# Assembly Crash Course

## Building Programs

Yan Shoshitaishvili
Arizona State University

# From Assembly to Binary

We built a quitter... Now we have to put it in an Assembly file:

```
# .intel_syntax tells the assembler that we are using Intel assembly syntax
# noprefix tells it that we will not prefix all register names with "%" (cause that looks silly)
.intel_syntax noprefix
mov rdi, 42 # our program's return code (e.g., for bash scripts)
mov rax, 60 # system call number of exit()
syscall # do the system call
```

Assembly is named after the Assembler. Let's use the assembler!

```
yans@ramoth ~/pwn $ gcc -nostdlib -o quitter quitter.s
/usr/bin/ld: warning: cannot find entry symbol _start; defaulting to 0000000000001000
yans@ramoth ~/pwn $ file quitter
quitter: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/
ld-linux-x86-64.so.2, BuildID[sha1]=31b3e4db70dd678441e67d155d58972d7f205777, not stripped
```

If that warning from ld annoys you, add this to the beginning of the program so that gcc doesn't have to guess at where your code starts:

```
.global _start
_start:
# then the rest of your code!
```

You've built your first assembly program!

# Running the Program

Your program runs like any other...

```
# ./quitter
```

You can check its return code with bash's special $? variable!

```
# ./quitter
# echo $?
42
```

# Reading Assembly

You can *disassemble* your program!

`# objdump -M intel -d quitter`

```
yans@ramoth ~/pwn $ objdump -M intel -d quitter

quitter:     file format elf64-x86-64


Disassembly of section .text:

0000000000001000 <start>:
    1000:       48 c7 c7 2a 00 00 00    mov    rdi,0x2a
    1007:       48 c7 c0 3c 00 00 00    mov    rax,0x3c
    100e:       0f 05                   syscall
```

# Extracting the Binary Code

gcc builds your Assembly into a full ELF program.

You can extract *just* your binary code:

```
# objcopy --dump-section .text=quitter_binary_code quitter
```

```
yans@ramoth ~/pwn $ objdump -M intel -d quitter

quitter:     file format elf64-x86-64


Disassembly of section .text:

0000000000001000 <start>:
    1000:       48 c7 c7 2a 00 00 00    mov    rdi,0x2a
    1007:       48 c7 c0 3c 00 00 00    mov    rax,0x3c
    100e:       0f 05                   syscall
yans@ramoth ~/pwn $ objcopy --dump-section .text=quitter_binary_code quitter
yans@ramoth ~/pwn $ hd quitter_binary_code
00000000  48 c7 c7 2a 00 00 00 48  c7 c0 3c 00 00 00 0f 05  |H..*...H..<.....|
00000010
```

# Bugs in the Program

Your program might have errors! This has been prophesied for centuries:

> *... an analysing process must equally have been performed in order to furnish the Analytical Engine with the necessary operative data; and that herein may also lie a possible source of error. Granted that the actual mechanism is unerring in its processes, the cards may give it wrong orders.*
> - Ada Lovelace, Notes on the Analytical Engine, 1843

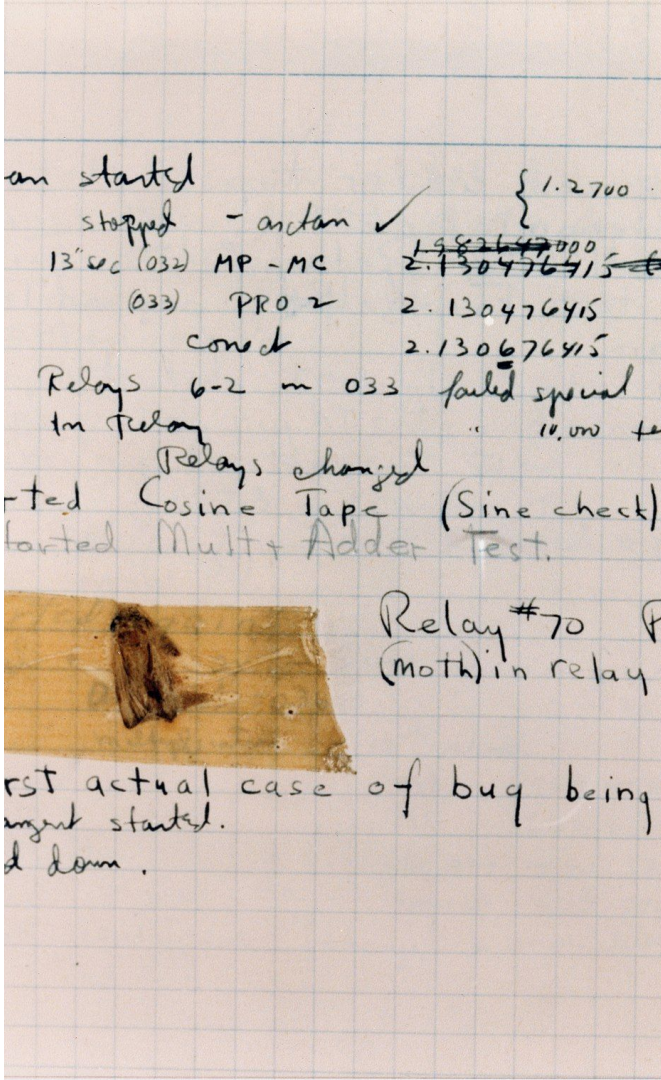
## Debugging Bugs through the ages.

The term "bug" to mean "fault" dates back a long time:

> *... difficulties arise–this thing gives out and [it is] then that "Bugs"–as such little faults and difficulties are called–show themselves*
> - Thomas Edison, letter, 1878

Popularly attributed to Grace Hopper for the moth to the right.

To remove bugs from the program, you de-bug them!




https://en.wikipedia.org/wiki/Software_bug

# Debugging

Debugging is done with *debuggers*, such as **`gdb`**.

Debuggers use (among other methods), a special *debug instruction*:

```
mov rdi, 42 // our program's return code (e.g., for bash scripts)
mov rax, 60 // system call number of exit()
int3 // trigger the debugger with a breakpoint!
syscall // do the system call
```

When the **`int3`** breakpoint instruction executes, the debugged program is interrupted and you can inspect its state!

Of course, the debugger itself can set breakpoints:

Overwrites the instruction at the breakpoint address with **`int3`**.
Emulates its effects when the breakpoint is executed instead!

In the Assembly Crash Course pwn.college challenges, we provide an automatic debugger for you; just put in an int3 and see!

# Other Resources

**GDB** is your go-to debugging experience.
You WILL become very good friends with it.

**strace** lets you figure out how your program is interacting with the OS.
A great first stop for debugging.

**Rappel** lets you explore the effects of instructions.
Get it from https://github.com/yrp604/rappel or just use the pre-installed version in the dojo!
Easily installable via https://github.com/zardus/ctf-tools.

**Documentation** of x86:
Opcode listing by byte value: http://ref.x86asm.net/coder64.html
Instruction documentation: https://www.felixcloutier.com/x86/
Intel's x86_64 architecture manual: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf