

Introduction to Software Vulnerabilities: Buffer Overflows

CMSC 23200, Spring 2026, Lecture 3

David Cash and Grant Ho

University of Chicago

Logistics

- **Assignment 1** has two parts, both due Thursday at 11:59pm
 - **1A:** Thread Modeling (“pen and paper”)
 - **1B:** Requires VM access. Very little code, but start a few days early in case you have a technical snag.
- **Assignment 2** will be out Friday and cover this weeks lectures
- No discussion section this week
- Discussion sections start next week on Monday

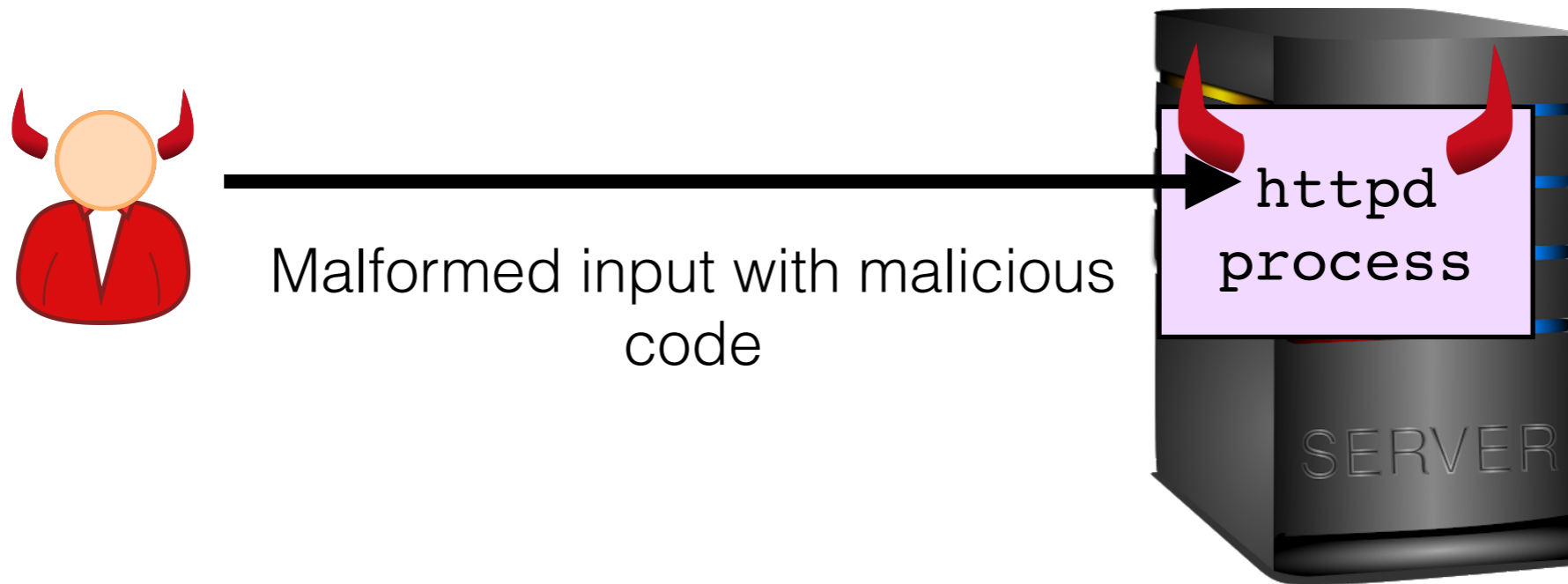
Outline for Lecture 3

1. Overview of software exploits
2. Memory layout and function calls in a process
3. Stack-based buffer overflow attacks
4. Heap-based buffer overflow attacks

Outline for Lecture 3

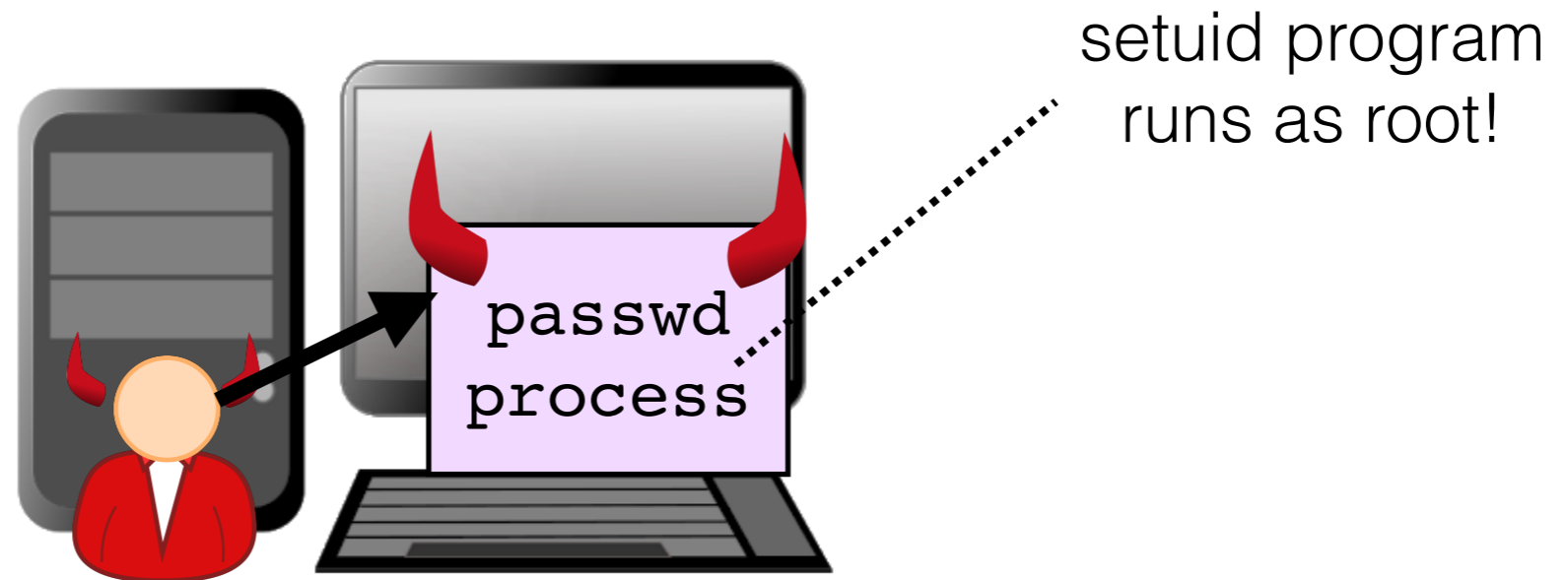
- 1. Overview of software exploits**
2. Memory layout and function calls in a process
3. Stack-based buffer overflow attacks
4. Heap-based buffer overflow attacks

Software Attacks: External Attacker Setting



- External attacker targets a remote resource like a web server
- Attacker aims to corrupt server process memory and run its own code
- Ultimate goal is often to monetize server as a bot
 - But crashing the system is sometimes good enough

Software Attacks: Local Attacker Settings



- Attacker has account on target system (e.g. like we do on CS dept servers)
- Attacker wants to escalate privilege (e.g. get a root account)
- setuid programs are good targets: if attacker can take control of them, it can run code as owner (frequently root)

Software Vulnerabilities are Very Common

- CVEs = “common vulnerability and Exposures”

The screenshot shows the CVE search interface. At the top, there is a navigation bar with links for About, Partner Information, Program Organization, Downloads, Resources & Support, and Report/Request. A search bar contains the term 'overflow' and a 'Search' button. Below the search bar, a notice states: "Expanded keyword searching of CVE Records (with limitations) is now available in the search box above. Learn more here." The main section is titled "Search Results" and indicates "Showing 1 - 25 of 25,780 results for overflow". There are controls for "Show:" (set to 25) and "Sort by:" (set to CVE ID (new to old)). Three results are visible:

- CVE-2026-5164** (CNA: Red Hat, Inc.): A flaw was found in virtio-win. The 'RhelDoUnMap()' function does not properly validate the number of descriptors provided by a user during an unmap request. A local user could exploit...
- CVE-2026-5121** (CNA: Red Hat, Inc.): A flaw was found in libarchive. On 32-bit systems, an integer overflow vulnerability exists in the zisofs block pointer allocation logic. A remote attacker can exploit this by providing a...
- CVE-2026-5046** (CNA: VulDB): A flaw has been found in Tenda FH1201 1.2.0.14(408). Affected is the function formWriExtraSet of the file /goform/WriExtraSet of the component Parameter Handler. Executing a manipulation of the argument GO...

- In 2021, while teaching this class, Linux sudo was broken!
- Department machines were vulnerable.
- Was it ok to test this?

