Machine-Level Programming: Buffer overflow

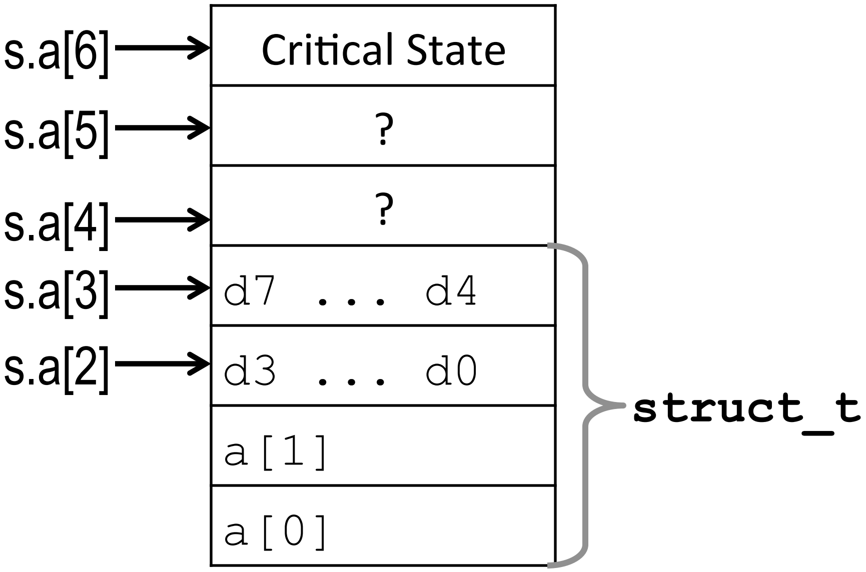
Jinyang Li

Slides adapted from Bryant and O'Hallaron

Recap: Memory Referencing Bug Example

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

double fun(int i) {
  volatile struct_t s;
  s.d = 3.14;
  s.a[i] = 1073741824;
  return s.d;
}
```

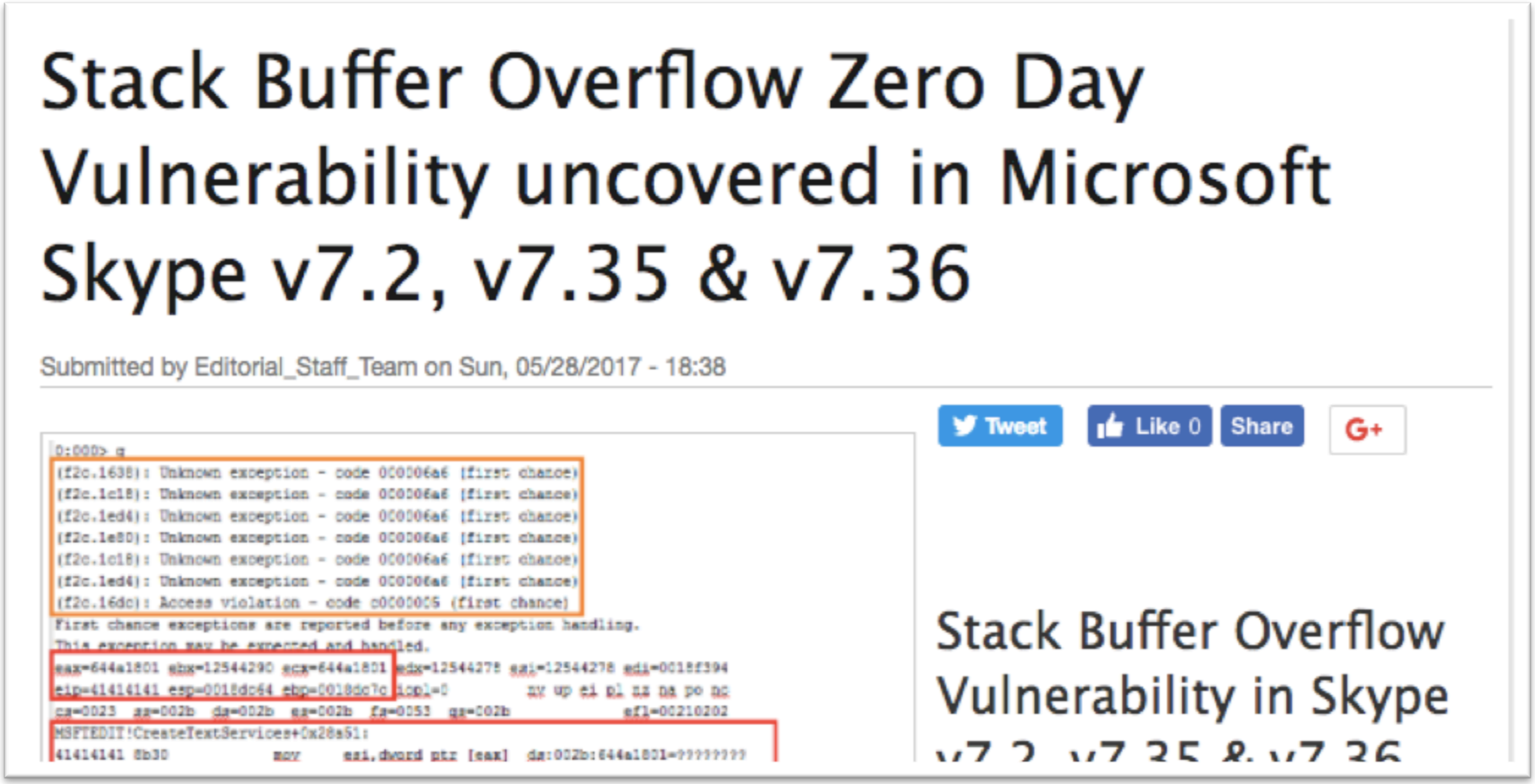


