Buffer Overflow

Jo, Heeseung

IA-32/Linux Memory Layout

Stack

- Runtime stack (8MB limit)

Heap

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

DLLs (shared libraries)

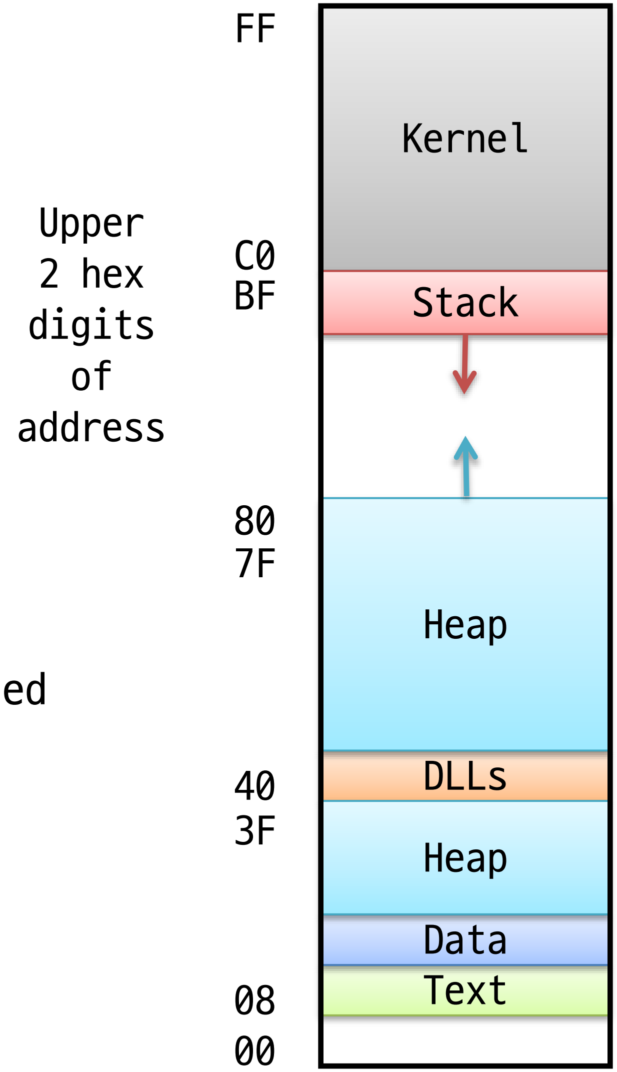
- Dynamically linked libraries
- Library routines (e.g., printf, gets)
- Linked into object code when first executed

Data

- Statically allocated data
- e.g., arrays & strings declared in code

Text

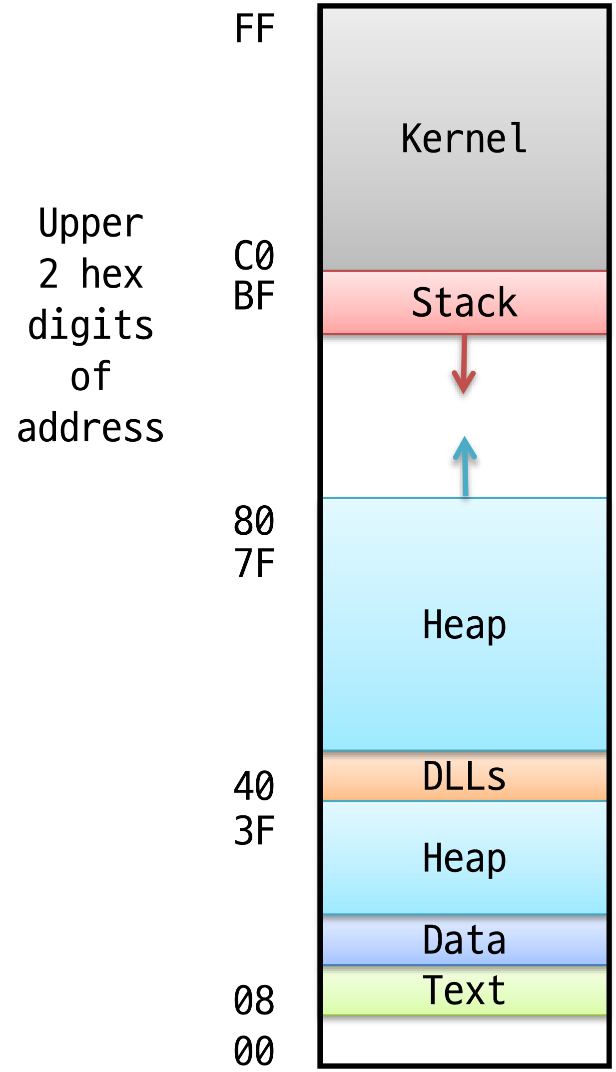
- Executable machine instructions
- Read-only



