LA-UR-18-25476
Approved for public release; distribution is unlimited.

Title: Anti-Reverse-Engineering: Malware Analysis Day 6

Author(s): Pearce, Lauren

Intended for: Presentation for two week course on malware analysis

Issued: 2018-06-21

Anti-Reverse-Engineering

Malware Analysis Day 6

laurenp@lanl.gov

Primary Goal of Anti-RE Techniques?

- Delay or prevent the analysis of malicious code by:
  - Increasing the skill level required by the malware analyst
  - Increasing the time it takes a malware analyst to extract indicators

Anti-Disassembly

How does a Disassembler Work?

- Let’s think about the problem of disassembly for a moment
  - Sequences of bytes can have multiple possible representations in assembly. It’s up to the disassembler to determine the correct one. Not a trivial problem.
- Anti-Disassembly techniques take advantage of the assumptions that disassemblers make and can cause even advanced disassemblers to present inaccurate code.

Linear Disassembly

- Iterate over a block of code disassembling one instruction at a time without deviating.
- Uses the size of the disassembled instruction to decide which byte to disassemble next – no regard to flow-control instructions
- Simple and easy to understand, BUT
  - Error prone, even in non malicious binaries
  - Will disassemble too much code – code section of binaries inevitable contains data that isn’t code.
  - Easy to defeat because they cannot distinguish code from data

Linear Assembly: Example

```
          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
```

Flow-Oriented Disassembly

- Doesn't blindly iterate over the buffer assuming everything is code. Examines each instruction and creates a list of locations to disassemble.

```
                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
```

Flow-Oriented Disassembly: Conditional Branches

- The flow oriented disassembler has a choice of two places to disassemble – the true or the false branch.
- In compiler generated code, there would be no difference in the resultant code if true or false was processed first – not so much for handwritten code.
- When there is a conflict, most compilers will take the false branch of the jump. Similarly, most compilers will take the bytes after a call rather than the call location.

Defeating Flow Oriented Disassemblers

Call Pop

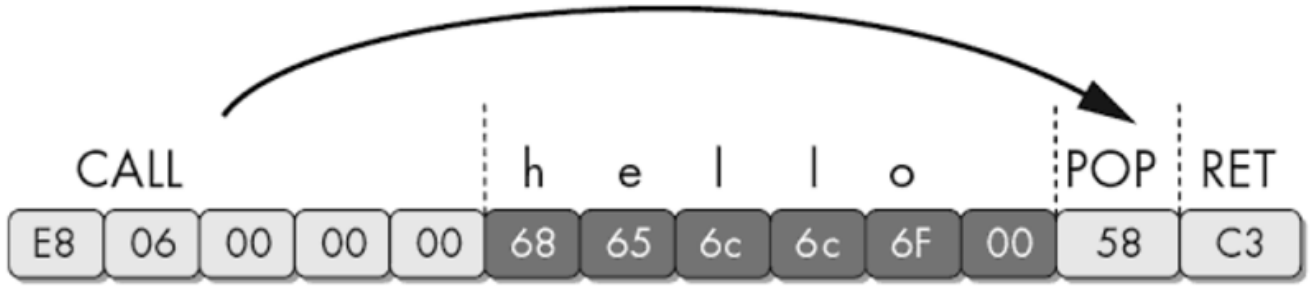


Figure 15-1: call instruction followed by a string

```
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
```

```
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
```

Practical Malware Analysis Chapter 15, Pages 333-334

Fixing Ida's Mistakes

- Sometimes you'll have to do some manual cleanup to tell Ida how to disassemble some instructions. You'll use the shortcuts C and D
  - Place your cursor over the instruction that was translated incorrectly
  - Press C to turn it into code
  - Press D to turn it into data

```
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
```

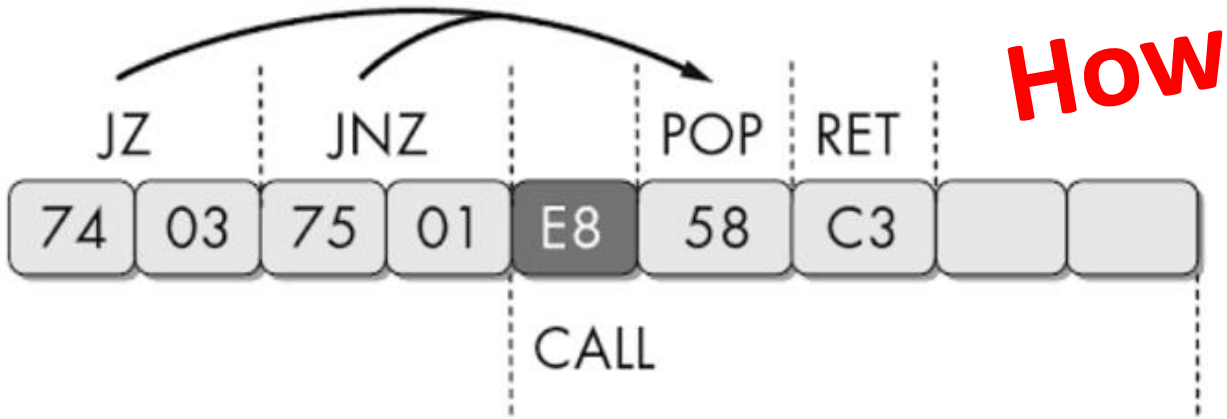
Practical Malware Analysis Chapter 15, Page 334

Jump Instructions with the Same Target

- Two back to back conditional jumps that both point to the same place and for which one or the other must be taken.
  - jz loc_512 followed by jnz loc_512
  - Technically it's an unconditional jump
- Why is this a problem?
  - Despite it being an unconditional jump, the disassembler only looks at a single instruction at a time and thus doesn't recognize it as such. It continues disassembling the false branch even though it can never be executed.

