Buffer Overflows

CSE 351 Spring 2024

Instructor:

Elba Garza

Teaching Assistants:

Ellis Haker
Adithi Raghavan
Aman Mohammed
Brenden Page
Celestine Buendia
Chloe Fong
Claire Wang
Hamsa Shankar
Maggie Jiang
Malak Zaki
Naama Amiel
Nikolas McNamee
Shananda Dokka
Stephen Ying
Will Robertson

Playlist: CSE 351 24Sp Lecture Tunes!

Relevant Course Information

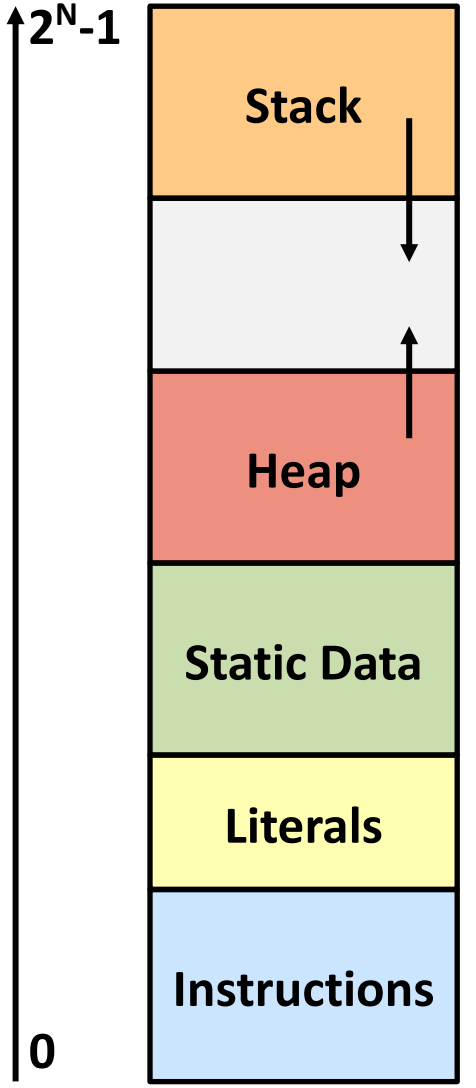
- Lab 2 due tonight & Lab 3 releasing today, due May 8th
  - You will have everything you need for it by the end of this lecture!
- HW13/14 due May 1st, HW15 due May 3rd
- Mid-Quarter Assessment results & write-up coming soon!
- Canvas Mid-Quarter Survey releasing on May 1st
  - Part of EPA grade
  - Particularly focusing on TA feedback
  - Due May 6th

Buffer Overflows

- Address space layout review
- Input buffers on the stack
- Overflowing buffers and injecting code
- Defenses against buffer overflows

Review: General Memory Layout

- Stack
  - Local variables (procedure context)
- Heap
  - Dynamically allocated as needed
  - `new`, `malloc()`, `calloc()`, …
- Statically-allocated Data
  - Read/write: global variables (Static Data)
  - Read-only: string literals (Literals)
- Code/Instructions
  - Executable machine instructions
  - Read-only



not drawn to scale

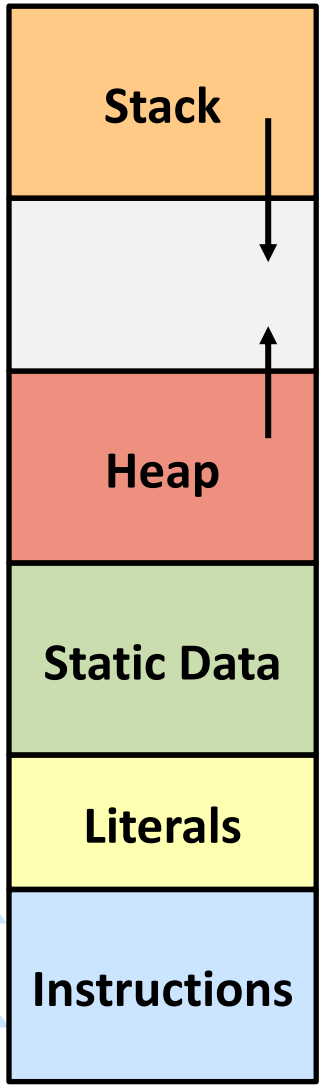
Memory Allocation Example

```
char big_array[1L<<24];  /* 16 MB */

int global = 0;

int useless() { return 0; }

int main() {
  void *p1, *p2;
  int local = 0;
  p1 = malloc(1L << 28);  /* 256 MB */
  p2 = malloc(1L << 8);   /* 256 B */
  /* Some print statements ... */
}
```



not drawn to scale!

Where does everything go?

