Qiling Framework: Introduction

November 2020

twitter: @qiling_io https://qiling.io

KaiJern Lau, kj -at- qiling.io

About xwings

JD.COM

Beijing, Stays in the lab 24/7 by hoping making the world a better place

- IoT Research
- Blockchain Research
- Fun Security Research

Qiling Framework

Cross platform and multi architecture advanced binary emulation framework

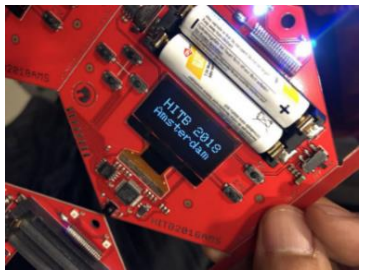
- https://qiling.io
- Lead Developer
- Founder

Badge Maker

Electronic fan boy, making toys from hacker to hacker

- Reversing Binary
- Reversing IoT Devices
- Part Time CtF player

Badge Designer for Hacking Conferences



Some Recent Talk (Partial)

- 2016, Qcon, Beijing, Speaker, nRF24L01 Hijacking
- 2016, Kcon, Beijing, Speaker, Capstone Unicorn Keystone
- 2017, Kcon, Beijing, IoT Hacking Trainer
- 2018, Kcon, Beijing, IoT Hacking Trainer
- 2018, Brucon, Brussel, Speaker, IoT Virtualization
- 2018, H2HC, San Paolo, Speaker, IoT Virtualization
- 2018, HITB, Beijing/Dubai, Speaker, IoT Virtualization
- 2018, beVX, Hong Kong, Speaker, HackCUBE - Hardware Hacking
- 2019, DEFCON USA, Qiling Framework Preview
- 2019, Zeronights, Qiling Framework to Public
- 2020, Nullcon GOA, Building Reversing Tools with Qiling
- 2020, HITB AMS, Building Reversing Tools with Qiling
- 2020, HITB Singapore, Training, How to Hack IoT with Qiling
- 2020, HITB UAE, Training, Lightweight Binary Analyzer
- 2020, Blackhat USA, Building IoT Fuzzer with Qiing
- 2020, Blackhat Singapore, Lightweight Binary Analyzer
- 2020, Blackhat Europe, Deep Dive Into Obfuscated Binary

Qiling Framework

- Cross platform and cross architecture binary instrumentation framework
- Emulate and instrument ARM, ARM64, MIPS, X86 and X8664
- Emulate and instrument Linux, MacOS, iphoneOS, Windows and FreeBSD
- High-level Python API access to register, CPU and memory
- 1,700+ Github star, more than 9,000+ pypi download, 60+ contributors worldwide
- Contributor from Dell, Intel, SentinalOne and etc

About lazymio && kabeor

~ $ whoami
Lazymio

~ $ file Lazymio
The sheperd lab, JD security, Security Engineer.
CTF player, member of Lancet.
GeekPwn 2019 Hall of Fame.

~ $ ls -l Lazymio
Reverse engineering.
Binary analysis.
Writing code for fun.

~ $ which Lazymio
Github: https://github.com/wtdcode
Blog: https://blog.lazym.io/
Twitter: https://twitter.com/pwnedmio

Name: kabeor

Security Engineer at The Shepherd Lab, JD Security.

Core developer of Qiling.

BlackHat Asia & Europe 2020 - Speaker

China kanxue SDC 2020 - Speaker

HITB Training 2020 - Speaker

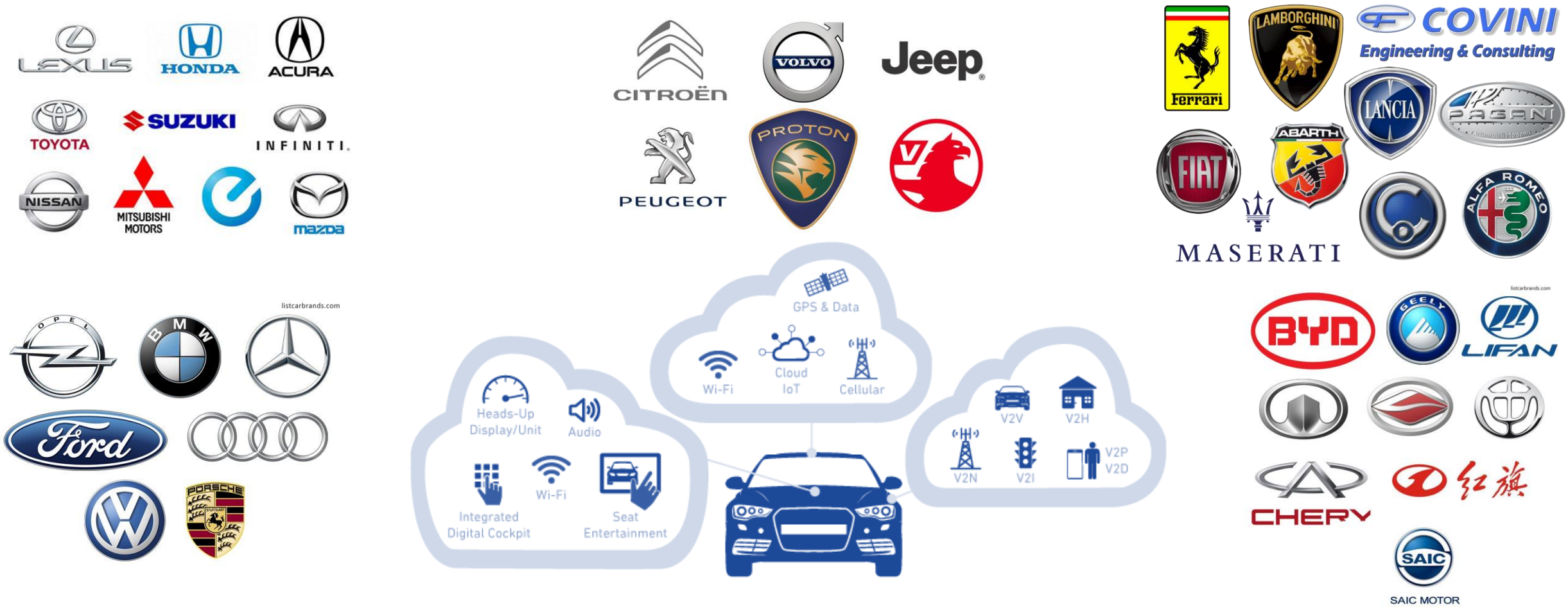
Github: https://github.com/kabeor

Blog: https://kabeor.cn

Twitter: https://twitter.com/Angrz3_K

Make IoT Reverse Engineering Great

Today’s IoT Analysis



To under a firmware

First, you need a IoT, If not too expensive to own one

How We Fix It

- GPS
- Wi-Fi
- Bluetooth
- Audio
- Screen

No Actual Hardware Needed

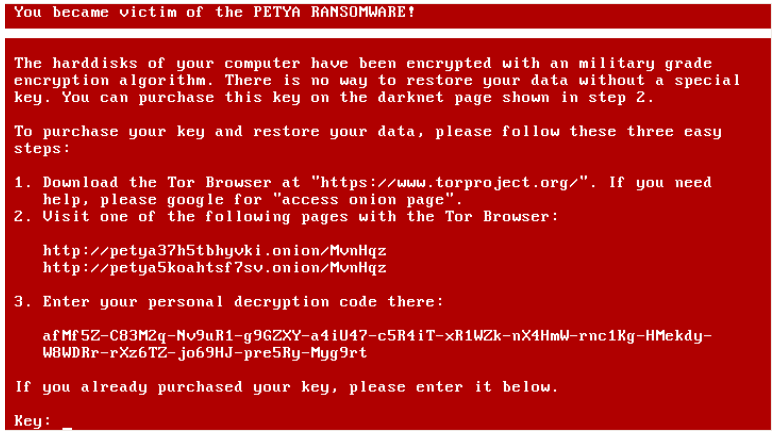
Qiling Framework

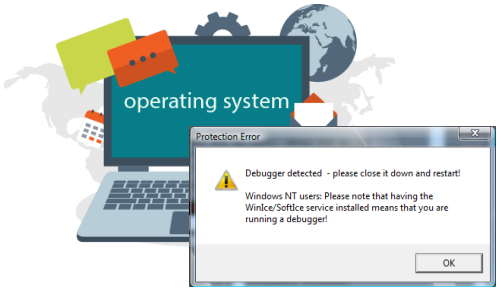
Bring the entire car firmware’s binary into emulation

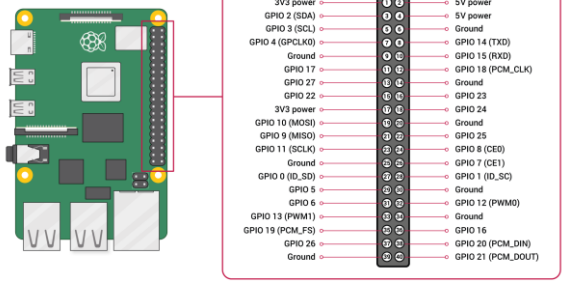
with virtual devices support

Wait, There are Virtual Machines

Current Virtual Machine Limitation







- MBR
- UEFI
- Smart Contract
- GPIO
- Anti-Anti Debug
- Qualcomm EDL

Most modern platform are either limited or NONE emulation or proper analysis tools

Qiling Framework

Overview

CPU Architecture

Loader

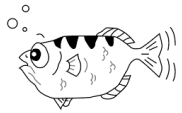
Posix

OS

EDL

MBR

Debugger



Extensions

coverage

idaplugin

report

sanitizers

windows sdk

Instrumentation

External Hardware Emulation

Qiling Framework

Features

- Cross platform: Windows, MacOS, Linux, BSD, UEFI, MBR
- Cross architecture: X86, X86_64, Arm, Arm64, MIPS, 8086
- Multiple file formats: PE, UEFI(PE), MachO, ELF, EDL (ELF), COM
- Emulate & sandbox machine code in a isolated environment
- Provide high level API to setup & configure the sandbox
- Fine-grain instrumentation: allow hooks at various levels (instruction/basic-block/memory-access/exception/syscall/IO/etc)
- Allow dynamic hotpatch on-the-fly running code, including the loaded library
- True Python framework, making it easy to build customized analysis tools on top
- GDBServer support - GDB/IDA/r2
- IDA Plugin
- OS profiling support

	8086	x86	x86-64	ARM	ARM64	MIPS
Windows (PE)	-	☑	☑	-	☐	-
Linux (ELF)	☐	☑	☑	☑	☑	☑
MacOS (MachO)	-	☐	☑	-	☐	-
BSD (ELF)	☐	☐	☑	☐	☐	☐
UEFI	-	☑	☑	-	-	-
DOS (COM)	☑	-	-	-	-	-
MBR	☑	-	-	-	-	-

Similarity

User Mode Emulation

qemu-usermode

- The TOOL
- Limited OS Support, Very Limited
- No Multi OS Support
- No Instrumentation
- **Syscall Forwarding**

usercorn

- Very good project !
- It’s a Framework !
- Mostly *nix based only
- Limited OS Support (No Windows)
- Go and Lua is not hacker’s friendly
- **Syscall Forwarding**

Binee

- Very good project too
- Only X86 (32 and 64)
- Limited OS Support
- Only PE Files
- Just a tool, we don’t need a tool
- Again, is GO

WINE

- Limited ARCH Support
- Limited OS Support, only Windows
- Not Sandbox Designed
- No Instrumentation

Speakeasy

- Very good project too
- X86 32 and 64
- PE files and Driver
- Limited OS Support
- Only Windows

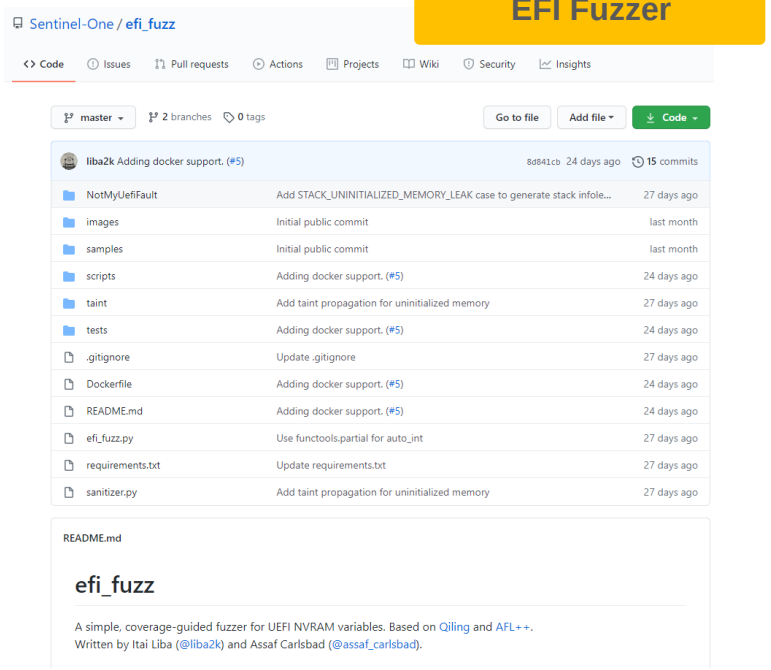
Zelos

- Very good project !
- It’s a Framework !
- Linux based only (No Windows)
- Incomplete support for Linux multi arch

