Security Principles (continued) and x86 Assembly

CS 161 Spring 2022 - Lecture 2

Heads up: These slides won't exactly match the recording, sorry, but the differences are all fairly minor, and both have the exact same content.

Announcements

- HW1 to be released after lecture
- Project 1 will be released next week
- Discussion sections start next week
  - Attend any one you want

Next: Security Principles (continued)

- Security principles
  - Know your threat model
  - Consider human factors
  - Security is economics
  - Detect if you can’t prevent
  - Defense in depth
  - Least privilege
  - Separation of responsibility
  - Ensure complete mediation
  - Don’t rely on security through obscurity
  - Use fail-safe defaults
  - Design in security from the start

Defense in Depth

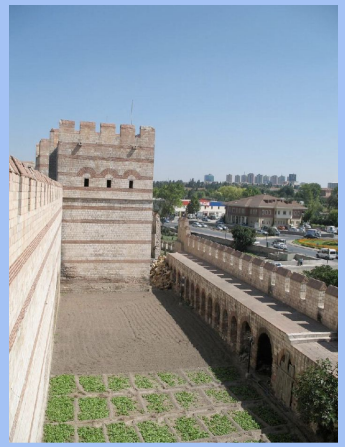
Textbook Chapter 1.5

Defense in Depth

- Multiple types of defenses should be layered together
- An attacker should have to breach all defenses to successfully attack a system
  - Ideally the strength of the defenses compounds somehow
- However, remember: security is economics
  - Defenses are not free.
  - Diminishing returns: Defenses are often less than the sum of their parts
    - 2 walls is much better than 1 wall
    - 101 walls is not much better than 100 walls

The Theodosian Walls of Constantinople

- The ancient capital of the Byzantine empire had a wall…
  - Well, they had a moat…
  - then a wall…
  - then a depression…
  - … and then an even bigger wall
- It also had towers to rain fire and arrows upon the enemy…
- Lasted until the Ottoman empire came along with cannons in 1453…
  - And now it's Istanbul not Constantinople
- **Takeaway**: Defense in depth: An attacker needed to breach all the walls
- **Takeaway**: Changing attacker technology changes defensive requirements



