SP2023 Week 05 • 2023-02-23

Reverse Engineering III

Richard

Announcements

- Cyber Tractor Challenge (application due 2023-03-13)
  - Travel to Des Moine to learn how to secure John Deere equipment

- ICSSP Informational Meeting (2023-03-02)
  - Scholarship and government internship opportunity
  - 5pm @ Siebel CS 2405

- Come to SAIL!
  - If you want to present, apply here by midnight on the 24th (tomorrow!)
  - Free shirt and food for presenters, teach with up to 5 people on April 8th!

ctf.sigpwny.com

sigpwny{vm_stands_for_very_mad}

Topics

- **Virtual Machine (VM) reversing**
  - Common VM architectures
  - Control flow graphs
  - Tips and tricks
- **Instrumentation**
  - Side channel attacks
- **Memoization**

Virtual Machines

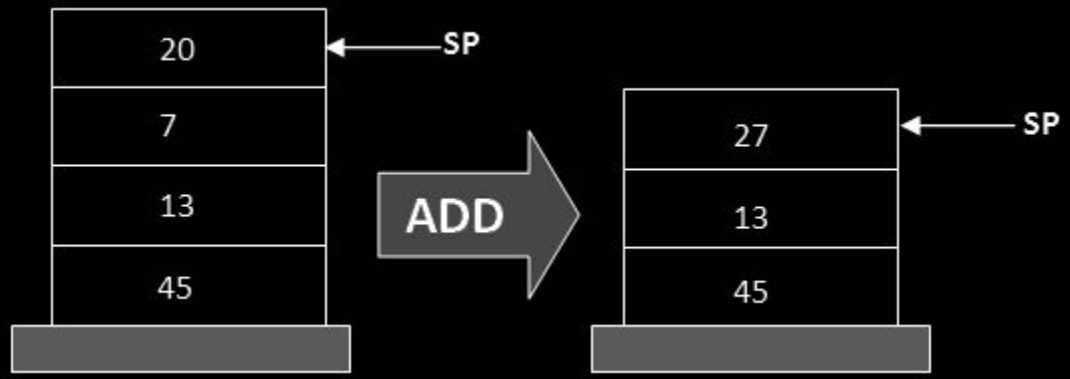
It's VMs all the way down…

What is a VM?

- Virtual machine: emulating a computer
  - **Instructions** that get interpreted to do certain commands
  - **I/O** to interact with the host computer
- Why?
  - To run one architecture on another (qemu)
  - To separate resources for security (KVM)
  - **To obfuscate code** (many CTF challenges)

Stack based VM

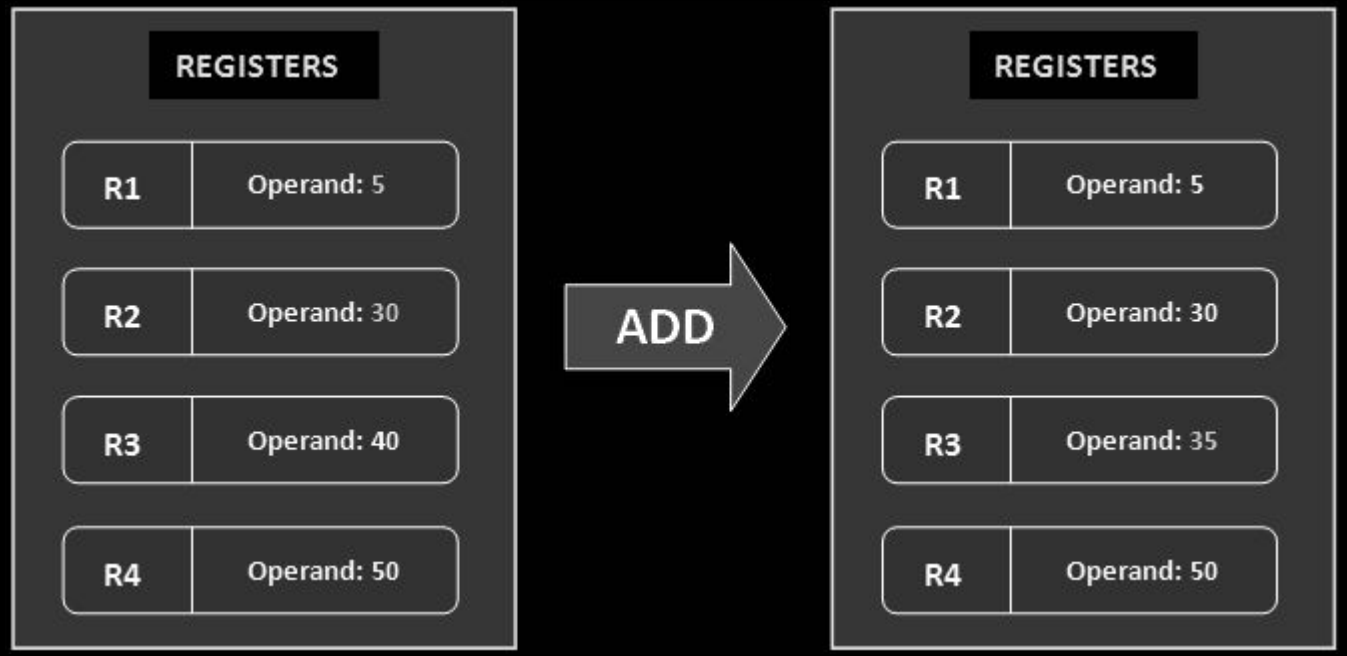
- separate instructions and data
- operations manipulate stack




```
PUSH 13
PUSH 45
PUSH 7
PUSH 20
ADD
```

