15

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.

All malware is designed with a particular goal in mind: keystroke logging, backdoor access, using a target system to send excessive email to cripple servers, and so on. Malware authors often go beyond this basic functionality to implement specific techniques to hide from the user or system administrator, using rootkits or process injection, or to otherwise thwart analysis and detection.

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. The time-sensitive investigative process is hindered by the malware analyst's inability to understand the malware's capabilities, derive valuable host and network signatures, and develop decoding algorithms. 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.

In addition to delaying or preventing human analysis, anti-disassembly is also effective at preventing certain automated analysis techniques. Many malware similarity detection algorithms and antivirus heuristic engines employ disassembly analysis to identify or classify malware. Any manual or automated process that uses individual program instructions will be susceptible to the anti-analysis techniques described in this chapter.

Understanding Anti-Disassembly

Disassembly is not a simple problem. Sequences of executable code can have multiple disassembly representations, some that may be invalid and obscure the real functionality of the program. 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. For 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. For example, examine the following fragment of disassembled code:

```
                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
```

This fragment of code was disassembled using the linear-disassembly technique, and the result is inaccurate. Reading this code, we miss the piece of information that its author is trying to hide. We see what appears to be a `call` instruction, but the target of the call is nonsensical ❶. The first instruction is a `jmp` instruction whose target is invalid because it falls in the middle of the next instruction.

Now examine the same sequence of bytes disassembled with a different strategy:

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

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

This fragment reveals a different sequence of assembly mnemonics, and it appears to be more informative. Here, we see a call to the API function `Sleep` at ❶. The target of the first `jmp` instruction is now properly represented, and we can see that it jumps to a `push` instruction followed by the call to `Sleep`. The byte on the third line of this example is 0xE8, but this byte is not executed by the program because the `jmp` instruction skips over it.

This fragment was disassembled with a flow-oriented disassembler, rather than the linear disassembler used previously. In this case, the flow-oriented disassembler was more accurate because its logic more closely mirrored the real program and did not attempt to disassemble any bytes that were not part of execution flow. We'll discuss linear and flow-oriented disassembly in more detail in the next section.

So, disassembly is not as simple as you may have thought. The disassembly examples show two completely different sets of instructions for the same set of bytes. This demonstrates how anti-disassembly can cause the disassembler to produce an inaccurate set of instructions for a given range of bytes.

Some anti-disassembly techniques are generic enough to work on most disassemblers, while some target specific products.

Defeating 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 is easier to implement, but it's also more error-prone.

Linear Disassembly

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 debuggers.

Linear disassembly uses the size of the disassembled instruction to determine which byte to disassemble next, without regard for flow-control instructions.

The following code fragment shows the use of the disassembly library libdisasm (*http://sf.net/projects/bastard/files/libdisasm/*) to implement a crude disassembler in a handful of lines of C using linear disassembly:

```
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();
```

In this example, a buffer of data named `buffer` contains instructions to be disassembled. The function `x86_disasm` will populate a data structure with the specifics of the instruction it just disassembled and return the size of the instruction. The loop increments the `position` variable by the `size` value ❶ if a valid instruction was disassembled; otherwise, it increments by one ❷.

This algorithm will disassemble most code without a problem, but it will introduce occasional errors even in nonmalicious binaries. The main drawback to this method is that it will disassemble too much code. The algorithm will keep blindly disassembling until the end of the buffer, even if flow-control instructions will cause only a small portion of the buffer to execute.

In a PE-formatted executable file, the executable code is typically contained in a single section. It is reasonable to assume that you could get away with just 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. The following disassembly fragment (from a nonlinear disassembler) shows a function that contains switch pointers immediately following the function code.

```
        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
```

The last instruction in this function is `retn`. In memory, the bytes immediately following the `retn` instruction are the pointer values beginning with 401020 at ❶, which in memory will appear as the byte sequence 20 10 40 00 in hex. These four pointer values shown in the code fragment make up 16 bytes of data inside the `.text` section of this binary. They also happen to disassemble to valid instructions. The following disassembly fragment would be produced by a linear-disassembly algorithm when it continues disassembling instructions beyond the end of the function:

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

Many of instructions in this fragment consist of multiple bytes. The key way that malware authors exploit linear-disassembly algorithms lies in planting data bytes that form the opcodes of multibyte instructions. For example, the standard local `call` instruction is 5 bytes, beginning with the opcode `0xE8`. If the 16 bytes of data that compose the switch table end with the value `0xE8`, the disassembler would encounter the `call` instruction opcode and treat the next 4 bytes as an operand to that instruction, instead of the beginning of the next function.

