Machine-level programming
Segmentation Fault
&
Buffer overflow

Jinyang Li

What we've learnt so far

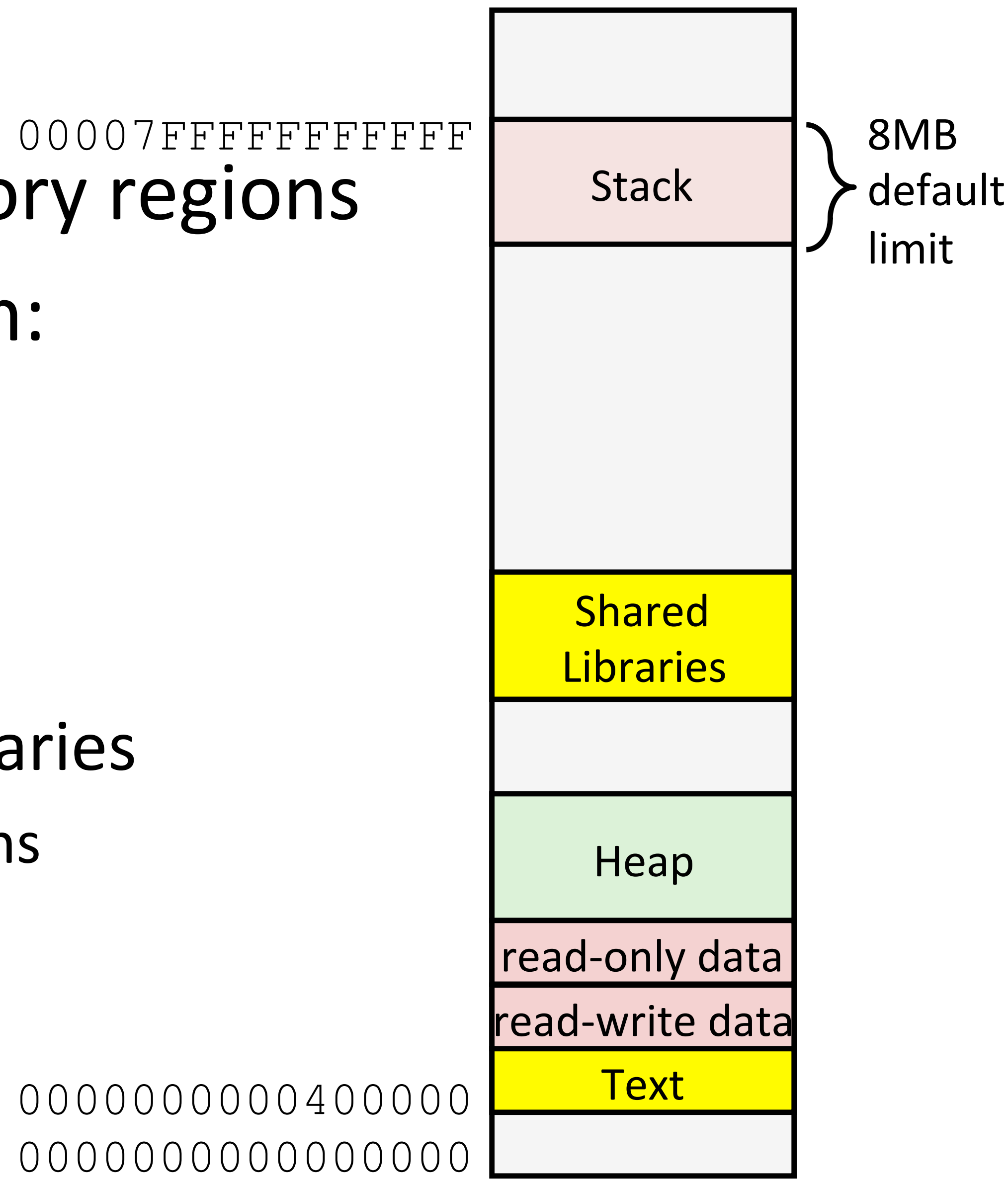
- Instructions and data stored in memory
  - some local variables are only stored in registers
- CPU execution
  - control flows: sequential, jumps, call/ret

Today’s lesson plan

- What’s segmentation fault?
- What’s buffer overflow?

