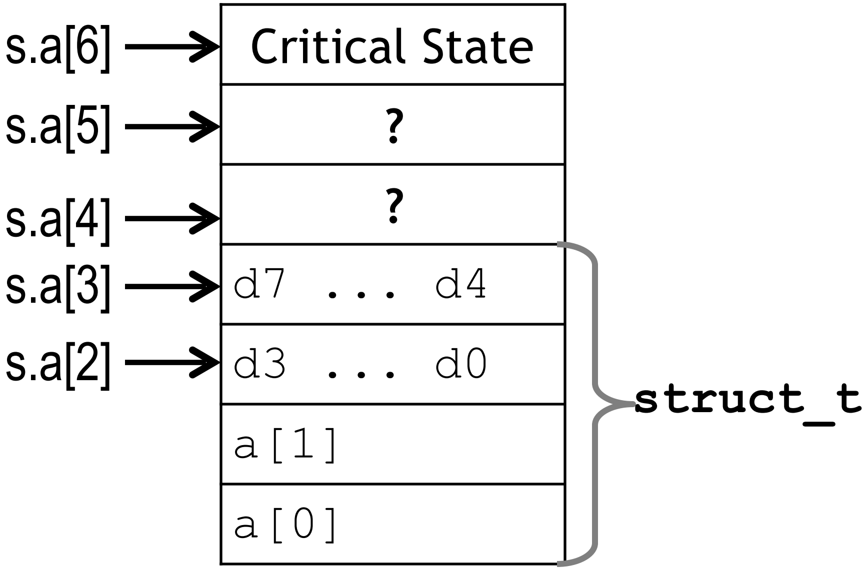
Machine-Level Programming V: Buffer overflow

Slides adapted from Bryant and O’Hallaron

Recall: 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
```

Such problems are a BIG deal

- **Generally called a "buffer overflow"**
  - when exceeding the memory size allocated for an array
- **#1 technical cause of security vulnerabilities**
- **Most common form**
  - Unchecked lengths on string inputs
  - Particularly for bounded character arrays on the stack

Bad APIs of stdlib make buffer overflow likely

- **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];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

ç how big
is big enough?

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

```
unix>./a.out
Type a string:0123456789012345678901 23
0123456789012345678901 23
```

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

Buffer Overflow Disassembly

echo:

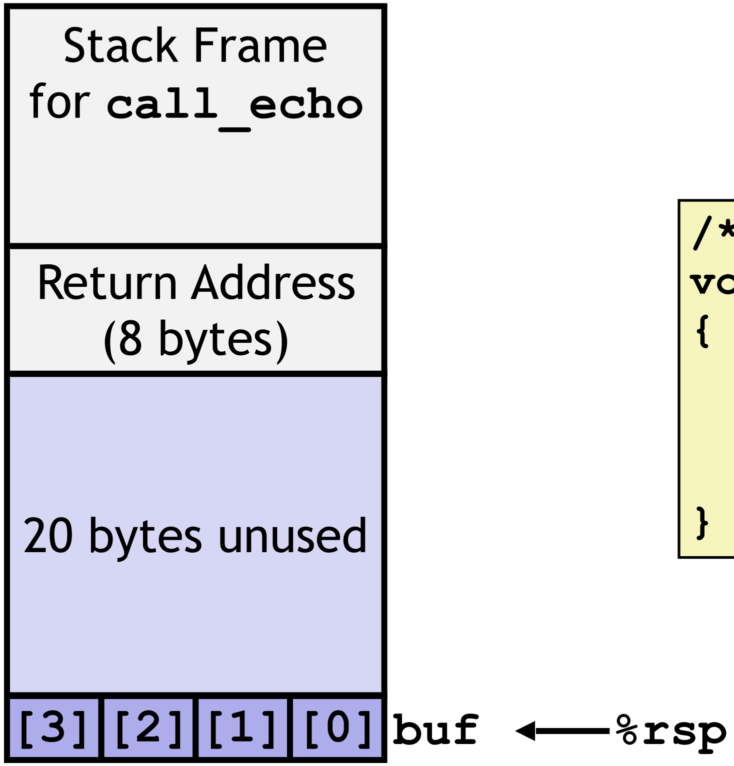
```
00000000004006cf <echo>:
 4006cf:  48 83 ec 18                sub    $0x18,%rsp
 4006d3:  48 89 e7                   mov    %rsp,%rdi
 4006d6:  e8 a5 ff ff ff             callq  400680 <gets>
 4006db:  48 89 e7                   mov    %rsp,%rdi
 4006de:  e8 3d fe ff ff             callq  400520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

