Improvements on Hiding x86-64 Instructions by Interleaving

William Mahoney[1], J. Todd McDonald[2], George Grispos[1] and Sayonnha Mandal[1]
[1] School of Interdisciplinary Informatics, University of Nebraska at Omaha, Omaha, Nebraska, USA
[2] Computer Science Department, University of South Alabama, Mobile, Alabama, USA

wmahoney@unomaha.edu
jtmcdonald@southalabama.edu
ggrispos@unomaha.edu
smandal@unomaha.edu

Abstract: This paper presents the results of a new method for interleaving CPU instructions in x86-64 machine code, such that one can hide executable code within other valid instructions. The aim is to make it more difficult to reverse-engineer software at a machine code level – to obfuscate instructions. A result is a hidden execution path within a visible main instruction path. While previous methods for this instruction obfuscation exist, we present a new method which builds upon past work, and which allows a greater flexibility in the selection of main instruction path instructions. The result of this new approach is to provide a methodology for instruction concealment which is free of restrictions present in prior work.

Keywords: Instruction selection, instruction set architecture, obfuscation, reverse engineering

1. Introduction

The process of reverse engineering software makes the software susceptible to various types of attacks if vulnerabilities can be discovered, intellectual property theft if key algorithms can be discerned, and violating service agreements if portions of licensing software can be disabled. In some cases, therefore, software authors may be motivated to make the reverse engineering process more difficult. Looking at things from the opposite perspective, malware authors also wish to make their malware difficult to reverse engineer. Regardless of the motive, a methodology which has been explored in the past is a form of "executable steganography"; one can embed machine instructions which can be executed but which are not obvious when reverse engineering the program.

Steganography involving instructions hidden in other data is simply a specialization of steganography in general. Numerous techniques already exist and are available for embedding digital data into image files, for example (Morkel 2005) and (Hamid 2012). If, using steganography, a picture can hide data that includes executable software, then it is reasonable to think that one can do the opposite and hide an image inside of executable code. And for that matter, one should be able to conceal executable code inside of executable code.

We are not the first researchers to propose executable programs as cover objects for steganographic purposes. Other researchers have explored executable steganography for hiding not only other programs but other data in general. One previous approach to executable steganography in particular will be described and reviewed in this paper, as this research uses the method as a starting point, presenting potential modifications and thoughts concerning the method. Specifically, we will refer at length to a technique described by Christopher Jämthagen, Patrik Lantz, and Martin Hell in 2013 (Jamthagen 2013). These authors present an algorithm involving what they refer to as a Main Execution Path (MEP), as well as a Hidden Execution Path (HEP). The goal of their method is to hide HEP within MEP according to certain constraints. In our research we will aim to provide improvements to the method, which we will refer to as "JLH" for Jämthagen, Lantz, and Hall.

The paper here initially describes, in section two, the common methods used to recover instructions via reverse engineering. These methods are typically static analysis (not executing the program), although the analysis may need to happen at runtime if code is dynamically unpacked or unencrypted at runtime.

Since we are specifically using JLH as our starting point, we forgo much of the traditional "prior work" section and instead present the background of their method in section three. Once an understanding of JLH is complete in section three we turn to and describe a related approach involving the scattering of instructions within an array of bytes, and the addition of MEP instructions using a depth-first approach. This approach is in section four. Section five provides some example results and thoughts, and conclusions and future work are in section six.

2. Reverse Assembling Binary Code

The problem of reverse engineering binary code comes down to reverse assembling the object code and instructions back into something that is human readable. In some cases this is easier than in other cases. For example, if the binary executable is for an ARM or other typical Reduced Instruction Set Computer (RISC), the process is helped by the fact that the instructions are typically a fixed length and "aligned" on certain addresses. In the case of Complex Instruction Set Computers (CISC) and the x86 family, the reverse assembly process is quite complex and cumbersome. Even the number of, and types of these instructions can be a mystery (Mahoney 2021).

