CS 88: Security and Privacy

02: Security Mindset

01-25-2024

SWARTHMORE COLLEGE

Reading Quiz

Announcements

- Please sign the ethics form this week to continue in the course
- Update clicker info + office hours that you can make.
- Update your preferences for the midterm exams.
- Please choose partnerships for Lab 1 (EdStem)

Recap: What is "*Security*"?

Security is about
computing or communicating
in the presence of ***adversaries*.**

Recap: What is "*Security*"?

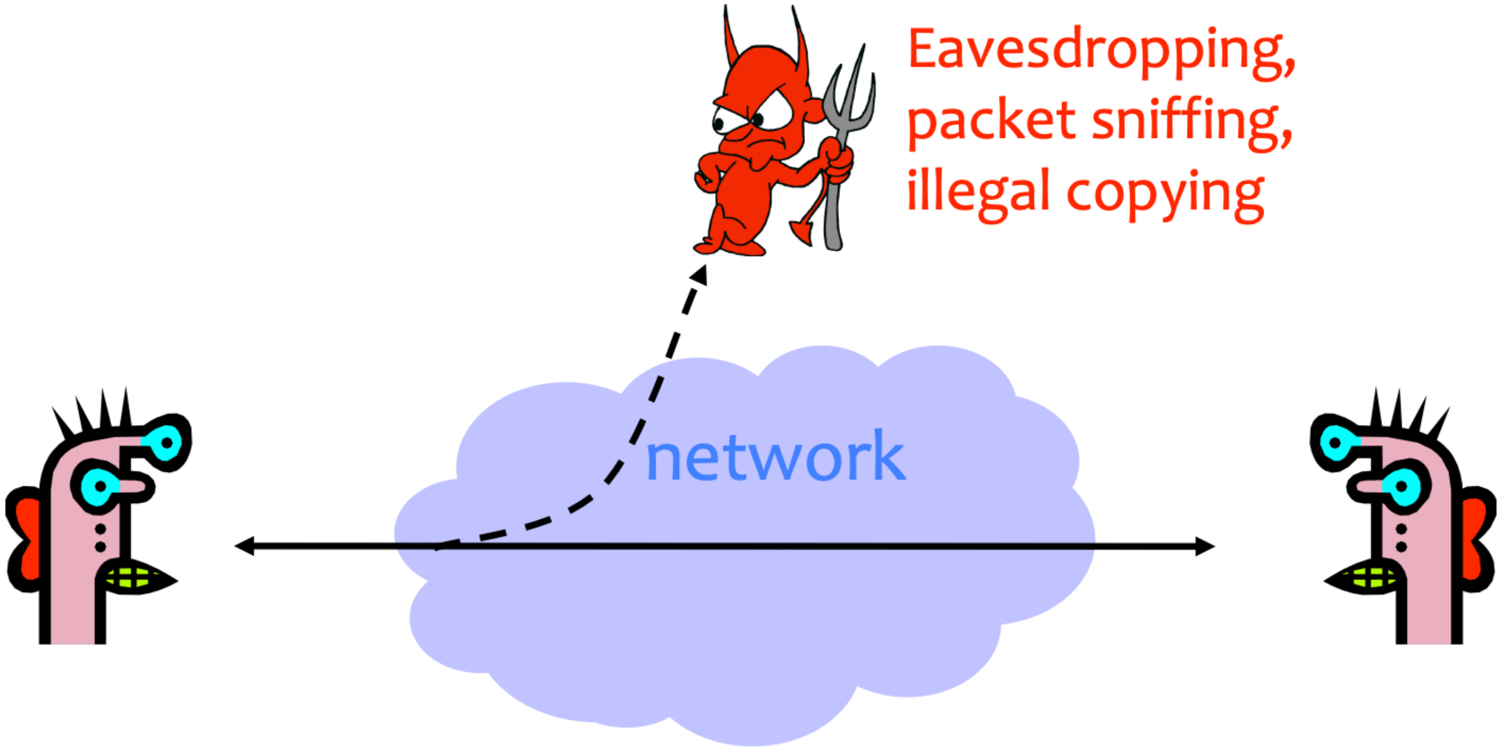
- Normally, we are concerned with the achieving correctness
  - e.g., does this software achieve the desired behavior
- Security is a form of correctness
  - does this software prevent "undesired" behavior?
- Security involves an adversary who is active and malicious
  - Attackers seek to circumvent protective measures

Recap: What is “*Security*”?

- General security goals: “CIA”
  - Confidentiality
  - Integrity
  - Availability

Confidentiality (Privacy)

Confidentiality is concealment of information

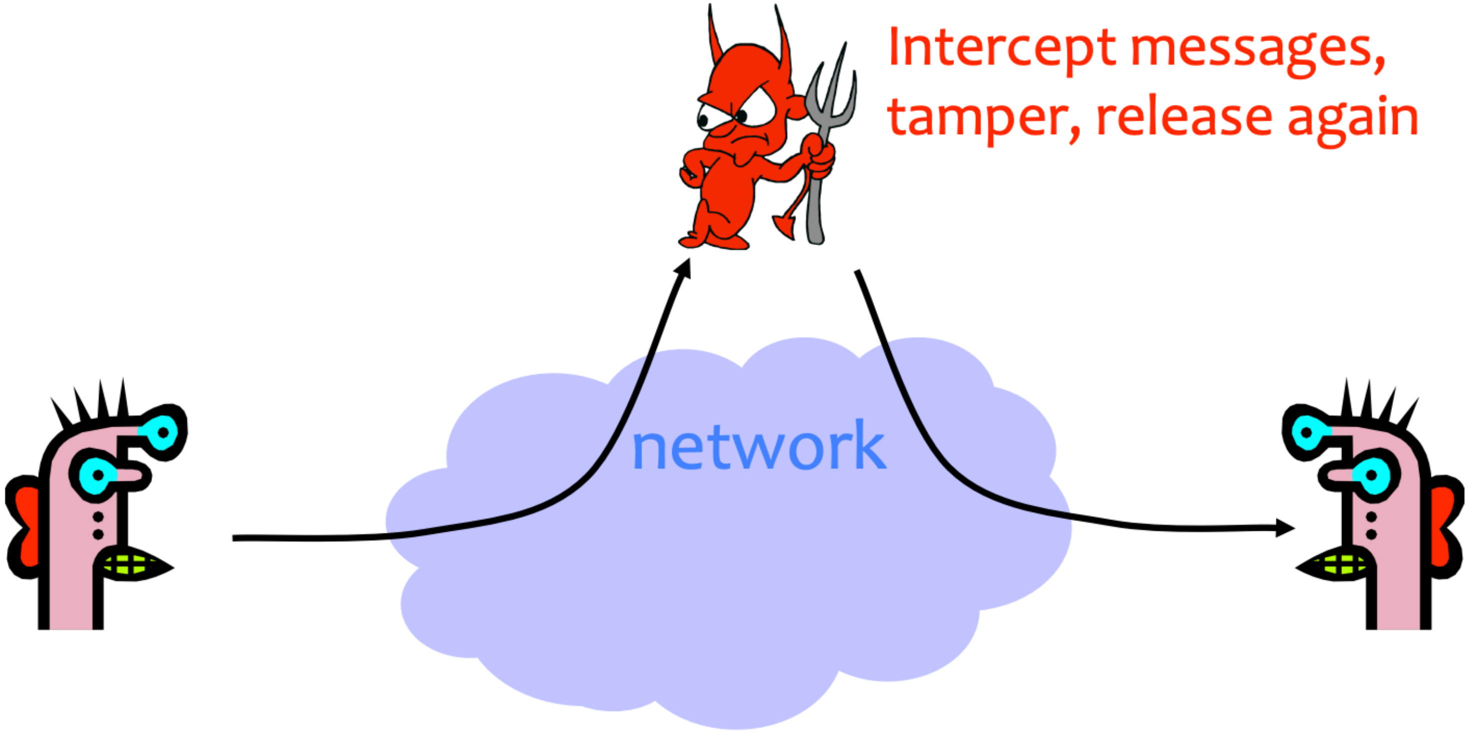


Eavesdropping,
packet sniffing,
illegal copying

Adapted from Franzi Roesner, Yoshi Kohno

Integrity

Integrity is prevention of unauthorized changes

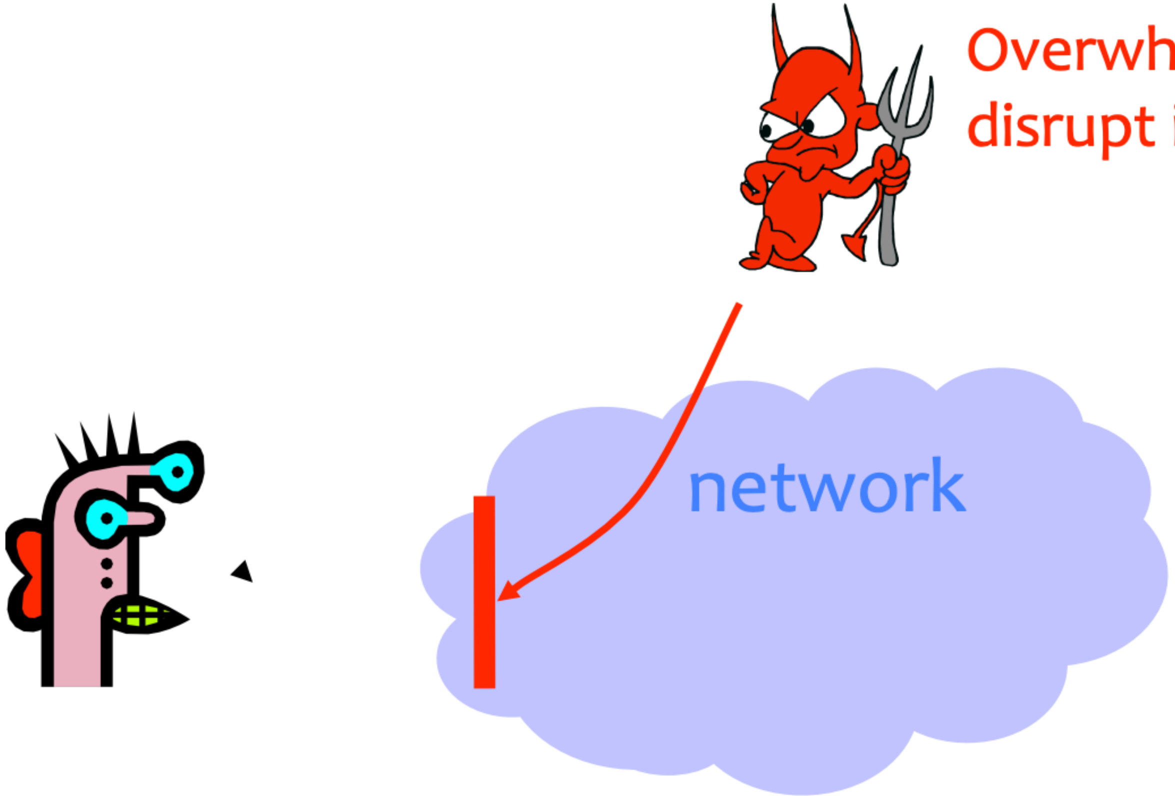


Intercept messages, tamper, release again

Adapted from Franzi Roesner, Yoshi Kohno

Availability

Availability is the ability to use information or resources



Overwhelm or crash servers, disrupt infrastructure

Adapted from Franzi Roesner, Yoshi Kohno

Recap: What is "*Security*"?

General security goals: "CIA"

- Confidentiality
- Integrity
- Availability

- …
- Authenticity
- Accountability and non-repudiation
- Access Control
- Privacy of collected information

Today

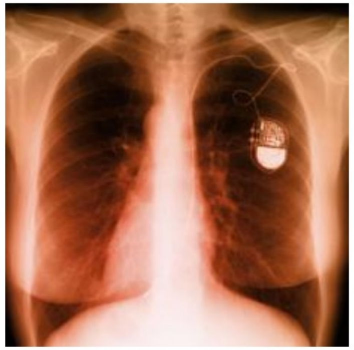
- Security Policy & Mechanism
  - Examples of security attacks
- Design principles of security
- Software Security

Security: System View: not just for computers

smartphones

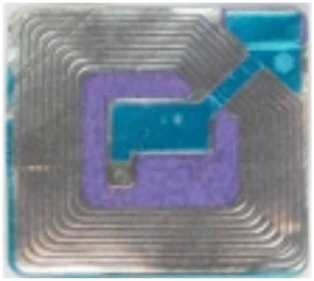
voting machines

EEG headsets



medical devices

wearables



RFID

mobile sensing platforms

cars

game platforms

airplanes

Functionality & Security

- A system normally has a desired functionality: *what ("good") things it should do in the absence of adversaries.*
- The system also normally has a security policy or security objective: *what ("bad") activities or events should be prevented and/or detected?*

Security Policy

Usually stated in terms of

1. Principals – actors or participants (perhaps in terms of their *roles,* including Adversary)
2. Set of *impermissible actions (or states)*
3. Relating to (classes of) objects

Security Mechanism

- AKA “Security Control”
- *Component, technique, or method for (attempting to) achieve or enforce security policy*.

Come up with at least one security policy for each of the following systems

1. Voting in an election
2. Access to /etc/shadow file on Unix Machines
3. Email delivery to Swat Mail users
4. Text messages sent from Alice to Bob

Security Policy is stated as:

1. Principals – actors or participants (perhaps in terms of their *roles,* including Adversary)
2. Set of *impermissible actions (or states)*
3. Relating to (classes of) objects

Come up with security mechanisms for the following systems

1. Voting in an election
2. Access to /etc/shadow file on Unix Machines
3. Email delivery to Swat Mail users
4. Text messages sent from Alice to Bob

Security Mechanism is stated as:

- *Component, technique, or method for (attempting to) achieve or enforce security policy*.

Two types of security mechanisms

- Prevention: keep security policy from being violated.
  - Examples: Fence, password, encryption
- Detection: Detect when security policy is violated.
  - Examples: Motion sensor, tamper-evident seal, storing hash of executable, virus scanner

Goal of Prevention

- to stop the "bad thing" from happening at all
- if prevention works its great
    - E.g. if you write in a memory-safe language (like Python) you are immune from buffer overflow exploits
- if prevention fails, it can fail hard
    - E.g. $68M stolen from a Bitcoin exchange, can’t be reversed

Detection & Recovery

- A *detection mechanism* often comes with an associated *recovery mechanism.*
  - E.g.: Remove intruder, remove virus, load files from backup.
- *Detection* may involve *deterrence:*
  - (Adversary risks being identified & being held accountable for security breach), which may help with *prevention.*

Detection & Response

- Detection: See that something is going wrong
- Response: Do something about it
  - Example: Reverse the harmful actions (restore from backup),
  - prevent future harm (block attacker)
  - Need both — no point in detection without a way to respond and remediate

False Positive and False Negatives

- False positive:
  - You alert when there is nothing there
- False negative:
  - You fail to alert when something is there
- Cost of detection:
  - Responding to false positives is not free, and if there are too many false positives, detector gets removed or ignored
  - False negatives mean a security failure

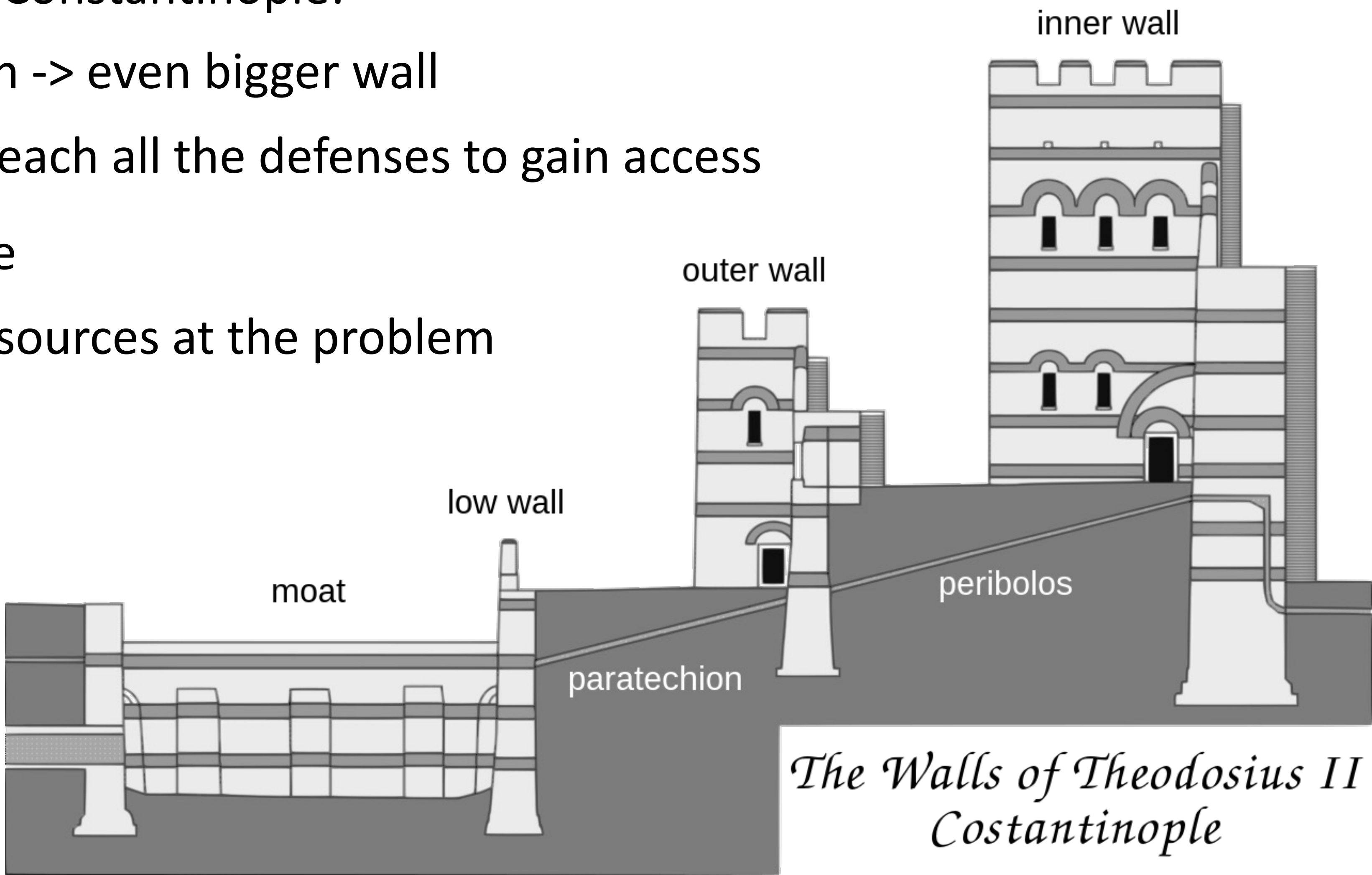
Design Principles of Security

- 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
- Detect if you can’t prevent
- Economics of Added Security (cost-benefit analysis)

