Computer Systems Security

CSE 466
Fall 2018
Yan Shoshitaishvili

http://pwn.college
http://groups.google.com/group/cse-466
https://goo.gl/F2hs9N

Homework 2 Retrospective

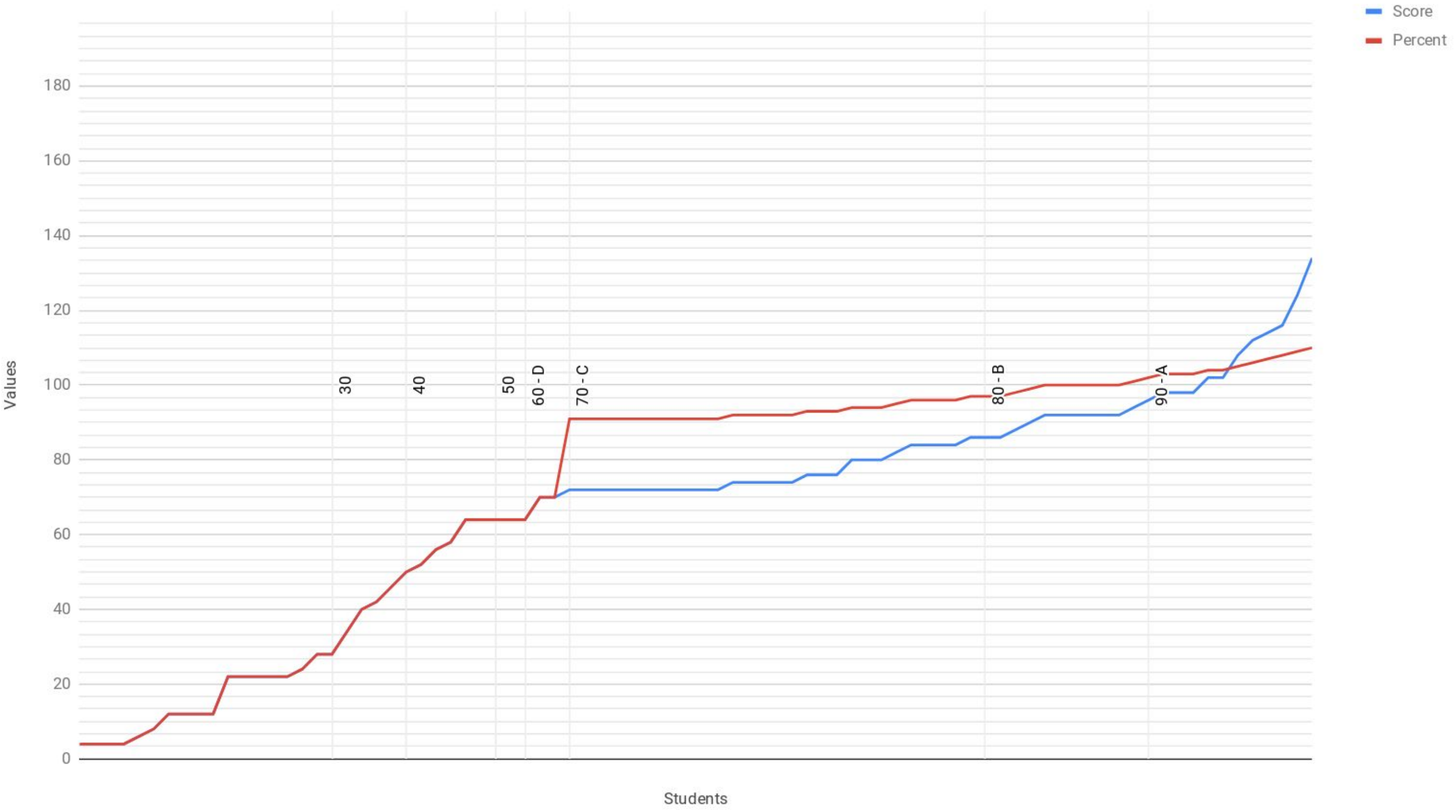
Statistics

88 graded students (+1 auditing)

20 A+ (>= 99%)
31 A (>= 90%)
0 B (>= 80%)
2 C (>= 70%)
5 D (>= 60%)
26 E (< 60%)
4 did not attempt

Because of smaller spread (fewer targets leading to many ties), anyone with over 70 flags got an A.

Hw1 Scores and Percents



Top 3

Master hackers:

1. potato - 134 points
2. blub - 124 points
3. smpwnd - 116 points

How did they do it?

Strategies

Lots of evidence of automation (bot acounts, crazy server load, etc).

- Definitely still useful, but less so.

Looks like people have started having pwntools-based "init" scripts.

- `export TERM`, `SHELL`, etc. before hitting `r.interactive()`

What was the "mastery" hack?

Mastery of the Loading Process

1. File access checks (CAP_DAC_OVERRIDE, then filesystem perms).
2. A new process entry is created.
3. The binary is loaded.
    a. the rise of Position Independent Executables
4. The libraries are located.
    a. LD_PRELOAD environment variable, and anything in /etc/ld.so.preload
        i. (LD_PRELOAD is functionally ignored for setuid binaries)
    b. DT_RPATH specified in the binary file (can be modified with patchelf)
    c. LD_LIBRARY_PATH environment variable (can be set in the shell)
    d. DT_RUNPATH specified in the binary file (can be modified with patchelf)
    e. system-wide configuration (/etc/ld.so.conf)
    f. /lib and /usr/lib
5. The libraries are loaded.
    a. conceptually the same as any other binary (including needing other libraries!)
    b. relocations updated

Mastery of the Loading Process

1. File access checks (CAP_DAC_OVERRIDE, then filesystem perms).
2. A new process entry is created.
3. The binary is loaded.
    a. the rise of Position Independent Executables
4. The libraries are located.
    a. LD_PRELOAD environment variable, and anything in /etc/ld.so.preload
        i. (**LD_PRELOAD is functionally ignored for setuid binaries**)
    b. DT_RPATH specified in the binary file (can be modified with patchelf)
    c. LD_LIBRARY_PATH environment variable (can be set in the shell)
    d. DT_RUNPATH specified in the binary file (can be modified with patchelf)
    e. system-wide configuration (/etc/ld.so.conf)
    f. /lib and /usr/lib
5. The libraries are loaded.
    a. conceptually the same as any other binary (including needing other libraries!)
    b. relocations updated

Mastery of the Loading Process

1. File access checks (CAP_DAC_OVERRIDE, then filesystem perms).
2. A new process entry is created.
3. The binary is loaded.
    a. the rise of Position Independent Executables
4. The libraries are located.
    a. LD_PRELOAD environment variable, and anything in **/etc/ld.so.preload**
        i. (LD_PRELOAD is functionally ignored for setuid binaries)
    b. DT_RPATH specified in the binary file (can be modified with patchelf)
    c. LD_LIBRARY_PATH environment variable (can be set in the shell)
    d. DT_RUNPATH specified in the binary file (can be modified with patchelf)
    e. system-wide configuration (**/etc/ld.so.conf**)
    f. **/lib** and /usr/lib
5. The libraries are loaded.
    a. conceptually the same as any other binary (including needing other libraries!)
    b. relocations updated

