# Attack Lab

Sungyong Ahn

# Agenda

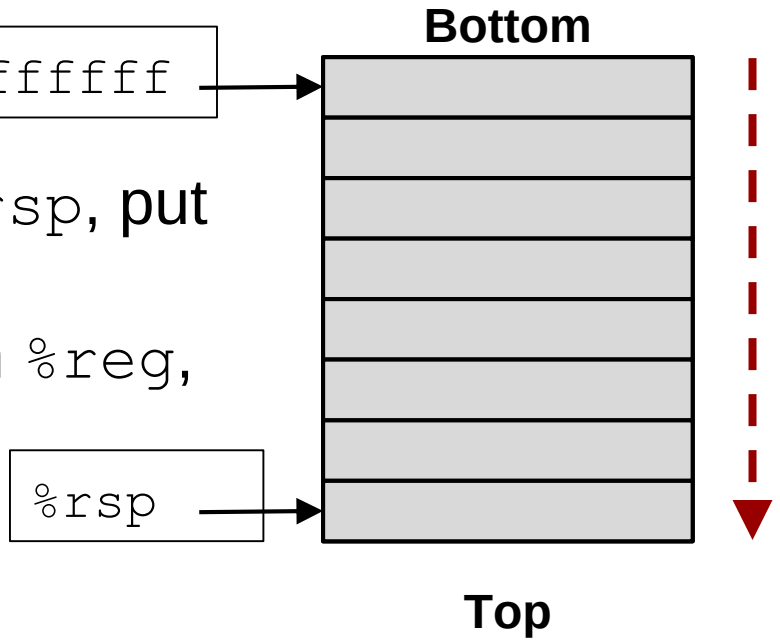
- Stack review
- Attack lab overview

# 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

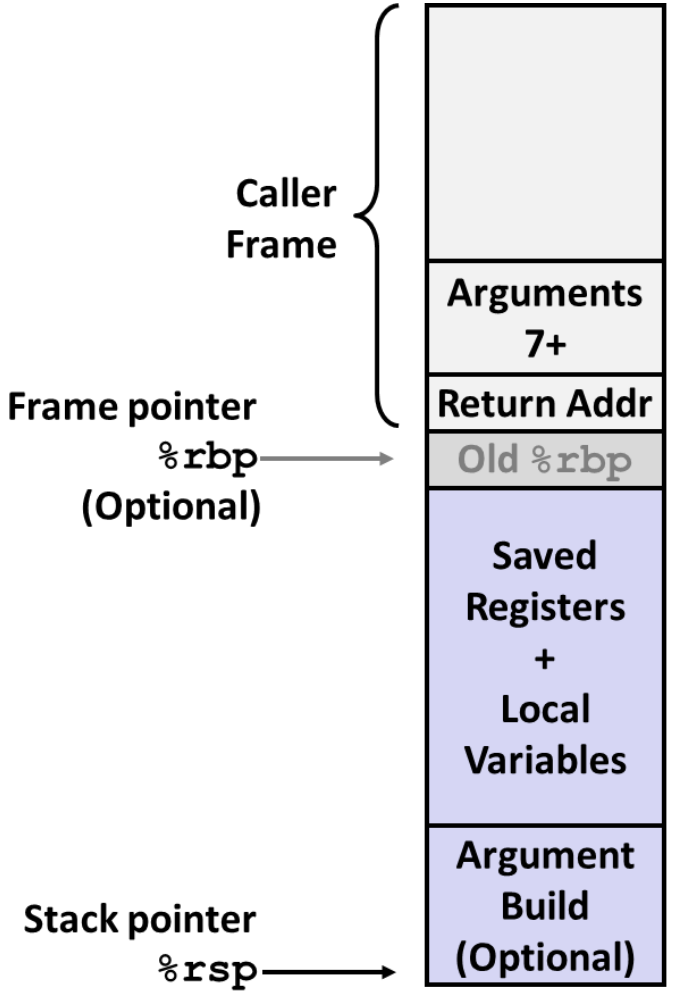
- Grows **downward** towards **lower** memory addresses
- `%rsp` points to **top** of stack