The screenshot shows the detail page for CVE-2021-3156. The title is "CVE-2021-3156 Detail".

Description

Sudo before 1.9.5p2 contains an off-by-one error that can result in a heap-based buffer overflow, which allows privilege escalation to root via "sudoedit -s" and a command-line argument that ends with a single backslash character.

Metrics

CVSS Version 4.0 | **CVSS Version 3.x** | CVSS Version 2.0

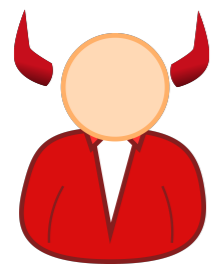
NVD enrichment efforts reference publicly available information to associate vector strings. CVSS information contributed by other sources is also displayed.

CVSS 3.x Severity and Vector Strings:

NIST: NVD	Base Score: 7.8 HIGH	Vector: CVSS:3.1/AV:L/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:H
ADP: CISA-ADP	Base Score: 7.8 HIGH	Vector: CVSS:3.1/AV:L/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:H

Two Basic Principles of Most Attacks

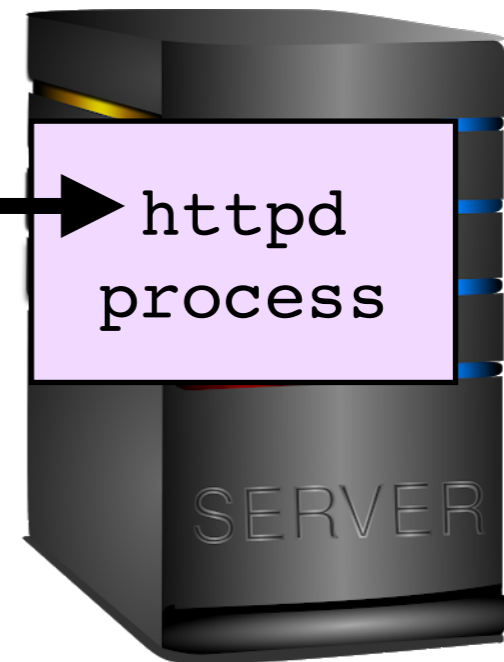
- Adversaries get to inject *their* bytes into *your* machine
- “Data” and “Code” are interchangeable; They are fundamentally the same “thing”.



GET /index.html HTTP/1.1

vs.

```
GET /index.htmlh6\ک??` :??  
L??S)??Z?vm??q`?%?~???M?  
EK???'?_?|Cg7L??s3?
```



Some Classes of Software Vulnerabilities

- Memory management
- Integer overflow and casting
- Unsanitized input fed to unprotected functions (e.g. `printf`)
- ...

Outline for Lecture 3

1. Overview of software exploits

2. Memory layout and function calls in a process

3. Stack-based buffer overflow attacks

4. Heap-based buffer overflow attacks

Memory Layout of a Process (in Linux)

.text: Machine executable code

.data: Global initialized static variables

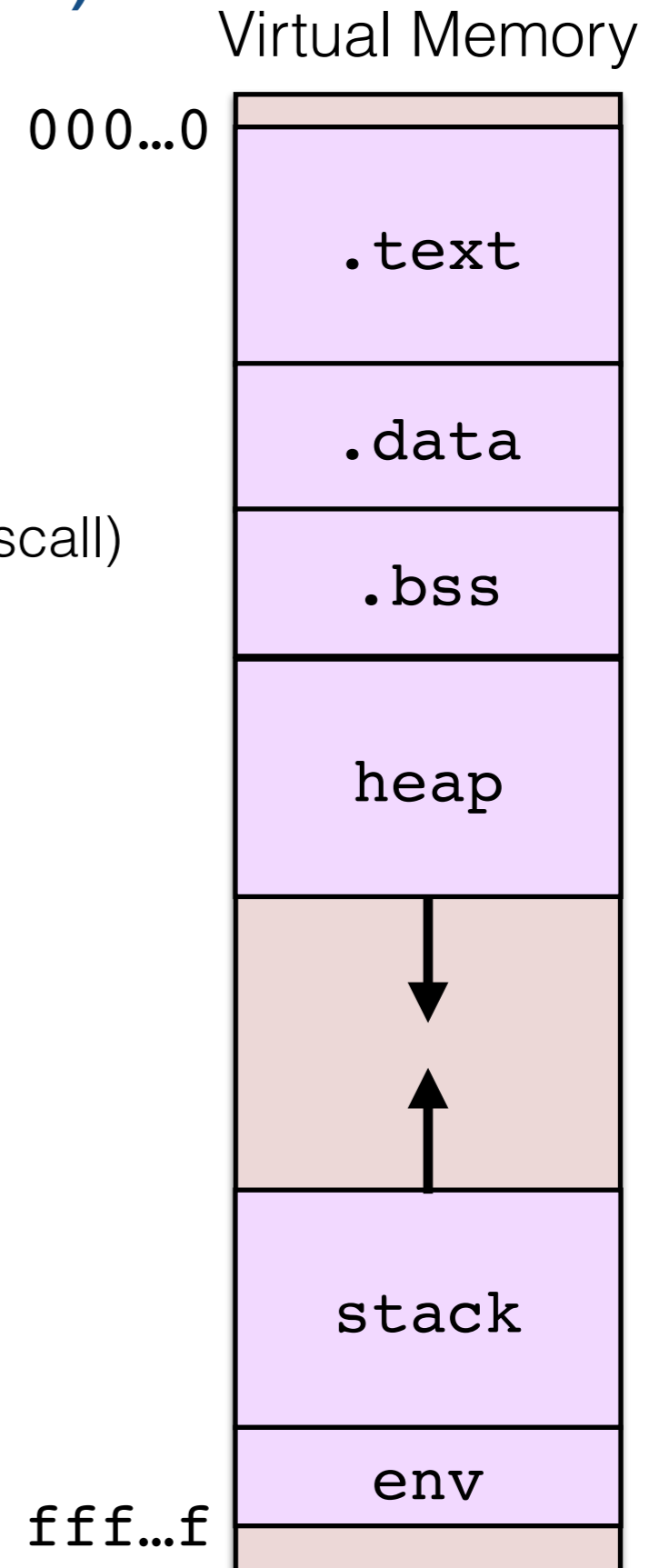
.bss: Global uninitialized variables (“block starting symbol”)

heap: Dynamically allocated memory (via `brk/sbrk/mmap` syscall)

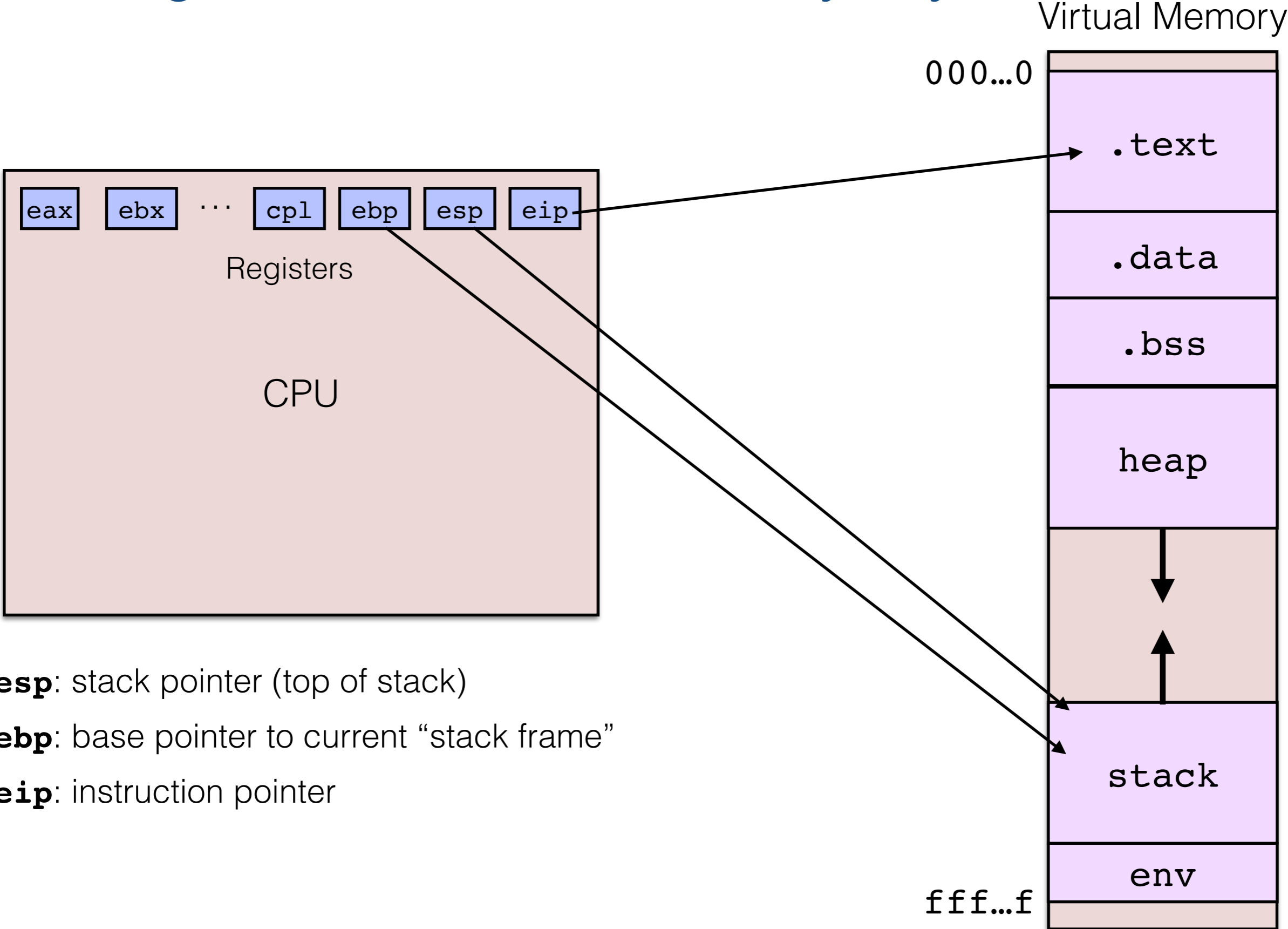
stack: Local variables and functional call info

