[CSED211] Introduction to Computer Software Systems

Lecture 8: Advanced Topics

Prof. Jisung Park

2023.10.16

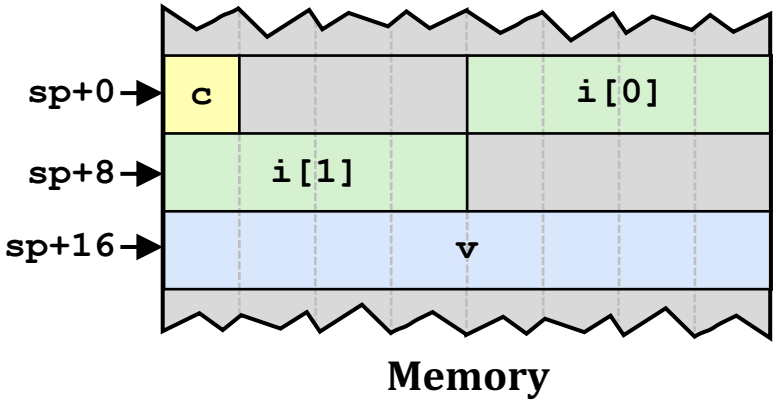
Lecture Agenda

- Unions
- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection

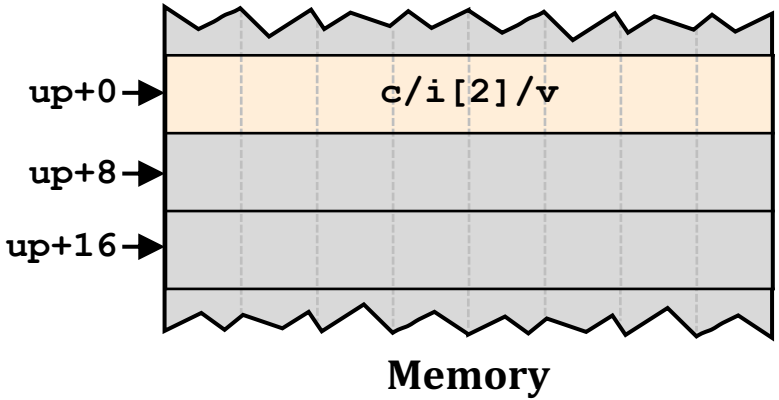
Union Allocation

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

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



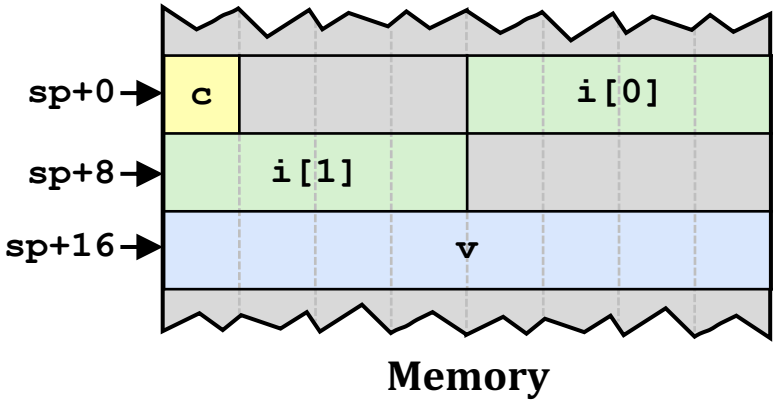

```
union U1 {
  char c;
  int i[2];
  double v;
} *up;
```



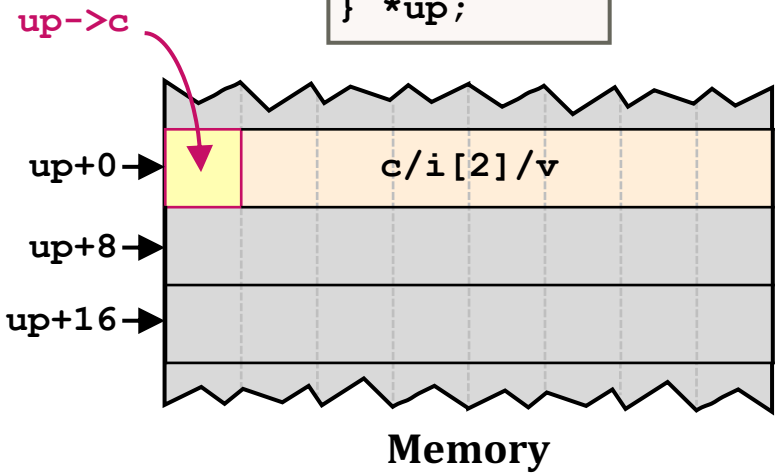
Union Allocation

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

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



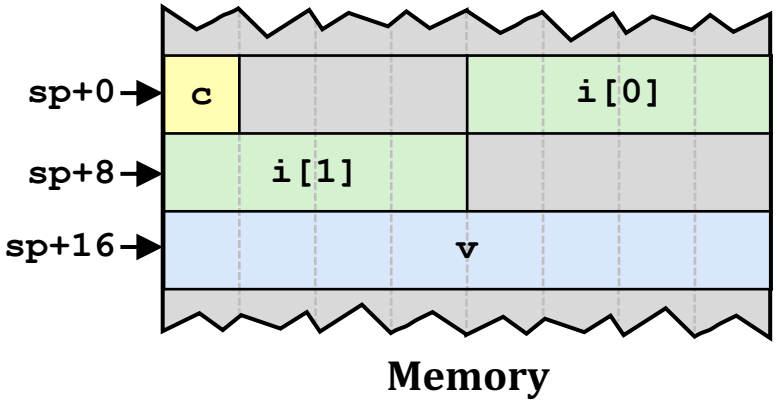

```
union U1 {
  char c;
  int i[2];
  double v;
} *up;
```



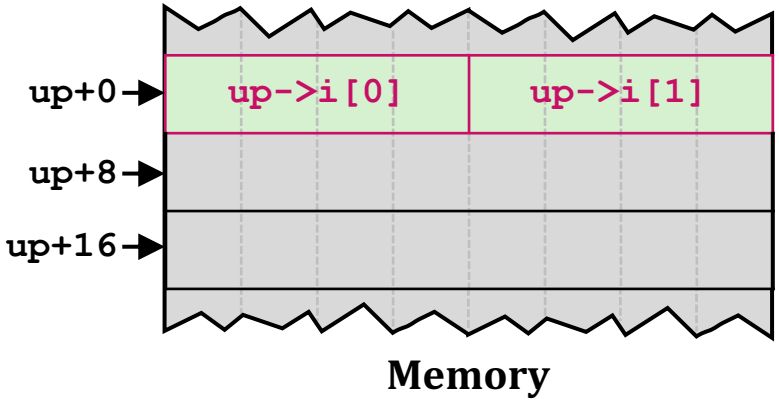
Union Allocation

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

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



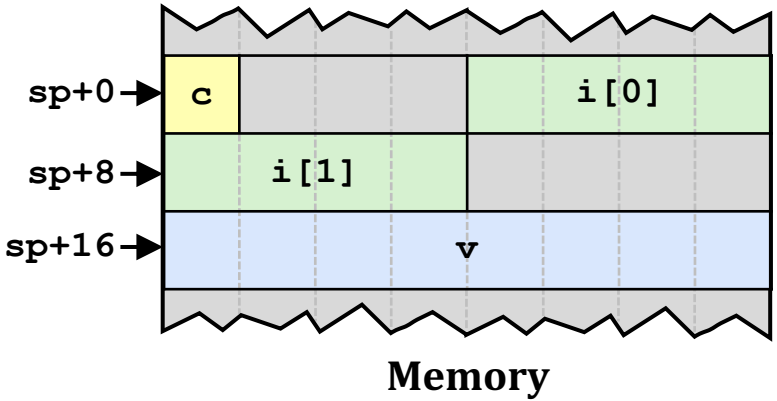

```
union U1 {
  char c;
  int i[2];
  double v;
} *up;
```



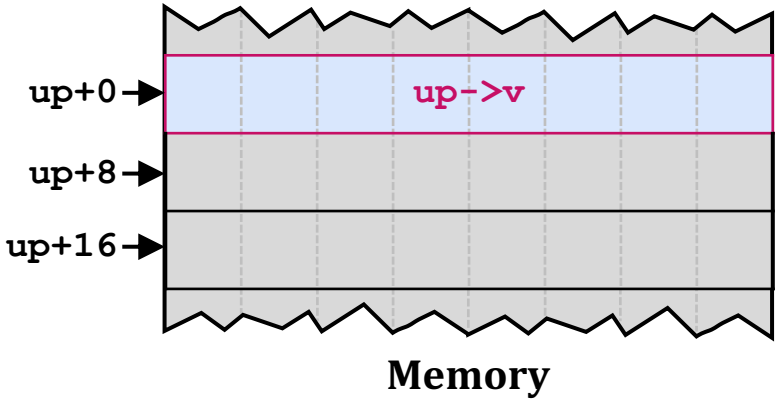
Union Allocation

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

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

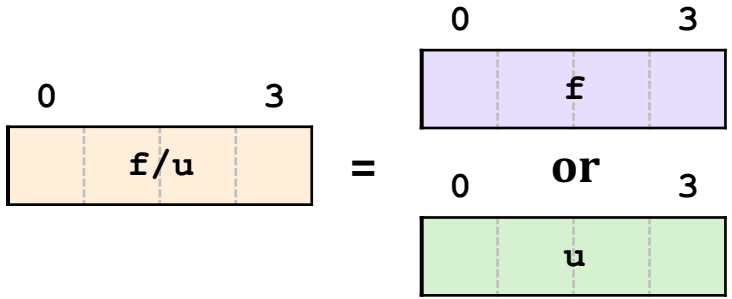



```
union U1 {
  char c;
  int i[2];
  double v;
} *up;
```



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?

