



HARVARD

School of Engineering
and Applied Sciences

Assembly Programming: Buffer overruns and stack exploits

CS61, Lecture 6

Prof. Stephen Chong

September 21, 2010

Announcements

- Lab 1 due today, 11:59pm!
- Lab 2 will be released today
 - Due 30 September (10 days)
 - See webpage for more info

Memory vulnerabilities

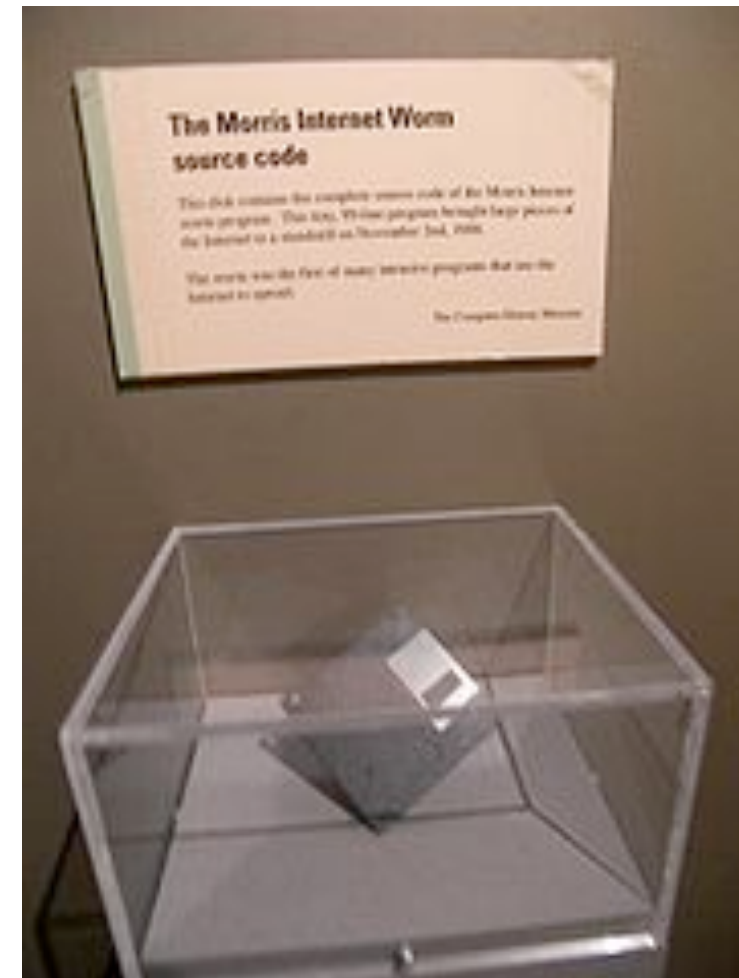
- Many C programs contain subtle bugs that can lead to remote exploits
- Most common case: **Buffer overflow attacks**
 - Program reads data into a fixed-size buffer
 - Remote attacker feeds program data that overflows the buffer
 - How can this lead to a security hole?
- Buffer overflow overwrites other memory used by the program
 - For example, the return address on the stack
- Attacker sends machine instructions that end up being executed by the remote host!
 - Allows the attacker to cause the remote machine to run (almost) any code.

Real vulnerabilities

- Internet Worm: 1988
 - First widespread worm on the Internet
 - Estimated infected 10% of machines on the Internet
- Code Red, Code Red II, NIMDA, SQL Slammer
 - Various worms that attacked Windows machines
 - Led to denial of service attacks, backdoors, web pages being defaced, etc.
- AOL vs. Microsoft in the Internet Messaging Wars
 - AOL exploiting a buffer overrun in its own AIM client
- iPhone jail breaking, Xbox modding, ...
- Lab 2!!

The Internet Worm

- November 2, 1988: One of first large-scale worm attacks on the Internet launched
 - At the time, just 60,000 machines on Internet
 - Most were VAX or Sun machines running BSD UNIX
- Worm repeatedly infected machines, causing huge load, slow down, lots of weird activity
 - At first it was not clear what was going on
 - Lots of universities and companies notice the attack
- Very rapid response by the community
 - Nov 3, teams at MIT and Berkeley “capture” worm and disassemble it
 - Within few days they have a basic understanding of how it works, and patches to prevent its spread
- See “The Internet Worm Program: An Analysis” by Eugene H. Spafford (Purdue Technical Report CSD-TR-823)



http://en.wikipedia.org/wiki/File:Morris_Worm.jpg

Details of the Worm

- Three basic attack mechanisms:
- 1) Exploited debugging “feature” of sendmail
 - Allowed remote user to send an email with a program as the recipient
 - Caused remote machine to interpret email message as a shell script!
 - Shell script extracted a C program from the message, compiled it, and ran it
- 2) Exploited rsh “.rhosts” feature
 - rsh allows users to create file of machines trusted to log in with no password!
 - Worm cracks user's password locally, sh to that user, then rsh to remotely
- 3) **Buffer overflow** in fingerd
 - Finger daemon (fingerd) provides info on users on machine
 - fingerd reads its input insecurely, allows arbitrary code to run within fingerd
 - Since fingerd generally runs as root, gives remote user root access!

Example: gets() library routine

```
/* 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;
}
```

- `char *gets(char*)` reads a string from stdin and stores it in buffer provided by caller
- What's wrong with this code?

Example: gets() library routine

```
/* 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;
}
```

- Does not check the size of buffer **dest**!
 - No way to check it: not passed in as an argument
- Similar problems with other Unix functions
 - **strcpy**: copy string of arbitrary length
 - **scanf**, **fscanf**, **sscanf**, when given %s specification

Example of badly written code

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

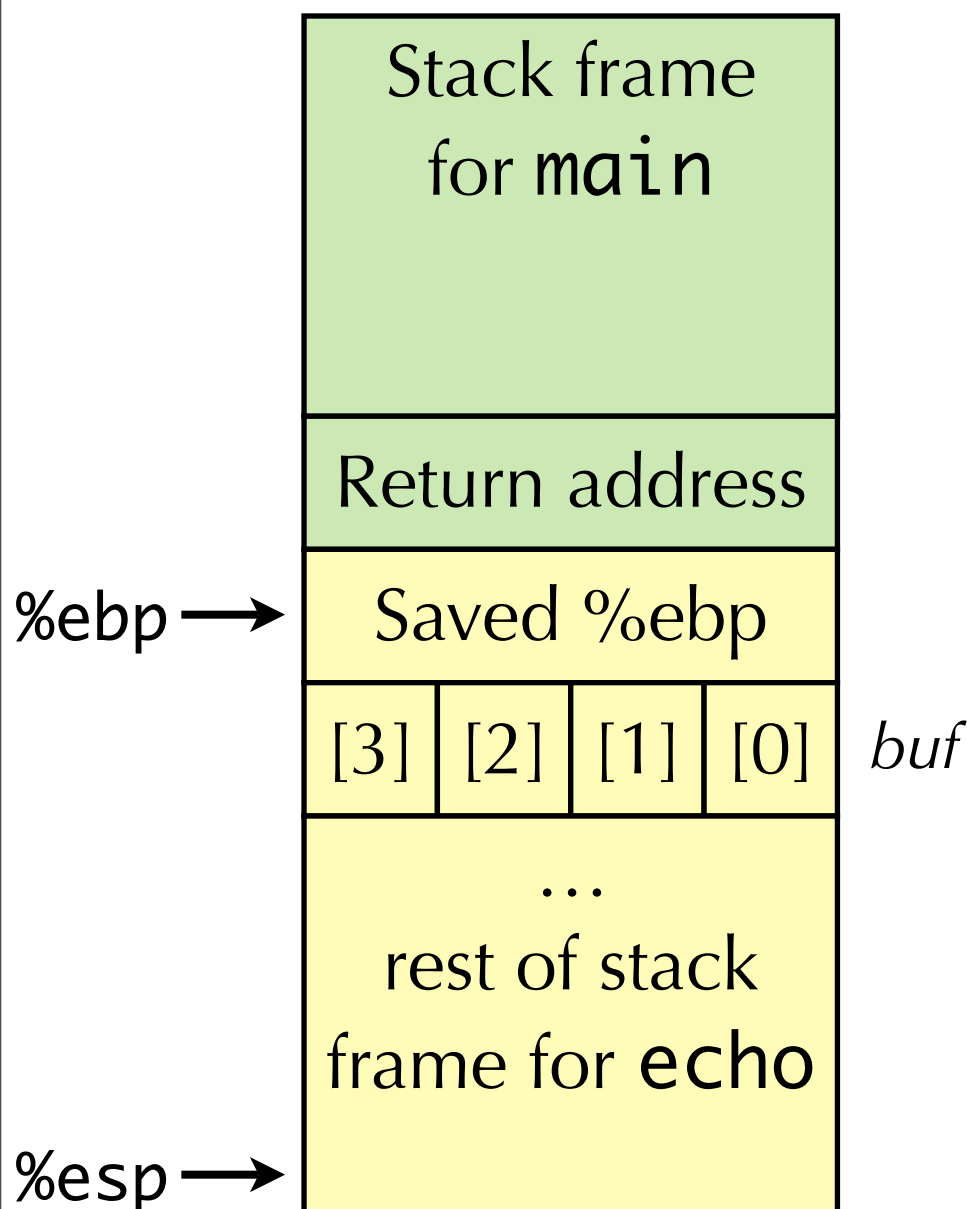
```
int main() {  
    printf("Type a string:");  
    echo();  
    return 0;  
}
```