Jump Instructions with 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
```



How do we fix it?

Figure 15-2: A jz instruction followed by a jnz instruction

Jump Instructions with the Same Target

```
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
```

Jump with a Constant Condition

- A conditional jump that will always be taken.
  - xor eax eax followed by jz
- Why could this create a problem for a disassembler?
  - Even though this is essentially an unconditional jump, the disassembler can’t see that. The disassembler views this as a conditional jump and so continues disassembling the false branch even though it can never be executed. This leads to conflicting instructions, and the disassembler defaults to the “false” branch – the one that’s technically impossible.

Jump with a Constant Condition

```
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
```

How do we fix it?

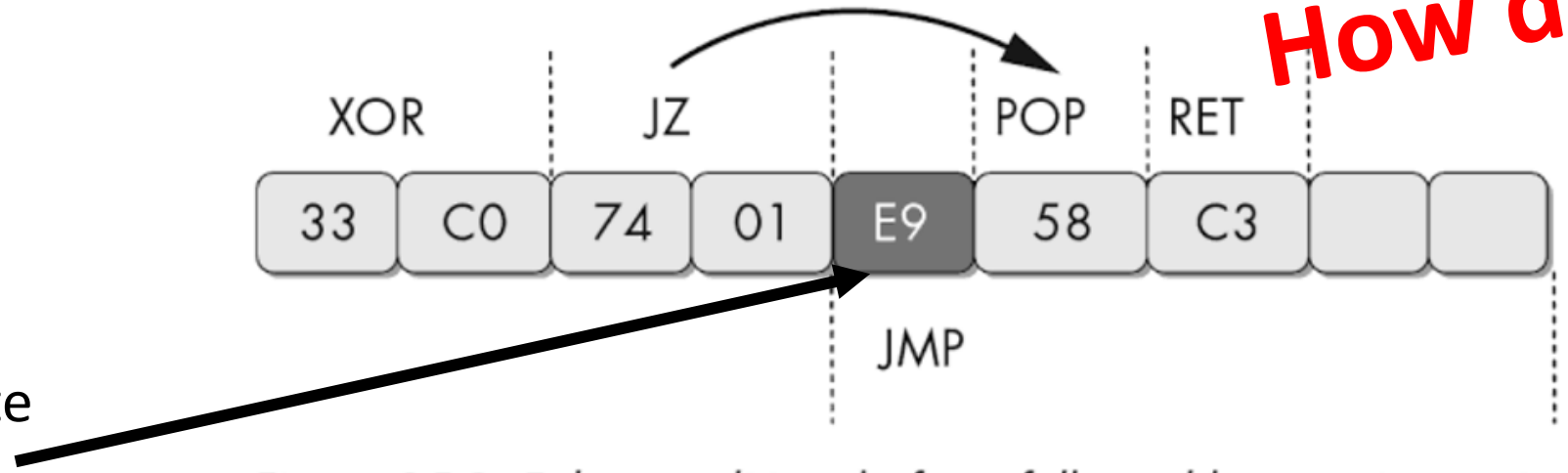


Figure 15-3: False conditional of xor followed by a jz instruction

Demo

Practical Malware Analysis Lab 15-1

Impossible Disassembly

- With the last issues, we could use Ida to fix the code and make it look how it should. There are some anti-disassembly techniques that prevent us from making the code look as it should.
- In the previous examples, the rogue byte could be completely disregarded – it was extraneous and there to get in our way.
- But what if that byte was necessary? A single byte that is part of two instructions,.
  - Possible on the processor level, but there's no disassembler that knows how to identify and represent this.

Impossible Disassembly

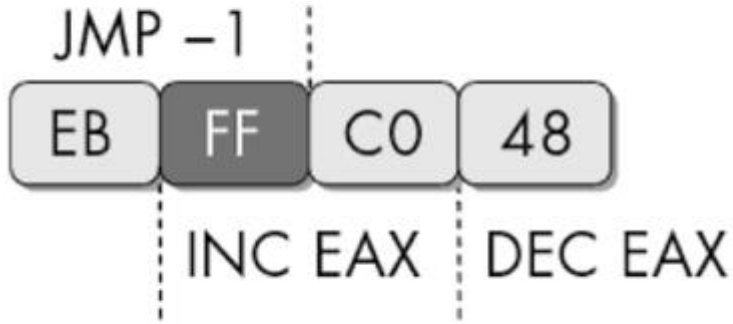


Figure 15-4: Inward-pointing `jmp` instruction

Practical Malware Analysis Chapter 15, Figure 15-4

Impossible Disassembly

How do we fix it?

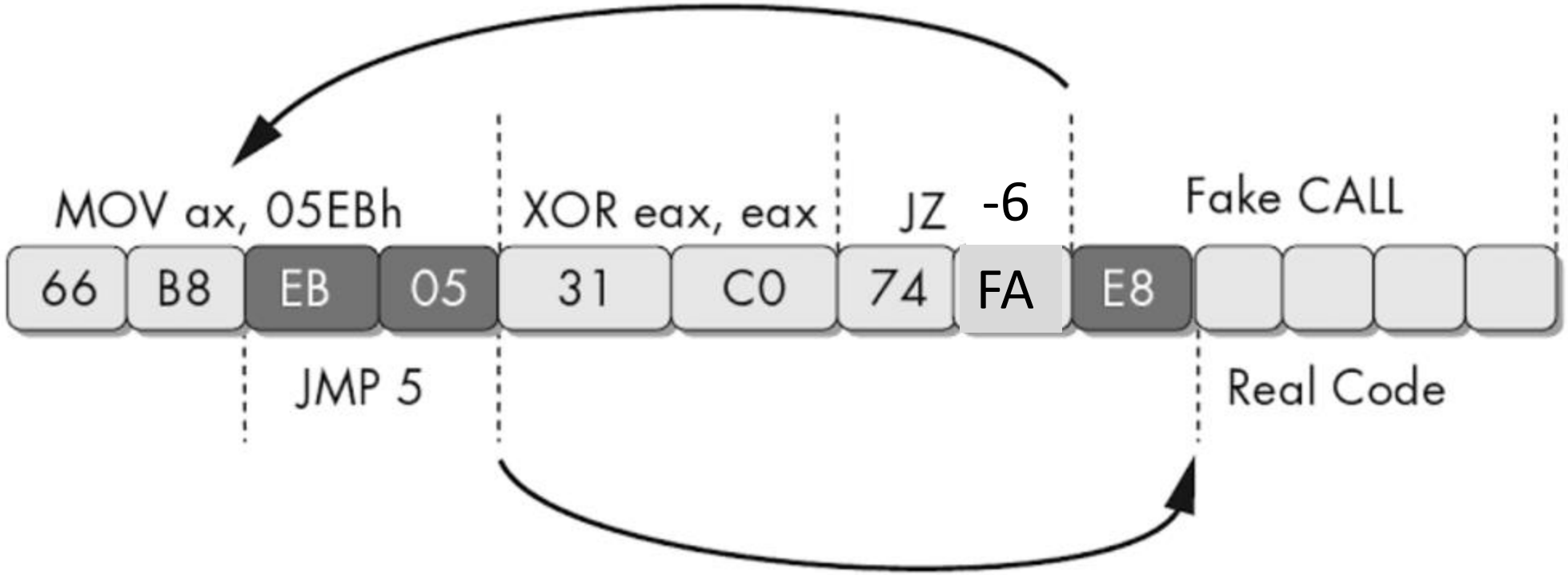


Figure 15-5: Multilevel inward-jumping sequence

Practical Malware Analysis Chapter 15, Figure 15-5

Obscuring Control Flow

Function Pointers

