Attack Lab

CSE251, Spring 2019

Recitation 1: Wed, March 13th, 2019

Changmin Yi

ulistar93@unist.ac.kr

Reference : CMU 15-213: Intro to Computer Systems Fall 2015
Recitation 3 - Dhruven Shah, Ben Spinelli

Agenda

- Stack review
- Attack lab overview
  - Phases 1-3: Buffer overflow attacks
  - Phases 4-5: ROP attacks

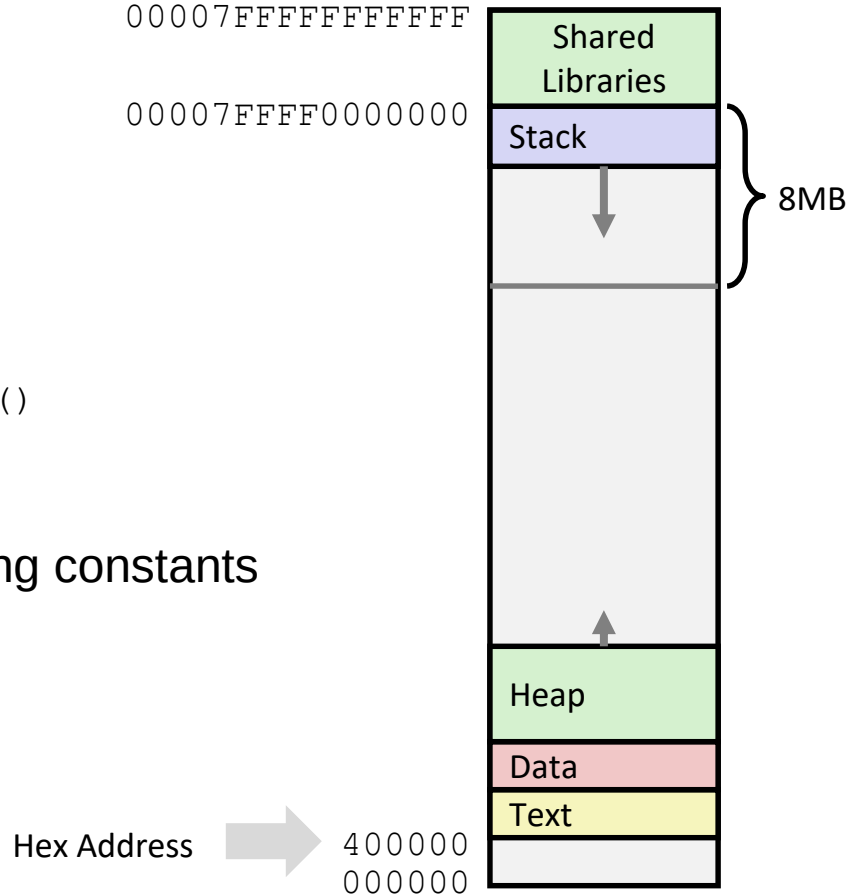
Notices

- Typo in pdf,
  Due date is **Apr 8. Monday.**

x86-64 Linux Memory Layout

not drawn to scale

- Stack
  - Runtime stack (8MB limit)
  - E. g., local variables
- Heap
  - Dynamically allocated as needed
  - When call `malloc(),calloc(),new()`
- Data
  - Statically allocated data
  - E.g., global vars, `static` vars, string constants
- Text / Shared Libraries
  - Executable machine instructions
  - Read-only

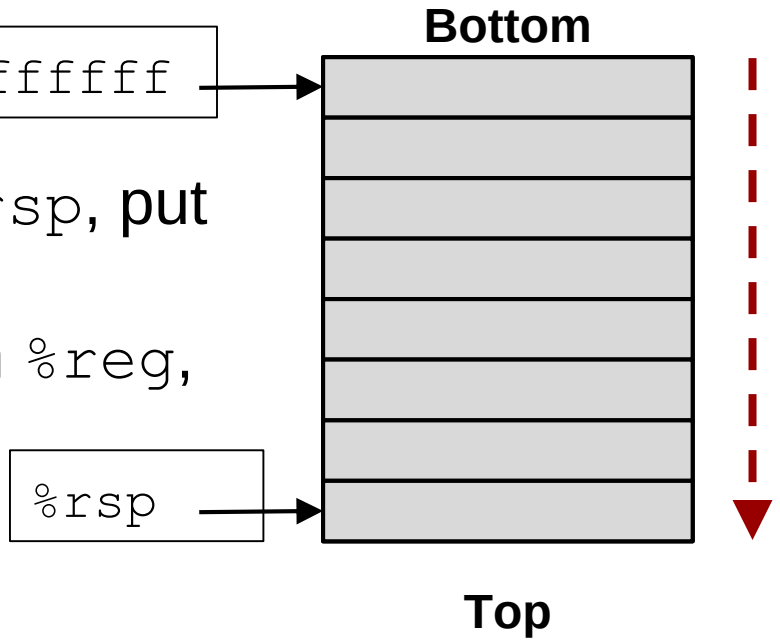


x86-64: Register Conventions

- 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`

x86-64: The Stack

- Grows **downward** towards **lower** memory addresses
- `%rsp` points to **top** of stack



