Introduction to Software Vulnerabilities: Buffer Overflows

CMSC 23200/33250, Winter 2021, Lecture 4

David Cash and Blase Ur

University of Chicago

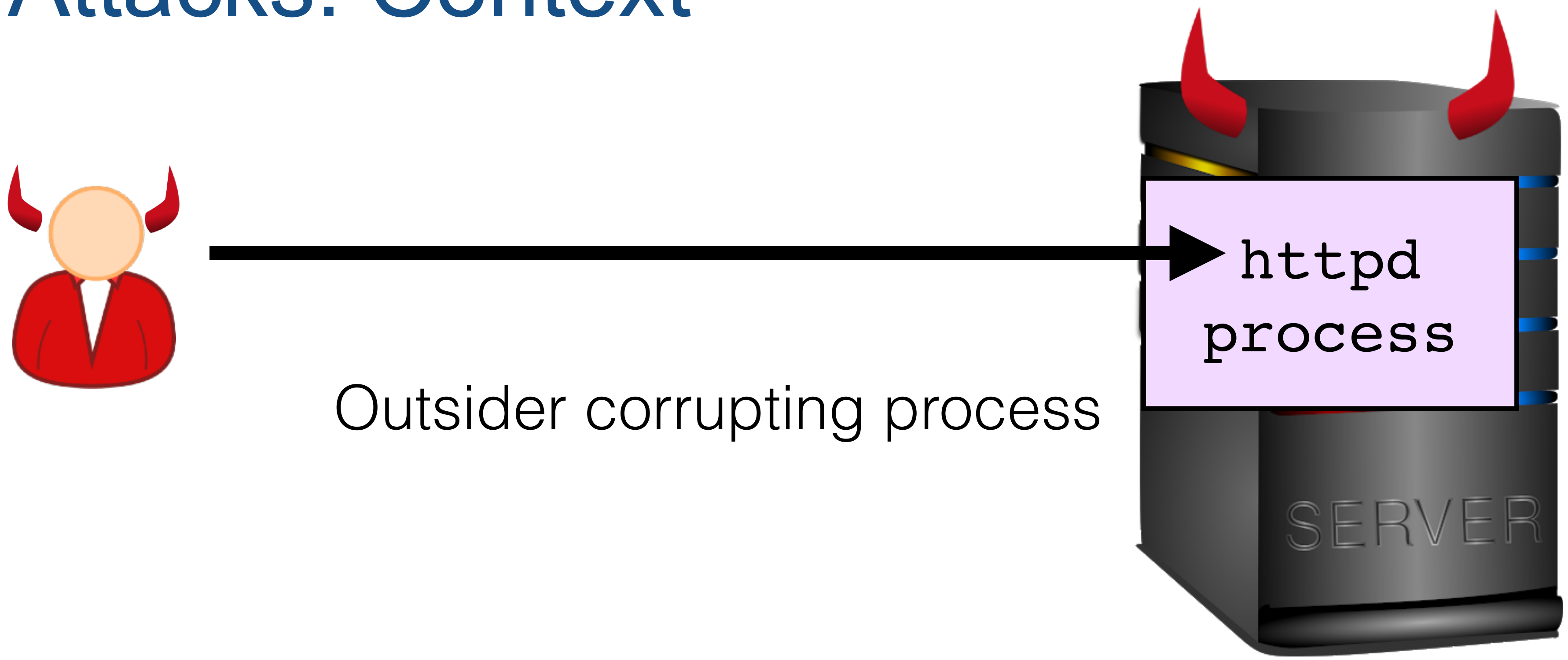
Outline for Lecture 4

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

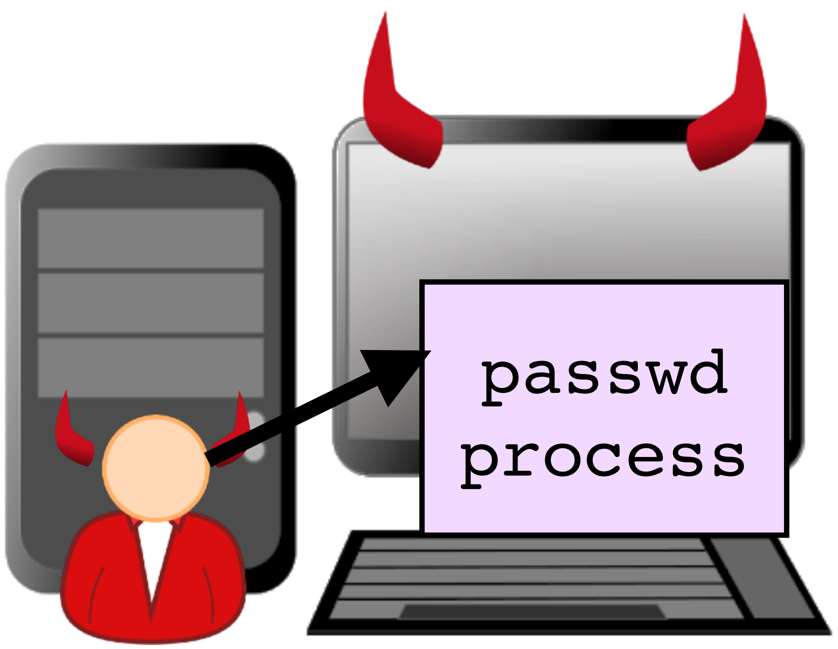
Outline for Lecture 4

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

Software Attacks: Context



Outsider corrupting process



Insider escalating privilege

- Usually want to monetize system
- Sometimes targeted espionage
- Happy crashing system as well!

