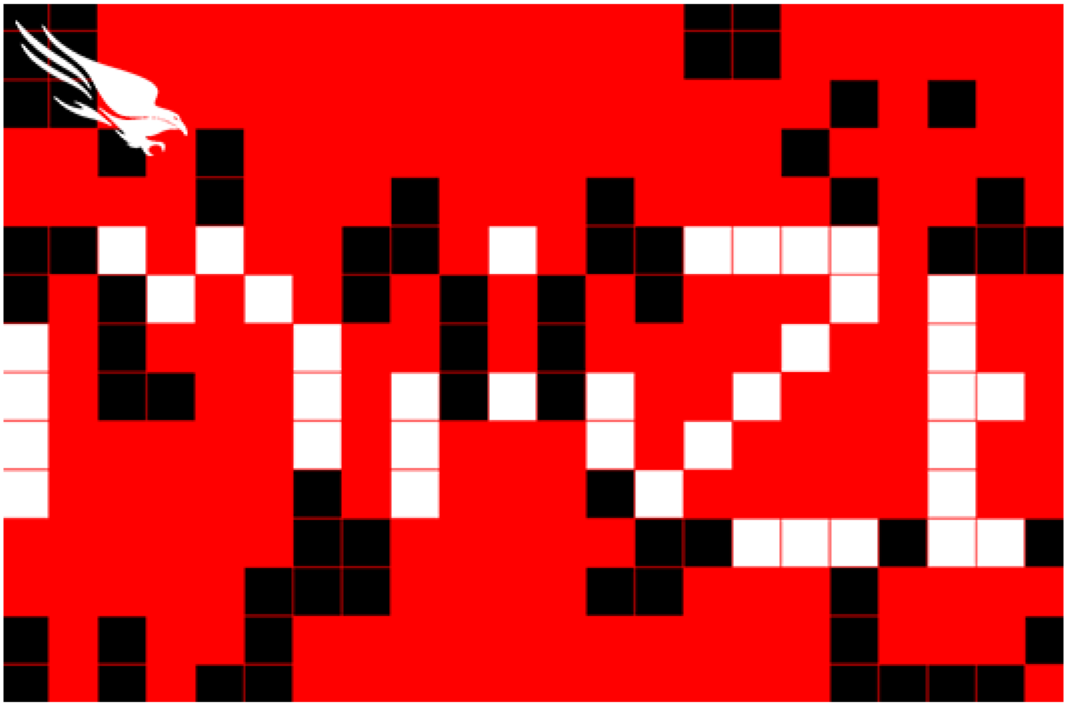
The Many Paths Through Maze

crowdstrike.com/blog/maze-ransomware-deobfuscation/

Shaun Hurley May 1, 2020



Maze ransomware is a recent addition to the ever-growing list of ransomware families. It stands out from the others by leveraging a technique called control flow obfuscation to make static and dynamic analysis difficult for anyone attempting to reverse engineer the binary. Maze lives up to its name — anyone doing analysis will get lost through the endless amount of paths that can be taken through the code. But automated deobfuscation of Maze is not a terribly difficult task.

Code obfuscations are legitimate techniques used to protect code from analysis by reverse engineers. The most common form of code obfuscation that a malware analyst will run into daily is a packed binary. Various forms of anti-disassembly and control flow obfuscations are next in line. Deobfuscation of these obfuscations range from trivial to bashing your head on the keyboard and hoping for a miracle. For more information on code obfuscation, read *Surreptitious Software* by Christian Collberg and Jasvir Nagra and *Practical Binary Analysis* by Dennis Andriesse.

The primary purpose of this blog post is to present an approach to attack and deobfuscate the various obfuscations leveraged by Maze's author. The primary tools leveraged are the IDA Pro disassembler and Python. We cover some foundation material, the obfuscations, the deobfuscated obfuscations, and how to accomplish deobfuscation using IDA's Python API.

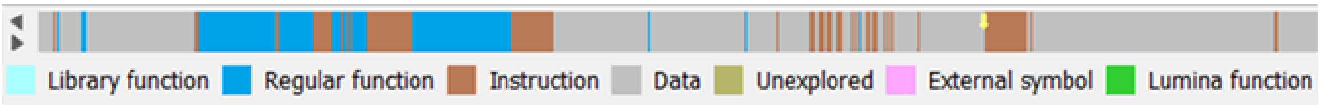


Figure 1. IDA’s Navigation Bar Depicting Maze Ransomware

Once any binary is opened up in IDA, the analyst should take a look at the navigation bar. This bar is a quick method to determine how difficult a binary may be to reverse engineer. The bar depicted in Figure 1 is of Maze. The vast majority of the gray and brown areas are legitimate code, junk code and undefined functions. The fact that there is so little blue (regular function code) lets an analyst know that they are about to have a rough day.

The approach taken in this blog post is blunt, effective and approachable by inexperienced reverse engineers. The primary method for identifying the obfuscations is to search for byte patterns and then deobfuscate all located patterns. The following section, “Understanding Program Control Flow,” covers the basic technical details to pave the way for the rest of the blog.

Understanding Program Control Flow

A program’s control flow is a path created out of the instructions that can be executed by the program. Disassemblers, like IDA, visualize control flow as a graph by creating a series of connected blocks (called “basic blocks”). Each basic block has a single implicit entry point and a single implicit exit point. The entry points for a basic block are reached by jump instructions, by call instructions or through the binary’s code entry point specified by the format (PE, ELF, etc). This section can be skipped if the reader already understands program control flow.

