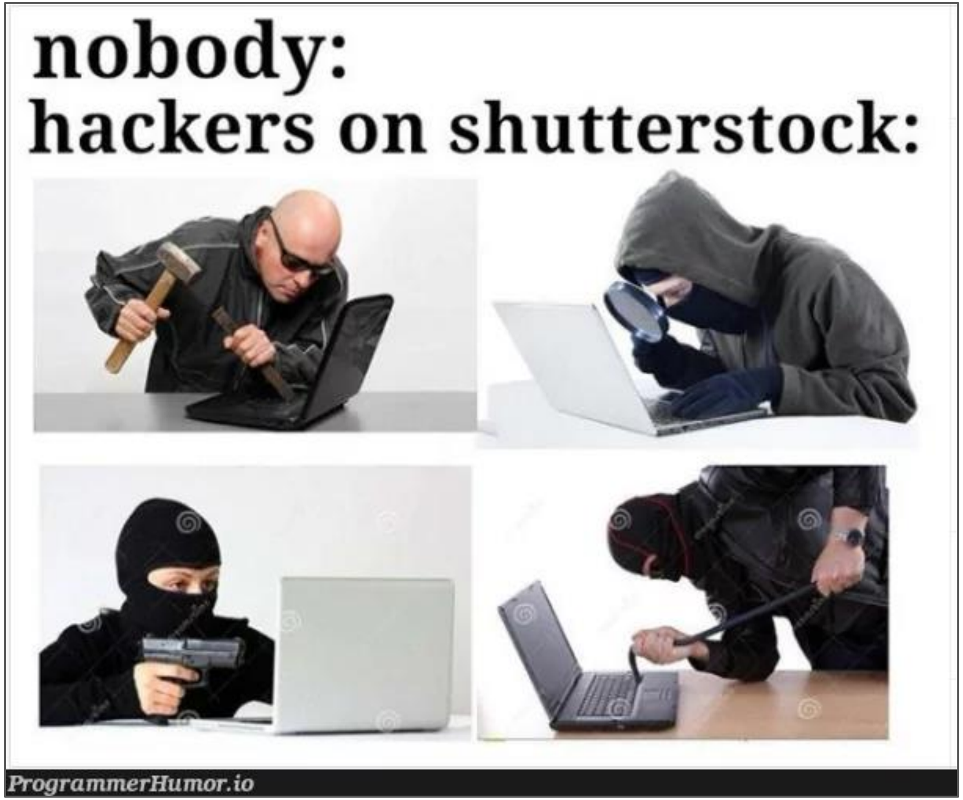
Buffer Overflow

CSE 351 Summer 2024

Instructor:
Ellis Haker

Teaching Assistants:
Naama Amiel
Micah Chang
Shananda Dokka
Nikolas McNamee
Jiawei Huang



Administrivia

- Today:
  - HW11 due (11:59pm)
  - **Mid-Quarter Survey due** (11:59pm)
  - Lab3 released! (due next Friday, 7/26)
- Friday, 7/19
  - RD14 due (1pm)
  - HW12 due (11:59pm)
  - **Lab2 due** (11:59pm)
    - Reminder: weekend counts as 1 late day
- **Quiz 2 released on Monday**

TA Applications are Open!

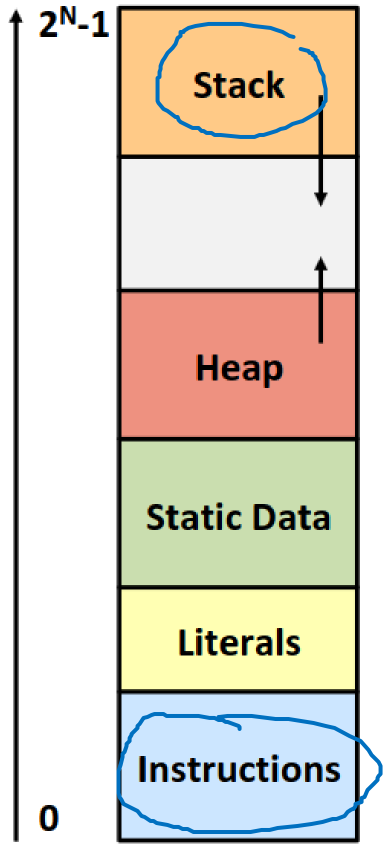
- Apply by Monday, 7/22 to TA for Fall
  - https://www.cs.washington.edu/students/ta
  - Same application for all CSE classes (besides intro)
- You are eligible to TA for 351 next quarter!
  - If interested, please also contact Ruth Anderson to let her know you're interested

Lecture Topics

- **Memory Layout Review**
- Buffer overflow
  - Input buffers on the stack
  - Overflow attacks and code injection
- Exploits Based on Buffer Overflows
- Defenses against buffer overflow
- Societal Impact

Review: Memory Layout

- **Stack**
  - Local variables, procedure context
- **Heap**
  - Dynamically allocated using `malloc()`
  - Future lecture topic!
- Statically-allocated data
  - Read/write: **Static Data** – global vars, etc.
  - Read-only: **Literals** – string literals, etc.
- **Instructions**
  - Machine code
  - Read-only

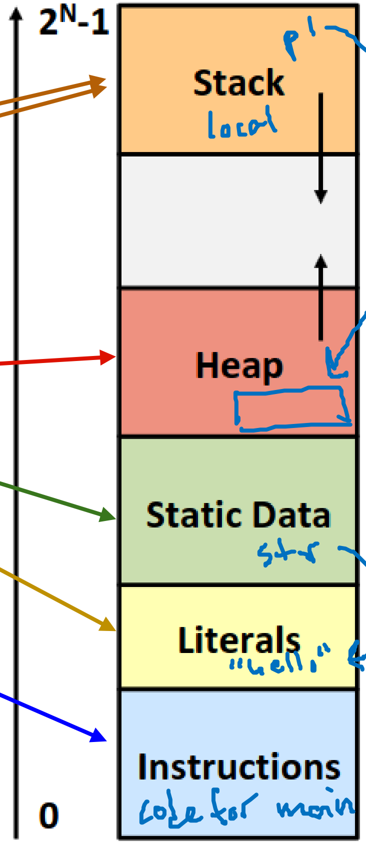


Memory Allocation Example

```
char* str = “hello!”;

int main() {
      void *p1;
      int local = 0;
      p1 = malloc(1L << 28); /* 256 MB */
      /* Some other code ... */
}
```