IA-32/Linux Memory Layout

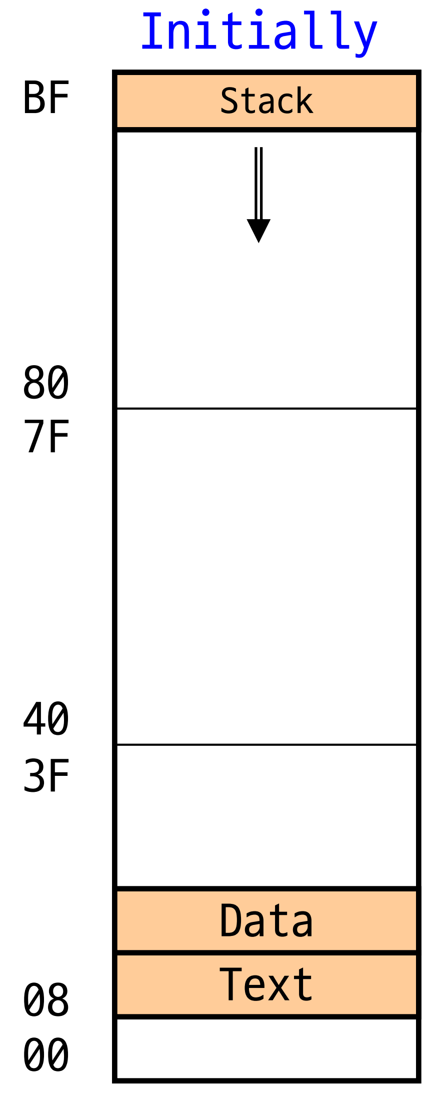
Where are the variables located?

```
int e = 7;

int main() {
  int i = 3;
  static int k;
  char c[256];
  char p*;

  p=(char *)malloc(256);
  printf("%p\n", p);
  i=a();

  exit(0);
}

int b = 5;
int a() {
  int i = 4;
  return b;
}
```



Text & Stack Example

```
(gdb) break main
(gdb) run
  Breakpoint 1, 0x0804856f in main ()
(gdb) print $esp
  $3 = (void *) 0xbffffc78
```



Vulnerable Buffer Code

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

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

```
$ ./bufdemo
Type:1234
1234

$ ./bufdemo
Type: 12345
Segmentation Fault

$ ./bufdemo
Type: 12345678
Segmentation Fault
```

What's wrong?

String Library Code

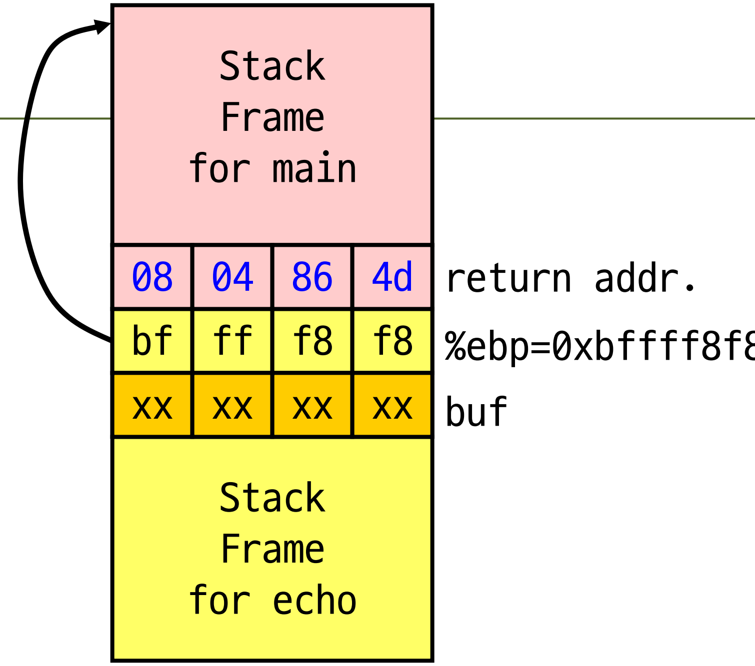
Implementation of Unix function gets()

- No way to specify limit on # of characters to read

```
/* Get string from stdin */
char *gets(char *dest) {
    int c = getc();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getc();
    }
    *p = '\0';
    return dest;
}
```

- Similar problems with other Unix functions
  - strcpy: copies string of arbitrary length
  - scanf/fscanf/sscanf, given %s conversion specification

Buffer Overflow (1)

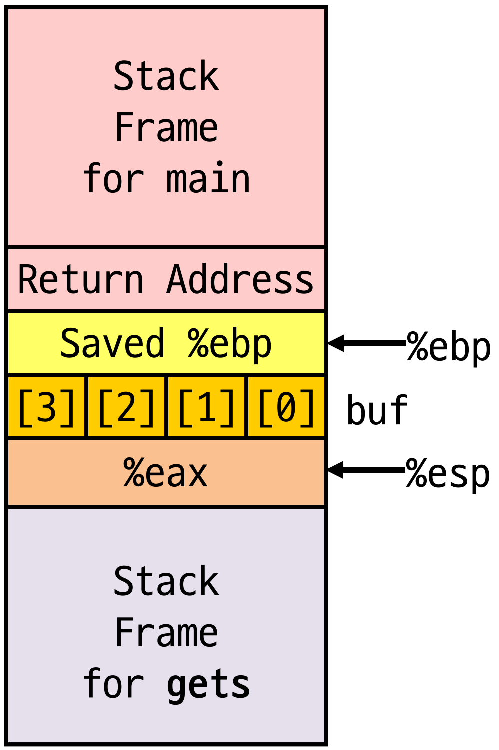



```
unix> gdb bufdemo
(gdb) break echo
Breakpoint 1 at 0x8048583
(gdb) run
Breakpoint 1, 0x8048583 in echo ()
(gdb) print /x *(unsigned *)$ebp
$1 = 0xbffff8f8
(gdb) print /x *((unsigned *)$ebp + 1)
$3 = 0x0804864d
```