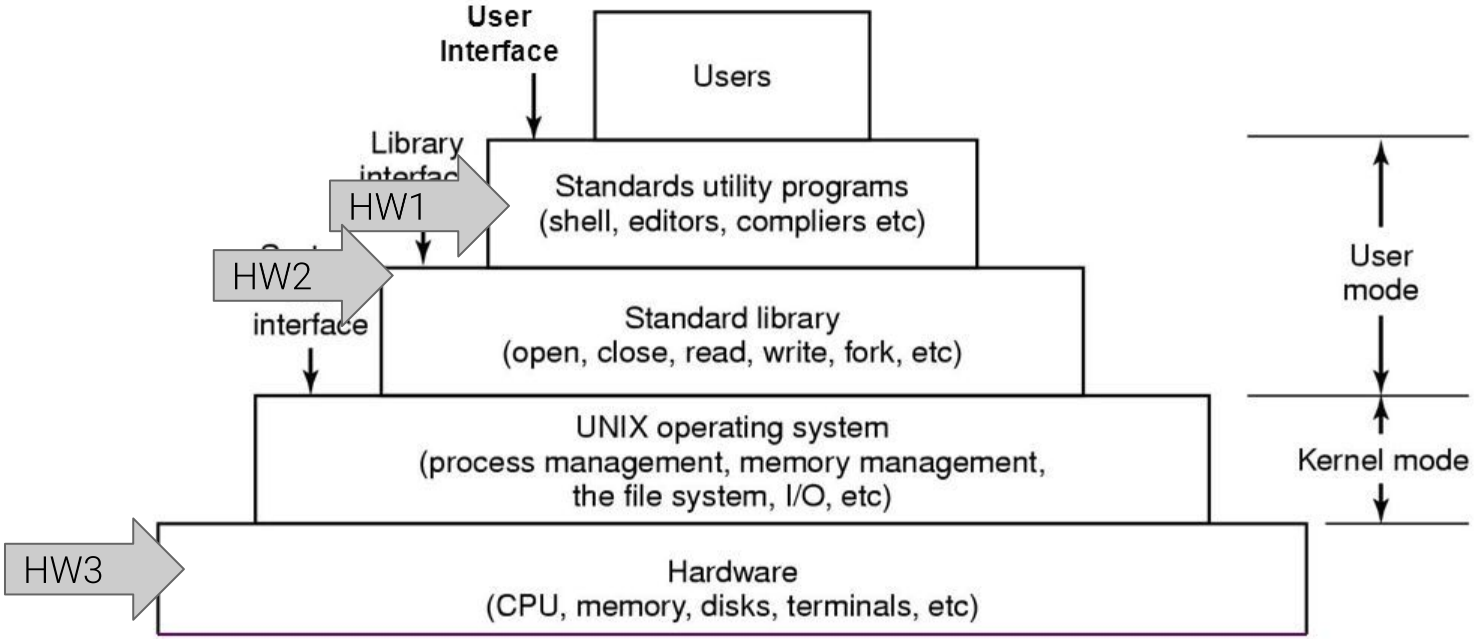
Mastery

Several options to turn *any* arbitrary write into arbitrary execution:

1. Change /etc/ld.so.conf to search somewhere where you put a decoy libc.
    a. Need to compile your patched "evil" libc.
    b. Doable, but tedious.
2. Overwrite /lib/x86_64-linux-gnu/libc.so.6
    a. Just as tedious.
3. Change /etc/ld.so.preload to preload a simple library that leaks the flag.
    a. /etc/ld.so.preload *DOES* work with SUID binaries.
    b. Props to: blub, with awesome solves such as `ed`, `tee`, `cp`, and others

CSE 466 Week 3

Binary Brillaince

User
Interface
Users
Library
interfac
HW1
Standards utility programs
(shell, editors, compliers etc)
HW2
interface
Standard library
(open, close, read, write, fork, etc)
UNIX operating system
(process management, memory management,
the file system, I/O, etc)
HW3
Hardware
(CPU, memory, disks, terminals, etc)
User
mode
Kernel mode

cat /flag

1. A process is created.
2. Cat is loaded.
    a. binary
    b. shared libraries
3. Cat is initialized.
    a. shared library initializers
    b. binary initializers
4. **Cat is launched.**
    a. __libc_start_main()
    b. main()
5. **Cat reads its arguments and environment.**
6. **Cat does its thing:**
    a. opens /flag
    b. reads the data
    c. writes the data to stdout
7. Cat terminates.

Software Runs on the CPU

All software eventually runs on your CPU, as low-level machine instructions.

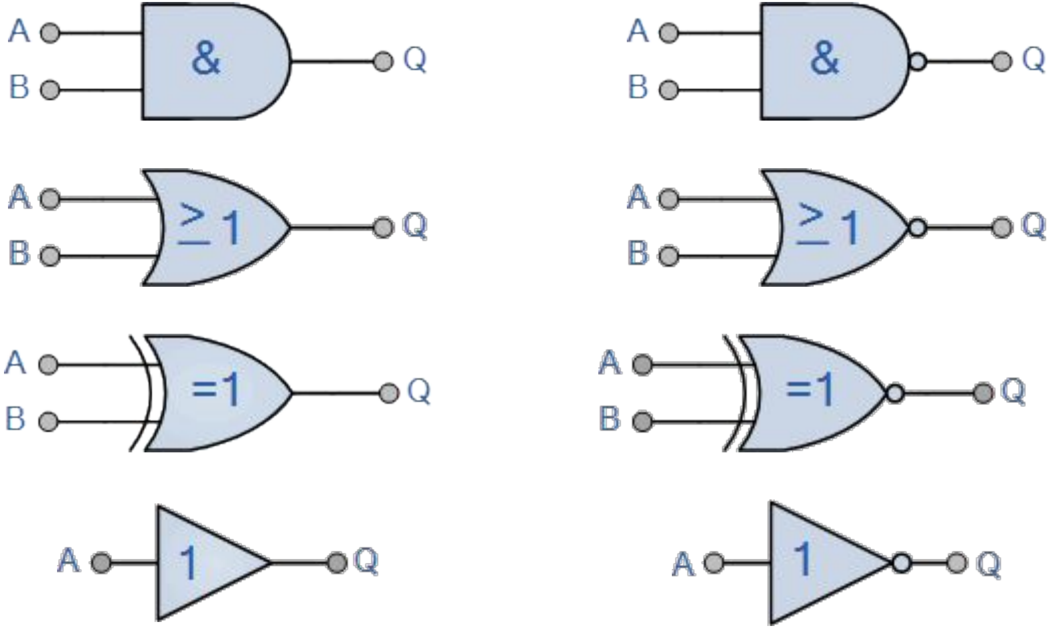
Programs written in compiled languages (C, C++, Rust, etc) run the machine code that they ship as.

Programs written in interpreted languages (Python, JavaScript, etc) are either JIT-compiled immediately before being run, or run by a compiled interpreter (in fancy cases, this interpreter is JIT-compiled itself).

To truly understand how all of this crazy stuff works, we need to understand what happens at the low level!

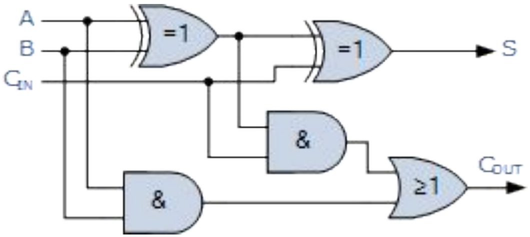
P.S. Your Computer Organization class should have taught you all of this.

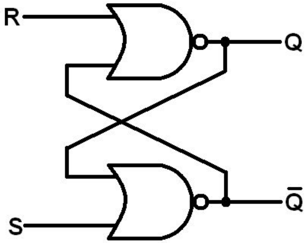
Where do binary files go to?

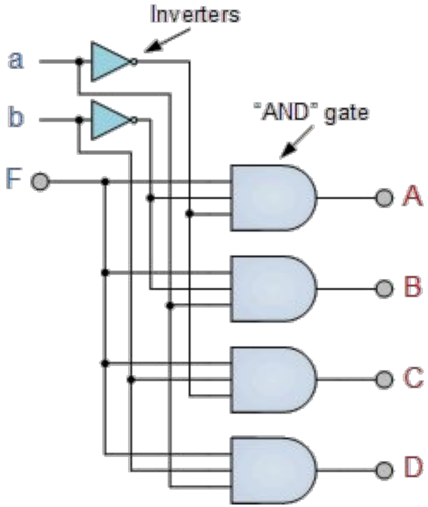


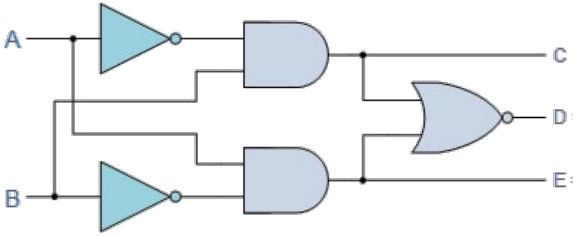
http://www.electronics-tutorials.ws/logic/logic_1.html

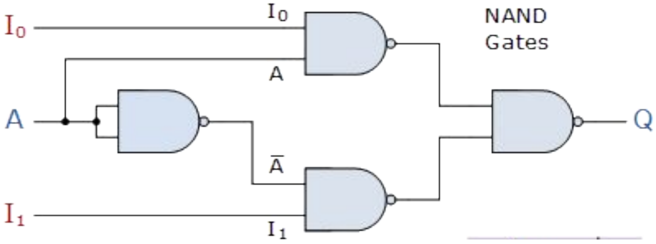
All our powers combined...