What happens when we run?

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

```
[chong@cs61 ~]$ ./bufdemo  
Type a string:123  
123  
[chong@cs61 ~]$ ./bufdemo  
Type a string:123456  
123456  
[chong@cs61 ~]$ ./bufdemo  
Type a string:1234567890  
1234567890  
Segmentation fault
```

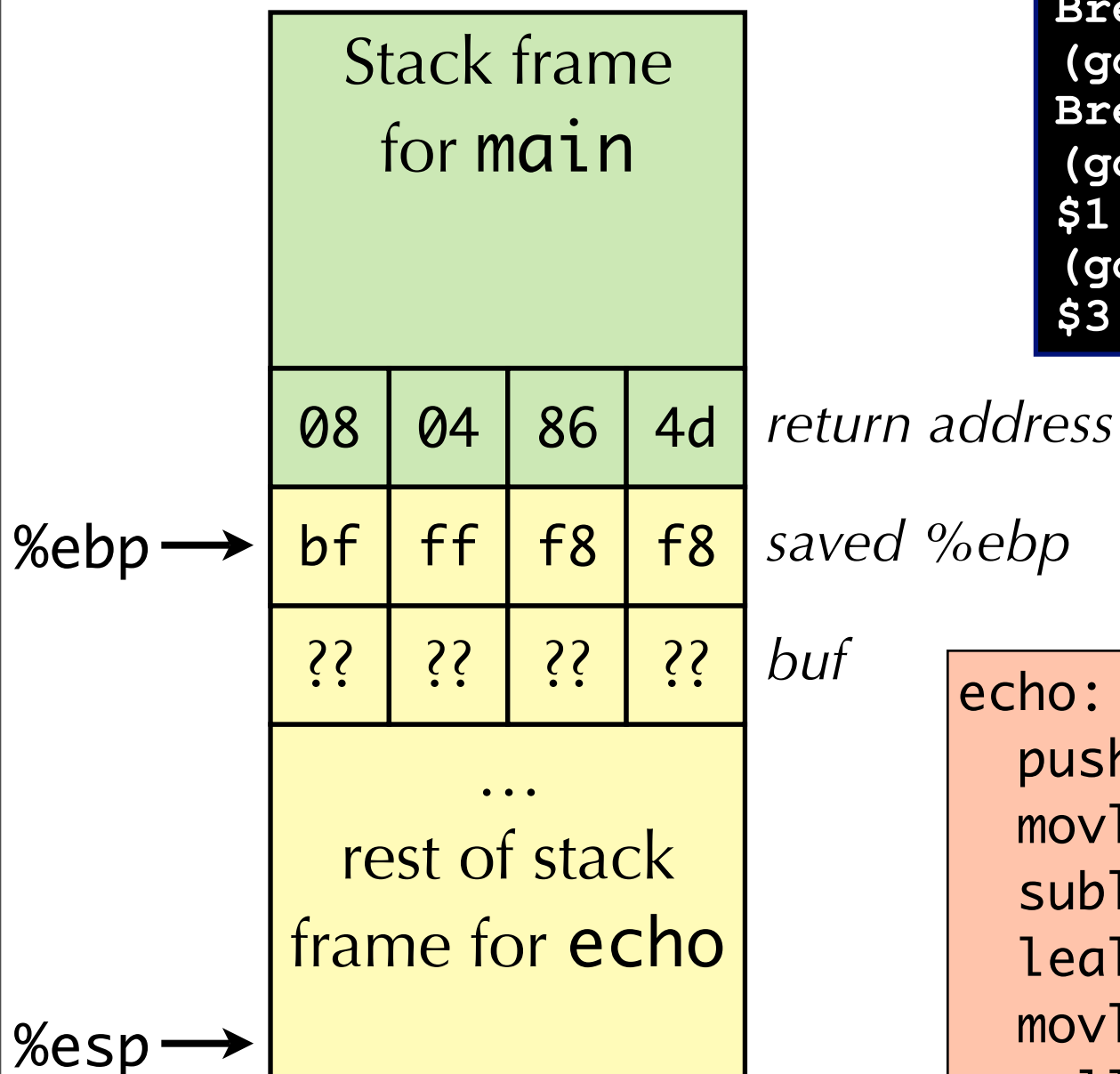
Code disassembly



```
/* 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  $12, %esp      # allocate space on stack
    leal  -4(%ebp), %eax # %eax = buf = %ebp-4
    movl  %eax, (%esp)   # push buf on stack
    call  gets           # call gets
    ...
```

Stack layout for echo()

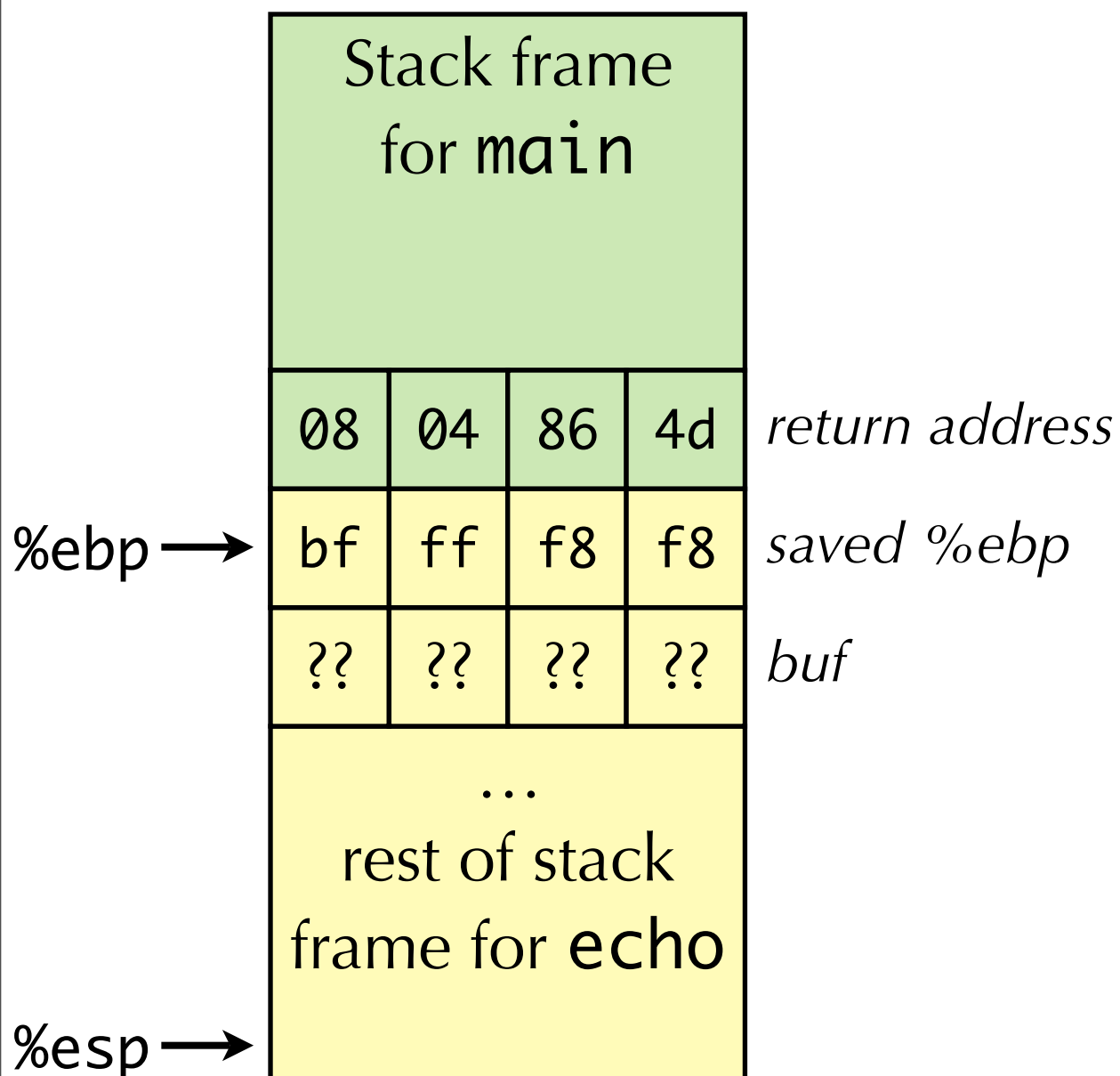


```
unix> gdb bufdemo
(gdb) break echo
Breakpoint 1 at 0x8048583
(gdb) run
Breakpoint 1, 0x8048583 in echo ()
(gdb) print /x *(unsigned *)$ebp
$1 = 0xbffff8f8
(gdb) print /x *((unsigned *)$ebp + 1)
$3 = 0x804864d
```

