CIS 450
Computer Architecture and Organization

Lecture 13: Buffer Overflow

Mitch Neilsen
neilsen@ksu.edu

219D Nichols Hall

Topics

Data/Control

- Buffer overflow
- Exploits

Buffer Overflow Attacks

November, 1988

- First Internet Worm spread over then-new Internet
- Many university machines compromised
- No malicious effect

Today

- Buffer overflow is still the initial entry for over 50% of network-based attacks

String Library Code

- Implementation of Unix function `gets()`
  - No way to specify limit on number 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`, when given %s conversion specification

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;
}
```

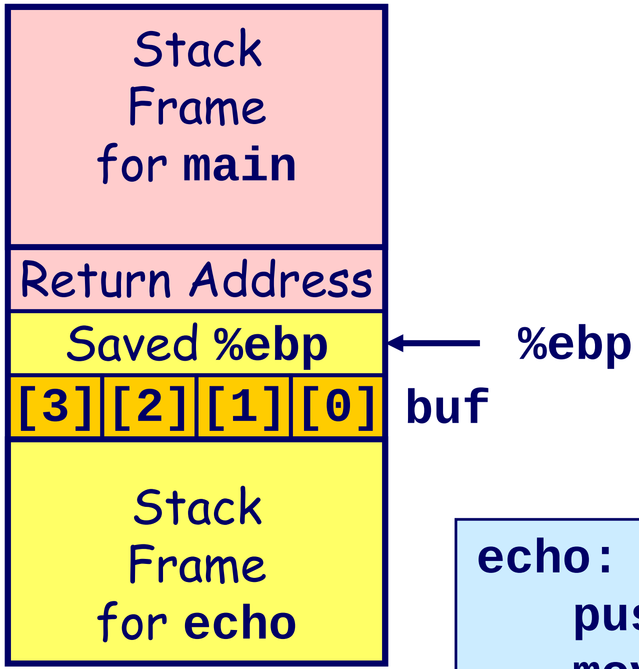
Buffer Overflow Executions

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

```
unix>./bufdemo
Type a string:12345
12345
→ note valid output, bad input
```

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

Buffer Overflow Stack (IA32)

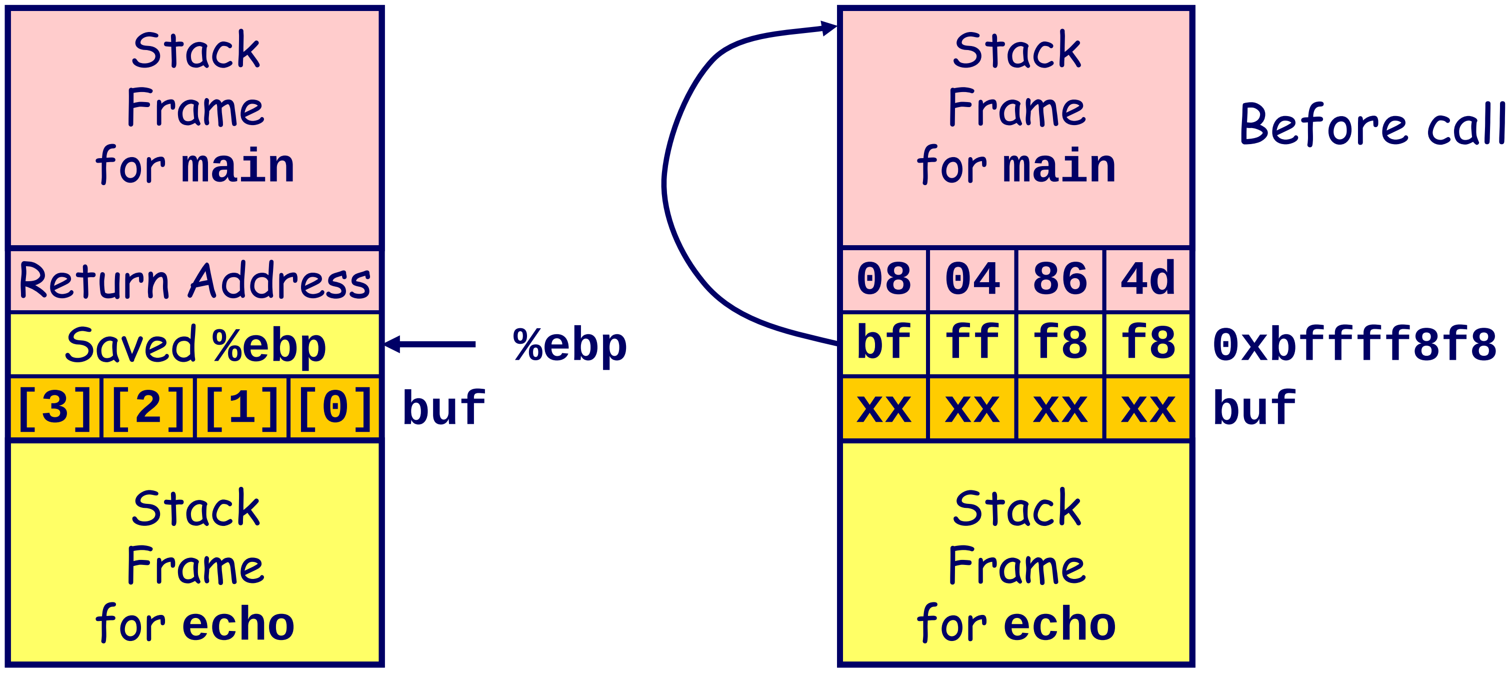