Software Vulnerabilities are Very Common

- According to vulnerability researcher and author Dave Aitel:

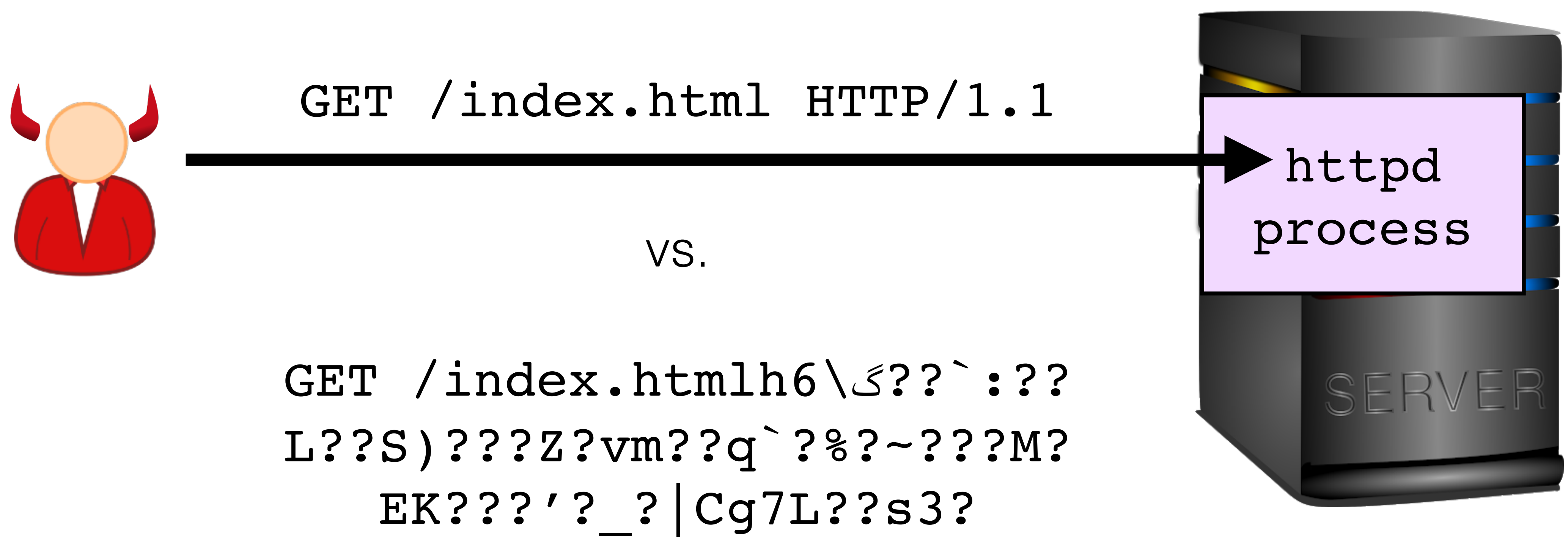
In **one hour** of analysis of a binary, one can find *potential* vulnerabilities

In **one week** of analysis of a binary, one can find *at least one good vulnerability*

In **one month** of analysis of a binary, one can find *a vulnerability that no one else will ever find.*

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”.



Some Classes of Software Vulnerabilities

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

Outline for Lecture 4

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

Memory Layout of a Process (in Linux)

.text: Machine executable code

.data: Global initialized static variables

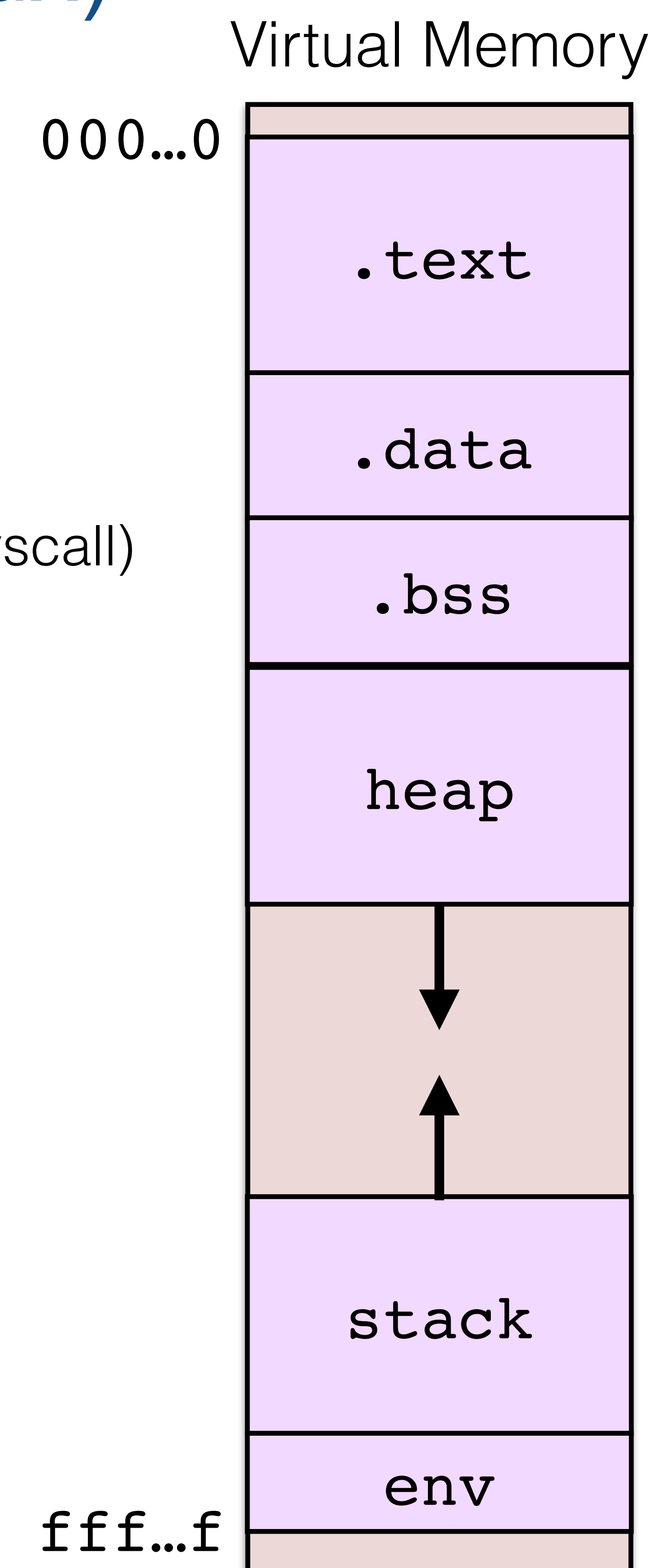
.bss: Global uninitialized variables ("block starting symbol")

