x86 Disassembly/Code Obfuscation

Code Obfuscation

Code Obfuscation is the act of making the assembly code or machine code of a program more difficult to disassemble or decompile. The term "obfuscation" is typically used to suggest a deliberate attempt to add difficulty, but many other practices will cause code to be obfuscated without that being the intention. Software vendors may attempt to obfuscate or even encrypt code to prevent reverse engineering efforts. There are many different types of obfuscations. Notice that many code optimizations (discussed in the previous chapter) have the side-effect of making code more difficult to read, and therefore optimizations act as obfuscations.

What is Code Obfuscation?

There are many things that obfuscation could be:

- Encrypted code that is decrypted prior to runtime.
- Compressed code that is decompressed prior to runtime.
- Executables that contain Encrypted sections, and a simple decrypter.
- Code instructions that are put in a hard-to read order.
- Code instructions which are used in a non-obvious way.

This chapter will try to examine some common methods of obfuscating code, but will not necessarily delve into methods to break the obfuscation.

Interleaving

Optimizing Compilers will engage in a process called **interleaving** to try and maximize parallelism in pipelined processors. This technique is based on two premises:

1. That certain instructions can be executed out of order and still maintain the correct output
2. That processors can perform certain pairs of tasks simultaneously.

x86 NetBurst Architecture

The Intel **NetBurst Architecture** divides an x86 processor into 2 distinct parts: the supporting hardware, and the primitive core processor. The primitive core of a processor contains the ability to perform some calculations blindingly fast, but not the instructions that you or I am familiar with. The processor first converts the code instructions into a form called "micro-ops" that are then handled by the primitive core processor.

The processor can also be broken down into 4 components, or modules, each of which is capable of performing certain tasks. Since each module can operate separately, up to 4 separate tasks can be handled *simultaneously* by the processor core, so long as those tasks can be performed by each of the 4 modules:

Port0
: Double-speed integer arithmetic, floating point load, memory store

Port1
: Double-speed integer arithmetic, floating point arithmetic

Port2
: memory read

Port3
: memory write (writes to address bus)

So for instance, the processor can simultaneously perform 2 integer arithmetic instructions in both Port0 and Port1, so a compiler will frequently go to great lengths to put arithmetic instructions close to each other. If the timing is just right, up to 4 arithmetic instructions can be executed in a single instruction period.

Notice however that writing to memory is particularly slow (requiring the address to be sent by Port3, and the data itself to be written by Port0). Floating point numbers need to be loaded to the FPU before they can be operated on, so a floating point load and a floating point arithmetic instruction cannot operate on a single value in a single instruction cycle. Therefore, it is not uncommon to see floating point values loaded, integer values be manipulated, and then the floating point value be operated on.

Non-Intuitive Instructions

Optimizing compilers frequently will use instructions that are not intuitive. Some instructions can perform tasks for which they were not designed, typically as a helpful side effect. Sometimes, one instruction can perform a task more quickly than other specialized instructions can.

The only way to know that one instruction is faster than another is to consult the processor documentation. However, knowing some of the most common substitutions is very useful to the reverser.

Here are some examples. The code in the first box operates more quickly than the one in the second, but performs exactly the same tasks.

Example 1

Fast

```
xor eax, eax
```

Slow

```
mov eax, 0
```

Example 2

Fast

```
shl eax, 3
```

Slow

```
push edx
push 8
```

```
mul dword [esp]
add esp, 4
pop edx ;# edx is not preserved by "mul"
```

Sometimes such transformations could be made to make the analysis more difficult:

Example 3

Fast

```
push $next_instr
jmp $some_function
$next_instr:...
```

Slow

```
call $some_function
```

Example 4

Fast

```
pop eax
jmp eax
```

Slow

```
retn
```

Common Instruction Substitutions

lea

The lea instruction has the following form:

```
lea dest, (XS:)[reg1 + reg2 * x]
```

Where XS is a segment register (SS, DS, CS, etc...), reg1 is the base address, reg2 is a variable offset, and x is a multiplicative scaling factor. What lea does, essentially, is load the memory address being pointed to in the second argument, into the first argument. Look at the following example:

```
mov eax, 1
lea ecx, [eax + 4]
```

Now, what is the value of ecx? The answer is that ecx has the value of (eax + 4), which is 5. In essence, lea is used to do addition and multiplication of a register and a constant that is a byte or less (-128 to +127).

Now, consider:

```
mov eax, 1
lea ecx, [eax+eax*2]
```

Now, ecx equals 3.