- `push %reg`: subtract 8 from `%rsp`, put val in `%reg` at `(%rsp)`
- `pop %reg`: put val at `(%rsp)` in `%reg`, add 8 to `%rsp`

Buffer Overflow Stack Example #1

After call to gets

Stack Frame for `call_echo`

00	00	00	00
00	40	06	f6
00	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf ← %rsp

```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
   subq   $24, %rsp
   movq   %rsp, %rdi
   call   gets
   . . .
```

call_echo:

```
   . . .
   4006f1:  callq  4006cf <echo>
   4006f6:  add    $0x8,%rsp
   . . .
```

```
unix>./bufdemo-nsp
Type a string:012345678901234567890 12
012345678901234567890 12
```

"012345678901234567890 12\0"

Overflowed buffer, but did not corrupt state

Buffer Overflow Stack Example #2

After call to gets

Stack Frame for `call_echo`

00	00	00	00
00	40	00	34
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf ← %rsp

```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
   subq   $24, %rsp
   movq   %rsp, %rdi
   call   gets
   . . .
```

call_echo:

```
   . . .
   4006f1:  callq  4006cf <echo>
   4006f6:  add    $0x8,%rsp
   . . .
```

```
unix>./bufdemo-nsp
Type a string:012345678901234567890123
Segmentation fault
```

Buffer Overflow Stack Example #3

After call to gets

Stack Frame for `call_echo`

00	00	00	00
00	40	06	00
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf ← %rsp

```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
   subq   $24, %rsp
   movq   %rsp, %rdi
   call   gets
   . . .
```

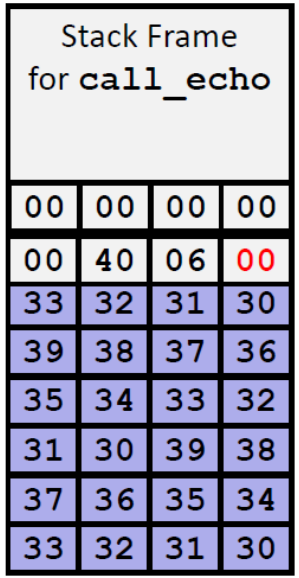
call_echo:

```
   . . .
   4006f1:  callq  4006cf <echo>
   4006f6:  add    $0x8,%rsp
   . . .
```

```
unix>./bufdemo-nsp
Type a string:012345678901234567890123
012345678901234567890123
```

Buffer Overflow Stack Example #3 Explained

After call to gets



buf ⟵ %rsp

register_tm_clones:

```
. . .
400600:  mov    %rsp,%rbp
400603:  mov    %rax,%rdx
400606:  shr    $0x3f,%rdx
40060a:  add    %rdx,%rax
40060d:  sar    %rax
400610:  jne    400614
400612:  pop    %rbp
400613:  retq
```

"Returns" to unrelated code
Lots of things happen, without modifying critical state
Eventually executes `retq` back to `main`

Attack Lab Overview: Phases 1-3

Overview

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

Key Advice

- Brush up on your x86-64 conventions!
- **Use objdump –d** to determine relevant offsets
- **Use GDB** to determine stack addresses

objdump/GDB – example1 / reci 2

- **Purpose**: To predict outputs of sample programs by analyzing assembly codes.

```
[cs20111100@uni06 gdb_ex]$ ls
ex1  ex2  ex3
[cs20111100@uni06 gdb_ex]$ objdump -d ex1

ex1:     file format elf64-x86-64


Disassembly of section .init:

0000000000400418 <_init>:
  400418:       48 83 ec 08             sub    $0x8,%rsp
  40041c:       48 8b 05 d5 0b 20 00    mov    0x200bd5(%rip),%rax
```