Register based VM

- use registers to store intermediate data
- similar to common ISAs (x86, ARM, MIPS)
- usually have a dedicated space for data



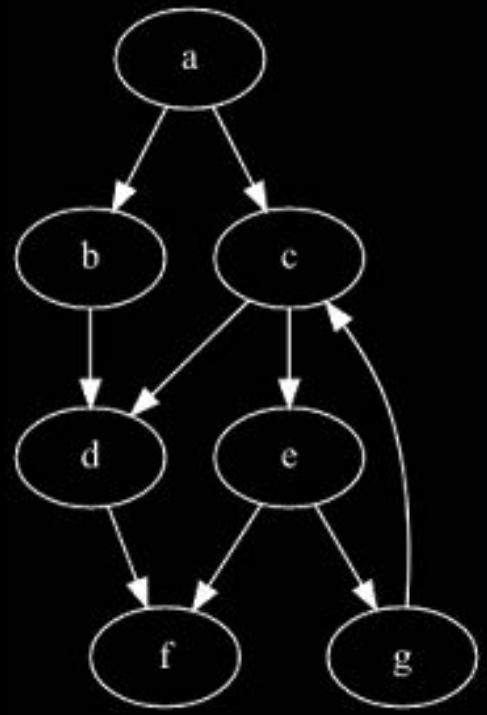

```
MOV R1, 5
MOV R2, 30
MOV R3, 40
MOV R4, 50
ADD R3, R1, R2
```

Reverse engineering VMs

- Write your own tooling!
  - control flow graphs
  - disassembler
  - debugger
- Tooling helpful for larger VMs

Control flow graphs

- shows how program behaves at a high level
- each node represents a **basic block** (one entry and one exit)
- two edges from a node could be an if statement
- an edge going "back" could be part of a loop



Control flow graphs (cont.)

- good questions to ask with a CFG
  - What is the win condition?
  - What branches take me to the win condition?
  - What code is unused?

Tips and tricks

- look up magic numbers
  - ex: 0x9e3779b9, golden ratio used in TEA (this came up in CSAW!)
- verify assumptions
  - can you run your own code in the VM?
  - ex: "I think opcode 0x9 performs an add". But does it?
- **brute force**
  - if the flag is checked one byte at a time, try every byte value

Brute force example

```
...
input = get_input();
const_array = [...];
for (int i = 0; i < input_len; i++) {
    if (input[i] != sum(const_array[0:i]) {
        print("Wrong!");
        break;
    }
}
...
```

- brute forcible, as the correct value of the ith character **only depends on the previous characters**
- try every character until we get one more loop iteration further

GDB Python template

```
import gdb
from struct import pack
import string
import sys

gdb.execute('file ./chal')
gdb.execute('b *(0x55555555539f)')
gdb.execute('set confirm off')

def count_correct(data):
    num_correct = int(gdb.parse_and_eval('$rax'))
    return len(data)
```