env: Environment variables (PATH etc)

(Demo!)



x86 Registers and Virtual Memory Layout

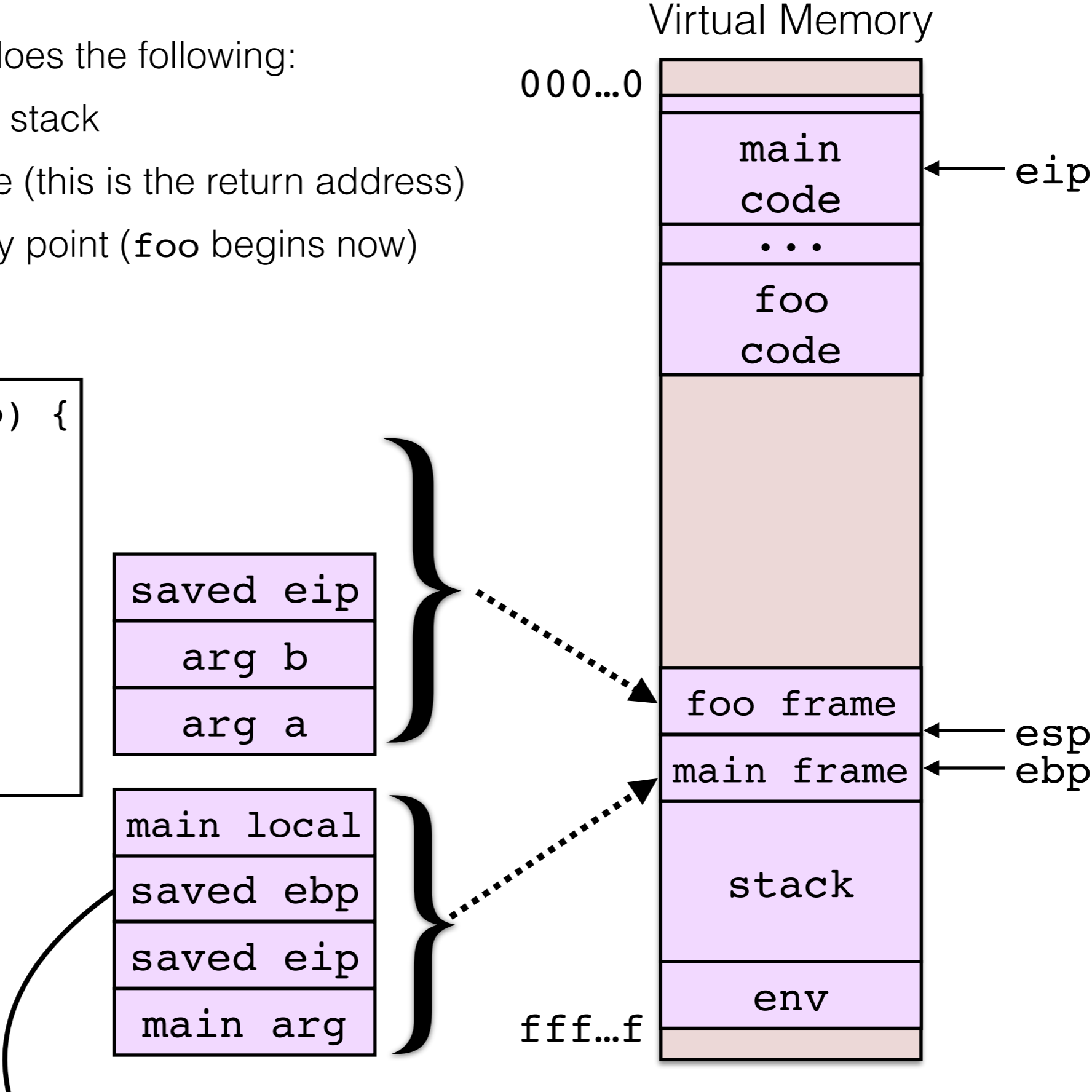


Calling a Function in C

To call `foo(a,b)`, `main` does the following:

- 1. Push args for `foo` onto stack
- 2. Push `eip` register value (this is the return address)
- 3. Point `eip` to `foo`'s entry point (`foo` begins now)

```
int foo(int a, int b) {  
    int d = 1;  
    return a+b+d;  
}  
  
int main(...) {  
    ...  
    int x = foo(5, 6);  
    ...  
}
```

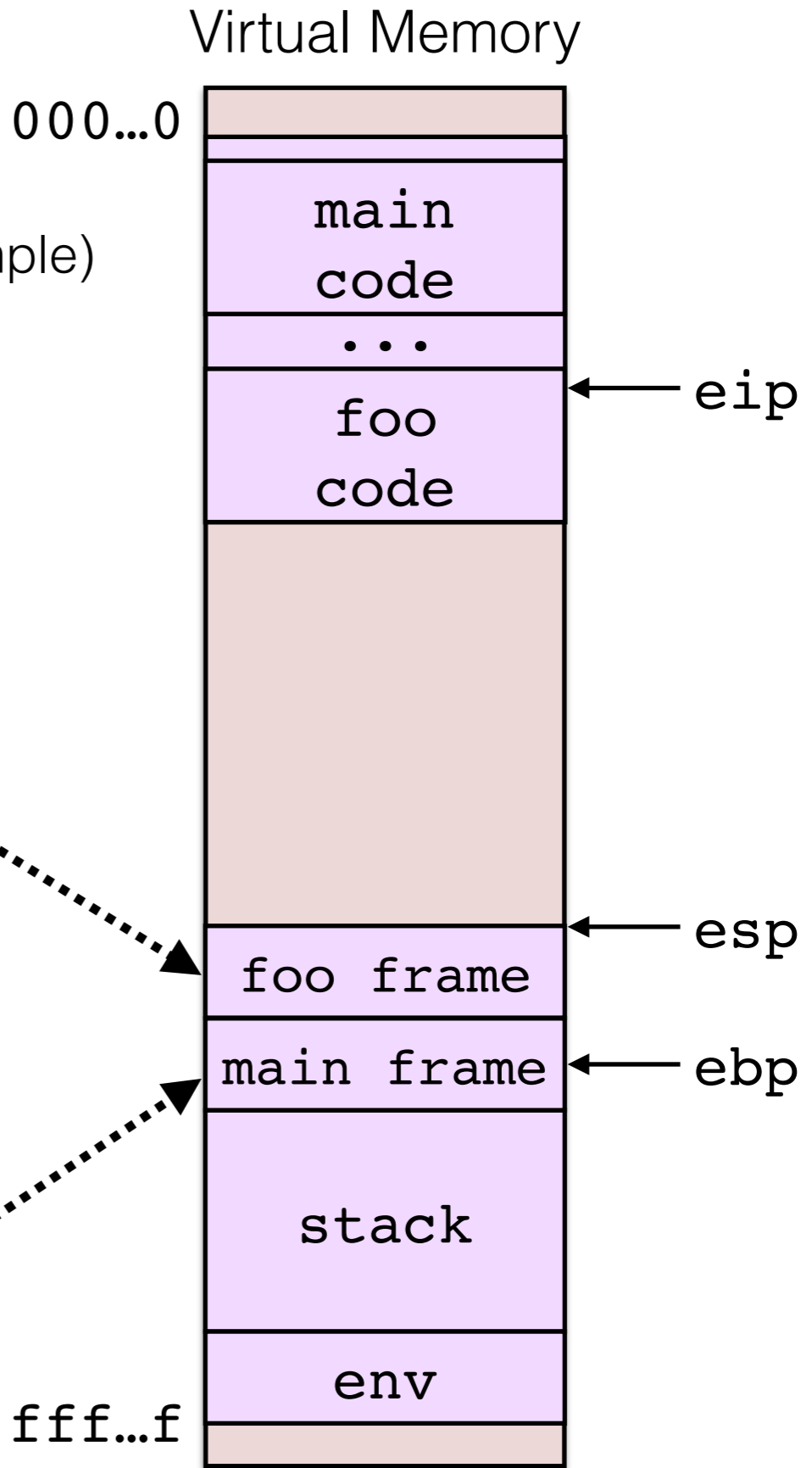
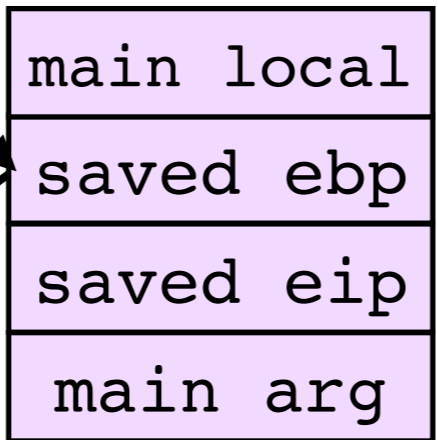
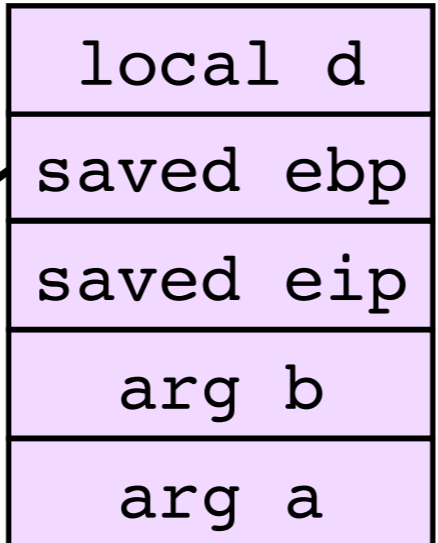


