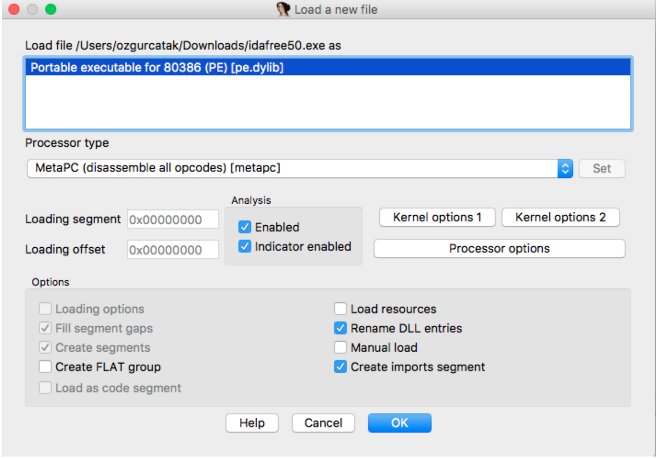
- The Interactive Disassembler Professional (IDA Pro) is an **extremely powerful disassembler** distributed by Hex-Rays.
- It is the disassembler of choice for many **malware analysts**, **reverse engineers**, and **vulnerability analysts**.
- "IDA Pro" generates assembly source code from executable files.
- Performs automatic code analysis.
- Works on Windows, Linux, Mac OS X
- Supports different processors (x86, x64, ARM, etc.)

IDA Pro II



- Some supported file formats
  - PE, COFF(Common Object File Format), ELF, Mach-O
  - Dalvik (Android bytecode)
  - Sony Playstation PSX
  - Plugin can be written using Python and IDC
  - Can generate C / C ++ code with HexRays plugin

IDA Pro III


Load a new file
Load file /Users/ozgurcatak/Downloads/idafree50.exe as
Portable executable for 80386 (PE) [pe.dylib]
Processor type
MetaPC (disassemble all opcodes) [metapc]
Set
Analysis
Loading segment 0x00000000
Loading offset 0x00000000
Enabled
Indicator enabled
Kernel options 1
Kernel options 2
Processor options
Options
Loading options
Fill segment gaps
Create segments
Create FLAT group
Load as code segment
Load resources
Rename DLL entries
Manual load
Create imports segment
Help
Cancel
OK

IDA Pro IV

Steps

- When an executable file is opened in IDA Pro application, IDA Pro analyzes this file and architecture.
- When the file is opened, it is first formatted as a *raw binary* directory.
- This feature may include information such as whether there is a *Shell* script in the code, encryption parameters, etc., and it is very useful for malware analysis.

Actions

- Detection of functions
- Stack Analysis
- Local variable identification
- Viewing Text

IDA Pro V

FLIRT

IDA Pro includes extensive code signatures within its **Fast Library Identification and Recognition Technology** (FLIRT), which allows it to recognize and l**abel a disassembled function**, especially library code added by a compiler.

- IDA Pro is meant to be interactive, and all aspects of its disassembly process can be modified, manipulated, rearranged, or redefined. One of the best aspects of IDA Pro is its ability to save your analysis progress:
  - You can add comments, label data, and name functions, and then save your work in an IDA Pro database (known as an idb) to return to later.
- IDA Pro also has robust support for **plug-ins**, so you can write **your own extensions** or leverage the **work of others**.

The IDA Pro Interface I

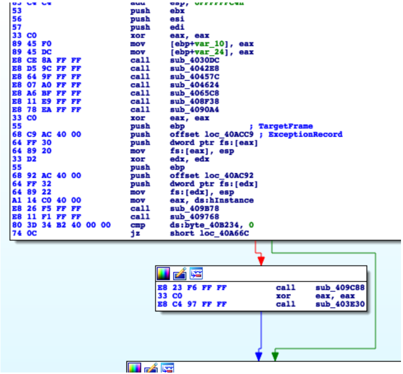
Disassembly Window Modes

- You can display the disassembly window in one of two modes: **graph** and **text**.
- To switch between modes, press the **spacebar**.

The IDA Pro Interface II

Graph Mode

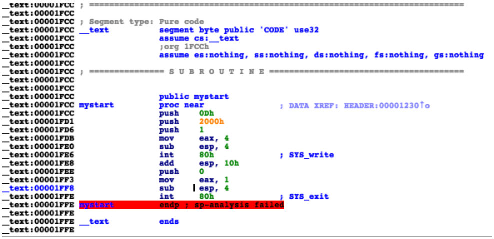
- In graph mode, IDA Pro excludes certain information that we recommend you display, such as **line numbers** and **operation codes**
- The color and direction of the arrows help show the program's flow during analysis
- The arrow's color tells you whether the path is based on a particular decision having been made:
  - **red** if a conditional jump is not taken,
  - **green** if the jump is taken,
  - **blue** for an unconditional jump.