Static analysis, looking at the instructions without executing the program, develops or discovers properties that hold true for all executions of a program. De-compilation (or reverse compilation), disassembly, control flow discovery, and data flow discovery are some examples of static analysis. Contrasting this is dynamic analysis, where the program is executed and data is collected. This leaves potential undiscovered pathways within the code; these paths will not be analyzed if they were not executed on a particular run of the program. Popular tools for static analysis include Ghidra (2022), objdump (LinuxDevCenter 2022), and udcli (Udis86 2022). The static analysis tool IdaPro (HexRays 2022), performs static analysis but also integrates with various debuggers, making this tool partially static and partially dynamic. Dynamic analysis tools include Binary Ninja (2022), Hopper (2022), OllyDbg (2022), Relyze (2022), x64dbg (2022), and more. Online web sites are also available for disassembly, including simple interfaces provided by – for example – Defuse (2022). All of these tools use several approaches to determine how the code operates.

Some are static analysis tools only, while some include integrated debuggers. Udcli and objdump programs are simple command-line static disassembly tools, while others such as OllyDbg and x64dbg are more traditional debugging platforms which are used mainly for dynamic analysis.

Whether the reverse assembly process is dynamic or static, there are two general approaches to reverse engineering the instruction stream: a linear scan method or a tree-based method (Linn 2003). Scanning starts at a program entry point and assumes that the instructions are contiguous. A scan simply converts the entire instruction area back to assembly language, translating one instruction at a time until there are no more instructions. A tree-based approach also starts at the entry point of the program, but then uses intelligence to queue the destinations of branch/jump instructions, starting the reverse assembly process at each destination. Simple reversing tools such as the Linux objdump utility perform a linear scan only, while advanced tools such as IdaPro use a combination of linear and tree-based scanning. If the target of a jump instruction is known, the reversal process is relatively straightforward, but branches to addresses that are calculated at runtime will not be known with static analysis and as a result, some code may be missed. Additionally, the x86 architecture includes several different categories of jump instructions (AMD 2022), (Intel 2006), including unconditional or conditional branches, a relative or absolute address as a destination, and may be indirect (a register holding the address). It is thus not possible to determine the target of every jump using only static analysis.

In the work described here we are primarily concerned with placing the hidden execution path (HEP) within the main execution path (MEP) in such a way that traditional reversing tools such as IdaPro (Pearce 2008) are confused into thinking that the MEP is the correct code. This is the aim of the approach described in JLH, but their approach has a very restricted view for selecting the instructions used for the MEP. Nonetheless it is necessary to next describe their approach, prior to suggesting improvements or changes.

3. The Jämthagen, Lantz, and Hall Method

3.1 Background: Anti-Disassembly

A well-known and understood technique for thwarting the reverse-assembly process (at least on x86) is the insertion of "junk" bytes to throw off the instruction decoding process. We have reported on this in the past and related as to which additional bytes of junk cause the most damage in terms of misrepresented instructions (Shinn, Mahoney 2011). An example is shown in Figure 1 below from Mahoney (2015), where the addition of 0x05 has caused the reverse-assembly process to mis several instructions.

If the inserted extraneous bytes ("junk" bytes) establish the beginning of an instruction, the disassembler will start producing the wrong code. The difficulty is that x86 instructions are varying in length, and in a generally short period the reversing tool will be synchronized again with the correct instructions.

Original Disassembly		
c745d401000000	mov	DWORD PTR [rbp-0x2c], 0x1
48c7c040a93c00	mov	rax, 0x3ca940
488d8024740300	lea	rax, [rax+0x37424]
ffe0	jmp	rax
Addition of 0x05		
05c745d401	add	eax, 0x1d445c7
0000	add	BYTE PTR [rax], al
0048c7	add	BYTE PTR [rax-0x39], cl
c073a93c	rol	BYTE PTR [rax-0x57], 0x3c
00488d	add	BYTE PTR [rax-0x73], cl
80247403	and	BYTE PTR [rsp+rsi*2], 0x3
00ff	add	bh, bh
e01d	.byte	0xe0

Figure 1: Confusing a reverse assembler by insertion of one byte of "junk"; the trailing 0xe0 will become a "loop" instruction with the next byte.

A related technique used to fool tree-based reverse engineering tools is to include a jump/branch instruction which has the address of the junk byte as the target of the branch. This will add this destination address to the list of code areas that the approach needs to work on, thus eventually forcing the disassembly to start at the wrong location. In Figure 1 we would include code that jumps to the address of the added 0x05, but with the caveat that the jump will never actually be taken when the program runs. This can easily be accomplished with an opaque predate – a small bit of code generating the results of a comparison, with the understanding that the comparison always results in a false (or true) result. This will be followed by a conditional jump based on the condition being true (or false).

