# Anti-Disassembly

CYBR473 – Malware and Reverse Engineering (2024/T1)

Lecturers: Arman Khouzani, Alvin Valera

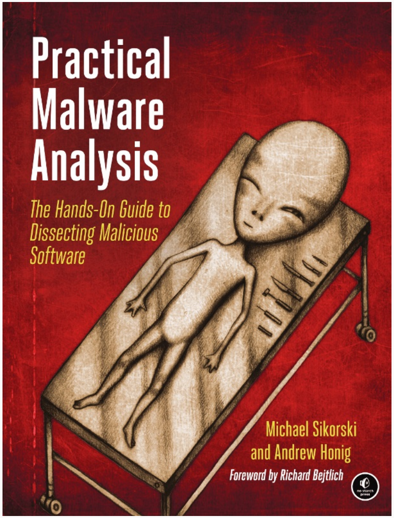
Victoria University of Wellington – School of Engineering and Computer Science

## Table of contents

## Reading Material

- Part V: Anti-Reverse-Engineering
  - Ch.16: Anti-Disassembly

*"Practical Malware Analysis: The Hands-on Guide to Dissecting Malicious Software", Michael Sikorski and Andrew Honig, 2012*


Practical Malware Analysis
The Hands-On Guide to Dissecting Malicious Software
Michael Sikorski and Andrew Honig
Foreword by Richard Bejtlich

## Anti-Disassembly

Anti-disassembly uses specially crafted code or data 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.

**Any code that executes successfully can be reverse-engineered, but anti-disassembly and anti-debugging techniques increase the time and level of skill required of the malware analyst.**

Anti-disassembly is also effective at preventing certain automated analysis: Any process that uses individual instructions (e.g. instruction-based similarity measures) will be susceptible.

## Understanding Anti-Disassembly

Understanding Anti-Disassembly

Defeating Disassembly Algorithms

Anti-Disassembly Techniques

Obscuring Flow Control

Thwarting Stack-Frame Analysis

## Understanding Anti-Disassembly

Anti-disassembly work by taking advantage of the assumptions and limitations of disassemblers. E.g., they can only represent each byte of a program as part of one instruction at a time. If tricked into disassembling at a wrong offset, a valid instruction could be hidden:

```
                jmp     short near ptr loc_2+1
; ---------------------------------------
loc_2:                     ; CODE XREF: seg000:00000000j
                call    near ptr 15FF2A71h
                or      [ecx], dl
                inc     eax
; ---------------------------------------
                db    0
```

The target of the `jmp` is invalid because it falls in the middle of the next instruction. The target of the `call` is nonsensical.

## Understanding Anti-Disassembly

```
                jmp     short loc_3
; ---------------------------------------
                db 0E8h
; ---------------------------------------
loc_3:                 ; CODE XREF: seg000:00000000j
                push    2Ah
                call    Sleep
```

The target of the first **jmp** instruction is now properly represented, 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.

## Defeating Disassembly Algorithms

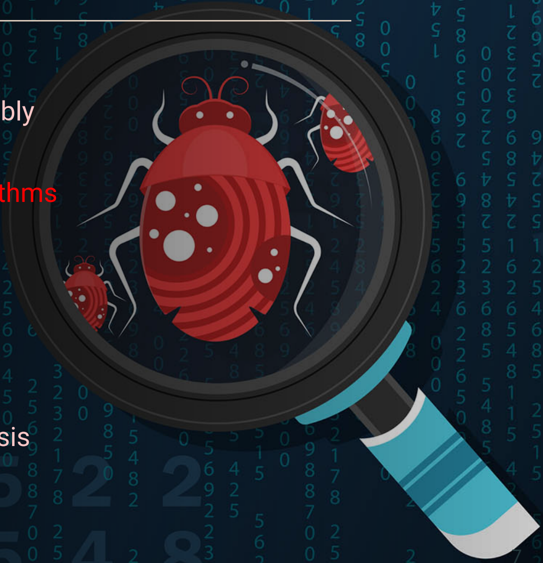
Understanding Anti-Disassembly

Defeating Disassembly Algorithms

Anti-Disassembly Techniques

Obscuring Flow Control

Thwarting Stack-Frame Analysis



## Defeating Disassembly Algorithms

There are two types of disassembler algorithms:

- **Linear**
- **Flow-Oriented**

The linear-disassembly strategy iterates over a block of code, disassembling one instruction at a time linearly, without deviating.

Linear disassembly uses the size of the disassembled instruction to determine which byte to disassemble next.