-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



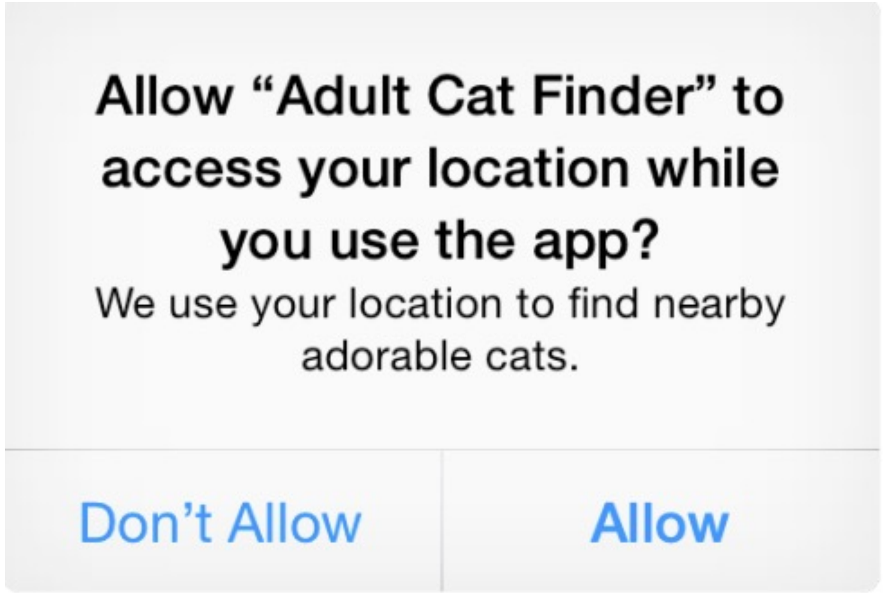
Password authentication

- 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

What principle does this follow? (if any ☺)



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)

Thinking About Least Privilege

- When assessing the security of a system's design, identify the Trusted Computing Base (TCB).
- What components does security rely upon?
- Security requires that the TCB:
  - Is correct
  - Is complete (can't be bypassed)
  - Is itself secure (can't be tampered with)
- Best way to be assured of correctness and its security?
  - KISS = Keep It Simple, Silly!
  - Generally, Simple = Small
- One powerful design approach: privilege separation
  - Isolate privileged operations to as small a component as possible

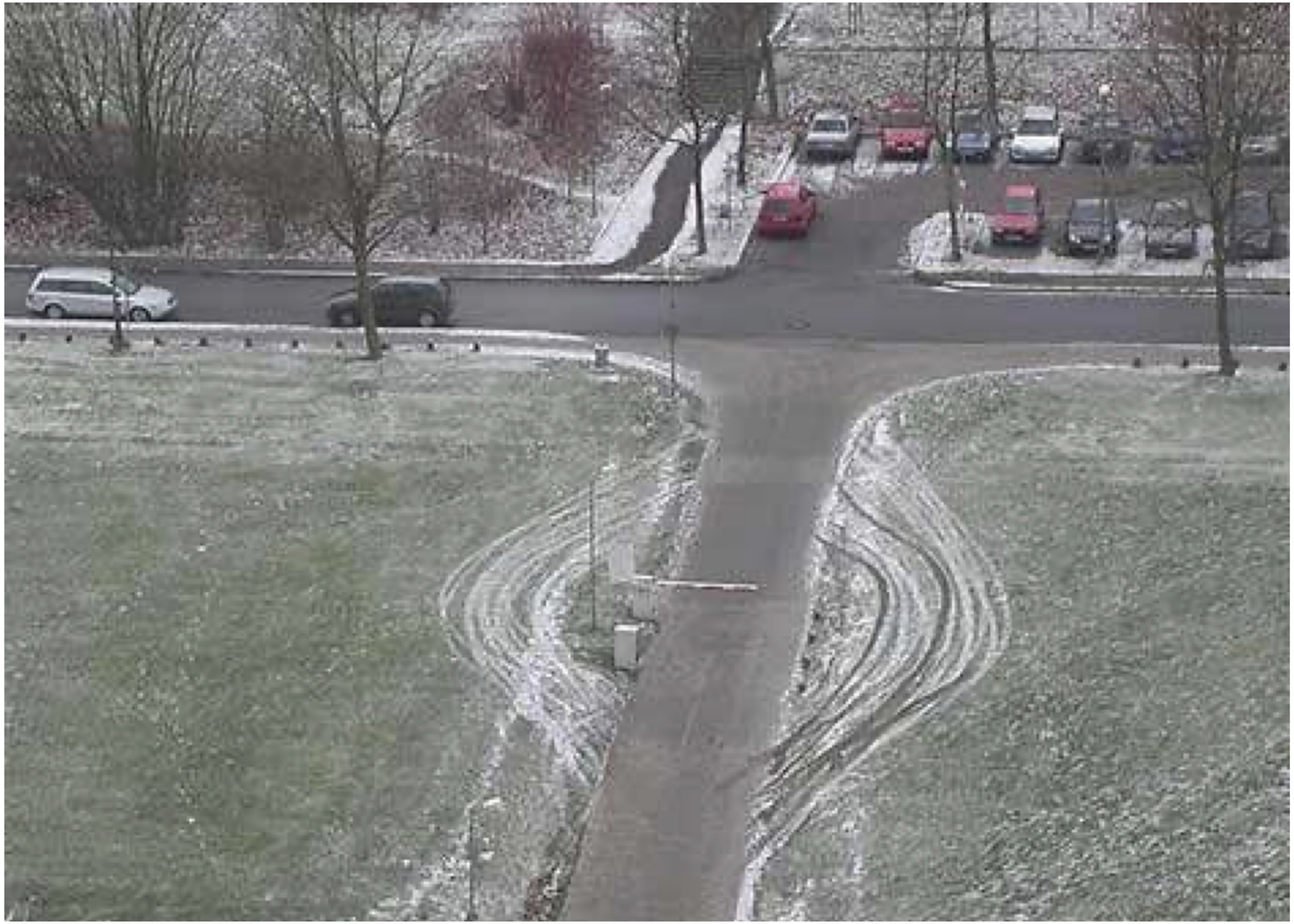
The Base for Isolation: The Operating System

- The operating system provides the following "guarantees"
  - Isolation: A process can not access (read OR write) the memory of any other process
  - Permissions: A process can only change files etc if it has permission to
    - This usually means "Anything that the user can do" in something like Windows or MacOS
    - It can be considerably less in Android or iOS
    - But even in Windows, MacOS, & Linux one can say "I don't want any permissions"

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



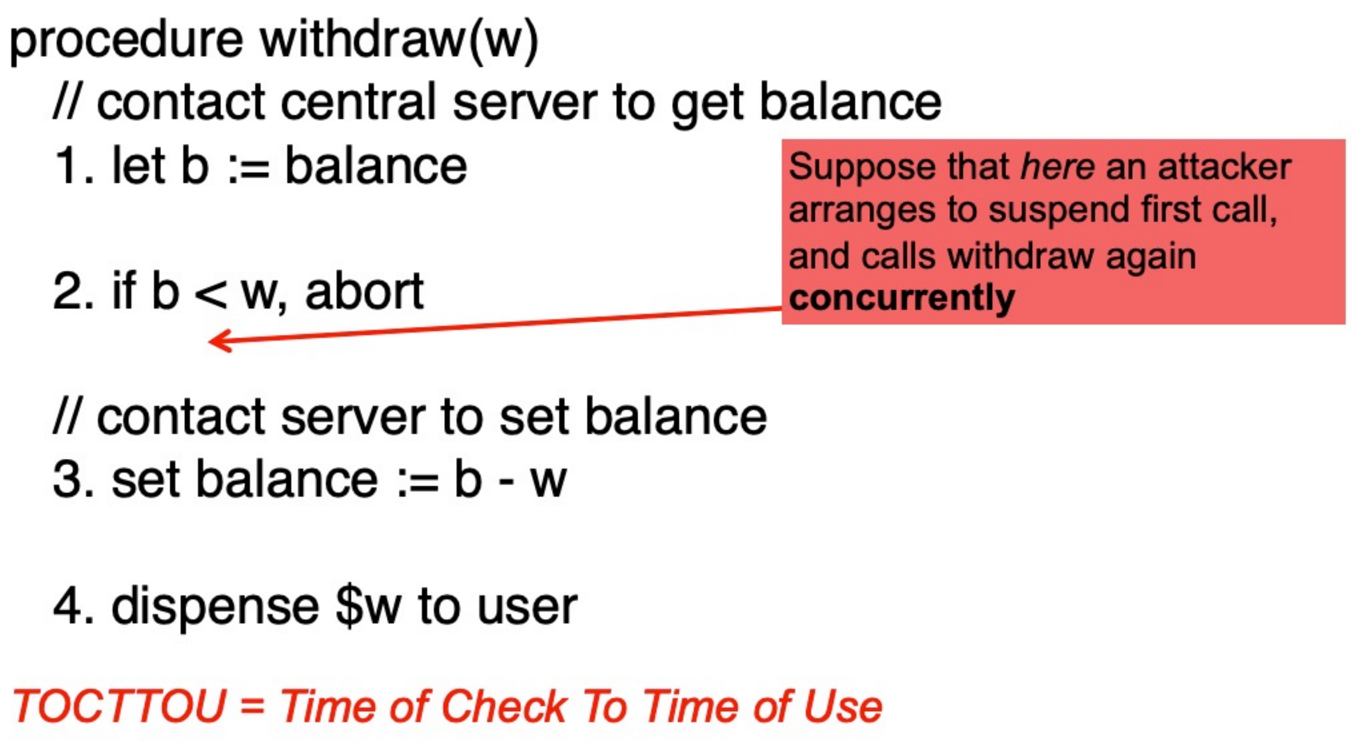
Time of Check to Time of Use Vulnerability: Race Condition

- A common failure of ensuring complete mediation involving race conditions
- Consider the following code:

```
procedure withdrawal(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. give w dollars to user
```

Suppose you have $5 in your account. How can you trick this system into giving you more than $5?

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

Time of Check to Time of Use Vulnerability: Race Condition

- Ethereum is a cryptocurrency which offers "smart" contracts
- Like a digital vending machine:
  - money + snack selection = snack dispensed
- The DAO (Distributed Autonomous Organization) venture capital fund for crypto
  - Participants could vote on "investments" that should be made
  - The DAO supported withdrawals as well

A "Feature" In The Smart Contract

- Code
  - Check the balance,
  - then send the money,
  - then update the balance
- Recursive call :
  - attacker asks the smart contract to give Ether back multiple times before the smart contract could update its balance

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
- Detect if you can't prevent
- Economics of Added Security (cost-benefit analysis)

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

Software Security

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

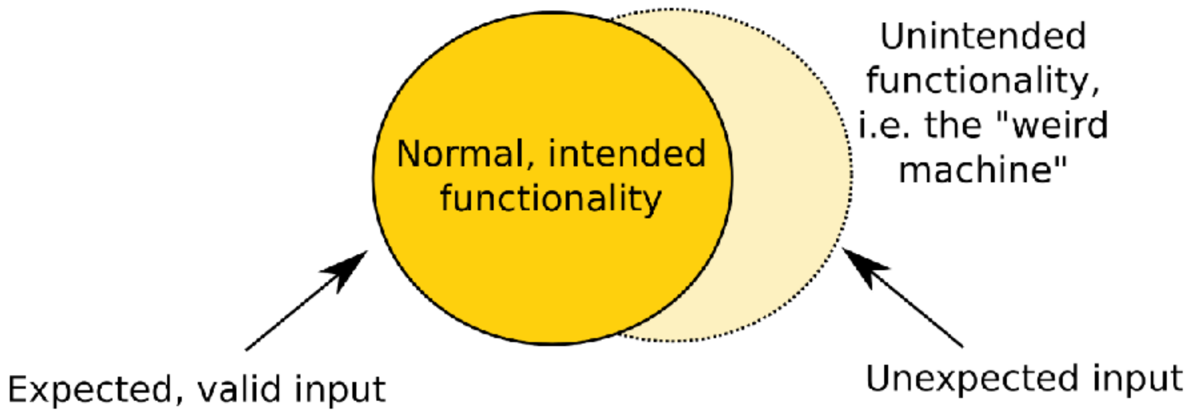
When is a program secure?

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

- is it secure?

Weird machines

- complex systems contain unintended functionality


Normal, intended functionality
Unintended functionality, i.e. the "weird machine"
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.

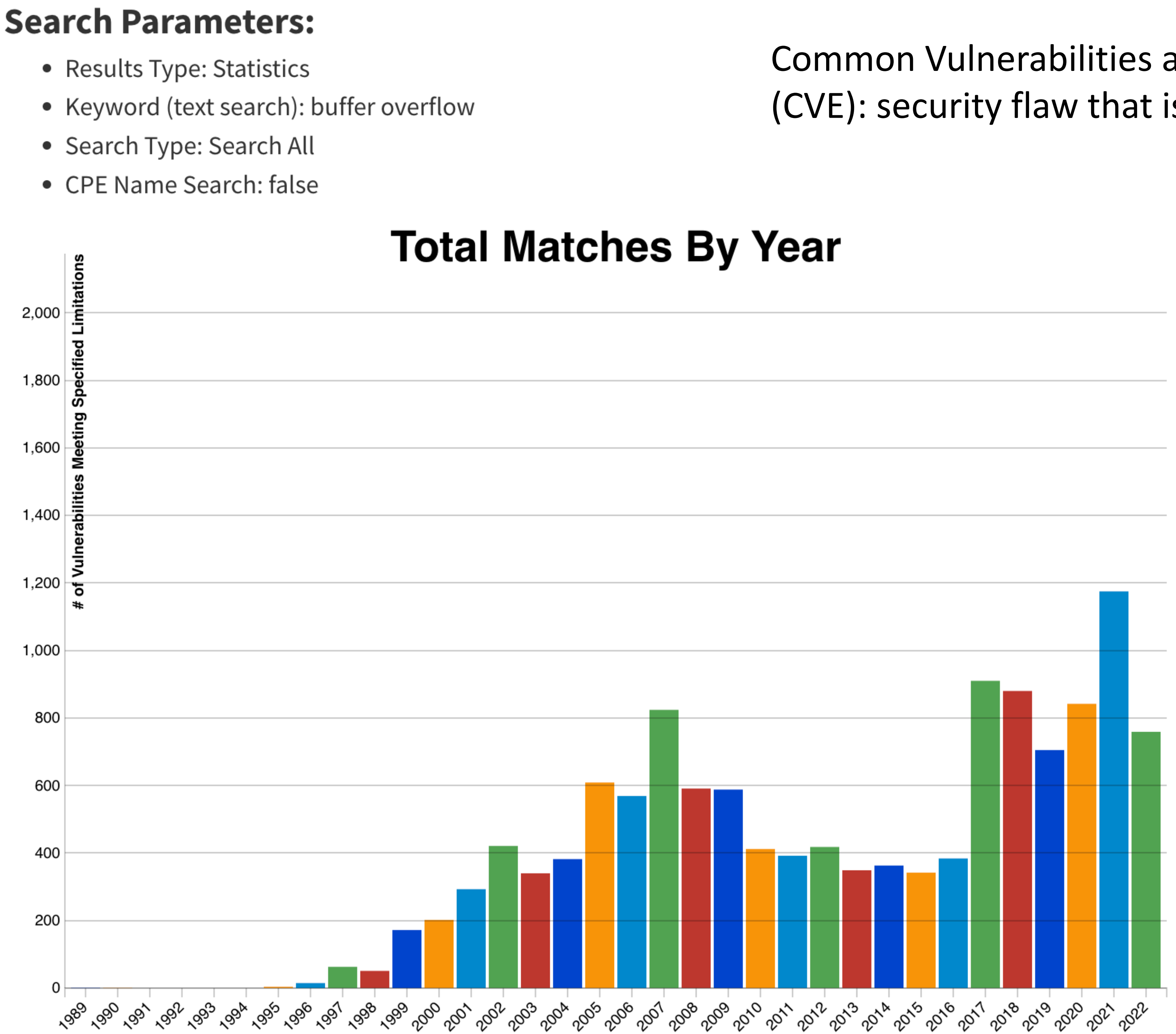
Today: stack buffer overflows

- 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 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

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



Search Parameters:

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

Total Matches By Year

Critical Systems are written in C/C++

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

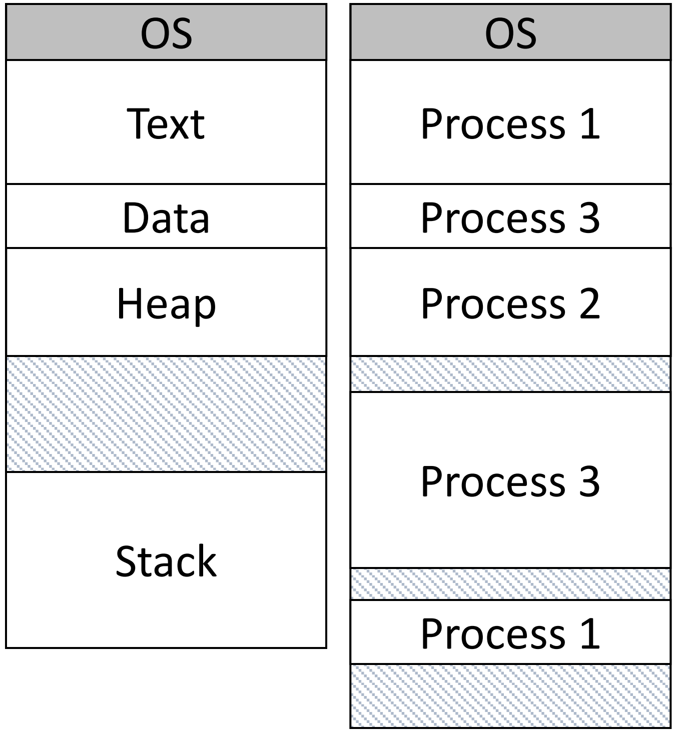
https://nvd.nist.gov/vuln/search

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.

