
WELCOME
TO
15-213
18-213
15-513

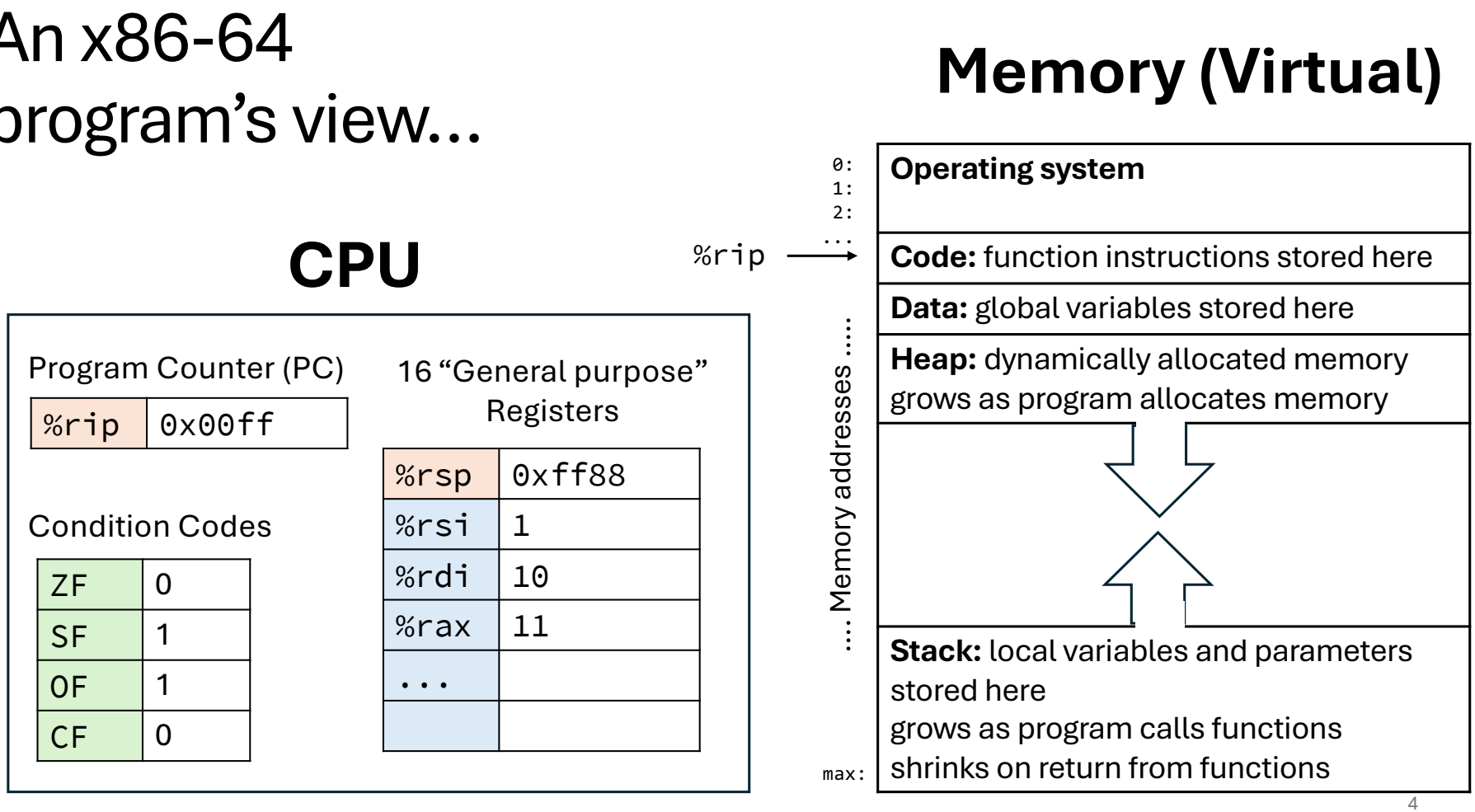
Announcements

- bomblab out, due Feb 6
- writing assignment 2 out, due Feb 5
- attacklab out Feb 6 , due Fe13

Today: How can we exploit x86-64 code? (So we can be better C programmers!)

- **Memory Layout**
- Buffer overflow
  - Vulnerability
  - Protection
  - Bypassing Protection
- Unions (won't get to today)

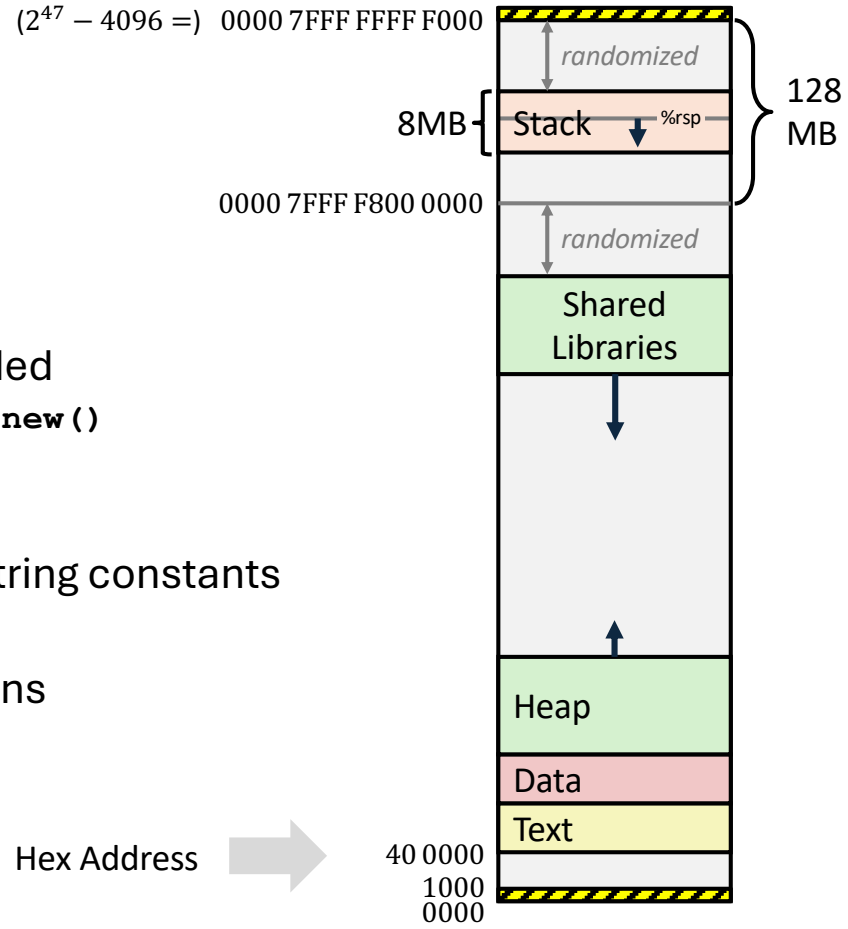
An x86-64 program's view...



x86-64 Linux Memory Layout

not drawn to scale

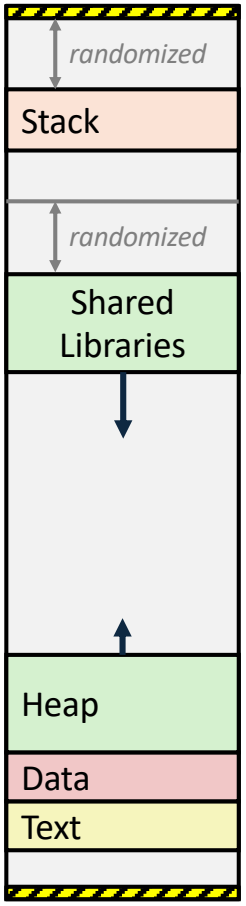
- Stack
  - Runtime stack (8MB limit)
  - e.g., local variables
- Heap
  - Dynamically allocated as needed
  - When call `malloc(), calloc(), new()`
- Data
  - Statically allocated data
  - e.g., global vars, `static` vars, string constants
- Text / Shared Libraries
  - Executable machine instructions
  - Read-only



Memory Allocation Example

not drawn to scale

```
char big_array[1L<<24];  /* 16 MB */
char huge_array[1L<<31]; /*  2 GB */

int global = 0;

int useless() { return 0; }

int main ()
{
    void *phuge1, *psmall2, *phuge3, *psmall4;
    int local = 0;
    phuge1 = malloc(1L << 28);  /* 256 MB */
    psmall2 = malloc(1L << 8);  /* 256  B */
    phuge3 = malloc(1L << 32);  /*   4 GB */
    psmall4 = malloc(1L << 8);  /* 256  B */
 /* Some print statements ... */
}
```



Where does everything go?