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

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

```
08048648:   call 804857c <echo>
0804864d:   movl $0, %eax             # Return Point
```

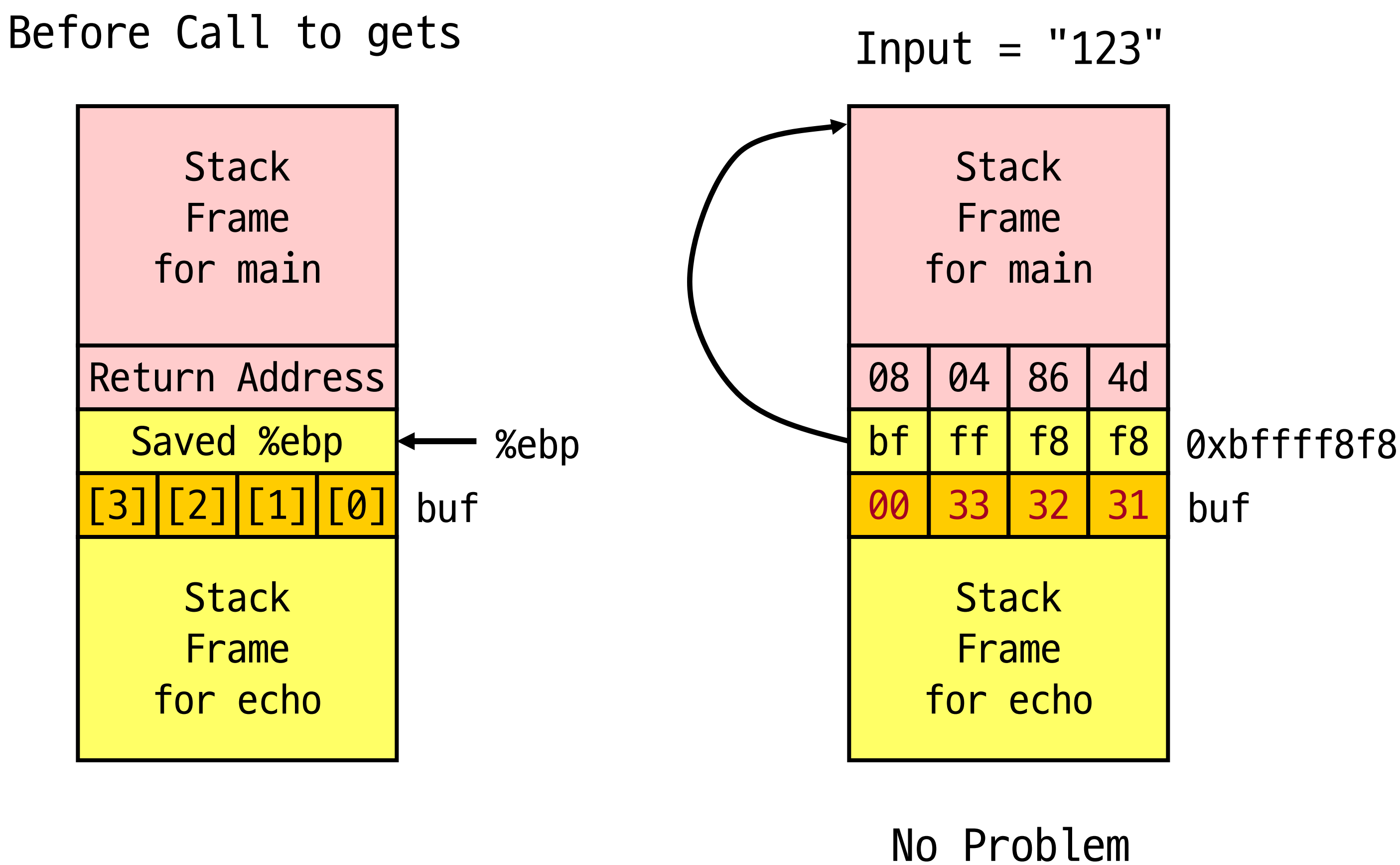
Buffer Overflow (2)



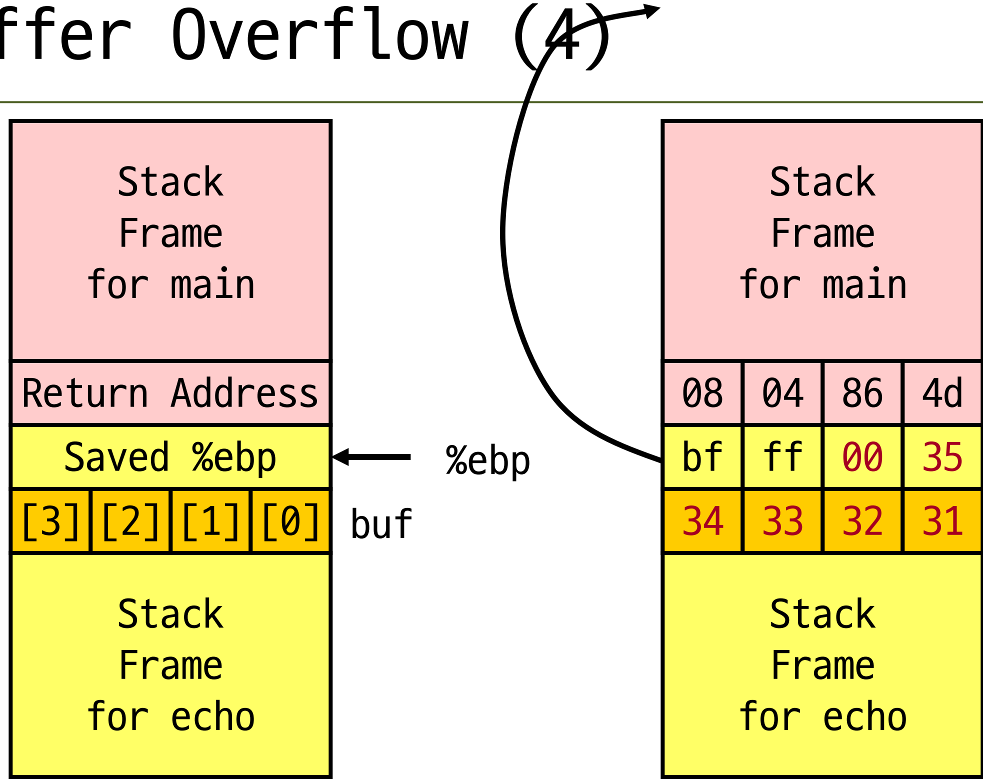

```
/* Echo Line */
void echo()
{
    char buf[4];        /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
 pushl %ebp                 # Save %ebp on stack
 movl %esp,%ebp
 leal -4(%ebp),%eax         # compute buf as %ebp-4
 movl %eax,(%esp)           # save to the stack
 call gets                  # Call gets
   . . .
```