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

# 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



# 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

# x86-64: Function Call Return

Callee:

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

# Attack Lab Overview

- What is Attack Lab?
  - Five buffer overflow attacks on two programs having different security vulnerabilities
  - You will learn different ways that attackers can exploit security vulnerabilities
    - For a better understanding of how to write programs that are more secure
    - To understand the stack and parameter-passing mechanisms
    - For a deeper understanding of how x86-64 instructions are encoded
    - More experience with GDB and OBJDUMP

# Attack Lab Overview: Phases 1-3

Overview

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

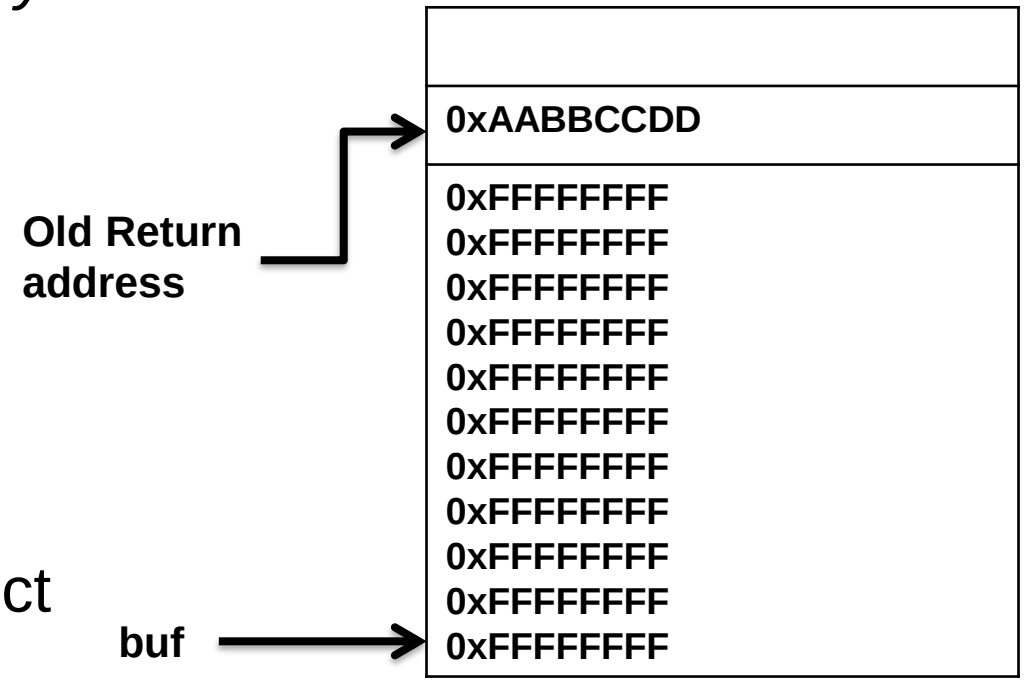
Key Advice

- Brush up on your x86-64 conventions!
- **Use objdump -d** to get this dissembled version
- Be careful about byte ordering