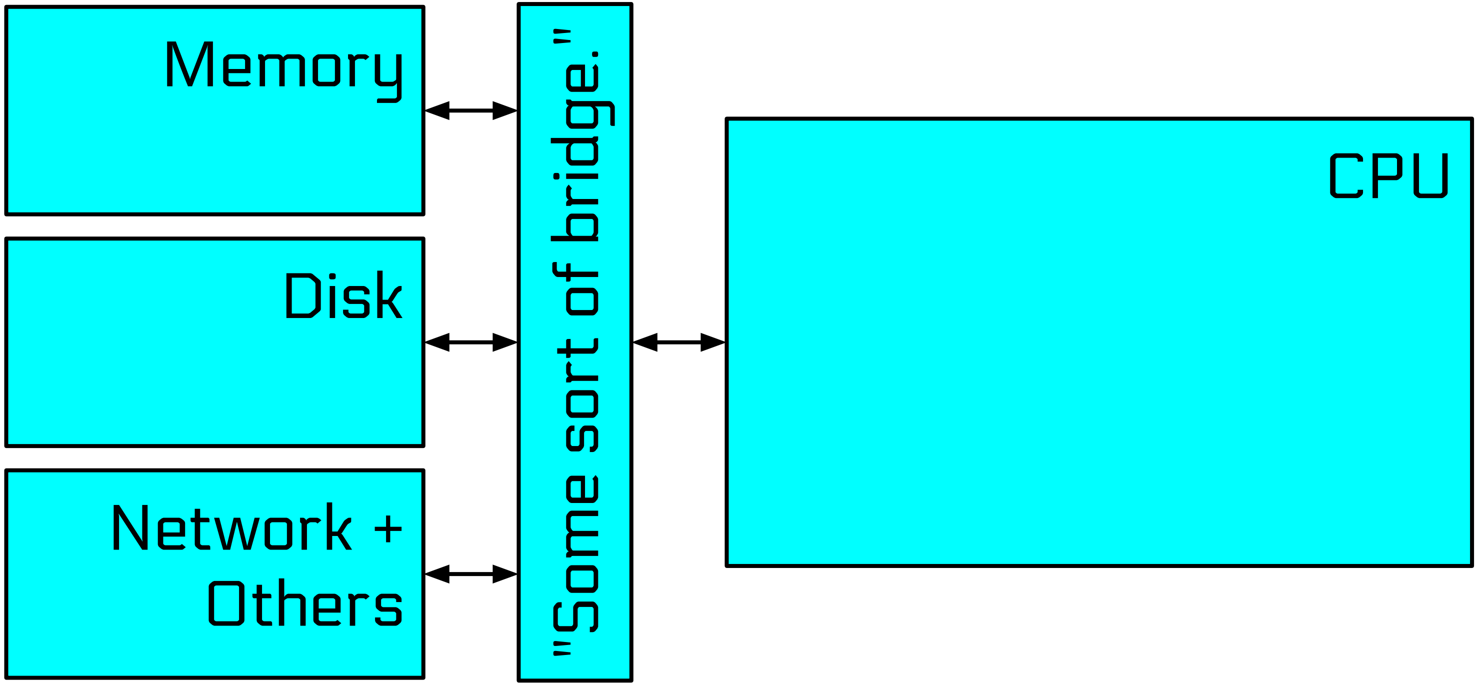




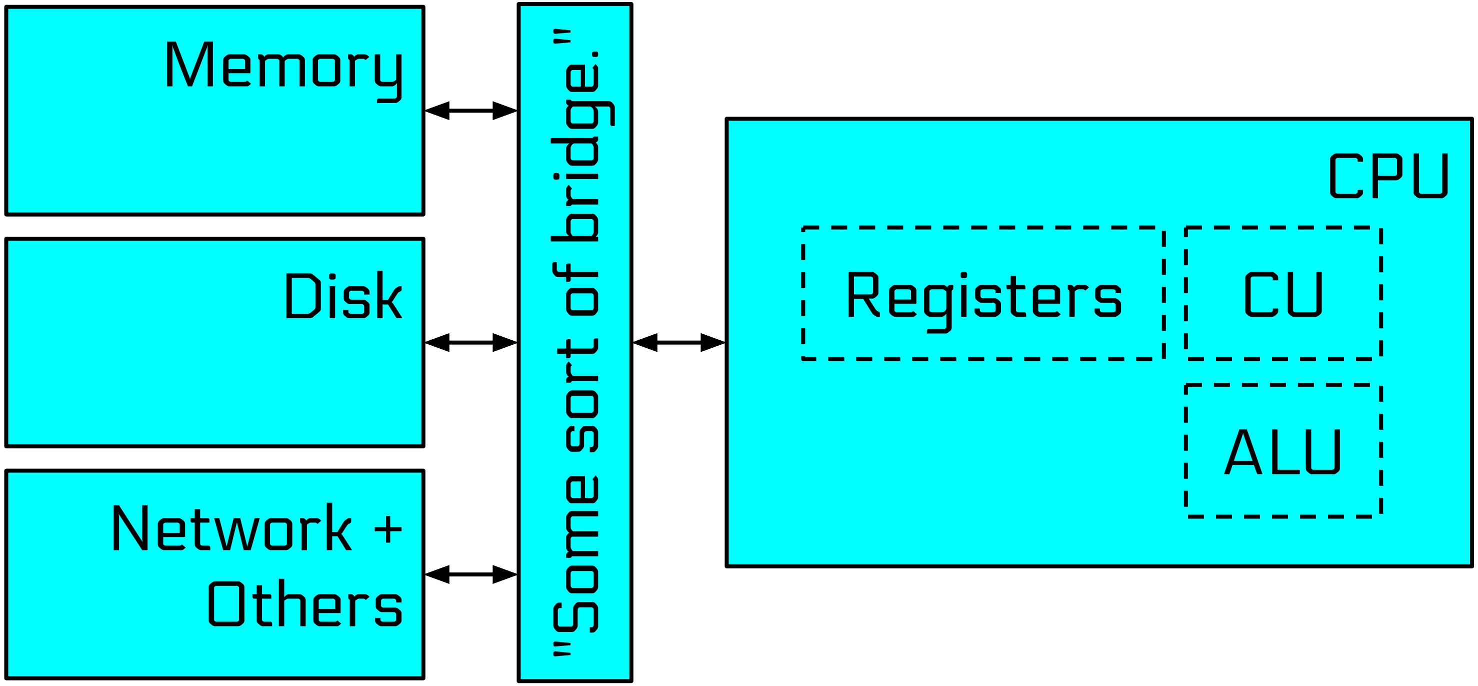


http://www.electronics-tutorials.ws/category/combination

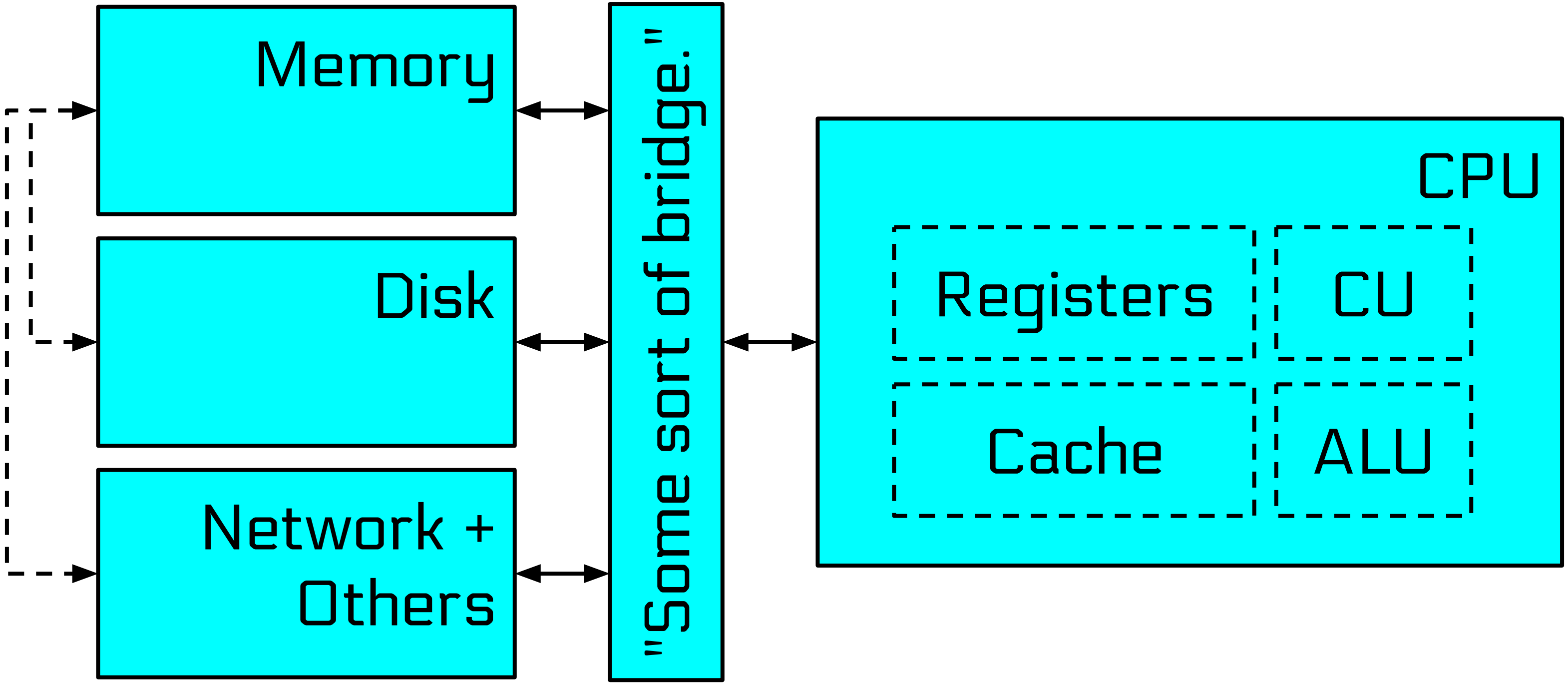