Set a breakpoint where it compares our input

Use register/memory operands to see how much of input was correct

```
flag = ''
best = len(flag)
alphabet = '_{}!' + string.ascii_lowercase +
string.digits + string.ascii_uppercase +
string.punctuation
for i in range(20):
    for c in alphabet:
        s = flag + c
        gdb.execute(f"run <<< '{s}'")
        count = count_correct()
        if count > best:
            best = count
            flag = s
            print(flag)
            break

gdb.execute('q')
```

Try every character until we find one that works!

Challenges

- ctf.sigpwny.com/challenges
  - VMWhere 0-2
  - Walkthrough of VMWhere 0 at the end of the meeting

Instrumentation

Computer go brrr

Background

- Two ways to find out what it does
  - Static analysis: looking at the binary without running it
  - **Dynamic analysis**: collecting information while running it
- Some common dynamic analysis tools:
  - gdb: classic debugger
  - **Pin: instrumentation tool**
  - angr: symbolic analysis

Motivation

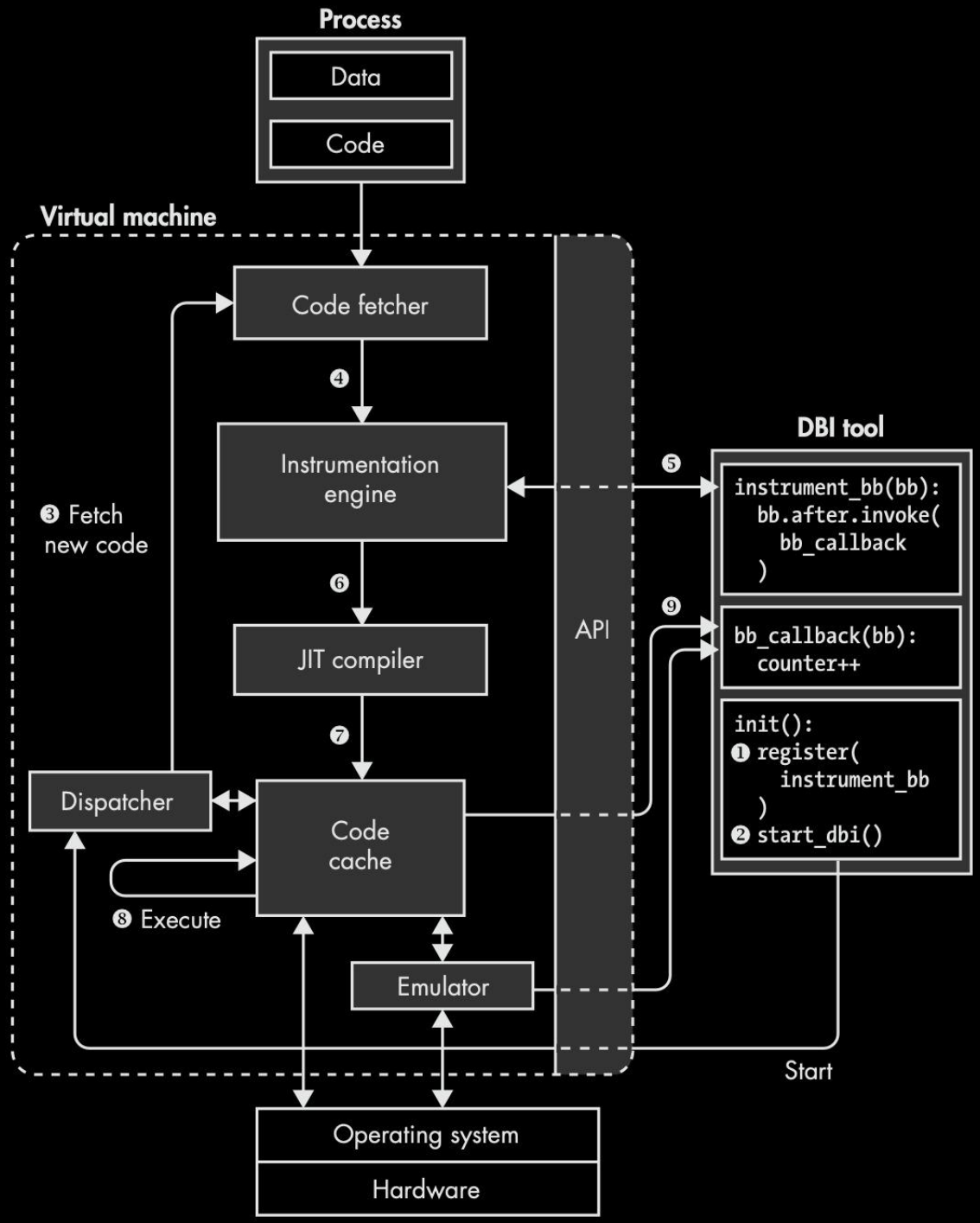
- What if you wanted to:
  - Print the arguments to every `strcmp` call?
  - Count the number of function calls/code lines/instructions?
  - Log every memory write?