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

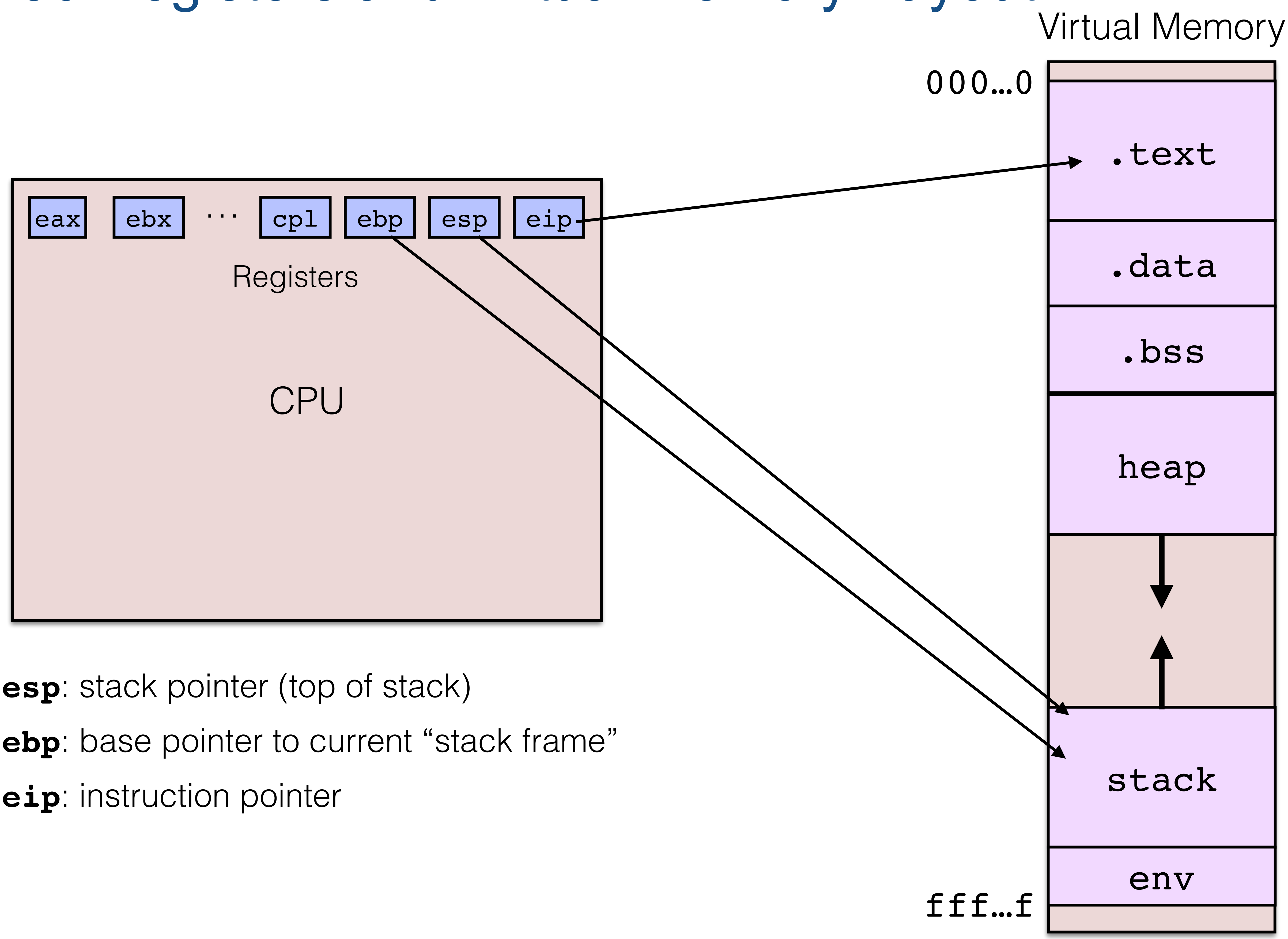
stack: Local variables and functional call info

env: Environment variables (PATH etc)

(Demo!)



x86 Registers and Virtual Memory Layout



esp: stack pointer (top of stack)

ebp: base pointer to current "stack frame"

