Many of the following slides are based on those from

Complete Powerpoint Lecture Notes for
Computer Systems: A Programmer's Perspective (CS:APP)

Randal E. Bryant and David R. O'Hallaron

http://www.cs.cmu.edu/afs/cs/academic/class/15213-f15/www/schedule.html

The book is used explicitly in CS 2505 and CS 3214 and as a reference in CS 2506.

Agenda

Stack review

Attack lab overview

- Phases 1-3: Buffer overflow attacks
- Phases 4-5: ROP attacks

x86-64 Registers

Return	%rax	%eax
	%rbx	%ebx
Arg 4	%rcx	%ecx
Arg 3	%rdx	%edx
Arg 2	%rsi	%esi
Arg 1	%rdi	%edi
Stack ptr	%rsp	%esp
	%rbp	%ebp

%r8	%r8d	Arg 5
%r9	%r9d	Arg 6
%r10	%r10d	
%r11	%r11d	
%r12	%r12d	
%r13	%r13d	
%r14	%r14d	
%r15	%r15d	

x86-64: Register Conventions

Arguments are passed in registers (default):

`%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`

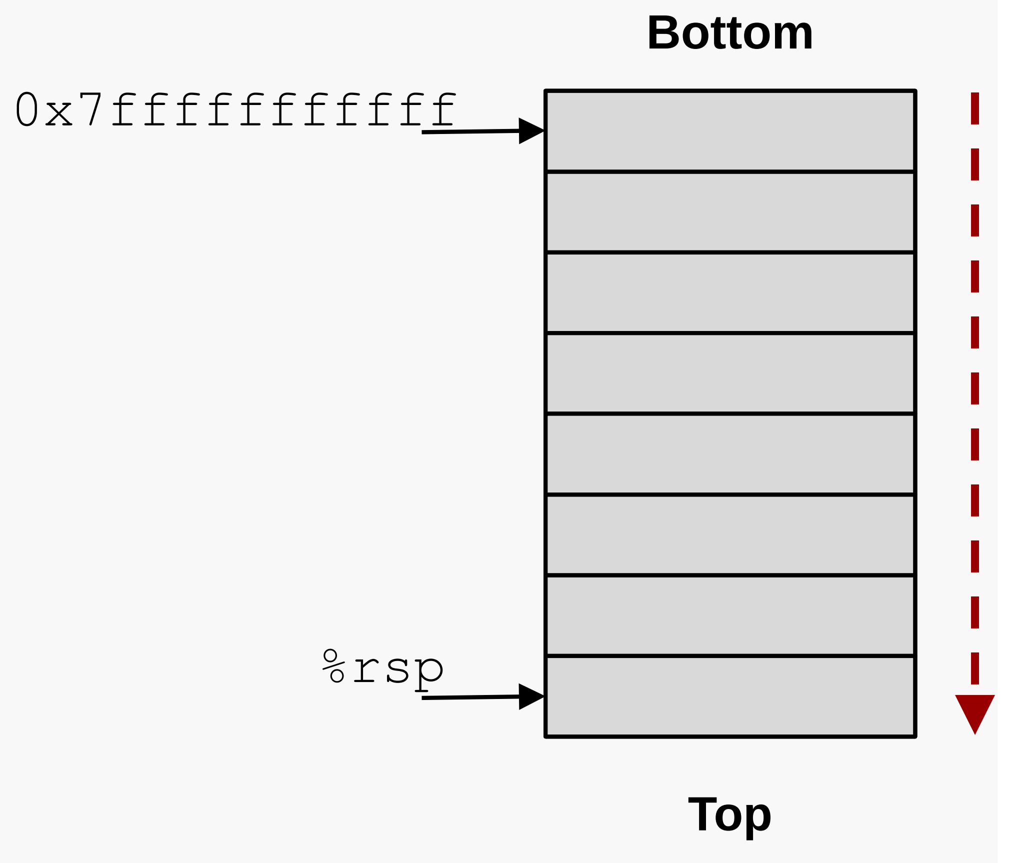
Stack pointer: `%rsp`

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`

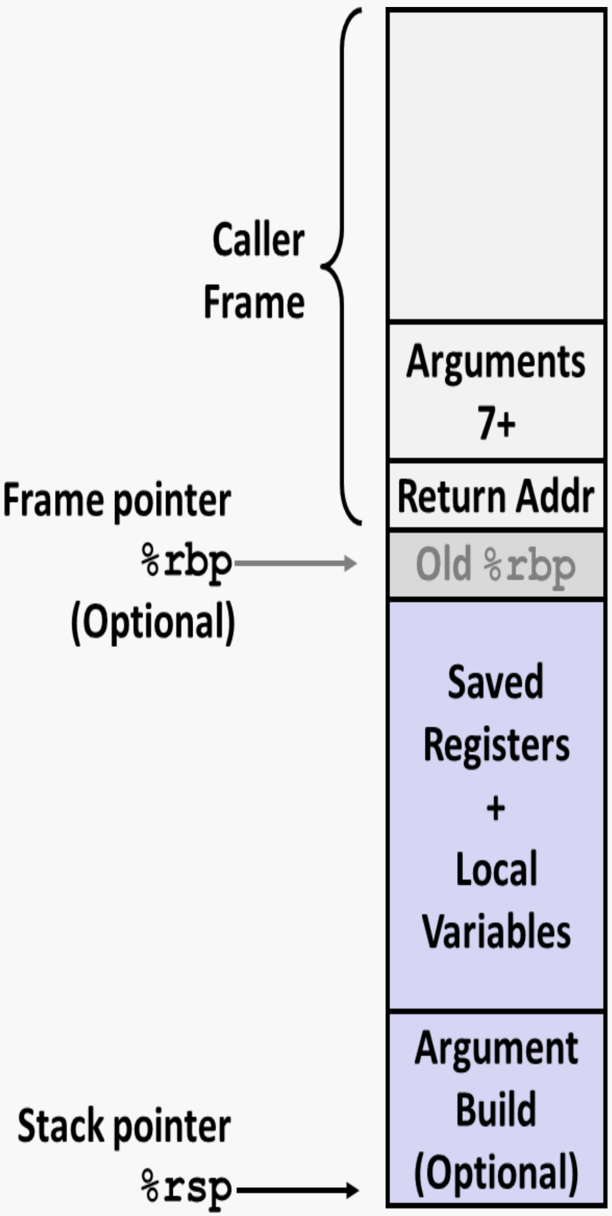
put val at `(%rsp)` in `%reg`,
add 8 to `%rsp`

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: 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 determine relevant offsets
- **Use GDB** to determine stack addresses

Exploit C String library vulneratilities 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?

old return address → 0xAABBCCDDEEFFGGHH

```
0xAABBCCDDEEFFGGHH
0xFFFFFFFFFFFFFFFF
0xFFFFFFFFFFFFFFFF
0xFFFFFFFFFFFFFFFF
0xFFFFFFFFFFFFFFFF
0xFFFFFFFFFFFFFFFF
0xFFFFFFFFFFFFFFFF
0xFFFFFFFFFFFFFFFF
0xFFFFFFFFFFFFFFFF
0xFFFFFFFFFFFFFFFF
0xFFFFFFFFFFFFFFFF
0xFFFFFFFFFFFFFFFF
```

buf → 0xFFFFFFFFFFFFFFFF (last row)

String Library Code

Implementation of Unix function **gets**

No way to specify limit on number of characters to read

```
// Get string from stdin
char *gets(char *dest)
{
    int c = getc();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getc();
    }
    *p = '\0';
    return dest;
}
```

So bad it's officially deprecated, but it has been used widely, is still used, and `gcc` still supports it.