A variation of junk insertion is code overlapping, as described in JLH and elsewhere. Code overlapping creates instructions that have their bytes in two (or more) other instructions. It is possible to create or select instructions with certain properties such that starting a block from two different addresses will execute two completely different sequences of code. By making sure that all instructions in the two (or more) paths always overlap, the instructions will not resynchronize until the end of the overlapped block.

3.2 Background: Anti-Disassembly

Many cybersecurity practitioners will be familiar with the x86 instruction with the operation code 0x90; it is a "no-operation" or "NOP". What may be less familiar is that this instruction exists in many forms; the one-byte form is just an easy one to remember. However, the byte sequence 0x66, 0x90 is also valid. The 0x66 is an "operand-size override prefix". This prefix indicates that the normal operand size for the instruction is not correct in this case. So, the two-byte sequence says, "override the normal operand size, for an instruction that does nothing". Other prefix bytes can also be prepended on x86 instructions, such as segment overrides; in fact, one can construct a 15-byte (the maximum allowed) sequence of bytes which form a valid instruction, which when executed does nothing. This knowledge is key to the approach by JLH.

Jämthagen uses a 9-byte NOP instruction. By formatting the operation code correctly, this no-operation instruction will contain five bytes, made up of an addressing mode and a memory address. Since the instruction itself is a no-operation the actual values in these five bytes are irrelevant since the memory address will never be accessed. This is similar to an approach where the contents of 64-bit immediate operands are used to hide executable instructions (Mahoney 2018). That method proves to be interesting in theory but not practical, due to the low likelihood of encountering 8-byte operands in actual typical programs (Mullins 2020), (Mullins 2022).

3.3 Overlapping

In order to take advantage of the 9-byte NOP and overlap instructions, Jämthagen posits the following requirements:

No alignment: the instructions must overlap each other and must never be aligned such that two instructions end at the same byte.

Both valid: each execution path – both the normal and hidden paths – must consist of valid instructions.

Both executable: each execution path must consist of executable instructions.

Recall the previous acronyms: we are overlapping a hidden execution path (HEP) with a main execution path (MEP). Requirement one is to avoid the case where a reverse assembler will re-synchronize with the hidden instruction stream part way through a block; this implies that all instructions in HEP are disjoint from those in MEP. Requirement two assures that the reverse engineering tool will not stop on an illegal instruction in the middle of the block, thus accidentally alerting an analyst that "something's wrong with this picture". This requirement is mainly concerned with MEP, as presumably HEP was hand-written in advance. Requirement three is the most difficult – if the code is executed, both MEP and HEP must execute and not generate – for example – an illegal memory reference, faulting the program. Requirement three is what forces the JLH method to use NOP instructions, as this requirement is primarily concerning MEP.

The work by Jämthagen considers instructions in MEP that will take care of obfuscating MEP: "… the exact effects the MEP has are not important since its primary function is to hide the HEP". Their idea is to then select "instrucitons which have as many "bytes that can be arbitrarily chosen as possible".

These instructions are divided into three sections, which are then referred to as "XXYYZZ". The idea is that the portion represented by "XX" contains the instruction prefixes as well as the actual operation code for the instruction. The middle "YY" section contains bytes whose actual value is not critical. These bytes might make up a memory address (which will not be accessed) or an immediate operand of an instruction. Lastly, "ZZ" needs to also be capable of having any byte assigned to it, with the note that the combination of "ZZ" and "XX" of the next instruction should decode to a valid instruction, thus satisfying requirement one. This combination of "ZZXX" is referred to as a "wrapping instruction". The HEP is embedded in a series of 9-byte NOP instructions, with the "ZZ" portion of each MEP instruction overlapping the NOP. The authors present the following example; here the top of Figure 2 shows the 9-byte NOP instructions which will hide the HEP instructions. But if instruction execution starts four bytes into the first block, at the 0x66, the hidden path is executed, as shown in the bottom of the figure.

Original Disassembly – MEP		
660f1f8466bb0100a9	nop	word ptr [rsi – 0x56fffe45]
660f1f8431c040cd80	nop	word ptr [rcx + rsi – 0x7f32bf40]
Executing From Byte Offset Four – HEP		
66bb0100	mov	bx,0x1
a9660f1f84	test	eax,0x841f0f66
31c0	xor	eax,eax
40	inc	eax
cd80	int	0x80

Figure 2: Main Execution Path (top), Hidden Execution Path (bottom)

Much of the remainder of the Jämthagen approach is concerned with improving the "look" of the MEP. For example, if a reverse engineering expert sees a long string of 9-byte NOP instructions they will start to be suspicious. An approach (which we will see again in section IV-a) is to scatter the instructions in HEP a bit farther apart and include at the end of each HEP instruction a jump to the next instruction. Although we will use the same idea, note that there are two drawbacks to this approach. First, including a jump takes away two bytes from the "payload" of the NOP instruction and pressures the author of the HEP instructions to select instructions which are very small in terms of byte count. Second, the 2-byte format of an x86 jump contains a byte displacement, limiting the distance to jumping from -128 to +127 bytes away. The JLH paper spends some time covering how to write short sections of code with these limited-length instructions. Our approach is to hand the compiler the "-Os" (optimize for size) command line option and see how things come out!

4. Materials and Methods

In this section we introduce a new methodology to see if we can improve upon the JLH method, while at the same time still holding to the three restrictions outlined in their paper. As mentioned above, previous work included an approach whereby HEP is encoded in the 8-byte operands of 64-bit instructions. The procedure works, but like in JLH, the output for MEP "looks suspicious" since there is a need for many 64-bit immediate

operands or addresses. Since our goal is to interleave instructions, we call the process "I²". The I² method consists of two main phases, a placement phase and a fill phase.

4.1 Instruction Placement

The first step in the process is to craft the code which will be in the hidden execution path (HEP). Here it is preferable to have instructions that are small. The algorithm will insert jump instructions and the displacement of the jump is limited; short instructions make for close jumps. There are also a few assumptions made with respect to the code placed in HEP. Jämthagen does not specify or indicate the surrounding context for the block of code. Here we assume that our method will produce a stand-alone function for MEP which includes HEP; the function will have a standard prologue and will jump over MEP to the epilogue, thus loosening requirement number three, above. This jump is currently unconditional but can easily be modified to use an opaque predicate to ensure that instructions in MEP that may cause a fault will not be executed. In fact this predicate might rely on a parameter passed to the function, making it more difficult for one to realize that it is always the same conditional result.

The instructions in HEP are initially (manually) placed into a data structure which is built from an assembly "listing" file, typically generated by the Gnu assembler "as". For example, consider the example HEP code outlined in Jämthagen:

```
MOV     BX, 0x0001
TEST    EAX, 0x841F0F66
XOR     EAX, EAX
INC     EAX
INT     0x80
```

Figure 3: Original Test Case, from Jämthagen et al.

The data structure corresponding to this is in Figure 4. Note that there is no need for us to include the second instruction since it serves no purpose.

```
Struct mal_inst decoder[] = {
{ 0,4,no_jump,1,1, { 0x66, 0xbb, 0x01, 0x00 }, "MOV BX,0x0001" },
{ 0,2,no_jump,2,2, { 0x31, 0xc0 }, "XOR EAX,EAX" },
{ 0,1,no_jump,3,3, { 0x40 }, "INC EAX" },
{ 0,2,no_jump,0,0, { 0xcd, 0x80 }, "INT 0x80" }
};
```

Figure 4: Data Structure for I² using Jämthagen et al.

The fields in this structure include the location where the instruction will be placed, the number of bytes, the type of instruction as well as the number of the next instruction, the actual opcode bytes, and the original string which is used for debugging. Thus, each instruction includes meta-data with the actual bytes necessary for the instruction, as well as other information including the instruction type. This type is either a "regular" instruction, a conditional jump, or an absolute jump. (Their example does not include any jumps.) The absolute jumps are limited to 2-byte jumps with the second byte as the offset, as per x86. In the case of conditional jumps, the structure maintains the number of the next instruction assuming the condition is met, as well as the next instruction number if the condition is not met.