What is it

- Modifying binaries on-the-fly
- Add our own code ("instruments")
- Control flow recovery
- Added code does not affect the binary

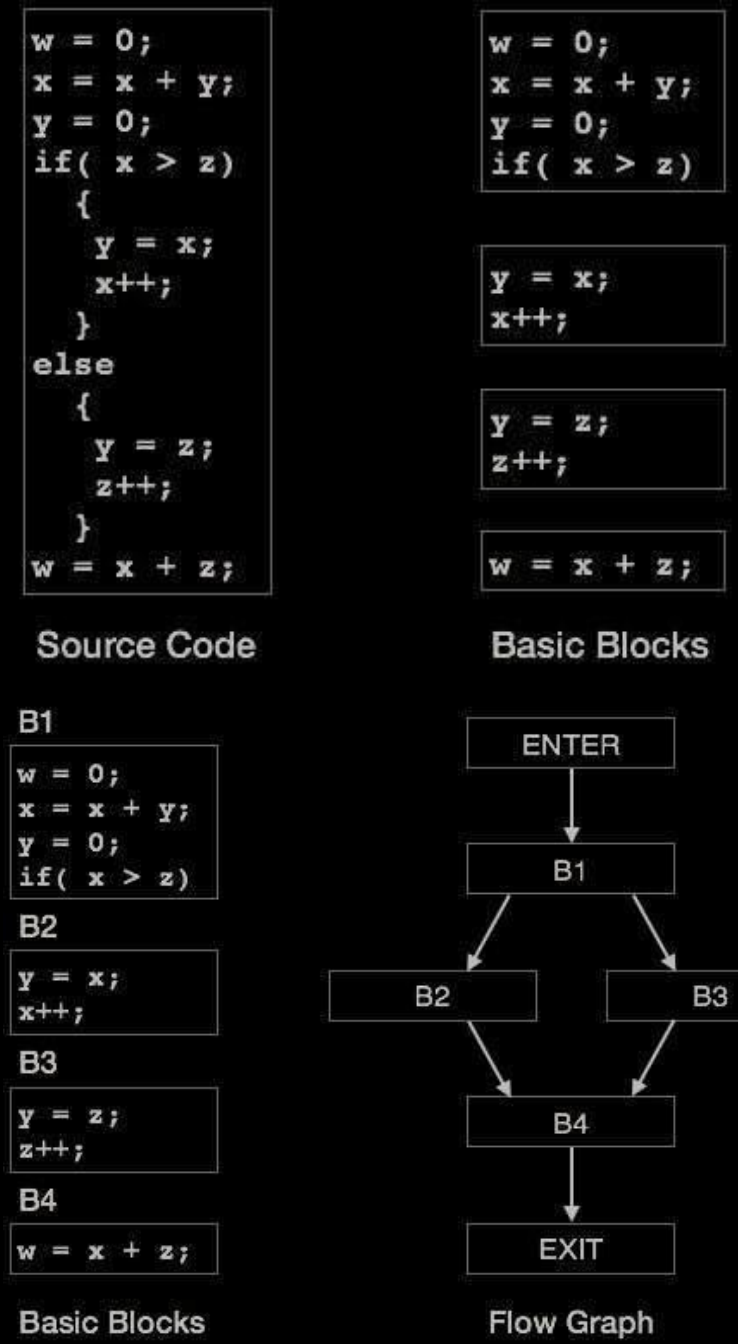
Basic overview

1. Load the binary
2. Disassemble, recover basic blocks
3. Add instrumentation
4. Add to cache
5. Run the binary
    a. Lazy; instrument more block only if necessary



