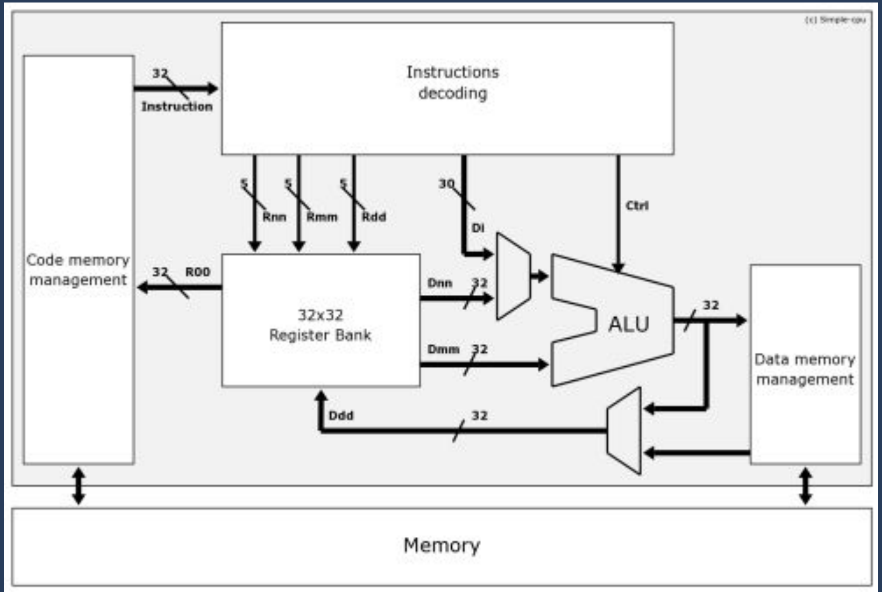
Reverse Engineering Introduction

CPU and ISA

What is a processor?

- CPU (Central Processing Unit)
- Performs arithmetic, logical, controlling and I/O operations
- Set of registers for specific purposes
- Memory for storing instructions and values
- Operations:
  - Bitwise (AND, OR, XOR)
  - Arithmetic (addition, subtraction)
  - Comparisons (>, <, >=, <=, ==, !=)
  - Read/write to memory



8-bit? 16-bit? 32-bit? 64-bit? 128-bit? … 1024-bit?

- Bit width specifies how many bits can be stored in a single register or how large memory addresses can be
- Examples:
  - 8-bit processor: single register can hold a max value of 255 (unsigned) - [1111 1111]
  - 64-bit processor: single register can hold a max value of (2^64)-1
- Address bus translates addresses to physical memory (read and write to RAM)
  - An address is just a binary represented integer of a specific bit-width
  - Example: Address [0xf00d] = what is stored in memory at the 61453'th consecutive location

Design tricks

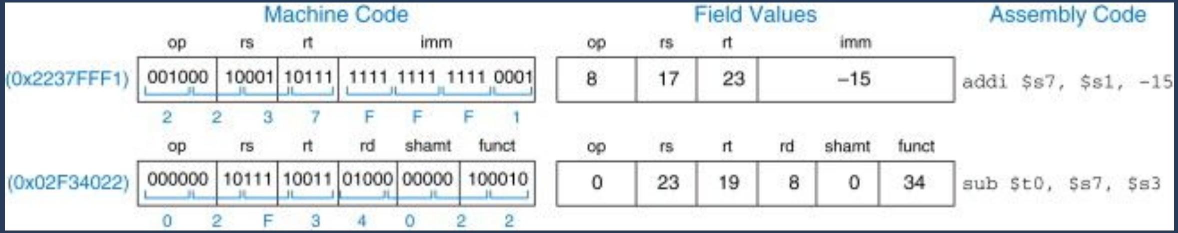
- Just because a processor has a specific bit-width for registers, it doesn’t mean that instructions are limited to that bit-width
- ???
- Example:
  - 8-bit processor has registers r0 - rn
  - Defined instruction Load Effective Address: **LEA r2, r0, r1**
  - What it does: Take the 8-bit value stored at the address gotten from concatenating registers r0 and r1 and store it in r2
  - Eg. at **0xf00d** is value **0xff**
    - r0 = 0xf0, r1 = 0x0d
    - r2 = [r0 | r1] = [0xf0 | 0x0d] = [0xf00d] = **0xff**
  - This means we have 16-bit addressing!

