CS 88: Security and Privacy

02: Buffer Overflows

09-06-2022

SWARTHMORE COLLEGE

Announcements

- Clickers available through the bookstore and TAP
- Lab 0 due today
- Reading quizzes count from this week
- Midterm dates on edstem later today

Reading Quiz

Today

- Design Principles of Security
- Software Vulnerabilities
- Recap functions and the stack
- Recap assembly instructions
- Stack Buffer Overflow

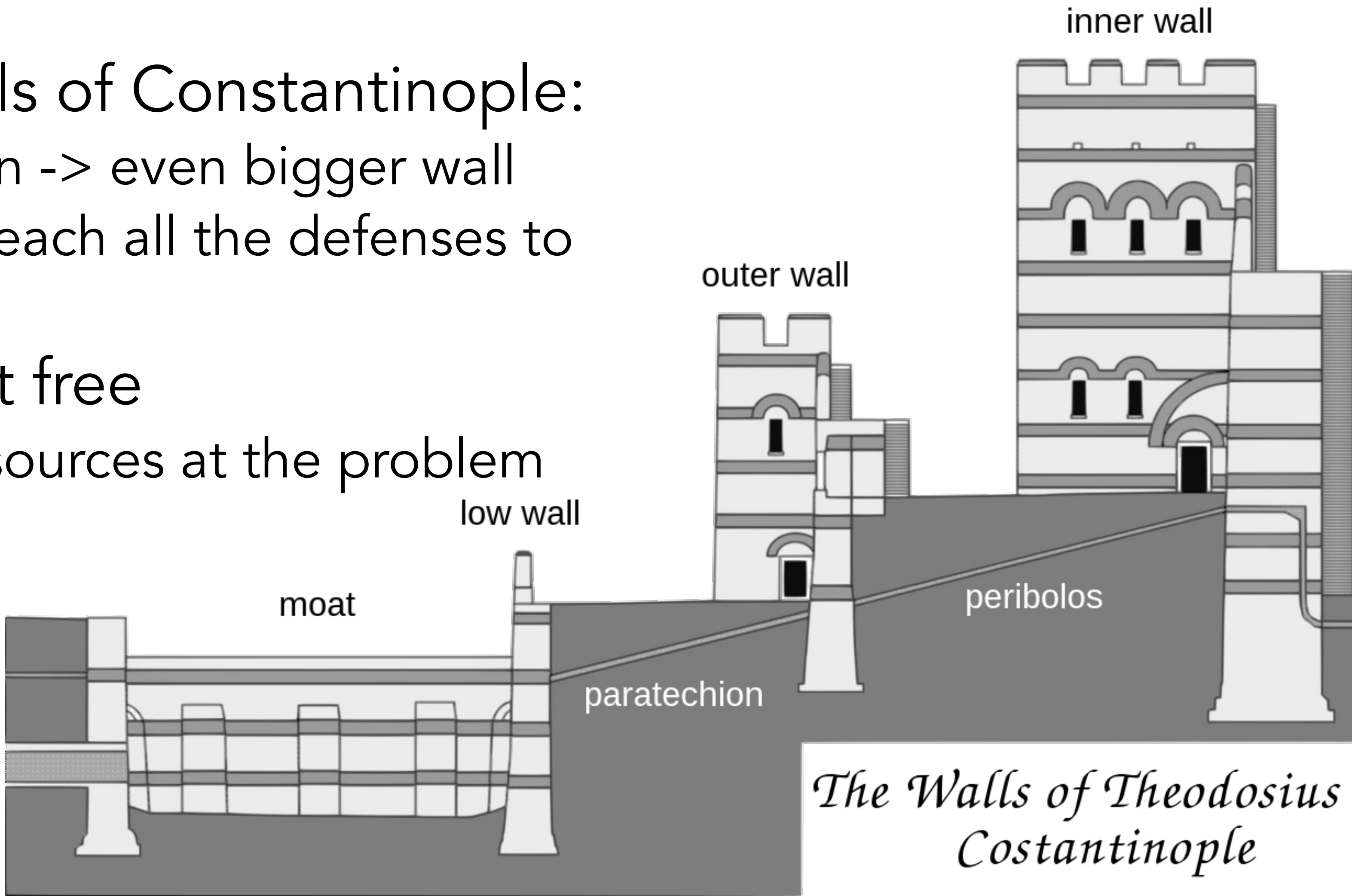
Last Class: Design Principles of Security

- Least Privilege
- Use Fail-Safe Defaults
- Separation of Privilege/Separation of responsibility
- Defense in Depth
- Complete Mediation: check access to every object
- Security *not* through obscurity
- Design Security as a core principal
- Keep it simple silly
- Ease of use

-Saltzer, J. "Protection and the Control of Information Sharing in MULTICS", CACM - 1974

Defense in Depth

- The notion of layering multiple types of protection together
- e.g., the Theodosian Walls of Constantinople:
  - Moat -> wall -> depression -> even bigger wall
  - Idea: attacker needs to breach all the defenses to gain access
- But defense in depth isn't free
  - You are throwing more resources at the problem



The Walls of Theodosius Costantinople

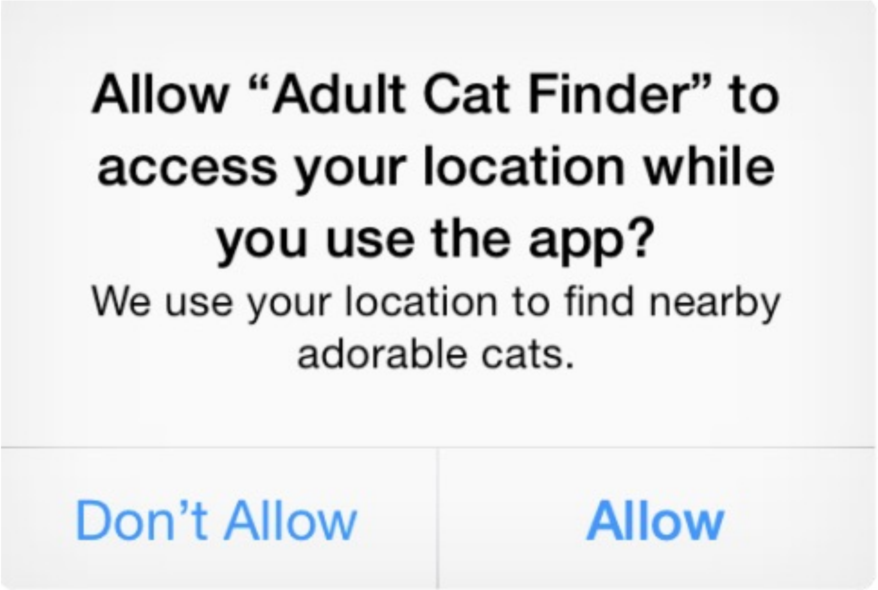
Password authentication

- People have a hard time remembering multiple strong passwords, so they reuse them on multiple sites
- Consequence: security breach of one site causes account compromise on other sites
- Solution: password manager
  - Remember one strong password, which unlocks access to site passwords
- Solution: two-factor authentication
  - Need both correct password and separate device to access account
- *Free advice: to protect yourself, use a password manager and two-factor authentication*

Least Privilege

- *Every program and every user of the system should operate using the least set of privileges necessary to complete the job*
- A subject should be given only those privileges necessary to complete its task
  - Function, not identity, controls
  - Rights added as needed, discarded after use
  - Minimal protection domain

Does this follow the principle of least privilege?



Allow "Adult Cat Finder" to access your location while you use the app?

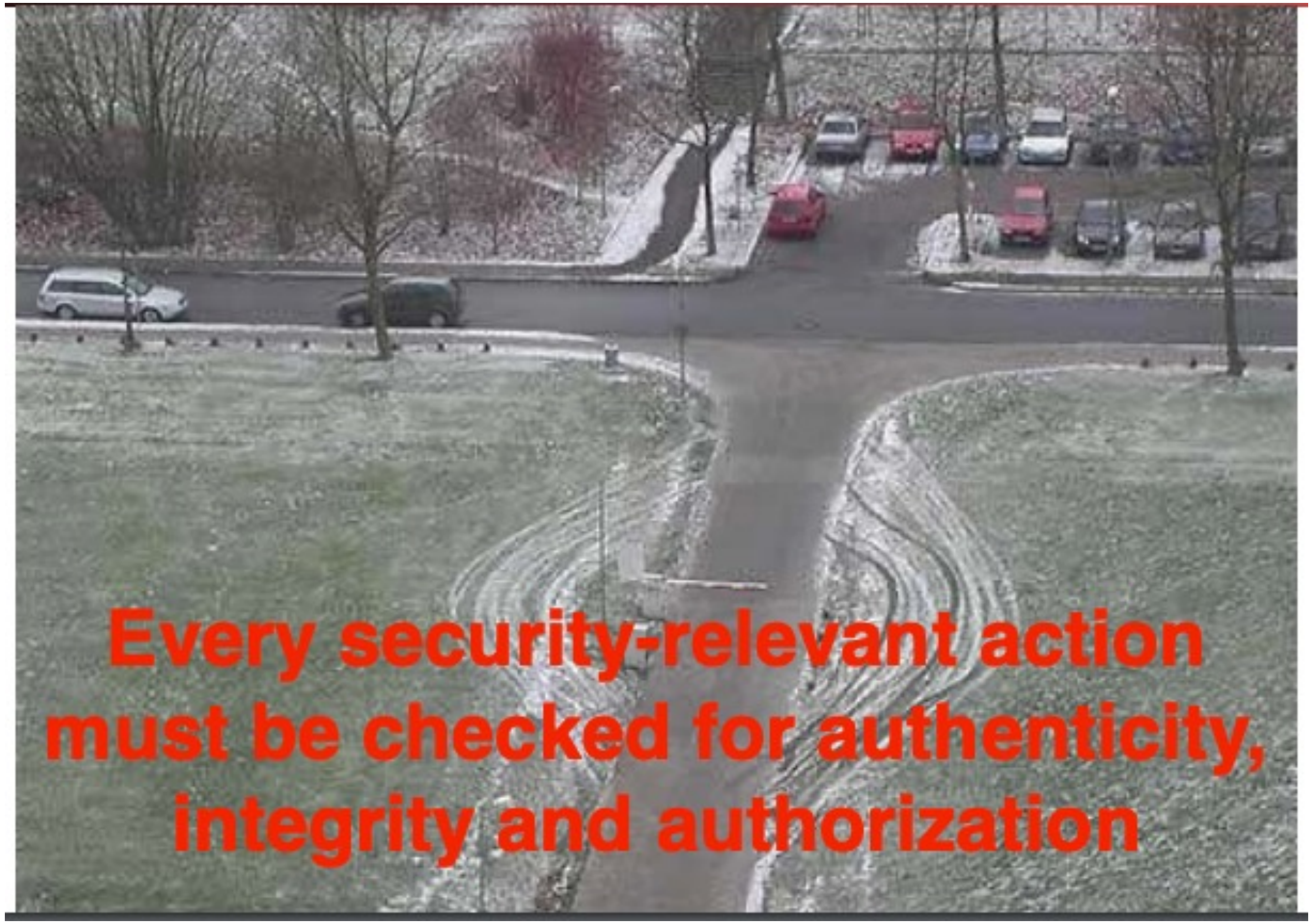
We use your location to find nearby adorable cats.

A. Yes
B. No
C. Maybe (Be prepared to explain)

Ensuring Complete Mediation

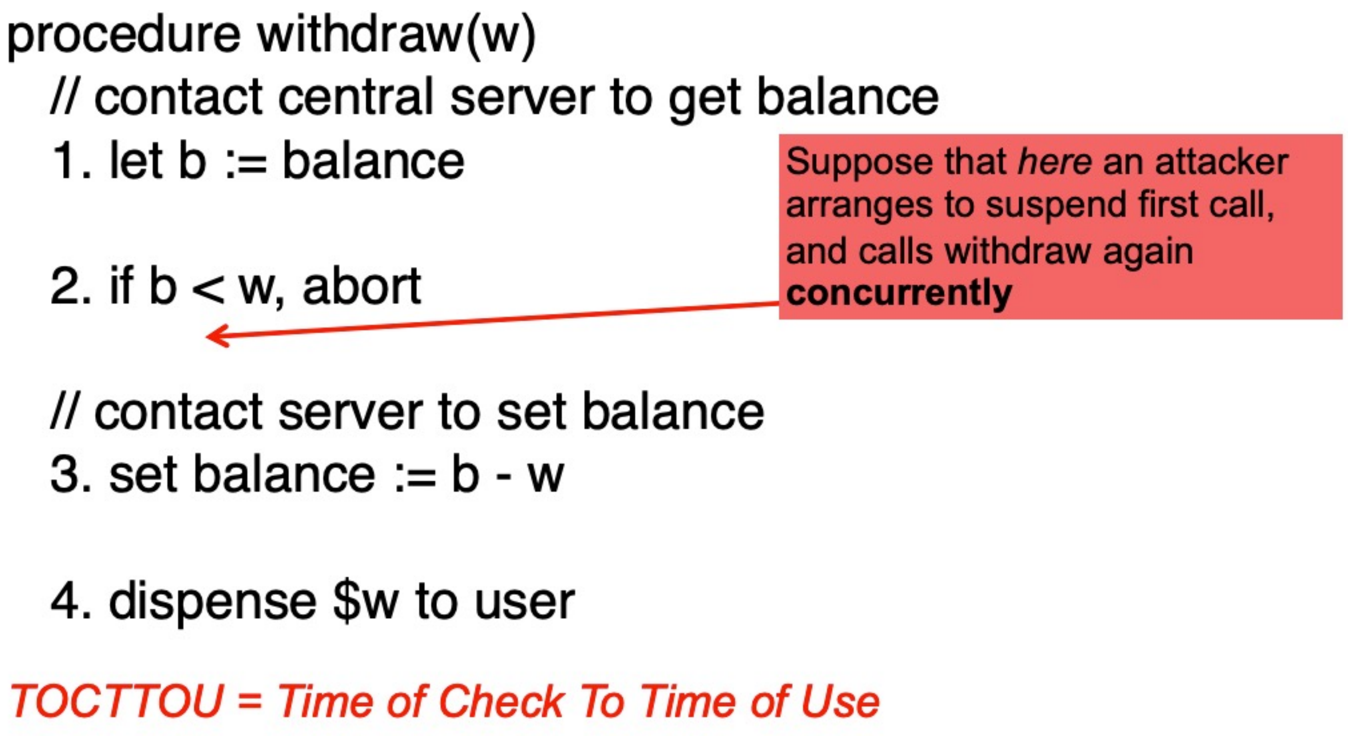
- To secure access to some capability/resource, construct a reference monitor
  - Single point through which all access must occur
    - E.g.: a network firewall
    - Desired properties: • Un-bypassable (“complete mediation”) •
    - Tamper-proof (is itself secure)
    - Verifiable (correct)
  - One subtle form of reference monitor flaw concerns race conditions

A Failure of Complete Mediation



Every security-relevant action
must be checked for authenticity,
integrity and authorization

Time of Check to Time of Use Vulnerability: Race Condition



```
procedure withdraw(w)
  // contact central server to get balance
  1. let b := balance

  2. if b < w, abort

  // contact server to set balance
  3. set balance := b - w

  4. dispense $w to user
```

Suppose that *here* an attacker arranges to suspend first call, and calls withdraw again **concurrently**

TOCTTOU = Time of Check To Time of Use

Security Reviews

- Least Privilege
- Use Fail-Safe Defaults
- Separation of Privilege/Separation of responsibility
- Defense in Depth
- Complete Mediation: check access to every object
- Security *not* through obscurity
- Design Security as a core principal
- Keep it simple silly
- Ease of use

Software Security

When is a program secure?

A. When it does what we want it to do

B. When we ensure that bad inputs do not result in unintended functionality

C. We need B + some more safeguards (be prepared to explain)

D. We can never have a secure program

Software Security

- Secure design and implementation
- Popular approach to software: black box approach
- Build defenses around vulnerable software – easily circumvented

When is a program secure?

- Formal approach: When it does exactly what it should
  - not more
  - not less
- But how do we know what it is supposed to do?

When is a program secure?

- Formal approach: When it does exactly what it should
  - not more
  - not less
- But how do we know what it is supposed to do?
  - somebody tells us (do we trust them?)
  - we write the code ourselves (what fraction of s/w have you written?)

When is a program secure?

- Pragmatic approach: when it doesn’t do bad things
- Often easier to specify a list of “bad” things:
    - delete or corrupt important files (integrity)
    - crash my system (availability)
    - send my password over the internet (confidentiality)
    - send phishing email

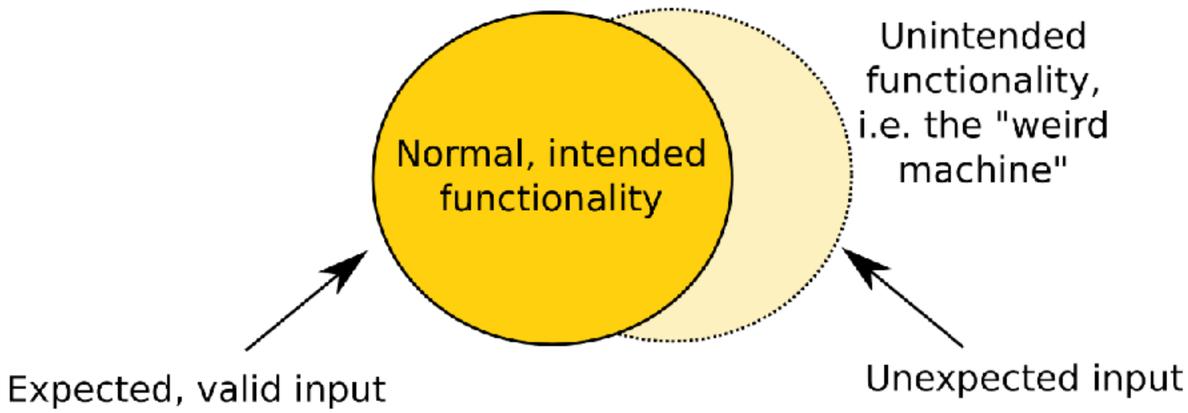
When is a program secure?

- But .. what if the program doesn’t do bad things, but could?

- is it secure?

Weird machines

- complex systems contain unintended functionality


Unintended functionality, i.e. the "weird machine"
Normal, intended functionality
Expected, valid input
Unexpected input

- attackers can trigger this unintended functionality
  - i.e. they are exploiting vulnerabilities

What is a software vulnerability?

- A bug in a program that allows an unprivileged user capabilities that should be denied to them.
- There are a lot of types of vulnerabilities
  - bugs that violate "control flow integrity"
  - why? lets attacker run code on your computer!
- Typically these involve violating assumptions of the programming language or its run-time

Exploiting vulnerabilities (the start)

- Dive into low level details of how exploits work
  - How can a remote attacker get a victim program to execute their code?

- Threat model: victim code is handling input that comes from across a security boundary
  - what are examples of this?

- Security policy: want to protect integrity of execution and confidentiality of data from being compromised by malicious and highly skilled users of our system.

Stack buffer overflows

- Understand how buffer overflow vulnerabilities can be exploited
- Identify buffer overflows and asses their impact
- Avoid introducing buffer overflow vulnerabilities
- Correctly fix buffer overflow vulnerabilities

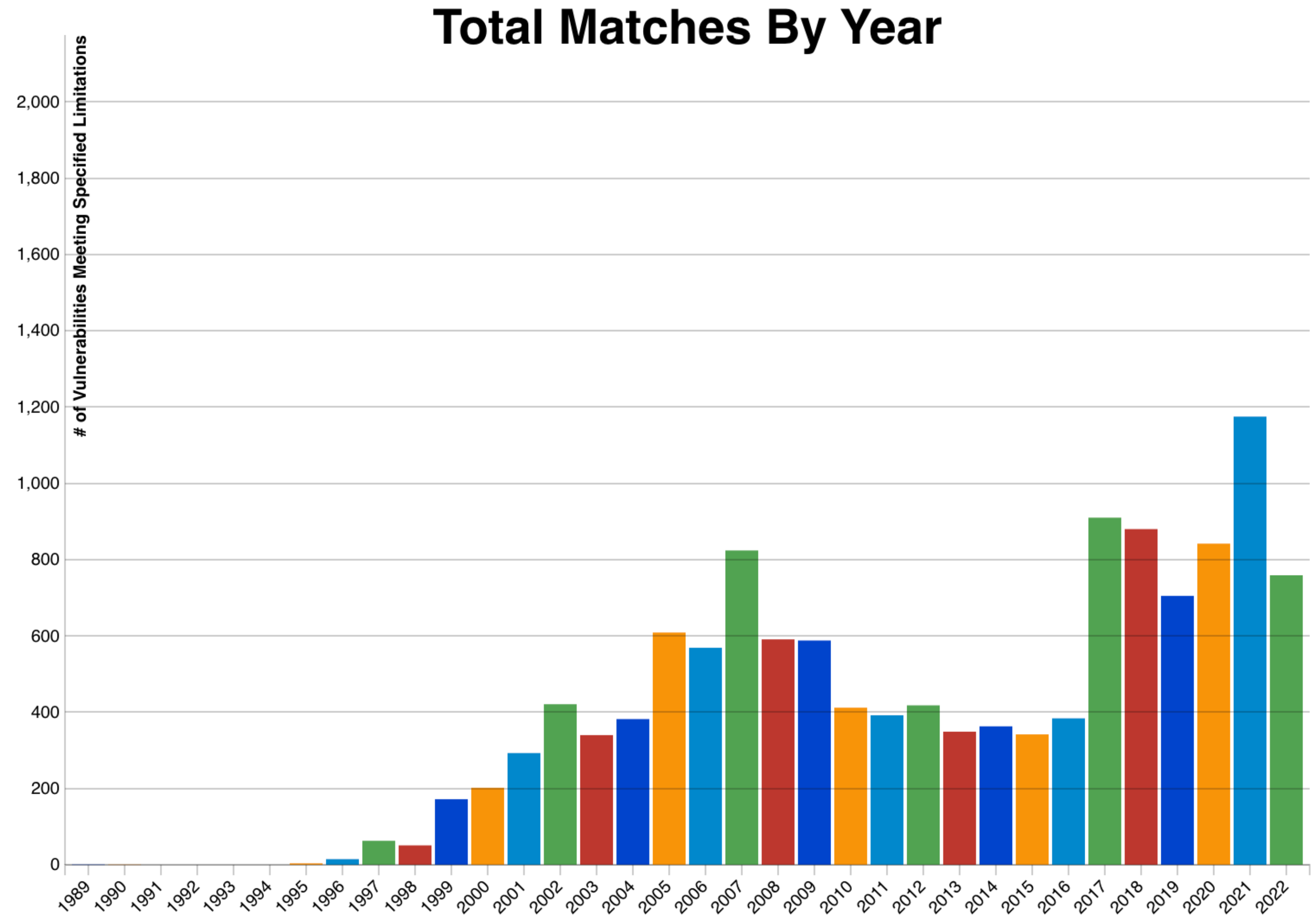
Buffer Overflows

- An anomaly that occurs when a program writes/reads data beyond the boundary of a buffer
- Canonical software vulnerability
  - ubiquitous in system software
  - OSes, web servers, web browsers
- If your program crashes with memory faults, you probably have a buffer overflow vulnerability

Search Parameters:

- Results Type: Statistics
- Keyword (text search): buffer overflow
- Search Type: Search All
- CPE Name Search: false

Common Vulnerabilities and Exposures (CVE): security flaw that is publicly known



Critical Systems are written in C/C++

- OS kernels
- High-performance servers
  - Apache, MySQL
- Embedded Systems
  - IoT deivices, “smart” vehicles, the MARs rover..

