Chapter 7

Machine-Level Programming V: Advanced Topics

SCS@TJU

Outline

Memory Layout

Buffer Overflow

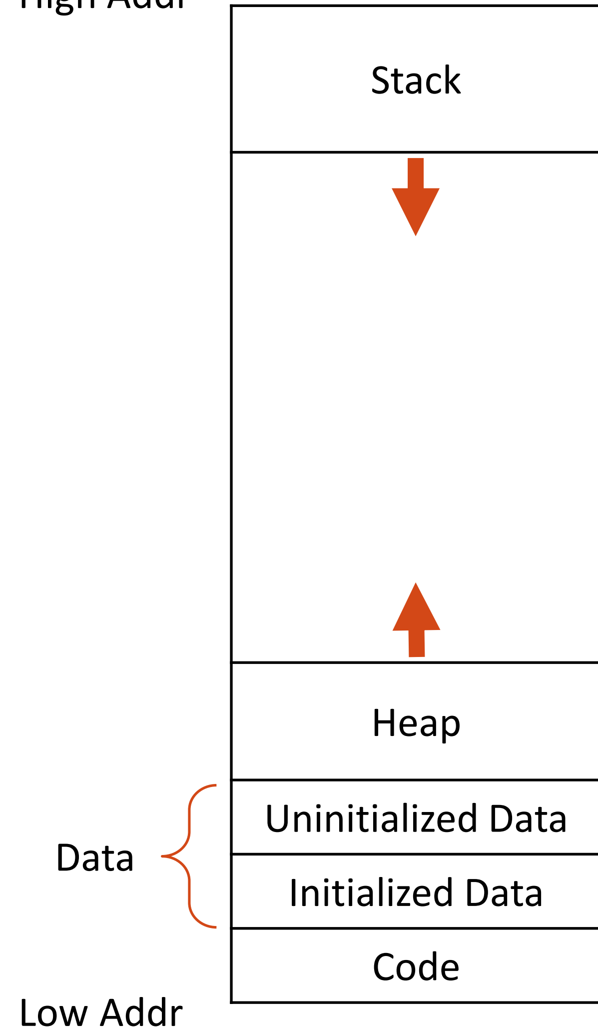
IA32 Linux Memory Layout

Stack
- Runtime stack (8MB Limit)
- E.g., local variables

Heap
- Dynamically allocated storage
- When call `malloc(), calloc(), new()`

Data
- Statically allocated data
- E.g., global vars, static vars, strings

Code (Text)
- Executable machine instructions
- Read-only



Memory Allocation Example

```
int a = 0;
char *p1;

void main()
{
    int b;
    char s[] = “abc”;
    char *p1, p2;
    char *p3 = “123456”;
    static int c = 0;
    p1 = (char *)malloc(10);
    p2 = (char *)malloc(20);
    free(p1);
    free(p2);
}
```

Memory Allocation Example

```
int a = 0;
char *p1;

void main()
{
     int b;
     char s[] = “abc”;
     char *p1, p2;
     char *p3 = “123456”;
     static int c = 0;
     p1 = (char *)malloc(10);
     p2 = (char *)malloc(20);
     free(p1);
     free(p2);
}
```

Stack
Heap
Uninitialized Data
Initialized Data a, “abc”, “123456”, c
Code

Memory Allocation Example

```
int a = 0;
char *p1;

void main()
{
    int b;
    char s[] = “abc”;
    char *p1, p2;
    char *p3 = “123456”;
    static int c = 0;
    p1 = (char *)malloc(10);
    p2 = (char *)malloc(20);
    free(p1);
    free(p2);
}
```

Stack
Heap
Uninitialized Data p1 (global)
Initialized Data a, “abc”, “123456”, c
Code

Memory Allocation Example

```
int a = 0;
char *p1;

void main()
{
    int b;
    char s[] = “abc”;
    char *p1, p2;
    char *p3 = “123456”;
    static int c = 0;
    p1 = (char *)malloc(10);
    p2 = (char *)malloc(20);
    free(p1);
    free(p2);
}
```

Memory segment	Contents
Stack	
Heap	*p1 (local), *p2
Uninitialized Data	p1 (global)
Initialized Data	a, “abc”, “123456”, c
Code	

Memory Allocation Example

```
int a = 0;
char *p1;

void main()
{
    int b;
    char s[] = "abc";
    char *p1, p2;
    char *p3 = "123456";
    static int c = 0;
    p1 = (char *)malloc(10);
    p2 = (char *)malloc(20);
    free(p1);
    free(p2);
}
```

