Lecture 11
Buffer Overflows

CS213 – Intro to Computer Systems
Branden Ghena – Spring 2021

Slides adapted from:
St-Amour, Hardavellas, Bustamente (Northwestern), Bryant, O'Hallaron (CMU), Garcia, Weaver (UC Berkeley)

Northwestern

Administrivia

- Bomb Lab due today
  - TIS-100 raffle for secret phase completion!

- Attack Lab released today
  - Employs all of the attack techniques taught today

  - Get started early! The last two labs are back-to-back

Today's Goals

- Introduce the domain of Computer Security

- Understand buffer overflows and return-oriented programming
  - What enables them
  - How they are used
  - How to protect against them

Why is computer security so important?

- Most public security happens at least in some portion on the honor system
  - Pretty easy to break a window
  - Keyed locks are easy to pick
  - Master keys can be determined and manufactured (Matt Blaze attack)
  - Laws apply after you've done it
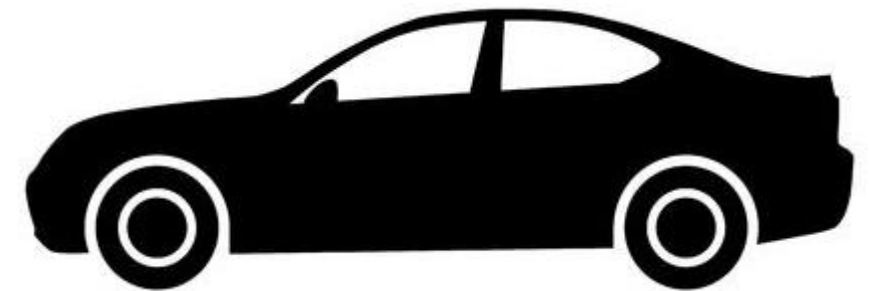
Early computers didn't have any security either

- Simple machines for doing computation do not have private files or contention

- Sometimes there were multiple users, but all were employees of the same company
  - Permissions needed to be as secure as a file in a locked drawer on a desk

"The act of breaking into a computer system has to have the same social stigma as breaking into a neighbor's house. It should not matter that the neighbor's door is unlocked."
- Ken Thompson, Turing Award Lecture, 1984

Connectivity of computers makes security a top concern

- Security of physical items is dependent on the fact that one person can only steal one thing at a time
  - And it's usually obvious when theft occurs
  - Not the case for private information on a computer

- The internet makes security incredibly important
  - Usually not people breaking into computers manually, one at a time
  - Instead it is computers breaking into computers by means of scripting
  - And you can access a computer from anywhere on Earth

- Breaking into or controlling one car is a crime
  - Controlling 100,000 cars remotely is a problem for the manufacturer

Outline

- **Buffer Overflows**

- Protecting Against Buffer Overflows

- Return-Oriented Programming

- Protecting Against Return-Oriented Programming

Memory Referencing Bug Example

```c
typedef struct {
  int a[2];
  double d;
} struct_t;

double fun(int i) {
  volatile struct_t s; // volatile ≈ don't optimize this away
  s.d = 3.14;
  s.a[i] = 1073741824; // Possibly out of bounds
  return s.d;
}
```

```
fun(0)  ∝    3.14
fun(1)  ∝    3.14
fun(2)  ∝    3.1399998664856
fun(3)  ∝    2.00000061035156
fun(4)  ∝    3.14
fun(5)  ∝    3.14
fun(6)  ∝    Segmentation fault (core dumped)
```
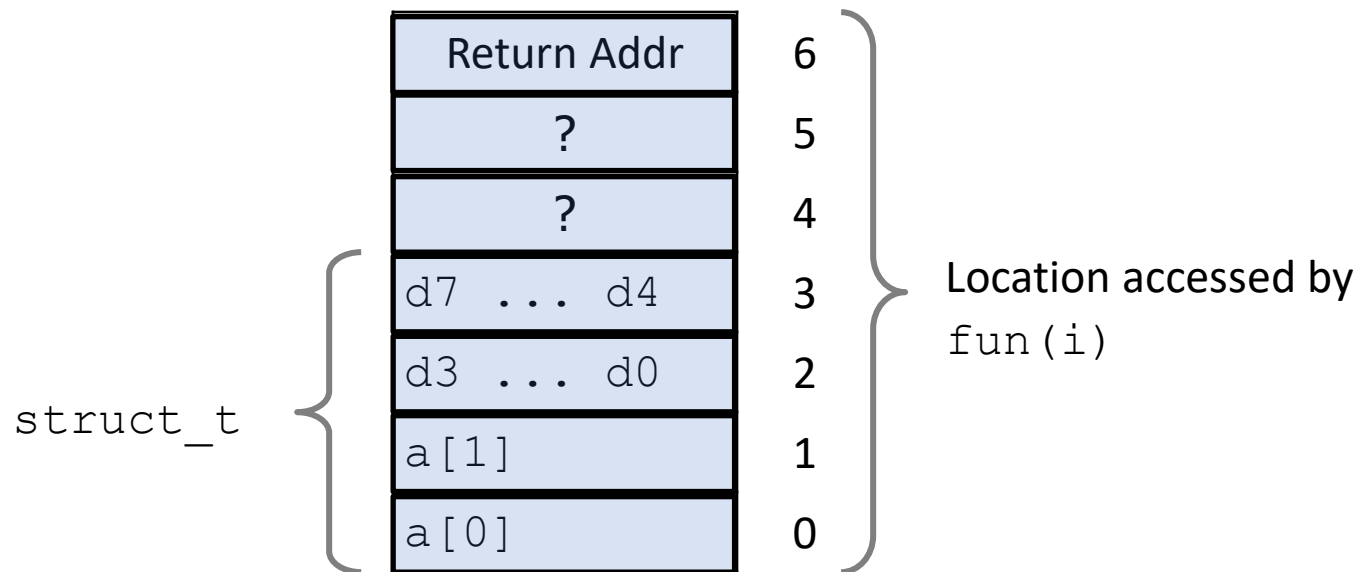
- Abuses undefined behavior
- Result is system specific

Memory Referencing Bug Example

```
typedef struct {
    int a[2];
    double d;
} struct_t;
```

fun(0)	∝	3.14
fun(1)	∝	3.14
fun(2)	∝	3.1399998664856
fun(3)	∝	2.00000061035156
fun(4)	∝	3.14
fun(5)	∝	3.14
fun(6)	∝	Segmentation fault

Explanation:

Return Addr	6	
?	5	
?	4	Location accessed by
d7 ... d4	3	`fun(i)`
d3 ... d0	2	
a[1]	1	
a[0]	0	

`struct_t`

Such problems are a *BIG* deal

- Generally called a "buffer overflow"
  - Going past end of memory allocated for an array (AKA buffer)

- Why is it a big deal?
  - #1 *technical* cause of security vulnerabilities
    - (#1 overall cause is social engineering)

- Most common form
  - Unchecked lengths on string inputs
  - Particularly with character arrays on the stack
    - Sometimes referred to as "stack smashing"

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 = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

No bounds checking!