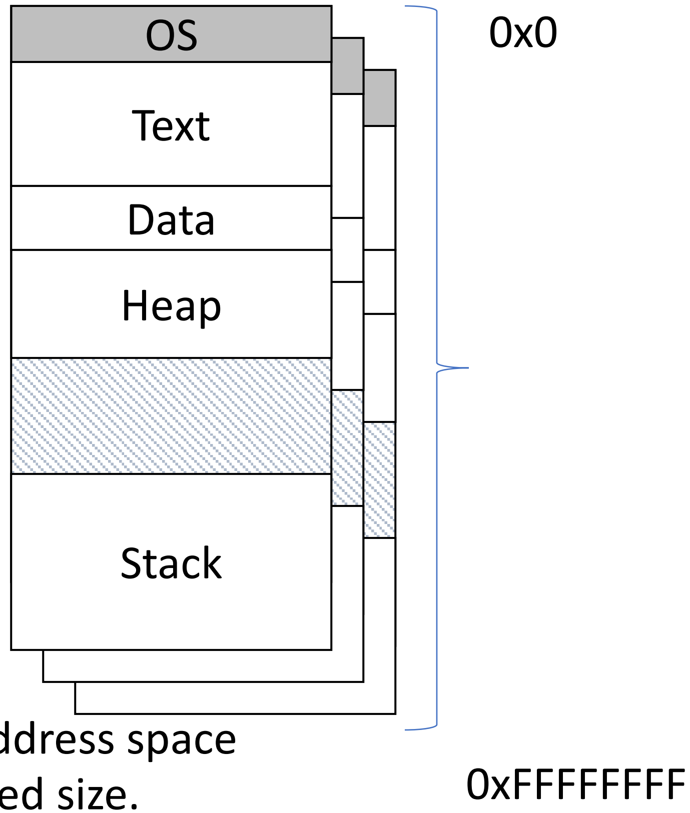
- Reality: there's only so much memory to go around, and no two processes should use the same (physical) memory addresses.



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.

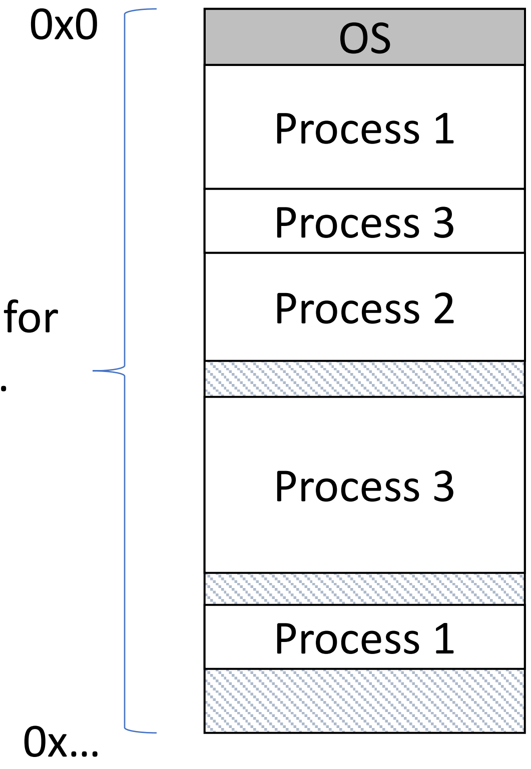


Virtual address space (VAS): fixed size.

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

The set of available storage locations.

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



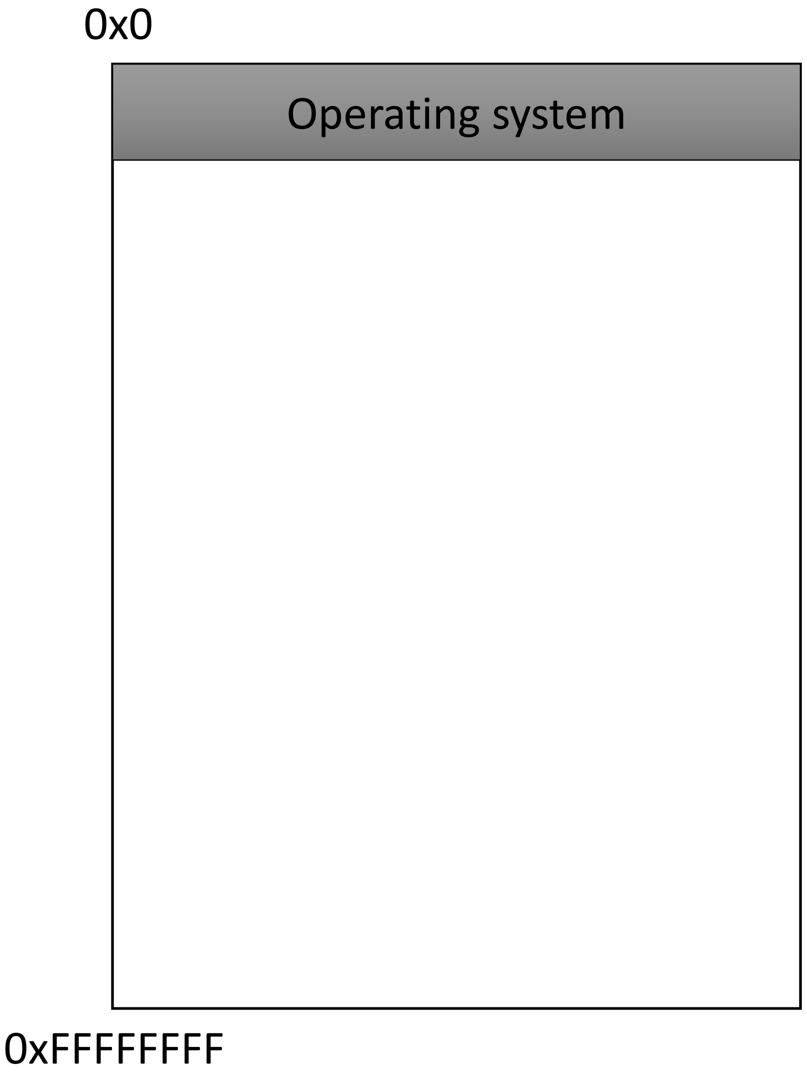
(Determined by amount of installed RAM.)

Memory

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

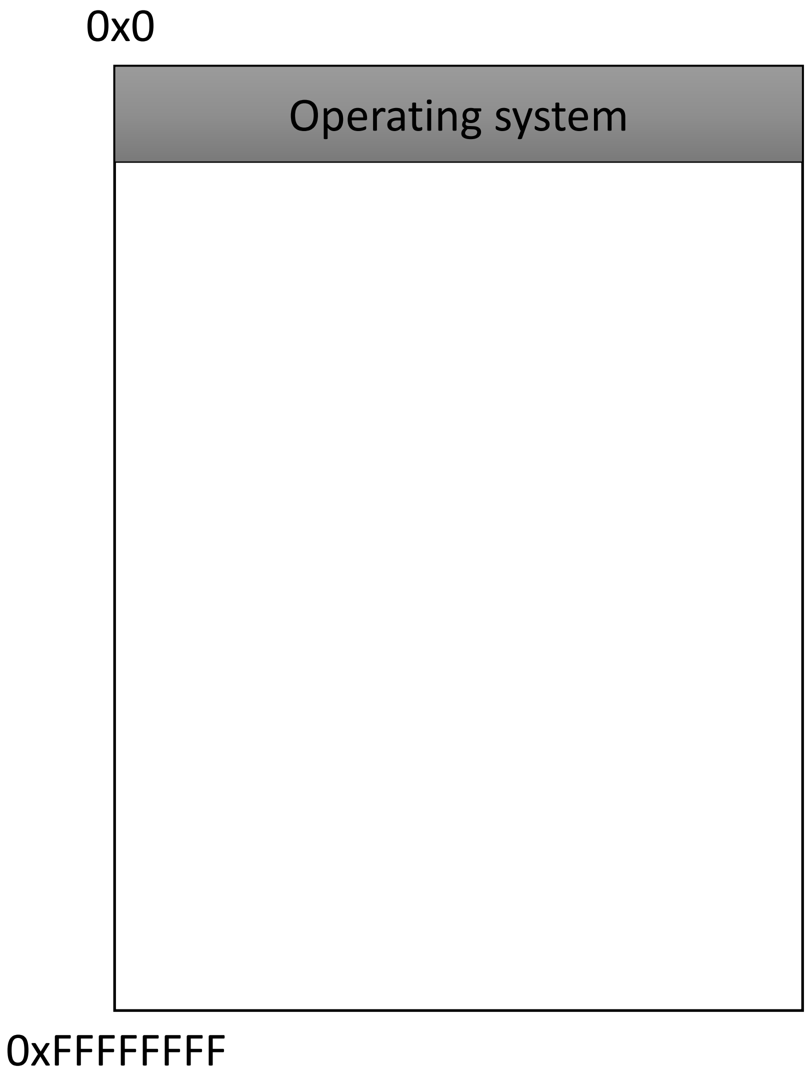
- By convention, we divide it into regions.

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



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.

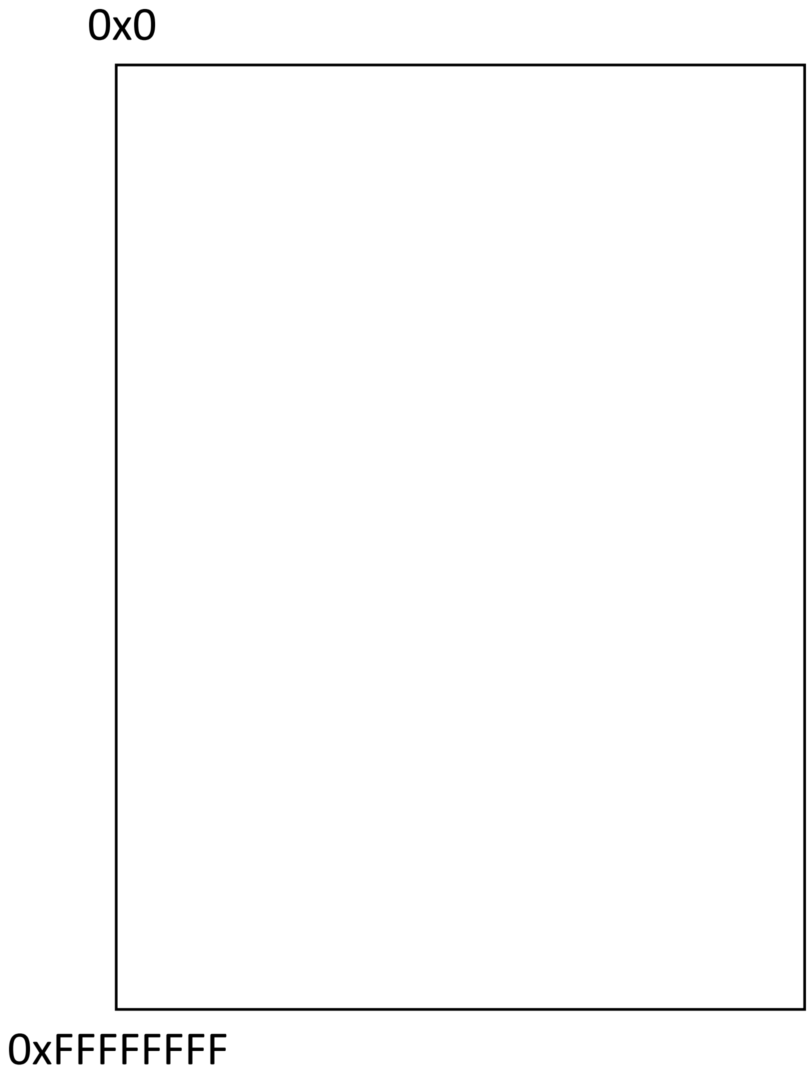


What happens if we launch an attack where we load an instruction to execute at 0x0

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

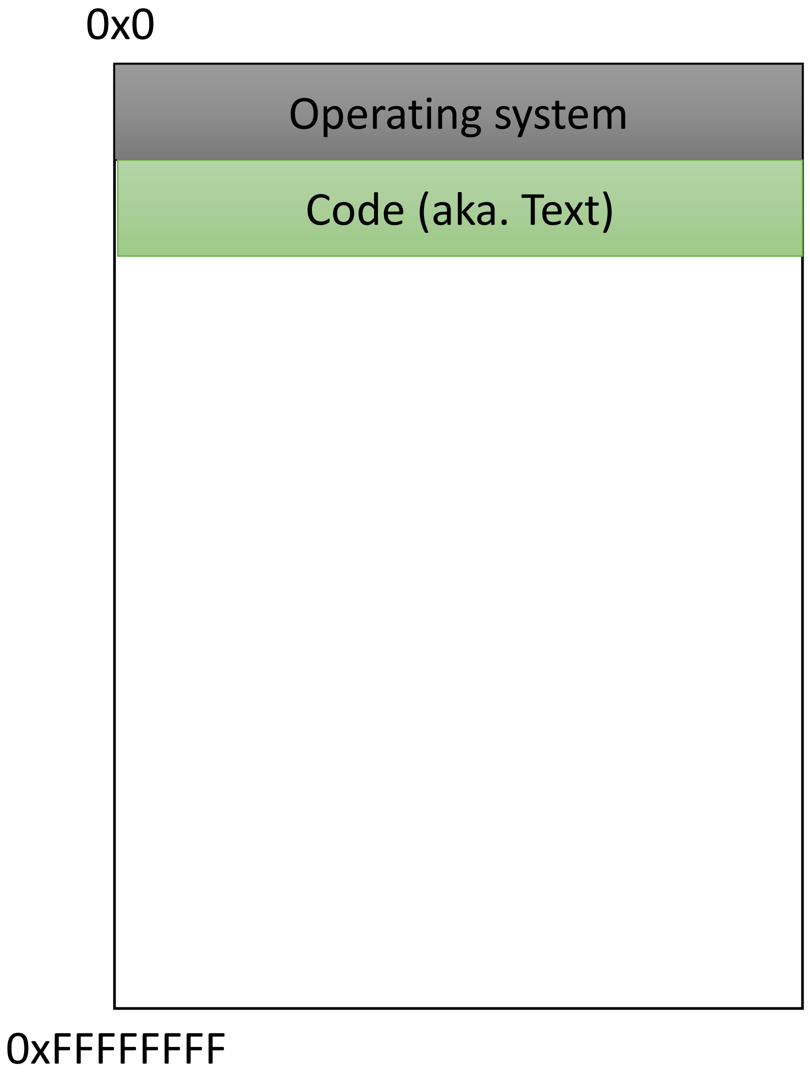
C. This will have a devastating effect.



Memory - Text

- After the OS, we store the program's code.

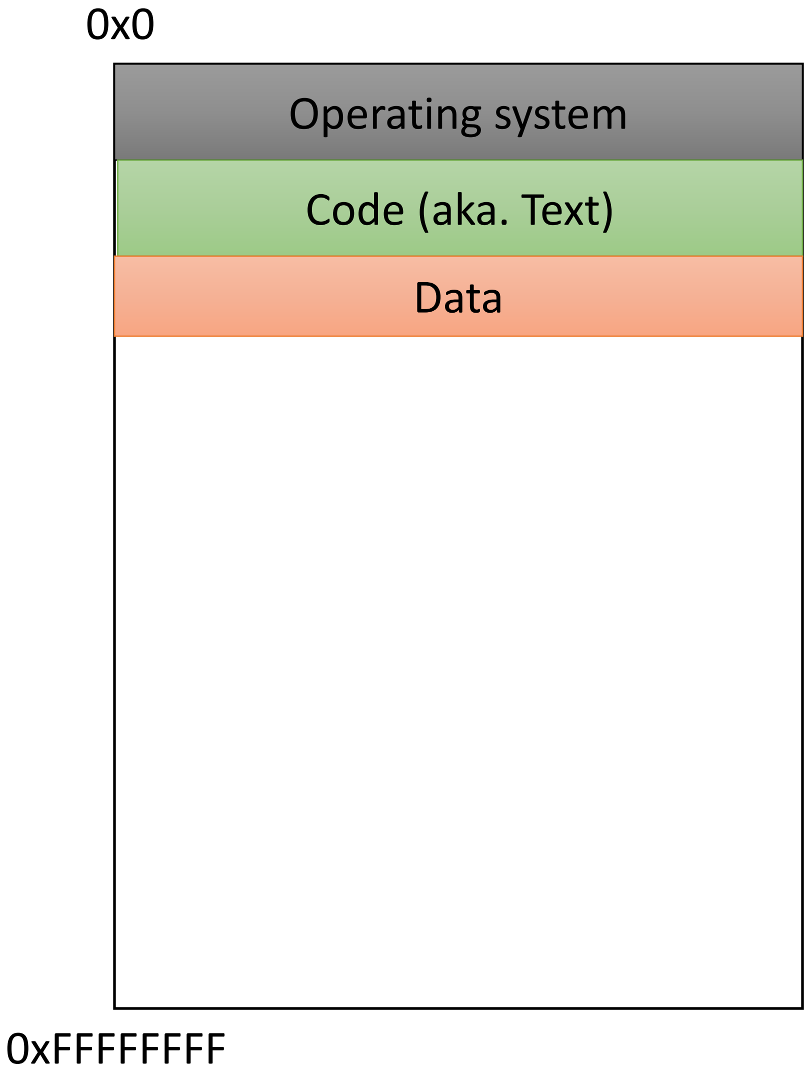
- Instructions generated by the compiler.