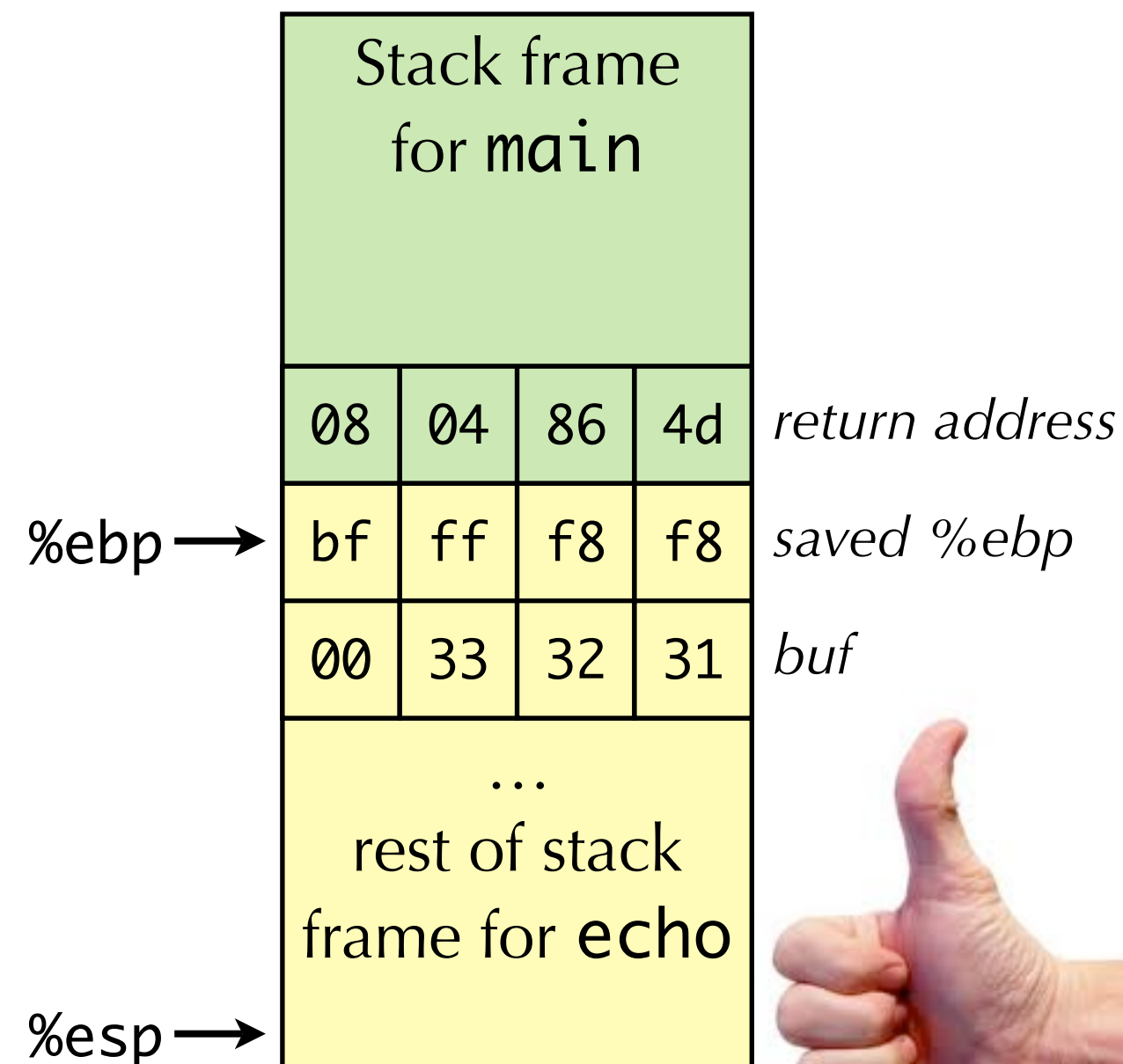
```
echo:
    pushl %ebp                # save %ebp on stack
    movl  %esp, %ebp
    subl  $12, %esp           # allocate space on stack
    leal  -4(%ebp), %eax      # %eax = buf = %ebp-4
    movl  %eax, (%esp)        # push buf on stack
    call  gets                # call gets
    ...
```

Entering a string that fits in buf[]

Before Call to `gets`



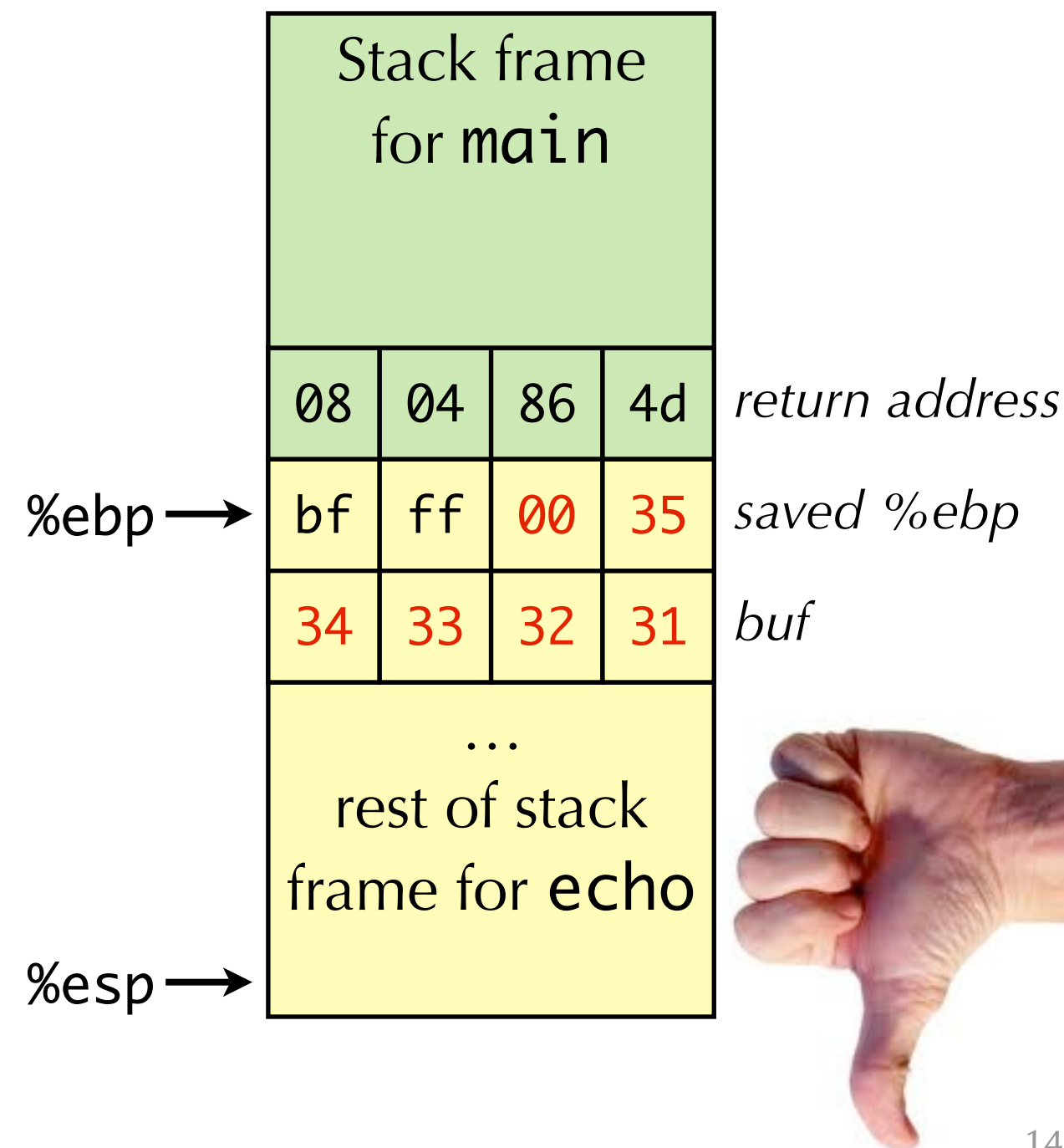
After Call to `gets`
with input "123"