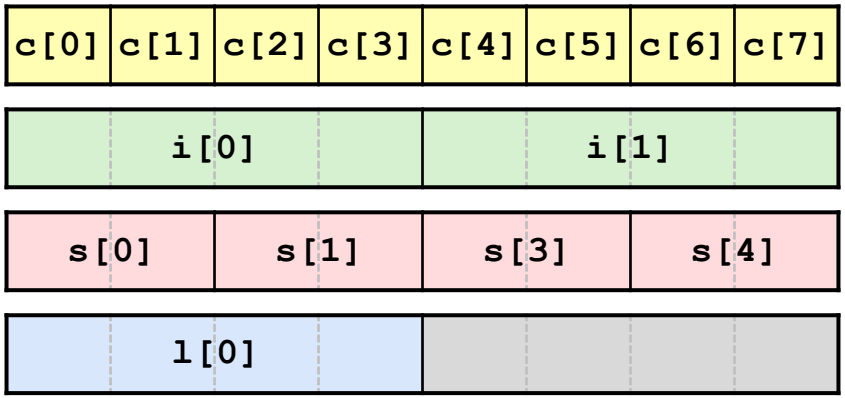
Byte Ordering Revisited

- Problem
  - `short`/`int`/`long` stored in memory as `2`/`4`/`8` consecutive bytes
  - Which is most (least) significant? – Which is the first byte: MSB or LSB?
  - Can cause problems when exchanging binary data between machines
- Big endian: the most significant byte has the lowest address
  - e.g., SPARC
- Little endian: the least significant byte has the lowest address
  - e.g., Intel x86
- Bi-endian: can be configured either way
  - e.g., ARM

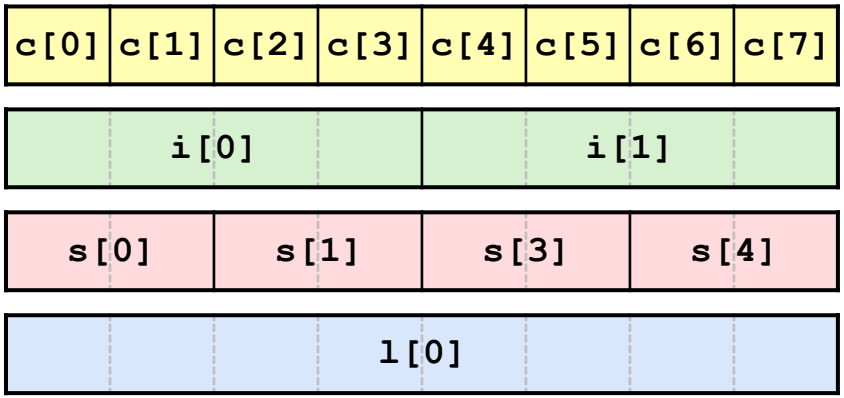
Byte Ordering Example

```
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?



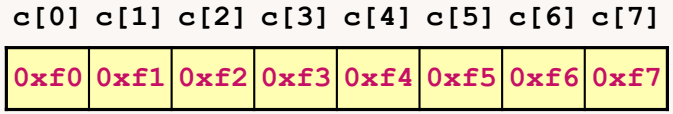
32-Bit Machines

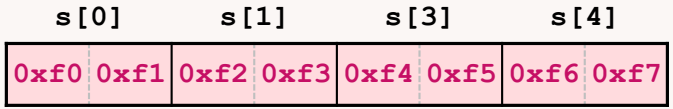


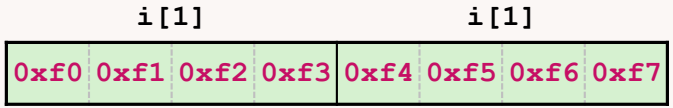
64-Bit Machines

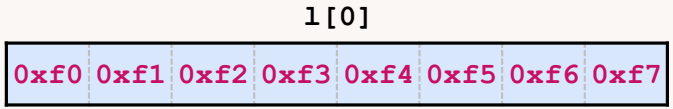
Byte Ordering Example (64-Bit Machines)

```
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]);
```



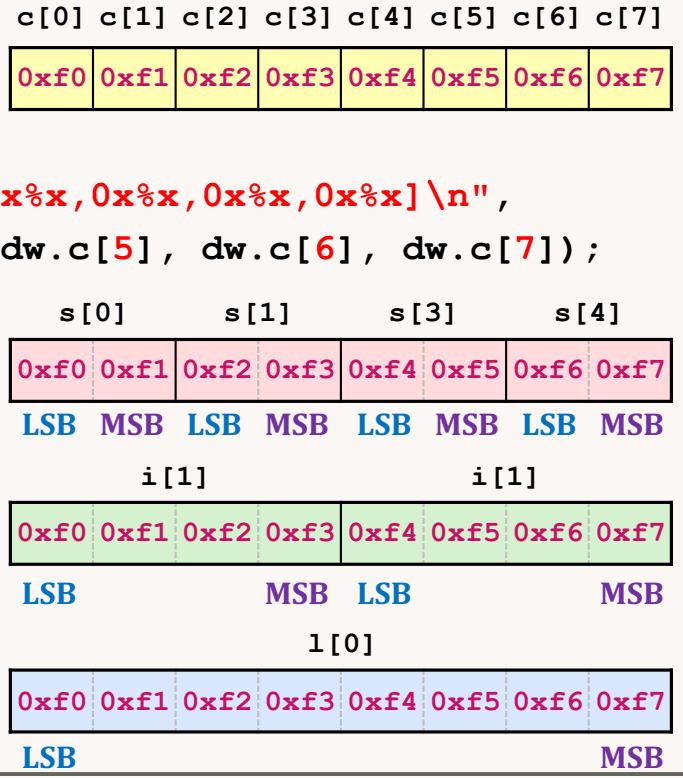






Byte Ordering Example (64-Bit, Little Endian)

```
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 Example (64-Bit, Little Endian)

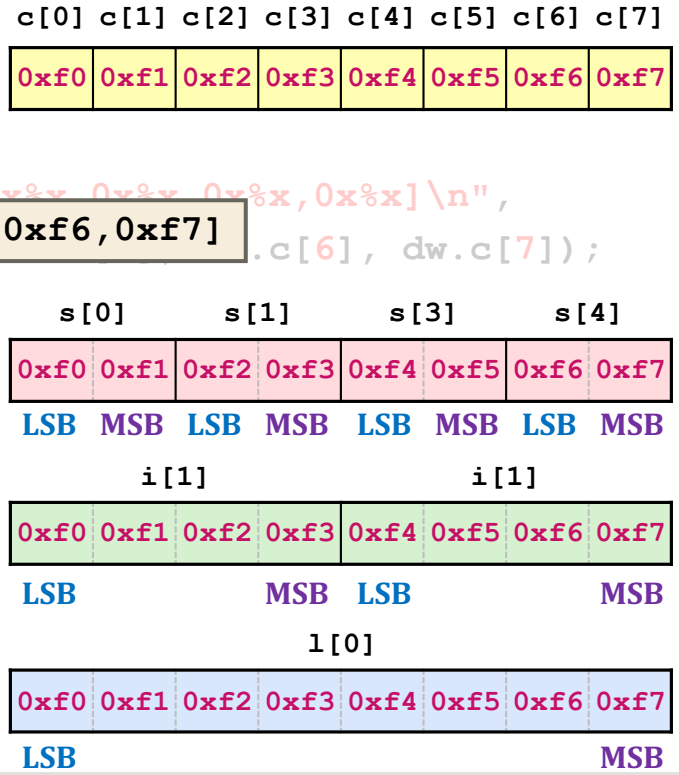
```
int j;
for (j = 0; j < 8; j++)
    dw.c[j] = 0xf0 + j;
```

```
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 Example (64-Bit, Big Endian)

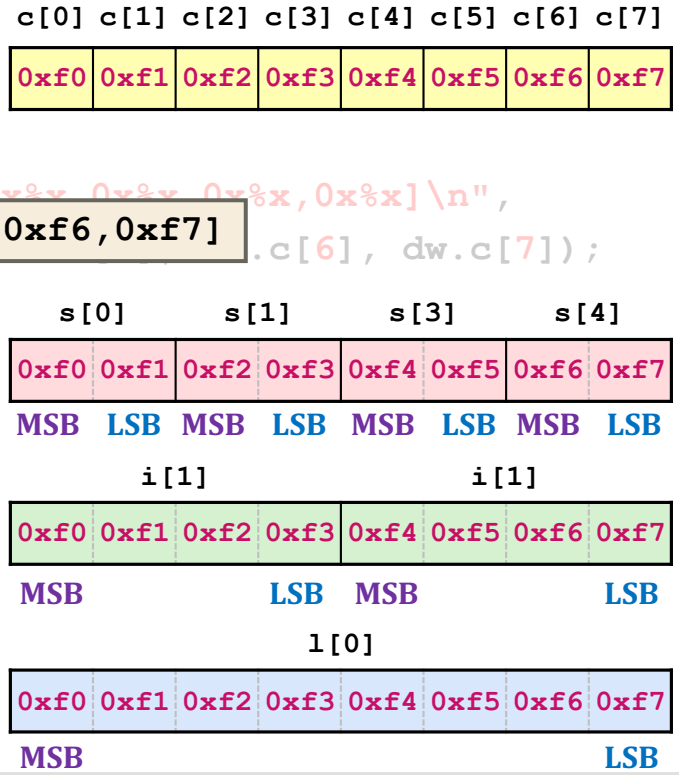
```
int j;
for (j = 0; j < 8; j++)
    dw.c[j] = 0xf0 + j;

printf("Characters 0-7 == [0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x]\n",
                                                    .c[6], dw.c[7]);
```