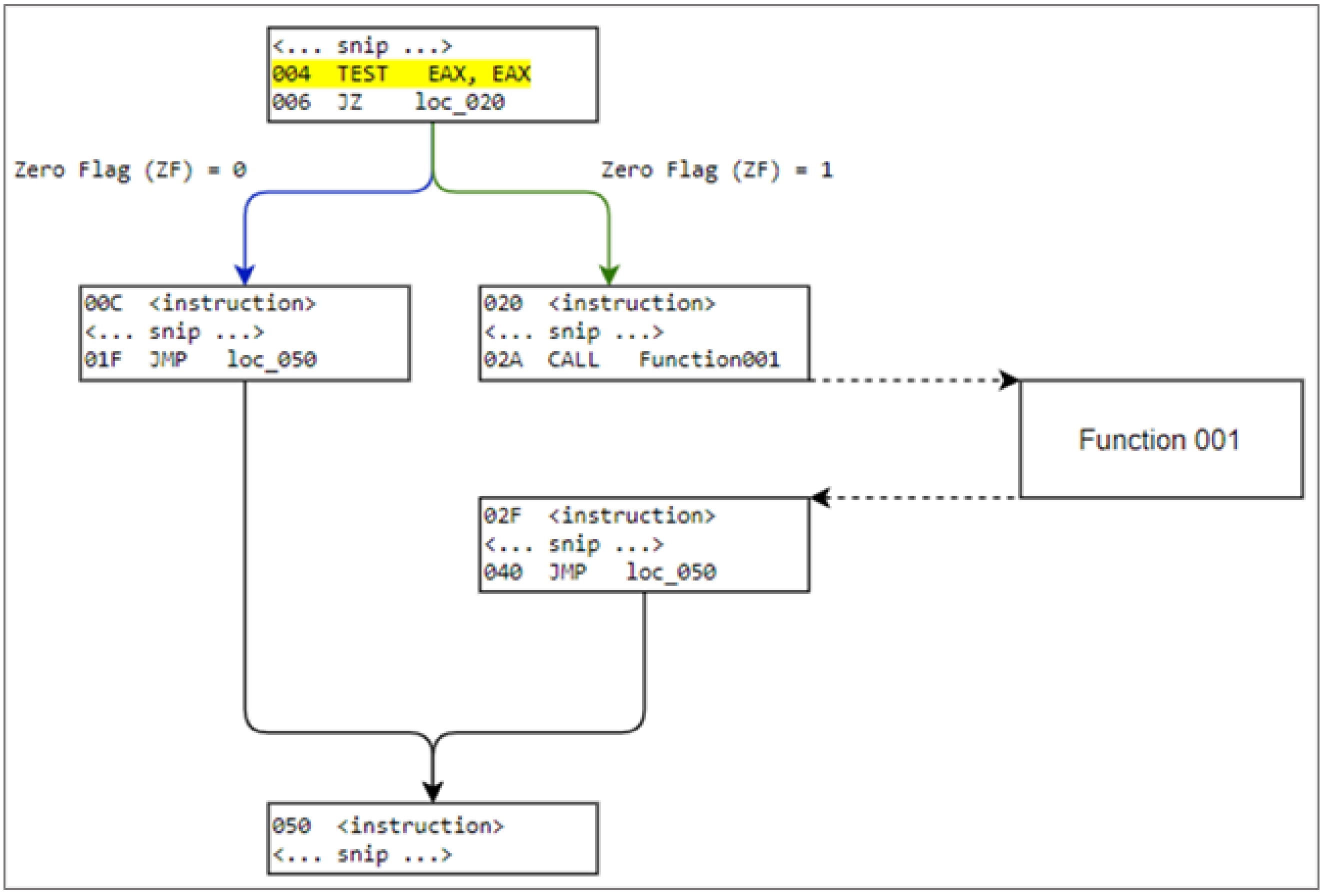


Figure 2. Standard Control Flow Example

Figure 2 is an example of visualizing a program's control flow. To start, take the highlighted TEST instruction at address `004`. A TEST instruction will perform a bitwise AND operation on the two operands (EAX, EAX). If the result of the bitwise AND is zero, then the CPU's Zero Flag (ZF) is set to one; otherwise, the ZF is set to zero. The ZF is used by the JZ ("jump if zero") instruction at address `006` to determine which path (left or right) to take. This is one example of something called a "conditional jump." If the ZF is set to one, then the jump is taken (right path), and instruction `020` will be executed next. Otherwise, the fall-through side (left path) of the conditional branch is taken, and the instruction at address `00C` is executed.

There are two more instructions that need to be discussed: CALL and JMP. The CALL instruction at `02A` will jump to the address of `Function001`, execute the body of the function and return the address following the call instruction (`02F`). This is explained in more detail in the section on function calls. A JMP instruction (`01F`, `040`) is called an "absolute jump" — the program will always jump to the specified target address.

Absolute Jumps, Conditional Jumps and Opcodes

This section is a quick overview of opcodes, jumps and conditional jumps. Before diving into the different types of jumps, know that all assembly instructions are human-readable representations of binary data. This binary data is put together into strings of bytes called “opcode bytes.” When patching code for binaries is discussed, what’s being patched are the opcode bytes and not an operation on the readable assembly instruction.

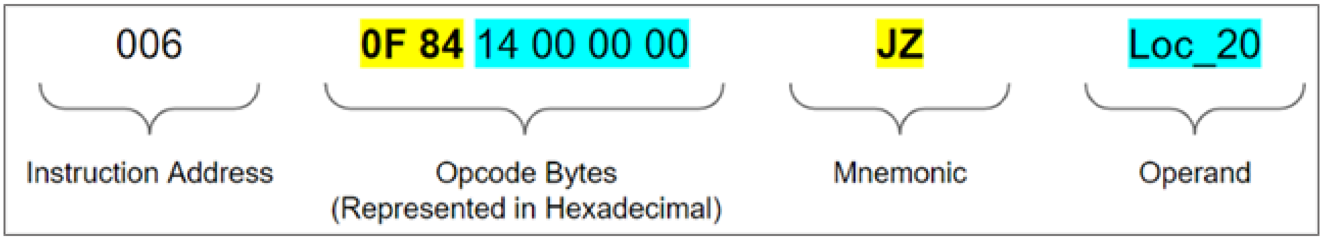


Figure 3. JZ Breakdown

Let’s take the JZ from Figure 2, break it down and explain each component at address `006`. This is done in Figure 3. The first component is the instruction address, which indicates the location in process memory where an instruction exists. The opcode bytes are the machine code that represent the mnemonic and the operand. The mnemonic is the human readable representation of the operation going to be performed on the operand.

In this example, the kind of conditional jump being performed here is a relative jump if zero. This can be determined based on the opcode bytes, `0F 84` (Figure 3). To calculate the jump target address for a relative jump, the last four bytes of the opcode are added to the address of the next instruction (`00C`). In this case, the value of the JZ operand is the four bytes following 0F `84` which are: `14 00 00 00`. The operand value is little endian and will be reversed when the instruction is executed, that is `14 00 00 00` is actually `00 00 00 14` or `14h`. So, the calculation to get a jump target address of `020` is `14h + 0Ch`, which adds up to `020h`.

Opcode (in hexadecimal)	**Mnemonic**	**Operand Value Size**	**Description**
EB *XX*	JMP	1 Byte	Short relative jump
E9 *XX XX XX XX*	JMP	2 or 4 Bytes	Near relative jump
75 *XX*	JNZ	1 Byte	Short relative jump if ZF = 0
0F 85 *XX XX XX XX*	JNZ	2 or 4 Bytes	Near relative jump if ZF = 0
74 *XX*	JZ	1 Byte	Short relative jump if ZF = 1
0F 84 *XX XX XX XX*	JZ	2 or 4 Bytes	Near relative jump if ZF = 1

Table 1. Absolute and Conditional Jump Instructions

A core part of deobfuscating Maze involves modifying the various jump instructions to remove control flow obfuscation. Table 1 describes the opcode, mnemonic and operand value size for each type of jump instruction that will be discussed.

How Function Calls Work

There are many different types of calling conventions for functions, but this section provides a general explanation into how function calls work. The primary takeaway from this section should be that a prologue sets up a function, an epilogue tears down a function, and a return address is where execution resumes when a `RETN` instruction is executed.

Every function starts by executing a series of instructions designed to allocate space that will contain the memory necessary for the function to complete its purpose. This is called a "function prologue." The allocated space is called a "stack frame," and a function needs it to store and reference arguments, variables, and the return address. Two registers are used when referencing the stack: a stack pointer and a base pointer. On an x86 system, the stack and base pointers are called ESP and EBP, respectively.

The stack pointer (ESP) always points to the top of the stack. That means the value of ESP will change when values are added or removed from the stack. This differs from EBP (the base pointer), which remains the same throughout the lifetime of the function. By not being modified, EBP can be used as a reference by the program's code to know where local variables (EBP-4), arguments (EBP+8) and the return address (EBP+4) are located on the stack.