Computer Architecture (at a very high level)



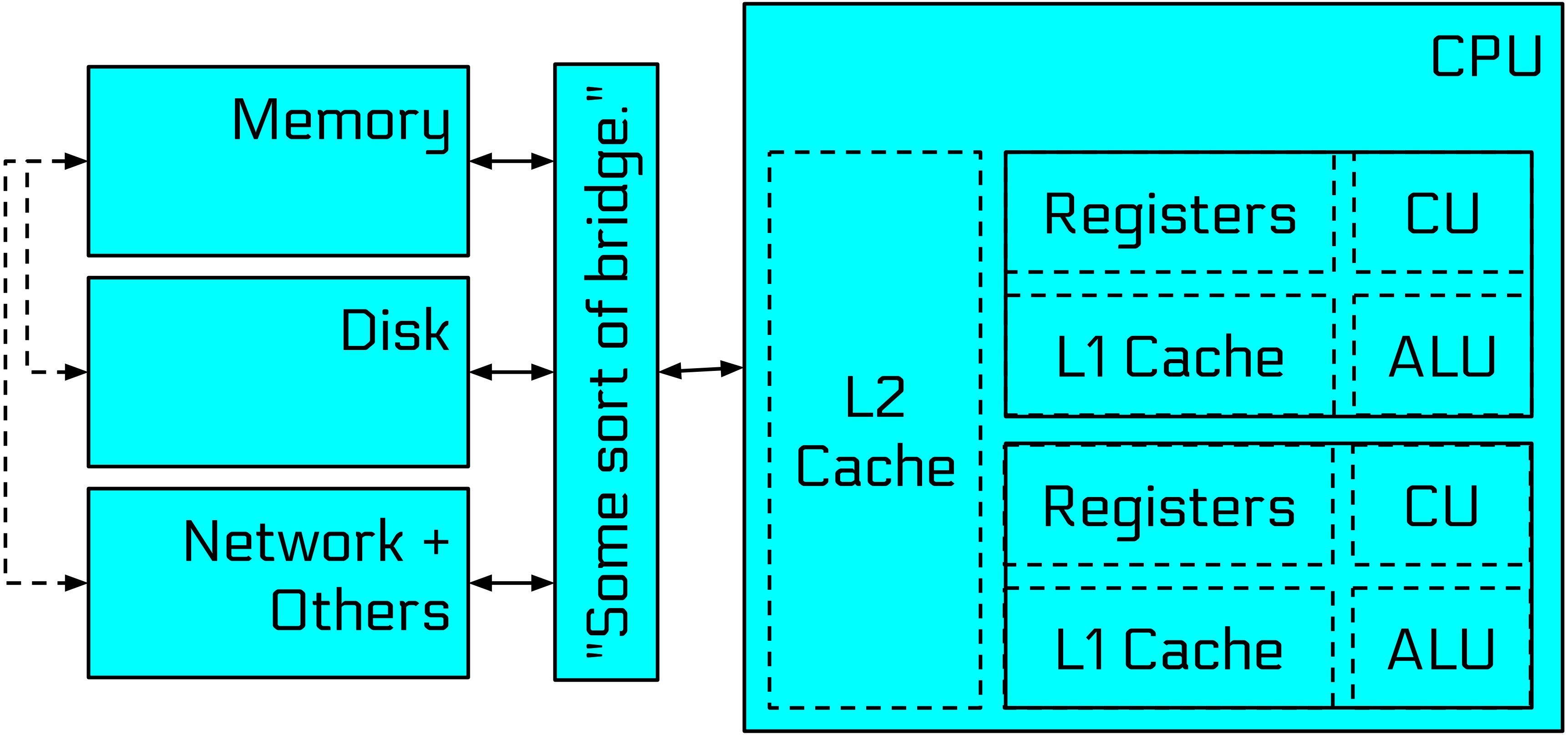
Computer Architecture (drilling down)

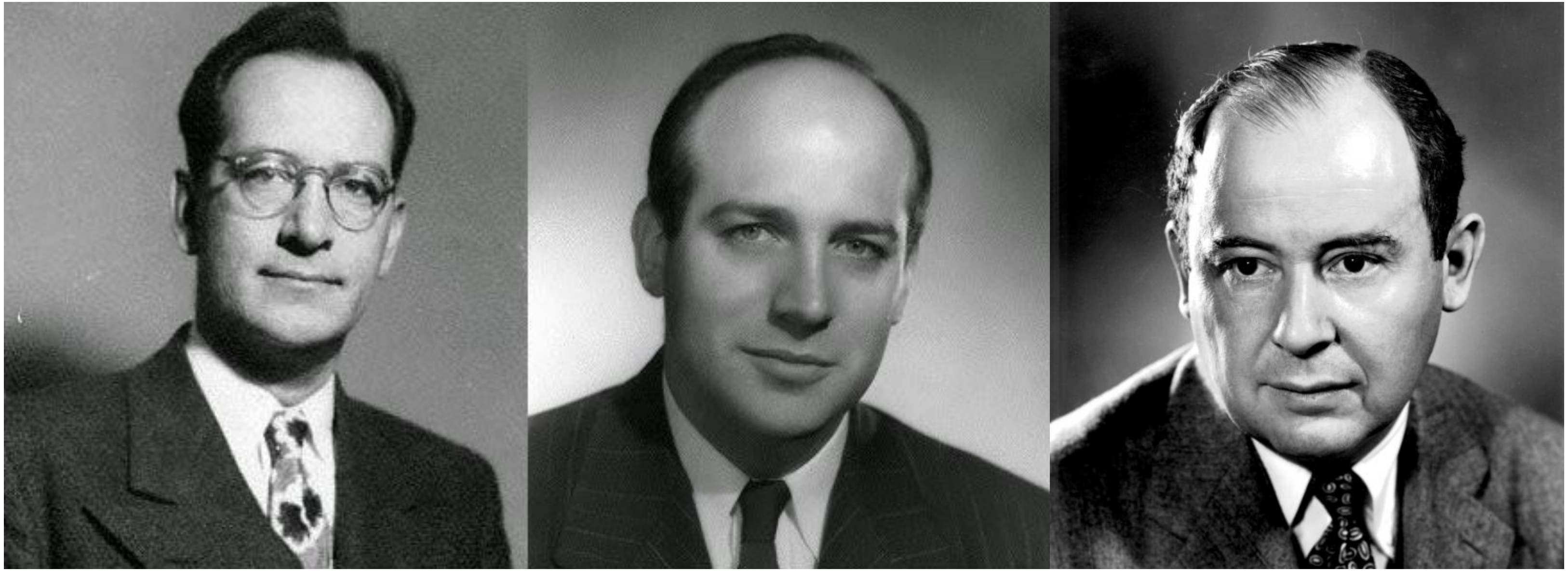


Computer Architecture (further down!)