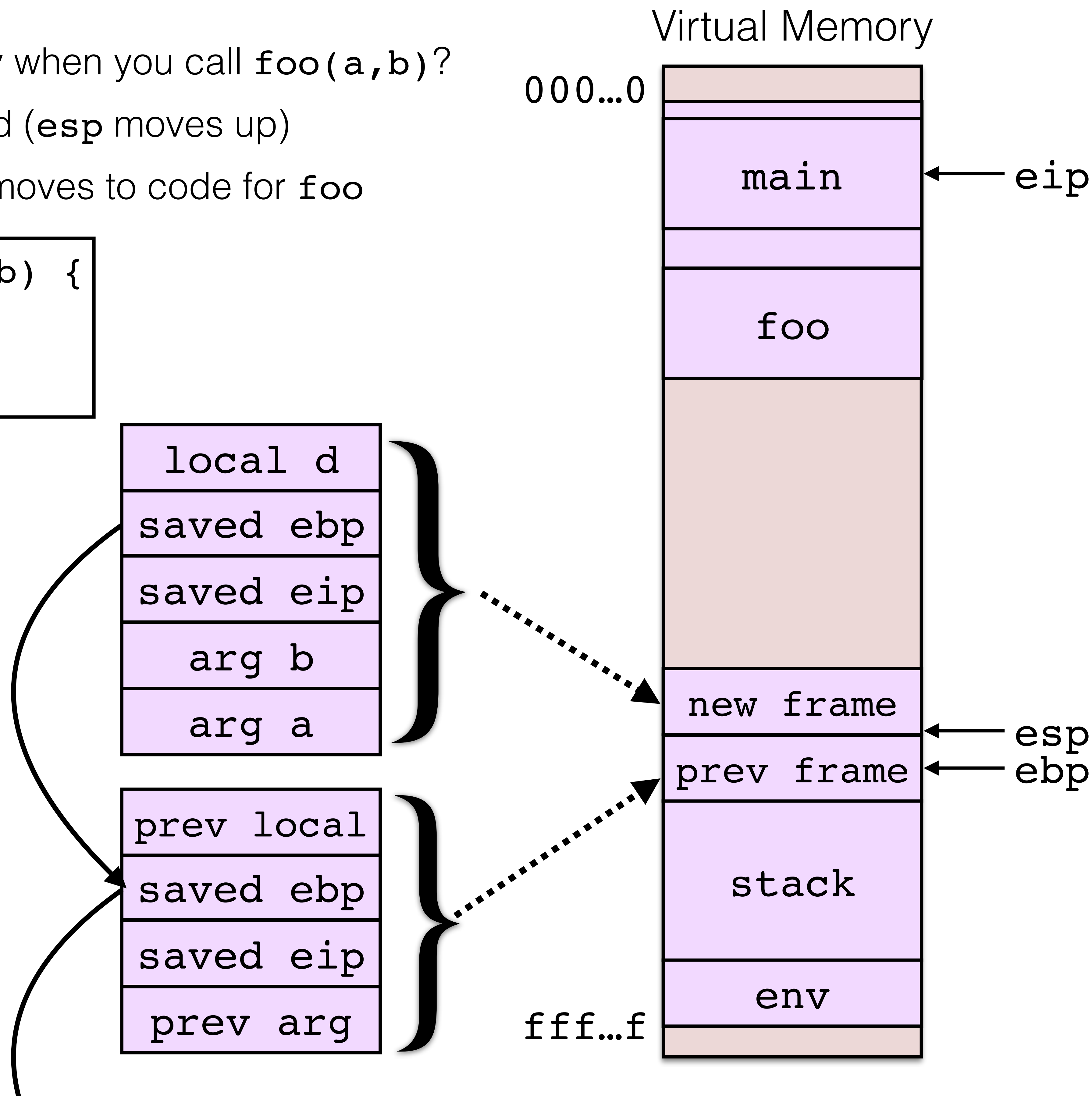
eip: instruction pointer

The Stack and Calling a Function in C

What happens to memory when you call `foo(a,b)`?

- A “stack frame” is added (`esp` moves up)
- Instruction pointer `eip` moves to code for `foo`

```
int foo(int a, int b) {
  int d = 1;
  return a+b+d;
}
```

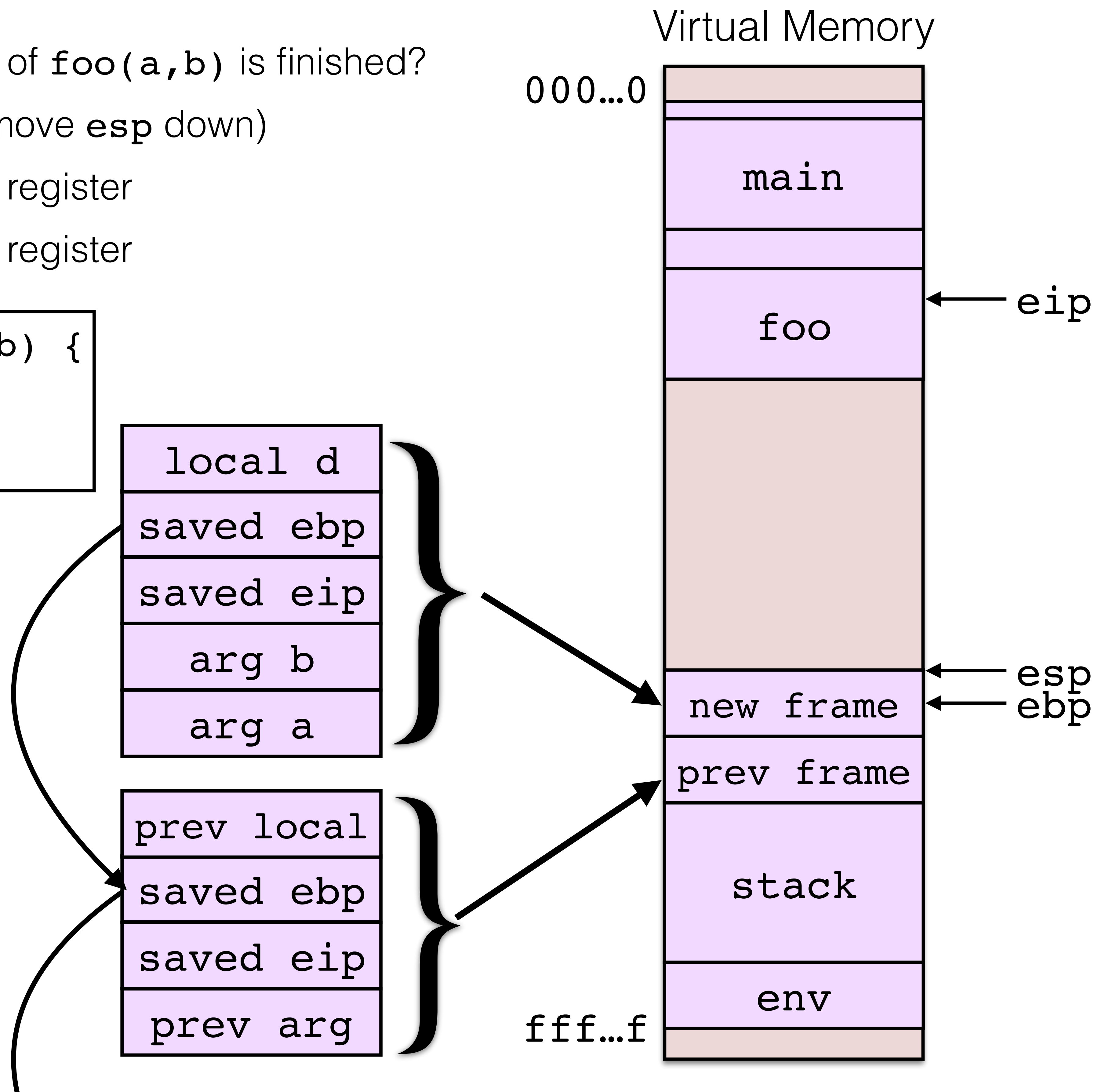


Returning from a function

What happens after code of **`foo(a,b)`** is finished?

- Pop frame off of stack (move **`esp`** down)
- Move saved **`ebp`** to **`ebp`** register
- Move saved **`eip`** to **`eip`** register

```
int foo(int a, int b) {
  int d = 1;
  return a+b+d;
}
```



Outline for Lecture 4

1. Overview of software exploits

2. Memory layout and function calls in a process

3. **Stack-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!