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

Flow-Oriented Disassembly

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.

The following fragment shows code that can be disassembled correctly only with a flow-oriented disassembler.

```
                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
```

This example begins with a `test` and a conditional jump. When the flow-oriented disassembler reaches the conditional branch instruction `jz` at ❶, it notes that at some point in the future it needs to disassemble the location `loc_1A` at ❺. Because this is only a conditional branch, the instruction at ❷ is also a possibility in execution, so the disassembler will disassemble this as well.

The lines at ❷ and ❸ are responsible for printing the string `Failed` to the screen. Following this is a `jmp` instruction at ❹. The flow-oriented disassembler will add the target of this, `loc_1D`, to the list of places to disassemble in the future. Since `jmp` is unconditional, the disassembler will not automatically disassemble the instruction immediately following in memory. Instead, it will step back and check the list of places it noted previously, such as `loc_1A`, and disassemble starting from that point.

In contrast, when a linear disassembler encounters the `jmp` instruction, it will continue blindly disassembling instructions sequentially in memory, regardless of the logical flow of the code. In this case, the `Failed` string would be disassembled as code, inadvertently hiding the ASCII string and the last two instructions in the example fragment. For example, the following fragment shows the same code disassembled with a linear-disassembly algorithm.

```
                test    eax, eax
                jz      short near ptr loc_15+5
                push    Failed_string
                call    printf
                jmp     short loc_15+9
Failed_string:
                inc     esi
                popa
loc_15:
                imul    ebp, [ebp+64h], 0C3C03100h
```

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. Though assumptions and choices might seem unnecessary, simple machine code instructions are complicated by the addition of problematic code aspects such as pointers, exceptions, and conditional branching.

Conditional branches give the flow-oriented disassembler a choice of two places to disassemble: the true or the false branch. In typical compiler-generated code, there would be no difference in output if the disassembler processes the true or false branch first. In handwritten assembly code and anti-disassembly code, however, the two branches can often produce different disassembly for the same block of code. When there is a conflict, most disassemblers trust their initial interpretation of a given location first. Most flow-oriented disassemblers will process (and thus trust) the false branch of any conditional jump first.

Figure 15-1 shows a sequence of bytes and their corresponding machine instructions. Notice the string `hello` in the middle of the instructions. When the program executes, this string is skipped by the `call` instruction, and its 6 bytes and NULL terminator are never executed as instructions.

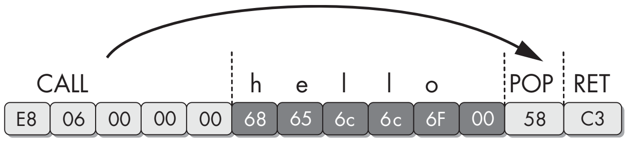

Figure 15-1: `call` instruction followed by a string

The `call` instruction is another place where the disassembler must make a decision. The location being called is added to the future disassembly list, along with the location immediately after the call. Just as with the conditional jump instructions, most disassemblers will disassemble the bytes after the `call` instruction first and the called location later. In handwritten assembly, programmers will often use the `call` instruction to get a pointer to a fixed piece of data instead of actually calling a subroutine. In this example, the `call` instruction is used to create a pointer for the string `hello` on the stack. The `pop` instruction following the call then takes this value off the top of the stack and puts it into a register (EAX in this case).

When we disassemble this binary with IDA Pro, we see that it has produced disassembly that is not what we expected:

```
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
```

As it turns out, the first letter of the string `hello` is the letter *h*, which is 0x68 in hexadecimal. This is also the opcode of the 5-byte instruction ❶ `push DWORD`. The null terminator for the `hello` string turned out to also be the first byte of another legitimate instruction. The flow-oriented disassembler in IDA Pro decided to process the thread of disassembly at ❶ (immediately following the `call` instruction) before processing the target of the `call` instruction, and thus produced these two erroneous instructions. Had it processed the target first, it still would have produced the first `push` instruction, but the instruction following the `push` would have conflicted with the real instructions it disassembled as a result of the `call` target.

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.