```
fun(0)  ➙    3.14
fun(1)  ➙    3.14
fun(2)  ➙    3.1399998664856
fun(3)  ➙    2.00000061035156
fun(4)  ➙    3.14
fun(6)  ➙    Segmentation fault
```

called the Buffer Overflow bug

Buffer overflows are a BIG deal

- **#1 technical cause of security vulnerabilities**
  - Many systems software written in C/C++
  - OS, file systems, database, compilers, network servers, shells,



Stack Buffer Overflow Zero Day Vulnerability uncovered in Microsoft Skype v7.2, v7.35 & v7.36

Submitted by Editorial_Staff_Team on Sun, 05/28/2017 - 18:38

```
0:000> g
(f2c.1638): Unknown exception - code 000006a6 (first chance)
(f2c.1c18): Unknown exception - code 000006a6 (first chance)
(f2c.1ed4): Unknown exception - code 000006a6 (first chance)
(f2c.1e80): Unknown exception - code 000006a6 (first chance)
(f2c.1c18): Unknown exception - code 000006a6 (first chance)
(f2c.1ed4): Unknown exception - code 000006a6 (first chance)
(f2c.16dc): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=644a1801 ebx=12544290 ecx=644a1801 edx=12544278 esi=12544278 edi=0018f394
eip=41414141 esp=0018dc64 ebp=0018dc7c iopl=0         nv up ei pl zr na po nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00210202
MSFTEDIT!CreateTextServices+0x28a51:
41414141 8b30            mov     esi,dword ptr [eax]  ds:002b:644a1801=????????
```

Stack Buffer Overflow Vulnerability in Skype v7.2, v7.35 & v7.36

Causes for buffer overflow: programming bugs

```
void foo() {
   int buffer[10];
   for (int i = 0; i <= 10; i++) {
       buffer[i] = i;
   }
   ...
}

int main() {
   foo();
}
```

Causes for buffer overflow: bad APIs

```
void copyString(char *dst, char *src) {
   while (*src != ‘\0’) {
     *dst = *src;
     src++;
     dst++;
   }
}

void bar() {
   char *s = “hello world”;
   char dst[10];
   copyString(dst, s);
}
```

C’s std library strcpy has the same bad API!

Causes for buffer overflow: Bad stdlib APIs

- **E.g. gets()**

```
// 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 way to specify limit on number of characters to read

- **Other examples: `strcpy`, `strcat`, `scanf`, `fscanf`, `sscanf`**

Vulnerable Buffer Code

```
/* Echo Line */
void echo()
{
    char buf[4];
    gets(buf);
    puts(buf);
}
void call_echo() {
    echo();
}
```

Nothing is big enough as gets() can always write more

```
unix>./a.out
Type a string:01234567890123456789012
01234567890123456789012
```

```
unix>./a.out
Type a string:0123456789012345678901234
Segmentation Fault
```

Buffer Overflow Disassembly

echo:

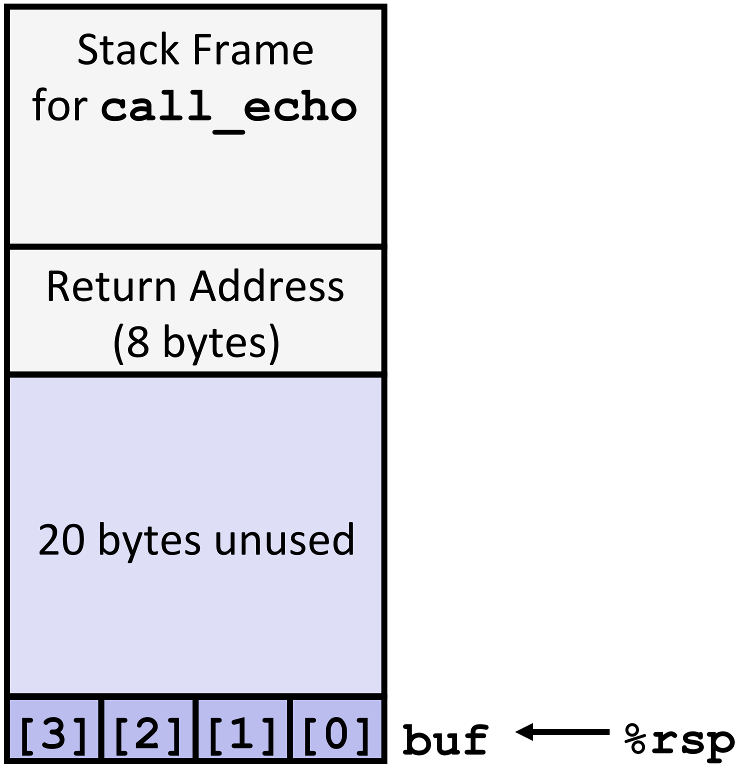
```
00000000004006cf <echo>:
 4006cf:  48 83 ec 18          sub    $0x18,%rsp
 4006d3:  48 89 e7             mov    %rsp,%rdi
 4006d6:  e8 a5 ff ff ff       callq  400680 <gets>
 4006db:  48 89 e7             mov    %rsp,%rdi
 4006de:  e8 3d fe ff ff       callq  400520 <puts@plt>
 4006e3:  48 83 c4 18          add    $0x18,%rsp
 4006e7:  c3                   retq
```

call_echo:

```
4006e8:   48 83 ec 08          sub    $0x8,%rsp
 4006ec:  b8 00 00 00 00       mov    $0x0,%eax
 4006f1:  e8 d9 ff ff ff       callq  4006cf <echo>
 4006f6:  48 83 c4 08          add    $0x8,%rsp
 4006fa:  c3                   retq
```

Buffer Overflow Stack

Before call to gets



Stack Frame for **`call_echo`**

Return Address (8 bytes)