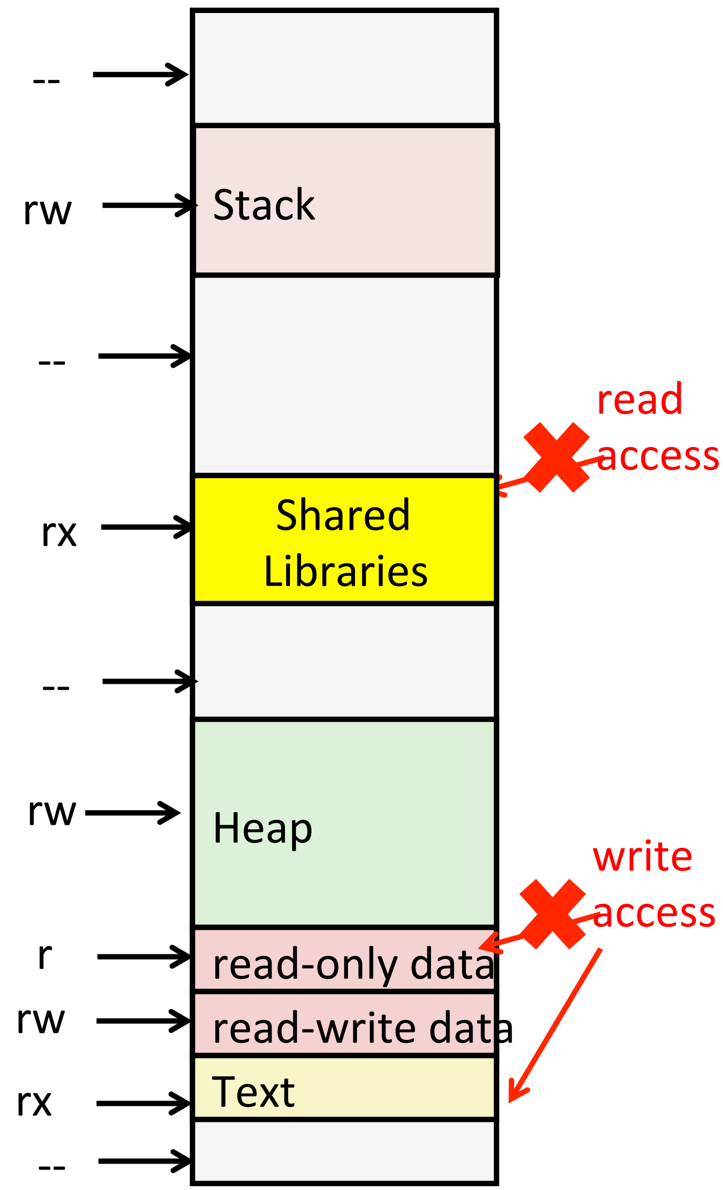
Recap: Linux Memory Layout

- OS allocates memory regions to a running program:
  - Stack
  - Heap
  - Data
  - Text / Shared Libraries
    - aka x86 instructions



Memory region has hardware enforced permission

- **Permissions are:**
  - readable (r),
  - executable (x)
  - writable(w)
  - no-access (-)
- **Segmentation fault occurs when an instruction tries to access "illegal" memory**
  - Read or write "no-access" memory
  - Write to "read-only" memory