Computer Architecture (as far as we'll go)





John Mauchly (Physicist), John Presper Eckert (Electrical Engineer), John Von Neumann (Mathematician)

`John von Neumann, First Draft of a Report on the EDVAC, 1945.`

Assembly

The only *true* programming language, as far as a CPU is concerned.

Concepts:

- registers
- memory
- instructions
- stack
- heap

Registers

Registers are very fast, temporary stores for data.

> x86: eax, ecx, edx, ebx, **esp**, **ebp**, esi, edi
>
> amd64: rax, rcx, rdx, rbx, **rsp**, **rbp**, rsi, rdi, r8, r9, r10, r11, r12, r13, r14, r15
>
> arm: r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, **r13**, **r14**

The address of the next instruction is in a register:

> eip (x86), rip (amd64), r15 (arm)

Various extensions add other registers (x87, MMX, SSE, etc).

Instructions

General form:

```
    OPCODE OPERAND OPERAND, ...
```

`OPCODE` - what to do

`OPERANDS` - what to do it on/with

```
mov rax, rbx
add rax, 1
cmp rax, rbx
jb some_location
```

`Useful reference:` `http://ref.x86asm.net`

Instructions (control flow)

Control flow is determined by conditional and unconditional jumps.

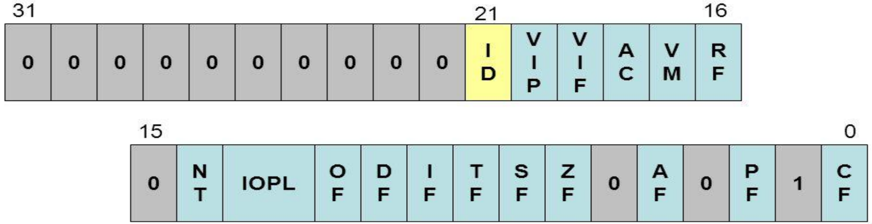
Unconditional: call, jmp, ret

Conditional:

je	jump if equal
jne	jump if not equal
jg	jump if greater
jl	jump if less
jle	jump if less than or equal
jge	jump if greater than or equal
ja	jump if above (unsigned)
jb	jump if below (unsigned)
jae	jump if above or equal (unsigned)
jbe	jump if below or equal (unsigned)
js	jump if signed
jns	jump if not signed
jo	jump if overflow
jno	jump if not overflow
jz	jump if zero
jnz	jump if not zero

Instructions (conditionals)

Conditionals key off of the "flags" register: eflags (x86), rflags (amd64), aspr (arm)



je	jump if equal	ZF=1
jne	jump if not equal	ZF=0
jg	jump if greater	ZF=0 and SF=OF
jl	jump if less	SF!=OF
jle	jump if less than or equal	ZF=1 or SF!=OF
jge	jump if greater than or equal	SF=OF
ja	jump if above (unsigned)	CF=0 and ZF=0
jb	jump if below (unsigned)	CF=1
jae	jump if above or equal (unsigned)	CF=0
jbe	jump if below or equal (unsigned)	CF=1 or ZF=1
js	jump if signed	SF=1
jns	jump if not signed	SF=0
jo	jump if overflow	OF=1
jno	jump if not overflow	OF=0
jz	jump if zero	ZF=1
jnz	jump if not zero	ZF=0

Updated by (x86/amd64):

- arithmetic operations
- cmp - subtraction (`cmp rax, rbx`)
- test - and (`test rax, rax`)

Memory (stack)

The stack fulfils four main uses:

1. Track the "callstack" of a program.
    a. return values are "`pushed`" to the stack during a call and "`popped`" during a ret
2. Contain local variables of functions.
3. Provide scratch space (to alleviate register exhaustion).
4. Pass function arguments (always on x86, only for functions with "many" arguments on other architectures).

Relevant registers (amd64): rsp, rbp

Relevant instructions (amd64): push, pop

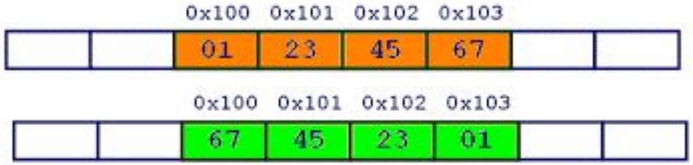
Memory (heap)

As we discussed last week, the heap is a libc-managed memory region from which you can allocate (malloc) and deallocate (free) memory.

It doesn't come into play heavily yet, but will feature prominently in future assignments.

Memory (endianess)

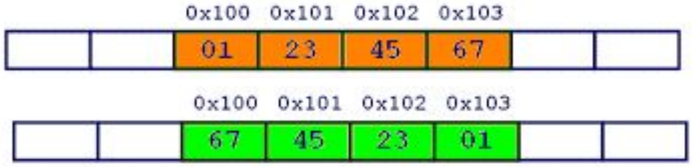
Data on most modern systems is stored *backwards*.



Why?

Memory (endianess)

Data on most modern systems is stored *backwards*.



		0x100	0x101	0x102	0x103		
		01	23	45	67		

		0x100	0x101	0x102	0x103		
		67	45	23	01		

Why?

- Performance (historical)
- Ease of addressing for different sizes.
- (apocryphal) 8086 compatibility

Calling conventions:

Callees and caller functions must agree on argument passing.

Linux x86: push arguments (in reverse order), then call (which pushes return address), return value in eax

Linux amd64: rdi, rsi, rdx, rcx, r8, r9, return value in rax

Linux arm: r0, r1, r2, r3, return value in r0

Registers are *shared* between functions, so calling conventions agree on what registers are protected.

Linux amd64: rbx, rbp, r12, r13, r14, r15 are "callee-saved"

Educational Resources

Rappel (https://github.com/yrp604/rappel) lets you explore the effects of instructions.

- easily installable via https://github.com/zardus/ctf-tools

pwndevils how2hack:

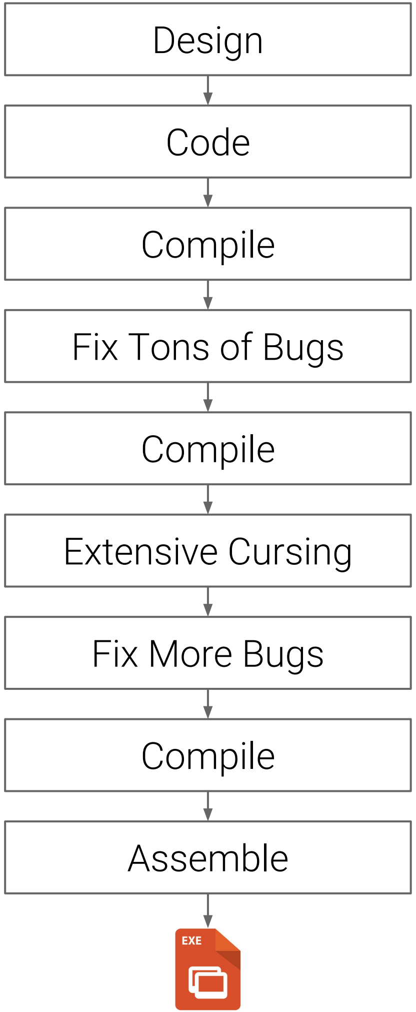
- http://pwndevils.com/hacking/howtwohack.html
- web interface to step-by-step understanding of x86
- NOTE: we are using x86_64 today, but many of the concepts are transferable

The Forward Engineering Process

"Forward Engineering" is an overloaded term, but in this context, it is the process of building a program.