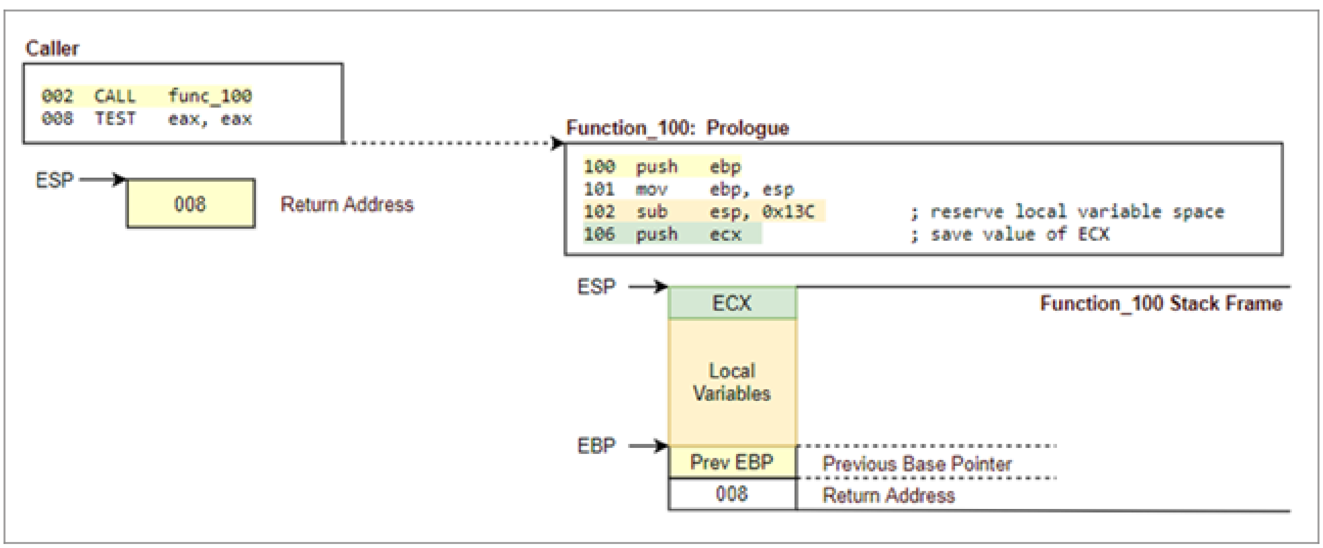


Figure 4. Function Prologue for Function_100

For example, in Figure 4, the Caller function (on the left) executes the CALL instruction at address `002`. The CALL pushes the return address, `008`, to the stack and shifts ESP so that it points to the top of the stack. On the right is the prologue for Function_100 that will set up the function's stack frame. Starting at address `100`, the base pointer for Caller (current value of EBP) is saved to the stack. The base pointer for `Function_100` is set to ESP. At address `102` , `13Ch` bytes are reserved for use by `Function_100` 's local variables. Finally, before anything else is done, the value stored in ECX is saved to the stack. This value will be restored in the function's epilogue.

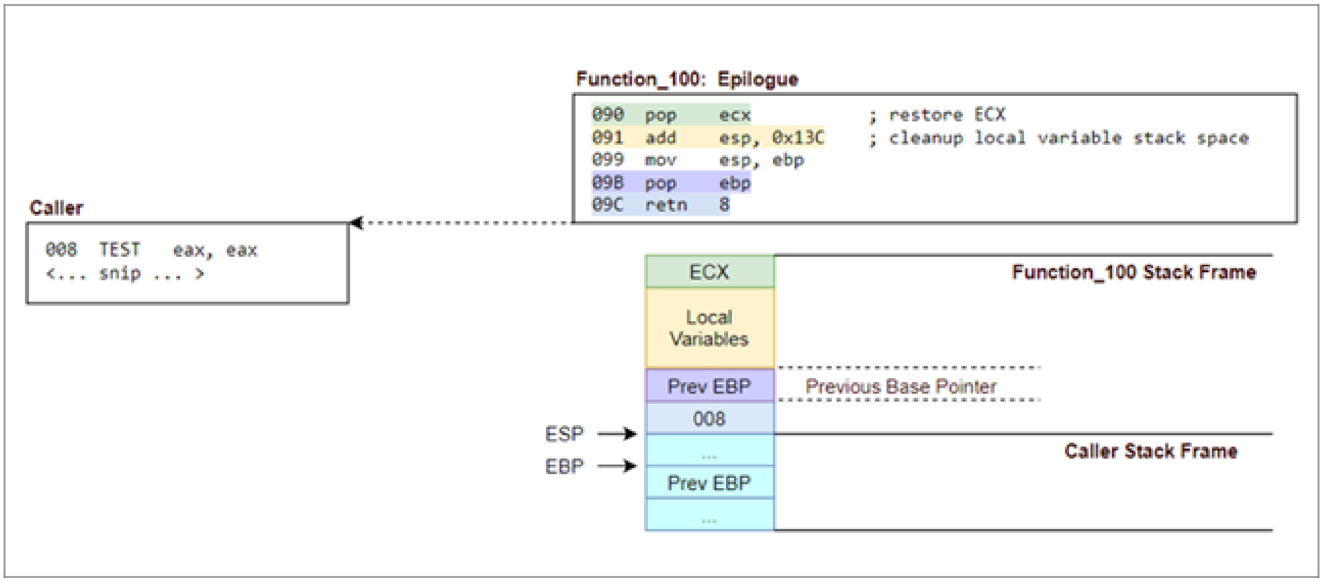


Figure 5. Function Epilogue for Function_100

The epilogue in Figure 5 undoes everything that was done by the prologue. The original value of ECX is restored, local variable stack space is removed, both ESP and EBP are restored to their original values, and the return instruction is executed. After the pop ebp instruction (09B), ESP points to the stack space where the return address 008 is stored. The RETN instruction at 09C will return to whatever value is pointed to by ESP, in this case 008. After the RETN instruction is executed, control flow returns to 008, and ESP will point to the top of the stack for Caller.

The information covered in this section can be a bit tricky to follow at first. However, a complete understanding is not necessary to follow along.

Absolute Jump Obfuscation

A simple obfuscation is to insert extraneous control flow, and Maze does this quite a bit. On the left side of Figure 6 is an absolute JMP instruction located at `004` . This will jump to location `020` . However, on the right side, a series of conditional jumps is used to ultimately end up at the same destination, loc_020. Walking through it, if the JZ instruction is followed, `loc_020` is reached. If it is not followed, however, the JNZ instruction will be followed, and the program jumps t0 `loc_010` . At `010` is another JNZ that will jump to `loc_020` . And, if the JNZ at `008` was followed because the zero flag is set to zero, then the JNZ instruction at `010` is also going to be followed because the ZF is going to be the same. Therefore, the program will always reach `loc_020` .