The main drawback to this method is that it will disassemble too much code: it keeps blindly disassembling, even if flow-control instructions will cause only a small portion to execute.

## 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);
  position += size;
 } else {
   /* invalid/unrecognized instruction */
  position++;
  }
}
x86_cleanup();
```

## Defeating Disassembly Algorithms: Linear Disassembly

A problem with linear disassembly is that the code section of nearly all binaries also contains data that isn't instructions.

For instance, one of the most common types of data items found in a code section, is a pointer value used in a table-driven switch idiom:

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

          ; switch cases omitted ...

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

## Defeating Disassembly Algorithms: Linear Disassembly

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, if 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
```

## Defeating Disassembly Algorithms: Linear Disassembly

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 reliable algorithm, used by most commercial disassemblers like IDA Pro, is the **flow-oriented** disassembly.

The key difference is that it doesn't blindly iterate over a buffer. Instead, it builds a list of locations to disassemble.

E.g., this can be disassembled only with a flow-oriented disassembler:

```
               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
```

## Flow-Oriented Disassembly

This example begins with a test and a conditional jump.

When the flow-oriented disassembler reaches the conditional branch instruction **`jz`**, it notes that at some point in the future it needs to disassemble the location **`loc_1A`**.

Because this is only a conditional branch, the following instruction is also a possibility in execution, so it will be disassembled as well.

The next two lines are responsible for printing the string `Failed`.

Following this is a **`jmp`** instruction. 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.

## Flow-Oriented Disassembly

In contrast, when a linear disassembler encounters the **`jmp`** instruction, it will continue blindly disassembling instructions sequentially in memory, as it has no choice to make about which instructions to disassemble at a given time:

```
                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
```

## Flow-Oriented Disassembly

Conditional branches give the flow-oriented disassembler a choice of two places to disassemble: the true or the false branch.

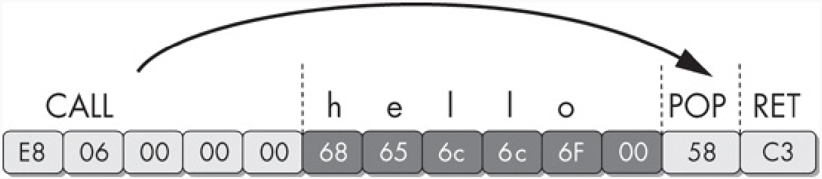
In typical compiler-generated code, there would be no difference in output if the disassembler processes the true or false branch first.

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.

## Defeating Disassembly Algorithms: Flow-Oriented Disassembly



A sequence of bytes and their corresponding machine instructions. When the program executes, the **`hello`** string is skipped by the **`call`** instruction, and its 6 bytes and `NULL` terminator are never executed as instructions.

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**, most disassemblers disassemble the bytes after the **`call`** first and the called location later.

## Defeating Disassembly Algorithms: Flow-Oriented Disassembly

In handwritten assembly, programmers often use **`call`** 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.

But disassembling this binary with IDA Pro, produces wrong results:

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

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

## Defeating Disassembly Algorithms: Flow-Oriented Disassembly

"`h`", the first letter of the string `hello`, is `0x68`. As it turns out, this is also the opcode of the 5-byte instruction `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 bytes immediately following the `call` before processing the target of the `call`, and thus produced these two erroneous instructions.

Had it processed the target first, it still would have produced the first `push`, but the instruction following it would have conflicted with the real instructions it disassembled as a result of the `call` target.

## Defeating Disassembly Algorithms: Flow-Oriented Disassembly

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

Understanding Anti-Disassembly

Defeating Disassembly Algorithms

Anti-Disassembly Techniques

Obscuring Flow Control

Thwarting Stack-Frame Analysis

## Anti-Disassembly Techniques: Jumps with the Same Target

A common anti-disassembly technique is two back-to-back conditional jump instructions that both point to the same target.

For example, a `jz loc_A` followed by `jnz loc_A` is, in effect, an unconditional `jmp`, but the disassembler continues disassembling the false branch of `jnz` even though it will never be reached:

```
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
```

In this example, the instruction immediately following the two conditional jumps appears to be a `call` instruction, beginning with the byte **`0xE8`**. This is not the case, however, as both conditional jump instructions actually point 1 byte beyond the **`0xE8`** byte.