20 bytes unused

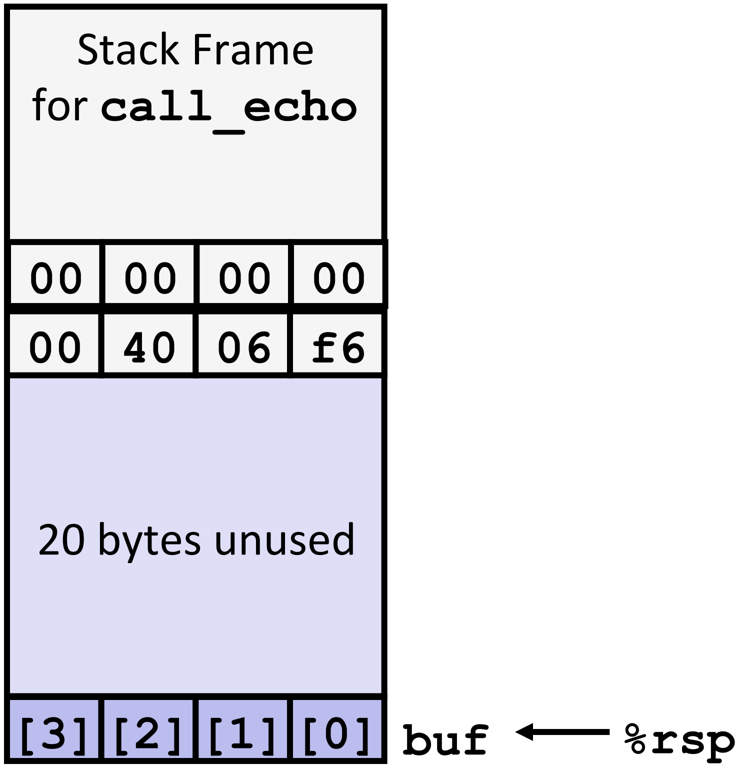
[3] [2] [1] [0] **`buf`** ← **`%rsp`**

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

```
echo:
  subq   $0x18, %rsp
  movq   %rsp, %rdi
  call   gets
  . . .
```

Buffer Overflow Stack Example

Before call to gets



Stack Frame for **`call_echo`**

00	00	00	00
00	40	06	f6

20 bytes unused

[3]	[2]	[1]	[0]

`buf` ← **`%rsp`**

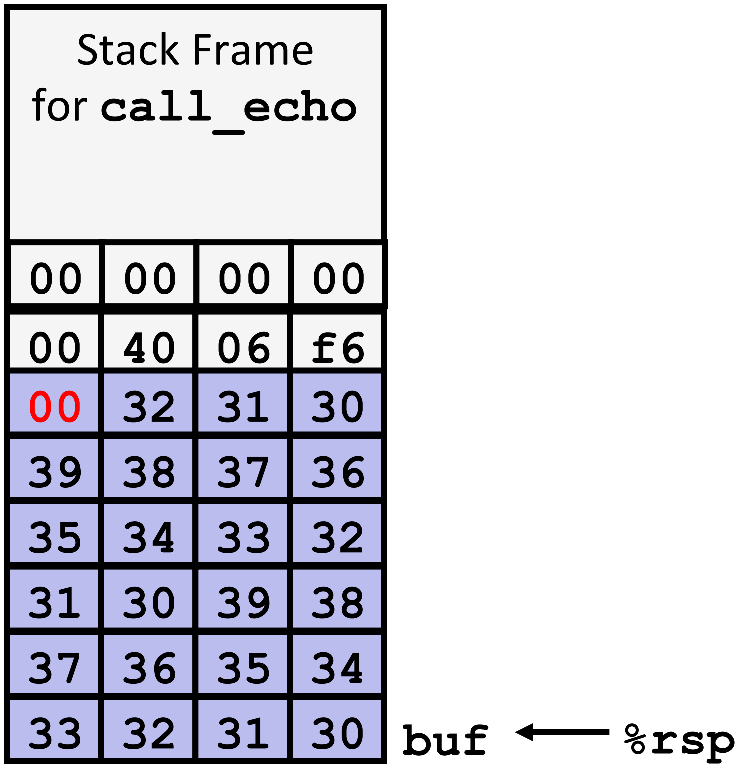
```
void echo()
{
    char buf[4];
    gets(buf);
    puts(buf);
}
```

```
echo:
  subq   $0x18, %rsp
  movq   %rsp, %rdi
  call   gets
  ...
```

```
call_echo:
  ....
  4006f1: callq  4006cf <echo>
  4006f6: add    $0x8,%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

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

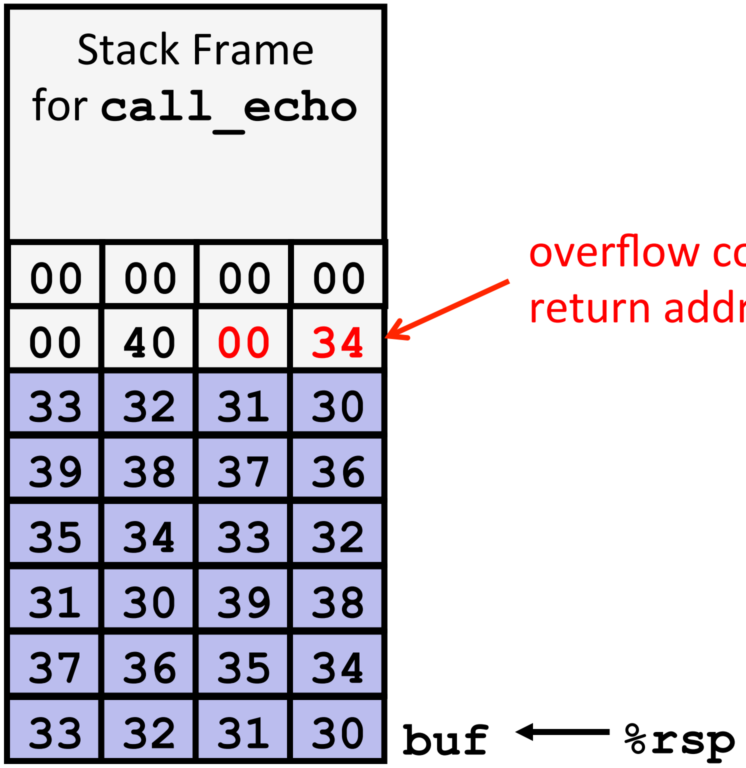
```
echo:
  subq   $0x18, %rsp
  movq   %rsp, %rdi
  call   gets
  ...
```

```
call_echo:
  ....
  4006f1: callq   4006cf <echo>
  4006f6: add     $0x8,%rsp
  ....
```

```
unix>./a.out
Type a string:01234567890123456789012
01234567890123456789012
```

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

overflow corrupted return address

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