Vulnerable Buffer Code

```
int main()
{
  printf("Type a string:");
  echo();
  return 0;
}
```

```
// Echo Line
void echo()
{
    char buf[4];  // Dangerously small!
                  // Stored on the stack!!

    gets(buf);    // Call to oblivious fn

    puts(buf);
}
```

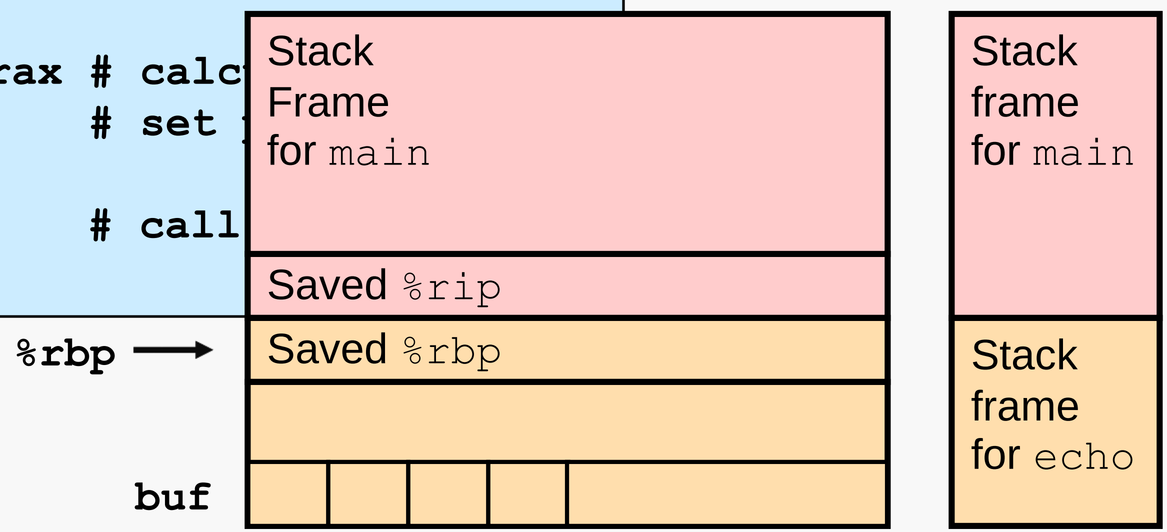
Buffer Overflow Executions

```
CentOS > ./bufdemo
Type a string:abcd
abcd


CentOS >  ./bufdemo
Type a string:abcdefghijklmnopqrst
abcdefghijklmnopqrst


CentOS >  ./bufdemo
Type a string:abcdefghijklmnopqrstuvwx
abcdefghijklmnopqrstuvwx
Segmentation fault (core dumped)
```