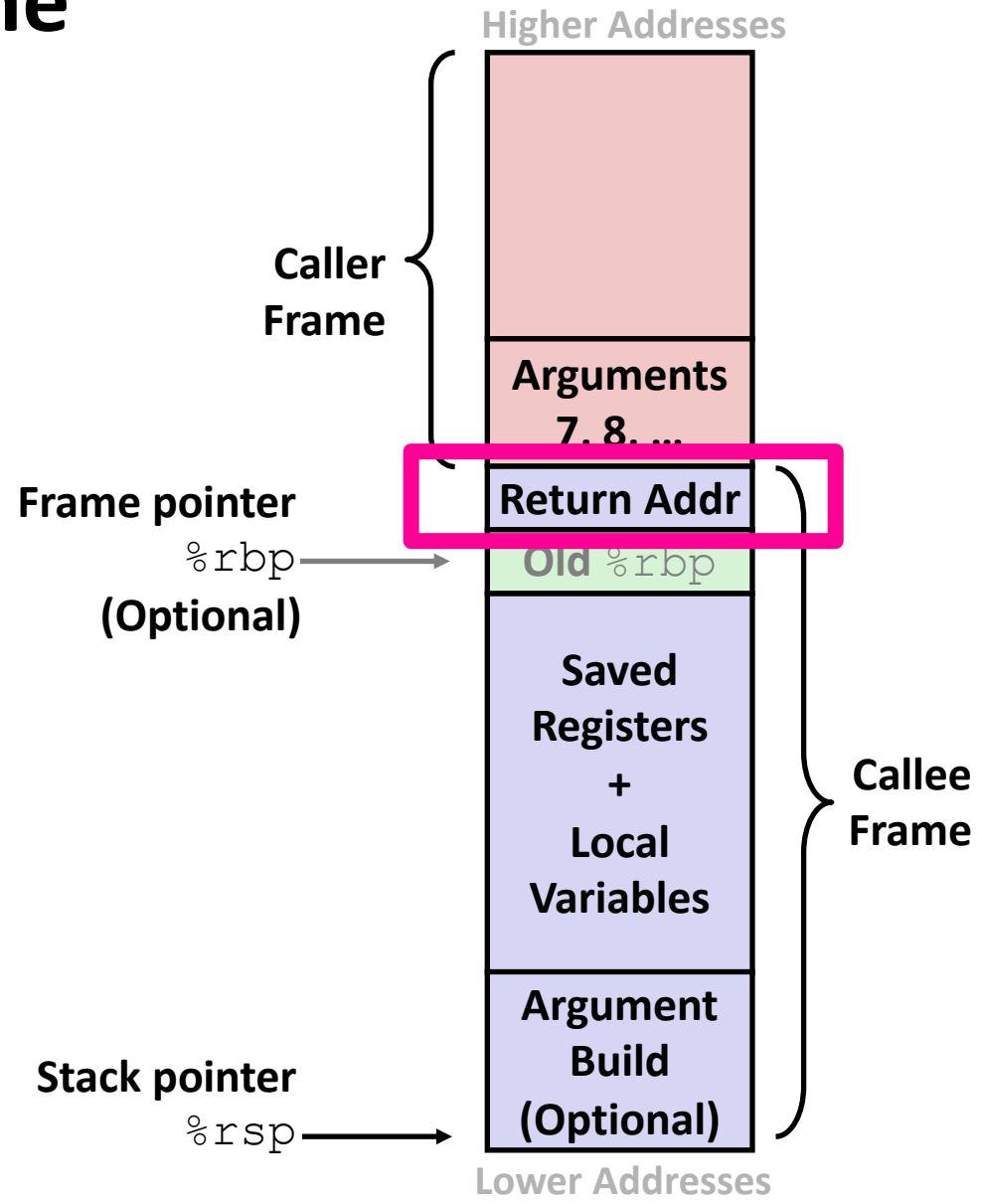
What Is a Buffer?

- A **buffer** is an array used to temporarily store data
- You've probably seen "video buffering…"
  - The video is being written into a buffer before being played

- Buffers can also be used to store user input… 🤔

Reminder: x86-64/Linux Stack Frame

- Caller's Stack Frame
  - Arguments (if > 6 args) for this call
- Current/Callee Stack Frame
  - Return address, pushed by `call` instruction
  - Old frame pointer (optional)
  - Caller-saved registers pushed before setting up arguments for a function call
  - Callee-saved registers pushed before using long-term registers
  - Local variables, if can't be kept in registers
  - "Argument build" area—Need to call a function with >6 arguments? Put them here!