- Similar problems with other Unix functions
  - **strcpy**, **strcat**: Copies string of arbitrary length
  - **scanf**, **fscanf**, **sscanf**, when given **%s** specifier

Vulnerable buffer code

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

```
void call_echo() {
  echo();
}
```

```
/* Prints whatever is read */
void echo(){
  char buf[4]; /* Way too small! */
  gets(buf);
  puts(buf);
}
```

```
unix>./bufdemo-nsp
Type a string:012
012
```

```
unix>./bufdemo-nsp
Type a string:0123456789012345678901
0123456789012345678901
```

Much more than 4 characters!

```
unix>./bufdemo-nsp
Type a string:012345678901234567890123 4
Segmentation Fault
```

Buffer Overflow Disassembly

echo:

```
00000000004006cf <echo>:
 4006cf:   48 83 ec 18              sub     $24,%rsp
 4006d3:   48 89 e7                 mov     %rsp,%rdi
 4006d6:   e8 a5 ff ff ff           callq   400680 <gets>
 4006db:   48 89 e7                 mov     %rsp,%rdi
 4006de:   e8 3d fe ff ff           callq   400520 <puts@plt>
 4006e3:   48 83 c4 18              add     $24,%rsp
 4006e7:   c3                       retq
```
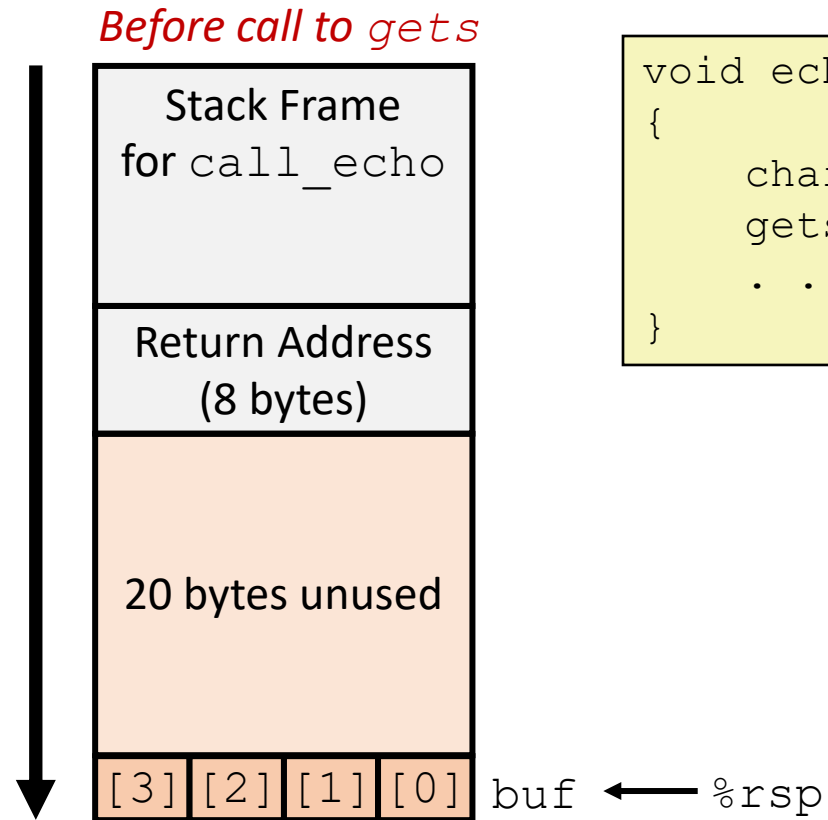
call_echo:

```
00000000004006e8 <call_echo>:
 4006e8:   48 83 ec 08              sub     $8,%rsp
 4006ec:   b8 00 00 00 00           mov     $0,%eax
 4006f1:   e8 d9 ff ff ff           callq   4006cf <echo>
 4006f6:   48 83 c4 08              add     $8,%rsp
 4006fa:   c3                       retq
```

Buffer Overflow Stack

Before call to `gets`

Stack Frame for `call_echo`
Return Address (8 bytes)
20 bytes unused
`[3]` `[2]` `[1]` `[0]`

`buf` ← `%rsp`
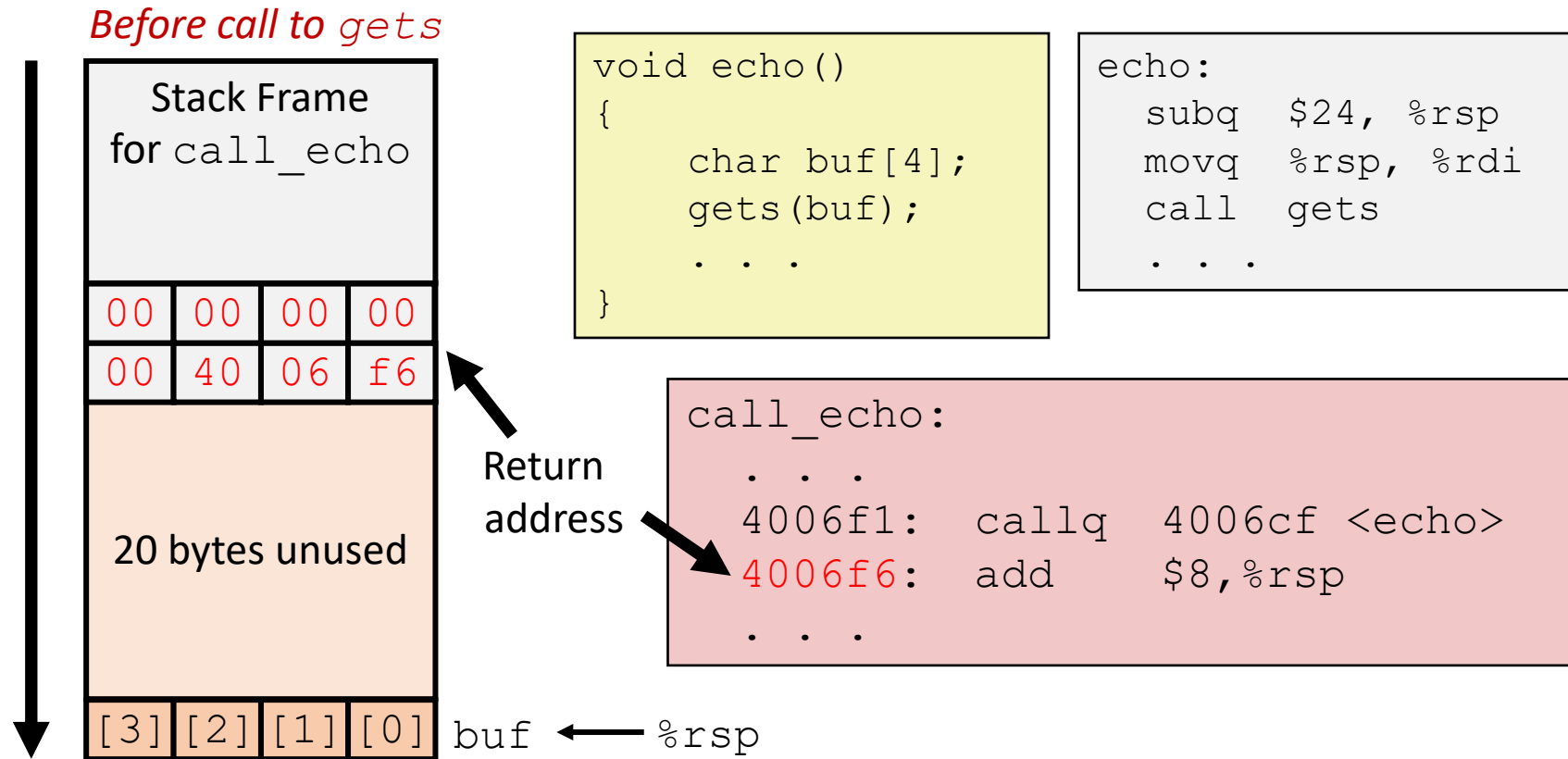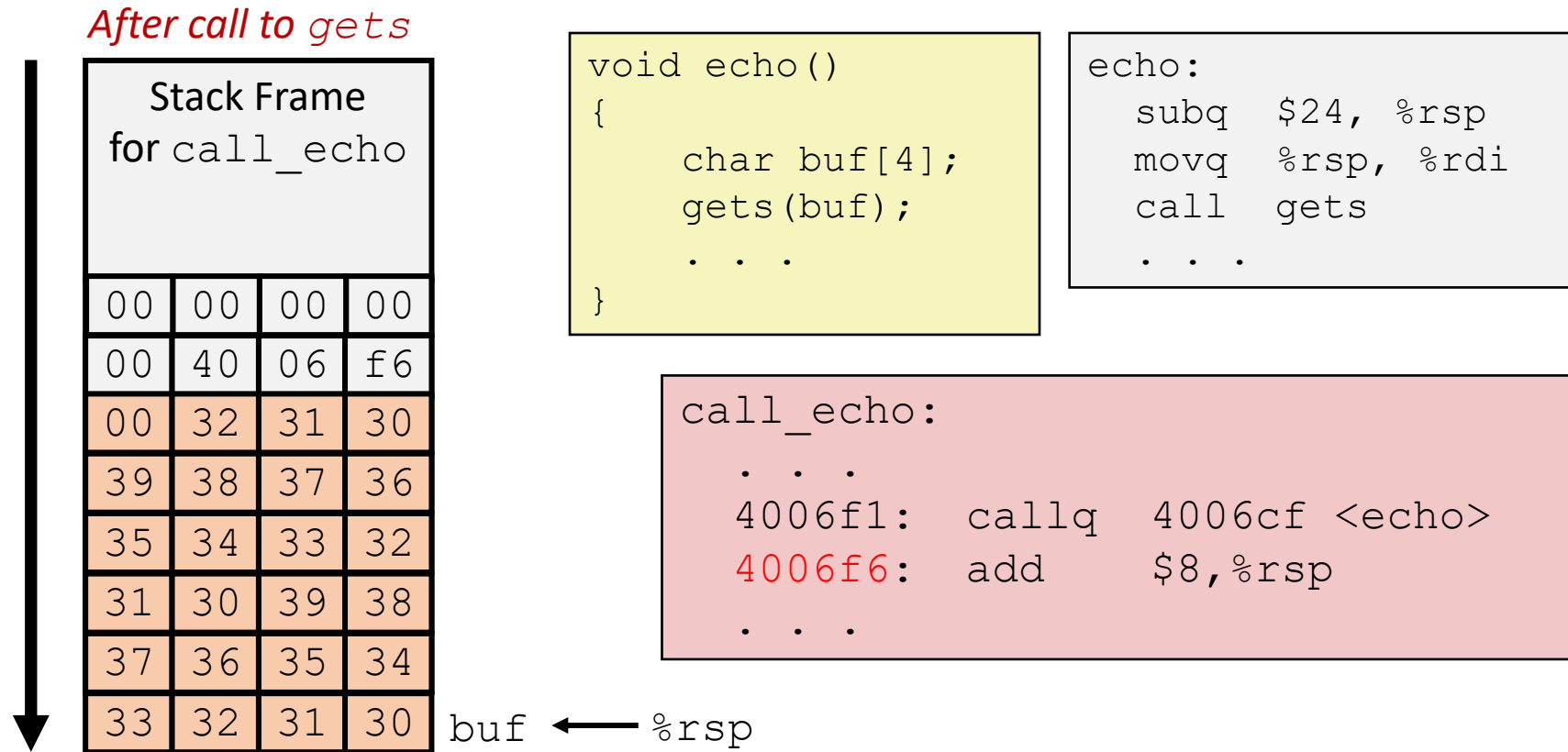
```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
  subq  $24, %rsp
  movq  %rsp, %rdi
  call  gets
  . . .
```