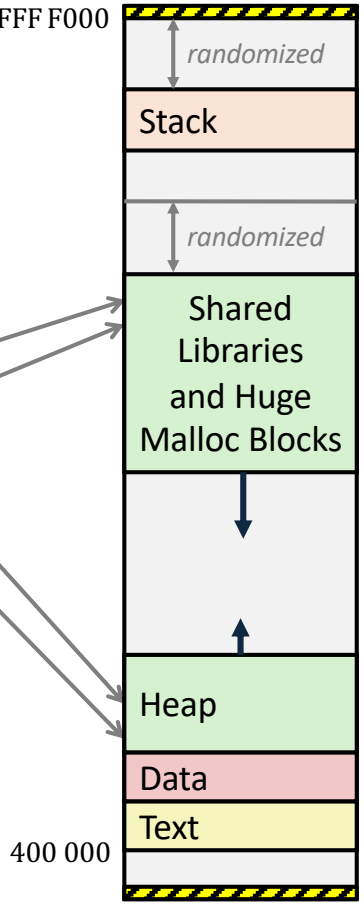
x86-64 Example Addresses

not drawn to scale

address range ~2^{47}

`local`	`0x00007ffe4d3be87c`
`phuge1`	`0x00007f7262a1e010`
`phuge3`	`0x00007f7162a1d010`
`psmall4`	`0x000000008359d120`
`psmall2`	`0x000000008359d010`
`big_array`	`0x0000000080601060`
`huge_array`	`0x0000000000601060`
`main()`	`0x000000000040060c`
`useless()`	`0x0000000000400590`

(Exact values can vary)



x86-64 is Little Endian

- Disassembly
  - Text representation of binary machine code
  - Generated by program that reads the machine code
- To read fragments, read them right to left

Address	Instruction Code	Assembly Rendition
8048365	5b	pop %ebx
8048366	81 c3 ab 12 00 00	add $0x12ab,%ebx
804836c	83 bb 28 00 00 00 00	cmpl $0x0,0x28(%ebx)

- Deciphering Numbers
  - Value: 0x12ab
  - Pad to 32 bits: 0x000012ab
  - Split into bytes: 00 00 12 ab
  - Reverse bytes: ab 12 00 00

The key to this lecture: Remember how `ret` instruction interacts with the stack and instruction pointer!

```
retq
```

is equivalent to

```
popq %rip
```

is equivalent to

```
movq  (%rsp), %rip
addq  $8, %rsp
```

Updates the instruction pointer (program counter) `%rip` to the return address.

This instruction assumes by the time a function gets to the `ret` instruction the return address is on top of the stack!

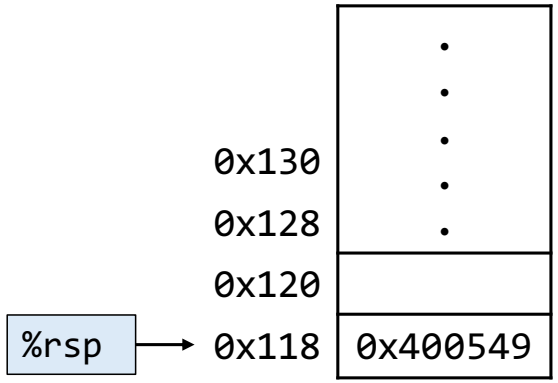
Recall …
Before `ret`

```
1  0000000000400540 <multstore>:
2    400540: push   %rbx              # Save %rbx
3    400541: mov    %rdx,%rbx         # Save dest
4    400544: call   400550 <mult2>    # mult2(x,y)
5    400549: mov    %rax,(%rbx)       # Save at dest
6    40054c: pop    %rbx              # Restore %rbx
7    40054d: ret                      # Return
```

```
1  0000000000400550 <mult2>:
2    400550:  mov    %rdi,%rax        # a
3    400553:  imul   %rsi,%rax        # a * b
4    400557:  ret                     # Return
```

%rip →

Stack



Registers	
%rsp	0x118
%rip	0x400557

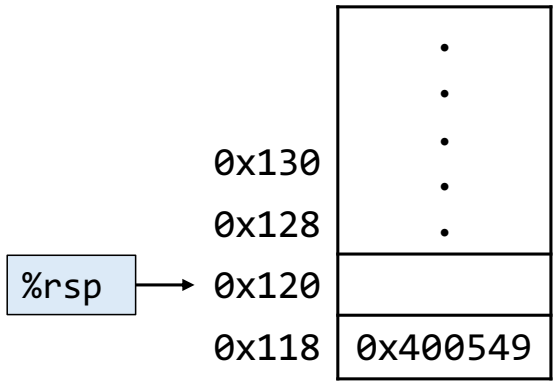
Executing `ret`: pop return address

```
1  0000000000400540 <multstore>:
2    400540: push   %rbx              # Save %rbx
3    400541: mov    %rdx,%rbx         # Save dest
4    400544: call   400550 <mult2>    # mult2(x,y)
5    400549: mov    %rax,(%rbx)       # Save at dest
6    40054c: pop    %rbx              # Restore %rbx
7    40054d: ret                      # Return
```

```
1  0000000000400550 <mult2>:
2    400550:  mov    %rdi,%rax        # a
3    400553:  imul   %rsi,%rax        # a * b
4    400557:  ret                     # Return
```

%rip →

Stack



Registers	
%rsp	0x120
%rip	0x400549

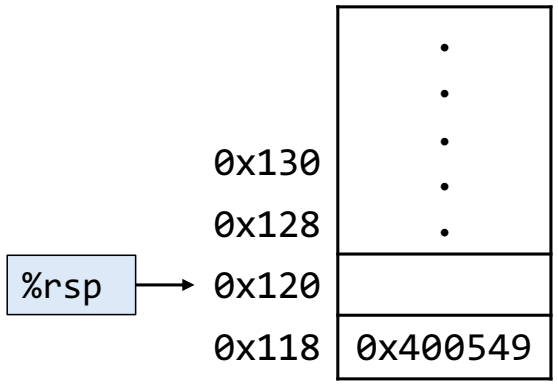
Executing `ret`: `jmp` to return address

```
1  0000000000400540 <multstore>:
2    400540: push   %rbx              # Save %rbx
3    400541: mov    %rdx,%rbx         # Save dest
4    400544: call   400550 <mult2>    # mult2(x,y)
5    400549: mov    %rax,(%rbx)       # Save at dest
6    40054c: pop    %rbx              # Restore %rbx
7    40054d: ret                      # Return
```

%rip →

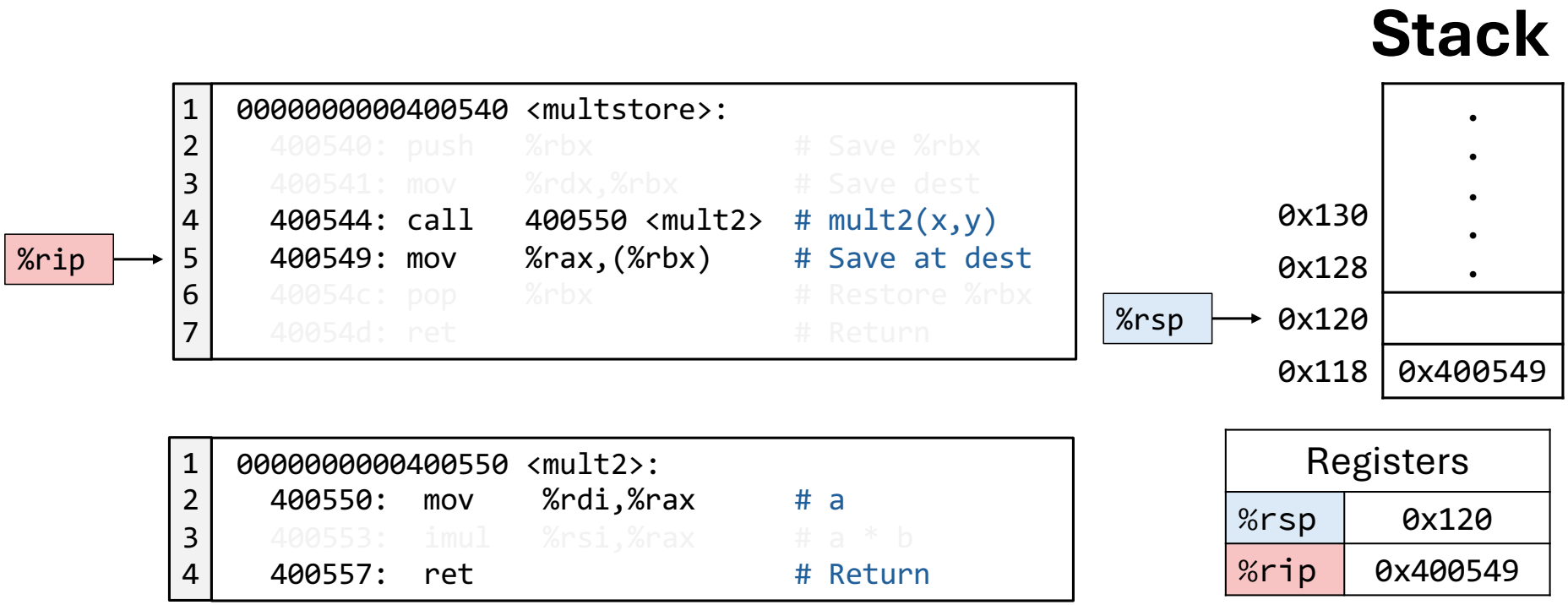
```
1  0000000000400550 <mult2>:
2    400550:  mov    %rdi,%rax        # a
3    400553:  imul   %rsi,%rax        # a * b
4    400557:  ret                     # Return
```