Buffer Overflow (3)



Buffer Overflow (4)



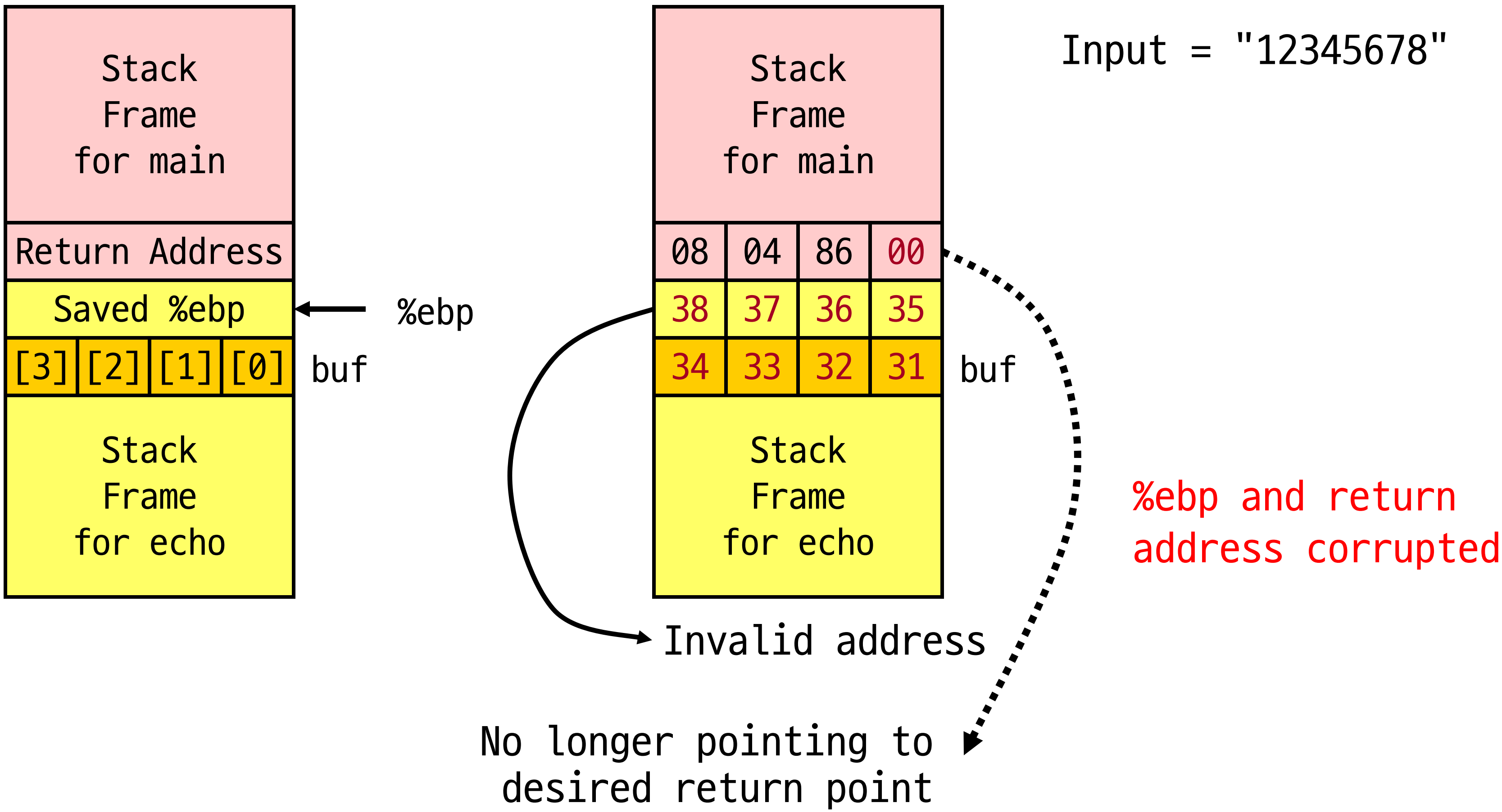
Input = "12345"