Entering a string TOO BIG for buf[]

- What if we enter the string "12345"?
 - Will overflow the buffer
 - Where do the extra bytes end up?
- Overwrite the saved %ebp on the stack!
 - What will this do to the program?

After Call to gets
with input "12345"

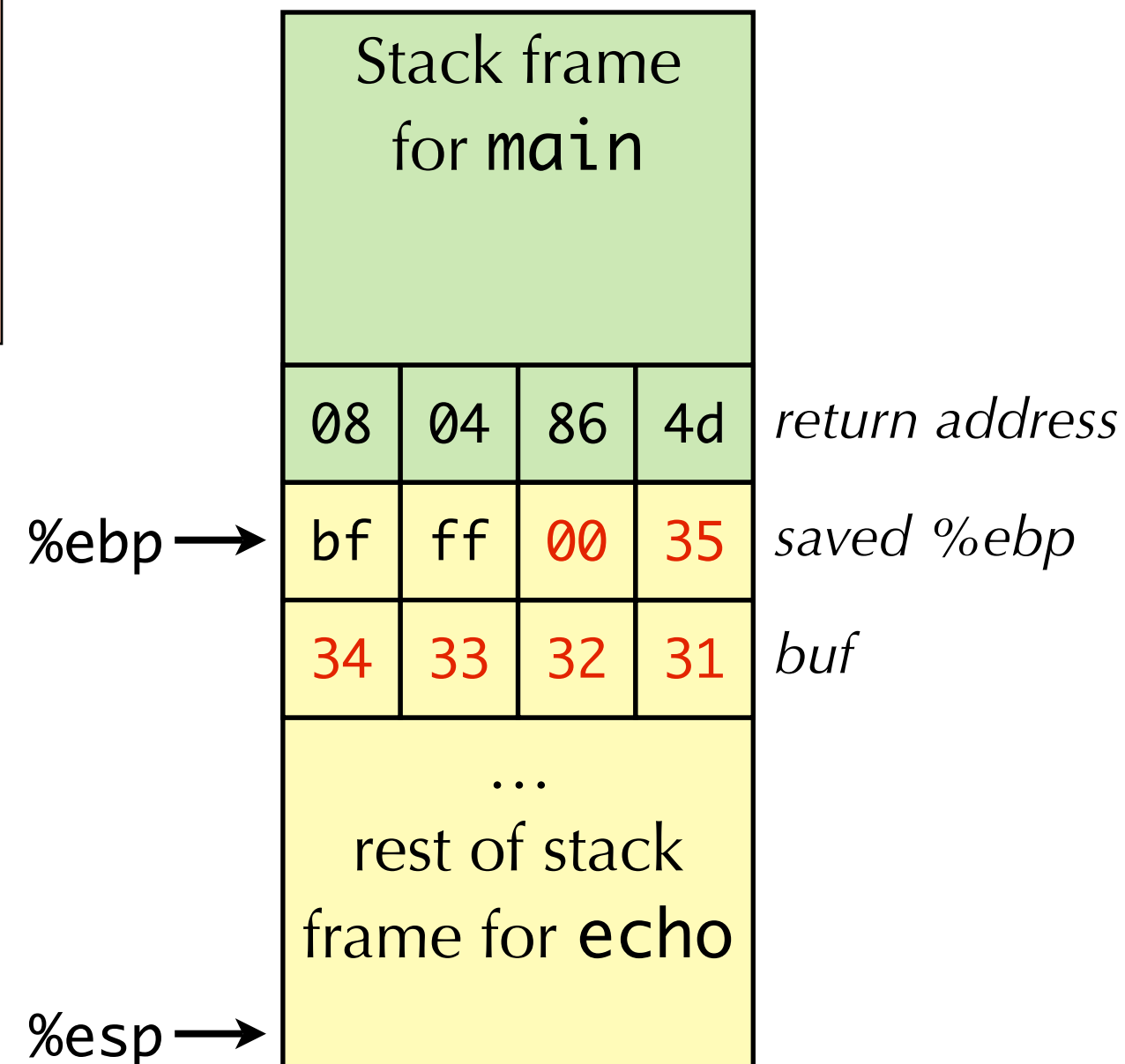


Entering a string TOO BIG for buf[]

```
echo:
...
call    gets          # call gets
...
movl    %ebp, %esp    # %esp = %ebp
popl    %ebp          # restore old %ebp
ret
```

- Restores incorrect value for %ebp!!!
 - Restores 0xbfff0035 instead of 0xbffff8f8

After Call to gets
with input "12345"