```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
   pushl %ebp               # Save %ebp on stack
   movl %esp,%ebp
   subl $20,%esp            # Allocate stack space
   pushl %ebx               # Save %ebx
   addl $-12,%esp           # Allocate stack space
   leal -4(%ebp),%ebx       # Compute buf as %ebp-4
   pushl %ebx               # Push buf on stack
   call gets                # Call gets
   . . .
```

Buffer Overflow Stack Example

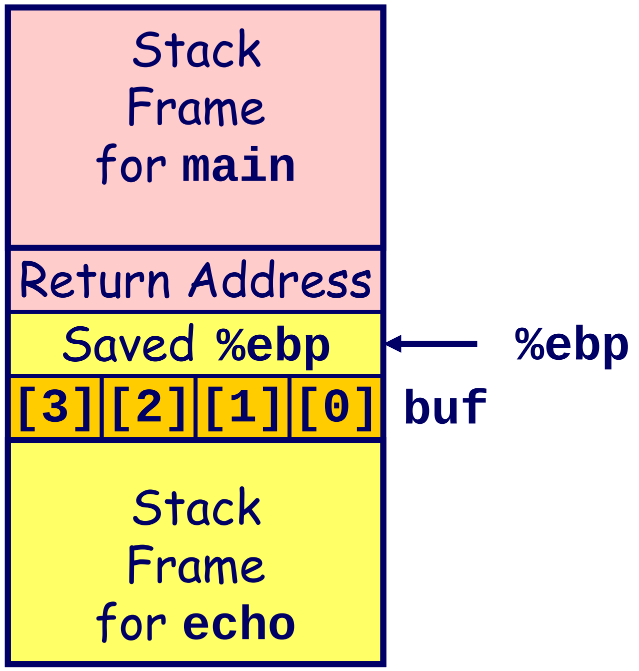
```
unix> gdb bufdemo
(gdb) break echo
Breakpoint 1 at 0x8048583
(gdb) run
Breakpoint 1, 0x8048583 in echo ()
(gdb) print /x *(unsigned *)$ebp
$1 = 0xbffff8f8
(gdb) print /x *((unsigned *)$ebp + 1)
$3 = 0x804864d
```



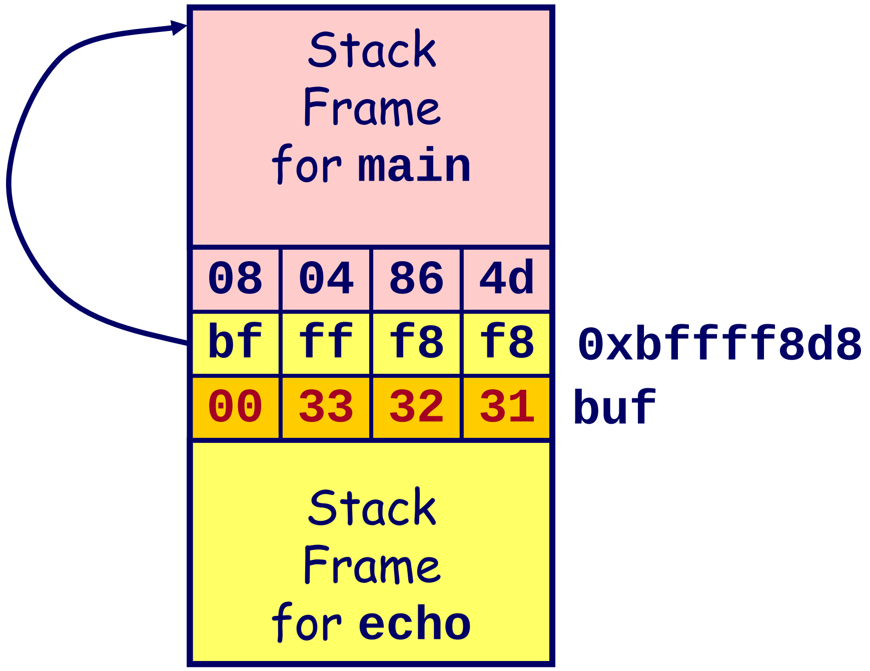
```
 8048648:  call 804857c <echo>
 804864d:  mov  0xffffffe8(%ebp),%ebx # Return Point
```

Buffer Overflow Example #1

Before Call to **gets**

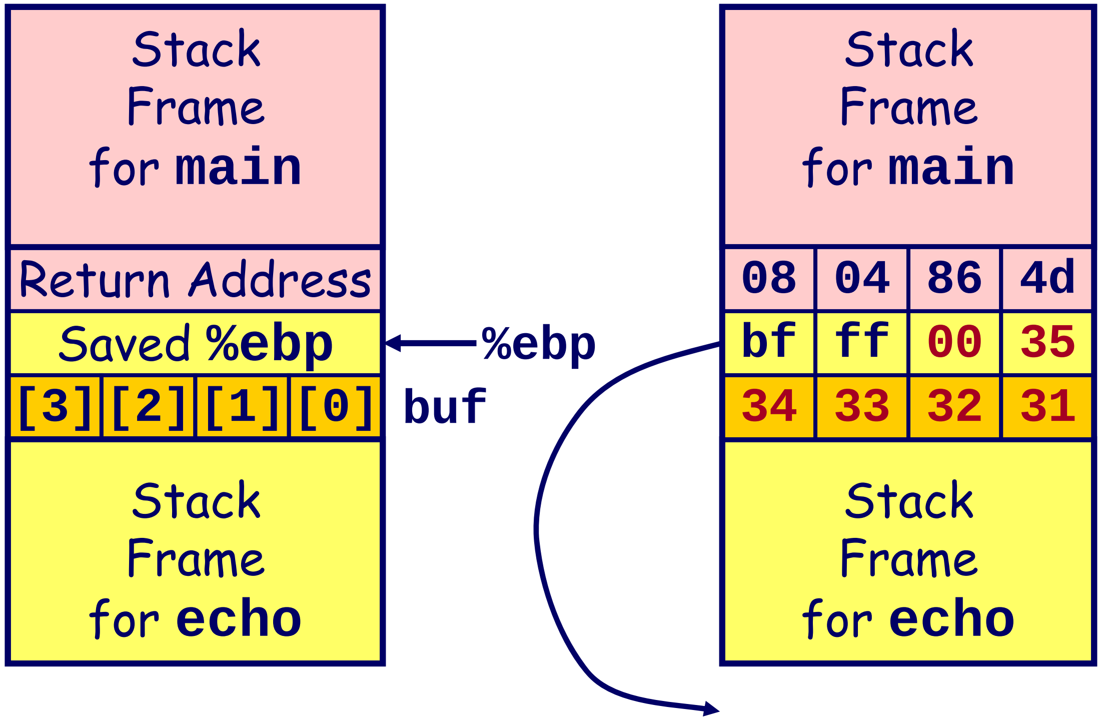


Input = "123"



No Problem

Buffer Overflow Stack Example #2



Input = "**12345**"

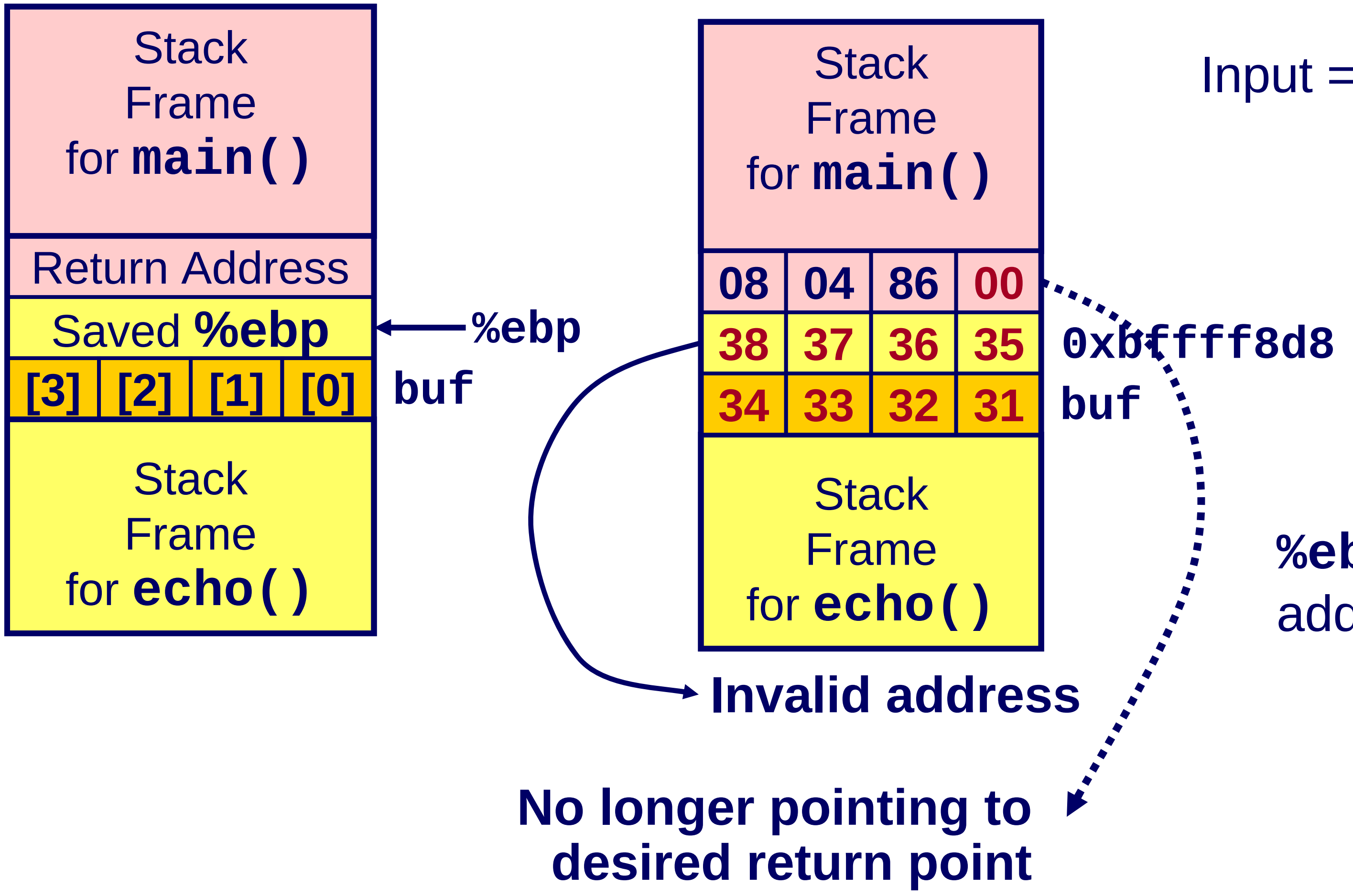
Saved value of **%ebp** set to **0xbfff0035**

Bad news when later attempt to restore **%ebp**

echo code:

```
8048592:  push   %ebx
8048593:  call   80483e4 <_init+0x50>   # gets
8048598:  mov    0xffffffe8(%ebp),%ebx
804859b:  mov    %ebp,%esp
804859d:  pop    %ebp     # %ebp gets set to invalid value
804859e:  ret
```

Buffer Overflow Stack Example #3

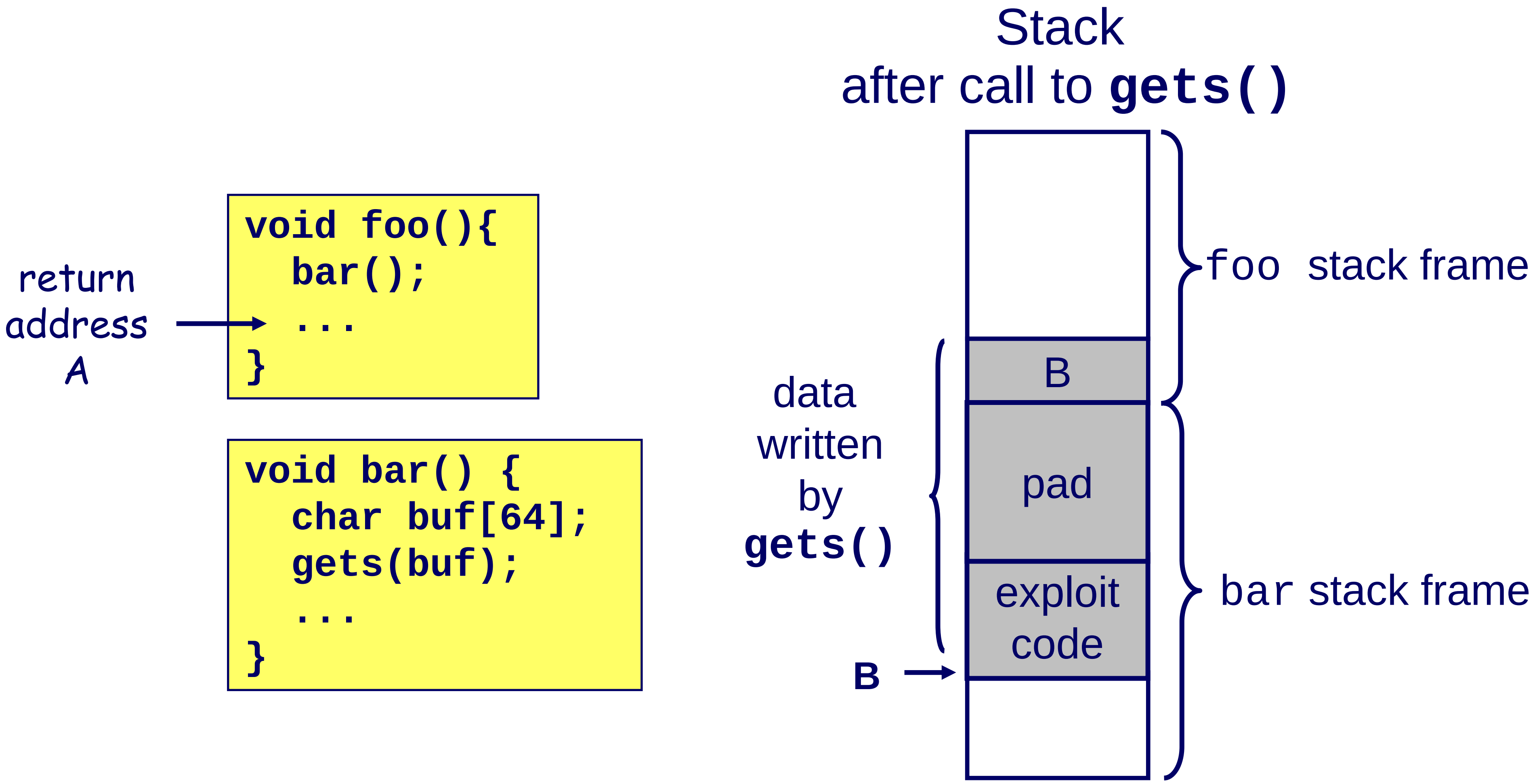


Input = “**12345678**”

%ebp and return address corrupted