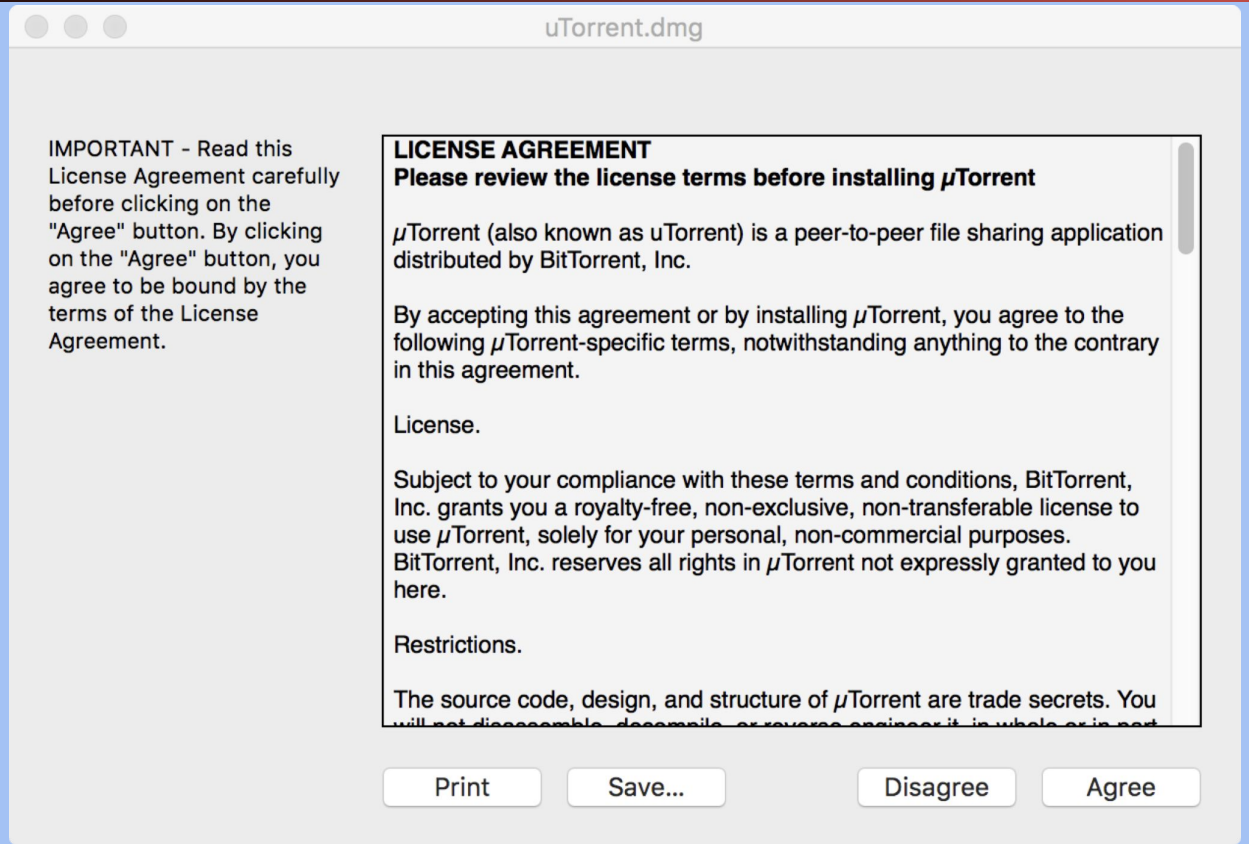
Least Privilege

Textbook Chapter 1.6

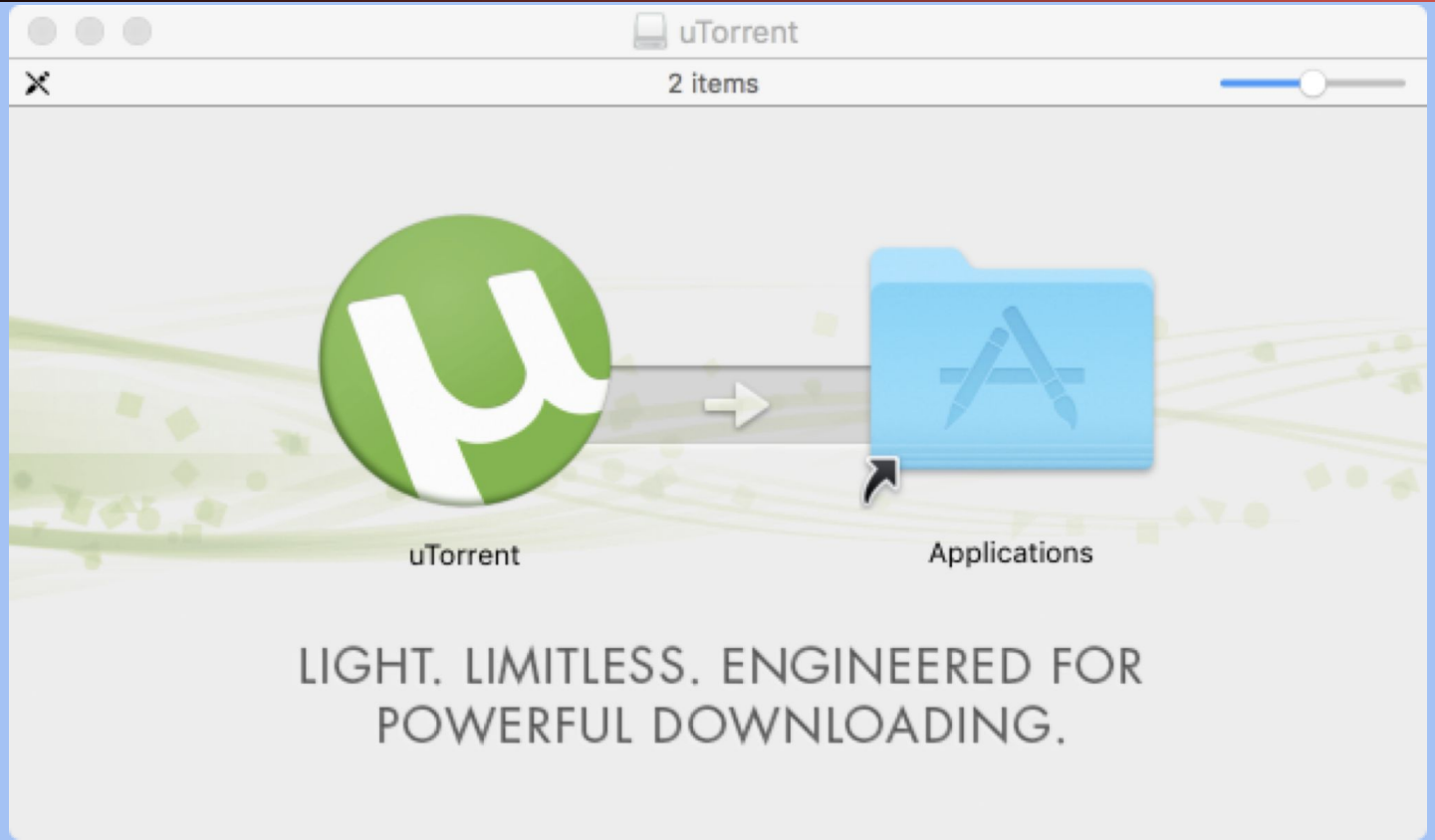
Least Privilege

- Consider the minimum permissions an entity or program *needs* to be able to do its job correctly, and grant only those permissions
  - If you grant unnecessary permissions, a malicious or hacked program could use those permissions against you

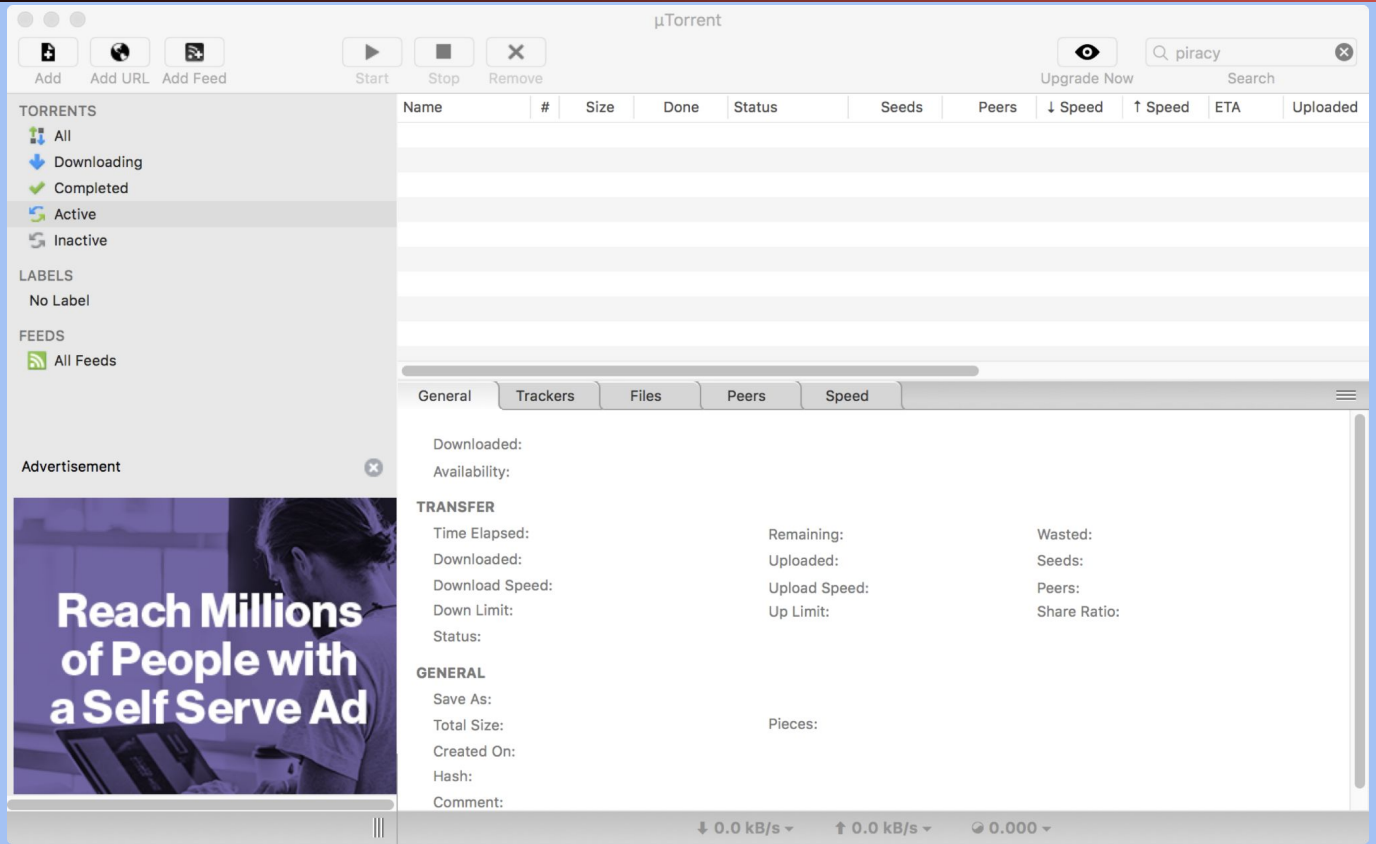
uTorrent


uTorrent.dmg
IMPORTANT - Read this License Agreement carefully before clicking on the "Agree" button. By clicking on the "Agree" button, you agree to be bound by the terms of the License Agreement.
LICENSE AGREEMENT
Please review the license terms before installing µTorrent
µTorrent (also known as uTorrent) is a peer-to-peer file sharing application distributed by BitTorrent, Inc.
By accepting this agreement or by installing µTorrent, you agree to the following µTorrent-specific terms, notwithstanding anything to the contrary in this agreement.
License.
Subject to your compliance with these terms and conditions, BitTorrent, Inc. grants you a royalty-free, non-exclusive, non-transferable license to use µTorrent, solely for your personal, non-commercial purposes. BitTorrent, Inc. reserves all rights in µTorrent not expressly granted to you here.
Restrictions.
The source code, design, and structure of µTorrent are trade secrets. You
Print
Save...
Disagree
Agree