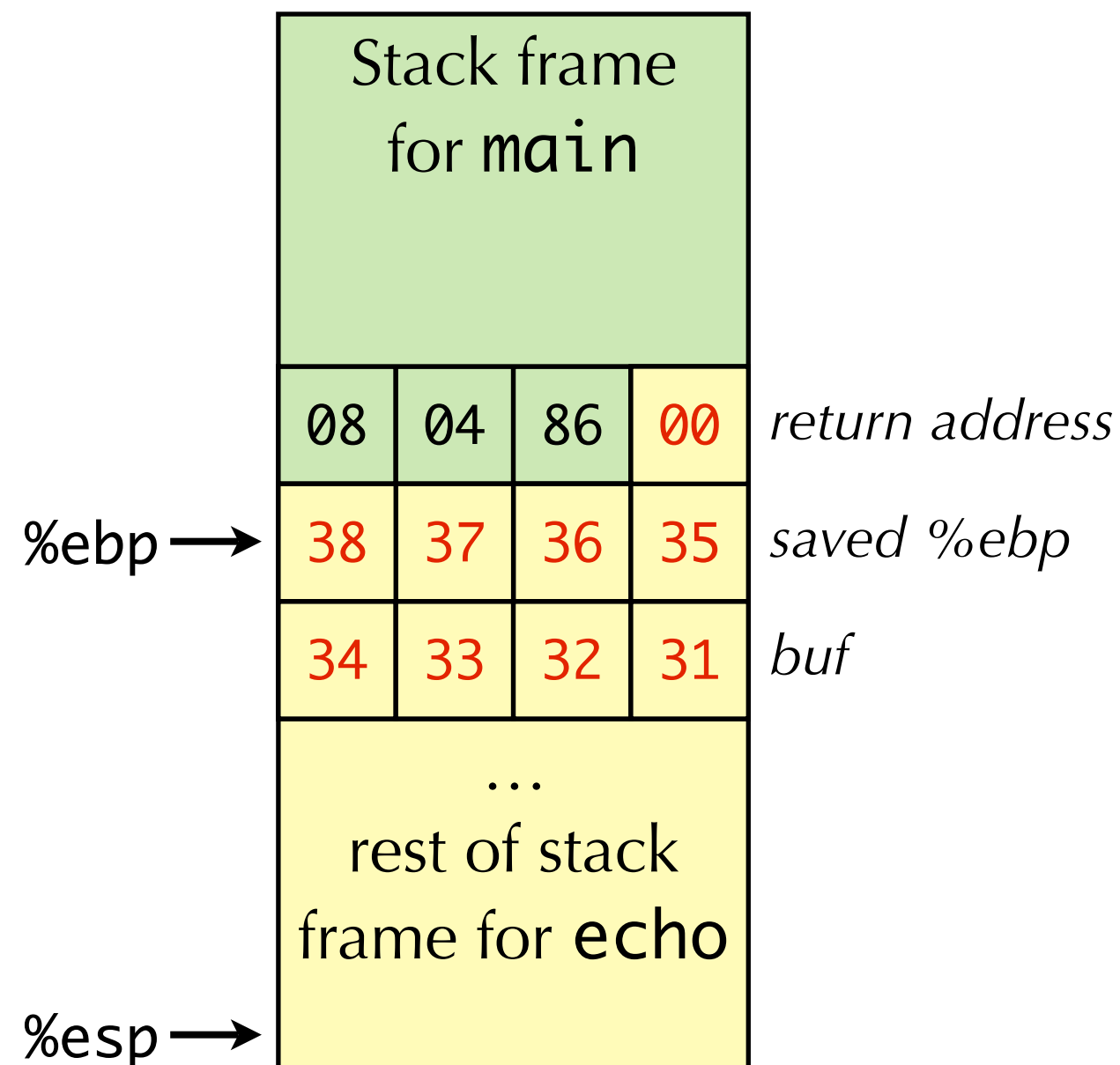


Entering EVEN BIGGER string

```
echo:
...
call    gets          # call gets
...
movl    %ebp, %esp    # %esp = %ebp
popl    %ebp          # restore old %ebp
ret
```

- Restores incorrect value for `%ebp`
- Jumps to wrong return address!!!

After Call to `gets`
with input "12345678"



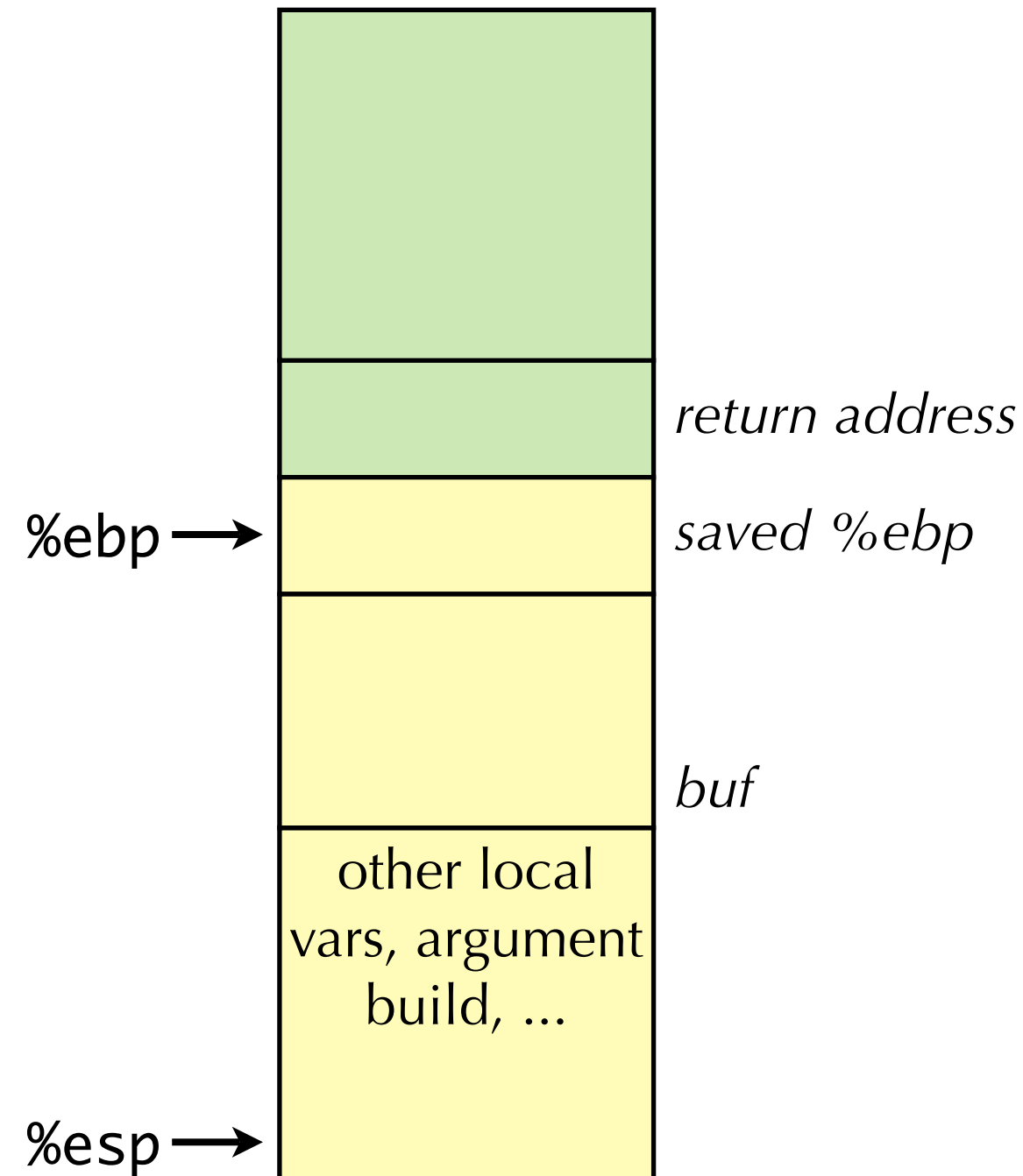
Malicious use of buffer overflow

- If we can overwrite portions of the stack, we can cause the program **to jump to an address of our choosing!**
- This can be used to do all kinds of nasty things.
- Say we knew the memory address of a routine that, say, deleted all of the files in the user's home directory.
 - Most programs would not contain such a routine, but it could happen ...
 - If we can coerce the program to jump to that routine, we can do major damage.
- This attack is fundamentally limited, however...
 - Can only cause the program to run code that's already part of the program.
- How can we **inject our own code** into the running program?

Injecting code

- Suppose routine puts data into buffer on stack (like previous example)
- Provide **x86 machine code** as input to the routine!
 - Fill buffer with instructions we want to run
 - Overwrite return address to point to buffer

