XJTU-ICS
BombLab && AttackLab

Yunguang Li, 2024-3-23 Some Contents from CMU-15-213

Outline

- ✓ Bomblab
- ✓ Attacklab
- ✓ Some Tools
- ✓ Command
- ✓ Some Cases

Some advice

- Start early !!!
- Think more before Asking !
- Keep a log while working !
- Compare and think after finishing !

What is Bomb Lab?

An exercise in reading x86-64 assembly code.

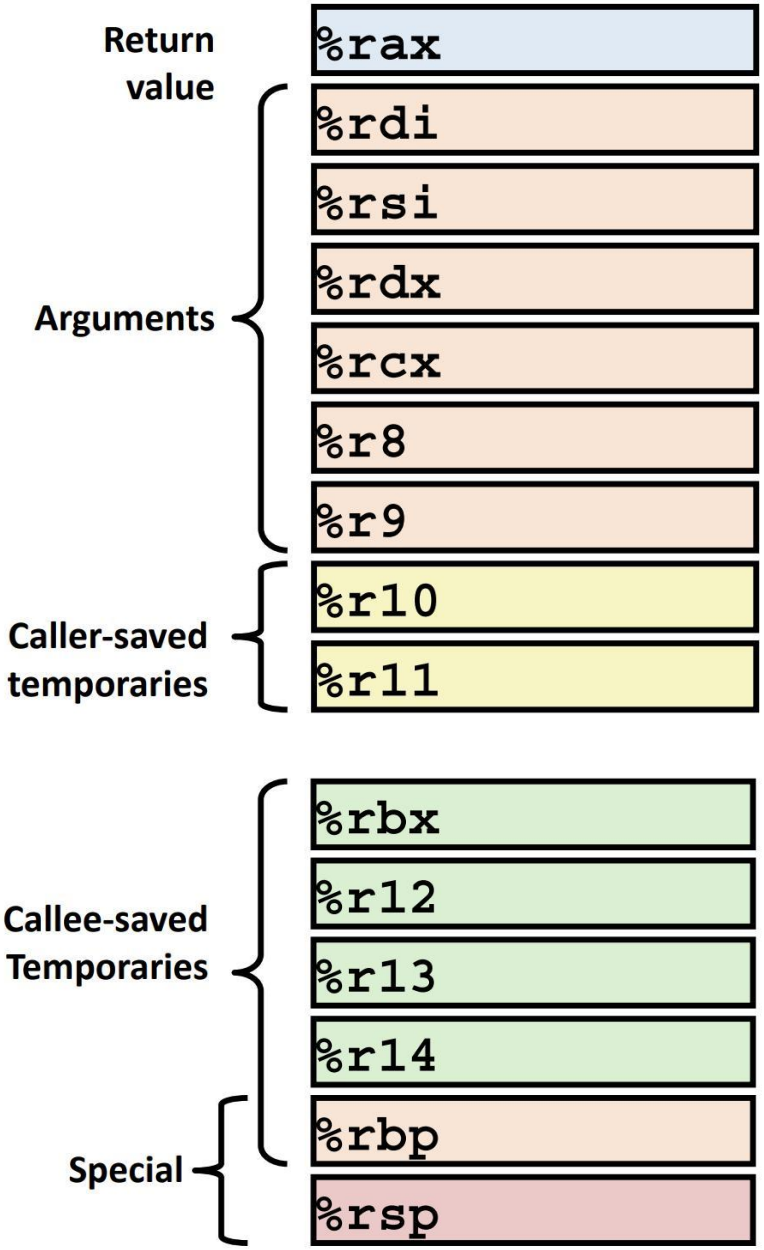
A chance to practice using GDB (a debugger).

Why ?

- x86 assembly is low level machine code. Useful for understanding security exploits or tuning performance.
- GDB can save you days of work in future labs (Malloc) and can be helpful long after you finish this class.