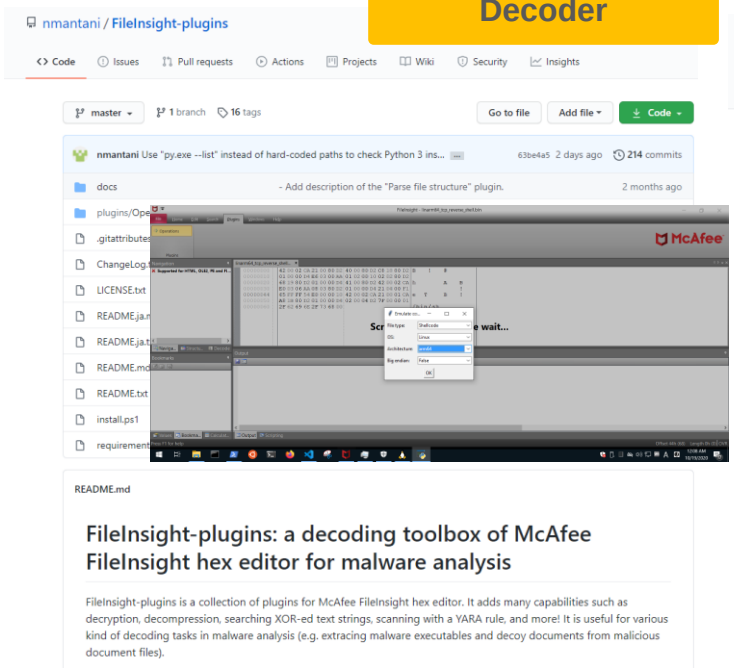
Framework

Framework, NOT Tools

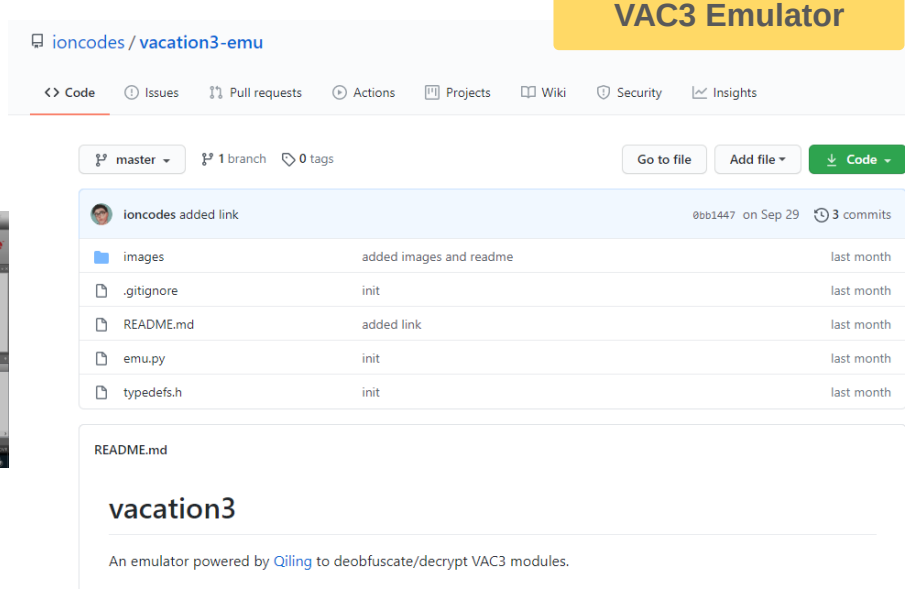
EFI Fuzzer



Decoder



VAC3 Emulator



Binary Fuzzer

IoT Fuzzer

Malware Sandbox

CTF Solver

IoT Emulator

MacOS Emulator

IOS Emulator

Binary Decrypt

Qiling Framework

CPU Architecture

Loader

OS

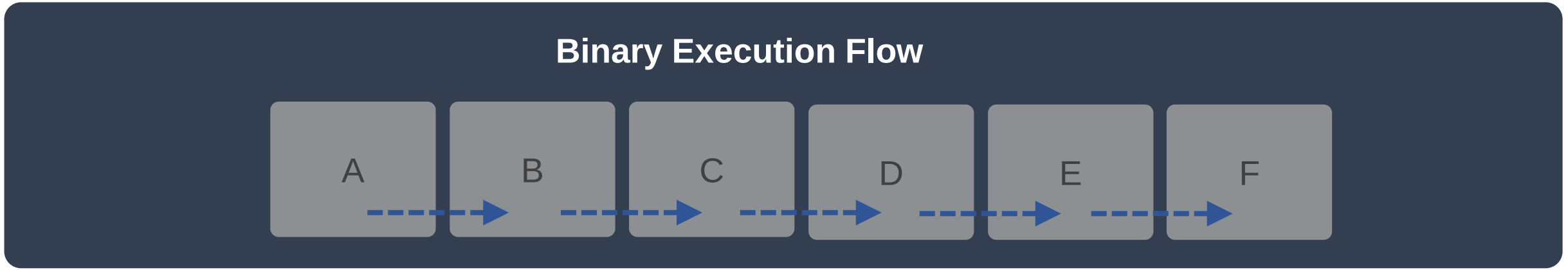
Debugger

Extensions

Instrumentation (Qiling's API)