```
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
```

Practical Malware Analysis Chapter 15, Page 341

Return Pointer Abuse

- We've looked at abuses of call and jmp, but ret also has an impact on the control flow of a program.
- What two things happen when the call instruction is issued?
  - Push
  - Jump
- What two things happen when the ret instruction is issued?
  - Pop
  - Jump
- Ret is logically used as the inverse of call, but nothing prevents it from being used wherever you want to use it.

```
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
```

NOP

Alt+P –set function end to 4011D6

Practical Malware Analysis Chapter 15, Page 342-343

SEH Abuse

- Remember when I said Structured Exception Handling would come up again later?
- SEH Chain – List of functions that are designed to handle exceptions in a given thread. Each function in the chain either handles the exception, or passes it to the next link.
  - In this context, what is an unhandled exception?

Traversing Structures to find the SEH Chain

- FS Segment Register → Thread Environment Block (TEB) → Thread Information Block (TIB)
  - The first element of the TIB is a pointer to the SEH Chain.
- The SEH chain itself is a linked list of 8 byte data structures, each of which is called an EXCEPTION_REGISTRATION record.
  - Conceptually, operates as a stack. The first record entered is the last record accessed.

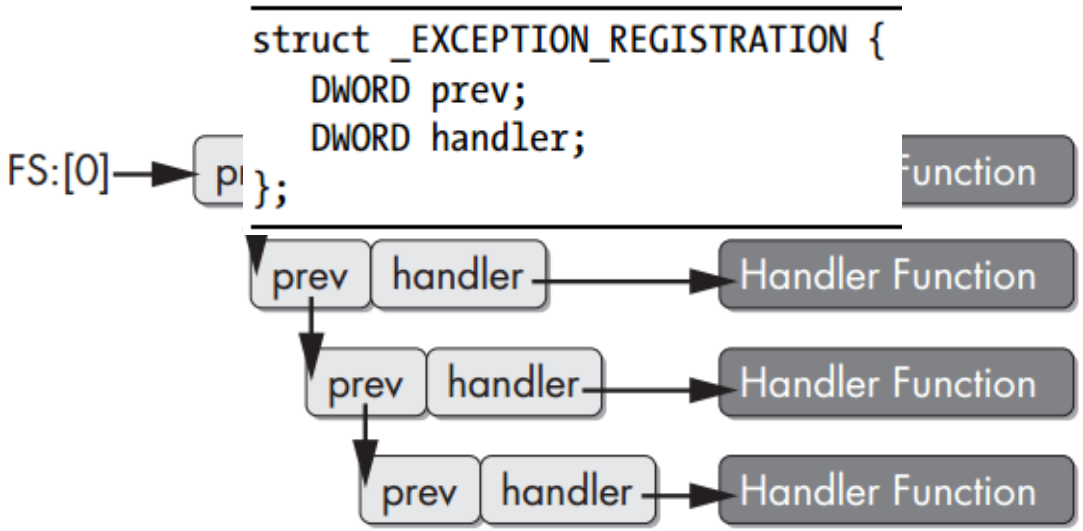


Figure 15-6: Structured Exception Handling (SEH) chain

Adding a Record to the SEH Chain

- If you can add a record to the SEH chain, you can then manipulate control flow in a way that is difficult for an analyst to follow.
- The structure for each record contains 2 DWORDs – easy to do with 2 pushes.
  - Stack grows UP – first push is the pointer to the handler function, second push is pointer to the next record.
- Pointer to the handler function is easy, but where do we find the pointer to the next record?
  - The next record is the one that is currently on top. It can be accessed “fs:[0]”