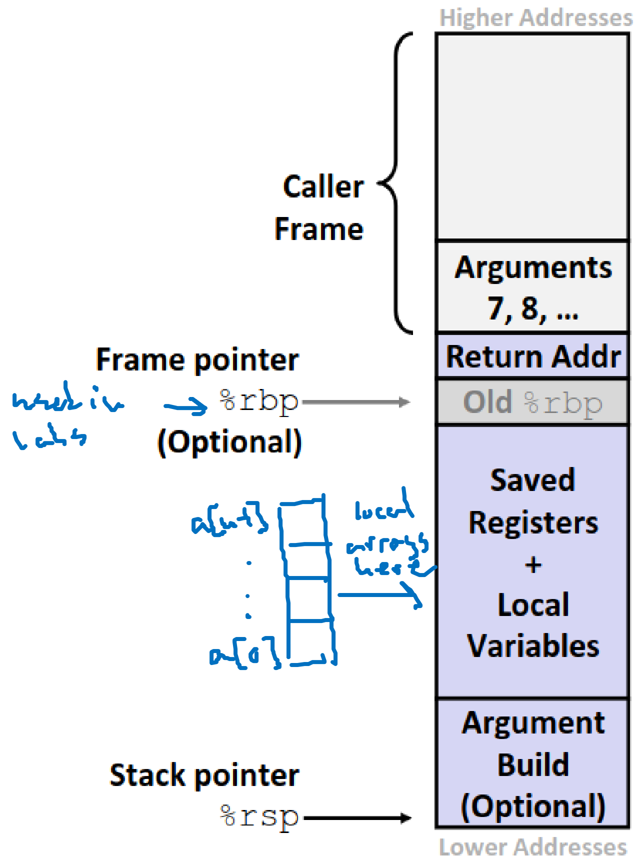
malloc(n) allocates n bytes on the heap, returns a pointer to it



Where does everything go?

Review: x86 Stack Frame

- Caller's stack frame
  - Arguments 7+ for this call
- Current stack frame
  - Return address pushed by `call` instruction
  - Old frame pointer (optional)
  - Local data
    - Callee-saved registers pushed before using
    - Caller-saved registers pushed before calling another function
  - Argument build = arguments 7+ for the *next* function



Lecture Topics

- Memory Layout Review
- **Buffer overflow**
    - **Input buffers on the stack**
    - **Overflow attacks and code injection**
- Exploits Based on Buffer Overflows
- Defenses against buffer overflow
- Societal Impact

What is a Buffer?

- An array used to temporarily store data
  - Typically some input or output
- Example: you've probably seen "video buffering"
  - Video data from the internet is written to a buffer before being played

Buffer Overflow in a Nutshell

- C does not check array bounds
  - **Buffer Overflow** = writing past the end of an array
- Characteristics of the Linux memory layout provide opportunities for malicious programs
  - Stack grows “backwards” in memory
  - Stack used for both data and control flow (return addresses)
  - Data and instructions both stored in memory

Buffer Overflow in a Nutshell (pt 2)

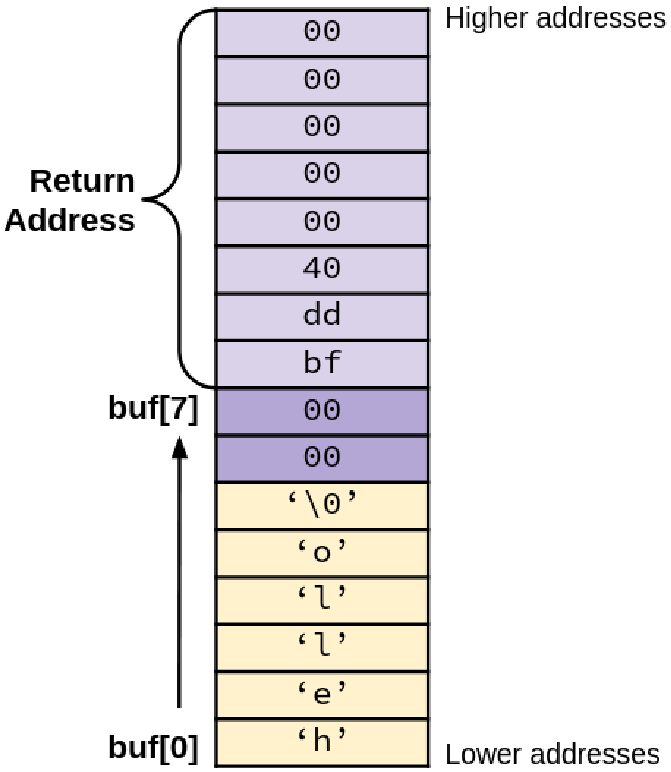
- Stack grows *down* towards lower addresses
- Buffer grows *up* towards higher addresses

- **Result**: if we overflow a buffer on the stack, we will overwrite other data!