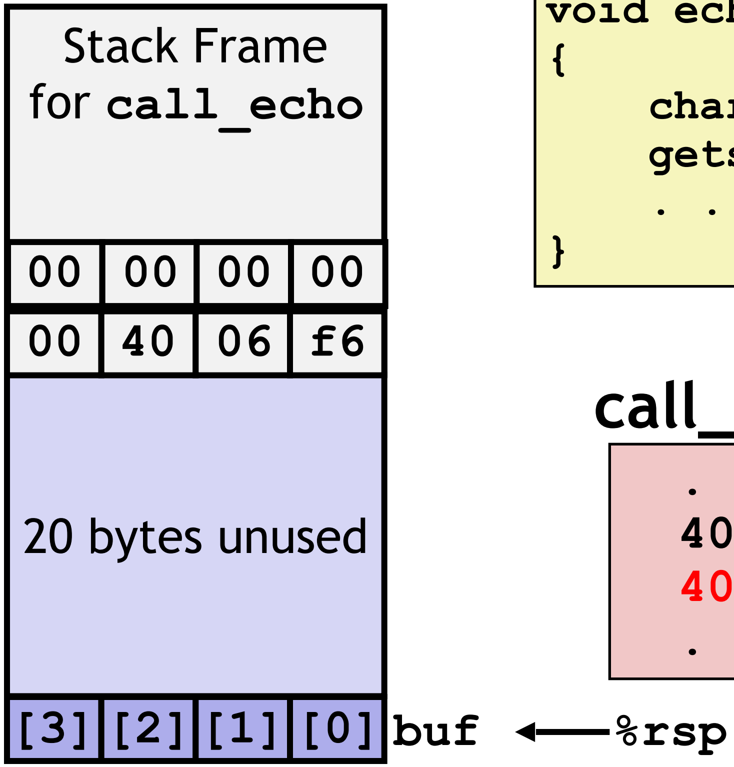


```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

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

Buffer Overflow Stack Example

Before call to gets



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

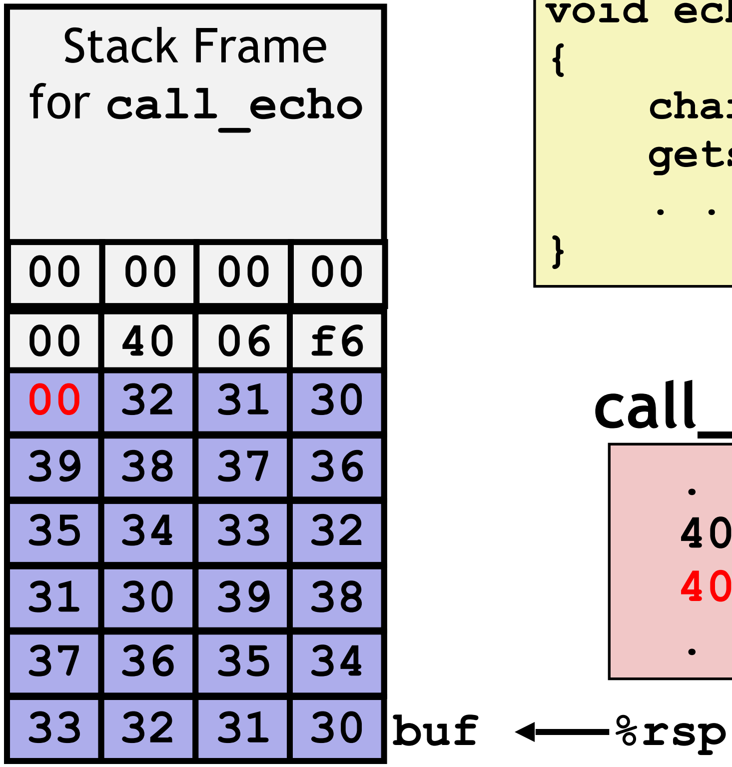
```
echo:
   subq  $24, %rsp
   movq  %rsp, %rdi
   call  gets
   . . .
```

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);
    . . .
}
```