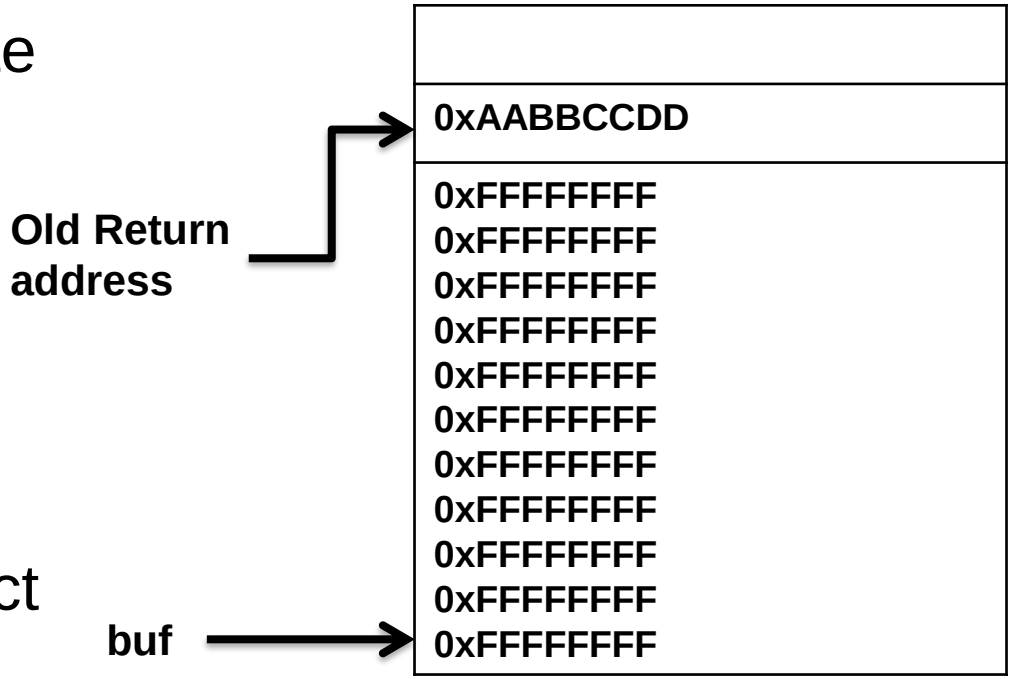
- There are 3 unknown binary files.
- "$ objdump –d ex1" command generates an assembly code for ex1.
- You can store it as a file by redirection.

  ex) $ objdump –d ex1 > ex1_assembly

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?



Attack Lab Overview: Phases 4-5

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

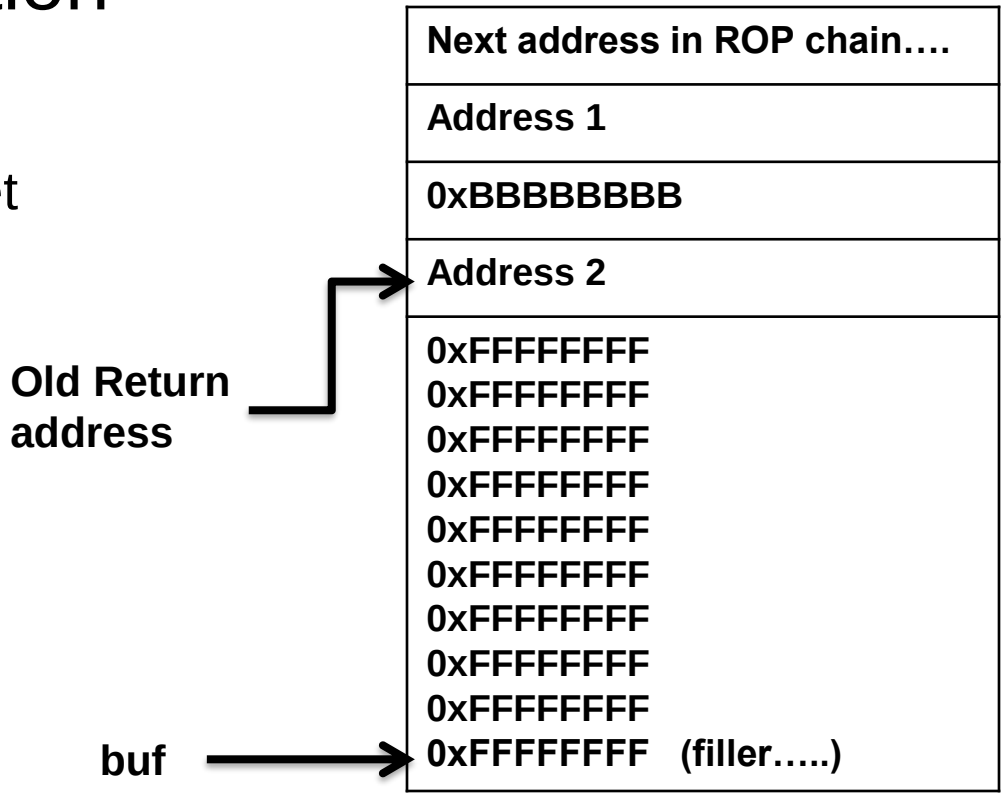
ROP Example: Solution

Gadgets:

Address 1: mov %rbx, %rax; ret

Address 2: pop %rbx; ret

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



Gadget example

- In rtarget

```
void setval_210(unsigned *p)
{
    *p = 3347663060U;
}
```

```
0000000000400f15 <setval_210>:
  400f15:       c7 07 d4 48 89 c7       movl   $0xc78948d4,(%rdi)
  400f1b:       c3                      retq
```

400f18

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

Tools

- **objdump –d**
  - View byte code and assembly instructions, determine stack offsets
- **./hex2raw**
  - Pass raw ASCII strings to targets
- **gdb**
  - Step through execution, determine stack addresses
- **gcc –c**
  - Generate object file from assembly language file

More Tips

- Draw stack diagrams
- Memo what data is assigned register
- Be careful of byte ordering (little endian)

Questions?