18-600: Recitation #4
Exploits

20th September 2016

Agenda

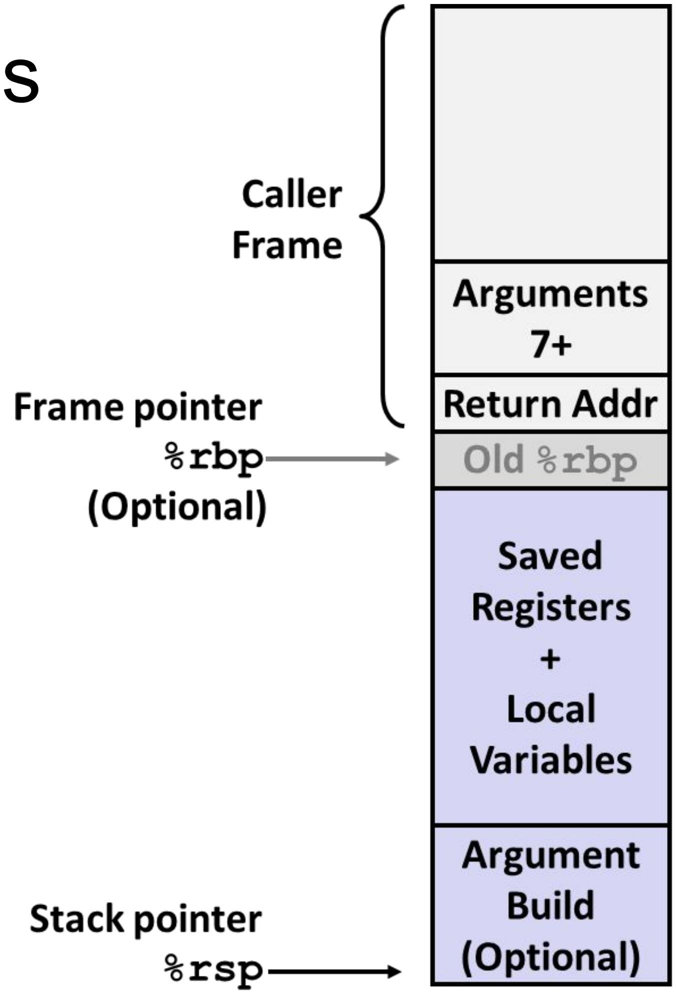
- More x86-64 assembly
- Buffer Overflow Attack
- Return Oriented Programming Attack

Recap: x86-64: Register Conventions

- Arguments passed in registers:
  `%rdi, %rsi, %rdx, %rcx, %r8, %r9`
- Return value: `%rax`
- Callee-saved: `%rbx, %r12, %r13, %r14, %rbp, %rsp`
- Caller-saved: `%rdi, %rsi, %rdx, %rcx, %r8, %r9, %rax, %r10, %r11`
- Stack pointer: `%rsp`
- Instruction pointer: `%rip`

Recap: x86-64: Stack Frames

- Every function call has its own **stack frame**.
- Think of a frame as a workspace for each call.
  - Local variables
  - Callee & Caller-saved registers
  - Optional arguments for a function call



Recap: x86-64: Function Call Setup

Caller:

- Allocates stack frame large enough for saved registers, optional arguments
- Save any caller-saved registers in frame
- Save any optional arguments (in **reverse order**) in frame
- `call foo`: push `%rip` to stack, jump to label `foo`

Callee:

- Push any callee-saved registers, decrease `%rsp` to make room for new frame

Recap: x86-64: Function Call Return

Callee:

- Increase `%rsp`, pop any callee-saved registers (in **reverse order**), execute `ret: pop %rip`

Control Hijacking

Buffer Overflow

- Exploit x86-64 by overwriting the stack
- Overflow a buffer, overwrite return address
- Execute injected code

Strcpy Vulnerability

```
int main(int argc, char *argv[]){
    foo(argv[1]);
    ...
}

void foo(char *input){
    char buf[32];
    ...
    strcpy (buf, input);
    return;
}
```

What is the potential issue with this program?

Buffer Overflows

- Exploit *strcpy vulnerability* to overwrite important info on stack
- When this function returns, where will it begin executing?
  - Recall

    `ret:pop %rip`
- What if we want to inject new code to execute?

Old Return address → 0xAABBCCDD

0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF

buf → 0xFFFFFFFF

Generating Byte Codes

- Use **gcc** and **objdump** to generate byte codes for assembly instruction sequences

```
mov     $0x497284,%edi
mov     $0x3b,%eax
syscall
```

exploit.s

```
gcc -c -o exploit.o exploit.s
              &
objdump -d exploit.o > exploit.txt
```

Values in little endian

```
0:   bf 84 72 49 00        mov     $0x497284,%edi
5:   b8 3b 00 00 00        mov     $0x3b,%eax
10:  0f 05                 syscall
```

exploit.txt

Buffer Overflows

- Exploit *strcpy vulnerability* to overwrite important info on stack
- When this function returns, where will it begin executing?
  - Recall

    `ret:pop %rip`
- What if we want to inject new code to execute?

New Return address

0xEEFF0000
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0x050F0000
0x003BB800
0x497284BF

buf

0xEEFF0000

Advanced Control Flow Hijacking

- What if the stack addresses are randomized at runtime, so that buf is always in a different place?
  - Potential solution: “nop” slide
- What if a “canary” or secret value has been placed at the end of the buffer, so that the program knows when it has been tampered with?
  - Potential solution: get canary value when generated, to fool checks
- Non-executable memory, e.g. DEP/NX
  - Potential solution: return oriented programming, no code on stack required!