```
push ExceptionHandler
push fs:[0]
mov fs:[0], esp
```

Practical Malware Analysis Chapter 15, Page 345

```
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_401050o
0040106B                  call    near ptr Sleep
00401070                  retn
00401070 sub_401050       endp ; sp-analysis failed
00401070
00401070 ; ---------------------------------------------------------------------
00401071                  align 10h
00401080                dd 824648Bh, 0A164h, 8B0000h, 0A364008Bh, 0
00401094                  dd 6808C483h
00401098                  dd offset aMysteryCode  ; "Mystery Code"
0040109C                  dd 2DE8h, 4C48300h, 3 dup(0CCCCCCCCh)
```

Breaking Stack-Frame Analysis

- Disassemblers like Ida can analyze the instructions contained in a function and deduce the construction of the stack-frame. This allows them to label parameters, local variables, etc.
  - This is not an exact science and can be broken by a determined malware author ☹

```
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 -F8❶                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]
00401572 -F8                 inc     eax
00401573 -F8                 mov     [edx], eax
00401575 -F8                 mov     eax, [esp-0F8h+arg_F4]
00401578 -F8                 mov     eax, [eax]
0040157A -F8                 add     esp, 8
0040157D -100                retn
0040157D     sub_401543      endp ; sp-analysis failed
```

Listing 15-1: A function that defeats stack-frame analysis

But Wait! There's MORE!

- We've talked about a few specific techniques, but there are many more.
- These techniques have demonstrated how a decompiler can be tricked, and all anti-reverse engineering techniques work by tricking the decompiler.
- Hopefully you will now be able to spot the trickery even when the actual technique is different.

Demo

Practical Malware Analysis Lab 15-2, How to NOP a byte

Anti-Debugging

What is Anti-Debugging?

- Techniques to
  - A) Detect that a program is being debugged
  - B) Fail or otherwise alter behavior if being debugged

- We focus on circumventing detection, because that can save us from part B. BUT, it’s important to keep in mind that a program that detects a debugger is not necessarily anti-debugging or malicious.

Broad Methods

1. Use the Windows API to detect the debugger
2. Manually checking OS data structures
3. Identifying Debugger Behavior
4. Interfering with the debugger functionality

Windows API

- **IsDebuggerPresent** – The most obvious of them all. Checks the PEB for the field IsDebugged.
- **CheckRemoteDebuggerPresent** – Behaves just like IsDebuggerPresent, but with the added capability of checking if a different process is being debugged.
- **NtQueryInformationProcess** – Native API function that retrieves information about a process. First parameter is a handle to the target process, the second is a value specifying what information it wants returned. Value 0x7 will return debugging information.
- **OutputDebugString** followed by GetLastError – OutputDebugString is a call that only works in the presence of a debugger. If called when not being debugged, it will error out.

Manually Checking Structures

- The API calls are just a little obvious. Malware authors may try to be a little more stealthy by manually checking the same structures the API calls rely on.

BeingDebugged Flag

```
typedef struct _PEB {
  BYTE                          Reserved1[2];
  BYTE                          BeingDebugged;
  BYTE                          Reserved2[1];
  PVOID                         Reserved3[2];
  PPEB_LDR_DATA                 Ldr;
  PRTL_USER_PROCESS_PARAMETERS  ProcessParameters;
  BYTE                          Reserved4[104];
```

Table 16-1: Manually Checking the BeingDebugged Flag

mov method	push/pop method
`mov eax, dword ptr fs:[30h]`	`push dword ptr fs:[30h]`

Practical Malware Analysis Chapter 16, Table 16-1

ProcessHeap Flag

- There is an undocumented location within Reserved4 called ProcessHeap.
- Located at 0x18 in the PEB structure.
- Contains field to tell the kernel whether the heap was created in a debugger, these are called ForceFlags and Flags fields.