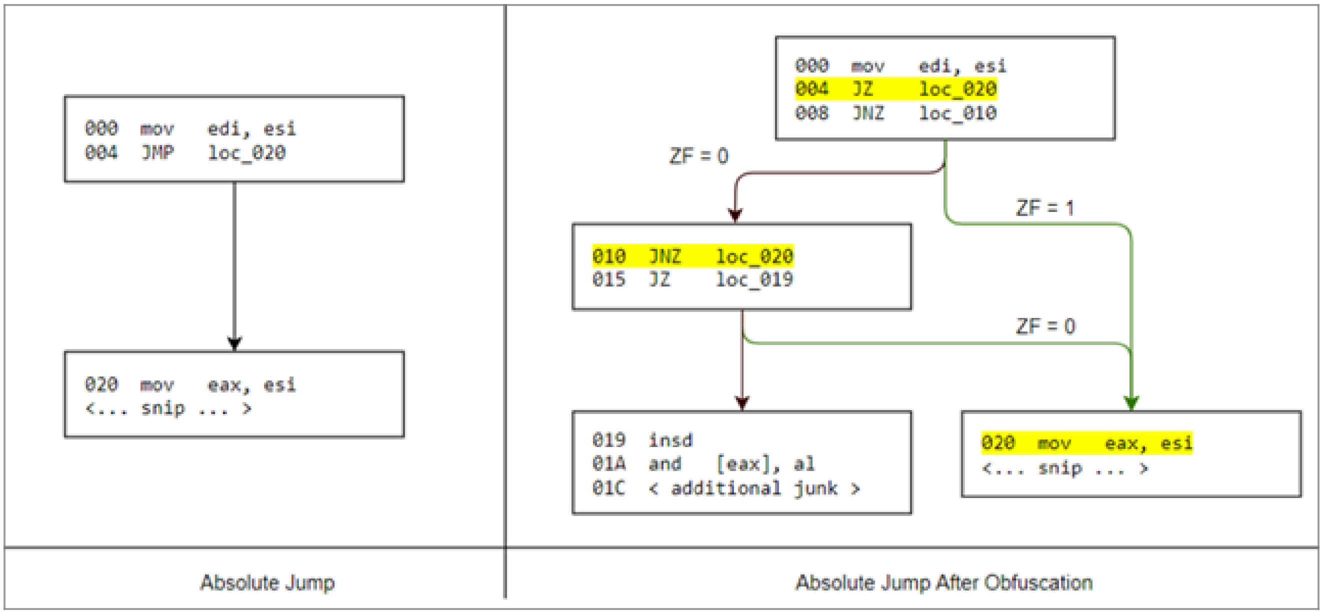


Figure 6: Absolute Jump Obfuscation

The fall-through branch for the JNZ instruction at `010` is the address `015`. This JZ instruction will never be reached because the JNZ will always jump. Conditional branches that will never be executed are called “opaque predicates.” As IDA starts to identify what is and isn’t code, it reaches these opaque predicates and displays them in the GUI. IDA expects the JZ instruction at `010` to have two branches: the code following `015` and the code located at address `019`. Although none of this will be reached, all of it is displayed as code in IDA, creating a cluttered mess. The analyst has to sift through all of this and decide which instructions are pertinent.

The following subsections describe the types of conditional jump patterns and how they can be deobfuscated.

JMP Type One: Absolute, Conditional Jump

This obfuscation takes an absolute jump and transforms it into a conditional jump that will always end up at one of two jump target addresses. As can be seen in Figures 7, 8 and 9, this obfuscation places the jump JZ and JNZ instructions in sequence. As mentioned previously, each conditional jump has a fall-through branch and a jump target. In the case of Figure 7, if the fall-through branch of the JZ instruction is executed (address `000`), that means ZF is set to zero. The next instruction, JNZ, is executed and the jump target will always be taken (`loc_004+4`). This means that the address at `004` will never be reached.

```
000: 74 2F              jz      short loc_6E4315FE
002: 75 04              jnz     short near ptr loc_004+4
004: A0 01 00 00 57     mov     al, ds:57000001h
009: 56                 push    esi
```

Figure 7. Short JMP Type One Obfuscation, Incorrect JNZ Target Label in IDA

IDA, however, expects the JNZ instruction at `002` to fall through if ZF is set to one. That means the bytes located at `004` should be marked and displayed as valid x86 code. The consequence of displaying the binary data at 004 as code is that IDA ends up incorrectly displaying both the operand (`loc_004+4`) for `002`, and the code at jump target address `008`. It is jumping to the last byte of the instruction at `004`. Confusing the disassembler in this manner is called "anti-disassembly." In IDA, this can be resolved by undefining the instruction at address `004` and marking the instruction at `008` as code (Figure 8). Throughout the rest of this post, each example will display the corrected code.

```
000: 74 2F                  jz      short loc_6E4315FE
002: 75 04                  jnz     short loc_008
; ---------------------------------------------------------------
004: A0 01 00 00            dd 1A0h
; ---------------------------------------------------------------
008: 57                     push    edi
009: 56                     push    esi
```

Figure 8. Short JMP Type One Obfuscation, Correct JNZ Target Label in IDA

Figure 9 is the same thing as Figure 7, but the instruction at `000` is a near (rather than a short) JZ. As expected, the bytecode starts with `0F 84` and the entire instruction length goes from two to six.

```
000: 0F 84 7E 02 00 00          jz      loc_6E431A7D
006: 75 0A                      jnz     short loc_6E43180B
008: FF 15 08 80 46 6E          call    ds:LsaFreeMemory
00E: A4                         movsb
00F: 02 00                      add     al, [eax]
```

Figure 9. Near JMP Type One Obfuscation

Deobfuscation

If the instruction at 000 is not taken, then the instruction at 002 will always be taken. That means that the JNZ instruction at 002 is actually a short JMP. The transformation here is simple — the JNZ instruction at address 002 (Figure 8) can be changed to a short JMP instruction by modifying the first byte from 75 to EB.

```
000: 74 2F                jz      short loc_6E4315FE
002: EB 04                jmp     short loc_008
; ------------------------------------------------------------
004: A0 01 00 00          dd 1A0h
; ------------------------------------------------------------
008: 57                   push    edi
009: 56                   push    esi
```

Figure 10. Short JMP Type One Obfuscation, Correct JNZ Target Label in IDA

JMP Type Two: Absolute, Multiple Conditional Short Jump

This obfuscation is similar to the JMP Type One obfuscation but with added JZ/JNZ blocks to further confuse IDA's ability to visualize control flow. In Figure 11, either the JZ instruction at `000` will be taken, or the JNZ instruction at 002 will be executed. The JNZ instruction at `002` jumps to the JNZ target at 008. A JNZ jumping to another JNZ is extraneous because the second JNZ jump will always be taken. In this example, one of two jump targets will always be reached: `loc_6E456049` or `loc_6E456049`.

```
000: 74 55                        jz      short loc_6E456049
002: 75 04                        jnz     short loc_008
004: 0E                           push    cs
005: 02 00                        add     al, [eax]
;------------------------------------------------------------
007: 00                           db    0
;------------------------------------------------------------
008: 75 0A                        jnz     short loc_6E456006
00A: 74 04                        jz      short loc_6E456002
00C: DB 1A                        fistp   dword ptr [edx]
```

Figure 11. Short JMP Type One Obfuscation, Correct JNZ Target Label in IDA

The instructions at `004`, `00A`, `00C` and `loc_6E456002` are all unreachable. This technique multiplies the number of paths through Maze that IDA believes are accessible. Each one is a dead end.

Like the JMP Type One obfuscation, the Type Two obfuscation has a short and a near version.

Deobfuscation

The deobfuscation here is similar to the JMP Type One obfuscation. The JNZ instruction at 002 (Figure 12) is changed to a short JMP by changing the first opcode byte `75` to `EB`. The JNZ instruction at `000` is not touched. The JNZ/JZ block at `008` is zeroed out.

```
000: 74 55                          jz      short loc_6E456049
002: EB 0E                          jmp     short loc_6E456006
; ---------------------------------------
004: 0E 02 00 00                    dw 1009h
008: 00 00 00 00                    dd      0
; ---------------------------------------
```

Figure 12. Near JMP Type Two Obfuscation, Correct JNZ Target Label in IDA

Call Instruction Obfuscation

Similar to the previous obfuscation types, Maze transforms CALL instructions into conditional jumps. Not only will this confuse control flow, it combines multiple functions into a single function. This process is called “function inlining,” and it makes it difficult for IDA to properly identify where functions begin and end. Figure 13 illustrates the obfuscation process.