Return Oriented Programming

Overview

- Utilize return-oriented programming to execute arbitrary code
  - Useful when stack is non-executable or randomized
- Find gadgets, string together to form injected code

Key Advice

- Use mixture of pop & mov instructions + constants to perform specific task

ROP Example

- Draw a stack diagram and ROP exploit to **pop a value 0xBBBBBBBB into %rbx** and **move it into %rax**

```
void foo(char *input){
    char buf[32];
    ...
    strcpy (buf, input);
    return;
}
```

Gadgets:

address_1: mov %rbx, %rax; ret

address_2: pop %rbx; ret

Inspired by content created by Professor David Brumley

ROP Example: Solution

Gadgets:

Address 1: mov %rbx, %rax; ret

Address 2: pop %rbx; ret

```
void foo(char *input){
   char buf[32];
   ...
   strcpy (buf, input);
   return;
}
```

Next address in ROP chain….

Address 1

0xBBBBBBBB

Old Return address → **Address 2**

0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
buf → **0xFFFFFFFF (filler…..)**

Looking for Gadgets

- How to identify useful gadgets in your code

```
000000000000013b <some_glibc_fn1>:
 13b:  55                   push   %rbp
 13c:  48 89 e5             mov    %rsp,%rbp
 13f:  48 89 7d f8          mov    %rdi,-0x8(%rbp)
 143:  48 8b 45 f8          mov    -0x8(%rbp),%rax
 147:  c7 00 48 89 e0 c3    movl   $0xc3e08948,(%rax)
 14d:  5d                   pop    %rbp
 14e:  c3                   retq
```

```
000000000000007c <some_glibc_fn2>:
 7c:  55                   push   %rbp
 7d:  48 89 e5             mov    %rsp,%rbp
 80:  48 89 7d f8          mov    %rdi,-0x8(%rbp)
 84:  48 8b 45 f8          mov    -0x8(%rbp),%rax
 88:  c7 00 c2 6a 30 58    movl   $0x58306ac2,(%rax)
 8e:  5d                   pop    %rbp
 8f:  c3
```

```
00000000000000b4 <setval_341>:
 b4:  55                   push   %rbp
 b5:  48 89 e5             mov    %rsp,%rbp
 b8:  48 89 7d f8          mov    %rdi,-0x8(%rbp)
 bc:  48 8b 45 f8          mov    -0x8(%rbp),%rax
 c0:  c7 00 cf 08 89 e0    movl   $0xe08908cf,(%rax)
 c6:  5d                   pop    %rbp
 c7:  c3
```

```
89 e0 : movl %esp, %eax
5d:     pop %rbp
c3:     ret
```

A. Encodings of `movq` instructions

`movq` S, D

Source S	Destination D							
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
%rax	48 89 c0	48 89 c1	48 89 c2	48 89 c3	48 89 c4	48 89 c5	48 89 c6	48 89 c7
%rcx	48 89 c8	48 89 c9	48 89 ca	48 89 cb	48 89 cc	48 89 cd	48 89 ce	48 89 cf
%rdx	48 89 d0	48 89 d1	48 89 d2	48 89 d3	48 89 d4	48 89 d5	48 89 d6	48 89 d7
%rbx	48 89 d8	48 89 d9	48 89 da	48 89 db	48 89 dc	48 89 dd	48 89 de	48 89 df
%rsp	48 89 e0	48 89 e1	48 89 e2	48 89 e3	48 89 e4	48 89 e5	48 89 e6	48 89 e7
%rbp	48 89 e8	48 89 e9	48 89 ea	48 89 eb	48 89 ec	48 89 ed	48 89 ee	48 89 ef
%rsi	48 89 f0	48 89 f1	48 89 f2	48 89 f3	48 89 f4	48 89 f5	48 89 f6	48 89 f7
%rdi	48 89 f8	48 89 f9	48 89 fa	48 89 fb	48 89 fc	48 89 fd	48 89 fe	48 89 ff

B. Encodings of `popq` instructions

Operation	Register R							
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
popq R	58	59	5a	5b	5c	5d	5e	5f

C. Encodings of `movl` instructions

`movl` S, D

Source S	Destination D							
	%eax	%ecx	%edx	%ebx	%esp	%ebp	%esi	%edi
%eax	89 c0	89 c1	89 c2	89 c3	89 c4	89 c5	89 c6	89 c7
%ecx	89 c8	89 c9	89 ca	89 cb	89 cc	89 cd	89 ce	89 cf
%edx	89 d0	89 d1	89 d2	89 d3	89 d4	89 d5	89 d6	89 d7
%ebx	89 d8	89 d9	89 da	89 db	89 dc	89 dd	89 de	89 df
%esp	89 e0	89 e1	89 e2	89 e3	89 e4	89 e5	89 e6	89 e7
%ebp	89 e8	89 e9	89 ea	89 eb	89 ec	89 ed	89 ee	89 ef
%esi	89 f0	89 f1	89 f2	89 f3	89 f4	89 f5	89 f6	89 f7
%edi	89 f8	89 f9	89 fa	89 fb	89 fc	89 fd	89 fe	89 ff

D. Encodings of 2-byte functional nop instructions

Operation	Register R			
	%al	%cl	%dl	%bl
andb R, R	20 c0	20 c9	20 d2	20 db
orb R, R	08 c0	08 c9	08 d2	08 db
cmpb R, R	38 c0	38 c9	38 d2	38 db
testb R, R	84 c0	84 c9	84 d2	84 db

Figure 3: Byte encodings of instructions. All values are shown in hexadecimal.

Summary

- Attack lab is posted for fun on Autolab
- TAs will be *very pleased* if you attempt it
- We expect you to be familiar with this content at a high level

Questions?