Buffer Overflow Stack Example

Before call to `gets`


```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
  subq  $24, %rsp
  movq  %rsp, %rdi
  call  gets
  . . .
```

00	00	00	00
00	40	06	f6

Stack Frame for call_echo

20 bytes unused

Return address

```
call_echo:
  . . .
  4006f1:  callq  4006cf <echo>
  4006f6:  add    $8,%rsp
  . . .
```

[3] [2] [1] [0] buf ← %rsp

Buffer Overflow Stack Example #1

After call to `gets`

Stack Frame for `call_echo`			
00	00	00	00
00	40	06	f6
00	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf ← %rsp

```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
   subq  $24, %rsp
   movq  %rsp, %rdi
   call  gets
   . . .
```

```
call_echo:
   . . .
   4006f1:  callq  4006cf <echo>
   4006f6:  add     $8,%rsp
   . . .
```
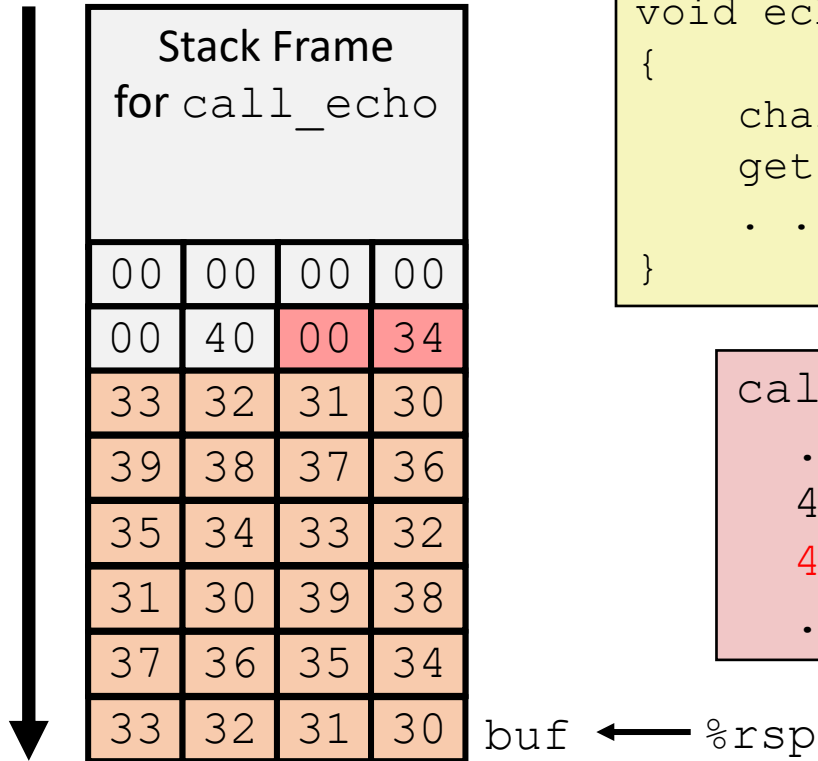
```
unix>./bufdemo-nsp
Type a string:01234567890123456789012
01234567890123456789012
```

Overflowed buffer, but did not corrupt state

16

Buffer Overflow Stack Example #2

After call to `gets`

Stack Frame for `call_echo`			
00	00	00	00
00	40	00	34
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf ← %rsp

```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
    subq   $24, %rsp
    movq   %rsp, %rdi
    call   gets
    . . .
```

```
call_echo:
    . . .
    4006f1:  callq   4006cf <echo>
    4006f6:  add     $8,%rsp
    . . .
```

Is it a string?
Is it an address?
Depends on context!

```
unix>./bufdemo-nsp
Type a string:01234567890123456789012340
Segmentation Fault
```

Overflowed buffer and corrupted return address. Could point to unmapped memory, etc.
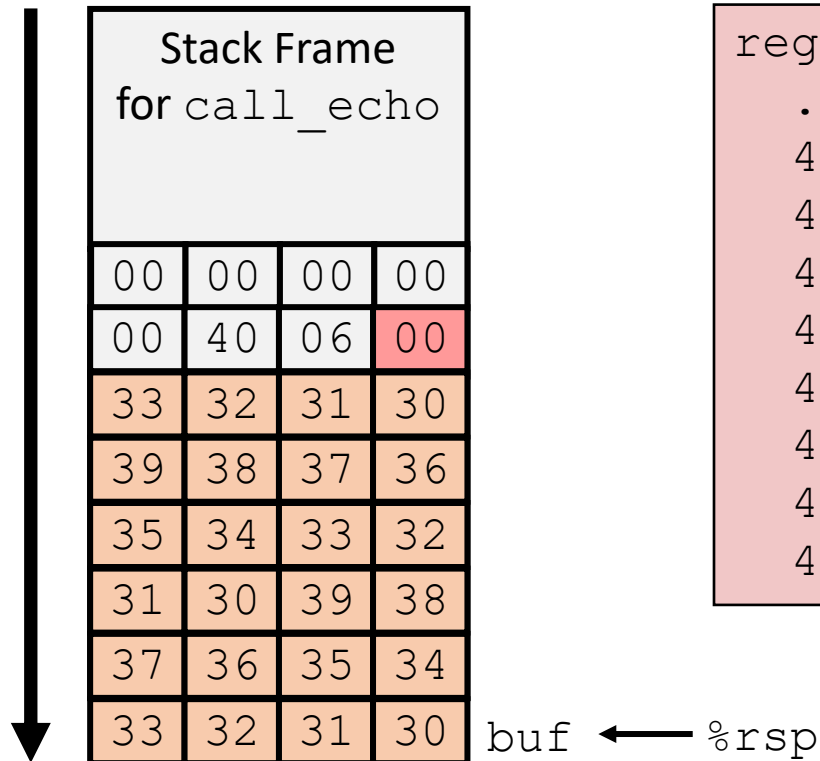
Buffer Overflow Stack Example #3