```
echo:
  subq   $0x18, %rsp
  movq   %rsp, %rdi
  call   gets
  . . .
```

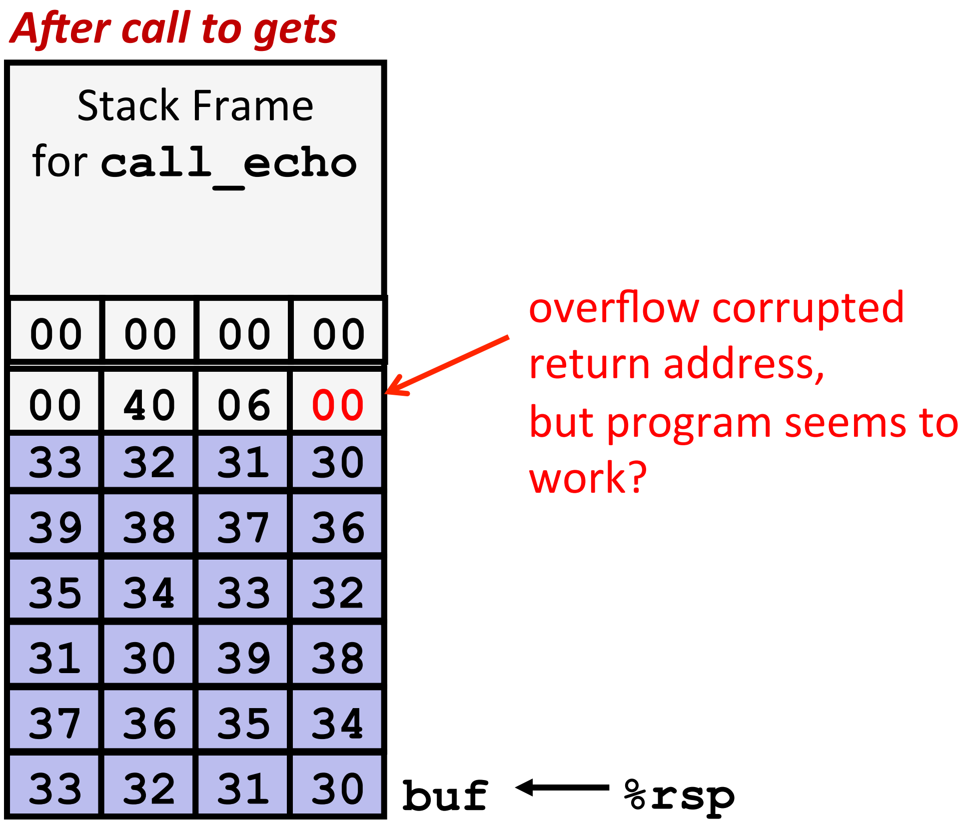
```
call_echo:
  . . . .
  4006f1: callq   4006cf <echo>
  4006f6: add     $0x8,%rsp
  . . . .
```

```
unix>./a.out
Type a string:0123456789012345678901234
Segmentation Fault
```

Q: what's the last instruction executed before seg fault?

1. ret of echo
2. ret of call_echo
3. ret of gets

Buffer Overflow Stack Example #3



overflow corrupted return address, but program seems to work?

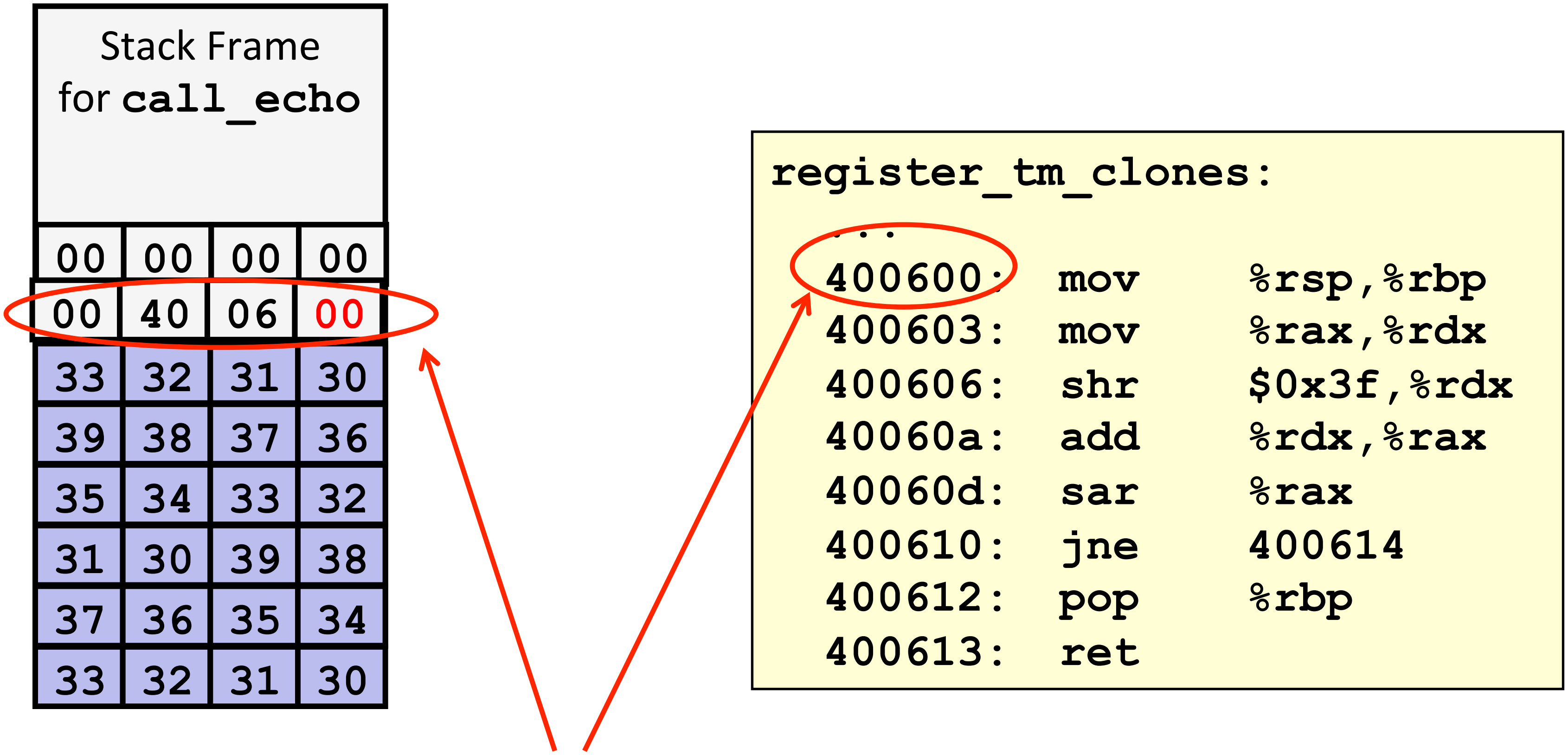
```
void echo()
{
    char buf[4];
    gets(buf);
    puts(buf);
}
```