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

call_echo:

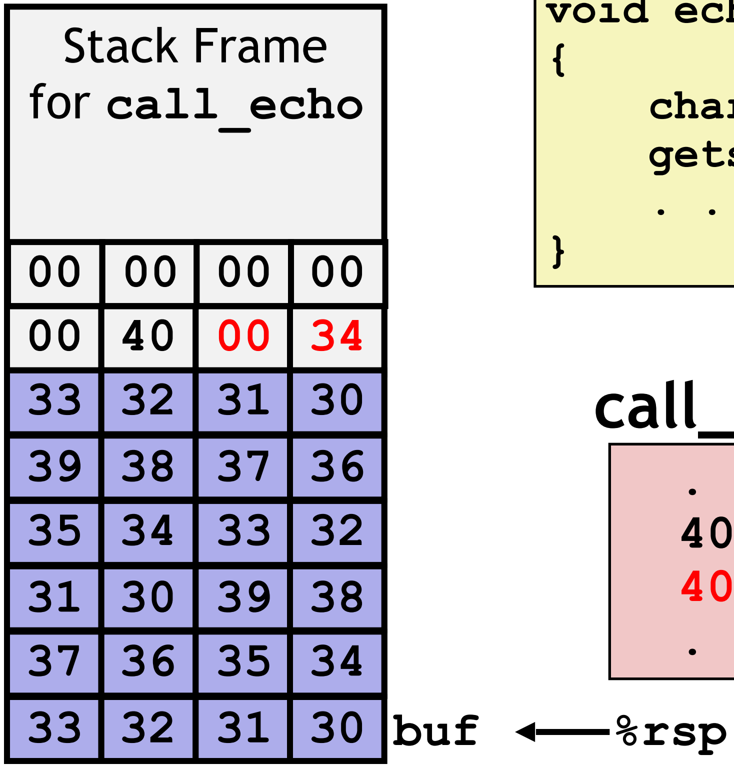
```
 . . .
 4006f1:  callq  4006cf <echo>
 4006f6:  add    $0x8,%rsp
 . . .
```

```
unix>./bufdemo-nsp
Type a string:01234567890123456789012
01234567890123456789012
```

Overflowed buffer, but did not corrupt state

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:

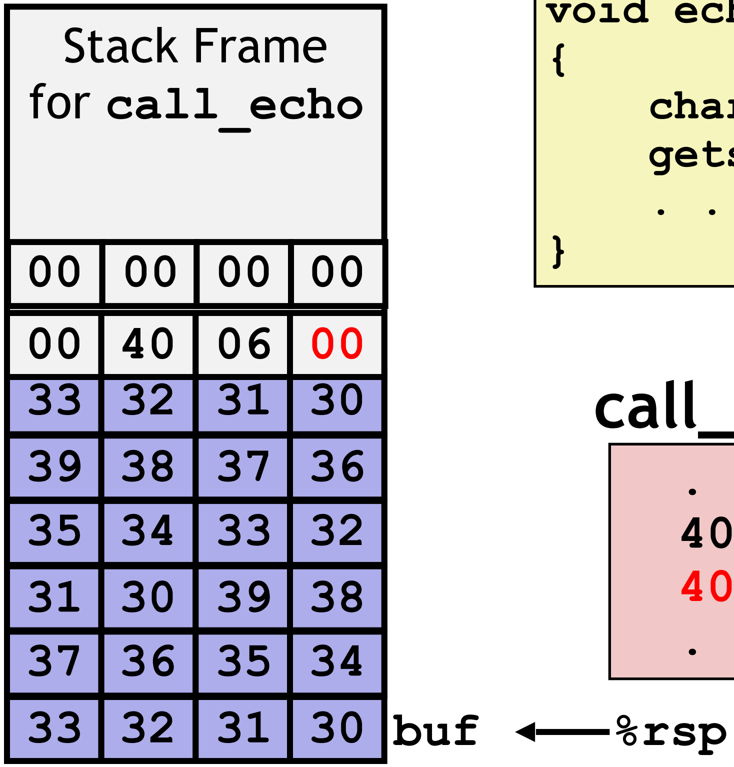
```
 . . .
 4006f1:  callq  4006cf <echo>
 4006f6:  add    $0x8,%rsp
 . . .
```

```
unix>./bufdemo-nsp
Type a string:0123456789012345678901234
Segmentation Fault
```