The IDA Pro Interface III

Text Mode

- The text mode of the disassembly window is a more traditional view.
- If you are still learning assembly code, you should find the auto comments feature of IDA Pro useful. To turn on this feature, select **Options** ← **General**, and then check the Auto comments checkbox. This adds additional comments throughout the disassembly window to aid your analysis.



```
__text:00001FCC ; ===========================================================================
__text:00001FCC
__text:00001FCC ; Segment type: Pure code
__text:00001FCC __text          segment byte public 'CODE' use32
__text:00001FCC                 assume cs:__text
__text:00001FCC                 ;org 1FCCh
__text:00001FCC                 assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
__text:00001FCC
__text:00001FCC ; =============== S U B R O U T I N E =======================================
__text:00001FCC
__text:00001FCC
__text:00001FCC                 public mystart
__text:00001FCC mystart         proc near               ; DATA XREF: HEADER:00001230↑o
__text:00001FCC                 push    0Dh
__text:00001FD1                 push    2000h
__text:00001FD6                 push    1
__text:00001FDB                 mov     eax, 4
__text:00001FE0                 sub     esp, 4
__text:00001FE6                 int     80h             ; SYS_write
__text:00001FE8                 add     esp, 10h
__text:00001FEE                 push    0
__text:00001FF3                 mov     eax, 1
__text:00001FF8                 sub     esp, 4
__text:00001FFE                 int     80h             ; SYS_exit
__text:00001FFE mystart         endp ; sp-analysis failed
__text:00001FFE
__text:00001FFE __text          ends
__text:00001FFE
```

Useful Windows for Analysis

Useful Windows for Analysis

Functions Lists all functions in the executable and shows the length of each.

- This window also associates flags with each function (*F*, *L*, *S*, and so on), the most useful of which, *L*, indicates library functions.
- The *L* flag can save you time during analysis, because you can identify and skip these compiler-generated functions.

Names Lists every address with a name, including functions, named code, named data, and strings.

Strings Shows all strings. By default, this list shows only ASCII strings longer than five characters.

Imports Lists all imports for a file.

Exports Lists all the exported functions for a file. This window is useful when you're analyzing DLLs.

Structures Lists the layout of all active data structures.

Lab I

Lab 1

- **Malware**: tnnbtib.exe
- **Tool**: IDA Pro Freeware 5.0
  - Investigation of the report of malware on the internet (gratis)
  - Opening malware with IDA Pro
  - Unpacking with UPX Unpacker
  - Examination of malware with IDA Pro

Lab 2

- **Malware**: Lab05-01.dll
- **Tool**: IDA Pro
  - What is the address of DllMain?
  - Use the Imports window to browse to *gethostbyname*. Where is the import located?
  - Focusing on the call to *gethostbyname* located at *0x10001757*, can you figure out which DNS request will be made?
  - Use the Strings window to locate the string **cmd.exe /c** in the disassembly. Where is it located?
  - What is happening in the area of code that references **cmd.exe /c**?

Table of Contents

Analyzing Malicious Windows Programs

- Most malware **targets Windows platforms** and **interacts closely with the OS**.
- A solid understanding of **basic Windows coding concepts** will allow you to identify host-based indicators of malware, **follow malware as it uses the OS** to execute code without a jump or call instruction, and determine the malware's purpose.

The Windows API I

The Windows API

- **The Windows API** is a **broad set of functionality** that governs the way that **malware interacts with the Microsoft libraries**.
- The Windows API is **so extensive** that developers of Windows-only applications have **little need for thirdparty libraries**.
- The Windows API uses **certain terms**, **names**, and **conventions** that you should become familiar with before turning to specific functions.

The Windows API II

Types and Hungarian Notation

- Much of the Windows API uses its own names to represent C types.
  - For example, the *DWORD* and *WORD* types represent 32-bit and 16-bit unsigned integers.
- Standard C types like *int*, *short*, and *unsigned int* are not normally used.
- Windows generally uses *Hungarian notation* for API function identifiers.
  - For example, if the third argument to the *VirtualAllocEx* function is *dwSize*, you know that it's a *DWORD*.

The Windows API III

Type and prefix	Description
WORD (w)	A 16-bit unsigned value.
DWORD (dw)	A double-WORD, 32-bit unsigned value.
Handles (H)	A reference to an object. The information stored in the handle is not documented, and the handle should be manipulated only by the Windows API. Examples include HModule, HInstance, and HKey.
Long Pointer (LP)	A pointer to another type. For example, LPByte is a pointer to a byte, and LPCSTR is a pointer to a character string. Strings are usually prefixed by LP because they are actually pointers. Occasionally, you will see Pointer (P)... prefixing another type instead of LP; in 32-bit systems, this is the same as LP. The difference was meaningful in 16-bit systems.
Callback	Represents a function that will be called by the Windows API. For example, the InternetSetStatusCallback function passes a pointer to a function that is called whenever the system has an update of the Internet status.