Segmentation fault example

```
void foo(int *p) {
    *p = 5;
}
```

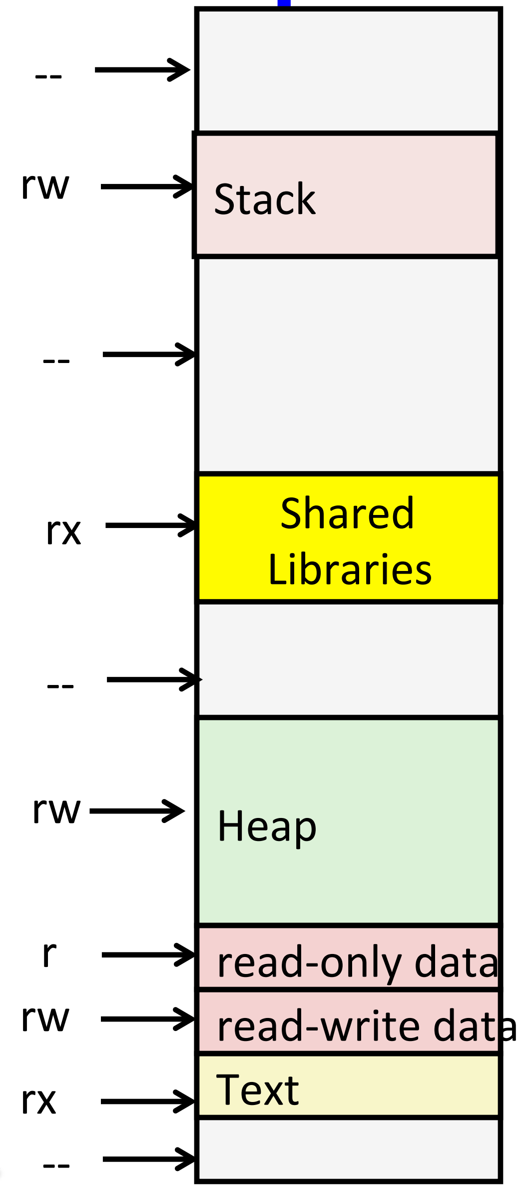
```
int main() {
    foo((int *)10);
    printf("finished\n");
}
```

```
(gdb) r
Starting program: /oldhome/jinyang/a.out

Program received signal SIGSEGV, Segmentation fault.
bar (p=p@entry=0xa) at haha.c:13
13              *p = 5;
(gdb) p p
$1 = (int *) 0xa
(gdb) x/4xb 0xa
0xa:    Cannot access memory at address 0xa
(gdb) 
```

Examine memory contents at address 0xa
4xb → 4 bytes in hex

Segmentation fault example

```
void foo(int *p) {
    *p = 5;
}

int main() {
    foo((int *)10);
    printf("finished\n");
}
```



Another segmentation fault example

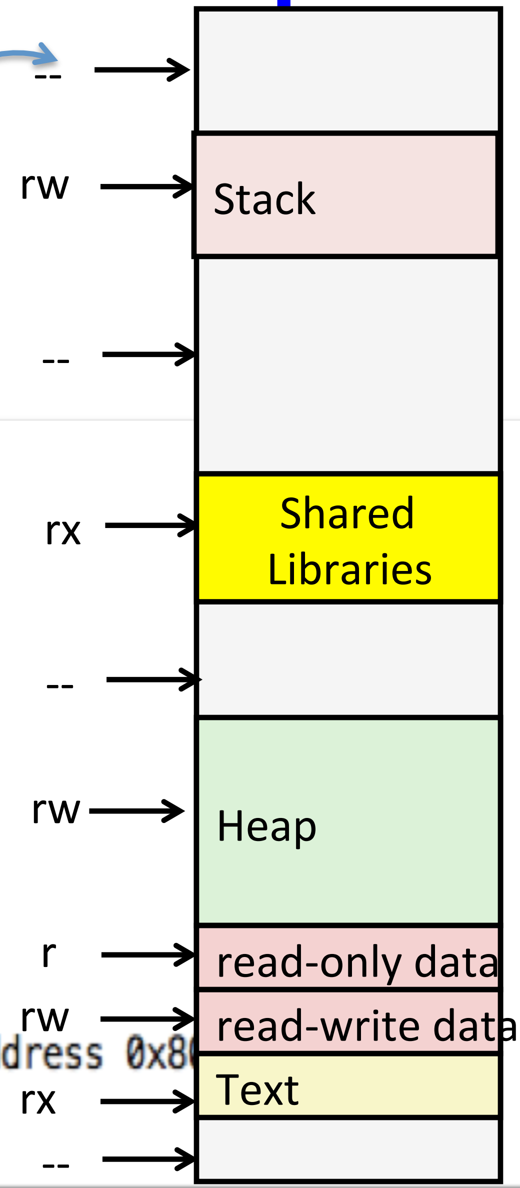
```
int main() {
   char s1[6] = “hello”;
   s1[10000] = ‘H’;
   printf(“finished\n”);
}
```

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /oldhome/jinyang/a.out

Program received signal SIGBUS, Bus error.
main () at haha.c:7
7               s1[10000] = 'H';
(gdb) p &s1[0]
$3 = 0x7fffffffdf70 "hello"
(gdb) p &s1[10000]
$4 = 0x800000000680 <error: Cannot access memory at address 0x800000000680>
(gdb)
```

Segmentation fault example

```
int main() {
   char s1[6] = “hello”;
   s1[10000] = ‘H’;
   printf(“finished\n”);
}
```

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /oldhome/jinyang/a.out

Program received signal SIGBUS, Bus error.
main () at haha.c:7
7               s1[10000] = 'H';
(gdb) p &s1[0]
$3 = 0x7fffffffdf70 "hello"
(gdb) p &s1[10000]
$4 = 0x800000000680 <error: Cannot access memory at address 0x8
(gdb)
```



Not all buggy memory references access "illegal" memory
→ buffer overflow exploits

Buggy code

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

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

read a line from stdin until a terminating newline or EOF, which it replaces with a NULL byte.

writes string and a trailing newline to stdout.

buffer overruns,
but things seem ok??

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

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

Buggy code Disassembly

echo:

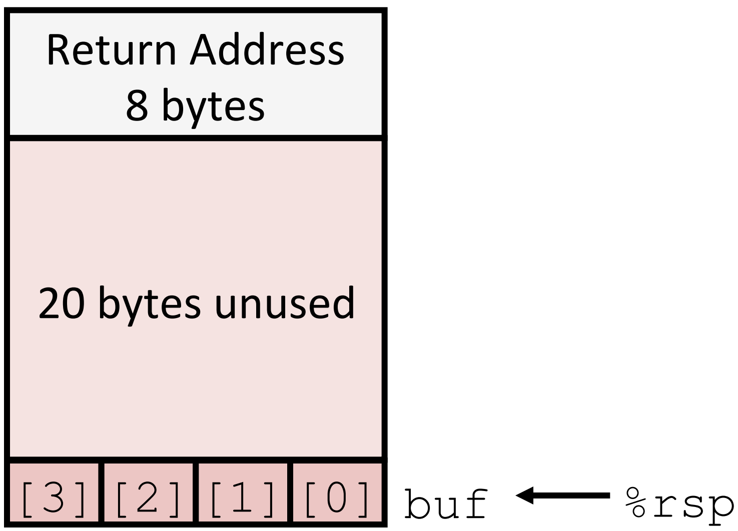
```
0000000004006cf <echo>:
 4006cf:  48 83 ec 18                sub    $0x18,%rsp
 4006d3:  48 89 e7                   mov    %rsp,%rdi
 4006d6:  e8 a5 ff ff ff             callq  400680 <gets>
 4006db:  48 89 e7                   mov    %rsp,%rdi
 4006de:  e8 3d fe ff ff             callq  400520 <puts@plt>
 4006e3:  48 83 c4 18                add    $0x18,%rsp
 4006e7:  c3                         retq
```

main:

```
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
```

Buggy code's stack

Before call to gets



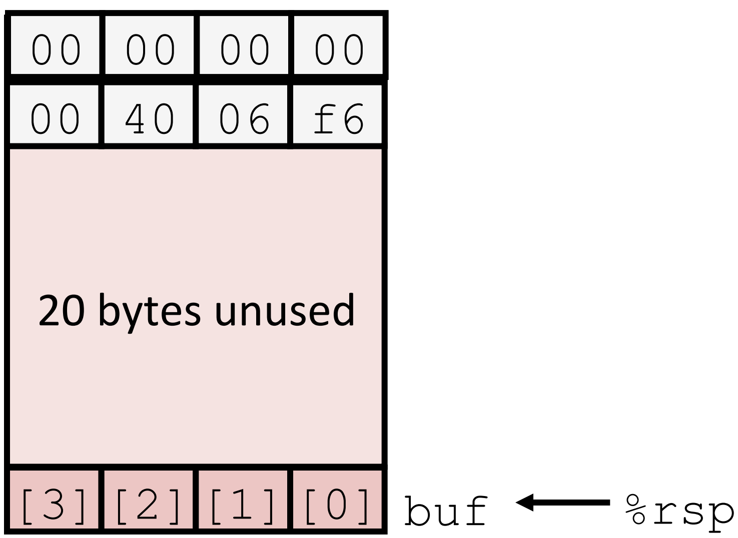

```
void echo()
{
    char buf[4];
    gets(buf);
    puts(buf);
}
```