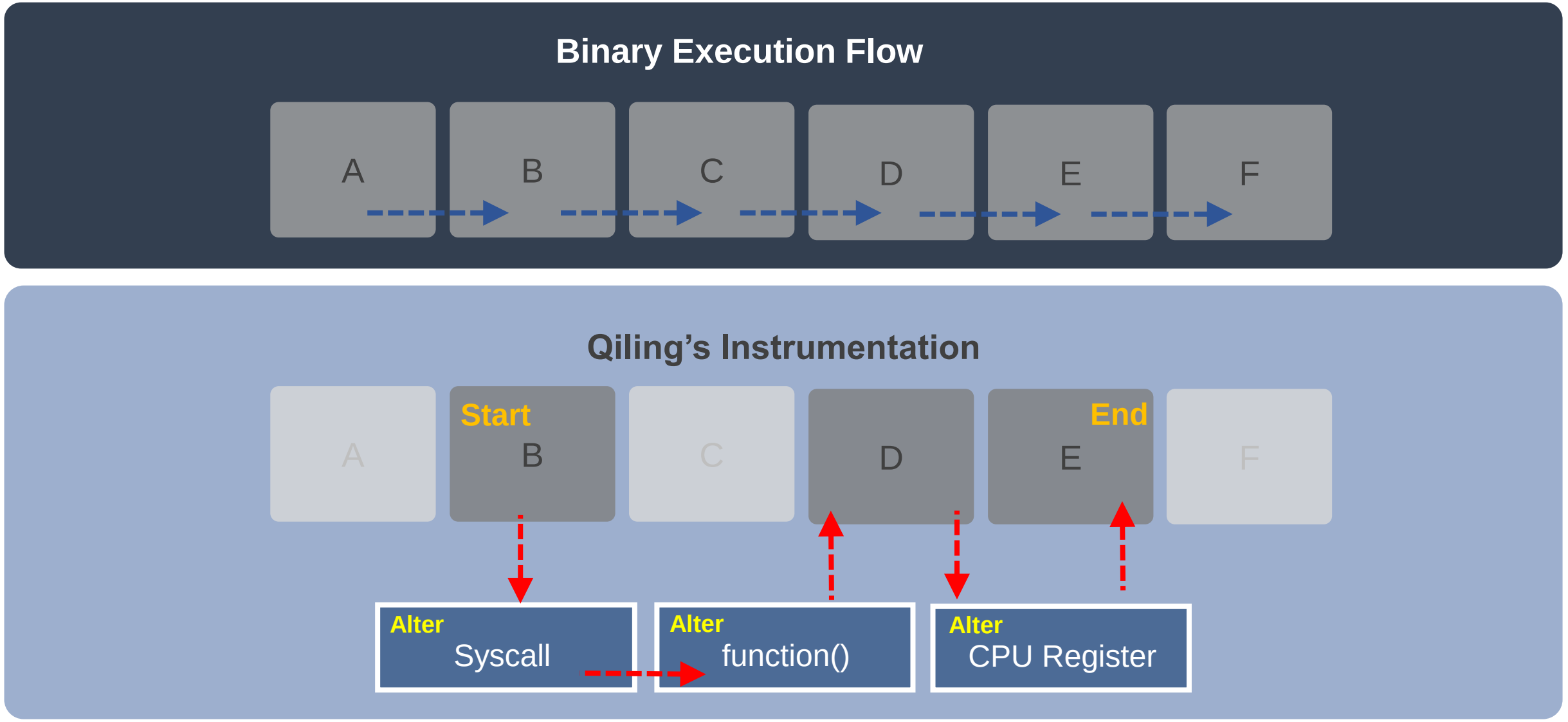
Instrumentation

What Is Instrumentation



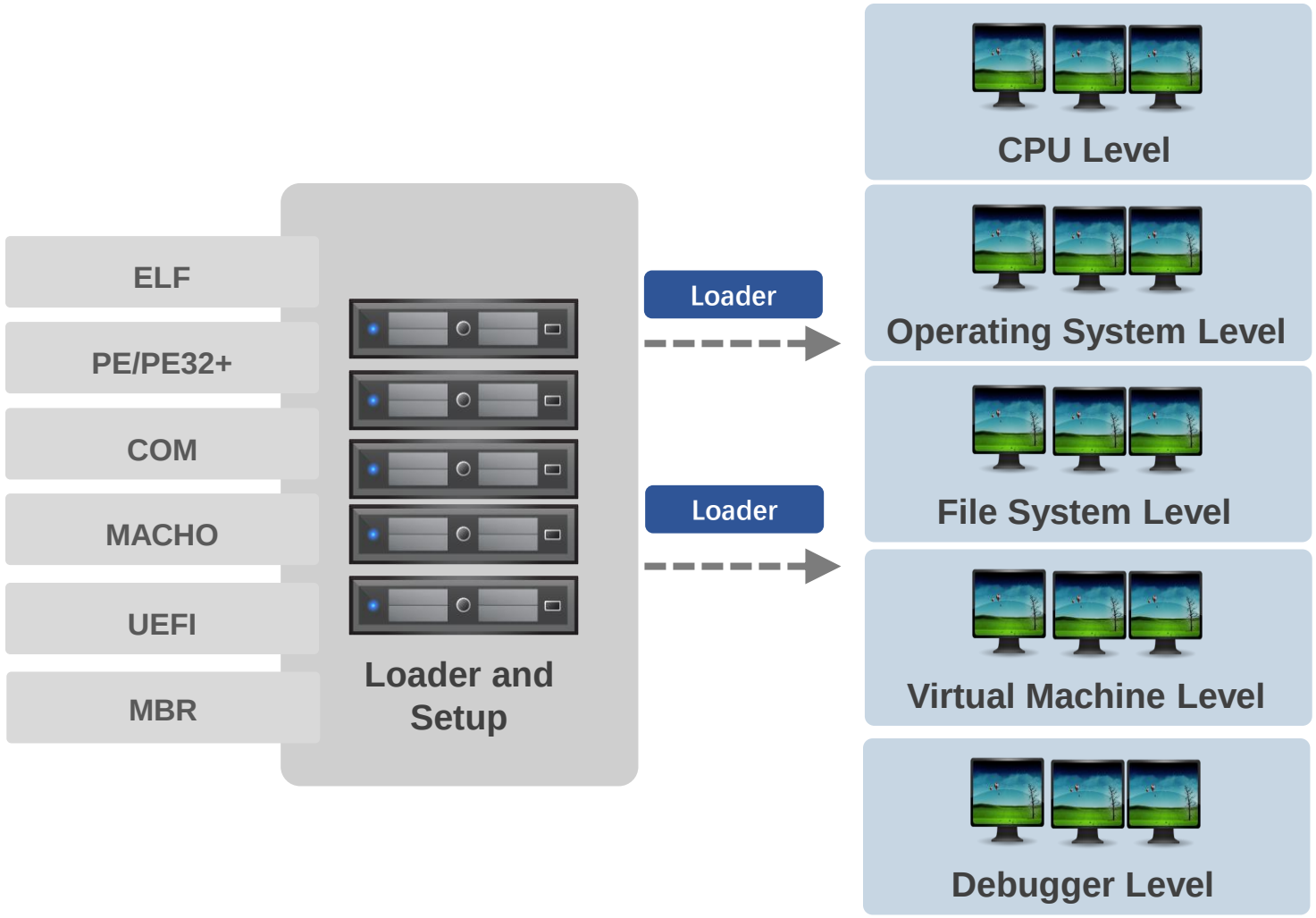
One Function After Another

What Is Instrumentation



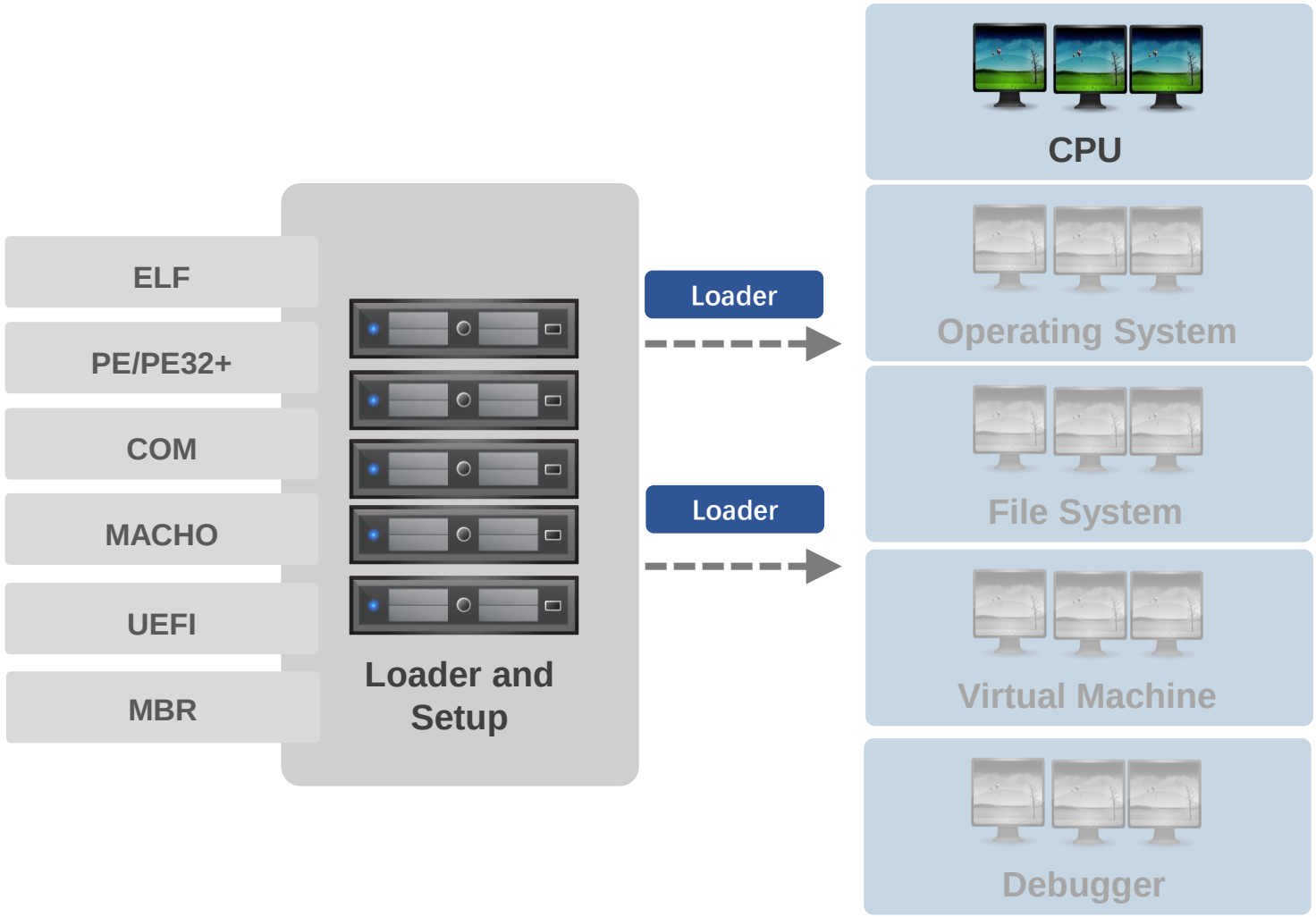
Qiling and APIs

Qiling Framework and Its Interface



More APIs: https://docs.qiling.io

CPU Instrumentation



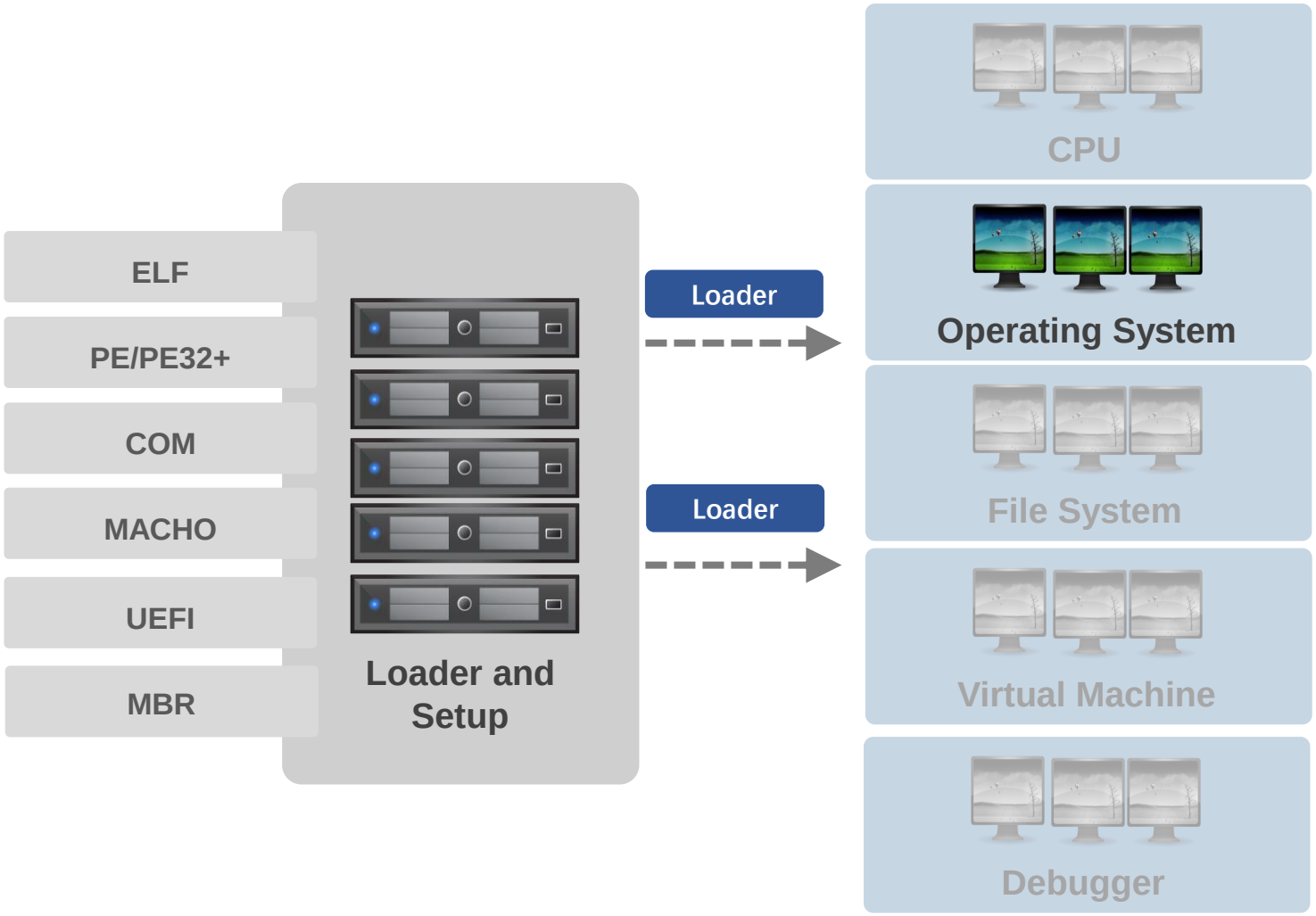
- Access to Register
- Reading register
  - ql.reg.eax
- Writing to register
  - ql.reg.eax = 0x41
- Differet Hooks
  - ql.hook_code()
  - ql.hook_address()
  - and more

CPU Instrumentation: Examples

```
def my_puts(ql):
    addr = ql.os.function_arg[0]
    print("puts(%s)" % ql.mem.string(addr))
    reg = ql.reg.read("rax")
    print("reg : 0x%x" % reg)
    ql.reg.rax = reg
    self.set_api = reg

def write_onEnter(ql, arg1, arg2, arg3, *args):
    print("enter write syscall!")
    ql.reg.rsi = arg2 + 1
    ql.reg.rdx = arg3 - 1
    self.set_api_onenter = True

def write_onexit(ql, arg1, arg2, arg3, *args):
    print("exit write syscall!")
    ql.reg.rax = arg3 + 1
    self.set_api_onexit = True

ql = Qiling(["../examples/rootfs/x8664_linux/bin/x8664_args","1234test", "12345678", "bin/x8664_hello"],  "../e
ql.set_syscall(1, write_onEnter, QL_INTERCEPT.ENTER)
ql.set_api('puts', my_puts)
ql.set_syscall(1, write_onexit, QL_INTERCEPT.EXIT)
ql.mem.map(0x1000, 0x1000)
ql.mem.write(0x1000, b"\xFF\xFE\xFD\xFC\xFB\xFA\xFB\xFC\xFC\xFE\xFD")
ql.mem.map(0x2000, 0x1000)
ql.mem.write(0x2000, b"\xFF\xFE\xFD\xFC\xFB\xFA\xFB\xFC\xFC\xFE\xFD")
ql.run()
```

- Access to Register
- Reading register
  - eax = ql.reg.eax
- Writing to register
  - ql.reg.eax = 0x41
- Differet Hooks
  - ql.hook_code()
  - ql.hook_address()
  - and more

Qiling Framework: Operating System



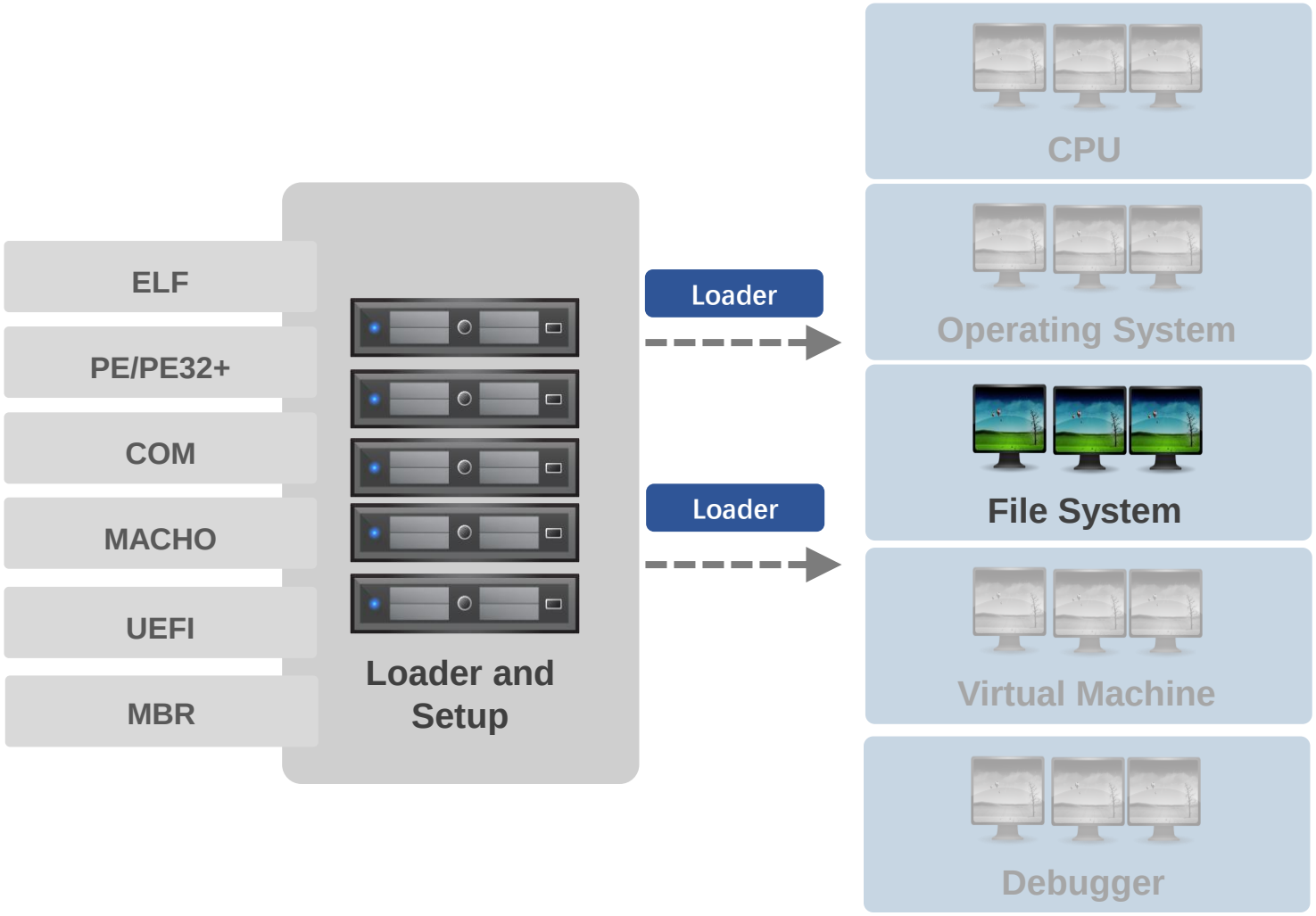
- Access to memory
  - ql.mem.read()
  - ql.mem.write()
- Search pattern from memory
  - ql.mem.search()
- Stack related operation
  - ql.stack_pop
  - ql.stack_push
- Syscall replacement
  - ql.set_syscall()
  - ql.set_api()
- Replace library call with
  - ql.set_api()

Example: Operating System

```python
from qiling import *

def my_syscall_write(ql, write_fd, write_buf, write_count, *args, **kw):
    regreturn = 0

    try:
        buf = ql.mem.read(write_buf, write_count)
        ql.nprint("\n+++++++++\nmy write(%d,%x,%i) = %d\n+++++++++" % (write_fd, write_buf, write_count, regreturn))
        ql.os.fd[write_fd].write(buf)
        regreturn = write_count
    except:
        regreturn = -1
        ql.nprint("\n+++++++++\nmy write(%d,%x,%i) = %d\n+++++++++" % (write_fd, write_buf, write_count, regreturn))
        if ql.output in (QL_OUTPUT.DEBUG, QL_OUTPUT.DUMP):
            raise

    ql.os.definesyscall_return(regreturn)


if __name__ == "__main__":
    ql = Qiling(["rootfs/arm_linux/bin/arm_hello"], "rootfs/arm_linux", output = "debug")
    # Custom syscall handler by syscall name or syscall number.
    # Known issue: If the syscall func is not be implemented in qiling, qiling does
    # not know which func should be replaced.
    # In that case, you must specify syscall by its number.
    ql.set_syscall(0x04, my_syscall_write)

    # set syscall by syscall name
    #ql.set_syscall("write", my_syscall_write)

    ql.run()
```

- Access to memory
  - ql.mem.read()
  - ql.mem.write()
- Search pattern from memory
  - ql.mem.search()
- Stack related operation
  - ql.stack_pop
  - ql.stack_push
- Syscall replacement
  - ql.set_syscall()
  - ql.set_api()
- Replace library call with
  - ql.set_api()

File System Instrumentation



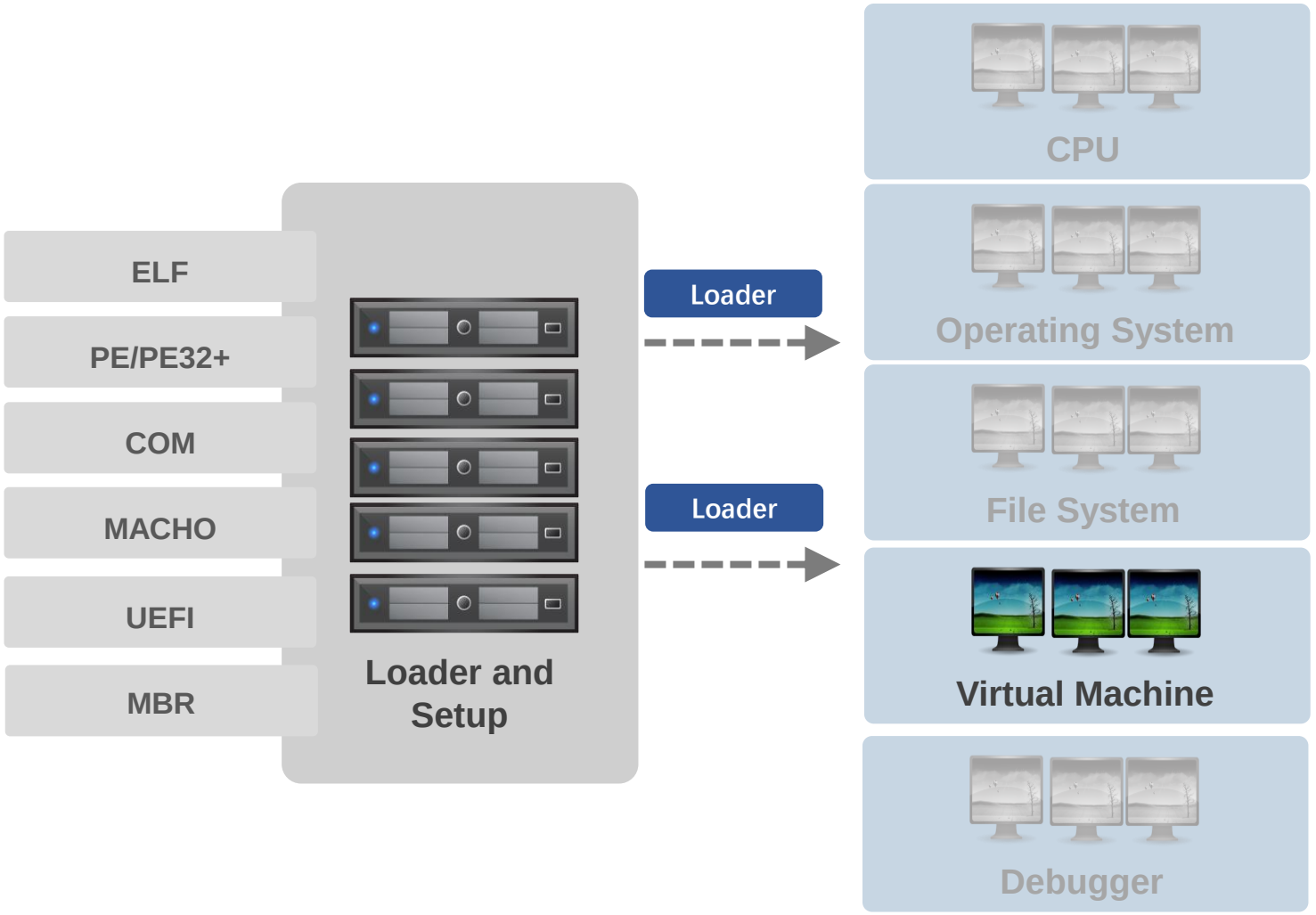
- Map host file
  - ql.fs_mapper()
- Hijack accessed file
  - ql.fs_mapper(hijack_func)
- Stdio replacement
  - stdin
  - stdout
  - stderr
- Patch file's memory before execution
  - ql.patch

File System Instrumentation

```python
from qiling import *
from qiling.os.mapper import QlFsMappedObject

class Fake_urandom(QlFsMappedObject):

    def read(self, size):
        return b"\x01" # fixed value for reading /dev/urandom

    def fstat(self): # syscall fstat will ignore it if return -1
        return -1

    def close(self):
        return 0

if __name__ == "__main__":
    ql = Qiling(["rootfs/x86_linux/bin/x86_fetch_urandom"], "rootfs/x86_linux")
    ql.add_fs_mapper("/dev/urandom", Fake_urandom())
    ql.run()
```

- Map host file
  - ql.fs_mapper()
- Hijack accessed file
  - ql.fs_mapper(hijack_func)
- Stdio replacement
  - stdin
  - stdout
  - stderr
- Patch file’s memory before execution
  - ql.patch