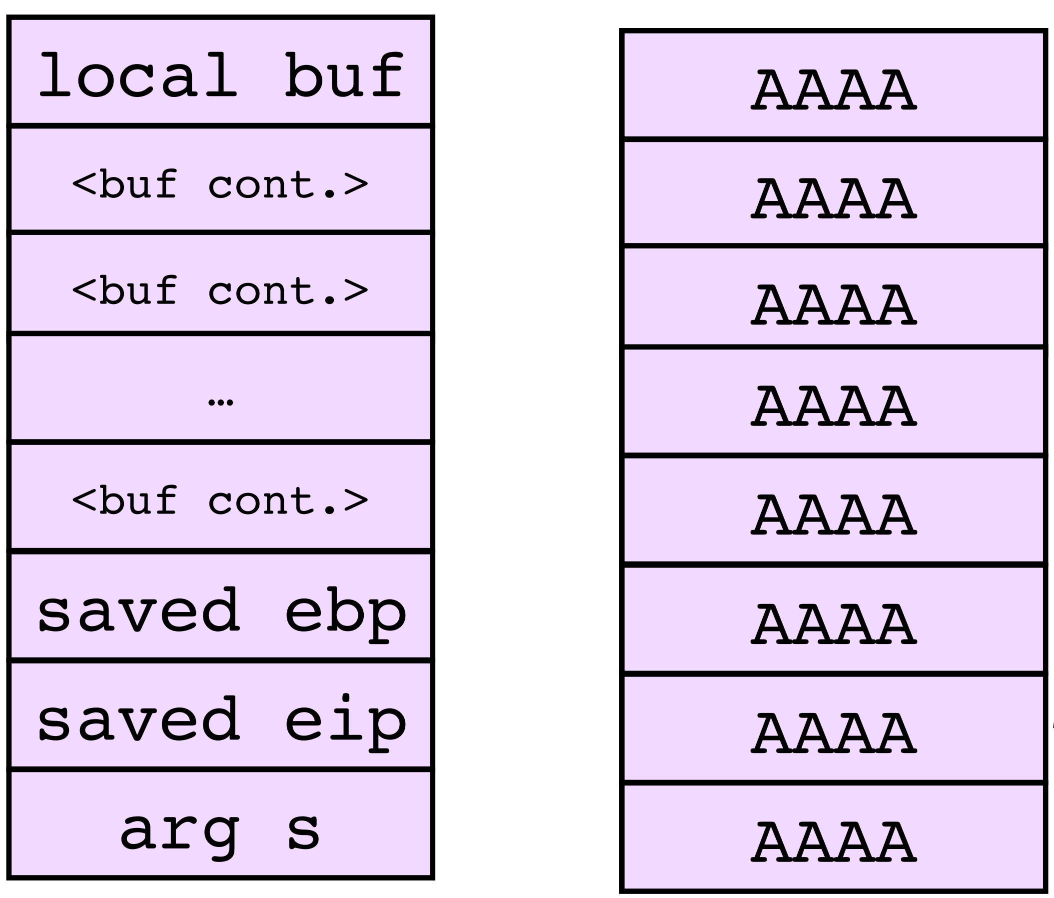
— If s points to longer string, this overwrites rest of stack frame.

— 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!

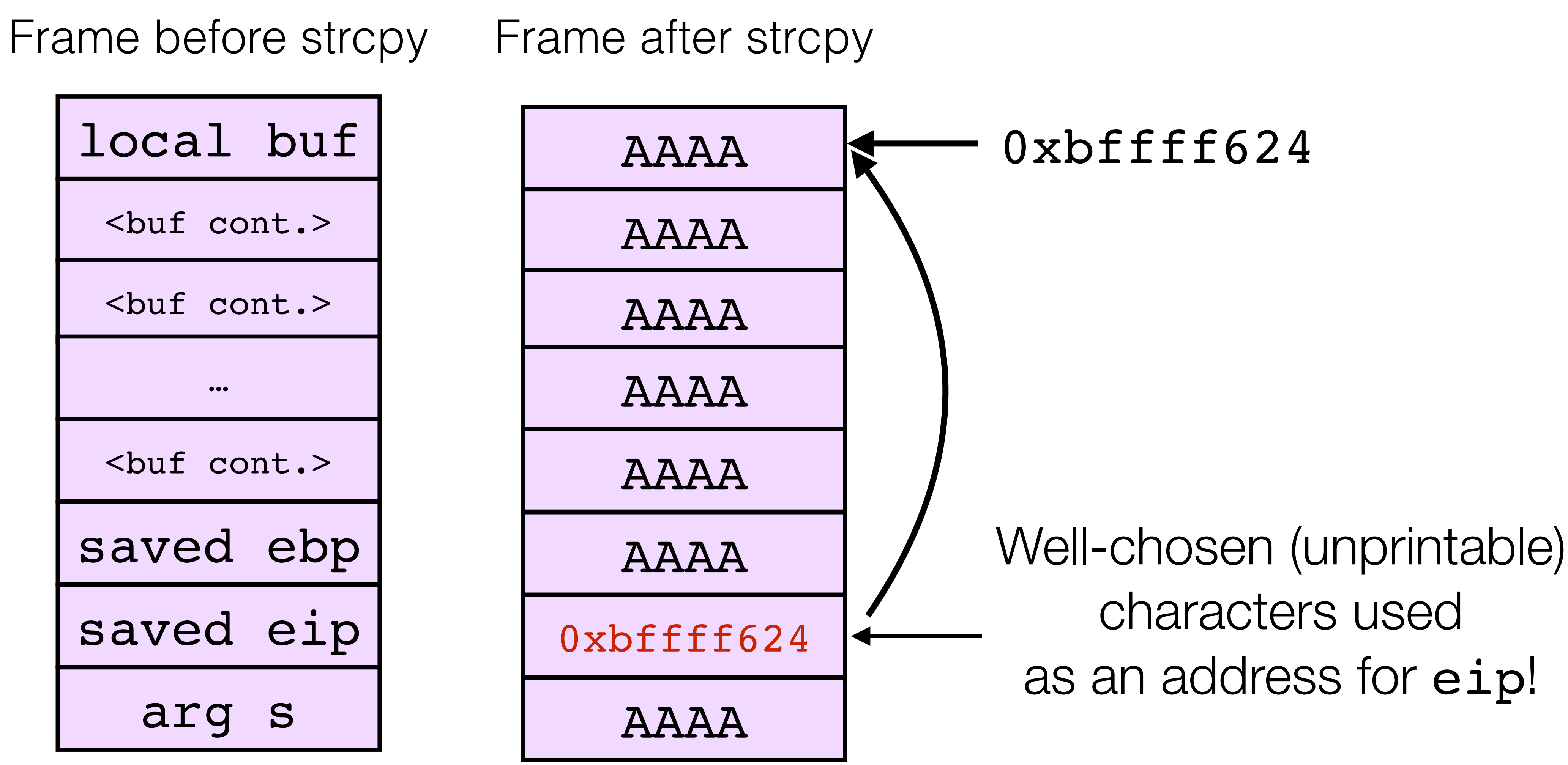
How to exploit a stack buffer overflow

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

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

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

s=“AAAAA…AAAA\x24\xf6\xff\xbfAAA…”



What will happen? Illegal instruction!

