Attack Lab

Agenda

- Reminders
- Stacks
- Attack Lab Activities

Stacks

- Last-in, first-out

- x86 stack grows down
  - lowest address is “top”
  - $rsp contains the address of the topmost element in the stack

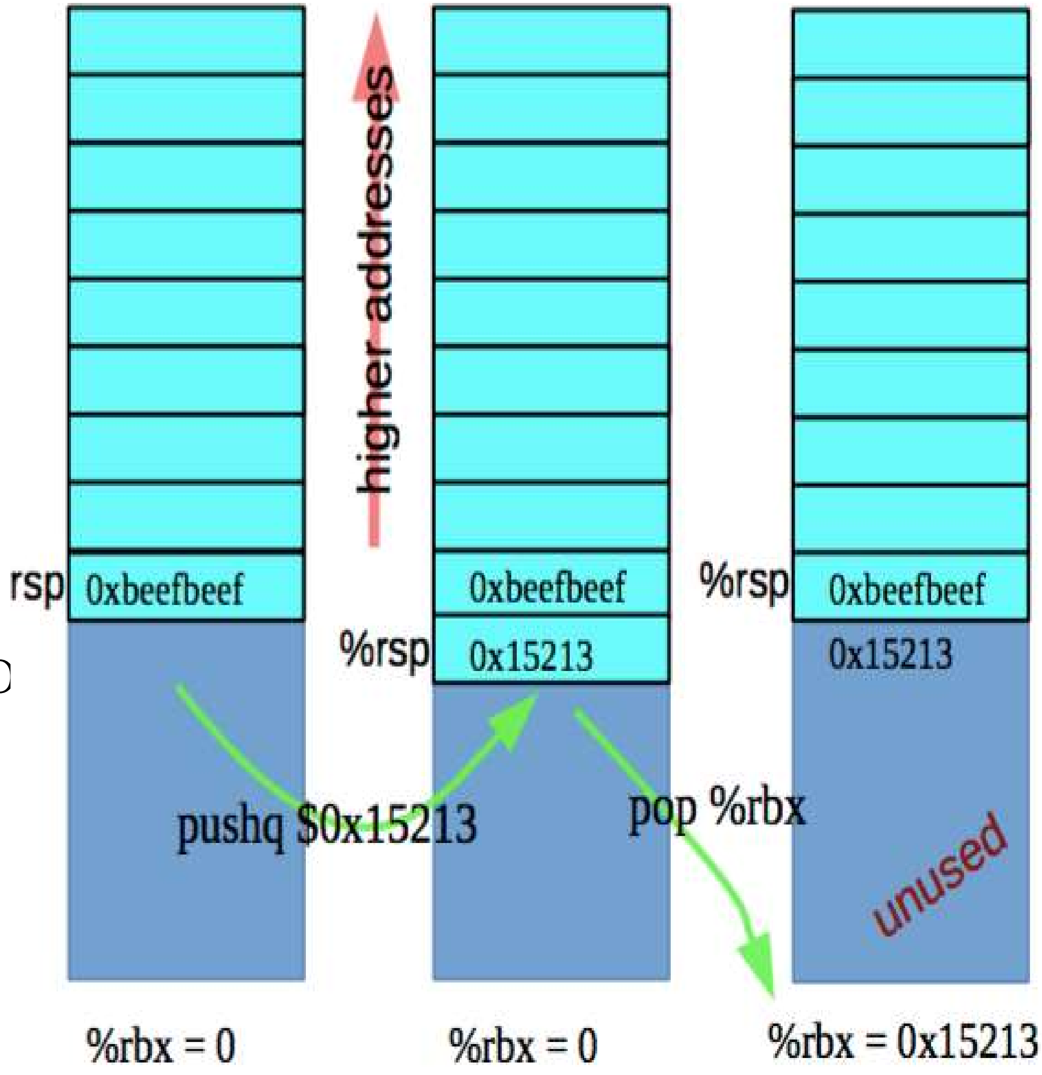
- Uses the pushq and popq instructions to push and pop registers/constants onto and off the stack

Stack – pushq & popq

- pushq {value} is equivalent to

  sub $8, %rsp
  mov {value}, (%rsp)

- popq {reg} is equivalent to
  mov (%rsp), {reg}
  add $8, %rsp



Stack - Caller vs. Callee

- Function A calls function B
  - A is the caller
  - B is the callee

- Stack space is allocated in “frames”
  - Represents the state of a single function invocation