Figure: Common Windows API Types

The Windows API IV

Handles

- Handles are items that have been opened or created in the OS, such as a **window**, **process**, **module**, **menu**, **file**, and so on.
- The **CreateWindowEx** function has a simple example of a handle. It returns an **HWND**, which is a handle to a window. Whenever you want to do anything with that window, such as call **DestroyWindow**, you'll need to use that handle.

File System Functions I

File System Functions

- One of the most common ways that malware interacts with the system is by **creating or modifying files**, and distinct filenames or **changes to existing filenames** can make good host-based indicators.

CreateFile

- **CreateFile** This function is used to create and open files. It can open existing **files**, **pipes**, **streams**, and **I/O devices**, and **create new files**.
  - The parameter **dwCreationDisposition** controls whether the **CreateFile** function creates a new file or opens an existing one.

File System Functions II

ReadFile & WriteFile

- **Read/WriteFile** These functions are used for reading and writing to files. Both operate on files as a stream.

```
BOOL WINAPI ReadFile(
  _In_         HANDLE       hFile,
  _Out_        LPVOID       lpBuffer,
  _In_         DWORD        nNumberOfBytesToRead,
  _Out_opt_    LPDWORD      lpNumberOfBytesRead,
  _Inout_opt_ LPOVERLAPPED lpOverlapped
);

BOOL WINAPI WriteFile(
  _In_         HANDLE       hFile,
  _In_         LPCVOID      lpBuffer,
  _In_         DWORD        nNumberOfBytesToWrite,
  _Out_opt_    LPDWORD      lpNumberOfBytesWritten,
  _Inout_opt_ LPOVERLAPPED lpOverlapped
);
```

File System Functions III

CreateFileMapping & MapViewOfFile

- *File mappings* are commonly used by malware writers because they allow a file to be loaded into memory and manipulated easily.
- The **CreateFileMapping** function loads a file from disk into memory.
- The **MapViewOfFile** function returns a pointer to the base address of the mapping, which can be used to access the file in memory.
- The program calling these functions can use the pointer returned from **MapViewOfFile** to read and write anywhere in the file.

```
HANDLE WINAPI CreateFileMapping(
  _In_     HANDLE                hFile,
  _In_opt_ LPSECURITY_ATTRIBUTES lpAttributes,
  _In_     DWORD                 flProtect,
  _In_     DWORD                 dwMaximumSizeHigh,
  _In_     DWORD                 dwMaximumSizeLow,
  _In_opt_ LPCTSTR               lpName
);

LPVOID WINAPI MapViewOfFile(
  _In_ HANDLE hFileMappingObject,
  _In_ DWORD  dwDesiredAccess,
  _In_ DWORD  dwFileOffsetHigh,
  _In_ DWORD  dwFileOffsetLow,
  _In_ SIZE_T dwNumberOfBytesToMap
);
```

Special Files I

Special Files

- Windows has a number of file types that can be accessed much like regular files, but that are not accessed by their drive letter and folder (like c:\\docs).
- Malicious programs often use special files.

Shared Files

- Shared files are special files with names that start with *serverName**share* or \\?*serverName**share*
- They access directories or files in a shared folder stored on a network.
- The \\?\ prefix tells the OS to disable all string parsing, and it allows access to longer filenames.

Special Files II

Files Accessible via Namespaces

- **The Win32 device namespace**, with the prefix \\.\, is often used by malware to access physical devices directly, and read and write to them like a file.
- For example, a program might use the \\.*PhysicalDisk*1 to directly access **PhysicalDisk1** while ignoring its file system, thereby allowing it to *modify the disk in ways that are not possible through the normal API*.
- For example, the *Witty worm* from a few years back accessed *Device**PhysicalDisk*1 via the NT namespace to corrupt its victim's file system.
  - It would open the *Device**PhysicalDisk*1 and write to a random space on the drive at regular intervals, eventually corrupting the victim's OS and rendering it unable to boot.

Special Files III

Alternate Data Streams

- The *Alternate Data Streams* (ADS) feature allows additional data to be added to an existing file within NTFS, essentially adding one file to another.
- ADS data is named according to the convention *normalFile.txt:Stream:$DATA*, which allows a program to read and write to a stream.

The Windows Registry I

The Windows Registry

- The Windows registry is used to store OS and program configuration information, such as settings and options.
- Like the file system, it is a good source of host-based indicators and can reveal useful information about the malware's functionality.

Root key The registry is divided into five top-level sections called **root keys**. Sometimes, the terms **HKEY** and **hive** are also used. Each of the root keys has a particular purpose, as explained next.

Subkey A subkey is like a subfolder within a folder.

Key A key is a folder in the registry that can contain additional folders or values. The root keys and subkeys are both keys.