1. Figure out what you want to code.
2. Code it.
3. Compile it (can include JIT).
4. Run it.

At every step, information is lost!



Design

Code

Compile

Fix Tons of Bugs

Compile

Extensive Cursing

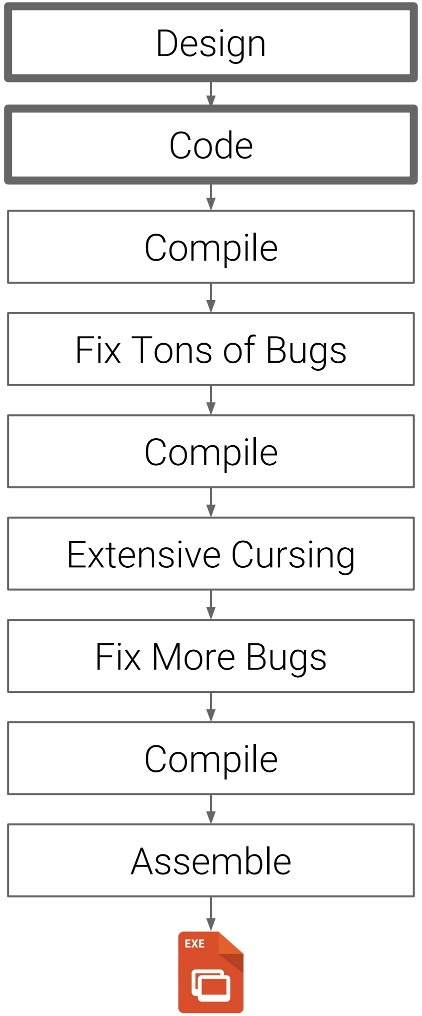
Fix More Bugs

Compile

Assemble

Discussion

What is lost in the transition between Design and Code?



Design

Code

Compile

Fix Tons of Bugs

Compile

Extensive Cursing

Fix More Bugs

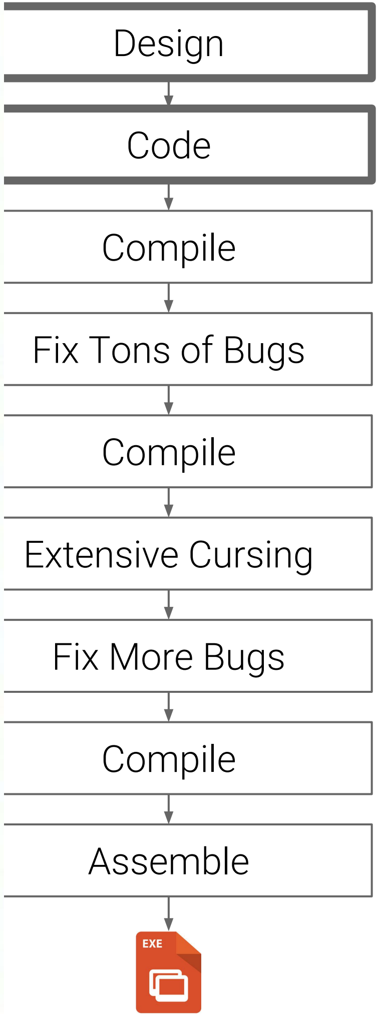
Compile

Assemble

Discussion

What is lost in tl

Oh God!! this code is horrible! who wrote this crap?!
shit, it was me

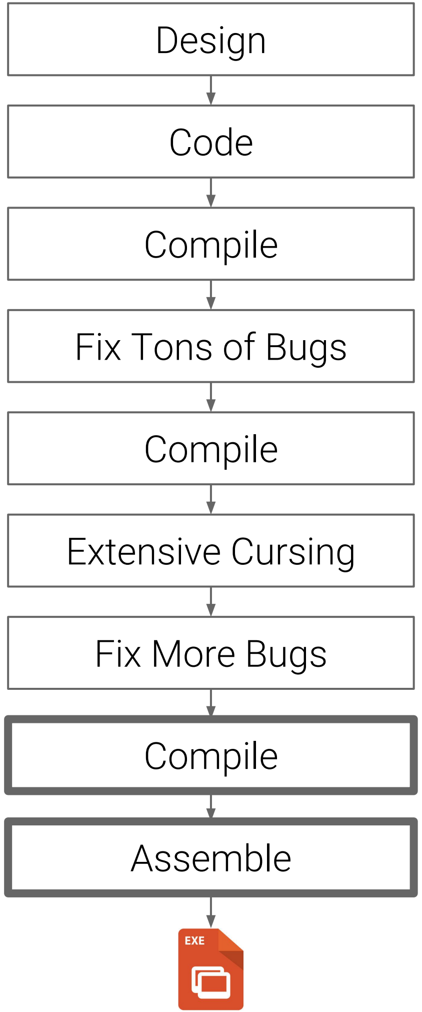


Discussion

What is lost in the compilation process?

- Comments
- Variable names.
- Function names.
- Structure (classes, structs, etc) data.
- Sometimes, entire algorithms (optimization)!

How do we get it back?



Design

Code

Compile

Fix Tons of Bugs

Compile

Extensive Cursing

Fix More Bugs

Compile

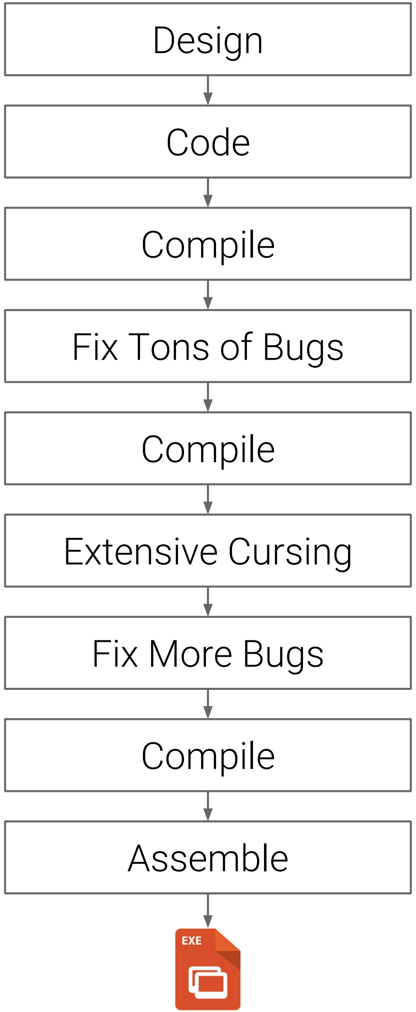
Assemble

Forward Engineering Tools

Let's look at some tools:

- Your IDE (we will not look at this).
- The GNU C Compiler.
- The GNU Assembler.
  - other assemblers
- The GNU Linker.

Viola! An ELF is born.



Design

Code

Compile

Fix Tons of Bugs

Compile

Extensive Cursing

Fix More Bugs

Compile

Assemble

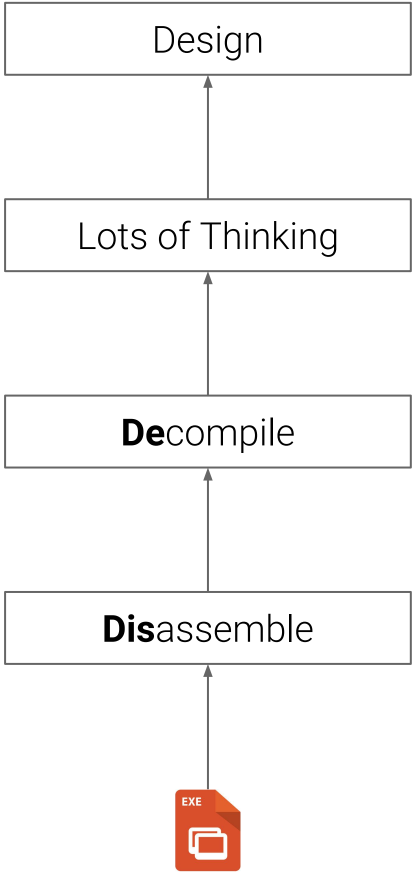
The **Reverse** Engineering Process

Every step in the reverse-engineering process is imperfect and relies on some amount of human help.