Memory – (Static) Data

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

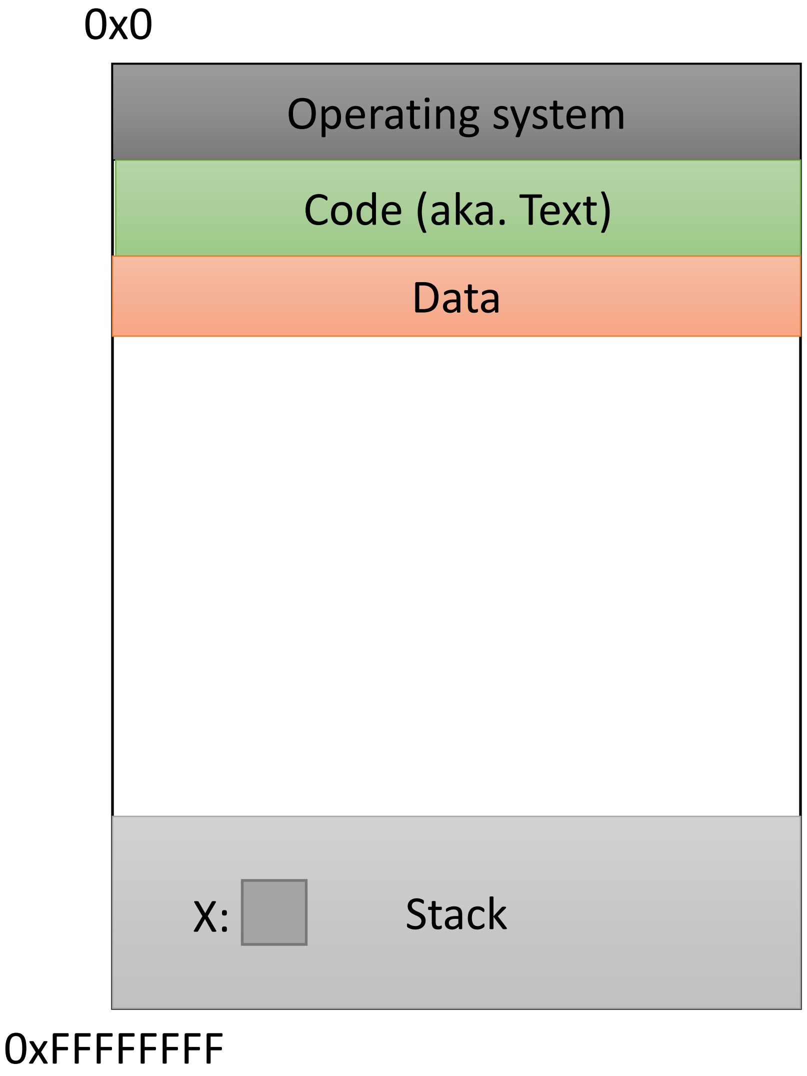
- This stores static variables that are known at compile time.
  - Global variables



Memory - Stack

- At high addresses, we keep the stack.

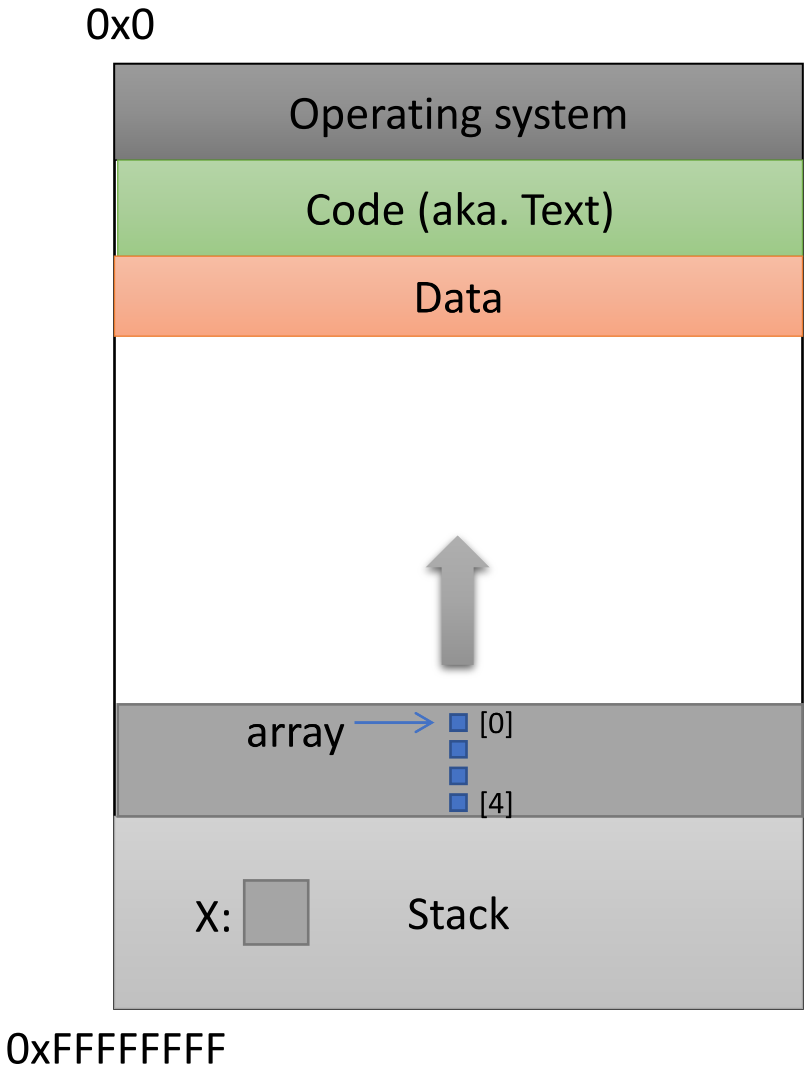
- This stores local (automatic) variables.
  - The kind we've been using in C so far.
  - e.g., int x;



Memory - Stack

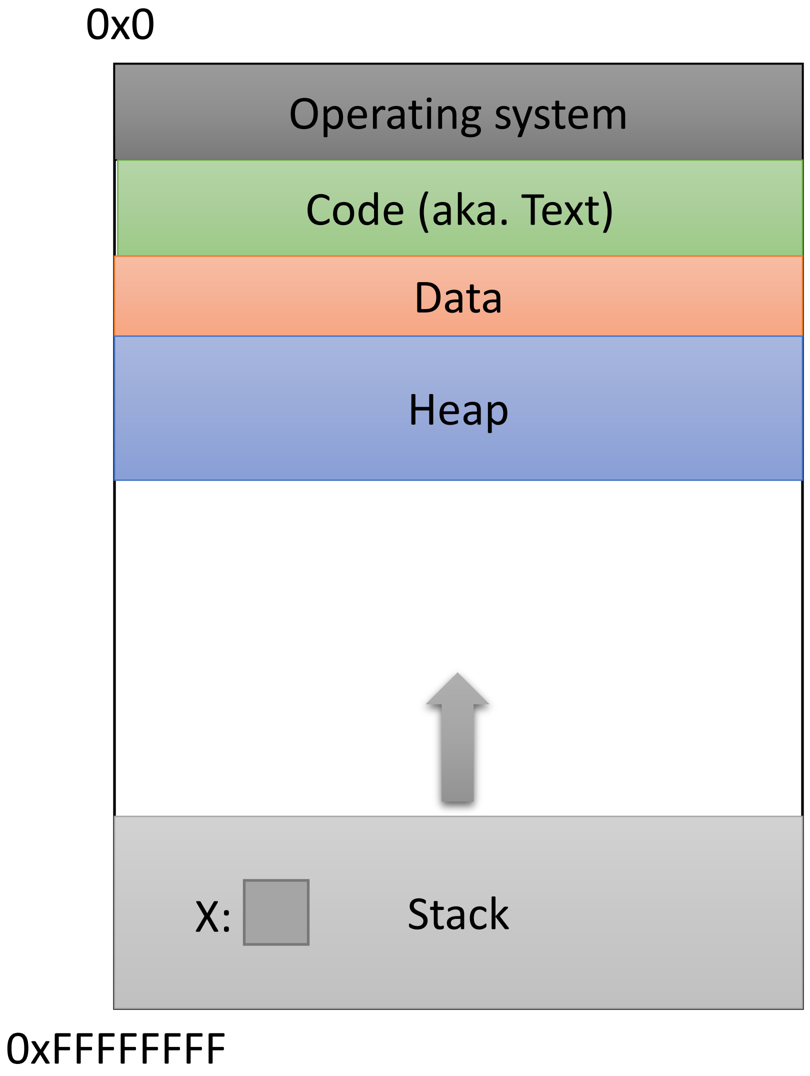
- The stack grows upwards towards lower addresses (negative direction).

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

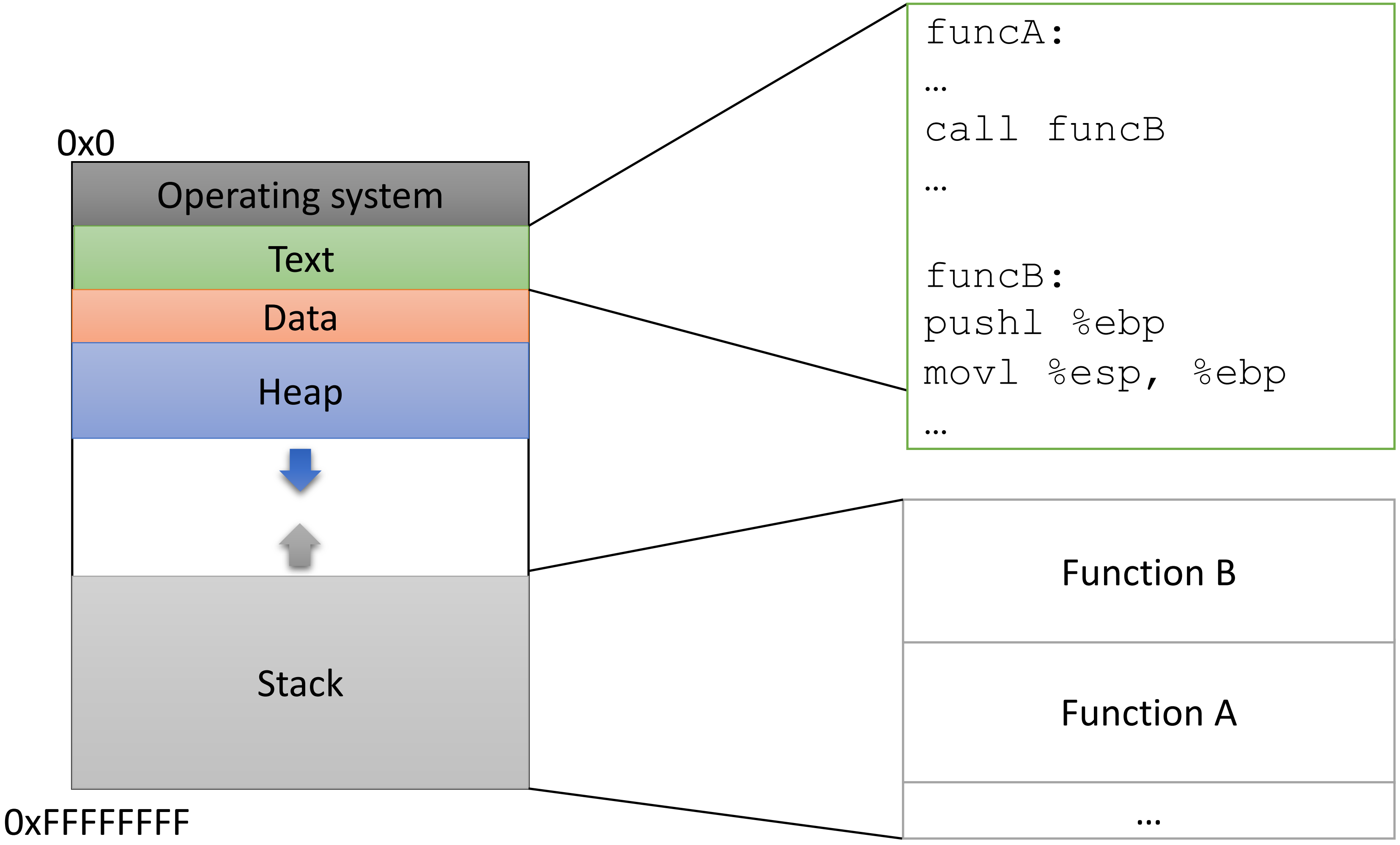


Memory - Heap

- 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

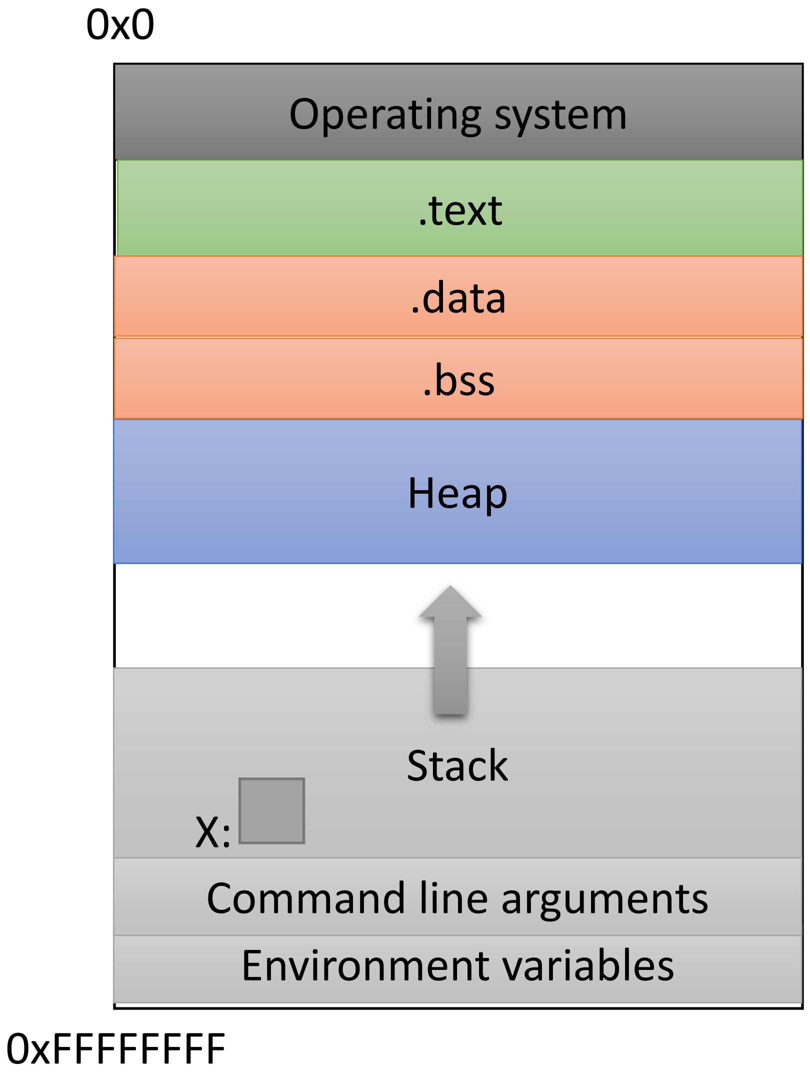
– Local variables

– 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

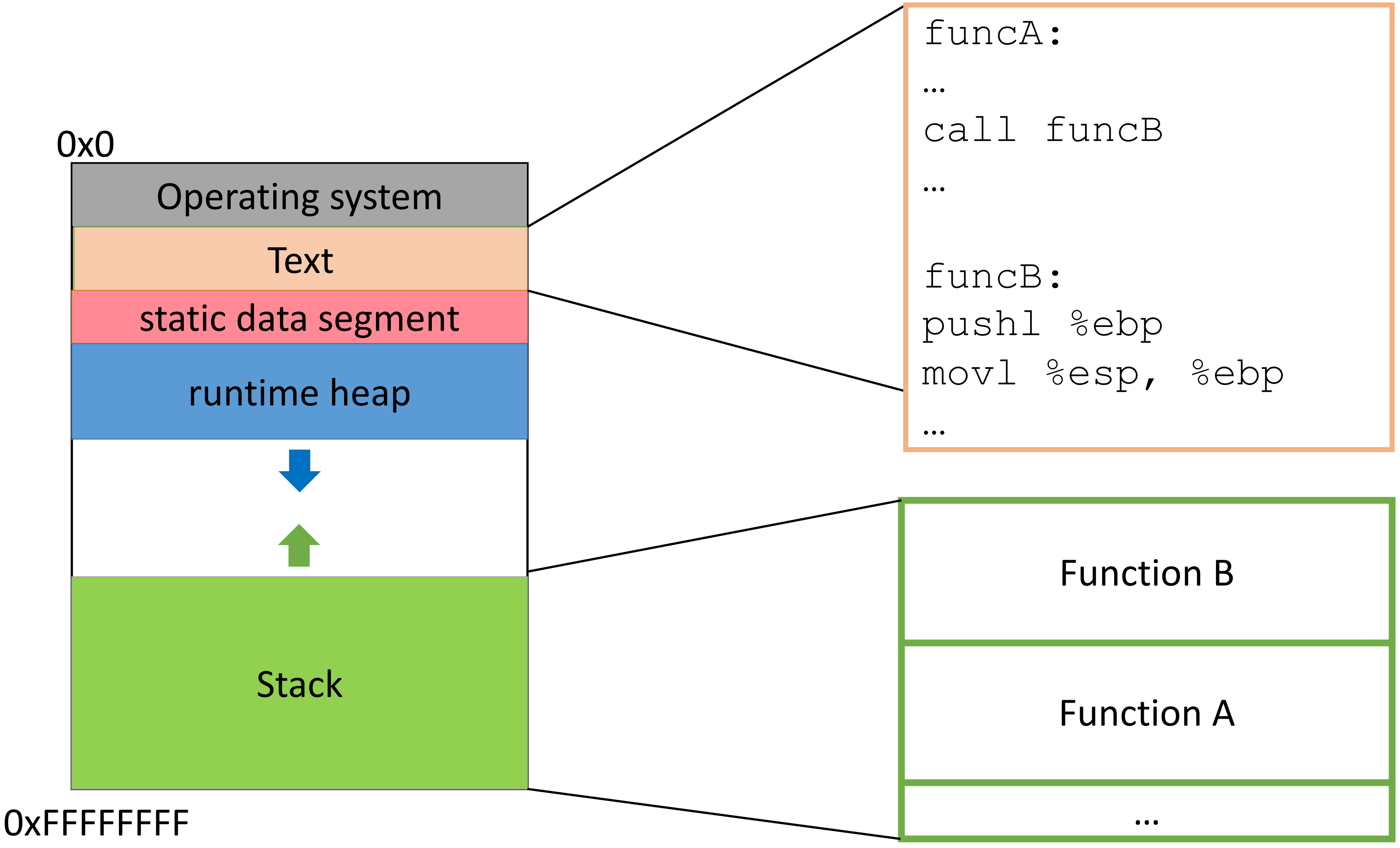
```
int i = 0;
int main()
{
   char *ptr = malloc(sizeof(int));
   char buf[1024]
   int j;
   static int y;
}
```