Overflowed buffer and corrupted return pointer

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:

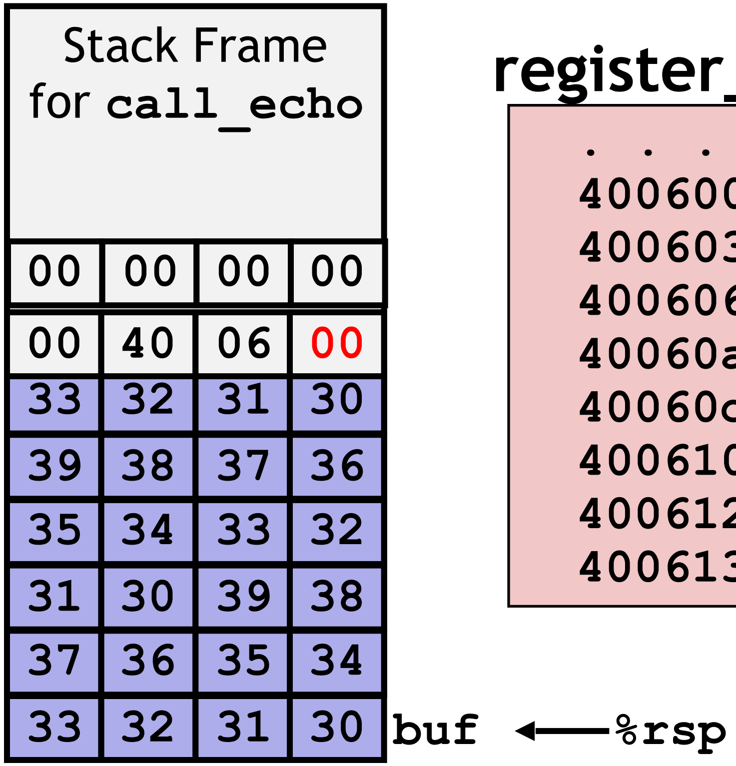
```
  . . .
  4006f1:  callq  4006cf <echo>
  4006f6:  add    $0x8,%rsp
  . . .
```

```
unix>./bufdemo-nsp
Type a string:01234567890123456789012
01234567890123456789012
```

Overflowed buffer, corrupted return pointer, but program seems to work!

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`

What’s the big deal about buffer overflow?

```
//webserver code
void
read_user_request() {
    char buf[200];
    gets(buf);
    ...
}
```

- Systems software is often written in C:
  - operating system, file systems, database, compilers, network servers, command shells, …
- How does attackers take advantage of this bug?
  1. overwrite with a carefully chosen return address
  2. executes malicious code (injected by attacker or elsewhere in the running process)
- What can attackers do once they are executing code?
  - 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 etc.