Design tricks - continued

- What purpose specific registers have is all defined by the architecture
- The instructions supported by the processor are also defined by the architecture
    - **ISA** (Instruction Set Architecture)
- In summary; what the processor is capable of is just a combination of architectural design and having hardware that supports it
- At its core, all values in registers or memory are just unsigned integers represented in binary.
- What gives them their semantic value is in the interpretation of the numerical value:
    - is **0x45** an: integer? (69), an ASCII character? (‘E’), an address?, a byte long bit flag?
    - All up to interpretation and architecture design!

Assembly instructions and machine code

- Instructions fed to a processor at runtime from code memory are just a string of bytes
- Bit-ranges specify
  - Operation (opcode)
  - Registers used
  - Immediate values
  - etc…

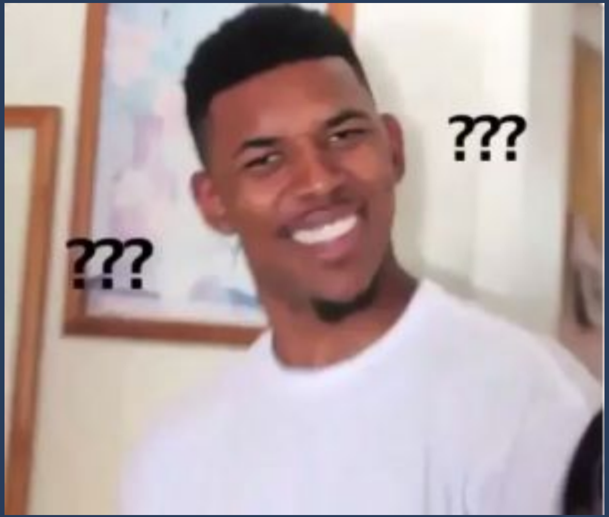


- When something goes wrong:
- Illegal opcode
  - The opcode specified in the byte string does not match a specification in the ISA

Shenanigans

- Given what we know, you might think that all instructions from machine code are executed sequentially
- Modern processors use things like:
  - Out-of-order execution - to execute code that requires more clock cycles ahead of when they sequentially would have been executed to avoid stalling
  - Branch prediction - guess which branch will be executed ahead of time to push more instructions in the pipeline
  - Speculative execution - execute instructions on the predicted branch even before knowing if the branch is actually taken

Shenanigans - continued



Memory

- There are many kinds of memory:
    - Cache - Extremely fast memory on the chipset of a processor
    - ROM - Read Only Memory (non-volatile), requires flashing, common on embedded systems
    - RAM - Random Access Memory (volatile), read and writeable memory
- Memory in a hardware sense is just an array of transistors capable of setting and storing binary values

OS and Assembly

Virtual Memory

- In a general purpose operating system, keeping track of very specific addresses used by individual processes would be very tricky and error prone
- Enter Virtual Memory
  - Virtual Memory abstracts the physical addresses from the running processes making them all get a unified view of their memory layout, even though it looks very different in hardware
- The hardware unit in charge of this is the Memory Management Unit (MMU)
  - Translates virtual addresses to hardware addresses
  - Caches frequently used translations in a Translation Lookaside Buffer (TLB)
- This is why you'll commonly see different processes using the same address ranges
- Virtual memory is not everywhere
  - Micro-controllers and embedded devices with limited resources commonly directly use physical memory

Theater time!