```
typedef struct _PEB {
  BYTE                          Reserved1[2];
  BYTE                          BeingDebugged;
  BYTE                          Reserved2[1];
  PVOID                         Reserved3[2];
  PPEB_LDR_DATA                 Ldr;
  PRTL_USER_PROCESS_PARAMETERS  ProcessParameters;
  BYTE                          Reserved4[104];
  PVOID                         Reserved5[52];
  PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;
  BYTE                          Reserved6[128];
  PVOID                         Reserved7[1];
  ULONG                         SessionId;
} PEB, *PPEB;
```

MSDN Documentation

ProcessHeap Flag

- XP – offset for ForceFlags is 0x10, offset for Flags is 0x0C.
- Win 7 32 bit offset for ForceFlags is 0x44, offset for Flags is 0x40
- Flags is almost always equal to ForceFlags, but it is stored ORed with 2.
  - No, I don't know why…

```
mov eax, large fs:30h
mov eax, dword ptr [eax+18h]
cmp dword ptr ds:[eax+10h], 0
jne DebuggerDetected
```

Listing 16-3: Manual ProcessHeap flag check

Practical Malware Analysis Chapter 16, Listing 16-3

NTGlobalFlag

- The heap is built differently when a process is run under a debugger – there's a location that stores the information that the system uses to determine how to create heap structures.
- This information is stored in the PEB at offset 0x68 as a series of flags. The following flags are set when a process is created by a debugger:
  - FLG_HEAP_ENABLE_TAIL_CHECK
  - FLG_HEAP_ENABLE_FREE_CHECK
  - FLG_HEAP_VALIDATE_PARAMETERS
- When those flags are set, the value of the byte is 0x70

```
mov eax, large fs:30h
cmp dword ptr ds:[eax+68h], 70h
jz DebuggerDetected
```

Listing 16-4: NTGlobalFlag check

Practical Malware Analysis Chapter 16, Table 16-4

Demo

Practical Malware Analysis Lab 16-1

Via Checking for System Residue

- Debuggers leave evidence of themselves on the system – malware can simply try to determine if Olly or Windbg exist on the system, and consider that enough to bail.
  - Traverse the file system, traverse the process list, look at registry keys, etc.

Via Looking for Debugger Behavior

- Breakpoints and single stepping modify the code – anti debugging techniques can look for this modification.
- 3 Methods:
  - INT Scanning
  - Checksum Checks
  - Timing Checks

INT Scanning

- Anyone remember how breakpoints work?
  - INT 3 is a software interrupt. To implement a breakpoint, a debugger temporarily replaces an instruction in a running program with INT 3. When the INT 3 is hit, the debug exception handler is called.
  - The opcode for INT 3 is 0xCC
  - In reality, any INT <immediate> instruction can set an interrupt. That makes a 2 byte opcode 0xCD <value>. Used less frequently.
- A process can scan its own code for an INT 3 modification, kill itself if it finds it.

INT Scanning

```
call $+5
pop edi
sub edi, 5
mov ecx, 400h
mov eax, 0CCh
repne scasb
jz DebuggerDetected
```

Listing 16-6: Scanning code for breakpoints

Practical Malware Analysis Chapter 16, Listing 16-6

Code Checksums

- Calculates a checksum on a section of code, compares it against its known good value.
- Look for malware to iterate over its own instructions, then compare some value to a constant value.

Timing Checks

- Take a time stamp, do some things (math is good), take another timestamp. If outside a certain threshold, you’re being debugged.
- Take a timestamp, raise an exception, take another timestamp – if outside a certain threshold, you’re being debugged.

Timing Checks - rdtsc

- rdtsc instruction – gives the ticks since the last reboot. 64 bit value placed in EDX:EAX.
- How is the code to the right using rdtsc?