Linux x86-64 ABI

- %rax
  - return value
- %rdi, ..., %r9
  - Arguments
- %r10, %r11
  - Caller-saved
- %rbx, %r12, %r13, %r14
  - Callee-saved
- %rsp %rbp: Special



What’s Attack Lab ?

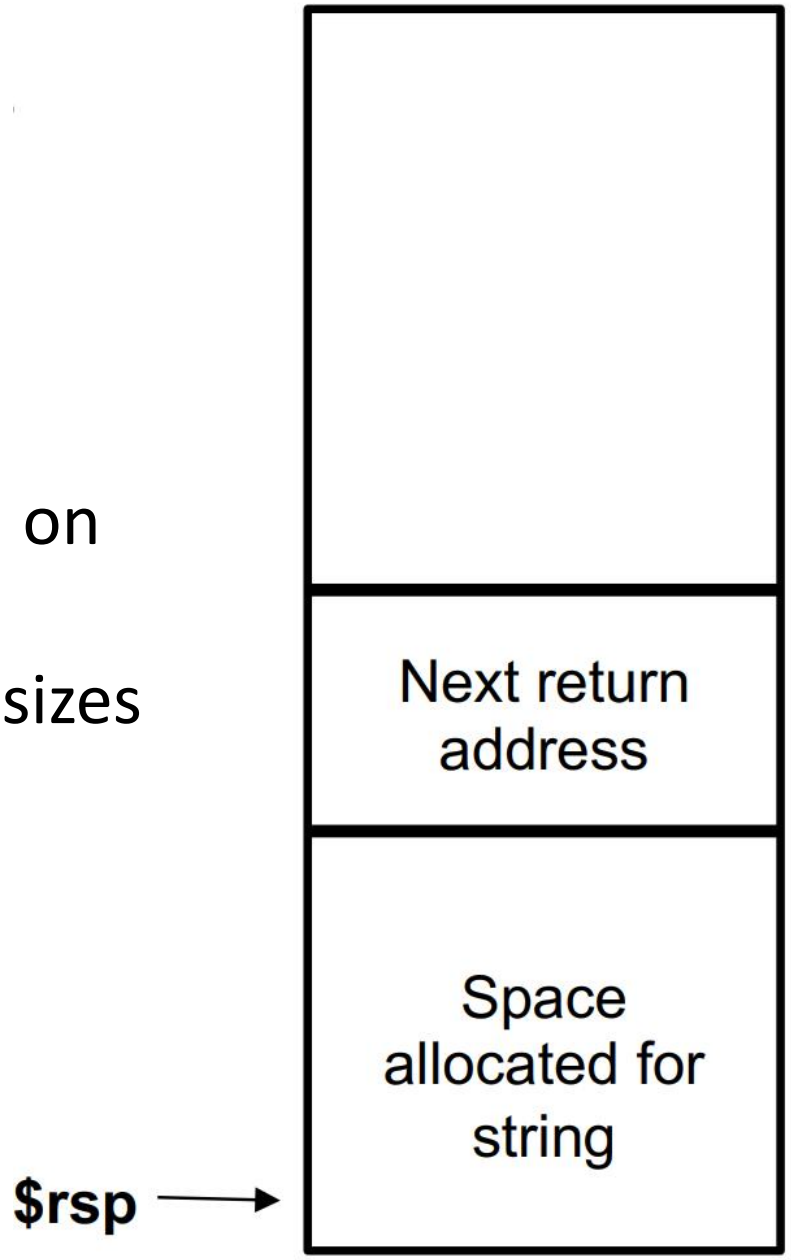
We’re letting you hijack programs by running buffer overflow attacks on them…