Memory region
Stack b, s[4], p1 (local), p2, p3
Heap *p1 (local), *p2
Uninitialized Data p1 (global)
Initialized Data a, “abc”, “123456”, c
Code

Memory Allocation Example

```
int a = 0;
char *p1;

void main()
{
    int b;
    char s[] = “abc”;
    char *p1, p2;
    char *p3 = “123456”;
    static int c = 0;
    p1 = (char *)malloc(10);
    p2 = (char *)malloc(20);
    free(p1);
    free(p2);
}
```

Stack b, s[4], p1 (local), p2, p3
Heap *p1 (local), *p2
Uninitialized Data p1 (global)
Initialized Data a, “abc”, “123456”, c
Code main

Buffer Overflow

Internet Worm

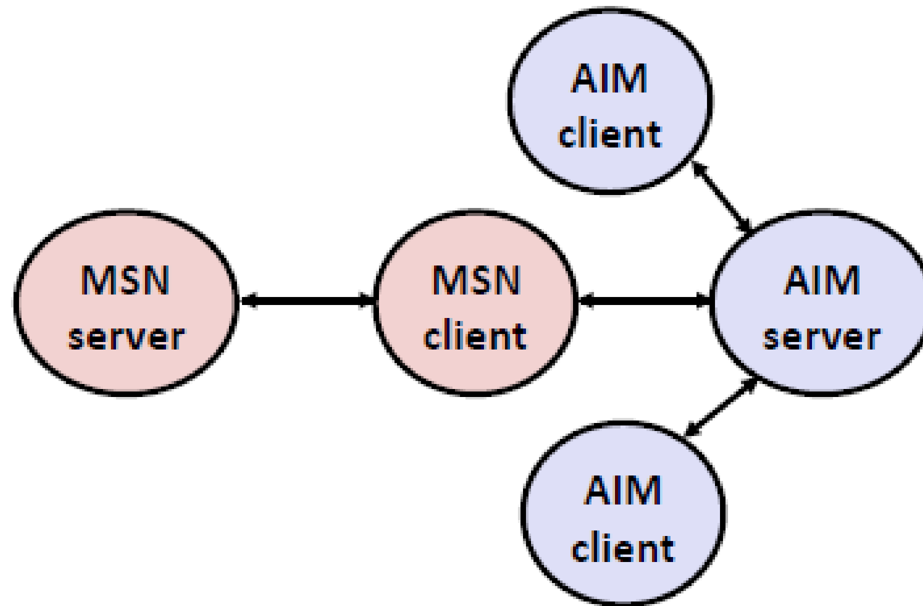
November, 1988

- Internet Worm attacks thousands of Internet hosts
- How did it happen?

IM War

July, 1999

- Microsoft launches MSN Messenger (instant messaging system)
- Messenger clients can access popular AOL Instant Messaging Service (AIM) servers



IM War (cont.)

August 1999

- Mysteriously, Messenger clients can no longer access AIM servers
- Microsoft and AOL begin the IM war:
  - AOL changes server to disallow Messenger clients
  - Microsoft makes changes to clients to defeat AOL changes
  - At least 13 such skirmishes
- How did it happen?

WannaCry / Bitcoin Worm

Wana Decrypt0r 2.0

Ooops, your files have been encrypted!

Chinese (simpli

我的电脑出了什么问题？

您的一些重要文件被我加密保存了。
照片、图片、文档、压缩包、音频、视频文件、exe文件等，几乎所有类型的文件都被加密了，因此不能正常打开。
这和一般文件损坏有本质上的区别。您大可在网上找找恢复文件的方法，我敢保证，没有我们的解密服务，就算老天爷来了也不能恢复这些文档。

有没有恢复这些文档的方法？

当然有可恢复的方法。只能通过我们的解密服务才能恢复。我以人格担保，能够提供安全有效的恢复服务。
但这是收费的，也不能无限期的推迟。
请点击 <Decrypt> 按钮，就可以免费恢复一些文档。请您放心，我是绝不会骗你的。
但想要恢复全部文档，需要付款点费用。
是否随时都可以固定金额付款，就会恢复的吗，当然不是，推迟付款时间越长对你不利。
最好3天之内付款费用，过了三天费用就会翻倍。
还有，一个礼拜之内未付款，将会永远恢复不了。
对了，忘了告诉你，对半年以上没钱付款的穷人，会有活动免费恢复，能否轮