```
8048648:  call 804857c <echo>
804864d:  mov  0xffffffe8(%ebp),%ebx # Return Point
```

Malicious Use of Buffer Overflow



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

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

- **Input string contains byte representation of executable code**
- **Overwrite return address with address of buffer**
- **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 neilsen@cis.ksu.edu`
- **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.**

C Call Stack

When a function call is made, the return address is pushed onto the stack.

Often the values of parameters passed to the function are put onto the stack (call-by-value).

Usually the function saves the stack frame pointer (old %ebp) on the stack.

Local variables are placed on the stack.

Stack Direction

On Linux (x86) the stack grows from high addresses to low.

Pushing something onto the stack moves the Top Of the Stack (%esp) towards address 0.

“Smashing the Stack”*

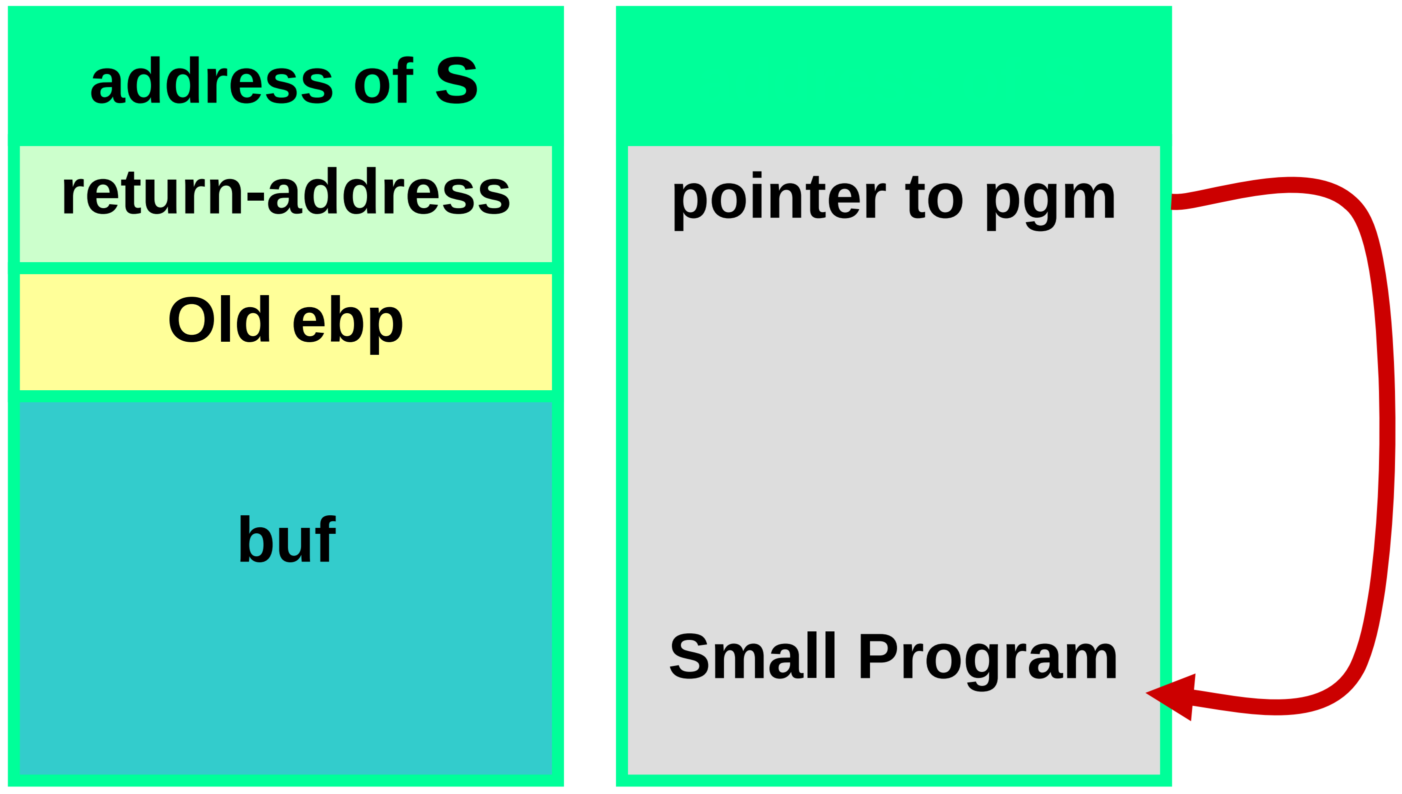
The general idea is to overflow a buffer so that it overwrites the return address.

When the function is done it will jump to whatever address is on the stack.

We put some code in the buffer and set the return address to point to it!

Before and After

```
void foo(char *s) {
          char buf[100];
          strcpy(buf,s);
          …
```



Issues

- How do we know what value the pointer should have (the new “return address”).
- It’s the address of the buffer, but how do we know what address this is?
- How do we build the “small program” and put it in a string?

Guessing Addresses

Typically you need the source code so you can *estimate* the address of both the buffer and the return-address.

An estimate is often good enough!

Building the small program

Typically, the small program stuffed in to the buffer does an `exec()`.

Sometimes it changes the password file or other files…

`exec()`

In Unix, the way to run a new program is with an `exec()` system call.

- There is actually a *family* of `exec()` system calls…
- This doesn't create a new process, it changes the current process to run a new program.
- To create a new process you need another system call (e.g., `fork()`).

exec() example