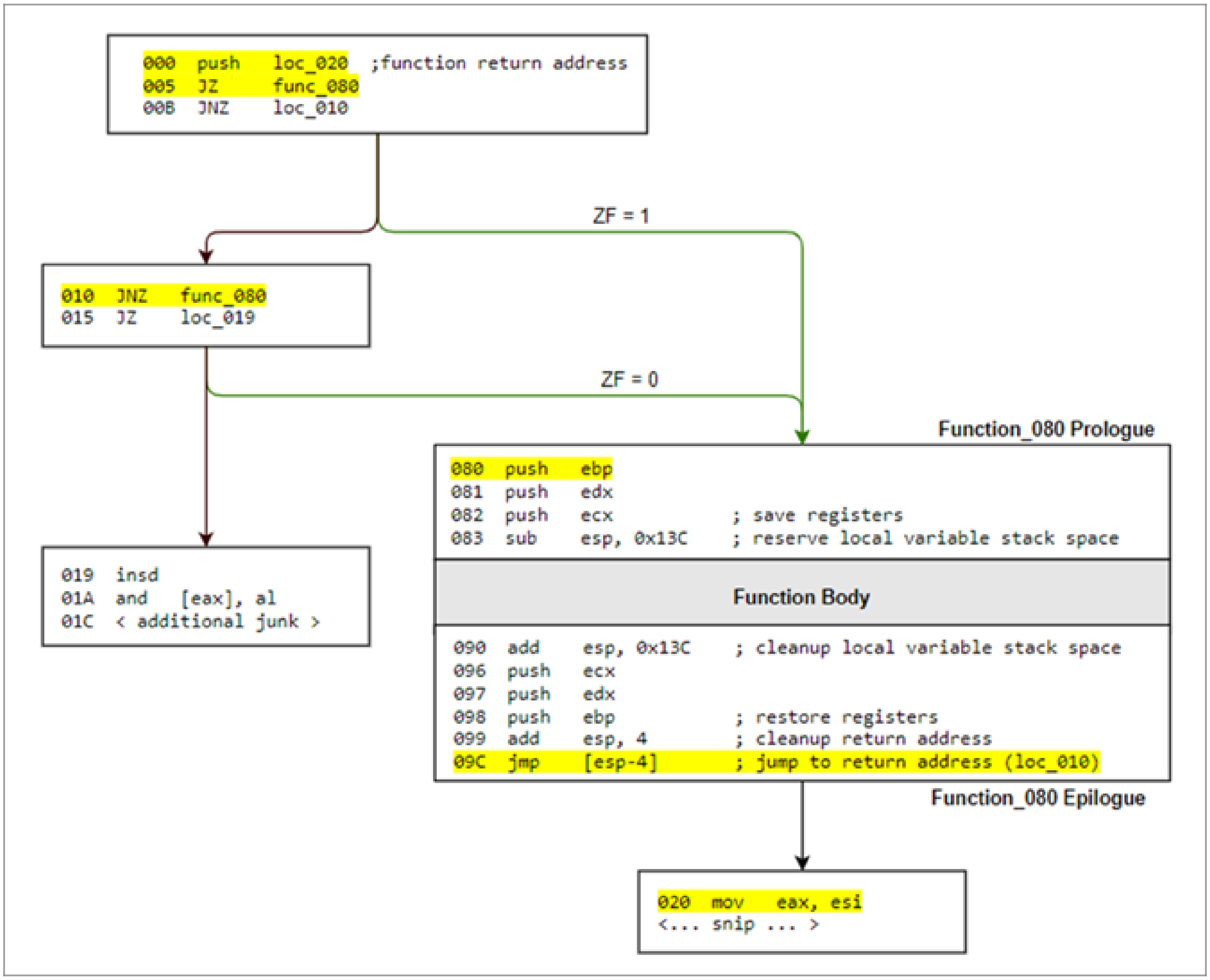


Figure 13. Obfuscated Function Call

Each function follows a similar template. At instruction `000` in Figure 13, the return address, `loc_020`, for the function call is pushed to the stack. The next instruction jumps to the address of the function being called. The function prologue instructions preserve the state of registers by pushing them on the stack and to allocate memory for local variables. After the prologue, the function body executes, and then the epilogue restores the state of the stack prior to the function being called. Once the state of the stack has been restored, control flow resumes to the address that was pushed at instruction `000`.

There are multiple minor variations to the template, but the core part of the control flow remains intact. The following series of sections breaks down each variation of the control flow obfuscation into numbered types.

Call Type One: Indirect Absolute Jump

This obfuscation pushes the return address to the stack and then executes an indirect jump. An indirect jump is a jump instruction where the operand of the JMP instruction is a register rather than the target address or the offset to a target address. The jump target address is stored in the register. In Figure 14, at address `005`, the JMP instruction operand is the EDI register. To reach the destination of the JMP, the CPU needs to get the address from EDI. This obfuscation breaks control flow, because IDA does not know where to jump and where to return.

```
000: 68 E8 52 44 6E      push    offset loc_6E4452E8
005: FF E7               jmp     edi                  ;VariantClear
```

Figure 14. Call Type One Obfuscation

Indirect jumps are a common instruction, and that on its own does not constitute control flow obfuscation. It's the PUSH instruction followed by `jmp edi` that makes this a Call Type One obfuscation. In Maze, these obfuscations are often used to execute a Windows procedure (`VariantClear`, in this example). When a Windows procedure is called, it is more or less always going to return to the instruction proceeding the call instruction. In this case, the return address for the function is pushed to the stack, and the JMP instruction is executed.

Deobfuscation

To deobfuscate, the indirect jump at `005` (Figure 14) gets changed to a call by subtracting `10h` from the second opcode byte (`E7`), the PUSH instruction at `000` gets changed to an absolute near jump, and the instructions are reordered so that the call instruction comes before the jump (Figure 15).

```
000: FF D7               call    edi
002: E9 24 00 00 00      jmp     loc_6E4452E8
```

Figure 15. Call Type One Deobfuscation

Call Type Two: Absolute, Conditional Jump

The primary difference between this obfuscation type and the JMP Type One obfuscation is the addition of pushing the return address at `001` (Figure 16). The first instruction at `000` pushes whatever value is stored to ESI as an argument to the function.

```
000: 56                    push    esi
001: 68 D1 21 43 6E        push    offset loc_6E4321D1
005: 0F 84 3D FF 01 00     jz      loc_6E4520E0
00a: 0F 85 37 FF 01 00     jnz     loc_6E4520E0
010: 23 25 00 00 65 19     and     esp, ds:19650000h
```

Figure 16. Call Type Two Obfuscation

Deobfuscation

This deobfuscation is a case where the deobfuscation instructions don't require as many opcode bytes as the original obfuscation instructions. There are a few different approaches to resolve this issue — in this case, no operation (NOP) instructions are used to overwrite the original bytes. After overwriting the unnecessary bytes, the JZ/JNZ instructions are replaced with a relative CALL instruction, and the return address pushed to the stack is used as the target for the absolute near jump at `00E`.

```
000: 56                       push    esi
001: 90                       nop
002: 90                       nop
003: 90                       nop
004: 90                       nop
005: 90                       nop
006: 90                       nop
007: 90                       nop
008: E8 3F FF 01 00           call    loc_6E4520E0
00E: E9 28 00 00 00           jmp     loc_6E4321D1
;-------------------------------------------------------------
013: 23                       db 23h
```

Figure 17. Call Type Two Obfuscation

A cleaner approach may be to move the CALL/JMP instruction sequence to address `001`, just after the PUSH instruction, and then zero out the unused bytes. This method would remove the extraneous NOP instructions from the basic block.