Saved value of %ebp set to 0xbfff0035

Bad news when later attempt to restore %ebp

```
 . . .                                                    <echo code>
8048600:   call   80482c4            # gets
8048605:   leal   -4(%ebp),%eax
8048608:   movl   %eax,(%esp)
804860b:   call   80482d4            # puts
8048610:   leave                     # movl %ebp, %esp; popl %ebp
8048611:   ret
```

Buffer Overflow (5)




```
8048648:    call 804857c <echo>
804864d:    mov  $0,%eax            # Return Point
```

Buffer Overflow Attack (1)

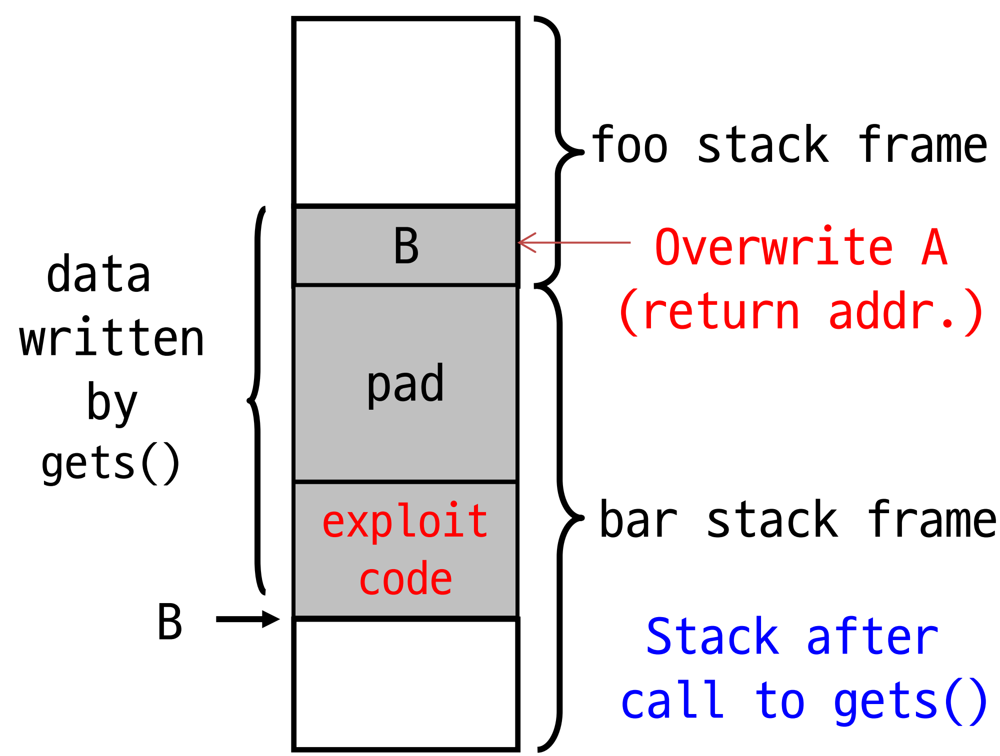
Malicious use of buffer overflow

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

return address A

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

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



Buffer Overflow Attack (2)

Exploits based on buffer overflows

- Buffer overflow bugs allow remote machines to execute arbitrary code

Internet worm

- Early versions of the finger server (fingerd) used gets() to read the argument sent by the client:
  - finger kildong@email.com
- 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

Code Red Worm (1)

History

- June 18, 2001. Microsoft announces buffer overflow vulnerability in IIS Internet server
- July 19, 2001. Over 250,000 machines infected by new virus in 9 hours
- White house must change its IP address
- Pentagon shut down public WWW servers for day

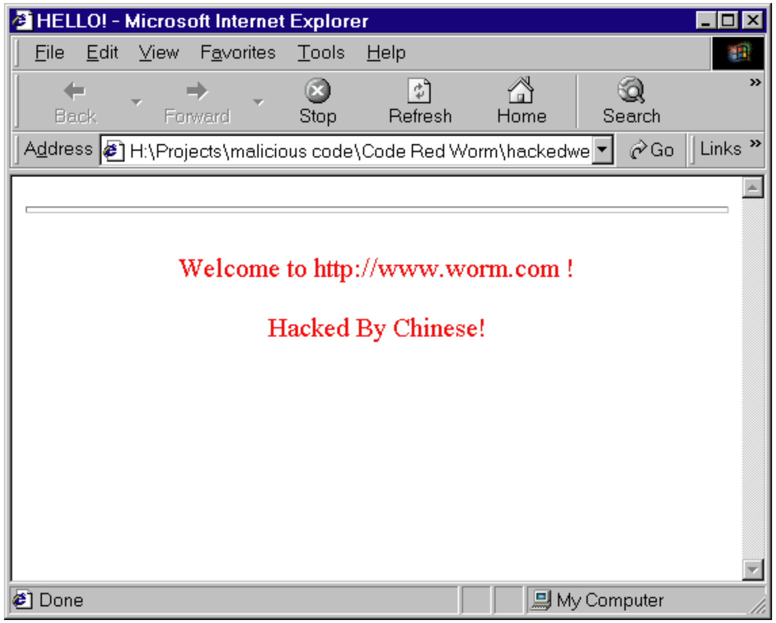
Received strings of form

```
GET /default.ida?NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN....NNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN%u9090%u6858%ucbd3%u7801%u9090%u6858%uc
bd3%u7801%u9090%u6858%ucbd3%u7801%u9090%u9090%u8190%u00c3%u0003%u8b00%
u531b%u53ff%u0078%u0000%u00=a
HTTP/1.0" 400 325 "-" "-"
```