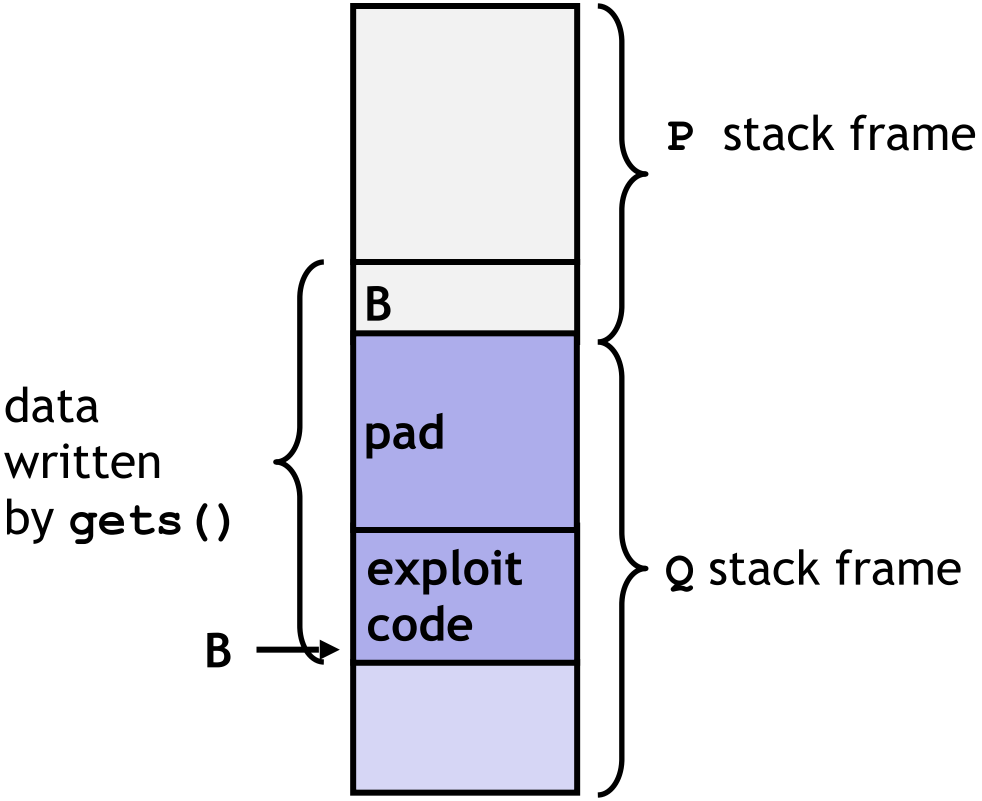
Example exploit: Code Injection Attacks

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

return address A

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

Stack after call to `gets()`



- **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**

Exploits Based on Buffer Overflows

- ***Buffer overflow bugs can allow remote machines to execute arbitrary code on victim machines***
- **Common in real progams**
- **Examples across the decades**
  - "Internet worm" (1988)
  - Attacks on Xbox
  - Jaibreaks on iPhone
  - …
- **Recent measures make these attacks much more difficult**

Example: the original Internet worm (1988)

- **Exploited a few vulnerabilities to spread**
  - Early finger server (fingerd) used `gets()` to read client inputs:
    - `finger sexton@nyu.edu`
  - Worm attacked fingerd server by sending phony argument:
    - `finger “exploit-code padding new-return-address”`
    - exploit code: executed a root shell on the victim machine with a direct TCP connection to the attacker.
- **Once on a machine, scanned for other machines to attack**
  - invaded ~6000 computers in hours (10% of the Internet ☺️)
    - see June 1989 article in *Comm. of the ACM*
  - the young author of the worm was prosecuted…

OK, what to do about buffer overflow attacks

- Write correct code: avoid overflow vulnerabilities

- Mitigate attack despite buggy code

Recap

- The size of a union
  - max(sizeof(field1), sizeof(field2)…)
- The size of a struct
  - sum(sizeof(field1), sizeof(field2)… + padding).
- There are four major memory segments, stack, heap, data and text
  - Each segment can be readable, writable and/or executable.
- Accessing a memory segment without correct permission will cause segmentation fault.
- Accessing an address that exceeds the memory size allocated for an array is called “buffer overflow”

Recap

- Buffer overflow does not always cause segmentation fault.
  - The stack can be overwritten with some noise or malicious code (the stack is readable and writable so there is no segmentation fault).
- Using the technique, we can attack some programs and machines.
  - Internet Worm developed by Robert Morris.

Applications

- Pirate a software…

Avoid Overflow Vulnerabilities in Code

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

- **Better coding practices**
  - e.g. use library routines that limit string lengths, `fgets` instead of `gets`, `strncpy` instead of `strcpy`
- **Use a memory-safe language instead of C**
- **bug finding tools?**

Mitigate BO attacks despite buggy code

- **A buffer overflow attack typically needs two components:**
  1. Control-flow hijacking
     - overwrite a code pointer (e.g. return address) that’s later invoked
  2. Need some interesting code in the process’ memory
     - e.g. put code in the buffer
     - Process already contains a lot of code in predictable location
- **How to mitigate attacks? make #1 or #2 hard**

Mitigate #1 (control flow hijacking)