After call to `gets`

Stack Frame
for `call_echo`

00	00	00	00
00	40	06	00
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf ← %rsp

```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
    subq  $24, %rsp
    movq  %rsp, %rdi
    call  gets
    . . .
```

```
call_echo:
    . . .
    4006f1:  callq   4006cf <echo>
    4006f6:  add     $8,%rsp
    . . .
```
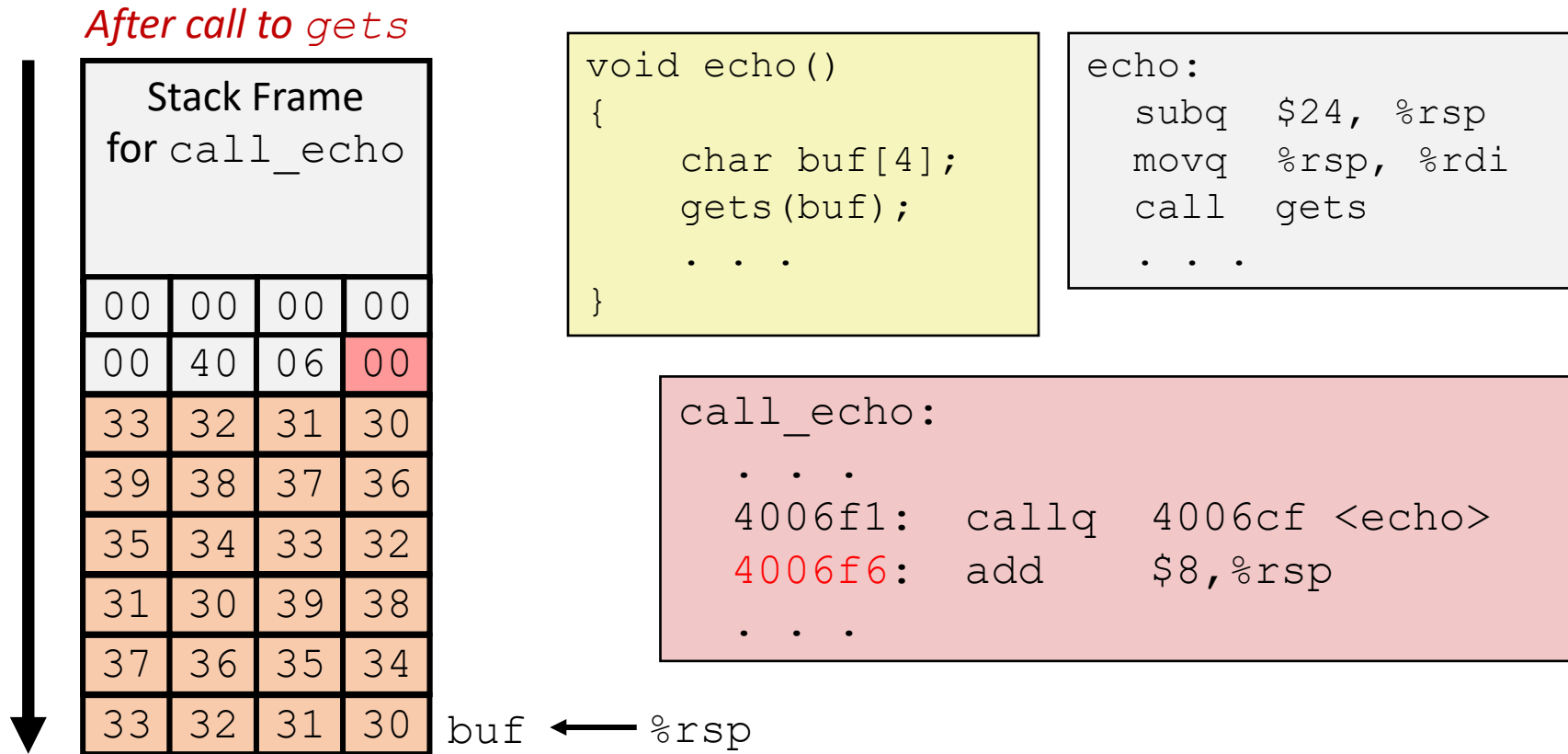
```
unix>./bufdemo-nsp
Type a string:01234567890123456789 0123
01234567890123456789 0123
```

Overflowed buffer, corrupted return address, but program *seems* to work! Latent bug!

18

Buffer Overflow Stack Example #3 Explained
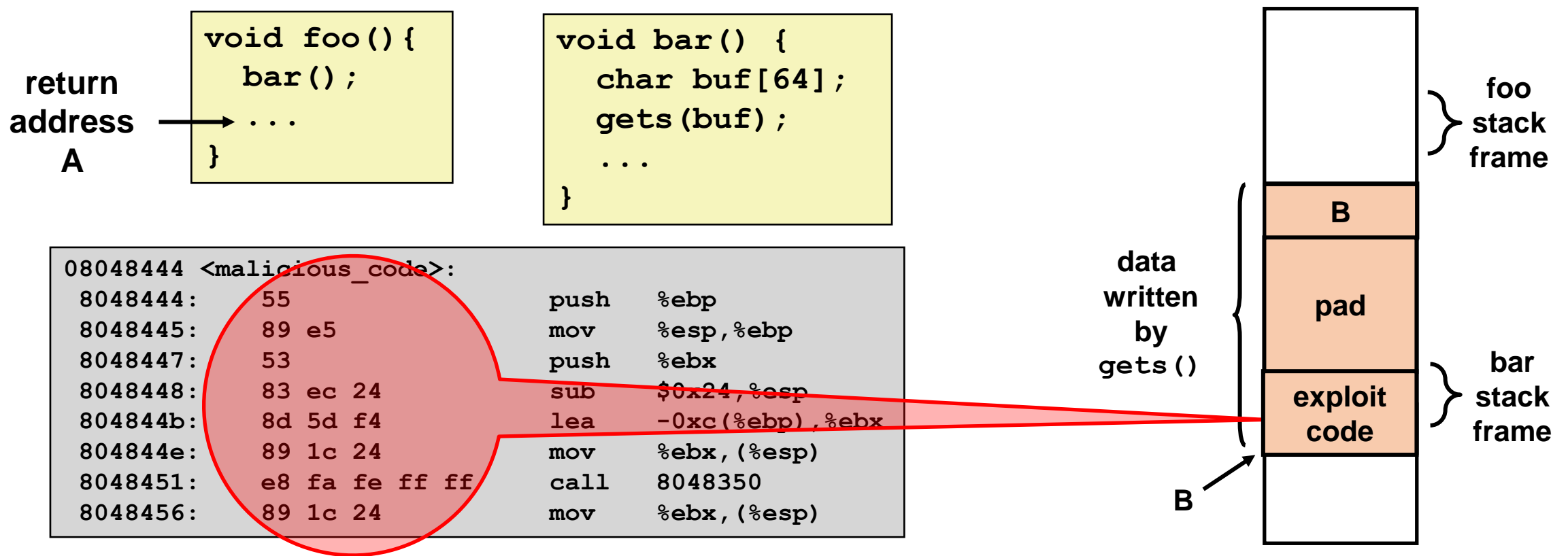
After call to `gets`

Stack Frame for `call_echo`			
00	00	00	00
00	40	06	00
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

`buf` ⟵ `%rsp`

```
register_tm_clones:
    . . .
    400600:   mov     %rsp,%rbp
    400603:   mov     %rax,%rdx
    400606:   shr     $0x3f,%rdx
    40060a:   add     %rdx,%rax
    40060d:   sar     %rax
    400610:   jne     400614
    400612:   pop     %rbp
    400613:   retq
```

"Returns" to unrelated code
Lots of things happen, without modifying critical state
Eventually executes `retq` back to `main` as if nothing happened…

Malicious use of buffer overflow

```
void foo(){
  bar();
  ...
}
```

```
void bar() {
  char buf[64];
  gets(buf);
  ...
}
```

return address A

```
08048444 <malicious_code>:
 8048444:    55              push    %ebp
 8048445:    89 e5           mov     %esp,%ebp
 8048447:    53              push    %ebx
 8048448:    83 ec 24        sub     $0x24,%esp
 804844b:    8d 5d f4        lea     -0xc(%ebp),%ebx
 804844e:    89 1c 24        mov     %ebx,(%esp)
 8048451:    e8 fa fe ff ff  call    8048350
 8048456:    89 1c 24        mov     %ebx,(%esp)
```

foo stack frame

B

data written by `gets()`

pad

exploit code

bar stack frame

B

- Input string contains binary representation of executable code
- Overwrite return address with address of buffer
- When `bar()` returns, where do we go?
  - Into the beginning of `malicious_code` on the stack! 😱

Memory Address 0

20

Exploits based on buffer overflows

- Buffer overflow bugs can allow remote machines to execute arbitrary code on victim machines

- Distressingly common in real programs
  - Programmers keep making the same mistakes 😭
  - Recent measures make these attacks much more difficult

- Examples across the decades
  - Original "Internet worm" (1988)
    - Attacked `fingerd` server, replicated itself across the internet
  - Stuxnet (2010)
    - Attack on Iran nuclear program, malicious code destroyed centrifuges
  - … and many, many more

- You will learn some of these tricks with the attack lab
  - Hopefully convincing you to never leave such holes in your programs!