```
echo:
  subq   $0x18, %rsp
  movq   %rsp, %rdi
  call   gets
  . . .
```

```
call_echo:
  . . . .
  4006f1: callq  4006cf <echo>
  4006f6: add    $0x8,%rsp
  . . . .
```

```
unix>./a.out
Type a string:012345678901234567890123
012345678901234567890123
```

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

```
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:  ret
```

"Returns" to unrelated code
Lots of things happen
(luckily no critical state modified)

How do attackers exploit buffer overflow?

- **First, take control over vulnerable program, called control flow hijacking**
  1. overwrite buffer with a carefully chosen return address
  2. executes malicious code (injected by attacker or elsewhere in the running program)
- **Second, gain broad access on host machine:**
  - To gain easier access, e.g. execute a shell
  - Take advantage of the permissions granted to the hacked process
    - if the process is running as “root”....
    - read user database, send spam, steal bitcoin!

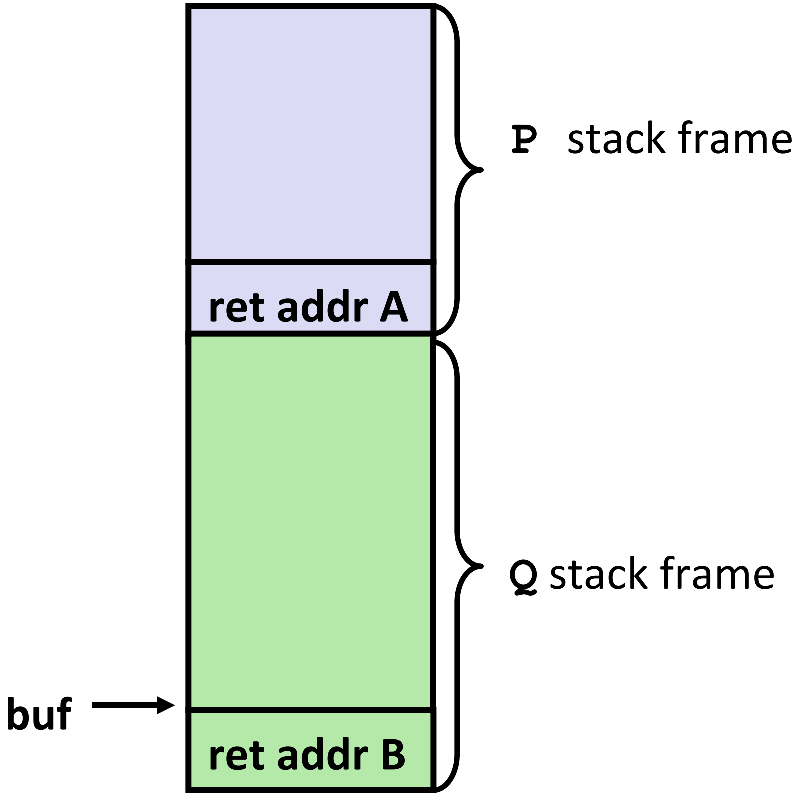
Example exploit: Code Injection Attacks

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

return address A

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

return address B



Stack upon entering `gets()`

Example exploit: Code Injection Attacks

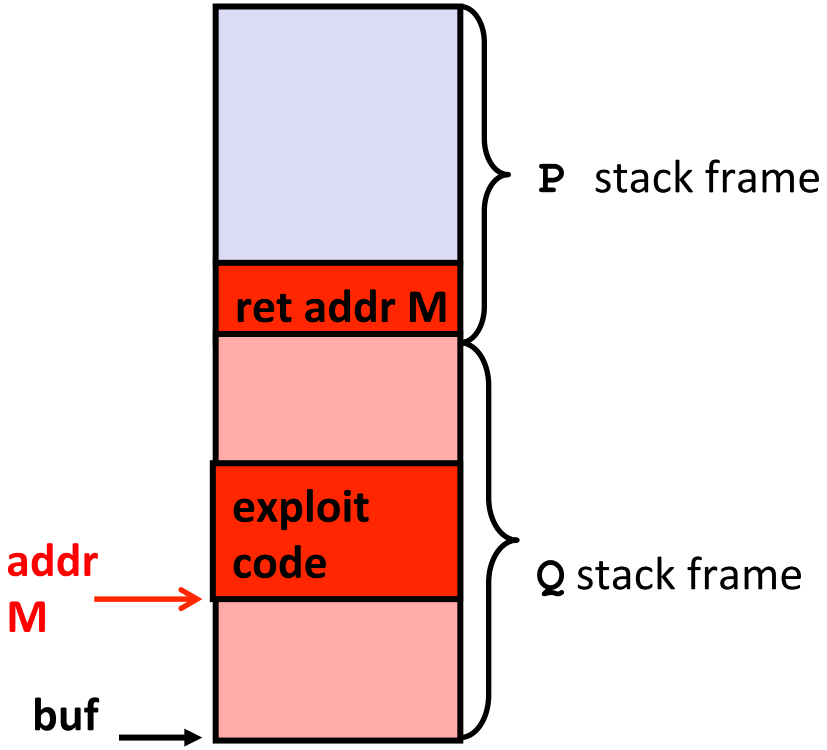
```
void P(){
   Q();
   ...
}
```

return address A

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

return address B

Upon executing this ret,
control is hijacked by exploit code



Stack after returning from `gets()`

Example Code Injection-based Buffer Overflow attacks

- **It all started with “Internet worm” (1988)**
  - A common network service (fingerd) used **`gets()`** to read inputs:
    - **`finger student123@nyu.edu`**
  - Worm attacked server by sending phony input:
    - **`finger “exploit-code...new-return-address”`**
  - Exploit-code executes a shell (with root permission) with inputs from a network connection to attacker.
  - Worm also scans other machines to launch the same attack
- **Recent measures make code-injection much more difficult**

Defenses against buffer overflow

- **Write correct code: avoid overflow vulnerabilities**
- **Mitigate attack despite buggy code**

Avoid Overflow Vulnerabilities in Code