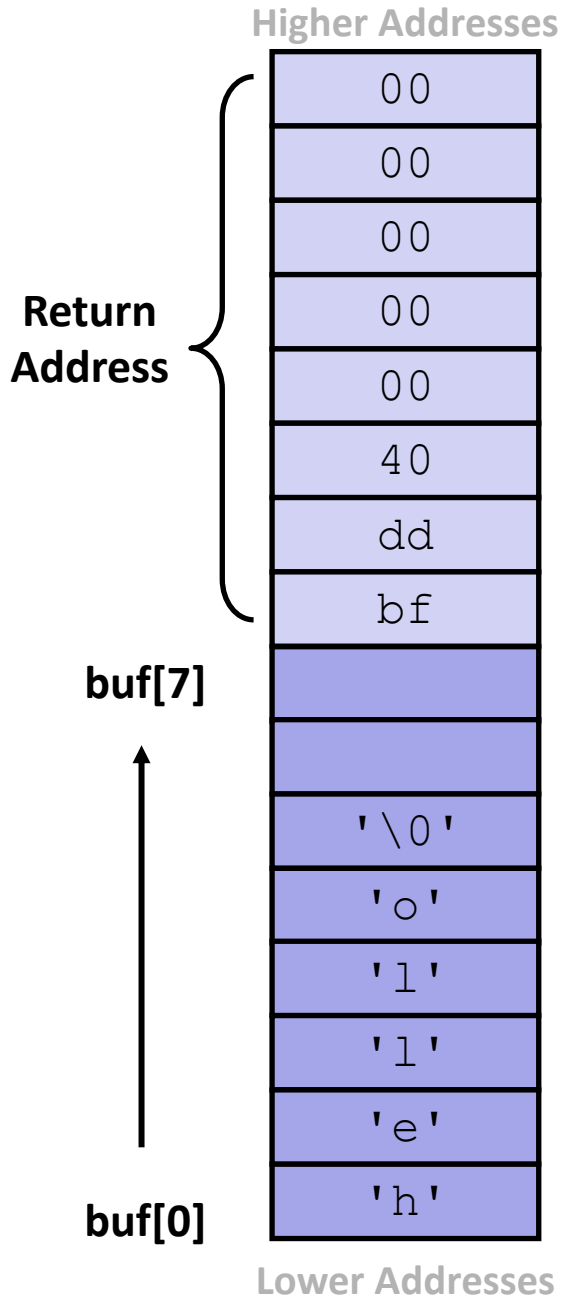


B

ut

Buffer Overflow in a Nutshell

- Stack grows *down* towards lower addresses
- Buffer grows *up* towards higher addresses
- If we write past the end of the array, we overwrite data on the stack!