Payment will be raised on
5/16/2017 02:26:59
Time Left
02:22:35:15

Your files will be lost on
5/20/2017 02:26:59
Time Left
06:22:35:15

bitcoin ACCEPTED HERE

Send $300 worth of bitcoin to this address:
13AM4VW2dhxYgXeQepoHkHSQuy6NgaEb94
Copy

About bitcoin
How to buy bitcoins?
Contact Us

Bai百科

Check Payment
Decrypt

Worms and Viruses

Worm: A program that

- Can run by itself
- Can propagate a fully working version of itself to other computers

Virus: Code that

- Add itself to other programs
- Cannot run independently

Both are (usually) designed to spread among computers and to wreak havoc

Reason

The Internet Worm and AOL/Microsoft War were both based on stack buffer overflow exploits!

- Many library functions do not check argument sizes
- Allows target buffers to overflow

Vulnerability(弱点) : 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

Deprecated Functions in C Library

Similar problems with other library functions

- `strcpy`, `strcat` : copy strings of arbitrary length
- `scanf`, `fscanf`, `sscanf` : when given %s conversion specification

Deprecated Functions
`char *strcpy(char* dest, const char *src);`
`char *strcat(char *dest, const char *src);`
`int scanf(const char *format, [argument...]);`
`int fscanf(FILE *stream, const char *format, [argument...]);`
`int sscanf(const char *buffer, const char *format, [argument...]);`

Vulnerable Buffer Code

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

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

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

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

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

Buffer Overflow Disassembly

echo

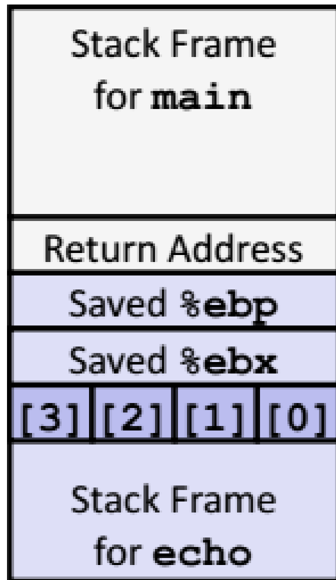
```
80485c5:    55                  push %ebp
80485c6:    89 e5               mov  %esp, %ebp
80485c8:    53                  push %ebx
80485c9:    83 ec 14            sub  $0x14, %esp
80485cc:    8d 5d f8            lea  0xfffffff8(%ebp), %ebx
80485cf:    89 1c 24            mov  %ebx, (%esp)
80485d2:    e8 9e ff ff ff      call 8048575 <gets>
80485d7:    89 1c 24            mov  %ebx, (%esp)
80485da:    e8 05 fe ff ff      call 80483e4 <puts@plt>
80485df:    83 c4 14            add  $0x14, %esp
80485e2:    5b                  pop  %ebx
80485e3:    5d                  pop  %ebp
80485e4:    c3                  ret
```

main

```
80485eb:    e8 d5 ff ff ff      call 80485c5 <echo>
80485f0:    c9                  leave
80485f1:    c3                  ret
```