```
rdtsc
xor ecx, ecx
add ecx, eax
rdtsc
sub eax, ecx
cmp eax, 0xFFF ❶
jb NoDebuggerDetected
rdtsc
push eax ❷
ret
```

Listing 16-7: The rdtsc timing technique

Practical Malware Analysis Chapter 16, Table 16-7

QueryPerformanceCounter and GetTickCount

- They do roughly the same thing and are used in the same way as rdtsc.
- How do you deal with these?
  - Place a breakpoint after the check, run through them quickly
    - Doesn’t always work – why?
  - Patch the compare or the jump after the compare (my favorite)

Interfering with Debugger Functionality

- TLS Callbacks
- Exceptions
- Interrupt Insertion

TLS Callbacks

- When debugging in Ida, I often use the checkbox “Break at process entry point”. Olly does this by default. How does the debugger know where the entry point is?
  - Entry point is defined in the PE Header
- TLS – A windows storage class that allows each thread to maintain a different value for a single variable.
- When a program uses TLS, you’ll typically see a .tls section in the PE header.
- TLS callback functions for initializing and terminating TLS objects run BEFORE the start of a program.

TLS Callbacks

- In Ida, after the program has loaded press ctrl+E – you will see all the entry points to a program, including the TLS callback functions.
- In Olly, Options → Debugging Options → Events, set “system breakpoint” as the place of the first pause.
- I’ve never seen this outside of class/bookwork, but good to be aware of.

Exceptions

- Remember how breakpoints work?
  - Interrupt → Exception → Control given to debugger?
- By default, debuggers are at the top of the list for exceptions. When a process raises an exception, it doesn't go directly to its own exception handler, it goes to the debugger for handling.
- When working with malware, you can change this default behavior and pass exceptions to the program.
  - In Olly, Options → Debugging Options → Exceptions

Inserting Interrupts – INT 3, INT 2D, 0xF1

- Sprinkle 0xCC opcodes into the code, drive the analyst crazy
- Remember there's a 2 byte version of this, 0xCD03
  - Wreaks havoc in WinDbg, Olly is fine, not sure about Ida
- INT 2D – breakpoint for the kernel debugger
- 0xF1 – This is the In-Circuit Emulator(ICE) breakpoint.
  - An undocumented Intel instruction.
  - Don't single step over this

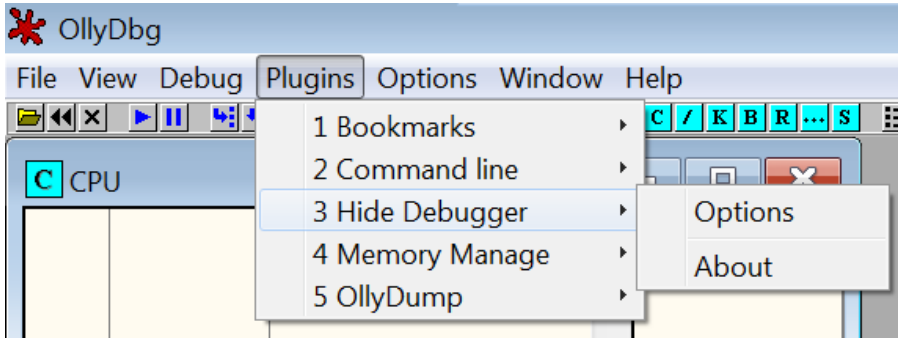
```
push offset continue
push dword fs:[0]
mov fs:[0], esp
int 3
//being debugged
continue:
//not being debugged
```

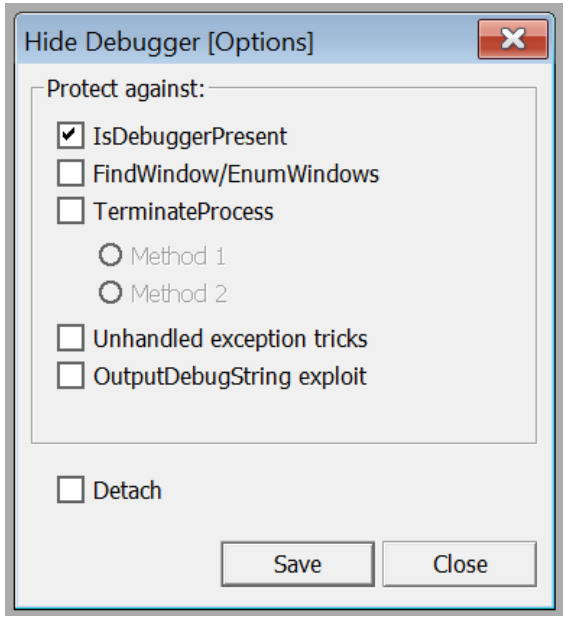
Listing 16-9: INT 3 technique

Practical Malware Analysis Chapter 16, Table 16-9

Olly Plugin: HideDebugger

- Available through OpenRCE.org
- Covers the basics – Not exceptionally useful with advanced actors.





Demo

Practical Malware Analysis Lab 16-3

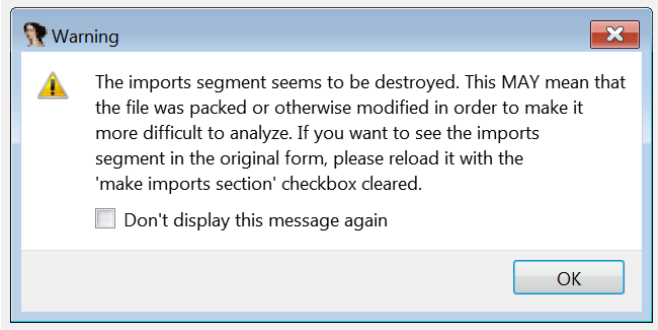
Packers and Unpacking

Review

- What does a packer do?
- Why are packers used?
- Is packing an indication of malware?

Review

- What are some indications of packing?



IDA View-A | Hex View-1

Address	Length	Type	String
.text:0040F118	0000000D	C	KERNEL32.DLL
.text:0040F125	0000000D	C	ADVAPI32.DLL
.text:0040F132	0000000D	C	NETAPI32.DLL

IDA View-A | Hex View-1 | Structures

Address	Ordinal	Name	Library
0040F09C		LoadLibraryA	KERNEL32
0040F0A0		GetProcAddress	KERNEL32
0040F0A4		VirtualProtect	KERNEL32
0040F0A8		VirtualAlloc	KERNEL32
0040F0AC		VirtualFree	KERNEL32
0040F0B0		ExitProcess	KERNEL32
0040F0B8		LookupPrivilegeValueA	ADVAPI32
0040F0C0		Netbios	NETAPI32

Name	Virtual Size	Virtual A...	Raw Size
Byte[8]	Dword	Dword	Dword
.code	0000E000	00001000	00000000
