15-213 Recitation 5
Attack Lab and Stacks

Your TAs
Monday, February 10th, 2020 (15-213, 18-213)
Wednesday, February 12th, 2020 (18-613)
Agenda

- Attack Lab Overview
- Stacks Review
- Activity 1
- Procedure Calling Review
- Activity 2
Learning objectives

By the end of this recitation, we want you to know:

- Stack discipline and calling conventions
- How to perform a simple buffer overflow attack

More discussion in lecture on Tuesday:
*Machine-Level Programming V: Advanced Topics*
Reminders and Lab Overview
Reminders

- Bomb Lab is due **Tuesday, Feb. 11** at 11pm ET!
- Attack Lab will be released at the same time, and is due the following **Tuesday, Feb. 18**
Attack Lab overview

- Attack programs by crafting buffer overflow attacks that hijack the control flow
- Provide inputs to the rtarget and ctarget programs that cause them to call certain functions
- Unlike in bomblab, the targets don't explode!
Stacks Review
Manipulating the stack

What instructions do we typically use to change the stack pointer, %rsp?

Growing the stack:  Shrinking the stack:
Manipulating the stack`

What instructions do we typically use to change the stack pointer, %rsp?

Growing the stack:  
- sub $0x28, %rsp
- push %rbx
- callq my_function

Shrinking the stack:
Manipulating the stack

What instructions do we typically use to change the stack pointer, %rsp?

Growing the stack:
- sub $0x28, %rsp
- push %rbx
- callq my_function

Shrinking the stack:
- add $0x28, %rsp
- pop %rbx
- retq
x86-64 Stack Frames

What kinds of data are stored on the stack?
x86-64 Stack Frames

What kinds of data are stored on the stack?

- Saved registers
- Local variables
- Arguments (7+)
- Saved return address
Which way does the stack grow?

- Up?
- Down?
- Left?
- Right?
Which way does the stack grow?

- Up?
- Down?
- Left?
- Right?

It depends on how you draw it!

The stack always grows towards **lower addresses** in x86-64.

(Informally, this usually means "down".)

Be aware of this possible ambiguity when reading diagrams.
Drawing memory

Stack diagrams

Addresses are displayed increasing to the **left**, and then **upwards**.

Everything else

Addresses are displayed increasing to the **right**, and then **downwards**.
Endianness

- Describes how integers are represented as bytes.
- Little-endian means that the least-significant 8 bits of an integer are stored at the lowest address.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01020304</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

"Big end"  "Little end"
Endianness

- Describes how integers are represented as bytes.
- Little-endian means that the least-significant 8 bits of an integer are stored at the lowest address.
Activity 1
Part 1: Introduction to solve()

Let's look at solve() in the src/activity.c file.

What is it doing?

Is it possible for the program to call win()?

```c
void solve(void) {
    long before = 0xb4;
    char buf[16];
    long after = 0xaf;

    Gets(buf);

    if (before == 0x3331323531)
        win(0x15213);

    if (after == 0x3331323831)
        win(0x18213);
}
```
Part 1: The gets() function

```c
char *gets(char *s);
```

- gets() reads from standard input and writes characters into `s` until it reaches a newline.
- Since it has no information about the size of the buffer `s`, its design is fundamentally flawed. **Never use gets() yourself!**
- Gets() is a CS:APP wrapper function that checks for errors, and exits if it encounters any.
Part 1: Activity setup

- Split up into groups of 2-3 people
- One person needs a laptop
- Log in to a Shark machine, and type:

  $ wget https://www.cs.cmu.edu/~213/activities/rec5.tar
  $ tar xvf rec5.tar
  $ cd rec5

- Take a look at the code in src/activity.c.
Part 1: Diving into assembly

- Look at the disassembly of `solve()`.
- Try drawing a stack diagram.
  - How large is the stack frame?
  - Where is the saved return address?
  - Where are `before`, `buf`, and `after`?
- Which variable will be overwritten if we perform a buffer overflow, `before` or `after`?
Part 1: Drawing the stack diagram

addresses increase towards the top of the slide

=> 0x4006b5 <+0>:  sub $0x38,%rsp

rsp

return address

rsp+0x38
Part 1: Drawing the stack diagram

0x4006b5 <+0>:     sub     $0x38,%rsp
=> 0x4006b9 <+4>:    movq    $0xb4,0x28(%rsp)

 Addresses increase towards the top of the slide

return address

 rsp+0x38

 rsp
Part 1: Drawing the stack diagram

Addresses increase towards the top of the slide.
Part 1: Drawing the stack diagram

0x4006b5 <+0>:     sub    $0x38,%rsp
0x4006b9 <+4>:     movq   $0xb4,0x28(%rsp)
0x4006c2 <+13>:    movq   $0xaf,0x8(%rsp)
0x4006cb <+22>:    lea    0x10(%rsp),%rdi
=> 0x4006d0 <+27>:    callq  0x40073f <Gets>
Part 1: Drawing the stack diagram

Addresses increase towards the top of the slide.

```
0x4006b5 <+0>: sub $0x38,%rsp
0x4006b9 <+4>: movq $0xb4,0x28(%rsp)
0x4006c2 <+13>: movq $0xaf,0x8(%rsp)
0x4006cb <+22>: lea 0x10(%rsp),%rdi
0x4006d0 <+27>: callq 0x40073f <Gets>
=> 0x4006d5 <+32>: mov 0x28(%rsp),%rdx
```
Part 1: Comparing with GDB output

Let's compare the stack diagram we drew with the actual values on the stack after `Gets()` returns.