Stack



Registers	
%rsp	0x120
%rip	0x400549

After `ret`


%rip
1 0000000000400540 <multstore>:
2 400540: push %rbx # Save %rbx
3 400541: mov %rdx,%rbx # Save dest
4 400544: call 400550 <mult2> # mult2(x,y)
5 400549: mov %rax,(%rbx) # Save at dest
6 40054c: pop %rbx # Restore %rbx
7 40054d: ret # Return
1 0000000000400550 <mult2>:
2 400550: mov %rdi,%rax # a
3 400553: imul %rsi,%rax # a * b
4 400557: ret # Return
Stack
%rsp
0x130
0x128
0x120
0x118
0x400549
Registers
%rsp 0x120
%rip 0x400549

Recall: Memory Referencing Bug Example

```
 1  typedef struct {
 2    int a[2];
 3    double d;
 4  } struct_t;
 5
 6  double fun(int i) {
 7    volatile struct_t s;
 8    s.d = 3.14;
 9    s.a[i] = 1073741824; /* Possibly out of bounds */
10    return s.d;
11  }
```

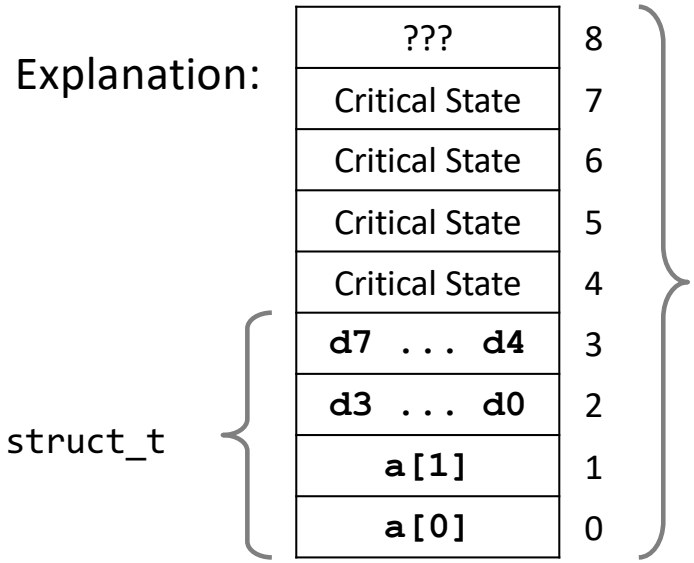
```
fun(0)  ->    3.1400000000
fun(1)  ->    3.1400000000
fun(2)  ->    3.1399998665
fun(3)  ->    2.0000006104
fun(6)  ->    Stack smashing detected
fun(8)  ->    Segmentation fault
```

Today: How can we exploit x86-64 code? (So we can be better C programmers!)

- Memory Layout
- **Buffer overflow**
  - **Vulnerability**
  - Protection
  - Bypassing Protection
- Unions

Memory Referencing Bug Example

Explanation:




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

```
fun(0)  ->     3.1400000000
fun(1)  ->     3.1400000000
fun(2)  ->     3.1399998665
fun(3)  ->     2.0000006104
fun(4)  ->     Segmentation fault
fun(8)  ->     3.1400000000
```

Buffer Overflow is a BIG Deal!

- “Buffer overflow” is generally when the memory size allocated for an array
- Buffer overflows are the #1 technical cause of security vulnerabilities
- The most common form is unchecked lengths on string inputs on bounded character arrays on the stack
  - Sometimes called “stack smashing”

The Unix function `gets()` does not check for a buffer overrun.

```
 1  /* Get string from stdin */
 2  char *gets(char *dest)
 3  {
 4      int c = getchar();
 5      char *p = dest;
 6      while (c != EOF && c != '\n') {
 7          *p++ = c;
 8          c = getchar();
 9      }
10      *p = '\0';
11      return dest;
12  }
```

There is nowhere to specify the number of characters to read.

Other problematic library functions:

- `strcpy`, `strcat`
- `scanf`, `fscanf`, `sscanf` when given %s conversion specification

Code vulnerable to a buffer overflow do not have large enough buffers for input.

```
 1  /* Echo Line */
 2  void echo()
 3  {
 4      char buf[4];  /* Way too small! */
 5      gets(buf);
 6      puts(buf);
 7  }
 8
 9  void call_echo() {
10      echo();
11  }
```

How big is big enough?

Buffer Overflow Example: Disassembly

```
1  000000000040069c <echo>:
2    40069c:  48 83 ec 18           sub    $0x18,%rsp
3    4006a0:  48 89 e7              mov    %rsp,%rdi
4    4006a3:  e8 a5 ff ff ff        call   40064d <gets>
5    4006a8:  48 89 e7              mov    %rsp,%rdi
6    4006ab:  e8 50 fe ff ff        call   400500 <puts@plt>
7    4006b0:  48 83 c4 18           add    $0x18,%rsp
8    4006b4:  c3                    retq
```

```
1  00000000004006b5 <call_echo>:
2    4006b5:  48 83 ec 08           sub    $0x8,%rsp
3    4006b9:  b8 00 00 00 00        mov    $0x0,%eax
4    4006be:  e8 d9 ff ff ff        call   40069c <echo>
5    4006c3:  48 83 c4 08           add    $0x8,%rsp
6    4006c7:  c3                    ret
```

On what lines does the program allocate space for buf?

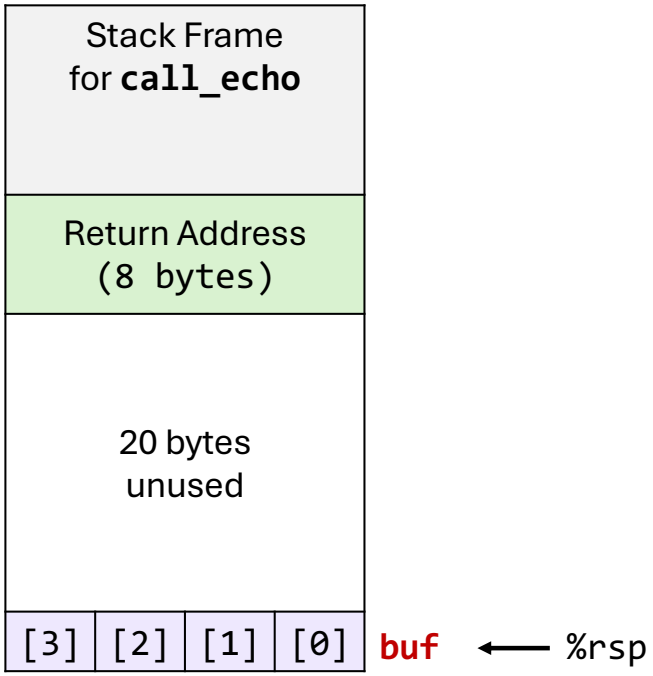
Buffer Overflow Example: Stack

%rip

```
1  40069c <echo>:
2    40069c:  sub    $0x18,%rsp
3    4006a0:  mov    %rsp,%rdi
4    4006a3:  call   40064d <gets>
5    4006a8:  mov    %rsp,%rdi
6    4006ab:  call   400500 <puts@plt>
7    4006b0:  add    $0x18,%rsp
8    4006b4:  ret
```

```
1  /* Echo Line */
2  void echo()
3  {
4      char buf[4];  /* Way too small! */
5      gets(buf);
6      puts(buf);
7  }
```