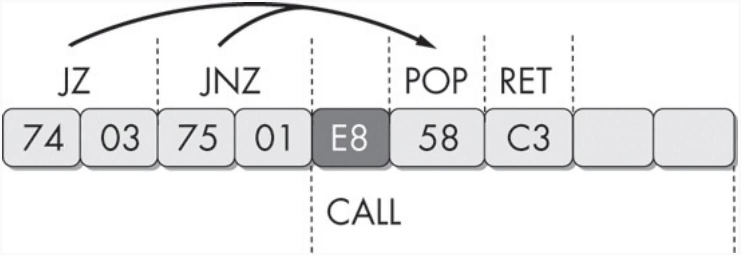
## Anti-Disassembly Techniques: Jumps with the Same Target

The following is disassembly of the same code, but this time fixed with the **D** key, to turn the byte immediately following the **jnz** 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
```

*Note:* To display the raw bytes (on the left) in IDA Pro, under *Options* ▶ *General*, set the *Number of Opcode Bytes* option appropriately.

## Anti-Disassembly Techniques: Jumps with the Same Target



A **`jz`** instruction followed by a **`jnz`** instruction.

## Anti-Disassembly Techniques: Jump with a Constant Condition

Another anti-disassembly technique is a single conditional jump instruction whose condition will always be the same:

```
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 `xor eax, eax` sets **EAX** to zero and, as a byproduct, sets the zero flag. Hence, the next instruction is not conditional at all.

As discussed before, the disassembler processes 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.

## Anti-Disassembly Techniques: Jump with a Constant Condition

After fixing the disassembly (using the **`D`** and **`C`** keyboard shortcuts):

```
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
```

## Anti-Disassembly Techniques: Jump with a Constant Condition

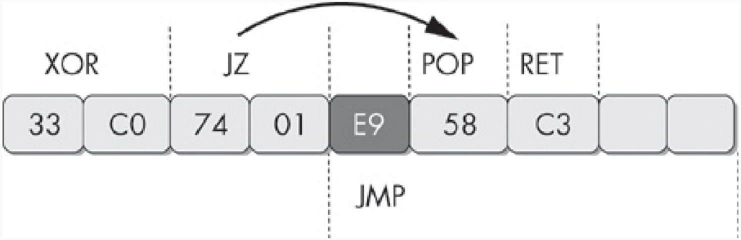


Illustration of the example of a conditional jump with a constant condition.

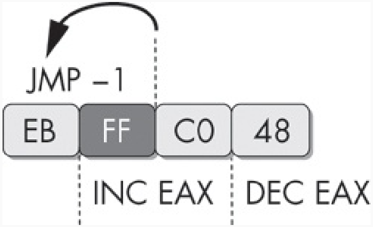
## Anti-Disassembly Techniques: Impossible Disassembly

So far, we examined code that was improperly disassembled first, but with an interactive disassembler, accurate results could be achieved.

However, sometimes no traditional listing accurately represents the executed instructions. The book calls them *impossible disassembly*.

The simple anti-disassembly techniques we have discussed use a strategically placed "rogue" data byte, which can be ignored.

But what if a byte is part of multiple instructions that are executed?



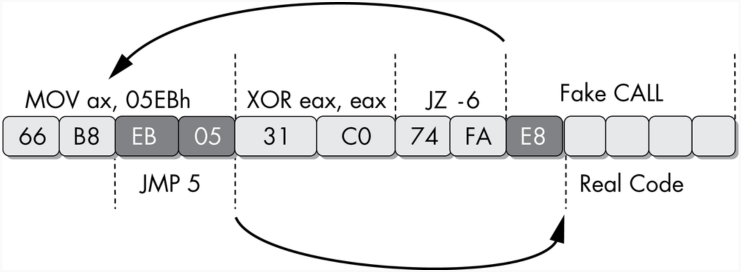
Inward-pointing **`jmp`** instruction (this is effectively a complicated **`NOP`** – why?)

## Anti-Disassembly Techniques: Impossible 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.

## Anti-Disassembly Techniques: Impossible Disassembly

Let's examine a more advanced specimen:



Multilevel inward-jumping sequence

The disassembler disassembles the instruction immediately following `jz`, which begins with `0xE8`, the opcode for a 5-byte `call`.

The disassembler can't disassemble the target of the `jz` since these bytes are already represented as part of the `mov` instruction.

## Anti-Disassembly Techniques: Impossible Disassembly

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 FA            jz    short near ptr sub_4011C0+2
        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. Which one should it be?

After manipulating the code (with **`D`** and **`C`** keys in IDA Pro), the result should look like the following:

## Anti-Disassembly Techniques: Impossible Disassembly

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

## Anti-Disassembly Techniques: Impossible Disassembly

However, this solution may interfere with analysis processes such as graphing. 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:

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

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

## Anti-Disassembly Techniques: Impossible Disassembly

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
```

## NOP-ing Out Instructions with IDA Pro

The following script establishes the hotkey **ALT-N**, allowing the analysts to **NOP**-out instructions as they see fit at the cursor location:

```
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

## Obscuring Flow Control: 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.

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.

