Reverse Engineering

Josh Hofing

Last Week:

- Basics of x86_64
  - Registers
  - Memory
  - Instructions
  - The Stack
  - Calling Conventions

This Week:

- Virtual Memory
- Page Tables
- The OS
- Theorem Provers

Virtual Memory

```
main:
00400895  55              push    rbp
00400896  4889e5          mov     rbp, rsp
00400899  4883ec10        sub     rsp, 0x10
0040089d  b800000000      mov     eax, 0x0
004008a2  e866ffffff      call    init
004008a7  bfa80a4000      mov     edi, 0x400aa8  {"Can you tell me where to mail th…"}
004008ac  e84ffdffff      call    puts
004008b1  b800000000      mov     eax, 0x0
004008b6  e876ffffff      call    get_number
004008bb  488945f0        mov     qword [rbp-0x10], rax
004008bf  488b45f0        mov     rax, qword [rbp-0x10]
004008c3  488945f8        mov     qword [rbp-0x8], rax
004008c7  488b45f8        mov     rax, qword [rbp-0x8]
004008cb  488b10          mov     rdx, qword [rax]
004008ce  48b8eecefa0d000d… mov     rax, 0xd000dfaceee
004008d8  4839c2          cmp     rdx, rax
004008db  751b            jne     0x4008f8
```

```
004008f8  bf000b4000      mov     edi, 0x400b00  {"That doesn't look right... try a…"}
004008fd  e8fefcffff      call    puts
00400902  b801000000      mov     eax, 0x1
```

```
004008dd  bfd80a4000      mov     edi, 0x400ad8  {"Got it! That's the right number!"}
004008e2  e819fdffff      call    puts
004008e7  b800000000      mov     eax, 0x0
004008ec  e88cfeffff      call    print_flag
004008f1  b800000000      mov     eax, 0x0
004008f6  eb0f            jmp     0x400907
```

```
00400907  c9              leave   {__saved_rbp}
00400908  c3              retn
```

Virtual Memory

- The address here is interesting…
- Is the first instruction of main *really* at 0x00400895?
- What if I run 2 copies of the program?
- What if I run 2 different programs?

```
main:
00400895  55                    push    rbp
```

Virtual Memory

- Most modern hardware supports *Virtual Memory*
- The addresses we tell the program it is running at are different than the physical locations in memory they live at
- How do we tell the hardware how a vaddr maps to a physical address?

Page Tables

- Mapping of a *Page* of Virtual Addresses to a *Page* of Physical Addresses

- How big is a Page?
  - It depends on the architecture
  - We'll only really ever look at 4KB pages

Page Tables

- Pages are *aligned*, meaning the bottom bits are all zeroes
- So, to map the region containing 0x00400895, you map 0x00400000 to 0x00400FFF
  - 0xFFF = 4KB - 1

Page Permissions

- Pages have permissions associated with them:
  - Read (R) = Can I read data from the page?
  - Write (W) = Can I write data to the page?
  - Execute (X) = Can I execute instructions on the page?
- This will matter when we talk about pwning!

The OS

- Modern Operating Systems are really complicated!
- They deal with input and output to a huge number of devices
    - Keyboards/Mice, Networking, GPUs, etc.
- They manage a bunch of programs running at once
- They manage page tables

Talking to the OS

- When a program wants the OS to do something, it issues a *System Call*
  - The *syscall* instruction does this on x86_64
- Registers are setup, just like function arguments
  - rax contains the *Syscall Number*
  - rdi, rsi, rdx, r10, r8, r9 are arguments

Syscalls

- There are lots of syscalls on modern Linux (300+)
    - And C functions that mimic the common ones!
- The main uses are things like dealing with files, networking, and allocating memory

Files (and things like them)

- In Unix-like operating systems, everything is a file
  - Files (duh)
  - Sockets (for networking)
  - Information about processes (/proc)
  - Attached devices (/dev)
  - System settings (/sys)

File Descriptors

- When you use the *open* syscall, you are given back a *handle* to a file, also known as a *file descriptor*
- A file descriptor is just a number
  - Some file descriptors that come built-in to the program:
    - 0 = standard input (stdin)
    - 1 = standard output (stdout)
    - 2 = standard error (stderr)

The network is files, too!

- The *socket* syscall creates a new socket
  - You can use it as a client or a server

Using Files

- There are 3 main operations all files support:
  - *read*, *write*, and *close*
- Same API for files, network, system information, randomness, …
  - Pretty neat!

Example

- https://godbolt.org/g/7ijzxF
- this will help: https://filippo.io/linux-syscall-table/

Theorem Proving

- Sometimes, programs involve complicated instructions

- *recurse* is a great example

Z3

- We imported from *z3*, which is a fantastic *theorem prover*

- Give it a set of constraints, and it will produce values that satisfy them (or say they are *unsatisfiable*)

    - Sometimes this will take awhile…

- A commonly used tool in CTFs

Ints, BitVecs, Solvers

- Z3 supports a bunch of types that it understands
  - You'll mostly see:
    - Ints (arbitrary-size integers)
    - BitVecs (Integers of a specific bit-length)
    - Bools (True or False)
    - a Solver (the API for checking constraints)

Basic Example

```
from z3 import Ints, Solver
a, b = Ints('a b')
s = Solver()
s.add(a + b == 1234)
s.add(a - b == 500)
print(s.check())
print(s.model())

>>> sat
>>> [a = 867, b = 367]
```

Another Example

```
from z3 import BitVecs, Solver
a, b = BitVecs('a b', 16)
s = Solver()
s.add(a ^ b == 0xbeef)
s.add(a == 0xdead)
print(s.check())
print(s.model())

>>> sat
>>> [b = 24642, a = 57005]
```

Recurse

```
int recurse(int a, int b, int c) {
    int sum = a + b;
    if (c == 16 && sum == 116369) {
        return 1;
    } else if (c < 16) {
        return recurse(b, sum, c + 1);
    } else {
        return 0;
    }
}
```

And in Python

```
def recurse(a, b):
    for _ in range(17):
        a, b = b, a + b
    return b

a, b = raw_input().split(' ')
a, b = int(a), int(b)
assert(recurse(a, b) == 116369)
```

Solving without doing math!

```
from z3 import Ints, Solver
a, b = Ints('a b')
rec = recurse(a, b)
s = Solver()
s.add(rec == 116369)
print(s.check())
print(s.model())

>>> sat
>>> [b = 37, a = 13]
```

This is really powerful

- Doing the solve is also super NP-Complete
- If your equations get too big or too complicated, you're gonna have a bad time
    - I've had equations that Z3 spent hours trying to solve
    - It's a powerful tool, but not always the most scalable