```
#include <stdio.h>

void execls(void) {
  execl("/bin/ls", "ls", NULL);
  printf("Line not printed if execl is
          successful.\n");
}
```

Generating a String

You can take code like the previous slide, and generate machine language.

Copy down the individual byte values and build a string.

To do a simple exec( ) requires less than 100 bytes.

Some important issues

The small program should be position-independent – able to run at any memory location.

Statically link the libraries to see the code generated for the exec( ) system call; e.g., gcc execExample.c, to see how the exec( ) system call is made. To statically link the libraries, use gcc –static execExample.c.

It can’t be too large, or we can’t fit the program and the new return-address on the stack!

A Sample Program/String

Does an exec( ) of /bin/ls:

```
unsigned char cde[] =
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0”
“\x88\x46\x07\x89\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c”
“\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
"\x80\xe8\xdc\xff\xff\xff/bin/ls";
```

Attacking a real program

Recall that the idea is to feed a server a string that is too big for a buffer.

This string overflows the buffer and overwrites the return address on the stack.

Assuming we put our small program in the string, we need to know it’s address.

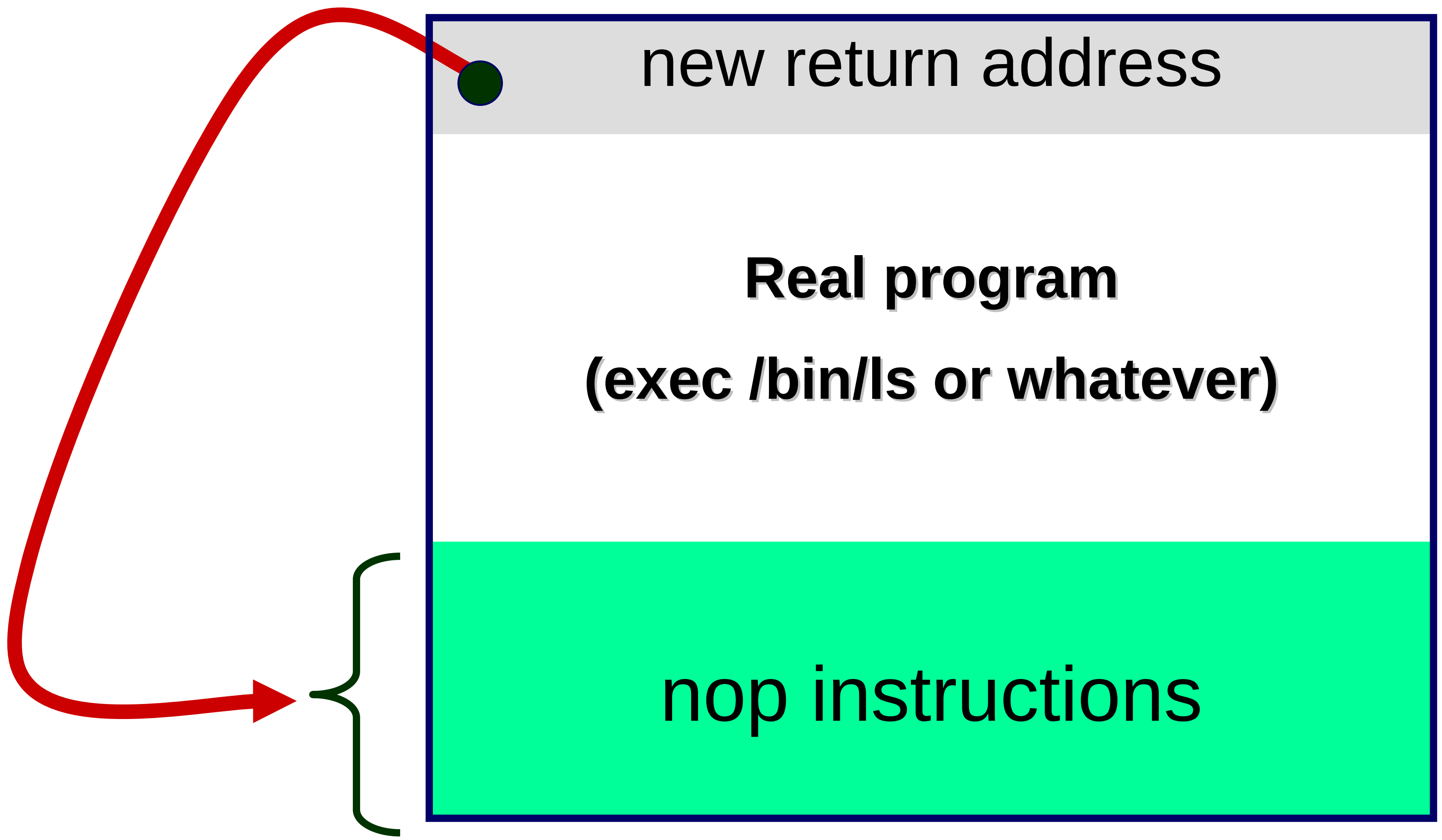
NOPs

Most CPUs have a *No-Operation* instruction – it does nothing but advance the instruction pointer.

Usually we can put a bunch of these ahead of our program (in the string).

As long as the new return-address points to a NOP we are OK.

Using NOPs

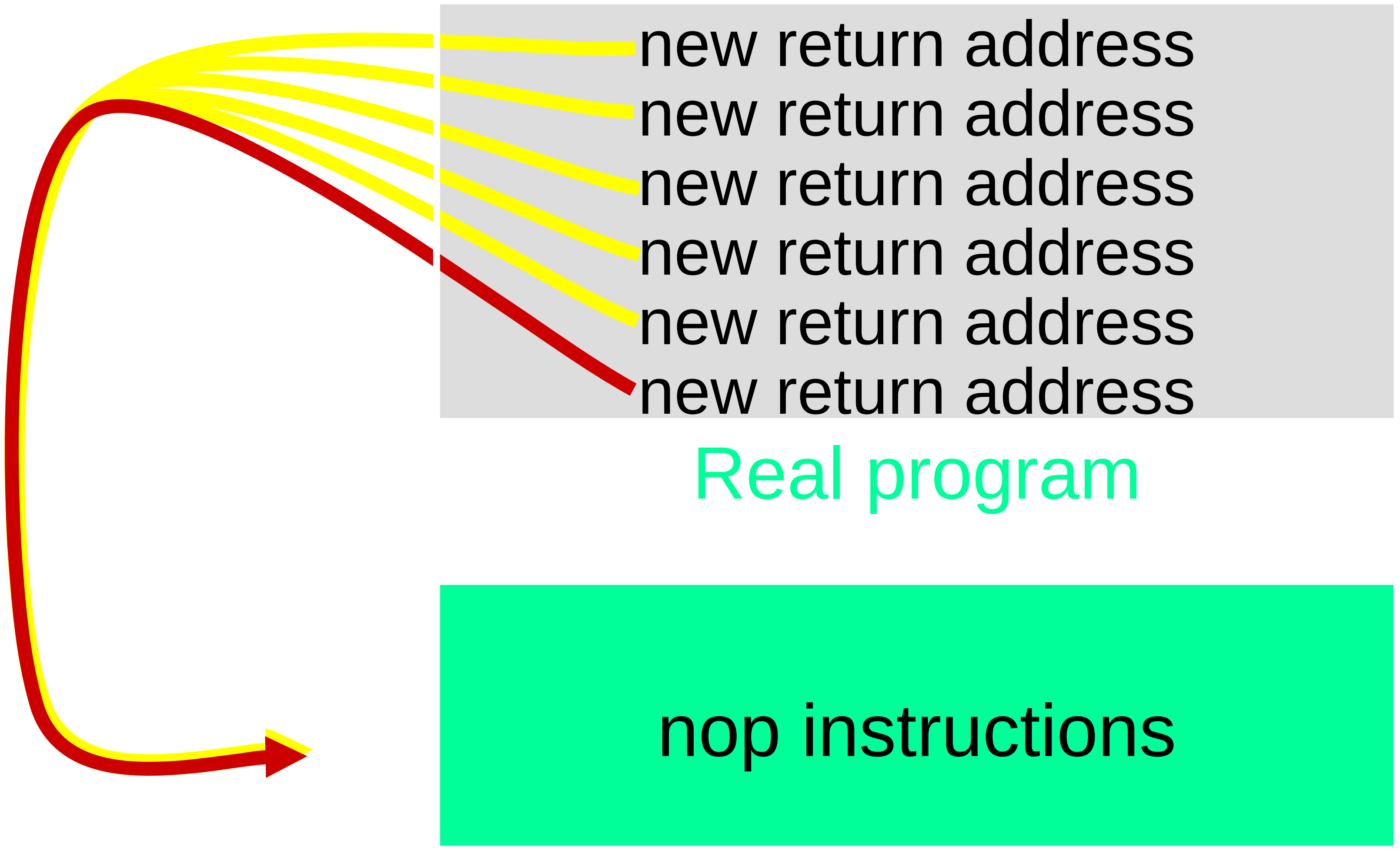

new return address
Real program
(exec /bin/ls or whatever)
nop instructions

Estimating the stack size

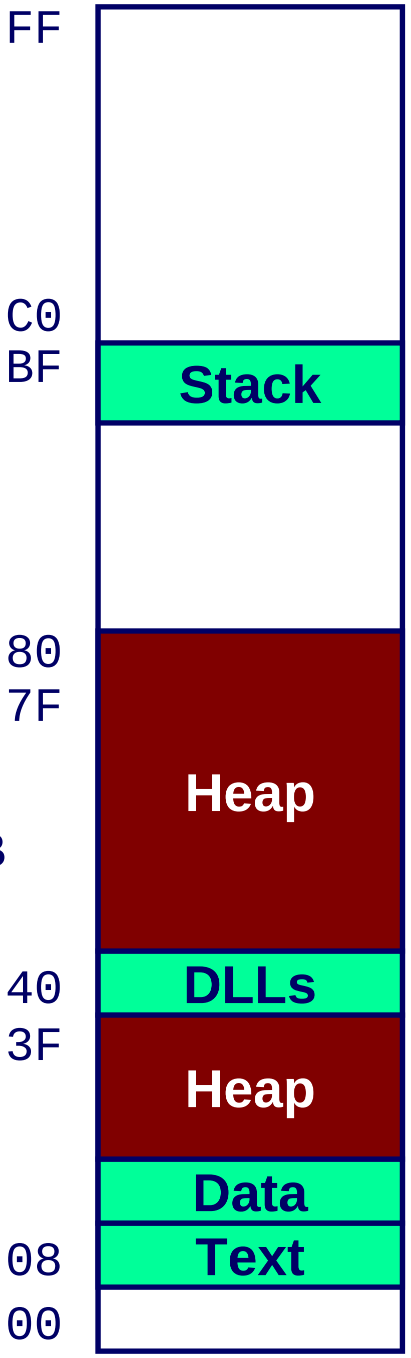
We can also guess at the location of the return address relative to the overflowed buffer.

Put in a bunch of new return addresses!

Estimating the Location



Linux Memory Layout



Upper 2 hex digits of address

Red Hat v. 6.2 ~1920MB memory limit

Stack

- **Runtime stack (8MB limit)**

Heap

- **Dynamically allocated storage**
- **When call `malloc`, `calloc`, `new`**

DLLs

- **Dynamically Linked Libraries**
- **Library routines (e.g., `printf`, `malloc`)**
- **Linked into object code when first executed**

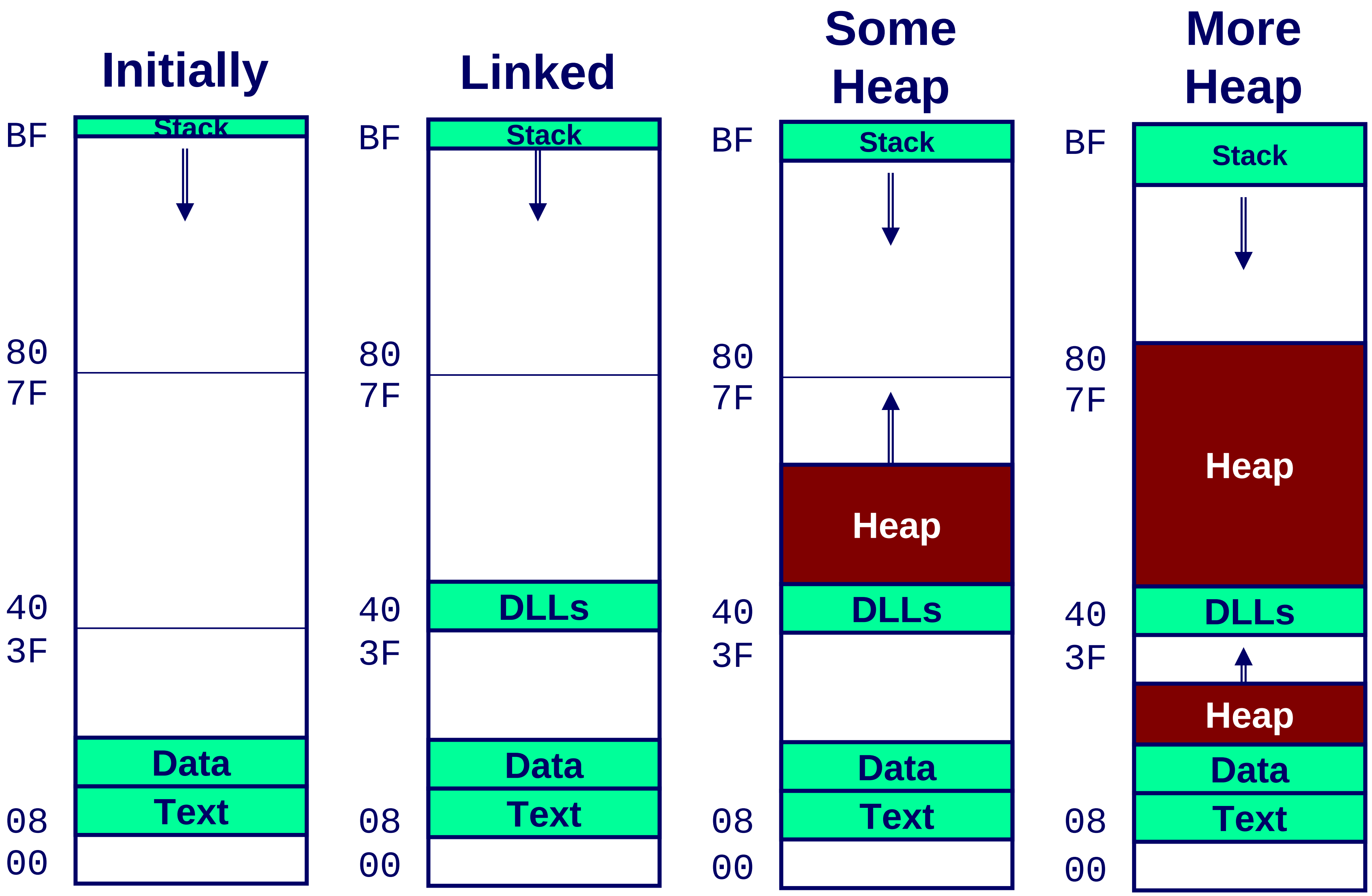
Data

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

Text

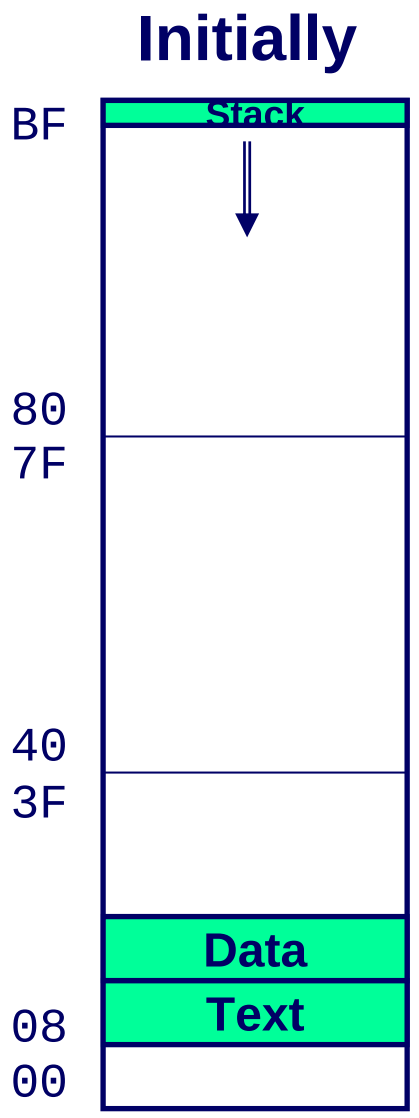
- **Executable machine instructions**
- **Read-only**

Linux Memory Allocation



Text & Stack Example