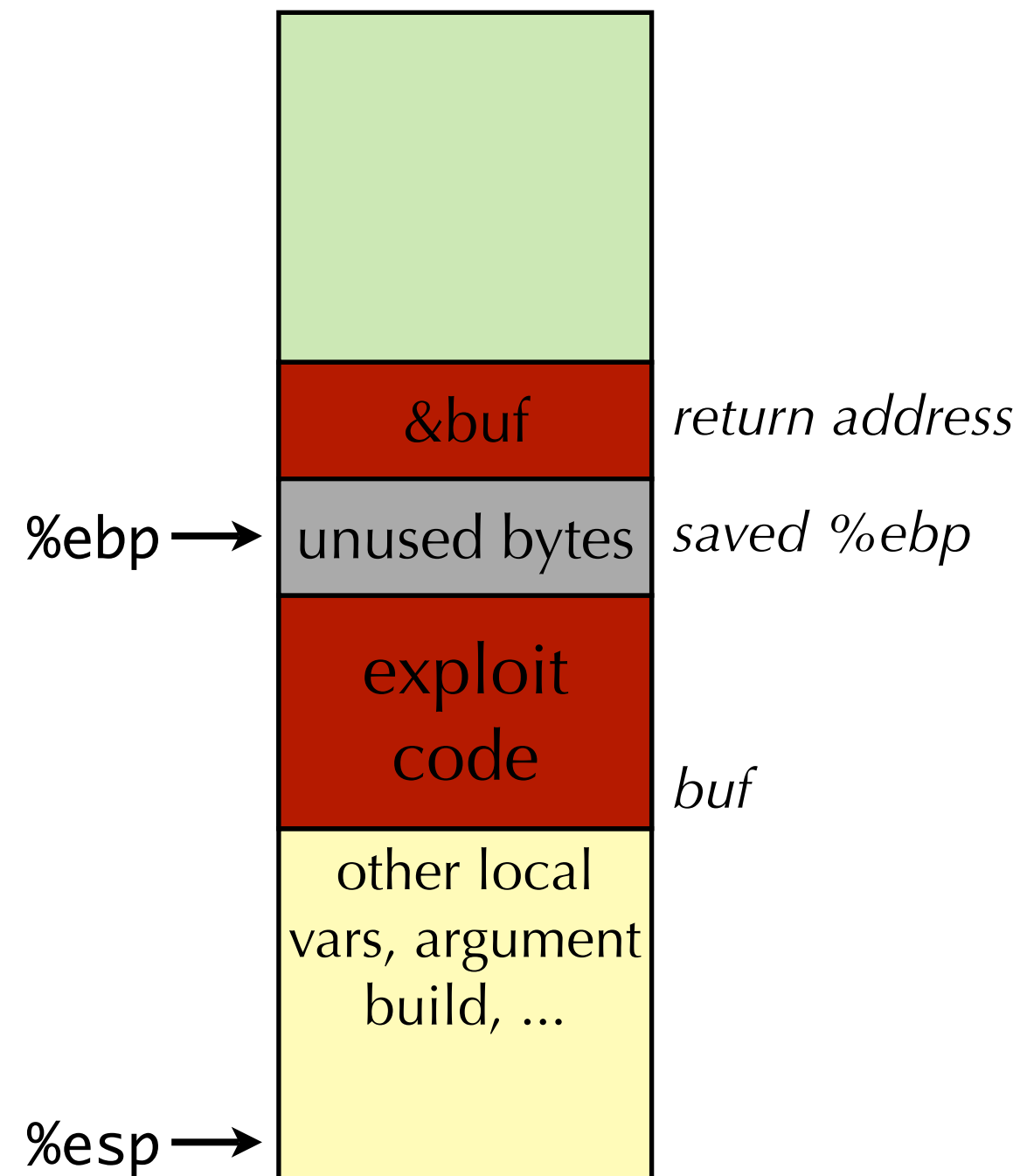
```
void some_routine() {  
    char buf[64];  
    gets(buf);  
}
```



Injecting code

- Suppose routine puts data into buffer on stack (like previous example)
- Provide **x86 machine code** as input
 - Fill buffer with instructions we want to run
 - Overwrite return address to point to buffer
- When routine tries to return...
 - `ret` pops return address off the stack
 - **But return address now points to buffer!**
 - Processor starts running code in buffer!

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



Some limitations of this attack

- Executing this attack on arbitrary programs is tricky.
 - 1) Need to know where on the stack the buffer is (and how big it is)
 - 2) Need to know where return address is on the stack (relative to the buffer).
 - Remember, you can only control what goes into the buffer (and any addresses beyond the end of the buffer).
- If you have access to the binary, this is not too difficult...
 - Can just use gdb, set breakpoints, inspect the stack, and figure it out.
- If you're attacking a service on the Internet and don't have the binary, this becomes much harder.
 - But it can be done, usually with a lot of trial and error.

Mitigating buffer overflow attacks

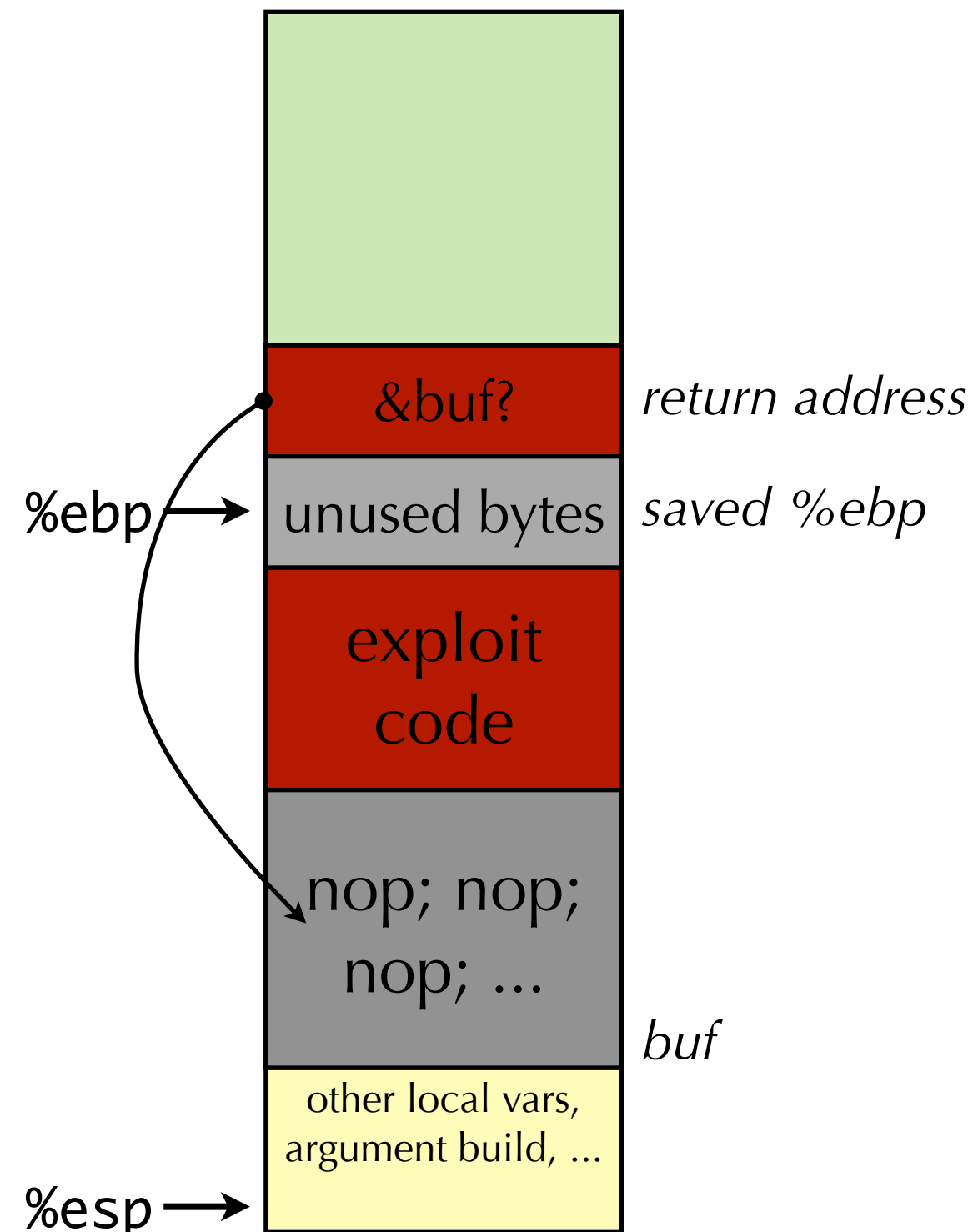
- Three common mechanisms
 - Stack randomization
 - Stack corruption detection
 - Non-executable memory

Stack randomization

- Exploiting stack-based buffer overflows requires knowing where buffer is in memory
 - Need to overwrite return address on stack with pointer to buffer
- One way to thwart this: **Address space randomization**
 - When kernel runs a program, it puts the stack at a (slightly) random location in memory each time.
 - Thus attacker unlikely to correctly guess buffer's address.
 - Implemented by recent Linux kernels by default.
- To thwart address space randomization...

The NOP Sled attack

- Idea: Start out buffer with long string of **nop** instructions
 - “No-op” instruction doesn’t do anything; just moves to next instruction.
- Put best guess of exploit code location in return address.
 - OK if we “undershoot” a bit.
- When program resumes execution within the NOP sled region, code will execute until it hits your exploit code.
 - Note: won't work if we “overshoot” guess of exploit code location.



Detecting stack corruption

- Try to detect when array on stack has overflowed
- Store special **canary value** (aka **guard value**) on stack
 - Generated randomly every time program is run
 - Attacker can't predict value
- Before returning from function check canary value unchanged
 - If changed, stop execution
- Recent versions of gcc do this for functions that may be vulnerable



Non-executable memory

- Idea: limit which memory regions can hold executable code
 - Modern operating systems and hardware support different forms of memory protection
 - Readable memory, writeable memory, executable memory
 - We'll learn more about the mechanisms that enable this
- Make stack readable, writeable, but not executable
- Note: some languages/programs dynamically generated code
 - E.g., Just-in-time (JIT) compilation of Java bytecode
 - Non-executable memory may not be a feasible in these settings

Avoiding Overflow Vulnerability

- Mitigations techniques (stack randomization, detecting stack corruption, non-executable memory) make it harder to perform buffer overflow attacks
 - But not impossible!
- How do we prevent all overflow vulnerabilities?