The difference is that lea is quick (because it only adds a register and a small constant), whereas the **add** and **mul** instructions are more versatile, but slower. lea is used for arithmetic in this fashion very frequently, even when compilers are not actively optimizing the code.

xor

The xor instruction performs the bit-wise exclusive-or operation on two operands. Consider then, the following example:

```
  mov al, 0xAA
  xor al, al
```

What does this do? Let's take a look at the binary:

```
    10101010 ;10101010 = 0xAA
xor 10101010
    --------
    00000000
```

The answer is that "xor reg, reg" sets the register to 0. More importantly, however, is that "xor eax, eax" sets eax to 0 *faster* (and the generated code instruction is smaller) than an equivalent "mov eax, 0".

mov edi, edi

On a 64-bit x86 system, this instruction clears the high 32-bits of the rdi register.

shl, shr

left-shifting, in binary arithmetic, is equivalent to multiplying the operand by 2. Right-shifting is also equivalent to integer division by 2, although the lowest bit is dropped. in general, left-shifting by N spaces multiplies the operand by 2^N, and right shifting by N spaces is the same as dividing by 2^N. One important fact is that resulting number is an integer with no fractional part present. For example:

```
  mov al, 31 ; 00011111
  shr al, 1  ; 00001111 = 15, not 15.5
```

xchg

xchg exchanges the contents of two registers, or a register and a memory address. A noteworthy point is the fact that xchg operates faster than a move instruction. For this reason, xchg will be used to move a value from a source to a destination, when the value in the source no longer needs to be saved.

As an example, consider this code:

```
mov ebx, eax
mov eax, 0
```

Here, the value in `eax` is stored in `ebx`, and then `eax` is loaded with the value zero. We can perform the same operation, but using `xchg` and `xor` instead:

```
xchg eax, ebx
xor eax, eax
```

It may surprise you to learn that the second code example operates significantly faster than the first one does.

Obfuscators

There are a number of tools on the market that will automate the process of code obfuscation. These products will use a number of transformations to turn a code snippet into a less-readable form, although it will not affect the program flow itself (although the transformations may increase code size or execution time).

Code Transformations

Code transformations are a way of reordering code so that it performs exactly the same task but becomes more difficult to trace and disassemble. We can best demonstrate this technique by example. Let's say that we have 2 functions, FunctionA and FunctionB. Both of these two functions are comprised of 3 separate parts, which are performed in order. We can break this down as such:

```
FunctionA()
{
  FuncAPart1();
  FuncAPart2();
  FuncAPart3();
}

FunctionB()
{
  FuncBPart1();
  FuncBPart2();
  FuncBPart3();
}
```

And we have our main program, that executes the two functions:

```
main()
{
  FunctionA();
  FunctionB();
}
```

Now, we can rearrange these snippets to a form that is much more complicated (in assembly):

```
main:
      jmp FAP1
FBP3: call FuncBPart3
      jmp end
FBP1: call FuncBPart1
      jmp FBP2
FAP2: call FuncAPart2
      jmp FAP3
FBP2: call FuncBPart2
      jmp FBP3
FAP1: call FuncAPart1
      jmp FAP2
FAP3: call FuncAPart3
      jmp FBP1
end:
```

As you can see, this is much harder to read, although it perfectly preserves the program flow of the original code. This code is much harder for a human to read, although it isn't hard at all for an automated debugging tool (such as IDA Pro) to read.

Opaque Predicates

An **Opaque Predicate** is a predicate inside the code, that cannot be evaluated during static analysis. This forces the attacker to perform a dynamic analysis to understand the result of the line. Typically this is related to a branch instruction that is used to prevent in static analysis the understanding which code path is taken.

Code Encryption

Code can be encrypted, just like any other type of data, except that code can also work to encrypt and decrypt *itself*. Encrypted programs cannot be directly disassembled. However, such a program can also not be run directly because the encrypted opcodes cannot be interpreted properly by the CPU. For this reason, an encrypted program must contain some sort of method for decrypting itself prior to operation.

The most basic method is to include a small stub program that decrypts the remainder of the executable, and then passes control to the decrypted routines.

Disassembling Encrypted Code

To disassemble an encrypted executable, you must first determine how the code is being decrypted. Code can be decrypted in one of two primary ways:

1. All at once. The entire code portion is decrypted in a single pass, and left decrypted during execution. Using a debugger, allow the decryption routine to run completely, and then dump the decrypted code into a file for further analysis.
2. By Block. The code is encrypted in separate blocks, where each block may have a separate encryption key. Blocks may be decrypted before use, and re-encrypted again after use. Using a debugger, you can attempt to capture all the decryption keys and then use those keys to decrypt the entire program at once later, or you can wait for the blocks to be decrypted, and then dump the blocks individually to a separate file for analysis.