Virtual Machine Instrumentation



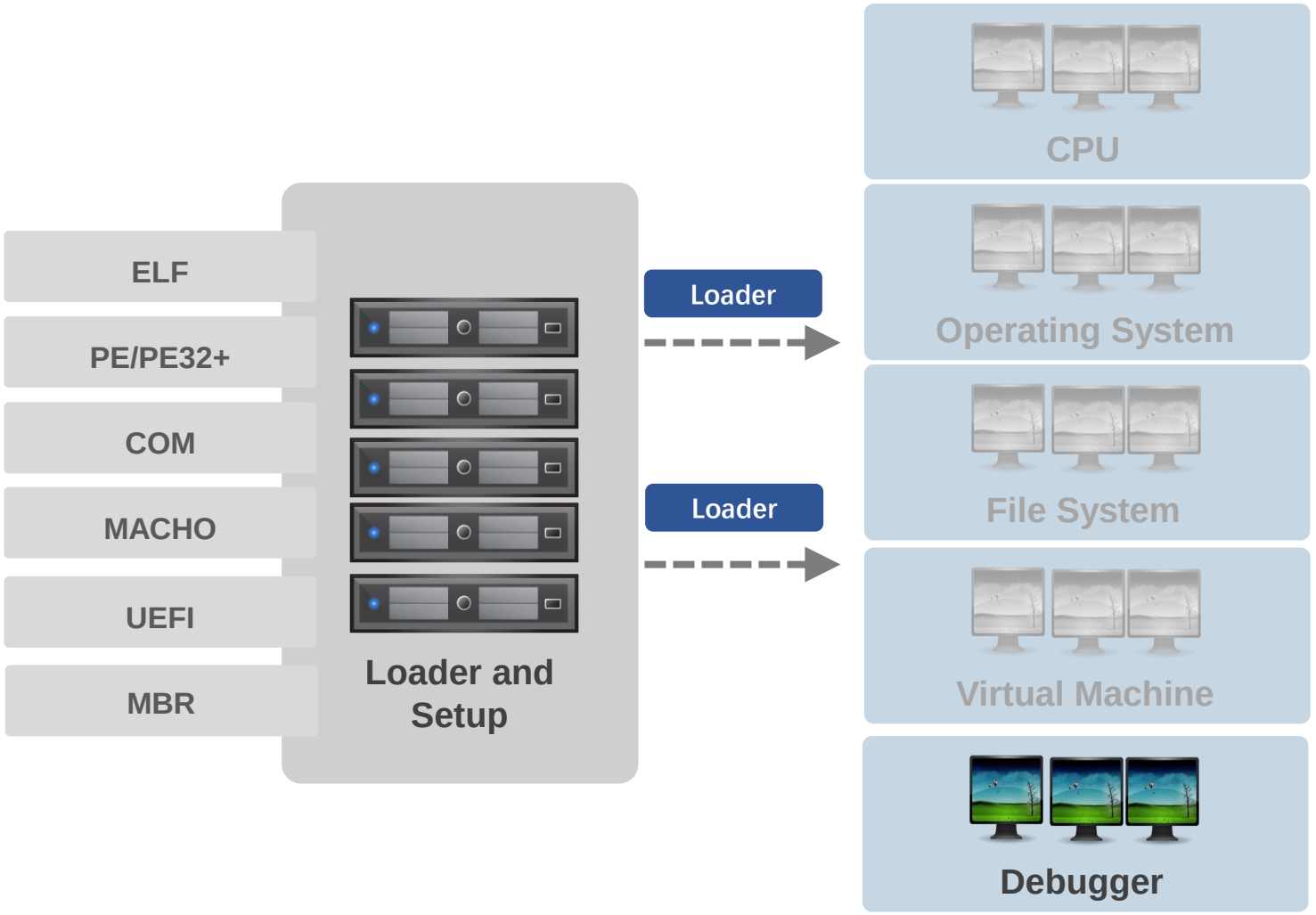
- Save current state
  - ql.save()
- Restore current state
  - ql.restore()
- Save/restore memory only
  - ql.mem.save()
- Save/restore register only
  - ql.reg.save()

Virtual Machine Instrumentation

```python
def save_context(ql, *args, **kw):
    ql.save(cpu_context=False, snapshot="snapshot.bin")

def patcher(ql):
    br0_addr = ql.mem.search("br0".encode() + b'\x00')
    for addr in br0_addr:
        ql.mem.write(addr, b'lo\x00')


def check_pc(ql):
    print("=" * 50)
    print("[!] Hit fuzz point, stop at PC = 0x%x" % ql.reg.arch_pc)
    print("=" * 50)
    ql.emu_stop()


def my_sandbox(path, rootfs):
    ql = Qiling(path, rootfs, output="debug", verbose=5)
    ql.add_fs_mapper("/dev/urandom","/dev/urandom")
    ql.hook_address(save_context ,0x10930)
    ql.hook_address(patcher, ql.loader.elf_entry)
    ql.hook_address(check_pc,0x7a0cc)
    ql.run()

if __name__ == "__main__":
    nvram_listener_therad =  threading.Thread(target=nvram_listener, daemon=True)
    nvram_listener_therad.start()
    my_sandbox(["rootfs/bin/httpd"], "rootfs")
```

- Save current state
  - ql.save()
- Restore current state
  - ql.restore()
- Save/restore memory only
  - ql.mem.save()
- Save/restore register only
  - ql.reg.save()

Debugger



- Open API for RSP compatible Debugger
- Build In debugger – Qdbg
  - Able to reverse debug

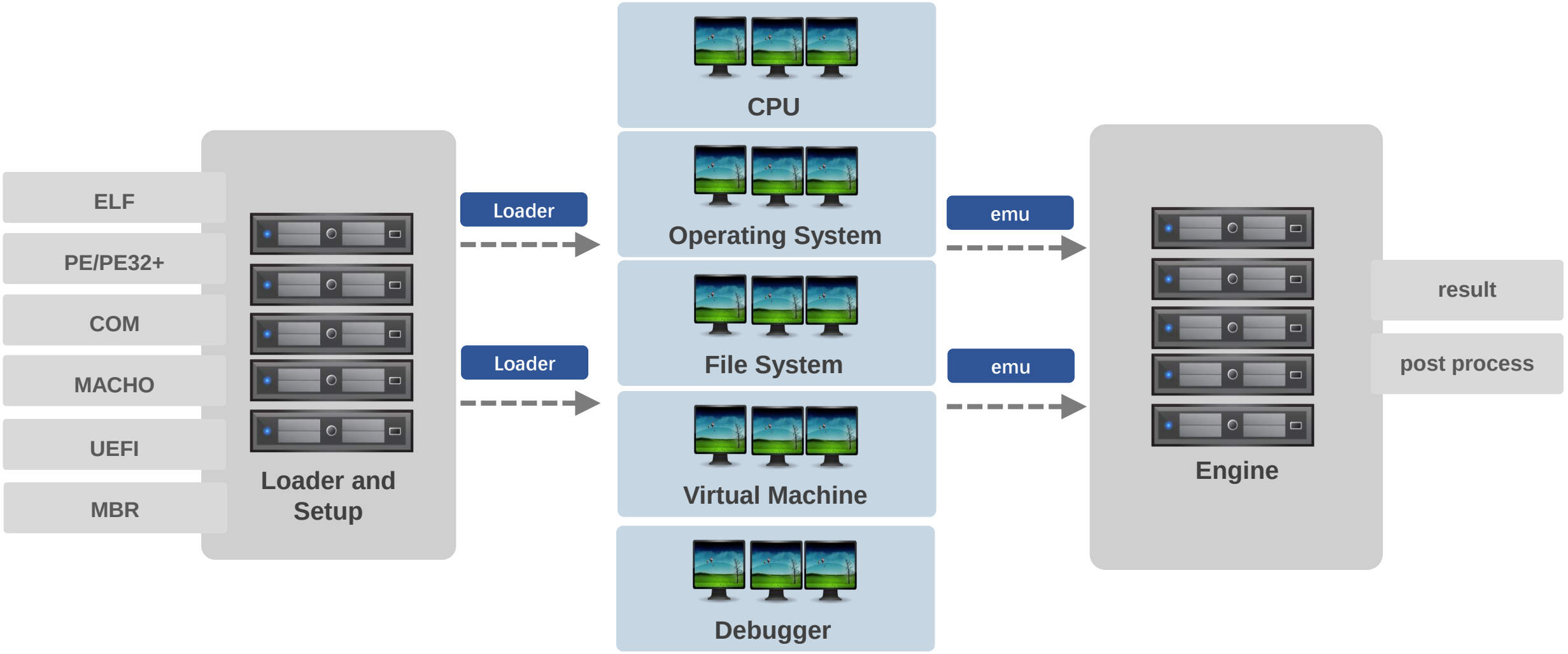
Debugger

```python
def run_sandbox(path, rootfs, output):
    ql = Qiling(path, rootfs, output = output)
    ql.multithread = False
    ql.debugger = "qdb:rr" # switch on record and replay with rr
    # ql.debugger = "qdb:" # enable qdb without options
    ql.run()


if __name__ == "__main__":
    run_sandbox(["rootfs/arm_linux/bin/arm_hello"], "rootfs/arm_linux", "debug")
```

- Open API for RSP compatible Debugger
- Build In debugger – Qdbg
  - Able to reverse debug

```python
from qiling import *

if __name__ == "__main__":
    ql = Qiling(["rootfs/x8664_linux/bin/x8664_hello"], "rootfs/x8664_linux", output = "debug")
    ql.debugger = "gdb:0.0.0.0:9999"
    ql.run()
```

Qiling Framework: In a Nutshell



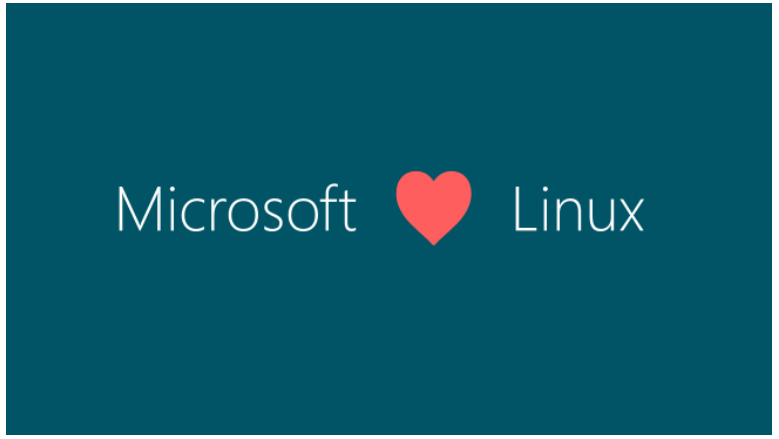
Base OS can be Windows/Linux/BSD or OSX

And not limited to ARCH

Preview

Demo Setup

- **ONLY If you wish to try yourself**



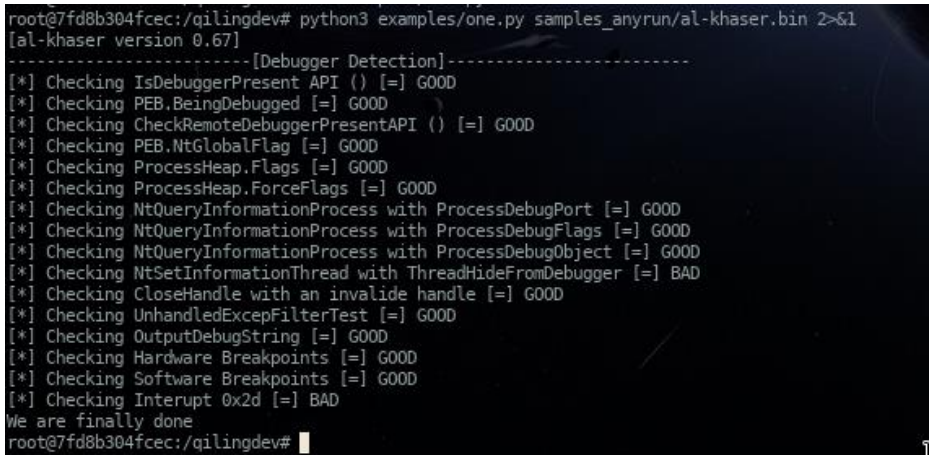
- Required OS
  - Ubuntu 18.04 / 20.04
  - WSL2

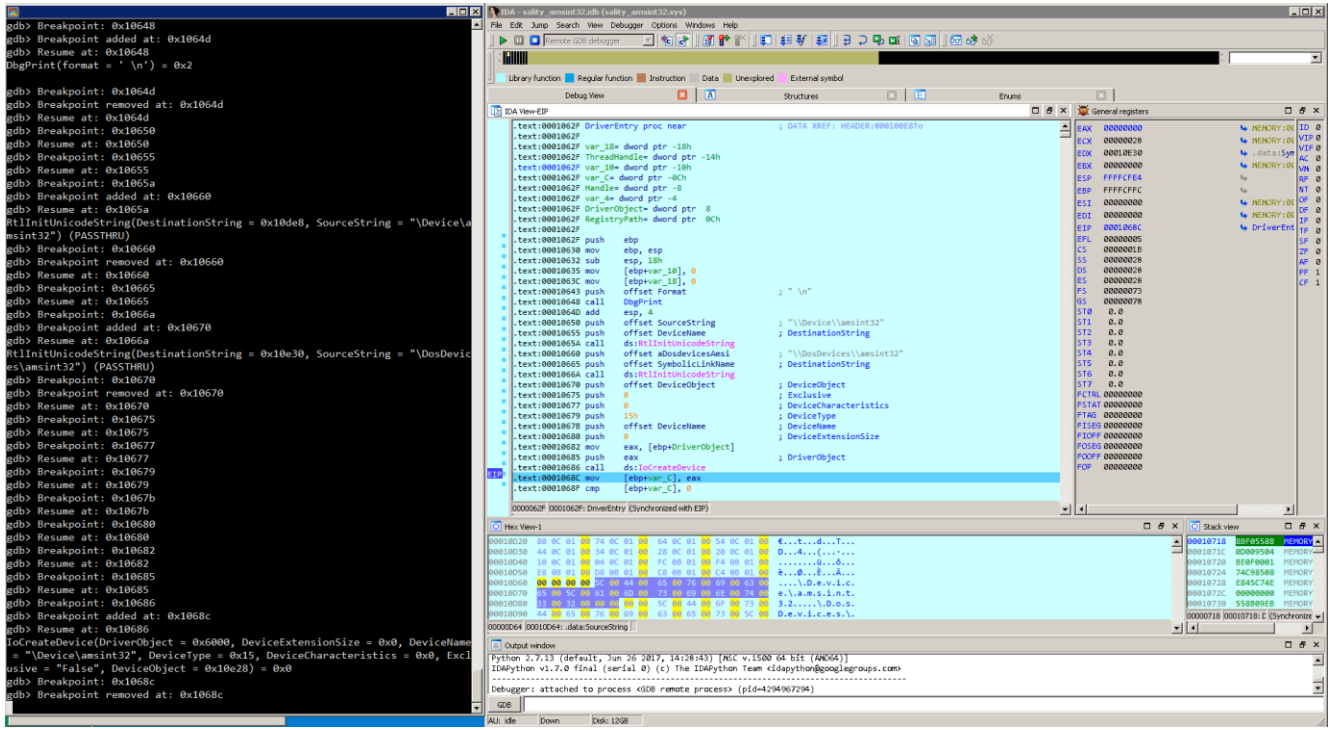
- Install Qiling Framework
  - sudo apt-get update
  - sudo apt-get upgrade
  - sudo apt install python3-pip git cmake build-essential libtool-bin python3-dev automake flex bison libglib2.0-dev libpixman-1-dev clang python3-setuptools llvm
  - git clone https://github.com/qilingframework/qiling.git
  - pip3 install --user https://github.com/qilingframework/qiling/archive/dev.zip

