Return-into-libc without Function Calls (on the x86)

Peng Xu

October 29, 2018
Table of Contents

- Introduction
- Problem
- Design
- Implementation
- Conclusion
Introduction

- Software-development with C/C++
- Memory Corruption
  - Stack overflow
  - Buffer overflow
- Code Injection Attacks
- Code Reuse Attacks
Introduction

- Software-development with C/C++
- Memory Corruption
  - Stack overflow
  - Buffer overflow
- Code Injection Attacks
- Code Reuse Attacks
Introduction - Code Injection

- Function-level
- External code injecting
Introduction - Code Injection

- Function-level
- External code injecting
Introduction - Code Reuse

- Function-level
- Internal code reuse
Introduction - Code Reuse

- Function-level
- Internal code reuse
Problem

- Removing certain functions from libc
- Changing the assembler’s code generation choices
- Defense against code reuse attacks
Design

- Return-oriented programming - ROP
- Instruction-level - gadgets
- Discovering useful instructions sequences in Libc
Design

- Return-oriented programming - ROP
- Instruction-level - gadgets
- Discovering useful instructions sequences in Libc
  - Useful code sequence
  - Ending with a ret instruction
  - Boring instructions
Design - ROP

Program 1

Module 1

Module 2

Module i

Module n

Func j

Mov eax, ebx
Inc eax
...
Ret

Func 1

Pop edx; ...
Ret

Func 2

Mov edx, edx
Add eax, ecx
Pop edx
Inc ebx
...

Func j

Mov eax, ebx; pip ebx; ret

Func m
Algorithm GALILEO:
create a node, root, representing the ret instruction;
place root in the trie;
for pos from 1 to textseg_len do:
    if the byte at pos is c3, i.e., a ret instruction, then:
        call BUILDFROM(pos, root).

Procedure BUILDFROM(index pos, instruction parent_insn):
for step from 1 to max_insn_len do:
    if bytes [(pos – step)…(pos – 1)] decode as a valid instruction in 
    ensure insn is in the trie as a child of parent_insn;
    if insn isn’t boring then:
        call BUILDFROM(pos – step, insn).

Figure 1: The GALILEO Algorithm.
Gadget - Load/Store

Loading a Constant

```
pop %reg; ret
```

![Diagram showing the gadget process with `pop %edx` and `ret` actions.]
Gadget - Load/Store
Loading from Memory

```assembly
movl 64(%eax), %eax; ret
```

```
+64
0xdeadbeef
```
movl %eax, 24(%edx); ret
Gadget - Arithmetic and Logic

Add

```
addl (%edx),%eax; push %edi; ret
```
Gadget - Arithmetic and Logic

Shifts and Rotates

```
roll %cl, 0x17383f8(%ebx);ret
```
Gadget - Control Flow

Unconditional Jumps

changing the value of %esp to point to a new gadget
pop %esp; ret
Phase One: Clear CF if %eax is zero, set CF if %eax is nonzero.
Gadget - Control Flow

Conditional Jumps

Phase Two: Store either 1 or 0 in the data word labeled "CF goes here," depending on whether CF is set or not.

```
movl %ecx, (%edx)
adc %cl, %cl
ret
```

```
0x00000000
```

```
%esp
```

```
(pop %ecx
pop %edx
ret
```

```
(CF goes here)
```
Phase Three: part one: Convert the word (labeled “CF here”) containing either 1 or 0 to contain either esp delta or 0. The data word labeled 0xbadc0ded is used for scratch.
Phase Three: two: Apply the perturbation in the word labeled “perturbation here” to the stack pointer. The perturbation is relative to the end of the gadget.
Gadget - System Calls

System call's number

- lcall %gs:0x10(0)
  - ret
- (call index)
  - pop %esp
  - ret
- pop %eax
  - ret
- %esp
Implementation

Buffer overflow

- Buffer overflow vulnerability
- No randomization
- No stack-protector
Implementation

Steps

- @.data (@ of .data for to place some strings)
- int $0x80 (for execute our payload)
- mov %eax,(%ecx) — pop %ebp — ret (for mov eax into buffer)
- inc %eax — ret (for increment eax to up to 11)
- pop %edx — pop %ecx — pop %ebx — ret (for pop address)
- pop %eax — pop %ebx — pop %esi — pop %edi — ret (here just pop %eax will be useful)
- xor %eax,%eax — ret (for put %eax to zero)
Questions?