Before calling `gets`



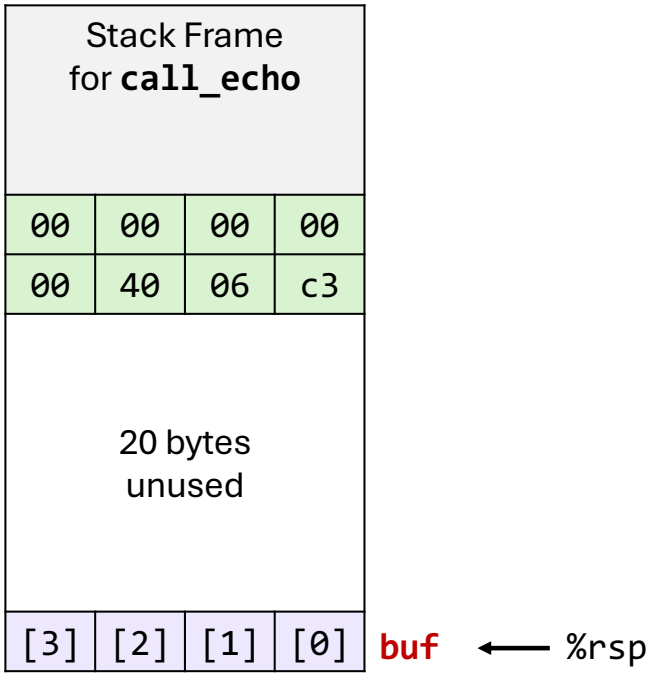
Buffer Overflow Example: Stack

%rip

```
1  40069c <echo>:
2    40069c:  sub    $0x18,%rsp
3    4006a0:  mov    %rsp,%rdi
4    4006a3:  call   40064d <gets>
5
```

```
1  4006b5 <call_echo>:
2
3    4006be:  call   40069c <echo>
4    4006c3:  add    $0x8,%rsp
5
```

Before calling `gets`



Buffer Overflow Example #1: Stack

```
1  40069c <echo>:
2    40069c:  sub    $0x18,%rsp
3    4006a0:  mov    %rsp,%rdi
4    4006a3:  call   40064d <gets>
5
```

```
1  4006b5 <call_echo>:
2
3    4006be:  call   40069c <echo>
4    4006c3:  add    $0x8,%rsp
5
```

%rip ➡

```
unix>./bufdemo-nsp
Type a string:0123456789012345678901 2
0123456789012345678901 2
```

"0123456789012345678901 2\0"

After calling gets

Stack Frame for **call_echo**			
00	00	00	00
00	40	06	c3
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

Overflowed buffer, but did not corrupt state.

%rip ➡

Buffer Overflow Example #2: Stack

```
1  40069c <echo>:
2    40069c:  sub    $0x18,%rsp
3    4006a0:  mov    %rsp,%rdi
4    4006a3:  call   40064d <gets>
5
```

```
1  4006b5 <call_echo>:
2
3    4006be:  call   40069c <echo>
4    4006c3:  add    $0x8,%rsp
5
```

```
unix>./bufdemo-nsp
Type a string:0123456789012345678901 23
0123456789012345678901 23
Segmentation fault
```

"0123456789012345678901 23\0"

Before calling 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

Program "returned" to 0x0400600, and then crashed.

Crafting Stack Smashing Attacks

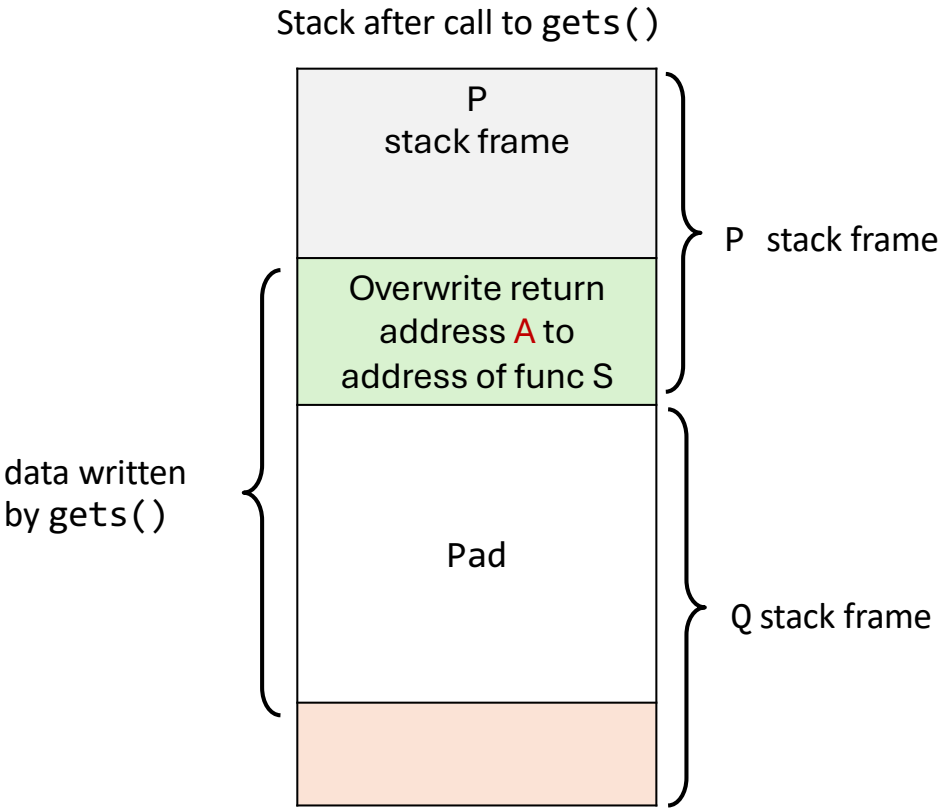
- Overwrite normal return address with the address of some other code S
- When Q executes ret, will jump to other code

```
void P(){
  Q();
  ...
}
```

return address A

```
int Q() {
  char buf[64];
  gets(buf);
  ...
  return ...;
}
```

```
void S(){
/* Something
   unexpected */
  ...
}
```



Crafting Smashing String Example

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

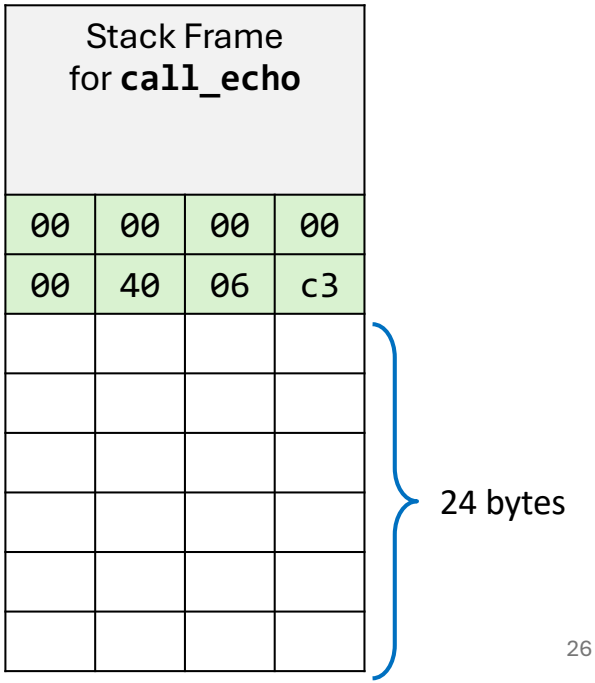
Attack String (Hex)

```
30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31
32 33 c8 06 40 00 00 00 00 00
```

Target Code

```
1  void smash() {
2    printf("I've been smashed!\n");
3    exit(0);
4  }
```

```
1  00000000004006c8 <smash>:
2    4006c8:       48 83 ec 08
```



Crafting Smashing String Example Effect

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

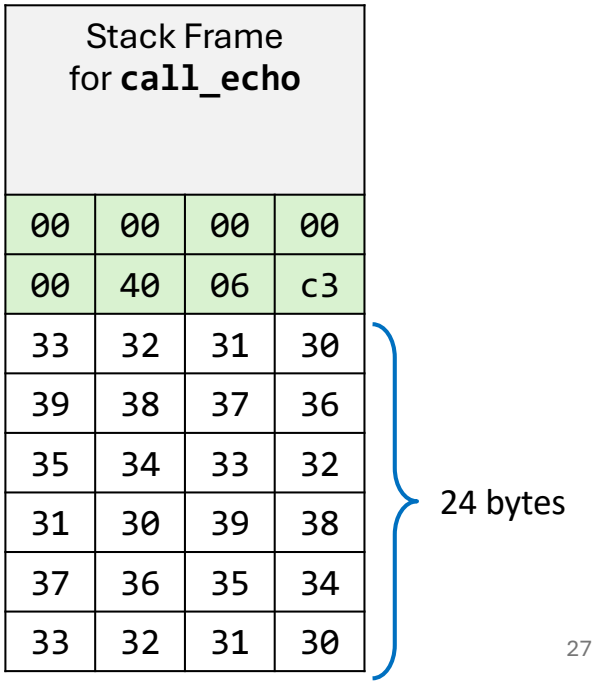
Attack String (Hex)

```
30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31
32 33 c8 06 40 00 00 00 00 00
```

