Malware Analysis

ANTI REVERSE ENGINEERING TECHNIQUES

COMPILED BY: DHARMESH DAVE | ASST. PROF. | NATIONAL FORENSIC SCIENCES UNIVERSITY

Anti Disassembly

- Anti-disassembly uses **specially crafted code or data** in a program to cause disassembly analysis tools to produce an incorrect program listing.
- This technique is crafted by malware authors manually, with a separate tool in the build and deployment process or interwoven into their malware's source code.
- 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

- These additional layers of protection may exhaust the in-house skill level at many organizations and require expert consultants or large research project levels of effort to reverse-engineer.
- Anti-disassembly is also effective at preventing certain automated analysis techniques.
- 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.
- Anti-disassembly techniques work by taking advantage of the assumptions and limitations of disassemblers.

Anti Disassembly

- Example: disassemblers can only represent each byte of a program as part of one instruction at a time. If the disassembler is tricked into disassembling at the **wrong offset**, a valid instruction could be hidden from view.

Disassembly Algorithms

- Anti-disassembly techniques are born out of inherent **weaknesses in disassembler algorithms**. Any disassembler must make certain assumptions in order to present the code it is disassembling clearly.
- When these assumptions fail, the malware author has an opportunity to fool the malware analyst.
- There are two types of disassembler algorithms: **linear** and **flow-oriented**

Linear Disassembly Algorithms

- The linear-disassembly strategy **iterates over a block of code**, disassembling one instruction at a time linearly, without deviating. This basic strategy is employed by disassembler writing tutorials and is widely used by debugger
- Linear disassembly uses the **size of the disassembled instruction** to determine which byte to **disassemble next**, without regard for flow-control instructions.
- This algorithm will disassemble most code without a problem, but it will introduce occasional errors even in non-malicious binaries.
- The main drawback to this method is that it will disassemble too much code.

Linear Disassembly Algorithms

- In a PE-formatted executable file applying this linear-disassembly algorithm to the .text section containing the code, but the problem is that the code section of nearly all binaries will also contain data that isn't instructions.
- One of the most common types of data items found in a code section is a pointer value, which is used in a table-driven switch idiom.

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

Linear Disassembly Algorithms

- Linear-disassembly algorithms are the easiest to defeat because they are unable to distinguish between code and data.

Flow-Oriented Disassembly Algorithms

- A more advanced category of disassembly algorithms is the flow-oriented disassembler. This is the method used by most commercial disassemblers such as IDA Pro.
- 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.

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

Difference Linear v/s Flow-Oriented

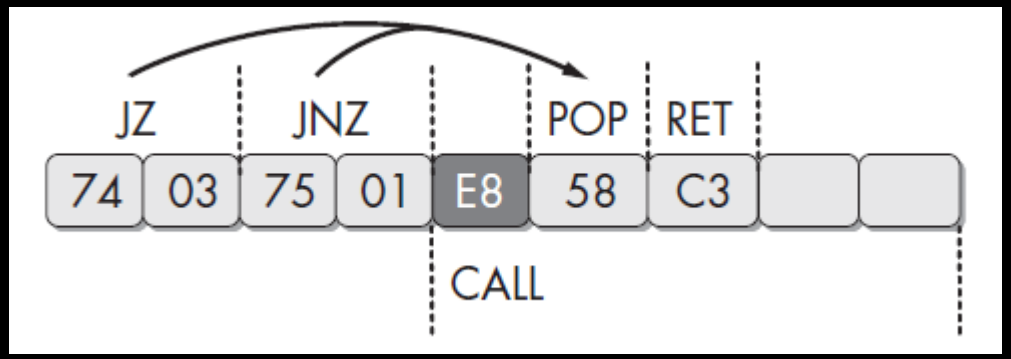
- In linear disassembly, the disassembler has no choice to make about which instructions to disassemble at a given time. Flow-oriented disassemblers make choices and assumptions.

Anti-Disassembly Techniques

- Jump Instructions with the Same Target
- A Jump Instruction with a Constant Condition
- Impossible Disassembly

Jump Instructions with the Same Target

- The most common anti-disassembly technique seen in the wild is two back-to-back conditional jump instructions that both point to the same target.
- Example, if a jz loc_512 is followed by jnz loc_512, the location loc_512 will always be jumped to.
- The combination of jz with jnz is, in effect, an unconditional jmp, but the disassembler doesn't recognize it as such because it only disassembles one instruction at a time.



Jump Instructions with the Same Target

- Both conditional jump instructions actually point 1 byte beyond the 0xE8 byte. When this fragment is viewed with IDA Pro, the code cross-references shown at loc_4011C4 will appear in red, rather than the standard blue, because the actual references point inside the instruction at this location, instead of the beginning of the instruction. As a malware analyst, this is your first indication that anti-disassembly may be employed in the sample you are analyzing.