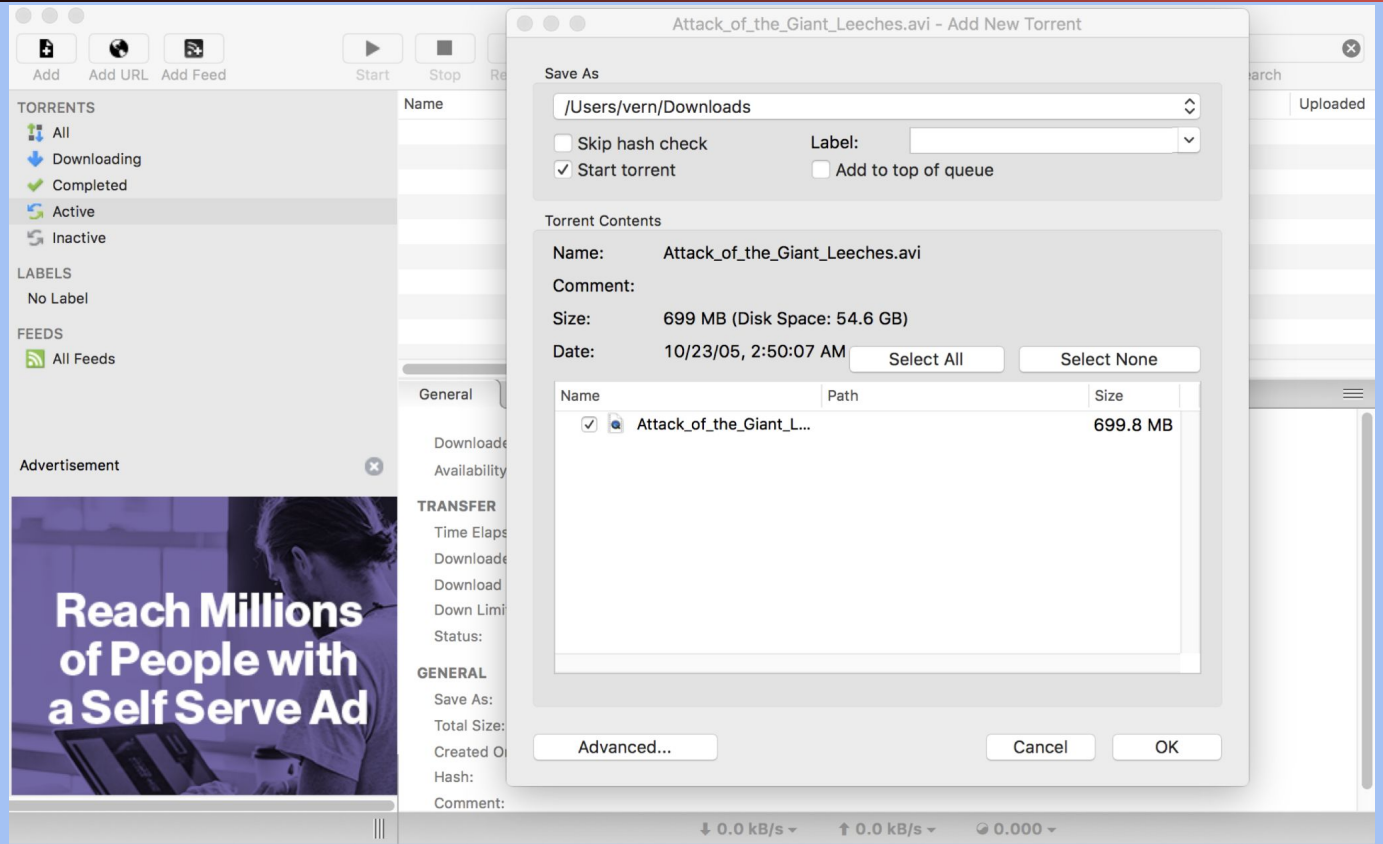
uTorrent


uTorrent
2 items
uTorrent
Applications
LIGHT. LIMITLESS. ENGINEERED FOR
POWERFUL DOWNLOADING.

uTorrent

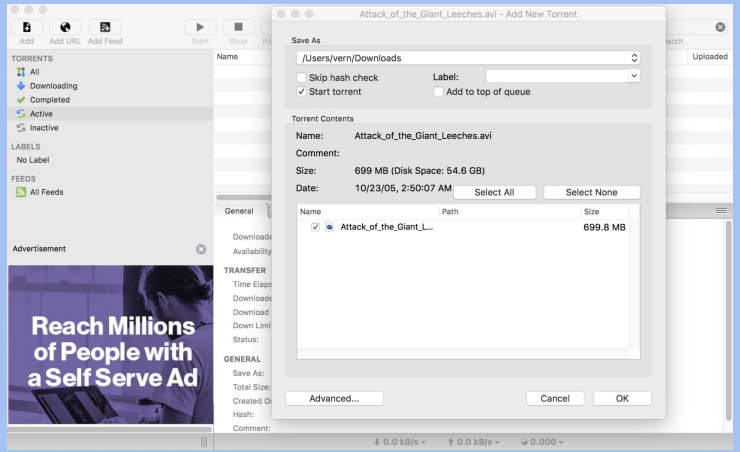

µTorrent
Add
Add URL
Add Feed
Start
Stop
Remove
Upgrade Now
piracy
Search
TORRENTS
All
Downloading
Completed
Active
Inactive
LABELS
No Label
FEEDS
All Feeds
Name
#
Size
Done
Status
Seeds
Peers
↓ Speed
↑ Speed
ETA
Uploaded
General
Trackers
Files
Peers
Speed
Downloaded:
Availability:
TRANSFER
Time Elapsed:
Remaining:
Wasted:
Downloaded:
Uploaded:
Seeds:
Download Speed:
Upload Speed:
Peers:
Down Limit:
Up Limit:
Share Ratio:
Status:
GENERAL
Save As:
Total Size:
Pieces:
Created On:
Hash:
Comment:
Advertisement
Reach Millions of People with a Self Serve Ad
0.0 kB/s
0.0 kB/s
0.000

uTorrent

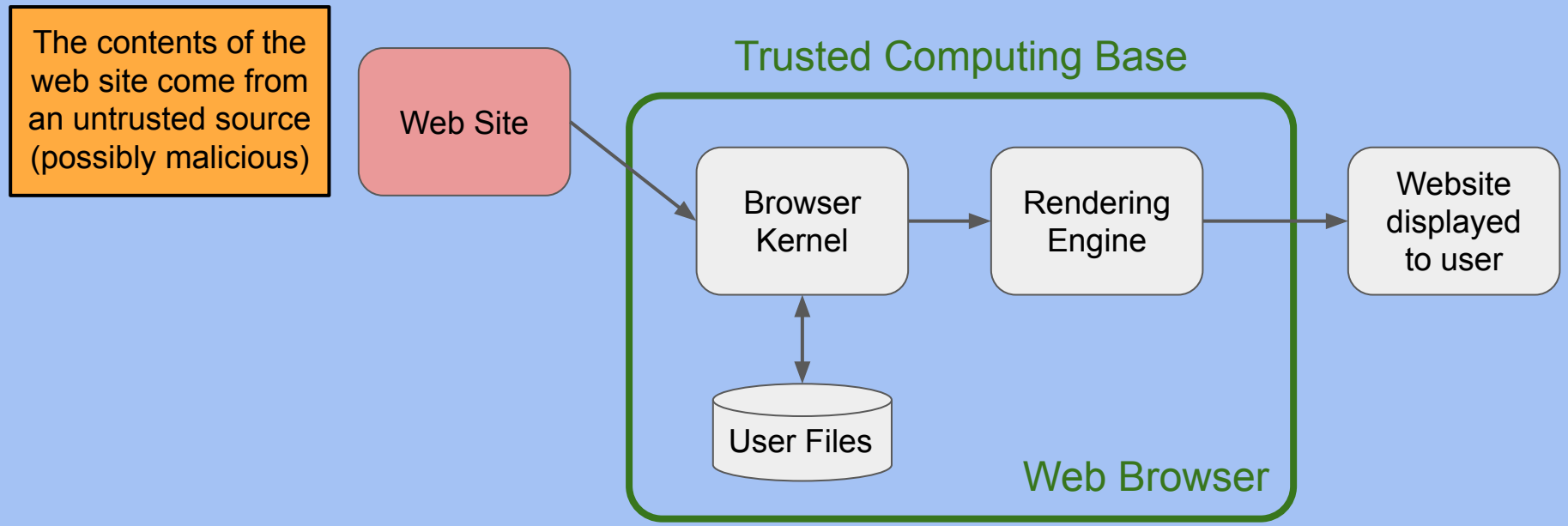

Add
Add URL
Add Feed
Start
Stop
TORRENTS
All
Downloading
Completed
Active
Inactive
LABELS
No Label
FEEDS
All Feeds
Advertisement
Reach Millions of People with a Self Serve Ad
Name
Uploaded
General
TRANSFER
GENERAL
Save As:
Total Size:
Hash:
Comment:
Attack_of_the_Giant_Leeches.avi - Add New Torrent
Save As
/Users/vern/Downloads
Skip hash check
Label:
Start torrent
Add to top of queue
Torrent Contents
Name: Attack_of_the_Giant_Leeches.avi
Comment:
Size: 699 MB (Disk Space: 54.6 GB)
Date: 10/23/05, 2:50:07 AM
Select All
Select None
Name
Path
Size
Attack_of_the_Giant_L...
699.8 MB
Advanced...
Cancel
OK
0.0 kB/s
0.0 kB/s
0.000

uTorrent

- What was this program able to do?
  - Leak your files
  - Delete your files
  - Send spam
  - Run another malicious program
- What does this program need to be able to do?
  - Access the screen
  - Manage some files (but not all files)
  - Make some Internet connections (but not all Internet connections)
- **Takeaway**: Least privilege