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

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

Buggy code's stack

Before call to gets




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

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

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

Buffer overflow on the stack

After call to gets

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);
    puts(buf);
}
```

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

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

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

Buffer overflow corrupts return address

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

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

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

After call to gets

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

overflow corrupted return address

```
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 main
3. ret of gets

Buffer Overflow corrupts return address

After call to gets

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

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
  ...
```

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

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

Buffer overflow corrupts return address, program jumps to random code

After call to gets

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?

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

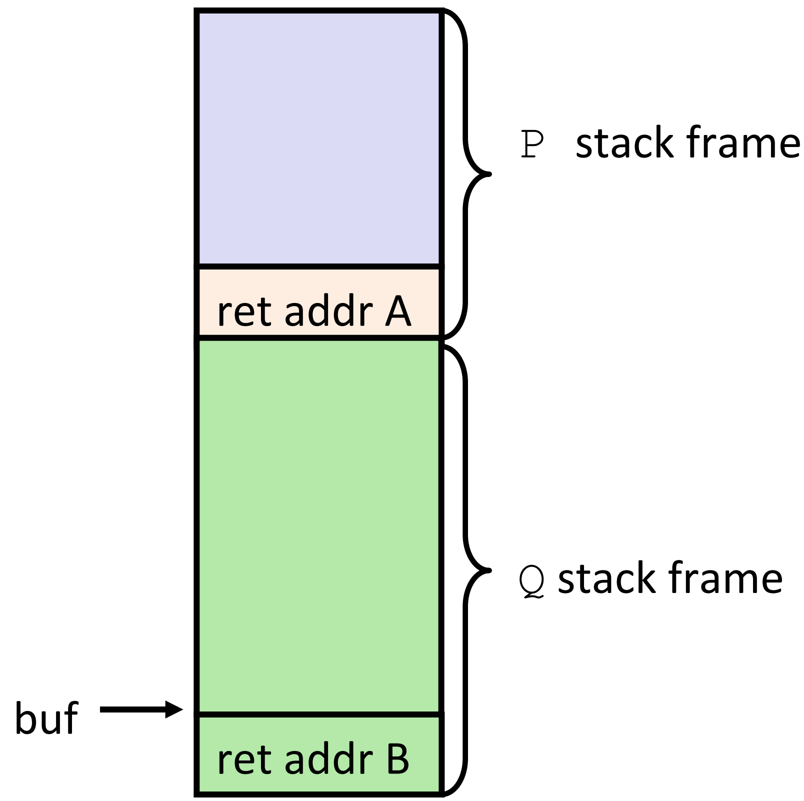
Example exploit: Code Injection Attacks