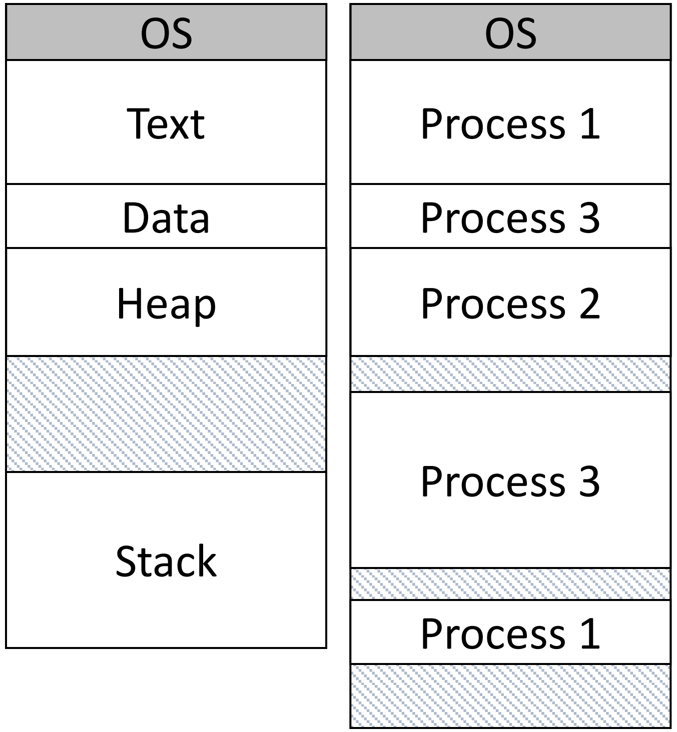
https://nvd.nist.gov/vuln/search

CS 31 Recap

Memory

- Abstraction goal: make every process think it has the same memory layout.
  - MUCH simpler for compiler if the stack always starts at 0xFFFFFFFF, etc.

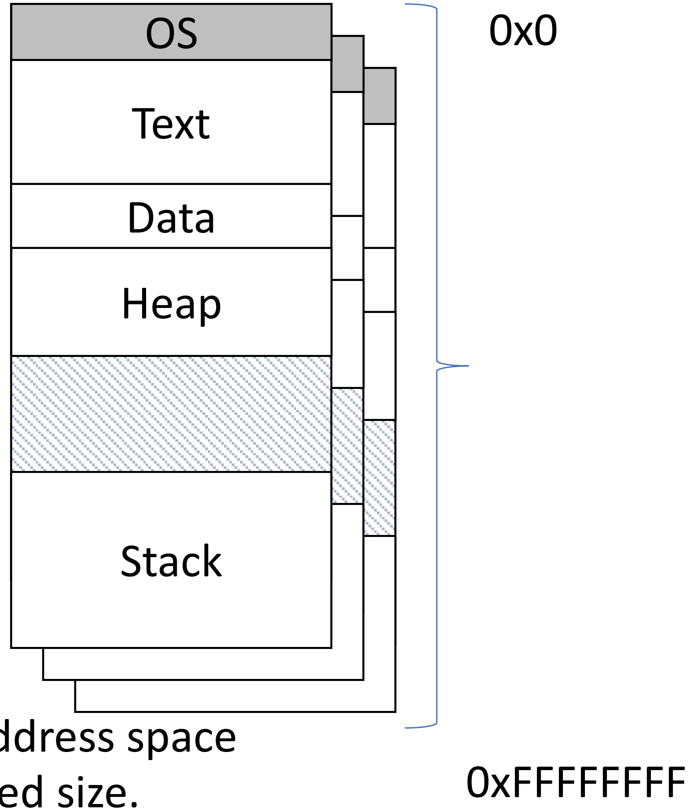
- Reality: there's only so much memory to go around, and no two processes should use the same (physical) memory addresses.



OS (with help from hardware) will keep track of who's using each memory region.

Memory Terminology

Virtual (logical) Memory: The abstract view of memory given to processes. Each process gets an independent view of the memory.

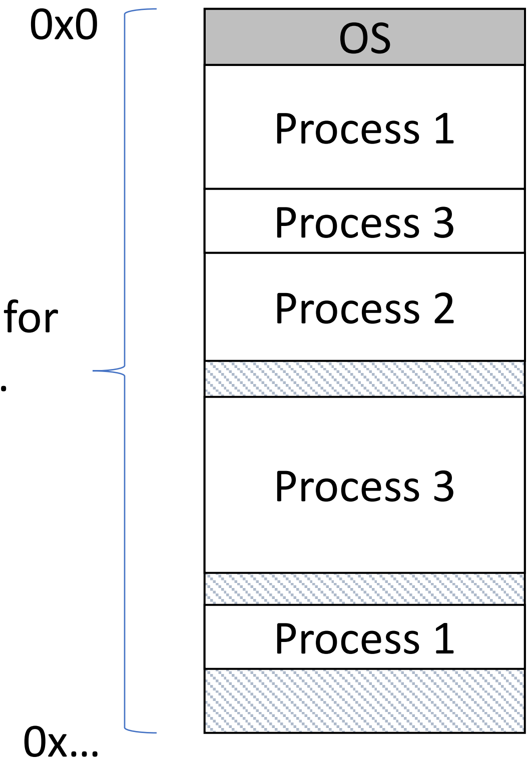


Virtual address space (VAS): fixed size.

Physical Memory: The contents of the hardware (RAM) memory. Managed by OS. Only ONE of these for the entire machine!

Address Space: Range of addresses for a region of memory.

The set of available storage locations.



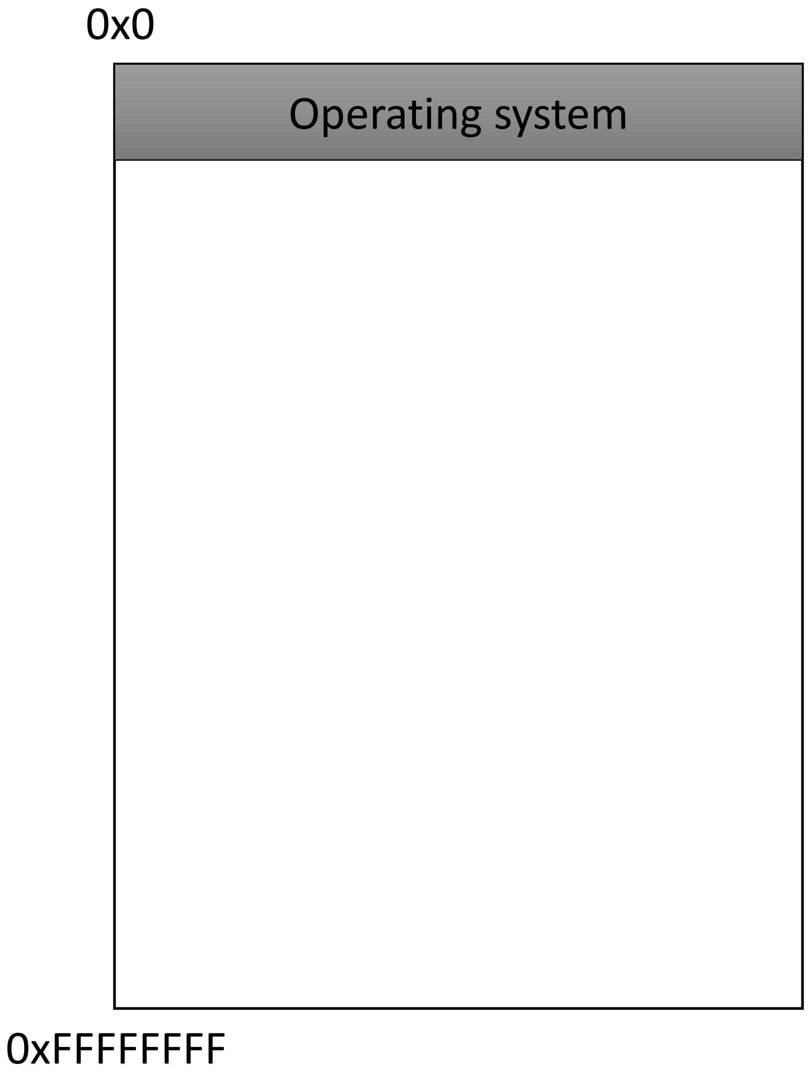
(Determined by amount of installed RAM.)

Memory

- Behaves like a big array of bytes, each with an address (bucket #).

- By convention, we divide it into regions.

- The region at the lowest addresses is usually reserved for the OS.



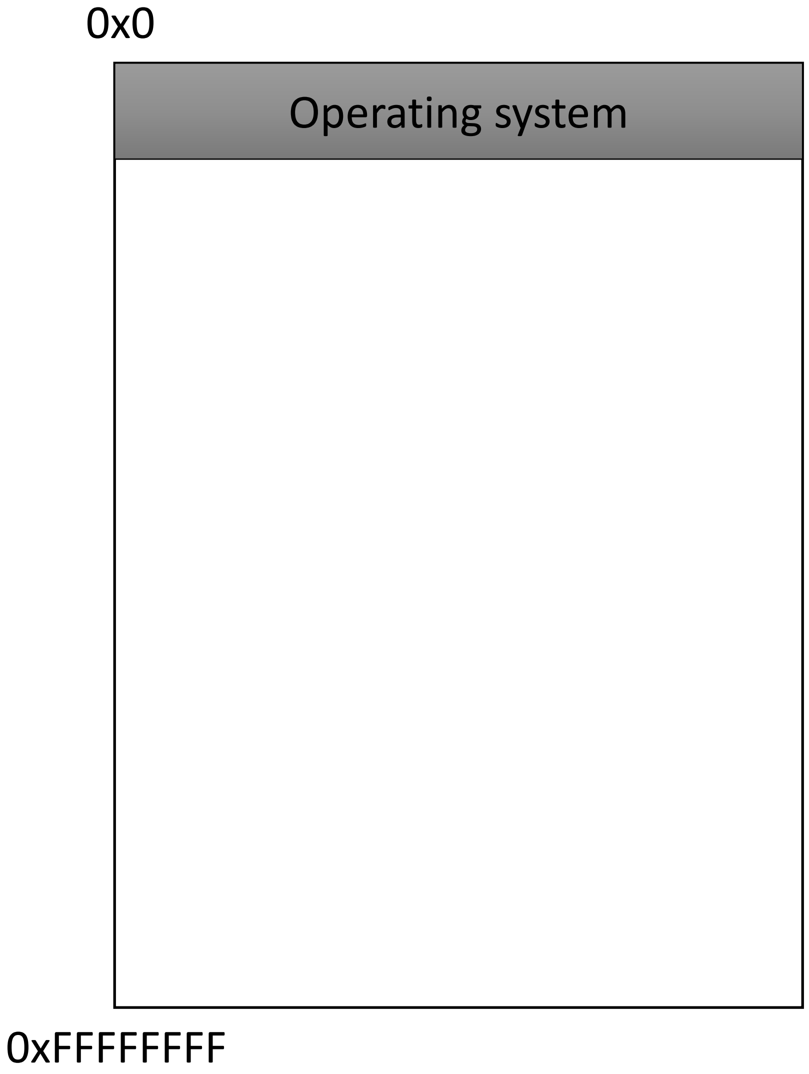
0x0

Operating system

0xFFFFFFFF

NULL: A special pointer value.

- NULL is equivalent to pointing at memory address 0x0. This address is NEVER in a valid segment of your program’s memory.
  - This guarantees a segfault if you try to deref it.
  - Generally a good ideal to initialize pointers to NULL.

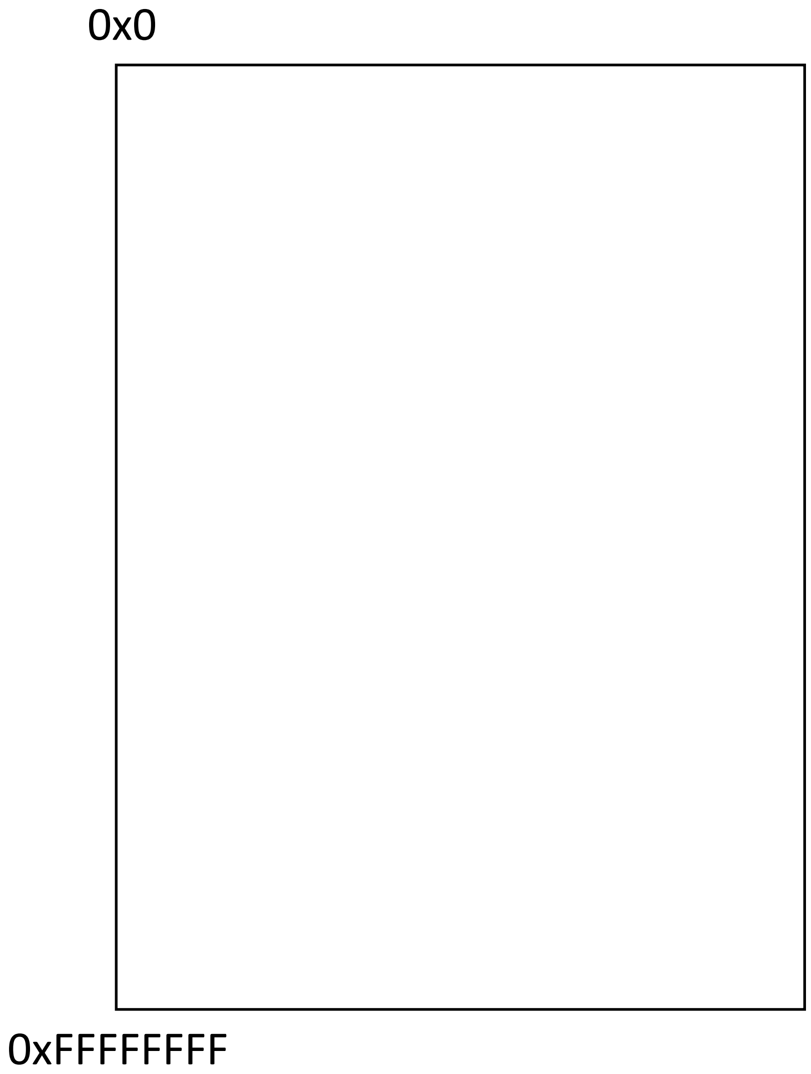


What happens if we launch an attack where we load an instruction to execute at 0x0

A. Nothing will happen, this region is mapped to the NULL pointer, which does not have any effect

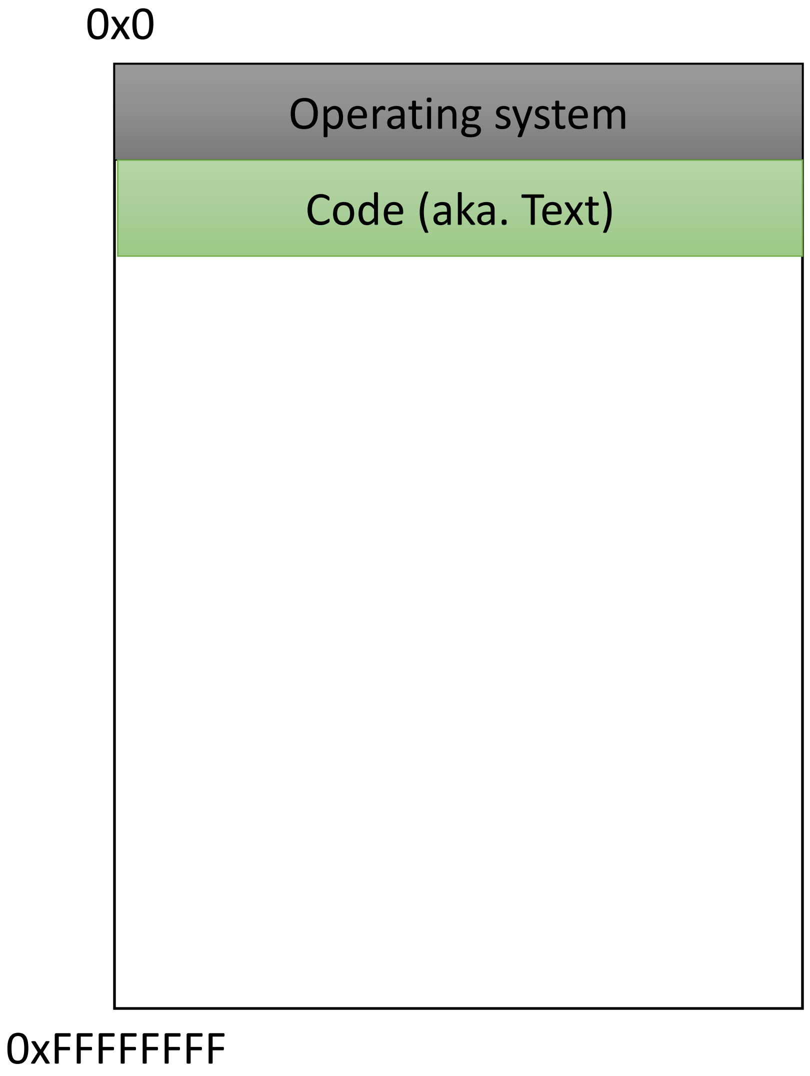
B. There will be some effect, but not necessarily devastating

C. This will have a devastating effect.



Memory - Text

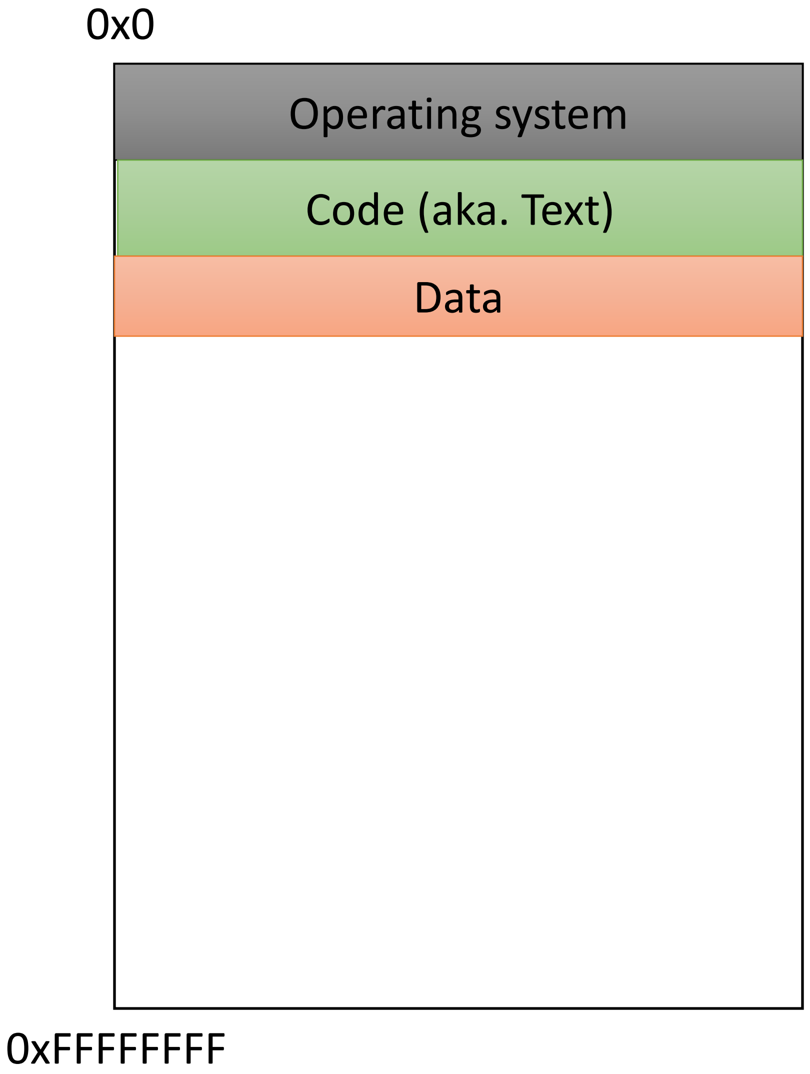
- After the OS, we store the program’s code.
- Instructions generated by the compiler.



Memory – (Static) Data

- Next, there’s a fixed-size region for static data.

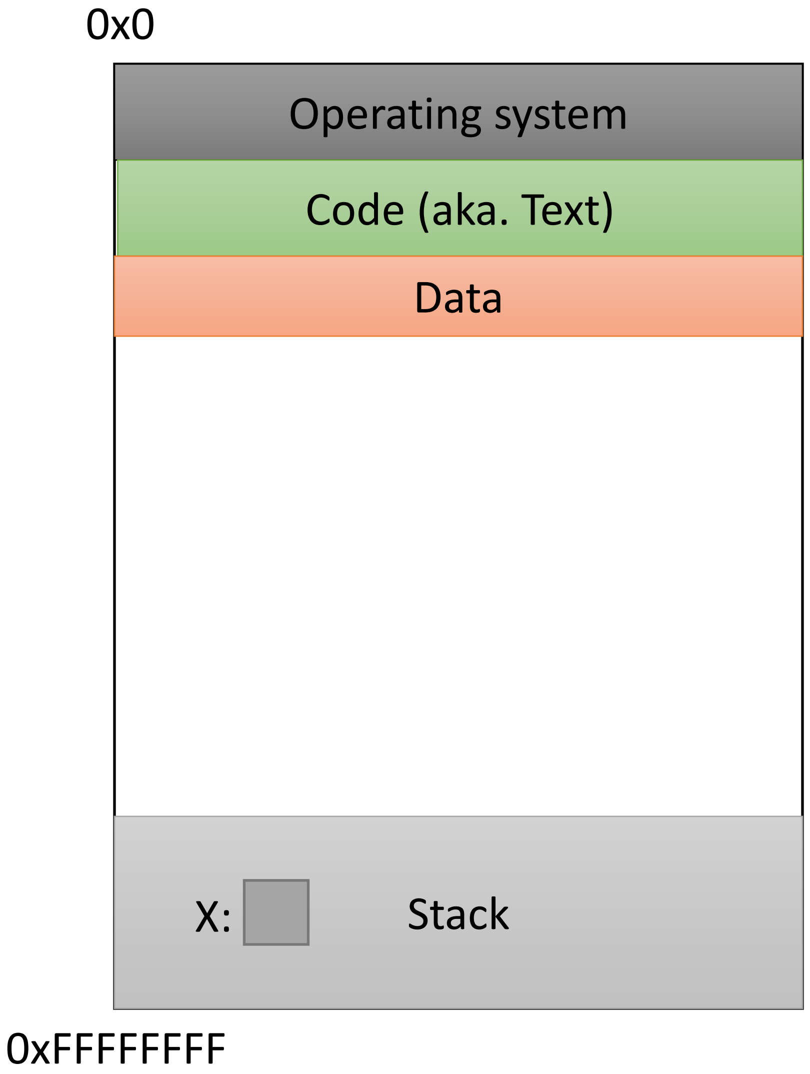
- This stores static variables that are known at compile time.
  - Global variables



Memory - Stack

- At high addresses, we keep the stack.