Target Code

```
1  void smash() {
2    printf("I've been smashed!\n");
3    exit(0);
4  }
```

```
1  00000000004006c8 <smash>:
2    4006c8:       48 83 ec 08
```



Performing Stack Smashing

```
linux> cat smash-hex.txt
30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 c8 06 40
00 00 00 00 00
linux> cat smash-hex.txt | ./hexify | ./bufdemo-nsp
Type a string:012345678901234567890123?@
I've been smashed!
```

Target Code

```
1  void smash() {
2    printf("I've been smashed!\n");
3    exit(0);
4  }
```

- Put hex sequence in file smash-hex.txt
- Use hexify program to convert hex digits to characters
- Provide as input to vulnerable program

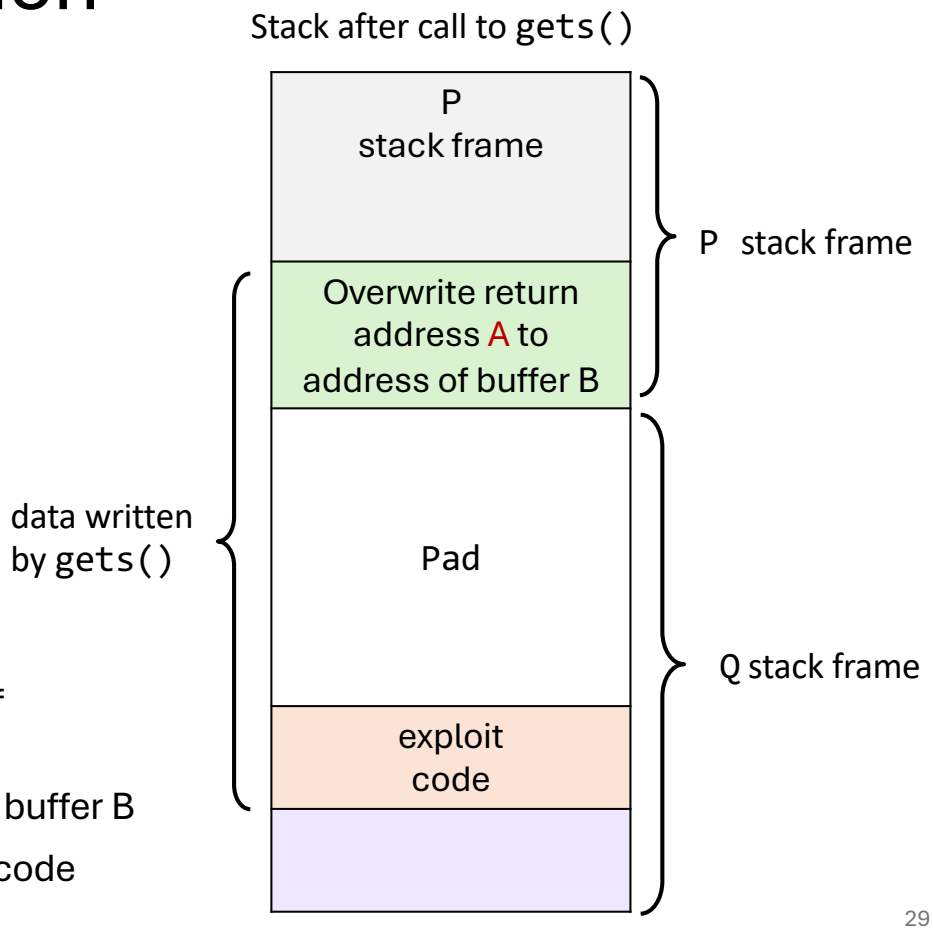
Crafting Code Injection Attacks

```
void P(){
  Q();
  ...
}
```

return address A

```
int Q() {
  char buf[64];
  gets(buf);
  ...
  return ...;
}
```

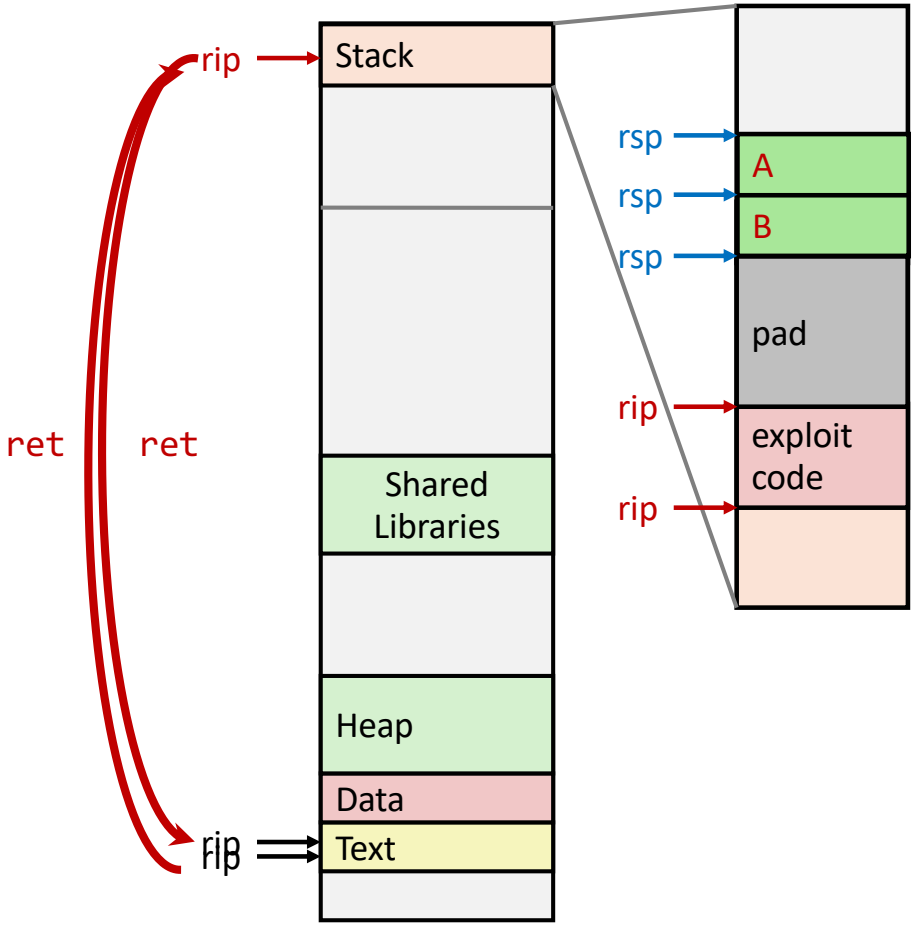
- Input string contains byte representation of executable code
- Overwrite return address A with address of buffer B
- When Q executes `ret`, will jump to exploit code



How Does The Attack Code Execute?

```
void P(){
  Q();
  ...
}
```

```
int Q() {
  char buf[64];
  gets(buf);  // A-> B
  ...
  return ...;
}
```



Today: How can we exploit x86-64 code? (So we can be better C programmers!)

- Memory Layout
- **Buffer overflow**
  - Vulnerability
  - **Protection**
  - Bypassing Protection
- Unions

What to do about buffer overflow attacks?

1. Avoid overflow vulnerabilities when writing code
2. Employ system-level protections
3. Have compilers use “stack canaries”