Call Type Three: Absolute, Multiple Conditional Jump

Once again, the primary difference between this obfuscation type and the JMP Type Two obfuscation is the addition of pushing the return address at `001` (Figure 16). The first instruction at `000` pushes whatever value is stored to EDI as an argument to the function.

```
000: 57                           push    edi
001: 68 D7 31 43 6E               push    offset loc_6E4331D7
006: 0F 84 B0 34 03 00            jz      sub_6E466630
00B: 75 0A                        jnz     short loc_017
;------------------------------------------------------------
00D: FF 15 E8 80                  dd 80E815FFh
011: 46 6E 68 21                  dd 21686E46h
015: 00 00                        dw 0
;------------------------------------------------------------
017: 0F 85 9E 34 03 00           jnz     sub_6E466630
01D: 74 04                       jz      short loc_6E433198
01F: 6A 0B                       push    0Bh
;------------------------------------------------------------
020: 00                          db    0
```

Figure 18. Call Type Three Obfuscation, Correct JNZ Target Label in IDA

The example in Figure 18 is only a double jump, but there are variations where three of these linked JNZ instructions ultimately end up in the same place. Deobfuscation has to be able to handle these cases.

Deobfuscation

The deobfuscation for this type is similar to the method used in Call Type Two.

```
000: 57                           push    edi
001: 90                           nop
002: E8 B5 34 03 00               call    sub_6E466630
007: E9 57 00 00 00               jmp     loc_6E4331D7
; ---------------------------------------------------------------
00C: FF 15 E8 80                  dd 80E815FFh
010: 46 6E 68 21                  dd 21686E46h
016: 00 00 00 00                  dd 0
01A: 00 00 00 00                  dd 0
; ---------------------------------------------------------------
```

Figure 19. Call Type Three Deobfuscation

Deobfuscation with IDA Python

IDA Python is built on the IDA SDK and this section will discuss how it can be leveraged to deobfuscate Maze. Deobfuscating the various control flow obfuscations discussed above gets redundant, so the only case covered is the Call Type One obfuscation. This example will be enough to follow along in both the source code and in the later sections.

Locating Call Type One Obfuscations

Prior to patching the IDA database (IDB) to remove the obfuscations, the locations for each obfuscation have to be identified. The simplest method for locating the obfuscations is by searching the IDB for all instances of a specific opcode pattern.

```
000: 68 E8 52 44 6E       push    offset loc_6E4452E8
005: FF E7                jmp     edi
```

Figure 20. Call Type One Obfuscation

In Figure 20, the opcode pattern for address `000` is `68 E8 52 44 6E`. This instruction is pushing the four-byte address `6E4452E8` (little endian byte order) onto the stack. Because the four bytes following the `68` opcode byte are going to change based on which address is being pushed, these will have to be excluded. This same issue pops up with the second opcode byte `E7` for the jump instruction at `005`. The search string ends up looking like: `68 ? ? ? ? FF`. The `?` is a wildcard match that will match any byte.

```
ctype1_opcodes = "68 ? ? ? ? FF"
ctype1_ea = ida_search.find_binary(0, end_ea, ctype1_opcodes, 0, SEARCH_DOWN
| SEARCH_CASE)

while ctype1_ea  !=  idc.BADADDR:
    #
    #  Iterate until the find_binary() function returns -1
    #

    <  code snip >

    ctype1_ea = ida_search.find_binary(ctype1_ea + 4, end_ea,
ctype1_opcodes, 0, SEARCH_DOWN | SEARCH_CASE)
```

Figure 21. Opcode Searching for Call Type One Obfuscation

The loop in Figure 21 will iterate over each address where the opcode pattern was found. The body of the loop (the snipped out code) will contain all of the logic used to verify that the found byte pattern is a Type One Obfuscation.

```
#
# Create an inst_t object to confirm that the pattern at this
#   address is a push  <address> instruction.
#
push_instr_ea = cfg1_ea
push_insn = ida_ua.insn_t()
ida_ua.decode_insn(push_insn, push_instr_ea)
push_insn_target_ea = GetInstuctionTargetAddress(push_insn)

if CheckValidTargettingInstr(push_insn, "push"):
```

Figure 22. Verify PUSH Instruction for Call Type One Obfuscation

Once an opcode pattern has been found, the instruction at that address needs to be interpreted. IDA may or may not have the correct instruction displayed here, but there is no way of knowing if that is the case. This blog post takes the approach of not trusting the current state of the IDB. So, an `ida_ua.insn_t` object is created and populated using the `ida_ua.decode_insn` (`insn_t` , `ea`) method. The reference for this instruction can be found in the online documentation. The key to success is using the online documentation along with searching the IDA Python source located in IDAs installation directory. In short, the `inst_t` object gives us access to the following information about the instruction:

- Instruction size
- Operand type
- Operand value
- Operand address

The two highlighted blue functions in Figure 22 are helper functions. The first retrieves the target address for the instruction (in this case, what is being pushed). The `CheckValidTargettingInstr()` function validates that the PUSH instructions operand is an `o_imm` type, `o_far` type or `o_near` type. After the type has been validated, the code address being pushed — `6E4452E8` (Figure 20) — needs to be verified that the address points to executable code in the Maze binary.

Once this has been confirmed, the JMP instruction located at `005` can be validated. The steps are mostly the same, but instead of validating the address, the operand type will be `o_reg` , `o_phrase` or `o_displ` .

Patching Call Type One Obfuscations

Once these instructions have been validated, the IDB can be patched to transform the obfuscation into the code in Figure 23. This is where things can get a bit frustrating. The goal is to get IDA to disassemble the code and present a cleaned-up version to whomever is analyzing the Maze binary. To ensure that this occurs, the deobfuscation script takes the steps discussed in this section.

```
000: FF D7                    call    edi
002: E9 24 00 00 00           jmp     loc_6E4452E8
```

Figure 23. Call Type One Deobfuscation

In order to accurately patch the program, the deobfuscation script needs to know the address for each of the obfuscated instructions, the address of the deobfuscated instructions, and the address that is going to be the jump target. The first item is the address of the byte pattern match from the previous section.

```
#
#   Get Obfuscated PUSH instruction data
#
obf_push_insn = ida_ua.insn_t()
ida_ua.decode_insn(obf_push_insn, obf_push_instr_addr)

#
#   Get Obfuscated JMP instruction data (JMP EDI)
#
obf_jmp_insn_addr = obf_push_instr_addr + obf_push_insn.size
obf_jmp_insn = ida_ua.insn_t()
ida_ua.decode_insn(jmp_insn, jmp_insn_addr)

#
#   Get address for patched JMP and CALL instructions
#
deobf_patch_jmp_addr = obf_push_instr_addr + obf_jmp_insn.size
deobf_patch_call_addr = obf_push_instr_addr
```