Executing a Function in C

foo will start by:

- 1. Push `ebp` value onto stack, update `ebp` register
- 2. Allocate local variables on stack (this is `d` in our example)
- 3. Run rest of `foo`'s code

```
int foo(int a, int b) {  
    int d = 1;  
    return a+b+d;  
}  
  
int main(...) {  
    ...  
    int x = foo(5, 6);  
    ...  
}
```

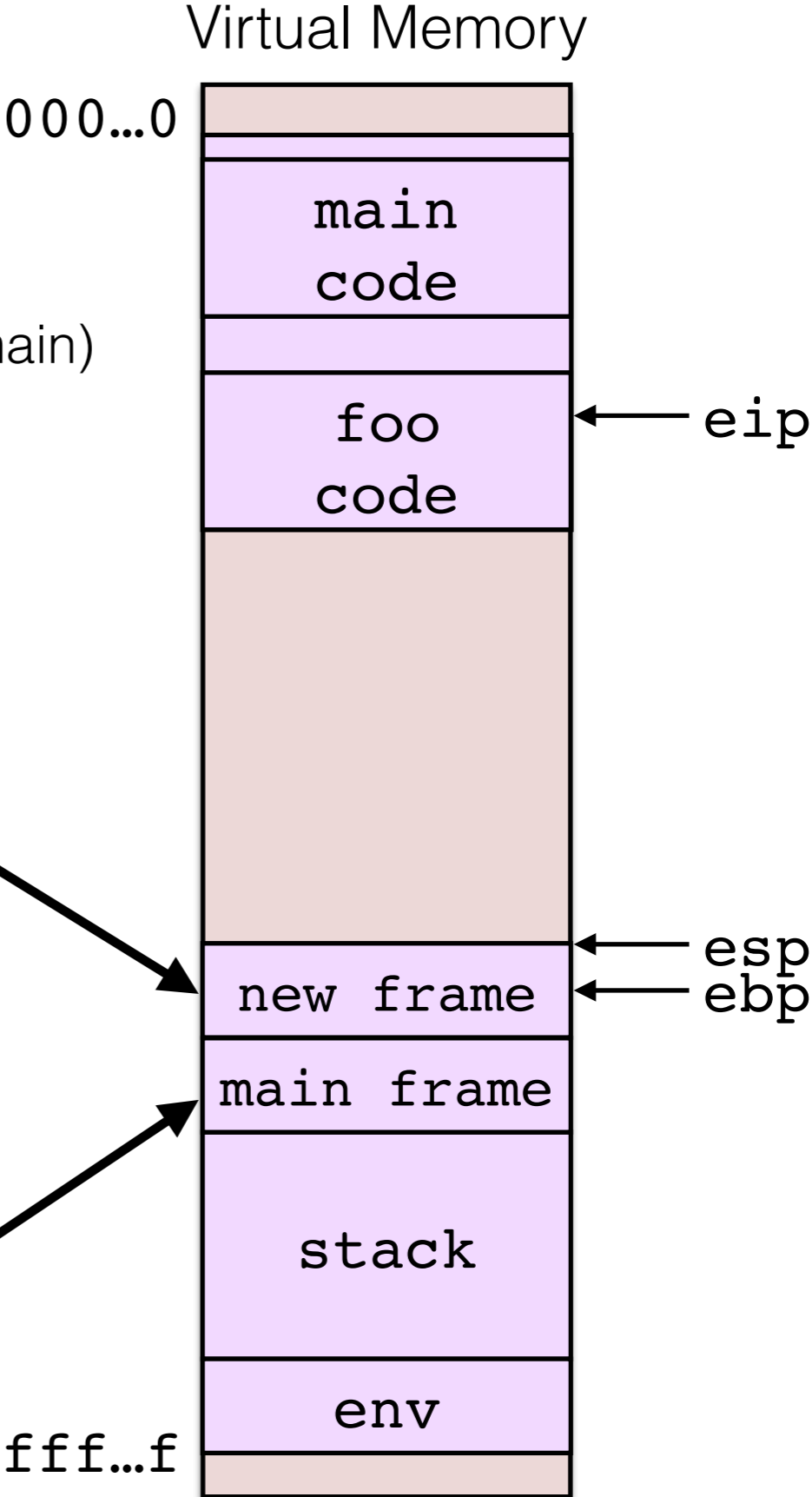
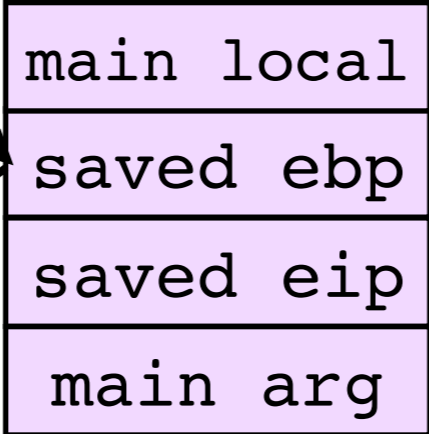
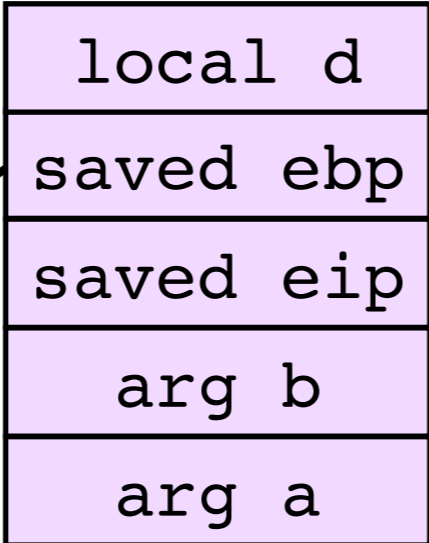


Returning from a function

When `foo` is finished:

- Pop local variables (move `esp` down)
- Pop (move) saved `ebp` to `ebp` register
- Pop (move) saved `eip` to `eip` register (this returns to main)
- `main` pops arguments

```
int foo(int a, int b) {  
    int d = 1;  
    return a+b+d;  
}  
  
int main(...) {  
    ...  
    int x = foo(5, 6);  
    ...  
}
```