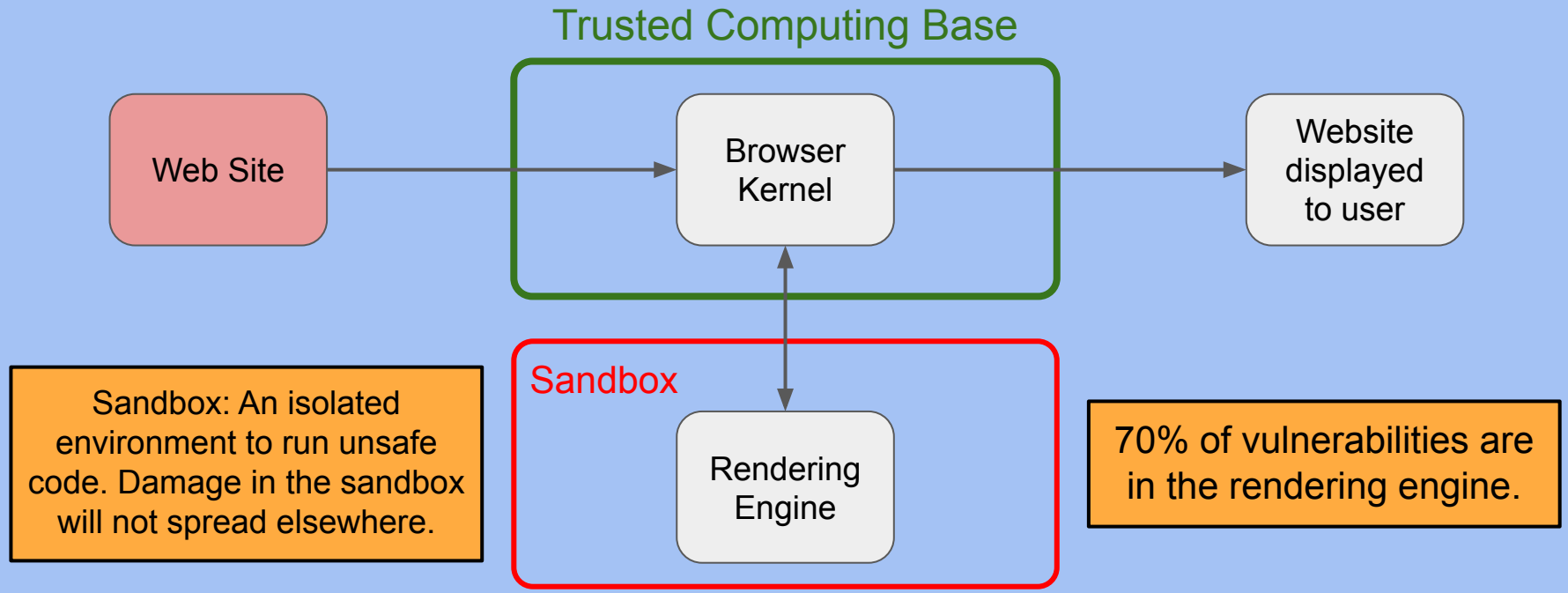


Browser Design with Poor Least Privilege



"Drive-by malware": A malicious web page exploits a browser bug to infect local files

Google Chrome Design: Apply Least Privilege



Prevent "drive-by malware," where a malicious webpage exploits a browser bug to infect local files

Enabling Least Privilege: Access Control

- How to control who has access to particular data
  - Access needs to be *necessary* to accomplish the tasks
  - Least privilege: Excessive access can be a problem
- Example: Access control for DSP data in CS 161
  - Separate Google shared Drive from the main Drive
    - This allows independent access control *defaults*
  - *Limit* access to that Drive to those who have a need to know
    - Nick, head TAs, DSP TAs, course manager

Access Control for Systems: The Operating System

- The OS is the TCB of most modern systems, and it provides access controls to restrict the privileges of user programs
- The OS provides the following “guarantees”:
  - **Isolation**: A process can’t read or write the memory of any other process
  - **Permissions**: A process can only change files, interact with devices, etc. if it has permission to

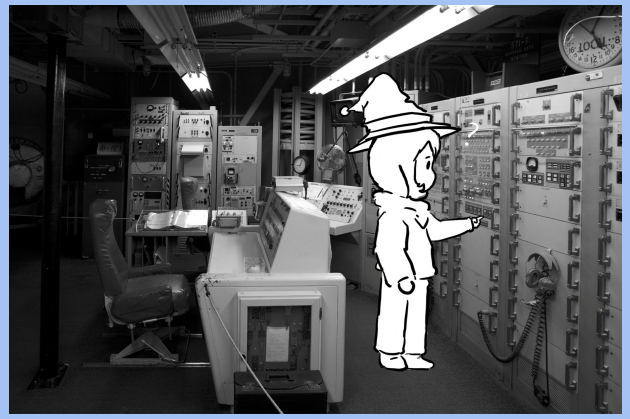
Separation of Responsibility

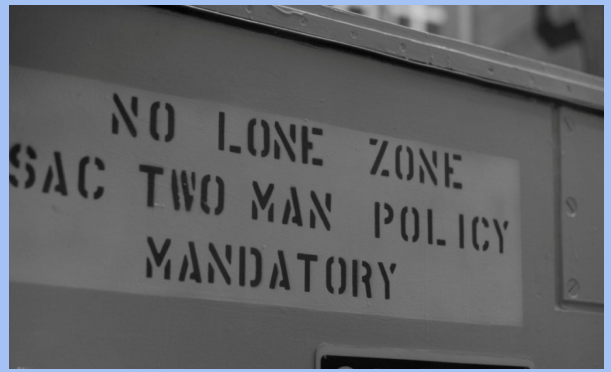
Textbook Chapter 1.7

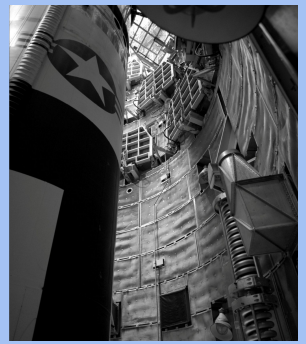
Separation of Responsibility

- Also known as distributed trust
- If you need to have a privilege, consider requiring multiple parties to work together (collude) to exercise it
  - It's much more likely for a single party to be malicious than for all multiple parties to be malicious and collude with one another

Welcome to a Nuclear Bunker







Welcome to a Movie Theater

Ensure Complete Mediation

Textbook Chapter 1.8 & 1.13

Security Principle: Ensure Complete Mediation

- Ensure that every access point is monitored and protected
- **Reference monitor**: Single point through which all access must occur
  - Example: A network firewall, airport security, the doors to the dorms
- Desired properties of reference monitors:
  - Correctness
  - Completeness (can't be bypassed)
  - Security (can't be tampered with)
  - Should be part of the TCB

