Lecture 26: Obfuscation

15411: Compiler Design
Robbie Harwood and Maxime Serrano

21 November 2013

1 Introduction

We have previously (lecture 20) considered the problem of doing compilation "backwards" (i.e., extracting a reasonable approximation of the original source from its compiled code). However, rather than analyzing our dataflow, we will be going one level beyond, and analyzing our analyses of dataflow.

The ethics of code obfuscation, DRM, and the like is a complicated topic, and we will not discuss it here. Regardless of one's ethical stance on the topic of obscurity, it is important as an proponent to understand the basic ideas behind implementing it, as an opponent to understand it well enough to defeat it, and as an intellectual to learn from it.

As alluded to above, if not the largest than certainly the most controversial application of code obfuscation is as a means to implement copy protection and Digital Rights Management Schemes (DRM). However, on occasion obfuscation can arise more organically. Hand-written assembly, despite being frowned upon as a software development process today, is still alarmingly prevalent within industry. Somewhat unfortunately, the less understandable snippets tend to be the most resistant to replacement, since writing code that performs the same function is difficult.

Finally, it is worth noting that these lecture notes are by no means the final word on obfuscation. Obfuscation is dependent on the workings of the tools used for reverse engineering, and so as they change, so to will the methods used for obfuscation. This lecture will in particular focus on defeating IDA Pro, `objdump`, and `gdb`.

2 Disassembly

The two disassembly tools, whlie they to some degree do a very similar job (converting machine code directly into assembly language) have very different philosophies about how one might do this. This is most likely due to the vastly higher degree of sophistication present in IDA, and to the effort that the authors of IDA make in order to defeat obfuscating compilers.

In a perfect world, disassembly is very similar. Much like an assembler can take every mnemonic to a single sequence of bytes, using a relatively simile one-to-one mapping, a disassembler ought to be able to read the sequence of bytes and decide which instructions map to those bytes. For example, `ret` maps to `C3`, so when a disassembler notices an instruction beginning with `C3`, it should produce a `ret` instruction.

There are, however, a few problems that appear. First of all, in modern executable file formats, the `.text` section that contains program code is also allowed to contain program data. While most program code is easy to disassemble, as long as no effort has been made to make it difficult, program *data* is very unlikely to disassemble to valid code - much less code that makes any sense. Therefore, modern disassemblers must be able to differentiate code from data, even when the entire block is marked as code.

Here is where the differences between the tools mostly lie. IDA attempts to solve this problem; `objdump` assumes your data is in the various data sections, and that `.text` is entirely made up of code. It should be noted that while in general, IDA makes the correct choice, there are ways to trick it as well. In particular, since the `00 00` bytes form a valid instruction (albeit `add %al, (%eax)`), IDA assumes that it is data. One can use this to trivially trick `objdump` by placing data in the `.text` section.

Both tools assume that a single sequence of bytes will only be executed in exactly one way. This is a wonderful efficiency gain for `objdump` in particular: rather than needing to do recursive-descent disassembly and "pretend" to execute the program, `objdump` can just do a single, straight-line disassembly pass and be done with it.

However, this causes a very significant issue the moment programs stop obeying this restriction. In particular, consider the sequence of instructions:

```
push %rax
xor %rax, %rax
.byte 0x74
.byte 0x01
.byte 0x0f
pop %rax
mov $r, %rax
```

Given this code, `gcc` happily produces the following sequence of bytes:

```
50 48 31 c0 74 01 0f 58 48 c7 c0 03 00 00 00
```

We can then run `objdump` on these bytes, and notice that the following is produced:

```
Disassembly of section .text:

0000000000000000 <.text>
   0:   50                      push %rax
   1:   48 31 c0                xor %rax, %rax
   4:   74 01                   je 0x07
   6:   0f 58 48 c7             addps -0x39(%rax), %xmm1
   a:   c0 03 00                rolb $0x0, (%rbx)
        ...
```

What has happened here is this: the bytes `74 01` lead to a jump that skips the following byte (here, `0f`), which happens to be the first byte of all two-byte opcodes. The presence of the `0f`, however, causes `objdump` to decide that the bytes following the jump must be a mapping From an instruction with a two-byte opcode (in this case, `addps`, a SIMD instruction). As the misalignment constinues (the `48 c7` bytes are the first two bytes of our `mov` instruction from the original assembly, but have been "eaten up" by the SIMD instruction), we continue to produce nonsensical assembly afterwards.

Now, a human reading the assembly can notice by the first three instructions that something fishy is going on, but unless they can read machine code, they will still have to modify the `0f` byte into something more like `90` in order to read correct disassembly.