Buffer Overflow Stack

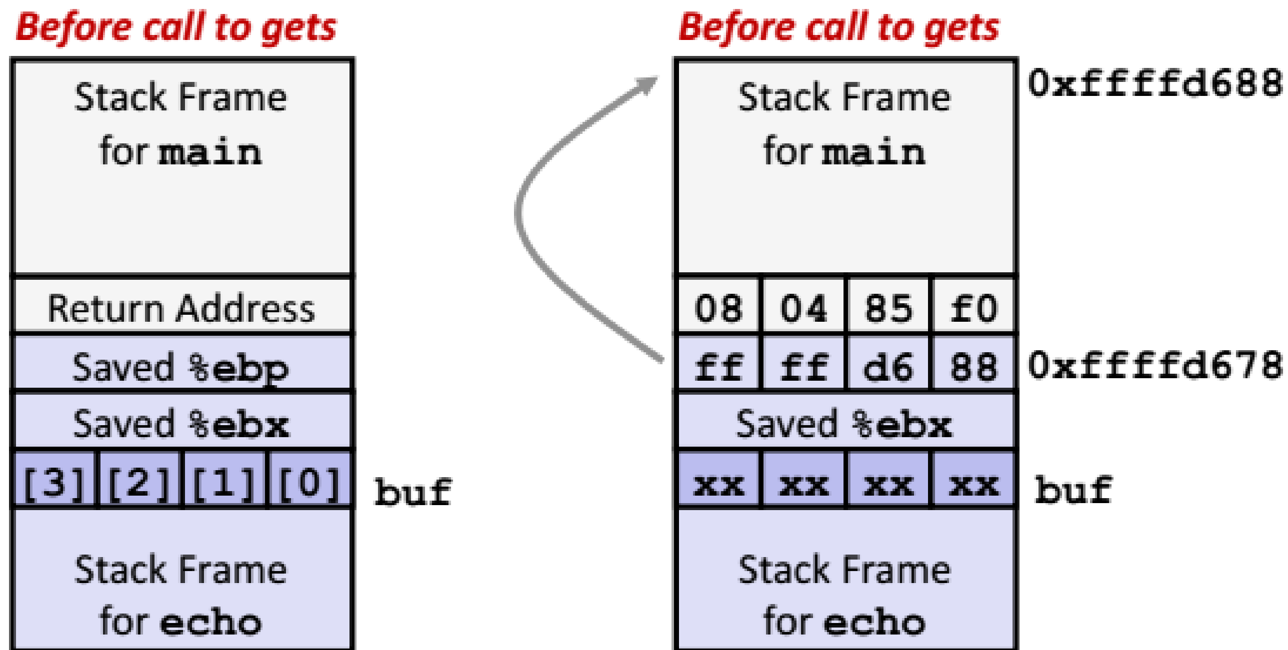
Before call to gets




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

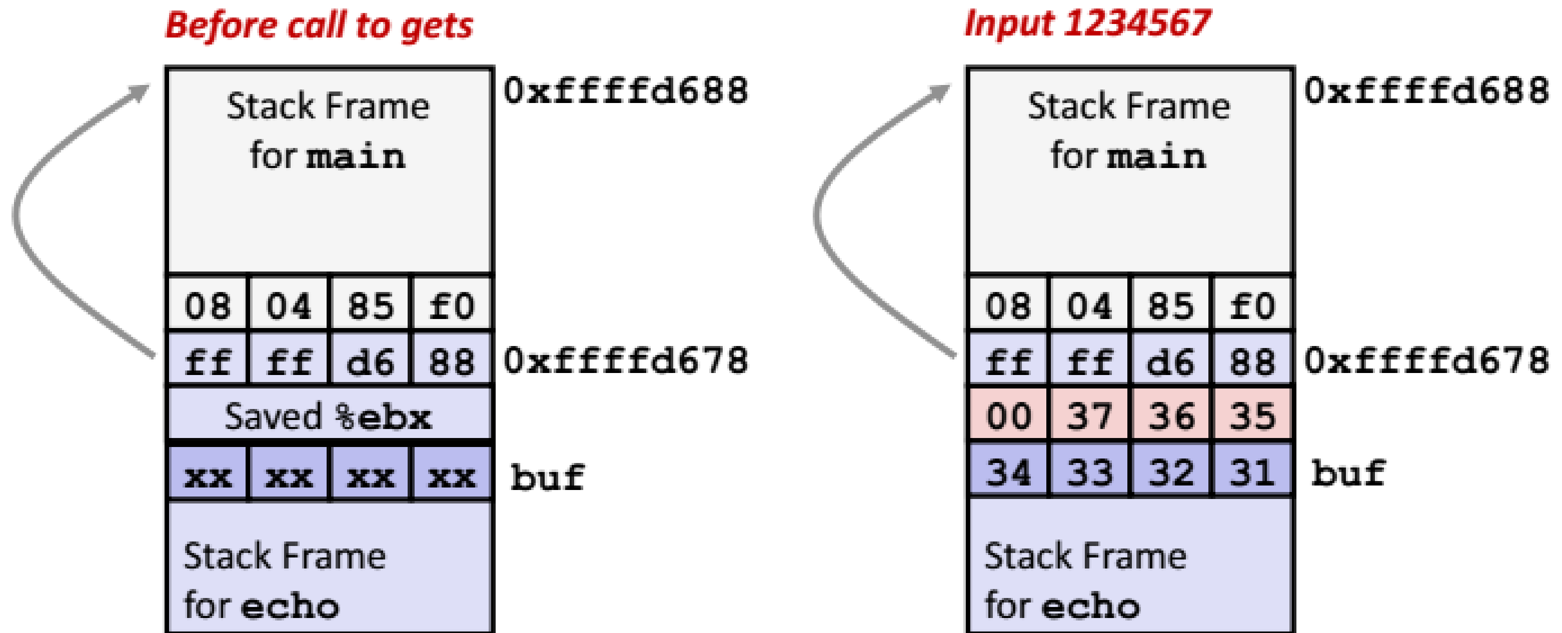
```
echo:
    pushl %ebp              # Save %ebp on stack
    movl  %esp, %ebp
    pushl %ebx              # Save %ebx
    subl  $20, %esp         # Allocate stack space
    leal  -8(%ebp),%ebx     # Compute buf as %ebp-8
    movl  %ebx, (%esp)      # Push buf on stack
    call  gets              #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

Buffer Overflow Stack Example