Outline for Lecture 3

1. Overview of software exploits
2. Memory layout and function calls in a process
- 3. Stack-based buffer overflow attacks**
4. Heap-based buffer overflow attacks

Typical Problem: Overflowing a buffer on the stack

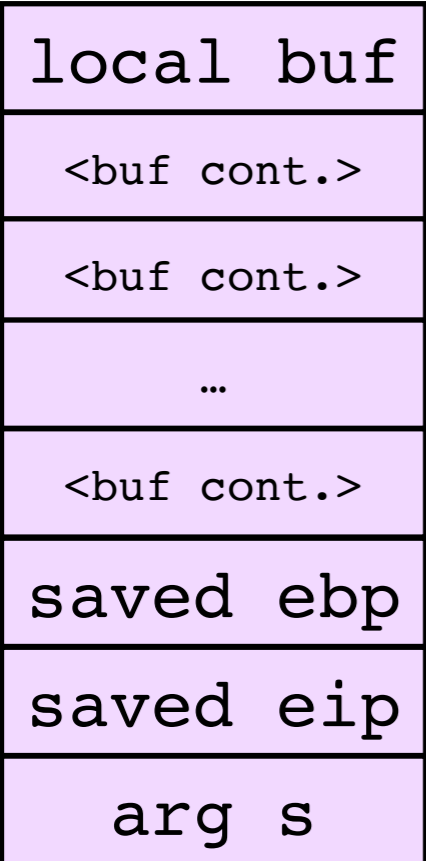
Function `bad` copies a string into a 64 character buffer.

- `strcpy` continues copying until it hits NULL character!
- If `s` points to longer string, this overwrites rest of stack frame.
- Most importantly saved `eip` is changed, altering control flow.

```
void bad(char *s) {  
    char buf[64];  
    strcpy(buf, s);  
}
```

`s="AAAA...AAAA"` (70 or more characters)

Frame before `strcpy` Frame after `strcpy`

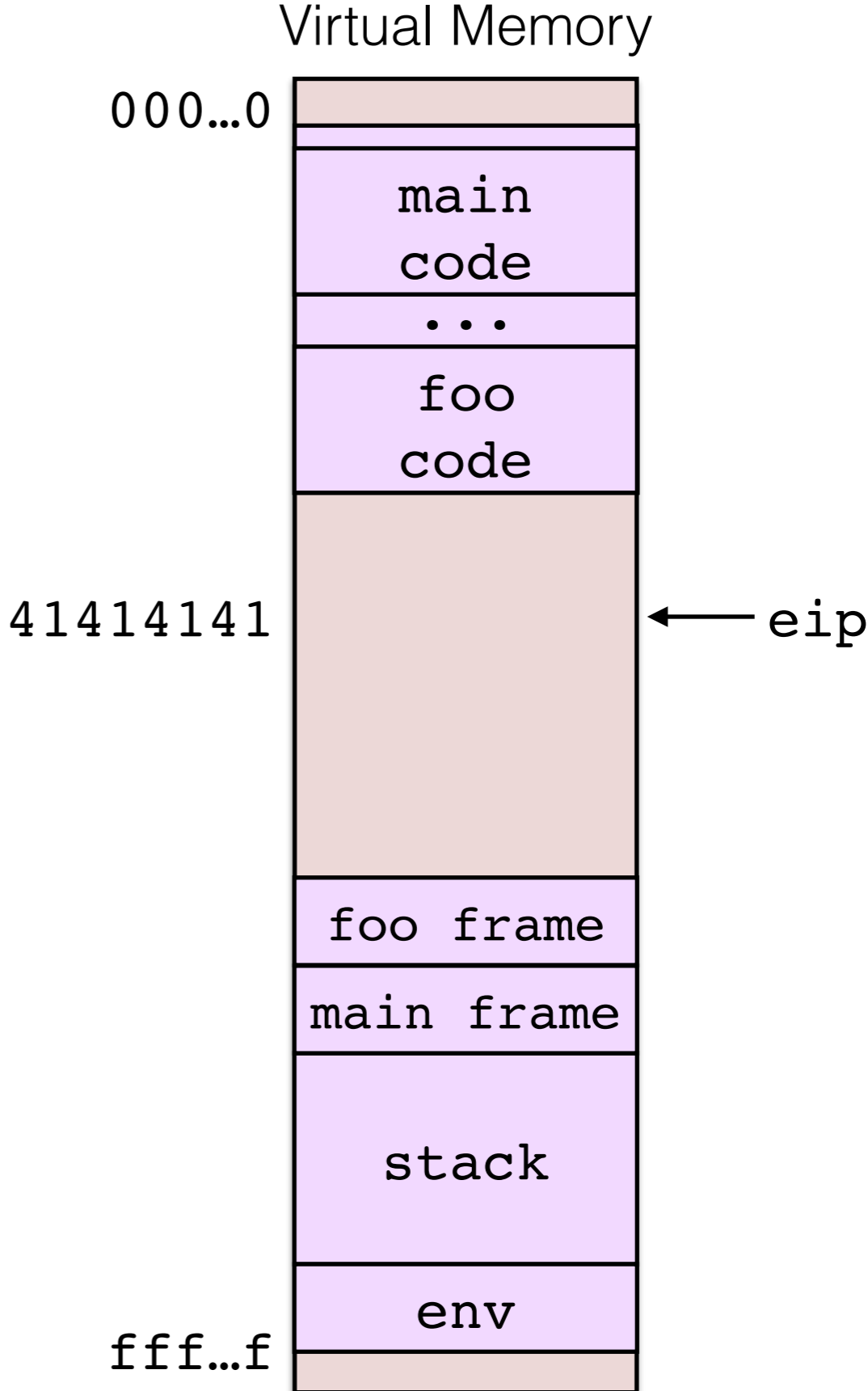


saved `eip` should be here!
`AAAA=0x41414141` will be used
as return address

What will happen? SEGFAULT!

Segmentation Fault

Address 0x41414141 is unlikely to be mapped.



How to exploit a stack buffer overflow

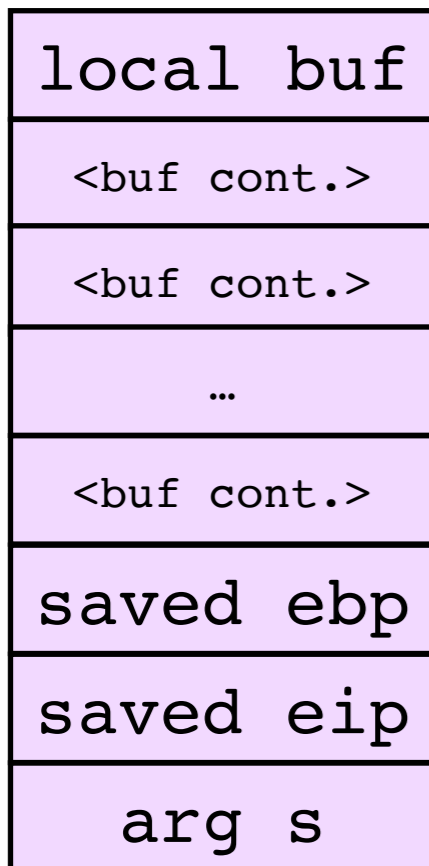
Suppose attacker can cause bad to run with an `s` it chooses.

- Trick 1: Set correct bytes to *point back to input(!)*

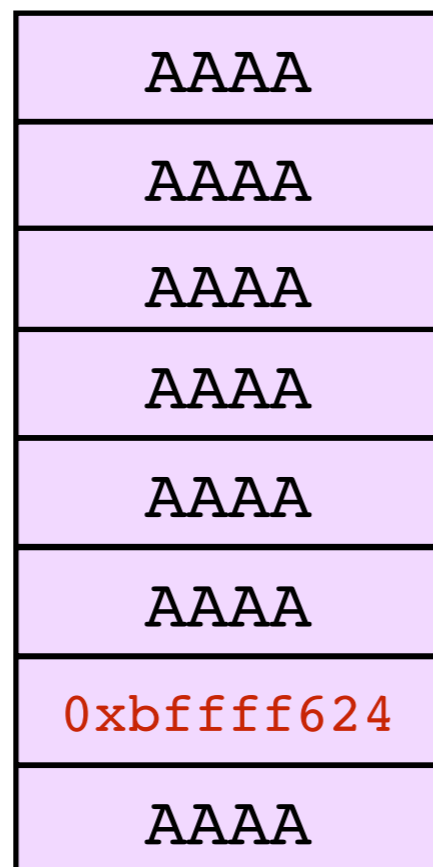
```
void bad(char *s) {  
    char buf[64];  
    strcpy(buf, s);  
}
```

`s="AAAAA...AAAA\x24\x66\xff\xbfAAA..."`

Frame before strcpy



Frame after strcpy



0xbffff624

Well-chosen (unprintable) characters used as an address for `eip`!