```
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

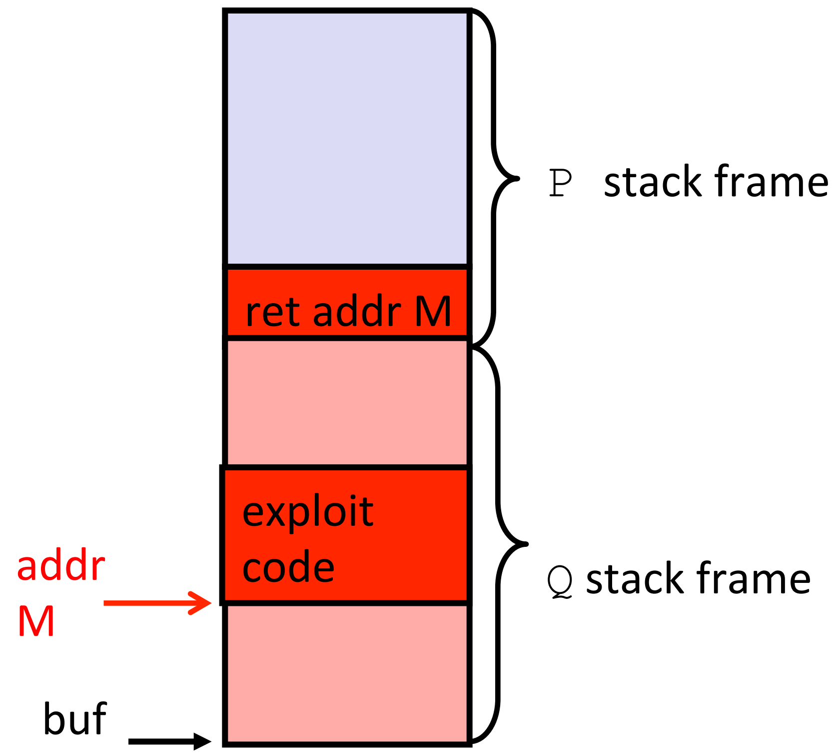
```
void P(){
  Q();
  ...
}
```

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()

Past Code-Injection 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 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 attacks 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 safe library APIs 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)
  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

