Obfuscation through randomness and parsing errors

Sebastiaan Peters

***Abstract*—**

I. Context

Obfuscation serves as a way to prevent distributed programs from being understood by unwanted actors. A legal use case of this would be protecting intellectual property. These same methods often also get used by malware writers to prevent their work from being analyzed by either a human or automated tools. Therefore it is vital to understand possible methods of obfuscation to counter act malicious use of obfuscation.

II. Background

Disassemblers are the corner stone of analysis for a program which its source has not been distributed. Typically they use one of two main methods to parse a binary.
When a disassembler uses a linear sweeping technique, they start parsing at the beginning of the $.text$ section and parse the entire section instruction for instruction, without any regard for which instruction it parses. In contrast with a recursive traversal algorithm typically a disassembler starts at one or more known entry points, and follows flow control which it parses.

For a long time disassemblers worked on the assumption that a byte can only be used for a single instruction, when this is not the case we will say that 2 or more instructions overlap. In the early days this was justified after Lin and Saumya showed that typically the CF (confusion factor) would lay around 1% [1], where the confusion factor is the percentile of instructions that would be disassembled incorrectly. The main reason for this being that even if 2 instructions would overlap, they would quickly converge to the same instruction sequence again. This is mainly due to how small x86-64 instructions in compiled binaries tend to be.

Since then work has gone into embedding instructions within each other to leave disassemblers that worked on this assumption confused. This is typically done in one of 3 ways:

1) Pre-pending a byte to an instruction and altering flow-control S.T the new byte would never be executed
2) Embedding instructions in the operands of long form nop instructions
3) Jumping into the middle of an instruction

However each one of those approaches leave something to be desired.
The approach of pre-pending bytes to an instruction is effective against linear sweepers, however it is costly to obfuscate a large chuck instructions and without using opaque predicates it is still vulnerable to recursive traversal disassemblers.

Embedding instructions in long nop instructions induces an enormous overhead, namely 9-11 bytes for a maximum of 4 bytes obfuscated per embedding[7] (meaning no 5+ byte instruction can be embedded), and due to the repetition it is also easy to spot by humans.

Jumping into the middle of instruction can be effective, however unless a lot of care has been taken it would fall prey to the same problems described earlier. This technique is typically used in small sections which heavy self modification is applied to [2].

While no general answer has been found for overlapping instructions have been found for disassemblers that use linear sweeping, Codisasm [2] have extended the general recursive traversal algorithm to include layers, S.T there would still be an accurate representation in the case of overlapping instructions.

III. Contribution

The aim of this paper is to contribute the following two things:

1) Show that it is feasible to create 70%[1] CF or higher based on overlapping instructions against linear sweepers
2) Introduce a novel technique to completely stop recursive traversal disassemblers

The first approach has an rather un-intuitive goal. Instead of making it harder to understand the output of the disassembler like many obfuscation techniques discussed by Cohen [3], we try to give the illusion that the program is not malicious while, when the binary is run, something else gets executed than what the disassembler shows as output. This requires greater control of the obfuscation that current techniques do not account for.

For our first finding we use a research tool called STOKE [4] that is meant to optimize existing binaries. Therefore it must be able to analyze the behavior and keep it intact, while changing instructions seemingly at random. This gives us a platform to search for binary representations of programs that exhibit a large quantity of overlapping instruction. Using this method we also gain tooling to quickly gain access to some of the redundancy within the x86-64 instruction set. Something that is of heavy need when attempting to obfuscate binaries in such a way. While STOKE can show us that it can be feasible with proper tooling, it still will require a lot of development (or augmentation of STOKE) to fully realize the potential of this method.

[1]This is around an $\sim 30\%$ increase over current methods

Our second finding depends on parsing bugs of disassemblers, no literature discusses on how to use such bugs and as of yet only a handful of studies concerns themselves with the testing of the correctness of disassemblers. However the fact that x86-64 instruction set is still getting extensions, combined with that Paleari, Roberto, et al[5] and Domas[6] showed that there are still bugs in current day disassemblers, leave me to believe that we will never life in a world where there will no parsing bugs.

An interesting property about wrongly parsed instructions is that most recursive traversal disassemblers stop completely (as opposed to linear sweepers). This allows us to hide nearly all of our instructions from automated analysis tools unless they identify instruction blocks through other means. However those can be quite erroneous (citation needed?).
While placing an instruction which gets parsed wrongly at the entry point would be heavily effective against automated analysis tools, a human observer would be able to easily remedy the situation and could still easily understand the program if he would change the instruction to any other. However carefully inserting these instructions could also serve as a method to silently stop the disassembly of any function, which could silently hide edges within a control flow graph and could easily go unnoticed by a human.