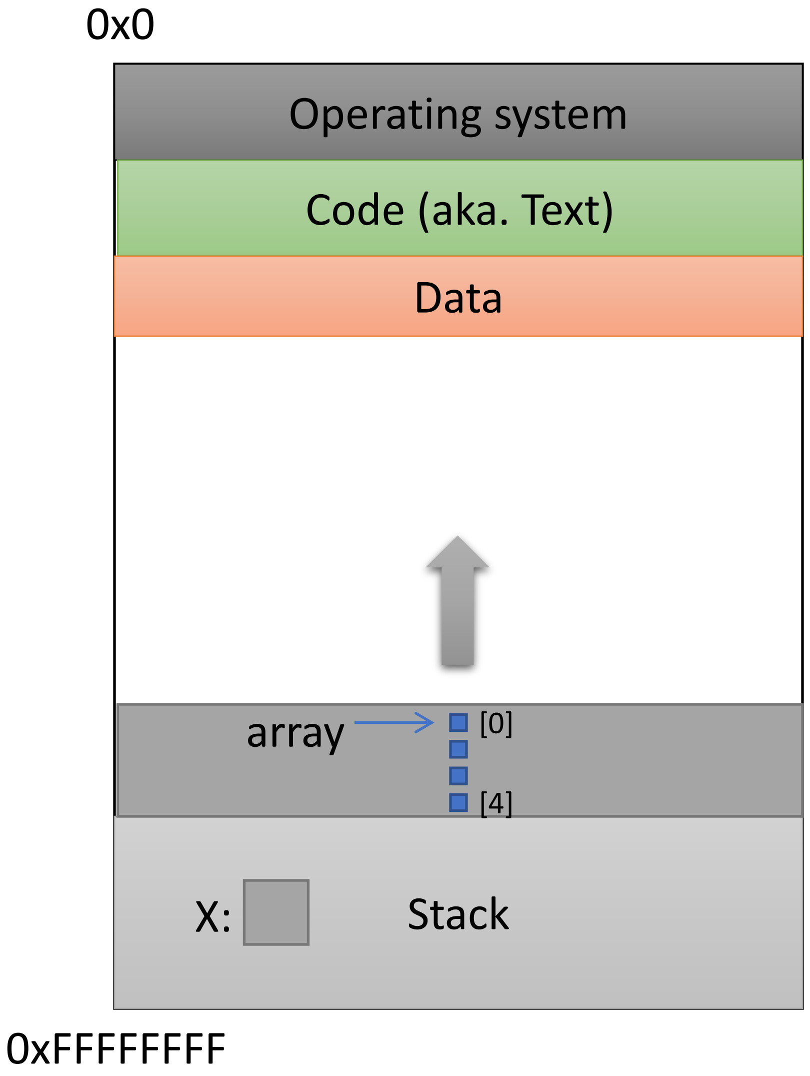
- This stores local (automatic) variables.
  - The kind we've been using in C so far.
  - e.g., int x;



Memory - Stack

- The stack grows upwards towards lower addresses (negative direction).

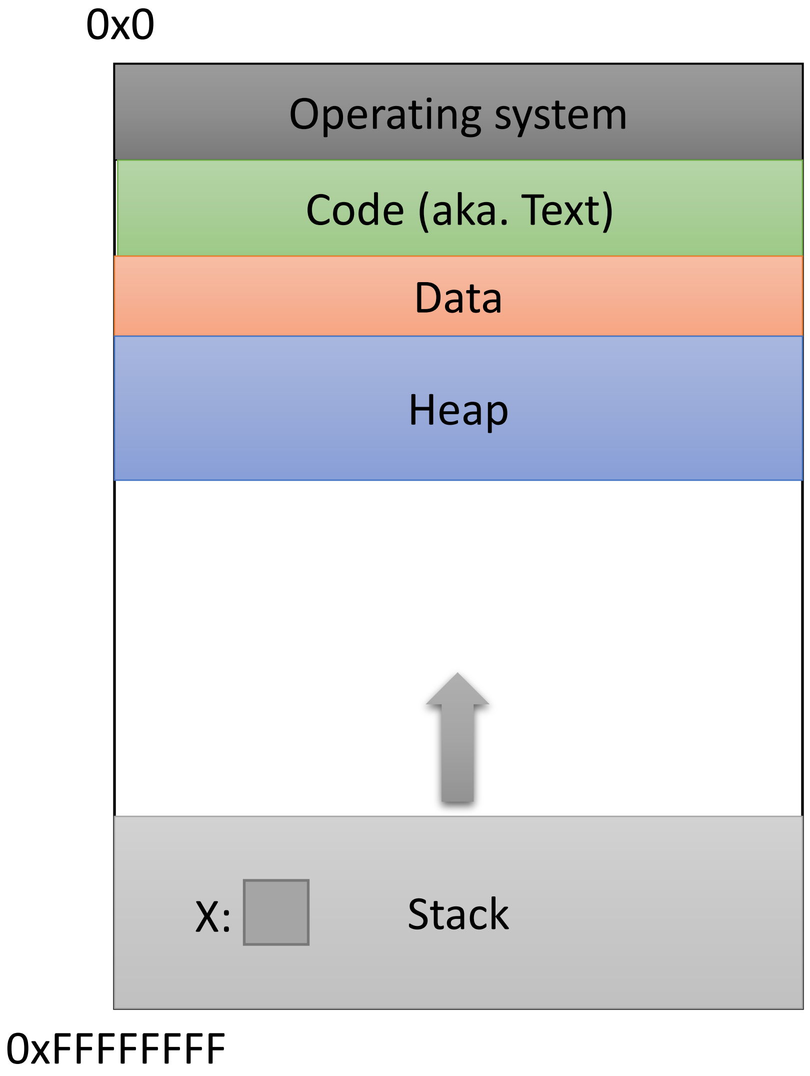
- Example: Allocating array
  - int array[4];



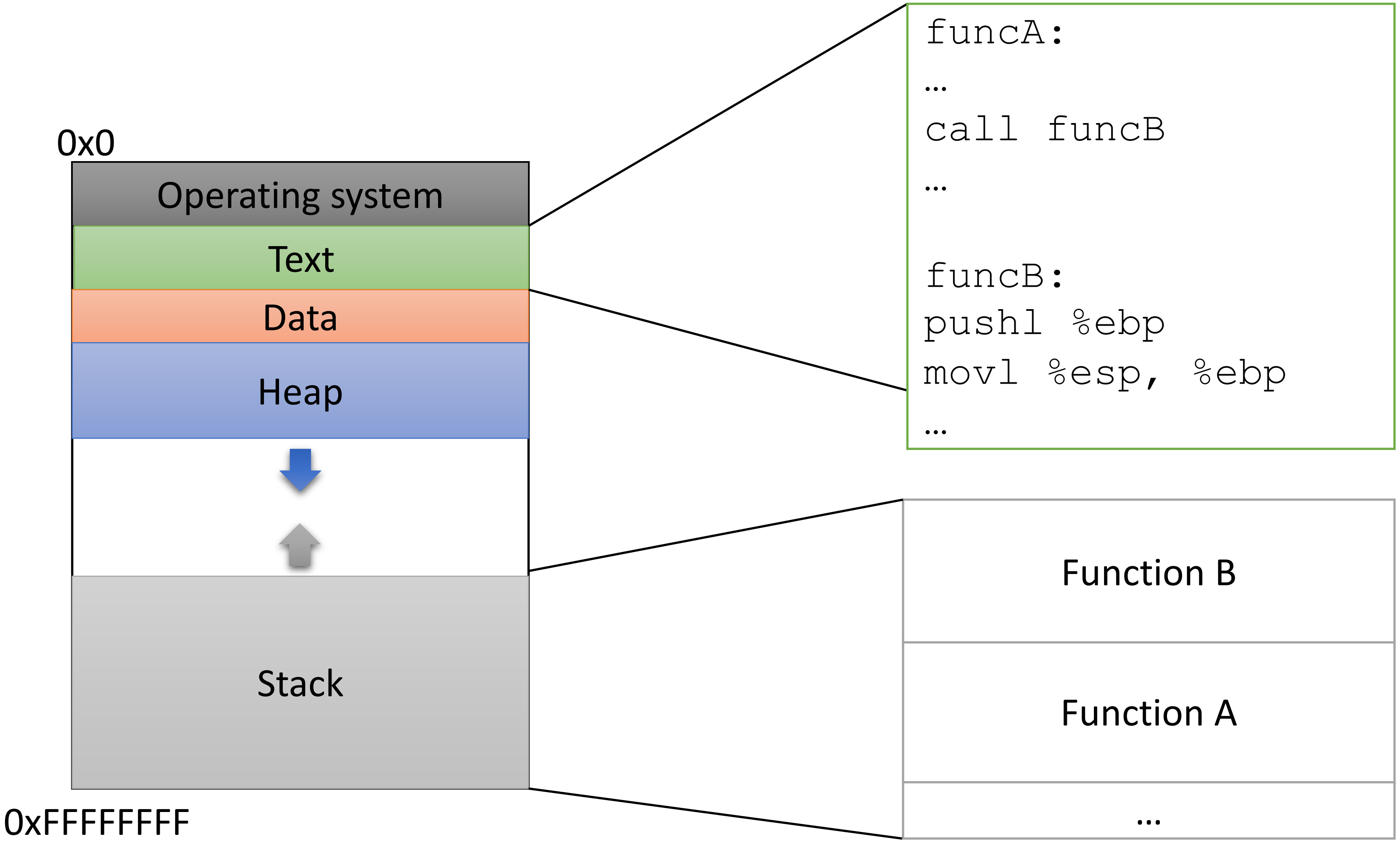
Memory - Heap

- The heap stores dynamically allocated variables.

- When programs explicitly ask the OS for memory, it comes from the heap.
  - malloc() function



Instructions in Memory



```
funcA:
…
call funcB
…

funcB:
pushl %ebp
movl %esp, %ebp
…
```

Process memory layout

.text

- Machine code of executable

.data

- Global initialized variables

.bss

- Below Stack Section

  global uninitialized vars

heap

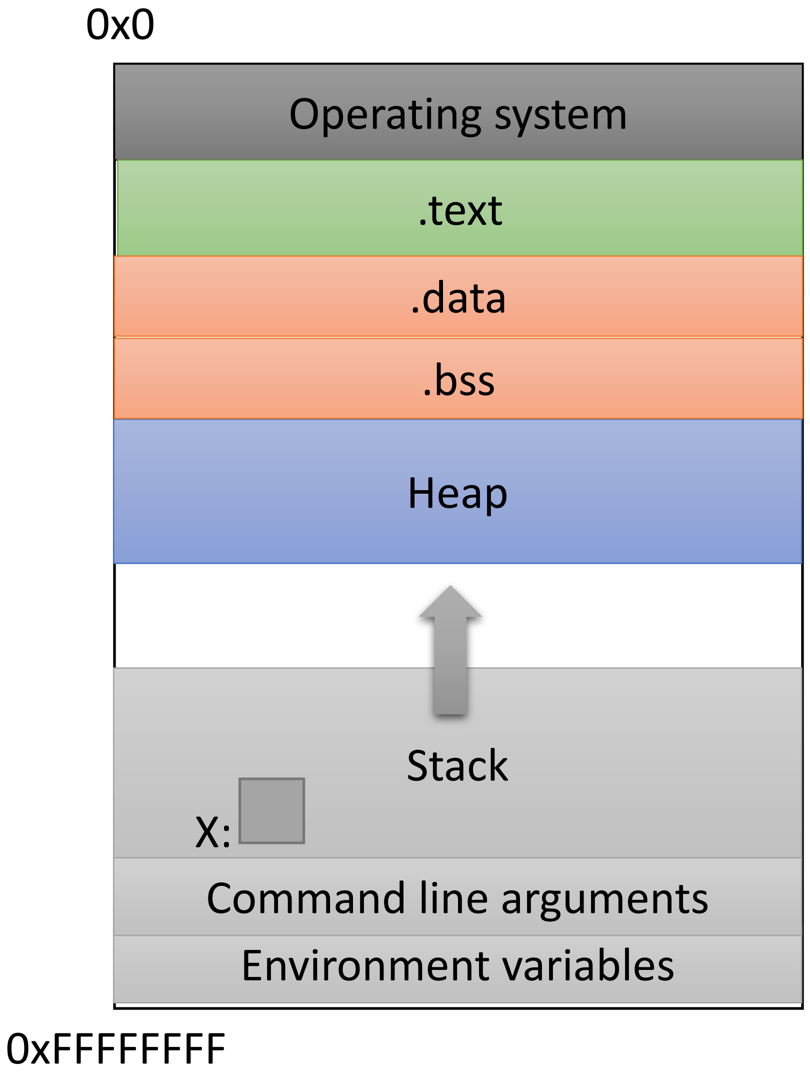
- Dynamic variables

stack

- Local variables
- Function call data

Env

- Environment variables
- Program arguments



Process memory layout

.text

- Machine code of executable

.data

- Global initialized variables

.bss

- Below Stack Section

  global uninitialized vars

heap

- Dynamic variables

stack

- Local variables
- Function call data

Env

- Environment variables
- Program arguments

```
int i = 0;
int main()
{
   char *ptr = malloc(sizeof(int));
   char buf[1024]
   int j;
   static int y;
}
```

Process memory layout

.text

- Machine code of executable

.data

- Global initialized variables

.bss

- Below Stack Section
  global uninitialized vars

heap

- Dynamic variables

stack

- Local variables
- Function call data

Env

- Environment variables
- Program arguments

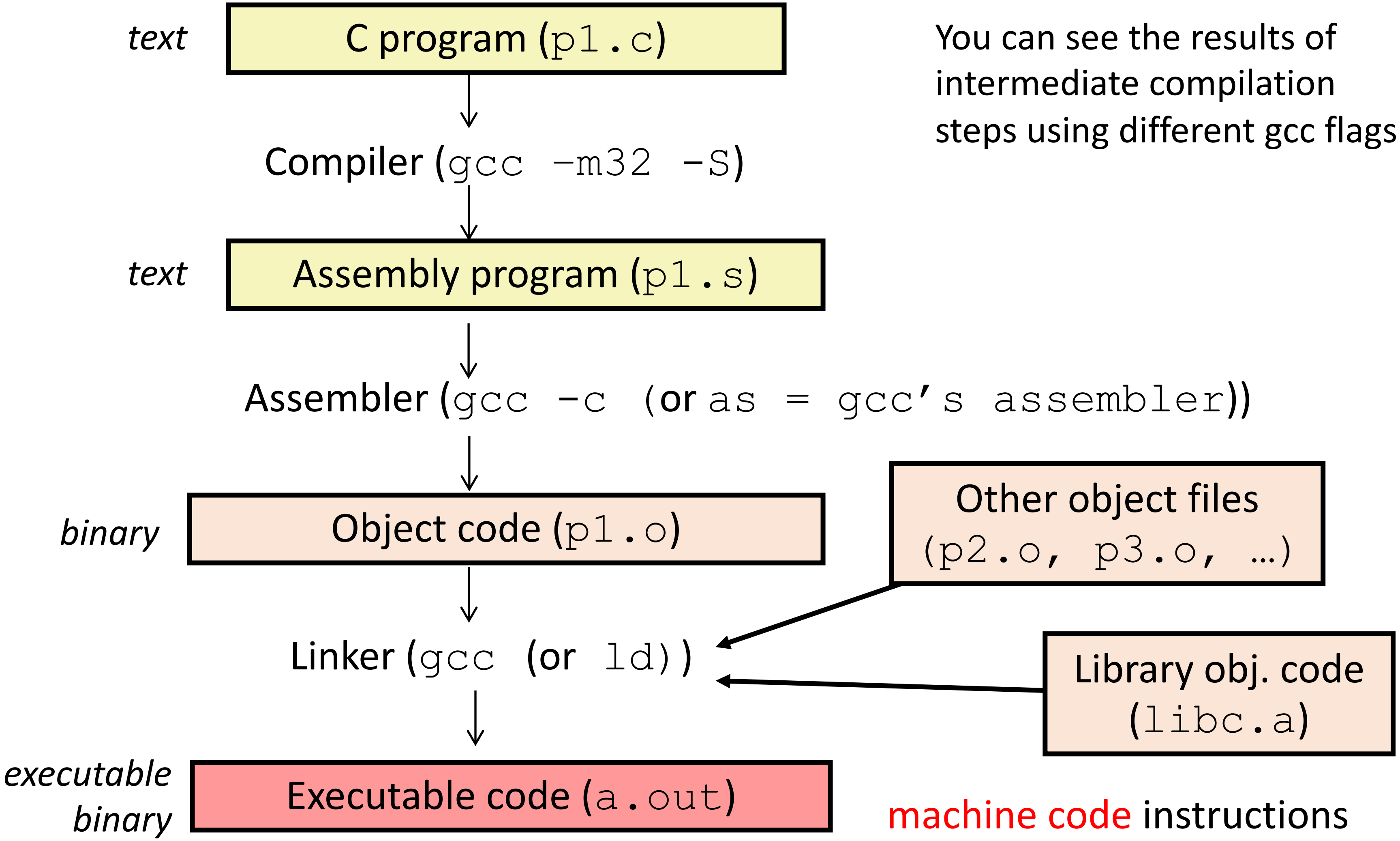
```
int i = 0;
int main()
{
   char *ptr = malloc(sizeof(int));
   char buf[1024]
   int j;
   static int y;
}
```

- i -> data segment
- ptr -> stack
  - data allocated on heap
- buf -> stack
- j -> stack
- y -> bss

X86: The De Facto Standard

- Extremely popular for desktop computers
- Alternatives
    - ARM: popular on mobile
    - MIPS: very simple
    - Itanium: ahead of its time
- CISC
    - 100 distinct opcodes
- Register poor
    - 8 registers of 32 bits
    - only 6 general purpose
- instructions are variable length
    - not aligned at 4 byte boundaries
- lots of backward compatibilities
    - defined in late 70s
    - exploit code that noone pays attention to
- we will use 32 bit because its more convenient.

Compilation Steps (.c to a.out)



text C program (`p1.c`)

Compiler (`gcc -m32 -S`)

text Assembly program (`p1.s`)

Assembler (`gcc -c` (or `as = gcc's assembler`))

binary Object code (`p1.o`)

Other object files (`p2.o, p3.o, …`)

Linker (`gcc` (or `ld`))

Library obj. code (`libc.a`)

executable binary Executable code (`a.out`)

You can see the results of intermediate compilation steps using different gcc flags

machine code instructions

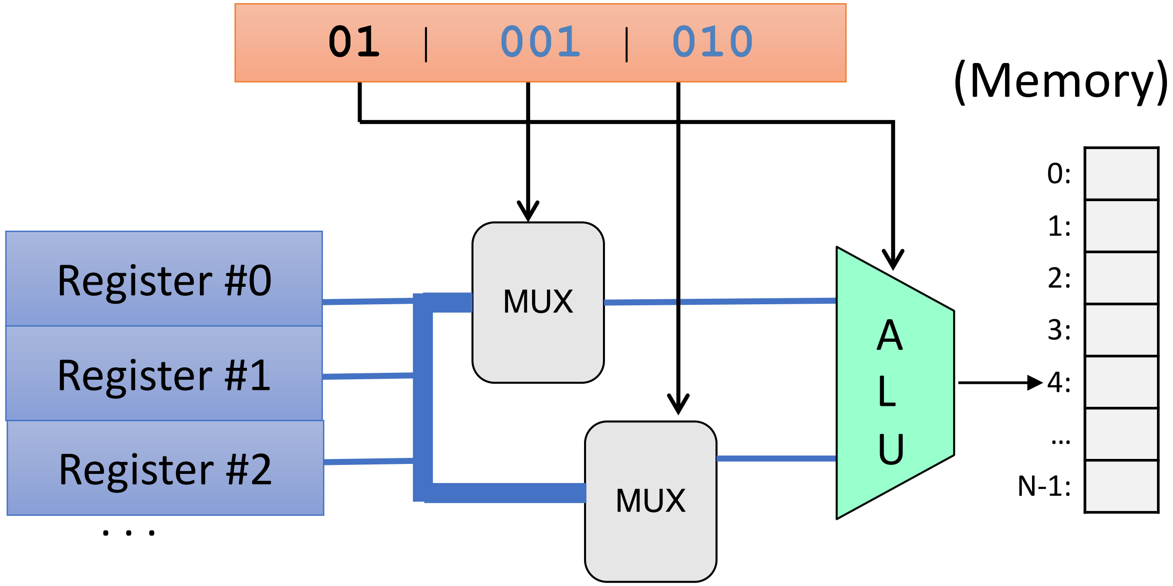
Machine Code

Binary (0's and 1's) Encoding of ISA Instructions

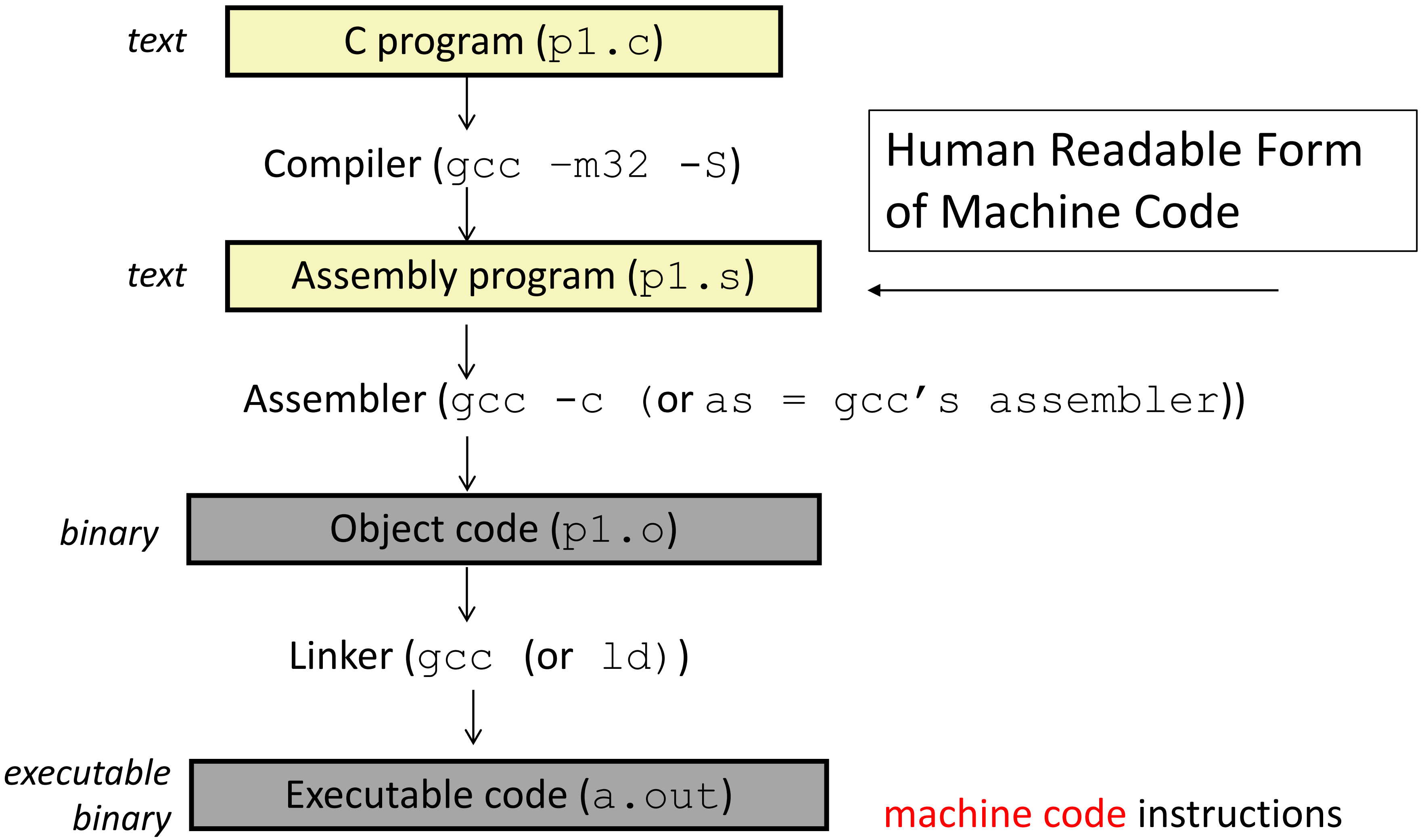
- some bits: encode the instruction (opcode bits)
- others encode operand(s)

```
(eg)  01001010   opcode operands
       01 001 010
     ADD %r1 %r2
```

- different bits fed through different CPU circuitry:



Assembly Code



text C program (`p1.c`)

Compiler (`gcc -m32 -S`)

text Assembly program (`p1.s`)

Human Readable Form of Machine Code

Assembler (`gcc -c` (or `as = gcc’s assembler`))

binary Object code (`p1.o`)