Outline

- Buffer Overflows

- **Protecting Against Buffer Overflows**

- Return-Oriented Programming

- Protecting Against Return-Oriented Programming

1. Avoiding Buffer Overflow Vulnerability

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    fgets(buf, 4, stdin); /* length limit! */
    puts(buf);
}
```

- Use safe library routines that limit string lengths
  - **fgets** instead of **gets**
  - **strncpy** instead of **strcpy**
  - Don't use **scanf** with **%s** conversion specification
    - Use **fgets** to read the string
    - Or use format specifier **%ns** where **n** is a suitable integer
- Also: don't write your programs in C, when possible
  - Fundamental design of C is to be fast, not to be secure

2. System-Level Protection: Randomized Stack

- Buffer overflow attack requires knowing the *absolute* address of the buffer
  - To overwrite return address to that

- At start of program, allocate a random amount of space on stack
  - Different every time the program runs

- Shifts stack addresses for entire program
  - Program still runs fine
  - Legitimate accesses to the stack are **relative** to `%rsp`

- But absolute addresses get randomly shifted
  - Don't know what return address should be!
  - Still not impossible to overcome (NOP sled)

Stack base

Random allocation

`main`

Application Code

B?

pad

exploit code

B? →
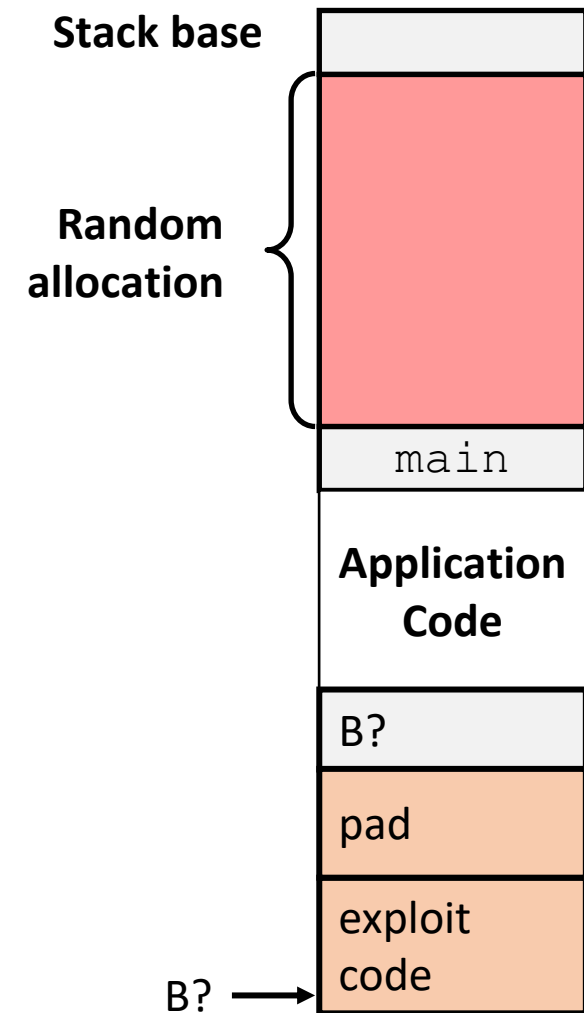
3. System-Level Protection: Explicit Execute Page Permissions

- Non-executable stack
  - On x86-64, can mark a region of memory as "non-executable"
  - Trying to execute something in that region → crash
  - More about page permissions in the virtual memory lecture

- OpenBSD goes further: W^X
  - A region of memory can be writeable or executable, but not both (xor!)
  - Causes trouble for JITs

Stack after call to `gets()`

P stack frame

B

data written by `gets()`

pad

exploit code

Q stack frame

B

Any attempt to execute this code will fail
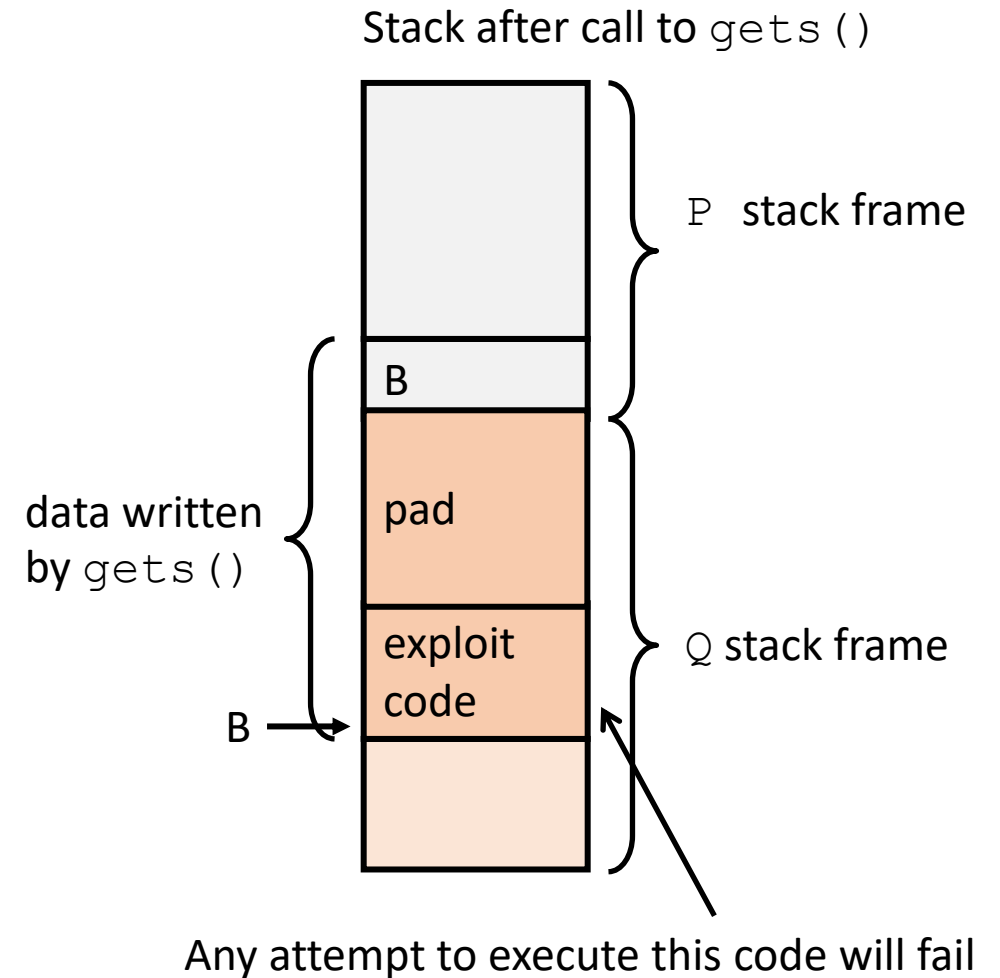
Break + Open Question

- **Why is a buffer overflow in a web browser so bad?**

Break + Open Question

- **Why is a buffer overflow in a web browser so bad?**

  - The buffer overflow will exist in *at least* all instances of the same version of the web browser installed on the same OS and architecture
    - Possibly many other versions too

  - If it can be triggered from a website, then you could run malicious code on computers without any manual effort
    - Any website could be suspect

  - Scale is enormous: Chrome has 2.65 billion users in 2020

Outline

- Buffer Overflows

- Protecting Against Buffer Overflows

- **Return-Oriented Programming**

- Protecting Against Return-Oriented Programming

Finding a new way to abuse a vulnerability

- Buffer overflows are still happening

- Even if they can't place malicious code directly on the stack, they can modify return addresses

- We can use that idea to build an attack from pieces of already existing program code that we reuse for malicious purposes
  - This is one of those ideas that sounds impossible to pull off in the real world
  - But actually it totally works and we'll have you do it in the attack lab!

Return-Oriented Programming (ROP)

- Challenge (for would-be hackers)
  - Stack randomization → predicting buffer location is hard
    - So it's hard to know where to jump and start executing

  - Making stack non-executable → injecting code doesn't work
    - We can inject anything we want, but we can't run it

- Alternative strategy: Don't inject your own code!
  - Use code that's already in the program!
  - It's in a predictable location!
    - Otherwise, don't know where to call/jump
  - It's executable
    - Otherwise, the program wouldn't run at all…

Return-Oriented Programming (ROP)

- But wait, the code I want to run isn't in the program!
  - Unlikely that, e.g., a mail client includes code to, e.g., launch missiles

- Key idea: construct the code you want to run from pieces that you find in the program!
  - We'll call these pieces **gadgets**

- Strategy: find machine code fragments that do *one small step* of the malicious program you want to run, then return
  - Then we'll put these small steps together to get the whole program
  - These return instructions will be the glue that tie them together

- "The program" includes the standard library!
  - Things like `printf`, `scanf`, etc.
  - That's a lot of code! So lots of gadgets to choose from

Gadget Examples

- Use the end of existing functions

