Obscuring Flow Control

Bui Viet Dung

June 26, 2025

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.

1 The function pointer problem

In the C programming language, a function pointer is a variable that stores the address of a function. This lets you call a function indirectly using that pointer. It's a common and powerful feature in C and C++, especially for callbacks, tables, and dynamic dispatch.But in disassembly and reverse engineering, function pointers cause problems.

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 reverseengineer without dynamic analysis.

1.1 Why is it a Problem for Disassemblers like IDA Pro

Disassemblers like IDA Pro try to figure out:

- What functions are being called
- The control flow of the program
- Where those functions are called from
- What arguments are passed

When a function is called **directly** like this:

```
call sub_4011C0
```

IDA can easily see:

- "Ah, `sub_4011C0` is called here"
- It adds a **cross-reference (xref)** to the call site

But when it's called using a **function pointer**, like this:

```
mov [ebp+var_4], offset sub_4011C0 ; stores function address
...
call [ebp+var_4]                   ; indirect call using pointer
```

IDA can't always figure out that the call is going to `sub_4011C0`. So, **no xref** is created for this call.

1.2 Example

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 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      [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
```

IDA Pro can easily detect direct calls like `call sub_4011C0` and create cross-references.But when the call is indirect through a pointer (e.g., `call [ebp+var_4]`), IDA does not know what function is being called. So, only the line `mov [ebp+var_4], offset sub_4011C0` gets recognized, and not the actual call. As a result, IDA shows only one cross-reference at:

```
004011D5 mov [ebp+var_4], offset sub_4011C0
```

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

2 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);
```

You can do the same in IDAPython with:

```
from ida_xref import add_code_xref
from ida_xref import XREF_USER
from ida_xref import fl_CF

add_code_xref(0x004011DE, 0x004011C0, fl_CF, XREF_USER)
add_code_xref(0x004011EA, 0x004011C0, fl_CF, XREF_USER)
```

3 Return Pointer Abuse

Return Pointer Abuse is a control flow manipulation technique that exploits the behavior of the `retn` instruction in x86 architecture. While `call` and `jmp` are commonly used for transferring control, `retn` is typically used to return from a function. However, since `retn` simply performs a `pop` to retrieve an address from the stack and jumps to it, it can be abused to redirect execution flow to an arbitrary, attacker-controlled location.

The technique often begins with a `call $+5` instruction, which pushes the address of the next instruction (i.e., the return address) onto the stack. This address is then modified using an instruction like `add [esp], x`, changing it to point to a new target. Finally, a `retn` instruction is executed, which pops the altered address and jumps to it. The following is an illustrative example:

```
004011C0 call $+5           ; Push 0x004011C5 onto the stack
004011C5 add [esp], 5       ; Modify it to 0x004011CA
004011C9 retn               ; Jump to 0x004011CA

004011CA push ebp           ; The actual function starts here
004011CB mov ebp, esp
004011CD mov eax, [ebp+8]
004011D0 imul eax, 2Ah
004011D3 mov esp, ebp
004011D5 pop ebp
004011D6 retn
```

In this example, the `retn` instruction is not used to return from a legitimate function call. Instead, it is abused to act like an indirect jump to an address that has been dynamically calculated on the stack. Disassemblers such as IDA Pro often fail to recognize this redirection. Since there is no explicit `jmp` or `call` to the real function at `0x004011CA`, IDA will not mark that address as the start of a new function. Moreover, the presence of the rogue `retn` breaks function boundary detection, leading to failed stack analysis and missing argument annotations.

This technique is commonly used for obfuscation and anti-disassembly purposes. Its benefits include:

- Hiding true control flow from static analysis tools.
- Avoiding cross-references in disassemblers.
- Terminating disassembly or function recognition prematurely.

To recover the correct function in IDA, the analyst must manually redefine function boundaries (e.g., using `ALT+P`) and optionally NOP-out the misleading instructions at the top.

In summary, **Return Pointer Abuse** is a simple yet powerful technique for covert control flow redirection, useful in obfuscation, malware, or reverse engineering challenges. By leveraging the return mechanism of the CPU, it effectively conceals execution paths and disrupts static analysis.

4 Misusing Structured Exception Handlers (SEH)

Structured Exception Handling (SEH) is a Windows-specific mechanism for handling runtime errors such as invalid memory access or divide-by-zero. While designed to support high-level languages like C++ and Ada in managing exceptions, SEH can also be abused for obfuscating control flow and evading static analysis by disassemblers such as IDA Pro.

SEH Chain Structure

Every thread in Windows maintains a chain of exception handlers stored in memory as a singly linked list. This list is accessed through the `fs:[0]` segment register, which points to the first exception record on the stack. Each record is represented by a structure:

```
struct _EXCEPTION_REGISTRATION {
    DWORD prev;     // Pointer to the previous handler
    DWORD handler;  // Pointer to the exception handler function
};
```

These records are added and removed as functions are entered or exited, and thus reside on the stack. The list operates in a LIFO (last-in, first-out) fashion.

Installing a Custom Handler

To hijack control flow, one can manually install a fake SEH handler by constructing a record on the stack using two `push` instructions:

```
push ExceptionHandler     ; push the address of the custom handler
push fs:[0]               ; push the previous SEH record
mov fs:[0], esp           ; update fs:[0] to point to the new record
```

Note that the `handler` is pushed before `prev` to preserve correct memory layout, because the stack grows downward.