Example:

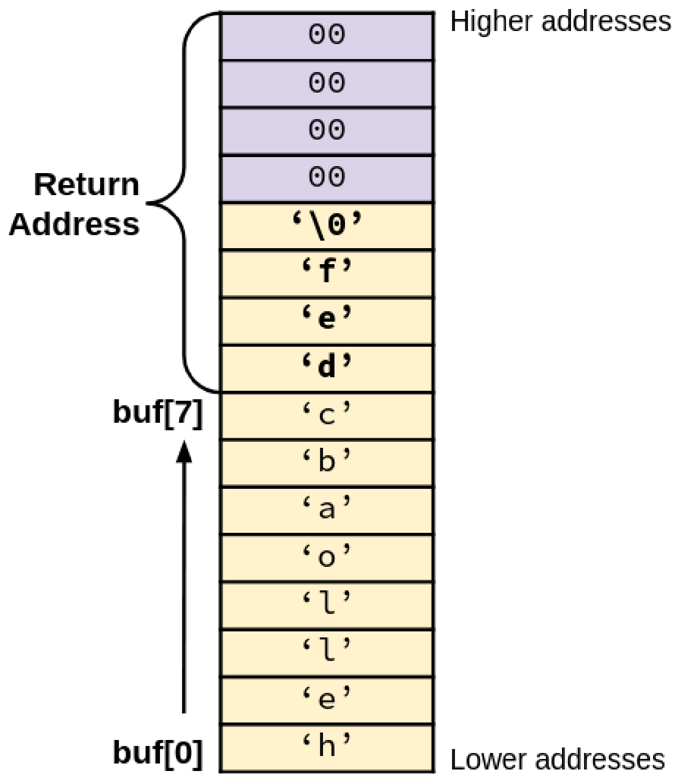
```
Enter input: hello
```

No overflow :)



Buffer Overflow in a Nutshell (pt 3)

- Stack grows *down* towards lower addresses
- Buffer grows *up* towards higher addresses
- **Result**: if we overflow a buffer on the stack, we will overwrite other data!



Example:

```
Enter input: helloabcdef
```

Buffer overflow :(

Buffer Overflow in a Nutshell (pt 4)

- Buffer overflows on the stack can overwrite important data
  - *e.g.*, the return address
  - A clever attacker can use this to their advantage
- Simplest form is **stack smashing**
  - Overwrite return address to change how a program runs
- More complex forms include **code injection**
  - Attacker can cause a program to run their own code!
- Why is this a big deal?
  - One of the most common *technical* causes of security vulnerabilities
    - Social engineering is more common than any technical cause

String Library 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;
}
```

similar to Java's
System.in.next()

gets input from user, stores
at dest.

Expects dest to point to
an allocated array of char

What could go wrong with this code?

String Library Code (pt 2)

remember from arrays lecture — need to pass in size as an argument

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;
}
```

- What if the function reads in more data than we have space for in `dest`?
- Similar problem in other standard library functions
  - `strcpy()`
  - `scanf()`, if given a `%s` specifier

Vulnerable Buffer Code

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

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

- `gets()` writes from stdin to `buf`
- `puts()` writes from `buf` to stdout
- What happens if `gets()` writes past the end of `buf`?

undefined behaviors!

```
unix:~$ ./run_echo
Enter string: 123456789012345
123456789012345
```

```
unix:~$ ./run_echo
Enter string: 1234567890123456
Segmentation fault (core dumped)
```

Vulnerable Buffer Code Disassembly

24

subtract from rsp = make 24B of space on the stack

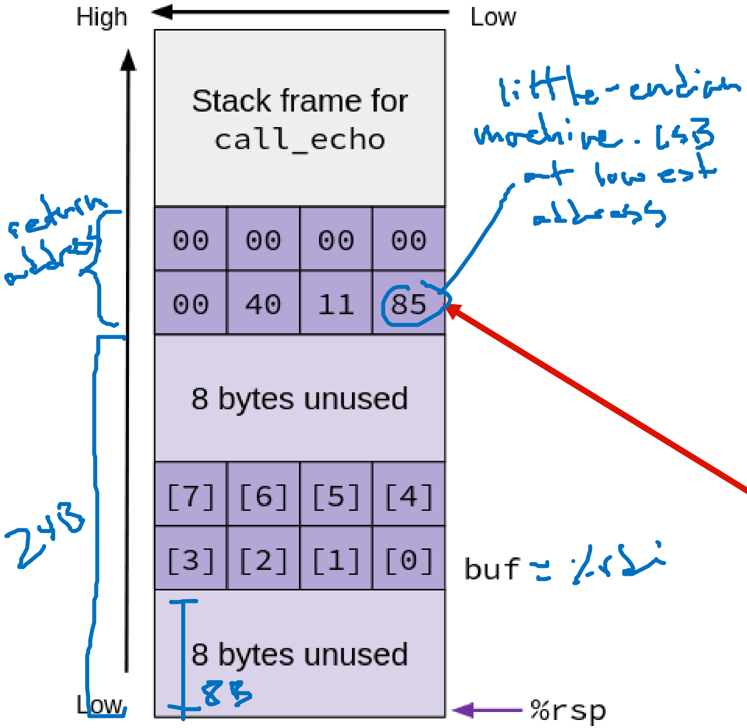
```
0000000000401146 <echo>:
  401146: 48 83 ec 18       sub $0x18,%rsp
  ...                       ...      # calls printf
  401159: 48 8d 7c 24 08    lea 0x8(%rsp),%rdi
  40115e: b8 00 00 00 00    mov $0x0,%eax
  401163: e8 e8 fe ff ff    callq 401050 <gets@plt>
  401168: 48 8d 7c 24 08    lea 0x8(%rsp),%rdi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

arg for gets = address rsp + 8

```
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

Vulnerable Code Stack (before `gets()`)