Linker (`gcc` (or `ld`))

executable binary Executable code (`a.out`)

machine code instructions

What is “assembly”?

```
push %ebp
mov  %esp, %ebp
sub  $16, %esp
movl $10, -8(%ebp)
movl $20, -4(%ebp)
movl -4(%ebp), $eax
addl $eax, -8(%ebp)
movl -8(%ebp), %eax
leave
```

Assembly is the “human readable” form of the instructions a machine can understand.

```
objdump -d a.out
```

Object / Executable / Machine Code

Assembly	Machine Code (Hexadecimal)
`push %ebp`	`55`
`mov  %esp, %ebp`	`89 E5`
`sub  $16, %esp`	`83 EC 10`
`movl $10, -8(%ebp)`	`C7 45 F8 0A 00 00 00`
`movl $20, -4(%ebp)`	`C7 45 FC 14 00 00 00`
`movl -4(%ebp), $eax`	`8B 45 FC`
`addl $eax, -8(%ebp)`	`01 45 F8`
`movl -8(%ebp), %eax`	`B8 45 F8`
`leave`	`C9`

Almost a 1-to-1 mapping to Machine Code
Hides some details like num bytes in instructions

Object / Executable / Machine Code

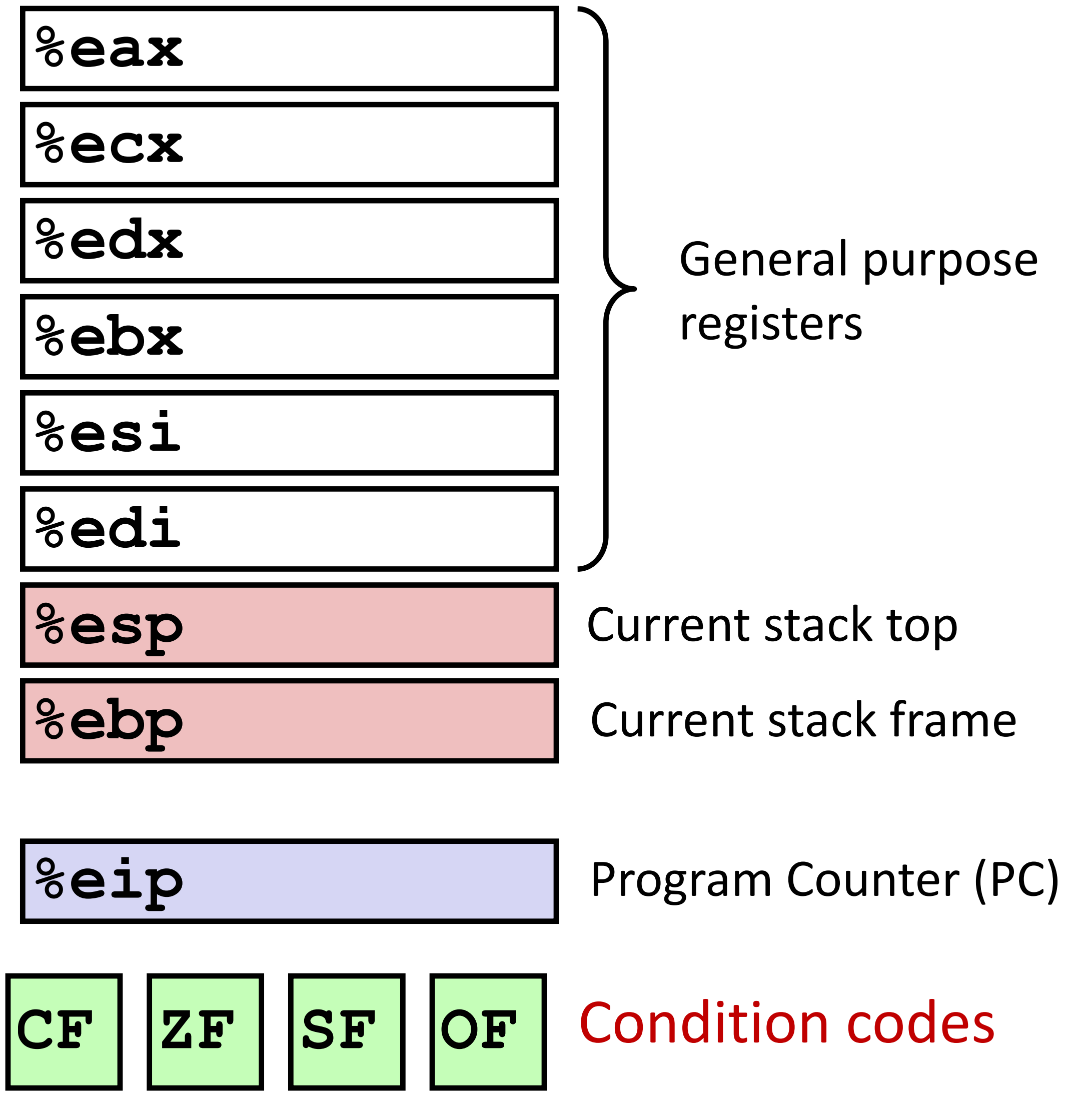
Assembly

```
push %ebp
mov  %esp, %ebp
sub  $16, %esp
movl $10, -8(%ebp)
movl $20, -4(%ebp)
movl -4(%ebp), $eax
addl $eax, -8(%ebp)
movl -8(%ebp), %eax
leave
```

```
int main() {
	int a = 10;
	int b = 20;

	a = a + b;

	return a;
}
```

Processor State in Registers

- Information about currently executing program
  - Temporary data
    (%eax - %edi)
  - Location of runtime stack
    (%ebp, %esp)
  - Location of current code control point (%eip, …)
  - Status of recent tests
    %EFLAGS
    (CF, ZF, SF, OF)



General purpose Registers

Six are for instruction operands

Can store 4 byte data or address value

The low-order 2 bytes %ax is the low-order 16 bits of %eax

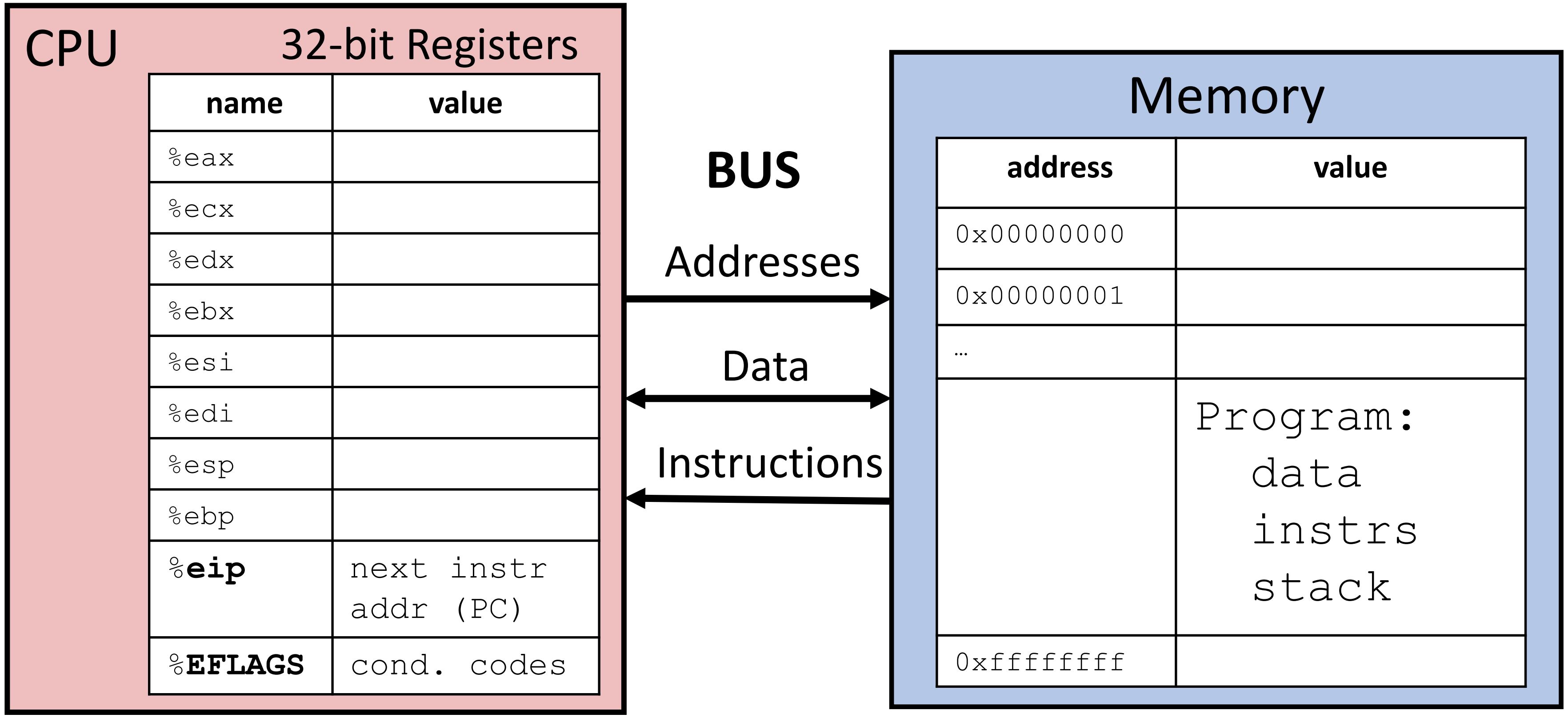
Two low-order 1 bytes %al is the low-order 8 bits of %eax

May see their use in ops involving shorts or chars

Register name
%eax
%ecx
%edx
%ebx
%esi
%edi
%esp
%ebp
%eip
%EFLAGS

bits: 16 31	15 8	7 0
%eax %ax	%ah	%al
%ecx %cx	%ch	%cl
%edx %dx	%dh	%dl
%ebx %bx	%bh	%bl
%esi %si		
%edi %di		
%esp %sp		
%ebp %bp		

Assembly Programmer's View of State



CPU

32-bit Registers

name	value
%eax	
%ecx	
%edx	
%ebx	
%esi	
%edi	
%esp	
%ebp	
%**eip**	next instr addr (PC)
%**EFLAGS**	cond. codes

Memory

address	value
0x00000000	
0x00000001	
…	
	Program: data instrs stack
0xffffffff	

Registers:

PC: Program counter (%eip)

Condition codes (%EFLAGS)

General Purpose (%eax - %ebp)

Memory:

- Byte addressable array
- Program code and data
- Execution stack

Types of IA32 Instructions

- Data movement
  - Move values between registers and memory
  - Example: `movl`

- Load: move data from memory to register

- Store: move data from register to memory

Instruction Syntax

Examples:

```
subl $16, %ebx

movl (%eax), %ebx
```

- Instruction ends with data length
- opcode, src, dst
- Constants preceded by $
- Registers preceded by %
- Indirection uses ( )

Addressing Mode: Memory

- Accessing memory requires you to specify which address you want.
  - Put address in a register.
  - Access with () around register name.

- `movl (%ecx), %eax`
  - Use the address in register ecx to access memory, store result in register eax

Addressing Mode: Memory

- `movl (%ecx), %eax`
  - Use the address in register ecx to access memory, store result in register eax

CPU Registers

name	value
`%eax`	`0`
`%ecx`	`0x1A68`
…	

(Memory)

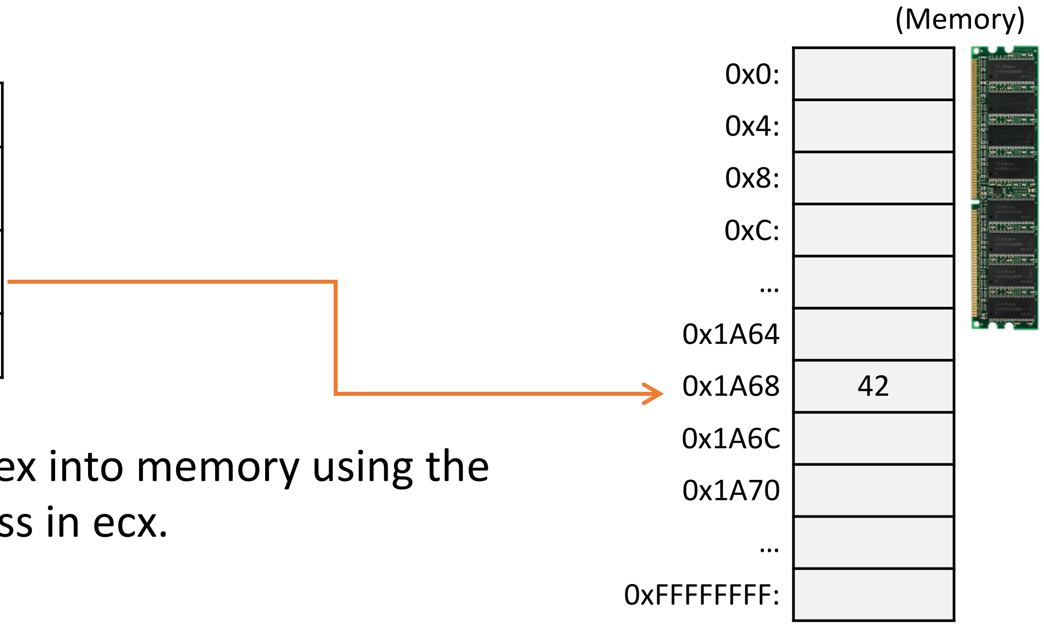
0x0:	
0x4:	
0x8:	
0xC:	
…	
0x1A64	
0x1A68	42
0x1A6C	
0x1A70	
…	
0xFFFFFFFF:	

Addressing Mode: Memory

- `movl (%ecx), %eax`
  - Use the address in register ecx to access memory, store result in register eax

CPU Registers

name	value
`%eax`	`0`
`%ecx`	`0x1A68`
…	



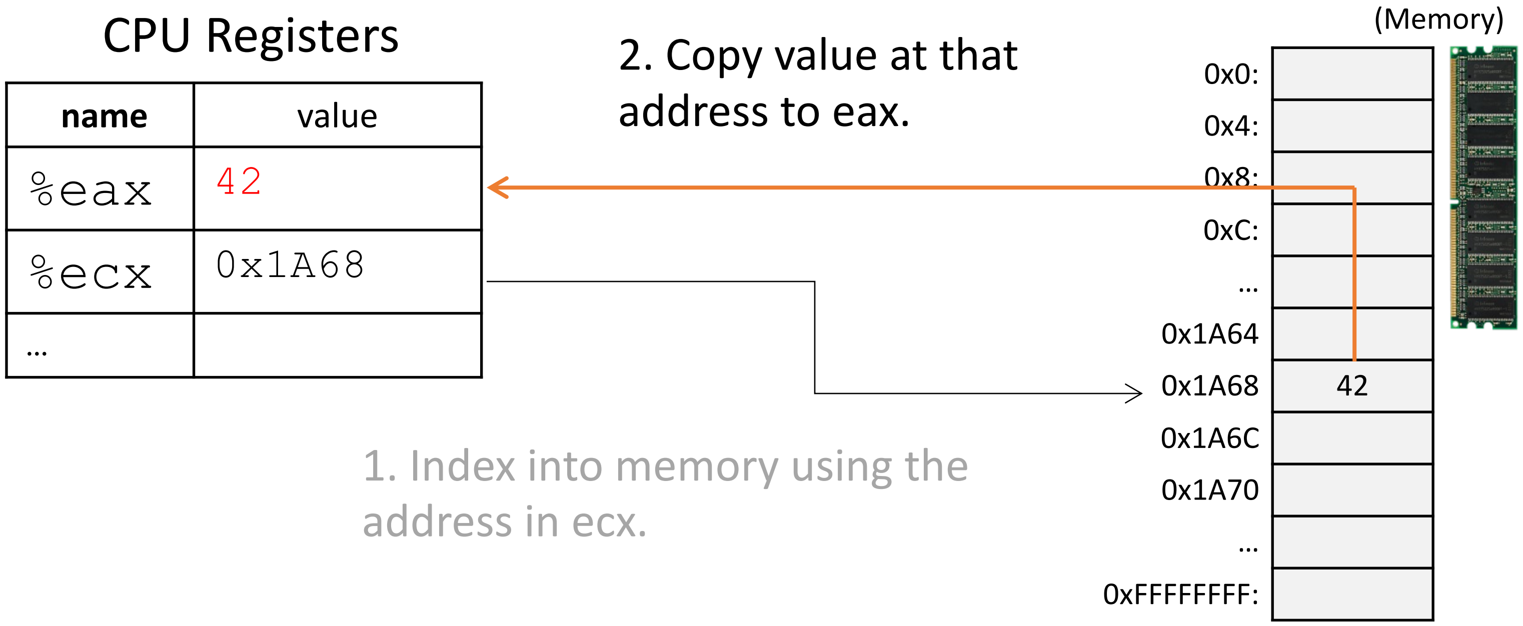
(Memory)

0x0:	
0x4:	
0x8:	
0xC:	
…	
0x1A64	
0x1A68	42
0x1A6C	
0x1A70	
…	
0xFFFFFFFF:	

1. Index into memory using the address in ecx.

Addressing Mode: Memory

- `movl (%ecx), %eax`
  - Use the address in register ecx to access memory, store result in register eax



CPU Registers

name	value
%eax	42
%ecx	0x1A68
…	

2. Copy value at that address to eax.

1. Index into memory using the address in ecx.

(Memory)

0x0:	
0x4:	
0x8:	
0xC:	
…	
0x1A64	
0x1A68	42
0x1A6C	
0x1A70	
…	
0xFFFFFFFF:	

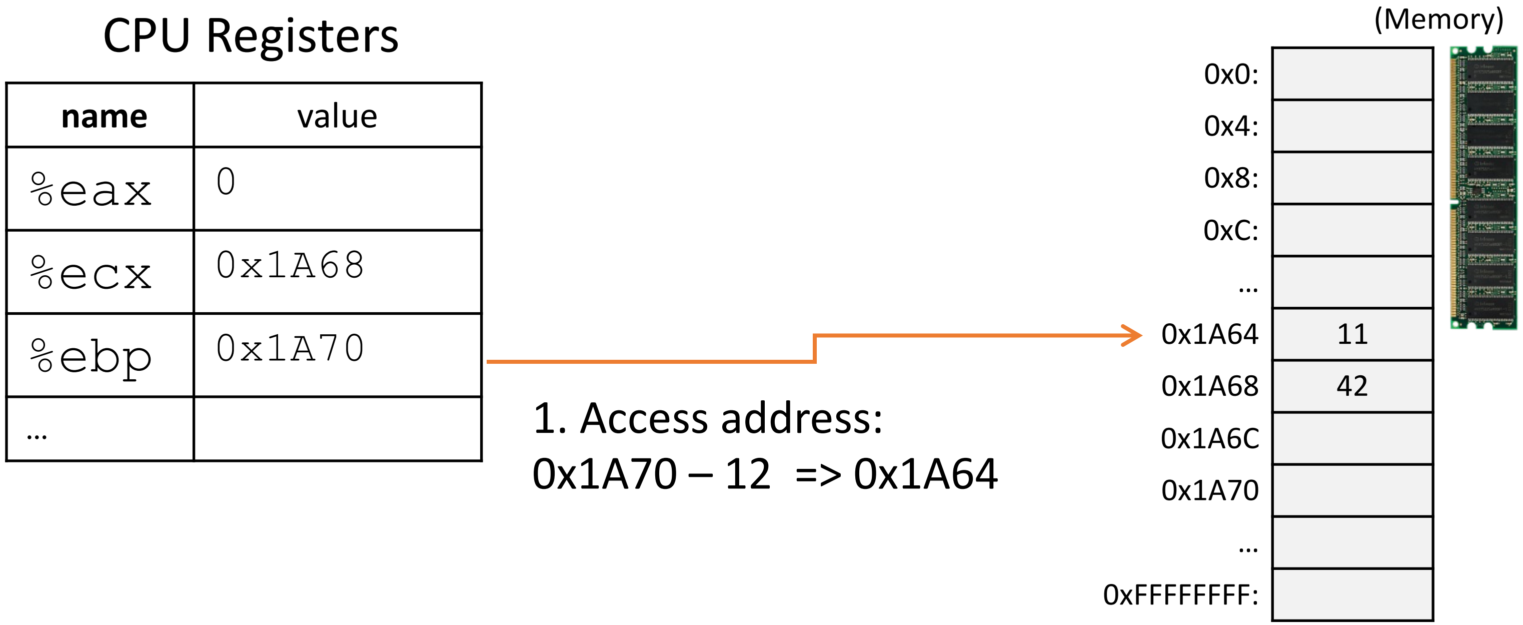
Addressing Mode: Displacement

- Like memory mode, but with constant offset
  - Offset is often negative, relative to %ebp

- `movl -12(%ebp), %eax`
  - Take the address in ebp, subtract twelve from it, index into memory and store the result in eax

Addressing Mode: Displacement

- `movl -12(%ebp), %eax`
  - Take the address in ebp, subtract twelve from it, index into memory and store the result in eax



CPU Registers

name	value
`%eax`	`0`
`%ecx`	`0x1A68`
`%ebp`	`0x1A70`
…	

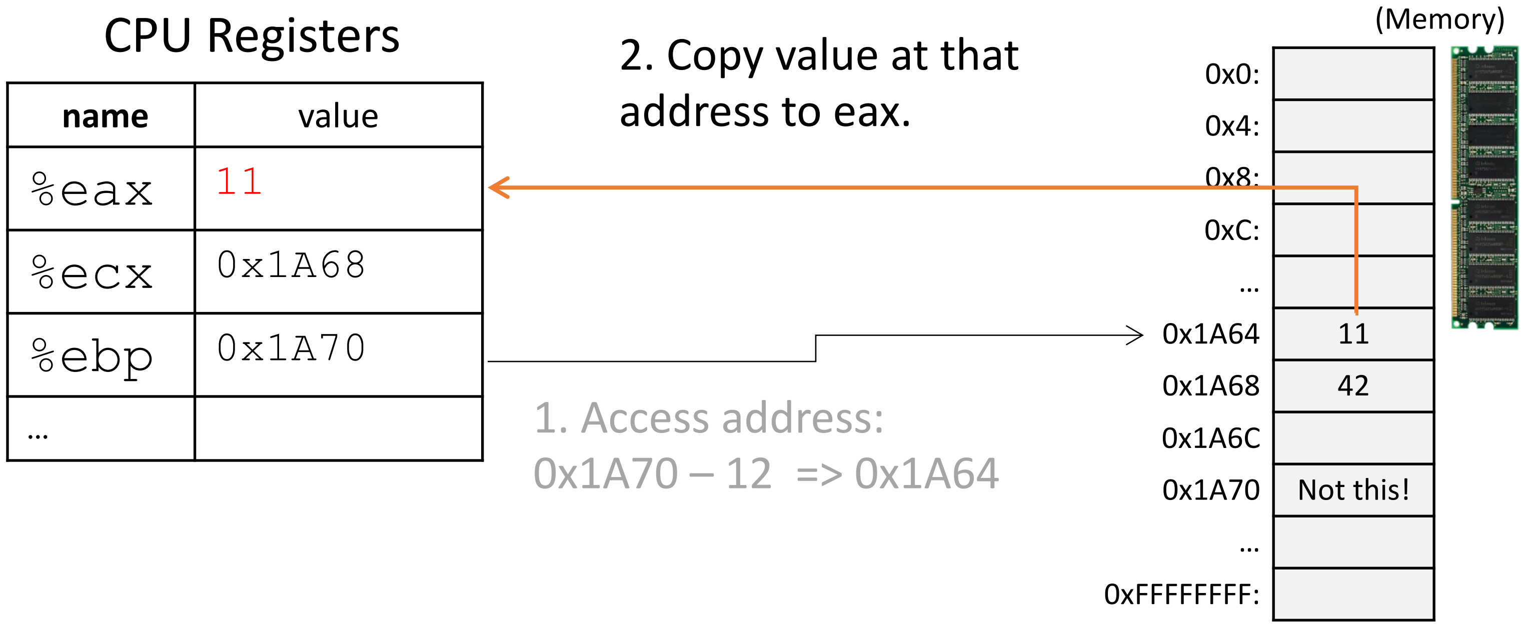
1. Access address:
0x1A70 – 12 => 0x1A64