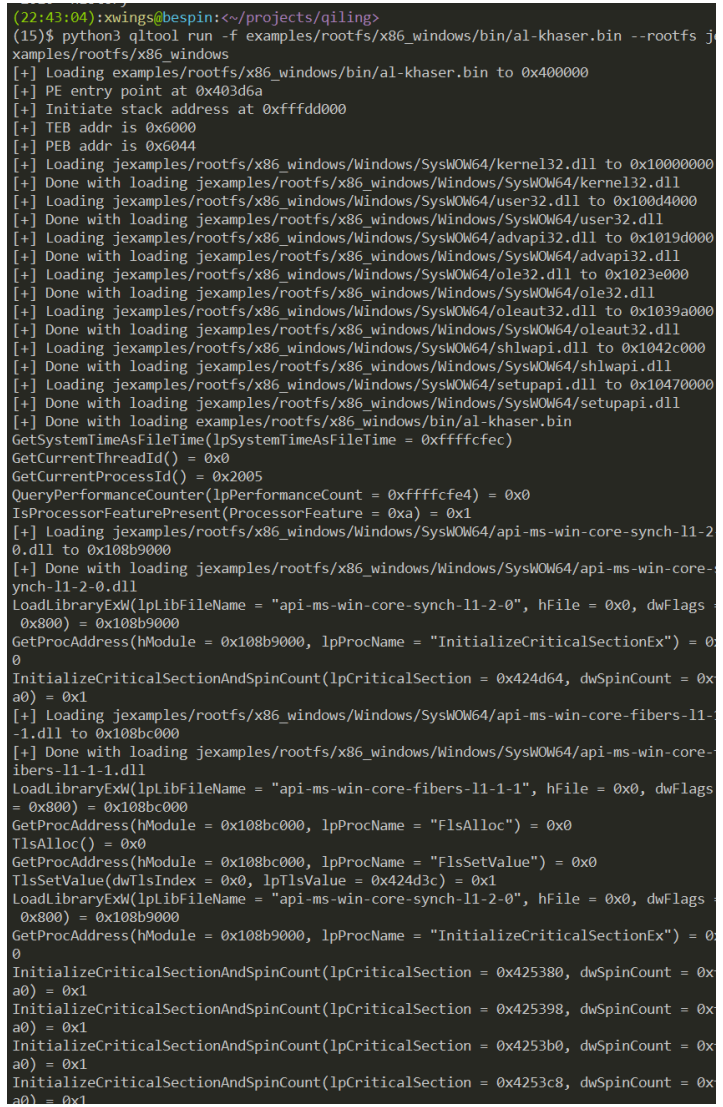
- Install AFL++
  - git clone https://github.com/AFLplusplus/AFLplusplus.git
  - cd AFLplusplus
  - make
  - cd unicorn_mode
  - ./build_unicorn_support.sh

Malware & Rootkit Analysis

- Support Both Win32/64
- Support PE and System Driver
- Anti-Anti Debug
- Script able
- Cross platform support

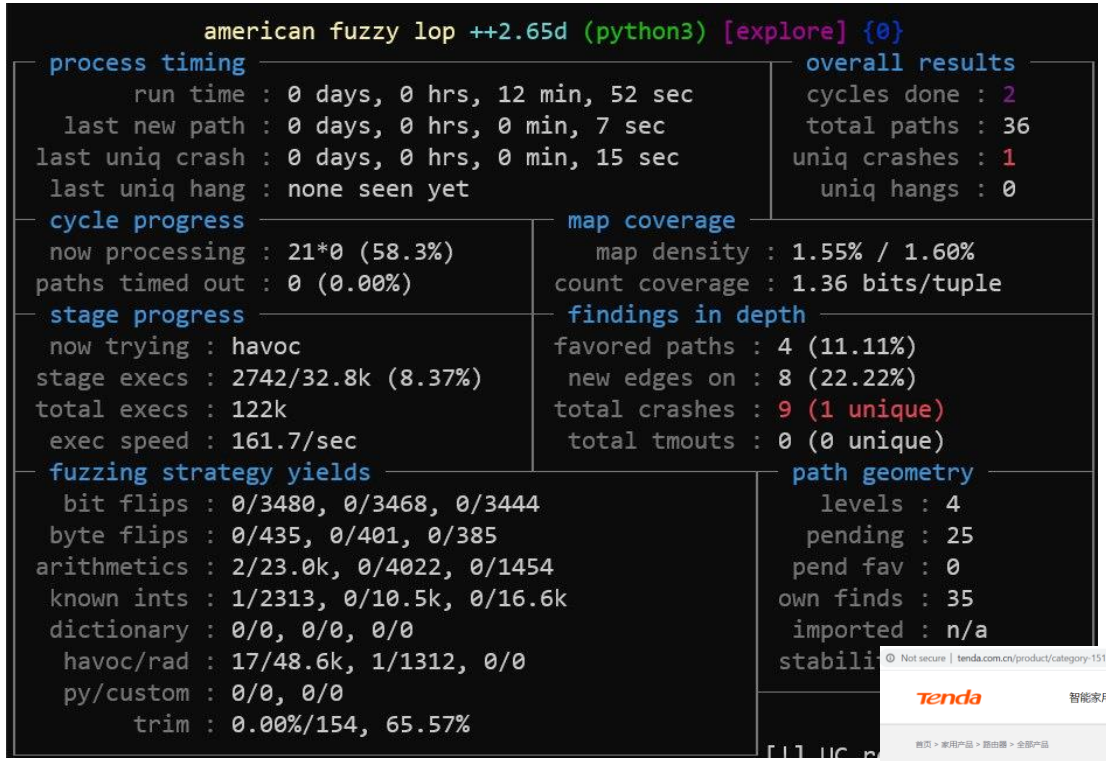


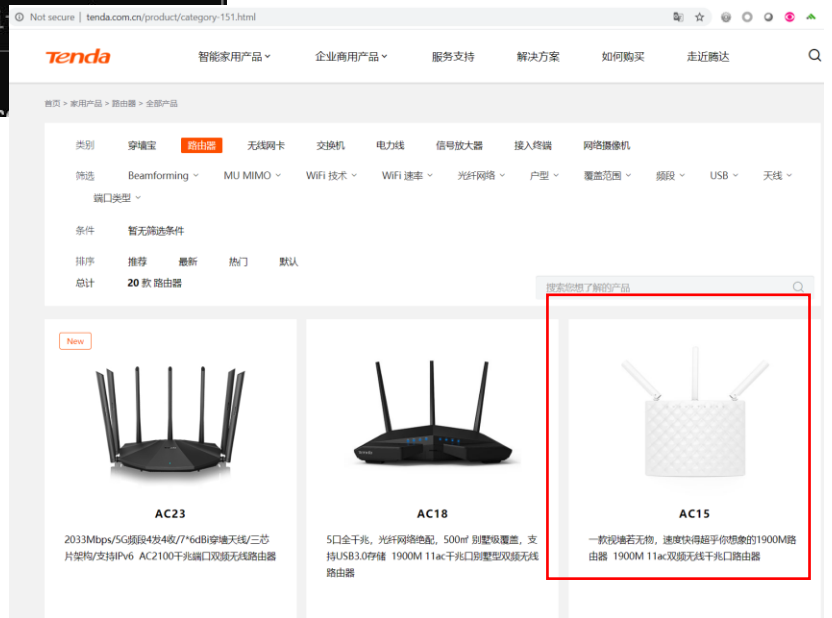




Fuzzer

- Required Firmware
  - AC15
- Run Tenda AC15
  - start_tendaac15_httpd.py
  - Test with browser
- Check crash point
  - addressNat_overflow.sh
- How to find and save snapshot
  - saver_tendaac15_httpd.py
- How to build and run fuzzer
  - fuzz_tendaac15_httpd.py

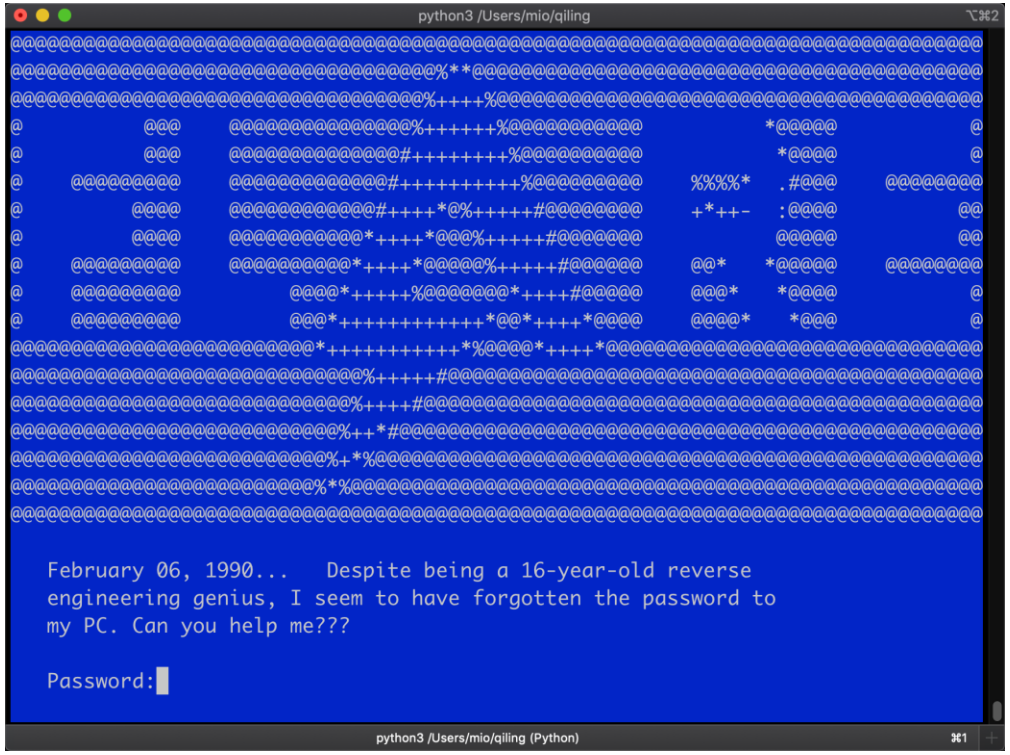


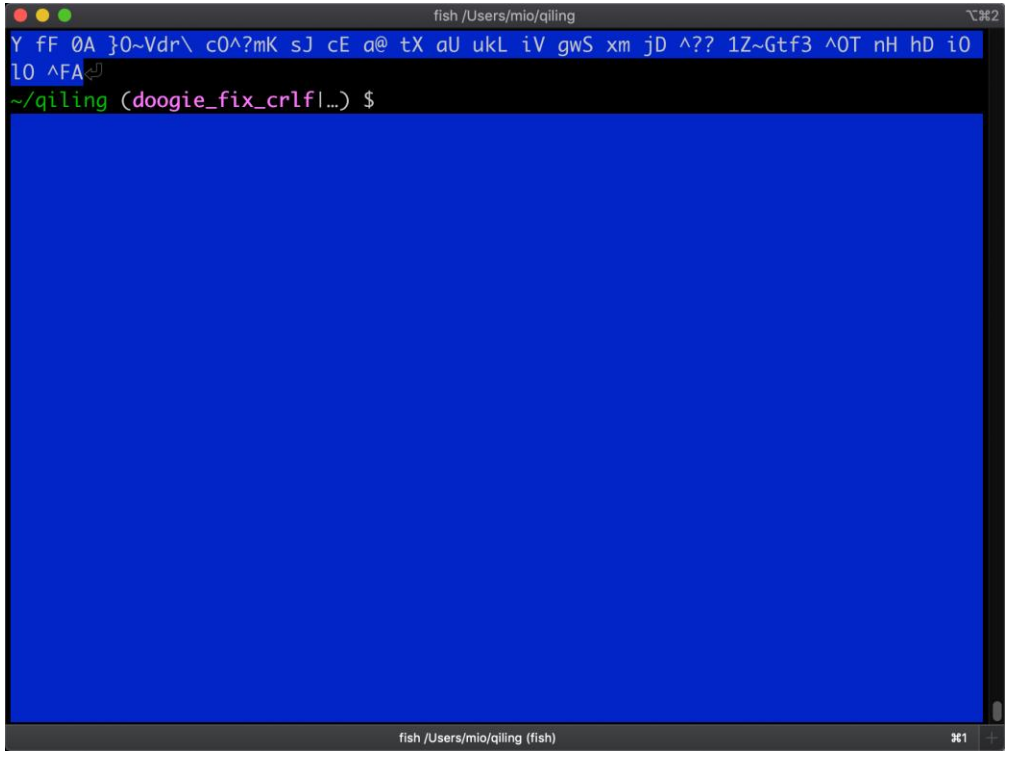


MBR Analysis

- Sample:
  - Flare-On 5 (2018) Challenge 8 - doogie
- MBR file
- Quick look by qltool.
  - python3 qltool run -f examples/rootfs/8086/doogie/doogie.bin --rootfs examples/rootfs/8086/ --console False
- Try some inputs, but only get gibberish.
- Tips: Feburary 06, 1990.

```
~/q/e/r/8/doogie (doogie|…) $ file doogie.bin
doogie.bin: DOS/MBR boot sector; partition 1 : ID=0x7, activ
e, start-CHS (0x0,32,33), end-CHS (0x3ff,254,63), startsecto
r 2048, 41938944 sectors
~/q/e/r/8/doogie (doogie|…) $
```