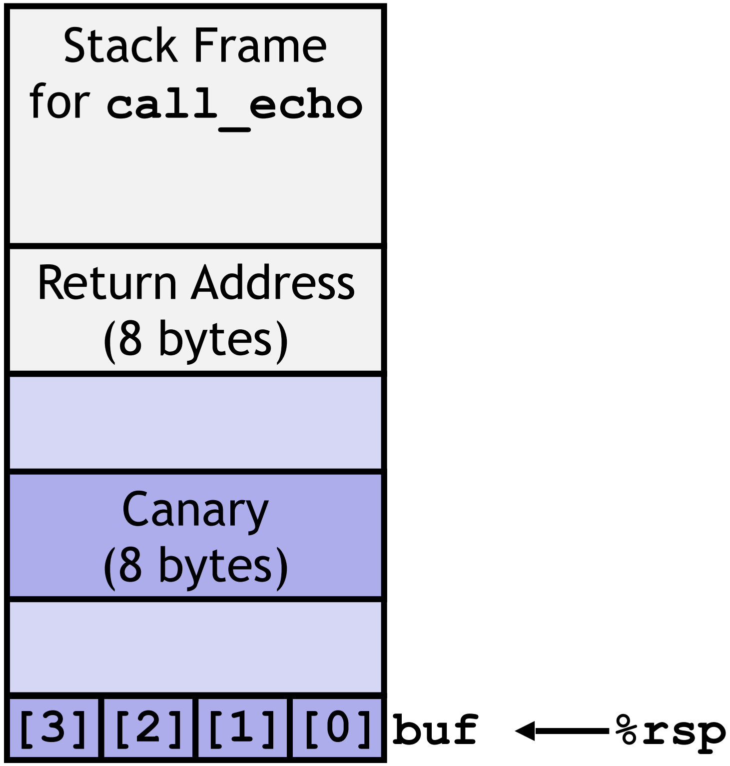
- **Idea: Catch over-written return address before invocation!**
  - Place special value (“canary”) on stack just beyond buffer
  - Check for corruption before exiting function
- **GCC Implementation**
  - `-fstack-protector`
  - Now the default