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

X86: The De Facto Standard

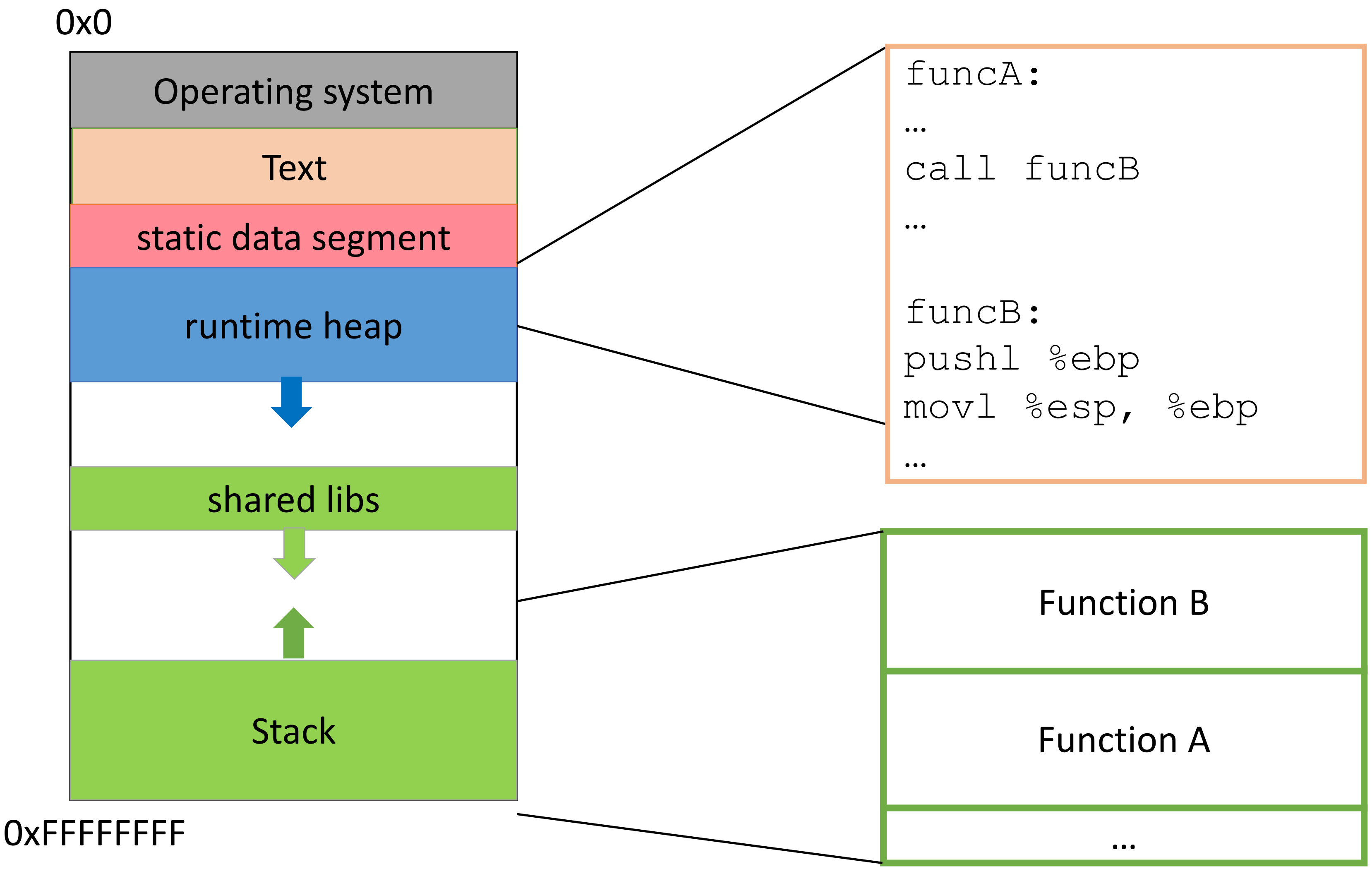
- Extremely popular for desktop computers
- Alternatives
  - ARM: popular on mobile
  - MIPS: very simple
  - Itanium: ahead of its time
- CISC
  - 100 distinct opcodes
- Register poor
  - 8 registers of 32 bits
  - only 6 general purpose
- instructions are variable length
  - not aligned at 4 byte boundaries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

Recall: Instructions in Memory



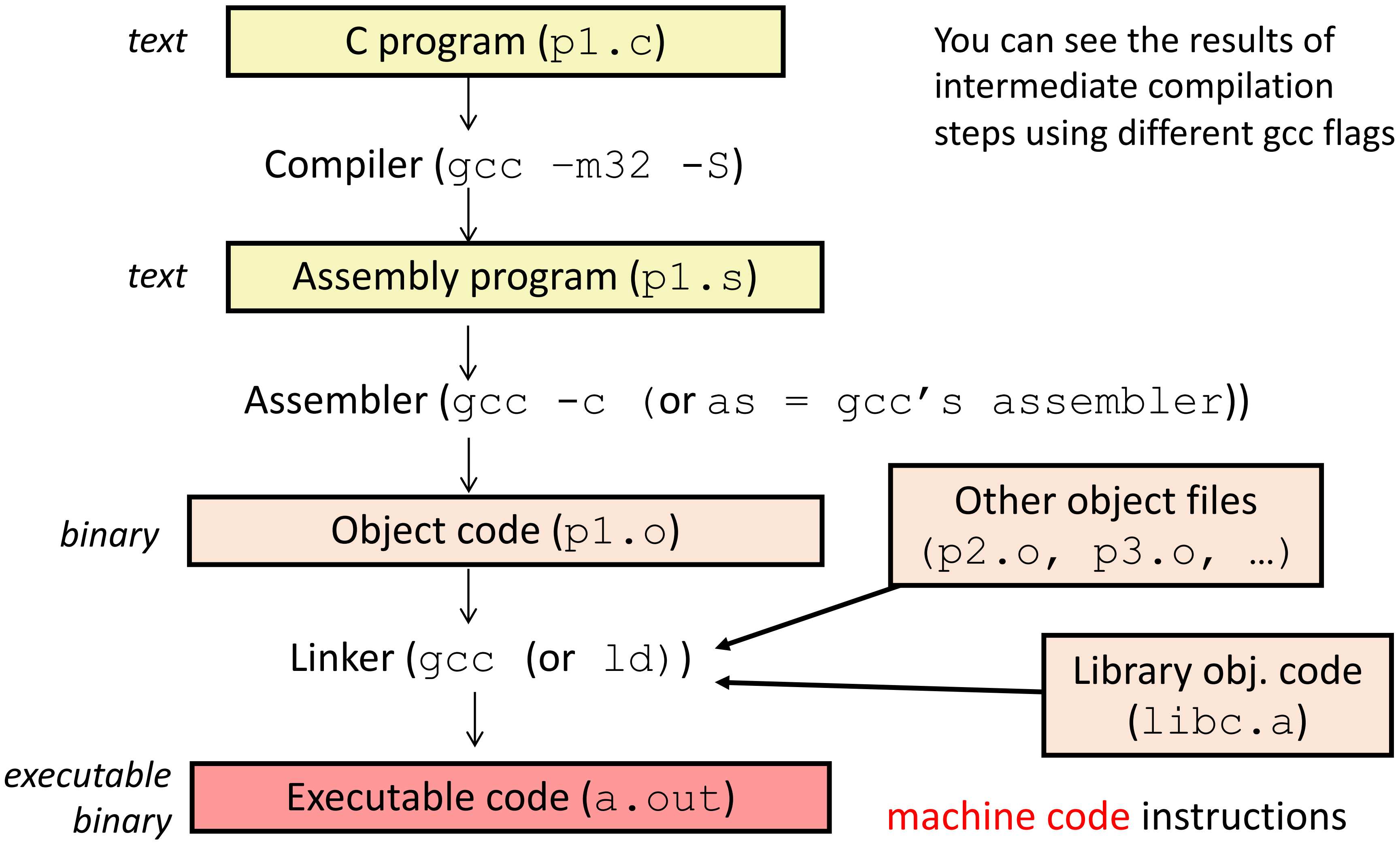
```
funcA:
…
call funcB
…

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

Recall: Instructions in Memory



```
funcA:
…
call funcB
…

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

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

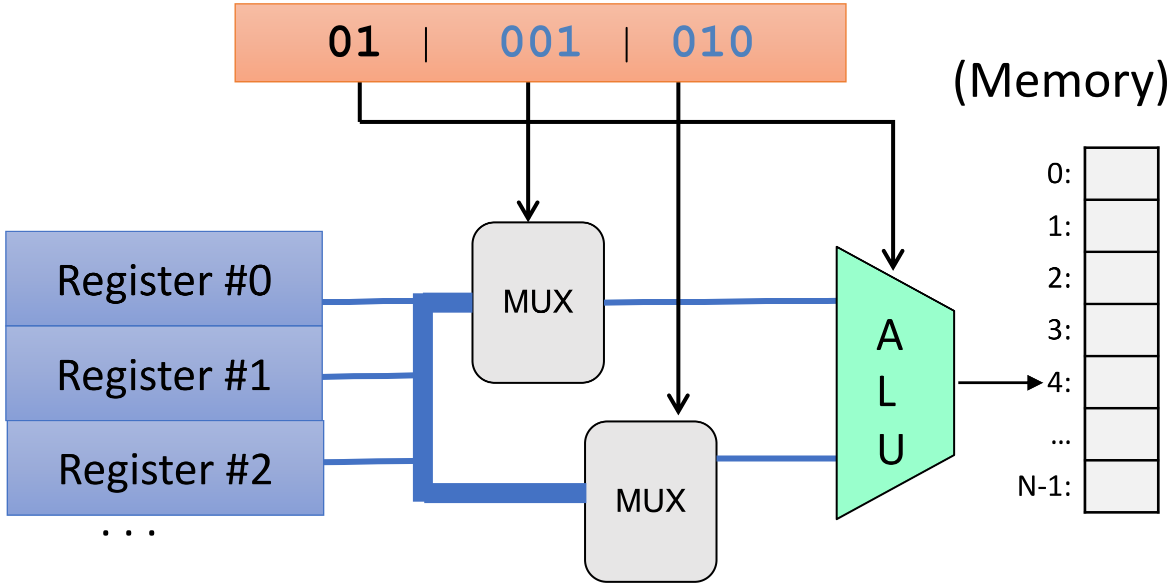
Machine Code

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

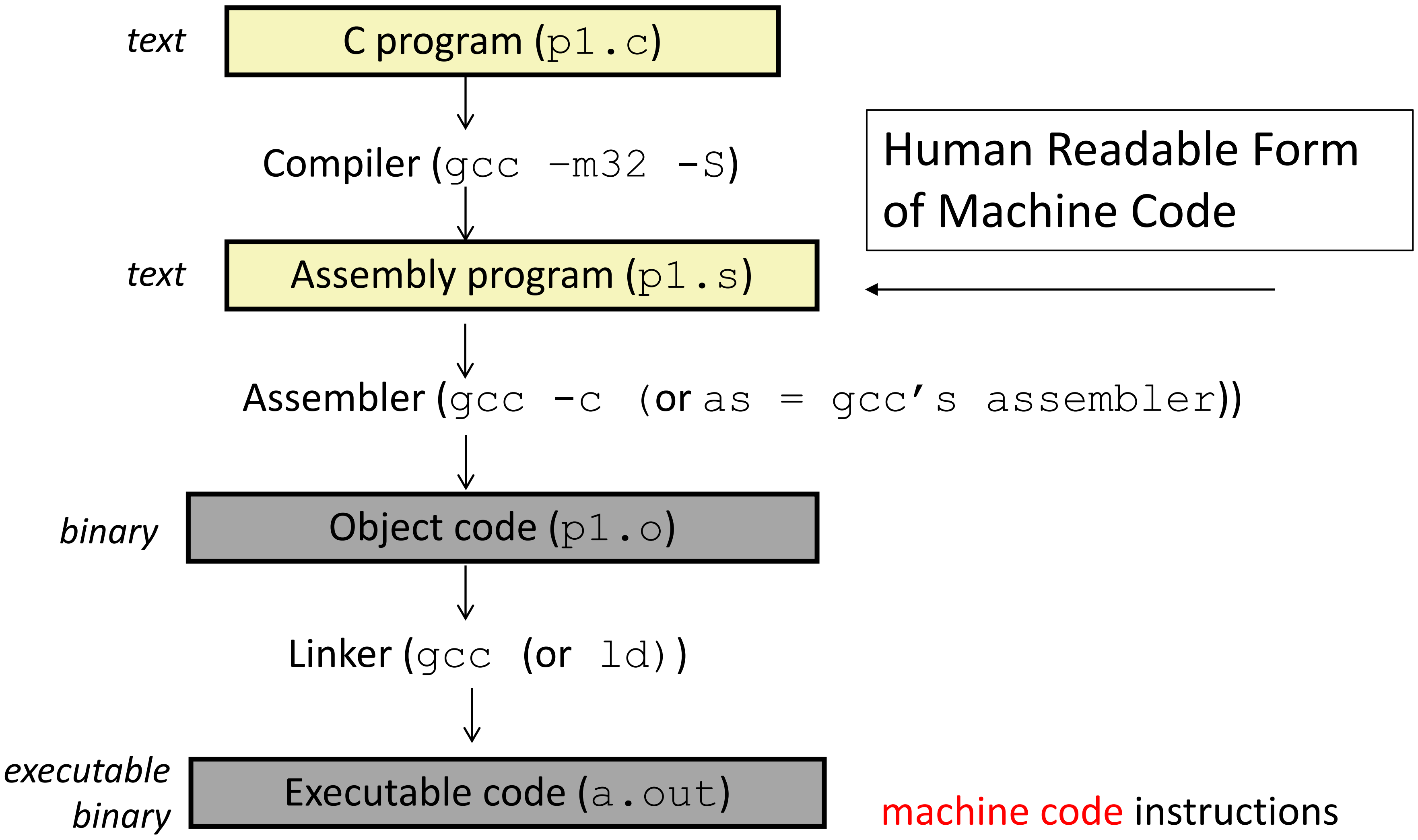
- some bits: encode the instruction (opcode bits)
- others encode operand(s)

```
(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

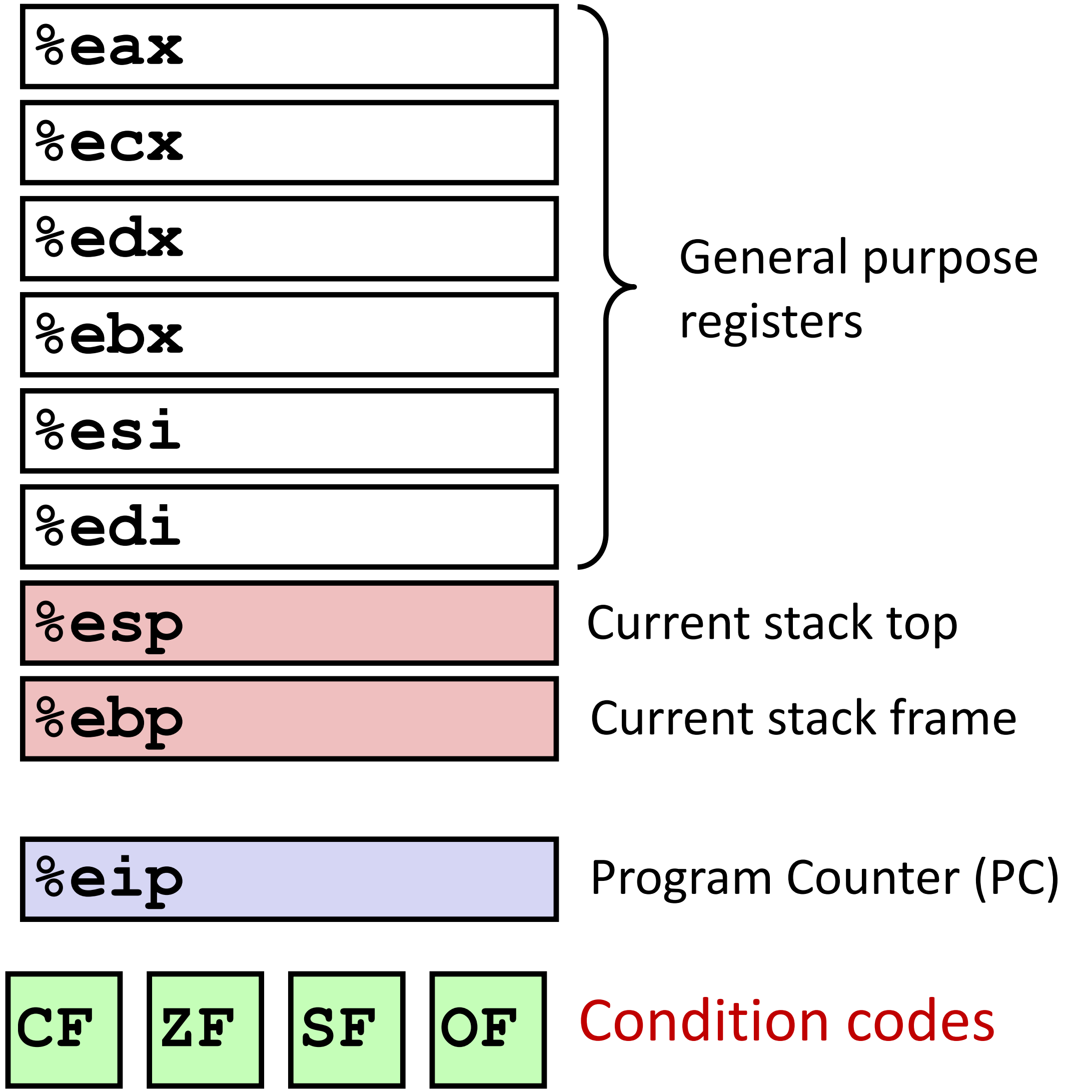
Assembly