## Obscuring Flow Control: The Function Pointer Problem

```
004011C0 sub_4011C0      proc near    ; DATA XREF: sub_4011D0+5o
004011C0
004011C0 arg_0           = dword ptr  8
004011C0
004011C0                 push    ebp
004011C1                 mov     ebp, esp
004011C3                 mov     eax, [ebp+arg_0]
004011C6                 shl     eax, 2
004011C9                 pop     ebp
004011CA                 retn
004011CA sub_4011C0      endp
004011D0 sub_4011D0      proc near   ; CODE XREF: _main+19p
004011D0                             ; sub_401040+8Bp
004011D0
004011D0 var_4           = dword ptr -4
004011D0 arg_0           = dword ptr  8
004011D0
004011D0                 push    ebp
004011D1                 mov     ebp, esp
...
```

## Obscuring Flow Control: The Function Pointer Problem

```
...
004011D3                push    ecx
004011D4                push    esi
004011D5                mov     [ebp+var_4], offset sub_4011C0
004011DC                push    2Ah
004011DE                call    [ebp+var_4]
004011E1                add     esp, 4
004011E4                mov     esi, eax
004011E6                mov     eax, [ebp+arg_0]
004011E9                push    eax
004011EA                call    [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
```

## Obscuring Flow Control: The Function Pointer Problem

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 within the **`sub_4011D0`** function, but it shows only one cross-reference.

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 it does not detect, however, is the fact that this function is then called twice. 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, such as function argument names, can be added manually as comments.

To add cross-references, we use the **IDC** language (or **IDAPython**), specifically, the **`AddCodeXref`** function. It takes three arguments:

- the location the reference is from;
- the location the reference is to;
- and a flow type.

The most useful flow types are either **`fl_CF`** for a normal call instruction or a **`fl_JF`** for a jump instruction.

To fix the previous example, the following script can be executed:

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

## Obscuring Flow Control: Return Pointer Abuse

The `call` and `jmp` instructions are not the only instructions to transfer control within a program (what is their difference?)

The counterpart to the `call` instruction is `retn` (also as `ret`).

As `call` is a combination of `jmp` and `push`, `retn` is a combination of `pop` and `jmp`. A disassembly technique is based on using `retn` in ways other than to return from a function call.

- ▷ A result of this technique is that the disassembler doesn't show any code cross-reference to the target being jumped to.
- ▷ Another effect of this technique is that the disassembler will prematurely terminate the function.

## Obscuring Flow Control: Return Pointer Abuse

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

## Obscuring Flow Control: Return Pointer Abuse

This function simply takes a number and returns its product with 42. Unfortunately, IDA Pro is unable to deduce any meaningful information about it as it has been defeated by a rogue **`retn`**.

The first three instructions just jump to the real start of the function:

- **`call $+5`**: calls the location immediately following itself, which results in a pointer to this location (`004011C5`) placed on the stack. This is a common instruction found in self-referential or position-independent code.
- **`add [esp+4+var_4], 5`**: you might think that this is referencing a stack variable var_4, but notice that at the top of the function, var_4 is defined as the constant -4. This means that we are just adding 5 to the value at the top of the stack, which will hence be `004011CA`.
- **`retn`**: takes the value off the top of the stack and jumps to it. Here, this is where the “real” function starts.

## Obscuring Flow Control: Return Pointer Abuse

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 (e.g. with a script, as discussed before).

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.

## Obscuring Flow Control: Misusing SEH

The **Structured Exception Handling (SEH)** mechanism provides a method of flow control that can defeat disassemblers and debuggers.

**SEH** is a feature of the x86 architecture and is intended to provide a way for the program to handle error conditions intelligently.

Exceptions can be triggered by e.g. access to an invalid memory region or dividing by zero, or calling the `RaiseException` function.

The **SEH chain** is a list of functions designed to handle exceptions within the thread.

- Each function either handles the exception or passes it to the next handler in the list.
- If gets to the last handler, “an unhandled exception has occurred.”

Exceptions happen regularly, but are handled silently before they crash the process.

## Obscuring Flow Control: Misusing SEH

To find the **SEH chain**, the OS examines the **`FS` segment register**, which contains a segment selector to access to the **Thread Environment Block (TEB)**.

The first structure within **TEB** is the **Thread Information Block (TIB)**.

