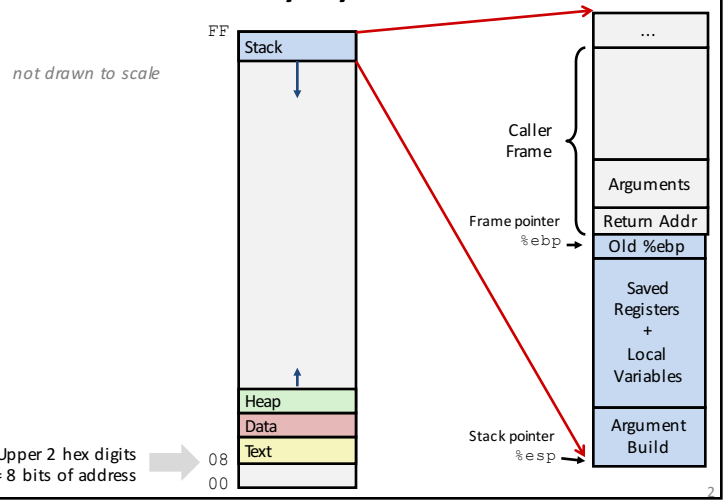
Buffer overflows (a security interlude)

How address space layout, the stack discipline, and C's lack of bounds-checking left us with a huge problem.

IA32 Linux Memory Layout



2

Buffer Overflow in a nutshell

Many classic Unix/Linux/C functions do not check argument sizes.

C does not check array bounds.

Allows overflowing (writing past the end of) buffers (arrays)

Overflows of buffers on the stack overwrite interesting data.

Attackers just choose the right inputs.

Common type of security vulnerability

3

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;
}
```

pointer to start of an array

same as:
*p = c;
p++;

What could go wrong in this code?

4

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;
}
```

No way to specify limit on number of characters to read

Similar problems with other Unix functions

`strcpy`: Copies string of arbitrary length

`scanf`, `fscanf`, `sscanf`, when given **`%s`** conversion specification

5

Vulnerable Buffer Code

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

```
int main() {
  printf("Type a string:");
  echo();
  return 0;
}
```

```
unix>./bufdemo
Type a string:1234567
1234567
```

```
unix>./bufdemo
Type a string:12345678
Segmentation Fault
```

```
unix>./bufdemo
Type a string:123456789ABC
Segmentation Fault
```

6

Buffer Overflow Disassembly

echo code

```
080484f0 <echo>:
 80484f0:  55              push   %ebp
 80484f1:  89 e5           mov    %esp,%ebp
 80484f3:  53              push   %ebx
 80484f4:  8d 5d f8        lea    0xfffffff8(%ebp),%ebx
 80484f7:  83 ec 14        sub    $0x14,%esp
 80484fa:  89 1c 24        mov    %ebx,(%esp)
 80484fd:  e8 ae ff ff ff  call   80484b0 <gets>
 8048502:  89 1c 24        mov    %ebx,(%esp)
 8048505:  e8 8a fe ff ff  call   8048394 <puts@plt>
 804850a:  83 c4 14        add    $0x14,%esp
 804850d:  5b              pop    %ebx
 804850e:  c9              leave
 804850f:  c3              ret
```

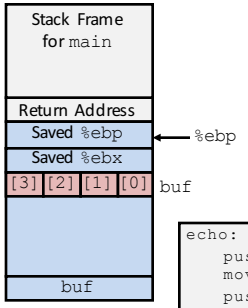
caller code

```
 80485f2:  e8 f9 fe ff ff  call   80484f0 <echo>
 80485f7:  8b 5d fc        mov    0xfffffffc(%ebp),%ebx
 80485fa:  c9              leave
 80485fb:  31 c0           xor    %eax,%eax
 80485fd:  c3              ret
```

7

Buffer Overflow Stack

Before call to gets

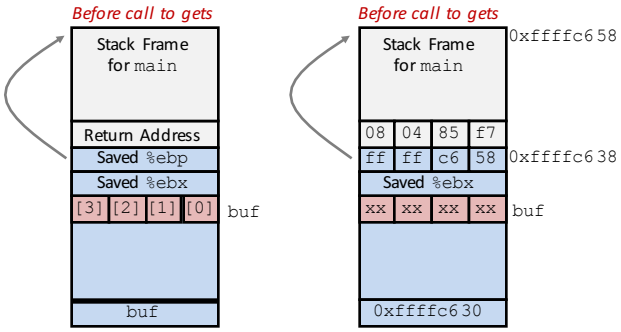



```
/* Echo Line */
void echo() {
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    pushl %ebp              # Save %ebp on stack
    movl  %esp, %ebp
    pushl %ebx              # Save %ebx
    leal  -8(%ebp),%ebx     # Compute buf as %ebp-8
    subl  $20, %esp         # Allocate stack space
    movl  %ebx, (%esp)      # Push buf addr on stack
    call  gets              # Call gets
    . . .
```