(Memory)

0x0:	
0x4:	
0x8:	
0xC:	
…	
0x1A64	11
0x1A68	42
0x1A6C	
0x1A70	
…	
0xFFFFFFFF:	

Addressing Mode: Displacement

- `movl -12(%ebp), %eax`
  - Take the address in ebp, subtract three from it, index into memory and store the result in eax



CPU Registers

name	value
`%eax`	`11`
`%ecx`	`0x1A68`
`%ebp`	`0x1A70`
…	

1. Access address: 0x1A70 – 12 => 0x1A64

2. Copy value at that address to eax.

(Memory)

0x0:	
0x4:	
0x8:	
0xC:	
…	
0x1A64	11
0x1A68	42
0x1A6C	
0x1A70	Not this!
…	
0xFFFFFFFF:	

What will memory look like after these instructions?

x is 2 at `%ebp-8`, y is 3 at `%ebp-12`, z is 2 at `%ebp-16`

```
movl -16(%ebp),%eax
sall  $3, %eax
imull $3, %eax
movl -12(%ebp), %edx
addl  -8(%ebp), %edx
addl  %edx, %eax
movl  %eax, -8(%ebp)
```

Registers

name	value
%eax	?
%edx	?
%ebp	**0x1270** →

Memory

address	value
0x1260	2
0x1264	3
0x1268	2
0x126c	
0x1270	
…	

What will memory look like after these instructions?

x is 2 at `%ebp-8`, y is 3 at `%ebp-12`, z is 2 at `%ebp-16`

```
movl -16(%ebp),%eax
sall  $3, %eax
imull $3, %eax
movl -12(%ebp), %edx
addl  -8(%ebp), %edx
addl  %edx, %eax
movl  %eax, -8(%ebp)
```

D:

address	value
0x1260	2
0x1264	3
0x1268	53
0x126c	
0x1270	
…	

A:

address	value
0x1260	53
0x1264	3
0x1268	24
0x126c	
0x1270	
…	

B:

address	value
0x1260	53
0x1264	3
0x1268	2
0x126c	
0x1270	
…	

C:

address	value
0x1260	2
0x1264	16
0x1268	24
0x126c	
0x1270	
…	

Solution

x is 2 at `%ebp-8`, y is 3 at `%ebp-12`, z is 2 at `%ebp-16`

```
movl   -16(%ebp), %eax
sall    $3, %eax
imull   $3, %eax
movl   -12(%ebp), %edx
addl   -8(%ebp), %edx
addl   %edx, %eax
movl   %eax, -8(%ebp)
```

Equivalent C code:

```
x = z*24 + y + x;
```

name	value
%eax	
%edx	
%ebp	**0x1270**

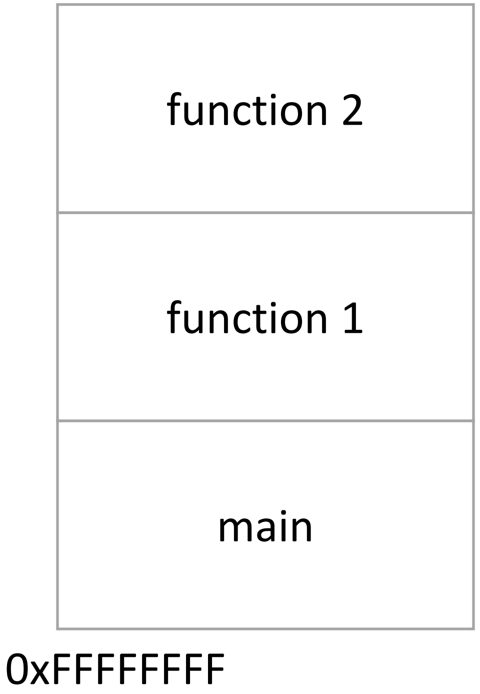
0x1260	2
0x1264	3
0x1268	2
0x126c	
0x1270	

Stack Frame Contents

- What needs to be stored in a stack frame?
    - Alternatively: What *must* a function know?

- Local variables
- Previous stack frame base address
- Function arguments
- Return value
- Return address

- Saved registers
- Spilled temporaries

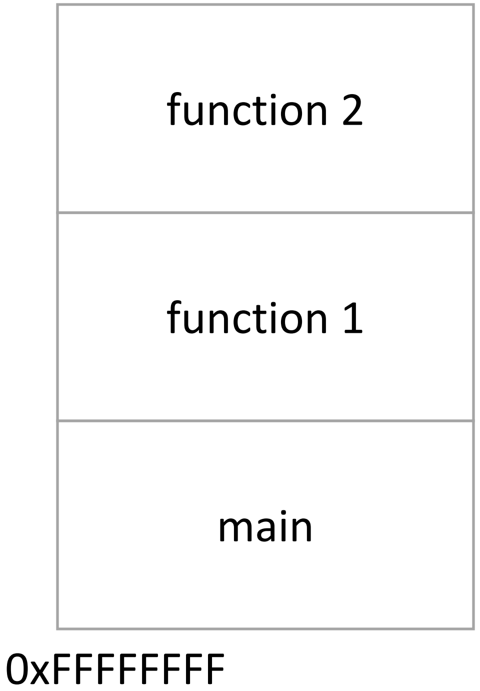


Stack Frame Contents

- What needs to be stored in a stack frame?
  - Alternatively: What *must* a function know?

- Local variables
- Previous stack frame base address
- Function arguments
- Return value
- Return address

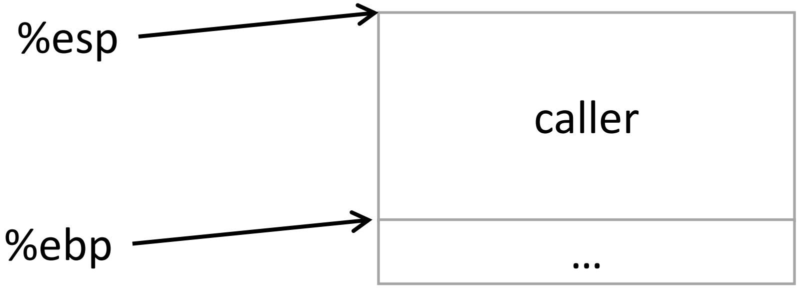
- Saved registers
- Spilled temporaries



Frame Pointer

- Must maintain invariant:
  - The current function's stack frame is always between the addresses stored in %esp and %ebp

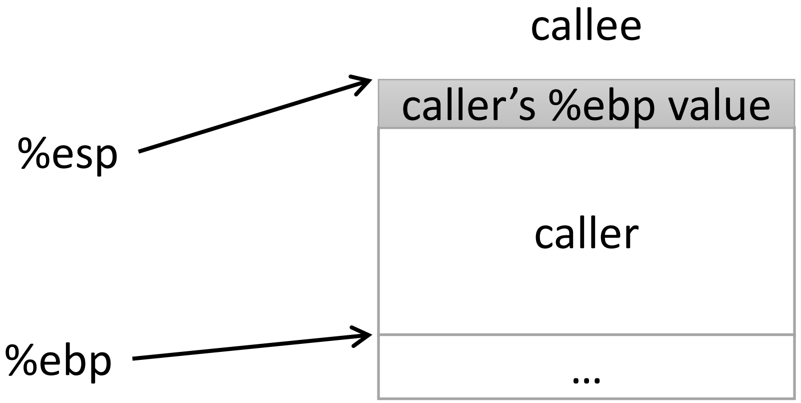
- Must adjust %esp, %ebp on call / return.



Frame Pointer

- Must maintain invariant:
  - The current function's stack frame is always between the addresses stored in %esp and %ebp

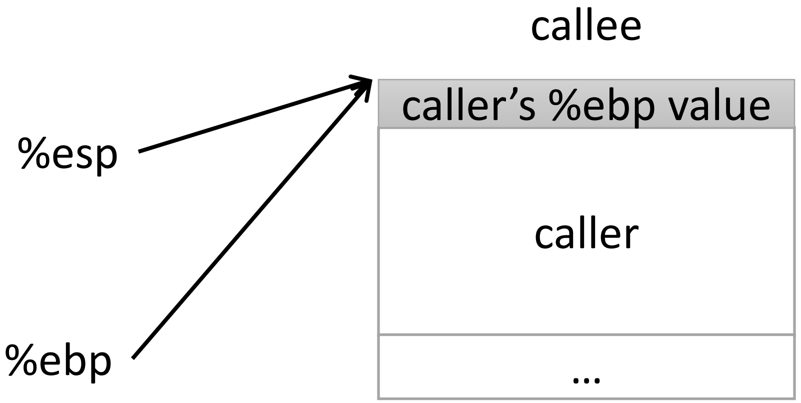
- Immediately upon calling a function:
  1. pushl %ebp



Frame Pointer

- Must maintain invariant:
  - The current function's stack frame is always between the addresses stored in %esp and %ebp

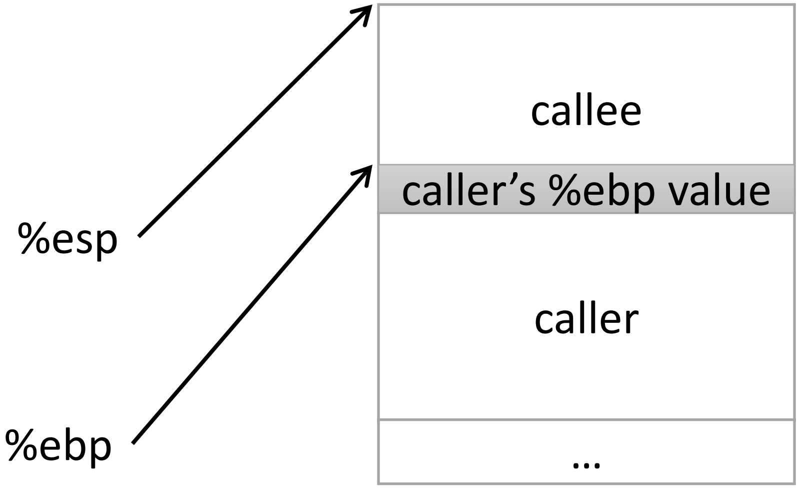
- Immediately upon calling a function:
  1. pushl %ebp
  2. Set %ebp = %esp



Frame Pointer

- Must maintain invariant:
  - The current function's stack frame is always between the addresses stored in %esp and %ebp

- Immediately upon calling a function:
  1. pushl %ebp
  2. Set %ebp = %esp
  3. Subtract N from %esp

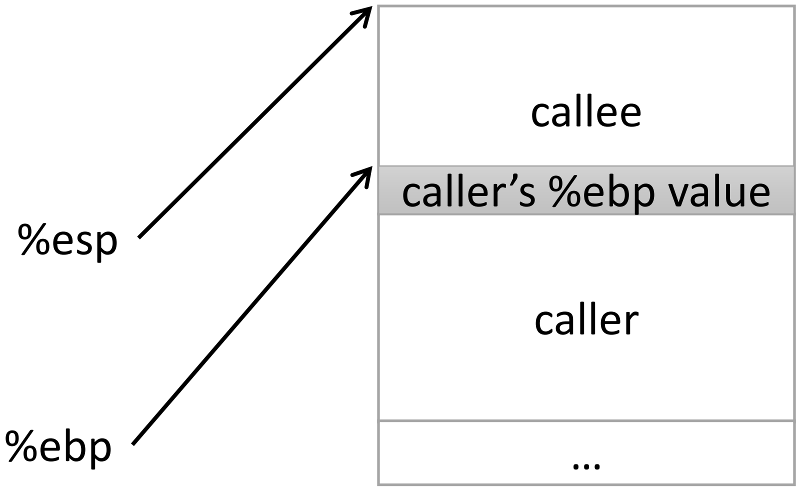


Callee can now execute.

Frame Pointer

- Must maintain invariant:
  - The current function's stack frame is always between the addresses stored in %esp and %ebp

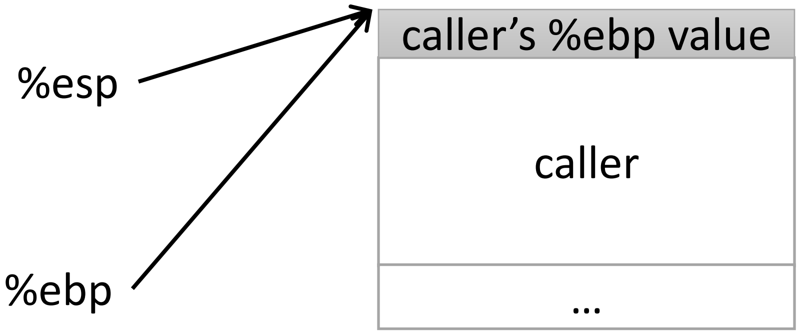
- To return, reverse this:



Frame Pointer

- Must maintain invariant:
  - The current function's stack frame is always between the addresses stored in %esp and %ebp

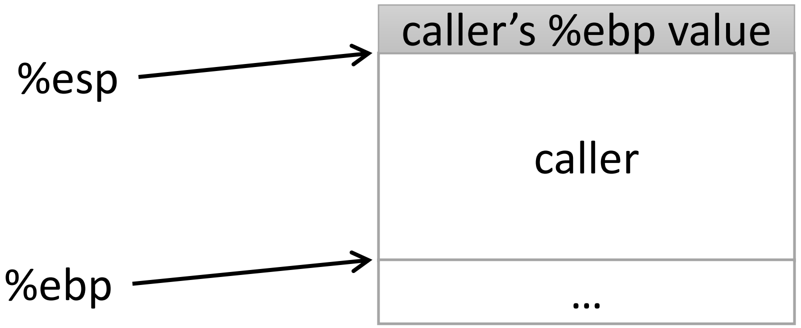
- To return, reverse this:
  1. set %esp = %ebp



Frame Pointer

- Must maintain invariant:
  - The current function's stack frame is always between the addresses stored in %esp and %ebp

- To return, reverse this:
  1. set %esp = %ebp
  2. popl %ebp

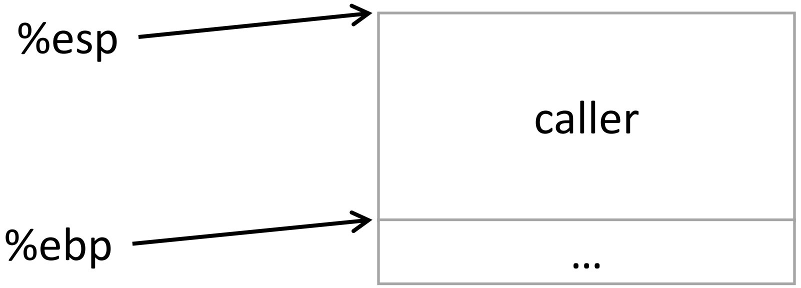


Frame Pointer

- Must maintain invariant:
  - The current function's stack frame is always between the addresses stored in %esp and %ebp

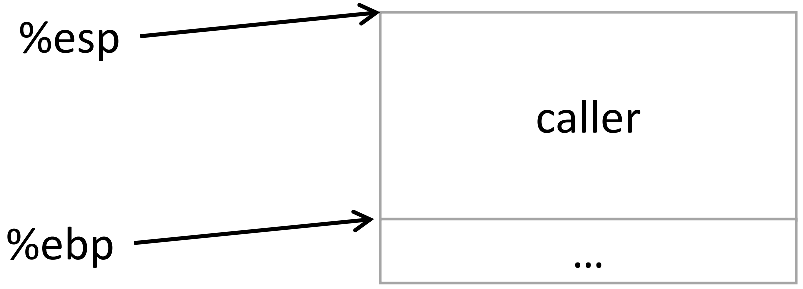
- To return, reverse this:
  1. set %esp = %ebp
  2. popl %ebp

IA32 has another convenience instruction for this: leave

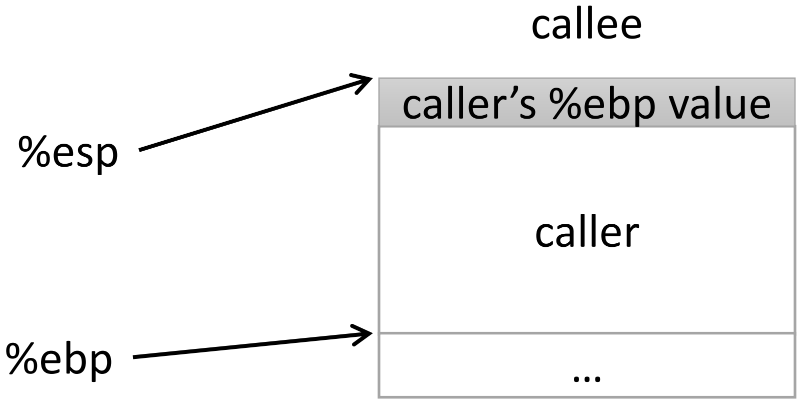


Back to where we started.

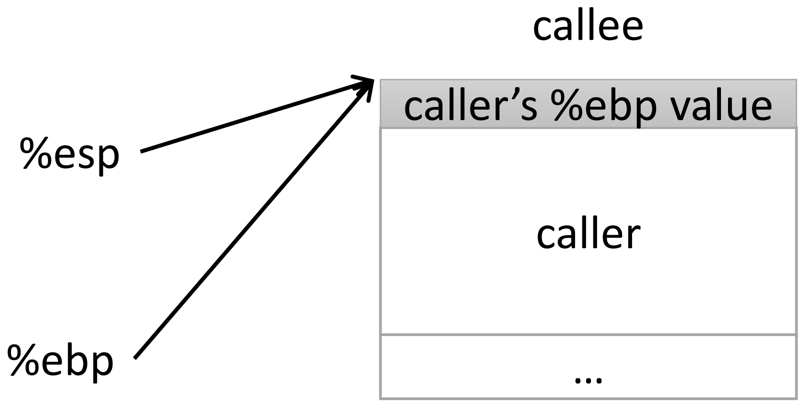
Frame Pointer: Function Call



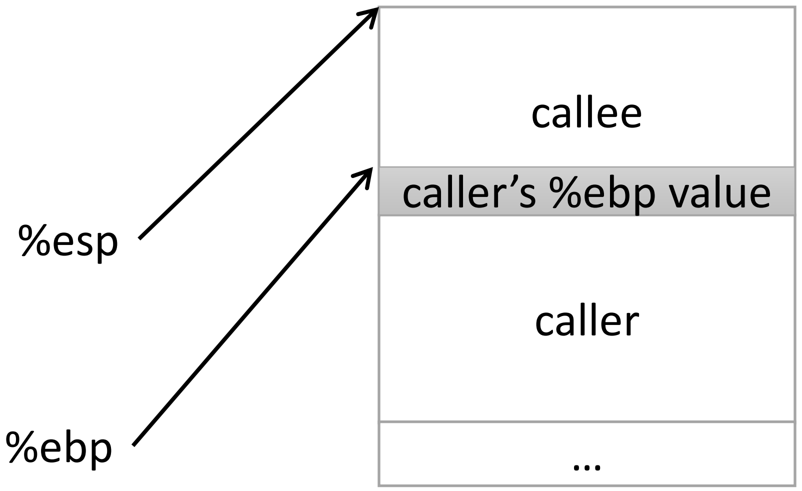
Initial state



pushl %ebp (store caller's frame pointer)

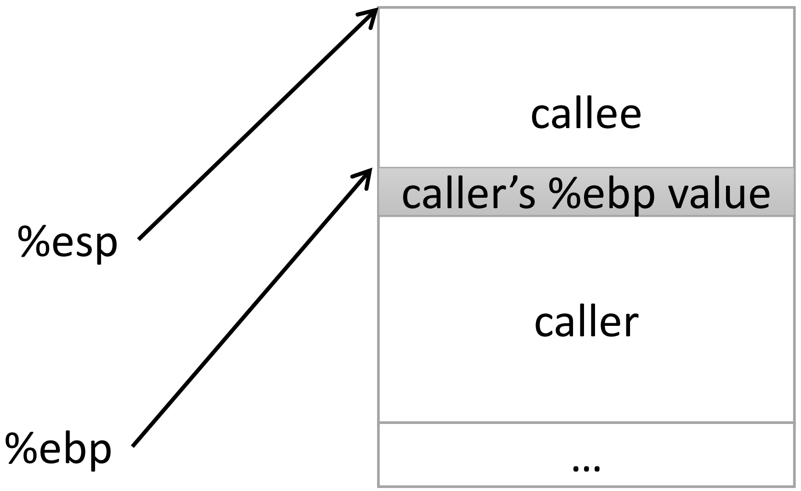


movl %esp, %ebp
(establish callee's frame pointer)

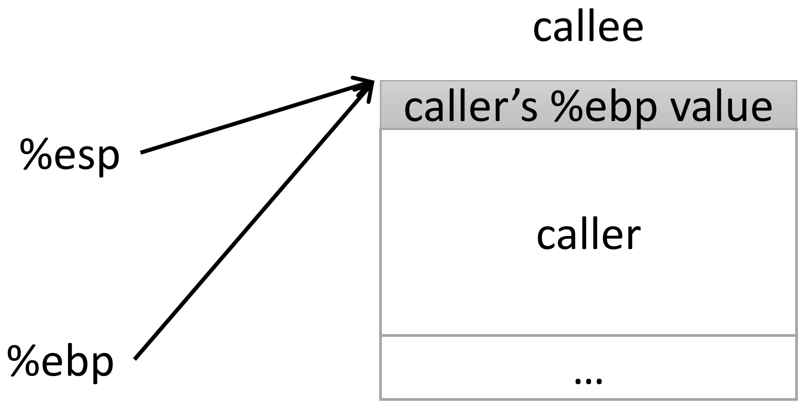


subl $SIZE, %esp
(allocate space for callee's locals)

Frame Pointer: Function Return



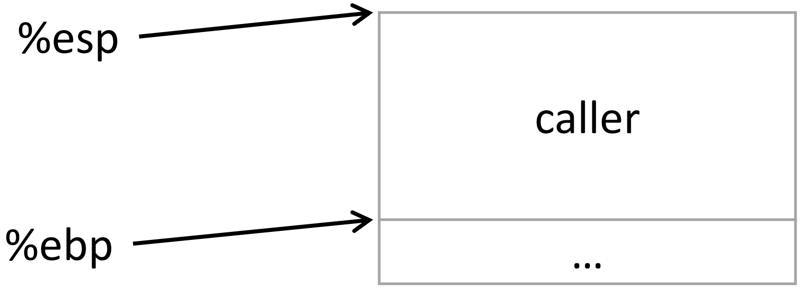
Want to restore caller's frame.



movl %ebp, %esp
(restore caller's stack pointer)

IA32 provides a convenience instruction that does all of this:

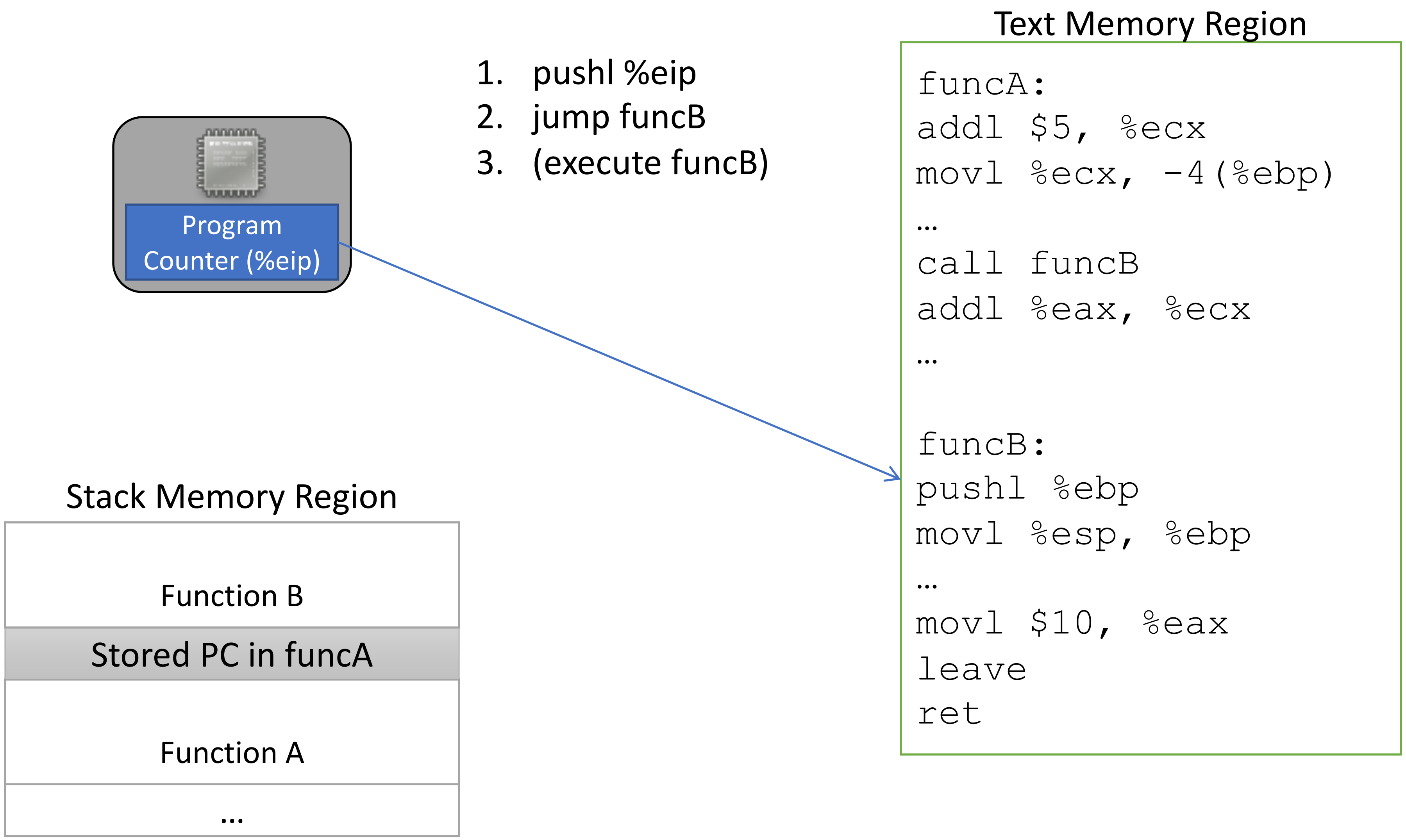
```
leave
```



popl %ebp (restore caller's frame pointer)

Functions and the Stack

1. pushl %eip
2. jump funcB
3. (execute funcB)



Stack Memory Region

Function B

Stored PC in funcA

Function A

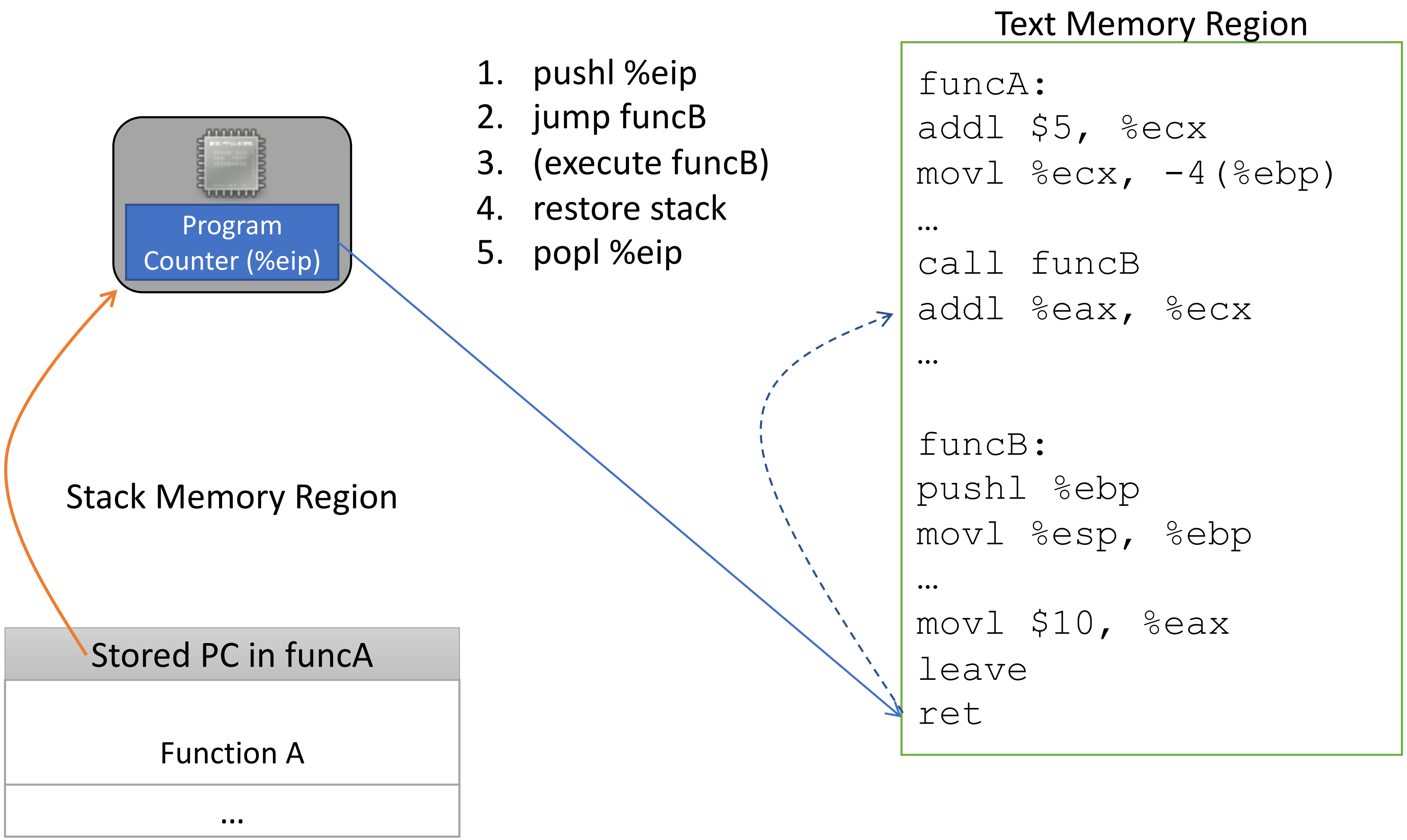
…

Text Memory Region

```
funcA:
addl $5, %ecx
movl %ecx, -4(%ebp)
…
call funcB
addl %eax, %ecx
…

funcB:
pushl %ebp
movl %esp, %ebp
…
movl $10, %eax
leave
ret
```

Functions and the Stack

1. pushl %eip
2. jump funcB
3. (execute funcB)
4. restore stack
5. popl %eip

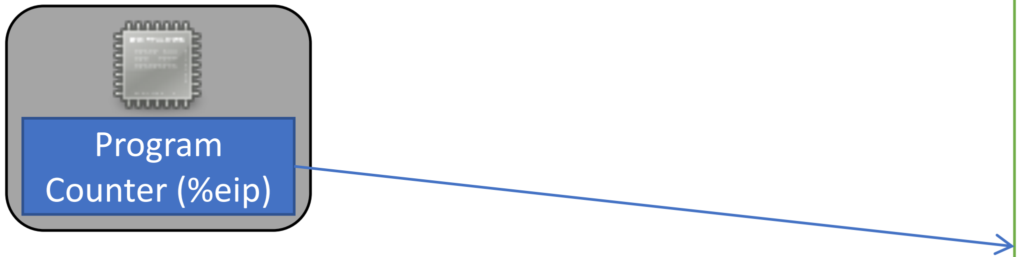


Text Memory Region

```
funcA:
addl $5, %ecx
movl %ecx, -4(%ebp)
…
call funcB
addl %eax, %ecx
…

funcB:
pushl %ebp
movl %esp, %ebp
…
movl $10, %eax
leave
ret
```

Functions and the Stack

6. (resume funcA)



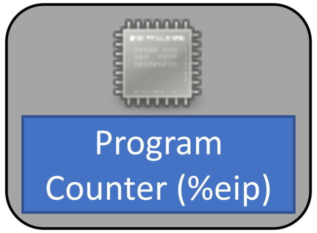
Text Memory Region

```
funcA:
addl $5, %ecx
movl %ecx, -4(%ebp)
…
call funcB
addl %eax, %ecx
…

funcB:
pushl %ebp
movl %esp, %ebp
…
movl $10, %eax
leave
ret
```

Stack Memory Region

Function A
…

Functions and the Stack



1. pushl %eip
2. jump funcB
3. (execute funcB)
4. restore stack
5. popl %eip
6. (resume funcA)

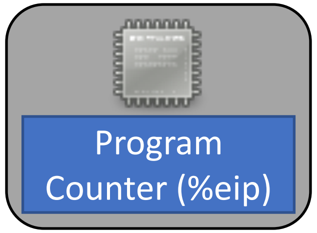
Stack Memory Region

Stored PC in funcA
Function A
…

Text Memory Region

```
funcA:
addl $5, %ecx
movl %ecx, -4(%ebp)
…
call funcB
addl %eax, %ecx
…

funcB:
pushl %ebp
movl %esp, %ebp
…
movl $10, %eax
leave
ret
```

Functions and the Stack



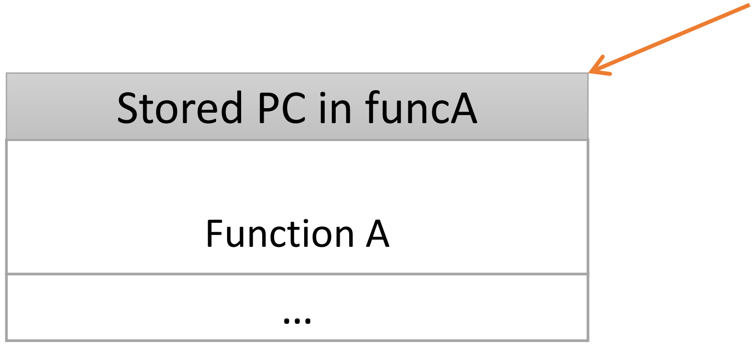
1. pushl %eip
2. jump funcB
3. (execute funcB)
4. restore stack
5. popl %eip
6. (resume funcA)

call

leave

ret

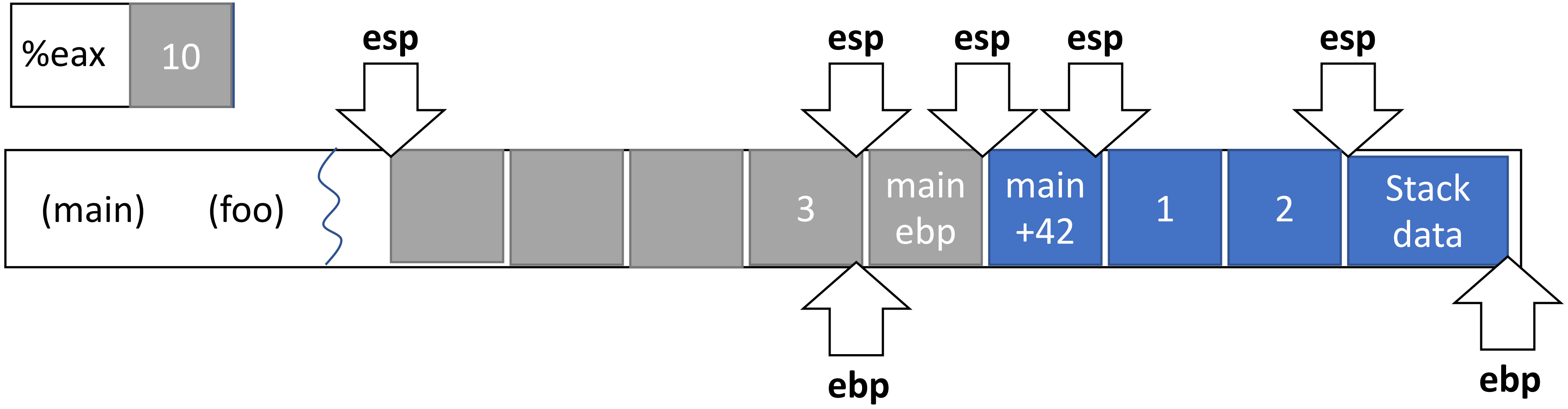
Stack Memory Region



Return address:

Address of the instruction we should jump back to when we finish (return from) the currently executing function.

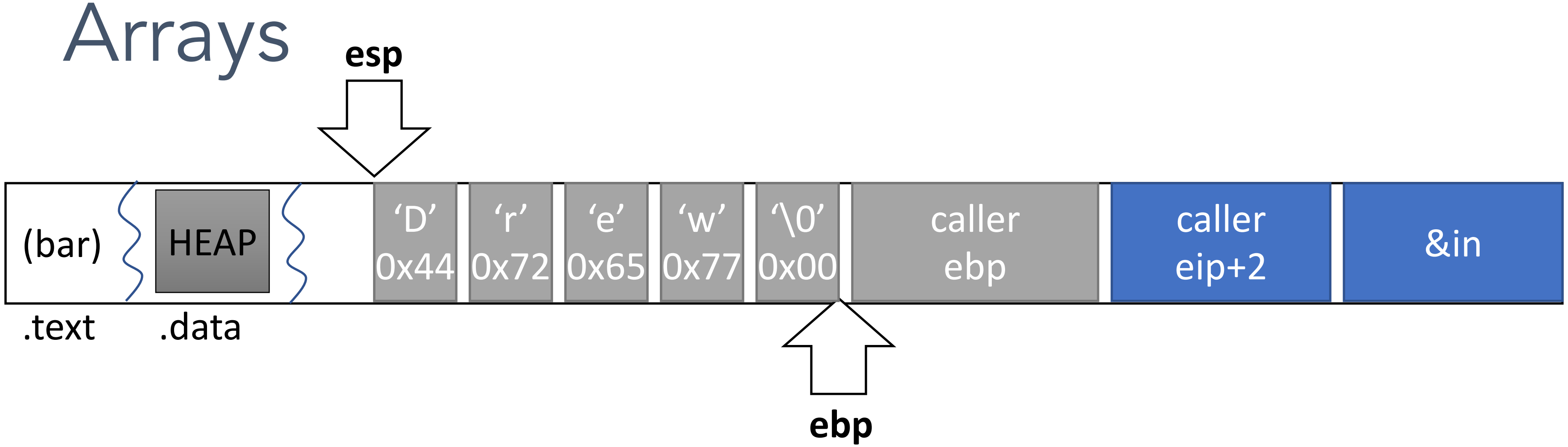
Implementing a function call



```
main:
        …
  eip   subl    $8, %esp
  eip   movl    $2, 4(%esp)
  eip   movl    $1, (%esp)
  eip   call    foo
  eip   addl    $8, %esp
        …
```

```
foo:
  eip   pushl   %ebp
  eip   movl    %esp, %ebp
  eip   subl    $16, %esp
  eip   movl    $3, -4(%ebp)
  eip   movl    8(%ebp), %eax
  eip   addl    $9, %eax
  eip   leave
  eip   ret
```

Arrays



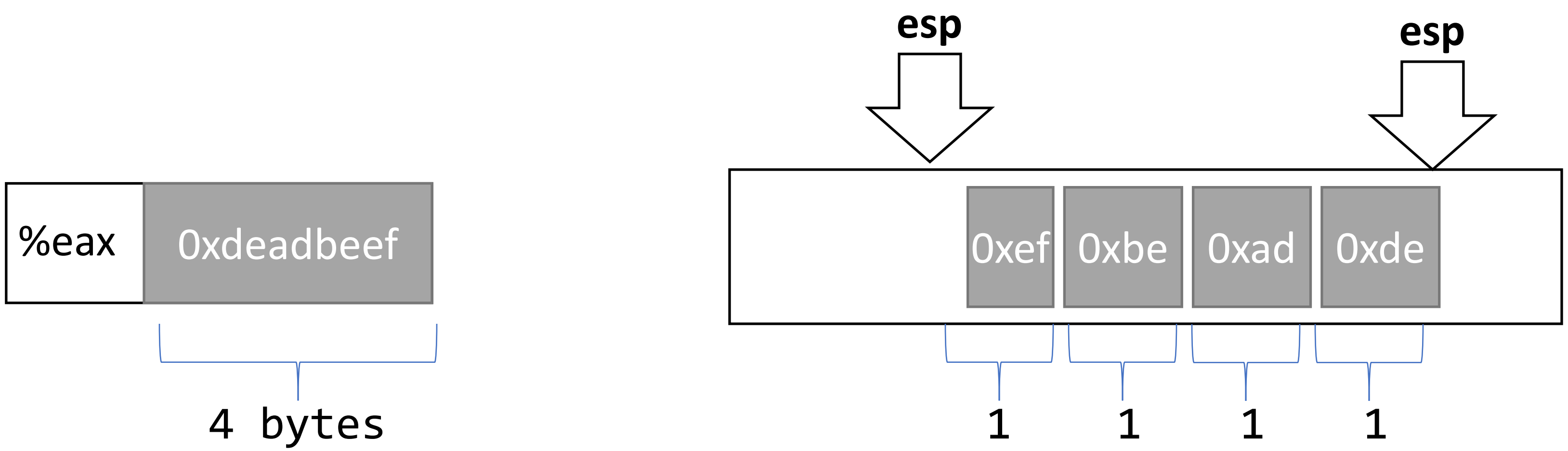
```
void bar(char * in){
  char name[5];
  strcpy(name, in);
}
```

```
bar:
  pushl  %ebp
  movl   %esp, %ebp
  subl   $5, %esp
  movl   8(%ebp), %eax
  movl   %eax, 4(%esp)
  leal   -5(%ebp), %eax
  movl   %eax, (%esp)
  call   strcpy
  leave
  ret
```

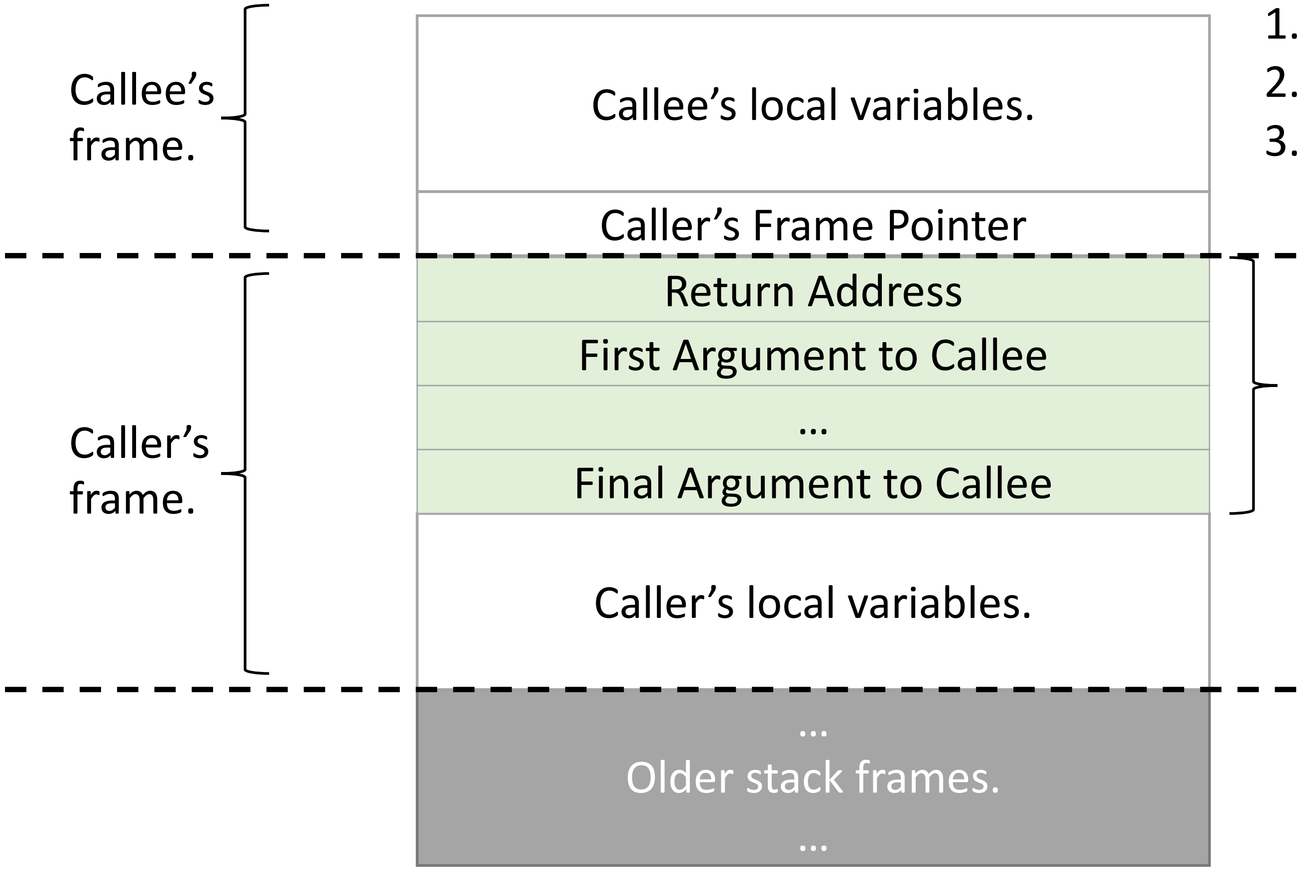
Data types / Endianness

- x86 is a little-endian architecture

```
pushl %eax
```



Putting it all together…



Callee's frame.

Callee's local variables.

Caller's Frame Pointer

Caller's frame.

Return Address

First Argument to Callee

…

Final Argument to Callee

Caller's local variables.

…

Older stack frames.

…

Callee Code

1. push frame pointer
2. move stack pointer to frame pointer
3. increase stack pointer

Caller Code

1. save address of next instruction
2. push arguments

Register Convention

- Caller-saved: %eax, %ecx, %edx
  - If the caller wants to preserve these registers, it must save them prior to calling callee
  - callee free to trash these, caller will restore if needed

This is why I've told you to only use these three registers.