```
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 == [0xf0f1f2f3f4f5f6f7]
```

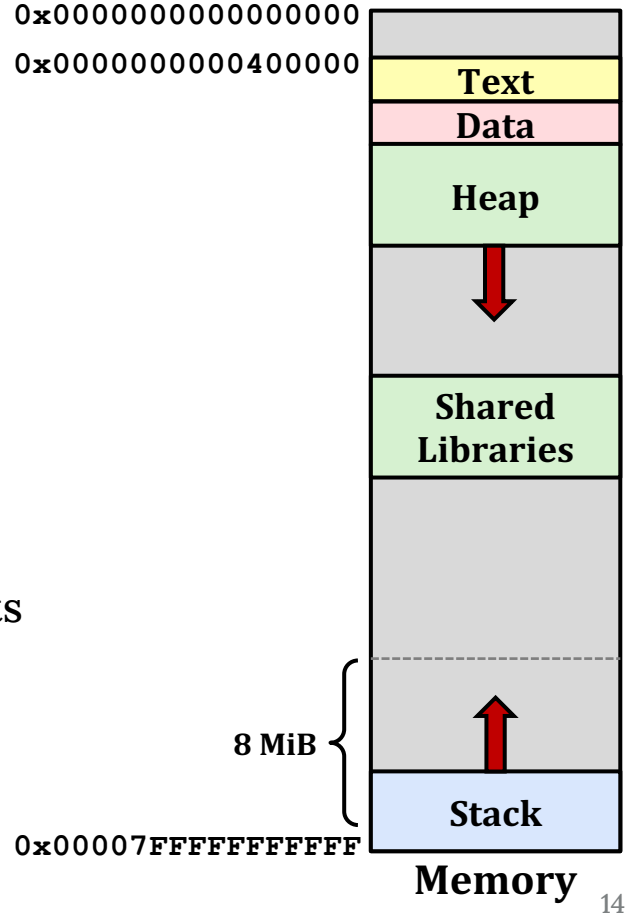


Lecture Agenda

- Unions
- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection

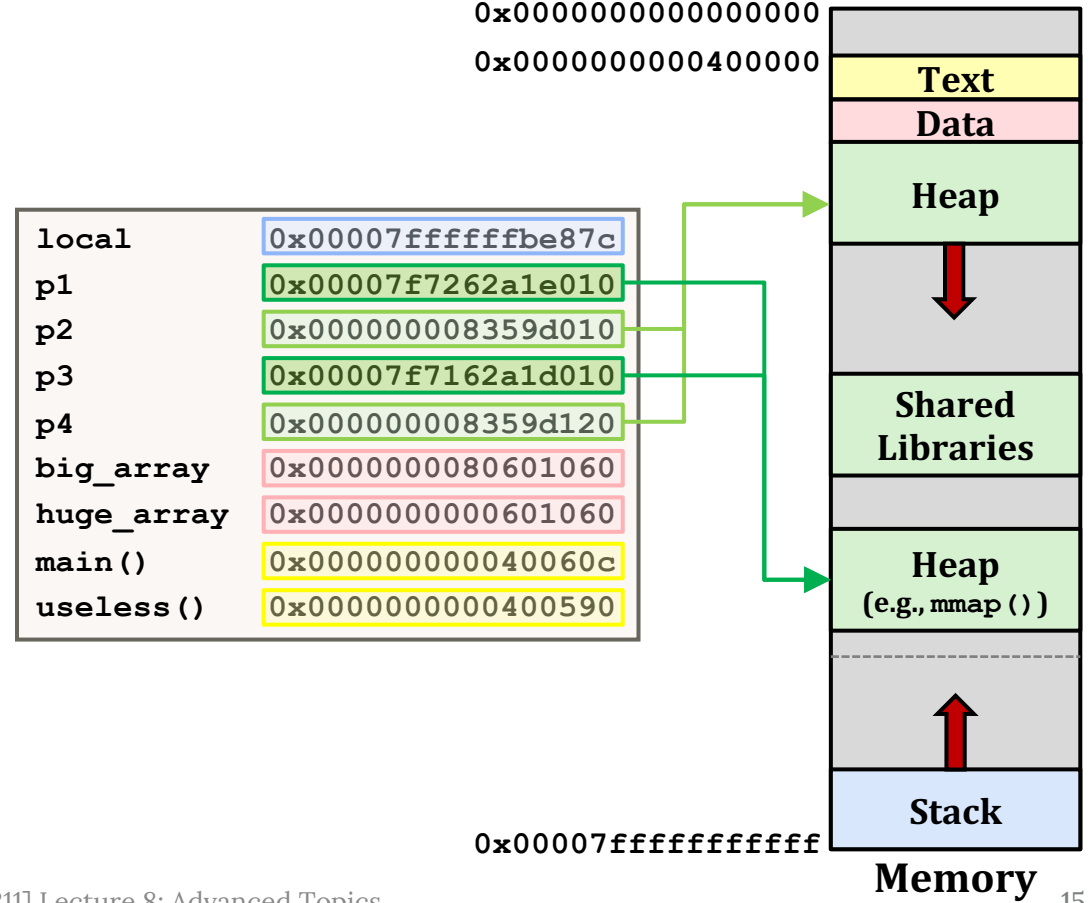
x86-64 Linux Memory Layout

- Stack
  - Runtime stack (8-MiB limit by default)
  - e.g., local variables
- Heap
  - Dynamically allocated as needed
  - e.g., `malloc()`, `calloc()`, `new()`, etc.
- Data
  - Statically allocated data
  - e.g., global variables, static variables, string constants
- Text and shared libraries
  - Executable machine instructions
  - Read-only



x86-64 Linux Memory Layout

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

int global = 0;

int useless(){ return 0; }

int main (){
    void *p1, *p2, *p3, *p4;
    int local = 0;
    p1 = malloc(1L<<28); // 256 MB
    p2 = malloc(1L<<8);  // 256  B
    p3 = malloc(1L<<32); //   4 GB
    p4 = malloc(1L<<8);  // 256  B
    // Some print statements below
     ⋮
}
```

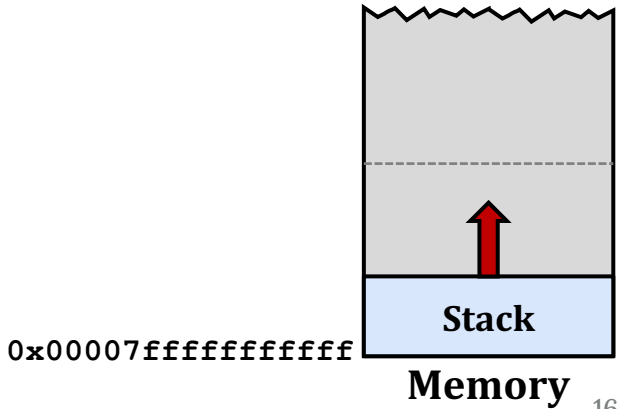


Runaway Stack Example

```
int recurse(int x) {
    int a[1<<15];  // 4*2^15 =  128 KiB
    printf("x = %d.  a at %p\n", x, a);
    a[0] = (1<<14)-1;
    a[a[0]] = x-1;
    if (a[a[0]] == 0)
        return -1;
    return recurse(a[a[0]]) - 1;
}
```

```
$ ./runaway 67
x = 67   a at 0x7ffd18aba930
x = 66   a at 0x7ffd18a9a920
x = 65   a at 0x7ffd18a7a910
...
x = 4    a at 0x7ffd182da540
x = 3    a at 0x7ffd182ba530
x = 2    a at 0x7ffd1829a520
Segmentation fault (core dumped)
```

- Functions store local data in their stack frame
- Recursive functions cause deep nesting of frames



Lecture Agenda

- Unions
- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection

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; /* 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(6)  →  Segmentation fault
```

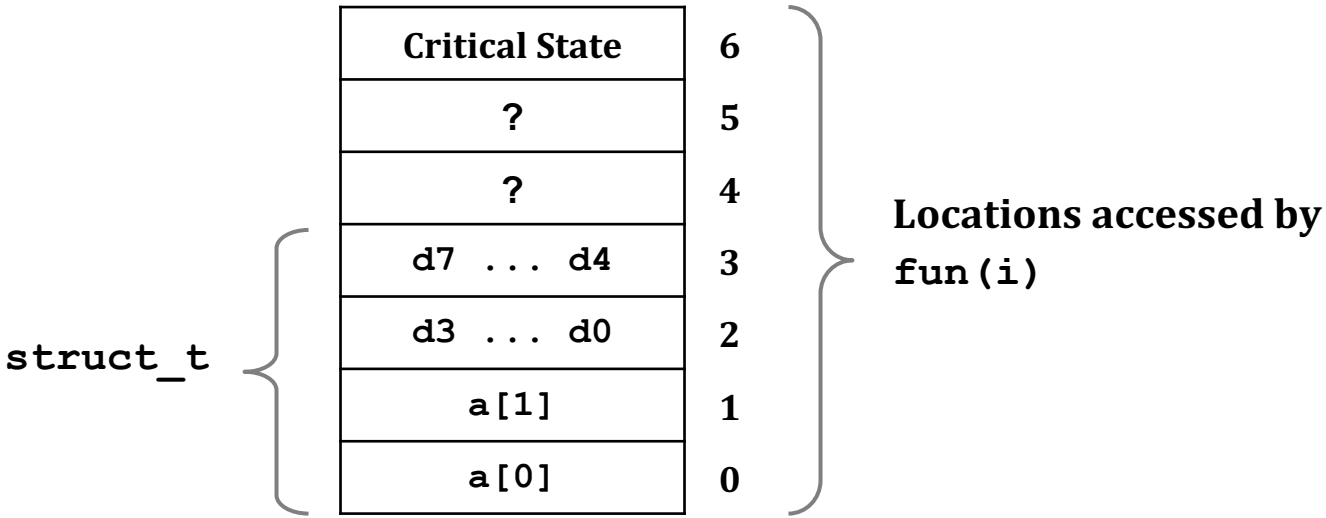
Result is system specific

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(6)  →  Segmentation fault
```

Explanation:



Such Problems Are a BIG Deal

- Generally called a buffer overflow
  - When exceeding the memory size allocated for an array

- Why a big deal?
  - It's the #1 technical cause of security vulnerabilities
    - #1 overall cause is social engineering/user ignorance

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

Vulnerable Buffer Code

- Implementation of Unix function 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;
}
```

  - Problem: no way to specify on the number of characters to read
- Similar problems with other library functions
  - `strcpy()` and `strcat()`: copy strings of arbitrary length
  - `scanf()`, `fscanf()`, and `sscanf()`: when given `%s` conversion specification

Vulnerable Buffer Code

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

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

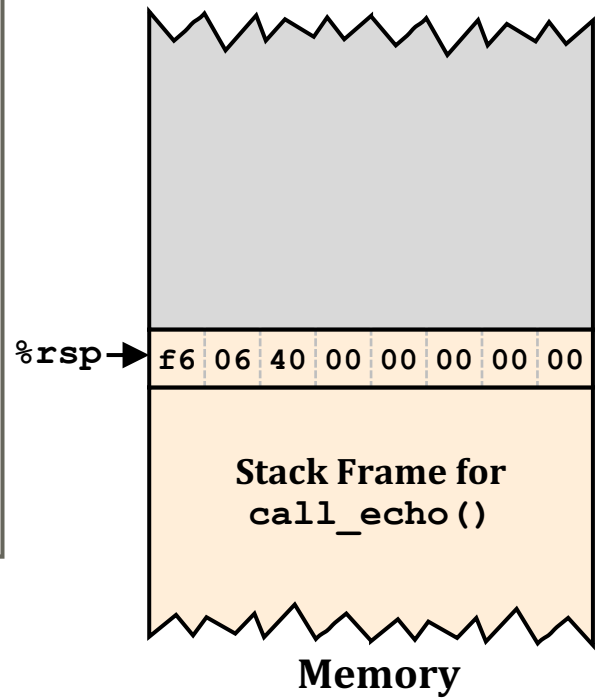
How big is big enough?