Avoiding Overflow Vulnerability

- **Rule #1: Don't program in C!**
- Java (and many other languages) make this kind of attack more or less impossible. How?
- In Java, all array accesses are bounds-checked at runtime.
 - No way to stuff data into an array beyond its size limit.
- Also, Java doesn't let you directly manipulate pointers.
 - No way to cause the program to jump to an arbitrary memory address.
- Of course, this relies on the Java Virtual Machine being free of any bugs itself...
 - No guarantees that this is the case!

Avoiding Overflow Vulnerability

- **Rule #2: Always check buffer lengths!**
 - Especially when reading data from the outside world – a user or a network socket.
- Use standard library routines that check buffer bounds
 - `fgets` instead of `gets`
 - `strncpy` instead of `strcpy` – checks length of string.
 - Don't use `scanf` with `%s` conversion specification
 - Use `fgets` to read the string

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

Buffer exploits over the network

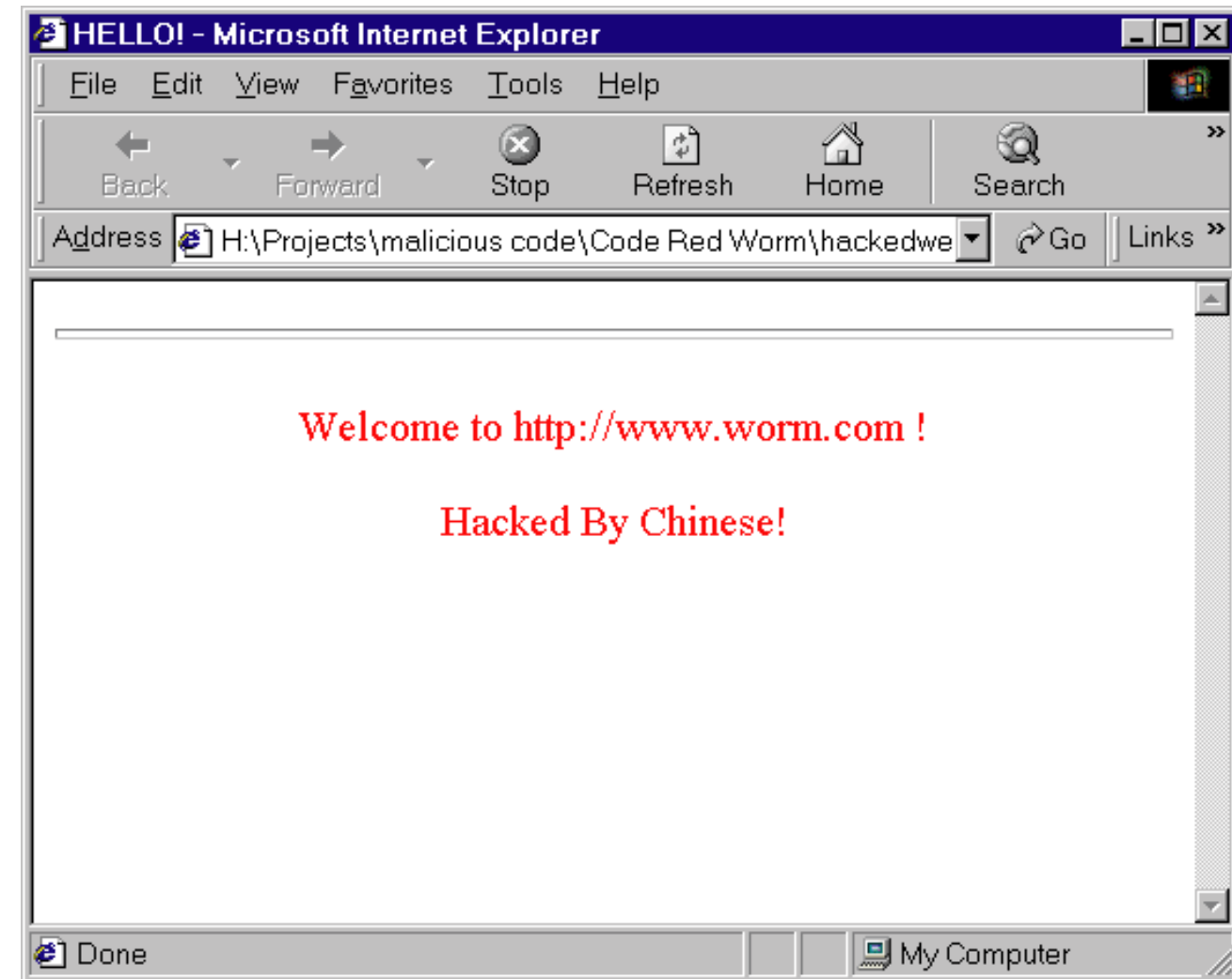
- Attacks so far use the `gets()` routine
 - Reads a string from standard input, typically user input, or from a file
- Problems also exists in programs that read data from the network.
 - Web browsers, IM clients, MP3 players, games,
 - Program reads data from network into buffer on the stack, and fails to check the data fits into the buffer \Rightarrow vulnerable to buffer overflow exploits
 - Happens a lot in the real world.
- More serious issue: Programs running as the “root” user
 - Many services on UNIX systems run as “root”: Admin user that can do anything on the machine.
 - Example: Web servers, file servers, ssh daemon, etc.
 - If you can attack these services, exploit code will run as root, and can do arbitrary damage to machine.

Code Red Worm

- June 18, 2001 – Buffer overflow vulnerability in Microsoft IIS Web server announced
- June 26, 2001 – Microsoft releases patch for vulnerability
- July 13, 2001 – Code Red v1 worm released
 - Infects machines and causes them to perform denial-of-service attacks
 - Bug in random number generator slows infection rate. New version released a few days later
- August 4, 2001 – Code Red II worm released
 - Same basic attack vector, but somewhat different behavior

How does Code Red work?

- Overflows stack of the IIS web server
 - Causes it to overwrite return address on the stack
 - IIS then jumps into the machine code in HTTP request
- Defaces server's home page



How does Code Red work?