Figure 24. Getting the Deobfuscation Instruction Addresses

For all cases, the `call edi` instruction is going to overwrite the `push offset loc_6E4452E8` (Figure 20) instruction. In Figure 24, this is accomplished by the following code: `deobf_patch_call_addr = obf_push_instr_ea.`

Now that the location of `call edi` has been determined (`000`), the address for the `jmp loc_6E4452E8` instruction needs to be calculated. Figure 23 shows that the JMP instruction will come after the two-byte CALL instruction, so the address is `002` . The question is, will that always be the case?

```
000: 68 E8 52 44 6E           push    offset loc_6E4452E8
005: FF 62 04                 jmp     dword ptr [edx+4]
```

Figure 25. Call Type One Obfuscation

Figure 25 shows that the length of the CALL will not always be two. However, calculating the correct address is not difficult. Remember that the length of the deobfuscated CALL instruction (`call edi`) will always be the same length as the obfuscated jump (`jmp edi`). The address of the obfuscated jump is already known, so the instruction can be decoded into an `ida_ua.isn_t` object. Once the instruction has been decoded, the address of the deobfuscated jump (`jmp loc_6E4452E8`) can be calculated: `deobf_patch_jmp_addr = obf_push_instr_addr + obf_jmp_insn.size` .

Now that the address of the deobfuscated jmp instruction is known, the JMP target address can be calculated. Take a look at the JMP instruction at `003` in Figure 23. The `E9` opcode indicates that this is going to be a relative near jump. The four opcode bytes after the `E9` byte are going to be the offset between the address of the next instruction and the return address pushed by the PUSH instruction.

```
#
#   Calculate the offset for the address that was pushed to the
#    stack by the push instruction and will now be used by the JMP
#
#       offset = target_address - address_of_insn_after_JMP
#
deobf_patch_jmp_target_addr = obf_push_insn_target_addr

#
#  Get address of the instruction after the JMP
#
next_insn_address = deobf_patch_jmp_ea + 5

#
#  Get offset
#
deobf_patch_jmp_dest_offset = (deobf_patch_jmp_target_addr -
next_insn_address) & 0xFFFFFFFF
```

Figure 26. Getting the JMP Target Address

The address of the next instruction is always going to be the address of the deobfuscated JMP instruction + five. This is known because a relative near JMP instruction has a size of five bytes, and that is what will always be used in the Call Type One deobfuscation code. The highlighted sections in Figure 26 show how the offset is calculated. The IDB can now be patched.

```
idx = 0
del_items(deobf_patch_call_ea,1)
for idx in range(obf_jmp_insn.size):
    #
    # Take each byte of the obfuscated JMP instruction (JMP EDI)
    #  and write that byte to the location of the deobfuscated
    #  call instruction.
    #

    byte = get_wide_byte(obf_jmp_insn_ea+idx)
    if idx == 1:

        #
        #   Convert the JMP into a CALL instruction by
        #    subtracting 16
        #
        patch_byte(deobf_patch_call_ea+idx, byte-0x10)
    else:
        patch_byte(deobf_patch_call_ea+idx, byte)
    idx += 1
create_insn(deobf_patch_call_ea)
```

Figure 27. Writing the Deobfuscated CALL Instruction

We start with the deobfuscated CALL instruction (Figure 27). First, the original PUSH instruction is deleted using the IDA Python `del_items` (`insn_addr, FLAG`)procedure. Next, the “for” loop walks over each byte of the obfuscated indirect JMP instruction (`jmp edi`). It writes the first byte to the address of the CALL instruction, subtracts 16 from the second byte to convert the JMP to a CALL, then writes the rest of the instruction. After the CALL instruction has been written, it is created using the `create_insn` (`insn_addr`) procedure.

```
#
#   Create JMP from PUSH instruction
#
CleanupPatchArea(deobf_patch_jmp_ea,5)
patch_byte(deobf_patch_jmp_ea, 0xE9)
patch_dword(deobf_patch_jmp_ea+1,deobf_patch_jmp_dest_offset)
create_insn(deobf_patch_jmp_ea)
plan_and_wait(deobf_patch_jmp_ea,deobf_patch_jmp_ea+5)
```

Figure 28. Writing the Deobfuscated Relative Near JMP Instruction

In Figure 28, the instruction that exists at this location is undefined using the `CleanupPatchArea` (`addr, size`) function (see source code). The first opcode byte (E9) is written to the location using `patch_byte` (`addr, byte`). Next, the offset address for the relative near jump is written using `patch_dword(addr` , `dword)` . The instruction is created, and `plan_and_wait` (`addr_start` , `addr_end`) is called to force IDA to reanalyze the instruction.

```
#
#   Make JMP destination code
#
del_items(deobf_patch_jmp_target_ea,1)
deobf_jmp_target_insn = ida_ua.insn_t()
ida_ua.decode_insn(deobf_jmp_target_insn, deobf_patch_jmp_target_ea)
create_insn(deobf_patch_jmp_target_ea)
```

Figure 29. Ensure that the Code at the Jump Target is Recognized as Code

The instruction located at the jump target address may not be recognized as code by IDA. To ensure that this will be interpreted not only as a valid instruction but the correct instruction, the steps in Figure 29 are followed. Once this has completed, the Call Type One obfuscation is now deobfuscated.

Windows Procedure Call Obfuscation

Some of the Windows API calls are obfuscated using the method outlined in this section. This obfuscation connects all of the obfuscations covered in the previous sections and adds a few twists. The purpose of this obfuscation is to call the `Winapi_LookupWindowsProcedure` function. This function looks up the address of a module name by hash and then executes the procedure. The method used to retrieve a Microsoft Windows procedure address will not be covered in this blog.

```
000: 68 00 C8 00 00            push    0C800h        ; Windows Proc Argument
005: 55                        push    ebp           ; Windows Proc Argument
006: 68 02 24 00 00            push    2402h         ; XOR Key
00B: 68 48 22 36 37            push    37362248h     ; Proc Hash
010: E8 0A 00 00 00            call    loc_01F
015: 67 64 69 33 32 2E 64 6C   imul    esi, fs:[bp+di], 6C642E32h
01D: 6C                        insb

;-------------------------------------------------------------
01E: 00                        db    0
;-------------------------------------------------------------

01F: 68 AE 3B 46 6E            push    offset loc_050
024: 0F 84 A5 D9 FC FF         jz      Winapi__LookupWindowsProcedure
02A: 75 04                     jnz     short loc_030

;-------------------------------------------------------------
02C: 41 1B 00 00               dd 1B41h
;-------------------------------------------------------------

030: 0F 85 99 D9 FC FF         jnz     Winapi__LookupWindowsProcedure
036: 74 04                     jz      short loc_6E463B8D
...
050: 68 BF 3B 46 6E            push    offset loc_080
055: FF E0                     jmp     eax         ; Call Windows Procedure
...
080: 6A 02                     push    2
< additional instructions >
```

Figure 30. Windows Procedure Call Obfuscation, Incorrect IDA Rendering

The CALL instruction at address `010` in Figure 30 is doing something shifty. Recall that a CALL instruction pushes the address of the next instruction to the stack — in this case, the address is going to be `015`. Following the CALL instruction, the `Winapi_LookupWindowsProcedure` is immediately called using a Call Type Three obfuscation (highlighted green). So the CALL pushes `015`, and the Call Type Three pushes the return address `loc_050`. This is somewhat new, so let's take a look at address `015`.

```
000: 68 00 C8 00 00            push    0C800h        ; Windows Proc Argument
005: 55                        push    ebp           ; Windows Proc Argument
006: 68 02 24 00 00            push    2402h         ; XOR Key
00B: 68 48 22 36 37            push    37362248h     ; Proc Hash
010: E8 0A 00 00 00            call    loc_01F
015: 67 64 69 33 32 2E 64 6C 6C 00     db 'gdi32.dll',0
<... trimmed ...>
01F: 68 AE 3B 46 6E            push    offset loc_050
024: 0F 84 A5 D9 FC FF         jz      Winapi__LookupWindowsProcedure
```

Figure 31. Windows Procedure Call Obfuscation, Correct IDA Rendering

Figure 31 shows what is actually located at address `015`. It is the name of a Microsoft Windows DLL (`gdi32.dll`, in this case). This DLL is going to be the first argument to the `Winapi_LookupWindowsProcedure` function. The CALL instruction is used to push the first argument. IDA interprets the bytes at `015` as code, and that makes it difficult to follow.