1. Avoid overflow vulnerabilities when writing code (!)

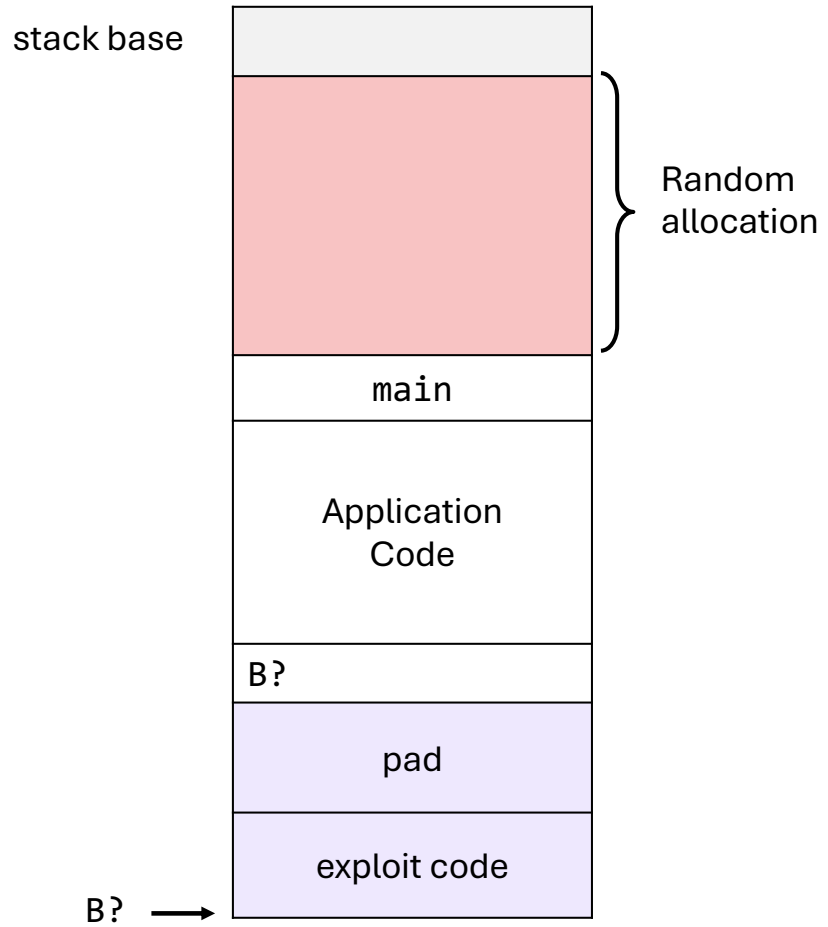
```
1  /* Echo Line */
2  void echo()
3  {
4      char buf[4];
5      fgets(buf, 4, stdin);
6      puts(buf);
7  }
```

Don’t use these...	Use these instead...
gets(buf)	fgets(buf, 4, stdin)
scanf("%s", buf)	scanf("%4s", buf)
strcpy(buf2, buf)	strncpy(buf2, buf, 4)
strcat(buf2, buf)	strncat(buf2, buf, 4)
sprintf(buf, "%d", num)	snprintf(buf, 4, "%d", num)

2. Employ system-level protections

Randomized stack offsets

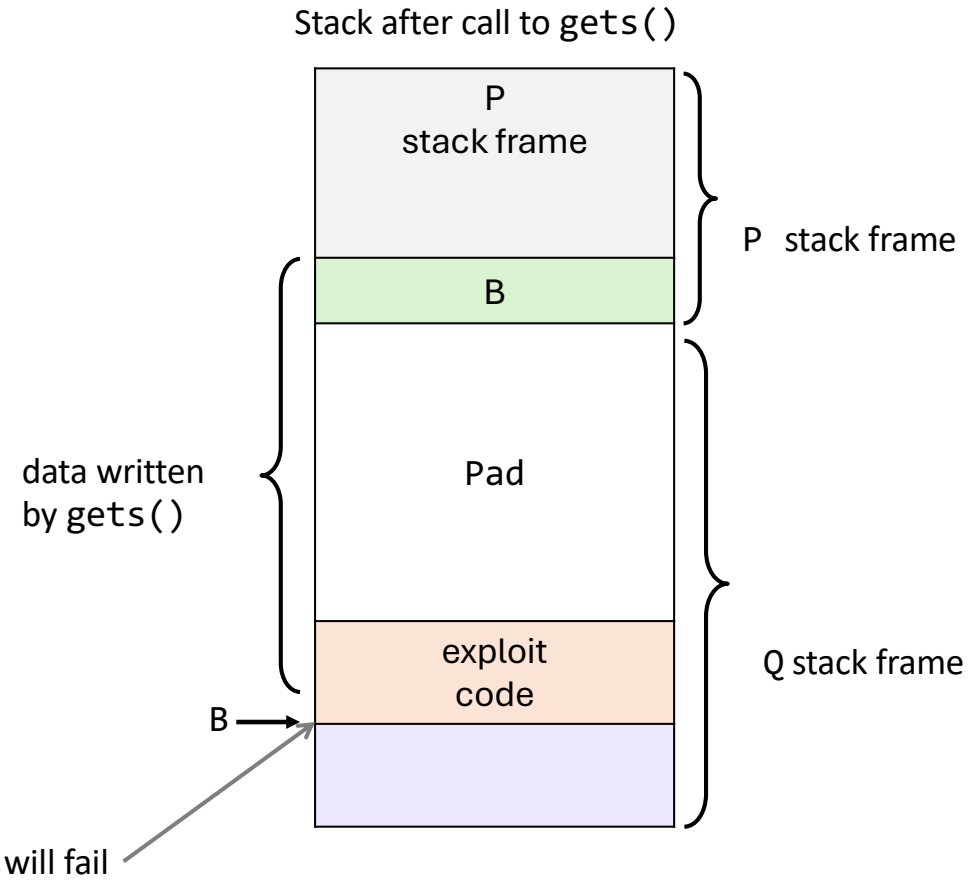
- At start of program, allocate random amount of space on stack
- Shifts stack addresses for entire program
- Makes it difficult for hacker to predict beginning of inserted code
- Stack repositioned each time program executes



2. Employ system-level protections

Non-executable memory

- Older x86 CPUs would execute machine code from any readable address
- x86-64 added a way to mark regions of memory as *not executable*
- Immediate crash on jumping into any such region
- Current Linux and Windows mark the stack this way



3. Have compilers use “stack canaries”

Place a special value (“canary”) on stack just beyond buffer. Check for corruption before exiting function.

GCC Implementation (default)

`-fstack-protector`

Example when not disabled:

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

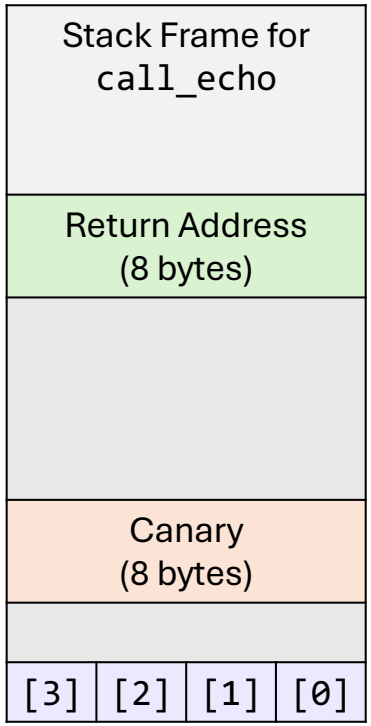
```
unix>./bufdemo-sp
Type a string:012345678
*** stack smashing detected ***
```

Protected Buffer Disassembly

```
 1  40072f <echo>:
 2    40072f:  sub    $0x18,%rsp
 3    400733:  mov    %fs:0x28,%rax
 4    40073c:  mov    %rax,0x8(%rsp)
 5    400741:  xor    %eax,%eax
 6    400743:  mov    %rsp,%rdi
 7    400746:  callq  4006e0 <gets>
 8    40074b:  mov    %rsp,%rdi
 9    40074e:  callq  400570 <puts@plt>
10    400753:  mov    0x8(%rsp),%rax
11    400758:  xor    %fs:0x28,%rax
12    400761:  je     400768 <echo+0x39>
13    400763:  callq  400580 <__stack_chk_fail@plt>
14    400768:  add    $0x18,%rsp
15    40076c:  retq
```

Setting Up Canary

Before call to `gets()`

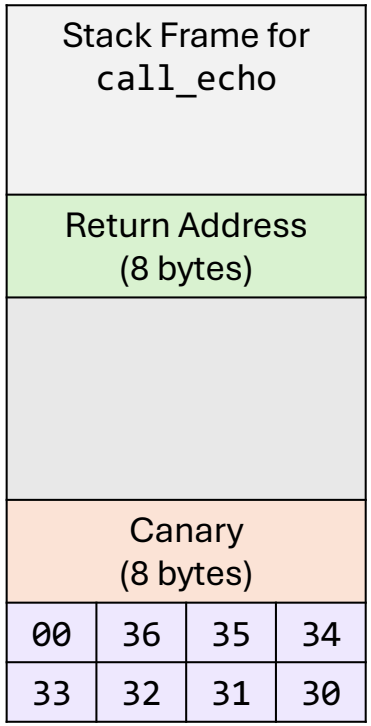



```
1  /* Echo Line */
2  void echo()
3  {
4      char buf[4];  /* Way too small! */
5      gets(buf);
6      puts(buf);
7  }
```

```
1  echo:
2      ...
3      mov    %fs:0x28,%rax   # Get canary
4      mov    %rax,0x8(%rsp)  # Place on stack
5      xor    %eax,%eax       # Erase register
6      ...
```

Checking Canary

After call to `gets()`




```
1  /* Echo Line */
2  void echo()
3  {
4      char buf[4];  /* Way too small! */
5      gets(buf);
6      puts(buf);
7  }
```

```
1  echo:
2      ...
3      mov 0x8(%rsp),%rax        # Retrieve from stack
4      xor %fs:0x28,%rax         # Compare to canary
5      je  .L6                   # If same, OK
6      call    __stack_chk_fail  # FAIL
```

Example Input: *0123456*

Today: How can we exploit x86-64 code? (So we can be better C programmers!)