```
(gdb) break main
(gdb) run
  Breakpoint 1, 0x804856f in main ()
(gdb) print $esp
  $3 = (void *) 0xbffffc78
```



Main

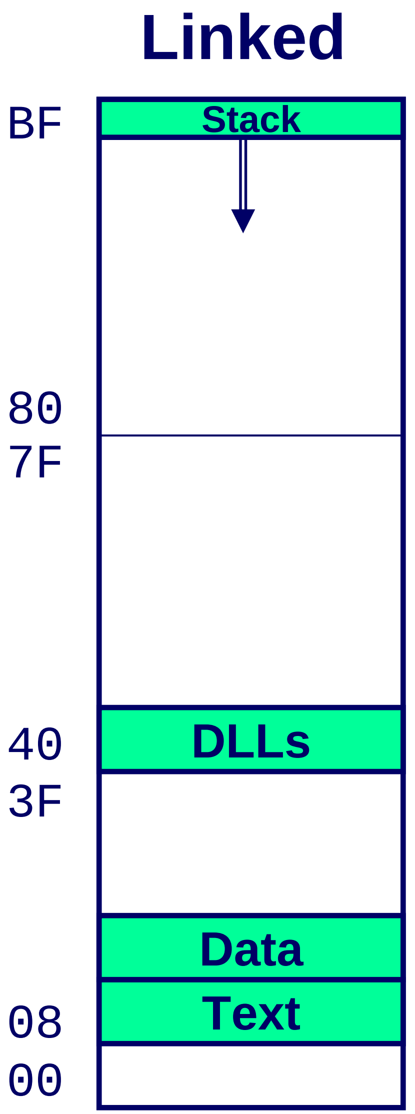
- Address `0x804856f` should be read `0x`*`08`*`04856f`

Stack

- Address `0x`*`bf`*`fffc78`

Dynamic Linking Example