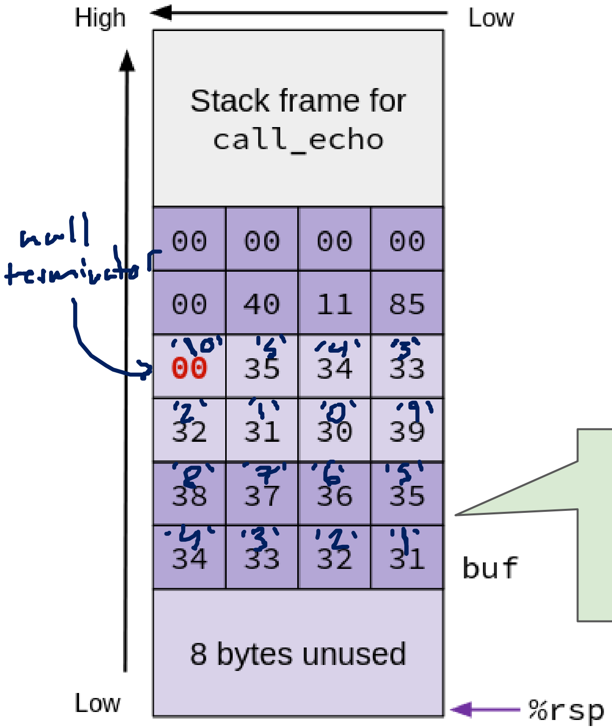



```c
void echo() {
    char buf[8];
    . . .
    gets(buf);
    . . .
}
```

```
echo:
   subq $0x18,%rsp
   ...
   leaq 0x8(%rsp),%rdi
   ...
   call gets
   ...
   addq $0x18,%rsp
   retq
```

```
call_echo:
   ...
   401180: call echo
   401185: addq $0x8,%rsp
   ...
```

Example #1 (after `gets()`)



Every digit *N* has the ASCII 0x3*N*

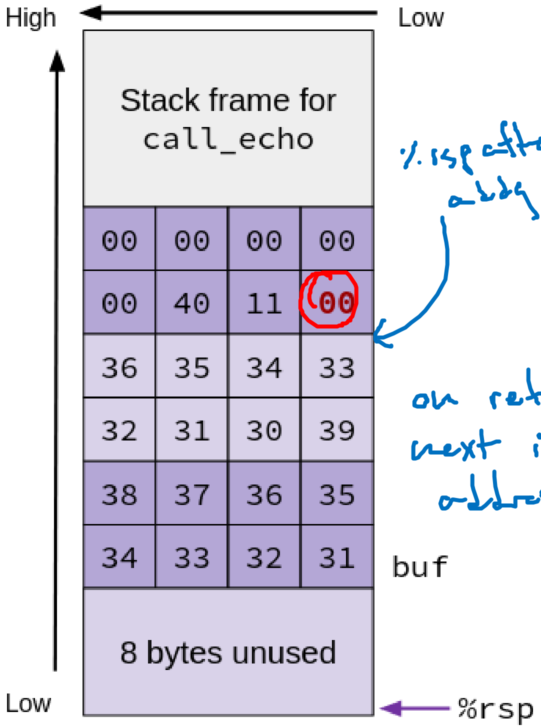
```c
void echo() {
    char buf[8];
    . . .
    gets(buf);
    . . .
}
```

```
echo:
   subq $0x18,%rsp
   ...
   leaq 0x8(%rsp),%rdi
   ...
   call gets
   ...
   addq $0x18,%rsp
   retq
```

```
unix:~$ ./run_echo
Enter string: 123456789012345
123456789012345
```

Overflowed buffer, but didn't corrupt important data

Example #2 (after `gets()`)



%rsp after addq

on return, CPU tries to find next instruction at address 0x401100 !!

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

```
echo:
   subq $0x18,%rsp
   ...
   leaq 0x8(%rsp),%rdi
   ...
   call gets
   ...
   addq $0x18,%rsp
   retq
```

```
unix:~$ ./run_echo
Enter string: 1234567890123456
Segmentation fault (core dumped)
```

Overwrote the return address!

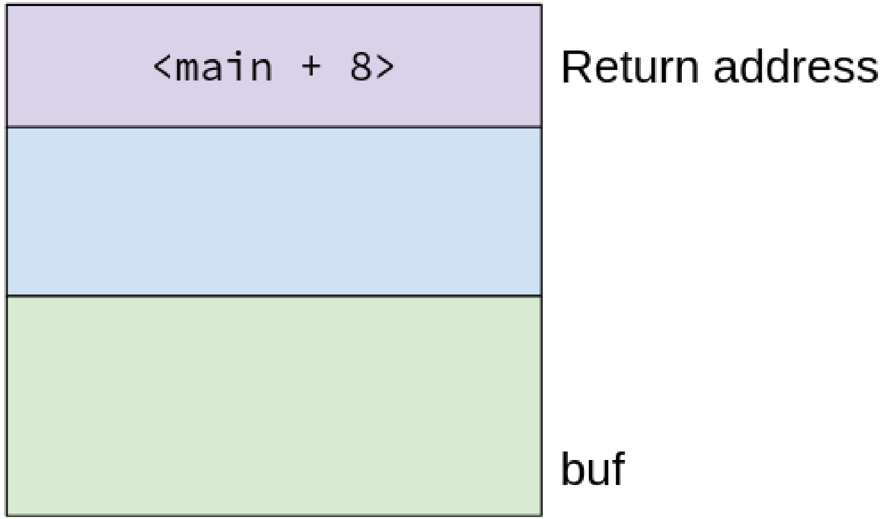
Attack Time

Buffer Overflow Attacks: Stack Smashing

Lab 2 phase 0

- Simpler attack
  - Overwrite the return address
- Usually execute another function in instruction memory

ex: say there's some other function in this program, called foo, that we want to execute

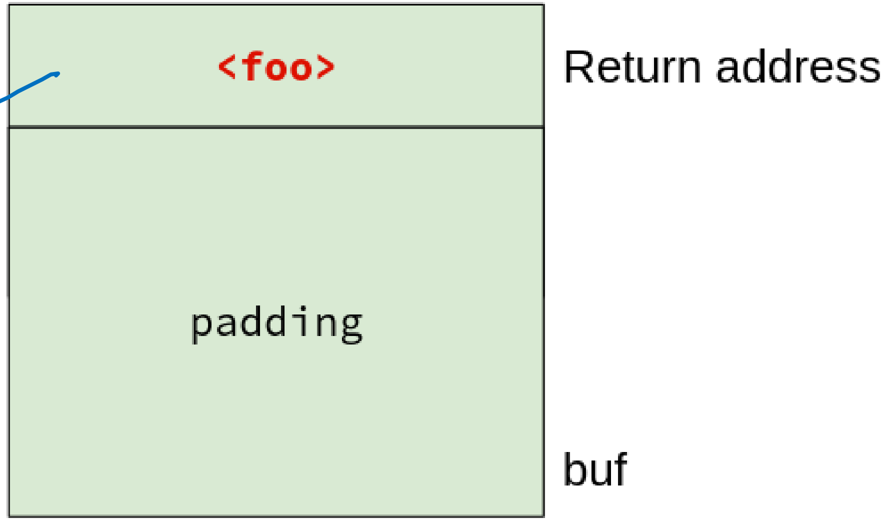


Buffer Overflow Attacks: Stack Smashing (pt 2)

- Simplest common attack
  - Overwrite the return address
- Usually execute another function in instruction memory

on return, executes foo

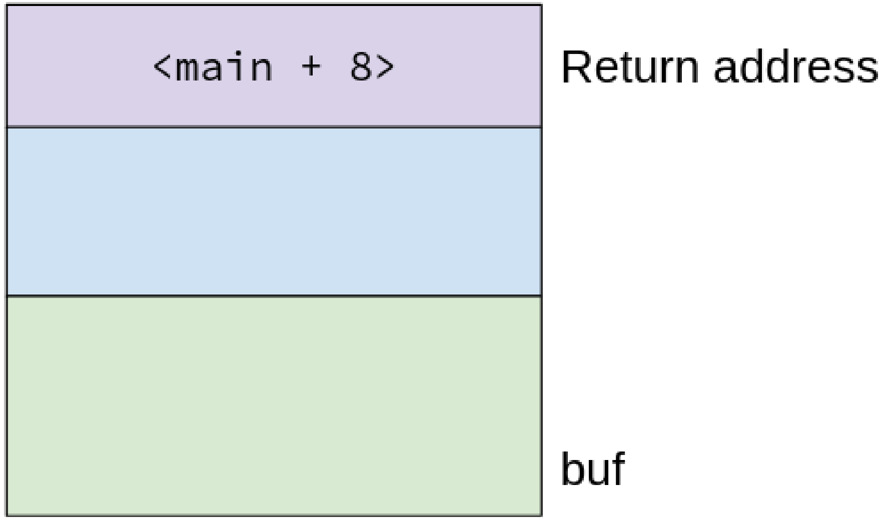
```
Enter string: <padding><foo>
```



Buffer Overflow Attacks: Code Injection

- Allows attacker to execute **arbitrary code** on victim machine!
- Write byte code into the buffer, then overwrite the return address to point to that code

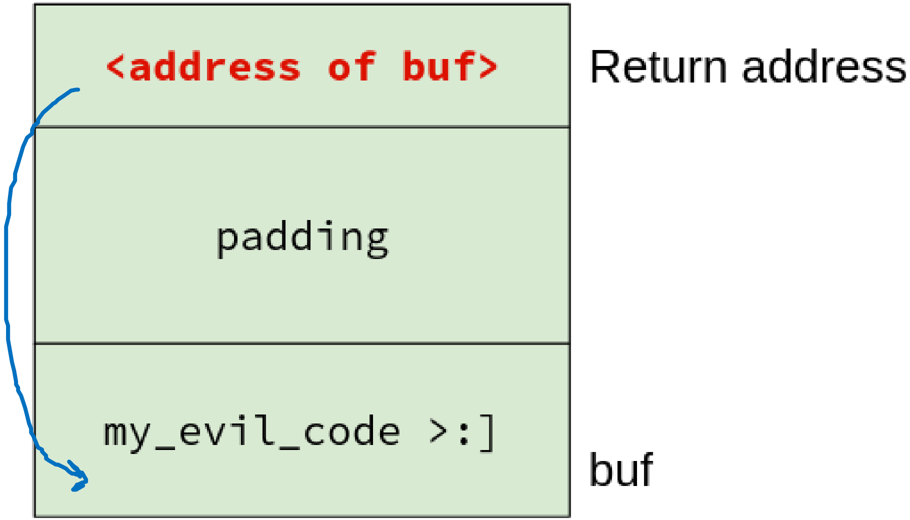
Lab2 phase 2



Buffer Overflow Attacks: Code Injection (pt 2)

- Allows attacker to execute **arbitrary code** on victim machine!
- Write byte code into the buffer, then overwrite the return address to point to that code
  - When current function returns, it will execute the code you put in the buffer!

```
Enter string: <evil_code><padding>
<address of buf>
```



Practice Question

buggy is vulnerable to stack smashing!

What is the minimum number of characters that gets must read in order for us to change the return address to a stack address?

(for example: 0x00 00 7f ff ca fe f0 0d)

don't need to write in 0s b/c they're already 0 in memory – write 6B of ret addr

A) 27
B) 20
C) 51
D) 54

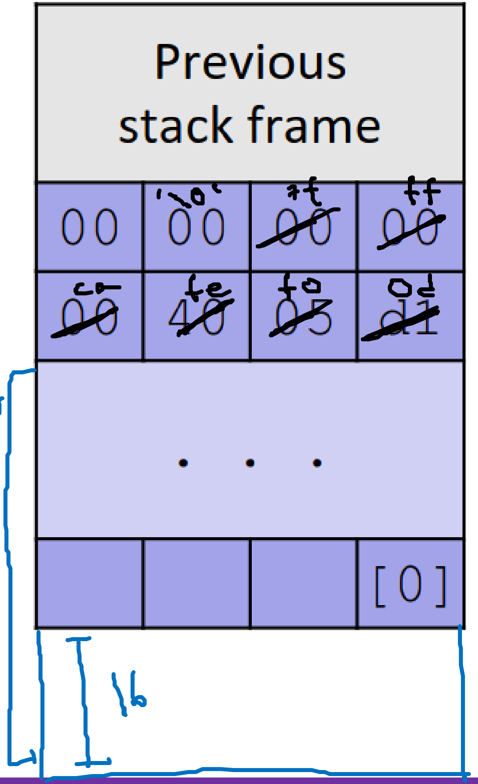
total = 64-16+6 = 54B

make 64B of stack space

```
buggy:
   subq   $0x40, %rsp
    ...
   leaq   16(%rsp), %rdi
   call   gets
    ...
```