What will happen? Illegal instruction!

How to exploit a stack buffer overflow

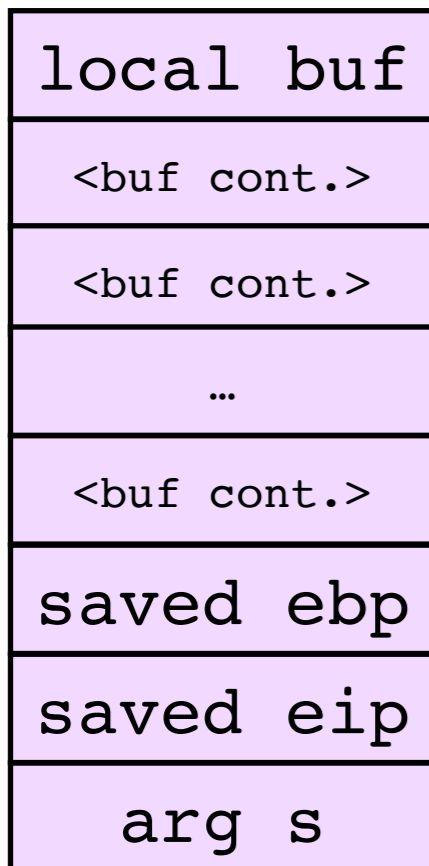
Suppose attacker can cause bad to run with an `s` it chooses.

- Trick 1: Set correct bytes to *point back to input(!)*
- Trick 2: Make input *executable machine code(!)*

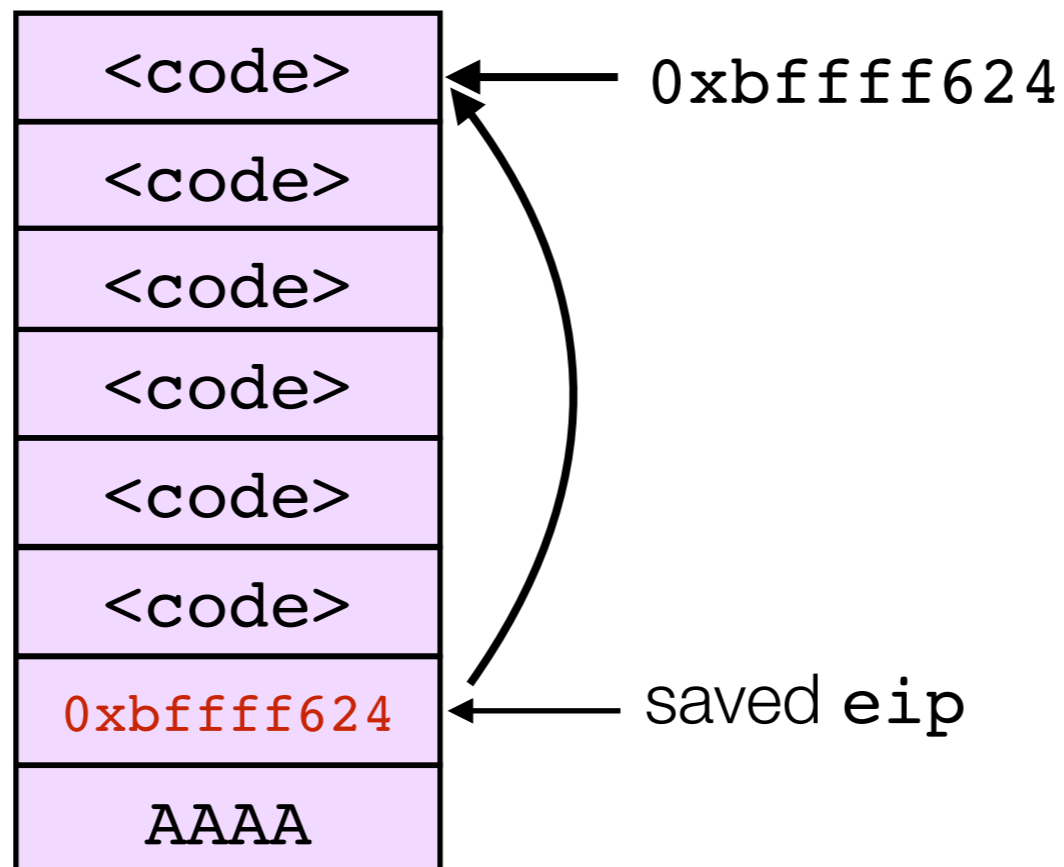
```
void bad(char *s) {  
    char buf[64];  
    strcpy(buf, s);  
}
```

`s="<machine code>\x24\x6\xff\xbfAAA..."`

Frame before strcpy



Frame after strcpy



What will happen?

Whatever we want!

What to put in for <code>?

The possibilities are endless!

- Spawn a shell
- Spawn a new service listening to network
- Change files
- ...

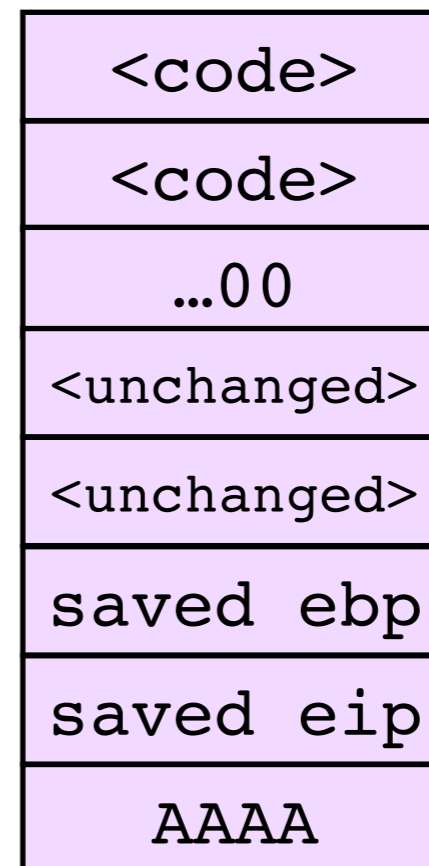
```
s="<machine code>\x24\x66\xff\xbfAAA..."
```

But wait... what about NULL bytes?

Solution: Find machine instructions with no NULLs!

- Can even find machine code with all alpha bytes.

Frame after strcpy



← strcpy
stopped here,
saving victim :(

Example Shellcode

```
char shellcode[] =  
"\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\x40xcd"  
"\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

- Basically equivalent to:

```
#include <stdio.h>  
void main() {  
    char *name[2];  
    name[0] = "/bin/sh";  
    name[1] = NULL;  
    execve(name[0], name, NULL);  
}
```

- Most bytes of this shellcode are not “printable”
- In some settings, attackers can only input printable bytes

Printable Shellcode

- By carefully choosing instructions, shellcode can consist of only ASCII (“printable”) letters/numbers

```
char shellcode[] =  
"IIIIIIIIIIIIIIIIII7QZjAXP0A0AkAAQ2AB2BB0BBABXP8ABuJIkLIxk2GpC0wpapk9IufQ9PpdLKF0"  
"dpLKSbv1NkQBB4LKcBq8d0lwrjUvVQYoNLu1U1SL32T1q0zaXO4M6ahGKRIBcbrwNkf2vplK3zE1Nkr"  
"lR1D88cRhfaKaRq1KaIa05Q9Cnksy4XzCdzBiNk5dlKgqn6dqYoL19QzoFmgqyWgHIpPuzV4CsMjXwK"  
"QmUtt5M4BxNk1HUtEQzs56nkF10KLKaHG1GqzslKwt1KGqJpK9PDTd7TCkckqq693jCaIom0sosobzn"  
"kr2XknmaMBHVSTrc0C0BHqgcCDr3oaDu8R1BW16c7K0XULxZ0S1C05PQ9jdqDrp3XEyOpBKgpyo9Eqz"  
"6kbyV08bIm2JfaqzTBU8zJ4okoYpIohUz72HFbePVqSlNi8fbJTPv6Rw0hJbKkVWRG1oKeLEIP1ev81"  
"GRHMgM9vXk09oHUqGBHadZL5k9qK08Ubw1WaxaerNrm0aIon51zwp1zfdaFV7u8eRJyxHaOk08UNc8x"  
"S0SNTmLKfVazqPsX5PfpS0EPaFazUP2Hbx0TbsIu9ozunsf3pj30Sf1CbwbH32HYhHQOKOjuos8xuPQ"  
"nUWwq8Cti9V1eIyZcAA";
```

English Shellcode

English Shellcode

Joshua Mason, Sam Small
Johns Hopkins University
Baltimore, MD
{josh, sam}@cs.jhu.edu

Fabian Monroe
University of North Carolina
Chapel Hill, NC
fabian@cs.unc.edu

Greg MacManus
iSIGHT Partners
Washington, DC
gmacmanus.edu@gmail.com

- shellcode can even be somewhat intelligible English text!

<pre>push %esp push \$20657265 imul %esi,20(%ebx),\$616D2061 push \$6F jb short \$22</pre>	54 68 65726520 6973 20 61206D61 6A 6F 72 20	There is a major
<pre>push \$20736120 push %ebx je short \$63 jb short \$22</pre>	68 20617320 53 74 61 72 20	h as Star
<pre>push %ebx push \$202E776F push %esp push \$6F662065 jb short \$6F</pre>	53 68 6F772E20 54 68 6520666F 72 6D	Show. The form
<pre>push %ebx je short \$63 je short \$67 jnb short \$22 inc %esp jb short \$77</pre>	53 74 61 74 65 73 20 44 72 75	States Dru
<pre>popad</pre>	61	a