Code Red Worm (2)

Code Red exploit code

- Starts 100 threads running
- Spread self
  - Between 1st & 19th of month
  - Generate random IP addresses & send attack string
- Denial of service attack to www.whitehouse.gov
  - Between 21st & 27th of month
  - Send 98,304 packets; sleep for 4.5 hours; repeat
- Deface server's home page
  - After waiting 2 hours



Code Red Worm (3)

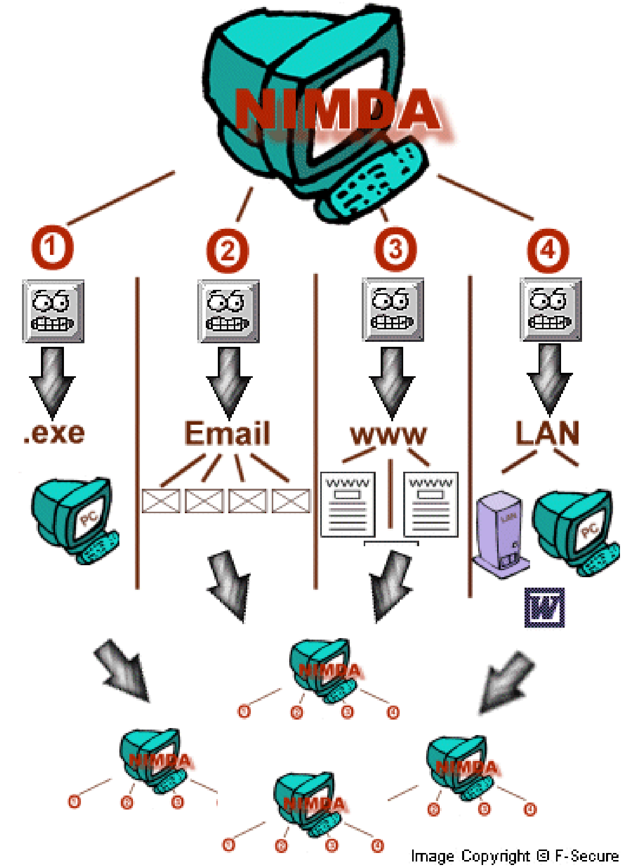
Code Red effects

- Later version even more malicious
  - Code Red II
  - As of April 2002, over 18,000 machines infected
  - Still spreading
- Paved way for NIMDA
  - Variety of propagation methods
  - One was to exploit vulnerabilities left behind by Code Red II

Nimda Worm

Nimda (2001)

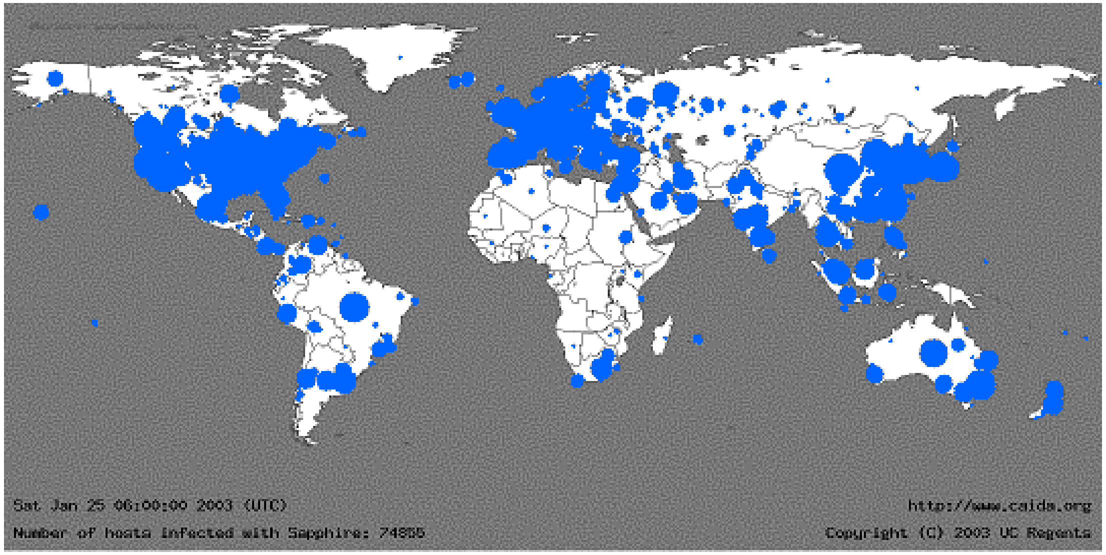
- Five different infection methods:
  - Via e-mail
  - Via open network shares
    - Window XP SP1 version
  - Via browsing of compromised web sites
  - Exploitation of various Microsoft IIS 4.0/5.0 directory traversal vulnerabilities
  - Via back doors left behind by the "Code Red II" and "Sadmind/IIS" worms
- One of the most widespread virus/worm



SQL Slammer Worm

SQL slammer (2003)

- Exploited two buffer overflow bugs in Microsoft's SQL Server and Desktop Engine
- Infected 75,000 victims within 10 minutes
- Generate random IP addresses and send itself out to those addresses, slowing down Internet traffic dramatically
- 1/25 nationwide Internet shutdown in South Korea