```
$ ./bufdemo
Type a string: 012345678901234567890123
012345678901234567890123
```

```
$ ./bufdemo
Type a string: 0123456789012345678901234
01234568901234567890123
Segmentation Fault
```

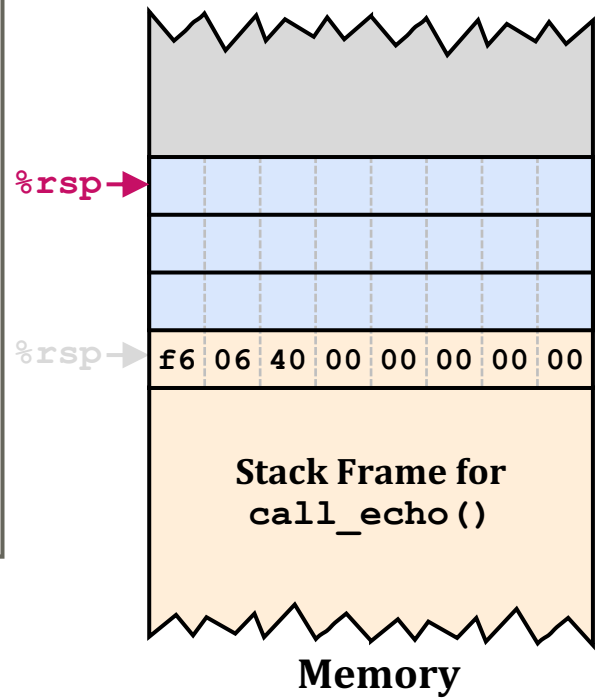
Vulnerable Buffer Code

```
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

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
```



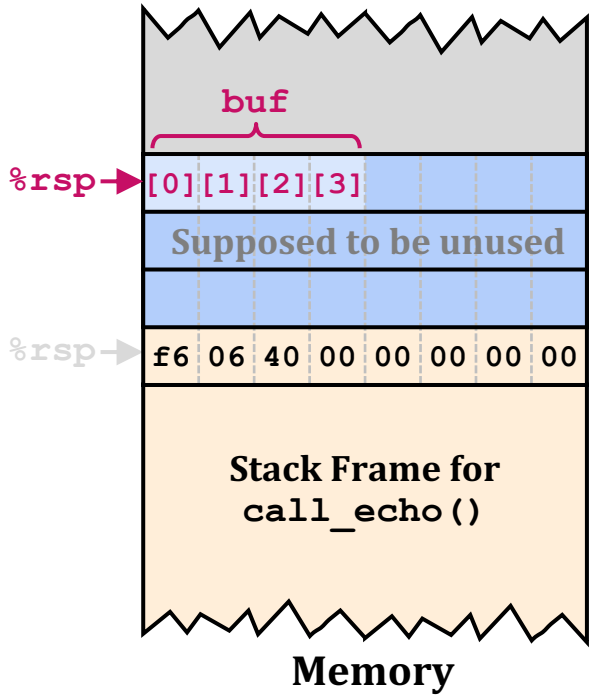
Vulnerable Buffer Code

```
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

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
```



Vulnerable Buffer Code

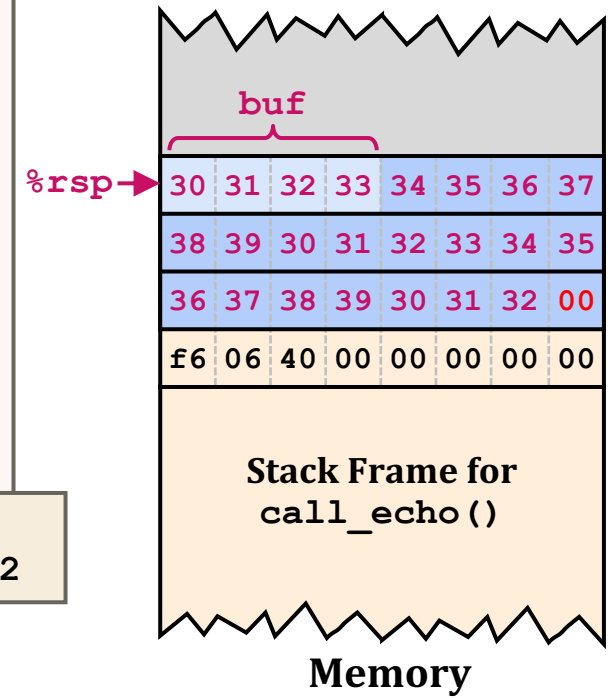
```
00000000004006cf <echo>:
 4006cf:       48 83 ec 18           sub    $24,%rsp
 4006d3:       48 89 e7              mov    %rsp,%rdi
 4006d6:       e8 a5 ff ff ff        callq  400680 <gets>
 4006db:       48 89 e7              mov    %rsp,%rdi
 4006de:       e8 3d fe ff ff        callq  400520 <puts@plt>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

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
```



Vulnerable Buffer Code

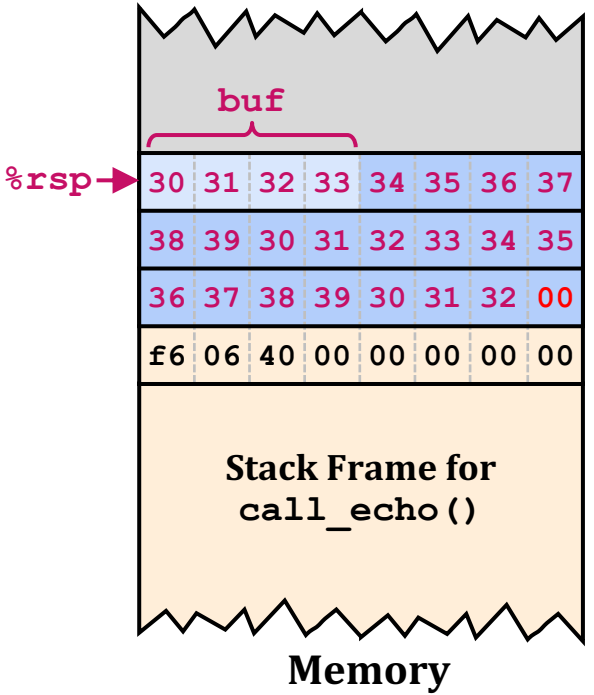
```
00000000004006cf <echo>:
  4006cf:       48 83 ec 18           sub    $24,%rsp
  4006d3:       48 89 e7              mov    %rsp,%rdi
  4006d6:       e8 a5 ff ff ff        callq  400680 <gets>
  4006db:       48 89 e7              mov    %rsp,%rdi
  4006de:       e8 3d fe ff ff        callq  400520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

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
```

```
$ ./bufdemo
Type a string: 012345678901234567890123
```



Vulnerable Buffer Code

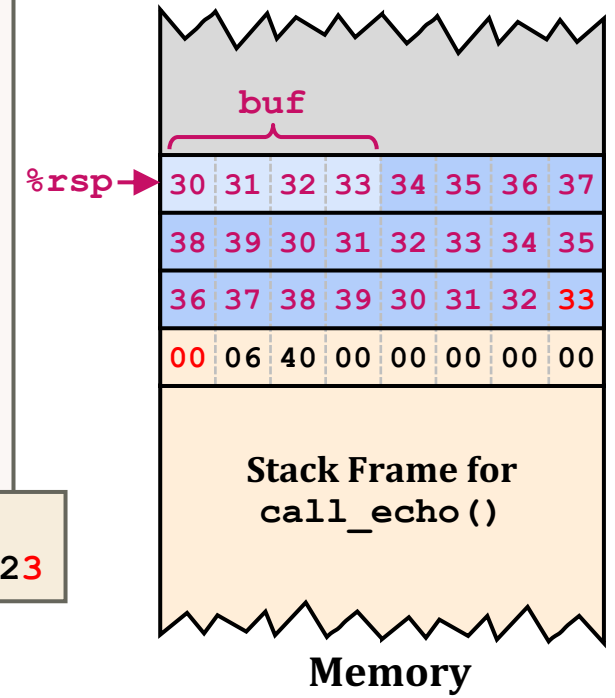
```
00000000004006cf <echo>:
 4006cf:       48 83 ec 18          sub    $24,%rsp
 4006d3:       48 89 e7             mov    %rsp,%rdi
 4006d6:       e8 a5 ff ff ff       callq  400680 <gets>
 4006db:       48 89 e7             mov    %rsp,%rdi
 4006de:       e8 3d fe ff ff       callq  400520 <puts@plt>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

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
```



Vulnerable Buffer Code

```
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

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
```

```
$ ./bufdemo
Type a string: 0123456789012345678901 23
```

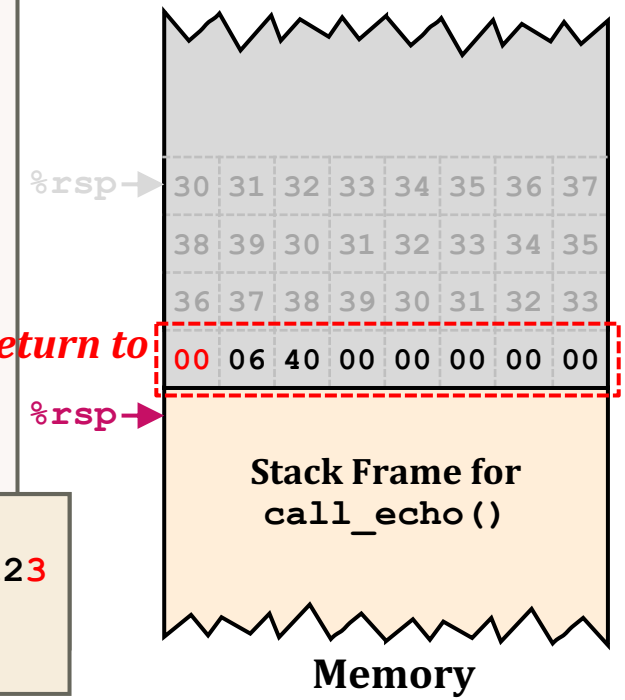


Vulnerable Buffer Code

```
00000000004006cf <echo>:
  4006cf:       48 8
  4006d3:       48 8
  4006d6:       e8 a                                    >
  4006db:       48 8
  4006de:       e8 3                                    @plt>
  4006e3:       48 8
  4006e7:       c3

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

```
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
```

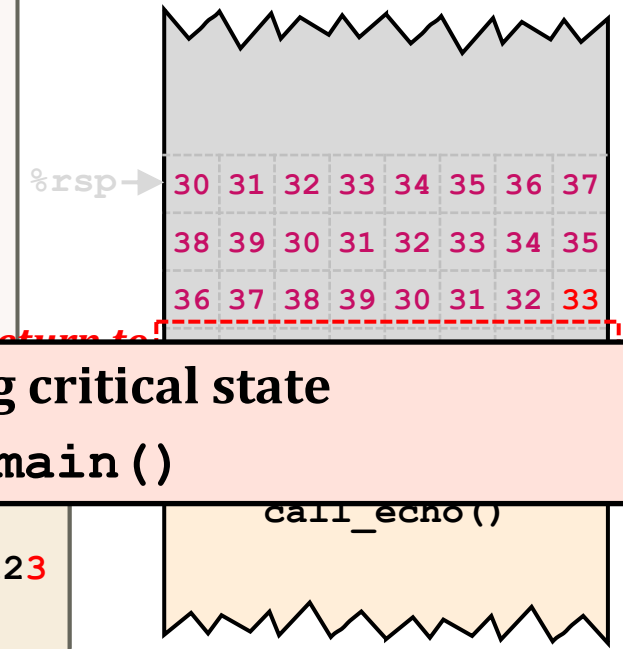
```
$ ./bufdemo
Type a string: 0123456789012345678901230123456789012345678901230
Segmentation Fault
```



Vulnerable Buffer Code

```
00000000004006cf <echo>:
 4006cf:        48 8
 4006d3:        48 8
 4006d6:        e8 a
 4006db:        48 8
 4006de:        e8 3
 4006e3:        48 8
 4006e7:        c3

00000000004006e8 <
 4006e8:        48 83
```

```
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
```



Lots of things happen, without modifying critical state

Eventually executes `retq` back to `main()`

```
$ ./bufdemo
Type a string: 0123456789012345678901230123456789012345678901230
Segmentation Fault
```