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

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

Setting Up Canary

Before call to gets



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

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

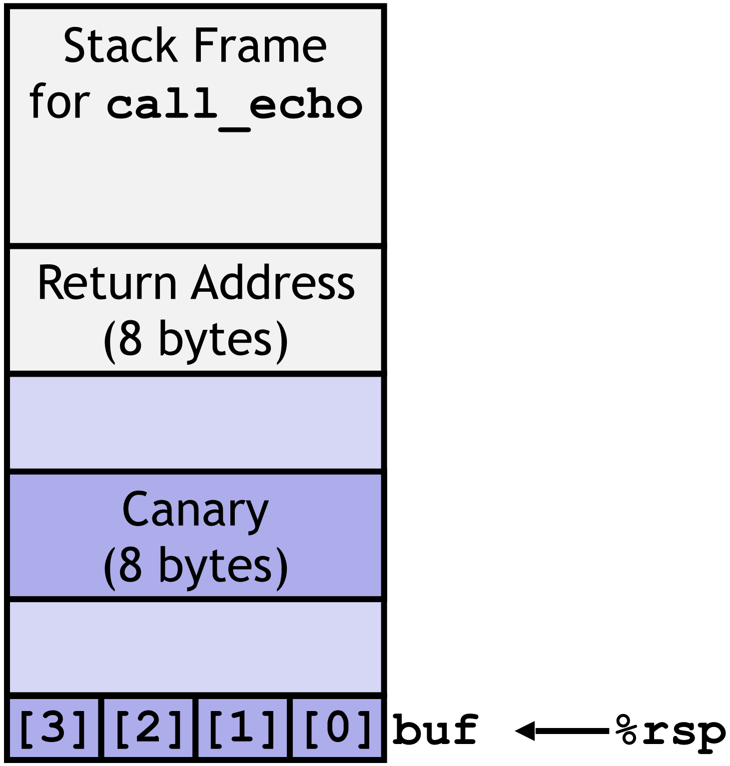
Stack canaries

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

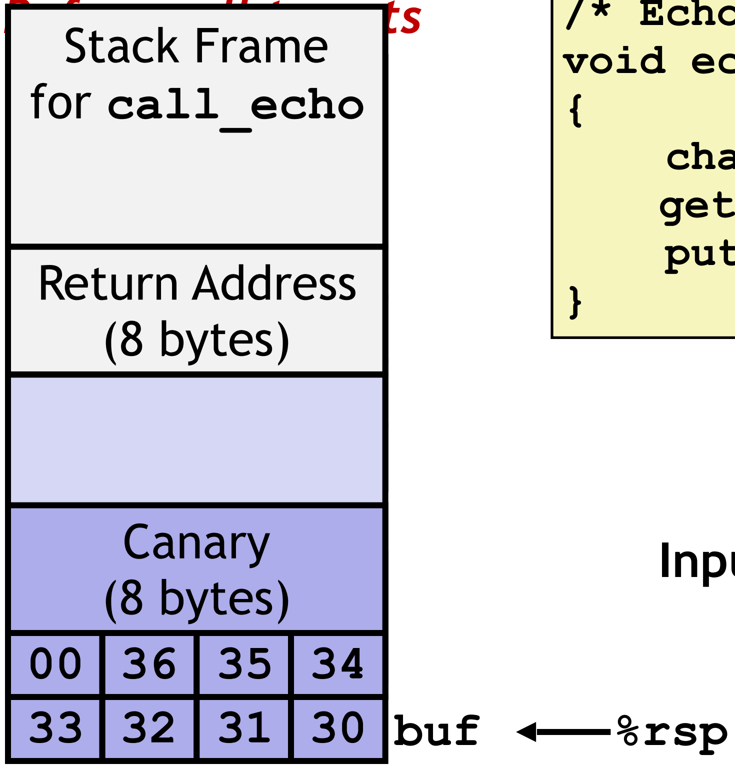


```
/* Echo Line */
void echo()
{
    char buf[4];
    gets(buf);
    puts(buf);
}
```

```
echo:
    . . .
    movq    %fs:40, %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

buf ← **%rsp**

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

Input: ***0123456***

```
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:        . . .
```

What isn’t caught by canaries?

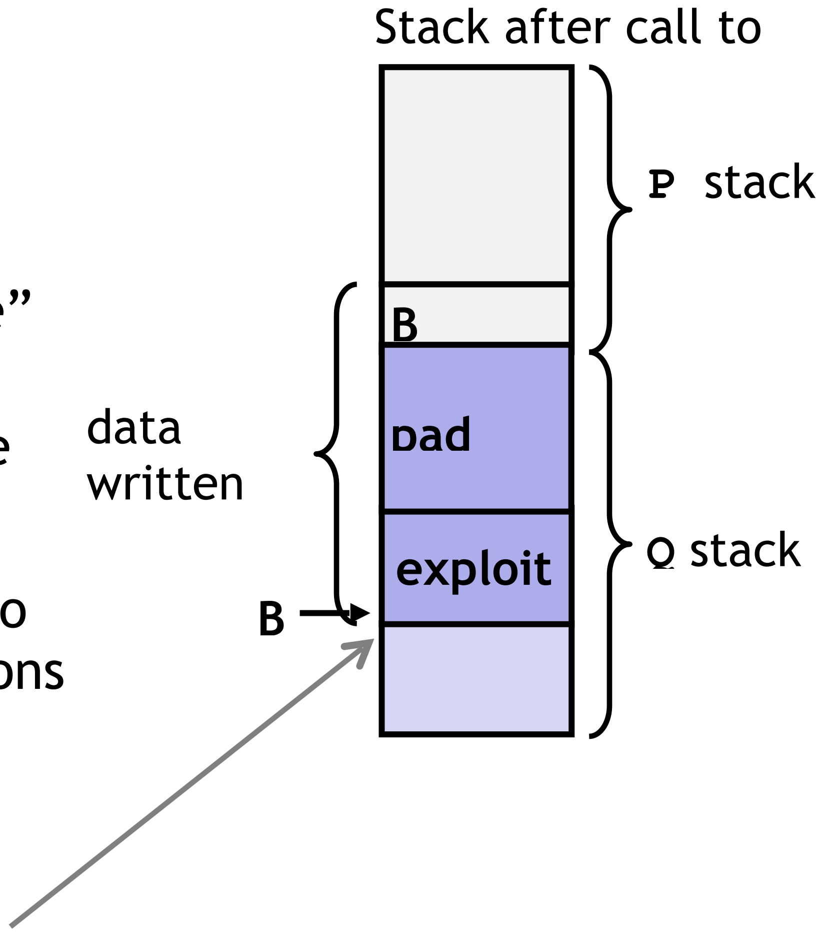
```
void myFunc(char *s) {
...
}
void echo()
{
    void (*f)(char *);
    f = myFunc;
    char buf[8];
    gets(buf);
    f();
}
```

```
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 attempts to craft “attacking code” (NX)

- **NX: Non-executable code segments**
  - Traditional 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.