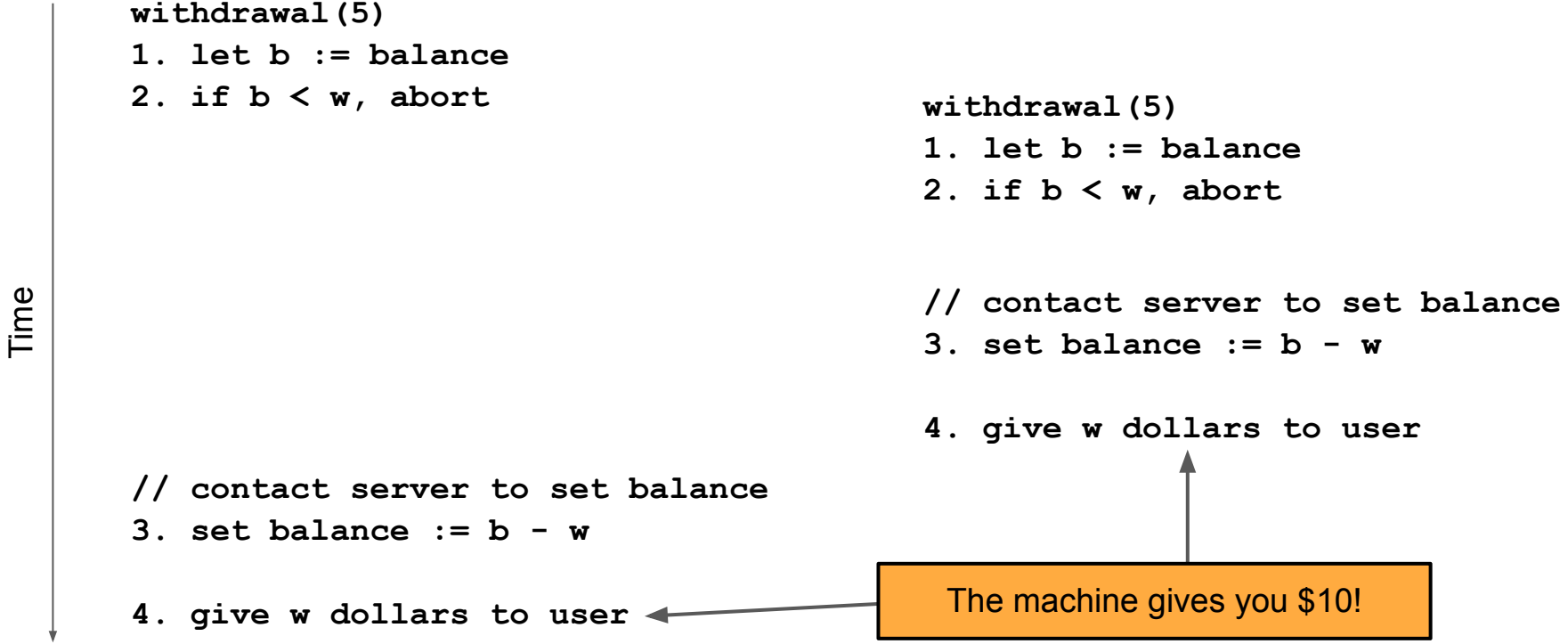
Time-of-Check to Time-of-Use

- 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



Don’t Rely on Security Through Obscurity

Textbook Chapter 1.9

Don’t Rely on Security Through Obscurity

- Also known as **Shannon’s Maxim**
  - “The enemy knows the system”
- Also known as **Kerckhoff’s Principle**
  - “The only part of a cryptographic system unknown to the adversary is the cryptographic keys”

Highway Signs

Here's a highway sign.

Here's the hidden computer inside the sign.

Here's the control panel. Most signs use the default password, **DOTS**.

Highway Signs

Note/Takeaway: Do not **ever** do this. Yes, some former CS 161 students did it once.

Highway Signs

Takeaway: Don’t rely on security through obscurity

Don’t Rely on Security Through Obscurity

- Always assume that the attacker knows every detail about the system you are working with (algorithms, hardware, defenses, etc.)
  - Sometimes, obscurity *can help*
    - Example: We don’t tell you how we detect academic dishonesty
  - However, systems that *rely* on obscurity are brittle, since the attacker may find out!
- Don’t do security through obscurity!

Assume the attacker knows where the “secret” control panel is located, and knows the default password.

Use Fail-Safe Defaults

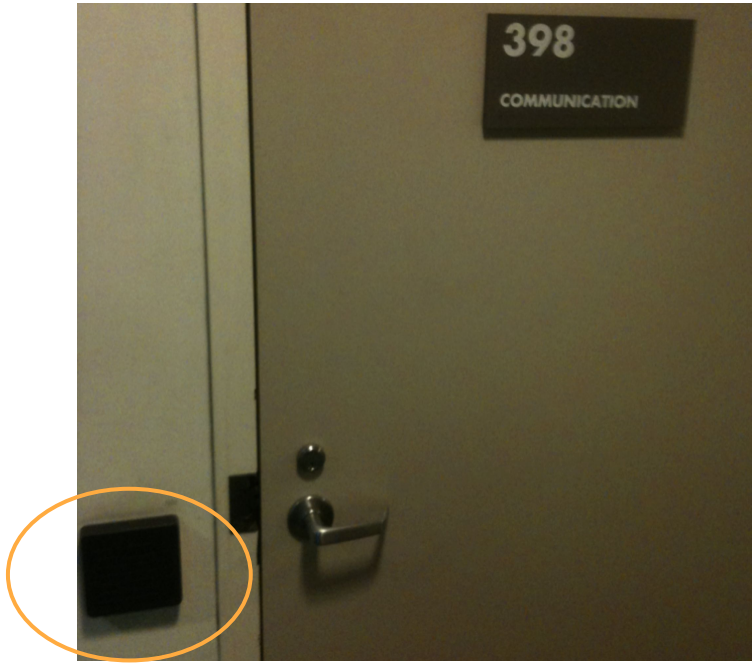
Textbook Chapter 1.10

Soda Hall

- Rooms in Berkeley's Soda Hall are guarded by electronic card keys
- What do you do if the power goes out?
  - Fail closed: No one can get in if the power is out
  - Fail open: Anyone can get in if the power goes out
- What's the best option to choose for closets with expensive equipment? What about emergency exit doors?
- **Takeaway**: Use fail-safe defaults… (if you can come up with one!)

Use Fail-Safe Defaults

- Choose default settings that “fail safe,” balancing security with usability when a system goes down
  - This can be hard to determine
- In the end, the right “default” often depends on context
  - Default open?
  - Default locked?



Design in Security from the Start

Textbook Chapter 1.11

Design in Security from the Start

- When building a new system, include security as part of the design considerations rather than patching it after the fact
  - A lot of systems today were not designed with security from the start, resulting in patches that don’t fully fix the problem!
- Keep these security principles in mind whenever you write code!

Summary: Security Principles

- **Know your threat model**: Understand your attacker and their resources and motivation
- **Consider human factors**: If your system is unusable, it will be unused
- **Security is economics**: Balance the expected cost of security with the expected benefit
- **Detect if you can't prevent**: Security requires not just preventing attacks but detecting and responding to them
- **Defense in depth**: Layer multiple types of defenses
- **Least privilege**: Only grant privileges that are needed for correct functioning, and no more
- **Separation of responsibility**: Consider requiring multiple parties to work together to exercise a privilege
- **Ensure complete mediation**: All access must be monitored and protected, unbypassable
- **Don't rely on security through obscurity**: Assume the enemy knows the system
- **Use fail-safe defaults**: Construct systems that fail in a safe state, balancing security and usability.
- **Design in security from the start**: Consider all of these security principles when designing a new system, rather than patching it afterwards

Next: x86 Assembly and Call Stack

- Part CS 61C review
  - How do computers represent numbers as bits and bytes?
  - How do computers interpret and run the programs we write?
  - How do computers organize segments of memory?
- Part new content
  - How does x86 assembly work?
  - How do you call a function in x86?