The focus of this class: how do we reverse the design from the binary?

Available tools:

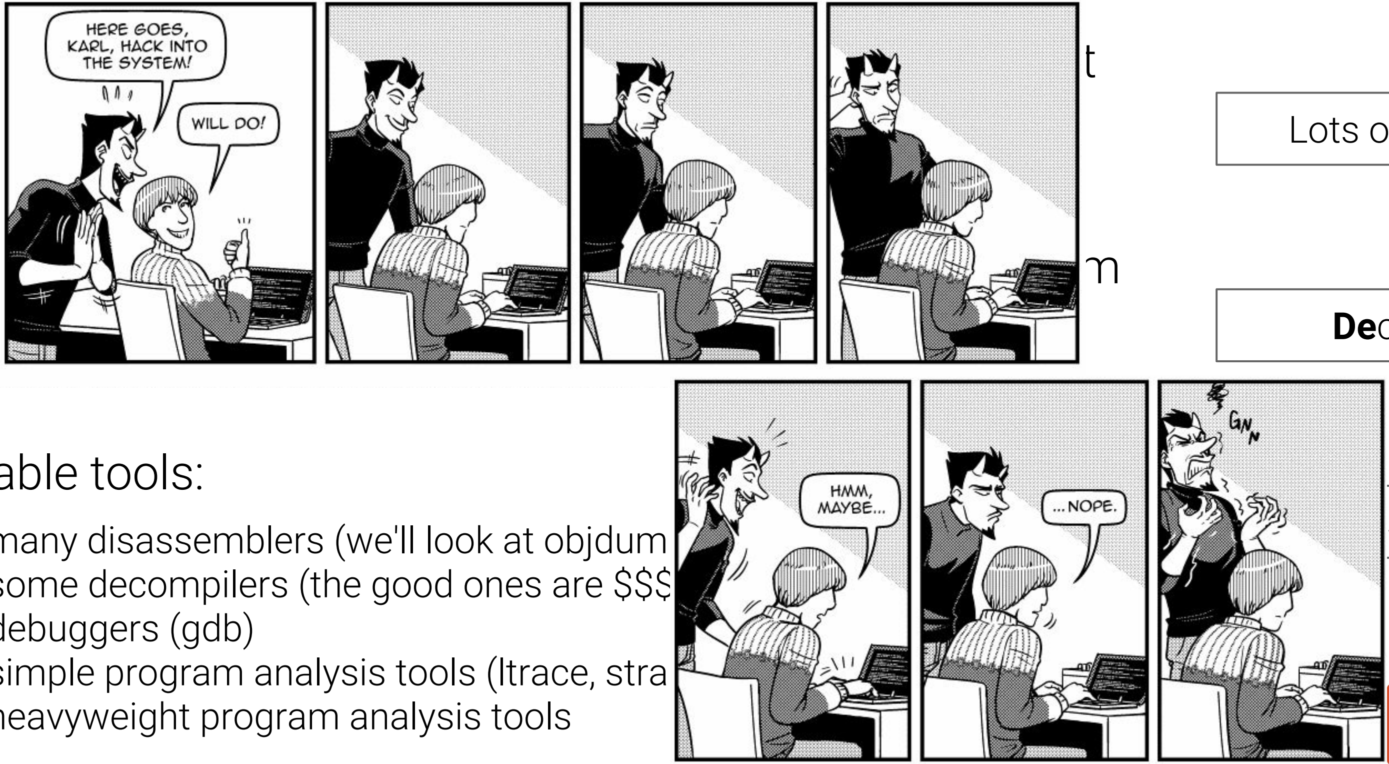
- many disassemblers (we'll look at objdump)
- some decompilers (the good ones are $$$$$$)
- debuggers (gdb)
- simple program analysis tools (ltrace, strace)
- heavyweight program analysis tools



The Reverse Engineering Process

Every t
and r

The f m
the b



Available tools:

- many disassemblers (we'll look at objdum
- some decompilers (the good ones are $$$
- debuggers (gdb)
- simple program analysis tools (ltrace, stra
- heavyweight program analysis tools

Design

Lots of Thinking

Decompile

ssemble

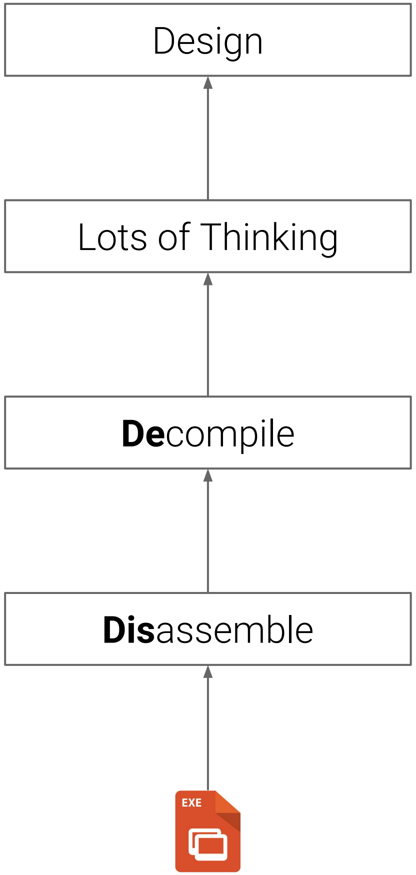
EXE

Reverse Engineering - Simple Example

Let's look at a simple "crackme" demo.

Crackmes are small programs designed to test a hacker's reverse engineering skillz.

Inspired by real-world license key verification systems.



Design

Lots of Thinking

Decompile

Disassemble

Crackme: https://en.wikipedia.org/wiki/Crackme
Many many crackmes: https://crackmes.one

License Checkers

Back in the stone age, before every cave had internet access, software had to be installable without internet access.