8

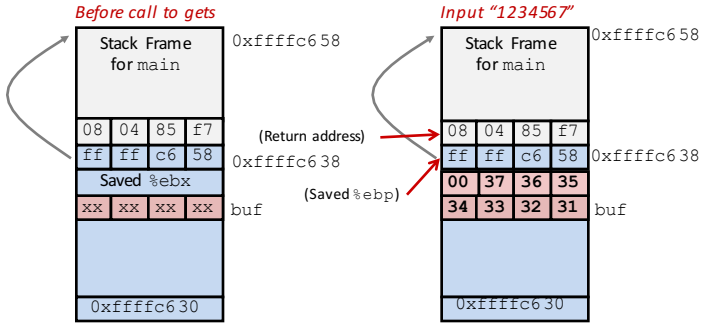
Buffer Overflow Stack Example




```
80485f2:call 80484f0 <echo>
80485f7:mov  0xffffffc(%ebp),%ebx # Return Point
```

9

Buffer Overflow Example #1

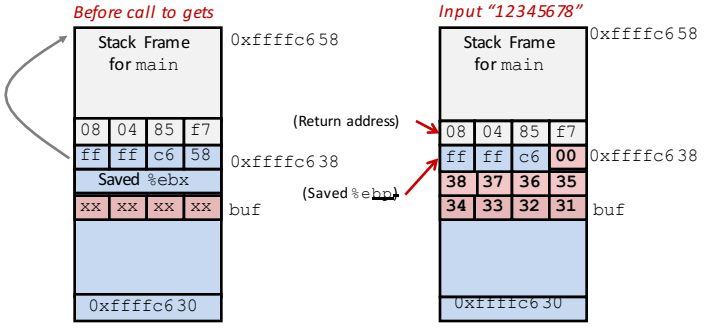


Overflow buf, and corrupt saved %ebx, but no problem, why?

What happens if input has one more byte?

10

Buffer Overflow Example #2

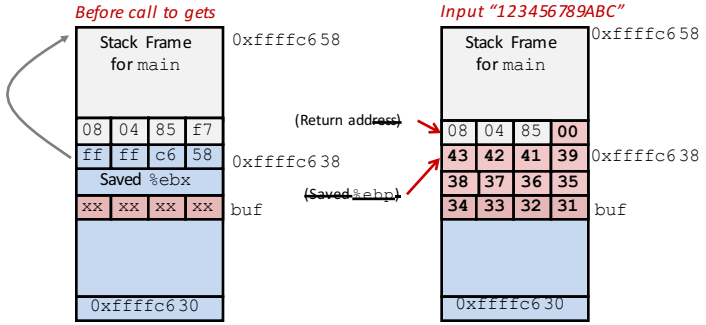


Frame pointer corrupted

```
. . .
804850a: 83 c4 14  add    $0x14,%esp  # deallocate space
804850d: 5b        pop    %ebx        # restore %ebx
804850e: c9        leave              # movl %ebp, %esp; popl %ebp
804850f: c3        ret                # Return
```

11

Buffer Overflow Example #3



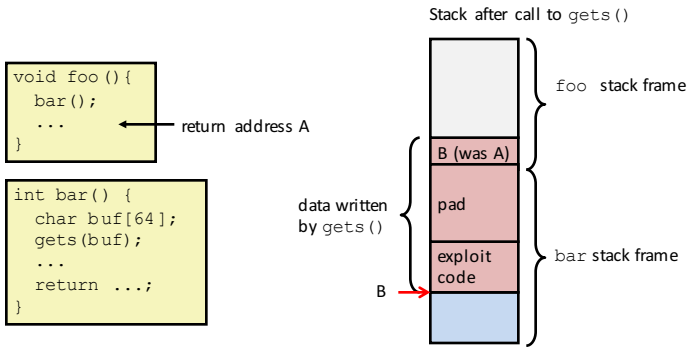
Return address corrupted

```
080485f2: call 80484f0 <echo>
080485f7: mov  0xffffffc(%ebp),%ebx # Return Point
```

Hmmm, what can you do with it?

12

Malicious Use of Buffer Overflow



Input string contains byte representation of executable code

Overwrite return address A with address of buffer (need to know B)

When `bar()` executes `ret`, will jump to exploit code (instead of A)

13

Exploiting Buffer Overflows

Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines.

1988: Internet worm

Early versions of the finger server daemon (fingerd) used **`gets()`** to read the argument sent by the client:

```
finger somebody@cs.wellesley.edu
```

commandline facebook of the 80s!

Attack buffer overflow by sending phony argument:

```
finger "exploit-code padding new-return-address"
```

…

2015: "Ghost" any many more…

Standard C library function **`gethostname()`**

exploitable buffer overflow

14

Avoiding Overflow Vulnerability

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

Use library routines that limit string lengths

- **`fgets`** instead of **`gets`** (second argument to fgets sets limit)
- **`strncpy`** instead of **`strcpy`**
- Don't use **`scanf`** with **`%s`** conversion specification
  - Use **`fgets`** to read the string
  - Or use **`%ns`** where **`n`** is a suitable integer

Other ideas?

15

System-Level Protections

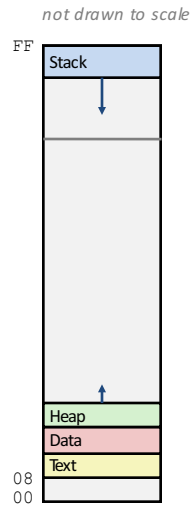
Randomized stack offsets

- At start of program, allocate random amount of space on stack
- Makes it difficult for exploit to predict beginning of inserted code

Techniques to *detect* stack corruption

Nonexecutable code segments

- Allow code to execute only from "text" segment
- Do NOT execute code in stack, data, or heap
- Hardware support needed



16