Value entry A value entry is an ordered pair with a name and value.

Value or data The value or data is the data stored in a registry entry.

The Windows Registry II

Registry Root Keys

- **HKEY_LOCAL_MACHINE (HKLM)**: Stores settings that are global to the local machine
- **HKEY_CURRENT_USER (HKCU)**: Stores settings specific to the current user
- **HKEY_CLASSES_ROOT** Stores information defining types
- **HKEY_CURRENT_CONFIG**: Stores settings about the current hardware configuration, specifically differences between the current and the standard configuration
- **HKEY_USERS**: Defines settings for the default user, new users, and current users

The Windows Registry III

Common Registry Functions

RegOpenKeyEx	Opens a registry for editing and querying.
RegSetValueEx	Adds a new value to the registry and sets its data.
RegGetValue	Returns the data for a value entry in the registry.

The Windows Registry IV

```
0040286F   push    2                  ; samDesired
00402871   push    eax                ; ulOptions
00402872   push    offset SubKey      ; "Software\\Microsoft\\Windows\\CurrentVersion\\Run"
00402877   push    HKEY_LOCAL_MACHINE ; hKey
0040287C ❶call    esi ; RegOpenKeyExW
0040287E   test    eax, eax
00402880   jnz     short loc_4028C5
00402882
00402882 loc_402882:
00402882   lea     ecx, [esp+424h+Data]
00402886   push    ecx                ; lpString
00402887   mov     bl, 1
00402889 ❷call    ds:lstrlenW
0040288F   lea     edx, [eax+eax+2]
00402893 ❸push    edx                ; cbData
00402894   mov     edx, [esp+428h+hKey]
00402898 ❹lea     eax, [esp+428h+Data]
0040289C   push    eax                ; lpData
0040289D   push    1                  ; dwType
0040289F   push    0                  ; Reserved
004028A1 ❺lea     ecx, [esp+434h+ValueName]
004028A8   push    ecx                ; lpValueName
004028A9   push    edx                ; hKey
004028AA   call    ds:RegSetValueExW
```

The Windows Registry V

Analyzing Registry Code in Practice

- The code calls the RegOpenKeyEx function at (1) with the parameters needed to open a handle to the registry key *HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run*.
- The value name at (5) and data at (4) are stored on the stack as parameters to this function, and are shown here as having been labeled by IDA Pro.
- The call to *lstrlenW* at (2) is needed in order to get the size of the data,
- which is given as a parameter to the RegSetValueEx function at (3)

Networking APIs I

Networking APIs

- Malware commonly relies on network functions to do its dirty work.
- There are many Windows API functions for network communication

Networking APIs II

Berkeley Compatible Sockets

- malware most commonly uses Berkeley compatible sockets, functionality that is almost identical on **Windows** and **UNIX** systems.
- Berkeley compatible sockets' network functionality in Windows is implemented in the **Winsock libraries**, primarily in **ws2_32.dll**.

Function	Description
socket	Creates a socket
bind	Attaches a socket to a particular port, prior to the accept call
listen	Indicates that a socket will be listening for incoming connections
accept	Opens a connection to a remote socket and accepts the connection
connect	Opens a connection to a remote socket; the remote socket must be waiting for the connection
recv	Receives data from the remote socket
send	Sends data to the remote socket

Networking APIs III

```
00401041  push    ecx                 ; lpWSAData
00401042  push    202h                ; wVersionRequested
00401047  mov     word ptr [esp+250h+name.sa_data], ax
0040104C  call    ds:WSAStartup
00401052  push    0                   ; protocol
00401054  push    1                   ; type
00401056  push    2                   ; af
00401058  call    ds:socket
0040105E  push    10h                 ; namelen
00401060  lea     edx, [esp+24Ch+name]
00401064  mov     ebx, eax
00401066  push    edx                 ; name
00401067  push    ebx                 ; s
00401068  call    ds:bind
0040106E  mov     esi, ds:listen
00401074  push    5                   ; backlog
00401076  push    ebx                 ; s
00401077  call    esi ; listen
00401079  lea     eax, [esp+248h+addrlen]
0040107D  push    eax                 ; addrlen
0040107E  lea     ecx, [esp+24Ch+hostshort]
00401082  push    ecx                 ; addr
00401083  push    ebx                 ; s
00401084  call    ds:accept
```

Figure: A simplified program with a server socket

Networking APIs IV

The WinINet API

- There is a higher-level API called the WinINet API. The WinINet API functions are stored in **Wininet.dll**.
- The WinINet API implements protocols, such as **HTTP** and **FTP**, at the **application layer**.
- You can gain an understanding of what malware is doing based on the connections that it opens.
  - **InternetOpen** is used to initialize a connection to the Internet.
  - **InternetOpenUrl** is used to connect to a URL (which can be an HTTP page or an FTP resource).
  - **InternetReadFile** works much like the **ReadFile** function, allowing the program to read the data from a file downloaded from the Internet.
- Malware can use the **WinINet API** to **connect to a remote server** and **get further instructions** for execution.