.text	00006000	0000F000	00005686
.rsrc	000009C6	00015000	00000000

Anatomy of a Packer – Unpacking Stub

- Performs 3 steps:
  1. Unpacks the original program into memory
  2. Resolves all of the imports of that original program
  3. Transfers execution to the original entry point (OEP)

- Note that the entry point to a packed program is the entry to the STUB, not the "original entry point" of the program. Recovering the OEP is a major step in unpacking packed samples.

Anatomy of a Packer – Loading the Executable

- The PE Header of a packed executable dictates the amount of space that each section of the soon-to-be unpacked program will need.
- The loader loads the unpacking stub with it’s seemingly excessive amount of space. The unpacking stub unpacks the original executable into the correct sections of the allocated space.
- What indicators of packing result from this methodology?

Anatomy of a Packer – Resolving Imports

- How are imports usually resolved? Why doesn't that work for this situation?
  - Resolving imports is the job of the windows loader. The loader resolved the imports for the packed version of the program, but it can't see the imports for the packed pieces to know what to do.

Anatomy of a Packer – Resolving Imports

- Method 1 – Rebuild the IAT.
  - Most common, mentioned on day 1
  - The stub imports LoadLibrary and GetProcAddress
  - After unpacking original program, reads import information. Then calls LoadLibrary for each library and loads the necessary DLLs into memory. Lastly, calls GetProcAddress for each function

Anatomy of a Packer – Resolving Imports

- Method 2 – Preserve the Original IAT
  - Simplest method – do not have to rebuild anything, the loader handles everything.
  - Reveals all of the imports to static analysis
  - Poor compression
  - I’ve never seen this in the wild

Anatomy of a Packer – Resolving Imports

- Method 3 – Have the loader load the libraries, find the function addresses manually.
  - Stub imports one function from each required DLL, then rebuilds the table by calling GetProcAddress for the required functions.

- What are the pros and cons of this method?
  - Simpler to write than method 1, harder than method 2
  - Still reveals all of the libraries imported, but reveals less functionality than method 2.

Anatomy of a Packer – Resolving Imports

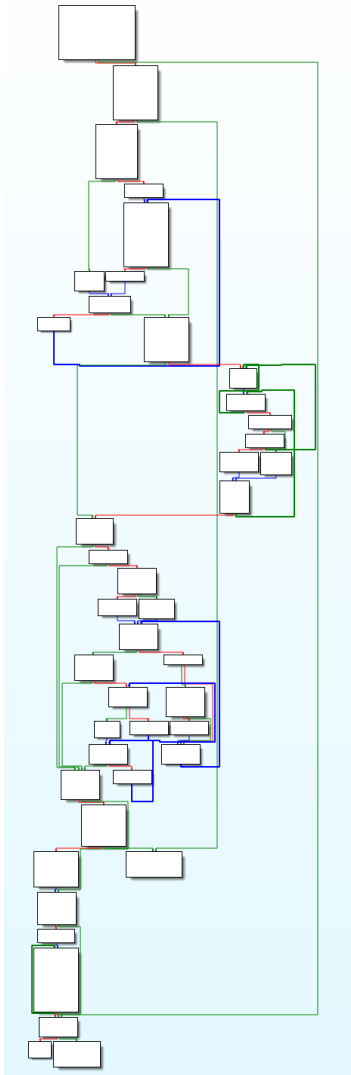
- Method 4 – Remove ALL imports
  - Zero imports, including LoadLibrary and GetProcAddress.
  - Stub must find those similarly to how shellcode does. If you're interested, reference chapter 19.
    - Be forewarned, it's pretty convoluted.

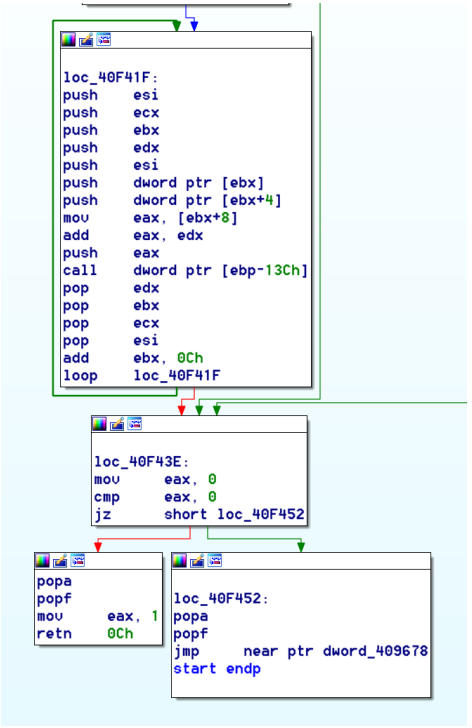
Anatomy of a Packer – The Tail Jump

- Once the unpacking stub has done its work, it transfers control to the entry point of the now unpacked program. This is the original entry point, or OEP.
- This called a tail jump, presumably because of how it looks in graph view in Ida.
- Typically this is done with a simple jump instruction, but malware authors may try to obscure it using some other instruction that can transfer control, such as ret or call.
- May also call an OS function such as NtContinue to transfer control

Anatomy of a Packer – The Tail Jump

- If we can identify the transfer of control, we can identify the OEP. If we can identify the OEP, we're significantly closer to an unpacked executable.





Unpacking - Automated

- Best case scenario? Someone’s already written an unpacker. If so, use it! But use it with caution – you’re running somebody else's code. I suggest a VM.
  - Most common example of this is UPX