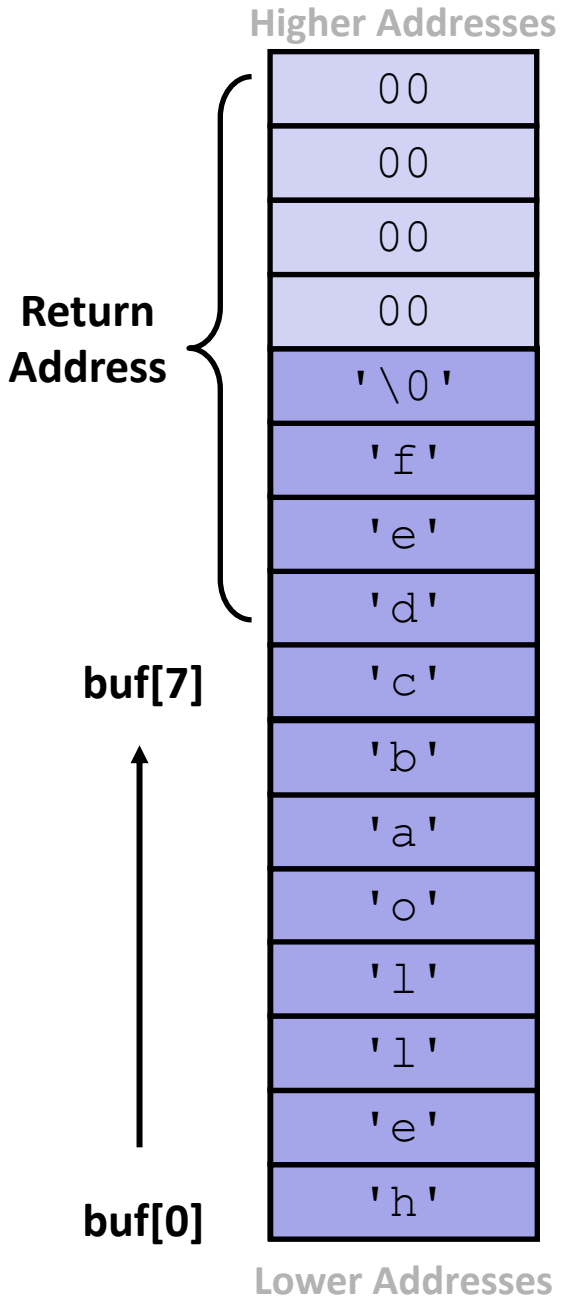

```
Enter input: hello
```

No overflow ☺

Buffer Overflow in a Nutshell

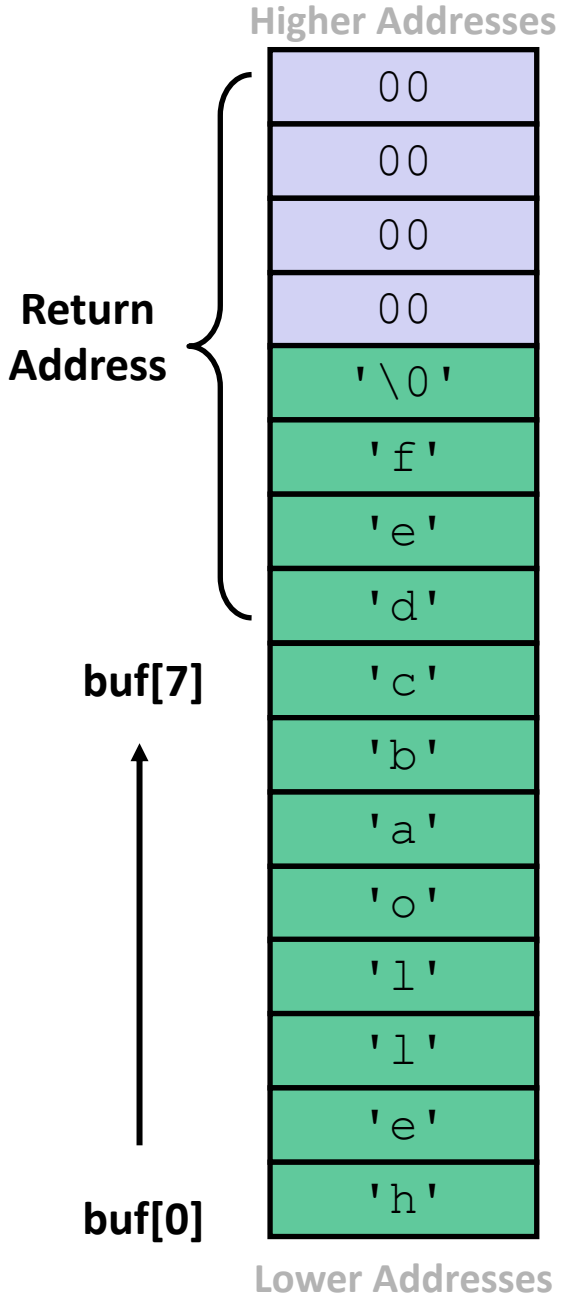
- Stack grows *down* towards lower addresses
- Buffer grows *up* towards higher addresses
- If we write past the end of the array, we overwrite data on the stack!

```
Enter input: helloabcdef
```



Buffer Overflow in a Nutshell

- Stack grows *down* towards lower addresses
- Buffer grows *up* towards higher addresses
- If we write past the end of the array, we overwrite data on the stack!




```
Enter input: helloabcdef
```

Buffer overflow! ☹

Buffer Overflow in a Nutshell

- Buffer overflows on the stack can overwrite "interesting" data
  - Attackers just choose the right inputs

- Simplest form: sometimes called "stack smashing"
  - Unchecked length on string input into bounded array causes overwriting of stack data
  - Specifically, try to change the return address of the current procedure!

- Why is this a big deal?
  - It was the #1 **technical** cause of security vulnerabilities!
    - e.g. Heartbleed, cloudbleed, etc.
    - #1 overall cause is social engineering/user ignorance 😬

So let's see how systems let us do this…

String Library Code

- Actual source code implementation of Unix function `gets()`:

```
/* Get string from stdin
one character at a time */
char* gets(char* dest) {
    int c = getchar(); // read 1 byte
    char* p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

pointer to start of an array

same as:

```
*p = c;
p++;
```

What could go wrong in this code?

String Library Code

- Actual source code implementation of Unix function `gets()`:

```
/* Get string from stdin
one character at a time */
char* gets(char* dest) {
    int c = getchar(); // read 1 byte
    char* p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

Similar problems with other Unix functions:

- `strcpy`: Copies string of arbitrary length to a dst
- `scanf`, `fscanf`, `sscanf`, when given `%s` specifier

🚨 No way to specify **limit** on number of characters to read! 🚨
The `man` page for `gets(3)` now says "BUGS: Never use `gets()`."

An Example: Vulnerable Buffer Code

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

```
/* Echo Line */
void echo() {
    char buf[8];  /* Way too small! */
    gets(buf);    /* Read input into buf */
    puts(buf);    /* Print output from buf */
}
```

```
unix> ./buf-nsp
Enter string: 123456789012345
123456789012345
```

```
unix> ./buf-nsp
Enter string: 1234567890123456
Segmentation fault (core dumped)
```

Buffer Overflow Disassembly (`buf-nsp`)

call_echo:

```
0000000000401177 <call_echo>:
  401177:  48 83 ec 08           sub    $0x8,%rsp
  40117b:  b8 00 00 00 00        mov    $0x0,%eax
  401180:  e8 c1 ff ff ff        callq  401146 <echo>
  401185:  48 83 c4 08           add    $0x8,%rsp
  401189:  c3                    retq
```

return address to place on stack

echo:

```
 0000000000401146 <echo>:
  401146:  48 83 ec 18           sub    $0x18,%rsp
    ...                           ... calls printf ...
  401159:  48 8d 7c 24 08        lea    0x8(%rsp),%rdi
  40115e:  b8 00 00 00 00        mov    $0x0,%eax
  401163:  e8 e8 fe ff ff        callq  401050 <gets@plt>
  401168:  48 8d 7c 24 08        lea    0x8(%rsp),%rdi
  40116d:  e8 be fe ff ff        callq  401030 <puts@plt>
  401172:  48 83 c4 18           add    $0x18,%rsp
  401176:  c3                    retq
```