```
char shellcode[] =
```

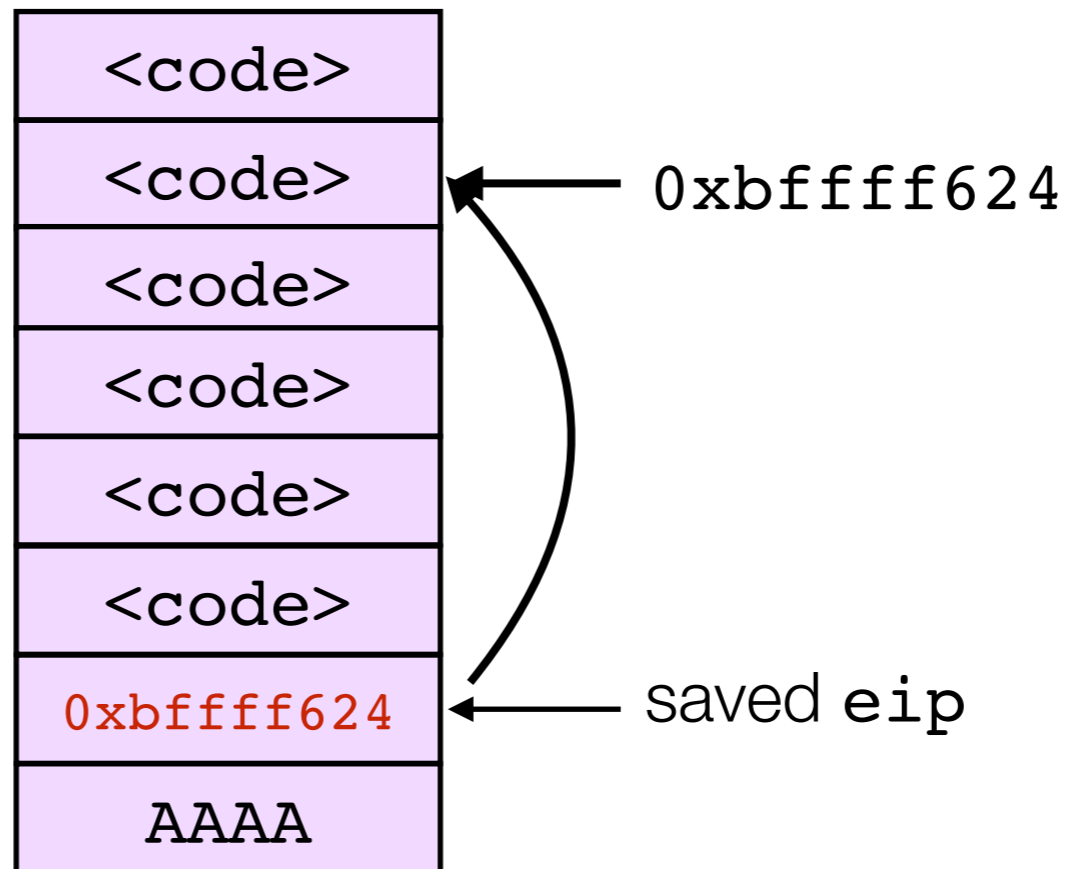
```
"There is a major center of economic activity, such as Star Trek, including The Ed "
"Sullivan Show. The former Soviet Union. International organization participation "
"Asian Development Bank, established in the United States Drug Enforcement "
"Administration, and the Palestinian territories, the International Telecommunication"
...
```

Finally, where did that magic address come from?

For Assignment 2: You'll use `gdb`

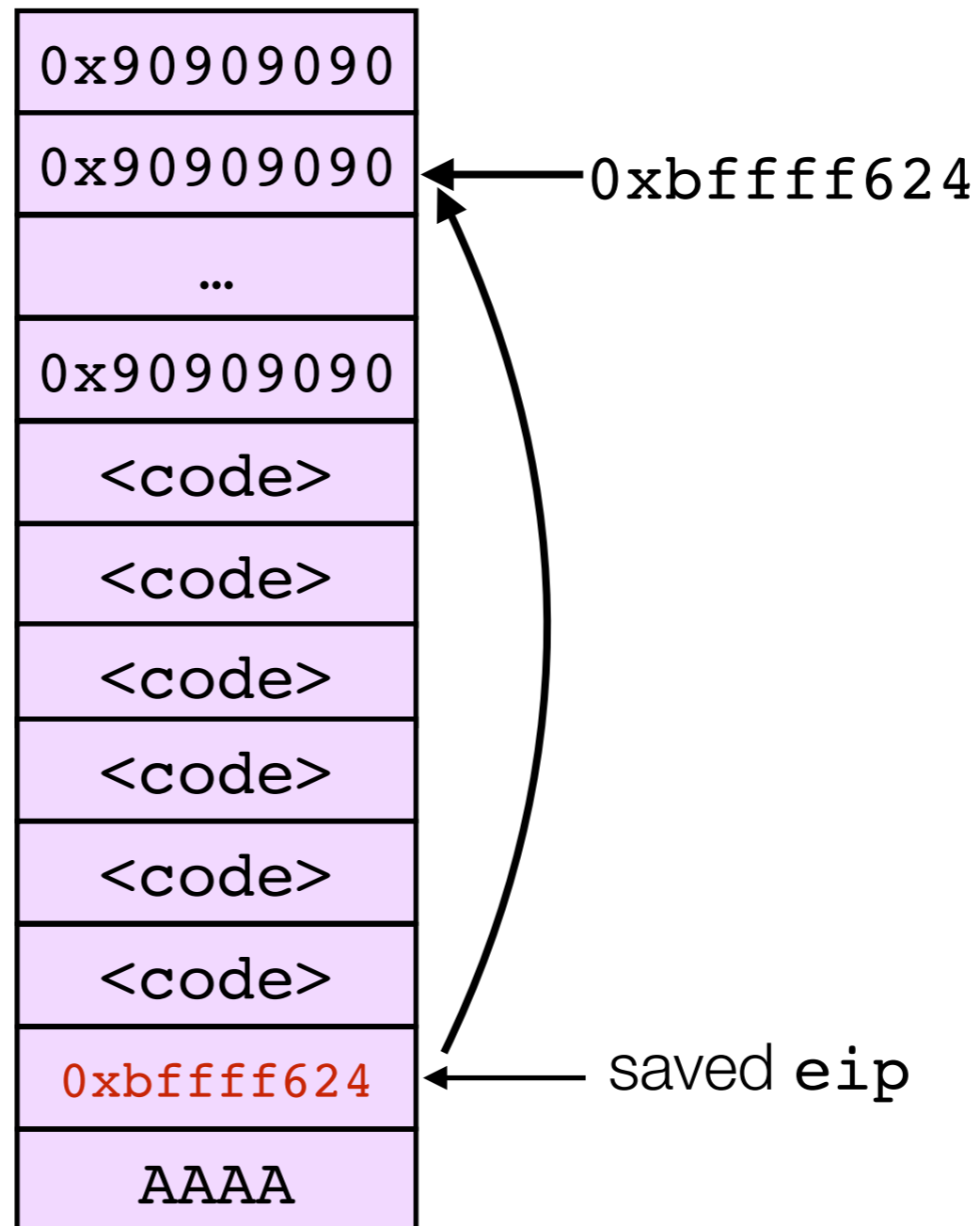
Two challenges in practice:

- Need to place address in correct spot
- Need address to jump to beginning of shellcode



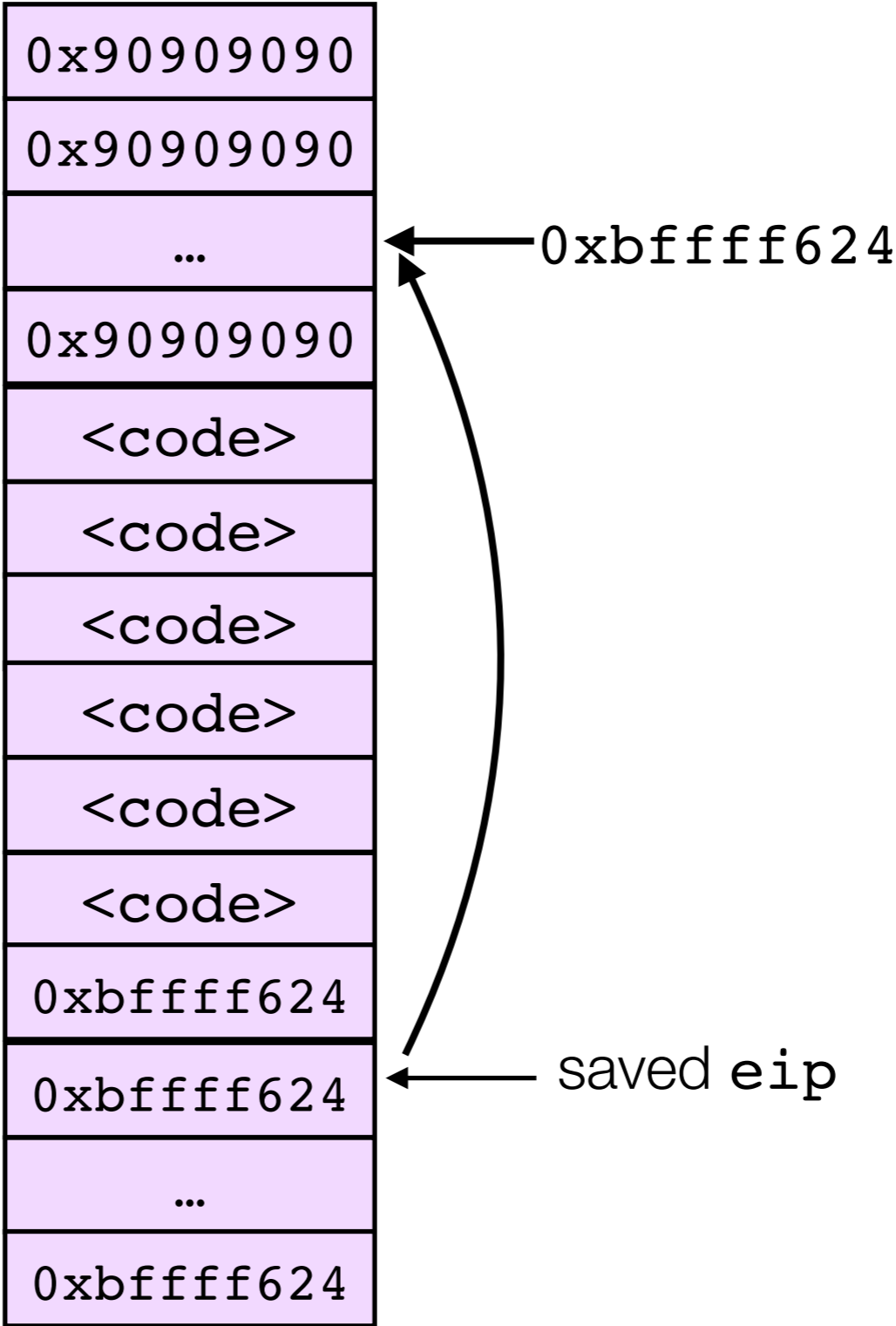
Technique #1: NOP Sleds

- Instruction `0x90` is `xchg eax, eax`, i.e. does nothing. This is a “No Op” or “NOP”.