```
long ab_plus_c
    (long a, long b, long c){
  return a*b + c;
}
```

```
00000000004004d0 <ab_plus_c>:
  4004d0:  48 0f af fe      imul %rsi,%rdi
  4004d4:  48 8d 04 17      lea (%rdi,%rdx,1),%rax
  4004d8:  c3               ret
```

Gadget: `rax ← rdi + rdx`
Address: `0x4004d4`

- Repurpose parts of instructions

```
void setval
    (unsigned *p) {

  *p = 3347663060u;
}
```

```
00000000004004d9 <setval>:
  4004d9:  c7 07 d4 48 89 c7   movl $0xc78948d4,(%rdi)
  4004df:  c3                  ret
```

Encodes: `movq %rax, %rdi`
Gadget: `rdi ← rax`
Address: `0x4004dc`

Combining Gadgets

- Let's say our malicious program is this:
  `%rax = (%rbx × %rcx) + %rdi`

- And let's say we found the following gadgets in the standard library

```
0000000000400474 <g1>:
  400474: 48 0f af cb    imul %rbx,%rcx
  400478: c3             retq

0000000000400479 <g2>:
  400479: 48 01 cf       add %rcx,%rdi
  40047c: c3             retq

000000000040047d <g3>:
  40047d: 48 89 f8       mov %rdi,%rax
  400480: c3             retq
```

Given a large enough standard library, can find gadgets that do pretty much anything we want! Enough code to pick from.

40047d
400479
400474	← `%rsp`
...
...	← `buf`

- Combine gadgets by adding pointers to them to the stack
  - Arrange on the stack by overflowing a buffer, like before
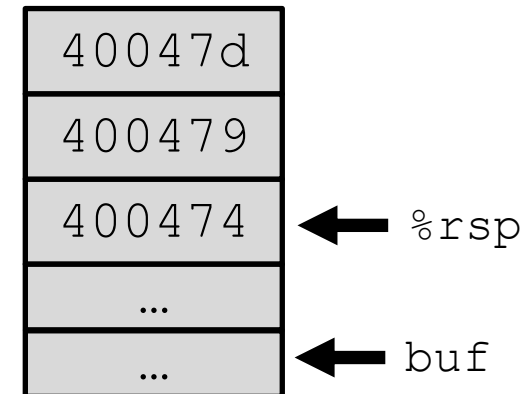
Gadget Execution

- Step 1: we overflowed the buffer, like before
  - We set up the stack with the gadget addresses, as on last slide
  - Now we're about to return from the vulnerable function (`echo`)

```
00000000004006cf <echo>:
 ...
 4006d6:  e8 a5 ff ff ff          callq  400680 <gets>
 ...
 4006e7:  c3                      retq
```

```
0000000000400474 <g1>:
  400474: 48 0f af cb     imul %rbx,%rcx
  400478: c3              retq
0000000000400479 <g2>:
  400479: 48 01 cf        add %rcx,%rdi
  40047c: c3              retq
000000000040047d <g3>:
  40047d: 48 89 f8        mov %rdi,%rax
  400480: c3              retq
```

`%rip = 4006e7`

…
40047d
400479
400474	← `%rsp`
…
…	← buf
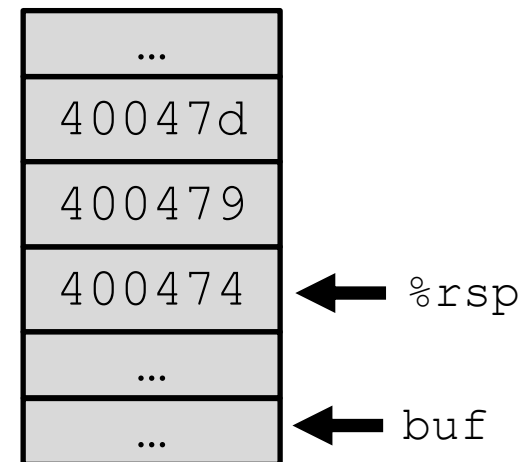
Gadget Execution

- Step 2: return from `echo`
  - Get the return address from **%rsp**
  - Oh, that's the address of the first gadget!

```
00000000004006cf <echo>:
  ...
  4006d6:  e8 a5 ff ff ff          callq  400680 <gets>
  ...
→ 4006e7:  c3                      retq
```

```
0000000000400474 <g1>:
  400474: 48 0f af cb    imul %rbx,%rcx
  400478: c3             retq
0000000000400479 <g2>:
  400479: 48 01 cf       add %rcx,%rdi
  40047c: c3             retq
000000000040047d <g3>:
  40047d: 48 89 f8       mov %rdi,%rax
  400480: c3             retq
```
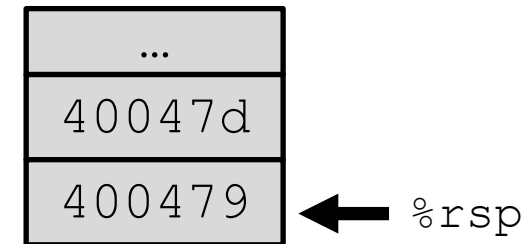
`%rip` = 400474

...
40047d
400479
400474	← `%rsp`
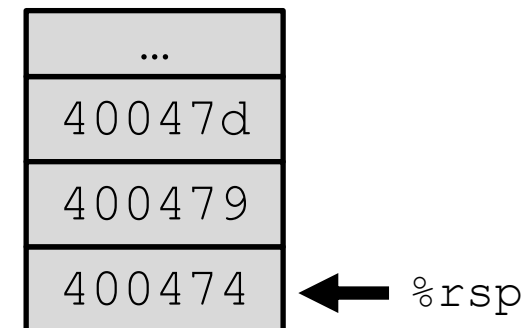
Gadget Execution

- Step 3: run the first gadget
  - **%rcx = %rbx × %rcx**

```
00000000004006cf <echo>:
 ...
 4006d6:  e8 a5 ff ff ff          callq  400680 <gets>
 ...
 4006e7:  c3                      retq
```

```
0000000000400474 <g1>:
 400474: 48 0f af cb     imul %rbx,%rcx
 400478: c3              retq
0000000000400479 <g2>:
 400479: 48 01 cf        add %rcx,%rdi
 40047c: c3              retq
000000000040047d <g3>:
 40047d: 48 89 f8        mov %rdi,%rax
 400480: c3              retq
```

`%rip = 400474`

...
40047d
400479

`%rsp`
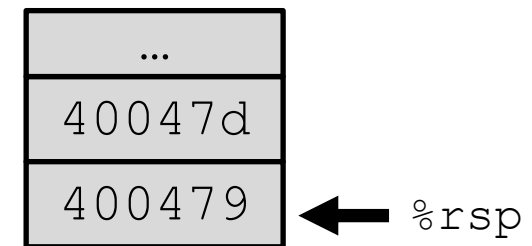
Gadget Execution

- Step 4: return from the first gadget
  - Get the return address from `%rsp`
  - **QUIZ**: where do we go next?   `400479`, that's gadget 2!