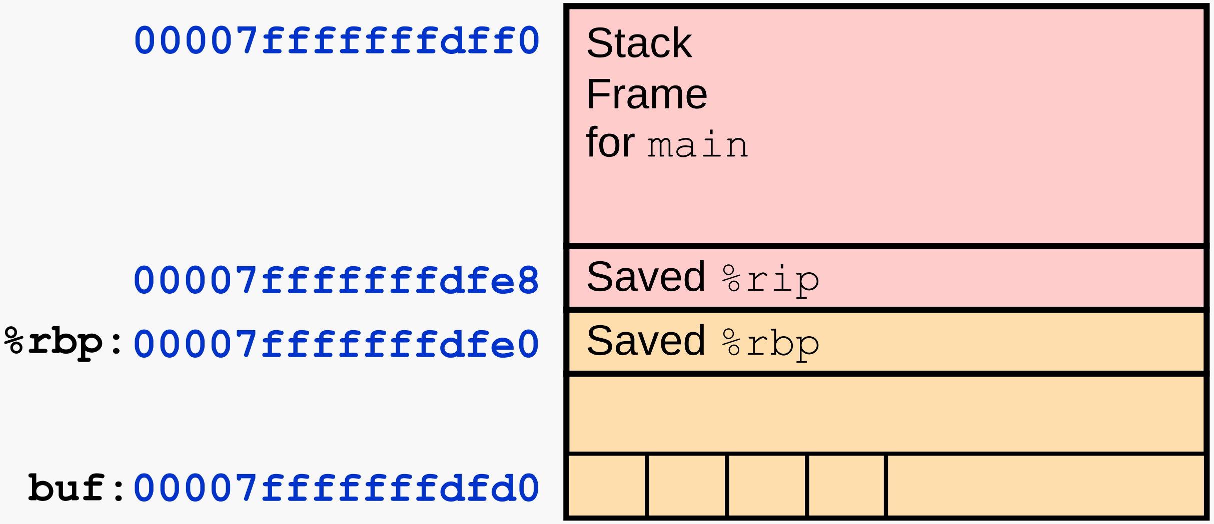
Buffer Overflow Stack

```
// Echo Line
void echo()
{
     char buf[4];   // Way too small!
     gets(buf);
     puts(buf);
}
```

```
echo:
    pushq   %rbp                # setup stack frame
    movq    %rsp, %rbp
    subq    $16, %rsp

    leaq    -16(%rbp), %rax # calc
    movq    %rax, %rdi          # set
    movl    $0, %eax
    call    gets                # call
    . . .
```



Buffer Overflow Stack

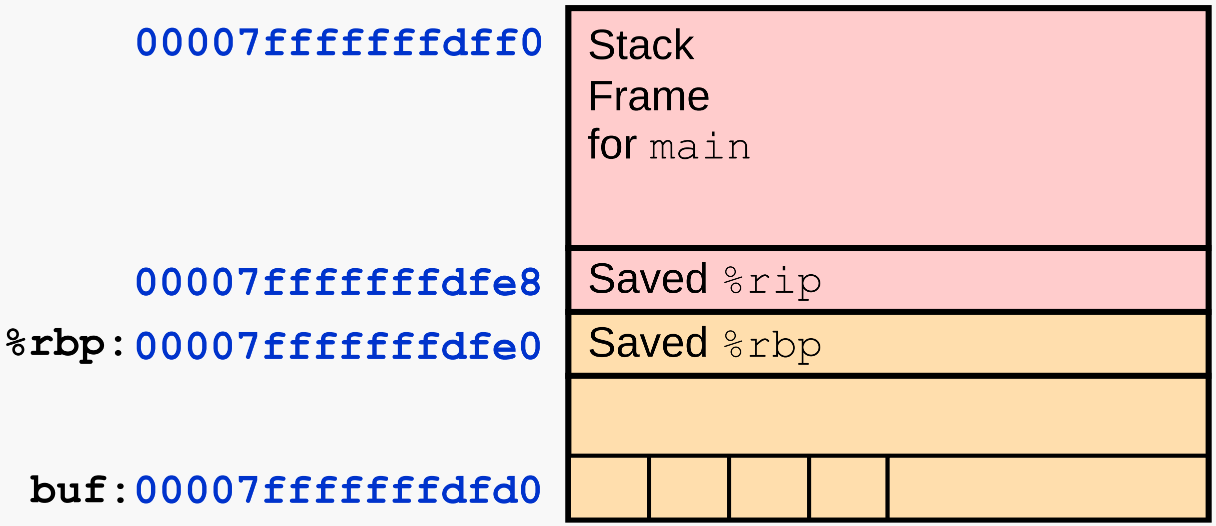



```
(gdb) info f
Stack level 0, frame at 0x7fffffffdff0:
 rip = 0x4005ec in echo (echo.c:4); saved rip 0x4005dd
 called by frame at 0x7fffffffe000
 source language c.
 Arglist at 0x7fffffffdfe0, args:
 Locals at 0x7fffffffdfe0, Previous frame's sp is 0x7fffffffdff0
 Saved registers:
  rbp at 0x7fffffffdfe0, rip at 0x7fffffffdfe8
(gdb) p/a $rbp
$6 = 0x7fffffffdfe0
(gdb) p/a &buf[0]
$7 = 0x7fffffffdfd0
```