Stack Smashing Attacks

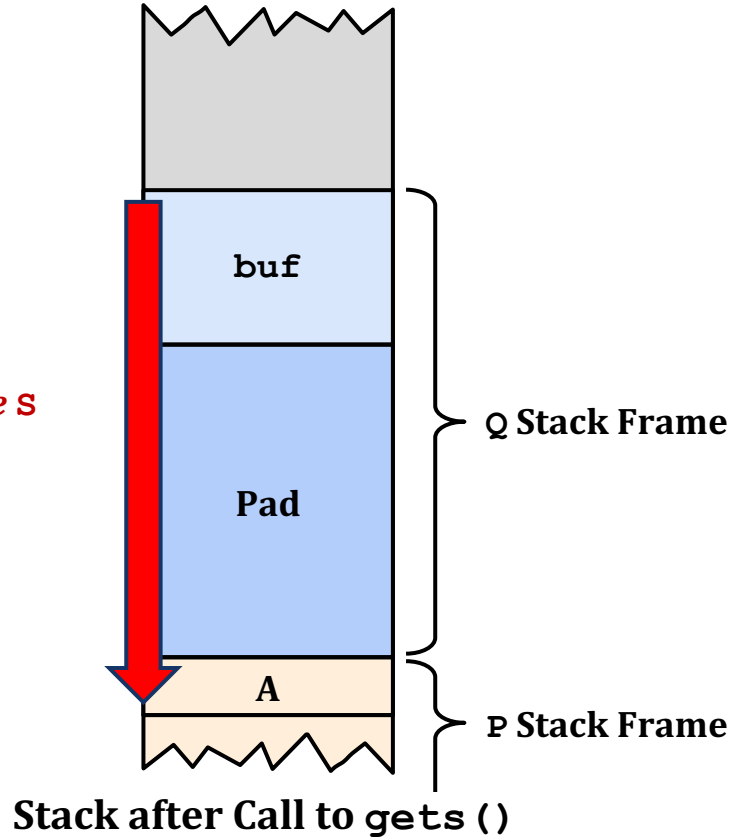
```
void P(){
    Q();
     ⋮
}
```

Return Addr. A

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

```
int S(){
/* Something
   unexpected */
     ⋮
}
```

`gets()`:
Overwrite the return address A with an address of some other code `S`



Stack after Call to `gets()`

Stack Smashing Attacks

```
void P(){
    Q();
     ⋮
}
```

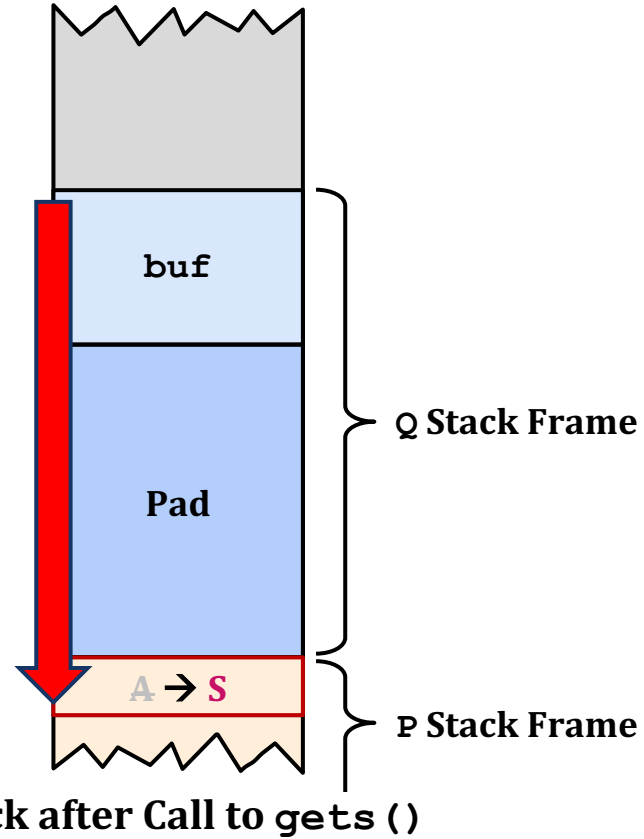
Return Addr. A

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

```
int S(){
/* Something
   unexpected */
     ⋮
}
```

`gets()`:
Overwrite the return address A with an address of some other code `S`

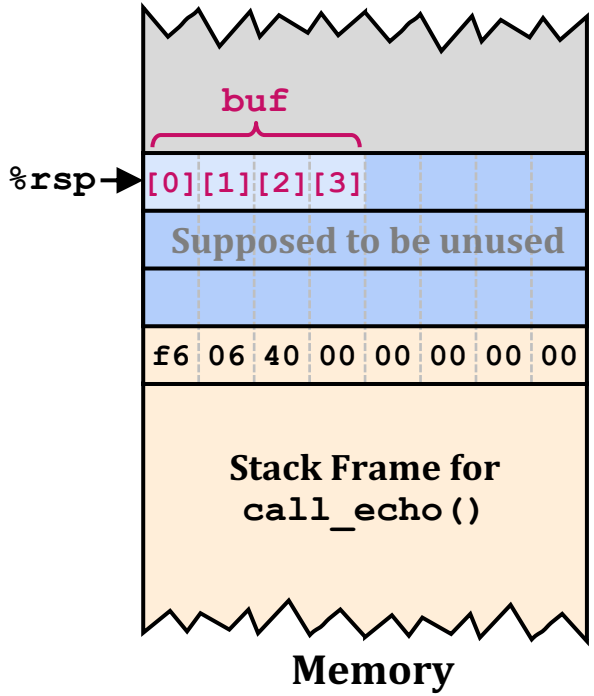
When `Q` *executes* `ret`, *will jump to the other code* `S`



Stack after Call to `gets()`

Crafting Smashing String

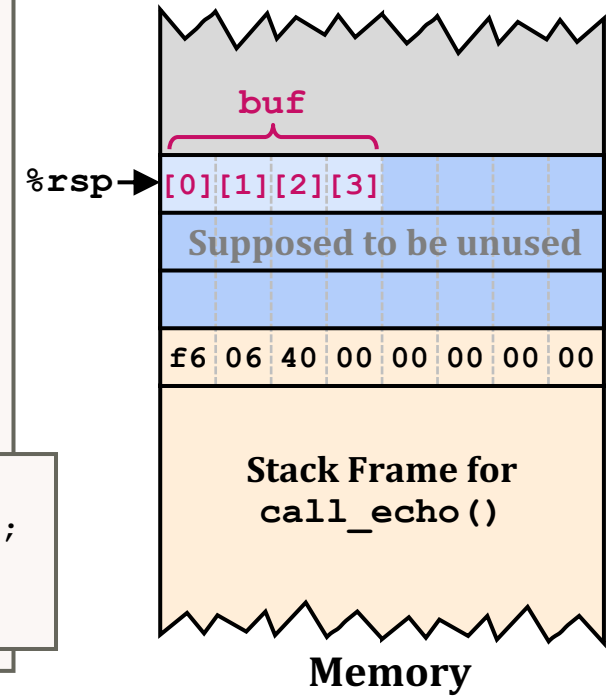
```
00000000004006cf <echo>:
 4006cf:       48 83 ec 18           sub    $24,%rsp
 4006d3:       48 89 e7              mov    %rsp,%rdi
 4006d6:       e8 a5 ff ff ff        callq  400680 <gets>
 4006db:       48 89 e7              mov    %rsp,%rdi
 4006de:       e8 3d fe ff ff        callq  400520 <puts@plt>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

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>
   ⋮

00000000004006fb <smash>:
 4006fb:       48 83 ec 08
   ⋮
```



Crafting Smashing String

```
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

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>
   ⋮

00000000004006fb <smash>:
 4006fb:       48 83 ec 08
   ⋮
```

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



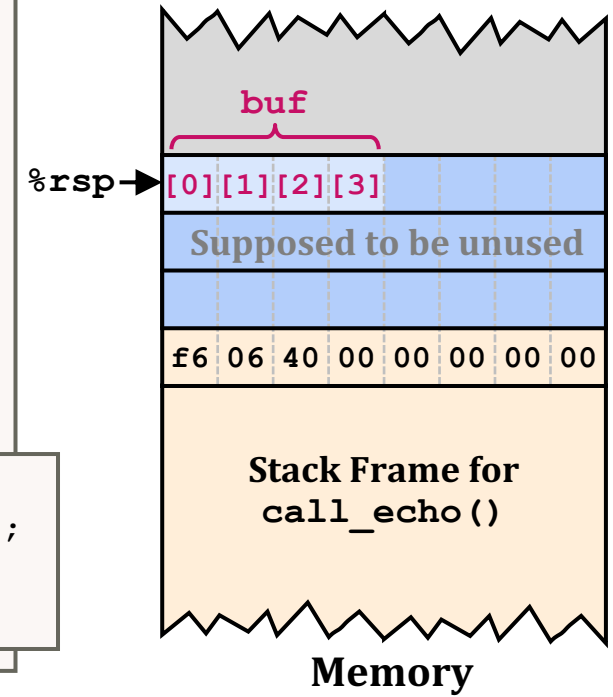
Crafting Smashing String

```
00000000004006cf <echo>:
  4006cf:       48 83 ec 18
  4006d3:       48 89 e7
  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

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>
    ⋮

00000000004006fb <smash>:
  4006fb:       48 83 ec 08
    ⋮
```

Attack String:

```
ffffffff ffffffff ffffffff ffffffff
ffffffff ffffffff fb064000 00000000
```

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



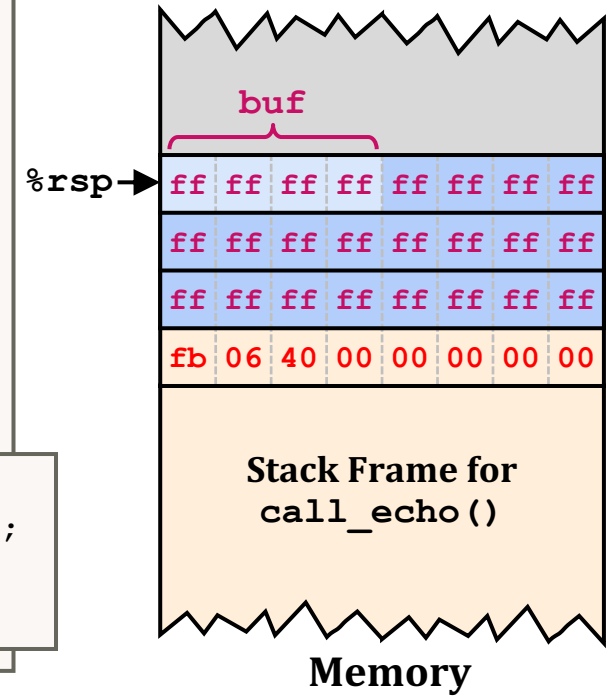
Crafting Smashing String

```
00000000004006cf <echo>:
  4006cf:       48 83 ec 18
  4006d3:       48 89 e7
  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

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>
    ⋮

00000000004006fb <smash>:
  4006fb:       48 83 ec 08
    ⋮
```

Attack String:

```
ffffffff ffffffff ffffffff ffffffff
ffffffff ffffffff fb064000 00000000
```

