Attack Lab

Agenda

- Stacks
- Attack Lab Activities

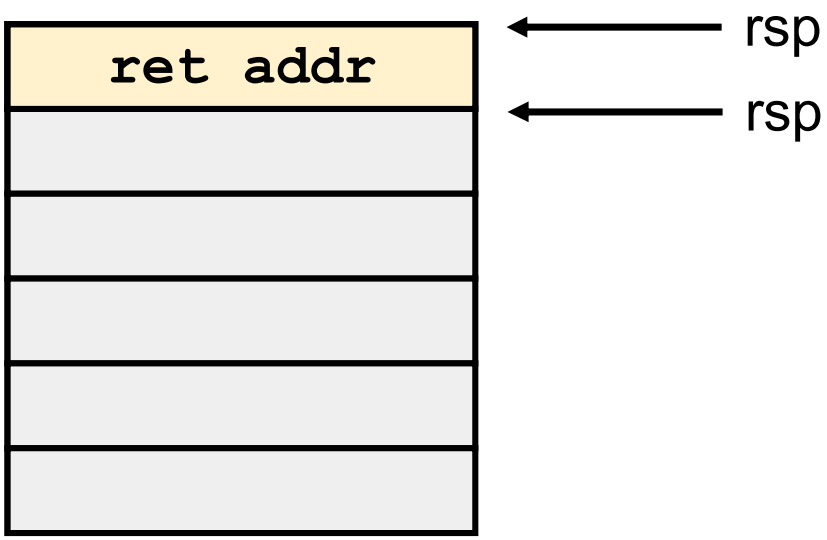
Attack Lab

- We're letting you hijack programs by running buffer overflow attacks on them.
  - Is that not justification enough?
- Helps you understand stack discipline and stack frames
- Also let you defeat relatively secure programs with return oriented programming

Stack Overview

Let's say you have the following stack diagram. What happens when you call a function?

What information always goes on the stack?



Attack Lab Activities

- **Three activities**
  - Each relies on a specially crafted assembly sequence to purposefully overwrite the stack

- **Activity 1 – Overwrites the return addresses**
- **Activity 2 – Writes an assembly sequence onto the stack**
- **Activity 3 – Uses byte sequences in libc as the instructions**

act1.c

```
#include <stdio.h>
#include <stdlib.h>
#include <alloca.h>

void clobber(char*, int);

 void printHi()
 {
     printf("Hi!\n");
  }

  char* buf;
```

```
int main(int argc, char** argv)
  {
     char* x = alloca(8);
     buf = malloc(16);
     *(long*) buf = (long)&printHi;
     *(long*) (buf + 8) = 0x0000000000400642;
     clobber(buf, 16);
     clobber(x, 8);
     return 0;
  }
```

Attack Lab Activities

- One student needs a laptop
- Login to bupt1 machine

```
$ tar xvf act123.tar
$ cd act123
$ make
$ gdb act1
```

Activity 1

(gdb) break clobber

(gdb) run

(gdb) x $rsp

(gdb) backtrace

Q. Does the value at the top of the stack match any frame?

Activity 1 Continued

(gdb) x /2gx $rdi // Here are the two key values

(gdb) stepi // Keep doing this until

```
(gdb)
clobber () at support.s:16
16              ret
```

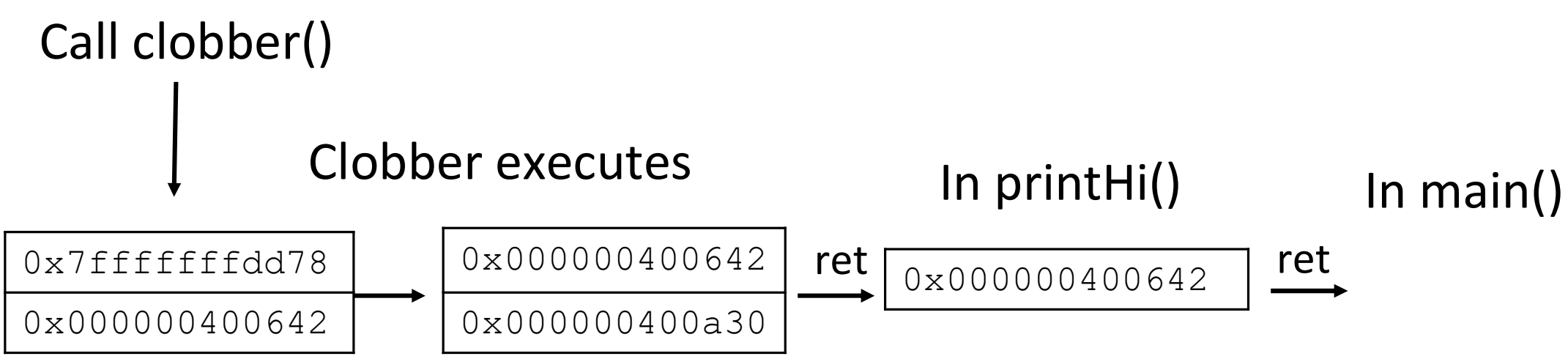
(gdb) x/gx $rsp

Q. Has the return address changed?

(gdb) finish // Should exit and print out “Hi!”