Here is the same function after manual cleanup:

```
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 Techniques

The primary way that malware can force a disassembler to produce inaccurate disassembly is by taking advantage of the disassembler's choices and assumptions. The techniques we will examine in this chapter exploit the most basic assumptions of the disassembler and are typically easily fixed by a malware analyst. More advanced techniques involve taking advantage of information that the disassembler typically doesn't have access to, as well as generating code that is impossible to disassemble completely with conventional assembly listings.

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. For 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. When the disassembler encounters the `jnz`, it continues disassembling the false branch of this instruction, despite the fact that it will never be executed in practice.

The following code shows IDA Pro's first interpretation of a piece of code protected with this technique:

```
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
```

In this example, the instruction immediately following the two conditional jump instructions appears to be a `call` instruction at ❶, beginning with the byte 0xE8. This is not the case, however, as 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 ❷ `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.

The following is disassembly of the same code, but this time fixed with the D key, to turn the byte immediately following the `jnz` instruction into data, and the C key to turn the bytes at `loc_4011C5` into instructions.

```
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
```

The column on the left in these examples shows the bytes that constitute the instruction. Display of this field is optional, but it's important when learning anti-disassembly. To display these bytes (or turn them off), select **Options ▸ General**. The Number of Opcode Bytes option allows you to enter a number for how many bytes you would like to be displayed.

Figure 15-2 shows the sequence of bytes in this example graphically.

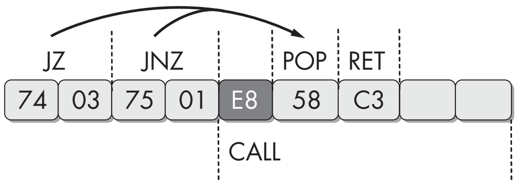

Figure 15-2: A jz instruction followed by a jnz instruction

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. The following code uses this technique:

```
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
```

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. The next instruction is a conditional jump that will jump if the zero flag is set. In reality, this is not conditional at all, since we can guarantee that the zero flag will always be set at this point in the program.

As discussed previously, the disassembler will process the false branch first, which will produce conflicting code with the true branch, and since it processed the false branch first, it trusts that branch more. As you've learned, 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
```

In this example, the 0xE9 byte is used exactly as the 0xE8 byte in the previous example. `E9` is the opcode for a 5-byte `jmp` instruction, and `E8` is the opcode for a 5-byte `call` instruction. In each case, by tricking the disassembler into disassembling this location, the 4 bytes following this opcode are effectively hidden from view. Figure 15-3 shows this example graphically.

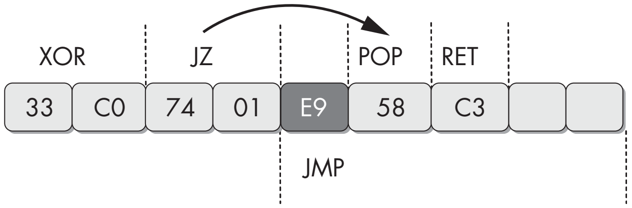

Figure 15-3: False conditional of `xor` followed by a `jz` instruction

Impossible Disassembly

In the previous sections, we examined code that was improperly disassembled by the first attempt made by the disassembler, but with an interactive disassembler like 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.

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. In all of these examples, the rogue byte can be ignored.

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? Here, we encounter a tricky scenario where any given byte may be a part of multiple instructions that are executed. No disassembler currently on the market will represent a single byte as being part of two instructions, yet the processor has no such limitation.

Figure 15-4 shows an example. The first instruction in this 4-byte sequence is a 2-byte `jmp` instruction. The target of the jump is the second byte of itself. This doesn't cause an error, because the byte FF is the first byte of the next 2-byte instruction, `inc eax`.

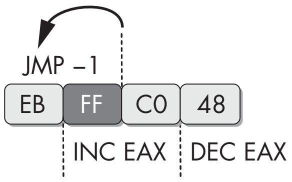

Figure 15-4: Inward-pointing `jmp` instruction

The predicament when trying to represent this sequence in disassembly is that if we choose to represent the FF byte as part of the `jmp` instruction, then it won't be available to be shown as the beginning of the `inc eax` instruction. The FF byte is a part of both instructions that actually execute, and our modern disassemblers have no way of representing this. This 4-byte sequence increments EAX, and then decrements it, which is effectively a complicated NOP sequence. It could be inserted at almost any location within a program to break the chain of valid disassembly. To solve this problem, a malware analyst could choose to replace this entire sequence with NOP instructions using an IDC or IDAPython script that calls the `PatchByte` function. Another alternative is to simply turn it all into data with the D key, so that disassembly will resume as expected at the end of the 4 bytes.