30 minutes after release

Avoiding Buffer Overflow

Use library routines that limit string lengths

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

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

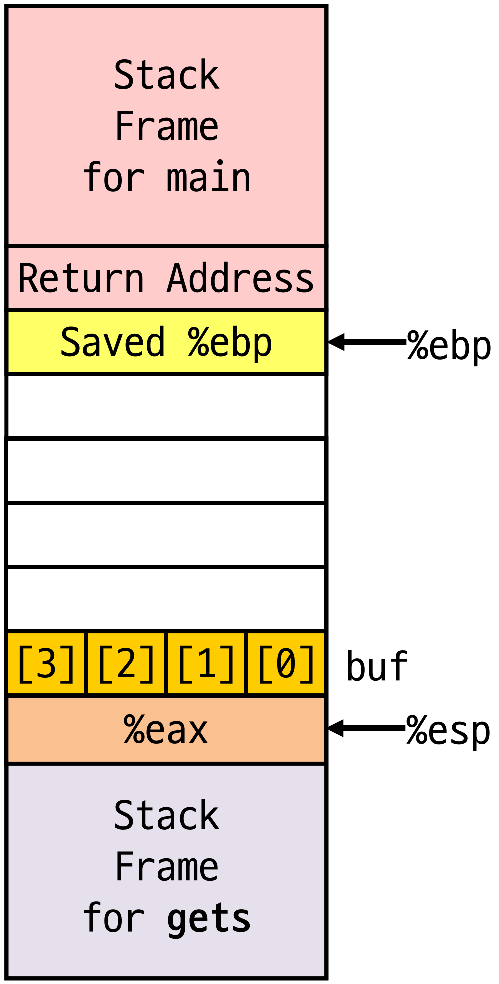
Why does not eliminate vulnerable functions?

System-Level Protections

Address Space Layout Randomization (ASLR)

- 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

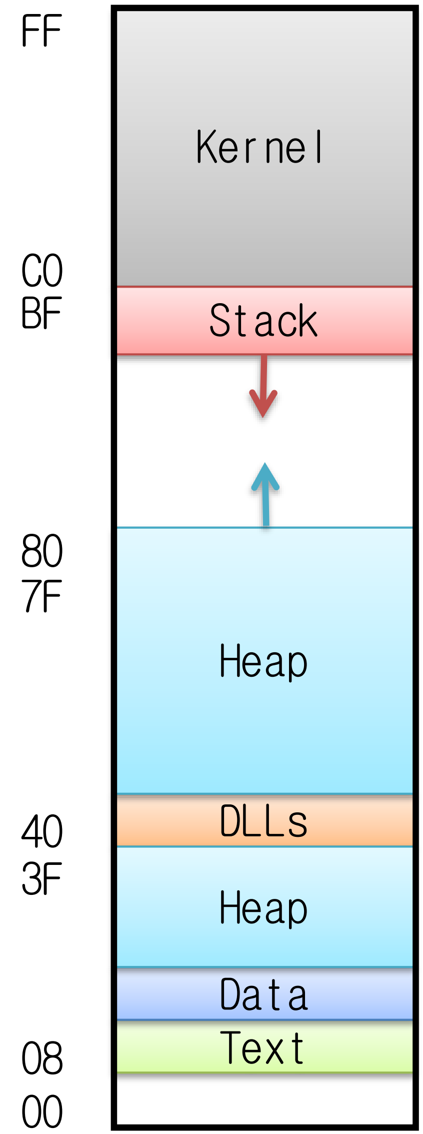
```
echo:
 pushl %ebp                 # Save %ebp on stack
 movl %esp,%ebp
 subl $24,%esp              # Allocate space
 leal -20(%ebp),%eax        # compute buf as %ebp-4
 movl %eax,(%esp)           # save to the stack
 call gets                  # Call gets
    . . .
```



System-Level Protections

Executable space protection

- Mark certain areas of memory as non-executable
- Hardware assistance:
  - Intel NX (No eXecute) bit
  - AMD XD (eXecute Disable) bit



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

```
unix>./bufdemo-protected
Type a string:123
123
```

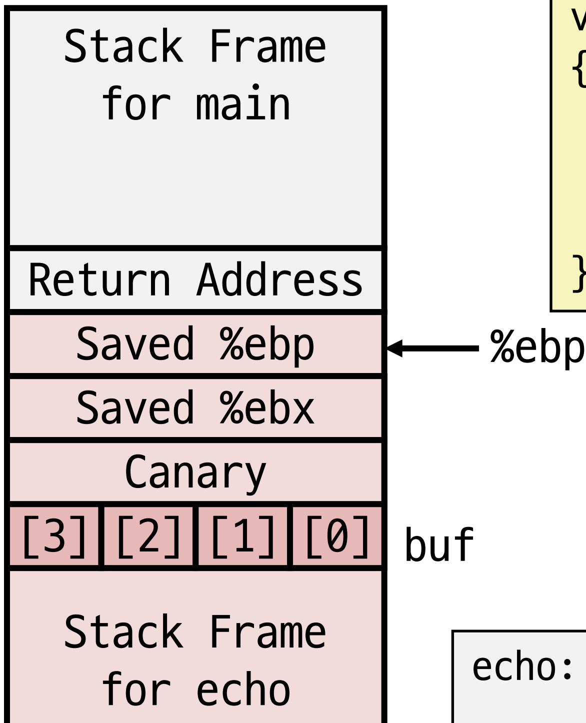
```
unix>./bufdemo-protected
Type a string:12345
*** stack smashing detected ***
```