```
00000000004006cf <echo>:
  ...
  4006d6:  e8 a5 ff ff ff          callq  400680 <gets>
  ...
  4006e7:  c3                      retq
```
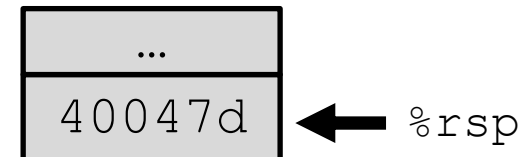
```
0000000000400474 <g1>:
  400474:  48 0f af cb    imul %rbx,%rcx
  400478:  c3             retq
0000000000400479 <g2>:
  400479:  48 01 cf       add %rcx,%rdi
  40047c:  c3             retq
000000000040047d <g3>:
  40047d:  48 89 f8       mov %rdi,%rax
  400480:  c3             retq
```

`%rip = 400478`

...
40047d
400479	← `%rsp`

Gadget Execution

- Step 5: run the second gadget
    - **%rdi = (%rbx × %rcx) + %rdi**

```
00000000004006cf <echo>:
 ...
 4006d6:  e8 a5 ff ff ff          callq  400680 <gets>
 ...
 4006e7:  c3                      retq
```

```
0000000000400474 <g1>:
  400474: 48 0f af cb     imul %rbx,%rcx
  400478: c3              retq
0000000000400479 <g2>:
  400479: 48 01 cf        add %rcx,%rdi
  40047c: c3              retq
000000000040047d <g3>:
  40047d: 48 89 f8        mov %rdi,%rax
  400480: c3              retq
```

%rip = 400479

```
  …
40047d    ← %rsp
```
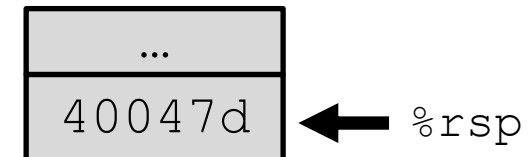
Gadget Execution

- Step 6: return from the second gadget
  - Get the return address from `%rsp`
  - Oh, that's the address of the third gadget!

```
00000000004006cf <echo>:
  ...
  4006d6:   e8 a5 ff ff ff          callq  400680 <gets>
  ...
  4006e7:   c3                      retq
```

```
0000000000400474 <g1>:
  400474:   48 0f af cb     imul %rbx,%rcx
  400478:   c3              retq
0000000000400479 <g2>:
  400479:   48 01 cf        add %rcx,%rdi
  40047c:   c3              retq
000000000040047d <g3>:
  40047d:   48 89 f8        mov %rdi,%rax
  400480:   c3              retq
```

`%rip` = 40047d

| … |
| 40047d | ← `%rsp`

Gadget Execution

- Step 7: run the third gadget
  - **%rax = (%rbx × %rcx) + %rdi**
  - We've run the program we wanted to run. Our job is done.

```
00000000004006cf <echo>:
  ...
  4006d6:  e8 a5 ff ff ff          callq  400680 <gets>
  ...
  4006e7:  c3                      retq
```

```
0000000000400474 <g1>:
  400474: 48 0f af cb     imul %rbx,%rcx
  400478: c3              retq
0000000000400479 <g2>:
  400479: 48 01 cf        add %rcx,%rdi
  40047c: c3              retq
000000000040047d <g3>:
  40047d: 48 89 f8        mov %rdi,%rax
  400480: c3              retq
```

$\texttt{\%rip} = \texttt{40047d}$

```
        …
```
%rsp

Gadget Execution

- Step 8: Return from the third gadget
  - At this point, return to whatever address we find on the stack.
  - That's past the data we put there ourselves, so it's whatever was there before. Maybe not meant to be an address! Could be anything!

- But we don't care about what the program does anymore!
  - We've run the code we wanted to run, nothing else matters!
  - (Maybe we stole from bank accounts, launched missiles, etc.)

```
0000000000400474 <g1>:
  400474: 48 0f af cb      imul %rbx,%rcx
  400478: c3               retq

0000000000400479 <g2>:
  400479: 48 01 cf         add %rcx,%rdi
  40047c: c3               retq

000000000040047d <g3>:
  40047d: 48 89 f8         mov %rdi,%rax
  400480: c3               retq
```
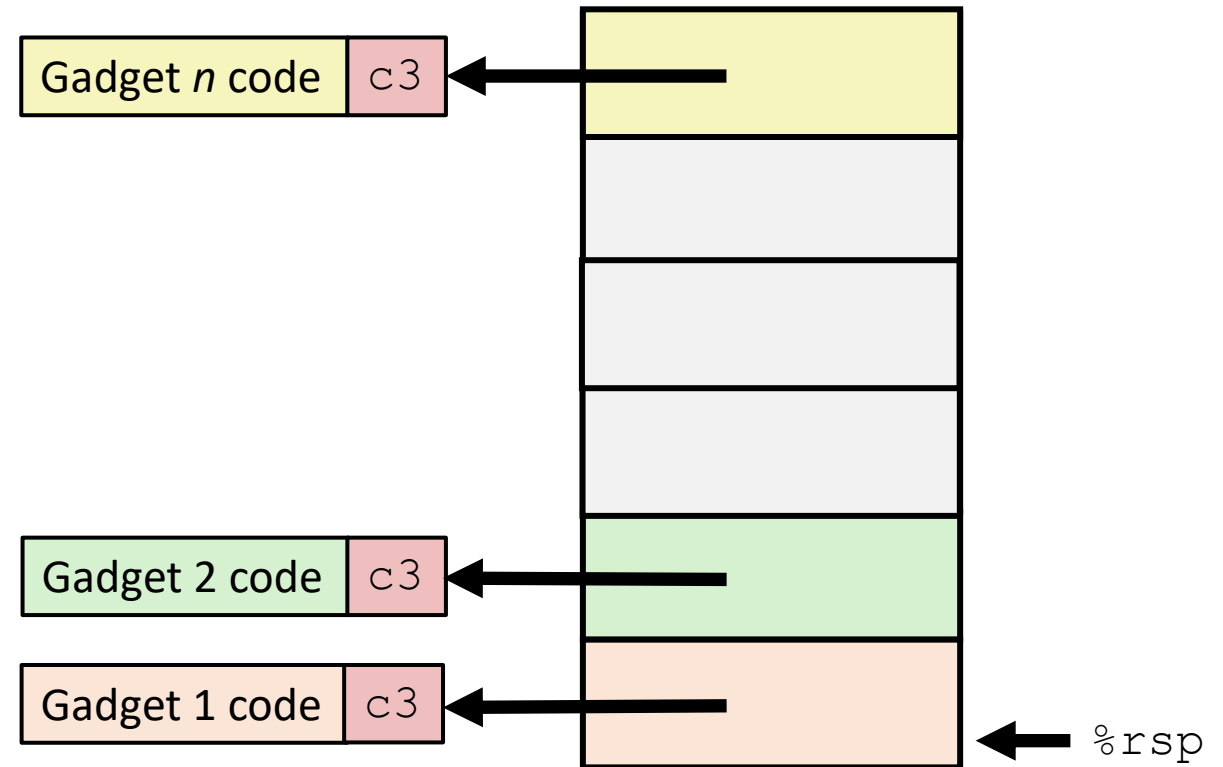
`%rip = ???`

... ← `%rsp`

Return-Oriented Programming Execution

- Trigger with `ret` instruction in the current function

- "Returns" to gadget 1, instead of to its caller

- Gadget 1 does its thing, then returns to gadget 2, etc.
  - Repeat as necessary

- Complete! You've "run" the "function" you wanted to run!

| Gadget *n* code | `c3` |

| Gadget 2 code | `c3` |

| Gadget 1 code | `c3` |

`%rsp`

Outline

- Buffer Overflows

- Protecting Against Buffer Overflows

- Return-Oriented Programming

- **Protecting Against Return-Oriented Programming**

1. Avoiding buffer overflow vulnerabilities

- Write better code please

- Return-oriented programming starts with a buffer overflow
    - To set up gadget addresses on the stack

- No buffer overflow, no return-oriented programming!

2. Stack Canaries

- Idea
  - Place special value ("canary") on stack just beyond buffer
  - Check for corruption before exiting function
  - So we can detect buffer overflows **before** we run malicious code
    - Then just crash the program instead of doing bad things
  - Analogy: canary in a coal mine

- GCC Implementation
  - **-fstack-protector**
  - Now the default for potentially vulnerable functions
    - (disabled in attack lab to show the vulnerability)