Using this data structure, the instructions are scattered in a random order into a "field" of bytes. The field has been initialized with random valued bytes, so the location of each instruction in HEP is tracked. Each regular opcode or conditional opcode requires an additional two bytes; these two bytes will encode an unconditional jump with the offset, and this jump will go to the next instruction in the original sequence. The instructions will not be in the field in the same order that they appear in the assembly code, and the displacements in the jump instructions are set so that the proper sequence is maintained.

The field of bytes might appear as follows, using a mixture of notations to better convey the idea. In this Figure 5, "Random" is a span of two yet unused bytes prior to the first instruction in the field, and between all subsequent instructions. The ordering of the instructions is dictated by the seed for a pseudo-random number generator, which can be set by the user. It is of course necessary to make a note of the correct entry point (the

"MOV" instruction) as this is the instruction to jump to in order to execute the HEP. If one were to write this in assembly language from scratch, it might look like Figure 5 below.

Since the data is an array of bytes there are no actual assembly language labels available; as each instruction includes a jump to the next instruction in sequence, the next step is to insert the correct 8-bit displacement into this jump instruction such that the normal instruction flow remains intact. Unfortunately, there is the potential that the instruction sequence is ordered such that the next instruction is "too far away" for this displacement. In this case since the algorithm discards this attempt and selects another random ordering of the instructions. Assuming this placement can be ordered, I^2 proceeds to the next stage.

Random bytes		
L1:	XOR	EAX, EAX
	JP	L2
Random bytes		
Entry:	MOV	BX, 0x0001
	JP	L1
Random bytes		
L3:	INT	0x80
Random bytes		
L2:	INC	EAX
	JP	L3
Random bytes		

Figure 5: Original Test Case Expanded Into Byte Field

4.2 Filling MEP

The next stage consists of randomly generating instructions to cover up those in HEP, while still satisfying the "No alignment" and "Both valid" restrictions. This is accomplished with a recursive depth-first algorithm, much like the search strategy of the language Prolog. The search function is called with a state, including the partially filled field as well as the current position. Several different events may now happen:

1. The current position may match with the beginning of an instruction in HEP.
2. The current position may be in the middle of an instruction in HEP.
3. The current position may be in the gap between HEP instructions.

Let us consider each case in turn, starting with the first. If the position corresponds to the location in the field of any instruction in HEP, this invalidates Jämthagen's "No alignment" condition; this is a failed state, and it is necessary to backtrack.

In case two, if we can decode a valid instruction from this point, we should advance to a new state. This new starting point will be the current location plus the length of the decoded instruction.

Case three is more complex. The current position is between instructions in HEP so the opportunity exists to have more flexibility. Recall from above that a technique for executable steganography is the addition of a byte of junk. Here values from 0..255 (0x00..0xff) are selected from a random permutation and placed at this location. The location is tested to see if a valid instruction results, and if so, we advance to a new state starting at the end of this new instruction. At any point in this search, the process may fail and return to a prior state. Here we may try all 256 junk bytes, with none of them yielding a valid operation code. When this happens, the process backtracks to a prior state, which may at that point try other values.

To use terminology from JLH, in this third case we are randomly selecting "XX" in the gaps between HEP and allowing "YY" to be the operation codes in the HEP instruction.

4.3 Finalizing the Sequences

Next, recall the three conditions outlined by JLH. "No alignment" states that the instructions must overlap. Since the starting point for each instruction in MEP is either inside a HEP instruction or in the gap between them, this condition is met.

"Both valid" states that both HEP and MEP are made up of legal (decodable) instructions. Those in HEP are valid from the onset. At any point in the I^2 algorithm if an instruction cannot be decoded at a certain position the search backtracks. It is true that the search may never find a valid sequence for MEP; but if a sequence is found it will be valid instructions, so this restriction is met by the algorithm.

The third condition, "Both executable", is the most difficult. As is pointed out (extensively) in JLH, certain instructions are safe and certain instructions are not safe. "Load effective address", for example, puts an address into a register but does not actually access the memory at that address. This is safe. Certain instructions are – by their very nature – not safe. Consider "LIDT" which loads the address of the interrupt dispatch table. This instruction can only be run in privilege level zero and will otherwise fault the program. Instructions which access memory are off limits since the address used will be arbitrary. Instructions that require any escalated privilege level are also bad since they will fault. But recall that here we are simply jumping over the MEP instructions, so illegal instructions are not executed. This condition is met by simply inserting a false predicate to avoid the code.