64

buffer starts 16B into stack



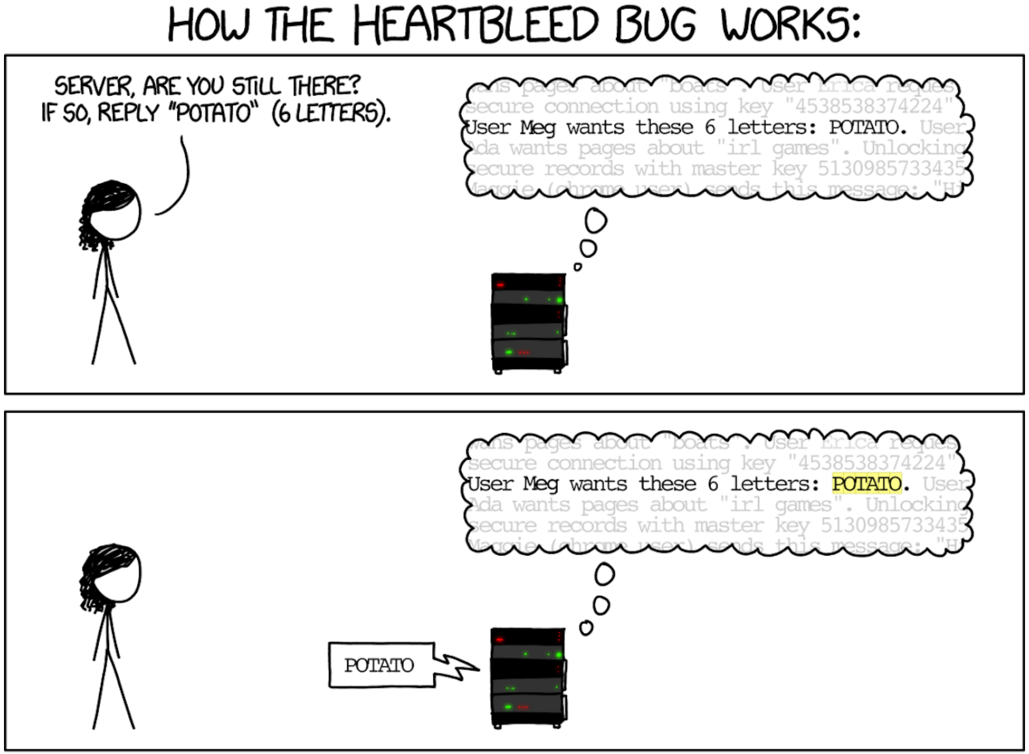
Lecture Topics

- Memory Layout Review
- Buffer overflow
  - Input buffers on the stack
  - Overflow attacks and code injection
- **Exploits Based on Buffer Overflows**
- Defenses against buffer overflow
- Societal Impact

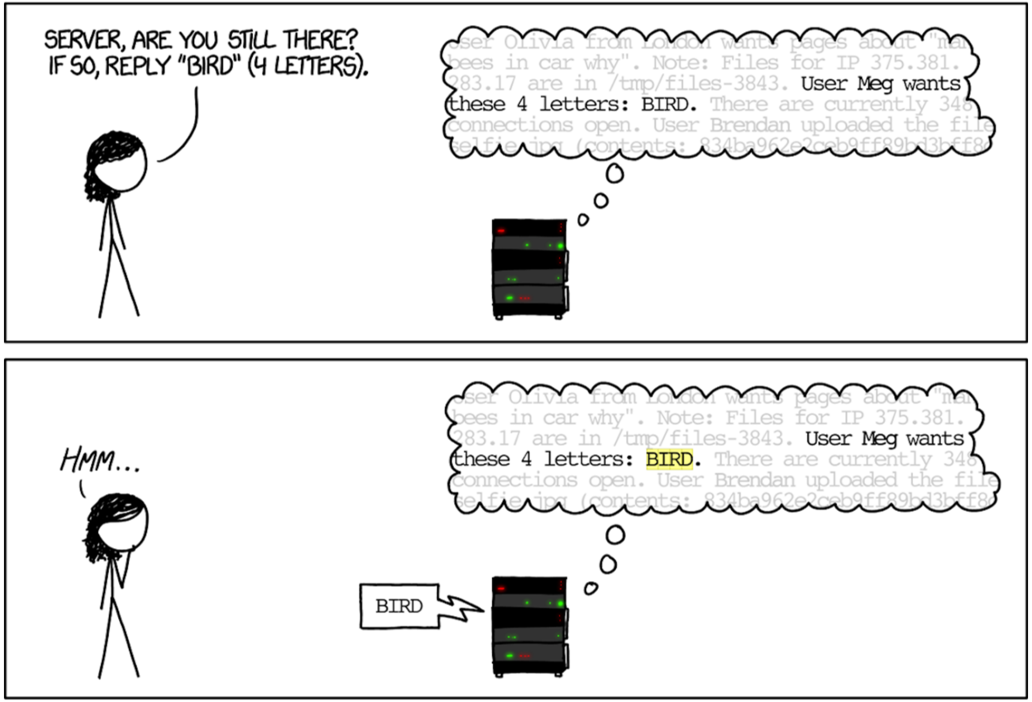
Morris Worm (1988)

- First ever internet worm
- Exploited finger server (`fingerd`), used gets to read the argument sent by the client
  - Attacked `fingerd` server with phony argument:
    - <u>Ex</u>: `finger "exploit-code padding new-return-addr"`
- Invaded ~6000 computers in hours (10% of the internet)
- The author, Robert Morris, was prosecuted
  - First conviction under 1986 Computer Fraud and Abuse Act
  - Now an MIT professor…