IDA Plugin

Setup
Reload User Scripts

Execute Till
Execute Selection
Continue
Set PC
Step ^⇧F9
Edit Register

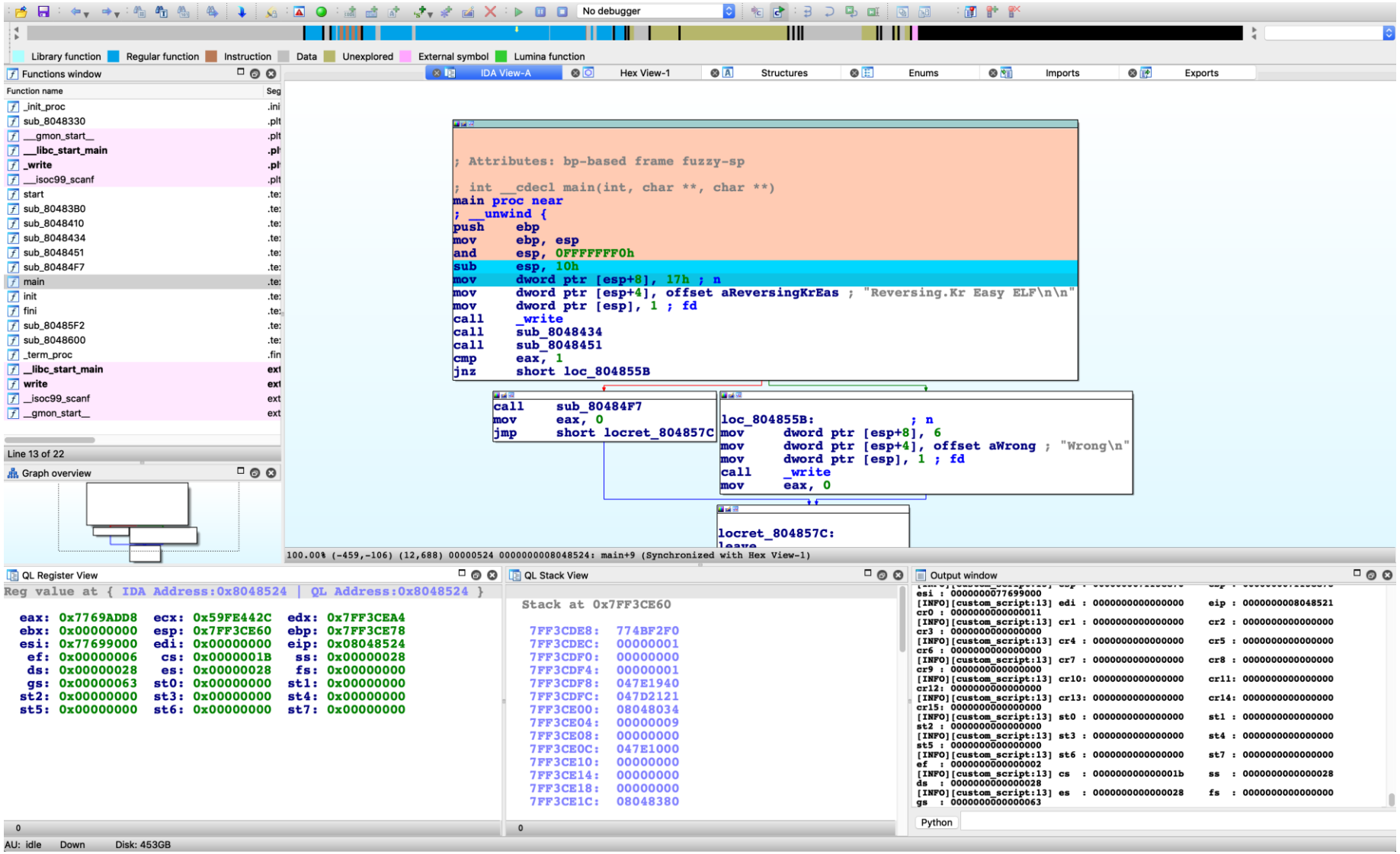
Restart

View Register
View Stack
View Memory

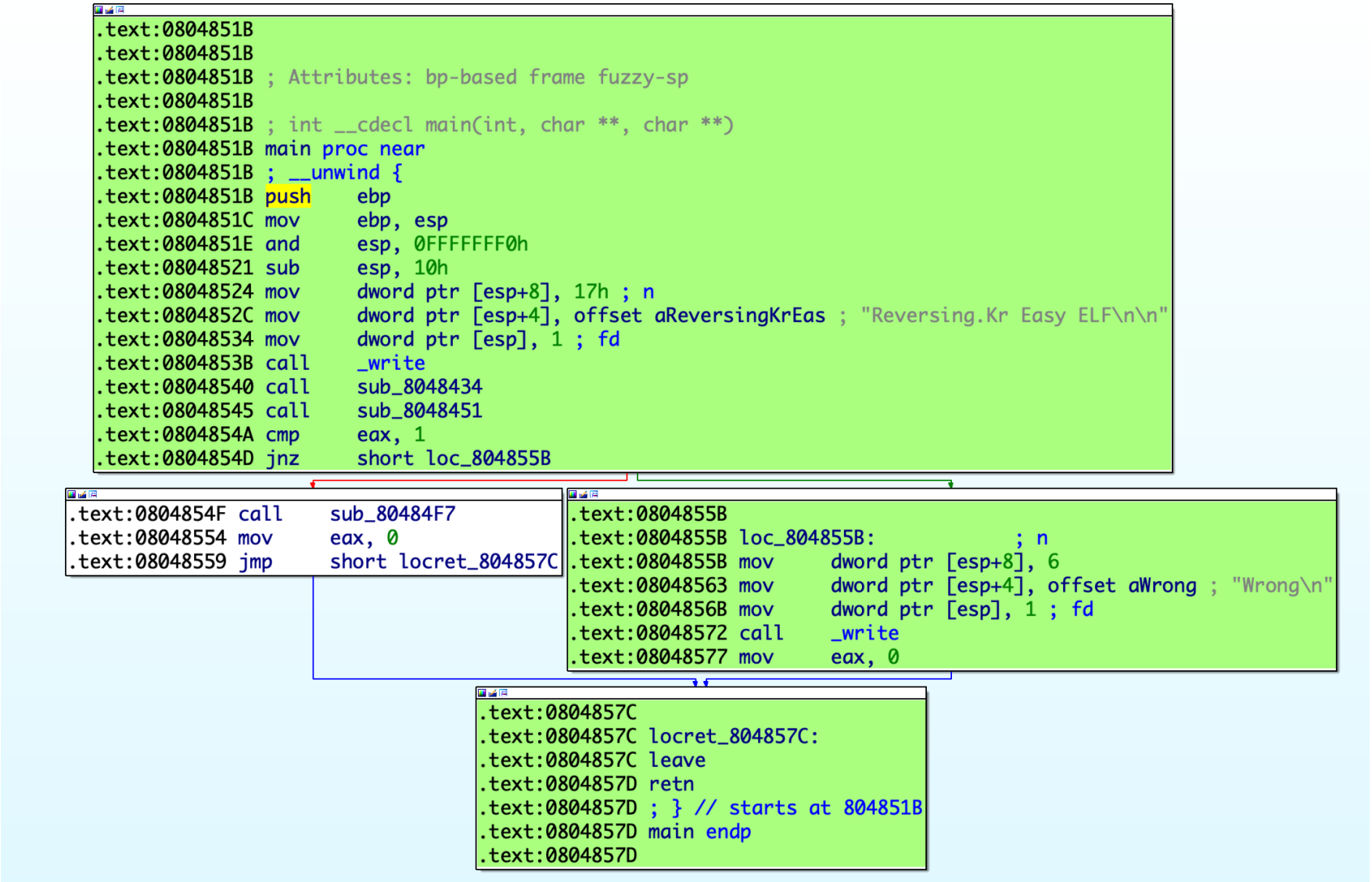
Save Snapshot
Load Snapshot

Auto Analysis For Deflat
Mark as Real Block
Mark as Fake Block
Mark as Return Block
Deflat

Remove Junk Code by Patterns
Nop Items without Color



Execution path drawing



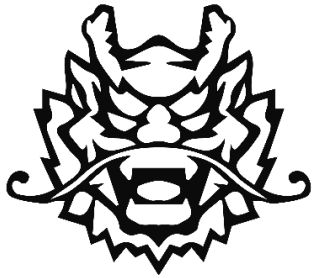
Demo 0

Prepare for Qiling IDA plugin

IDA 7.4+

Python3

Qiling



Demo

- Analysis sample:
  - Flare-On 5(2018) Challenge 6 - magic

- This challenge required enter 666 keys
- Only call validation function after each input

```
v20 = 0x2C16022663LL;
memset(v21, 0, sizeof(v21));
for ( i = 0; i < strlen("Run, Forrest, run!!"); i += 4 )
  seed ^= *&aRunForrestRun[i];
srand(seed);                                    // seed
if ( sub_402D47() )
{
  v3 = strlen(a2[1]);
  sub_402DCF(a2[1], v3, v11);
  sub_4037BF(*a2);
  puts("Generated first permutation!");
  exit(0);
}
puts("Welcome to the ever changing magic mushroom!");
printf("%d trials lie ahead of you!\n", 666LL);
for ( j = 0; j < 666; ++j )
{
  printf("Challenge %d/%d. Enter key: ", j + 1, 666LL);
  if ( !fgets(v10, 128, stdin) )                // input len=128
    return 0xFFFFFFFFLL;
  v5 = strlen(v10);
  sub_402DCF(v10, v5, v11);                     // validate input && change self
  for ( k = 0; k < strlen(v10); ++k )
    *(&v12 + k) ^= v10[k];
  sub_4037BF(*a2);
}
printf("Congrats! Here is your price:\n%s\n", &v12);
return 0LL;
}
```

Prepare for Qiling IDA plugin

https://raw.githubusercontent.com/kabeor/qiling/dev/examples/extensions/idaplugin/FlareOn5_6.py
https://github.com/asuna-amawaka/Flareon5-2018/raw/master/Ch6%20-%20magic.7z

```
python3 –m venv qiling_env
source qiling_env/bin/activite
cd qiling.git
python –m pip –e .
```

```
import sys
sys.path.append('/Users/home/Pipenv/qiling_run/lib/python3.7/site-packages')
sys.path.append('/Users/home/Documents/Codes/Python/qiling_dev/qiling')
```

Sample Analysis

- The validation function validates the user input in a loop.
- In each iteration of the validation loop, a different part of the entered key is validated.

```
__int64 __fastcall sub_402DCF(__int64 a1, unsigned __int64 a2, __int64 a3)
{
  __int64 result; // rax
  unsigned int i; // [rsp+2Ch] [rbp-4h]

  for ( i = 0; ; ++i )
  {
    result = i;
    if ( i >= 33uLL )
      break;
    if ( dword_605108[72 * i + 1] + dword_605108[72 * i + 2] > a2 )
      sub_402CC7();
    sub_402CDF((&off_605100)[36 * i], dword_605108[72 * i], *(_QWORD *)&dword_605108[72 * i + 4]);
    if ( !((unsigned int (__fastcall *)(__int64, _QWORD, char (**)[3]))(&off_605100)[36 * i])(
            dword_605108[72 * i + 1] + a1,
            dword_605108[72 * i + 2],
            &off_605100 + 36 * i + 4) )
    {
      sub_402CDF((&off_605100)[36 * i], dword_605108[72 * i], *(_QWORD *)&dword_605108[72 * i + 4]);
      sub_402CC7();
    }
    sub_402CDF((&off_605100)[36 * i], dword_605108[72 * i], *(_QWORD *)&dword_605108[72 * i + 4]);
    memcpy(
      (void *)(dword_605108[72 * i + 3] + a3),
      (const void *)(dword_605108[72 * i + 1] + a1),
      dword_605108[72 * i + 2]);
  }
  return result;
}
```

Sample Analysis

- After analysis, it is found that the verification algorithm is stored in magic_table.
- ”magic_table” is **encrypted** but it will **decrypt in memory** in each iteration of the validation loop.
- The verification algorithm will automatically change after each round of verification.

```
__int64 __fastcall sub_402DCF(__int64 input, unsigned __int64 input_len, __int64 a3)
{
  __int64 result; // rax
  unsigned int i; // [rsp+2Ch] [rbp-4h]

  for ( i = 0; ; ++i )
  {
    result = i;
    if ( i >= 33uLL )
      break;
    if ( magic_table[i].input_offset + magic_table[i].input_len > input_len )
      failed();                                 // exit
    Enc_Dec(magic_table[i].func, magic_table[i].func_sz, magic_table[i].key);
    if ( !(magic_table[i].func)(magic_table[i].input_offset + input, magic_table[i].input_len, magic_table[i].target) )
    {
      Enc_Dec(magic_table[i].func, magic_table[i].func_sz, magic_table[i].key);
      failed();                                 // exit
    }
    Enc_Dec(magic_table[i].func, magic_table[i].func_sz, magic_table[i].key);
    memcpy((magic_table[i].output_offset + a3), (magic_table[i].input_offset + input), magic_table[i].input_len);// change opcode
  }
  return result;
}
```

```
.text:000000000400C55 87 50 57                byte_400C55     db 87h, 50h, 57h        ; DATA XREF: .data:magic_table+o
.text:000000000400C58 83 07 A1 BE 47 08 46 21+                dq 21460847BEA10783h, 5A16E746A4712F14h, 0E4CD87E5A60282A1h
.text:000000000400C58 14 2F 71 A1 46 E7 16 5A+                dq 673331E1E4E5AD46h, 1A36CCFEF05AF9DFh, 08FCB72A88EC020C8h
.text:000000000400C58 A1 82 02 A6 E5 87 CD E4+                dq 0AFB41EBA3D16771h, 229CE926ABDFEECFh, 709D77A2B3E0FBC0h
.text:000000000400C58 46 AD E5 E4 E1 31 33 67+                dq 0D4827E8F9DB83497h, 102B48575AE4368Eh, 3C32B492AD4ED883h
.text:000000000400C58 DF F9 5A F0 FE CC 36 1A+                dq 27642822234B6E4Ch, 52F30846E9A3BBA9h, 4F53707FC9CC5A72h
.text:000000000400C58 CB 20 C0 8E A8 72 CB BF+                dq 25AC30BBB082C6FEh, 18589736A52A1A5h, 5778ED6C9C8702CDh
.text:000000000400C58 71 67 D1 A3 E8 41 FB 0A+                dq 0FA9166EB90710E00h, 51689656F17A71DEh, 0C2A31F2BCD9812FFh
.text:000000000400C58 CF EE DF A8 26 E9 9C 22+                dq 309E3569C80F2DE8h, 34DE28FAEFD0D2C3h, 6EBE43DECE96C71Dh
.text:000000000400C58 C0 FB E0 B3 A2 77 9D 70+                dq 0CE54F23EB3FF8C98h, 4F0B2641A29A1D9Ch, 2E4420BD6E5178B8Eh
.text:000000000400C58 97 34 B8 9D 8F 7E 82 D4+                dq 0B6119B48845861A4h, 36287197E8C8F151h, 0EAFC1B71DBAD0266h
.text:000000000400C58 BE 36 E4 5A 57 48 2B 10+                dq 0EE348DDA58FF7A03h, 48B4EA4A4C0A0B59h, 0E9104BBBFE93DB8h
.text:000000000400C58 83 D8 4E AD 92 B4 32 3C+                dq 4AFD99760E705A87h, 4735203A8D636911h, 8D2103044F8E6C44h
.text:000000000400C58 4C 6E 4B 23 22 28 64 27+                dq 0CE30FABD11E7DB73h, 66412C0F99611B72h, 6B14D7434EC5EEEEh
.text:000000000400C58 A9 8B A3 E9 46 08 F3 52+                dq 0E2227D5028FE7135h, 0BD4D67CB2AAE1EAAh, 0D3AF8CE52AFBDDB7h
.text:000000000400C58 72 5A CC C9 7F 70 53 4F+                dq 24B8BFFC86999CE5h, 6CB8525C8D47C3C8h, 36387D23D66561F9h
.text:000000000400C58 FE C6 82 B0 BB 30 AC 25+                dq 12914DFDF41727EFh, 480AF64E8AFA03A0h, 5460661174208F0Ah
.text:000000000400C58 A5 A1 52 6A 73 89 85 01+                dq 17D0DC565EDD23C4h, 73C47CDC8066B038h, 67F8F40AEF64FBC6h
.text:000000000400C58 CD 02 87 9C 6C ED 78 57+                dq 8D88ED7B4F72461Dh, 51562D093DF4F5C7h, 08E16E52A0D898F49h
.text:000000000400C58 00 0E 71 90 E8 66 91 FA+                dq 5681803E602C9412h, 600ED9FE5F021977h, 082F308E06707F999h
```

Sample Analysis

- We will use qiling to:
  - emulate this challenge
  - decrypt magic_table
  - cover bytes to asm