- Frame used primarily for two things:
  - Storing callee saved registers
  - Storing the return address of a funciton

Registers – Caller-saved vs. Callee-saved

- Caller-saved
  - Registers used for function arguments are always caller-saved
  - $rax is also caller-saved
  - Called function may do as it wishes with the registers
  - Must save/restore register in caller's stack frame if it still needs the value after a function call
- **Callee-saved**
  - If the function wants to change the register, it must save the original value in its stack frame and restore it before returning
  - The calling function may store temporary values in callee-saved registers

x86-64 Register Usage Conventions

%rax	return value	%r8	argument #5
%rbx	callee saves	%r9	argument #6
%rcx	argument #4	%r10	caller saves
%rdx	argument #3	%r11	caller saves
%rsi	argument #2	%r12	callee saves
%rdi	argument #1	%r13	callee saves
%rsp	stack pointer	%r14	callee saves
%rbp	callee saves	%r15	callee saves

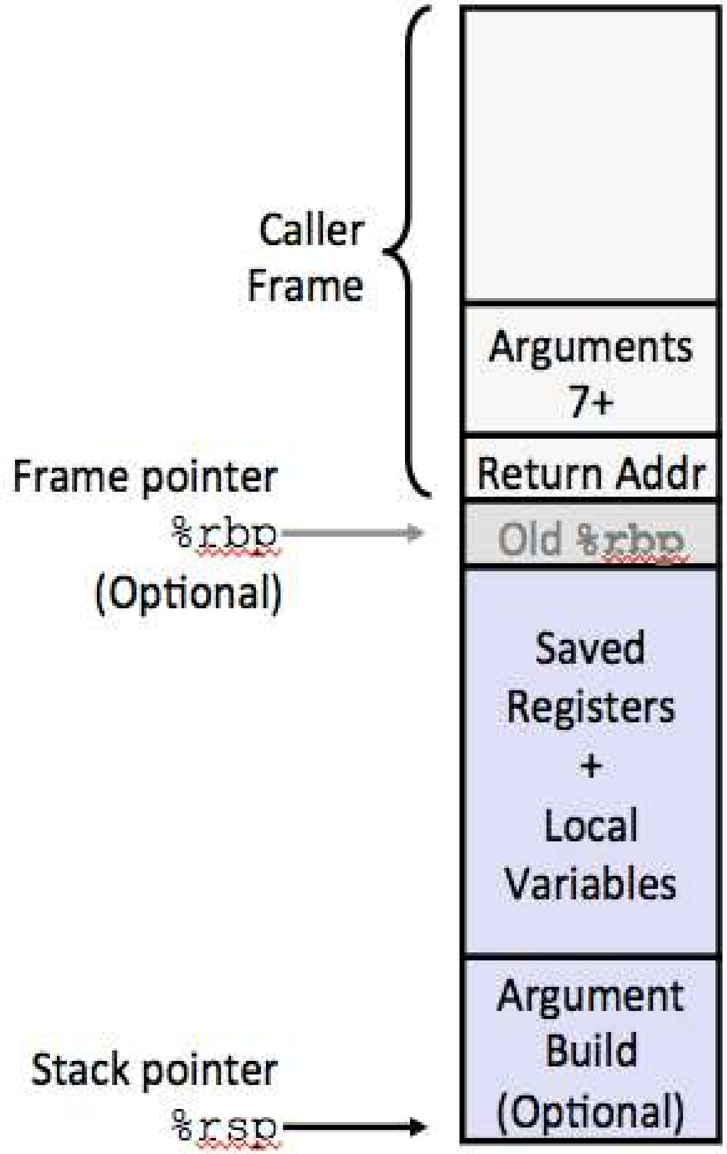
Registers – Caller-saved vs. Callee-saved

- Before function call
  - rdi = first argument
  - rsi = second argument
  - rax = some temporary value
  - rbx = some important number to use later (15213)
  - rsp = pointer to some important buffer (0x7fffffffaaaa)

- After function call
  - rdi = garbage
  - rsi = garbage
  - rax = return value
  - rbx = some important number to use later (15213)
  - rsp = pointer to some important buffer (0x7fffffffaaaa)

x86-64/Linux Stack Frame

- Current Stack Frame ("Top" to Bottom)
  - "Argument build:"
    - Parameters for function about to call
  - Local variables
    - If can't keep in registers
  - Saved register context
  - Old frame pointer (optional)
- Caller Stack Frame
  - Return address
    - Pushed by call instruction
  - Arguments for this call