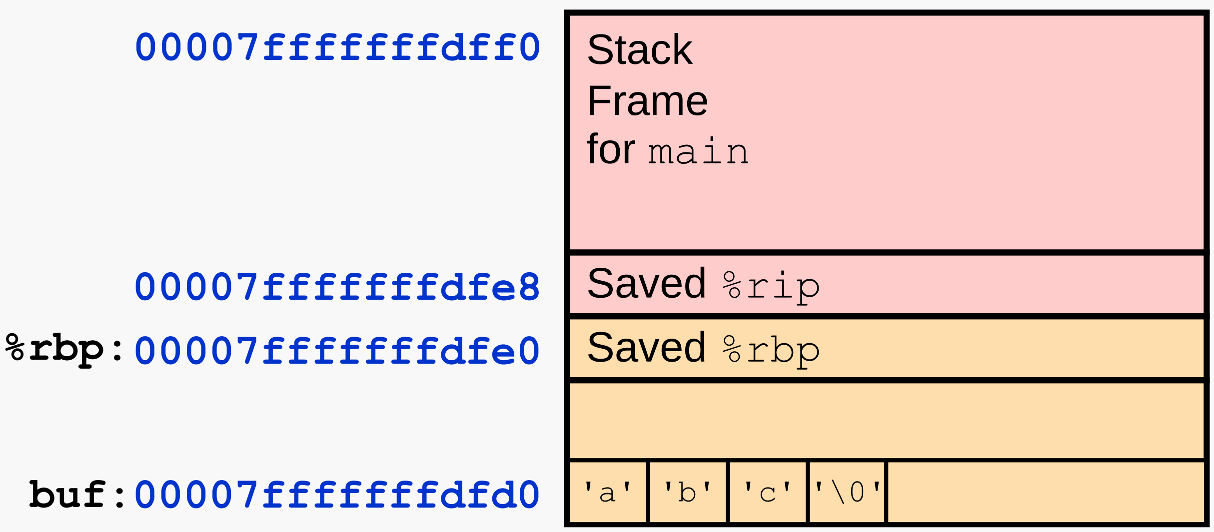

Buffer Overflow Example #1

Before call to gets()

Entered string was "abc"



After call to gets()