```
void echo()
{
    char buf[4];
    fgets(buf, 4, stdin);
    puts(buf);
}
```

- **Better coding practices**
  - e.g. use library routines that limit buffer lengths, **`fgets`** instead of **`gets`**, **`strncpy`** instead of **`strcpy`**
- **Use a memory-safe language instead of C**
  - Java programs do not have buffer overflow problems, except in
    - naive methods (e.g. awt image library)
    - JVM itself
- **heuristic-based bug finding tools**
  - valgrind's SGCheck

Mitigate BO attacks despite buggy code

- **A buffer overflow attack needs two components:**
  1. Control-flow hijacking
     - overwrite a code pointer (e.g. return address) that's later invoked
  2. Call to "useful" code
     - Inject executable code in buffer
     - Re-use existing code in the running process (easy if code is in a predictable location)
- **How to mitigate attacks? make #1 or #2 hard**

Mitigate #1 (control flow hijacking)

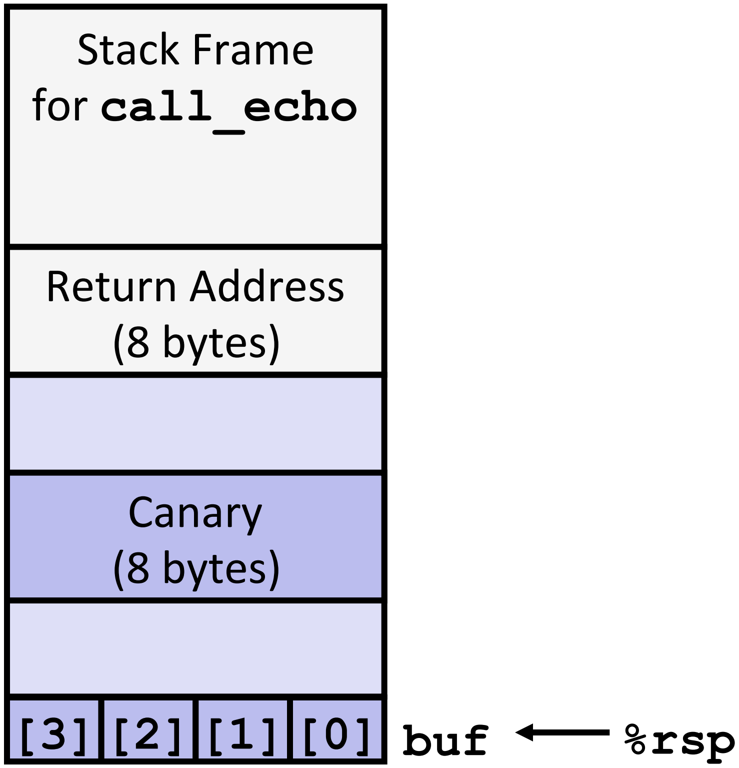
- **Idea: Catch over-written return address before invocation!**
  - Place special value (“canary”) on stack just beyond buffer
  - Check for corruption before exiting function
- **GCC Implementation**
  - `-fstack-protector`
  - Now the default

```
unix>./a.out
Type a string:0123456
0123456
```

```
unix>./a.out
Type a string:01234567
*** stack smashing detected ***
```

Setting Up Canary

Before call to gets



Stack Frame
for `call_echo`

Return Address
(8 bytes)

Canary
(8 bytes)

[3] [2] [1] [0] `buf` ← `%rsp`

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

- **Where should canary go?**
- **When should canary checking happen?**
- **What should canary contain?**

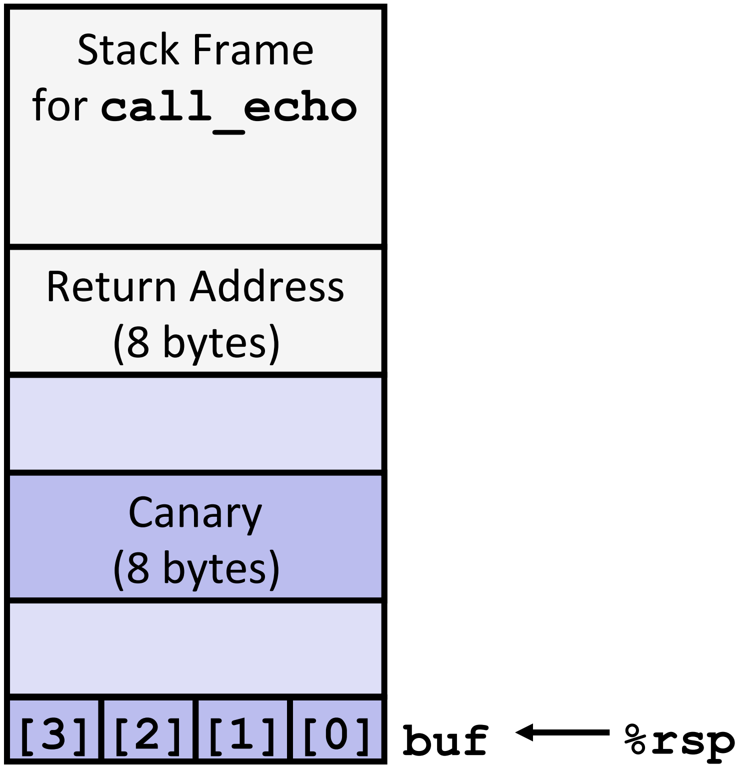
Stack canaries

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
```