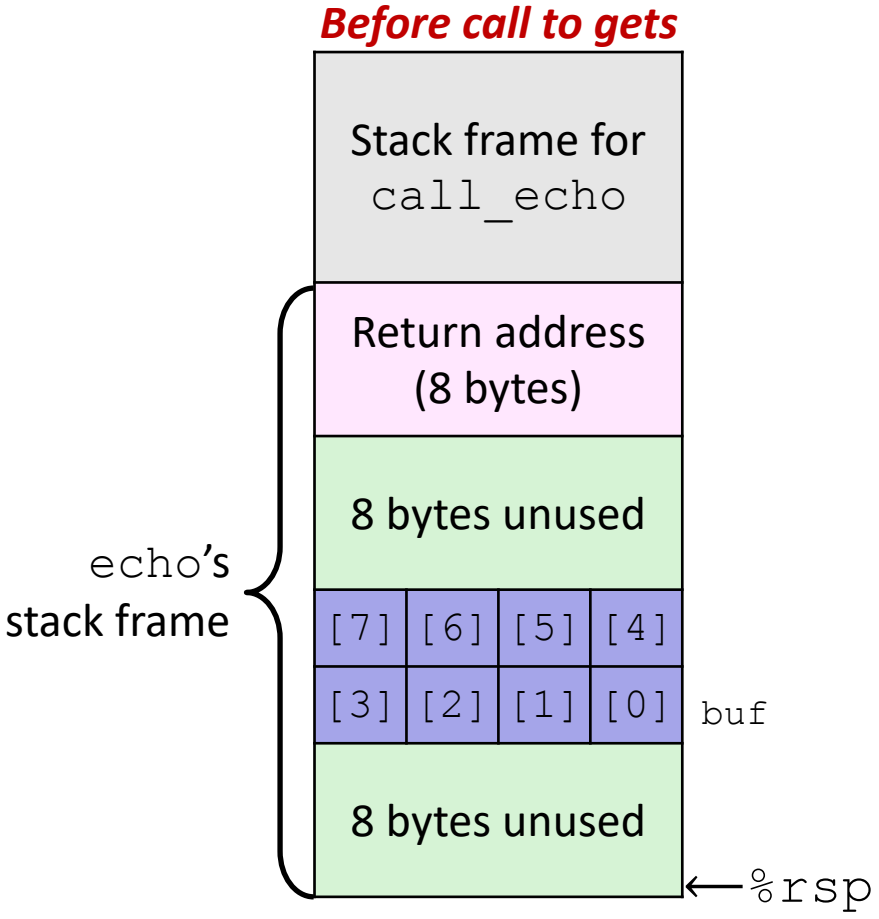
Allocate 24 bytes in stack (compiler's choice)

Calculate address location to be passed to `gets`

Calculate address location to be passed to `puts`

Clean up stack & return

Buffer Overflow Stack

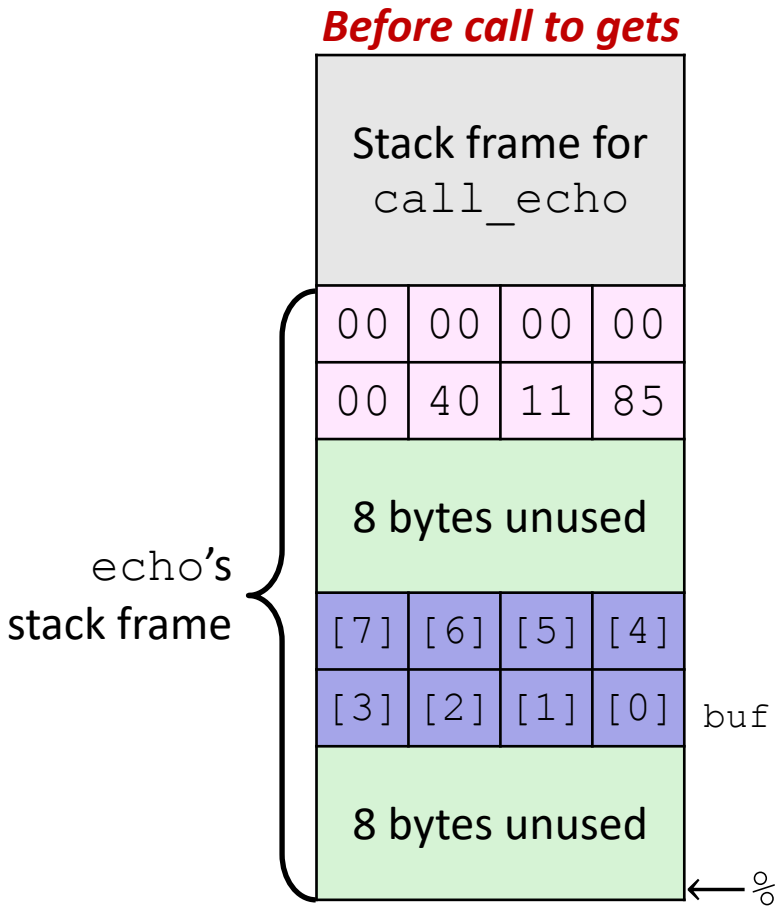



```
/* Echo Line */
void echo()
{
    char buf[8];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
  subq   $24, %rsp
    ...
  leaq   8(%rsp), %rdi
  mov    $0x0,%eax
  call   gets
    ...
```

Note: addresses increasing right-to-left, bottom-to-top

Buffer Overflow Setup



call_echo:

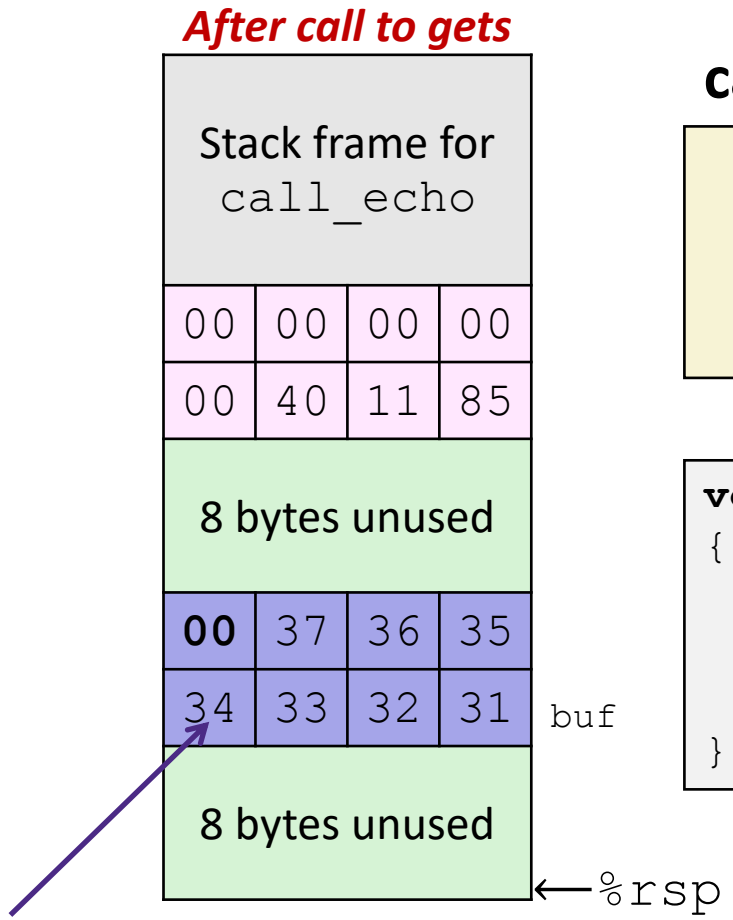
```
   . . .
   401180:  callq  401146 <echo>
   401185:  add    $0x8,%rsp
   . . .
```

```
void echo()
{
    char buf[8];
    gets(buf);
    . . .
}
```

```
echo:
   subq   $24, %rsp
    ...
   leaq   8(%rsp), %rdi
   mov    $0x0,%eax
   call   gets
    ...
```

Note: addresses increasing right-to-left, bottom-to-top

Buffer Overflow Example #1: 1234567



Note: Digit "*N*" is just 0x3*N* in ASCII!

call_echo:

```
   . . .
   401180:  callq  401146 <echo>
   401185:  add    $0x8,%rsp
   . . .
```

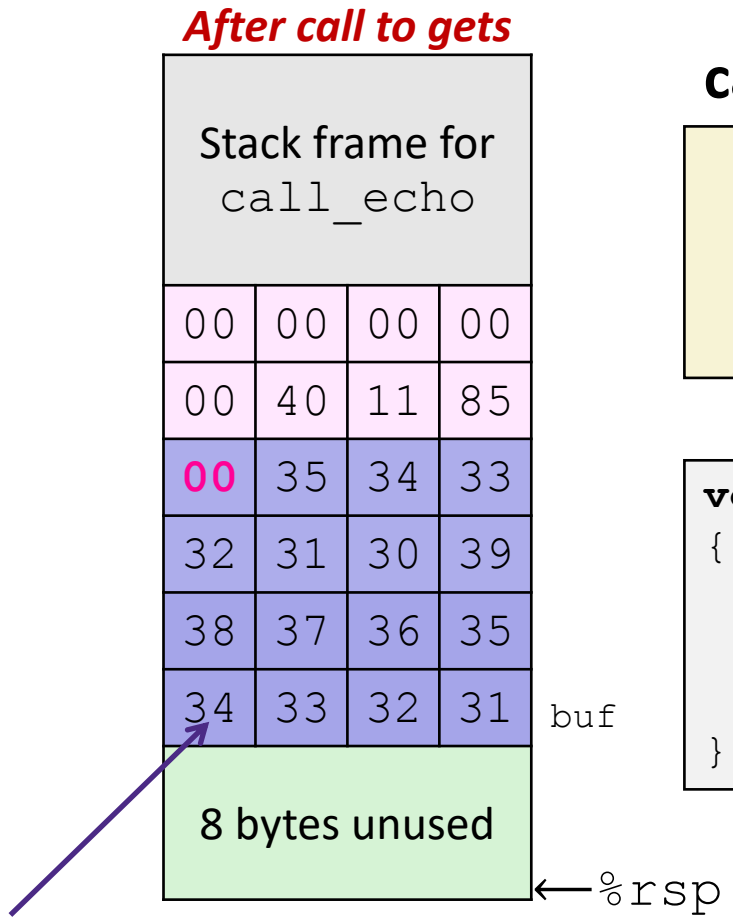
```
void echo()
{
    char buf[8];
    gets(buf);
    . . .
}
```

```
echo:
   subq   $24, %rsp
    ...
   leaq   8(%rsp), %rdi
   mov    $0x0,%eax
   call   gets
    ...
```

```
unix> ./buf-nsp
Enter string: 1234567
1234567
```

All good! No overflow. Phew!