Number Representation

Textbook Chapter 2.1

Units of Measurement

- In computers, all data is represented as bits
  - **Bit**: a binary digit, 0 or 1
- Names for groups of bits
  - 4 bits = 1 **nibble**
  - 8 bits = 1 **byte**
- How many bits/nibbles/bytes in `0b 1000 1000 1000 1000`?
  - 16 bits, or 4 nibbles, or 2 bytes

Hexadecimal

- 4 bits can be represented as 1 hexadecimal digit (base 16)
- How would you write `0b11000110` in hex?
  - `0xC6`
  - Note: For clarity, we add `0b` in front of bits and `0x` in front of hex

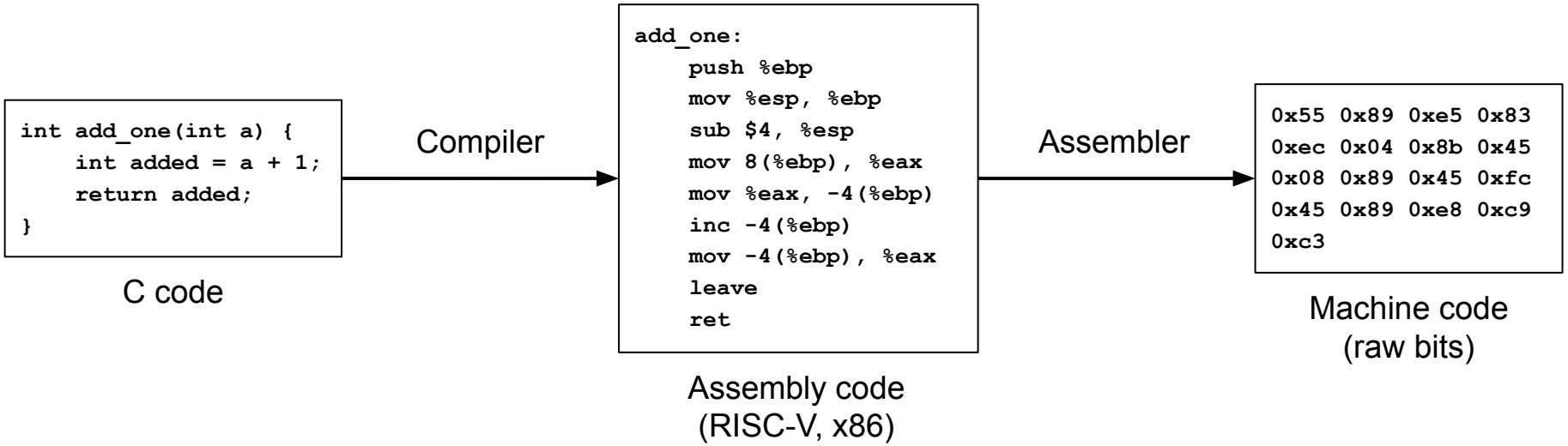
Binary	Hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7

Binary	Hexadecimal
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Running C Programs

Textbook Chapter 2.2

CALL (Compiler, Assembler, Linker, Loader)



CALL (Compiler, Assembler, Linker, Loader)

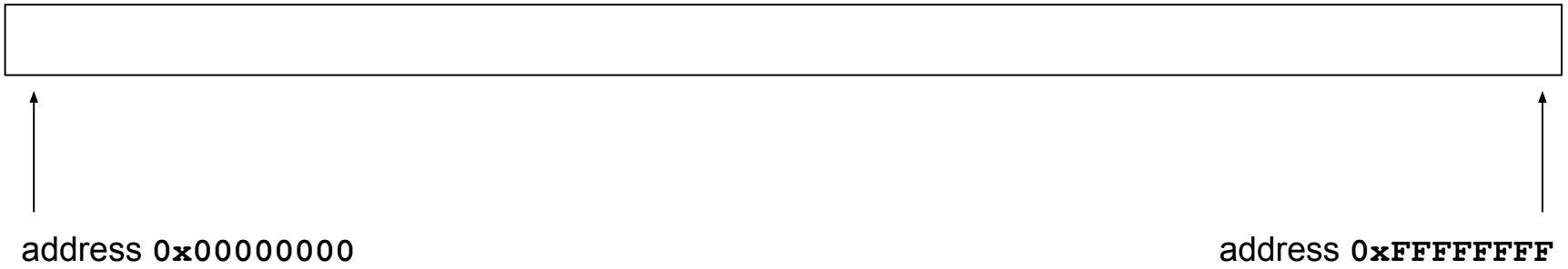
- Compiler: Converts C code into assembly code (RISC-V, x86)
- Assembler: Converts assembly code into machine code (raw bits)
  - Think 61C's RISC-V "green sheet"
- Linker: Deals with dependencies and libraries
  - You can ignore this part for 161
- Loader: Sets up memory space and jumps into the machine code
  - Sometimes, there is an additional linking step right before loading, called dynamic loading
    - Result: The executable doesn't need to contain all the dependencies
    - This also provides additional mitigations, as we'll see later
- After these steps, execution begins, and C runtime library calls **`main`**!

Memory Layout

Textbook Chapter 2.3 & 2.5

C Memory Layout

- At runtime, the loader tells your OS to give your program a big blob of memory
- On a 32-bit system, the memory has 32-bit addresses
  - On a 64-bit system, memory has 64-bit addresses
  - We use 32-bit systems in this class
- Each address refers to one byte, which means you have 2^{32} bytes of memory

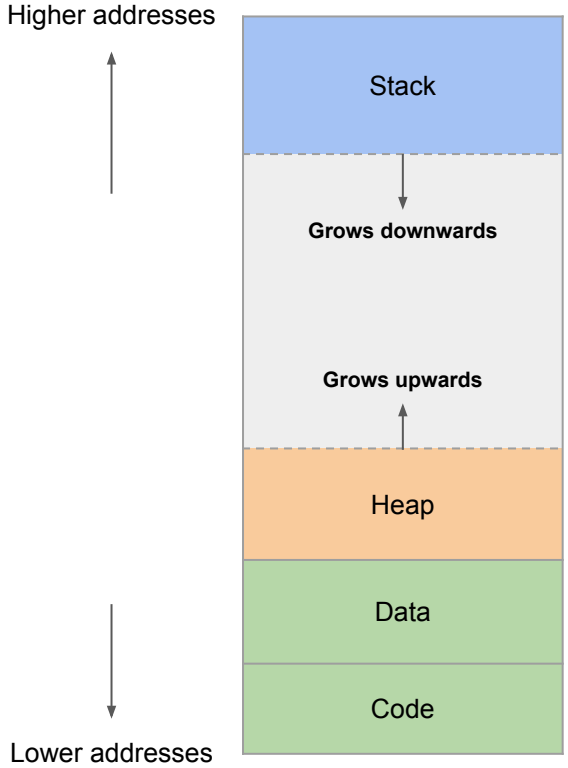


C Memory Layout

- Often drawn vertically for ease of viewing
  - But memory is still just a long array of bytes
- Each process has its own address space (virtual memory)
  - Isolation: Two processes can read and write the same address but access different pieces of physical memory
  - Least privilege: Virtual memory also enables the OS to control what memory a process can “see”