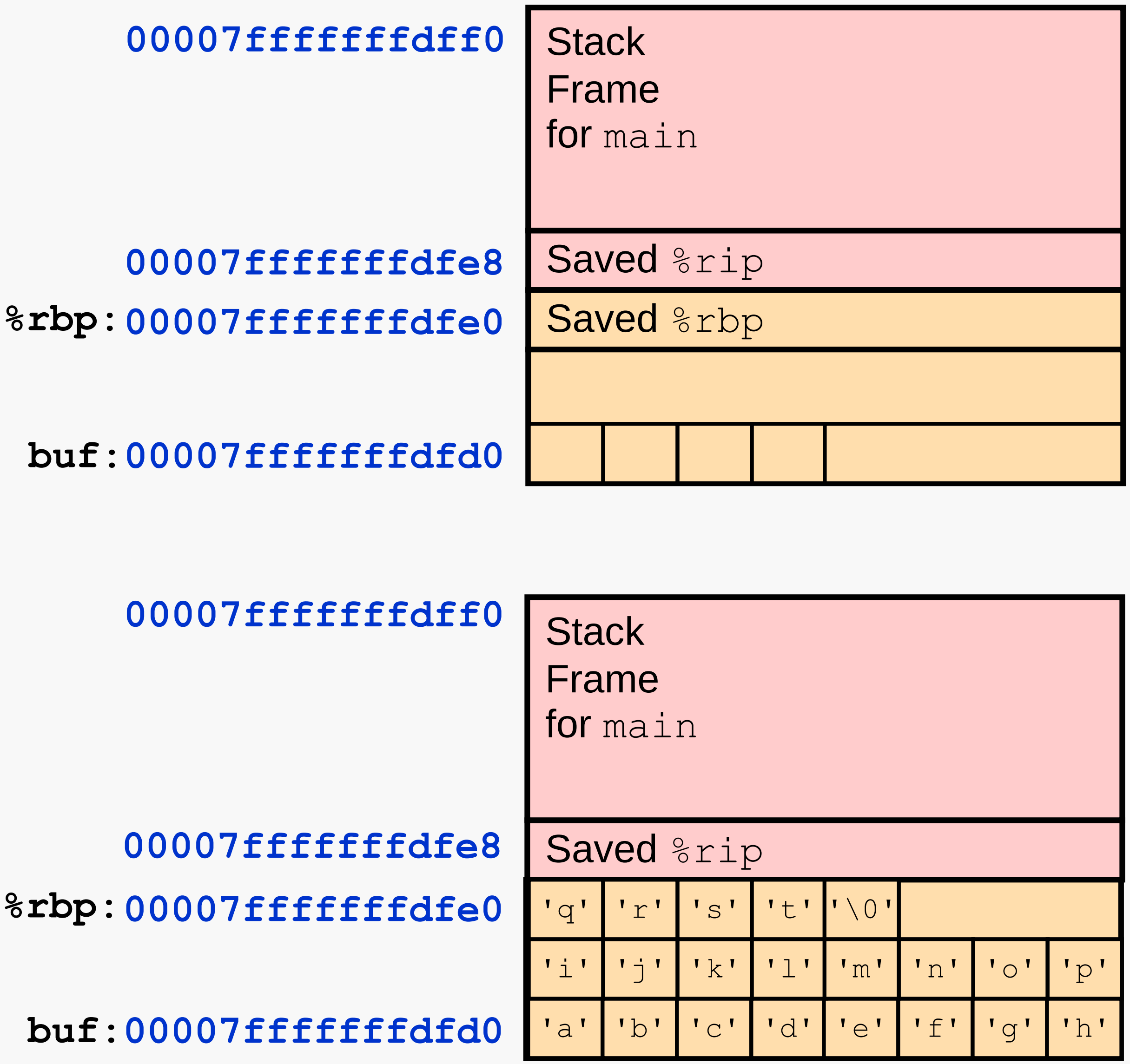


No problem

Buffer Overflow Example #2

Before call to gets()

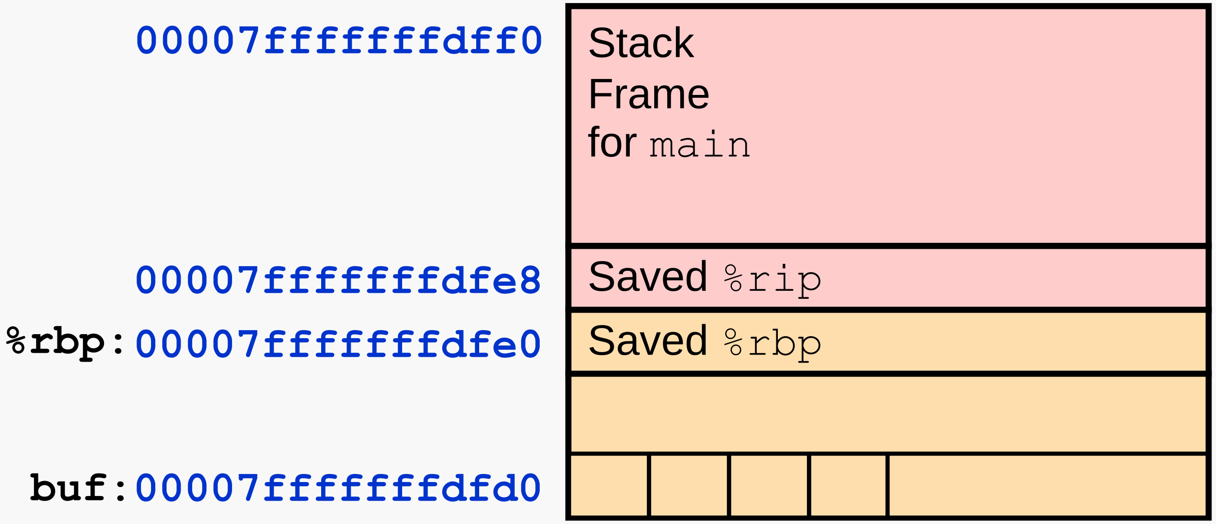
Entered string was "abcd...qrst"