One might expect IDA, which does not perform straight-line disassembly but rather uses a recursive-descent algorithm, to not be fooled by this trick. HOwever, IDA's recursive descent follows branch-not-taken before it follows branch-taken, and it also assumes that each sequence of bytes can only be disassembled one way. Therefore, when it follows branch-taken (which would produce correct output!) it notices that the target of the jump is marked as disassembled already and hence does not bothered to disassemble it correctly.

It should be noted, however, that the processor has no issues executing this code. Since `xor %rax, %rax` will *always* cause the jump to be triggered, furthermore, it does not execute the potentially fatal SIMD instruction (which dereferences `-0x39(%rax)`, which is probably not a good place).

3 Anti-Analysis

The anti-analysis techniques described here are mostly targeted at IDA Pro, though one could probably also adapt them to target CMU's BAP system.[1].

IDA's analysis capabilities are formidable. With 32-bit code, in fact, they are often (though not always) able to generate C code that preserves the semantics of the original code that was compiled. While the 64-bit version of IDA does not yet have decompilation support, it does have a variety of other analysis tricks up its sleeve.

One thing in particular that is exceedingly useful to reverse engineer is IDA's capacity to recognize function boundaries. However, different compilers tend to produce vastly different function boundary code!

In 32-bit code, the differences are relatively small. In particular, many Microsoft compilers produce code that follows the `stdcall` calling convention rather than the more standard `cdecl` convention, which causes end-of-function code to be more complicated. Similarly, the `fastcall` convention is different depending on platform and compiler.

In 64-bit code, however, there is a significant difference between code produced by compilers that follow the Microsoft calling conventions and the code produced by compilers that follow the System V calling conventions (such as `gcc`, `clang`, and others).

In order to support all of these different conventions, as users of IDA expect it to, IDA uses a variety of heuristics to decide where functions begin and end. One can exploit the characteristics of these heuristics in order to cause IDA to believe that functions end before they do, or simply refuse to believe that they end at all.

Other analyses performed by IDA involve stack variables. In order to confuse IDA here, one can temporarily confuse the stack pointer. As IDA depends on symbolic execution of the code being analyzed, and expects the stack pointer to remain at "sensible" values, this causes the analysis steps to fail entirely. It is possible to produce 32-bit code that fails entirely to be recompiled by IDA, even if the disassembly step succeeds.

Confusing the stack pointer and IDA's function boundary detection can be done by faking a sequence of instructions that are similar to those done on return From a function, and transforming the "normal" return sequence.

For example, the fake sequence:

```
push %rcx
push %rbx
push %rdx
.byte 0xe8
.byte 0x00
.byte 0x00
.byte 0x00
.byte 0x00
pop %rdx
add $08, %rdx
push %rdx
```

[1]More information on BAP can be found at `http://bap.ece.cmu.edu/`

```
ret
.byte 0x0f
pop %rdx
pop %rbx
pop %rcx
```

These bytes `e8 00 00 00` decode to `call $+0`, or simply "`push %rip`" (were such a thing allowed). This confuses IDA's stack pointer, since it resets the "relative" value of `%rsp` (which is all it keeps track of) upon executing a `call` instruction. Following it up with a `pop` causes the sign of the stack pointer adjustment to be something IDA fails to understand.

Further, the presence of the `ret` instruction so soon after a `pop` instruction causes some versions of IDA to believe that this is the end of a function. Unfortunately (or fortunately, depending on your perspective), there does not seem to be such a sequence for all versions.

Afterward, one can modify the real return sequence very slightly. Rather than ending functions with a `ret` instruction, one could end them with, say:

```
pop %r15
pop %r14
pop %r13
pop %r12
pop %rbx
pop %r11
pop %r10
pop %rdi
jmp *%rdi
```

IDA attempts to decide what it believes the indirect jump is, and often labels the function a "chunk" rather than a full function and does not list it in the functions list.

To understand what this means, some understanding of IDA's internals is required. The way the function detection works appears to be as follows:

1. Notice a standard function-beginning sequence.
2. "Evaluate" the code from the start (except backwards edges) until every piece of control flow reaches a function end sequence.
3. If the pieces of the function are spread across memory, and don't end in return statements, label them as "chunks".

If one can cause IDA to recognize a given piece of code as a chunk that has no corresponding function, then it analyzes it and promptly tosses out its analysis.