```
unix>./bufdemo-sp
Type a string:0123456
0123456
```

```
unix>./bufdemo-sp
Type a string:01234567
*** stack smashing detected ***
```

2. Stack Canaries - Disassembly

echo:

```
40072f:   sub     $0x18,%rsp
400733:   mov     %fs:0x28,%rax
40073c:   mov     %rax,0x8(%rsp)
400741:   xor     %eax,%eax
400743:   mov     %rsp,%rdi
400746:   callq   4006e0 <gets>
40074b:   mov     %rsp,%rdi
40074e:   callq   400570 <puts@plt>
400753:   mov     0x8(%rsp),%rax
400758:   xor     %fs:0x28,%rax
400761:   je      400768 <echo+0x39>
400763:   callq   400580 <__stack_chk_fail@plt>
400768:   add     $0x18,%rsp
40076c:   retq
```
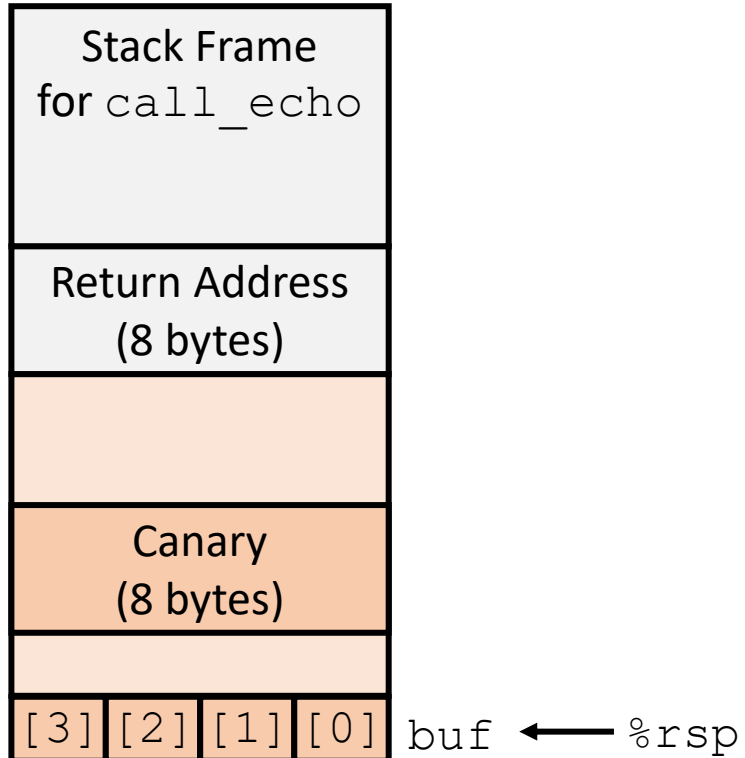
Read value from a special, read-only segment in memory

Store it on the stack at offset 8 from `%rsp`

Check the canary is fine using `xorl` (0 if the two values are identical)

2. Stack Canaries - Setting up canary

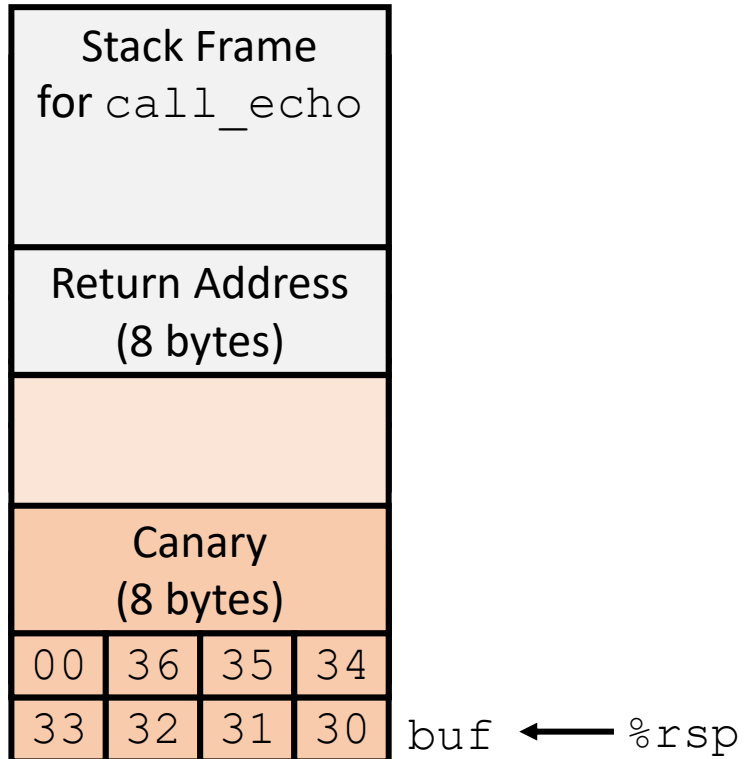*Before call to* `gets`


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

```
echo:
    . . .
    movq     %fs:40, %rax   # Get canary
    movq     %rax, 8(%rsp)  # Place on stack
    xorl     %eax, %eax     # Erase canary
    . . .
```

2. Stack Canaries - Setting up canary

After call to `gets`

Stack Frame for `call_echo`
Return Address (8 bytes)
Canary (8 bytes)

| 00	36	35	34
33	32	31	30

`buf` ⟵ `%rsp`

Input: *0123456*

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

```
echo:
    . . .
    movq      8(%rsp), %rax     # Retrieve from stack
    xorq      %fs:40, %rax      # Compare to canary
    je        .L6               # If same, OK
    call      __stack_chk_fail  # FAIL
.L6:          . . .
```

3. Address space layout randomization (ASLR)

- Like stack randomization, generalized to all of memory
  - ***Especially***: executable code

- Code, stack, heap all start in random locations
  - Determined when program starts up
  - You know the gadget you want is at the end of `ab_plus_c`
  - But if you don't know where `ab_plus_c` *is*, that's no use!

- Can be circumvented by clever side-channel attacks
  - But really hard! Much harder than ROP
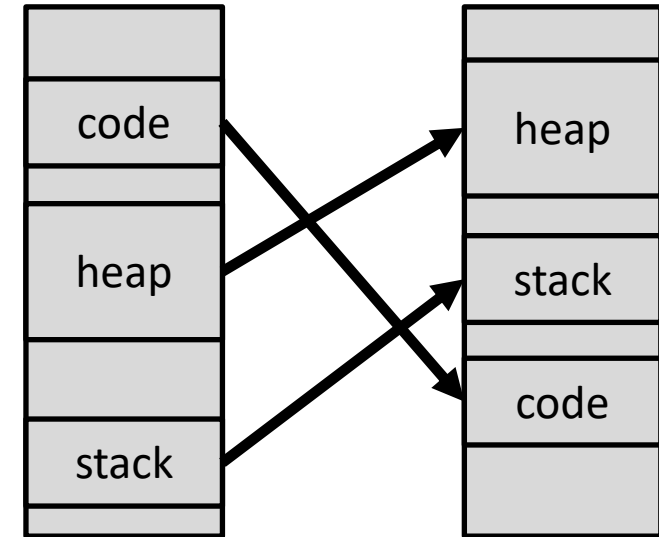
```
???? <ab_plus_c>:
    ????: 48 0f af fe
    ????: 48 8d 04 17
    ????: c3
```

Security is an arms race

- There is no single fix for system security
    - New attacks are constantly being discovered
    - New solutions are constantly being applied

1. Find a vulnerability and how it can be exploited
2. Fix vulnerability
3. Go back to 1

- A good goal is to at least avoid all the simple known attacks
- Designing with security in mind can make vulnerabilities harder to find in the first place

Outline

- Buffer Overflows

- Protecting Against Buffer Overflows

- Return-Oriented Programming

- Protecting Against Return-Oriented Programming