Lab

Lab

Analyze the malware found in the file Lab07-01.exe.

- How does this program ensure that it continues running (achieves persistence) when the computer is restarted? (MalService)
- What is a good network-based signature for detecting this malware? (Network API search)
- What is the purpose of this program? (DDoS Attack)

Table of Contents

Packers

Packers

- Packers have become extremely popular with malware writers because they help malware
  - hide from antivirus software,
  - complicate malware analysis,
  - shrink the size of a malware.
- Most packers are easy to use and are freely available.
- Basic static analysis isn't useful on a packed program; packed malware must be **unpacked before it can be analyzed statically**, which makes analysis more complicated and challenging.

Packer Anatomy I

Packer Anatomy

- In order to unpack an executable, we must **undo the work performed by the packer**.
- All packers take **an executable file as input** and produce **an executable file as output**.
- The packed executable is **compressed**, **encrypted**, or otherwise **transformed**, making it harder to recognize and reverse-engineer.

Packer Anatomy II

The Unpacking Stub

- Nonpacked (Normal) executables are loaded by the OS.
- packed programs, **the unpacking stub** is loaded by the OS, and then the **unpacking stub loads the original program**.
- The **code entry point** for the executable **points to the unpacking stub** rather than the original code.
- The unpacking stub performs three steps:
  - Unpacks the original executable into memory
  - Resolves all of the imports of the original executable
  - Transfers execution to the original entry point (OEP)

Packer Anatomy III

Loading the Executable

- When regular executables load, a loader **reads the PE header on the disk**, and **allocates memory** for each of the executable's sections **based on that header**.
- Packed executables also format the PE header so that the loader will allocate space for the sections, which can come from the original program, or the unpacking stub can create the sections.
- The unpacking stub unpacks the code for each section and copies it into the space that was allocated.

Packer Anatomy IV

The Tail Jump

- Once the unpacking stub is complete, it must transfer execution to the OEP (original entry point).
- The instruction that transfers execution to the OEP is commonly referred to as the *tail jump*.
- A *jump* instruction is the simplest and most popular way to transfer execution.
- Since it's so common, many malicious packers will attempt to obscure this function by using a *ret* or *call* instruction.
- Sometimes the tail jump is obscured with OS functions that transfer control, such as *NtContinue* or *ZwContinue*.

Packer Anatomy V

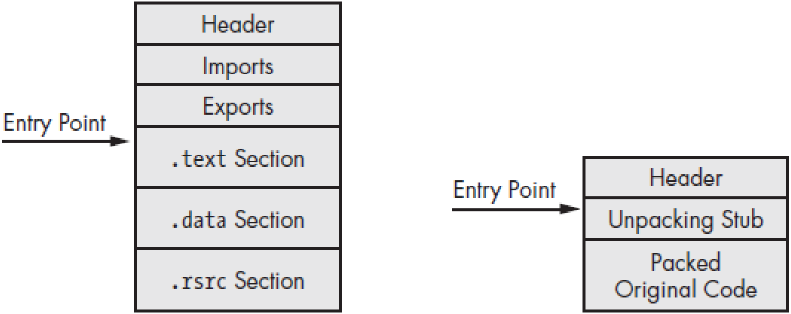
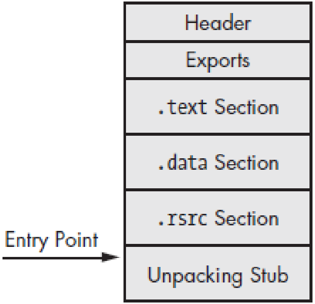


Figure: **(A)** The original executable, prior to packing **(B)** The packed executable, after the original code is packed and the unpacking stub is added

Packer Anatomy VI



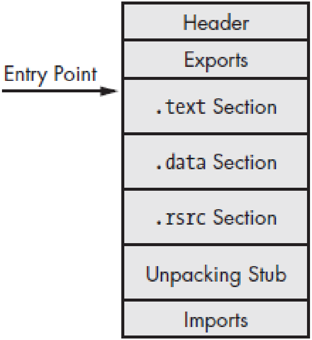


Figure: **(A)** The program after being unpacked and loaded into memory. The unpacking stub unpacks everything necessary for the code to run. The program's starting point still points to the unpacking stub, and there are no imports. **(B)** The fully unpacked program. The import table is reconstructed, and the starting point is back to the original entry point (OEP).

Identifying Packed Programs I

Indicators of a Packed Program

- The program has few imports, and particularly if the only imports are *LoadLibrary* and *GetProcAddress*.
- When the program is opened in IDA Pro, only a small amount of code is recognized by the automatic analysis.
- When the program is opened in OllyDbg, there is a warning that the program may be packed.
- The program shows section names that indicate a particular packer (such as UPX0).
- The program has abnormal section sizes, such as a .text section with a Size of Raw Data of 0 and Virtual Size of nonzero.