- Just add a ton of NOPs (as many as you can, even many MB) and hope pointer lands there



Technique #2: Placing malicious EIP

— Simple: Just copy it many times



Outline for Lecture 3

1. Overview of software exploits
2. Memory layout and function calls in a process
3. Stack-based buffer overflow attacks
- 4. Heap-based buffer overflow attacks**

Heap Memory Vulnerabilities

Assume that:

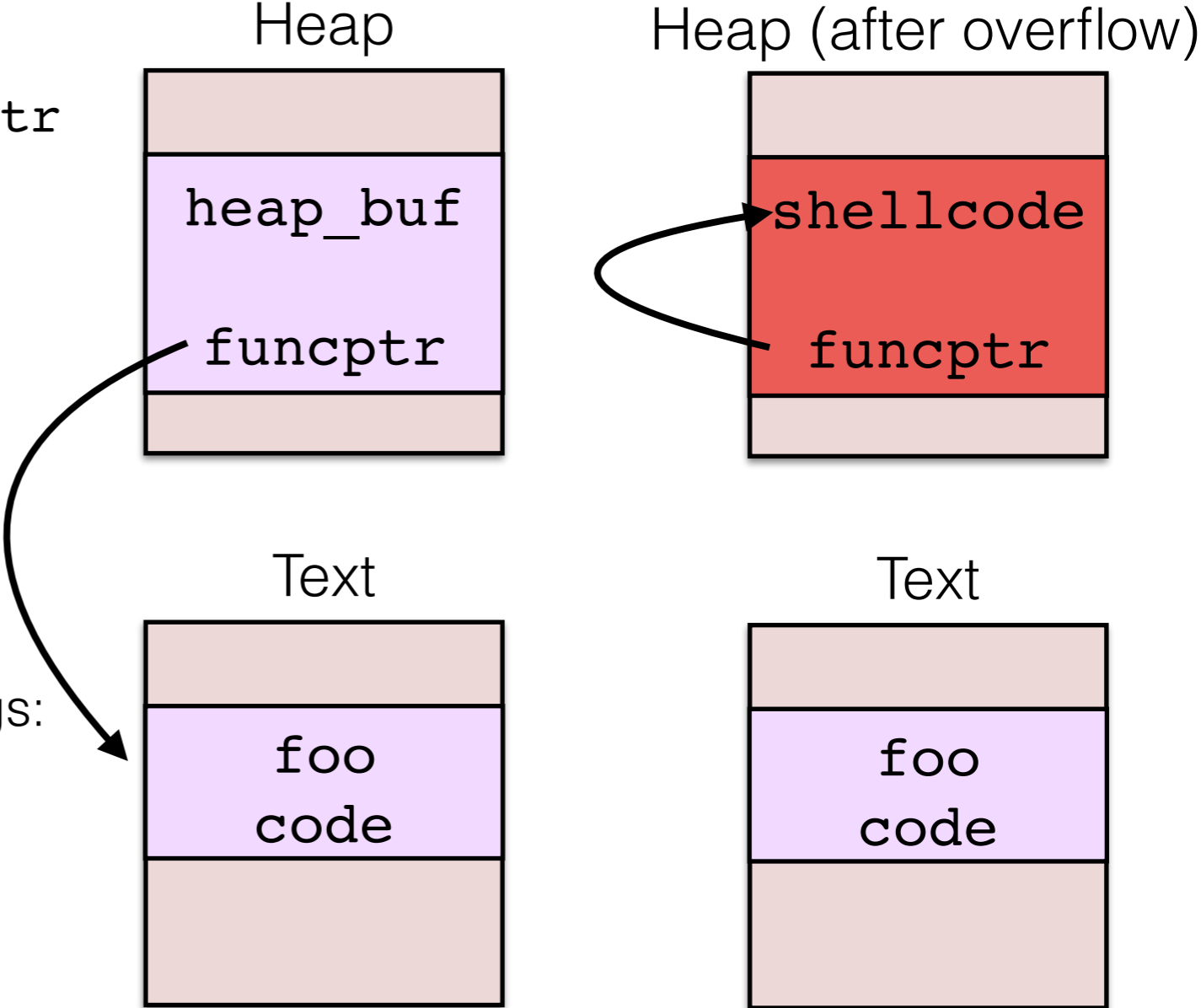
1. Program has heap variable (`heap_buf`)
2. A function pointer, stored in `heap_buf`, points to `foo`

Attack:

1. Overflow `heap_buf` and corrupt `funcptr`
2. Later calls using `funcptr` will call `shellcode`

Other notes:

- Attacker can overwrite program vars
 - e.g. change to `admin=true`
- Vulnerabilities possible from many bugs:
 - use-after-free
 - double-free
 - ...



The End