Buffer Overflow Example #2: 123456789012345



Note: Digit "*N*" is just 0x3*N* in ASCII!

call_echo:

```
   . . .
   401180:  callq  401146 <echo>
   401185:  add    $0x8,%rsp
   . . .
```

```
void echo()
{
    char buf[8];
    gets(buf);
    . . .
}
```

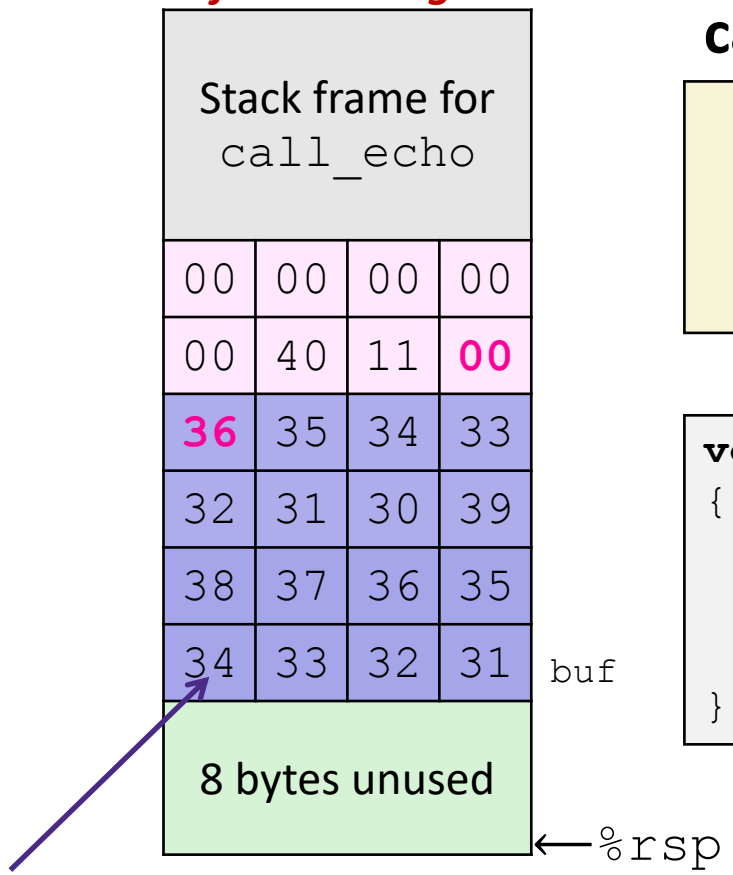
```
echo:
   subq   $24, %rsp
    ...
   leaq   8(%rsp), %rdi
   mov    $0x0,%eax
   call   gets
    ...
```

```
unix> ./buf-nsp
Enter string: 123456789012345
123456789012345
```

Overflowed buffer, but did not corrupt state.

Buffer Overflow Example #3: 1234567890123456

After call to gets



Note: Digit "*N*" is just 0x3*N* in ASCII!

call_echo:

```
   . . .
   401180:  callq  401146 <echo>
   401185:  add    $0x8,%rsp
   . . .
```

```
void echo()
{
    char buf[8];
    gets(buf);
    . . .
}
```

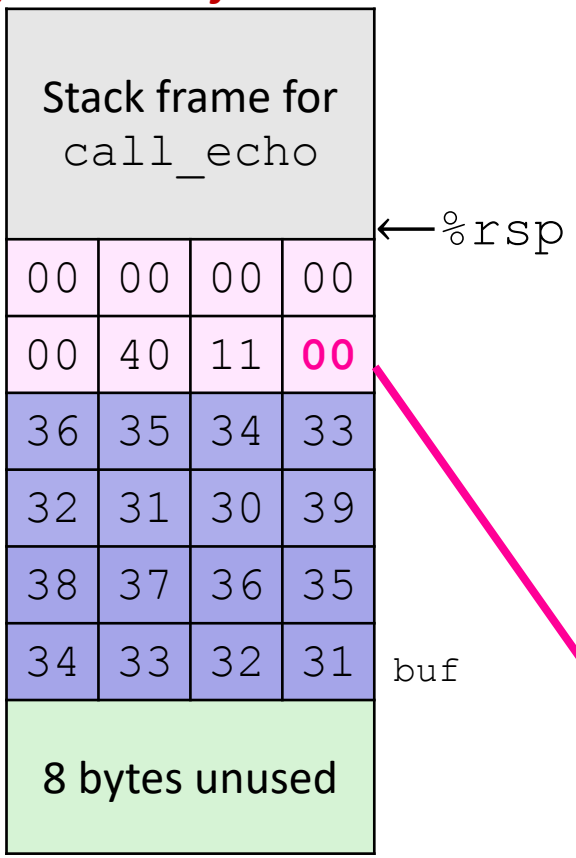
```
echo:
   subq   $24, %rsp
    ...
   leaq   8(%rsp), %rdi
   mov    $0x0,%eax
   call   gets
    ...
```

```
unix> ./buf-nsp
Enter string: 12345678901234566
Segmentation fault (core dumped)
```

Overflowed buffer and corrupted return pointer!