How to exploit a stack buffer overflow

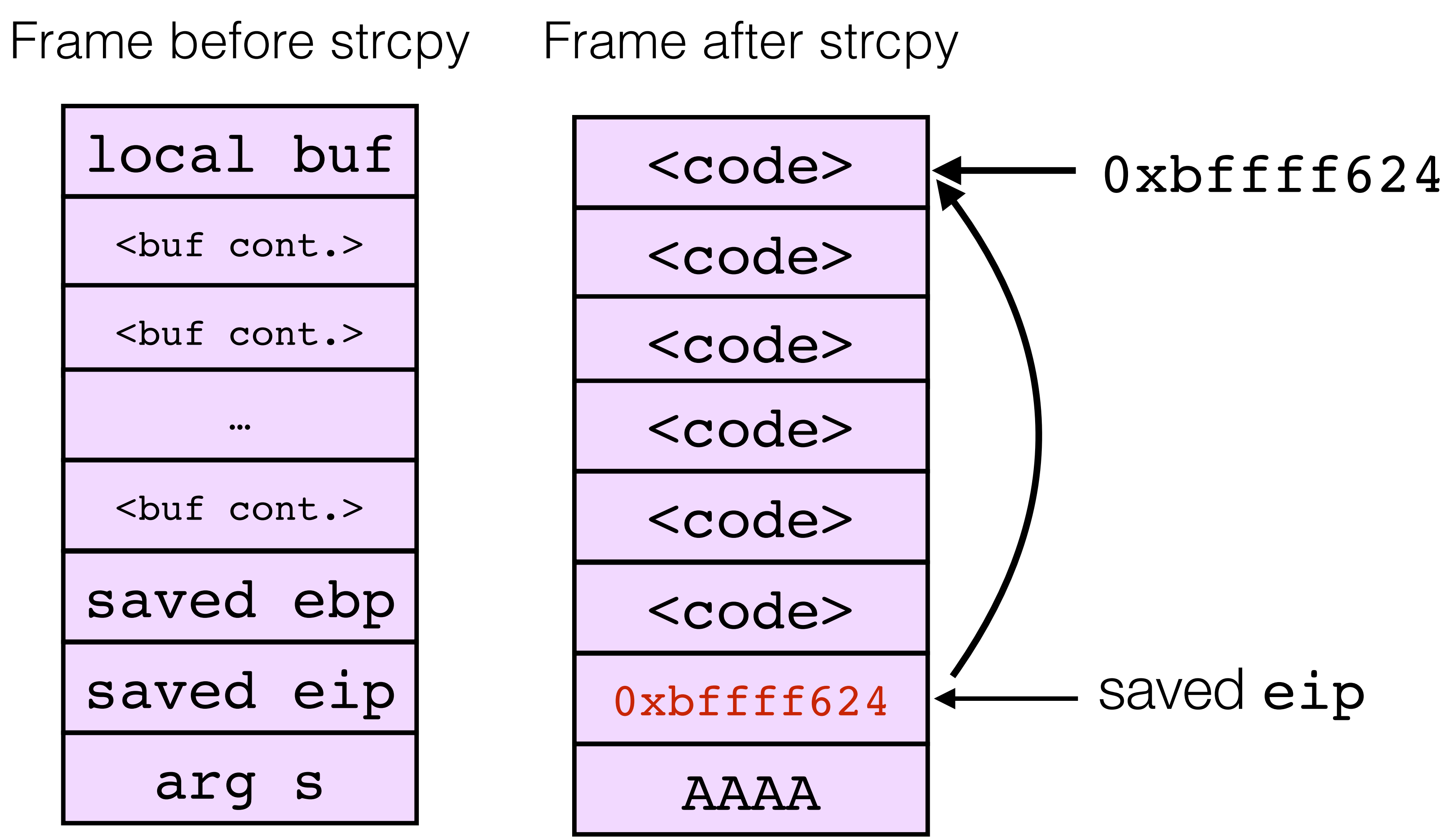
Suppose attacker can cause bad to run with an **s** it chooses.

- 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><u>\x24\xf6\xff\xbf</u>AAA…”



What will happen?

What to put in for <code>?

The possibilities are endless!

— Spawn a shell

— Spawn a new service listening to network

— Overview files

— …

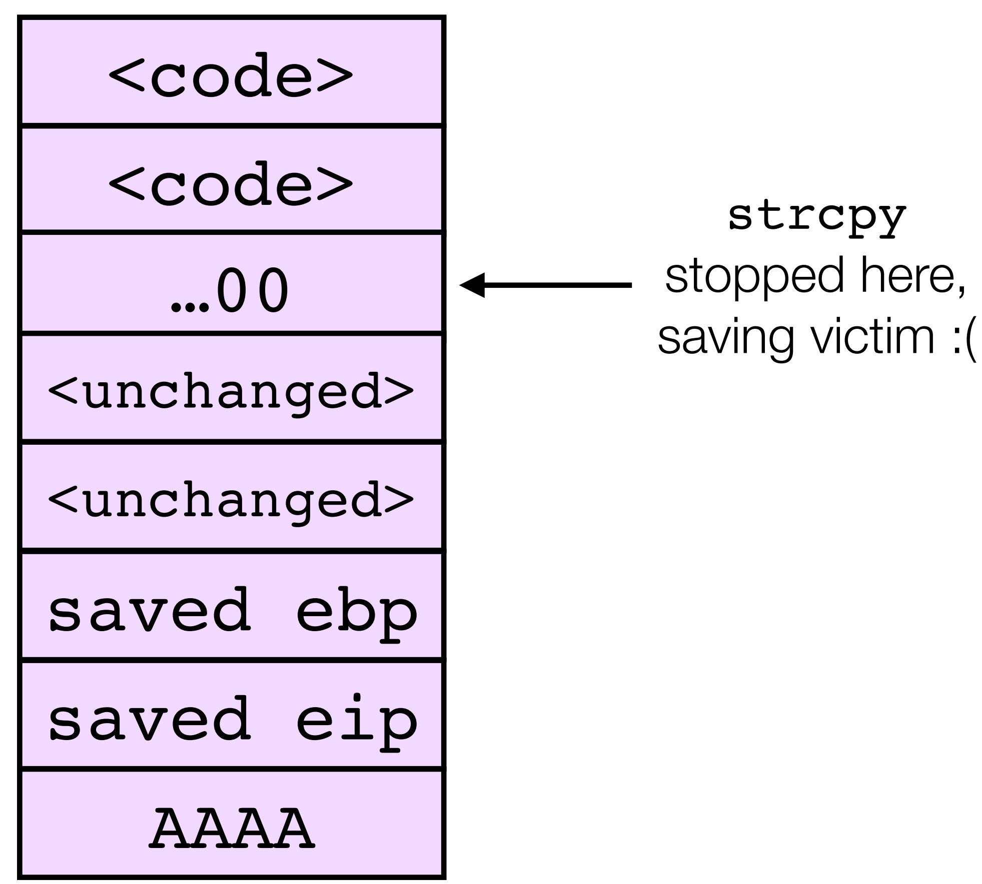
`s="<machine code>\x24\xf6\xff\xbfAAA…"`