```
80485eb: e8 d5 ff ff ff       call 80485c5 <echo>
80485f0: c9                   leave
```

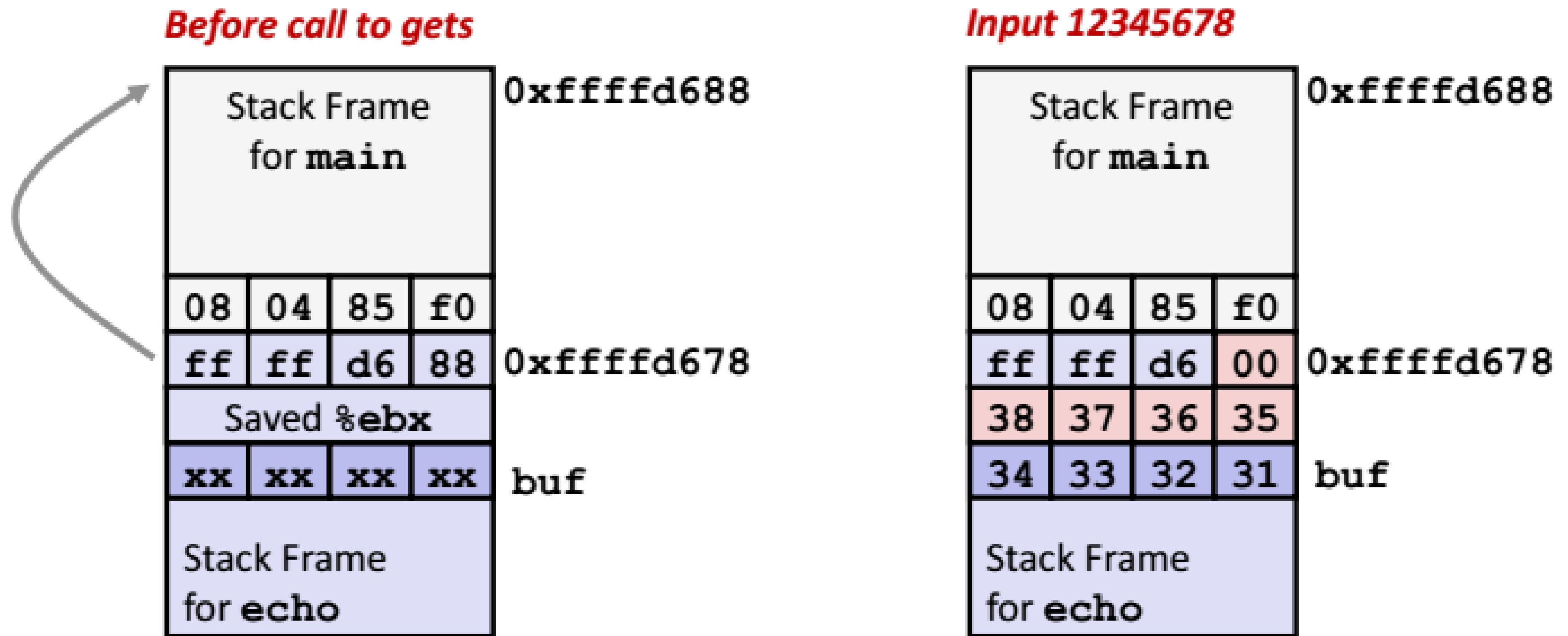
Buffer Overflow Example #1