0x4006d0 <+27>: callq 0x40073f <Gets>
=> 0x4006d5 <+32>: mov 0x28(%rsp),%rdx

(gdb) break *0x4006d5
(gdb) run
Starting program: act1
abcdefgh12345678
(gdb) x/8gx $rsp
(gdb) x/64bx $rsp
Part 1: Comparing with GDB output

(gdb) x/8gx $rsp
0x602020: 0x0000000000000000 0x00000000000000af
0x602030: 0x6867666564636261 0x3837363534333231
0x602040: 0x0000000000000000 0x00000000000000b4
0x602050: 0x0000000000000000 0x0000000000000000

(gdb) x/64bx $rsp
0x602020: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xaf 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x602028: 0x61 0x62 0x63 0x64 0x65 0x66 0x67 0x68
0x602030: 0x31 0x32 0x33 0x34 0x35 0x36 0x37 0x38
0x602038: 0xb4 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x602040: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x602048: 0x83 0x07 0x40 0x00 0x00 0x00 0x00 0x00
0x602050: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x602058: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00

Addresses increase towards bottom of the slide

Addresses increase towards top of the slide
Part 1: Exploitation

- Try to find an input string that wins 1 cookie!
  - What do we need to overwrite before with if we want to have before == 0x3331323531?

- Constructing an exploit
  - `gets()` stops reading once it sees a newline. In the buffer, it replaces the newline with a null terminator.
  - `gets()` does **not** stop reading at a null terminator.
Part 1: Recap

- Buffer overflows can **overwrite** parts of the stack frame, including other local variables.
- Stack frames may include **padding**, so looking at the assembly is crucial to drawing a correct diagram.
- GDB prints output starting at the **lowest** address, whereas our stack diagrams start at the **highest**.
Procedure Calling Review
Call and return instructions

Which registers do callq and retq change?

<table>
<thead>
<tr>
<th>%rax</th>
<th>%rbx</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>%r12</td>
</tr>
<tr>
<td>%rsi</td>
<td>%r13</td>
</tr>
<tr>
<td>%rdx</td>
<td>%r14</td>
</tr>
<tr>
<td>%rcx</td>
<td>%rbp</td>
</tr>
<tr>
<td>%r8</td>
<td>%rsp</td>
</tr>
<tr>
<td>%r9</td>
<td>%rip</td>
</tr>
<tr>
<td>%r10</td>
<td></td>
</tr>
<tr>
<td>%r11</td>
<td></td>
</tr>
</tbody>
</table>
Call and return instructions

Which registers do callq and retq change?

%rax
%rdi
%rsi
%rdx
%rcx
%r8
%r9
%r10
%r11
%rbx
%r12
%r13
%r14
%rbp
%rsp
%rip
Stack/Procedure Review

0000000000400540 <multstore>:
  
  
=> 400544: callq 400550 <mult2>
  400549: mov %rax, (%rbx)
  
0000000000400550 <mult2>:
  400550: mov %rdi, %rax
  
  400557: retq

%rsp 0x120
%rip 0x400544
Stack/Procedure Review

00000000000400540 <multstore>:
   
   
=> 400544: callq 400550 <mult2>
  400549: mov %rax, (%rbx)
   
   
What happens next?

00000000000400550 <mult2>:
  400550: mov %rdi, %rax
   
   
  400557: retq
Stack/Procedure Review

0000000000400540 <multstore>:
  •
  •
  400544: callq 400550 <mult2>
  400549: mov %rax,(%rbx)
  •
  •

0000000000400550 <mult2>:
  =>400550: mov %rdi,%rax
  •
  •
  400557: retq
Stack/Procedure Review

00000000000400540 <multstore>:
  •
  •
  400544: callq 400550 <mult2>
  400549: mov %rax,(%rbx)
  •
  •

00000000000400550 <mult2>:
  400550: mov %rdi,%rax
  •
  •
  =>400557: retq

0x400550
0x400557

Stack/Procedure Review

0x400549
0x118
0x128
0x120
0x118
0x130

%rip 0x400557
%rsp 0x118
Stack/Procedure Review

0x400549: mov %rax, (%rbx)

0x400550: mov %rdi, %rax

0x400557: retq

Stack pop to %rip
Let’s Rewind…

What if we mess up the return address?
Activity 2
Part 2: Exploitation

- Hijacking control flow
  - Is it possible to overwrite after? If not, what parts of the stack frame can we overwrite?
  - Is there anywhere we could jump to call \texttt{win(0x18213)}?

- Constructing an exploit

`inputs/input2.txt`

```
48 65 6c 6c 6f 20 31 35 32 31 33 21
```

# comment

`inputs/input2.bin`

```
Hello 15213!
```

make

( runs hex2raw )
Part 2: Recap

- `retq` always jumps to the **saved return address**, which it pops off the stack (at `rsp`).

- **Overwriting** the saved return address on the stack allows us to "fool" `retq`, and transfer control to an arbitrary instruction.
Attack Lab Tools

- $ gcc -c test.s
  $ objdump -d test.o
  Compiles the assembly code in test.s, then shows the disassembled instructions along with the actual bytes.

- $ ./hex2raw < exploit.txt > exploit.bin
  Convert hex codes into raw binary strings to pass to targets.

- (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 carefully.** Not everything will make sense on the first read-through.

- Other resources you can make use of:
  - CS:APP Chapter 3
  - Lecture slides and videos
  - x86-64 and GDB cheat sheets under [Resources](#)