Identifying Packed Programs II

Entropy Calculation

- Entropy is a measure of the disorder in a system or program,
- Compressed or encrypted data more closely resembles random data and therefore has **high entropy**; executables that are **not encrypted or compressed have lower entropy**.

Automated Unpacking

Automated Unpacking

- Automated static unpacking programs decompress and/or decrypt the executable.
- This is the **fastest method, and when it works**,
- The best method, since it **does not run the executable**, and it restores the executable to its original state.
- *PE Explorer* comes with several static unpacking plug-ins as part of the default setup.
  - The default plug-ins support *NSPack*, *UPack*, and *UPX*.
  - If PE Explorer detects that a file you've chosen to open is packed, it will automatically unpack the executable.
- Automated dynamic unpackers **run the executable and allow the unpacking stub** to unpack the original executable code

Manual Unpacking

Manual Unpacking

- Sometimes, packed malware can be unpacked automatically by an existing program, but more often it must be unpacked manually.
- There are two common approaches to manually unpacking a program:
  - Discover the packing algorithm and write a program to run it in reverse. By running the algorithm in reverse, the program undoes each of the steps of the packing program. **still inefficient**
  - Run the packed program so that the unpacking stub does the work for you, and then dump the process out of memory, and manually fix up the PE header so that the program is complete. **more efficient**

Anti-disassembly I

Anti-disassembly

- Anti-disassembly uses specially crafted code or data in a program to cause disassembly analysis tools to producean incorrect program listing.
- Malware authors use anti-disassembly techniques to delay or prevent analysis of malicious code. Any code that executes successfully can be reverse-engineered,
- But by armoring their code with anti-disassembly and anti-debugging techniques, **malware authors increase the level of skill required of the malware analyst**.

Anti-disassembly II

Understanding Anti-Disassembly

- When implementing anti-disassembly, the malware author creates a sequence that tricks the disassembler into showing a list of instructions that differ from those that will be executed.
- If the disassembler is tricked into disassembling at the wrong offset, a valid instruction could be hidden from view.

```
                jmp     short near ptr loc_2+1
; ---------------------------------------------------------------------------

loc_2:                                  ; CODE XREF: seg000:00000000j
                call    near ptr 15FF2A71h ❶
                or      [ecx], dl
                inc     eax
; ---------------------------------------------------------------------------
                db    0
```

```
                jmp     short loc_3
; ---------------------------------------------------------------------------
                db 0E8h
; ---------------------------------------------------------------------------

loc_3:                                  ; CODE XREF: seg000:00000000j
                push    2Ah
                call    Sleep ❶
```

Anti-disassembly III

Defeating Disassembly Algorithms

- There are two types of disassembler algorithms: **linear** and **flow-oriented**.
  - **Linear Disassembly**: The linear-disassembly strategy iterates over a block of code, disassembling one instruction at a time linearly, without deviating.

Anti-disassembly IV

```
char buffer[BUF_SIZE];
int position = 0;

while (position < BUF_SIZE) {
   x86_insn_t insn;
   int size = x86_disasm(buf, BUF_SIZE, 0, position, &insn);

   if (size != 0) {
      char disassembly_line[1024];
        x86_format_insn(&insn, disassembly_line, 1024, intel_syntax);
        printf("%s\n", disassembly_line);
      ❶position += size;
   } else {
        /* invalid/unrecognized instruction */
      ❷position++;
      }
}
x86_cleanup();
```

Anti-disassembly V

```
        jmp     ds:off_401050[eax*4] ; switch jump

        ; switch cases omitted ...

        xor     eax, eax
        pop     esi
        retn
; ---------------------------------------------------------------------------
off_401050 ❶dd offset loc_401020    ; DATA XREF: _main+19r
            dd offset loc_401027    ; jump table for switch statement
            dd offset loc_40102E
            dd offset loc_401035
```

```
and [eax],dl
inc eax
add [edi],ah
adc [eax+0x0],al
adc cs:[eax+0x0],al
xor eax,0x4010
```

Anti-disassembly VI

Flow-Oriented Disassembly

- The key difference between flow-oriented and linear disassembly is that the disassembler doesn't blindly iterate over a buffer assuming the data is nothing but instructions packed neatly together.
- Instead, it examines each instruction and builds a list of locations to disassemble.

