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

Peng Xu
peng@sec.in.tum.de

Chair for IT Security / I20
Prof. Dr. Claudia Eckert
Technical University of Munich
Introduction

Useful instruction sequences in Libc

Return-oriented Programming

Return-oriented Shellcode

Conclusion and Future works
Introduction

- Code injection attacks
- $W \oplus X$ defense
- Return-into-libc
- Remove certain functions from libc
- Change the assembler’s code generation
- Return Oriented Programming
- Gadgets - A set of short instructions
Background: Attacks and Defenses

- **Exploitation** - subverts the program’s control flow
  - Buffer overflow on the stack/heap
  - Integrity overflow
  - Format String

- **Attacks**
  - Find some way to subvert the program’s control flow
  - Cause the program to act in the manner of attacker’s choosing

- **Code Inject attacks:** return to the injected code
- **Code Reuse attacks:** return to the existed code

- **Defenses**
  - Defenses against the first step: shadow stack, stack canaries
  - Defenses against the second step: $W \oplus X$, ASLR
Useful instruction sequences in Libc

- What’s the useful instruction sequence?
- What are the “boring” instructions?
  - leave ; ret ;
  - pop %ebp; ret;
  - a return or an unconditional jump
- GALILEO algorithm: find useful instruction sequences
Return-oriented Programming

- **What is the Gadget?**
  - Each gadget is an Intermediate organizational unit
  - Each gadget specifies certain values to be placed on the stack
  - Each gadget could be found in libc
  - Each gadget performs well-defined operations, such as load, an xor, or a jump

- **Return-oriented programming**
  - ROP consists in putting gadgets together that will perform the desired operations
  - ROP is Turing complete by inspection

- **How to enter into Gadget?**
  - The processor executes a `ret` with the stack pointer, `%esp`, pointing to the bottom word of the gadget, or
  - The first gadget should be placed so that its bottom word overwrites some functions’ saved return address on the stack
Return-oriented Programming - Load/Store

- Loading a constant into a register
  - `pop %reg;ret`: incremental %esp firstly, and then pop stack word into %reg

- Loading the contents of a memory location into a register
  - `pop %eax;ret; movl 64(%eax),%eax;ret`: incremental %esp firstly, and then load the address into %eax, and pop stack word into %eax

- Writing the contents of a register into a memory location
  - `pop %edx;ret; movl %eax,24(%edx);ret`: Load the address to be written into %edx using the constant-load method, and then move value in %eax to 24(%edx)
Return-oriented Programming - Arithmetic and Logic

- For all operations, one operand is %eax; the other is memory location
- Depending on what is more convenient, either %eax or memory location receives the computed value
- Simple Add
  - `pop %edi;ret`
  - `pop %edx;ret`
  - `addl (%edx),%eax; push %edi; ret`
- Repeatable add
  - `pop %ecx;pop %edx; ret`
  - `addl (%edx),%eax; push %edi; ret`
  - `movl %ecx,(%edx);ret`
  - `pop %edi;ret`
  - `pop %edx;ret`
Return-oriented Programming - Arithmetic and Logic

- Other Arithmetic Operations: `neg %eax;ret`
- Exclusive Or:
  - `pop %ebx;ret;`
  - `xor %al,0x48908c0(%ebx);and $0xff,%al;push %ebp;or $0xc9,%al;ret;`
  - `ror $0x08,%eax;ret;`
- And, Or, Not
  - `andb %al,0x5d5e0cc4(%ebx);ret`
  - `orb %al,0x40e4602(%ebx);ret`
- Shifts and Rotates:
  - `pop %ebx;ret;`
  - `pop %ecx;%Sedx;ret;`
  - `roll %cl;0x017383f8(%ebx);ret`
Return-oriented Shellcode

invokes a system call to run a shell

Syscalls

- Making a syscall: **int $0x80** or **syscall** or **swi 0x0**
  - Syscalls’ number: eax/rax/r7
  - Syscalls’ parameters
    - Param 1: ebx/rdi/r0
    - Param 2: ecx/rsi/r1
    - Param 3: edx/rdx/r2
    - Param 4: esi/r10/r3
    - Param 5: edi/r8/r4
    - Param 6: ebp/r9/r5
    - Param 7: -/r6
  - Return value: eax/rax/r0
Return-oriented Shellcode

- `11` `sys_execve(const char *filename, const char *const argv[], const char *const envp[]) [X86]`
- invokes the `execve` system call to run a shell
  - Setting the system call index, in `%eax`, to `0xb`
  - Setting the path of the program to run, in `%ebx`, to the string `/bin/sh`
  - Setting the argument vector `argv`, in `%ecx`, to array of two pointers, the first of which points to the string `/bin/sh` and the second of which is null
  - Setting the environment vector `envp`, in `%edx`, to an array of one pointer, which is null
Conclusion and Future works

- Present a new way of organizing return-into-libc exploits on the x86
- Combine short instruction sequences into gadgets that allow an attacher to perform arbitrary computation.
- Towards greater automation and integration with existing technologies
- Attempt to validate (or invalidate) by examining C libraries in other platforms.
Suggestion and Questions

Suggestions?
Q&A?