Input: "1234567"

Overflow buf, and corrupt %ebx, but no problem

Buffer Overflow Example #2



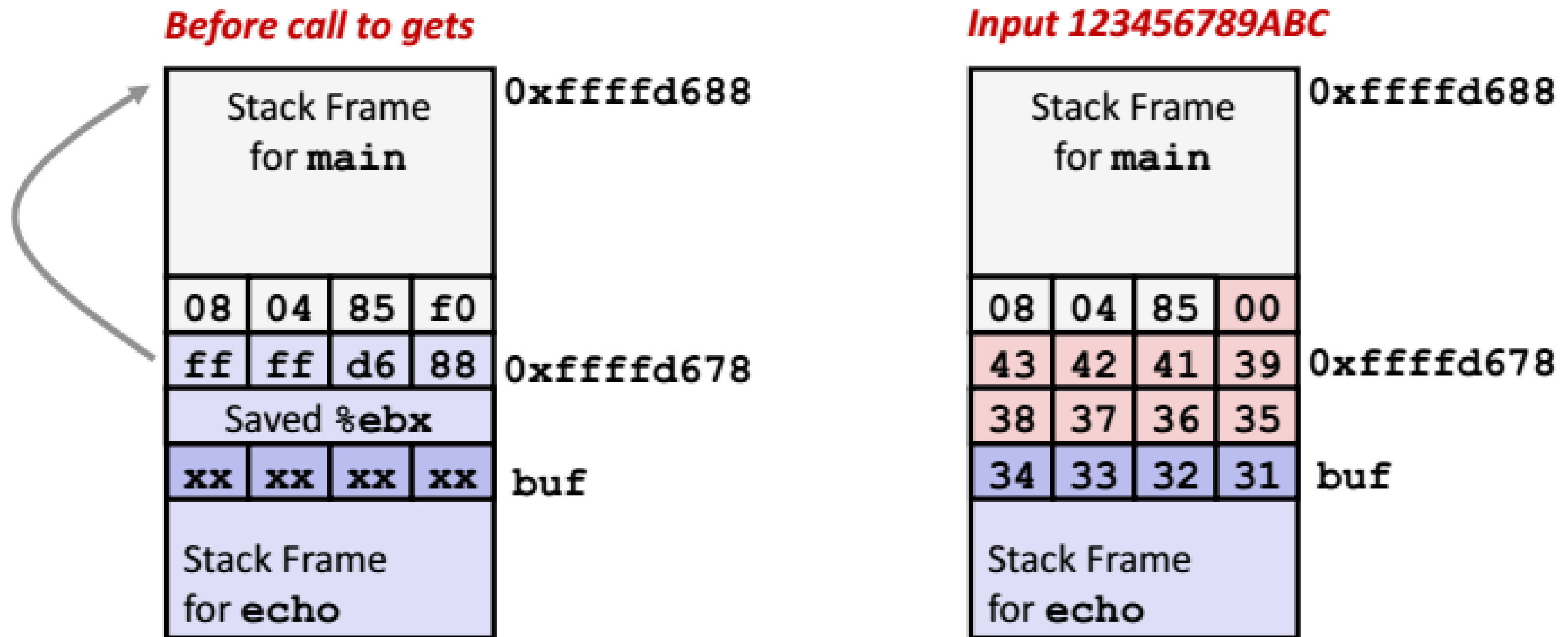
Input : “12345678”
Base pointer corrupted