- Use IDA Pro Hex-Rays Decompiler to:
  - decompile the verification algorithm

Encrypted → Decrypted → Decompile

```
byte_400C55     db 87h, 50h, 57h        ; DATA XREF: .data:magi
46 21+          dq 21460847BEA10783h, 5A16E746A4712F14h, 0E4CD8
16 5A+          dq 673331E1E4E5AD46h, 1A36CCFEF05AF9DFh, 0BFCB7
CD E4+          dq 0AFB41EBA3D1671h, 229CE926ABDFEECFh, 709D77
33 67+          dq 0D4827E8F9D883497h, 102B48575AE436BEh, 3C32B
36 1A+          dq 276428222234B6E4Ch, 52F30846E9A3BBA9h, 4F5370
CB BF+          dq 25AC30BBB082C6FEh, 18589736A52A1A5h, 5778ED6
FB 0A+          dq 0FA9166EB90710E00h, 51689656F17A71DEh, 0C2A3
9C 22+          dq 309E3569C80F2DE8h, 34DE28FAEFD0D2C3h, 6EBE43
9D 70+          dq 0CE54F23EB3FF8C98h, 4F0B2641A29A1D9Ch, 2E442
82 D4+          dq 0B6119B48845861A4h, 36287197E8C8F151h, 0EAFC
2B 10+          dq 0EE348DDA5BFF7A03h, 48B4EA4A4C0A0B59h, 0E910
32 3C+          dq 4AFD99760E705AB7h, 47352D3A8D636911h, 8D2103
64 27+          dq 0CE30FABD11E7DB73h, 66412C0F99611B72h, 6B14D
F3 52+          dq 0E2227D5028FE7135h, 0BD4D67CB2AAE1EAAh, 0D3A
53 4F+          dq 24BBBFFC86999CE5h, 6CB8525C8D47C3C8h, 363B7D
AC 25+          dq 12914DFDF41727EFh, 480AF64E8AFA03A0h, 546066
85 01+          dq 17D0DC565EDD23C4h, 73C47CDC8066B03Bh, 67F8F4
78 57+          dq 8D88ED7B4F72461Dh, 5156D093DF4F5C7h, 0BE16E
91 FA+          dq 56B1803E602C9412h, 600ED9FE5F021977h, 0B2F30
68 51+          dq 158349D548DAC94Fh, 18C333FB0C6BE8A6h, 0CEF39
A3 C2+          dq 0A8A7605B2DB998ADh, 68FE2EAA2BD42F88h, 0FAE1
9E 30+          dq 87C3AF489E97732Ch, 9105793D3F62A8F1h, 0A9ED0
DE 34+          dq 0FCC6C80A76ADAE0Bh, 0E088546B20A50859h, 0E4F
BE 6E+          dq 52742720FDB93280h, 0C309AFB3D71AB9EBh, 2C3F6
54 CE+          dq 7DA7274301FF612Bh, 0CBA3F22455335539h, 8F0AC
0B 4F+          dq 99B9847A292D4A2Ch, 0C3C2939831AD4756h, 0B62E
44 2E+          dq 0D0ED590E5A4C43C3h, 0C939E780645BFB3Eh, 0B0B
11 B6+          dq 9B0E8CFD5323D736h, 0B0F7DE0A7F561D18h, 0D3BE
28 36+          dq 427C8469E6675A3Ah, 78FF332BF2A2F5C7h, 4B3B7D
FC EA+          dq 0BAC446F96BC591EAh, 0BA627DF99F75648Fh, 8F33
```

```
400C55                  sub_400C55      proc near               ;
400C55
400C55                  var_58          = qword ptr -58h
400C55                  var_4C          = dword ptr -4Ch
400C55                  var_48          = qword ptr -48h
400C55                  var_38          = qword ptr -38h
400C55                  var_30          = qword ptr -30h
400C55                  var_28          = qword ptr -28h
400C55                  var_20          = qword ptr -20h
400C55                  var_18          = qword ptr -18h
400C55                  var_10          = qword ptr -10h
400C55                  var_6           = byte ptr -6
400C55                  var_5           = byte ptr -5
400C55                  var_4           = dword ptr -4
400C55
400C55 55                               push    rbp
400C56 48 89 E5                         mov     rbp, rsp
400C59 48 89 7D B8                      mov     [rbp+var_48], rdi
400C5D 89 75 B4                         mov     [rbp+var_4C], esi
400C60 48 89 55 A8                      mov     [rbp+var_58], rdx
400C64 C7 45 FC 00 00 00 00             mov     [rbp+var_4], 0
400C6B E9 19 01 00 00                   jmp     loc_400D89
400C70                          ; ---------------------------------------------------------------------------
400C70
400C70                  loc_400C70:                             ;
400C70 8B 45 FC                         mov     eax, [rbp+var_4]
400C73 48 8D 14 C5 00 00 00 00          lea     rdx, ds:0[rax*8]
400C7B 48 8B 45 A8                      mov     rax, [rbp+var_58]
400C7F 48 01 D0                         add     rax, rdx
400C82 48 8B 00                         mov     rax, [rax]
400C85 48 85 C0                         test    rax, rax
400C88 75 7A                            jnz     short loc_400D04
400C8A 8B 45 FC                         mov     eax, [rbp+var_4]
400C8D 48 8D 14 C5 00 00 00 00          lea     rdx, ds:0[rax*8]
```

```
__int64 __fastcall sub_400C55(__int64 a1, unsigned int a2, __int64 a3)
{
  __int64 v4; // [rsp+20h] [rbp-38h]
  __int64 v5; // [rsp+28h] [rbp-30h]
  __int64 v6; // [rsp+30h] [rbp-28h]
  __int64 v7; // [rsp+38h] [rbp-20h]
  __int64 v8; // [rsp+40h] [rbp-18h]
  __int64 v9; // [rsp+48h] [rbp-10h]
  char v10; // [rsp+52h] [rbp-6h]
  char v11; // [rsp+53h] [rbp-5h]
  unsigned int i; // [rsp+54h] [rbp-4h]

  for ( i = 0; i < a2; ++i )
  {
    if ( *(8LL * i + a3) )
    {
      v11 = *(i + a1);
      v6 = 0LL;
      v5 = 1LL;
      v4 = 0LL;
      while ( v11 )
      {
        v6 = v5 + v4;
        v4 = v5;
        v5 = v6;
        --v11;
      }
      if ( v6 != *(8LL * i + a3) )
        return 0LL;
    }
    else
    {
      v10 = *(i + a1);
      v9 = 0LL;
```

Demo 1

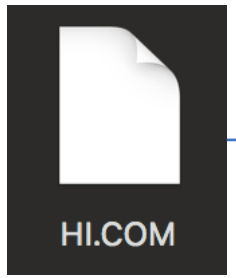
Agenda

- Diving into Qiling Framework – 10min
  - Show how real mode emulation is implemented.
  - Learn the internal design of the Qiling Framework.
  - A good start if you would like become a contributor.
- Solve a CTF Challenge with MBR emulation. – 10min + 10min
  - How Qiling API helps our analysis.

Qiling Internals

Basics

- Core problem: How Qiling emulates a real mode binary? Two Layers.
  - Loader: Parse binary and load it into memory.
  - OS: Implement interrupts.
- While the instrumentation is provided as an API, it is also heavily used internally to implement the OS layer.




```
seg000:0100 ; Attributes: noreturn
seg000:0100
seg000:0100                     public start
seg000:0100 start               proc near
seg000:0100                     mov     ah, 9
seg000:0102                     mov     dx, 10Dh
seg000:0105                     int     21h
seg000:0105
seg000:0107                     mov     ax, 4C00h
seg000:010A                     int     21h
seg000:010A start               endp
```

Emulation starts here.

Trap into Qiling.

Loader Layer

- Target binary: MBR file && COM file.
  - DOS EXE support is still WIP.
- Pretty similar, memory image without any header.
  - Setup registers, memory map and write the file into memory.
  - But disk image should be mounted for MBR file.
- qiling/loader/dos.py

OS Layer

- The place where we implement traditional interrupts.
- Example: INT 13h, ah=42h, read disk sectors.
  - Implemented with fs_mapper API.
  - Map any object which implements FsMappedObject interface to an emulated device/path.
  - QlDisk is inherited from FsMappedObject with CHS and LBA support.
  - Note that we mount the MBR file itself in loader.
- os/dos/dos.py
- os/mapper.py
- os/disk.py

Loader

```
if not self.ql.os.fs_mapper.has_mapping(0x80):
    self.ql.os.fs_mapper.add_fs_mapping(0x80, QlDisk(path, 0x80))
```

OS

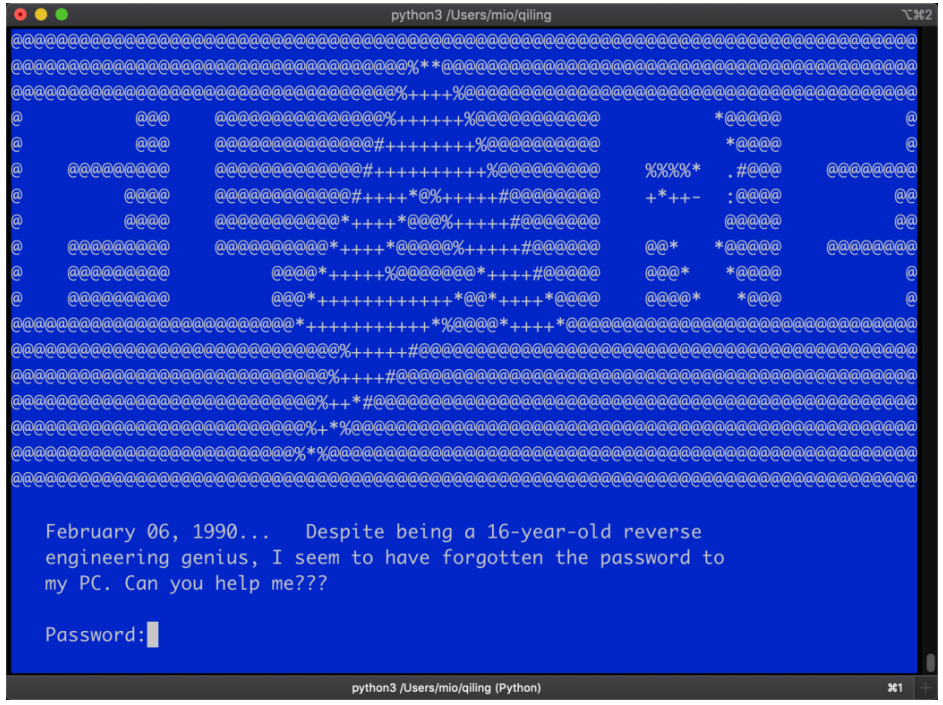
```
disk = self.ql.os.fs_mapper.open(idx, None)
content = disk.read_sectors(lba, cnt)
```

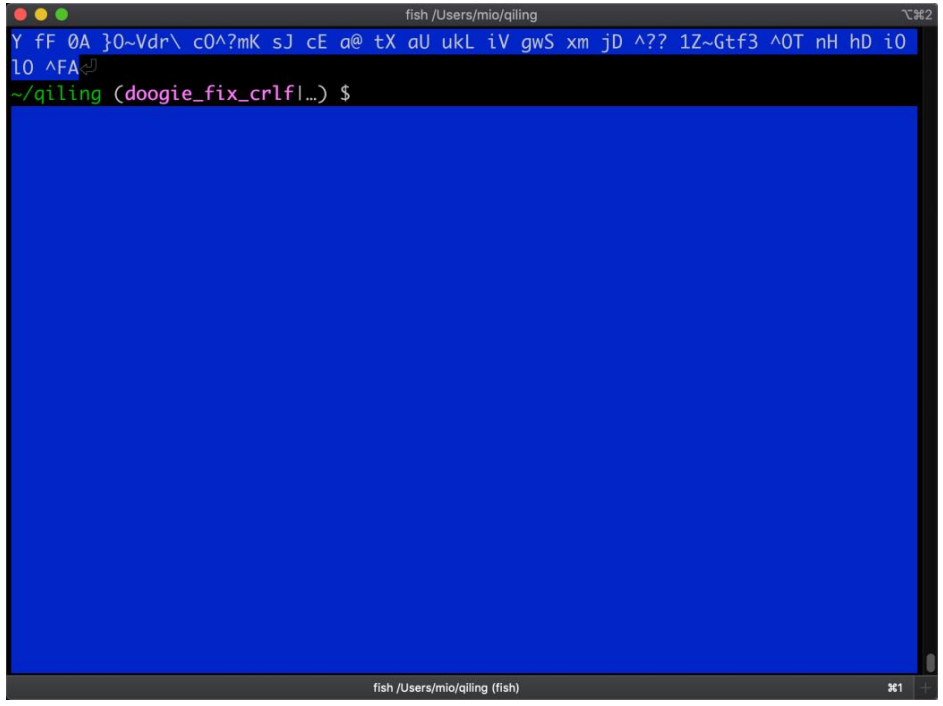
Solve a CTF Challenge

Sample Analysis

- Sample:
  - Flare-On 5 (2018) Challenge 8 – doogie
  - examples/rootfs/8086/doogie/doogie.bin
- MBR file
- Quick look by qltool.
  - python3 qltool run -f examples/rootfs/8086/doogie/doogie.bin --rootfs examples/rootfs/8086/ --console False
- Try some inputs, but only get gibberish.
- Tips: Feburary 06, 1990.

```
~/q/e/r/8/doogie (doogie|…) $ file doogie.bin
doogie.bin: DOS/MBR boot sector; partition 1 : ID=0x7, activ
e, start-CHS (0x0,32,33), end-CHS (0x3ff,254,63), startsecto
r 2048, 41938944 sectors
~/q/e/r/8/doogie (doogie|…) $
```

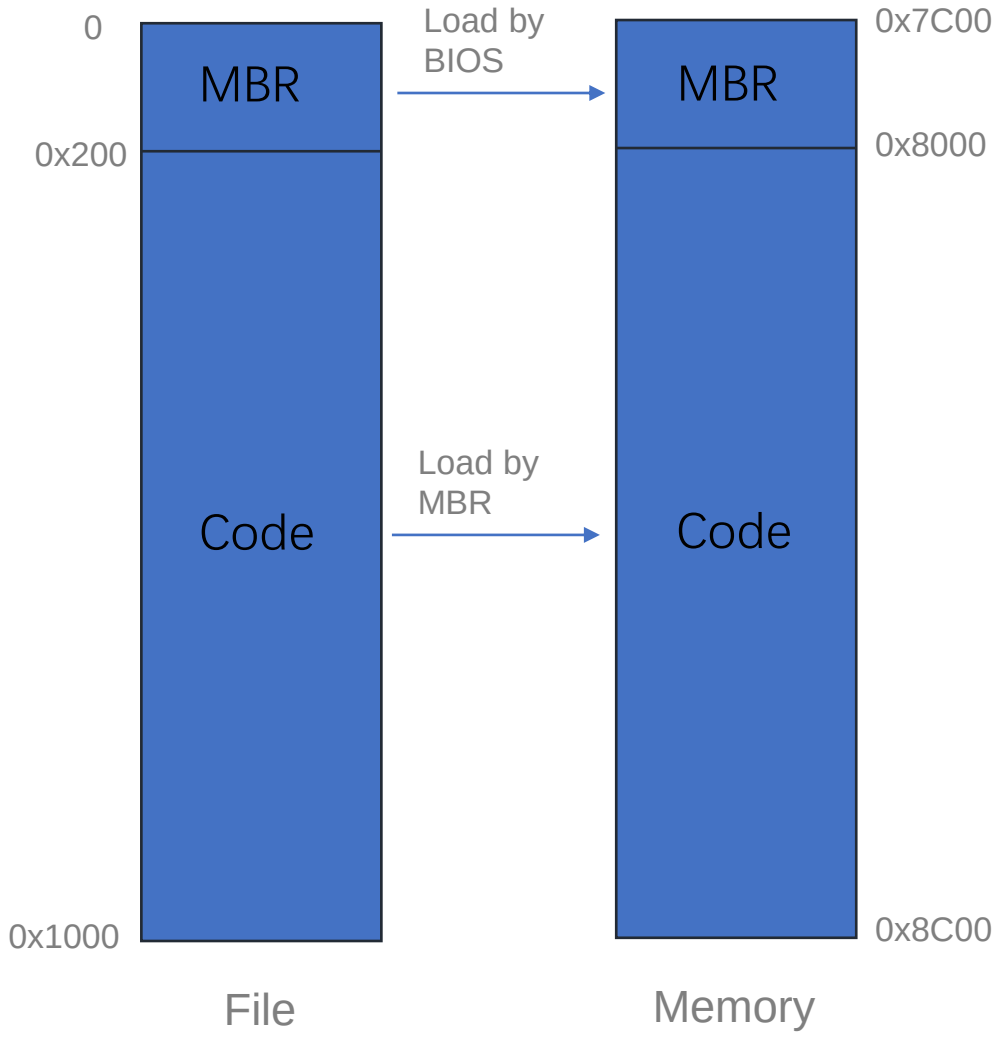




Sample Analysis: First Stage

- Like most operating systems, the program runs in two stage
  - MBR is responsible for loading code from file into 0x8000
  - Then it jumps to 0x8000 and execute the rest code

```
loc_7C00:                               ; DATA XREF: seg000:7C09↓o
                cli
                xor     ax, ax
                mov     ds, ax
                mov     ss, ax
                mov     es, ax
                lea     sp, loc_7C00
                sti
                mov     eax, 20h ; ' '
                mov     ds:byte_7C45, dl
                mov     ebx, 1
                mov     cx, 8000h
                call    sub_7C27
                jmp     near ptr byte_7C4C+3B4h ; jump to 0x8000

; =============== S U B R O U T I N E =======================================


sub_7C27        proc near               ; CODE XREF: seg000:7C21↑p
                xor     eax, eax
                mov     di, sp
                push    eax
                push    ebx             ; sectors offset = 1
                push    es
                push    (offset byte_7C4C+3B4h) ; destination address = 0x8000
                push    7               ; sectors count = 7
                push    10h
                mov     si, sp
                mov     dl, ds:byte_7C45
                mov     ah, 42h ; 'B'
                int     13h             ; DISK - IBM/MS Extension - EXTENDED READ
                mov     sp, di
                retn
sub_7C27        endp
```