# Buffer Overflows

- Exploit *gets 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

- 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

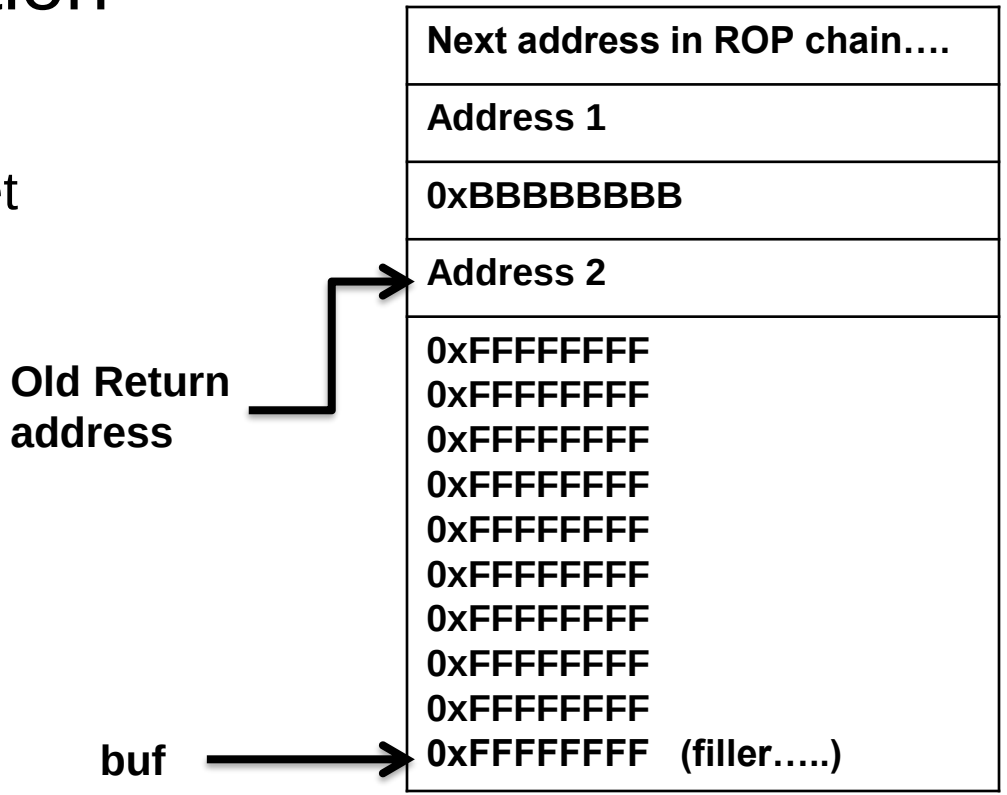
# ROP Example: Solution

**Gadgets:**

Address 1: mov %rbx, %rax; ret

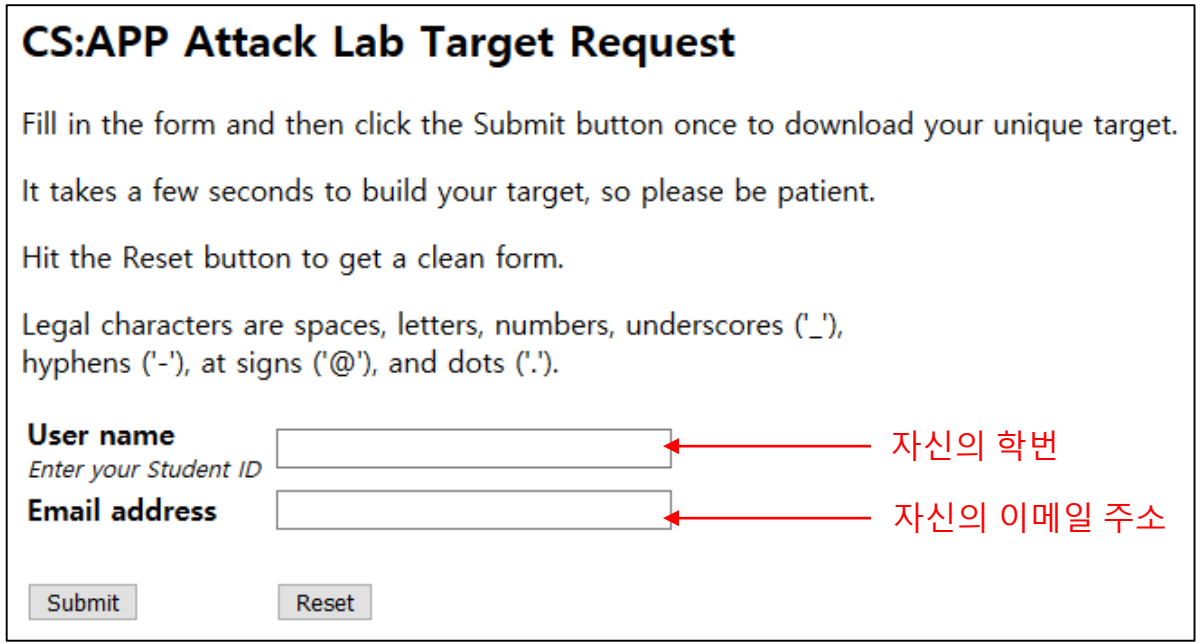