Process memory (Operating Systems)

- Processes are programs running on an Operating System
- Memory layout:
  - Code segment ->
  -
  - Heap ->
  -
  - libc ->
  -
  -
  - vDSO ->
  - ld ->
  -
  - Stack ->
  - Look at memory mapping of a process in: /proc/[pid]/maps

[Legend: Code | Heap | Stack]

Start	End	Offset	Perm	Path
0x00000000400000	0x00000000401000	0x00000000000000	r--	/home/kali/presentations/example
0x00000000401000	0x00000000402000	0x00000000001000	r-x	/home/kali/presentations/example
0x00000000402000	0x00000000403000	0x00000000002000	r--	/home/kali/presentations/example
0x00000000403000	0x00000000404000	0x00000000002000	r--	/home/kali/presentations/example
0x00000000404000	0x00000000405000	0x00000000003000	rw-	/home/kali/presentations/example
0x00000000405000	0x00000000426000	0x00000000000000	rw-	[heap]
0x007ffff7de0000	0x007ffff7de2000	0x00000000000000	rw-	
0x007ffff7de2000	0x007ffff7e08000	0x00000000000000	r--	/usr/lib/x86_64-linux-gnu/libc-2.33.so
0x007ffff7e08000	0x007ffff7f50000	0x00000000026000	r-x	/usr/lib/x86_64-linux-gnu/libc-2.33.so
0x007ffff7f50000	0x007ffff7f9b000	0x0000000016e000	r--	/usr/lib/x86_64-linux-gnu/libc-2.33.so
0x007ffff7f9b000	0x007ffff7f9c000	0x000000001b9000	---	/usr/lib/x86_64-linux-gnu/libc-2.33.so
0x007ffff7f9c000	0x007ffff7f9f000	0x000000001b9000	r--	/usr/lib/x86_64-linux-gnu/libc-2.33.so
0x007ffff7f9f000	0x007ffff7fa2000	0x000000001bc000	rw-	/usr/lib/x86_64-linux-gnu/libc-2.33.so
0x007ffff7fa2000	0x007ffff7fad000	0x00000000000000	rw-	
0x007ffff7fc6000	0x007ffff7fca000	0x00000000000000	r--	[vvar]
0x007ffff7fca000	0x007ffff7fcc000	0x00000000000000	r-x	[vdso]
0x007ffff7fcc000	0x007ffff7fcd000	0x00000000000000	r--	/usr/lib/x86_64-linux-gnu/ld-2.33.so
0x007ffff7fcd000	0x007ffff7ff1000	0x00000000001000	r-x	/usr/lib/x86_64-linux-gnu/ld-2.33.so
0x007ffff7ff1000	0x007ffff7ffb000	0x00000000025000	r--	/usr/lib/x86_64-linux-gnu/ld-2.33.so
0x007ffff7ffb000	0x007ffff7ffd000	0x0000000002e000	r--	/usr/lib/x86_64-linux-gnu/ld-2.33.so
0x007ffff7ffd000	0x007ffff7fff000	0x00000000030000	rw-	/usr/lib/x86_64-linux-gnu/ld-2.33.so
0x007ffffffde000	0x007ffffffff000	0x00000000000000	rw-	[stack]

Memory permissions

- Sections of process memory have different permissions

```
[ Legend:  Code | Heap | Stack ]
Start              End                Offset
0x0000000000400000 0x0000000000401000 0x0000000000000000 r--
0x0000000000401000 0x0000000000402000 0x0000000000001000 r-x
0x0000000000402000 0x0000000000403000 0x0000000000002000 r--
0x0000000000403000 0x0000000000404000 0x0000000000002000 r--
0x0000000000404000 0x0000000000405000 0x0000000000003000 rw-
```

- r = Read, w = Write, x = Execute
- [r - -] = read only; for example hard-coded constants
- [r - x] = read and execute; machine instructions

Calling conventions

- Specifies where arguments should be placed before calling a function
- Example x86:
  - arguments are pushed on the stack
  - return value is some address placed in register **eax**
- Caller-saved registers
  - the function you call takes no responsibility for what happens to these registers. Save them before calling if you care about them
- Callee-saved registers
  - caller can expect that the registers will hold the same values after the call as they did before

Calling conventions - continued

- Calling conventions differ between processor architectures
- Example:
    - As opposed to **x86**, **x64** (or **x86-64**) arguments are not pushed on the stack, but rather put in specific registers
    - These are (in **System V AMD64**): **rdi, rsi, rdx, rcx, r8, r9**
    - The rest go on the stack
- Calling conventions are arbitrarily defined, but you have to follow them closely when programming for a specific target platform.