Buffer Overflow Example #3 Explained

After return from echo




```
0000000000004010d0 <register_tm_clones>:
  4010d0:  lea    0x2f61(%rip),%rdi
  4010d7:  lea    0x2f5a(%rip),%rsi
  4010de:  sub    %rdi,%rsi
  4010e1:  mov    %rsi,%rax
  4010e4:  shr    $0x3f,%rsi
  4010e8:  sar    $0x3,%rax
  4010ec:  add    %rax,%rsi
  4010ef:  sar    %rsi
  4010f2:  je     401108
  4010f4:  mov    0x2efd(%rip),%rax
  4010fb:  test   %rax,%rax
  4010fe:  je     401108
  401100:  jmpq   *%rax
  401102:  nopw   0x0(%rax,%rax,1)
  401108:  retq
```

"Returns" to a valid instruction, but **bad indirect jump**
so program signals `SIGSEGV, Segmentation fault`

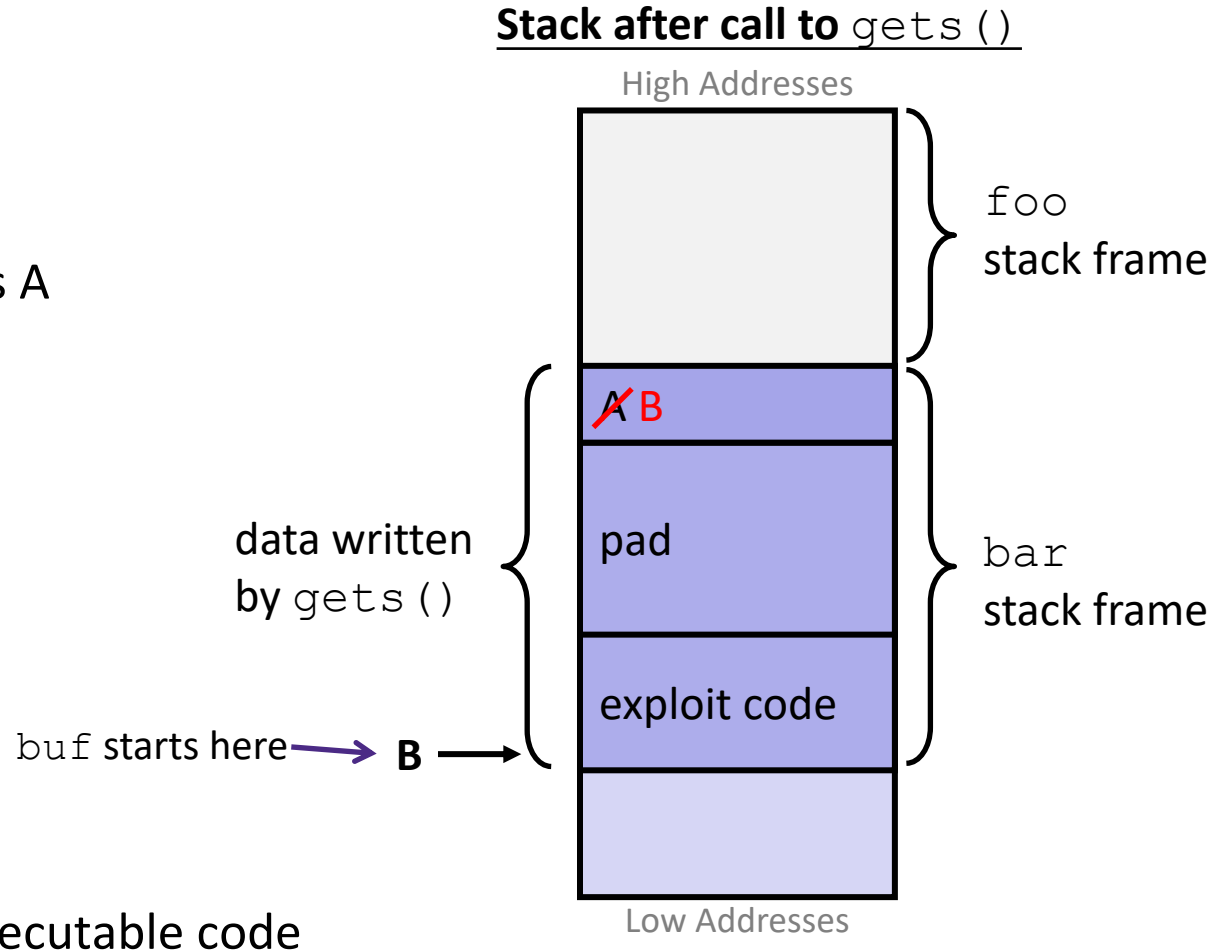
Attack Time

Malicious Use of Buffer Overflow: Code Injection Attacks

```
void foo(){
  bar();
A:...
}
```

return address A

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



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