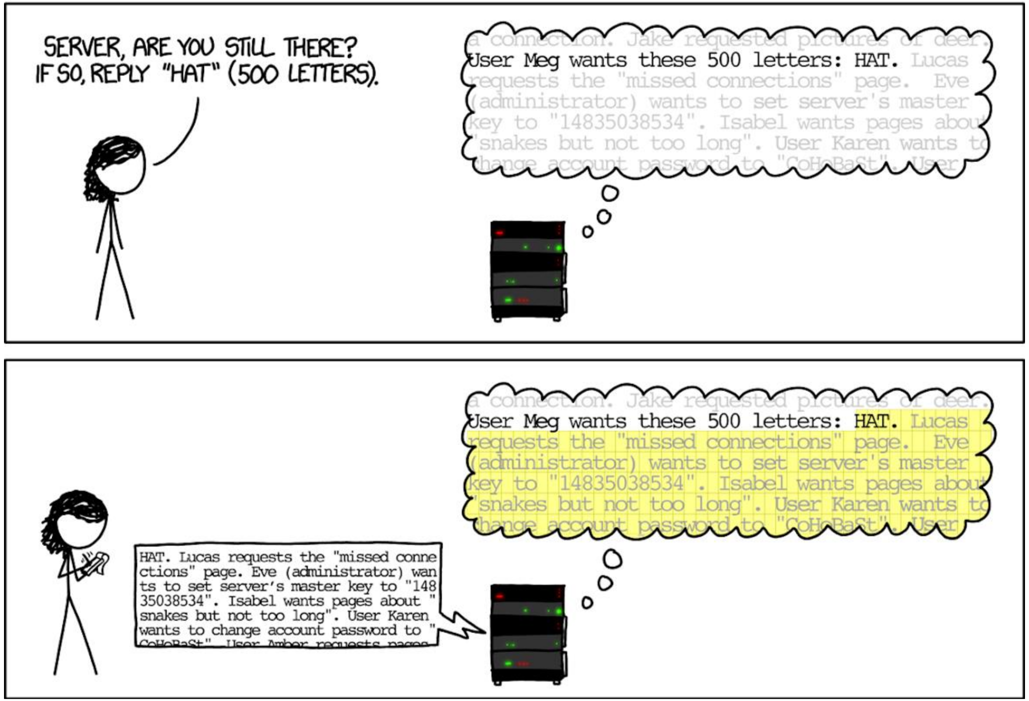
Heardbleed (2014)



Heardbleed (2014) (pt 2)

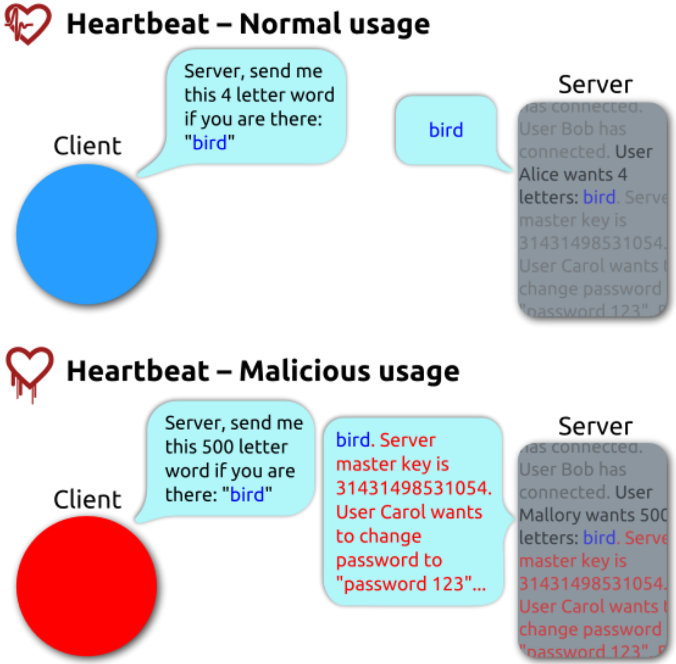


Heardbleed (2014) (pt 3)



Heartbleed Explained

- Exploited vulnerability in OpenSSL
  - Open-source security library
- “Heartbeat” packet: message and length
  - Server echos message back
  - Trusted the given length!
    - Allowed attackers to read contents of memory
- ~17% of the internet affected
  - GitHub, Yahoo, Amazon Web Services, etc.



By FenixFeather - Own work, CC BY-SA 3.0,
https://commons.wikimedia.org/w/index.php?curid=32276981

Lecture Topics

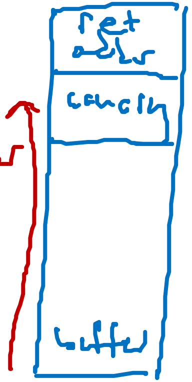
- Memory Layout Review
- Buffer overflow
  - Input buffers on the stack
  - Overflow attacks and code injection
- Exploits Based on Buffer Overflows
- **Defenses against buffer overflow**
- Societal Impact

System-Level Protections

- **Non-executable memory segments**
  - In traditional x86, only “read” and “write” permissions, could execute anything
  - x86-64 added “execute” permissions
    - Only instruction memory marked executable
    - Attempting to execute non-executable memory will cause a segfault
- **Randomized stack offsets**
  - At start of program, allocate a random amount of stack space
    - Shifts addresses for the rest of the program
    - Addresses will be different every time it’s run
- **Pros**: automatic (programmer doesn’t have to do anything)
- **Cons**: requires hardware support, doesn’t stop all attacks (*e.g.*, return to libc)

Compiler-Level Protections

wants to overwrite canary to get to ret addr



- **Stack canaries**
  - Place special value (“canary”) in the stack just beyond the buffer
    - Check value for corruptio before exiting function
  - GCC implementation: `-fstack-protector`
  - **Pros**:
    - Easy to implement
  - **Cons**:
    - Only detects errors, doesn’t stop them
    - *Slow*

```
unix:~$ ./run_echo
Enter string: 12345678
12345678
```

```
unix:~$ ./run_echo
Enter string: 123456789
*** stack smashing detected ***
```