Calling conventions - continued

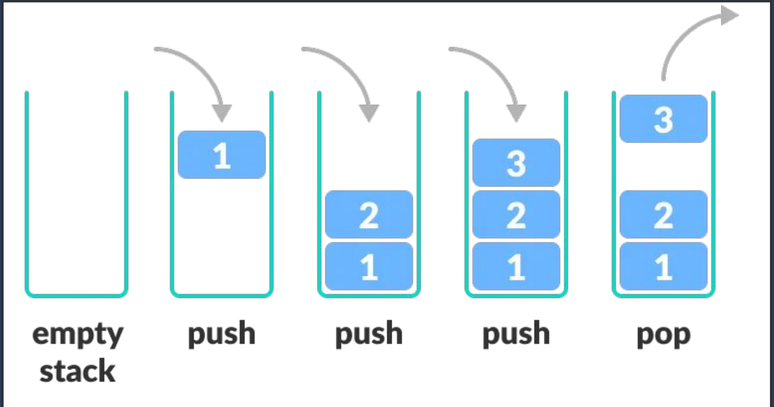
```c
int main(int argc, char** argv){
    char* test = "This is a test";
    char* test2 = "Hello";
    char* heapy = malloc(6);
    for(int i = 0; i<6; i++){
        heapy[i] = test2[i];
    }
    printf("%s\n",test);
    printf("%s\n",heapy);
    return 0;
}
```

```
→ 0x5555555551c4 <main+123>      call   0x555555555030 <printf@plt>
   ↳  0x555555555030 <printf@plt+0>    jmp    QWORD PTR [rip+0x2fe2]
      0x555555555036 <printf@plt+6>    push   0x0
      0x55555555503b <printf@plt+11>   jmp    0x555555555020
      0x555555555040 <malloc@plt+0>    jmp    QWORD PTR [rip+0x2fda]
      0x555555555046 <malloc@plt+6>    push   0x1
      0x55555555504b <malloc@plt+11>   jmp    0x555555555020

printf@plt (
   $rdi = 0x00555555556019 → 0x1000000000a7325 ("%s\n"?),
   $rsi = 0x00555555556004 → "This is a test"
)
```

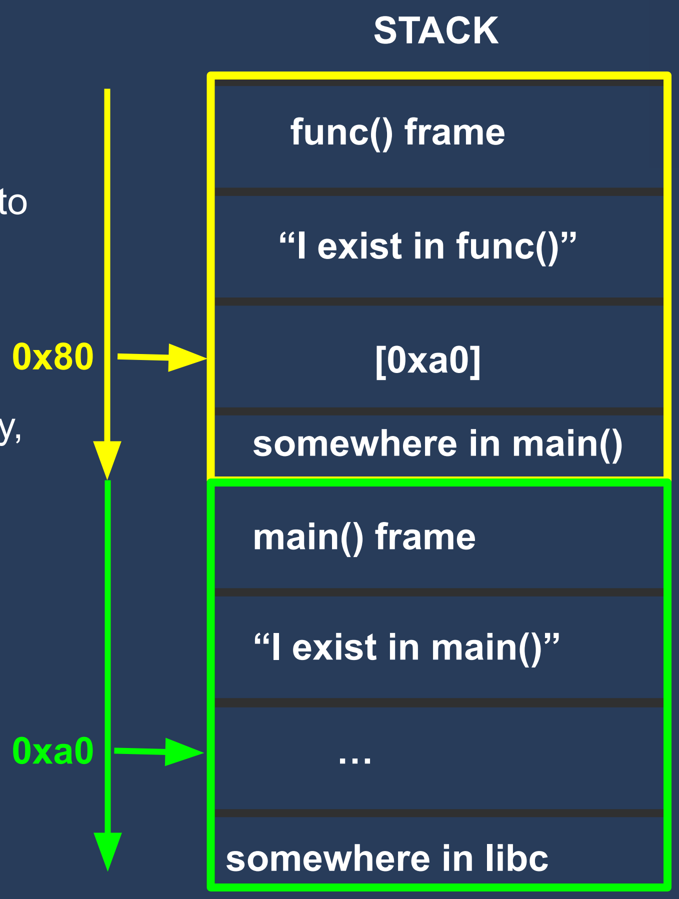
Stack

- The Stack is a data structure that is used for storing information in RAM
- Operations:
  - PUSH - Place on top of stack
  - POP - Get value from top of stack
- Used for storing values when registers are not enough
- RSP - stack pointer
- RBP - base pointer, contains address pointing to the base of the previous stack frame