For a glimpse of the complexity that can be achieved with these sorts of instruction sequences, let's examine a more advanced specimen. Figure 15-5 shows an example that operates on the same principle as the prior one, where some bytes are part of multiple instructions.

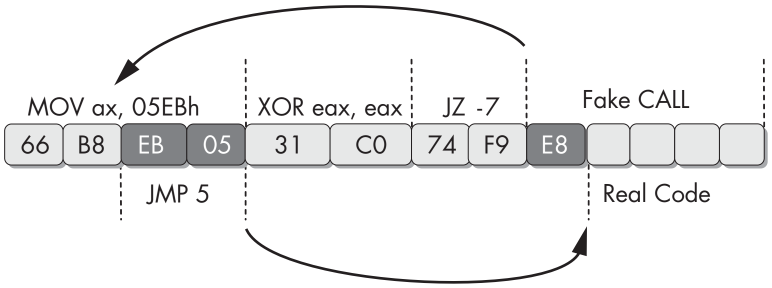

Figure 15-5: Multilevel inward-jumping sequence

The first instruction in this sequence is a 4-byte mov instruction. The last 2 bytes have been highlighted because they are both part of this instruction and are also their own instruction to be executed later. The first instruction populates the AX register with data. The second instruction, an xor, will zero out this register and set the zero flag. The third instruction is a conditional jump that will jump if the zero flag is set, but it is actually unconditional, since the previous instruction will always set the zero flag. The disassembler will decide to disassemble the instruction immediately following the jz instruction, which will begin with the byte 0xE8, the opcode for a 5-byte call instruction. The instruction beginning with the byte E8 will never execute in reality.

The disassembler in this scenario can't disassemble the target of the jz instruction because these bytes are already being accurately represented as part of the mov instruction. The code that the jz points to will always be executed, since the zero flag will always be set at this point. The jz instruction points to the middle of the first 4-byte mov instruction. The last 2 bytes of this instruction are the operand that will be moved into the register. When disassembled or executed on their own, they form a jmp instruction that will jump forward 5 bytes from the end of the instruction.

When first viewed in IDA Pro, this sequence will look like the following:

```
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
```

Since there is no way to clean up the code so that all executing instructions are represented, we must choose the instructions to leave in. The net side effect of this anti-disassembly sequence is that the EAX register is set to zero. If you manipulate the code with the D and C keys in IDA Pro so that the only instructions visible are the xor instruction and the hidden instructions, your result should look like the following.

```
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
```

This is a somewhat acceptable solution because it shows only the instructions that are relevant to understanding the program. However, this solution may interfere with analysis processes such as graphing, since it's difficult to tell exactly how the `xor` instruction or the `pop` and `retn` sequences are executed. A more complete solution would be to use the `PatchByte` function from the IDC scripting language to modify remaining bytes so that they appear as NOP instructions.

This example has two areas of undisassembled bytes that we need to convert into NOP instructions: 4 bytes starting at memory address 0x004011C0 and 3 bytes starting at memory address 0x004011C6. The following IDAPython script will convert these bytes into NOP bytes (0x90):

```
def NopBytes(start, length):
   for i in range(0, length):
     PatchByte(start + i, 0x90)
   MakeCode(start)

NopBytes(0x004011C0, 4)
NopBytes(0x004011C6, 3)
```

This code takes the long approach by making a utility function called `NopBytes` to NOP-out a range of bytes. It then uses that utility function against the two ranges that we need to fix. When this script is executed, the resulting disassembly is clean, legible, and logically equivalent to the original:

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

The IDAPython script we just crafted worked beautifully for this scenario, but it is limited in its usefulness when applied to new challenges. To reuse the previous script, the malware analyst must decide which offsets and which length of bytes to change to NOP instructions, and manually edit the script with the new values.

NOP-ing Out Instructions with IDA Pro

With a little IDA Python knowledge, we can develop a script that allows malware analysts to easily NOP-out instructions as they see fit. The following script establishes the hotkey ALT-N. Once this script is executed, whenever the user presses ALT-N, IDA Pro will NOP-out the instruction that is currently at the cursor location. It will also conveniently advance the cursor to the next instruction to facilitate easy NOP-outs of large blocks of code.

```
import idaapi

idaapi.CompileLine('static n_key() { RunPythonStatement("nopIt()"); }')

AddHotkey("Alt-N", "n_key")

def nopIt():

      start = ScreenEA()
      end = NextHead(start)
      for ea in range(start, end):
            PatchByte(ea, 0x90)
      Jump(end)
      Refresh()
```

Obscuring Flow Control

Modern disassemblers such as IDA Pro do an excellent job of correlating function calls and deducing high-level information based on the knowledge of how functions are related to each other. This type of analysis works well against code written in a standard programming style with a standard compiler, but is easily defeated by the malware author.