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



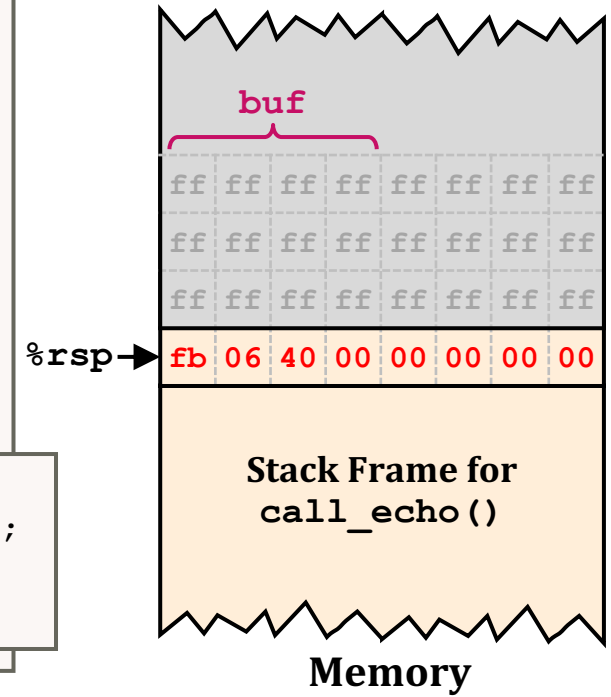
Crafting Smashing String

```
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

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>
    ⋮

00000000004006fb <smash>:
  4006fb:    48 83 ec 08
    ⋮
```

Attack String:

```
ffffffff ffffffff ffffffff ffffffff
ffffffff ffffffff fb064000 00000000
```

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



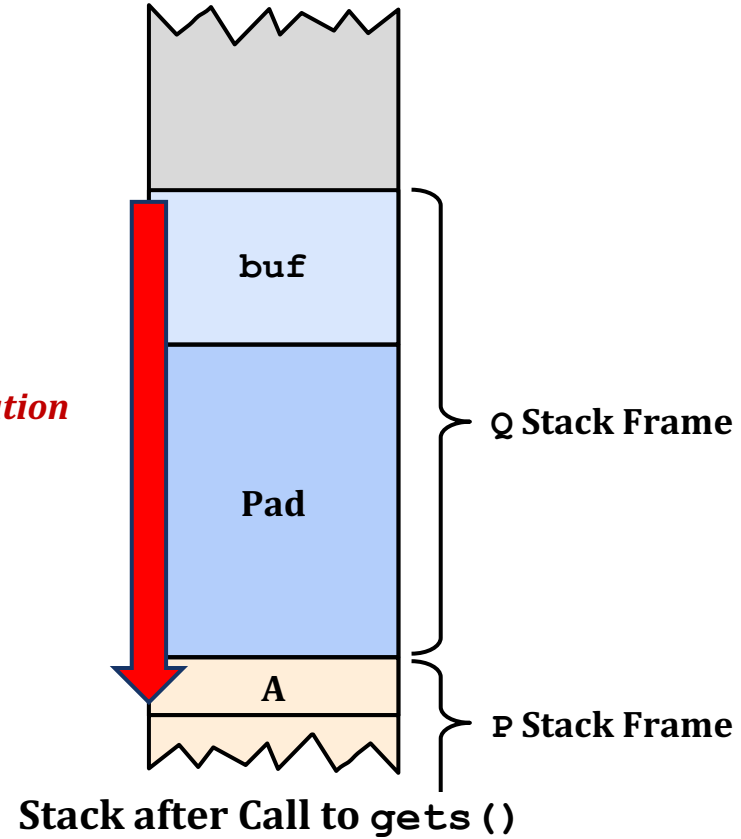
Code Injection Attacks

```
void P(){
    Q();
     ⋮
}
```

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

Return Addr. A

gets():
Overwrite the return address A with the address of buffer (B) + fill the buffer with byte representation of executable code



Stack after Call to gets()

Code Injection Attacks

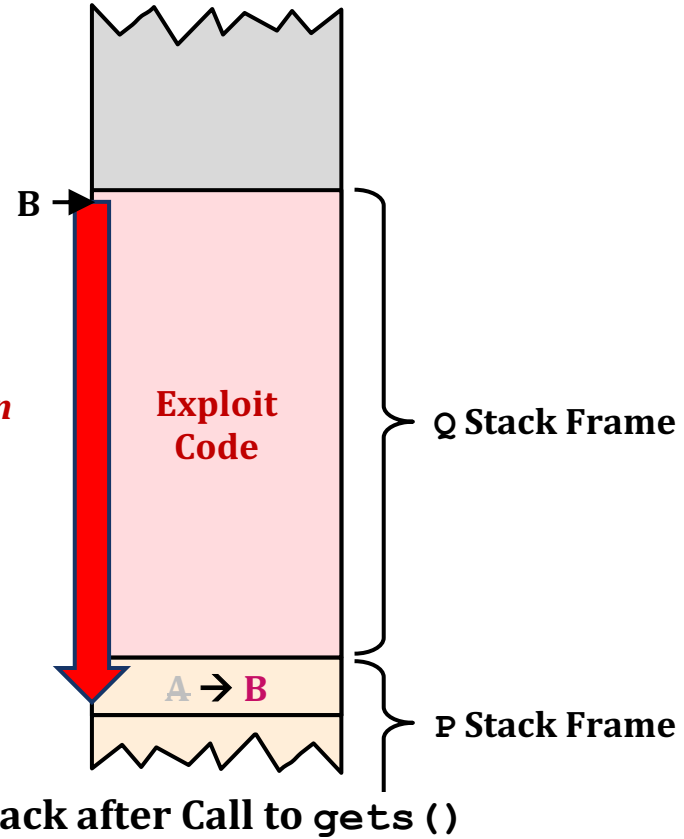
```
void P(){
    Q();
     ⋮
}
```

Return Addr. A

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

`gets()`:
Overwrite the return address A with the address of buffer (B) + fill the buffer with byte representation of executable code