After call to gets()

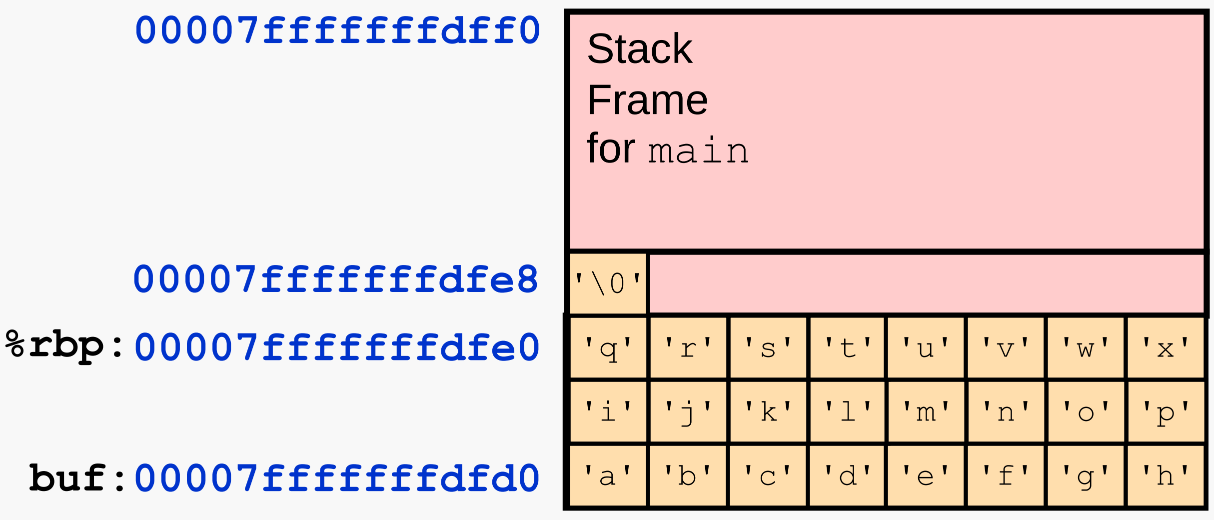
No problem??

Buffer Overflow Example #3



Before call to gets()

Entered string was "abcd...qrstuvwx"



After call to gets()

Problem!!

Demonstration: Generating Byte Codes

Use **gcc** and **objdump** to generate machine code bytes for assembly instruction sequences:

```
CentOS >  cat exploit.s
movq  %rbx, %rdx
addq  $5, %rdx
movq  %rdx, %rax
retq
CentOS >  gcc -c exploit.s
CentOS >  objdump -d exploit.o

exploit.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0:    48 89 da                 mov    %rbx,%rdx
   3:    48 83 c2 05              add    $0x5,%rdx
   7:    48 89 d0                 mov    %rdx,%rax
   a:    c3                       retq
```

Demonstration: Trying the Exploit

Edit the `objdump` output as shown.