Anti-disassembly VII

```
              test    eax, eax
            ❶jz      short loc_1A
            ❷push    Failed_string
            ❸call    printf
            ❹jmp     short loc_1D
; ---------------------------------------
Failed_string:  db 'Failed',0
; ---------------------------------------
loc_1A: ❺
              xor     eax, eax
loc_1D:
              retn
```

- When the floworiented disassembler reaches the conditional branch instruction jz at (1), it notes that at some point in the future it needs to disassemble the location loc_1A at (5)
- Because **this is only a conditional branch**, the instruction at (2) is also a possibility in execution, so the disassembler will disassemble this as well.
- The lines at (2) and (3) are responsible for printing the string *Failed* to the screen.
- Following this is a jmp instruction at (4). The flow-oriented disassembler will add the target of this, *loc_1D*, to the list of places to disassemble in the future.
-

Anti-disassembly VIII

IDA Pro

If IDA Pro produces inaccurate results, you can manually switch bytes from data to instructions or instructions to data by using the C or D keys on the keyboard, as follows:

- Pressing the C key turns the cursor location into code.
- Pressing the D key turns the cursor location into data.

```
E8 06 00 00 00        call    near ptr loc_4011CA+1
68 65 6C 6C 6F       ❶push    6F6C6C65h

                      loc_4011CA:
00 58 C3              add     [eax-3Dh], bl
```

```
E8 06 00 00 00                   call    loc_4011CB
68 65 6C 6C 6F 00   aHello       db 'hello',0
                                 loc_4011CB:
58                               pop     eax
C3                               retn
```

Anti-disassembly IX

Anti-Disassembly Techniques

- Jump Instructions with the Same Target
- A Jump Instruction with a Constant Condition
- Impossible Disassembly
- NOP-ing Out Instructions with IDA Pro
- The Function Pointer Problem
- Adding Missing Code Cross-References in IDA Pro
- Return Pointer Abuse
- Misusing Structured Exception Handlers
- Thwarting Stack-Frame Analysis

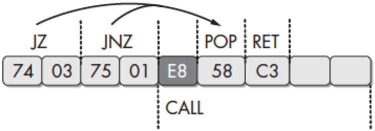
Jump Instructions with the Same Target I

Jump Instructions with the Same Target

- Back-to-back conditional jump instructions that both point to the same target.

```
74 03                jz      short near ptr loc_4011C4+1
75 01                jnz     short near ptr loc_4011C4+1
                     loc_4011C4:                 ; CODE XREF: sub_4011C0
                                                 ; ❷sub_4011C0+2j
E8 58 C3 90 90      ❶call    near ptr 90D0D521h
```

```
74 03                jz      short near ptr loc_4011C5
75 01                jnz     short near ptr loc_4011C5
        ; ---------------------------------------------------------------
E8                   db 0E8h
        ; ---------------------------------------------------------------
                     loc_4011C5:                 ; CODE XREF: sub_4011C0
                                                 ; sub_4011C0+2j
58                   pop     eax
C3                   retn
```



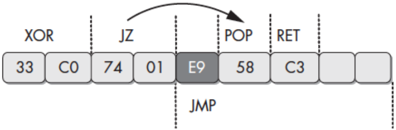
A Jump Instruction with a Constant Condition

Fixed Conditional Branching

- A conditional branching is used to branch unconditionally.

```
33 C0                 xor     eax, eax
74 01                 jz      short near ptr loc_4011C4+1
        loc_4011C4:                              ; CODE XREF: 004011C2j
                                                 ; DATA XREF: .rdata:004020ACo
E9 58 C3 68 94        jmp     near ptr 94A8D521h
```

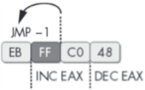
```
33 C0                 xor     eax, eax
74 01                 jz      short near ptr loc_4011C5
        ; ------------------------------------------------------------------
E9                    db 0E9h
        ; ------------------------------------------------------------------
        loc_4011C5:                              ; CODE XREF: 004011C2j
                                                 ; DATA XREF: .rdata:004020ACo
58                    pop     eax
C3                    retn
```

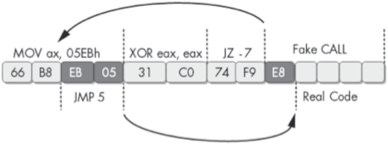


Impossible Disassembly I

Impossible Disassembly

- The simple anti-disassembly techniques, use a data byte placed strategically after a conditional jump instruction,
- We'll call this a *rogue byte* because it is not part of the program and is only in the code to throw off the disassembler





Impossible Disassembly II

```
66 B8 EB 05              mov     ax, 5EBh
31 C0                    xor     eax, eax
74 F9                    jz      short near ptr sub_4011C0+1
                loc_4011C8:
E8 58 C3 90 90           call    near ptr 98A8D525h
```

```
66                       byte_4011C0     db 66h
B8                                       db 0B8h
EB                                       db 0EBh
05                                       db    5
                ; -----------------------------------------------------
31 C0                                    xor     eax, eax
                ; -----------------------------------------------------
74                                       db 74h
F9                                       db 0F9h
E8                                       db 0E8h
                ; -----------------------------------------------------
58                                       pop     eax
C3                                       retn
```