How it works

- Disassemble the binary (recursive, linear)
  - Surprisingly non-trivial, esp. w/ variable-length instruction ISAs
- Analyze the disassembly and get "basic blocks"
  - Boundary at jumps/calls/rets
- Each basic block is individually analyzed



Use Cases

- Instruction counting
- Function call statistics
- VM instruction tracing
- Memory watching
- Syscall tracing

Tool

- **Intel's PIN**
  - Fast, but steep learning curve
- **GDB Python**
  - Slow, but easy to use
- DynamoRIO
  - Open source

Real World Example

- "ropfuscated" from UIUCTF 2021
- VM reversing chal using ROP gadgets as the instructions
- difficult due to large VM program
- solve process:
  - use Pintool to dump instructions
  - analyze CFG
  - realized instructions follow a pattern (compiled from ELVM)
  - decompile to ELVM
  - step through program to find flag comparison checks

Dump executed instructions

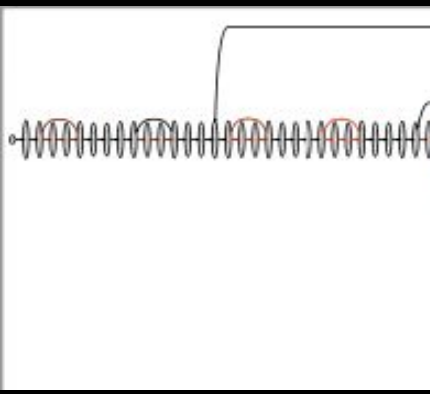
```
004013c1 00414078     0x4013c1              0x800 0x414078 pop rsi ; ret
00401352 00414088     0x401352                    0x414088 xchg rax, rsi ; ret
00401355 00414090     0x401355                    0x414090 mov rcx, rax ; ret
0040133d 00414098     0x40133d                    0x414098 xchg rcx, rbx ; ret
00401350 004140a0     0x401350           0x4040b4 0x4140a0 pop rax ; ret
0040136d 004140b0     0x40136d                    0x4140b0 mov qword ptr [rax], rbx ; ret
00401350 004140b8     0x401350             0x44 0x4140b8 pop rax ; ret
00401337 004140c8     0x401337                    0x4140c8 xchg rax, rbx ; ret
0040133d 004140d0     0x40133d                    0x4140d0 xchg rcx, rbx ; ret
00401350 004140d8     0x401350           0x404094 0x4140d8 pop rax ; ret
00401369 004140e8     0x401369                    0x4140e8 mov qword ptr [rax], rcx ; ret
00401350 004140f0     0x401350              0x0 0x4140f0 pop rax ; ret
004013b7 00414100     0x4013b7           0x40408c 0x414100 pop rbx ; ret
00401371 00414110     0x401371                    0x414110 mov qword ptr [rbx], rax ; ret
004013b7 00414118     0x4013b7           0x40408c 0x414118 pop rbx ; ret
0040135d 00414128     0x40135d                    0x414128 mov rax, qword ptr [rbx] ; ret
```