The Function Pointer Problem

Function pointers are a common programming idiom in the C programming language and are used extensively behind the scenes in C++. Despite this, they still prove to be problematic to a disassembler.

Using function pointers in the intended fashion in a C program can greatly reduce the information that can be automatically deduced about program flow. If function pointers are used in handwritten assembly or crafted in a nonstandard way in source code, the results can be difficult to reverse-engineer without dynamic analysis.

The following assembly listing shows two functions. The second function uses the first through a function pointer.

```
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
```

While this example isn't particularly difficult to reverse-engineer, it does expose one key issue. The function `sub_4011C0` is actually called from two different places (❷ and ❸) within the `sub_4011D0` function, but it shows only one cross-reference at ❶. This is because IDA Pro was able to detect the initial reference to the function when its offset was loaded into a stack variable on line `004011D5`. What IDA Pro does not detect, however, is the fact that this function is then called twice from the locations ❷ and ❸. Any function prototype information that would normally be autopropagated to the calling function is also lost.

When used extensively and in combination with other anti-disassembly techniques, function pointers can greatly compound the complexity and difficulty of reverse-engineering.

Adding Missing Code Cross-References in IDA Pro

All of the information not autopropagated upward, such as function argument names, can be added manually as comments by the malware analyst. In order to add actual cross-references, we must use the IDC language (or IDAPython) to tell IDA Pro that the function `sub_4011C0` is actually called from the two locations in the other function.

The IDC function we use is called `AddCodeXref`. It takes three arguments: the location the reference is from, the location the reference is to, and a flow type. The function can support several different flow types, but for our purposes, the most useful are either `fl_CF` for a normal `call` instruction or a `fl_JF` for a jump instruction. To fix the previous example assembly code listing in IDA Pro, the following script was executed:

```
AddCodeXref(0x004011DE, 0x004011C0, fl_CF);
AddCodeXref(0x004011EA, 0x004011C0, fl_CF);
```

Return Pointer Abuse

The `call` and `jmp` instructions are not the only instructions to transfer control within a program. The counterpart to the `call` instruction is `retn` (also represented as `ret`). The `call` instruction acts just like the `jmp` instruction, except it pushes a return pointer on the stack. The return point will be the memory address immediately following the end of the `call` instruction itself.

As `call` is a combination of `jmp` and `push`, `retn` is a combination of `pop` and `jmp`. The `retn` instruction pops the value from the top of the stack and jumps to it. It is typically used to return from a function call, but there is no architectural reason that it can't be used for general flow control.

When the `retn` instruction is used in ways other than to return from a function call, even the most intelligent disassemblers can be left in the dark. The most obvious result of this technique is that the disassembler doesn't show any code cross-reference to the target being jumped to. Another key benefit of this technique is that the disassembler will prematurely terminate the function.

Let's examine the following assembly fragment:

```
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
```

This is a simple function that takes a number and returns the product of that number times 42. Unfortunately, IDA Pro is unable to deduce any meaningful information about this function because it has been defeated by a rogue `retn` instruction. Notice that it has not detected the presence of an argument to this function. The first three instructions accomplish the task of jumping to the real start of the function. Let's examine each of these instructions.

The first instruction in this function is `call $+5`. This instruction simply calls the location immediately following itself, which results in a pointer to this memory location being placed on the stack. In this specific example, the value `0x004011C5` will be placed at the top of the stack after this instruction executes. This is a common instruction found in code that needs to be self-referential or position-independent, and will be covered in more detail in Chapter 19.

The next instruction is `add [esp+4+var_4], 5`. If you are used to reading IDA Pro disassembly, you might think that this instruction is referencing a stack variable `var_4`. In this case, IDA Pro's stack-frame analysis was incorrect, and this instruction was not referencing what would be a normal stack variable, autonamed to `var_4` in an ordinary function. This may seem confusing at first, but notice that at the top of the function, `var_4` is defined as the constant `-4`. This means that what is inside the brackets is `[esp+4+(-4)]`, which can also be represented as `[esp+0]` or simply `[esp]`. This instruction is adding five to the value at the top of the stack, which was `0x004011C5`. The result of the addition instruction is that the value at the top of the stack will be `0x004011CA`.