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

Before call to gets



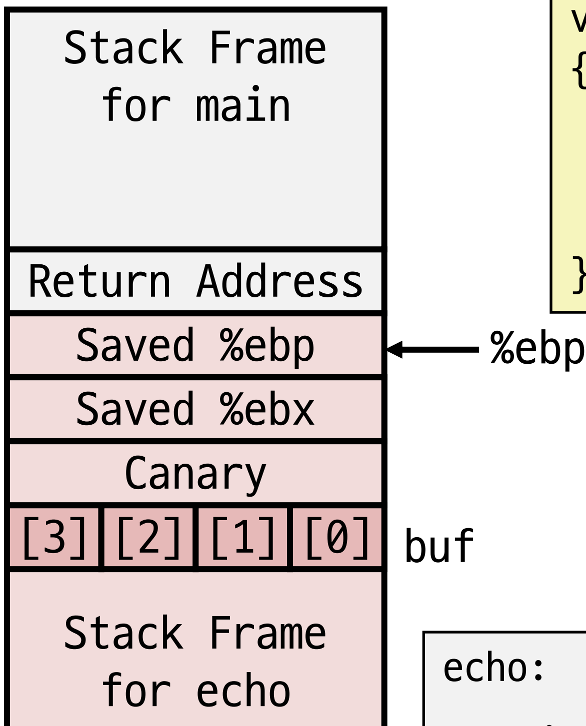

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

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

%gs : one of segment register

Checking Canary

Before call to gets

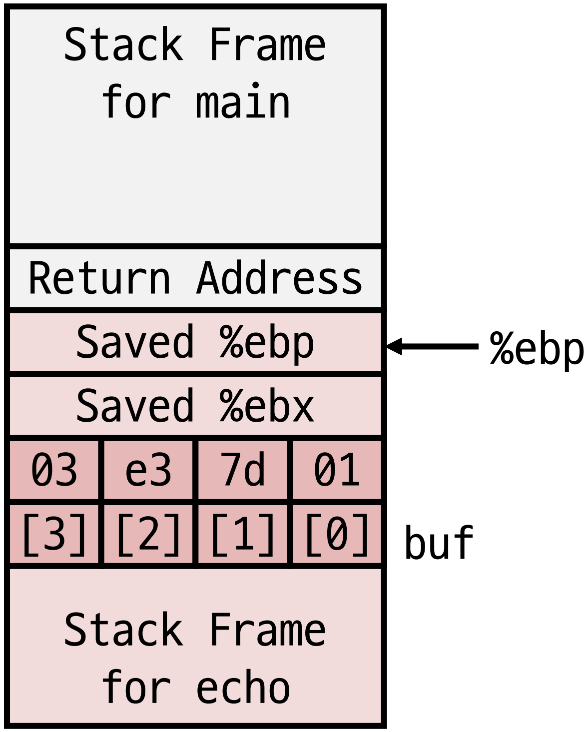



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

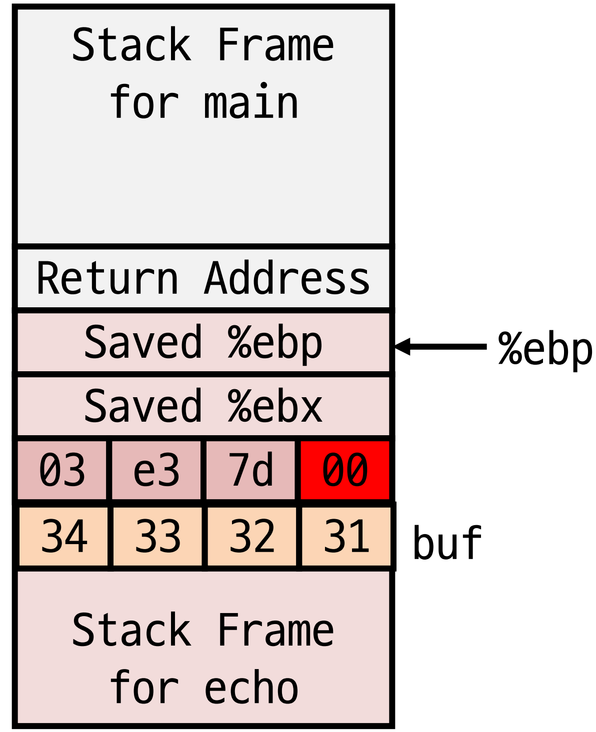
```
echo:
    . . .
    movl    -8(%ebp), %eax       # Retrieve from stack
    xorl    %gs:20, %eax         # Compare with Canary
    je      .L24                 # Same: skip ahead
    call    __stack_chk_fail     # ERROR
.L24:
    . . .
```

Canary Example

Before call to gets



Input 1234




```
(gdb) break echo
(gdb) run
(gdb) stepi 3
(gdb) print /x *((unsigned *) $ebp - 2)
$1 = 0x03e37d01
```

False negative possible !
but, low possibility

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

Summary

Memory layout

- OS/machine dependent (including kernel version)
- Basic partitioning:
  - stack, data, text, heap, DLL found in most machines

Avoiding buffer overflow vulnerability

- Important to use library routines that limit string lengths
- Why does not eliminate vulnerable functions?

Working with strange code

- Important to analyze nonstandard cases
  - e.g., what happens when stack corrupted due to buffer overflow
- Helps to step through with GDB