- Second best case? Your program unpacks in an automated dynamic unpacker.
  - Runs the program to unpack it. Depends on identifying the OEP.
  - If can’t identify OEP, unpacking fails. This occurs so frequently I don’t use them in my regular analysis flow.
  - The word DYNAMIC means the code executes. Use a VM.
  - Ex: PackerAttacker

Unpacking - Manual

- Two methods:
  1. Write an unpacker
     a) Study the unpacking stub, write something to do what it does (minus the anti-RE stuff)
     b) Makes the rest of the community happy if you make it public
     c) I've never done this. IMO, this is a research problem. There is no time for it in operations. We will not do this in class.
  2. Manipulate the stub into unpacking the packed material, then dump the unpacked program from memory.
     1. Easier, typically takes significantly less time
     2. A few standard methods that work *most* of the time

Manual Unpacking - Olly

- When it comes to unpacking, or anything having to do with memory, Olly is vastly superior to Ida.
- You will need the OllyDump plugin
- Whenever possible, use HARDWARE BREAKPOINTS when attempting to unpack. Why?

Methods of Manual Unpacking

Tail Jumps

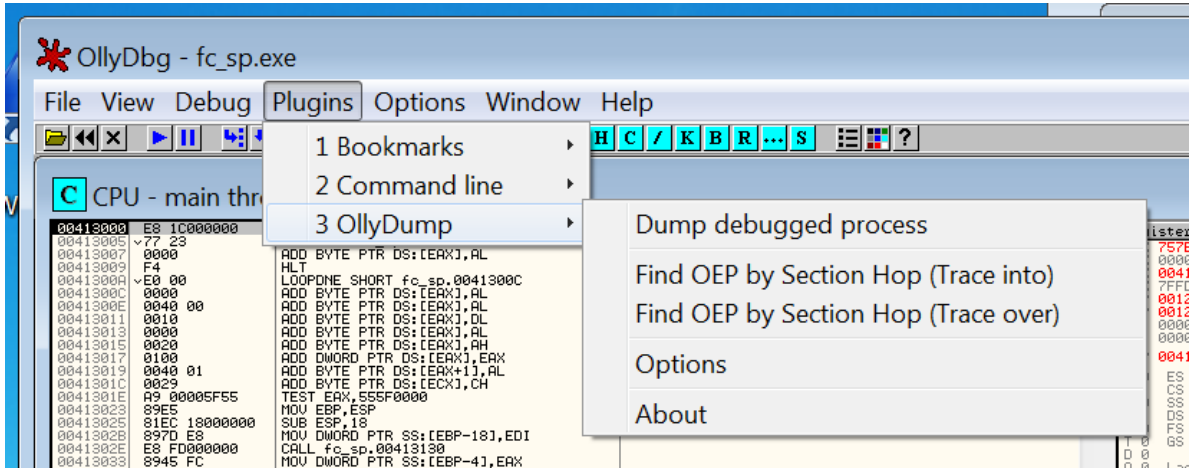
- We talked about Tail Jumps and why they occur in packed executables, but how does that knowledge help us unpack?
  - Identify the OEP, use OllyDump to dump from there.

Demo

Lab 18-1

OllyDump - Find OEP By Section Hop

1. “Find OEP by Section Hop”
2. Place hardware breakpoint at location indicated
3. Run program to breakpoint
4. Use OllyDump to dump Memory



OllyDump - Find OEP By Section Hop

- How does it work?
  - Relies on the fact that section hops (jumping from one section of a PE to another), are rare in executables that are not packed.
  - Single steps through the program, checking each instruction to see if it executes a section hop.
  - When a section hop is identified, you have found the OEP.
  - How might malware be written to cause this not to work?

Demo

Lab 18-2

pusha and popa

- pusha pushes all the general purpose registers onto the stack in a specified order. It is used to save the state of the registers.
- popa pops all the general purpose registers off the stack in the reverse order of pusha. This is used to restore the state of the registers at some point after a pusha.
- Why would a packer use these instructions?
  - The unpacking stub will frequently start with a pusha instruction, then, when the unpacking is finished, issue a popa.

pusha and popa

- To leverage these instructions for unpacking:
  - Put a breakpoint on the pusha instruction
  - Debug
  - Look at EIP at the breakpoint – set a hardware breakpoint to break on access to that location
  - Run the program, you'll break either on or just before the OEP
  - Dump

Assumptions of the Above Methods

- The stub is allowing you the privilege of running in a VM
- The stub is not killing itself when it sees the debugger
- The stub is not placing section hops, tail jumps, and random encoding/decoding algorithms all over the place.

Demo

18-3

Labwork Tonight

- Labs 15-3, 16-2, and 18-4
- Only 2 afternoons left to work on Obfuscated Malware Lab – be sure to give it some attention this afternoon.

Sources/Questions/Comments/Corrections

- As usual, much credit to Andrew Honig and Michael Sikorski's Practical Malware Analysis.
- Note that animations (mostly highlighting on click) are extremely useful when teaching from this slide deck. Email me for slide originals.
- Questions/Comments/Corrections to Lauren Pearce – Laurenp@lanl.gov