Activity 1 Post

- **Clobber overwrites part of the stack with memory at $rdi, including the all-important return address**
- **In act1 it writes two new return addresses:**
  - 0x400a30: address of printHi()
  - 0x400642: address in main



act2.c

```
void clobber(char*, int);
const char hiStr[] = "Hi\n";

int main(int argc, char** argv)
{
   char* x = alloca(32);
   unsigned char* m = malloc(128);

   puts("Activity 2!");
   if (m == NULL)
   {
      fprintf(stderr, "Allocation failure\n");
      return -1;
   }
```

```
   if (mprotect((void*)(((uint64_t)x) & (~0xfff))…
   {
      perror("MPROTECT");
      free(m);
      return -1;
   }
   *(uint64_t*) m = (uint64_t)(x);
   m[8] = 0xbf;
   *(uint32_t*) (m + 9) = (unsigned int)(uint64_t) hiStr;
   *(uint32_t*) (m + 13) = 0x410100be;
   *(uint32_t*) (m + 18) = 0xd6ff;
   *(uint32_t*) (m + 20) = 0x40ec40be;
   *(uint32_t*) (m + 25) = 0xd6ff;
   clobber(m, 32);
   return 0;
}
```

Activity 2

$gdb act2

(gdb) break clobber

(gdb) run

(gdb) x $rsp

Q. What is the address of the stack and the return address?

(gdb) x /4gx $rdi

Q. What will the new return address be?

(i.e., what is the first value?)

Activity 2 Continued

(gdb) x/5i $rdi + 8 // Display as instructions

Q. Why rdi + 8?

Q. What are the three addresses?

(gdb) break puts

(gdb) break exit

Q. Do these addresses look familiar?

Activity 2 Post

- **Normally programs cannot execute instructions on the stack**
  - Main used mprotect to disable the memory protection for this activity

- **Clobber wrote an address that's on the stack as a return address**
  - Followed by a sequence of instructions
  - Three addresses show up in the exploit:
    - 0x4a186d → "Hi\n" string
    - 0x410100 → puts() function
    - 0x40ec40→ exit() function

act3.c

```
void clobber(char*, int);
const char hiStr[] = "Hi\n";

void printAndExit(char* s)
{
    puts(s);
    exit(0);
 }

int main(int argc, char** argv)
{
    char* x = alloca(48);
    unsigned char* m = malloc(128);

    puts("Activity 3!");
    if (m == NULL)
    {
        fprintf(stderr, "Allocation failure\n");
        return -1;
    }

    *(uint64_t*) m = (uint64_t)(0x40374b);
    *(uint64_t*) (m + 8) = (uint64_t)(hiStr);
    *(uint64_t*) (m + 16) = (uint64_t)(0x400643);
    *(uint64_t*) (m + 24) = (uint64_t)(&printAndExit);
    *(uint64_t*) (m + 32) = (uint64_t)(0x402dcd);
    clobber(m, 40);

    return 0;
}
```

Activity 3

$gdb act3

(gdb) break clobber

(gdb) run

(gdb) x /5gx $rdi

Q. Which value will be first on the stack?

Q. At the end of clobber, where will the function return to?

Activity 3 Continued

(gdb) x /2i <return address>

Q. What does this sequence do?

Q. Do the same for the other addresses. Note that some are return addresses and some are for data. When you continue, what will the code now do?

Activity 3 Post

- It's harder to stop programs from running existing pieces of code in the executable.

- Clobber wrote multiple return addresses (aka gadgets) that each performed a small task, along with data that will get popped off the stack while running the gadgets.

  - 0x40374b: pop %rdi; retq
  - 0x4a13a4: Pointer to the string "Hi\n"
  - 0x400643: pop %rbx; retq
  - 0x400a30: Address of a printing function
  - 0x402dcd: callq *%rbx

Activity 3 Post

- Note that some of the return addresses actually cut off bytes from existing instructions

```
465b54:    5b       pop    %rbx
465b55:    5d       pop    %rbp
465b56:    41 5c    pop    %r12
465b58:    41 5d    pop    %r13
465b5a:    41 5e    pop    %r14
465b5c:    41 5f    pop    %r15
465b5e:    c3       retq
465b5f:    90       nop
```

0x465b5c …0c …0d

----

pop %r15 retq

41 5f

c3

pop %rdi

retq

5f

c3

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

Attack Lab Tools

- **gcc –c test.s; objdump –d test.o > test.asm**

  Compiles the assembly code in test.s and shows the actual bytes for the instructions

- **./hex2raw < exploit.txt > converted.txt**

  Convert hex codes in exploit.txt into raw ASCII strings to pass to targets

  See the writeup for more details on how to use this

- **(gdb) display /12gx $rsp** **(gdb) display /2i $rip**

  Displays 12 elements on the stack and the next 2 instructions to run

GDB is also useful to for tracing to see if an exploit is working

If you get stuck

- **Please read the writeup.** ***Please read the writeup.*** ***Please read the writeup.*** ***Please read the writeup!***
- CS:APP Chapter 3
- man gdb, gdb's help command