The last instruction in this sequence is the `retn` instruction, which has the sole purpose of taking this value off the stack and jumping to it. If you examine the code at the location 0x004011CA, it appears to be the legitimate beginning of a rather normal-looking function. This "real" function was determined by IDA Pro to not be part of any function due to the presence of the rogue `retn` instruction.

To repair this example, we could patch over the first three instructions with NOP instructions and adjust the function boundaries to cover the real function.

To adjust the function boundaries, place the cursor in IDA Pro inside the function you wish to adjust and press ALT-P. Adjust the function end address to the memory address immediately following the last instruction in the function. To replace the first few instructions with `nop`, refer to the script technique described in "NOP-ing Out Instructions with IDA Pro" on page 340.

Misusing Structured Exception Handlers

The Structured Exception Handling (SEH) mechanism provides a method of flow control that is unable to be followed by disassemblers and will fool debuggers. SEH is a feature of the x86 architecture and is intended to provide a way for the program to handle error conditions intelligently. Programming languages such as C++ and Ada rely heavily on exception handling and translate naturally to SEH when compiled on x86 systems.

Before exploring how to harness SEH to obscure flow control, let's look at a few basic concepts about how it operates. Exceptions can be triggered for a number of reasons, such as access to an invalid memory region or dividing by zero. Additional software exceptions can be raised by calling the `RaiseException` function.

The SEH chain is a list of functions designed to handle exceptions within the thread. Each function in the list can either handle the exception or pass it to the next handler in the list. If the exception makes it all the way to the last handler, then it is considered to be an *unhandled exception*. The last exception handler is the piece of code responsible for triggering the familiar message box that informs the user that "an unhandled exception has occurred." Exceptions happen regularly in most processes, but are handled silently before they make it to this final state of crashing the process and informing the user.

To find the SEH chain, the OS examines the FS segment register. This register contains a segment selector that is used to gain access to the Thread Environment Block (TEB). The first structure within the TEB is the Thread Information Block (TIB). The first element of the TIB (and consequently the first bytes of the TEB) is a pointer to the SEH chain. The SEH chain is a simple linked list of 8-byte data structures called `EXCEPTION_REGISTRATION` records.

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

The first element in the `EXCEPTION_REGISTRATION` record points to the previous record. The second field is a pointer to the handler function.

This linked list operates conceptually as a stack. The first record to be called is the last record to be added to the list. The SEH chain grows and shrinks as layers of exception handlers in a program change due to subroutine calls and nested exception handler blocks. For this reason, SEH records are always built on the stack.

In order to use SEH to achieve covert flow control, we need not concern ourselves with how many exception records are currently in the chain. We just need to understand how to add our own handler to the top of this list, as shown in Figure 15-6.

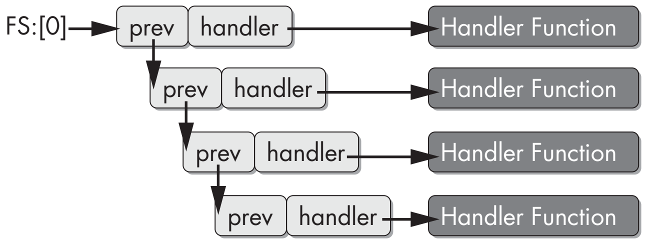

Figure 15-6: Structured Exception Handling (SEH) chain

To add a record to this list, we need to construct a new record on the stack. Since the record structure is simply two DWORDs, we can do this with two push instructions. The stack grows upward, so the first push will be the pointer to the handler function, and the second push will be the pointer to the next record. We are trying to add a record to the top of the chain, so the next record in the chain when we finish will be what is currently the top, which is pointed to by fs:[0]. The following code performs this sequence.

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

The ExceptionHandler function will be called first whenever an exception occurs. This action will be subject to the constraints imposed by Microsoft's Software Data Execution Prevention (Software DEP, also known as SafeSEH).

Software DEP is a security feature that prevents the addition of third-party exception handlers at runtime. For purposes of handwritten assembly code, there are several ways to work around this technology, such as using an assembler that has support for SafeSEH directives. Using Microsoft's C compilers, an author can add /SAFESEH:NO to the linker command line to disable this.