`hex2raw` will generate an appropriate binary version of that for input to ctarget:

```
CentOS >  cat exploit.hex
48 89 da                    /* mov    %rbx,%rdx */
48 83 c2 05                 /* add    $0x5,%rdx */
48 89 d0                    /* mov    %rdx,%rax */
c3                          /* retq     */

CentOS >  hex2raw -i exploit.hex
H??H??H???

CentOS >  hex2raw -i exploit.hex | rtarget -q
Cookie: 0x754e7ddd
Type string:No exploit.  Getbuf returned 0x1
Normal return
```

(The strange output from `hex2raw` is due to the presence of bytes in the machine code that correspond to unprintable ASCII characters.)

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

Draw a stack diagram and ROP exploit to:

- **pop the value 0xBBBBBBBB into %rbx,** and
- **move it into %rax**

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

Gadgets:

address_1: `pop %rbx; ret`

address_2: `mov %rbx, %rax; ret`

Inspired by content created by Professor David Brumley

ROP Example: Solution

Gadgets:

Address 1: `pop %rbx; ret`

Address 2: `mov %rbx, %rax; ret`

More ROP entries...
Address 2
0x00000000BBBBBBBB
Address 1 ← old return address was here
0xFFFFFFFFFFFFFFFF
0xFFFFFFFFFFFFFFFF
0xFFFFFFFFFFFFFFFF
0xFFFFFFFFFFFFFFFF
0xFFFFFFFFFFFFFFFF
0xFFFFFFFFFFFFFFFF
0xFFFFFFFFFFFFFFFF
0xFFFFFFFFFFFFFFFF
0xFFFFFFFFFFFFFFFF
0xFFFFFFFFFFFFFFFF ← buf

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

ROP Demonstration: Looking for Gadgets

How to identify useful gadgets in your code:

- a gadget must come from the supplied gadget "farm"
- a gadget must end with the byte `c3` (corresponding to `retq`)
- `pop` and `mov` instructions are useful
- so are carefully-chosen constants

You must use `objdump` to examine `farm.o` for candidate gadgets.

Then, you must determine the correct virtual addresses for those gadgets within `rtarget`.

Use `objdump` to examine `rtarget`...

ROP Demonstration: Looking for Gadgets

Creative "subsetting" of the machine code bytes may be useful:

```
0000000000000000 <start_farm>:
   0:    b8 01 00 00 00              mov    $0x1,%eax
   5:    c3                          retq

0000000000000006 <getval_168>:
   6:    b8 48 89 c7 94              mov    $0x94c78948,%eax
   b:    c3                          retq

000000000000000c <getval_448>:
   c:    b8 7e 58 89 c7              mov    $0xc789587e,%eax
  11:    c3                          retq

0000000000000012 <getval_387>:
  12:    b8 48 89 c7 c3              mov    $0xc3c78948,%eax
  17:    c3                          retq

0000000000000018 <getval_247>:
  18:    b8 18 90 90 90              mov    $0x90909018,%eax
  1d:    c3                          retq

000000000000001e <addval_452>:
  1e:    8d 87 48 89 c7 c3           lea    -0x3c3876b8(%rdi),%eax
  24:    c3                          retq
```

ROP Demonstration: Looking for Gadgets

Gadgets:

Address 1: `pop %rbx; ret`

Address 2: `mov %rbx, %rax; ret`

```
5b c3
```

```
48 89 d8 c3
```

```
4018d2:        c7 07 5b c3 4f 58
4018d8:        c3
. . .
4018e4:        48 89 d8
4018e7:        c3
```

Tools

`objdump -d`

- View byte code and assembly instructions, determine stack offsets

`./hex2raw`

- Pass raw ASCII strings to targets

`gdb`

- Step through execution, determine stack addresses

`gcc -c`

- Generate object file from assembly language file

More Tips

Draw stack diagrams

Be careful of byte ordering (little endian)

READ the assignment specification!

A video presentation of CMU's version is available as Recitation 5 at the link below:

https://scs.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=60c65748-2026-463f-8c57-134fd6661cdf