- Memory Layout
- Buffer overflow
  - Vulnerability
  - Protection
  - **Bypassing Protection**
- Unions

How can we bypass buffer overflow protections?

Two challenges we can overcome:

1. Stack randomization makes it hard to predict buffer location
2. Marking stack as non-executable makes it hard to insert binary code

Solution: Use existing code!

- There is code already in the program and in linked C libraries.
- String together fragments to achieve overall desired outcome

Note: This does not overcome stack canaries.

We can use existing program code using return-oriented programming (ROP) attacks.

The core principles:

- Find existing instructions in code and string them together to execute exploits.
- Use gadgets: Sequence of instructions ending in `ret`.
- Use the stack to store the addresses of the gadgets we want to execute.

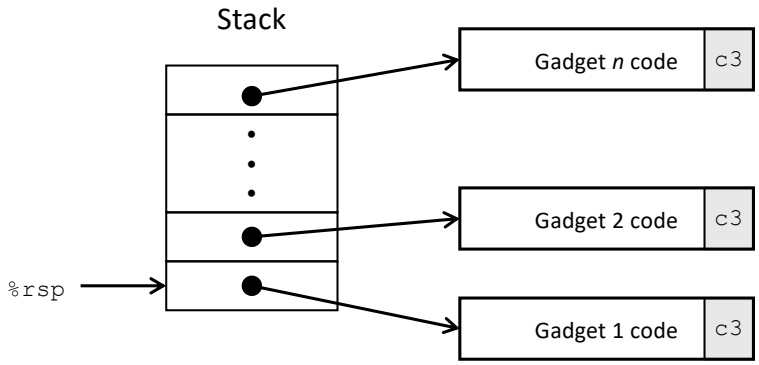
Why does ROP work?

We can use buffer overflow to insert arbitrary things on the stack.

If we put the address of code on the stack that ends in a `ret` instruction, this will update %rip to the address on top of the stack.

The next executed instruction will be what `%rip` now points to, which is our gadget!

Since the gadget also ends with `ret` it will also update the stack pointer and execute the next gadget



We can get instructions from existing functions: Gadget Example #1

```
1  /* Echo Line */
2  void echo()
3  {
4      char buf[4];
5      gets(buf);
6      ...
7  }
```

```
1  0000000004004d0 <ab_plus_c>:
2    4004d0:  48 0f af fe  imul %rsi,%rdi
3    4004d4:  48 8d 04 17  lea (%rdi,%rdx,1),%rax
4    4004d8:  c3           retq
```

Gadget address = 0x4004d4

Encodes:

rax ← rdi + rdx

ret

Somewhere else in code:

```
1  long ab_plus_c
2    (long a, long b, long c) {
3     return a*b + c;
4  }
```

Example: Use tail-end of existing instructions

We can get instructions from existing functions: Gadget Example #2

```
1  /* Echo Line */
2  void echo()
3  {
4      char buf[4];
5      gets(buf);
6      ...
7  }
```

```
1  <setval>:
2    4004d9:  c7 07 d4 48 89 c7  movl  $0xc78948d4,(%rdi)
3    4004df:  c3
```

Gadget address = 0x4004d

Encodes: movq %rax, %rdi

rdi ← rax

ret

Somewhere else in code:

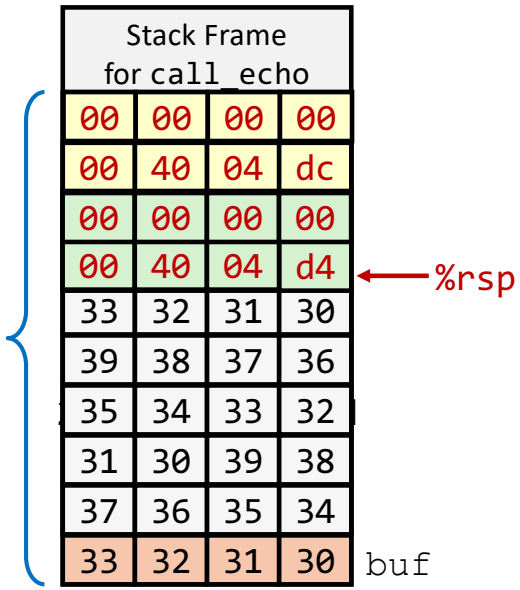
```
1  void setval(unsigned *p) {
2      *p = 3347663060u;
3  }
```

Example: Re-purpose bytes

Often "parts" of instructions are also valid instructions!

Look for `c3` (`ret`) and work backwards.

Crafting a ROP Attack String



- Gadget #1
  - **0x4004d4** rax ← rdi + rdx
- Gadget #2
  - **0x4004dc** rdi ← rax
- Combination

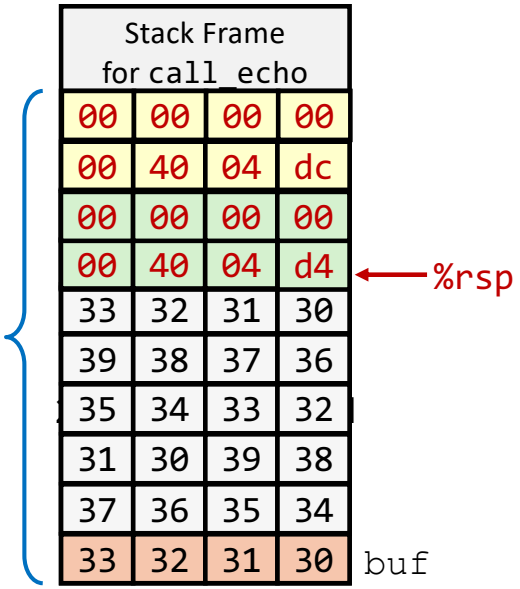
  rdi ← rdi + rdx

Attack String (Hex)

```
30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33
d4 04 40 00 00 00 00 00 dc 04 40 00 00 00 00 00
```

Multiple gadgets will corrupt stack upwards

What Happens When echo Returns?




```
1  /* Echo Line */
2  void echo()
3  {
4      char buf[4];
5      gets(buf);
6      ...
7  }
```

1. Echo executes `ret`
   - **Starts Gadget #1**
2. Gadget #1 executes `ret`
   - **Starts Gadget #2**
3. Gadget #2 executes `ret`
   - **Goes off somewhere …**

Multiple gadgets will corrupt stack upwards

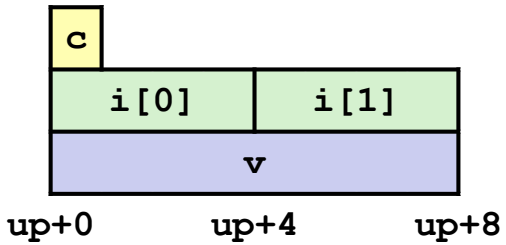
Today: How can we exploit x86-64 code? (So we can be better C programmers!)

- Memory Layout
- Buffer overflow
  - Vulnerability
  - Protection
  - Bypassing Protection
- **Unions**

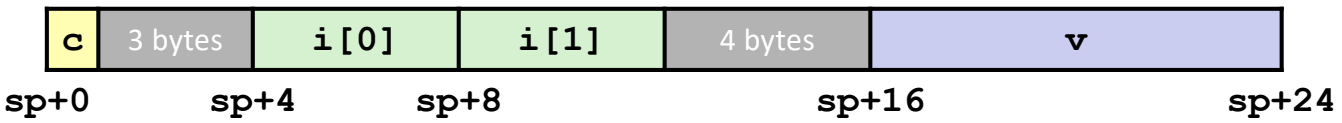
Union Allocation

- Allocate according to largest element
- Can only use one field at a time

```
union U1 {
  char c;
  int i[2];
  double v;
} *up;
```

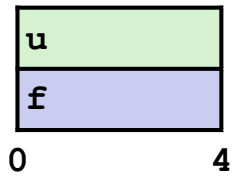



```
struct S1 {
  char c;
  int i[2];
  double v;
} *sp;
```



Using Union to Access Bit Patterns

```
typedef union {
  float f;
  unsigned u;
} bit_float_t;
```




```
float bit2float(unsigned u)
{
  bit_float_t arg;
  arg.u = u;
  return arg.f;
}
```

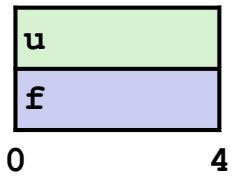
Same as **`(float) u`** ?