Triggering an Exception

Once the handler is installed, an artificial exception is triggered to divert execution to the custom handler. A common method is to use a divide-by-zero:

```
xor ecx, ecx
div ecx                  ; triggers exception → jumps to ExceptionHandler
```

Restoring Stack and Unlinking the Handler

When the custom handler is invoked, the OS may have inserted its own internal SEH handler as well. Therefore, to cleanly restore the state, the handler must:

- Restore the original stack pointer
- Remove both the OS handler and the attacker's handler from the SEH chain

This is typically achieved with the following sequence:

```
mov esp, [esp+8]         ; restore original ESP before exception
mov eax, fs:[0]          ; eax = OS-injected handler
mov eax, [eax]           ; eax = attacker's handler
mov eax, [eax]           ; eax = original handler
mov fs:[0], eax          ; restore fs:[0] to original state
add esp, 8               ; clean up the stack
```

The double `mov eax, [eax]` steps are necessary because the OS temporarily installs a wrapper handler before calling the attacker's handler. Skipping one of them would leave the attacker's or OS's handler in place, leading to inconsistencies or crashes.

Example of Control Flow Obfuscation

Consider the following obfuscated code fragment extracted from a Visual C++ binary:

```
00401050 mov eax, offset loc_40106B + 1
00401055 add eax, 14h                ; eax = address 0x401080 (handler)
00401058 push eax
00401059 push fs:[0]
00401060 mov fs:[0], esp
00401067 xor ecx, ecx
00401069 div ecx                     ; trigger exception
```

This code installs a fake SEH handler and triggers a fault. The real handler code begins at `0x401080`, which is not referenced anywhere, making it invisible to disassemblers:

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

Conclusion

By exploiting the SEH mechanism, attackers can implement hidden control flow transfers that confuse both static analysis tools and runtime debuggers. This technique not only bypasses traditional control-flow references like `jmp` or `call`, but also seamlessly returns to normal execution after the exception, leaving little trace. Understanding this technique is critical for reverse engineers and malware analysts to correctly interpret disassembled binaries and detect obfuscation.

5 Thwarting Stack-Frame Analysis

Modern disassemblers such as IDA Pro attempt to reconstruct a function's stack frame by analyzing instructions that manipulate the stack pointer. This allows them to infer local variables, function arguments, and produce high-level pseudocode through tools like the Hex-Rays Decompiler. For malware analysts, this information is extremely valuable, as it provides insights into the function's logic and interface.

However, stack-frame analysis is inherently heuristic. Disassemblers must make educated guesses about how the stack is constructed and used. These

guesses can be deliberately manipulated and misled by carefully crafted assembly instructions.

Anti-Analysis Technique

The following function demonstrates a method to confuse stack analysis by injecting misleading stack pointer manipulations and conditional branches:

```
00401543 sub esp, 8
00401546 sub esp, 4
00401549 cmp esp, 1000h
0040154F jl short loc_401556
00401551 add esp, 4
00401554 jmp short loc_40155C

00401556 loc_401556:
00401556 add esp, 104h

0040155C loc_40155C:
0040155C mov [esp-0F8h+arg_F8], 1E61h
00401564 lea eax, [esp-0F8h+arg_F8]
...
0040157A add esp, 8
0040157D retn
```

This function pretends to perform normal stack allocation using `sub esp`, but then introduces a conditional check:

- It compares `esp` against `0x1000`.
- If less than, it executes `add esp, 0x104`.
- Otherwise, it executes `add esp, 4` and continues.

From the disassembler's point of view, both branches are valid, and it must track both possible outcomes. However, this comparison is deceptive. In reality, the ESP register is never less than `0x1000`, since the lower memory pages are typically unmapped and protected on Windows. Therefore, the "less than" branch is never taken during actual execution. But IDA Pro cannot know this; it blindly analyzes both paths and assumes the worst-case stack growth.

Impact on Disassembly

This confusion results in:

- Incorrect stack pointer offsets in the display (e.g., negative offsets like `esp-0F8h`).
- Incorrect assumptions about the number of arguments (e.g., IDA may infer 62 arguments).
- Malformed function prototypes in the decompiler.

- Broken stack variable tracking and corrupted pseudocode output in Hex-Rays.

Pressing `Y` in IDA to define a function prototype will often yield bizarre results due to the broken stack assumptions.

Why It Works

This technique leverages the disassembler's inability to reason about:

- Real-world constraints on memory layout (e.g., ESP cannot be less than `0x1000`).
- Constant condition evaluations (e.g., CMP followed by always-false Jcc).
- Purposeful stack misalignment that eventually cancels itself out (e.g., `sub esp; add esp` pairs).

In practice, this causes IDA to misalign the function's stack frame and propagate that confusion through the rest of the analysis.

Remediation and Manual Recovery

Analysts encountering this situation can:

- Use `ALT+K` in IDA to manually adjust the ESP offset at specific instructions.
- NOP out misleading stack manipulation instructions and reanalyze the function.
- Manually clean up the decompiler's variable list by removing incorrect arguments or stack slots.
- Use `ALT+P` to redefine function boundaries and limit propagation of stack misinterpretation.

Conclusion

This anti-analysis method is simple yet highly effective. By crafting misleading stack pointer operations and opaque control flow, malware authors can break static stack-frame reconstruction. As a result, reverse engineers are forced to fall back on manual analysis, increasing the difficulty and time required to understand a function's true behavior.