5. Discussion and Example Results

Next, we take several examples to demonstrate the operation of the method. The goal is to produce code which will, when disassembled, appear as just another part of the program. (One can argue as to whether a sequence of very strange instructions "stands out", but one can also argue that a long sequence of 9-byte NOP instructions stands out just as much.) The aim is to prevent HEP from appearing, unless the reverse engineer knows where to look and what to do.

We will examine the binary code artifacts with two tools, the popular IdaPro (HexRays 2022) reverse engineering tool and the open source Ghidra (2022) tool.

5.1 Repeating Jämthagen

We start first with the same example described in the Jämthagen paper, as shown in Figures 3, 4, and 5 above. This code is designed to run on a 32-bit Linux system, as interrupt 0x80 is a system call. In this case the system call will cause a program exit.

When this code is used for our algorithm, IdaPro shows only the function prolog and epilog, as in Figure 6:

```
; =============== S U B R O U T I N E =======================================

; Attributes: bp-based frame

sub_1199        proc near               ; CODE XREF: .text:0000000000001158↑p
                push    rbp
                mov     rbp, rsp
                jmp     short loc_11BA
; ---------------------------------------------------------------------------
                db 0C8h
                dq 403DEC9D0FEB80CDh, 0F7EBC0318183F6EBh, 0EB0001BB663B25D4h
                db 0F3h, 0F4h
; ---------------------------------------------------------------------------

loc_11BA:                               ; CODE XREF: sub_1199+4↑j
                pop     rbp
                retn
sub_1199        endp
```

Figure 6: Original Test Case, from Jämthagen et al.

Assuming the reverse engineer notices the constants within the function, the next step would be to manually instruct IdaPro to start disassembly at the first non-decoded byte. This results in uncovering more of MEP as in the following Figure 7 below.

Note that although this appears to be a strange sequence of instructions, it decodes with no illegal operations and still hides the actual aim of the instructions above. While not shown here, the Ghidra tool gives very similar results.

```
; =============== S U B R O U T I N E =======================================

; Attributes: bp-based frame

sub_1199        proc near               ; CODE XREF: .text:0000000000001158↑p
                push    rbp
                mov     rbp, rsp
                jmp     short loc_11BA
; ---------------------------------------------------------------------------
                enter   0FFFFFFFFFFFF80CDh, 0EBh
                setnl   ah
                cmp     eax, 83F6EB40h
                xor     dword ptr [rcx], 0D4F7EBC0h
                and     eax, 1BB663Bh
                add     bl, ch
                rep hlt
; ---------------------------------------------------------------------------

loc_11BA:                               ; CODE XREF: sub_1199+4↑j
                pop     rbp
                retn
sub_1199        endp
```

Figure 7: Original Test Case, from Jämthagen et al., additional instruction decoding

5.2 A More Serious Example

In terms of malware or nefarious deeds, code that simply exits a program, as above, is not too serious. We next test the following sequence of x86-64 instructions as shown in figure 8.

```
XOR             EAX, EAX
MOV             AL, 0xA9
MOV             EDI, 0xFEE1DEAD
MOV             ESI, 0x28121969
MOV             EDX, 0x4321FEDC
XOR             R10, R10
SYSCALL
```

Figure 8: Malware Testcase, Halt System

This is a Linux 64-bit operating system call and, assuming it is run as root, will immediately halt and power off the computer. (This, as a side note, makes testing and debugging occasionally problematic.) Obfuscating this code using our algorithm results in the following view in Ghidra. This tool has missed the hidden instructions entirely, as in the following (partial) Figure 9:

```
                        *************************************************************
                        *                          FUNCTION                         *
                        *************************************************************
                        undefined FUN_00101199()
          undefined         AL:1           <RETURN>
                        FUN_00101199                                    XREF[1]:     00101158(c)
     00101199 55            PUSH        RBP
     0010119a 48 89 e5      MOV         RBP,RSP
     0010119d eb 45         JMP         LAB_001011e4
     0010119f 03            ??          03h
     001011a0 72            ??          72h    r
     001011a1 a5            ??          A5h
     001011a2 e9            ??          E9h
     001011a3 c3            ??          C3h
     001011a4 eb            ??          EBh
     001011a5 04            ??          04h
```