Stack frame

- The stack grows from a high address to a lower one
  - In Linux, the “start” of the stack is right next to kernel space virtual memory
    - 0x00007fffffff…
- Each function has its own stack frame
- Frame:
  - **stack pointer** - points to the top (numerically, the bottom) of the stack frame
  - **local function variables**
  - **base pointer** - points to the bottom (numerically, the top) of the previous stack frame
  - **return address** - where we resume execution after returning from the function



Stack frame - continued

```c
void func2(){
    int lol = 0x42;
    int boll = 42;
    printf("%s\n%d\n","Stack frame and shit",lol+boll);
    return;
}
```

Current frame ->

```
0x007fffffffdf00│+0x0000: 0x0055555555555240  →  <__libc_csu_init+0> push r15      ← $rsp
0x007fffffffdf08│+0x0008: 0x000000420000002a ("*"?)
0x007fffffffdf10│+0x0010: 0x007fffffffdf50  →  0x0000000000000000      ← $rbp
0x007fffffffdf18│+0x0018: 0x005555555555522f  →  <main+165> mov eax, 0x0
```

<- rip

Previous frame ->

```
0x007fffffffdf20│+0x0020: 0x007fffffffe048  →  0x007fffffffe395  →  "/home/kali/presenta
0x007fffffffdf28│+0x0028: 0x0000000155555240
0x007fffffffdf30│+0x0030: 0x00555555555592a0  →  0x00006f6c6c6548 ("Hello"?)
0x007fffffffdf38│+0x0038: 0x005555555555602f  →  0x7325006f6c6c6548 ("Hello"?)
0x007fffffffdf40│+0x0040: 0x0055555555556020  →  "This is a test"
0x007fffffffdf48│+0x0048: 0x0000000600000000
0x007fffffffdf50│+0x0050: 0x0000000000000000
0x007fffffffdf58│+0x0058: 0x007ffff7e097ed  →  <__libc_start_main+205> mov edi, eax
```

<- prev $rbp

<- rip

Symbols - Names for addresses

```c
#include <stdio.h>

const char *coolio = "cool!";

void cool_function() {
   puts(coolio);
}

void amazing_function() {
   puts("Amazing!");
}

int main() {
   cool_function();
   amazing_function();
}
```

```
$ gcc -o binary main.c
$ nm ./binary
000000000000039c r __abi_tag
000000000000114f T amazing_function
0000000000004020 B __bss_start
0000000000004020 b completed.0
0000000000001139 T cool_function
0000000000004018 D coolio
                 w __cxa_finalize@GLIBC_2.2.5
...              ...
                 w _ITM_registerTMCloneTable
                 U __libc_start_main@GLIBC_2.34
0000000000001165 T main
                 U puts@GLIBC_2.2.5
00000000000010a0 t register_tm_clones
0000000000001040 T _start
0000000000004020 D __TMC_END__
```

Stripped binaries

```
$ strip ./binary -o binary-stripped
$ nm ./binary-stripped
nm: binary-stripped: no symbols
```

```
$ file ./binary
./binary: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter
/lib64/ld-linux-x86-64.so.2, BuildID[sha1]=8fc9f669ae4a4a27683c69bd1af9de3146c919c1, for GNU/Linux
4.4.0, with debug_info, not stripped
$ file ./binary-stripped
./binary-stripped: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=8fc9f669ae4a4a27683c69bd1af9de3146c919c1, for
GNU/Linux 4.4.0, stripped
```