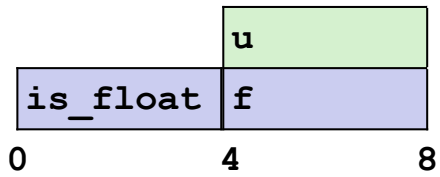
```
unsigned float2bit(float f)
{
  bit_float_t arg;
  arg.f = f;
  return arg.u;
}
```

Same as **`(unsigned) f`** ?

Using Unions as Sum Types

```
typedef union {
  float f;
  unsigned u;
} num_t;

typedef struct {
  bool is_float;
  num_t val;
} value_t;
```





(technically is_float only takes 1 byte and then there's 3 bytes of padding)

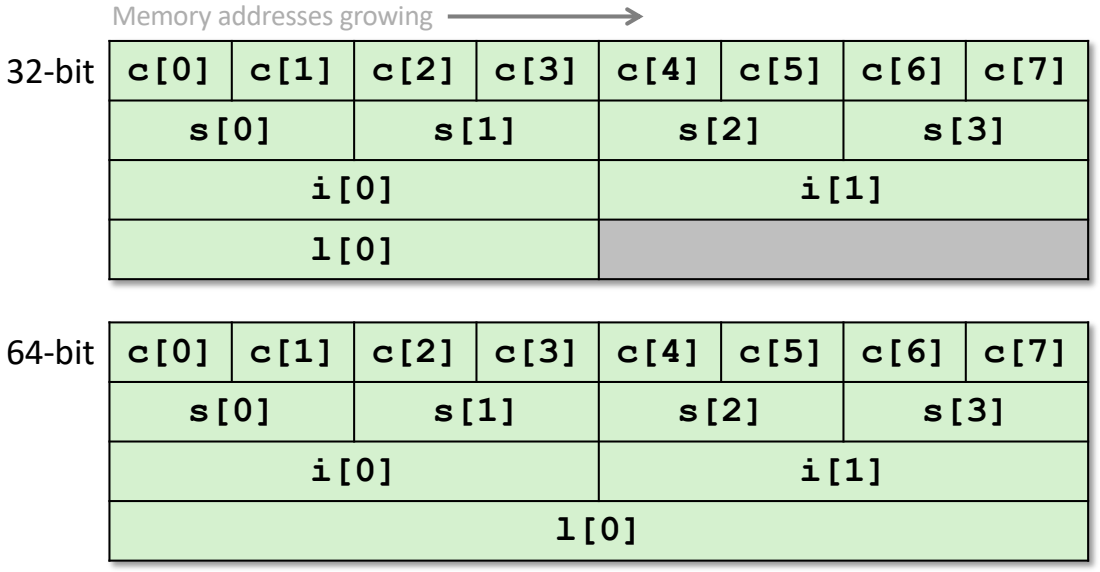
Byte Ordering Revisited

- Idea
  - Short/long/quad words stored in memory as 2/4/8 consecutive bytes
  - Which byte is most (least) significant?
  - Can cause problems when exchanging binary data between machines
- Big Endian
  - Most significant byte has lowest address
  - Sparc, *Internet*
- Little Endian
  - Least significant byte has lowest address
  - Intel x86, ARM Android and IOS
- Bi Endian
  - Can be configured either way
  - ARM

Byte Ordering Example

```
union {
  unsigned char c[8];
  unsigned short s[4];
  unsigned int i[2];
  unsigned long l[1];
} dw;
```

How are the bytes inside short/int/long stored?

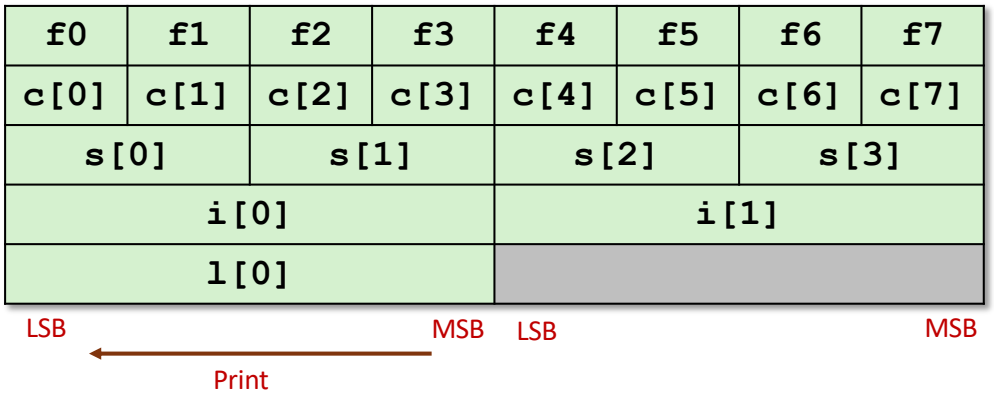


Byte Ordering Example (Cont).

```
int j;
for (j = 0; j < 8; j++)
    dw.c[j] = 0xf0 + j;

printf("Characters 0-7 ==
[0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x]\n",
    dw.c[0], dw.c[1], dw.c[2], dw.c[3],
    dw.c[4], dw.c[5], dw.c[6], dw.c[7]);

printf("Shorts 0-3 == [0x%x,0x%x,0x%x,0x%x]\n",
    dw.s[0], dw.s[1], dw.s[2], dw.s[3]);

printf("Ints 0-1 == [0x%x,0x%x]\n",
    dw.i[0], dw.i[1]);

printf("Long 0 == [0x%lx]\n",
    dw.l[0]);
```

Byte Ordering on IA32

Little Endian

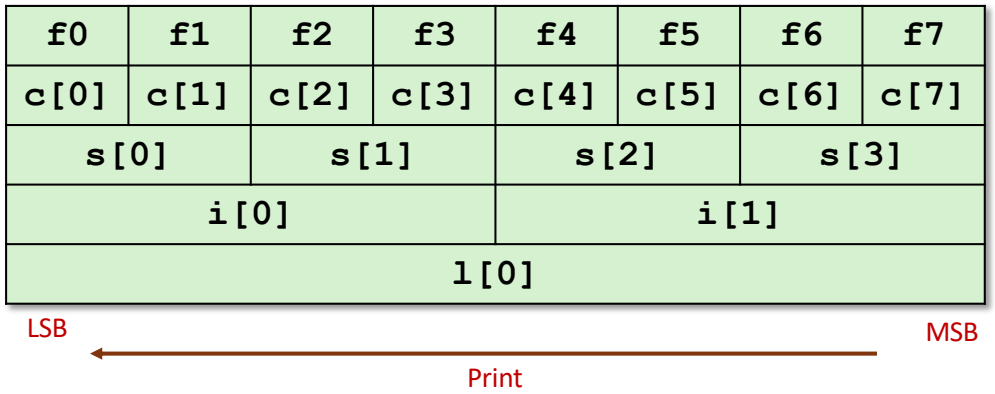


Output:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints       0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long       0   == [0xf3f2f1f0]
```

Byte Ordering on x86-64

Little Endian

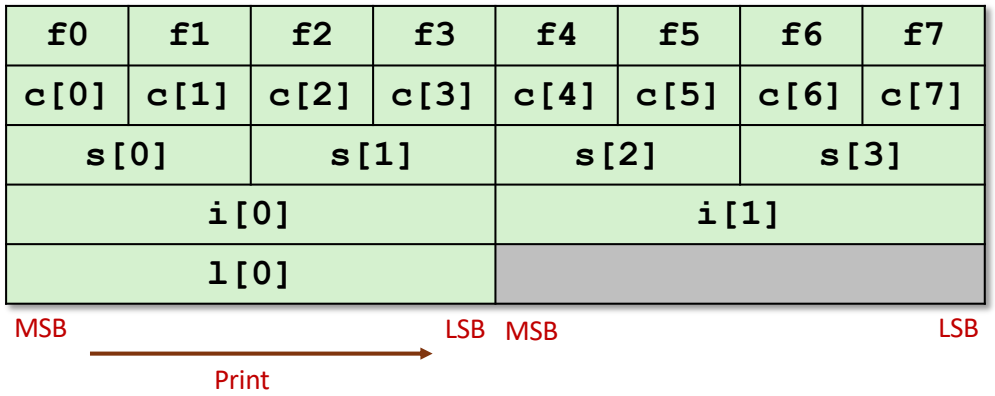


Output on x86-64:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints       0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long       0   == [0xf7f6f5f4f3f2f1f0]
```

Byte Ordering on Sun

Big Endian



Output on Sun:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf0f1,0xf2f3,0xf4f5,0xf6f7]
Ints       0-1 == [0xf0f1f2f3,0xf4f5f6f7]
Long       0   == [0xf0f1f2f3]
```