When the ExceptionHandler code is called, the stack will be drastically altered. Luckily, it is not essential for our purposes to fully examine all the data that is added to the stack at this point. We must simply understand how to return the stack to its original position prior to the exception. Remember that our goal is to obscure flow control and not to properly handle program exceptions.

The OS adds another SEH handler when our handler is called. To return the program to normal operation, we need to unlink not just our handler, but this handler as well. Therefore, we need to pull our original stack pointer from esp+8 instead of esp.

```
mov esp, [esp+8]
mov eax, fs:[0]
mov eax, [eax]
mov eax, [eax]
mov fs:[0], eax
add esp, 8
```

Let's bring all this knowledge back to our original goal of obscuring flow control. The following fragment contains a piece of code from a Visual C++ binary that covertly transfers flow to a subroutine. Since there is no pointer to this function and the disassembler doesn't understand SEH, it appears as though the subroutine has no references, and the disassembler thinks the code immediately following the triggering of the exception will be executed.

```
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              ❶dd 824648Bh, 0A164h, 8B0000h, 0A364008Bh, 0
00401094                dd 6808C483h
00401098                dd offset aMysteryCode  ; "Mystery Code"
0040109C                dd 2DE8h, 4C48300h, 3 dup(0CCCCCCCCh)
```

In this example, IDA Pro has not only missed the fact that the subroutine at location 401080 ❶ was not called, but it also failed to even disassemble this function. This code sets up an exception handler covertly by first setting the register EAX to the value `40106C` ❷, and then adding `14h` to it to build a pointer to the function `401080`. A divide-by-zero exception is triggered by setting ECX to zero with `xor ecx, ecx` followed by `div ecx` at ❸, which divides the EAX register by ECX.

Let's use the C key in IDA Pro to turn the data at location 401080 into code and see what was hidden using this trick.

```
00401080                mov     esp, [esp+8]
00401084                mov     eax, large fs:0
0040108A                mov     eax, [eax]
0040108C                mov     eax, [eax]
0040108E                mov     large fs:0, eax
00401094                add     esp, 8
00401097                push    offset aMysteryCode ; "Mystery Code"
0040109C                call    printf
```

Thwarting Stack-Frame Analysis

Advanced disassemblers can analyze the instructions in a function to deduce the construction of its stack frame, which allows them to display the local variables and parameters relevant to the function. This information is extremely valuable to a malware analyst, as it allows for the analysis of a single function at one time, and enables the analyst to better understand its inputs, outputs, and construction.

However, analyzing a function to determine the construction of its stack frame is not an exact science. As with many other facets of disassembly, the algorithms used to determine the construction of the stack frame must make certain assumptions and guesses that are reasonable but can usually be exploited by a knowledgeable malware author.

Defeating stack-frame analysis will also prevent the operation of certain analytical techniques, most notably the Hex-Rays Decompiler plug-in for IDA Pro, which produces C-like pseudocode for a function.

Let's begin by examining a function that has been armored to defeat 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 -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

Stack-frame anti-analysis techniques depend heavily on the compiler used. Of course, if the malware is entirely written in assembly, then the author is free to use more unorthodox techniques. However, if the malware is crafted with a higher-level language such as C or C++, special care must be taken to output code that can be manipulated.

In Listing 15-1, the column on the far left is the standard IDA Pro line prefix, which contains the segment name and memory address for each function. The next column to the right displays the stack pointer. For each instruction, the stack pointer column shows the value of the ESP register relative to where it was at the beginning of the function. This view shows that this function is an ESP-based stack frame rather than an EBP-based one, like most functions. (This stack pointer column can be enabled in IDA Pro through the Options menu.)

At ❶, the stack pointer begins to be shown as a negative number. This should never happen for an ordinary function because it means that this function could damage the calling function's stack frame. In this listing, IDA Pro is also telling us that it thinks this function takes 62 arguments, of which it thinks 2 are actually being used.

NOTE *Press* CTRL-*K in IDA Pro to examine this monstrous stack frame in detail. If you attempt to press Y to give this function a prototype, you'll be presented with one of the most ghastly abominations of a function prototype you've ever seen.*

As you may have guessed, this function doesn't actually take 62 arguments. In reality, it takes no arguments and has two local variables. The code responsible for breaking IDA Pro's analysis lies near the beginning of the function, between locations 00401546 and 0040155C. It's a simple comparison with two branches.