Prevent control flow hijacking

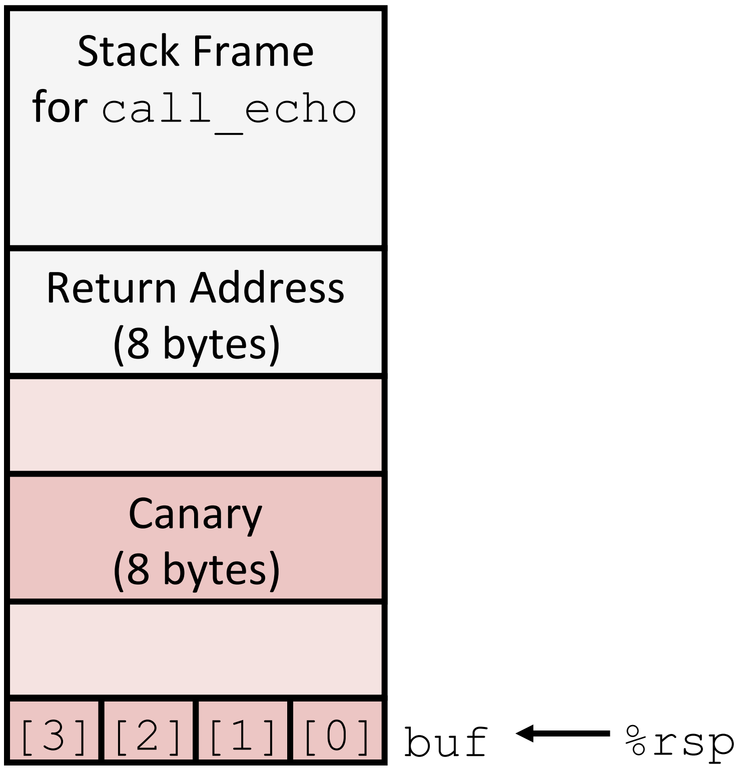
- Idea: Catch over-written return address before invocation!
  - Place special value ("canary") on stack just beyond buffer
  - Check for corruption before exiting function
- GCC Implementation
  - **`-fstack-protector`**
  - Now the default

```
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




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

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

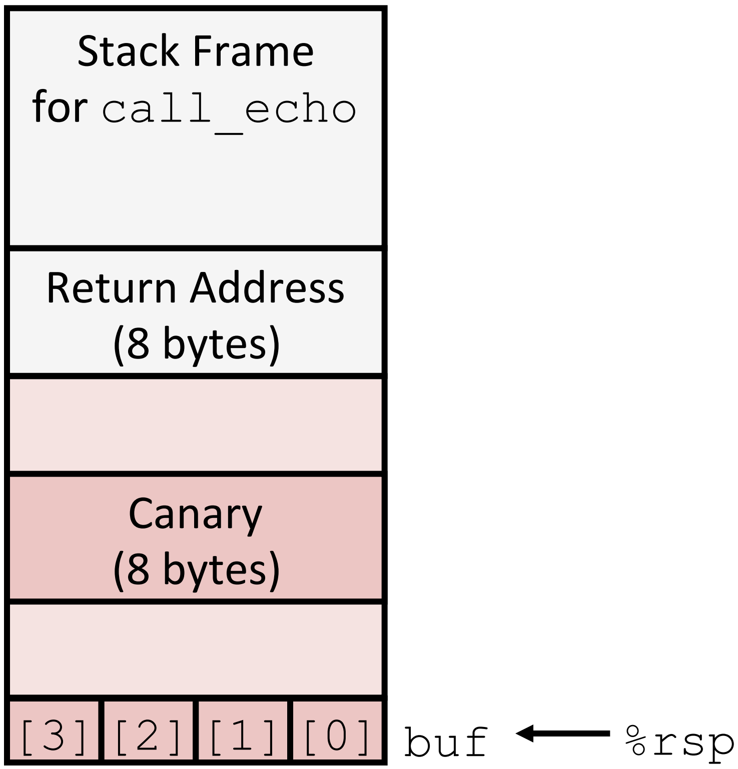
Stack canaries

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

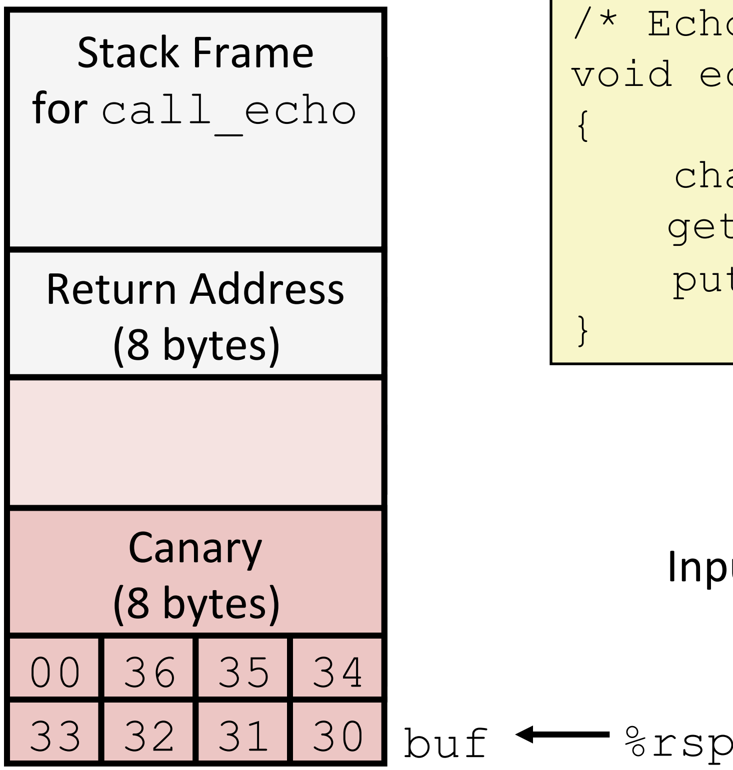



```
/* Echo Line */
void echo()
{
    char buf[4];
    gets(buf);
    puts(buf);
}
```

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

Checking Canary

After call to gets




```
/* 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?