The first element of **TIB** is a pointer to the **SEH chain**, which is a linked list of 8-byte structures called **`EXCEPTION_REGISTRATION`** records:

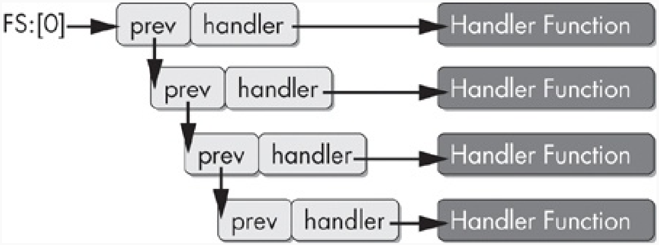
```
struct _EXCEPTION_REGISTRATION {
   DWORD prev;
   DWORD handler;
};
```

## Obscuring Flow Control: Misusing SEH

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.

To achieve covert flow control using SEH, we do not need to concern with how many exception records are currently in the chain; just how to add our handler to the top of this list:



## Obscuring Flow Control: Misusing SEH

To add a record to this list, we need to construct a new record on the stack. Since the record structure is simply two **`DWORD`**s, 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 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
```

## Obscuring Flow Control: Misusing SEH

When our **`ExceptionHandler`** code is called, the stack will be drastically altered. Luckily, we just need 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
```

## Obscuring Flow Control: Misusing SEH

The following fragment covertly transfers the flow to a subroutine:

```
00401050      mov   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      div   ecx
0040106B
0040106B loc_40106B:            ; DATA XREF: sub_401050o
0040106B      call  near ptr Sleep
00401070      retn
00401070 sub_401050      endp ; sp-analysis failed
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)
```

## Obscuring Flow Control: Misusing SEH

IDA Pro has not only missed that the subroutine at location 401080 was not called, but it also failed to even disassemble this function.

This code sets up an exception handler covertly by first setting **`EAX`** to **`40106C`**, then adding **`14h`** to build a pointer to the function **`401080`**.

A divide-by-zero exception is then triggered.

Let's use the C key at **`401080`** to 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

Understanding Anti-Disassembly

Defeating Disassembly Algorithms

Anti-Disassembly Techniques

Obscuring Flow Control

Thwarting Stack-Frame Analysis

## Thwarting Stack-Frame Analysis

Disassemblers deduce the construction of a function’s stack frame, allowing them to display its local variables and parameters.

This information is extremely valuable to a malware analyst, as it allows for the analysis of a single function at one time.

However, analyzing a function to determine the construction of its stack frame is not an exact science: it must make certain assumptions and guesses that are reasonable but can usually be exploited by a knowledgeable malware author.

Defeating stack-frame analysis also disrupts certain analytical techniques, e.g. the **`Hex-Rays Decompiler`** plug-in for **`IDA Pro`**.

## Thwarting Stack-Frame Analysis

```
00401543   sub_401543     proc near ; CODE XREF: sub_4012D0+3Cp
00401543                            ; sub_401328+9Bp
00401543
00401543    arg_F4         = dword ptr  0F8h
00401543    arg_F8         = dword ptr  0FCh
00401543
00401543 000                sub     esp, 8
00401546 008                sub     esp, 4
00401549 00C                cmp     esp, 1000h
0040154F 00C                jl      short loc_401556
00401551 00C                add     esp, 4
00401554 008                jmp     short loc_40155C
00401556     ; ------------------------------------
00401556
00401556    loc_401556:       ; CODE XREF: sub_401543+Cj
00401556 00C                add     esp, 104h
0040155C
...
```

## Thwarting Stack-Frame Analysis

```
    ...
0040155C    loc_40155C:      ; CODE XREF: sub_401543+11j
0040155C -F8                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
```

## Thwarting Stack-Frame Analysis

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 stack pointer column can be enabled in IDA Pro through the **Options** menu.)

This view shows that this function is an `ESP`-based stack frame rather than an `EBP`-based one, like most functions.

At `0040155C`, 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 damages 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.

## Thwarting Stack-Frame Analysis

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`**. 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.

## Thwarting Stack-Frame Analysis

Earlier, we discussed conditional branches with constant conditions.

Here, too, `cmp esp, 1000h` will always produce a fixed result: 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.

The disassembly program doesn’t have this level of intuition: 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.

In IDA Pro to make adjustment to the stack pointer, you can press **`ALT-K`** on a particular line. In many cases, such as this example, it is more fruitful to patch the stack-frame manipulation instructions.

## Next: Anti-Debugging