- Callee-saved: %ebx, %esi, %edi
  - If the callee wants to use these registers, it must save them first, and restore them before returning
  - caller can assume these will be preserved

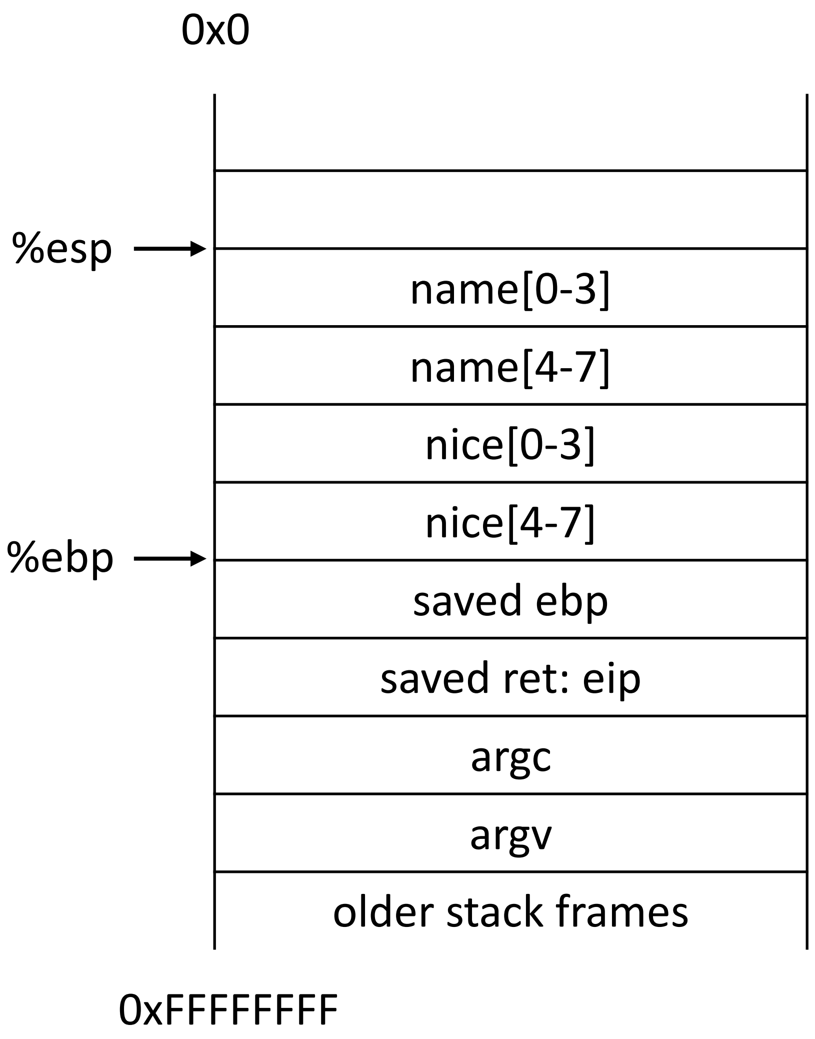
Buffer Overflows

When is a program secure?

- Formal approach: When it does exactly what it should
  - not more
  - not less
- But how do we know what it is supposed to do?

Example 1

```
#include <stdio.h>
#include <string.h>

int main(int argc, char**argv){
   char nice[] = “is nice.”;
   char name[8];
   gets(name);
   printf(“%s %s\n”, name, nice);
   return 0;
}
```

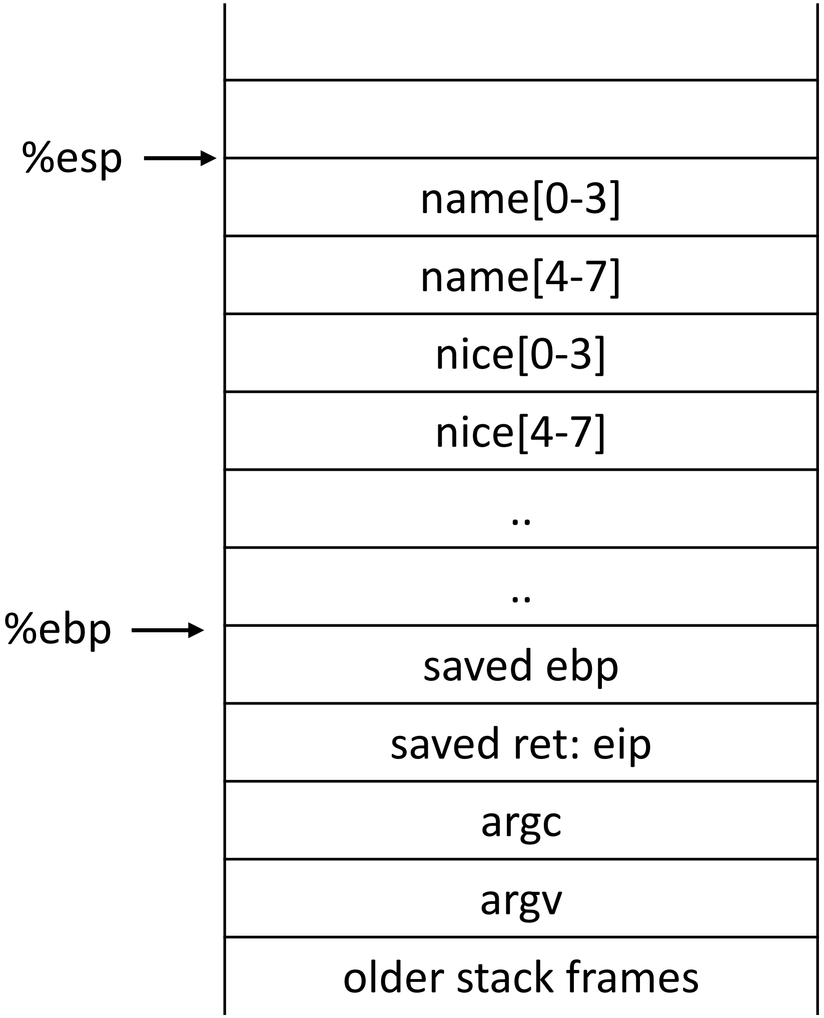


Function call stack

What happens if we <u>read</u> a long name?

```
#include <stdio.h>
#include <string.h>

int main(int argc, char**argv){
   char nice[] = “is nice.”;
   char name[8];
   gets(name);
   printf(“%s %s\n”, name, nice);
   return 0;
}
```

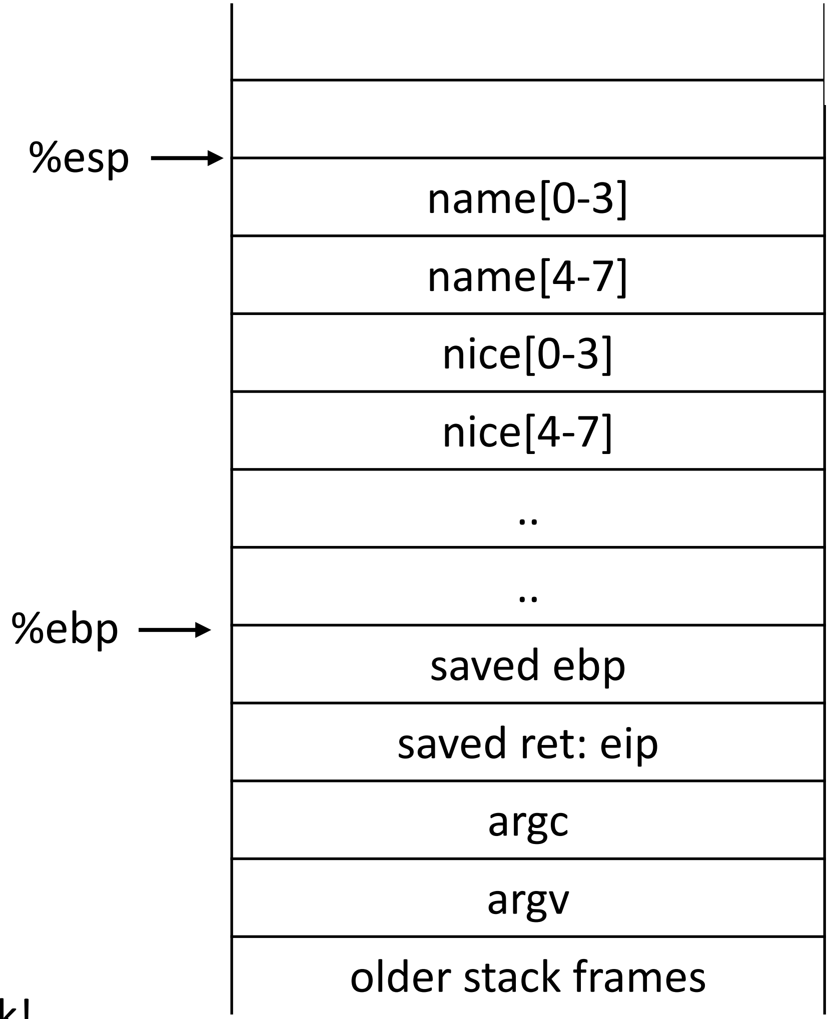
A. Nothing bad will happen
B. Something nonsensical will result
C. Something terrible will result



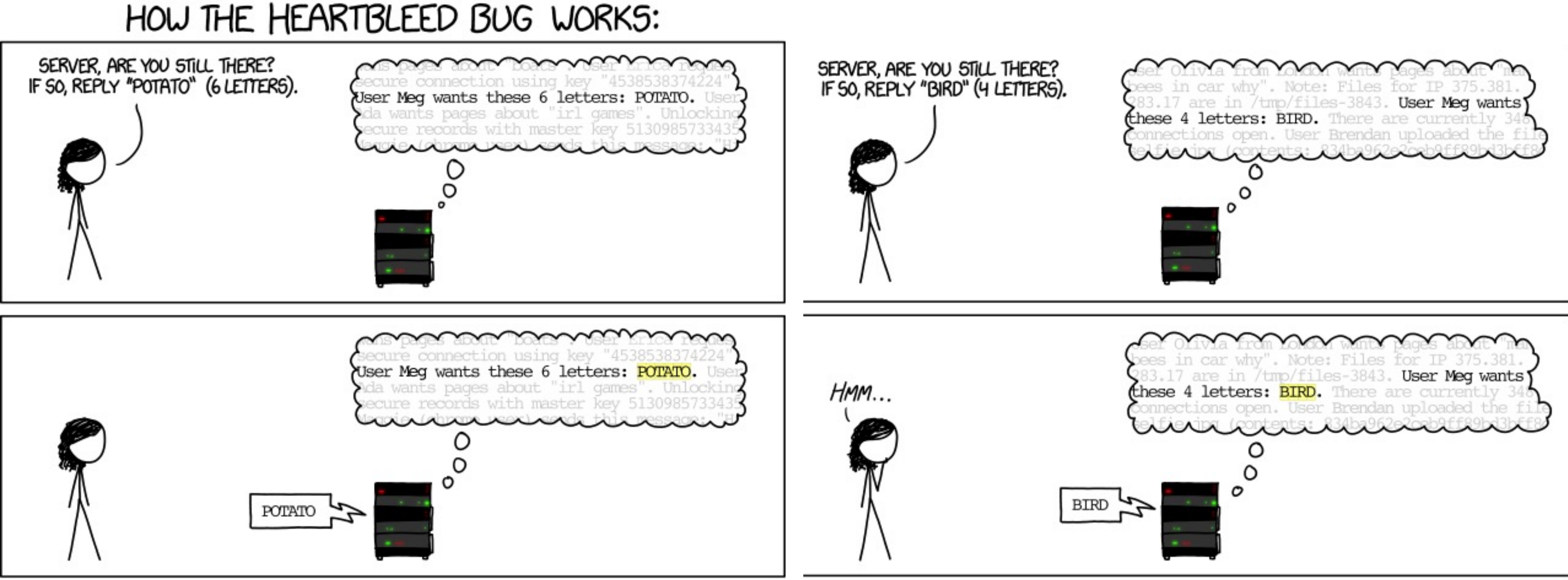
Function call stack

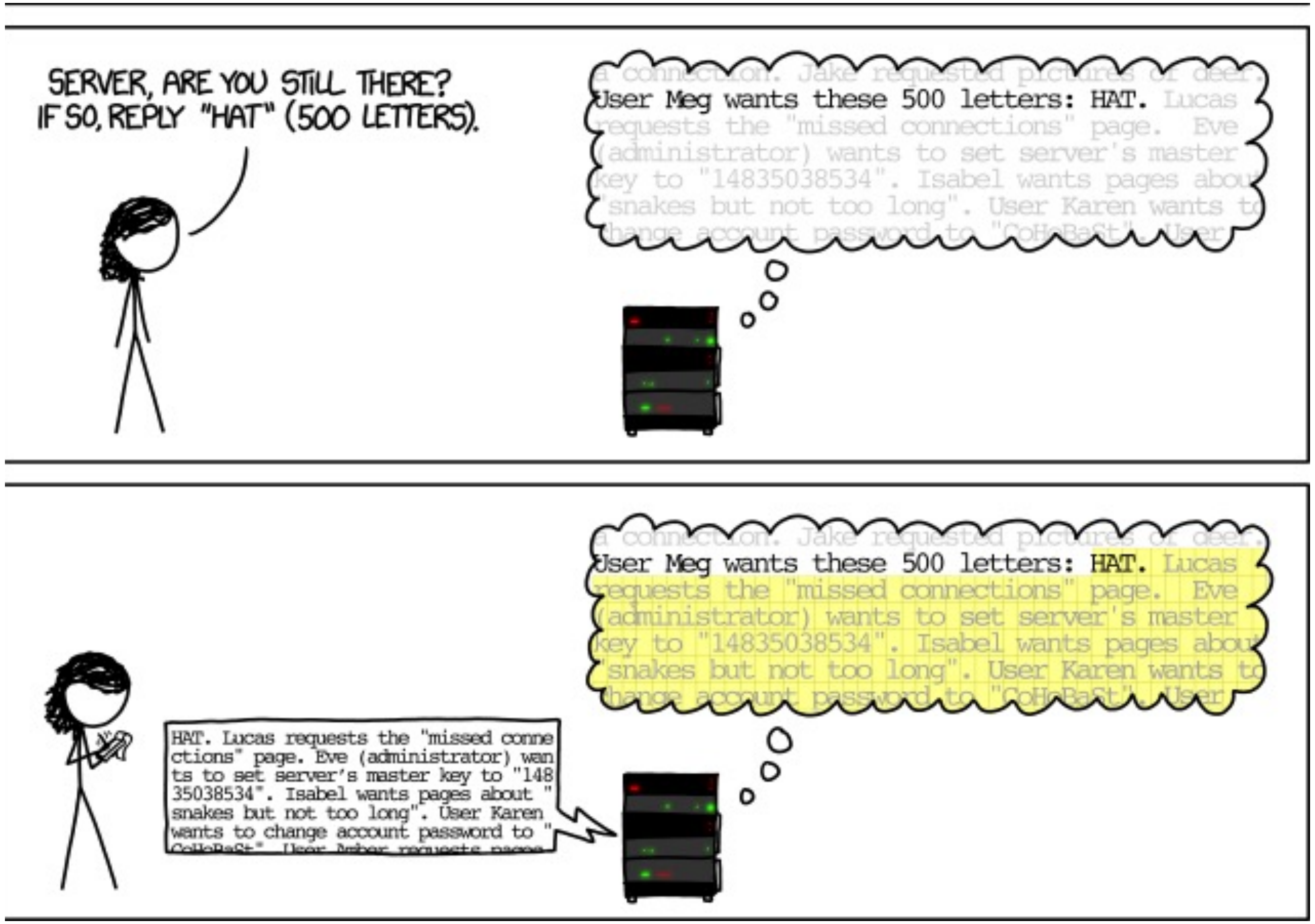
What happens if we read a long name?

```
#include <stdio.h>
#include <string.h>

int main(int argc, char**argv){
   char nice[] = “is nice.”;
   char name[8];
   gets(name);
   printf(“%s %s\n”, name, nice);
   return 0;
}
```

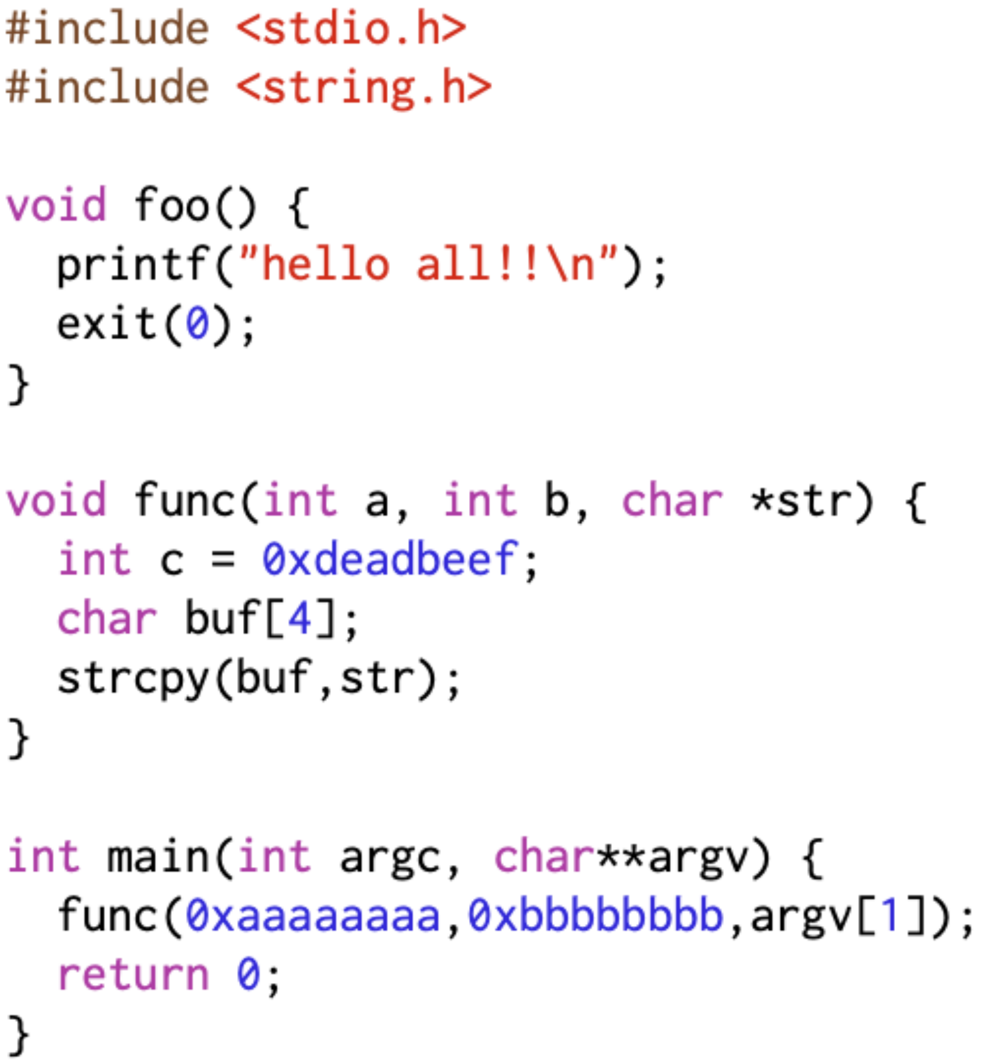


It it is not null terminated it can read a lot more of the stack!


HOW THE HEARTBLEED BUG WORKS:
SERVER, ARE YOU STILL THERE? IF SO, REPLY "POTATO" (6 LETTERS).
ants pages about "boats". User Erica reques secure connection using key "4538538374224" User Meg wants these 6 letters: POTATO. User Ada wants pages about "irl games". Unlocking secure records with master key 5130985733435 Maggie (chrome user) sends this message: "H
SERVER, ARE YOU STILL THERE? IF SO, REPLY "BIRD" (4 LETTERS).
User Olivia from London wants pages about "ma bees in car why". Note: Files for IP 375.381. 283.17 are in /tmp/files-3843. User Meg wants these 4 letters: BIRD. There are currently 346 connections open. User Brendan uploaded the file selfie.jpg (contents: 834ba962e2ceb9ff89bd3bff8
ants pages about "boats". User Erica reques secure connection using key "4538538374224" User Meg wants these 6 letters: POTATO. User Ada wants pages about "irl games". Unlocking secure records with master key 5130985733435 Maggie (chrome user) sends this message: "H
POTATO
HMM...
User Olivia from London wants pages about "ma bees in car why". Note: Files for IP 375.381. 283.17 are in /tmp/files-3843. User Meg wants these 4 letters: BIRD. There are currently 346 connections open. User Brendan uploaded the file selfie.jpg (contents: 834ba962e2ceb9ff89bd3bff8
BIRD

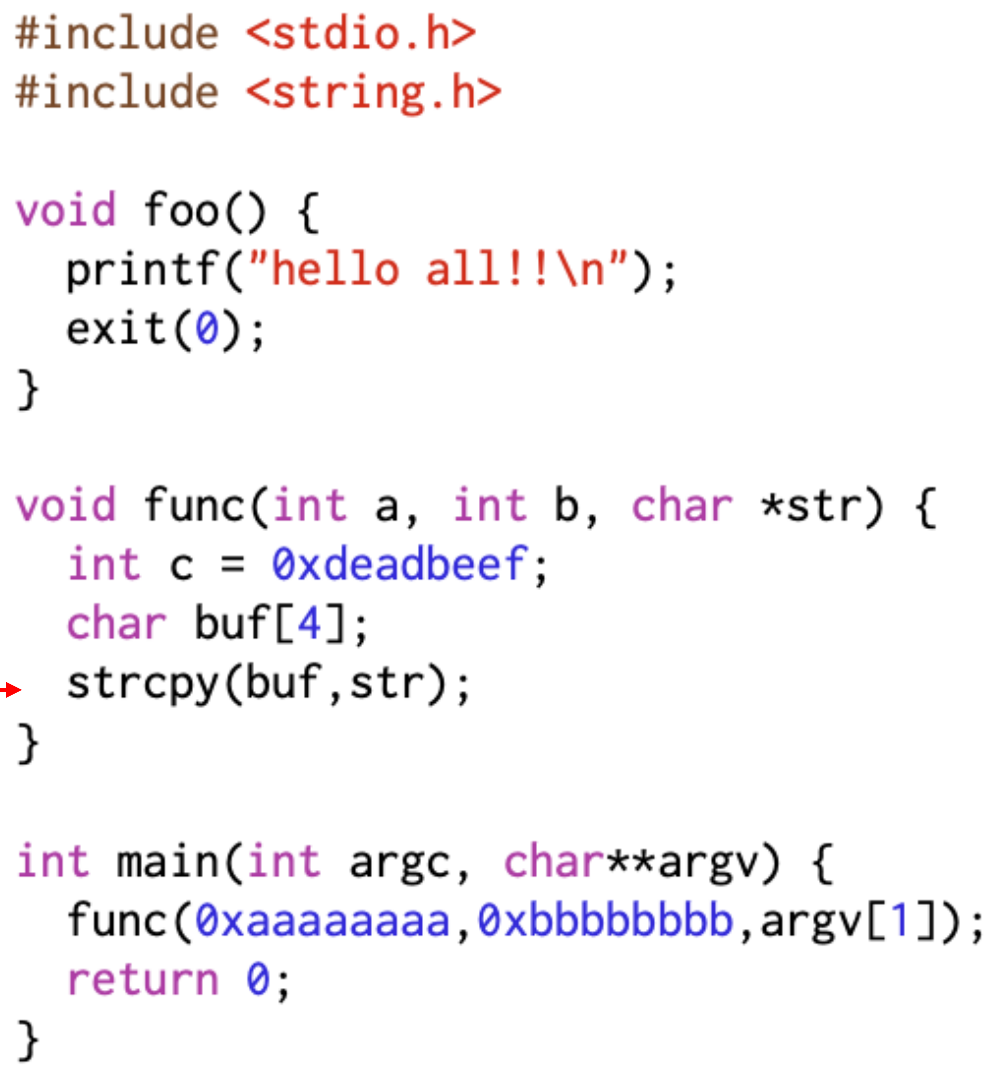

SERVER, ARE YOU STILL THERE? IF SO, REPLY "HAT" (500 LETTERS).
a connection. Jake requested pictures of deer. User Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages abou snakes but not too long". User Karen wants to hange account password to "CoHoBaSt". User
a connection. Jake requested pictures of deer. User Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages abou snakes but not too long". User Karen wants to hange account password to "CoHoBaSt". User
HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to change account password to "CoHoBaSt". User Amber requests pages