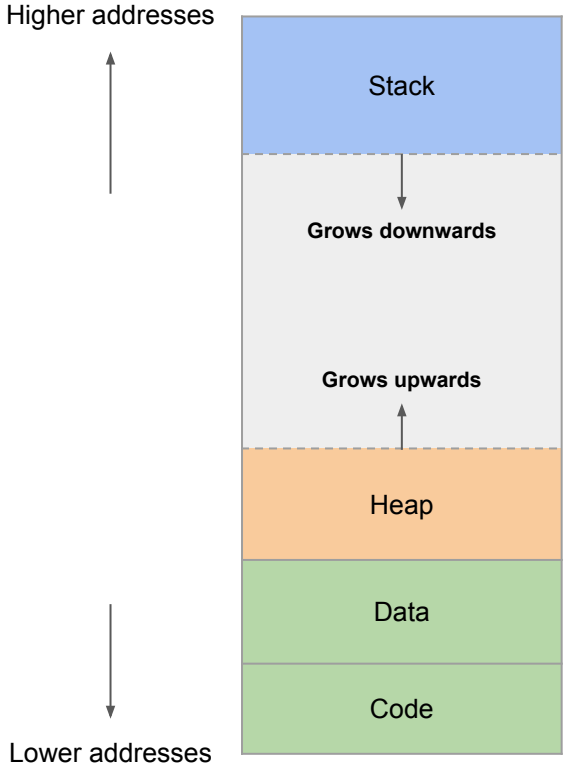
x86 Memory Layout

- Code
  - The program code itself (also called “text”)
- Data
  - Static variables, allocated when the program is started
- Heap
  - Dynamically allocated memory using **`malloc`** and **`free`**
  - As more and more memory is allocated, it grows *upwards*
- Stack:
  - Local variables and stack frames
  - As you make deeper and deeper function calls, it grows *downwards*



Registers

- Recall registers from CS 61C
  - Examples of RISC-V registers: **`a0`**, **`t0`**, **`ra`**, **`sp`**
- Registers are located on the CPU
  - This is different from the memory layout
  - Memory: addresses are 32-bit numbers
  - Registers: addresses are names (**`ebp`**, **`esp`**, **`eip`**)



x86 Architecture

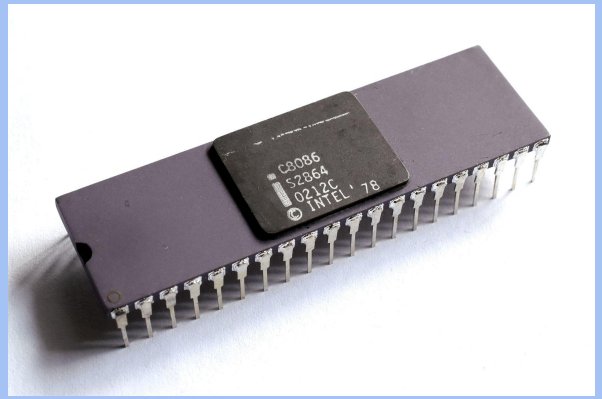
Textbook Chapter 2.4 & 2.7

Why x86?

- It’s the most commonly used instruction set architecture in consumer computers!
  - You are probably using an x86 computer right now… unless you’re on a phone, tablet, or M1 Mac
- You only need enough to be able to read it and know what is going on
  - We will make comparisons to RISC-V, but it’s okay if you haven’t taken 61C and don’t know RISC-V; you don’t need to understand the comparisons to understand x86
- However, if you have a choice, choose ARM
  - Significantly higher performance for the same cost
  - 64b ARM has some unique security features we will discuss later

What is x86?

- Complex instruction set computer (CISC) architecture
  - The opposite of reduced instruction set computer (RISC) architecture
  - There are a lot of instructions… The full ISA manual is over **5,000 pages long**!
  - We will not be teaching the full instruction set (obviously), just enough of the calling convention to be able to do Project 1
- Launched in 1978 with the Intel 8086 processor and eventually took over much of computing
  - Over 40 years ago! The ISA is very bloated because of this…
- 64-bit variant x86-64 launched in 1999 (but we won't study it)



x86 Fact Sheet

- Little-endian
  - The least-significant byte of multi-byte numbers is placed at the first/lowest memory address
  - Same as RISC-V
- Variable-length instructions
  - When assembled into machine code, instructions can be anywhere from 1 to 16 bytes long
  - Contrast with RISC-V, which has fixed-length, 4-byte instructions (or, optionally, 2-byte instructions)

```
int main(void) {
    uint32_t num = 0xdeadbeef;

    // This prints "deadbeef".
    printf("%x", num);

    // This prints "ef be ad de".
    uint8_t *bytes = (uint8_t *) &num;
    for (size_t i = 0; i < 4; i++) {
        printf("%x ", bytes[i]);
    }
}
```

x86 Registers

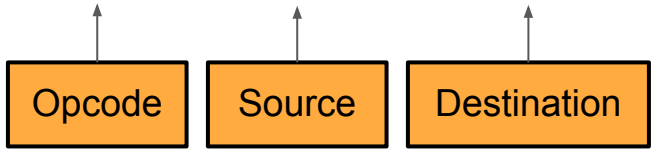
- Storage units as part of the CPU architecture (not part of memory)
- Only 8 main general-purpose registers:
  - EAX, EBX, ECX, EDX, ESI, EDI: General-purpose
  - ESP: Stack pointer (similar to **`sp`** in RISC-V)
  - EBP: Base pointer (similar to **`fp`** in RISC-V)
  - We will discuss ESP and EBP in more detail later
- Instruction pointer register: EIP
  - Similar to PC in RISC-V

x86 Syntax

- Register references are preceded with a percent sign `%`
  - Example: **`%eax`**, **`%esp`**, **`%edi`**
- Immediates are preceded with a dollar sign **`$`**
  - Example: **`$1`**, **`$161`**, **`$0x4`**
- Memory references use parentheses and can have immediate offsets
  - Example: **`8(%esp)`** dereferences memory 8 bytes above the address contained in ESP

x86 Assembly

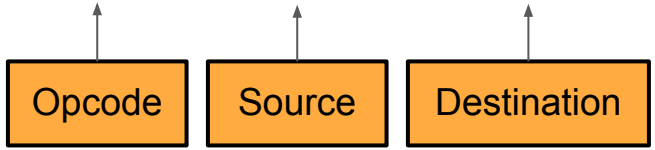
- Instructions are composed of an opcode and zero or more operands.
- `add $0x8, %ebx`



- Pseudocode: **`EBX = EBX + 0x8`**
- The destination comes last
  - Contrast with RISC-V assembly, where the destination (RD) is first
- The **`add`** instruction only has two operands; and the destination is an input
  - Contrast with RISC-V, where the two source operands are separate (RS1 and RS2)
- This instruction uses a register and an immediate

x86 Assembly

- `xor 4(%esi), %eax`



- Pseudocode: **`EAX = EAX ^ *(ESI + 4)`**
- This is a memory reference, where the value at 4 bytes above the address in ESI is dereferenced, XOR'd with EAX, and stored back into EAX
  - Most instructions can be register-register, register-immediate, register-memory, or memory-immediate (but not memory-memory)
    - How can you achieve a memory-memory operation?

Stack Layout

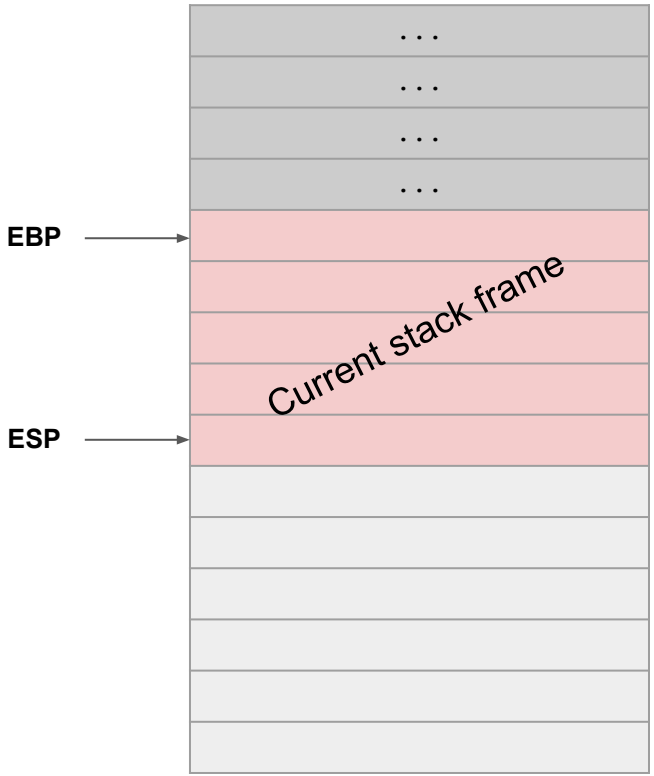
Textbook Chapter 2.6

Stack Frames

- When your code calls a function, space is made on the stack for local variables
  - This space is known as the **stack frame** for the function
  - The stack frame goes away once the function returns
- The stack starts at higher addresses. Every time your code calls a function, the stack makes extra space by growing down
  - Note: Data on the stack, such as a string, is still stored from lowest address to highest address. “Growing down” only happens when extra memory needs to be allocated.