Address 2: pop %rbx; ret

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



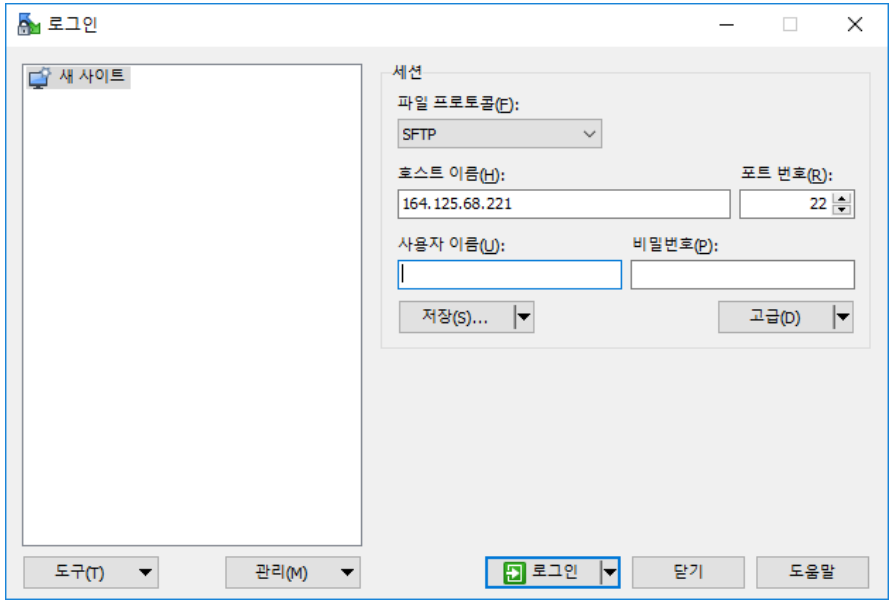
# Get your target file

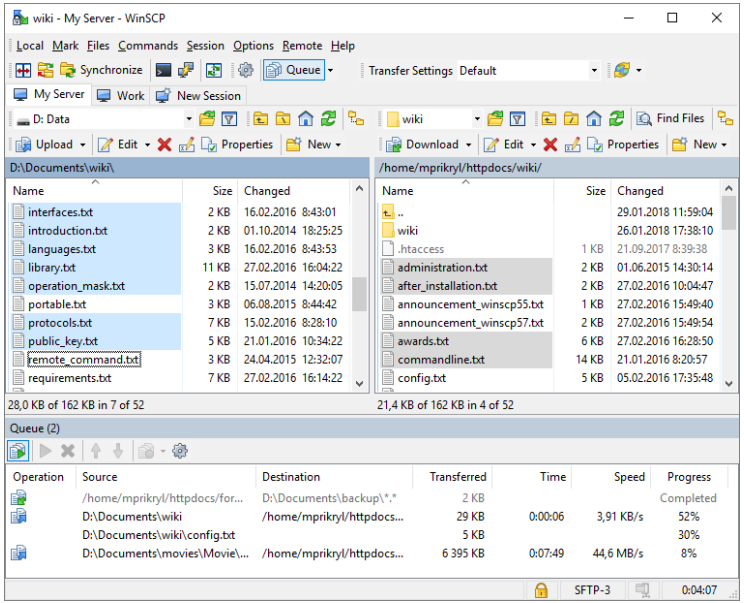
- You can obtain your target from http://164.125.68.221:15213/
  - `target`*k*`.tar` (*k* is the unique number)