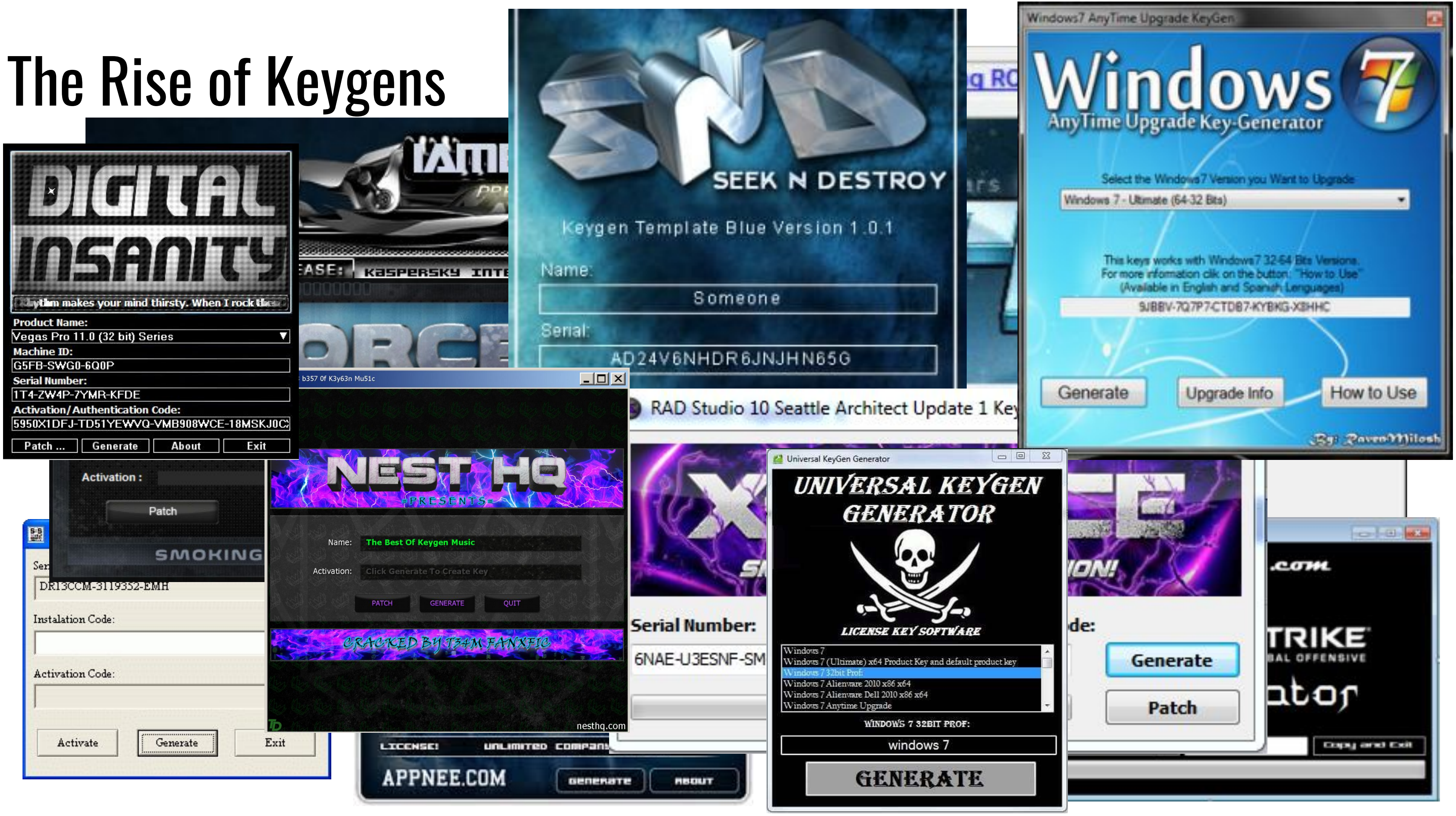
How does the software ensure that it is legitimate?

License key checks!

- Have a secret algorithm that takes in input, performs some calculation, and validates the result.
- Ideally, the company selling the software can generate multiple valid keys.
- Ideally, pirates cannot generate valid keys.

This method implicitly truststhe binary code to keep the secret license key verification algorithm safe!

The Rise of Keygens



Product Name: Vegas Pro 11.0 (32 bit) Series

Machine ID: G5FB-SWG0-6Q0P

Serial Number: 1T4-ZW4P-7YMR-KFDE

Activation/Authentication Code: 5950X1DFJ-TD51YEWVQ-VMB908WCE-18MSKJ0C

Keygen Template Blue Version 1.0.1

Name: Someone

Serial: AD24V6NHDR6JNJHN65G

Windows7 AnyTime Upgrade KeyGen

Select the Windows7 Version you Want to Upgrade: Windows 7 - Ultimate (64-32 Bits)

This keys works with Windows7 32-64 Bits Versions.
For more information clik on the button: "How to Use"
(Available in English and Spanish Languages)

9JBBV-7Q7P7-CTDB7-KYBKG-X8HHC

Name: The Best Of Keygen Music

Activation: Click Generate To Create Key

RAD Studio 10 Seattle Architect Update 1 Key

Serial Number: 6NAE-U3ESNF-SM

Universal KeyGen Generator

Windows 7
Windows 7 (Ultimate) x64 Product Key and default product key
Windows 7 32bit Prof.
Windows 7 Alienware 2010 x86 x64
Windows 7 Alienware Dell 2010 x86 x64
Windows 7 Anytime Upgrade

WINDOWS 7 32BIT PROF: windows 7

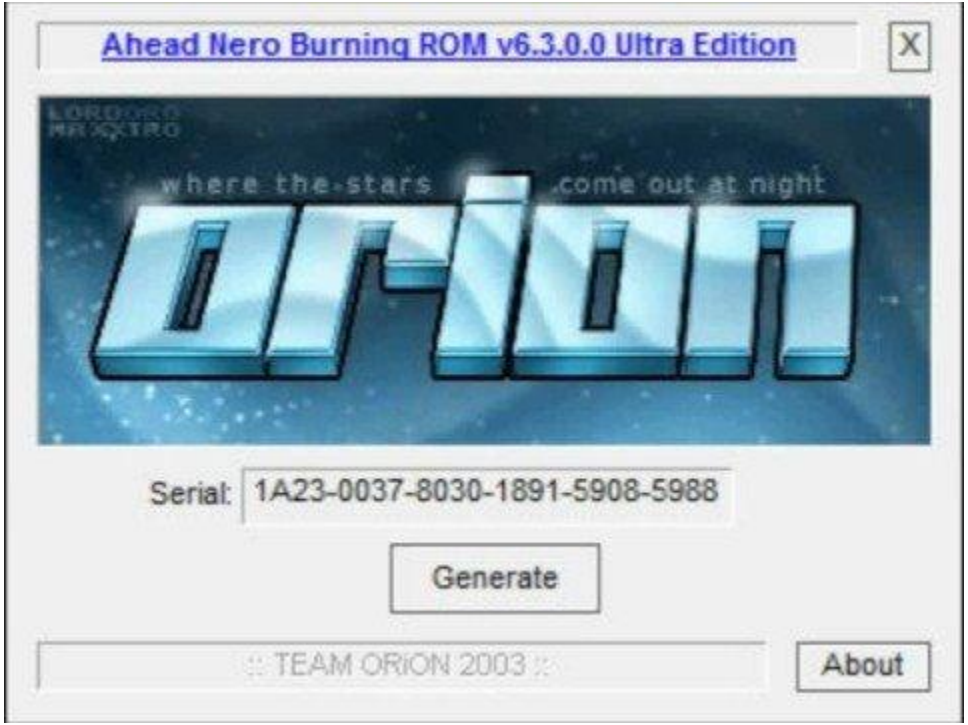
Ser DR13CCM-3119352-EMH

Instalation Code:

Activation Code:

Activation :

The Rise of Keygens



Ahead Nero Burning ROM v6.3.0.0 Ultra Edition

Serial: 1A23-0037-8030-1891-5908-5988

:: TEAM ORiON 2003 ::

Alternatives to Keygens

1. Cracks that patch the executable to remove the check altogether.
2. Legally purchasing the software.

With ubiquitous internet access (and increased ability to have unbreakable server-side license key checks), keygenning diminished in viability.

But we will explore this lost art in Homework 3!

Homework 3 - Binary Brilliance

Unlike homework 1 and 2, we're diving *in-depth* into the inner workings of binaries, rather than just learning how to use them. It's time to reverse engineer!

The homework:

- netcat to cse466.pwn.college, port 23 (the password will be posted on the class mailing list)
- enter your hacker alias (you can choose a new one) and ASU ID.
- Choose one of your personalized 132 challenges (in /pwn) to attempt. It will be made SUID, and will be able to call "/get_flag".
- If you give that challenge the correct input, it will call "/get_flag" and give you the "/flag"!
- Log out ("exit" command).
- Provide the flag when asked.
- Scoring is done automatically.
- **2 points** per challenge.

Useful Reversing Tools (and demo)

For simple crackmes, ltrace might be sufficient!

- more info than strace, but still not enough for anything complex

Running the program with multiple different inputs might get you farther.

- ltrace it with input A
- ltrace it again with input B
- see if you can reverse the algorithm from looking at input and output
- still does not scale to complex algorithms

Others:

- gdb (and gdb scripting!)
  - The authority on gdb: https://sourceware.org/gdb/onlinedocs/gdb/
- objdump

Useful Reversing Tools (advanced)

This assignment is very doable with ltrace, objdump, and gdb, and I recommend you take that route! That being said, the "magic bullets" for this assignment are advanced program analysis tools.

angr (full disclosure: I am the cofounder of this project)

- advanced program analysis suite that lends itself quite nicely to crackmes
- hard to install (though there is a docker image!), and hard to use
    - you'll probably have to understand how to solve things manually before you can angr them

Dynamic analysis tools:

- https://github.com/zardus/ctf-tools has helpful install scripts for tools
- taintgrind will tell you what the program is doing with your input