Deobfuscation

In Figure 30, instruction `01F` is a Call Type Three obfuscation, and instruction `050` is a Call Type One obfuscation. Each obfuscation is deobfuscated using previously mentioned methods.

```
000: 68 00 C8 00 00                push    0C800h         ; Windows Proc Argument
005: 55                            push    ebp            ; Windows Proc Argument
006: 68 02 24 00 00                push    2402h            ; XOR Key
00B: 68 48 22 36 37                push    37362248h        ; Proc Hash
010: 68 1A 00 00 00                push    offset loc_01F   ; Module Name
015: E8 0A 00 00 00                call    Winapi__LookupWindowsProcedure
01A: E9 36 00 00 00                JMP     loc_050
01F: 67 64 69 33 32 2E 64 6C 6C 00    db 'gdi32.dll',0
<... trimmed ...>
050: FF D0                         call    eax           ; Call Windows Procedure
055: E9 0A 00 00 00                jmp     loc_05F
```

Figure 32. Windows Procedure Call Deobfuscation

In order to pass the first argument (module name) to `Winapi_LookupWindowsProcedure`, the CALL instruction at `010` (Figure 31) will be replaced with a PUSH instruction. To accomplish this, the module name is shifted from `015` to address `01F` (Figure 32). Now, the PUSH instruction can be added at `010` to be used as the first argument for the CALL to `Winapi_LookupWindowsProcedure` (`015`).

Function by the Slice

All of the obfuscations that have been discussed turn the code into spaghetti. This confuses disassemblers, like IDA, and makes it difficult to automatically identify and create functions. After deobfuscation has occurred, it may not be the case that IDA can automatically identify the functions. This can be resolved by using IDA Python to identify and create functions based on byte search patterns.The approach identifies an artifact from the prologue and an artifact from the epilogue, and then connects the two using a recursive descent parser.

```
000: 83 C4 38                    add     esp, 38h
003: 5E                          pop     esi
004: 5F                          pop     edi
005: 5D                          pop     ebp
006: 44                          inc     esp
007: 44                          inc     esp
008: 44                          inc     esp
009: 44                          inc     esp
00A: FF 64 24 FC                 jmp     dword ptr [esp-4]
```

Figure 33. Function Epilogue

The function identification algorithm starts by locating a set of potential function epilogues. The algorithm for epilogue identification:

- Search for all `add esp` , `value` opcodes (`000`): `83 C4` or `81 C4`
- Track the amount being added to ESP (38h)
- Starting at `000` , walk over the instructions until the return instruction is reached at `00A`
- Track the registers being popped off the stack (ESI, EDI, EBP)

This algorithm can be repeated until all of the epilogues have been identified.

```
000: 55                          push    ebp
001: 53                          push    ebx
002: 57                          push    edi
003: 56                          push    esi
004: 83 EC 38                    sub     esp, 38h
```

Figure 34. Function Prologue

To locate a set of all of the potential prologues, the start address for each prologue (`000` , in this case) needs to be identified. This is difficult because none of the functions that need to be identified have a standard prologue (Figure 34). Since each prologue will have an equivalent epilogue and the epilogues are known, this can be used to identify the start address for each prologue:

- Search for all `sub esp` , `value` opcodes `(` `004): 83 EC` or `81 EC`

- Starting with the first SUB instruction located:
  - Compare the number of bytes subtracted from ESP (`38h`) to what was added to ESP for each of the function epilogues that were found.
    - The expectation is that the same amount subtracted in the prologue will be added in the epilogue.
  - Walk backward, from `004` to `000` , and verify that each register being pushed to the stack is in the same order as what was popped from the stack in the set of potential epilogues for this function.

If the ESP manipulation value (`38h`) and the register value PUSH/POP instructions match, then it is safe to say that a function prologue has been identified and the start address has been located.

Control Flow Graph Recovery

Both the start address and the end address for the function have been identified, and now the pieces in between, the basic blocks, need to be walked to confirm that the prologue and epilogue actually connect. A recursive descent parser is used to walk the control flow graph. The implementation used in the source code is based on the example from the book *Practical Binary Analysis by Dennis Andriesse*.

Function Creation

The only thing left to do is create the functions that were identified and cleaned up using the algorithm outlined in the CFG Recovery section.

1. Create the identified functions during the CFG Recovery process using IDA's `add_func` (`start_address, end_address`) method.
2. Redefine functions that were undefined during the CFG Recovery process.

Conclusion

As demonstrated throughout this discussion, Maze's obfuscated control flow can cause quite a headache for an analyst, but with a little bit of planning (and Python), these obfuscations can be removed. In the opening paragraphs of this blog post an IDA navigation bar of the obfuscated Maze binary was shown (Figure 1). The expectation is that this will be significantly different after deobfuscation.

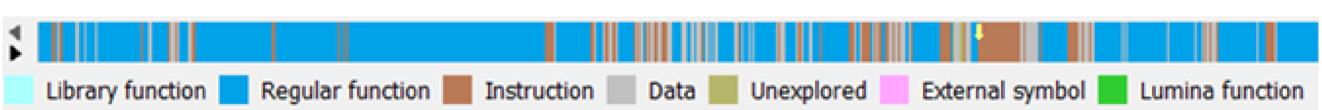


Figure 35. IDA Navigation Bar for Deobfuscated Maze Binary

Figure 35 shows a significant improvement. Still, there is plenty of brown and gray space. It's to be expected. Not everything is going to be part of a regular function. The presented solution is not going to deobfuscate 100% of the code. It will provide an analyst with enough deobfuscated code that the control flow is both easier to follow and easier to manually correct.

For further reading on binary analysis and software obfuscation, both "Surreptitious Software" by Christian Collberg and Jasvir Nagra and "Practical Binary Analysis" are excellent resources.

The source code has been published on GitHub.

Additional Resources

- *Download the CrowdStrike® 2020 Global Threat Report.*
- *See the CrowdStrike Falcon® platform in action, and learn how true next-gen AV performs against today's most sophisticated threats — get a full-featured free trial of CrowdStrike Falcon Prevent™ today.*
- *To learn more about how to incorporate intelligence on threat actors and their tactics techniques and procedures (TTPs) into your security strategy, please visit the Falcon X™ Threat Intelligence page.*
- *For more information on the cellular automation depicted in the blog header image, read about John Conway's Game of Life.*