# Get your target file

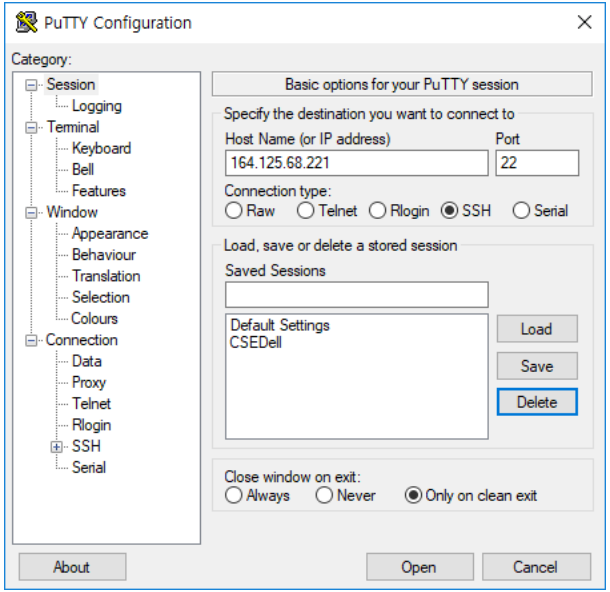
- Move your `targetk.tar` file to **`CSEDell`** (164.125.68.221)
  - Use WinSCP
    - https://winscp.net/eng/download.php



# Get your target file

- Extract your `targetk.tar` file on **`CSEDell`** (164.125.68.221)
  - Use putty for SSH connection
    - https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html
  - `tar -xvf targetk.tar`



# Get your target file

- The files in `target`*k* include
  - `README.txt`: A file describing the contents of the directory
  - `ctarget`: An executable program vulnerable to code-injection attacks
  - `rtarget`: An executable program vulnerable to return-oriented-programming attacks
  - `cookie.txt`: An 8-digit hex code that you will use as a unique identifier in your attacks.
  - `farm.c`: The source code of your target's "gadget farm," which you will use in generating return-oriented programming attacks.
  - `hex2raw`: A utility to generate attack strings.
- IMPORTANT NOTE: You can work on your solution only on the **`CSEDell`** machine.

# Target Programs

- Both CTARGET and RTARGET read strings from standard input with the function `getbuf`
  - It has buffer overflow vulnerability

```
1 unsigned getbuf()
2 {
3     char buf[BUFFER_SIZE];
4     Gets(buf);
5     return 1;
6 }
```

# Target Programs

- If the string typed by the user and read by `getbuf` is sufficiently short

```
unix> ./ctarget
Cookie: 0x1a7dd803
Type string: Keep it short!
No exploit.  Getbuf returned 0x1
Normal return
```

# Target Programs

- Typically an error occurs if you type a long string

```
unix> ./ctarget
Cookie: 0x1a7dd803
Type string: This is not a very interesting string,
Ouch!: You caused a segmentation fault!
Better luck next time
```

# Target Programs

- Command line arguments
  - `-h`: Print list of possible command line arguments
  - `-q`: Don't send results to the grading server
  - `-i FILE`: Supply input from a file, rather than from standard input

# Attack Lab Overview

- HEX2RAW
  - Your exploit strings will typically contain byte values that do not correspond to the ASCII values for printing characters.
  - The program HEX2RAW can help you generate these raw strings.

```
# echo "30 31 32 33 34 35" | ./hex2raw
012345
```

```
# cat exploit.txt | ./hex2raw | ./ctarget
```