```
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

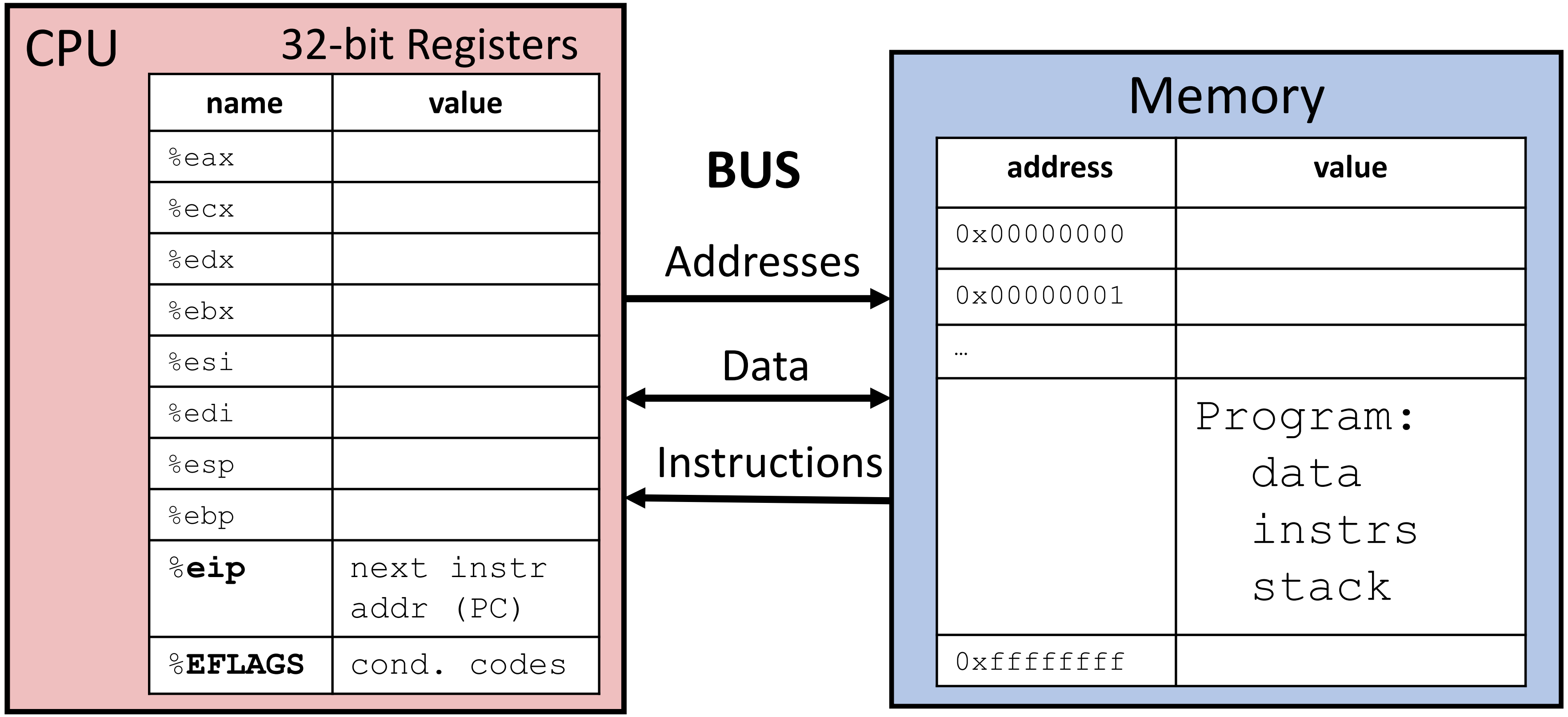
Two low-order 1 bytes %al is the low-order 8 bits of %eax

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

BUS

Addresses

Data

Instructions

Memory

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

Registers:

PC: Program counter (%eip)

Condition codes (%EFLAGS)

General Purpose (%eax - %ebp)

Memory:

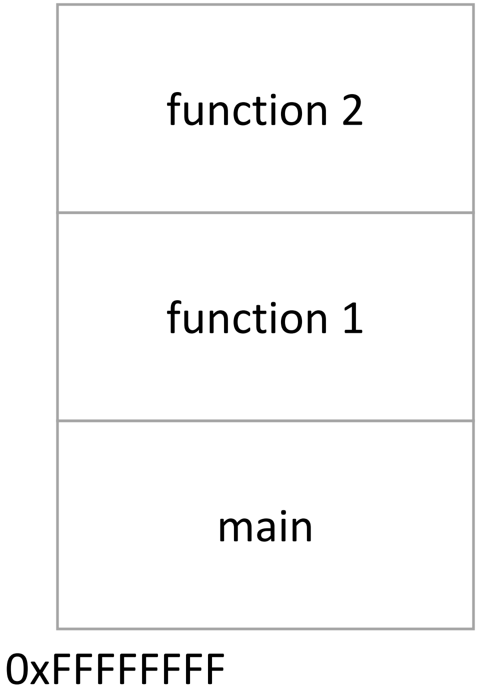
- Byte addressable array
- Program code and data
- Execution stack

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

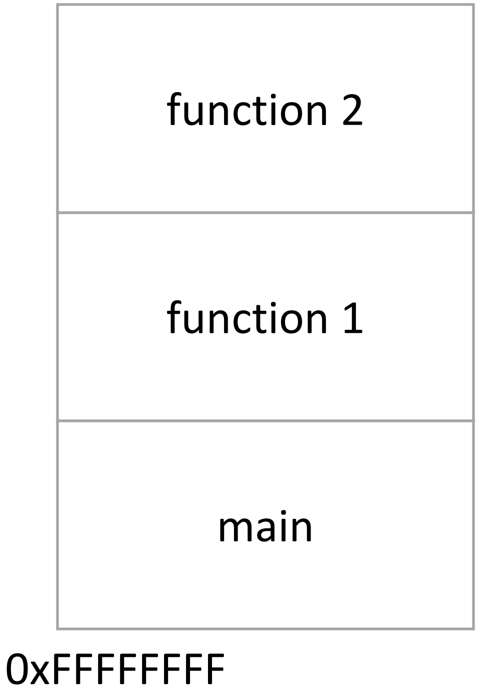


Stack Frame Contents

- 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



function 2

function 1

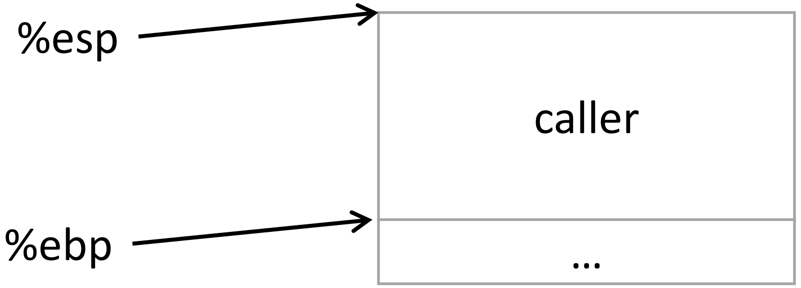
main

0xFFFFFFFF

Frame Pointer

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

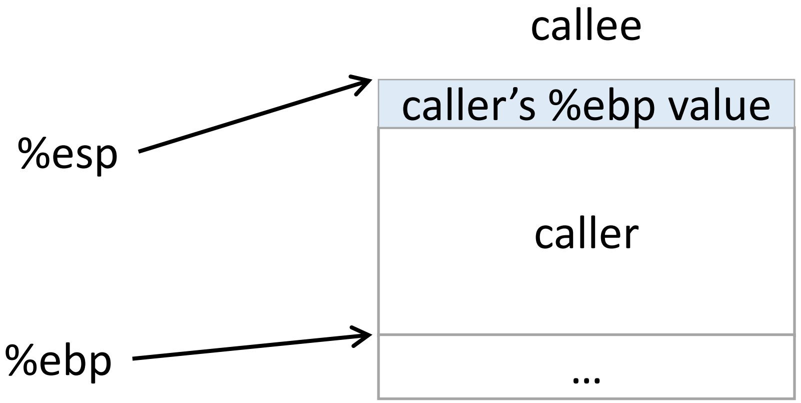
- Must adjust %esp, %ebp on call / return.



Frame Pointer

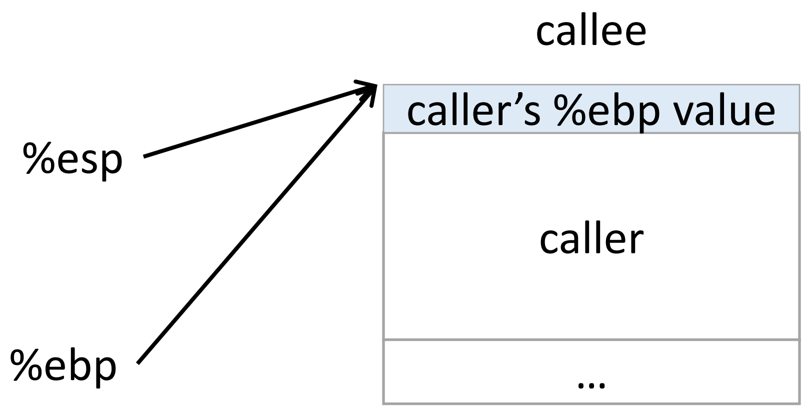
- Must maintain invariant:
  - The current function's stack frame is always between the addresses stored in %esp and %ebp

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

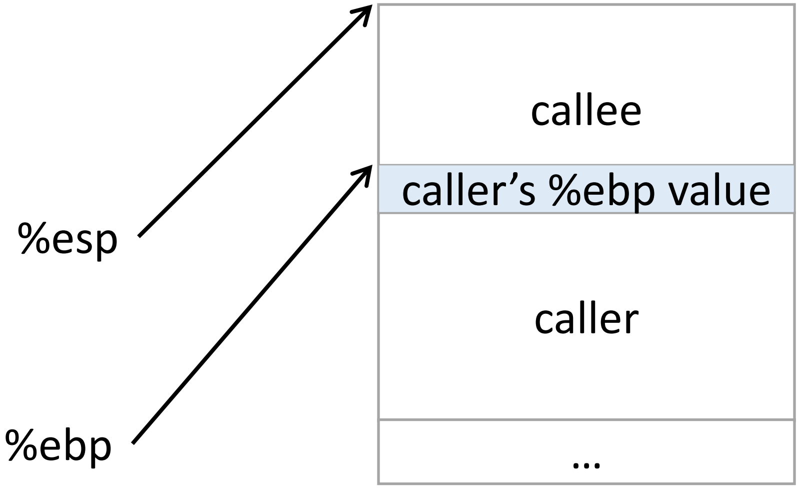


Frame Pointer

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

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

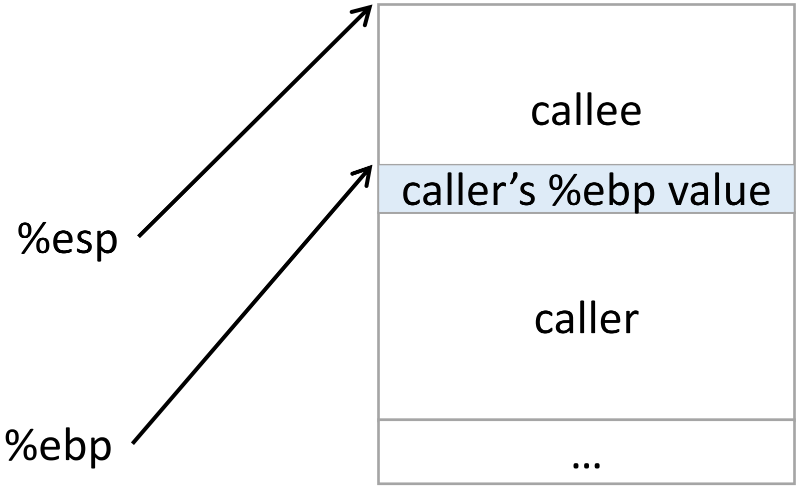
Callee can now execute.



Frame Pointer

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

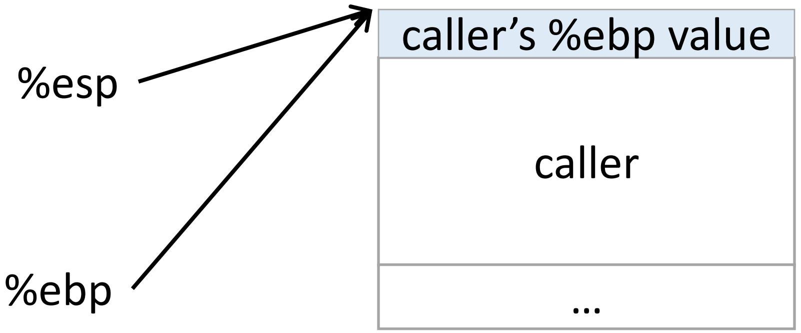
- To return, reverse this:



Frame Pointer

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

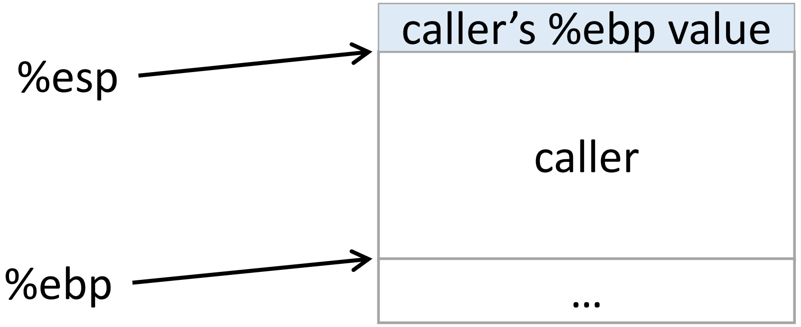
- To return, reverse this:
  - set %esp = %ebp



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:
  - set %esp = %ebp
  - popl %ebp



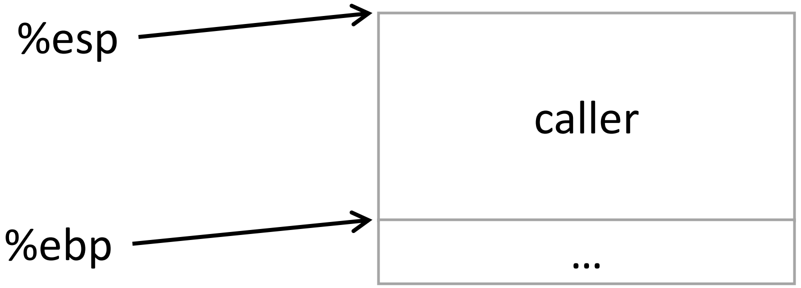
Frame Pointer

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

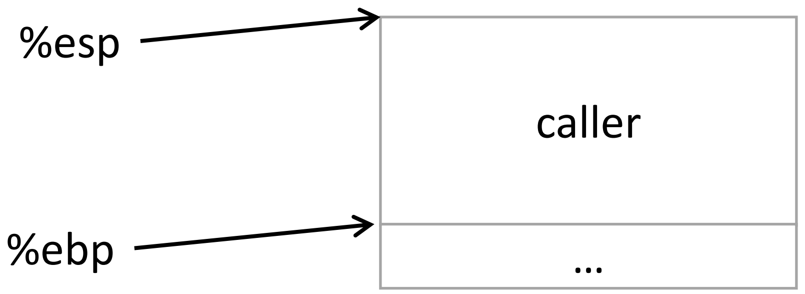
- To return, reverse this:
  - set %esp = %ebp
  - popl %ebp

IA32 has another convenience instruction for this: leave

Back to where we started.