```
...

80485eb: e8 d5 ff ff ff  call 80485c5 <echo>
80485f0: c9              leave              # Set %ebp to corrupted value
80485f1: c3              ret
```

```
leave  =>   movl   %ebp, %esp
            popl   %ebp
```

Buffer Overflow Example #3



Input: "123456789ABC"
Return address corrupted

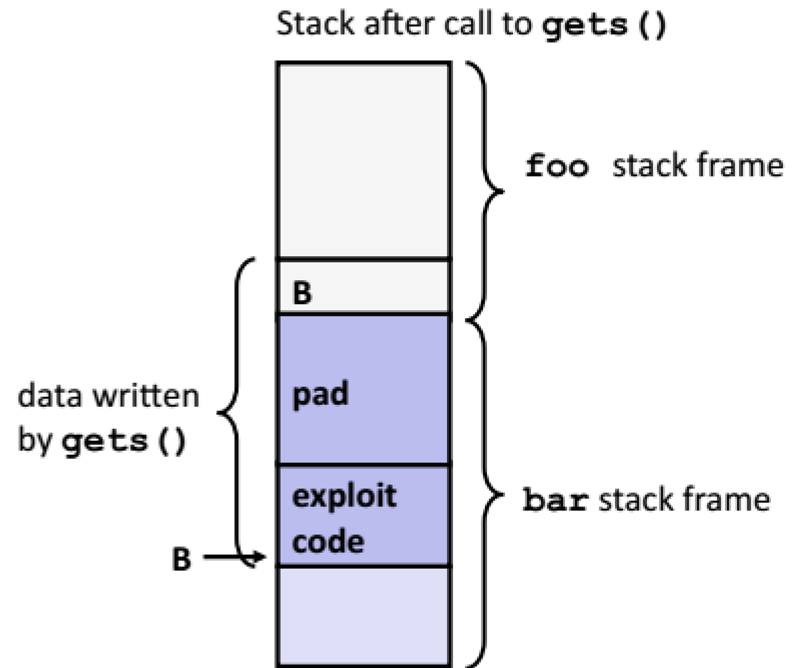
```
80485eb: e8 d5 ff ff ff  call 80485c5 <echo>
80485f0: c9              leave                   # Desired return point
```

Malicious(恶意的) Use of Buffer Overflow

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

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

Return address A



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

Exploits Based on Buffer Overflows

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

Internet worm

- Early versions of the finger server (fingerd) used gets() to read the argument sent by the client:
  - **`finger liyoumeng@tju.edu.cn`**
- Worm attacked fingerd server by sending phony argument:
  - **`finger “exploit-code  padding  new-return-address”`**
- Exploit code: executed a root shell on the victim machine with a direct TCP connection to the attacker

Exploits Based on Buffer Overflows

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

IM War

- AOL exploited existing buffer overflow bug in AIM clients
- Exploit code: returned 4-byte signature (the bytes at some location in the AIM client) to server
- When Microsoft changed code to match signature, AOL changed signature location

Date: Wed, 11 Aug 1999 11:30:57 -0700 (PDT)
From: Phil Bucking <philbucking@yahoo.com>
Subject: AOL exploiting buffer overrun bug in their own software!
To: rms@pharlap.com

Mr. Smith,

I am writing you because I have discovered something that I think you might find interesting because you are an Internet security expert with experience in this area. I have also tried to contact AOL but received no response.

I am a developer who has been working on a revolutionary new instant messaging client that should be released later this year.
...
It appears that the AIM client has a buffer overrun bug. By itself this might not be the end of the world, as MS surely has had its share. But AOL is now *exploiting their own buffer overrun bug* to help in its efforts to block MS Instant Messenger.
....
Since you have significant credibility with the press I hope that you can use this information to help inform people that behind AOL's friendly exterior they are nefariously compromising peoples' security.

Sincerely,
Phil Bucking
Founder, Bucking Consulting
philbucking@yahoo.com

It was later determined that this email originated from within Microsoft!

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 gets
- 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

System-Level Protections

Randomized stack offsets

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

Nonexecutable code segments

- In traditional x86, can mark region of memory as either “read-only” or “writeable”
  - Can execute anything readable
- x86-64 added explicit “execute” permission

Stack Canaries

Idea

- Place special value ("canary") on stack just beyond buffer
- Check for corruption before exiting function

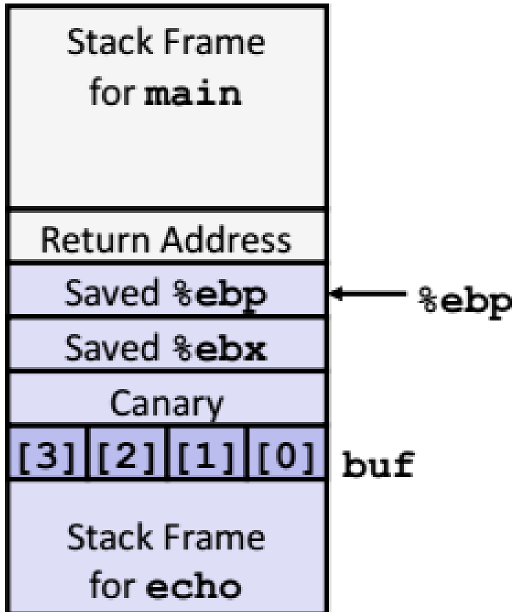
GCC Implementation

- `-fstack-protector`
- `-fstack-protector-all`

Protected Buffer Disassembly