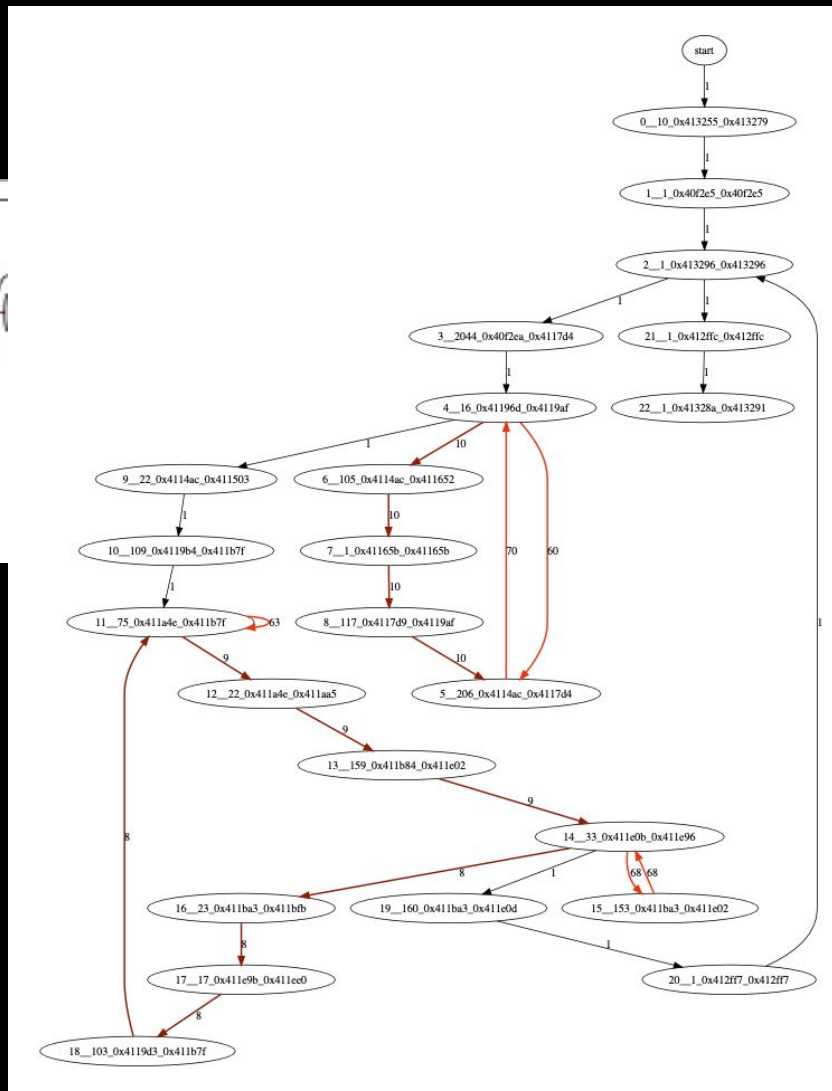
Decompile to ELVM

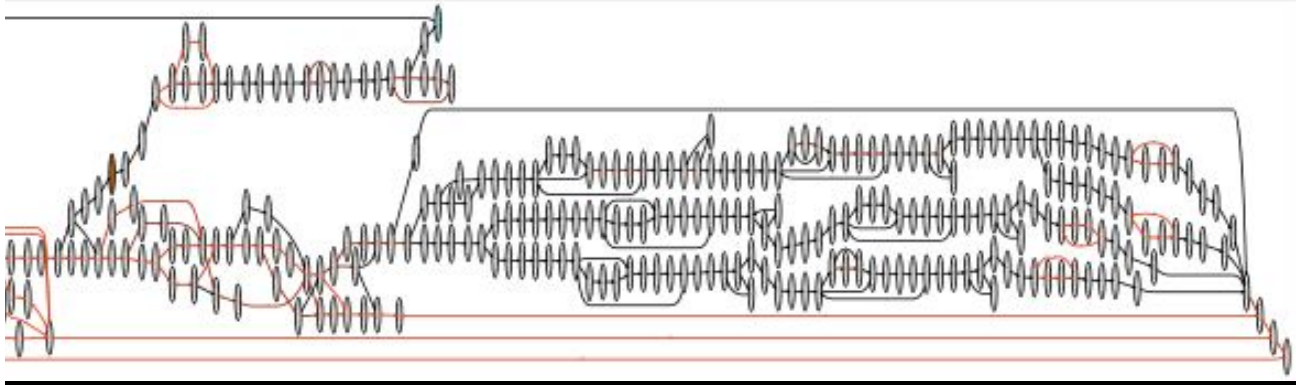
```
mov SP, imm 0x800          0x414078 ... 0x4140b0 (6)
mov B, imm 0x44            0x4140b8 ... 0x4140e8 (5)
mov F, imm 0x0             0x4140f0 ... 0x414110 (3)
mov A, F                   0x414118 ... 0x414140 (4)
add A, A                   0x414148 ... 0x4141a0 (9)
add A, A                   0x4141a8 ... 0x414200 (9)
add A, A                   0x414208 ... 0x414260 (9)
mov E, imm 0x4040c4        0x414268 ... 0x414290 (4)
add A, E                   0x414298 ... 0x4142f0 (9)
```