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



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\x40\xcd"
"\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

Basically equivalent to:

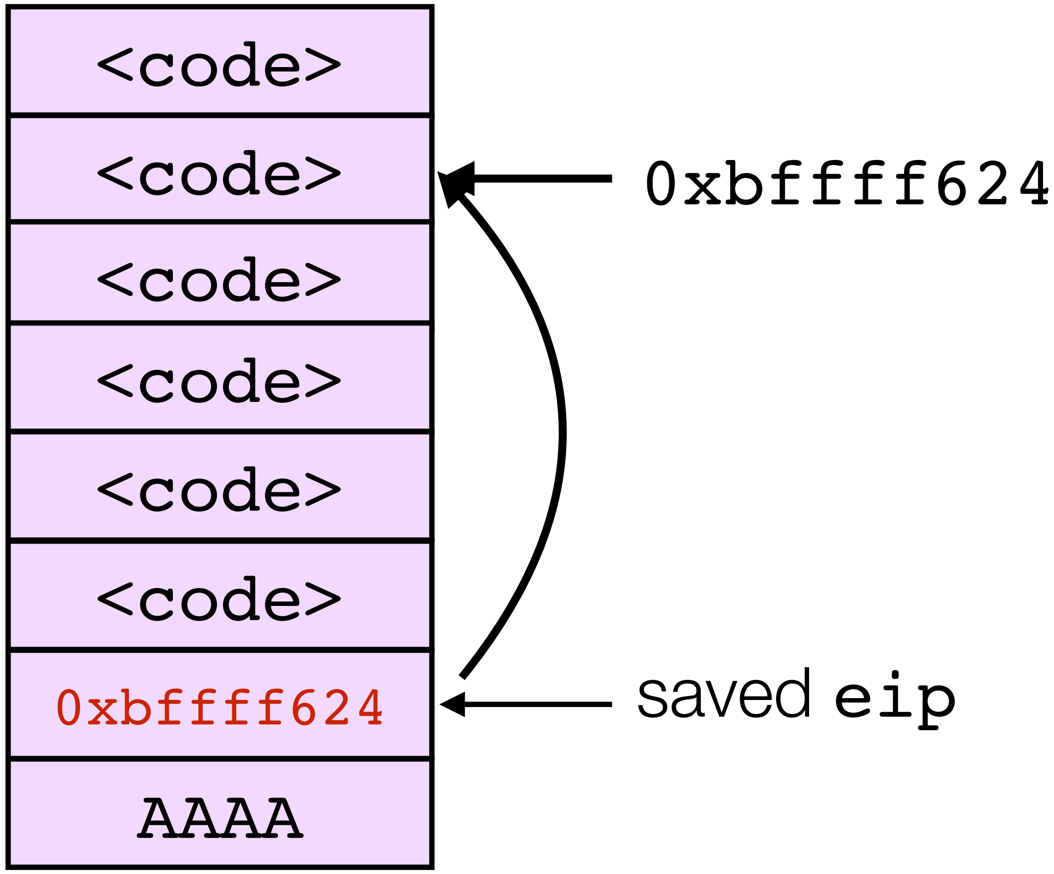
```
#include <stdio.h>
void main() {
  char *name[2];
  name[0] = “/bin/sh";
  name[1] = NULL;
  execve(name[0], name, NULL);
}
```

Finally, where did that magic address come from?

Two issues:

— Need to place address in correct spot

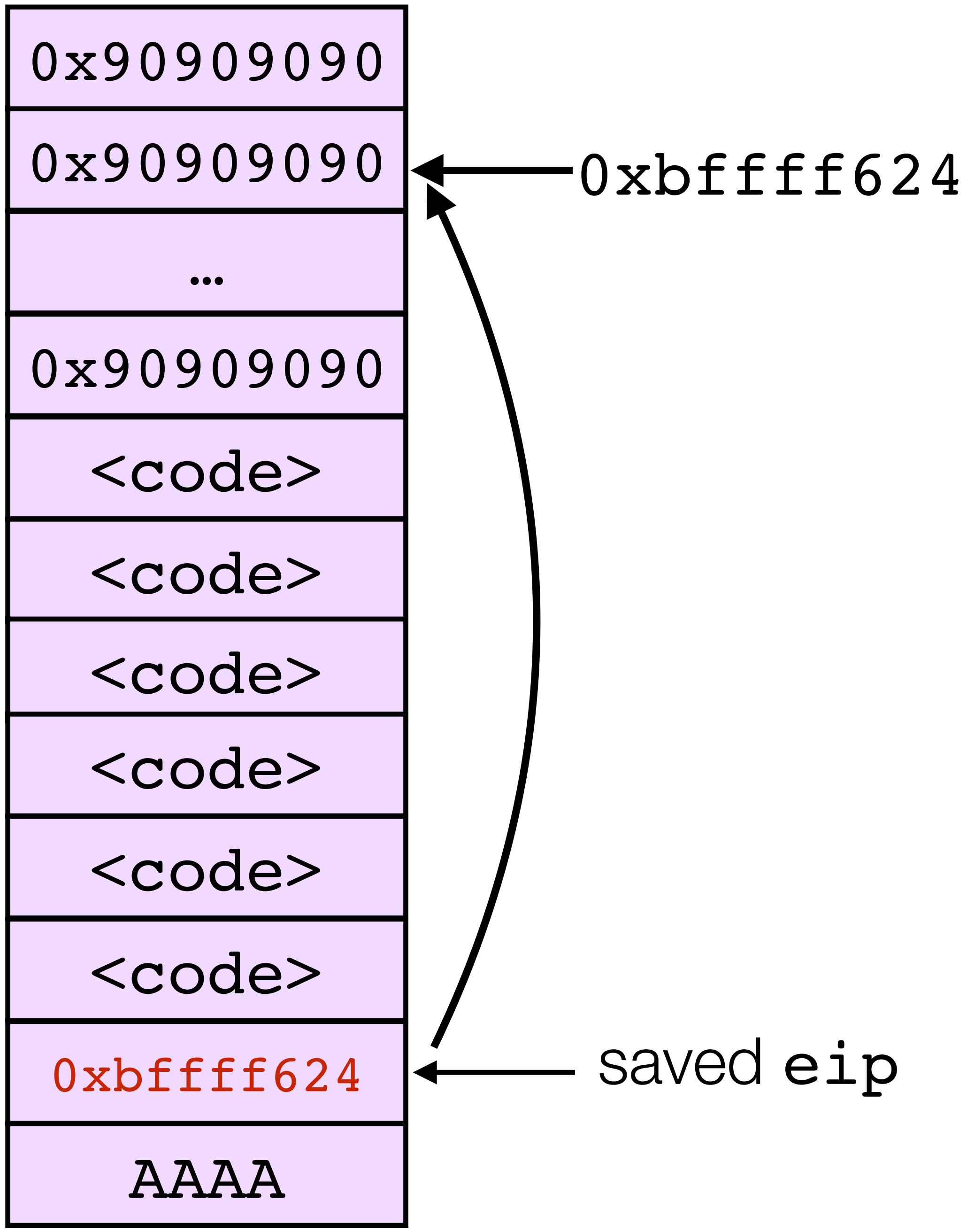
— Need address to jump to beginning of shellcode



Technique #1: NOP Sleds

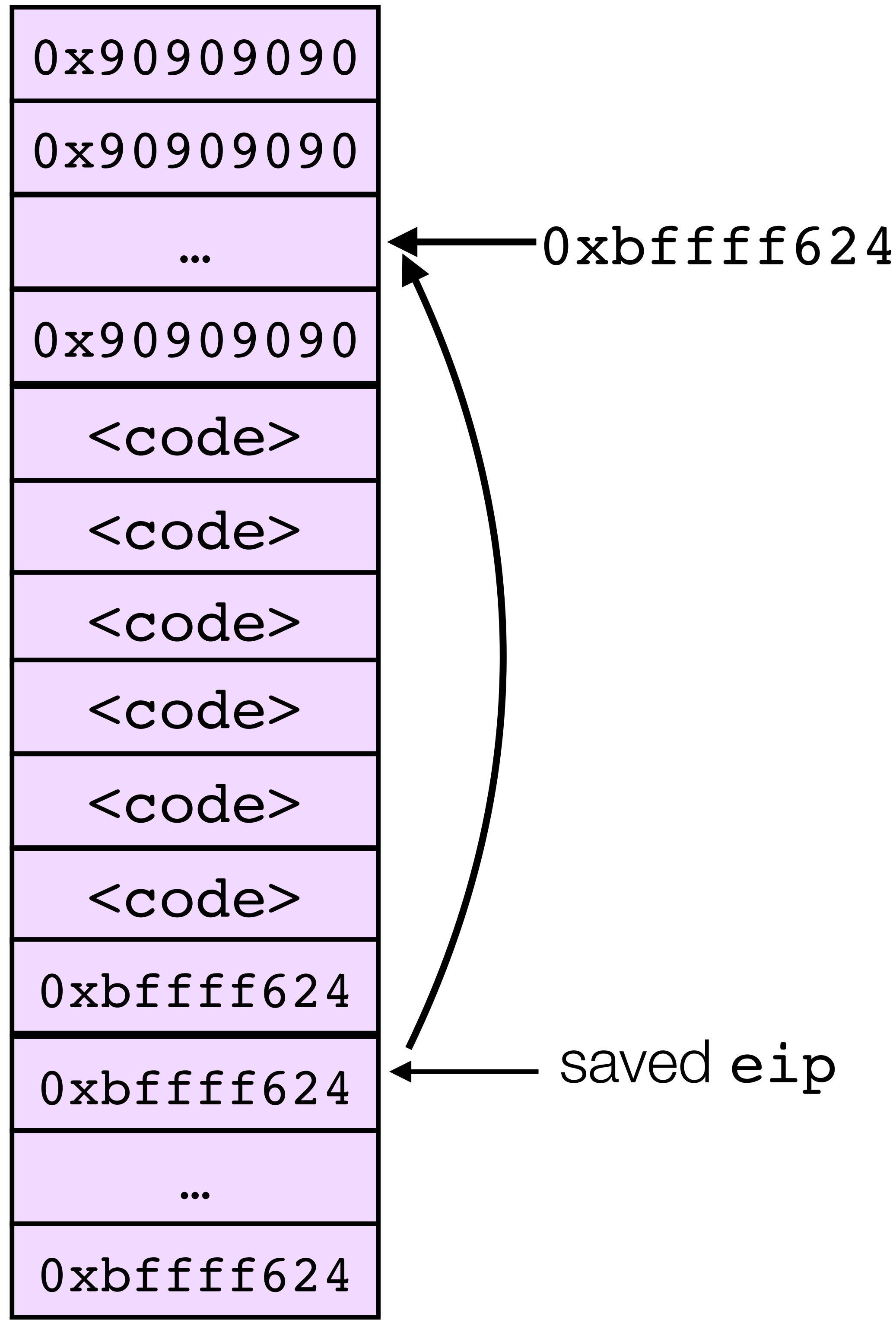
— Instruction `0x90` is "`xchg eax, eax`", i.e. does not thing. This is a "No Op" or "NOP".

— 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



The End