When `Q` executes `ret`, will jump to the exploit code



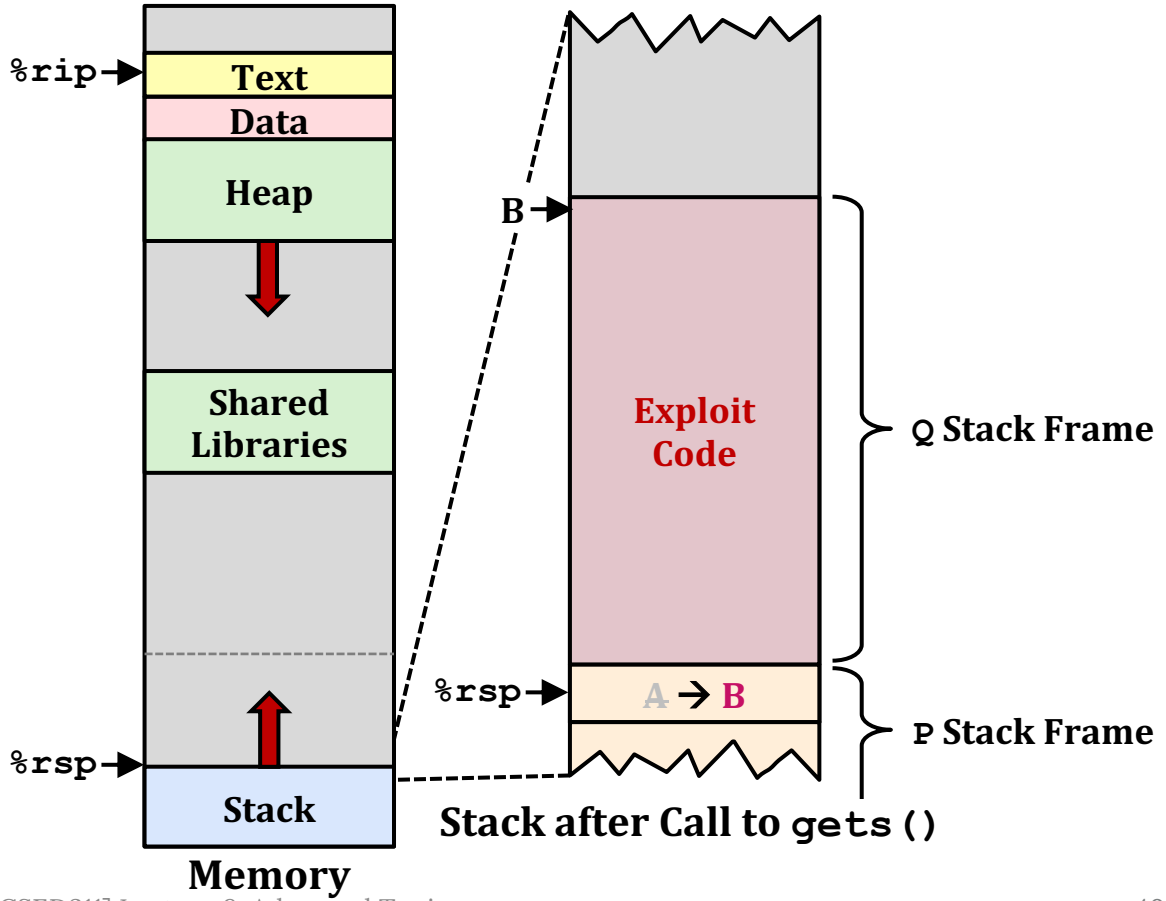
Stack after Call to `gets()`

Code Injection Attacks

```
void P(){
    Q();
     ⋮
}
```

Return Addr. A

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

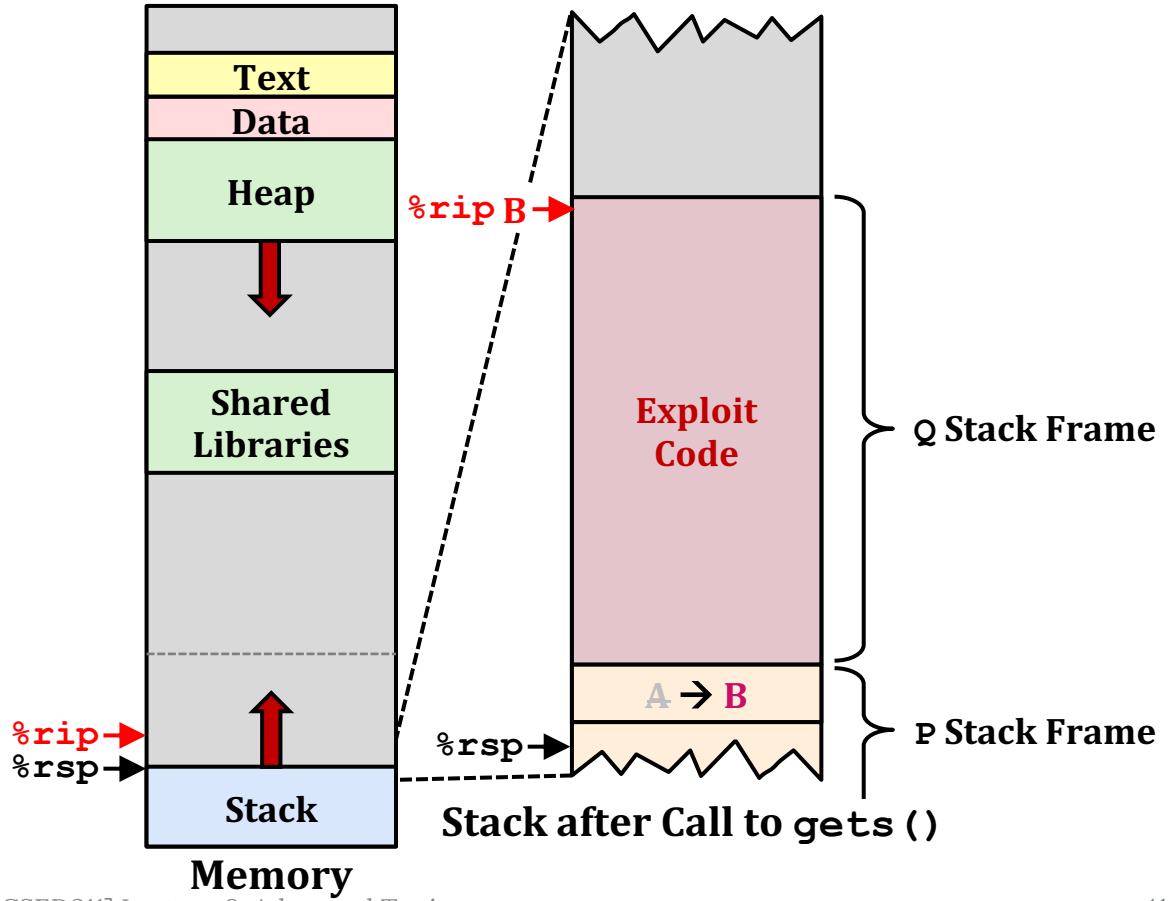


Code Injection Attacks

```
void P(){
    Q();
     ⋮
}
```

Return Addr. A

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



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: Internet Worm (1988), IM Wars (1999), Twilight hack on Wii (2000s), and many, many more
- You will learn some of the tricks in Lab 4. Attack Lab
  - Hopefully to convince you to never leave such security holes in your program

Counter Measures Against Buffer Overflow Attacks

- Avoid overflow vulnerabilities
- Employ system-level protections
- Have compiler use stack canaries

1. Avoid Overflow Vulnerabilities in Code

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

- Example: use library routines that limit string lengths
  - `fgets` instead of `gets`
    - Do not use `scanf` with `%s` conversion specification;
      Use `fgets` to read the string or `%ns` where ***n*** is a suitable integer
  - `strncpy` instead of `strcpy`

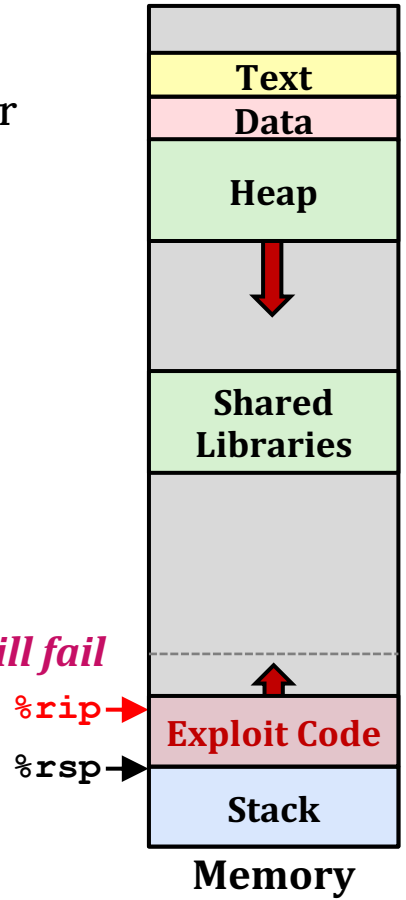
2. System-Level Protections

- Randomized stack offsets: ASLR (Address Space Layout Radomization)
  - At start of program, allocate a random amount of space on stack
  - Shift stack addresses for the entire program
  - Make it difficult for adversaries to predict the beginning of the inserted code
  - e.g., 5 executions of memory allocation code: stack repositioned each time program executes

```
Local
0x7ffe4d3be87c
0x7fff75a4f9fc
0x7ffeadb7c80c
0x7ffeaea2fdac
0x7ffcd452017c
```

2. System-Level Protections (Cont.)

- Nonexecutable code segments
  - In traditional x86, a region of memory can be marked as either read-only or writeable
    - Can execute anything readable
  - x86-64 added explicit executable permission
    - Stack marked as non-executable



OS ensures (cooperating CPU):
any attempt to execute an instruction in non-executable region will fail

3. Compiler-Level Protection: Stack Canaries

- Key idea
  - Place a special value, called carnary, on stack just beyond buffer
  - Check the value before exiting the function
    - The value is change → Something wrong must have happened

- GCC implementation
  - `-fstack-protector`
  - Now the default (used to be disabled earlier)

```
$ ./bufdemo-protected
Type a string: 0123456
0123456
```

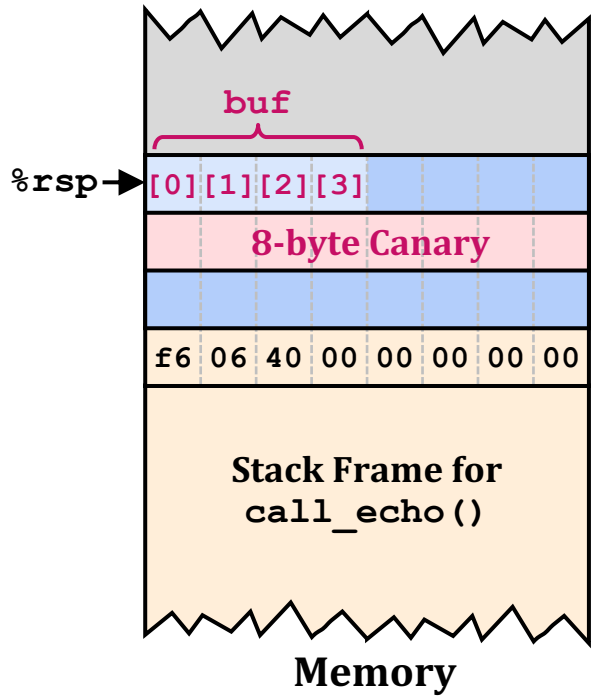
```
$ ./bufdemo-protected
Type a string: 01234567
01234567
*** stack smashing detected ***
```

3. Stack Canaries: Protected Buffer Disassembly

```
00000000004006cf <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
```

3. Stack Canaries: Setting Up Canary

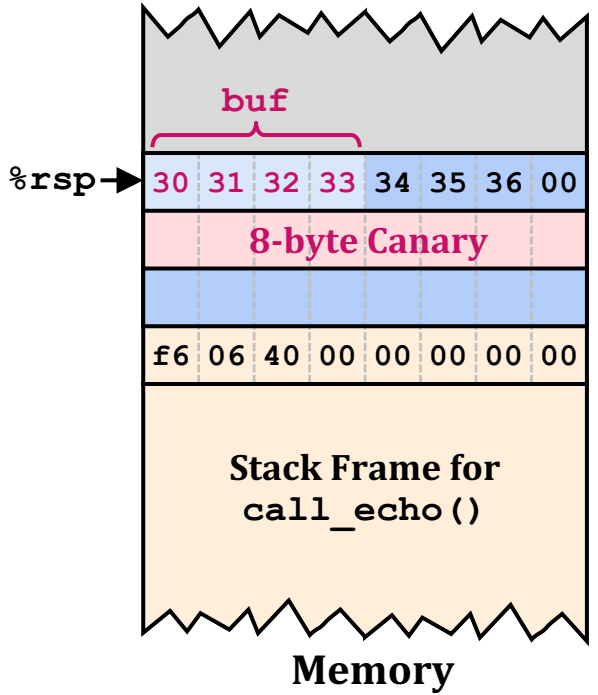
```
00000000004006cf <echo>:
 40072f:       sub    $0x18,%rsp
 400733:       mov    %fs:0x28,%rax  # Get canary
 40073c:       mov    %rax,0x8(%rsp) # Place on Stack
 400741:       xor    %eax,%eax      # Delete canary
 400743:       mov    %rsp,%rdi
 400746:       callq  4006e0 <gets>
 40074b:       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
```



3. Stack Canaries: Checking Canary

```
00000000004006cf <echo>:
  40072f:       sub    $0x18,%rsp
  400733:       mov    %fs:0x28,%rax  # Get canary
  40073c:       mov    %rax,0x8(%rsp) # Place on Stack
  400741:       xor    %eax,%eax      # Delete canary
  400743:       mov    %rsp,%rdi
  400746:       callq  4006e0 <gets>
  40074b:       mov    %rsp,%rdi
  40074e:       callq  400570 <puts@plt>
  400753:       mov    0x8(%rsp),%rax # Retrieve from stack
  400758:       xor    %fs:0x28,%rax  # Compare to canary
  400761:       je     400768 <echo+0x39> # If same, OK
  400763:       callq  400580 <__stack_chk_fail@plt> # Fail
  400768:       add    $0x18,%rsp
  40076c:       retq
```

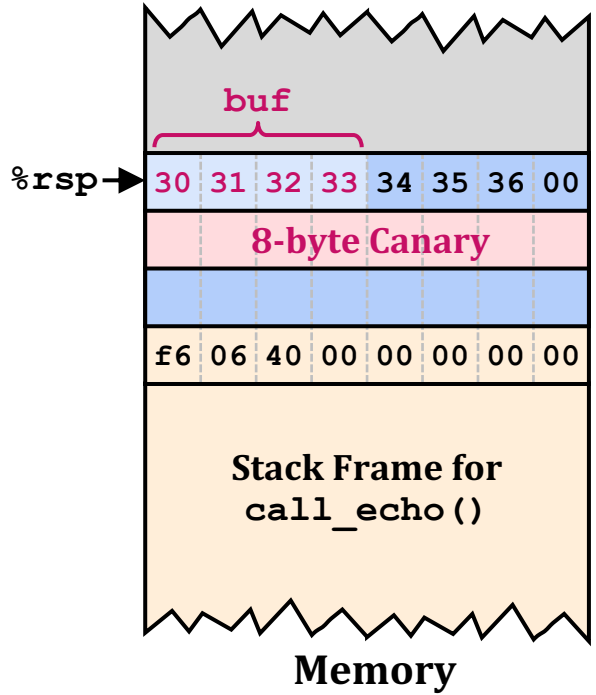
```
$ ./bufdemo-protected
Type a string: 0123456
```