Frame Pointer: Function Call



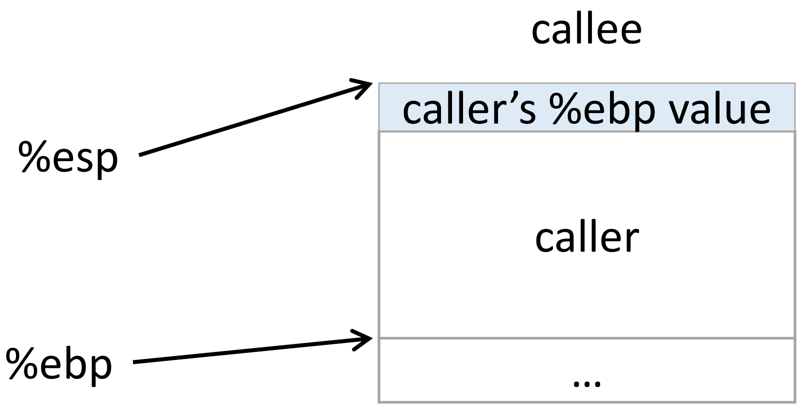
%esp

caller

%ebp

…

Initial state



callee

caller’s %ebp value

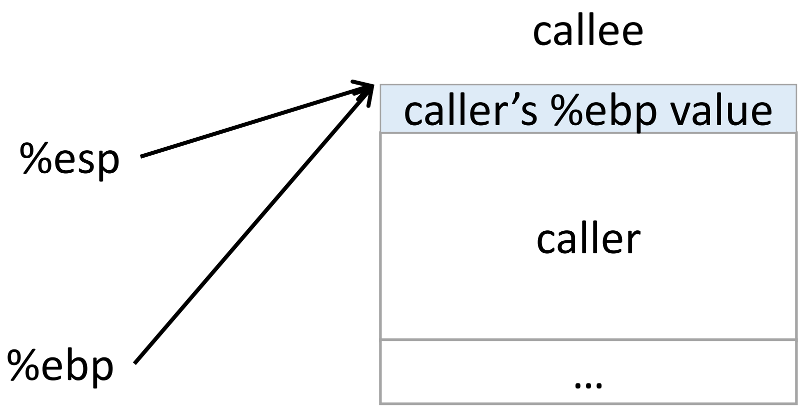
%esp

caller

%ebp

…

pushl %ebp (store caller’s frame pointer)



callee

caller’s %ebp value

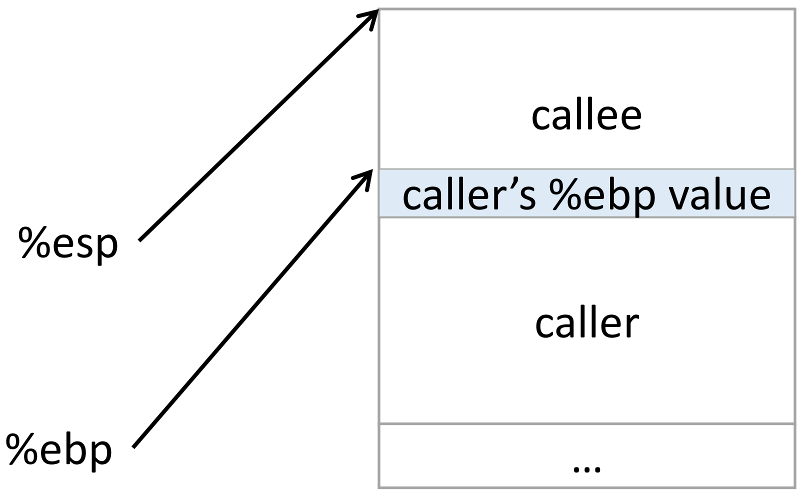
%esp

caller

%ebp

…

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



callee

caller’s %ebp value

%esp

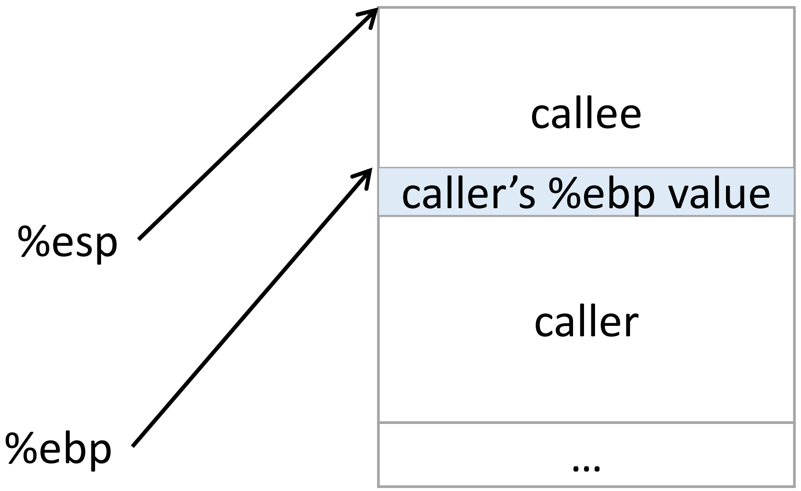
caller

%ebp

…

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

Frame Pointer: Function Return



callee

caller's %ebp value

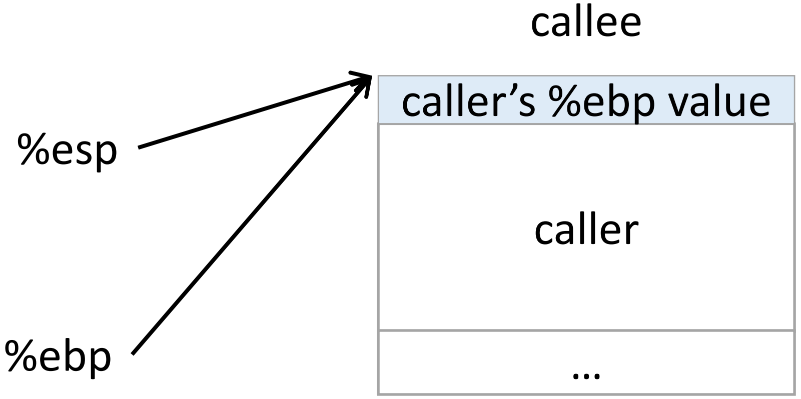
caller

...

%esp

%ebp

Want to restore caller's frame.



callee

caller's %ebp value

caller

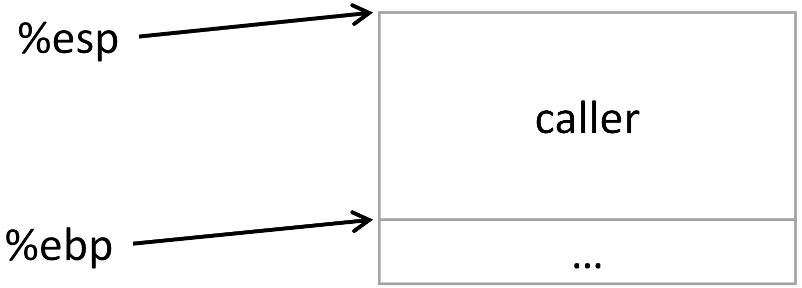
...

%esp

%ebp

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

IA32 provides a convenience instruction that does all of this:
`leave`



%esp

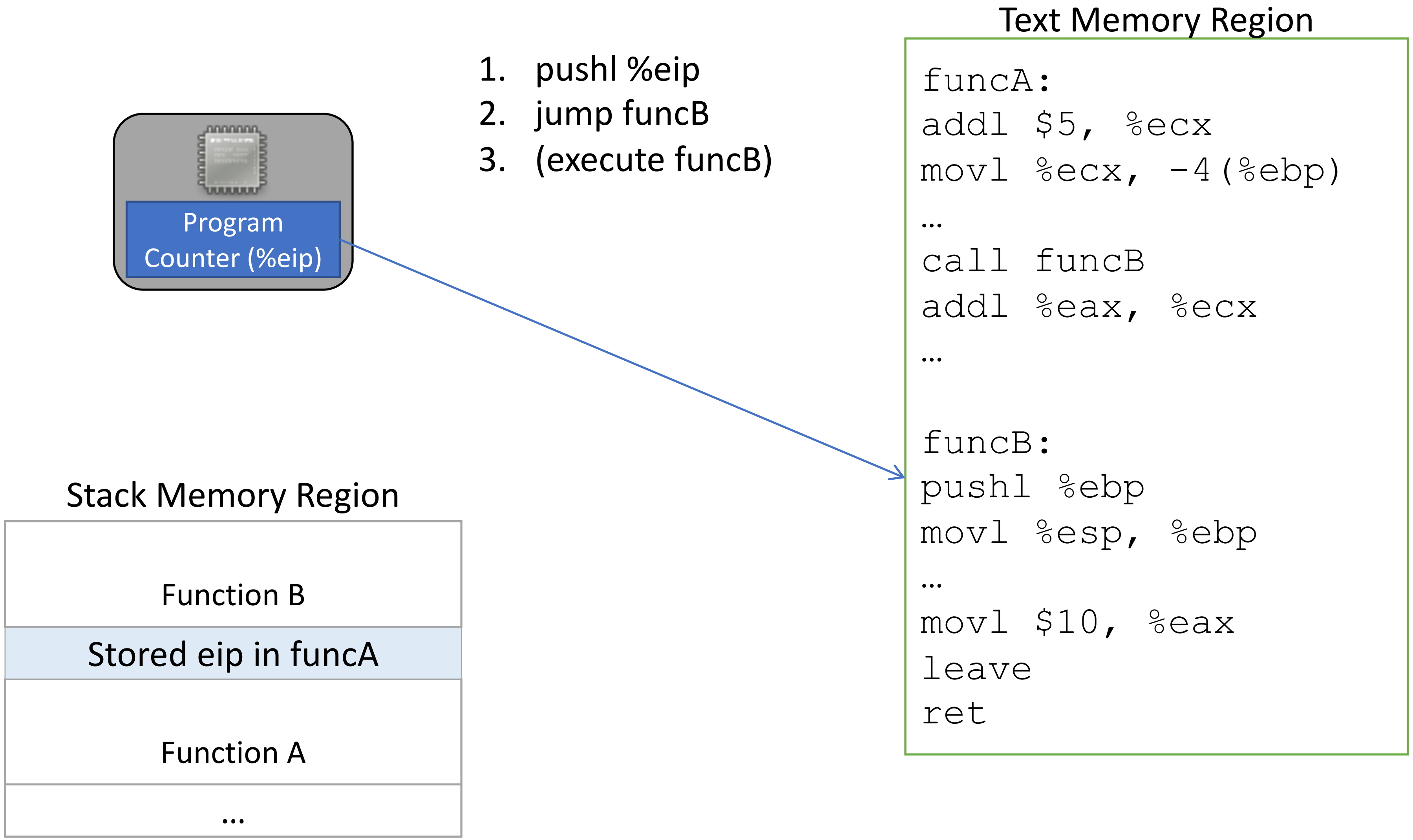
caller

%ebp

...

popl %ebp (restore caller's frame pointer)

Functions and the Stack



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

Text Memory Region

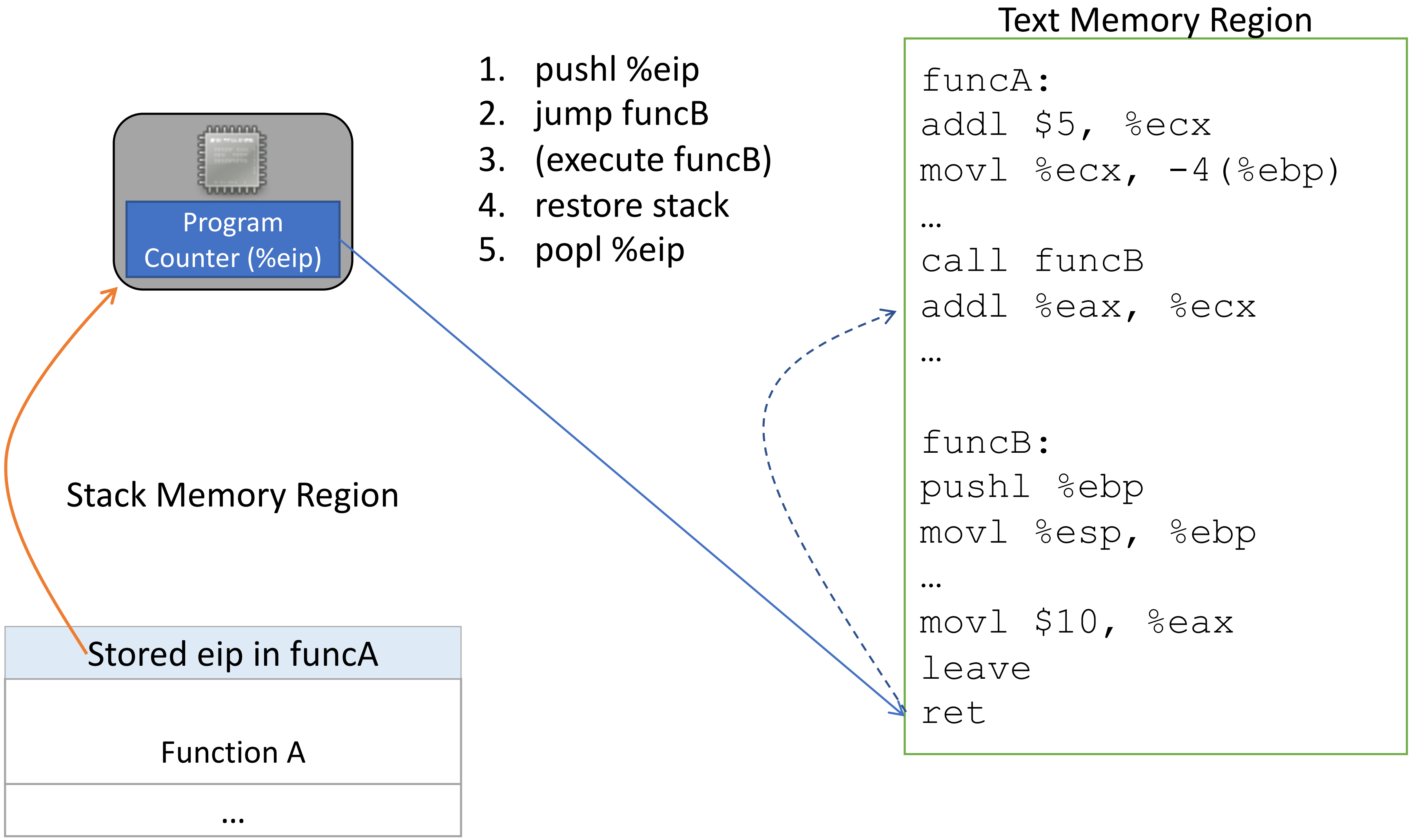
```
funcA:
addl $5, %ecx
movl %ecx, -4(%ebp)
…
call funcB
addl %eax, %ecx
…

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

Stack Memory Region

Function B
Stored eip in funcA
Function A
…

Functions and the Stack

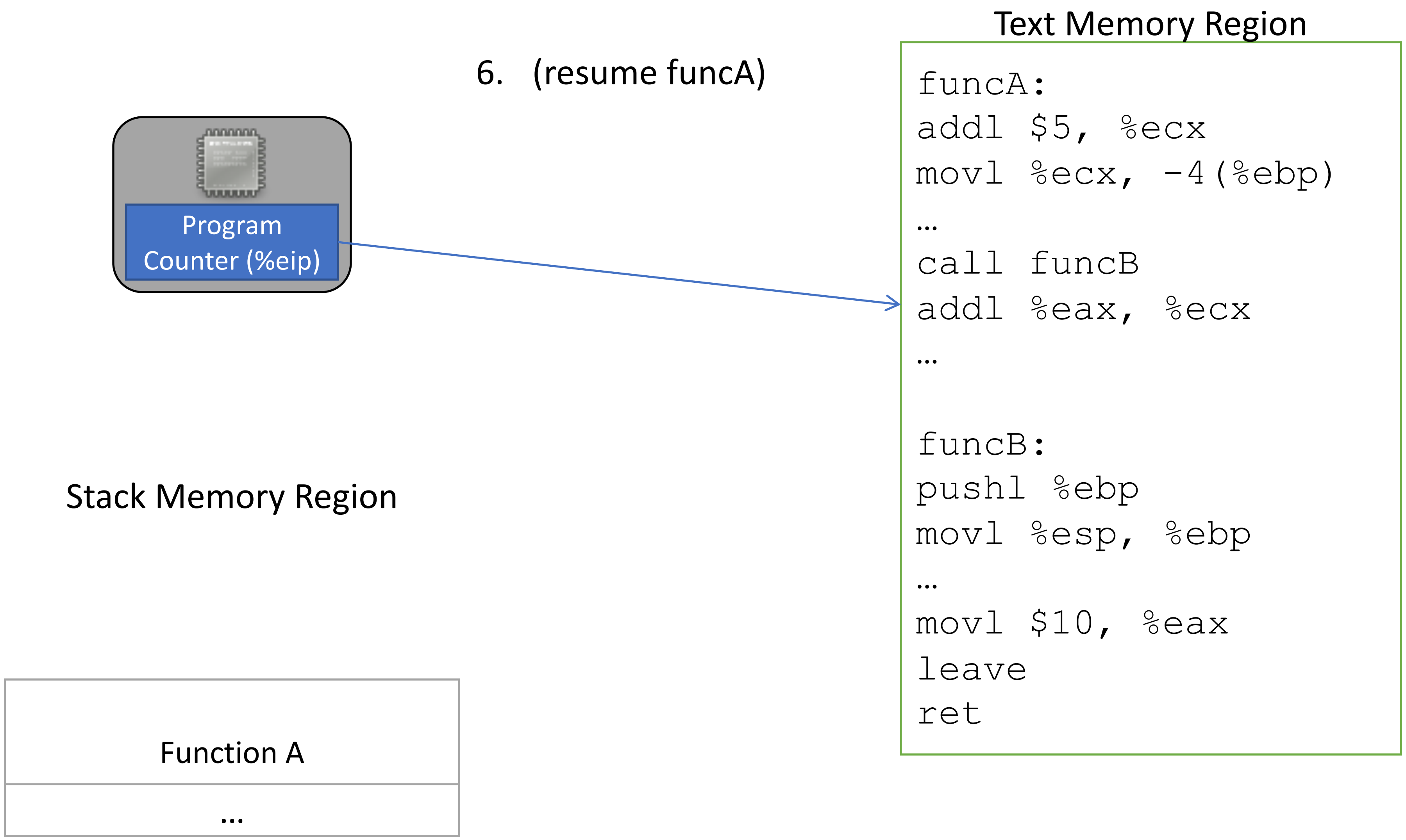
1. pushl %eip
2. jump funcB
3. (execute funcB)
4. restore stack
5. popl %eip



```
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)



Program Counter (%eip)

Stack Memory Region

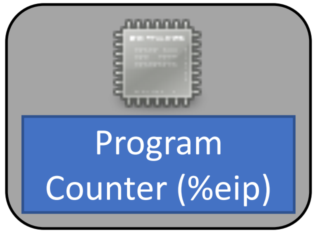
Function A

…

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)