Programmer-Level Protections

- Avoid using unsafe standard library functions
  - `gets()`, `strcpy()`, etc.
    - No way to pass in array size!
  - Most have been replaced with safer alternatives (`fgets()`, `strncpy()`, etc.)
- Don't use `scanf()` with a `%s` conversion specifier
  - Use `fgets()` to read the string
  - Use `%ns` (where `n` is the max size you can read in not including the null-terminator)
- Keep track of array bounds
  - Define macros for array sizes
  - Watch out for off-by-1 errors and integer overflow

Programmer-Level Protections (pt 2)

- Alternatively, use another language that does array index bounds check
  - Most modern languages check at runtime
- What if I need a low-level systems language?
  - **Rust** is a systems language designed with security in mind
    - Does compile-time array bounds checking
- Not always possible, some projects are better suited for C

Lecture Topics

- Memory Layout Review
- Buffer overflow
  - Input buffers on the stack
  - Overflow attacks and code injection
- Exploits Based on Buffer Overflows
- Defenses against buffer overflow
- **Societal Impact**

Discussion

Take a few minutes to think about the question, and then share your thoughts with the class.

- Although it’s not as common as it once was, C is still the default language in certain areas of the industry (operating systems, embedded systems, etc.).
- Why do we still use C if it’s so insecure?
    - What benefits are there to using C?
    - What kinds of things does C allow us to do that we can’t do in other languages?
    - What might dissuade developers from using another language?

Security vs. Functionality

- Not always mutually exclusive, but often in tension
  - “The only system which is truly secure is one which is switched off and unplugged locked in a titanium lined safe, buried in a concrete bunker, and is surrounded by nerve gas and very highly paid armed guards. Even then, I wouldn't stake my life on it.” *-Gene Stafford*
- Many things we do in systems programming use C features like pointer casting etc.
  - Even Rust has “unsafe”! necessary if writing OS code
- Security checks incur overhead

Two Narratives in C

1. “I think programmers should know enough to not access array elements out of bounds. It’s a relatively simple check to insert at the language level, and if **you** can’t remember to add it, **you** shouldn’t write C.”
   a. Emphasis on the **individual**

2. “C is an absolutely awful language; why on earth doesn’t it implement bounds checking? It’s an expense, but a relatively nominal one, and **the language** would be so much easier to use.”
   a. Emphasis on **structures**

Accessibility and Computer Science

```c
/*
 * If the new process paused because it was
 * swapped out, set the stack level to the last call
 * to savu(u_ssav).  This means that the return
 * which is executed immediately after the call to aretu
 * actually returns from the last routine which did
 * the savu.
 *
 * You are not expected to understand this.
 */
if(rp->p_flag&SSWAP) {
        rp->p_flag =& ~SSWAP;
        aretu(u.u_ssav);
}
```

Unix 6th Edition Source Code

- Is C accessible?
  - “C is good for two things: being beautiful and creating catastrophic day-0s in memory management.”
- Is *programming* accessible?
  - A notoriously difficult task to do correctly (even for experts!)
  - Ideological foundations tend to over-emphasize individuals
- **You** know how to program. What now?

Discussion (pt 2)

Discuss the following questions in groups of 2-4. Then we'll share as a class.

- What do you think of when you hear the word "hacker"? Where did your beliefs about hacking come from?
- What are some of the possible consequences & objectives of hacking (i.e., to what ends might someone engage in hacking)?

What is a “hacker”?

- Very different from what you see in the movies!
    - Real hacking is much more tedious
- Stereotype is a single (usually male) person
    - Emphasizes “rugged individualism”
    - Plays into dominant narratives about who programmers are
    - Romanticizes crime (though “ethical hacking” does exist)
- *Where do these stereotypes come from?*

Some history

- Programming used to be thought of as “women’s work”
  - Played into gender stereotypes: tedious, detail-oriented work
- So what changed?
  - Between the 1960s-80s, computing culture shifted
    - Focus on individualism
    - Competition (think hackathons, etc.)
    - Higher barriers to entry (specialized CS degrees)
  - These stereotypes were pushed to turn programing into a “legitimate” science
- The “hacker” stereotype was a part of this cultural shift!

Think this is cool?

- You'll love Lab 3 :)
- Take CSE 484 (Security)
  - 1st lab is a more in-depth version of Lab 3
- More examples in bonus slides
  - Talk to Tadayoshi Kohno or Franzi Roesner if you want to know more about these
- Optional readings on Ed
- Nintendo fun!
  - Flappy bird in Mario: https://www.youtube.com/watch?v=hB6eY73sLV0

please watch this :)

BONUS SLIDES

You won’t be tested on this material, but it’s interesting nonetheless :)

Hacking Cars (2010)

- UW CSE research demonstrated wirelessly hacking a car using buffer overflow
  - http://www.autosec.org/pubs/cars-oakland2010.pdf
- Overwrote the onboard control system's code
  - Disable brakes, unlock doors, turn engine on/off

Hacking DNA Sequencing Tech (2017)

Computer Security and Privacy in DNA Sequencing

Paul G. Allen School of Computer Science & Engineering, University of Washington

- DNA Sequencer reads in DNA, encodes in binary, stores in a buffer
  - Potential for malicious code to be encoded in DNA!
  - Attacker can gain control of DNA sequencing machine when malicious DNA is read

Ney et al. (2017): https://dnasec.cs.washington.edu/

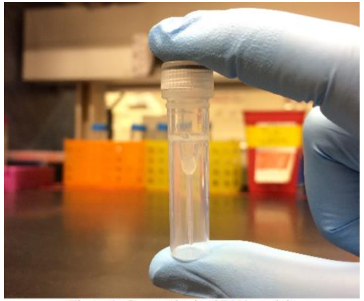


Figure 1: Our synthesized DNA exploit