To understand stack discipline and stack frames

To defeat relatively secure programs with return oriented programming

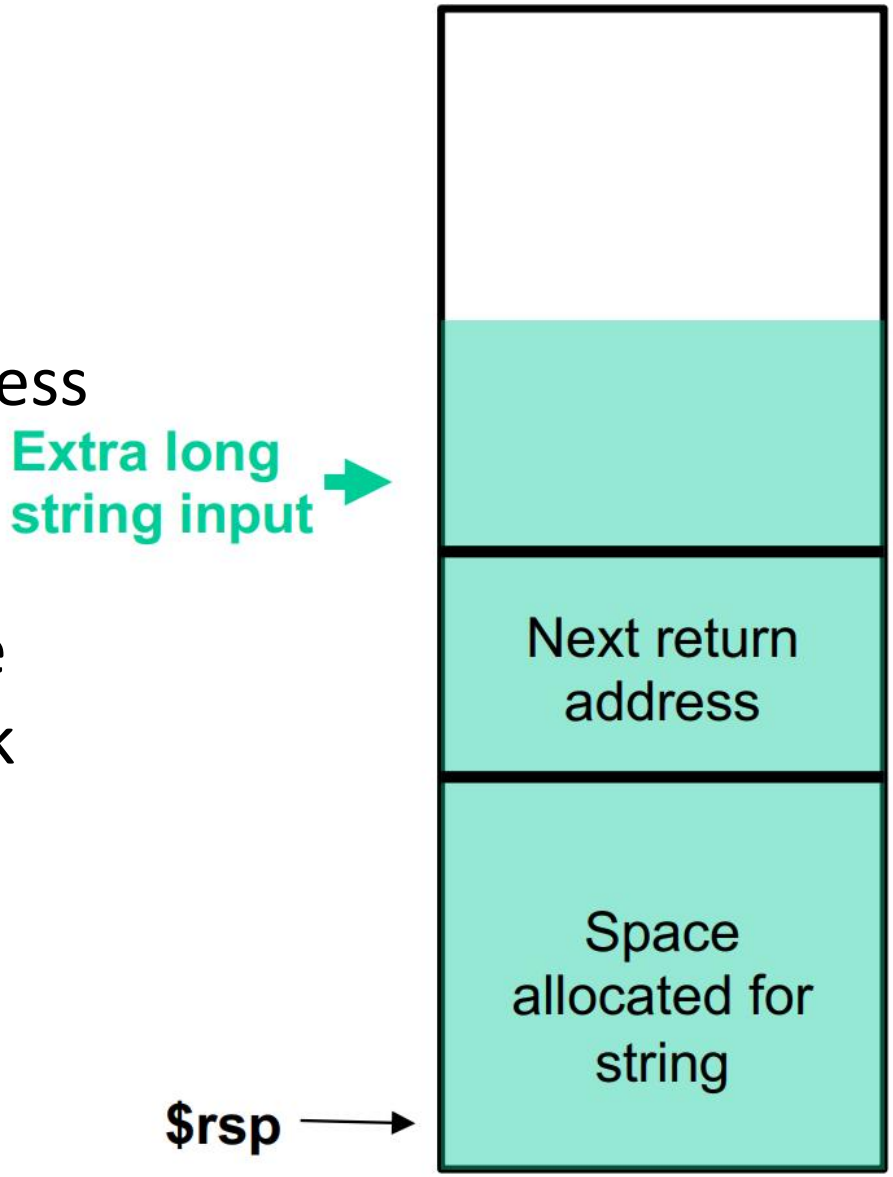
Buffer Overflows

- Local string variables are stored on the stack
- Some C functions do not check sizes of strings



Buffer Overflows

- You can write a string that overwrites the return address
- Activity 1 steps through an example of overwriting the return address on the stack



Tools: Objdump

- Disassemble to generate assembly file
- $ objdump -d [name of executable] > [any file name]
  - Saves the assembly code of the executable into the file.
    - The objdump assembly file address is not real virtual address!!!

```
1
2 bomb:     file format elf64-x86-64
3
4
5 Disassembly of section .init:
6
7 0000000000001000 <_init>:
8     1000: f3 0f 1e fa           endbr64
9     1004: 48 83 ec 08           sub    $0x8,%rsp
10    1008: 48 8b 05 d9 3f 00 00  mov    0x3fd9(%rip),%rax        # 4fe8 <__gmon_start__@Base>
11    100f: 48 85 c0              test   %rax,%rax
12    1012: 74 02                 je     1016 <_init+0x16>
13    1014: ff d0                 call   *%rax
14    1016: 48 83 c4 08           add    $0x8,%rsp
15    101a: c3                    ret
```