```
(gdb) print malloc
  $1 = {<text variable, no debug info>}
    0x8048454 <malloc>
(gdb) run
  Program exited normally.
(gdb) print malloc
  $2 = {void *(unsigned int)}
    0x40006240 <malloc>
```



Initially

- Code in text segment that invokes dynamic linker
- Address 0x8048454 should be read 0x08048454

Final

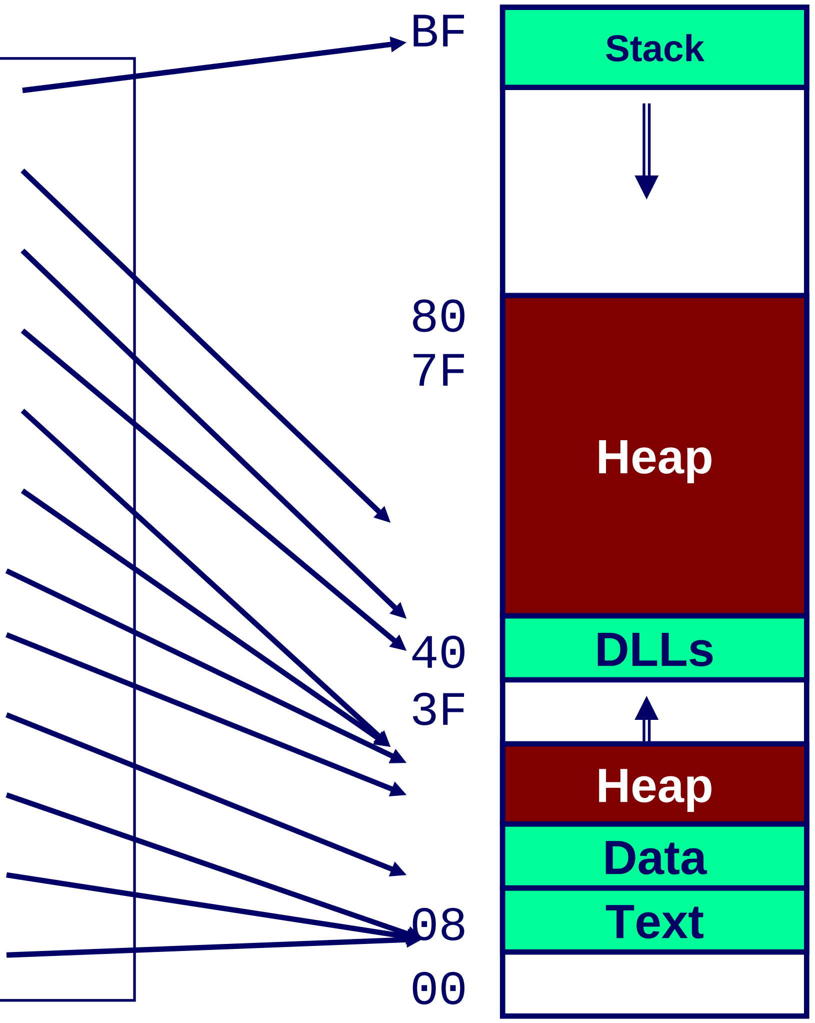
- Code in DLL region

Memory Allocation Example

```
char big_array[1<<24];  /*  16 MB */
char huge_array[1<<28]; /* 256 MB */

int beyond;
char *p1, *p2, *p3, *p4;

int useless() {  return 0; }

int main()
{
 p1 = malloc(1 <<28);  /* 256 MB */
 p2 = malloc(1 << 8);  /* 256 B  */
 p3 = malloc(1 <<28);  /* 256 MB */
 p4 = malloc(1 << 8);  /* 256 B  */
 /* Some print statements ... */
}
```

Example Addresses

`$esp`	`0xbffffc78`
`p3`	`0x500b5008`
`p1`	`0x400b4008`
Final `malloc`	`0x40006240`
`p4`	`0x1904a640`
`p2`	`0x1904a538`
`beyond`	`0x1904a524`
`big_array`	`0x1804a520`
`huge_array`	`0x0804a510`
`main()`	`0x0804856f`
`useless()`	`0x08048560`
Initial `malloc`	`0x08048454`



C operators

Operators	Associativity
`()  []  ->  .`	left to right
`!  ~  ++  --  +  -  *  & (type) sizeof`	right to left
`*  /  %`	left to right
`+  -`	left to right
`<<  >>`	left to right
`<  <=  >  >=`	left to right
`==  !=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right
`?:`	right to left
`= += -= *= /= %= &= ^= != <<= >>=`	right to left
`,`	left to right

Note: Unary +, -, and * have higher precedence than binary forms

C pointer declarations

`int *p`	p is a pointer to int
`int *p[13]`	p is an array[13] of pointer to int
`int *(p[13])`	p is an array[13] of pointer to int
`int **p`	p is a pointer to a pointer to an int
`int (*p)[13]`	p is a pointer to an array[13] of int
`int *f()`	f is a function returning a pointer to int
`int (*f)()`	f is a pointer to a function returning int
`int (*(*f())[13])()`	f is a function returning ptr to an array[13] of pointers to functions returning int
`int (*(*x[3])())[5]`	x is an array[3] of pointers to functions returning pointers to array[5] of ints

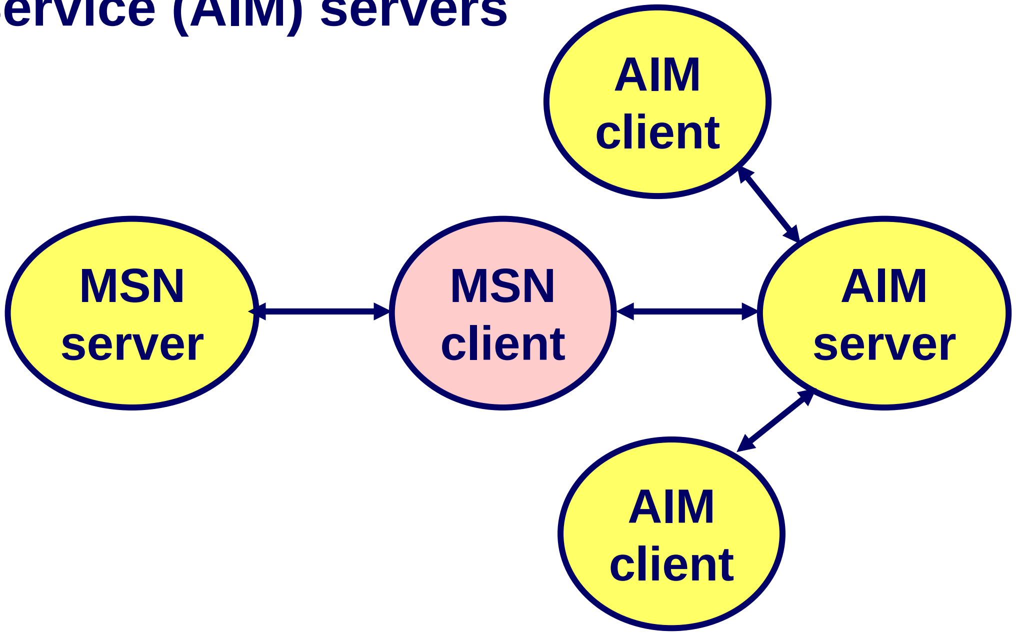
Internet Worm and IM War

November, 1988

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

July, 1999

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



Internet Worm and 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?

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

  - many Unix functions do not check argument sizes.
  - allows target buffers to overflow.

Summary

Arrays in C

- Contiguous allocation of memory
- Pointer to first element
- No bounds checking

Structures

- Allocate bytes in order declared
- Pad in middle and at end to satisfy alignment

Unions

- Overlay declarations
- Way to circumvent type system

Buffer Overflow

- Overrun stack state with externally supplied data
- Potentially contains executable code