IV. Current results

Through STOKE I have been able to find an example of an binary with 70 bytes (20-22 instructions) overlapping. Although it must be stated that while transforming an existing function into something that inhibited this property, some of the correctness has been lost. At the end of this document are 2 outputs are given, one being from objdump and the other being from intel's XED. Both of which used linear sweeping.
I have found multiple parsing errors in nearly every major disassembler available for Linux (including ghidra, capstone (radare2) and objdump).

V. FUTURE WORK

While STOKE opened the door for finding binaries that contain a large amount of overlap between instructions, it has the problem that it can only obfuscate one function at a time. So far I have not been able to let it obfuscate functions that contain:

1) system calls
2) function calls
3) complex flow-control logic

no tooling as of yet is designed to find and explore this through the use of the redundancy of the x86. In addition it might be beneficial to take a closer look at how disassemblers differ from each other with regard to error recovery and coverage of the instruction set.

A major problem when generating the binaries is that of finding and selecting possible candidate instructions that fits the requirements of both being a valid instruction for both streams, and in addition to that we want it have a desirable side effect. In the later stages of my research I have become more and more convinced that it would be possible to model our problem as complex geometric objects. This would allow us to reduce the problem to, for example, finding intersection high dimensional planes, these would most likely have a heavily periodic form with regards to form (byte length of an instruction) while the side effects might be reduced to find a path in space.

This research thus far has only concerned it self with linear sweepers and recursive traversal disassemblers but it in recent years other methods are discussed in literature. It might be fruitful to also look at how the results would do in such a context.

References

[1] Linn, Cullen, and Saumya Debray. "Obfuscation of executable code to improve resistance to static disassembly." Proceedings of the 10th ACM conference on Computer and communications security. ACM, 2003.

[2] Bonfante, Guillaume, et al. "CoDisasm: medium scale concatic disassembly of self-modifying binaries with overlapping instructions." Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM, 2015.

[3] Cohen, Frederick B. "Operating system protection through program evolution." Computers Security 12.6 (1993): 565-584.

[4] Schkufza, Eric, Rahul Sharma, and Alex Aiken. "Stochastic superoptimization." ACM SIGPLAN Notices. Vol. 48. No. 4. ACM, 2013.

[5] Paleari, Roberto, et al. "N-version disassembly: differential testing of x86 disassemblers." Proceedings of the 19th international symposium on Software testing and analysis. ACM, 2010.

[6] Domas, Christopher. "Breaking the x86 ISA." Black Hat, USA (2017).

[7] Jmthagen, Christopher, Patrik Lantz, and Martin Hell. "A new instruction overlapping technique for anti-disassembly and obfuscation of x86 binaries." 2013 Workshop on Anti-malware Testing Research. IEEE, 2013.

```
0f                      prefetch (bad)
0d c0 90 90 31          or     $0x319090c0,%eax
ff 81 ff 80 ff ff       incl   -0x7f01(%rcx)
ff 48 b8                decl   -0x48(%rax)
c0 ff ff                sar    $0xff,%bh
00 00                   add    %al,(%rax)
00 00                   add    %al,(%rax)
ff c5                   inc    %ebp
fc                      cld
77 f7                   ja     4004e7 <main+0x1a>
e7 3c                   out    %eax,$0x3c
05 3d f9 ff ff          add    $0xfffff93d,%eax
ff 83 ef fb 90 e8       incl   -0x176f0411(%rbx)
b5 ff                   mov    $0xff,%ch
ff                      (bad)
ff 89 45 f8 83 7d       decl   0x7d83f845(%rcx)
f8                      clc
05 7e 07 b8 02          add    $0x2b8077e,%eax
00 00                   add    %al,(%rax)
00 eb                   add    %ch,%bl
05 b8 01 00 00          add    $0x1b8,%eax
00 c9                   add    %cl,%cl
c3                      retq
```

Fig. 1. Output objdump

```
0F0DC0                  nop eax, eax
90                      nop
90                      nop
31FF                    xor edi, edi
81FF80FFFFFF            cmp edi, 0xffffff80
48B8C0FFFF00000000FF    mov rax, 0xff00000000ffffc0
C5FC77                  vzeroall
F7E7                    mul edi
3C05                    cmp al, 0x5
3DF9FFFFFF              cmp eax, 0xfffffff9
83EFFB                  sub edi, 0xfffffffb
90                      nop
E8B5FFFFFF              call 0x4004b6 <calc>
8945F8                  mov dword ptr [rbp-0x8], eax
837DF805                cmp dword ptr [rbp-0x8], 0x5
7E07                    jle 0x400511 <main+0x44>
B802000000              mov eax, 0x2
EB05                    jmp 0x400516 <main+0x49>
B801000000              mov eax, 0x1
C9                      leave
C3                      ret
```

Fig. 2. Output xed