Setting Up Canary

Before call to gets



Stack Frame for `call_echo`

Return Address (8 bytes)

Canary (8 bytes)

[3] [2] [1] [0]

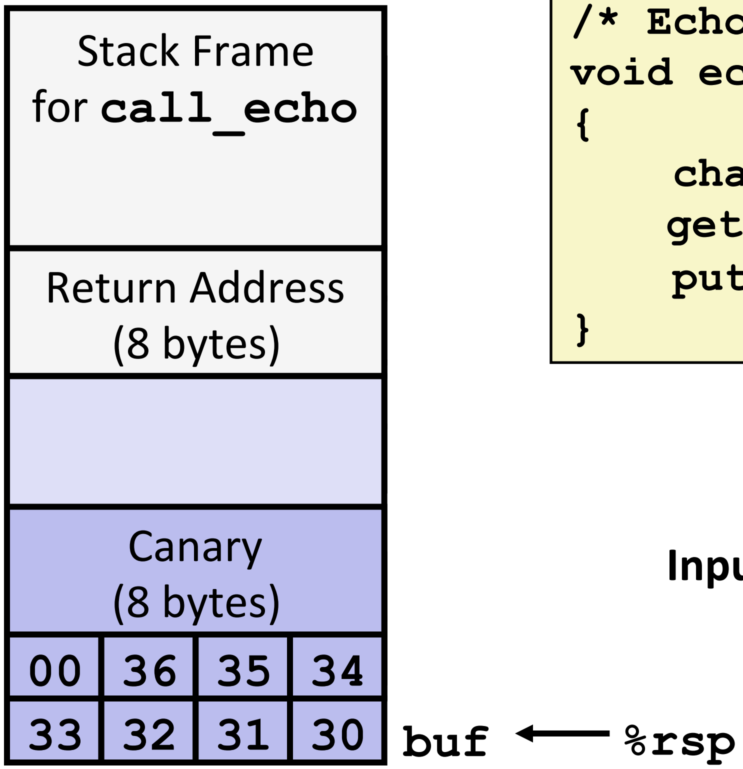
`buf` ← `%rsp`

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

```
echo:
    . . .
    movq    %fs:0x28, %rax  # Get canary
    movq    %rax, 8(%rsp) # Place on stack
    xorl    %eax, %eax    # Erase canary
    . . .
```

Checking 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

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

Input: ***0123456***

```
echo:
    . . .
    movq    8(%rsp), %rax       # Retrieve from stack
    xorq    %fs:0x28, %rax        # Compare to canary
    je      .L6                 # If same, OK
    call    __stack_chk_fail    # FAIL
.L6:        . . .
```

What isn’t caught by canaries?

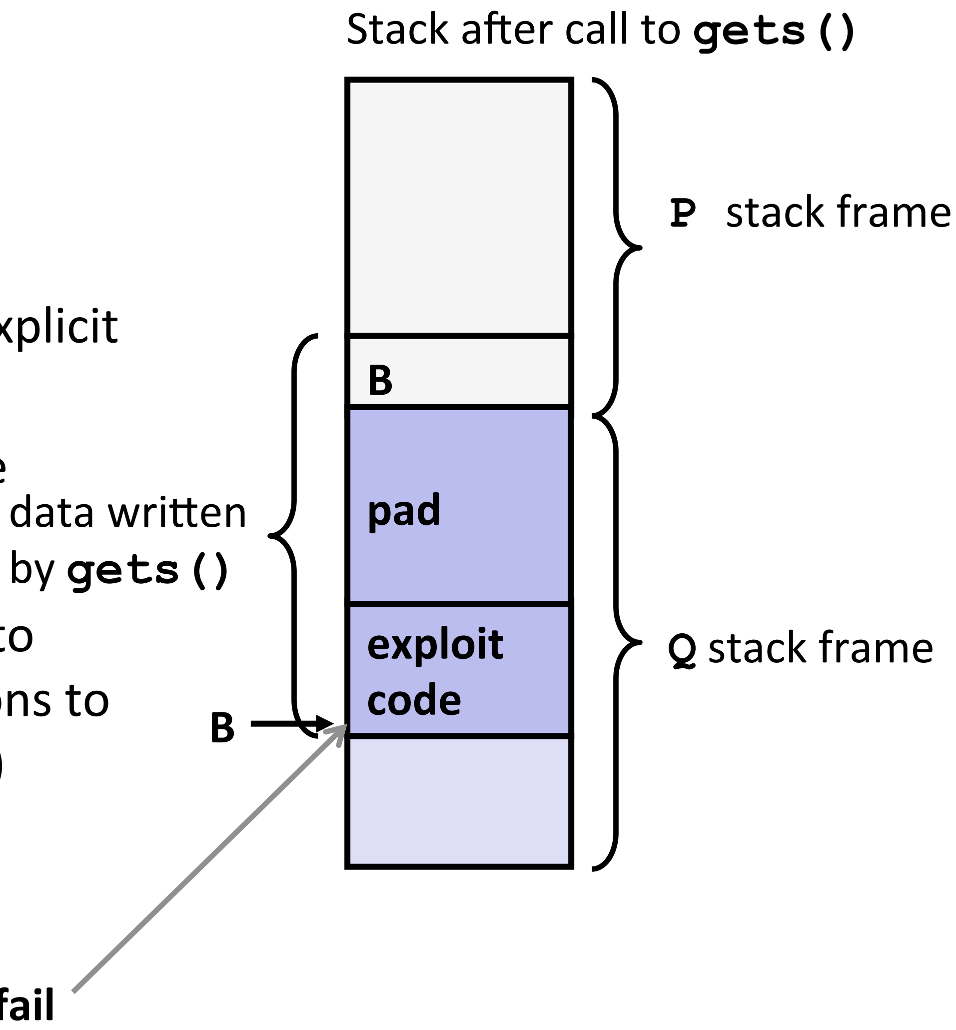
```
void myFunc(char *s)  {
...
}
void echo()
{
    void (*f)(char *);
    f = myFunc;
    char buf[8];
    gets(buf);
    f();
}
```

```
void echo()
{
    long *ptr;
    char buf[8];
    gets(buf);
    *ptr = *(long *)buf;
}
```