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

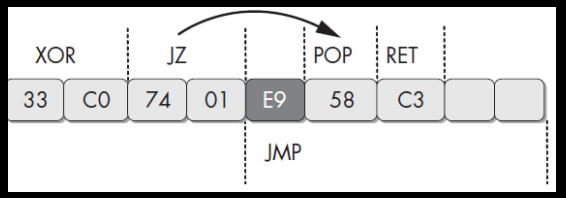
Solution

- 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.

```
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

- Another anti-disassembly technique commonly found in the wild is composed of a single conditional jump instruction placed where the condition will always be the same.
- Notice that this code begins with the instruction xor eax, eax. This instruction will set the EAX register to zero and, as a byproduct, set the zero flag.




```
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
```

Solution

- you can use the D key on the keyboard while your cursor is on a line of code to turn the code into data, and pressing the C key will turn the data into code. Using these two keyboard shortcuts, a malware analyst could fix this fragment and have it show the real path of execution, as follows:

```
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
```

Rogue Byte

- The **simple anti-disassembly** techniques we have discussed use a data byte placed strategically after a conditional jump instruction, with the idea that disassembly starting at this byte will prevent the real instruction that follows from being disassembled because the byte that is inserted is the opcode for a multibyte 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.
- But what if the rogue byte can't be ignored? What if it is part of a legitimate instruction that is actually executed at runtime? These is where **Impossible Disassembly** takes place.

Impossible Disassembly

- With IDA Pro, we were able to work with the disassembly and have it produce accurate results. However, under some conditions, no traditional assembly listing will accurately represent the instructions that are executed.
- We use the term impossible disassembly for such conditions, but the term isn't strictly accurate. You could disassemble these techniques, but you would need a vastly different representation of code than what is currently provided by disassemblers.

Impossible Disassembly

- With IDA Pro, we were able to work with the disassembly and have it produce accurate results. However, under some conditions, no traditional assembly listing will accurately represent the instructions that are executed.
- We use the term impossible disassembly for such conditions, but the term isn't strictly accurate. You could disassemble these techniques, but you would need a vastly different representation of code than what is currently provided by disassemblers.

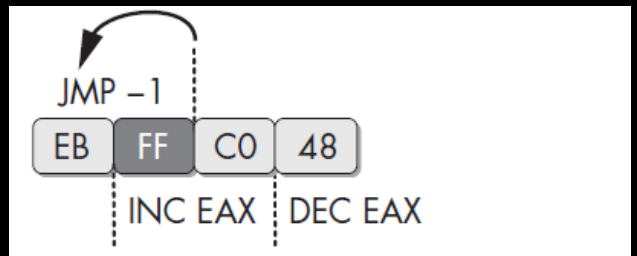


Figure 15-4: Inward-pointing jmp instruction

Impossible Disassembly

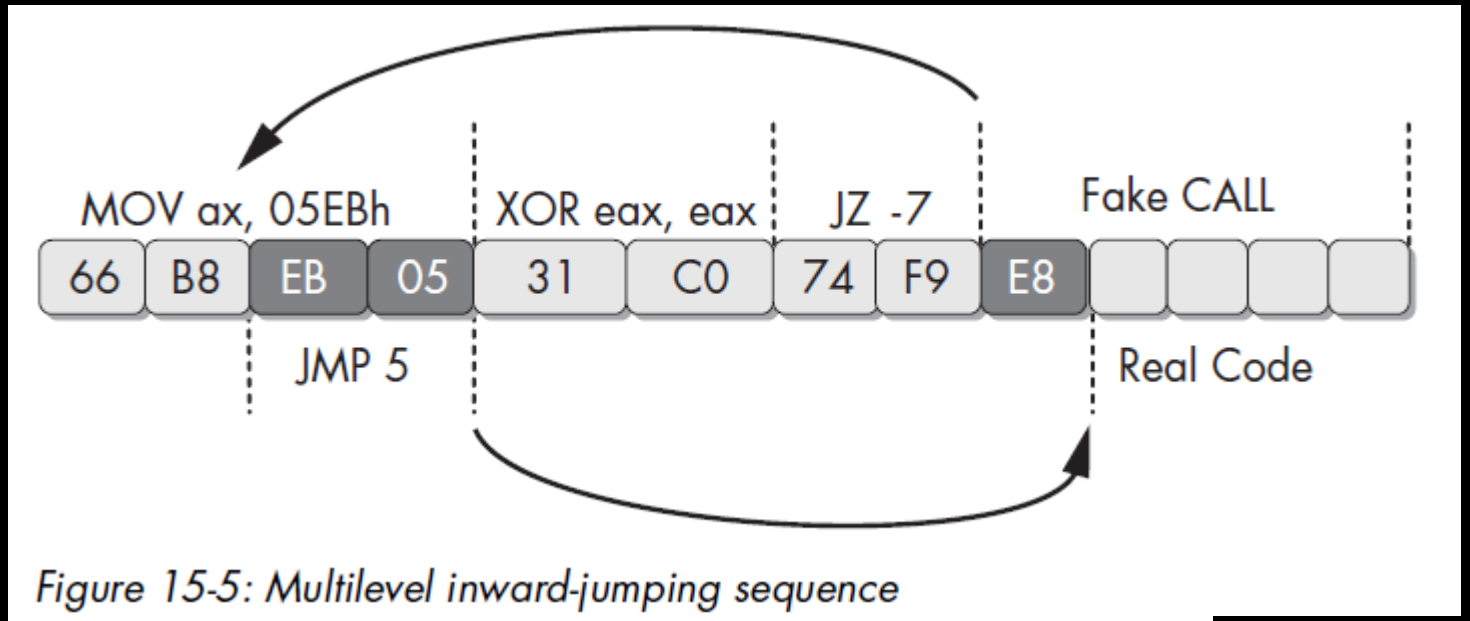


Figure 15-5: Multilevel inward-jumping sequence