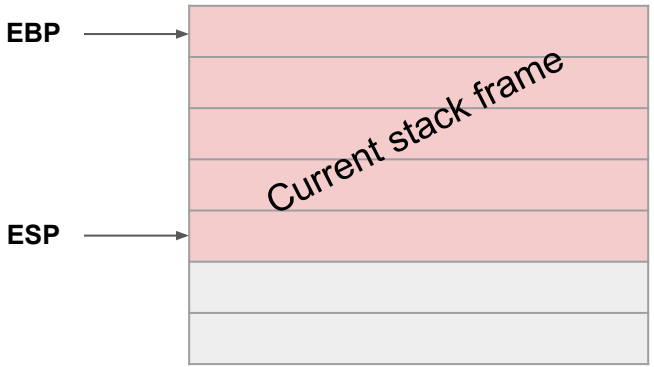
Stack Frames

- To keep track of the current stack frame, we store two pointers in registers
  - The EBP (base pointer) register points to the top of the current stack frame
    - Equivalent to RISC-V **`fp`**
  - The ESP (stack pointer) register points to the bottom of the current stack frame
    - Equivalent to RISC-V **`sp`** (but x86 moves the stack pointer up and down a lot more than RISC-V does)



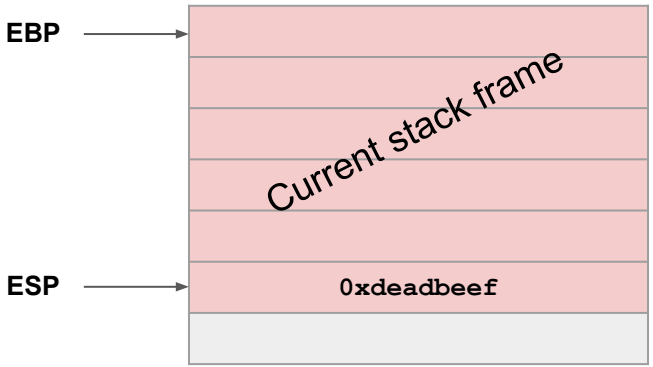
Pushing and Popping

- The **`push`** instruction adds an element to the stack
  - Decrement ESP to allocate more memory on the stack
  - Save the new value on the lowest value of the stack



EAX = `0xdeadbeef`
EBX = ...

Before **`push %eax`**

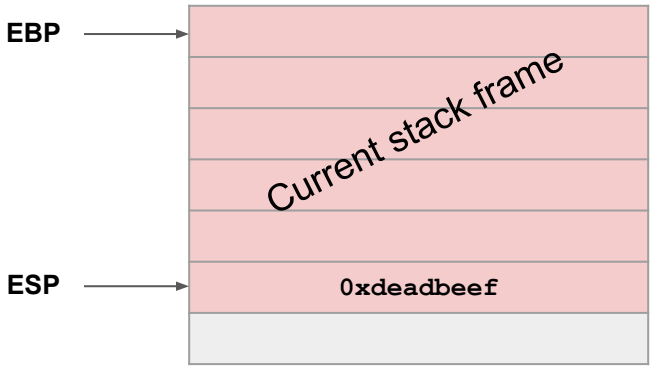


EAX = `0xdeadbeef`
EBX = ...

After **`push %eax`**

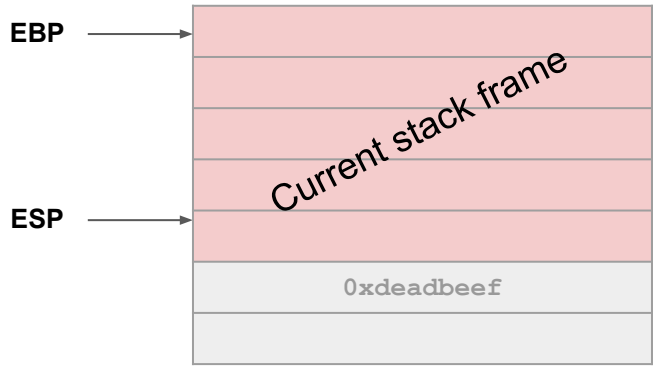
Pushing and Popping

- The **`pop`** instruction removes an element from the stack
  - Load the value from the lowest value on the stack and store it in a register
  - Increment ESP to deallocate the memory on the stack



EAX = `0x00000000`
EBX = ...

Before **`pop %eax`**



EAX = `0xdeadbeef`
EBX = ...

After **`pop %eax`**

Why `push` and `pop`?

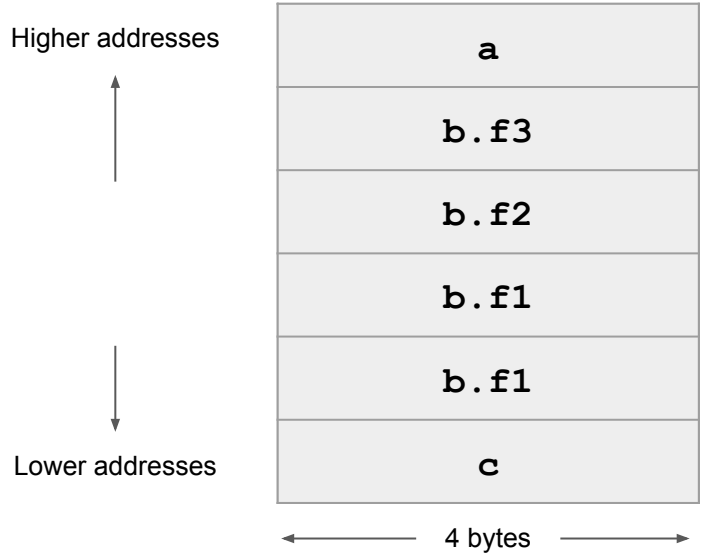
- This allows x86 to use “stack-machine”-style logic
  - Operations always operate on the top items on the stack and push the result back onto the stack
  - Example: *a* + *b* * *c* + *d*
    - `PUSH a // a`
    - `PUSH b // a, b`
    - `PUSH c // a, b, c`
    - `MUL  // a, (b*c)`
    - `ADD  // (a+(b*c))`
    - `PUSH  // (a+(b*c)), d`
    - `ADD  // (a+(b*c))+d`
    - Result is the last remaining item on the stack!
- Used to think this allowed for smaller code and simpler code generation
  - Reality? Not so much. RISC logic is equally efficient in code size in practice, especially with the compressed instructions for RISC-V.
  - But this misconception is why the Java VM is a stack machine too…
- (Not really a) Takeaway: Take 164 if you want to learn more about this

x86 Stack Layout

- Local variables are always allocated on the stack
  - Contrast with RISC-V, which has plenty of registers that can be used for variables
- Individual variables within a stack frame are stored with the first variable at the *highest* address
- Members of a struct are stored with the first member at the *lowest* address
- Global variables are stored with the first variable at the *lowest* address

Stack Layout


```
struct foo {
    long long f1; // 8 bytes
    int f2;       // 4 bytes
    int f3;       // 4 bytes
};

void func(void) {
    int a;        // 4 bytes
    struct foo b;
    int c;        // 4 bytes
}
```



How would you fill out the boxes in this stack diagram?
Options:
a **b.f1** **b.f2** **b.f3** **c**