The ESP register is being compared against the value `0x1000`. If it is less than `0x1000`, then it executes the code at 00401556; otherwise, it executes the code at 00401551. Each branch adds some value to ESP—0x104 on the "less-than" branch and 4 on the "greater-than-or-equal-to" branch. From a disassembler's perspective, there are two possible values of the stack pointer offset at this point, depending on which branch has been taken. The disassembler is forced to make a choice, and luckily for the malware author, it is tricked into making the wrong choice.

Earlier, we discussed conditional branch instructions, which were not conditional at all because they exist where the condition is constant, such as a `jz` instruction immediately following an `xor eax, eax` instruction. Innovative disassembler authors could code special semantics in their algorithm to track such guaranteed flag states and detect the presence of such fake conditional branches. The code would be useful in many scenarios and would be very straightforward, though cumbersome, to implement.

In Listing 15-1, the instruction `cmp esp, 1000h` will always produce a fixed result. An experienced malware analyst might recognize that the lowest memory page in a Windows process would not be used as a stack, and thus this comparison is virtually guaranteed to always result in the "greater-than-

or-equal-to" branch being executed. The disassembly program doesn't have this level of intuition. Its job is to show you the instructions. It's not designed to evaluate every decision in the code against a set of real-world scenarios.

The crux of the problem is that the disassembler assumed that the `add esp, 104h` instruction was valid and relevant, and adjusted its interpretation of the stack accordingly. The `add esp, 4` instruction in the greater-than-or-equal-to branch was there solely to readjust the stack after the `sub esp, 4` instruction that came before the comparison. The net result in real time is that the ESP value will be identical to what it was prior to the beginning of the sequence at address 00401546.

To overcome minor adjustments to the stack frame (which occur occasionally due to the inherently fallible nature of stack-frame analysis), in IDA Pro, you can put the cursor on a particular line of disassembly and press ALT-K to enter an adjustment to the stack pointer. In many cases, such as in Listing 15-1, it may prove more fruitful to patch the stack-frame manipulation instructions, as in the previous examples.

Conclusion

Anti-disassembly is not confined to the techniques discussed in this chapter. It is a class of techniques that takes advantage of the inherent difficulties in analysis. Advanced programs such as modern disassemblers do an excellent job of determining which instructions constitute a program, but they still require assumptions and choices to be made in the process. For each choice or assumption that can be made by a disassembler, there may be a corresponding anti-disassembly technique.

This chapter showed how disassemblers work and how linear and flow-oriented disassembly strategies differ. Anti-disassembly is more difficult with a flow-oriented disassembler but still quite possible, once you understand that the disassembler is making certain assumptions about where the code will execute. Many anti-disassembly techniques used against flow-oriented disassemblers operate by crafting conditional flow-control instructions for which the condition is always the same at runtime but unknown by the disassembler.

Obscuring flow control is a way that malware can cause the malware analyst to overlook portions of code or hide a function's purpose by obscuring its relation to other functions and system calls. We examined several ways to accomplish this, ranging from using the `ret` instruction to using SEH handlers as a general-purpose jump.

The goal of this chapter was to help you understand code from a tactical perspective. You learned how these types of techniques work, why they are useful, and how to defeat them when you encounter them in the field. More techniques are waiting to be discovered and invented. With this solid foundation, you will be more than prepared to wage war in the anti-disassembly battlefield of the future.

LABS

Lab 15-1

Analyze the sample found in the file *Lab15-01.exe*. This is a command-line program that takes an argument and prints "Good Job!" if the argument matches a secret code.

Questions

1. What anti-disassembly technique is used in this binary?
2. What rogue opcode is the disassembly tricked into disassembling?
3. How many times is this technique used?
4. What command-line argument will cause the program to print "Good Job!"?

Lab 15-2

Analyze the malware found in the file *Lab15-02.exe*. Correct all anti-disassembly countermeasures before analyzing the binary in order to answer the questions.

Questions

1. What URL is initially requested by the program?
2. How is the User-Agent generated?
3. What does the program look for in the page it initially requests?
4. What does the program do with the information it extracts from the page?

Lab 15-3

Analyze the malware found in the file *Lab15-03.exe*. At first glance, this binary appears to be a legitimate tool, but it actually contains more functionality than advertised.

Questions

1. How is the malicious code initially called?
2. What does the malicious code do?
3. What URL does the malware use?
4. What filename does the malware use?