Figure 9: Malware Testcase, Halt System, Ghidra View

Manually instructing Ghidra to convert this to code results in uncovering only the main execution path as shown in figure 10 below:

```
                    undefined FUN_00101199()
        undefined         AL:1           <RETURN>
                    FUN_00101199                                XREF[1]:     00101158(c)
00101199 55              PUSH        RBP
0010119a 48 89 e5        MOV         RBP,RSP
0010119d eb 45           JMP         LAB_001011e4
0010119f 03 72 a5        ADD         ESI,dword ptr [RDX + -0x5b]
001011a2 e9 c3 eb        JMP         LAB_ffffffffe614fd6a
         04 e6
001011a7 57              PUSH        RDI
001011a8 a0 b8 31        MOV         AL,[DAT_bab1cf15ebc031b8]
         c0 eb 15
         cf b1 ba
001011b1 dc fe           FDIV        ST6,ST0
001011b3 21 43 eb        AND         dword ptr [RBX + -0x15],EAX
001011b6 05 c7 ea        ADD         EAX,0xff4eac7
         f4 0f
001011bb 69 4d 31        IMUL        ECX,dword ptr [RBP + 0x31],0x770aebd2
         d2 eb 0a 77
001011c2 34 b0           XOR         AL,0xb0
001011c4 a9 eb 0a        TEST        EAX,0xa1870aeb
         87 a1
001011c9 98              CWDE
001011ca 69 0f 05        IMUL        ECX,dword ptr [RDI],0x39d4eb05
         eb d4 39
001011d0 81 bf ad        CMP         dword ptr [RDI + -0x11e2153],0xadb005eb
         de e1 fe
         eb 05 b0 ad
```

Figure 10: Malware Testcase, Halt System, Ghidra View, Manual Conversion

Lastly, we consider a case where a nefarious malware author may wish to hide an encryption algorithm of some sort. Consider the following 'C' function to do a simple block copy with an exclusive-or:

```
void decrypt( unsigned char *a, unsigned char *b, unsigned n )
{
    for( unsigned I = 0; i < n; i++ )
        *b++ = *a++ ^ 0x42;
}
```

Figure 11: Simple 'C' Block Decryption Function – XOR with 0x42

This code is compiled using GCC and optimized for space, generating an assembly function with nine instructions, which is then inserted as the hidden execution path. The following figure shows the (partial) result in IdaPro:

```
sub_11AF        proc near               ; DATA XREF: .text:0000000000001145↑o

; FUNCTION CHUNK AT .text:000000000000121F SIZE 00000046 BYTES

                push    rbp
                mov     rbp, rsp
                jmp     short loc_11FB
; ---------------------------------------------------------------------------
                insd
                insb
                movsb
                imul    ecx, [rdx+22EB070Ch], 24h
                imul    esi, [rsi+31h], 0B1CFF5EBh
                xor     [rsi-14F9F378h], bh
                add     al, bh
                adc     eax, 0EBC0FF48h
                adc     eax, 77E58F20h
                cmp     dword ptr [rcx], 87E3EBC2h
                xchg    eax, edi
                or      eax, 0EB42F183h
```

Figure 12: Malware Testcase, Decoder Function in IdaPro

It appears here that IdaPro has again defaulted to the simple view of performing a linear disassembly over all of the main execution path, never seeing any of the hidden instructions. Calling externally to this function address plus (in this case) 0x3E causes the decoder to execute. Ghidra and other tools show the same view of this code.

6. Conclusions and Future Work

The methods outlined by Jämthagen, Lantz, and Hell do an excellent job of hiding executable instructions inside of a series of no-operation instructions. The advantage of their method lies in the fact that one can still safety execute the main instruction sequence with no ill effects. However, this severely limits the available

opportunities to hide instructions; providing an appropriate opaque predicate to avoid executing the main path opens better opportunities for protecting the secret steganographic instruction payload.

The method discussed here opens up greater opportunities in terms of instruction hiding. First, it removes the restriction that the main execution path consist of no-operation instructions. Our argument is that a long string of these NOP instructions may stand out when reverse engineering, and if the sequence is already "somewhat odd" it doesn't matter which instructions make it look odd. Second, if one opens the possibility of using any hiding instruction, as opposed to only NOP instructions, those instructions which are hidden within are no longer restricted. As a result, longer code sequences (relative to the original work) can be hidden and still execute correctly.