Tools: man

- $ man sscanf
  - you are allowed to look up documentation of functions
  - man pages are your friend :)
- sscanf: string scan format
  - parses a string provided as an argument to the function

```
char *example_string = "123, 456";
int a, b;
sscanf(example_string, "%d, %d", &a, &b)
```

After this code snippet is run, a = 123 and b = 456

Tools: GDB

GDB is a powerful debugger-- let's you inspect your program as it's executing.

Fundamental Instruction:

- You can open gdb by typing into the shell:
  - $ gdb
- Type gdb and then a binary to specify which program to run
  - $ gdb <binary>
- This is the notation we'll be using for the rest of the slides:
  - $ cd // The command should be typed in the bash shell
  - (gdb) break // The command should be typed in GDB

Helpful GDB Commands

Disassemble: displays assembly

- (gdb) disas(disassemble) + (func) // show the assembly code of specific func

```
 3 int pass = 1;
 4
 5 void add(int* x)
 6 {
 7   ++*x;
 8 }
 9
10 int main() {
11   // Initialize
12   int input = 0;
13
14   scanf("%d", &input);
15   add(&pass);
16   if (pass != input)
17   {
18     printf("Something Wrong\n");
19     return 0;
20   }
21   printf("Everything good!\n");
22   return 0;
23 }
```

```
(gdb) disas add
Dump of assembler code for function add:
   0x00005555555551a9 <+0>:     endbr64
   0x00005555555551ad <+4>:     push   %rbp
   0x00005555555551ae <+5>:     mov    %rsp,%rbp
   0x00005555555551b1 <+8>:     mov    %rdi,-0x8(%rbp)
   0x00005555555551b5 <+12>:    mov    -0x8(%rbp),%rax
   0x00005555555551b9 <+16>:    mov    (%rax),%eax
   0x00005555555551bb <+18>:    lea    0x1(%rax),%edx
   0x00005555555551be <+21>:    mov    -0x8(%rbp),%rax
   0x00005555555551c2 <+25>:    mov    %edx,(%rax)
   0x00005555555551c4 <+27>:    nop
   0x00005555555551c5 <+28>:    pop    %rbp
   0x00005555555551c6 <+29>:    ret
End of assembler dump.
```

Helpful GDB Commands

Breakpoints: stops execution of program when it reaches certain point

- break function_name: breaks once you call a specific function
- break *0x…: breaks when you execute instruction at a certain address
- info b: displays information about all breakpoints currently set
- disable #: disables breakpoint with id equal to #

Helpful GDB Commands

Navigating through assembly:

- stepi: moves one instruction forward, will step into functions encountered
- nexti: moves one instruction forward, skips over functions called
- c: continues execution until next breakpoint is hit

```
Breakpoint 1, 0x000055555555520e in main ()
(gdb) ni
0x0000555555555213 in main ()
(gdb)
```

```
Breakpoint 1, 0x000055555555520e in main ()
(gdb) si
0x00005555555551a9 in add ()
(gdb)
```

What to do

Don’t understand what a big block of assembly does? GDB
Need to figure out what’s in a specific memory address? GDB
Can’t trace how 4 – 6 registers are changing over time? GDB
Have no idea how to start the assignment? BombLab/Attacklab Tutorial
Need to know how to use certain GDB commands? BombLab Tutorial
    Also useful: http://ics.dfshan.net/GDB-Usage-Tutorial
Don’t know what an assembly instruction does? Lecture slides
Confused about control flow or stack discipline? Lecture slides