The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)

Enrico Galli

Introduction

- Background
- The Attack
- Return-Oriented Programming
- Avoding spurious ***ret***s
- Conclusion

Background

- Buffer-overflows
- Defenses against buffer-overflows (W⊕X)
- Traditional return-to-libc attack

Buffer-overflow

- Abuse unsafe versions of string functions
- Override return pointer and inject code

Defenses against buffer-overflows (W⊕X)

- Non-executable stack
- W⊕X
- Supported by Linux(PaX patches), Windows(since XP SP2)

Traditional return-to-libc attack

- Use existing function
- Chain multiple function calls

Why implement W⊕X?

- Limited execution to straight-line code
- Only use existing functions

The Attack

- Purpose/Thesis
- How does it work
- What are gadgets and why use them
- Previous use of short code sequences
- Finding gadgets

Purpose/Thesis

In any sufficiently large body of x86 executable code there will exist sufficiently many useful code sequences that an attacker who controls the stack will be able, by means of the return-into-libc techniques we introduce, to cause the exploited program to undertake arbitrary computation.

How does it work

- Traditional return-to-libc
- Gadgets rather than functions

What are gadgets and why use them

- Collection of short sequences of instructions ending in *ret* (ex. pop %eax; ret;)
- x86 ISA is very dense
  - Explicit(ret[c3] added by the compiler)
  - Implicit
    - Part of other op codes(add imm32, %ebx, 81 c3)
    - Part of displacement

Previous use of short code sequences

- Use short sequences as glue
- Set function arguments in traditional return-to-libc(args on reg calling conventions)

Finding gadgets

- Galileo Algorithm
  - Scan libc for c3(ret)
  - Build trie(prefix tree) from c3 in reverse
- Remove ”boring” sequences
  - Standard exits(leave; ret;)
  - Return or unconditional jmp

Return-Oriented Programming

- Load/Store
- Arithmetic and Logic
- Control Flow
- System Calls
- Function Calls

Load/Store

- Loading a constant
  - pop %reg; ret;
- Loading from memory
  - movl 64(%eax), %eax; ret
- Storing to memory
  - movl %eax, 24(%edx); ret

Arithmetic and Logic

- Add
  - addl (%edx), %eax; push %edi; ret
  - Push causes problem
- No multiplication
- Xor, And, Or, Not
  - Libc doesn't contain clean version.
  - xorb %al, 0x48908c0(%ebx); and $0xff, %al;push %ebp; or $0xc9, %al; ret.

Control Flow

- Unconditional Jump
  - pop %esp; ret
- Conditional Jumps
  - Use memory location to change esp
  - Change memory location based on condition

System Calls

- Use any of the simple wrappers in libc
- Jump directly to __kernel_vsyscall

Function calls

- Traditional return-to-libc
- Frame pointer to part not used

Avoding spurious *ret*s

- Single exit point
- Use other reg instead of ebx
  - Avoid add imm32, %ebx
- Unable to remove unintentional sequences

Conclusion and Future work

- Able to run arbitrary code using gadgets
- Backend for GCC
- Other platforms

Questions?