The Function Pointer Problem

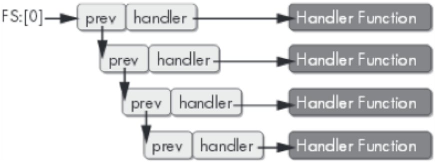
```
004011C0 sub_4011C0      proc near              ; DATA XREF: sub_4011D0+5o
004011C0
004011C0 arg_0           = dword ptr  8
004011C0
004011C0                 push    ebp
004011C1                 mov     ebp, esp
004011C3                 mov     eax, [ebp+arg_0]
004011C6                 shl     eax, 2
004011C9                 pop     ebp
004011CA                 retn
004011CA sub_4011C0      endp

004011D0 sub_4011D0      proc near              ; CODE XREF: _main+19p
004011D0                                        ; sub_401040+8Bp
004011D0
004011D0 var_4           = dword ptr -4
004011D0 arg_0           = dword ptr  8
004011D0
004011D0                 push    ebp
004011D1                 mov     ebp, esp
004011D3                 push    ecx
004011D4                 push    esi
004011D5                 mov     ❶[ebp+var_4], offset sub_4011C0
004011DC                 push    2Ah
004011DE                 call    ❷[ebp+var_4]
004011E1                 add     esp, 4
004011E4                 mov     esi, eax
004011E6                 mov     eax, [ebp+arg_0]
004011E9                 push    eax
004011EA                 call    ❸[ebp+var_4]
004011ED                 add     esp, 4
004011F0                 lea     eax, [esi+eax+1]
004011F4                 pop     esi
004011F5                 mov     esp, ebp
004011F7                 pop     ebp
004011F8                 retn
004011F8 sub_4011D0      endp
```

Return Pointer Abuse

```
004011C0 sub_4011C0      proc near               ; CODE XREF: _main+19p
004011C0                                         ; sub_401040+8Bp
004011C0
004011C0 var_4           = byte ptr -4
004011C0
004011C0                 call    $+5
004011C5                 add     [esp+4+var_4], 5
004011C9                 retn
004011C9 sub_4011C0      endp ; sp-analysis failed
004011C9
004011CA ; ------------------------------------------------------------
004011CA                 push    ebp
004011CB                 mov     ebp, esp
004011CD                 mov     eax, [ebp+8]
004011D0                 imul    eax, 2Ah
004011D3                 mov     esp, ebp
004011D5                 pop     ebp
004011D6                 retn
```

Misusing Structured Exception Handlers




```
struct _EXCEPTION_REGISTRATION {
   DWORD prev;
   DWORD handler;
};
```

```
00401050                ❷mov     eax, (offset loc_40106B+1)
00401055                 add     eax, 14h
00401058                 push    eax
00401059                 push    large dword ptr fs:0 ; dwMilliseconds
00401060                 mov     large fs:0, esp
00401067                 xor     ecx, ecx
00401069                ❸div     ecx
0040106B
0040106B loc_40106B:                             ; DATA XREF: sub_4010500
0040106B                 call    near ptr Sleep
00401070                 retn
00401070 sub_401050      endp ; sp-analysis failed
00401070
00401070 ; ---------------------------------------------------------------
00401071                 align 10h
00401080                ❶dd 824648Bh, 0A164h, 8B0000h, 0A364008Bh, 0
00401094                 dd 6808C483h
00401098                 dd offset aMysteryCode  ; "Mystery Code"
0040109C                 dd 2DE8h, 4C48300h, 3 dup(0CCCCCCCCh)
```

Thwarting Stack-Frame Analysis

```
00401543      sub_401543      proc near           ; CODE XREF: sub_4012D0+3Cp
00401543                                          ; sub_401328+9Bp
00401543
00401543      arg_F4          = dword ptr  0F8h
00401543      arg_F8          = dword ptr  0FCh
00401543
00401543 000                  sub     esp, 8
00401546 008                  sub     esp, 4
00401549 00C                  cmp     esp, 1000h
0040154F 00C                  jl      short loc_401556
00401551 00C                  add     esp, 4
00401554 008                  jmp     short loc_40155C
00401556      ; ---------------------------------------------------------------
00401556
00401556      loc_401556:                        ; CODE XREF: sub_401543+Cj
00401556 00C                  add     esp, 104h
0040155C
0040155C      loc_40155C:                        ; CODE XREF: sub_401543+11j
0040155C -F8 (1)              mov     [esp-0F8h+arg_F8], 1E61h
00401564 -F8                  lea     eax, [esp-0F8h+arg_F8]
00401568 -F8                  mov     [esp-0F8h+arg_F4], eax
0040156B -F8                  mov     edx, [esp-0F8h+arg_F4]
0040156E -F8                  mov     eax, [esp-0F8h+arg_F8]
```