Looking forward, the method outlined here utilizes the "xed" (2022) package to determine whether the bytes at a certain location in our "field" decode to an instruction. Potential future research is to restrict the decoded instructions to ones that use only immediate operands, for example, to make the main execution path appear more like normal code. Alternatively, one can construct, using "xed", instructions with certain characteristics, and have the package return back the actual opcode bytes. These opcode bytes could be used instead of the random selection method currently used by the algorithm.

References

AMD, *AMD64 Architecture Programmer's Manual— Volume 3: General-Purpose and System Instructions*, [online], https://www.amd.com/system/files/TechDocs/24594.pdf, accessed November 2022.

Binary Ninja, [online], https://binary.ninja, accessed November 2022

Defuse, [online], https://defuse.ca/online-x86-assembler.htm, accessed November 2022.

Ghidra, [online], https://ghidra-sre.org, accessed November 2022.

HexRays IdaPro, [online], http://www.hex-rays.com/products/ida/index.shtml, accessed November 2022.

Hamid, N., Yahya, A., Ahmad, R. B., & Al-Qershi, O. M. (2012). Image steganography techniques: an overview. *International Journal of Computer Science and Security (IJCSS)*, *6*(3), 168-187.

Hopper, [online], https://www.hopperapp.com, accessed November 2022.

Intel (2006). *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-M*, 2006.

Jamthagen, C, Lantz, P. and Hell, M. (2013). "*A new instruction overlapping technique for anti-disassembly and obfuscation of x86 binaries*", 2013 Workshop on Anti-malware Testing Research (WATeR), 978-1-4799-2476-9, IEEE.

Linn, C. and Debray, C. (2003). "*Obfuscation of Executable Code to Improve Resistance to Static Disassembly*", Proceedings of the ACM Conference on Computer and Communications Security (CCS 2003), Washington, DC, USA, Oct. 27-30, 2003.

LinuxDevCenter, [online], http://www.linuxdevcenter.com/cmd/cmd.csp?path=o/objdump, accessed November 2022.

Mahoney, W. (2015) "*Modifications to GCC for Increased Software Privacy*", International Journal of Information and Computer Security, Vol. 7, No. 2, 3, 4.

Mahoney, W., Franco, J., Hoff, G. and McDonald, J. T. (2018) "*Leave it to Weaver*", 8th Software Security, Protection, and Reverse Engineering Workshop, Puerto Rico, December 3-4.

Mahoney, W., McDonald, J. T., [online], "*Enumerating x86-64 – It's Not as Easy as Counting*", https://www.unomaha.edu/college-of-information-science-and-technology/research-labs/_files/enumerating-x86-64-instructions.pdf, accessed November 2022.

Morkel, T., Eloff, J. H., & Olivier, M. S. (2005). "An overview of image steganography", In Information Security South Africa (ISSA) (Vol. 1, No. 2, pp. 1-11)

Mullins, J. A., McDonald, J. T., Mahoney, W. and Andel, T. (2020) "*Evaluating Security of Executable Steganography for Digital Software Watermarking*", 2020 Cybersecurity Symposium, Moscow, Idaho.

Mullins, J. A., McDonald, J. T., Mahoney, W. and Andel, T. (2022) "*Evaluating Security of Executable Steganography for Digital Software Watermarking*", IEEE SoutheastCon 2022, Mobile, AL, March 31 – April 3, 2022.

OllyDbg, [online], http://www.ollydbg.de/, accessed November 2022.

Pearce, W., (2008). *Reverse Engineering Code with IDA Pro*. Elsevier Science.

Relyze, [online], https://www.relyze.com/overview.html, accessed November 2022.

Shinn, S. and Mahoney, W. (2011) "*Optimal Values for Disrupting x86-64 Reverse Assemblers*", International Journal of Computer Science and Network Security, Volume 11, Number 11.

Udis86, [online], http://udis86.sourceforge.net/, accessed November 2022.

x64dbg, [online], https://x64dbg.com, accessed November 2022.

XED, [online], https://intelxed.github.io/ref-manual/, accessed November 2022.