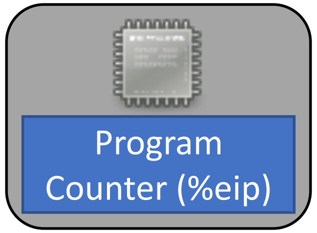
Text Memory Region

```
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

Stored eip in funcA
Function A
…

Functions and the Stack



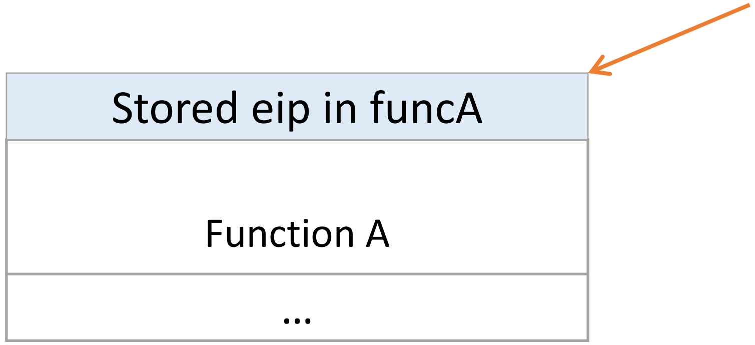
1. pushl %eip
2. jump funcB
3. (execute funcB)
4. restore stack
5. popl %eip
6. (resume funcA)

call (1–2)

leave (4)

ret (5)

Stack Memory Region



Stored eip in funcA
Function A
…

Return address:

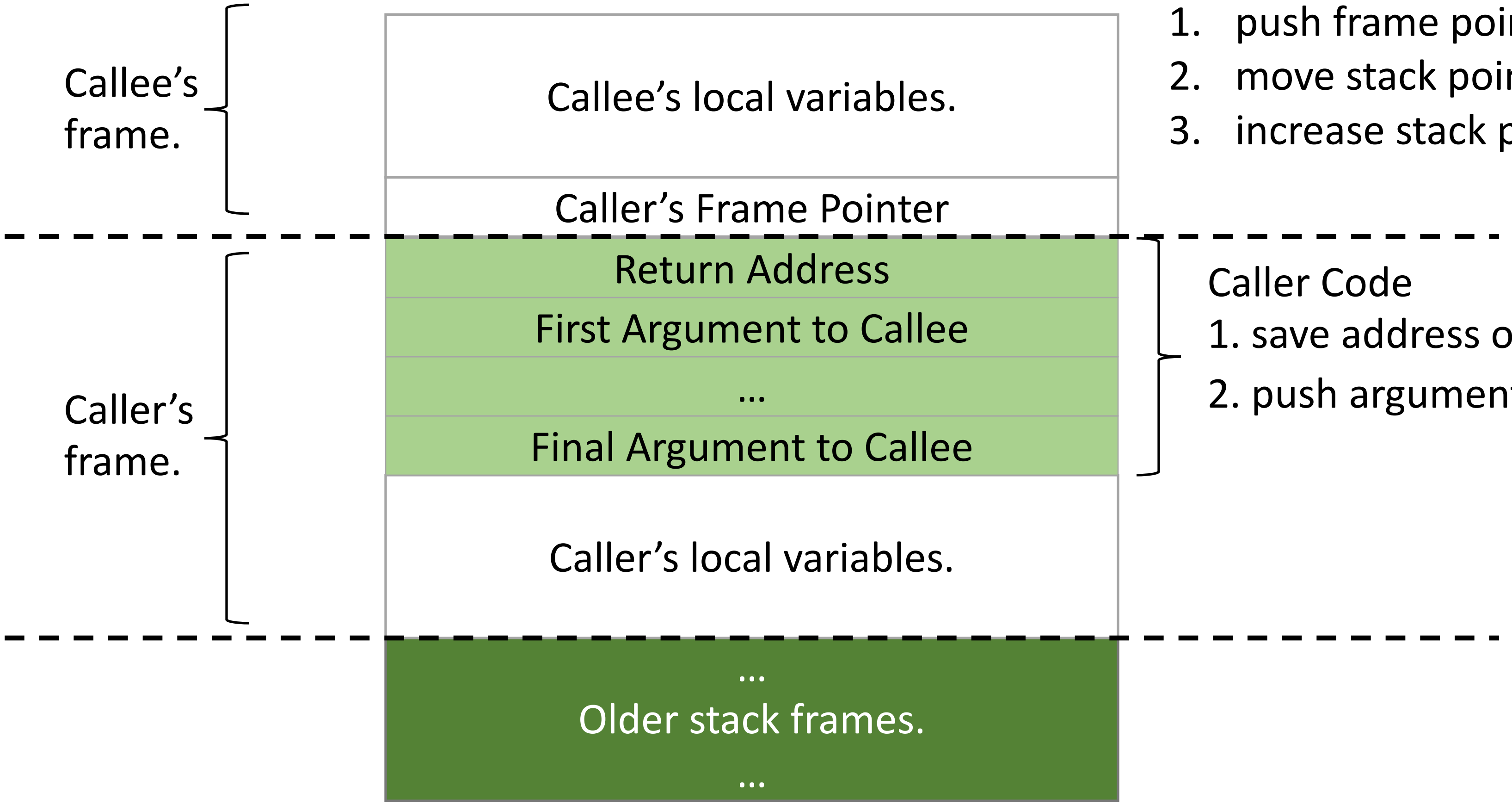
Address of the instruction we should jump back to when we finish (return from) the currently executing function.

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

- 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

Putting it all together…



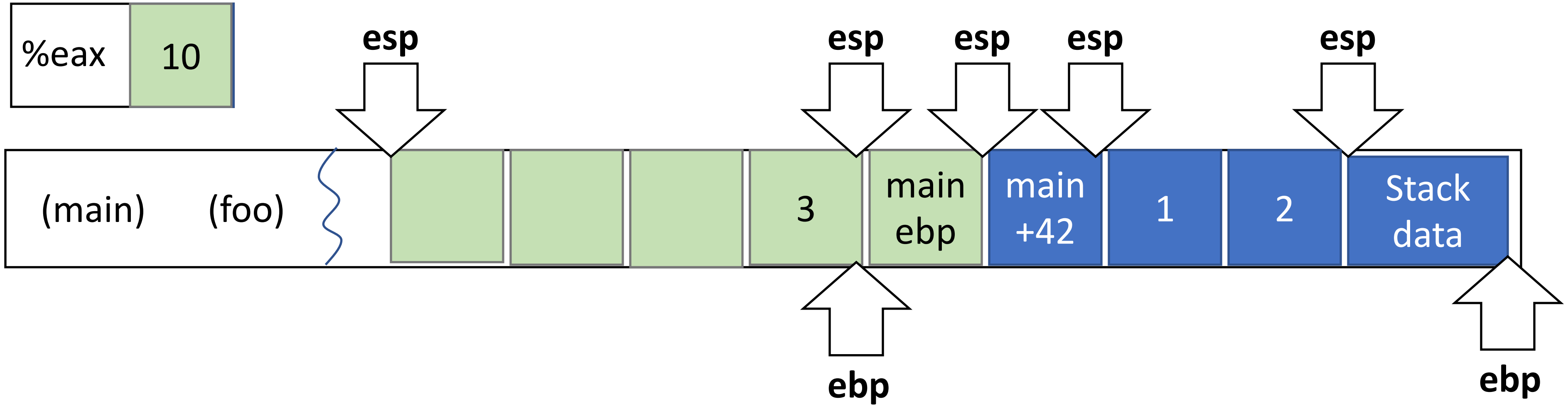
Callee Code

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

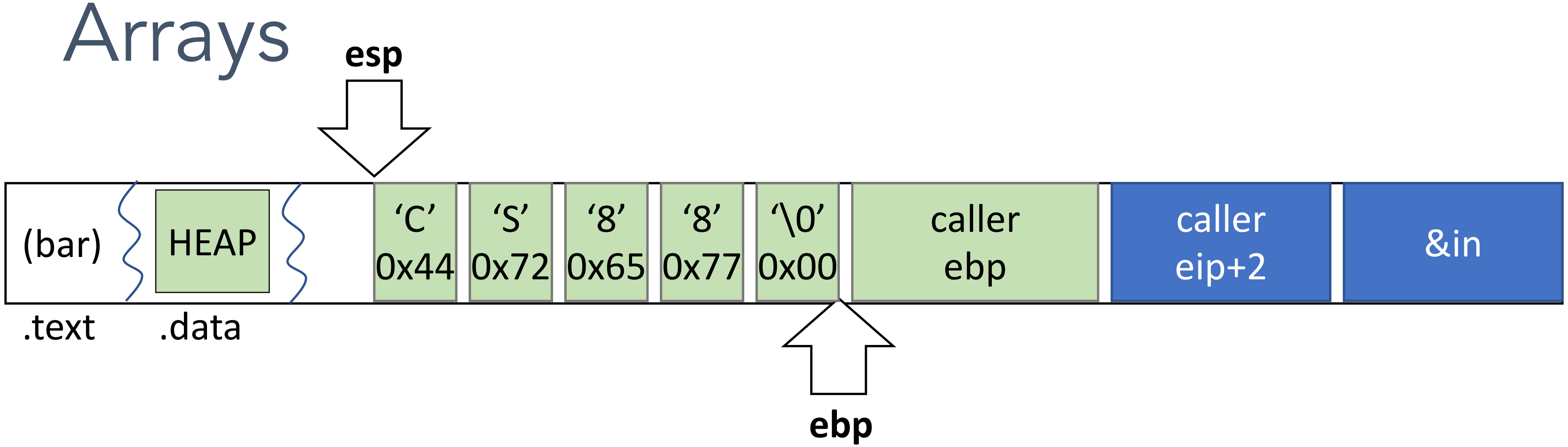
Implementing a function call



```
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



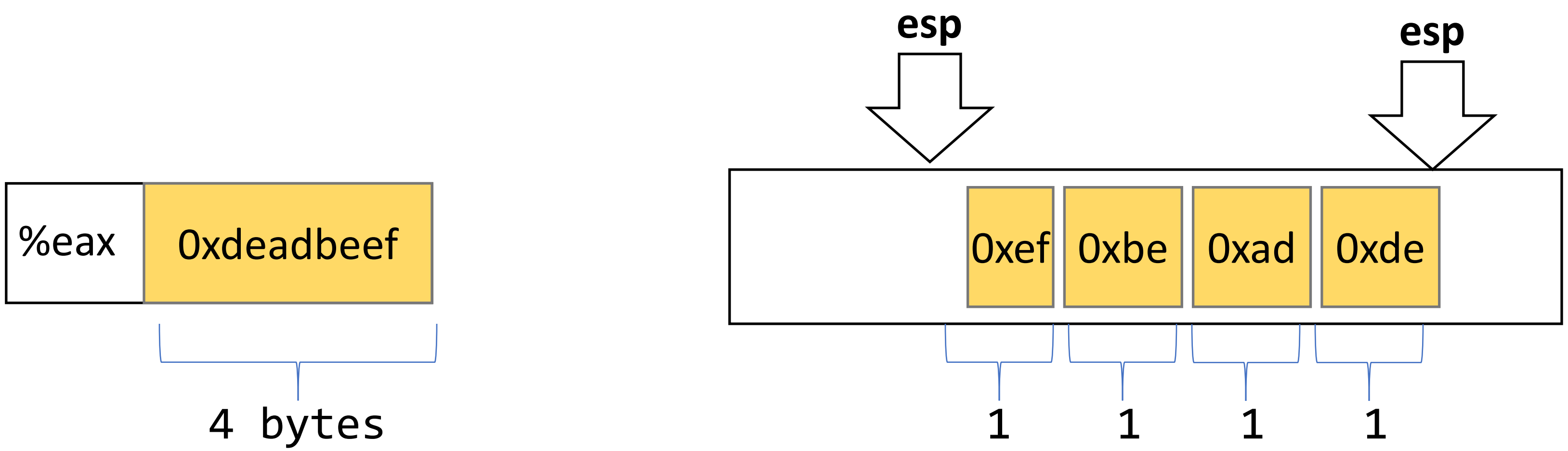
```
void bar(char * in){
  char name[5];
  strcpy(name, in);
}
```

```
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
```



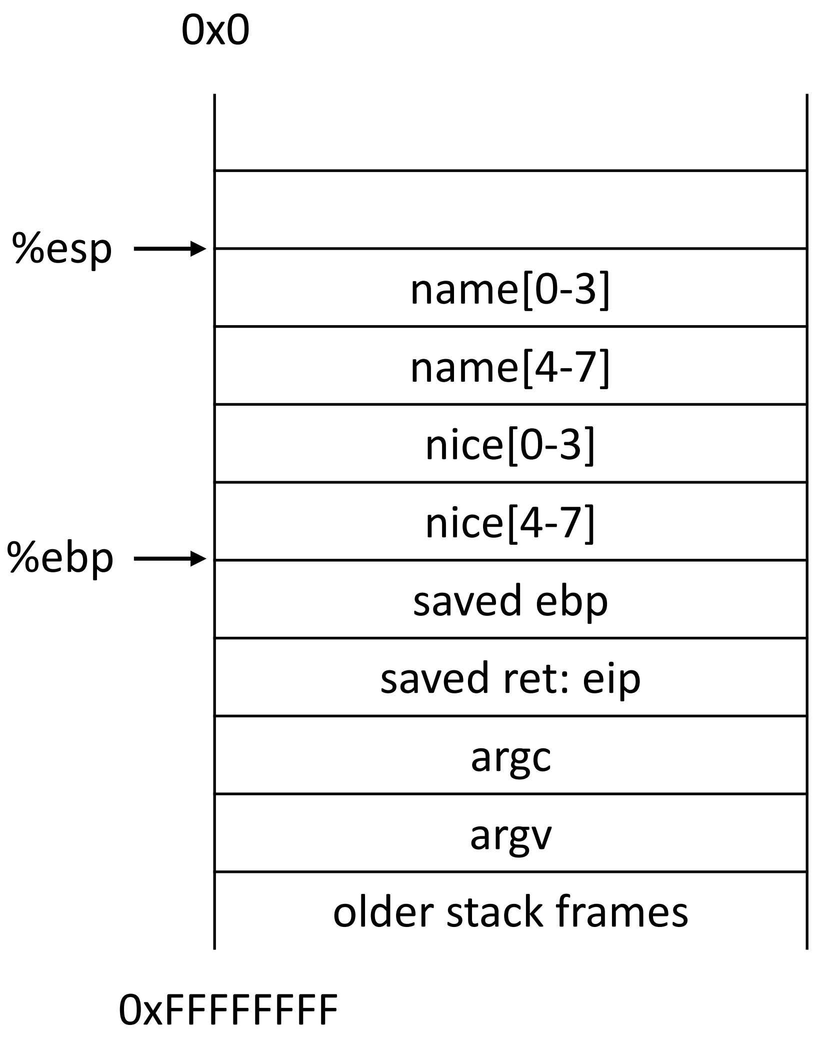
Buffer Overflows

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;
}
```



0x0

%esp

%ebp

name[0-3]

name[4-7]

nice[0-3]

nice[4-7]

saved ebp

saved ret: eip

argc

argv

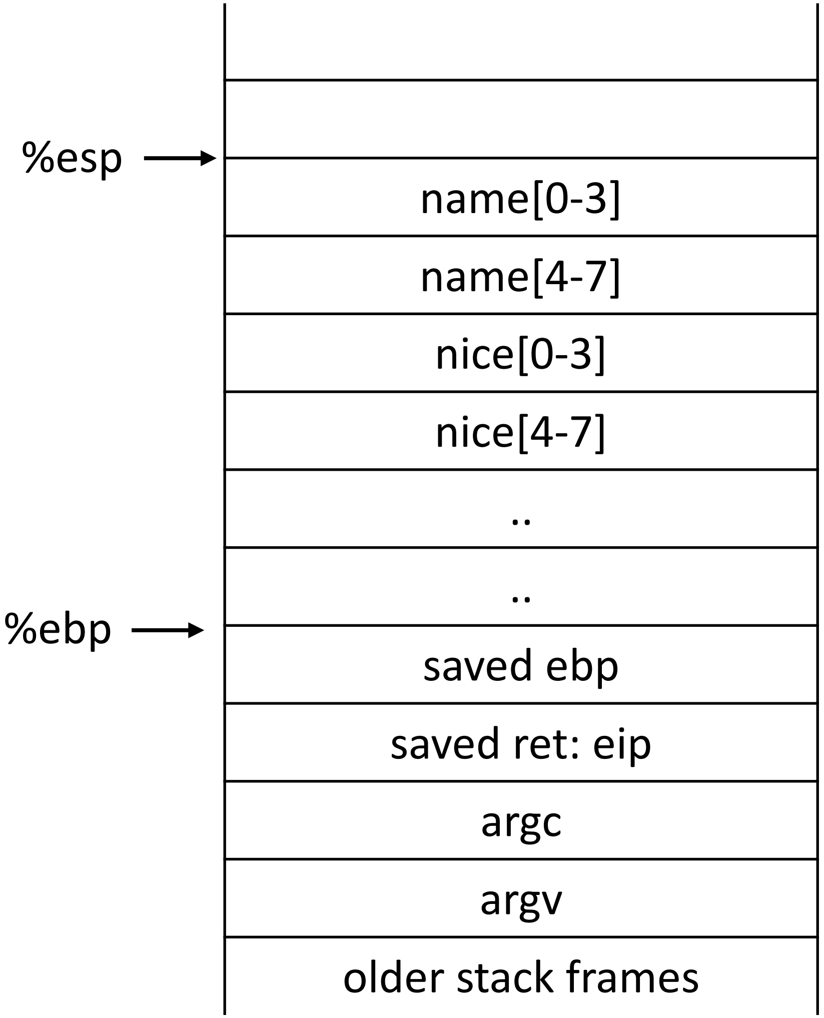
older stack frames

0xFFFFFFFF

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;
}
```



%esp →

name[0-3]

name[4-7]

nice[0-3]

nice[4-7]

..

..

%ebp →

saved ebp

saved ret: eip

argc

argv

older stack frames

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