4 Anti-Debugging

Often, when a reverse engineer cannot figure out the details of the program statically (either due to massive program scale, unreadable assembly, or any of a variety of other factors), he or she will attempt to do so dynamically, observing the program's behavior as it runs. Debuggers such as `gdb` are a critical part of this effort. Therefore, any full anti-reverse-engineering effort would need to have some anti-debugging component.

The primary method for anti-debugging is to abuse the fact that each program can only be debugged by one debugger. To use this, we inserted a `ptrace()` call into the program that causes the program's parent to attach itself to it as a debugger. If the program already has a debugger attached, this call fails and the program segfaults.

This is perhaps the simplest anti-debugging mechanism, and can be countered by a simple modification of the binary, by very careful use of the debugger, or by interposing the `ptrace()` call. However, in combination with anti-disassembly it is generally enough to delay reverse engineers for a significant amount of time, especially since the `ptrace()` call can be well-hidden.

A more sophisticated trick is as follows:

1. The program begins, and before calling `main()`, calls a *constructor* function `c()`. The list of such functions is held in the `ctors` section, and can be user-controlled.
2. This constructor forks the program. One fork is marked as the child, and the other is marked as the parent.
3. The parent `ptrace()`s the child, and the child `ptrace()`s the parent.
4. Each inserts a critical structure into the memory of the other, or modifies the other's code, or alters the `main()` function to be a no-op, while inserting the address of the "real" main into the `dtors` section to be called at exit.
5. One of the two runs the program as "normal".

This trick (circular debugging) is very difficult to get around. It is, however, also very difficult to implement at the compiler level. In general, mitigating this anti-debugging technique involves writing a custom loader for the target program, and loading the structure into memory oneself. In general, understanding what the structure is entails a full static analysis of the original program, effectively defeating the entire purpose of debugging in the first place.

The last technique is trivial both in implementation and in mitigation: forced breakpoints. The `int 3` instruction, on most platforms, forces a breakpoint. Inserting `int 3` in various locations in the program would cause the reverse engineer to need to manually deal with vastly more breakpoints than desired (especially in tight loops) and hence hinder his or her debugging efforts. However, simply replacing all instances of that instruction with a `nop` instruction trivially defeats this technique, so it is mostly just a nuisance to any reverse engineer.

5 Other techniques

(This section was not reached in lecture since we did a demo.)

5.1 SSA

Obfuscation interacts quite well with SSA (and basic block analysis), since one can then use it to do IR-level hiding of information. To do this, one could add extra unnecessary control flow nodes, without losing *too* much in performance. Further, given the basic block information, one can then reorder blocks in the binary, making reading disassembler output an exercise in scrolling.

5.2 Packers

A technique that has become more prevalent in recent years is the use of packers. The general idea here is to keep a compressed or encrypted version of the binary on the disk ("packed"), as well as a small loader stub which can then decrypt/decompress ("unpack") the rest of the binary in memory. Advanced packers can even unpack individual functions or basic blocks one at a time, and then re-pack them when they are no longer running. This completely defeats static analysis, as the bytes that make up the target program are quite simply not decipherable until they are unpacked.

Packing is generally defeated by dumping the contents of memory at runtime, though the specific implementation of this technique can make memory dumps much harder. Packing has perhaps the highest return of any anti-reverse-engineering technique, but is itself very time-consuming to write.

5.3 Calling conventions

Per-function calling conventions are the extreme of making human analysis difficult, as then the human would need to keep track of which function takes which convention. From an implementation perspective, however, it requires quite a lot of information be passed around at compile time and instruction selection time to make it workable, and so is less viable than some other techniques.

5.4 Program bugs

The use of bugs in `objdump` and IDA can make them entirely useless. While writing our obfuscating compiler, for instance, we successfully produced a binary that caused `objdump` to segfault upon attempted disassembly. (It's not exploitable as far as we could tell.) However, such transformations can be unreliable in implementation, since they require the construction of decidedly odd corner cases.

6 Want more?

The biggest problem with writing obfuscating compilers is debugging the compilers themselves. In effect, the author is attempting to shoot himself or herself in the foot by preventing debugging while demanding correctness, since determining why correctness was not maintained requires debugging. In effect, it requires the author to mitigate their own anti-reverse-engineering transformations in order to reverse-engineer their own program.

Finally, we should mention that if reversing and code obfuscation are topics you find interesting, the PPP Security Research Group and Hacking Team may be for you. For more information, visit our webiste `http://pwning.net`, sign up for our mailing list, or join our IRC channel `irc://freenode.net/#pwning`. Our faculty advisor is Professor David Brumley; more information on his projects can be found at his website `https://users.ece.cmu.edu/~dbrumley/`.