```
804864d: 55                     push %ebp
804864e: 89 e5                  mov  %esp,%ebp
8048650: 53                     push %ebx
8048651: 83 ec 14               sub  $0x14,%esp
8048654: 65 a1 14 00 00 00      mov  %gs:0x14,%eax
804865a: 89 45 f8               mov  %eax,0xfffffff8(%ebp)
804865d: 31 c0                  xor  %eax,%eax
804865f: 8d 5d f4               lea  0xfffffff4(%ebp),%ebx
8048662: 89 1c 24               mov  %ebx,(%esp)
8048665: e8 77 ff ff ff         call 80485e1 <gets>
804866a: 89 1c 24               mov  %ebx,(%esp)
804866d: e8 ca fd ff ff         call 804843c <puts@plt>
8048672: 8b 45 f8               mov  0xfffffff8(%ebp),%eax
8048675: 65 33 05 14 00 00 00   xor  %gs:0x14,%eax
804867c: 74 05                  je   8048683 <echo+0x36>
804867e: e8 a9 fd ff ff         call 804842c <FAIL>
8048683: 83 c4 14               add  $0x14,%esp
8048686: 5b                     pop  %ebx
8048687: 5d                     pop  %ebp
8048688: c3                     ret
```

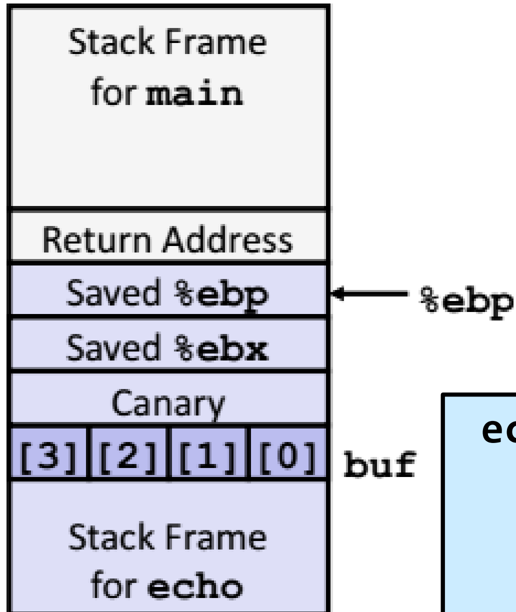
Setting Up Canary




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

```
echo:
    . . .
    movl %gs:20, %eax    # Get canary
    movl %eax, -8(%ebp)  # Put on stack
    xorl %eax, %eax      # Erase canary
    . . .
```

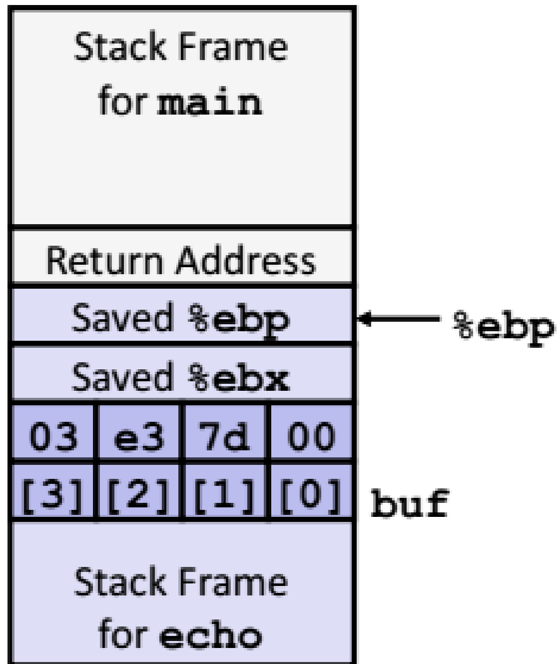
Checking Canary

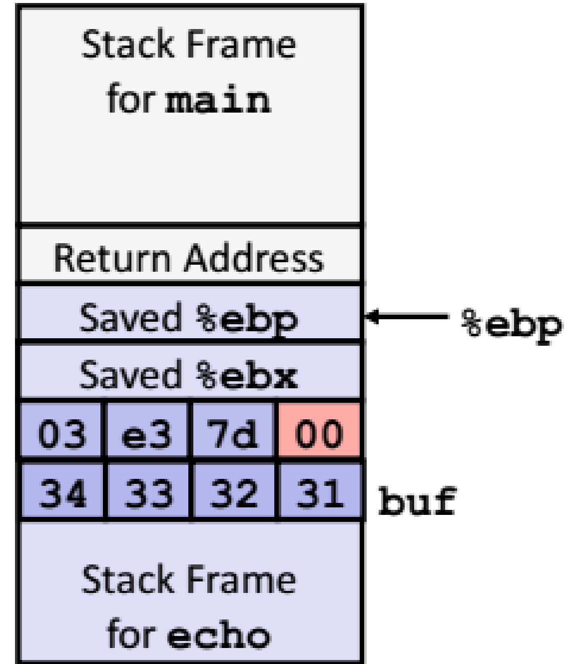



```
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
  . . .
  movl -8(%ebp), %eax    # Retrieve from stack
  xorl %gs:20, %eax      # Compare with Canary
  je   .L24              # Same: skip ahead
  call __stack_chk_fail  # ERROR
.L24:
  . . .
```

Canary Example





Summary

Memory Layout

Buffer Overflow

- Vulnerability
- Protection