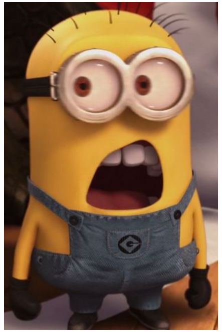
Static vs. dynamic linking

```
$ gcc -static -o binary-static main.c
$ ls -lh
-rwxr-xr-x 1 mkg mkg  21K Sep 20 20:09 binary
-rwxr-xr-x 1 mkg mkg 764K Sep 20 20:28 binary-static
```

```
$ nm ./binary | wc -l
33
$ nm ./binary-static | wc -l
1925
```

```
$ file ./binary
./binary: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter
/lib64/ld-linux-x86-64.so.2, BuildID[sha1]=8fc9f669ae4a4a27683c69bd1af9de3146c919c1, for GNU/Linux
4.4.0, with debug_info, not stripped
$ file ./binary-static
binary-static: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically linked,
BuildID[sha1]=8cd04fb0a025a2482b6ef956c3e56c63f8183baf, for GNU/Linux 4.4.0, with debug_info, not
stripped
```

The pain square

	Dynamically linked	Statically linked
With symbols		
Stripped		

Reverse Engineering

Mindset


I'M LEARNDING.

Reverse Engineering is Speed Science

- Using binary files as a case study; they could be considered artifacts of engineering
- An engineering process where design decisions where made and source code was written that would later be compiled into the end result - the binary executable
- Reverse engineering is the process of inferring the engineering process, design decisions made and source code written only from having access to the behavior of the end result
- The reverse engineering process usually follows something similar to the Hypothetico-Deductive Model:
    - 1. Use your experience of similar problems to ->
    - 2. Generate a hypothesis H which can be used to ->
    - 3. Construct testable true/false statements that will then be used in an ->
    - 4. Experiment that evaluates the hypothesis based on the constructed statements
    - If all statements evaluate to true, you can increase your confidence in the hypothesis
    - If some experiment fails, consider reforming the hypothesis
- For complex problems, you will most likely never reach perfect accuracy in your reversing, but through an iterative process, you will get a stronger understanding of how it behaves and have stronger foundations for inferring the engineering process
- The more experience you gather, the better hypotheses you'll be able to form!

More stuff on methodology

- In general, a good way of getting started is to make it clear to yourself what the problem **is not**
  - Decrease search space by pruning off branches of where your hypothesizing could go
  - Example: The goal is to find some way of bypassing an algorithmic authentication mechanism
    - Are there functions that do not concern the input or evaluation in any way?
    - if so, mentally mark them as **Don’t Care** and move on
- There is no text book “right” approach. Through experience and trial and error you will discover the methodology that fits you
- A reverse engineering result is graded based on how close its approximation is to the source
  - Even though end result is the same, two people might have vastly different approaches
- Hilariously enough, in some cases, reverse engineering a binary executable might lead to the reverser having a clearer and deeper understanding of how the program actually works than that engineer that created it
  - “Magic” compiler optimizations might have caused the actual machine code and assembly instructions to be semantically different from the higher level language source code

Demo

Workflow

- If you are dealing with obfuscated stuff, using regexes to clean up symbol or decompilation output is useful
- Examples:
  - grep (to match on specific patterns)
  - awk (extremely useful scripting language for programmatically treating text, grep is a subset of awk)
  - regex matching in text editors
- If you need to implement necessary program logic, Python scripting is generally the quickest and easiest way to go

Static analysis

- IDA, Ghidra, Binary ninja, Hopper, Cutter, radare2, etc.
- objdump
- strings, rabin2 -z

Dynamic analysis (debugging)

- gdb (Tutorial)
- GEF (GDB Enhanced Features)
- ltrace and strace

Questions?

Upcoming CTFs and meetups

- DownUnderCTF - 23 sept, 11:30 – 25 sept 2022, 11:30
- LakeCTF - 24 sept, 20:00 – 25 sept, 20:00
- Meetup - Hang out, solve challs - next Thursday
- Meetup - PWN or guest lecture - next next Thursday

What do you know about PWN?

- ?