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
  - *Does not overcome stack canaries*
- **How to concoct an arbitrary mix of instructions from the current running program?**
  - Gadgets: Sequence of instructions ending in **`ret`**
    - Encoded by single byte **`0xc3`**
  - Code positions fixed from run to run
  - Code is executable

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
```

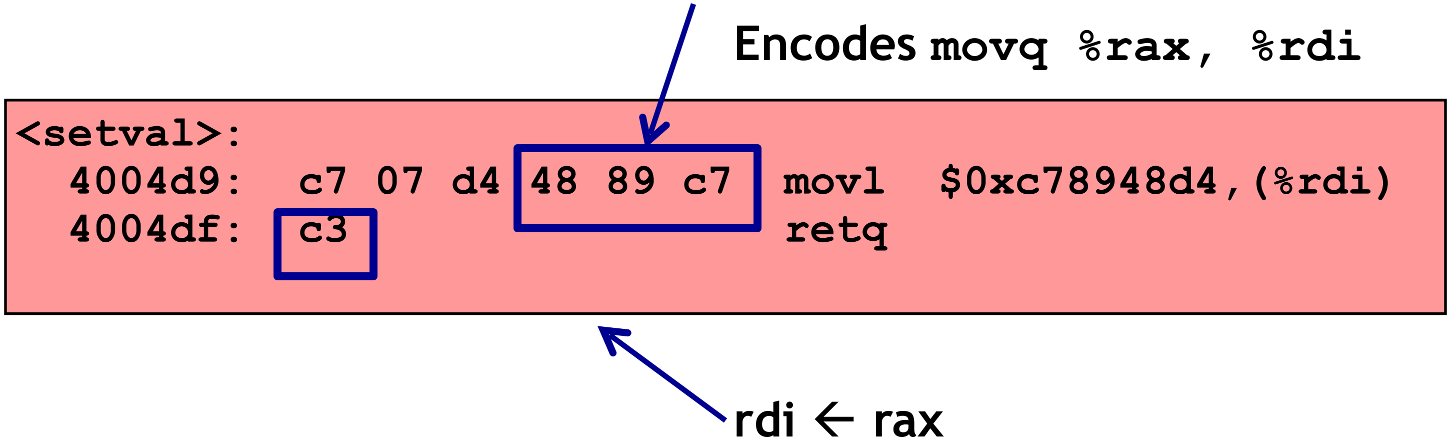
rax ← rdi + rdx

Gadget address = `0x4004d4`

- Use tail end of existing functions

Gadget Example #2

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



Encodes `movq %rax, %rdi`

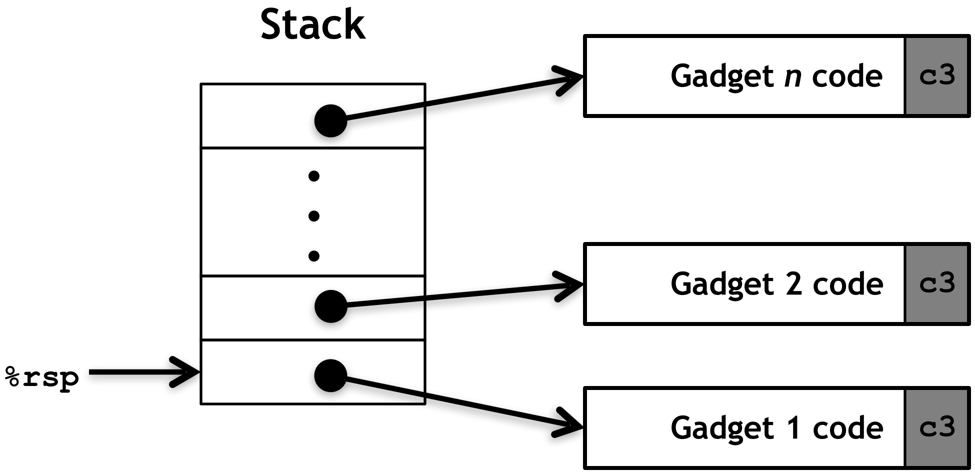
```
<setval>:
  4004d9:   c7 07 d4 48 89 c7  movl   $0xc78948d4,(%rdi)
  4004df:   c3                 retq
```

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