Sample Analysis: Second Stage

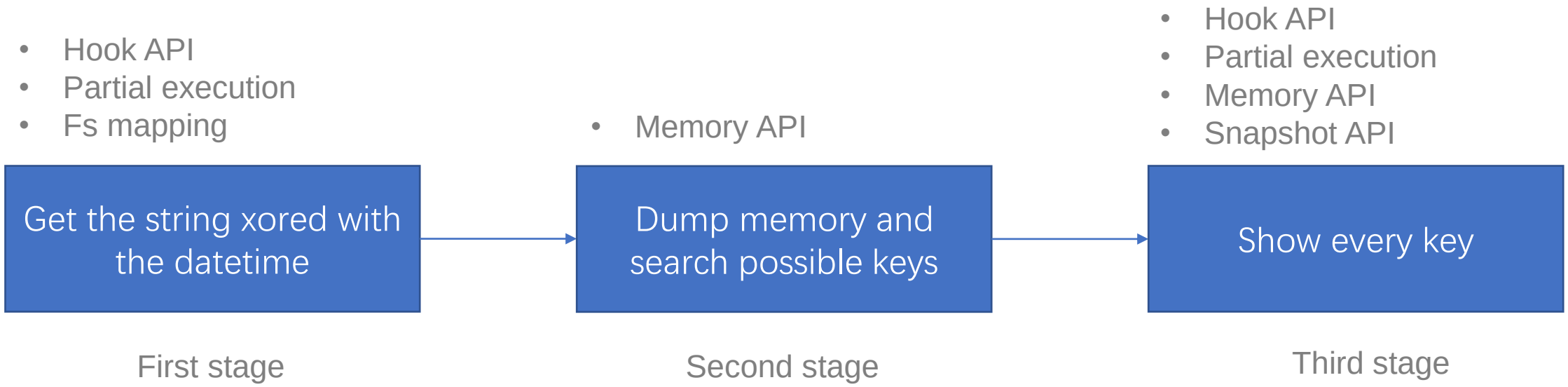
- The logic which starts from 0x8000 is pretty clear
  - Firstly, it gets current datetime by INT 1a and then xors the string at 0x8809 with that datetime
  - Then, it reads user input and xors the same string at 0x8809 with the input
  - Lastly, it initializes the screen and print the ascii art

```
seg000          segment byte public 'CODE' use16
                assume cs:seg000
                ;org 8000h
                assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
                call    sub_805B
                push    ds:word_87F2
                call    get_date_and_write_to_87EE
                push    4
                push    offset date_87EE
                call    xor8809
                add     sp, 4
                push    87F4h
                call    read_input
                add     sp, 4
                push    ax
                push    87F4h
                call    xor8809
                add     sp, 4
                call    initialize_screen
                push    0
                push    8809h
                call    print_ascii_art
                add     sp, 4
                mov     cx, 2607h
                mov     ah, 1
                int     10h             ; - VIDEO - SET CURSOR CHARACTERISTICS
                                        ; CH bits 0-4 = start line for cursor in character cell
                                        ; bits 5-6 = blink attribute
                                        ; CL bits 0-4 = end line for cursor in character cell

loc_803D:                               ; CODE XREF: seg000:803E↓j
                hlt
; ---------------------------------------------------------------------------
                jmp     short loc_803D
```

Sample Analysis: Qiling's work

- What Qiling can do to speed up our analysis and find the key
  - Emulate the binary
  - Hook interrupts like INT 1a to give the program a specific time
  - Dynamic memory read/write API
  - Automatically test every key with partial execution and snapshot API
- The crack process can be divided into three stages
  - The key of this challenge is not unique, so we have to show every one

```
if __name__ == "__main__":
    ql = first_stage()
    # resume terminal
    curses.endwin()
    keys = second_stage(ql)
    for key in keys:
        print(f"Possible key: {key}")
    # The key of this challenge is not unique. The real
    # result depends on the last ascii art.
    print("Going to try every key.")
    time.sleep(3)
    third_stage(keys)
    # resume terminal
    curses.endwin()
```



Crack: First stage

- The quick look before suggests that we have to set the datetime to Feburary 06, 1990
  - It's extremely easy to achieve that with **ql.set_api**
- The program also read disks directly
  - Use **ql.fs_mapper** API to emulate a disk
- Execute the program until 0x8018
  - At this time, the string at 0x8809 has been xored with date
- Dump memory at 0x8809 for the next stage
  - Use **ql.mem.read** API to dump memory

```
# In this stage, we get the encrypted data which xored with the specific date.
def first_stage():
    ql = Qiling(["rootfs/8086/doogie/doogie.bin"],
                "rootfs/8086",
                console=False,
                log_dir=".")
    ql.add_fs_mapper(0x80, QlDisk("rootfs/8086/doogie/doogie.bin", 0x80))
    # Doogie suggests that the datetime should be 1990-02-06.
    ql.set_api((0x1a, 4), set_required_datetime, QL_INTERCEPT.EXIT)
    # A workaround to stop the program.
    hk = ql.hook_code(stop, begin=0x8018, end=0x8018)
    ql.run()
    ql.hook_del(hk)
    return read_until_zero(ql, 0x8809)
```

```
seg000:8000    call    sub_805B
seg000:8003    push    ds:word_87F2
seg000:8007    call    get_date_and_write_to_87EE
seg000:800A    push    4
seg000:800C    push    offset date_87EE
seg000:800F    call    xor8809
seg000:8012    add     sp, 4
seg000:8015    push    87F4h
seg000:8018    call    read_input
```

February 06, 1990... Despite being a 16-year-old reverse engineering genius, I seem to have forgotten the password to my PC. Can you help me???

```
def set_required_datetime(ql: Qiling):
    ql.nprint("Setting Feburary 06, 1990")
    ql.reg.ch = BIN2BCD(19)
    ql.reg.cl = BIN2BCD(1990%100)
    ql.reg.dh = BIN2BCD(2)
    ql.reg.dl = BIN2BCD(6)
```

Crack: Second stage

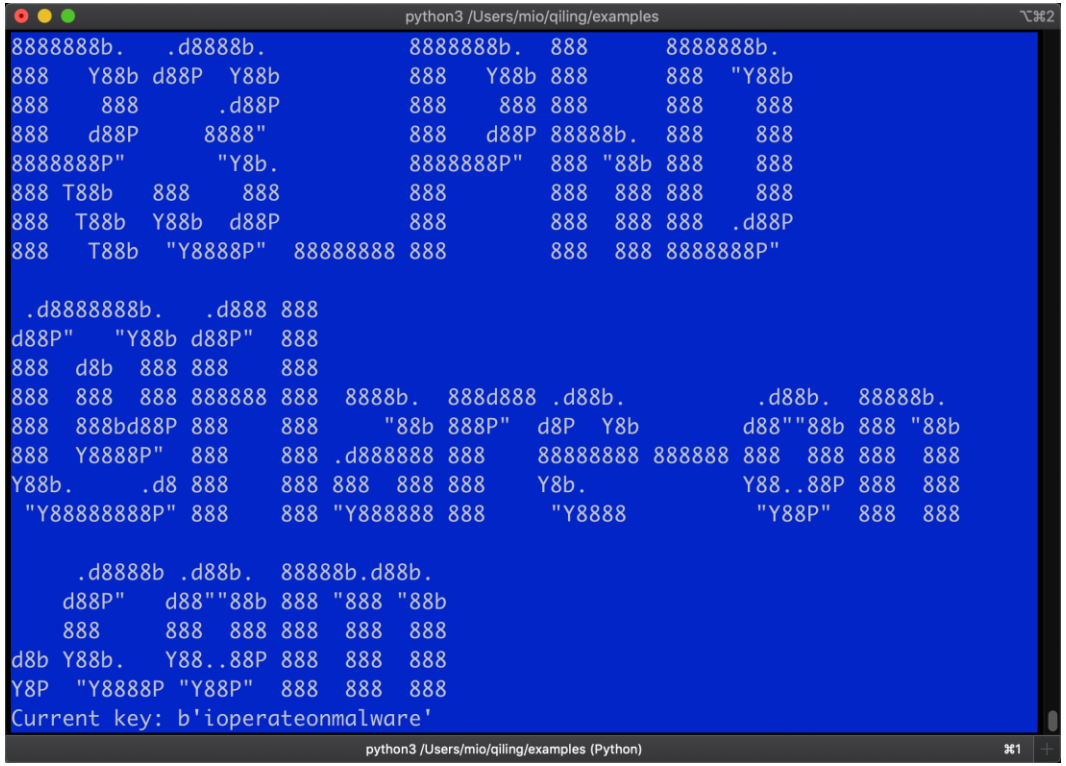
- Dump memory at 0x8809 for the next stage
  - Use ‘ql.mem.read’ API to dump memory
- Utilize some algorithms[1] to guess key size and search possible keys with the assumption that all the result should be printable ascii since it is likely an ascii art

```python
# In this stage, we crack the encrypted buffer.
def second_stage(ql: Qiling):
    data = bytes(read_until_zero(ql, 0x8809))
    key_size = guess_key_size(data) # Should be 17
    seqs = []
    for i in range(key_size):
        seq = b""
        j = i
        while j < len(data):
            seq += bytes([data[j]])
            j += key_size
        seqs.append(seq)
    seqs_keys = cal_count_for_seqs(seqs)
    keys = search_possible_key(seqs, seqs_keys)
    return keys


def read_until_zero(ql: Qiling, addr):
    buf = b""
    ch = -1
    while ch != 0:
        ch = ql.mem.read(addr, 1)[0]
        buf += pack("B", ch)
        addr += 1
    return buf
```

[1]: https://trustedsignal.blogspot.com/2015/06/xord-play-normalized-hamming-distance.html

Crack: Third stage

- Execute until 0x8018 and take a snapshot
- Fill in the key in the memory and Skip reading user input
- After completing one round, resume the program to previous snapshot and try next key
- One of the correct keys is: ‘ioperateonmalware’




```python
def show_once(ql: Qiling, key):
    klen = len(key)
    ql.reg.ax = klen
    ql.mem.write(0x87F4, key)
    # Partial exectution to skip input reading
    ql.run(begin=0x801B, end=0x803d)
    echo_key(ql, key)
    time.sleep(3)

# In this stage, we show every key.
def third_stage(keys):
    # To setup terminal again, we have to restart the whole program.
    ql = Qiling(["rootfs/8086/doogie/doogie.bin"],
                "rootfs/8086",
                console=False,
                log_dir=".")
    ql.add_fs_mapper(0x80, QlDisk("rootfs/8086/doogie/doogie.bin", 0x80))
    ql.set_api((0x1a, 4), set_required_datetime, QL_INTERCEPT.EXIT)
    hk = ql.hook_code(stop, begin=0x8018, end=0x8018)
    ql.run()
    ql.hook_del(hk)
    # Snapshot API.
    ctx = ql.save()
    for key in keys:
        show_once(ql, key)
        ql.restore(ctx)
```

Planning

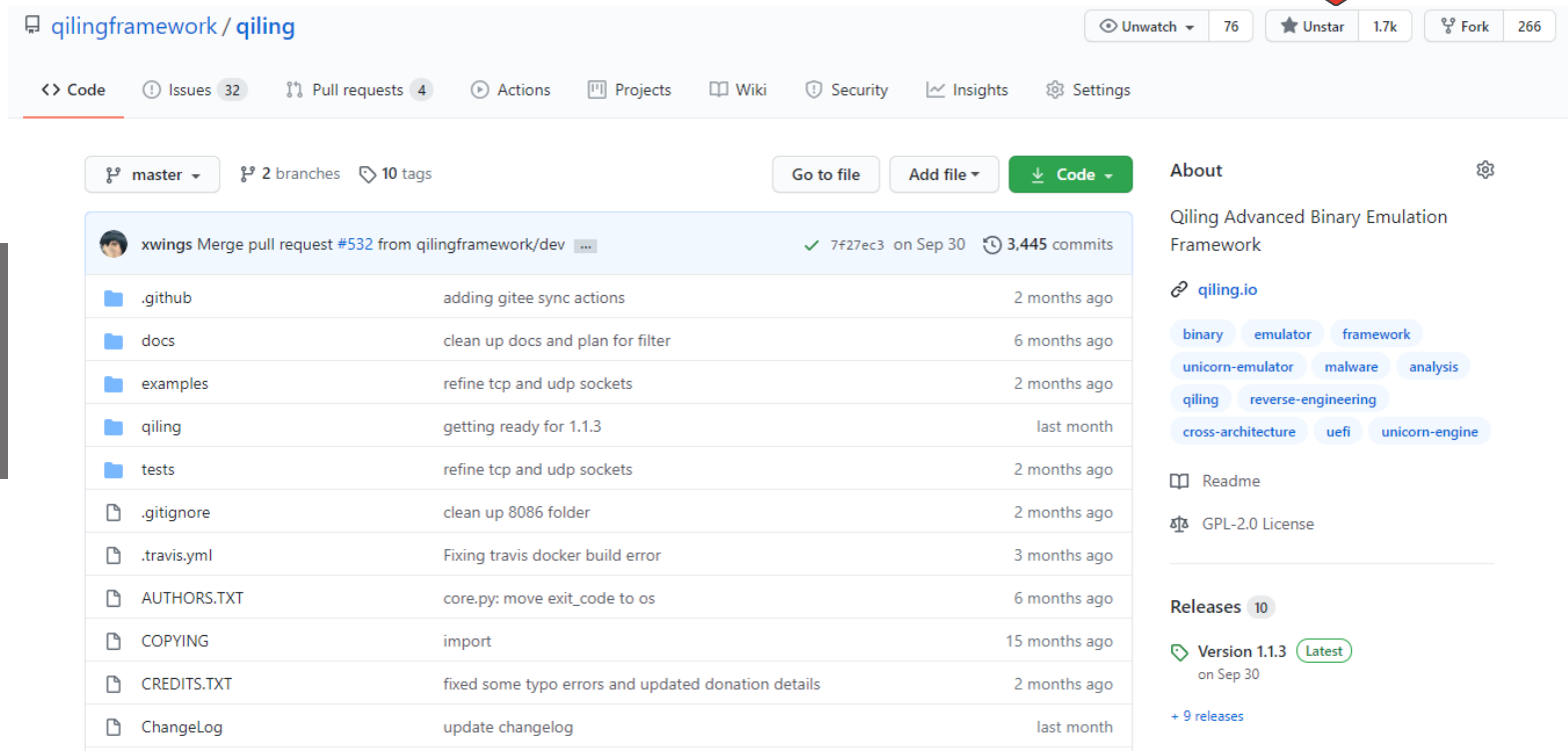
Roadmap

- Force Unicorn Engine sync with QEMU 5
  - More architectures, more CPU instructions set
- Android Java bytecode layer instrumentation
- IOS emulation support
- More robust Windows emulation
  - Introduce wine && Cygwin or something
- Smart Contract emulation (EVM, WASM)
- MCU emulation

Everything Else

- About Qiling Framework
  - https://qiling.io
  - https://github.com/qilingframework/qiling
  - https://docs.qiling.io
  - @qiling_io

Star us



Questions