```
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
```

Solution

```
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
```

```
90                      nop
90                      nop
90                      nop
90                      nop
31 C0                   xor     eax, eax
90                      nop
90                      nop
90                      nop
58                      pop     eax
C3                      retn
```

Anti – Debugging Techniques

- **Anti-debugging** is a popular anti-analysis technique used by malware to **recognize when it is under the control of a debugger** or to thwart debuggers.

- Once malware realizes that it is running in a debugger, it may alter its normal code execution path or modify the code to cause a crash, thus interfering with the analysts' attempts to understand it, and adding time and additional overhead to their efforts.

Anti – Debugging Techniques

- **Windows Debugger Detection**
- Malware uses a variety of techniques to scan for indications that a debugger is attached, including using the for debugging **Windows API, manually checking memory structure** artifacts, and searching the system for residue left by a debugger.
- The **Windows API** provides several functions that can be used by a program to determine if it is being debugged.
- Some of these functions were designed for debugger detection
  - ***IsDebuggerPresent***
  - ***CheckRemoteDebuggerPresent***
  - ***NtQueryInformationProcess***
  - ***OutputDebugString***

Anti – Debugging Techniques

- **Manually Checking Structures**
- manually checking structures is the most common method used by malware authors.
- In performing manual checks, several flags within the PEB (Process Environment Block) structure provide information about the presence of a debugger.
- Checking the **BeingDebugged Flag**:
- A Windows PEB structure is maintained by the OS for each running process. It contains all user-mode parameters associated with a process.
- While a process is running, the location of the PEB can be referenced by the location fs:[30h]. For anti-debugging, malware will use that location to check the BeingDebugged flag.

mov method	push/pop method
mov eax, dword ptr fs:[30h] mov ebx, byte ptr [eax+2] test ebx, ebx jz NoDebuggerDetected	push dword ptr fs:[30h] pop edx cmp byte ptr [edx+2], 1 je DebuggerDetected

Anti – Virtual Machine Techniques

- The malware attempts to detect whether it is being run inside a virtual machine. If a virtual machine is detected, it can act differently or simply not run.
- Anti-VM techniques are most commonly found in malware that is widely deployed, such as bots, scareware, and spyware.

Anti – Virtual Machine Techniques

- **VMware Artifacts**
- The VMware environment leaves many artifacts on the system, especially when **VMware Tools is installed**. Malware can use these artifacts, which are present in the filesystem, registry, and process listing, to detect VMware.
- VMware processes are running: **VMwareService.exe**, **VMwareTray.exe**, and **VMwareUser.exe**.
- Any one of these can be found by malware as it searches the process listing for the VMware string.

Anti – Virtual Machine Techniques

- **VMware Artifacts**
- The virtual machine to have its own virtual network interface card (NIC). Because VMware must virtualize the NIC, it needs to create a MAC address for the virtual machine, and, depending on its configuration, the network adapter can also identify VMware usage.
- The first three bytes of a MAC address are typically specific to the vendor, and **MAC addresses starting with 00:0C:29** are associated with **VMware**.
- Malware can also detect VMware by other hardware, such as the motherboard.

Anti – Virtual Machine Techniques

- **There are a couple of ways to avoid this detection:**
- **Patch the binary** while debugging so that the jump at 0x40xxxx *(VM detect function)* will never be taken.
- Use a **hex editor to modify** the VMwareTray.exe **string** to read XXXareTray.exe to make the comparison fail since this is not a valid process string.
- **Uninstall VMware Tools** so that VMwareTray.exe will no longer run.

Anti – Virtual Machine Techniques

- **Using the Red Pill Anti-VM Technique**
- Red Pill is an anti-VM technique that executes the ***sidt* instruction** to grab the value of the **IDTR** (interrupt descriptor table register).
- The virtual machine monitor must relocate the guest's IDTR to avoid conflict with the host's IDTR.
- Since the virtual machine monitor is not notified when the virtual machine runs the sidt instruction, the IDTR for the virtual machine is returned.
- The Red Pill tests for this discrepancy to detect the usage of VMware.

Anti – Virtual Machine Techniques

- **Using the Red Pill Anti-VM Technique**
- The IDTR is 6 bytes, and the fifth byte offset contains the start of the base memory address. **That fifth byte is compared to 0xFF, the VMware signature**.
- Red Pill succeeds **only on a single-processor machine**. It won't work consistently against multicore processors because each processor (guest or host) has an IDT assigned to it.

References

- Practical Malware Analysis by Michael Sikorski