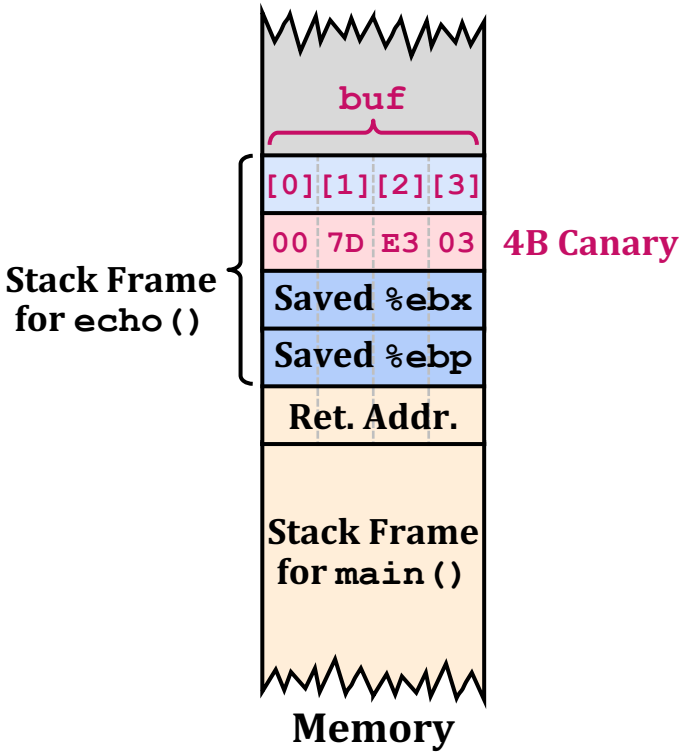
3. Stack Canaries: Checking Canary

```
00000000004006cf <echo>:
  40072f:       sub    $0x18,%rsp
  400733:       mov    %fs:0x28,%rax  # Get canary
  40073c:       mov    %rax,0x8(%rsp) # Place on Stack
  400741:       xor    %eax,%eax      # Delete canary
  400743:       mov    %rsp,%rdi
  400746:       callq  4006e0 <gets>
  40074b:       mov    %rsp,%rdi
  40074e:       callq  400570 <puts@plt>
  400753:       mov    0x8(%rsp),%rax # Retrieve from stack
  400758:       xor    %fs:0x28,%rax  # Compare to canary
  400761:       je     400768 <echo+0x39> # If same, OK
  400763:       callq  400580 <__stack_chk_fail@plt> # Fail
  400768:       add    $0x18,%rsp
  40076c:       retq
```

```
$ ./bufdemo-protected
Type a string: 0123456
0123456
```

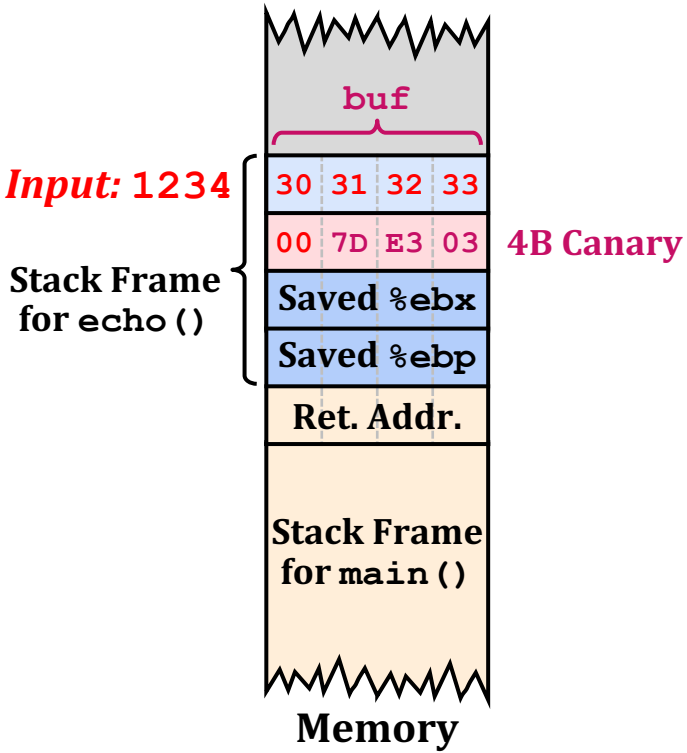


3. Stack Canaries: Example




```
(gdb) break echo
(gdb) run
(gdb) stepi 3
(gdb) print /x *((unsigned *) $ebp - 2)
$1 = 0x3e37d00
```

3. Stack Canaries: Example




```
(gdb) break echo
(gdb) run
(gdb) stepi 3
(gdb) print /x *((unsigned *) $ebp - 2)
$1 = 0x3e37d00
```

Benign corruption (tolerated)!:

Allows a programmer to make silent off-by-one errors

Return-Oriented Programming Attacks

- Challenges (for hackers)
  - Stack randomization makes it hard to predict buffer location
  - Marking stack nonexecutable makes it hard to insert 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

- Construct program from gadgets
  - Sequence of instructions ending in `ret` that is 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 the tail end of existing functions

Gadget Example#2

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

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

Encodes `movq %rax, %rdi`

`%rdi ← %rax`

Gadget Address = `0x4004dc`

- Use the tail end of existing functions

Gadget Example#2

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

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

Encodes `movq %rax, %rdi`

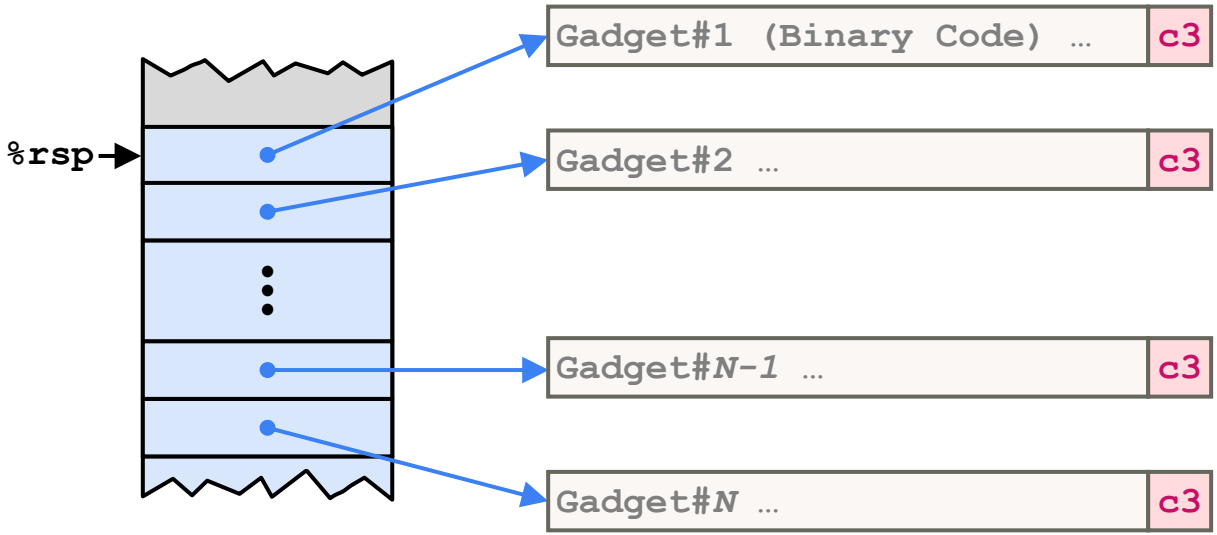
`%rdi ← %rax`

Gadget Address = `0x4004dc`

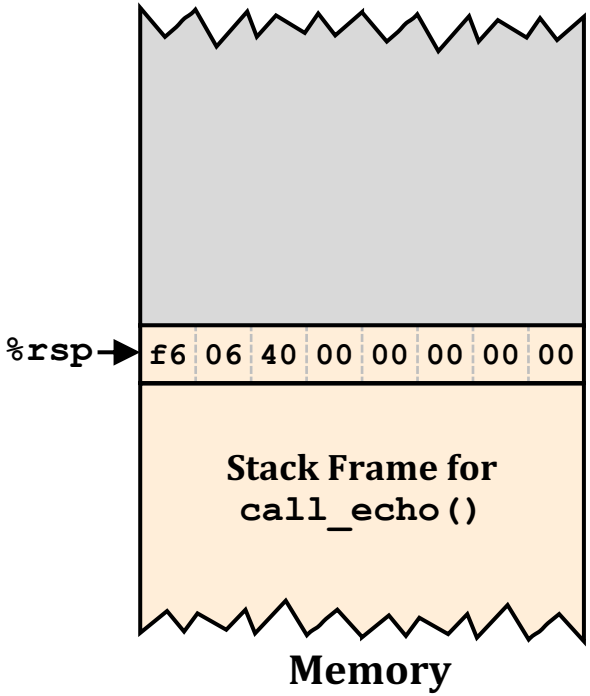
- Use the tail end of existing functions

ROP Execution

- Trigger with ret instruction
  - Will start executing Gadget 1
  - `ret` → `pop %rip`
- Final ret in each gadget will start next one



Crafting an ROP Attack String



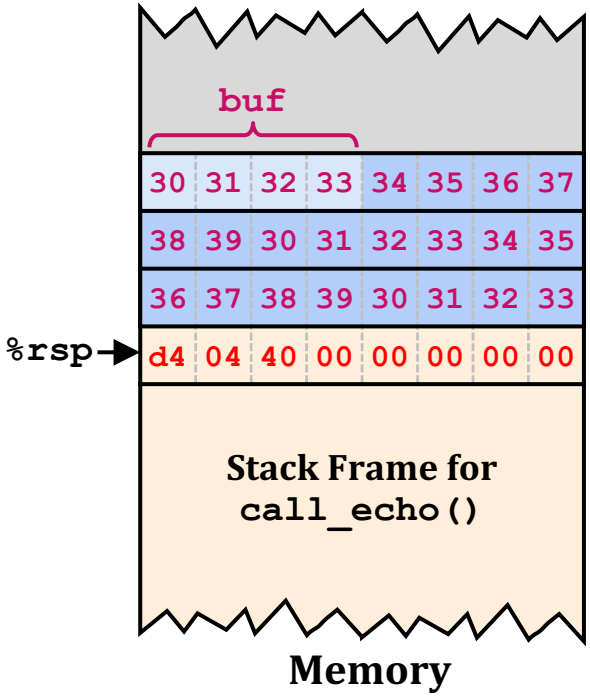

```
int echo(){
    char buf[4];
    gets(buf);
     ⋮
    return ret
}
```

Attack: Makes `echo()` return `%rdi+%rdx`

Gadget: `%rax` ← `%rdi` + `%rdx` (+ `ret`)

```
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
```

Crafting an ROP Attack String




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

Attack: Makes `echo()` return `%rdi+%rdx`

Gadget: `%rax` ← `%rdi` + `%rdx` (+ `ret`)

```
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
```

Attack String (HEX)

```
30 31 ... 38 39 30 ... 39 30 ... 33 d4 04 40 00 00 00 00 00
```

Multiple gadgets can corrupt stack upwards

[CSED211] Introduction to Computer Software Systems

Lecture 8: Advanced Topics

Prof. Jisung Park

2023.10.16