  - HEX2RAW also supports C-style block comments

```
bf 66 7b 32 78 /* mov $0x78327b66,%edi */
```

# Attack Lab Overview

- Important points
  - Your exploit string must not contain byte value 0x0A ('`\n`') at any intermediate position
  - `HEX2RAW` expects two-digit hex values separated by a whitespace

```
../hex2raw < ctarget.l2.txt | ../ctarget
Cookie: 0x1a7dd803
Type string:Touch2!: You called touch2(0x1a7dd803)
Valid solution for level 2 with target ctarget
PASSED: Sent exploit string to server to be validated.
NICE JOB!
```

# Attack Lab Overview

- Auto grading server
  - When you have correctly solved one of the levels, your target program will automatically send a notification to the grading server.
    - Unlike the Bomb Lab, there is no penalty for making mistakes in this lab.
  - You can view the scoreboard by pointing your Web browser at http://164.125.68.221:15213/scoreboard

# Phase1: Code Injection Attacks: CTARGET

- Level 1

```
1 void test()
2 {
3     int val;
4     val = getbuf();
5     printf("No exploit.  Getbuf returned 0x%x\n",
6 }
```

```
1 void touch1()
2 {
3     vlevel = 1;        /* Part of validation protocol */
4     printf("Touch1!: You called touch1()\n");
5     validate(1);
6     exit(0);
7 }
```

Your task is to get CTARGET to execute the code for touch1 when getbuf executes its return statement, rather than returning to test.

# Phase2: Code Injection Attacks: CTARGET

- Level 2

```
1  void touch2(unsigned val)
2  {
3      vlevel = 2;        /* Part of validation protocol */
4      if (val == cookie) {
5          printf("Touch2!: You called touch2(0x%.8x)\n", val);
6          validate(2);
7      } else {
8          printf("Misfire: You called touch2(0x%.8x)\n", val);
9          fail(2);
10     }
11     exit(0);
12 }
```

Your task is to get CTARGET to execute the code for touch2 rather than returning to test. In this case, however, you must make it appear to touch2 as if you have passed your cookie as its argument.

# Phase3: Code Injection Attacks: CTARGET

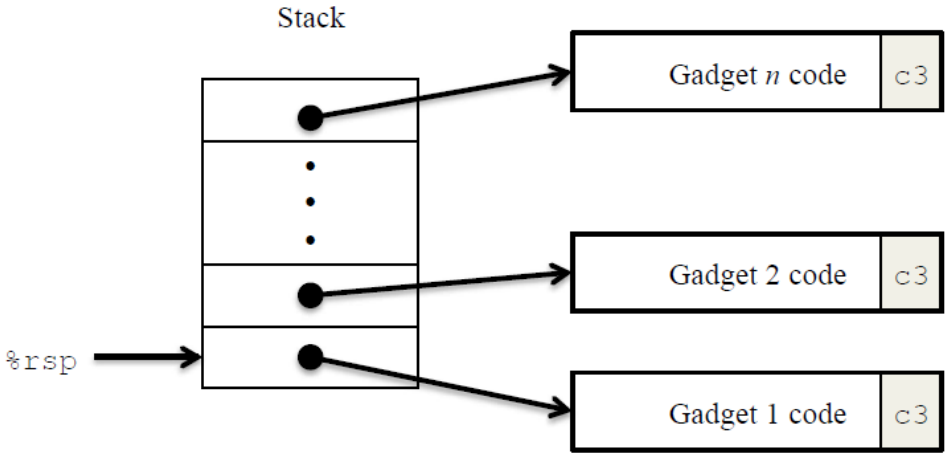
- Level 3

```
1  /* Compare string to hex represention of unsigned value */
2  int hexmatch(unsigned val, char *sval)
3  {
4      char cbuf[110];
5      /* Make position of check string unpredictable */
6      char *s = cbuf + random() % 100;
7      sprintf(s, "%.8x", val);
8      return strncmp(sval, s, 9) == 0;
9  }
10
11 void touch3(char *sval)
12 {
13     vlevel = 3;        /* Part of validation protocol */
14     if (hexmatch(cookie, sval)) {
15         printf("Touch3!: You called touch3(\"%s\")\n", sval);
16         validate(3);
17     } else {
18         printf("Misfire: You called touch3(\"%s\")\n", sval);
19         fail(3);
20     }
21     exit(0);
22 }
```