Stack Maintenance

- Functions free their frame before returning
- Return instruction looks for the return address at the top of the stack
  - *…What if the return address has been changed?*

Attack Lab

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

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

Attack Lab Activities

- One student needs a laptop
- Login to bupt1 machine

```
$ tar xf lab3-rec5.tar
$ cd rec5
$ make
$ gdb act1
```

act1.c

```
(gdb) break clobber
(gdb) run
(gdb) x $rsp
(gdb) backtrace
```

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

act1.c

```
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) = 0x00000000004003d0;
      clobber(buf, 16);
      clobber(x, 8);
      return 0;
  }
```

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
```

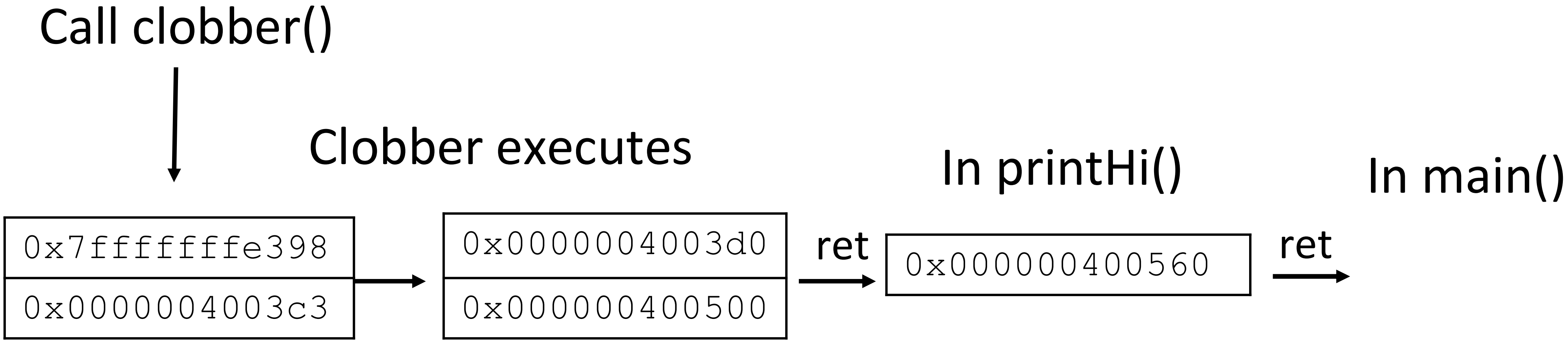
(gdb) x $rsp

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:
  - 0x400500: address of printHi()
  - 0x400560: 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) = 0x402290be;
    *(uint32_t*) (m + 18) = 0xd6ff;
    *(uint32_t*) (m + 20) = 0x401150be;
    *(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?)

Activitity 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:
    - 0x48644d → “Hi\n” string
    - 0x402290 → puts() function
    - 0x401150 → 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)(0x457c07);
    *(uint64_t*) (m + 8) = (uint64_t)(hiStr);
    *(uint64_t*) (m + 16) = (uint64_t)(0x429aaa);
    *(uint64_t*) (m + 24) = (uint64_t)(&printAndExit);
    *(uint64_t*) (m + 32) = (uint64_t)(0x42dfbb);
    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.
  - 0x457c07: pop %rdi; retq
  - 0x47fbb0: Pointer to the string "Hi\n"
  - 0x429aaa: pop %rax; retq
  - 0x400520: Address of a printing function
  - 0x42dfbb: callq *%rax

Activity 3 Post

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

```
457cfa: 48 83 c4 28          add    $0x28,%rsp
457cfe: 5b                   pop    %rbx
457cff: 4a 8d 44 3d 00       lea    0x0(%rbp,%r15,1),%rax
457d04: 5d                   pop    %rbp
457d05: 41 5c                pop    %r12
457d07: 41 5d                pop    %r13
457d09: 41 5e                pop    %r14
457d0b: 41 5f                pop    %r15
457d0d: c3                   retq
457d0e: 48 83 7c 24 10 00    cmpq   $0x0,0x10(%rsp)
457d14: 74 8a                je     457ca0 <_IO_getline_info+0xd0>
```

0x457d0b	...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

If you get stuck

- ***Please read the writeup.*** ***<u>Please read the writeup.</u>*** ***Please read the writeup!***
- CS:APP Chapter 3
- View lecture notes
- man gdb, gdb's help command

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