Control flow graphs

Original ROP chain CFG







Decompile ELVM CFG

Big idea: finding patterns

- no one will hand write 4 MB of assembly
- find patterns to work your way up the "abstraction chain"
- in this challenge, the abstraction chain was:
  - C code
  - ELVM IR
  - ROP chain
  - ROP (in x86 assembly)
- each layer added 5-10x more instructions

Memoization

a.k.a dynamic programming

Fibonacci

```
def fib(n):
    if n < 2: return n
    return fib(n-1) + fib(n-2)
```

- What happens if we try to calculate `fib(100)`?
- Is there a way to optimize this?

Memoization

- Save function return values in a lookup table
- Only works if the function has no side effects
  - Same output for a given input

Real World Example

- "to_inefficient" from nitectf 2021
- two recursive calls
- calculates "n choose k" by the recurrence:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Recursive calls

```
1
2  long main.a(ulong **param_1,undefined8 param_2,undef
3              ,undefined param_6,undefined8 param_7,und
4
5  {
6    long in_RAX;
7    long lVar1;
8    long lVar2;
9    char cVar3;
10   undefined extraout_DL;
11   undefined extraout_DL_00;
12   long unaff_RBX;
13   long unaff_R14;
14   undefined8 in_stack_fffffffffffffd8;
15   undefined8 in_stack_fffffffffffffe0;
16
17   while (&stack0x00000000 < *(undefined **)(ulong *)
18          &stack0x00000000 == *(undefined **)(ulong *
19     runtime.morestack_noctxt.abi0(param_1,param_2);
20     param_3 = extraout_DL_00;
21   }
22   if (unaff_RBX <= in_RAX) {
23     if ((unaff_RBX != 0) && (unaff_RBX != in_RAX)) {
24       cVar3 = (char)unaff_RBX + -1;
25       lVar1 = main.a(param_1,param_2,param_3,cVar3,p
26                      in_stack_fffffffffffffe0);
27       lVar2 = main.a(param_1,param_2,extraout_DL,cVa
28                      in_stack_fffffffffffffe0);
29       return lVar2 + lVar1;
30     }
31     return 1;
32   }
33   return 0;
34 }
35
```

Solution: GDB Python

- Use GDB Python to cause function to immediately return with correct value
- I also patched original binary to replace entire function with a `ret`

```
# set silent breakpoint (see next slide)
...

lookup = []
# precomputed values for n choose k
with open('cycle', 'r') as f:
    for line in f:
        lookup += [int(line)]

for i in range(1000):
    # at the stopped breakpoint, do the lookup
    # and skip running the function
    rax = int(gdb.parse_and_eval('$rax'))
    cur = lookup[rax]
    gdb.execute('set $rax={}'.format(cur))
    gdb.execute('c')
```

Full script

```
import gdb
import math

gdb.execute('file ./chal_mod')
gdb.execute('set args flag.fig out4')
gdb.execute('set confirm off')

class MyBreakpoint(gdb.Breakpoint):
    def __init__(self):
        super().__init__('*0x4815f6')
        self.silent = True

    def stop(self):
        return True

MyBreakpoint()
```

Break at the function

```
gdb.execute('run')

lookup = []
with open('cycle', 'r') as f:
    for line in f:
        lookup += [int(line)]

for i in range(1000):
    rax = int(gdb.parse_and_eval('$rax'))
    cur = lookup[rax]
    gdb.execute('set $rax={}'.format(cur))
    gdb.execute('c')
```

Replace the return value

Challenges

- VMWhere 0-2: VM reversing
- Hell: VM reversing, instrumentation (very hard)
- Ropfuscated: briefly went over solve
  - Download here: https://2021.uiuc.tf/challenges#ropfuscated-49

Next Meetings

2023-02-26 - This Sunday

- Nintendo DSi Browser Exploit
- Nathan will share how he hacked the DSi web browser

2023-03-02 - Next Thursday

- Quantum computation with George

sigpwny{vm_stands_for_very_mad}

VMWhere 0 demo