Your task is to get CTARGET to execute the code for touch3 rather than returning to test. You must make it appear to touch3 as if you have passed a string representation of your cookie as its argument.

# Return-Oriented Programming(ROP)

- It uses randomization so that the stack positions differ from one run to another. This makes it impossible to determine where your injected code will be located.
- It marks the section of memory holding the stack as nonexecutable, so even if you could set the program counter to the start of your injected code, the program would fail with a segmentation fault.



# Return-Oriented Programming(ROP)

- For example, `rtarget` contains the following C function.

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

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

48 89 c7 → movq %rax, %rdi ← starts at address 0x400f18

`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 |

# Return-Oriented Programming(ROP)

- Your code for `RTARGET` contains a number of functions similar to the `setval_210` function
  - We refer to as the *gadget farm*

- Your job will be to identify useful gadgets in the *gadget farm* and use these to perform attacks similar to those you did in Phases 2 and 3

# Phase4: ROP Level2 : `RTARGET`

- You will repeat the attack of Phase 2, but do so on program `RTARGET` using gadgets from your *gadget farm*.
- You can construct your solution using gadgets consisting of the following instruction types, and using only the first eight x86-64 registers (`%rax`–`%rdi`).

# Phase4: ROP Level2 : RTARGET

- You can find byte encodings of more instructions on the ***writeup***
  - *movq, popq, movl*
  - *ret* : This instruction is encoded by the single byte 0xc3.
  - *nop* : This instruction (pronounced “no op,” which is short for “no operation”) is encoded by the single byte 0x90. Its only effect is to cause the program counter to be incremented by 1.

# Phase5 : ROP Level 3: `RTARGET`

- Phase 5 requires you to do an ROP attack on `RTARGET` to invoke function `touch3` with a pointer to a string representation of your cookie.

# Generating Byte Codes

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

example.s

```
# Example of hand-generated assembly code
push $0xabcdef      # Push value onto stack
add $17,%eax        # Add 17 to %eax
movl %eax,%edx      # Copy lower 32 bits to %edx
```

```
ubuntu# gcc -c example.S
ubuntu# objdump -d example.o > example.d
```

example.d

```
example.o: file format elf64-x86-64
Disassembly of section .text:
0000000000000000 <.text>:
0:  68 ef cd ab 00      pushq   $0xabcdef
5:  48 83 c0 11         add     $0x11,%rax
9:  89 c2               mov     %eax,%edx
```

This string can then be passed through `HEX2RAW` to generate an input string for the target programs

```
68 ef cd ab 00 48 83 c0 11 89 c2
```

**OR**

```
68 ef cd ab 00 /* push $0xabcdef */
83 c0 11          /* add $0x11,%eax */
89 c2             /* mov %eax,%edx */
```

# Summary

- Download your target and attack it!
  - http://164.125.68.221:15213/
- 제출기한 : 12월 6일 11:59 PM
  - 하루 딜레이당 20점 감점
- 반드시 실습 서버(164.125.68.221)에서 수행할 것!!
  (그 외의 환경에서는 실행 불가)
- 제출물
  - **TargetID_학번.docx**: 각 phase에서 자신이 attack을 수행한 과정을 간략히 설명
    - MS워드 파일로 작성
    - 표지없이 간단히 첫 장 상단에 이름과 학번만 명시
    - 5장을 넘지 말 것, 초과시 보고서 점수 감점 사유.
    - PLMS로 제출

# Questions?