- **Overwrite a code pointer before canary**
- **Overwrite a data pointer before canary**

Mitigate #2 prevent code injection

- **NX: Non-executable code segments**
  - Old x86 has no "executable" permission bit, X86-64 added explicit "execute" permission
  - Stack marked as non-executable
- **Does not defend against:**
  - Modify return address to point to code in stdlib (which has functions to execute any programs e.g. shell)



Any attempt to execute this code will fail

Mitigate #2 attempts to craft “attacking code” (ASLR)

- **Insight: attacks often use hard-coded address → make it difficult for attackers to figure out the address to use**
- **Address Space Layout Randomization**
  - Stack randomization
    - Makes it difficult to determine where the return addresses are located
  - Randomize the heap, location of dynamically loaded libraries etc.

The rest of the slides are optional

Return-Oriented Programming Attacks

- **Challenge (for hackers)**
  - Stack randomization makes it hard to predict buffer location
  - Non-executable stack makes it hard to insert arbitrary binary code
- **Alternative Strategy**
  - Use existing code
    - E.g., library code from stdlib
  - String together fragments to achieve overall desired outcome
- **How to concoct an arbitrary mix of instructions from the current running program?**
  - Gadgets: A short sequence of instructions ending in `ret`
    - Encoded by single byte `0xc3`

Gadget Example #1

```
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            retq
```

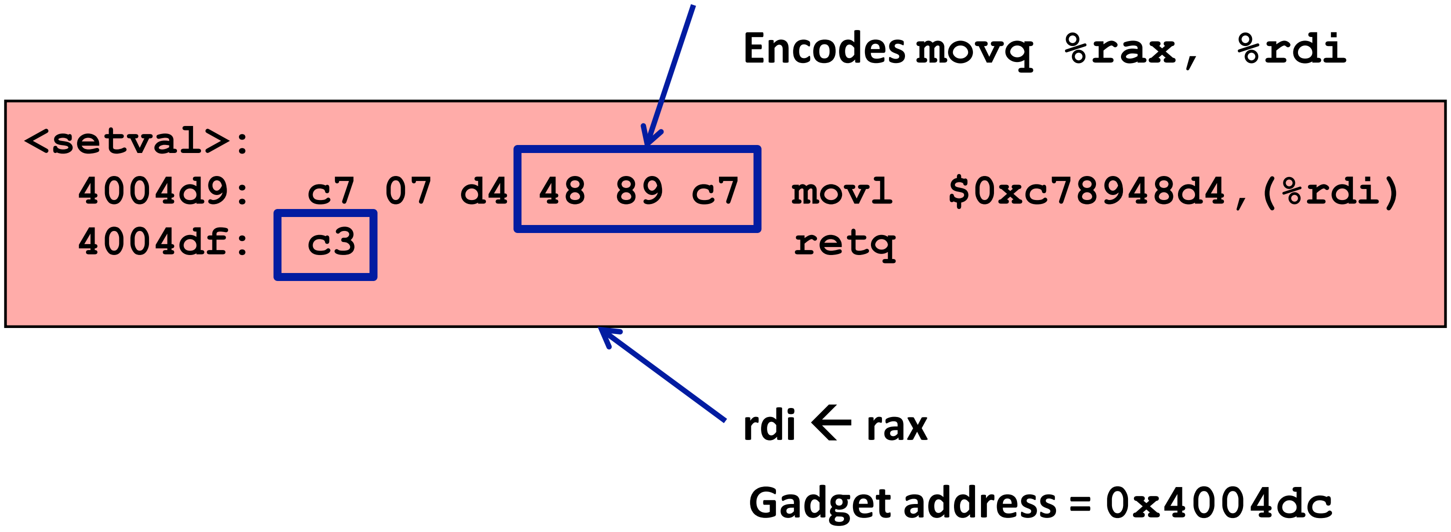
rax ← rdi + rdx

Gadget address = `0x4004d4`

- Use tail end of existing functions

Gadget Example #2

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



Encodes `movq %rax, %rdi`

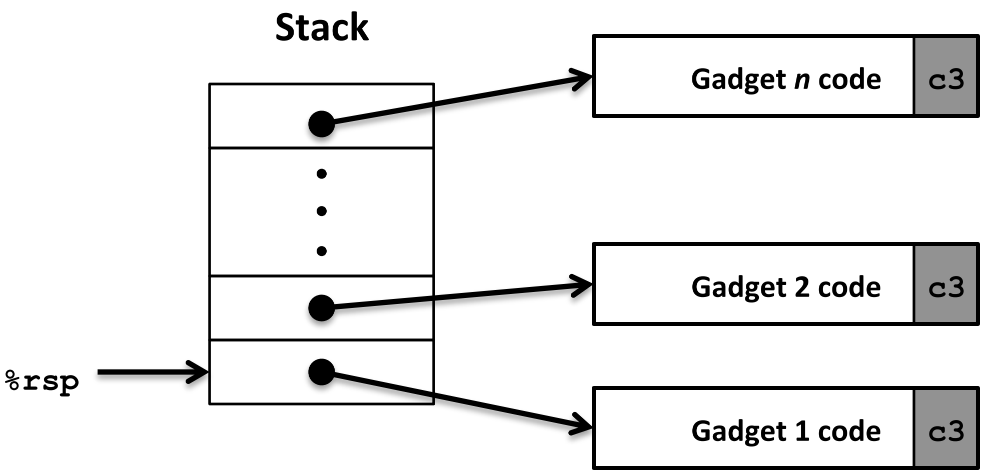
```
<setval>:
  4004d9:  c7 07 d4 48 89 c7  movl  $0xc78948d4,(%rdi)
  4004df:  c3                 retq
```

rdi ← rax

Gadget address = `0x4004dc`

- Repurpose byte codes

ROP Execution



- **Trigger with `ret` instruction**
  - Will start executing Gadget 1
- **Final `ret` in each gadget will start next one**