Buffer Overflow example

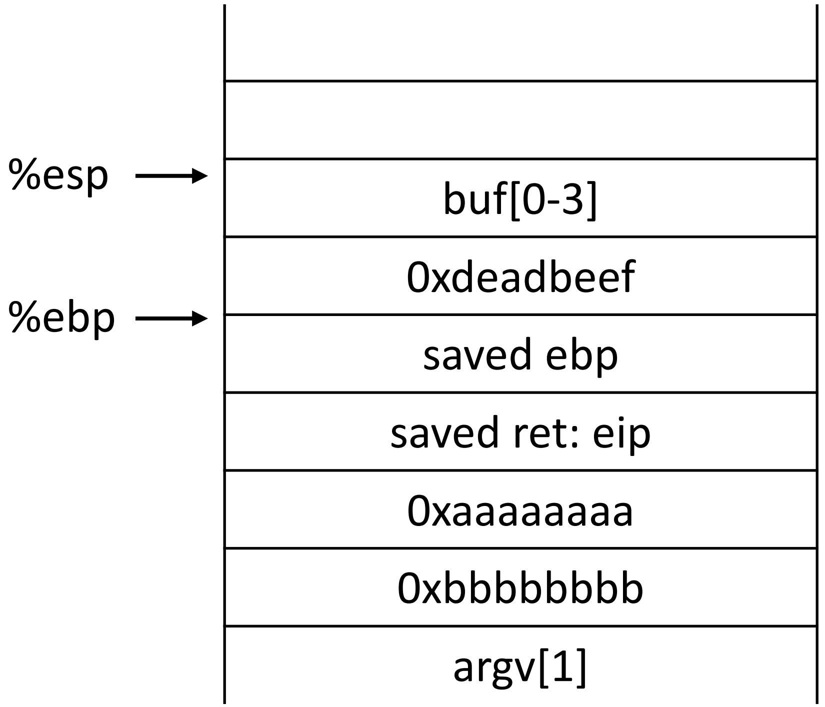


```
#include <stdio.h>
#include <string.h>

void foo() {
  printf("hello all!!\n");
  exit(0);
}

void func(int a, int b, char *str) {
  int c = 0xdeadbeef;
  char buf[4];
  strcpy(buf,str);
}

int main(int argc, char**argv) {
  func(0xaaaaaaaa,0xbbbbbbbb,argv[1]);
  return 0;
}
```

Buffer Overflow example



```
#include <stdio.h>
#include <string.h>

void foo() {
  printf("hello all!!\n");
  exit(0);
}

void func(int a, int b, char *str) {
  int c = 0xdeadbeef;
  char buf[4];
  strcpy(buf,str);
}

int main(int argc, char**argv) {
  func(0xaaaaaaaa,0xbbbbbbbb,argv[1]);
  return 0;
}
```



buf[0-3]

0xdeadbeef

saved ebp

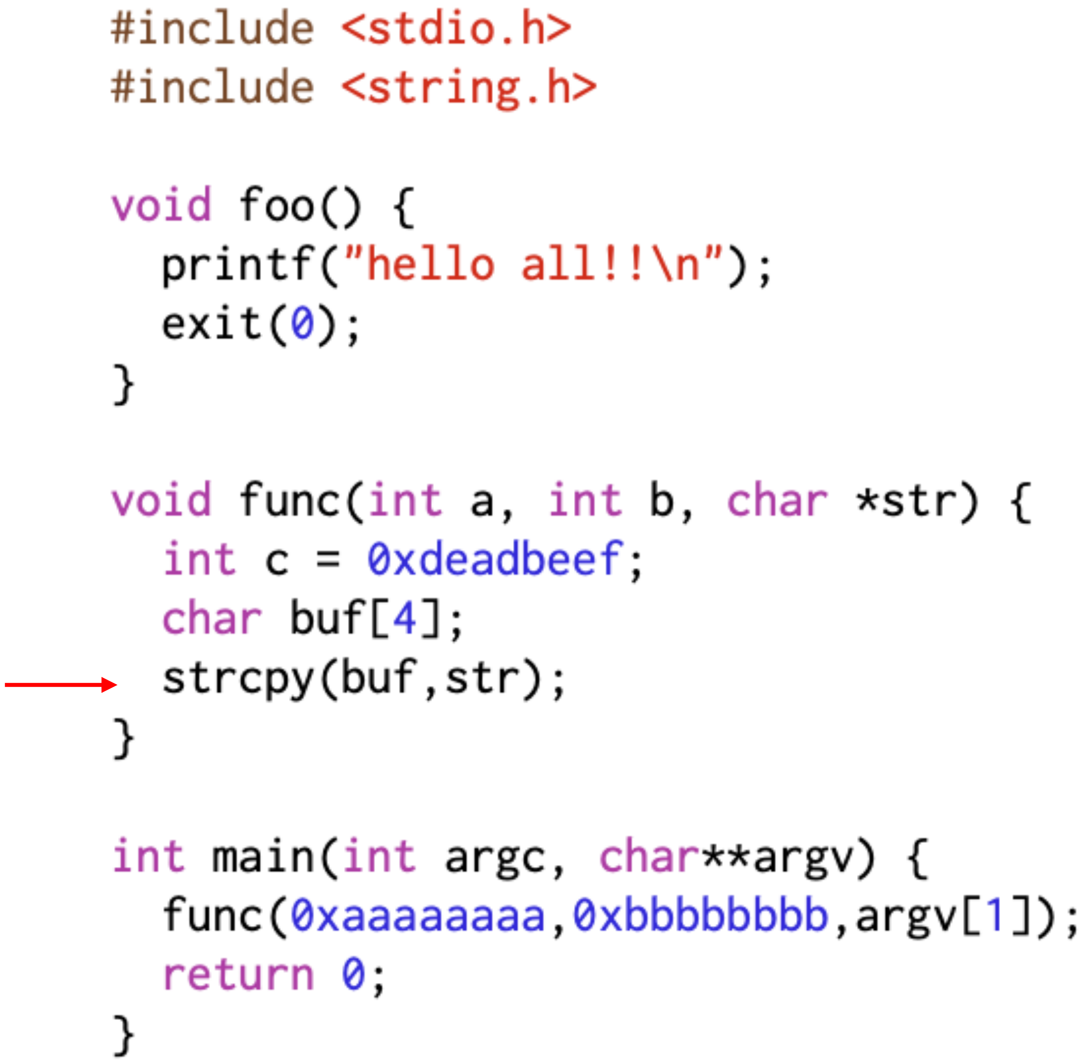
saved ret: eip

0xaaaaaaaa

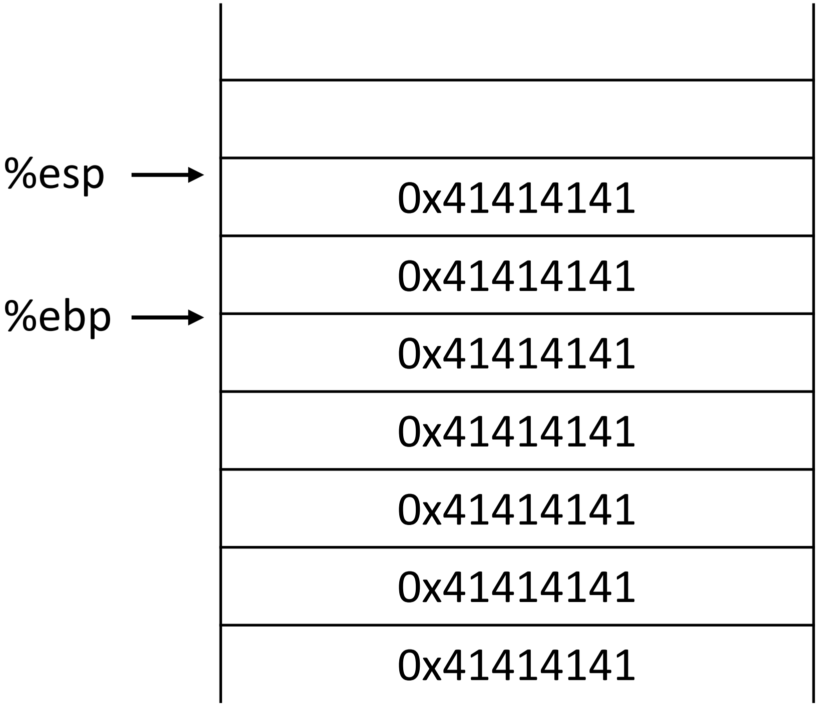
0xbbbbbbbb

argv[1]

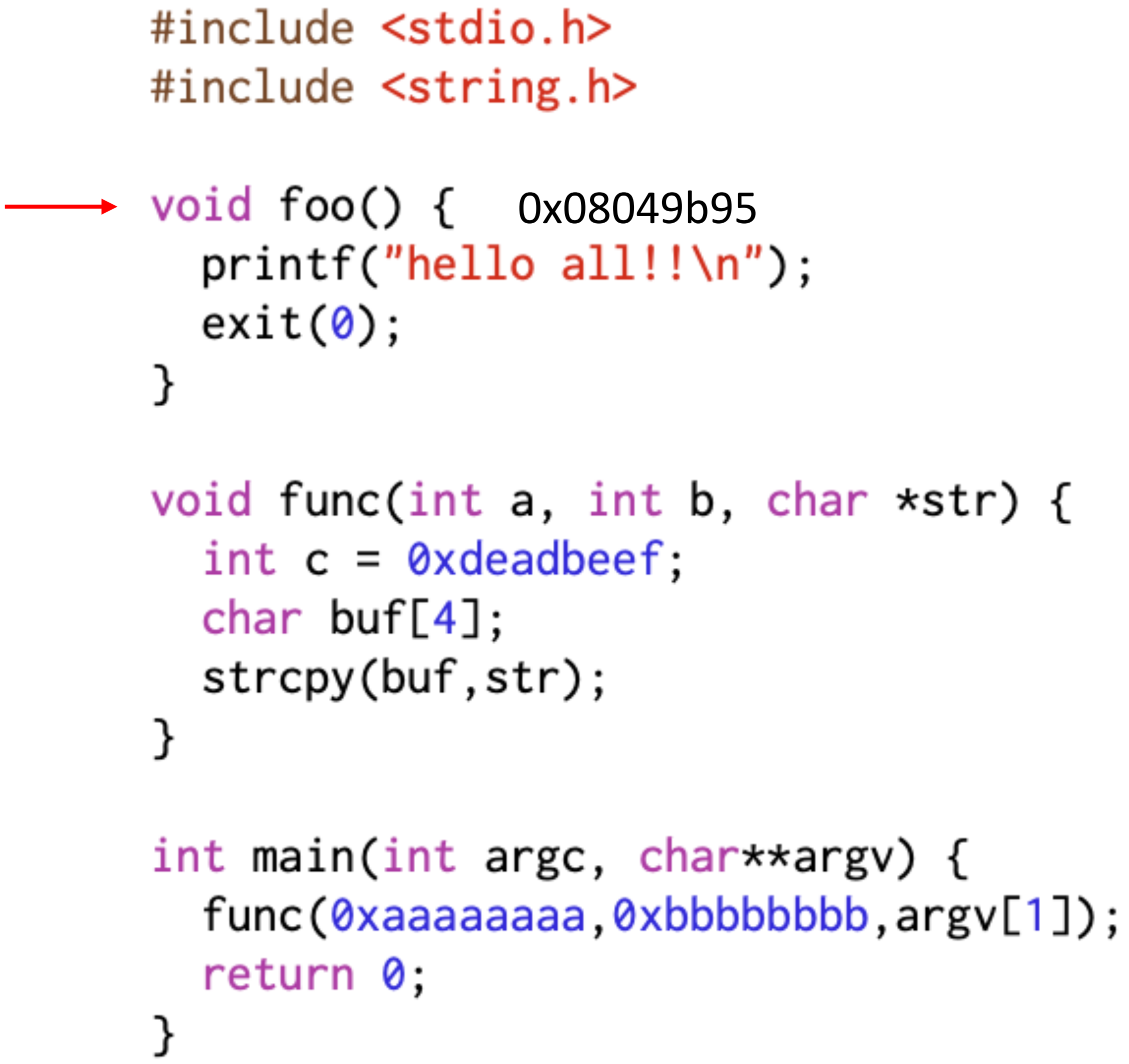
Buffer Overflow example: If the first input is “AAAAAAAAAAAAAAA”



```
#include <stdio.h>
#include <string.h>

void foo() {
  printf("hello all!!\n");
  exit(0);
}

void func(int a, int b, char *str) {
  int c = 0xdeadbeef;
  char buf[4];
  strcpy(buf,str);
}

int main(int argc, char**argv) {
  func(0xaaaaaaaa,0xbbbbbbbb,argv[1]);
  return 0;
}
```

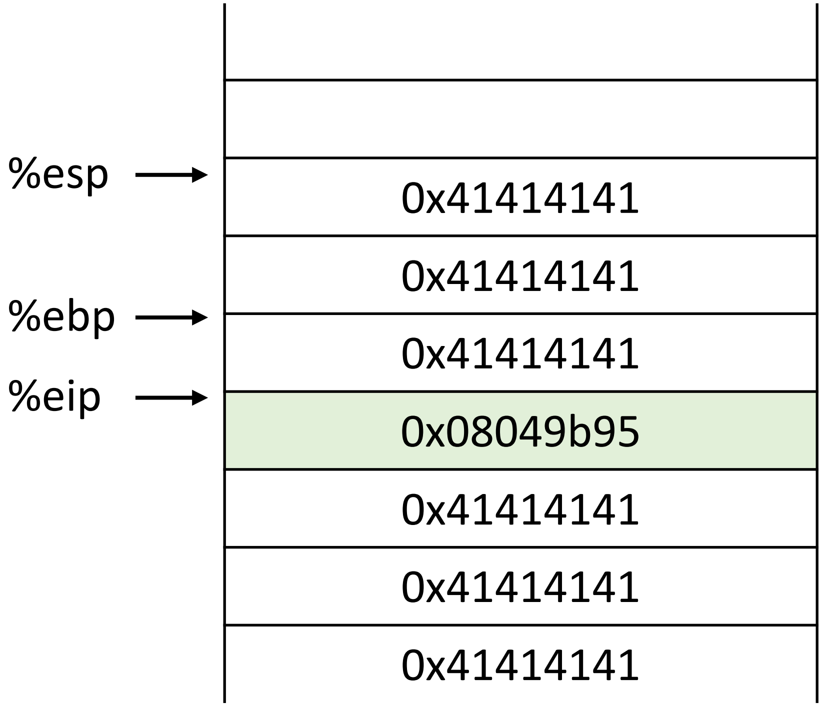


Buffer Overflow example: If the first input is “AAAAAAAAAAAAAAAA”



```
#include <stdio.h>
#include <string.h>

void foo() {
  printf("hello all!!\n");
  exit(0);
}

void func(int a, int b, char *str) {
  int c = 0xdeadbeef;
  char buf[4];
  strcpy(buf,str);
}

int main(int argc, char**argv) {
  func(0xaaaaaaaa,0xbbbbbbbb,argv[1]);
  return 0;
}
```

0x08049b95



%esp → 0x41414141

0x41414141

%ebp → 0x41414141

%eip → 0x08049b95

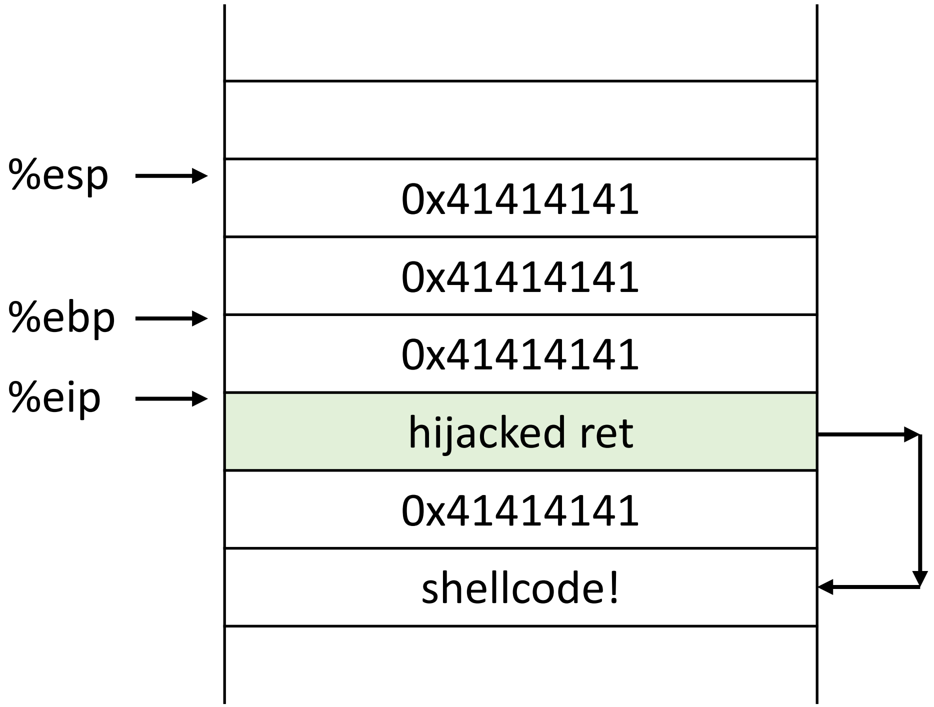
0x41414141

0x41414141

0x41414141

Better Hijacking Control

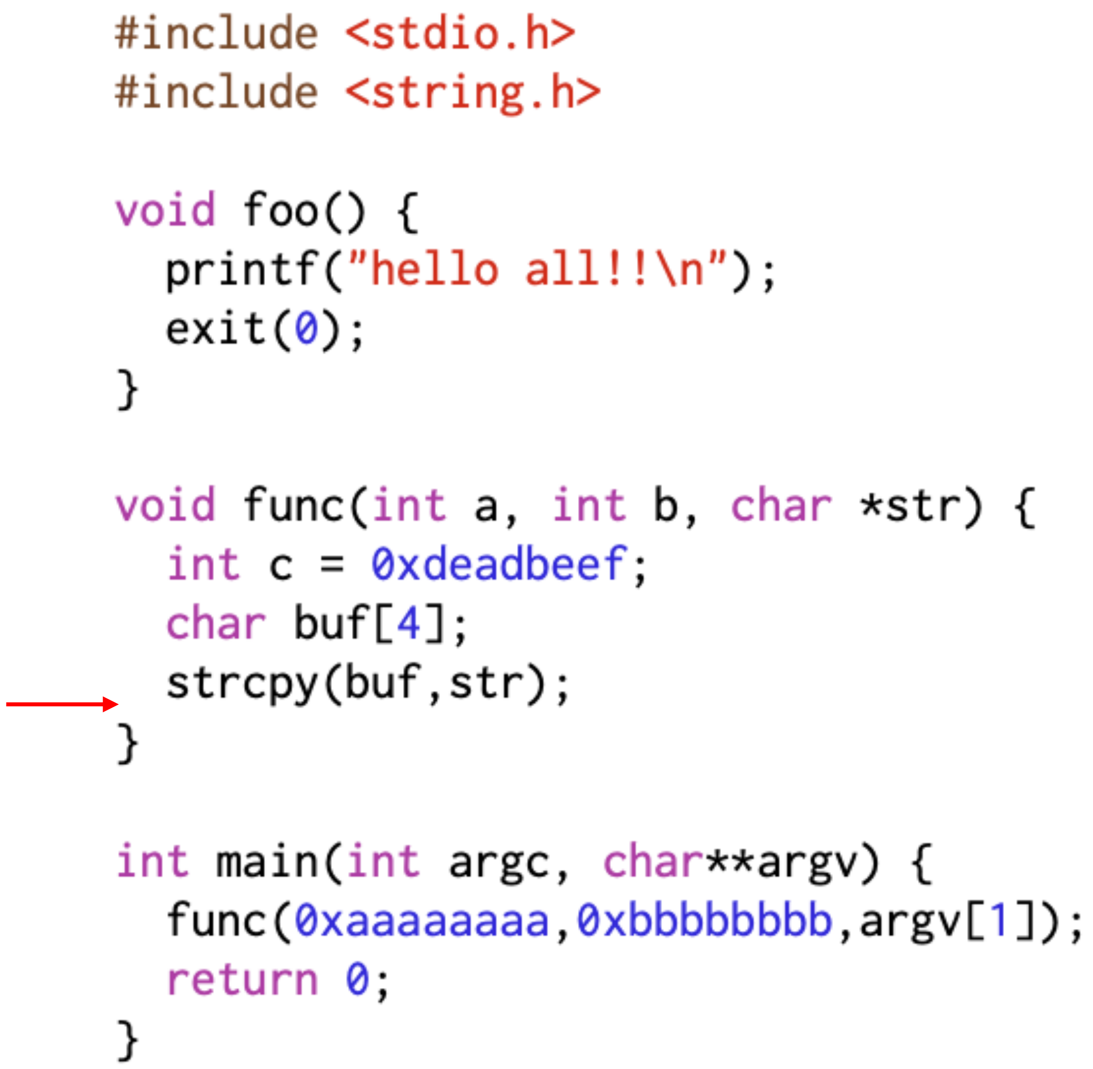
```c
#include <stdio.h>
#include <string.h>

void foo() {
  printf("hello all!!\n");
  exit(0);
}

void func(int a, int b, char *str) {
  int c = 0xdeadbeef;
  char buf[4];
  strcpy(buf,str);
}

int main(int argc, char**argv) {
  func(0xaaaaaaaa,0xbbbbbbbb,argv[1]);
  return 0;
}
```



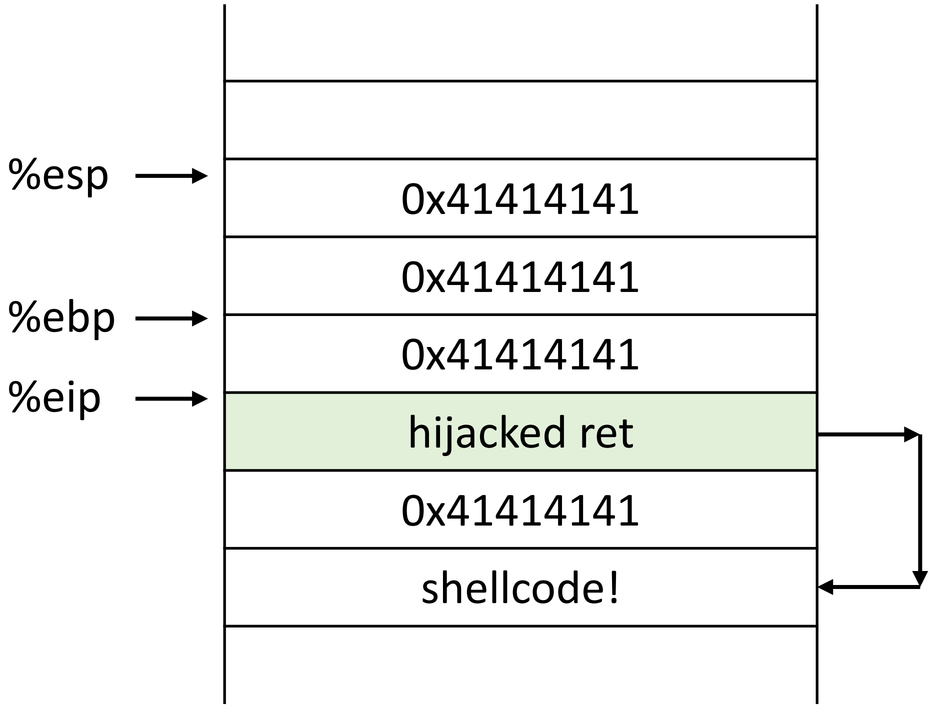
Jump to attacker supplied code where?

- put code in the string
- jump to start of the string

Better Hijacking Control



```
#include <stdio.h>
#include <string.h>

void foo() {
  printf("hello all!!\n");
  exit(0);
}

void func(int a, int b, char *str) {
  int c = 0xdeadbeef;
  char buf[4];
  strcpy(buf,str);
}

int main(int argc, char**argv) {
  func(0xaaaaaaaa,0xbbbbbbbb,argv[1]);
  return 0;
}
```



%esp

%ebp

%eip

0x41414141

0x41414141

0x41414141

hijacked ret

0x41414141

shellcode!

Jump to attacker supplied code where?

- put code in the string
- jump to start of the string