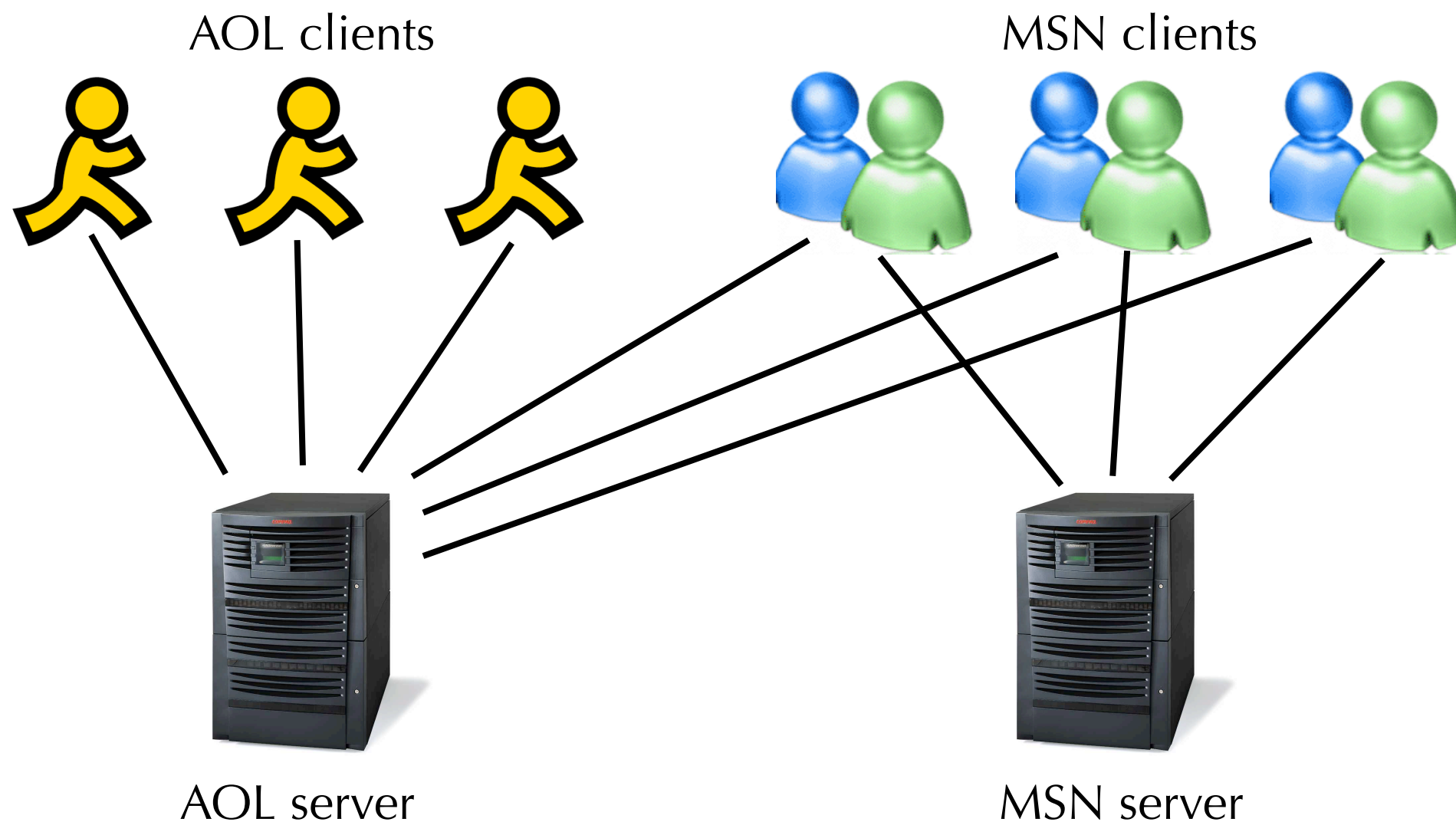
- Start 100 threads running
- Spread self
 - Open connections to random IP addresses and send attack string
 - May or may not be IIS
 - Between 1st and 19th of month
- Attack several static IP addresses, including `www.whitehouse.gov`
 - Send 98,304 packets; sleep for 4-1/2 hours; repeat
 - This is called a **denial-of-service** attack
 - Between 21st and 27th of month
 - White House had to change IP address

The Code Red II Attack

<i>Padding (overflows buffer)</i>	0000	47	45	54	20	2f	64	65	66	61	75	6c	74	2e	69	64	61	GET /default.ida
	0010	3f	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	?XXXXXXXXXXXXXXXXX
	0020	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	XXXXXXXXXXXXXXXXXX
	0030	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	XXXXXXXXXXXXXXXXXX
	0040	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	XXXXXXXXXXXXXXXXXX
	0050	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	XXXXXXXXXXXXXXXXXX
	0060	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	XXXXXXXXXXXXXXXXXX
	0070	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	XXXXXXXXXXXXXXXXXX
	0080	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	XXXXXXXXXXXXXXXXXX
	0090	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	XXXXXXXXXXXXXXXXXX
	00a0	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	XXXXXXXXXXXXXXXXXX
	00b0	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	XXXXXXXXXXXXXXXXXX
	00c0	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	XXXXXXXXXXXXXXXXXX
	00d0	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	XXXXXXXXXXXXXXXXXX
	00e0	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	XXXXXXXXXXXXXXXXXX
	00f0	58	25	75	39	30	39	30	25	75	36	38	35	38	25	75	63	X%u9090%u6858%uc
<i>Machine code for exploit</i>	0100	62	64	33	25	75	37	38	30	31	25	75	39	30	39	30	25	bd3%u7801%u9090%
	0110	75	36	38	35	38	25	75	63	62	64	33	25	75	37	38	30	u6858%ucbd3%u780
	0120	31	25	75	39	30	39	30	25	75	36	38	35	38	25	75	63	1%u9090%u6858%uc
	0130	62	64	33	25	75	37	38	30	31	25	75	39	30	39	30	25	bd3%u7801%u9090%
	0140	75	39	30	39	30	25	75	38	31	39	30	25	75	30	30	63	u9090%u8190%u00c
	0150	33	25	75	30	30	30	33	25	75	38	62	30	30	25	75	35	3%u0003%u8b00%u5
	0160	33	31	62	25	75	35	33	66	66	25	75	30	30	37	38	25	31b%u53ff%u0078%
	0170	75	30	30	30	30	25	75	30	30	3d	61	20	20	48	54	54	u0000%u00=a HTTP
	0180	50	2f	31	2e	30	0d	0a	43	6f	6e	74	65	6e	74	2d	74	P/1.0..Content-t
	0190	79	70	65	3a	20	74	65	78	74	2f	78	6d	6c	0a	43	6f	ype: text/xml.Co
	01a0	6e	74	65	6e	74	2d	6c	65	6e	67	74	68	3a	20	33	33	ntent-length: 33
	01b0	37	39	20	0d	0a	0d	0a	c8	c8	01	00	60	e8	03	00	00	79`....
	01c0	00	cc	eb	fe	64	67	ff	36	00	00	64	67	89	26	00	00dg.6..dg.&..
	01d0	e8	df	02	00	00	68	04	01	00	00	8d	85	5c	fe	ff	ffh.....\...
	01e0	50	ff	55	9c	8d	85	5c	fe	ff	ff	50	ff	55	98	8b	40	P.U...\...P.U..@
	01f0	10	8b	08	89	8d	58	fe	ff	ff	ff	55	e4	3d	04	04	00X....U.=...

The Instant Messaging Wars of 1999

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



The Instant Messaging Wars of 1999

- AOL wanted to prevent MSN clients from accessing its servers.
 - But, the MSN clients mimicked the AIM protocol exactly.
 - And, AOL didn't want to change their protocol – that would require that all of their users download a new client.
- Instead, AOL exploited a buffer overrun bug in their own client!
 - One case of the protocol reads a string into a buffer of size 0x100
 - AIM code was not checking that the string would fit into this size buffer
- AOL crafted an attack on their own client that would:
 - Overflow buffer with about 40 bytes of x86 code
 - Exploit code causes client to read data from a portion of the AIM binary
 - Send that data back to the server, as a kind of “signature”
 - AOL server would only accept the client if it sent back the right signature
- This attack would not work on the MSN client, of course.
 - So MSN clients could not send back the correct signature, and would be rejected.

The Instant Messaging Wars of 1999

- Microsoft caught onto this pretty quick.
 - Changed the MSN client so it would send back the right signature.
- AOL just changed the attack code slightly so a different signature would be sent back to the server.
- Microsoft changed their clients again...
- This skirmish went back and forth 13 times!

Worm vs. Virus

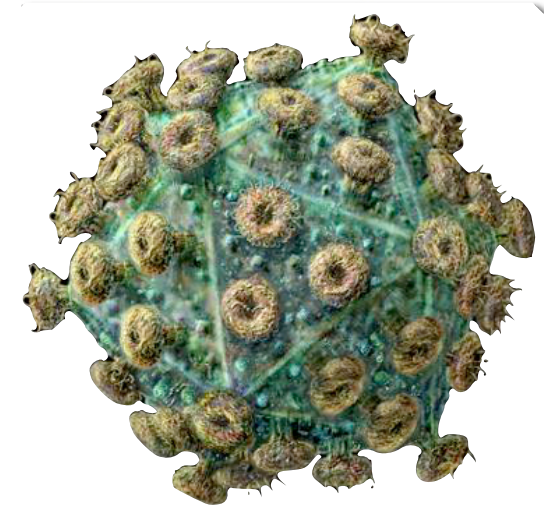
- Worm

- Spreads from one computer to another
- Can propagate fully working version of itself to another machine
- Can spread without human interaction
- Derived from *tapeworm*: a parasite that lives inside a host and uses its resources to maintain itself.



- Virus

- Spreads from one computer to another
- Attaches itself to program or file
- Cannot exist independently
- Requires a human action to spread (e.g., executing or opening a file)



Next Lecture

- Program optimizations
 - How to greatly improve the performance of your C programs with some simple changes
- Code motion
- Strength reduction
- Common subexpression elimination
- Loop unrolling