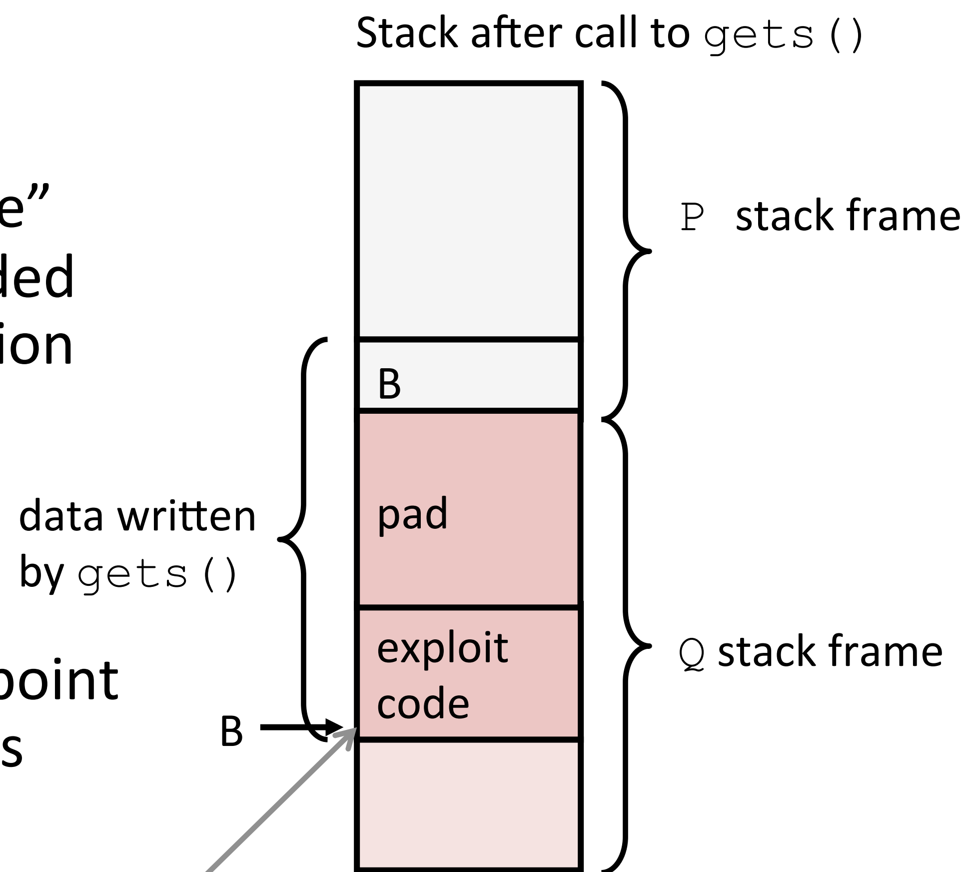
```
void myFunc(char *s) {
...
}
void echo()
{
    void (*f)(char *);
    f = myFunc;
    char buf[8];
    gets(buf);
    f();
}
```

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

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

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

Prevent attempts to inject "useful" return addresses

- 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

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

Encodes `movq %rax, %rdi`

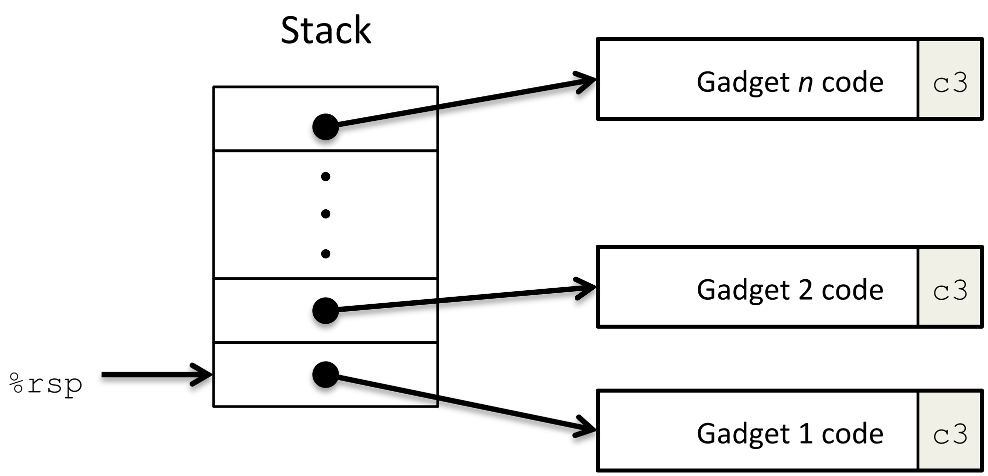
```
<setval>:
  4004d9:  c7 07 d4 48 89 c7  movl  $0xc78948d4,(%rdi)
  4004df:  c3                 retq
```

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