Qiling Framework: Introduction

November, 2020

twitter: @qiling_io https://qiling.io

Ziqiao Kong

First of All…

- About Qiling Framework
  - https://qiling.io
  - https://github.com/qilingframework/qiling
  - https://docs.qiling.io
  - http://t.me/qilingframework
  - Weibo: @qiling_io
  - QQ: 486812017

Star us

麒麟框架交流群

群号：486812017

qilingframework / qiling

Unwatch 76 | Unstar 1.7k | Fork 266

Code | Issues 32 | Pull requests 4 | Actions | Projects | Wiki | Security | Insights | Settings

master | 2 branches | 10 tags | Go to file | Add file | Code

xwings Merge pull request #532 from qilingframework/dev ... 7f27ec3 on Sep 30 3,445 commits

.github	adding gitee sync actions	2 months ago
docs	clean up docs and plan for filter	6 months ago
examples	refine tcp and udp sockets	2 months ago
qiling	getting ready for 1.1.3	last month
tests	refine tcp and udp sockets	2 months ago
.gitignore	clean up 8086 folder	2 months ago
.travis.yml	Fixing travis docker build error	3 months ago
AUTHORS.TXT	core.py: move exit_code to os	6 months ago
COPYING	import	15 months ago
CREDITS.TXT	fixed some typo errors and updated donation details	2 months ago
ChangeLog	update changelog	last month

About

Qiling Advanced Binary Emulation Framework

qiling.io

binary | emulator | framework | unicorn-emulator | malware | analysis | qiling | reverse-engineering | cross-architecture | uefi | unicorn-engine

Readme

GPL-2.0 License

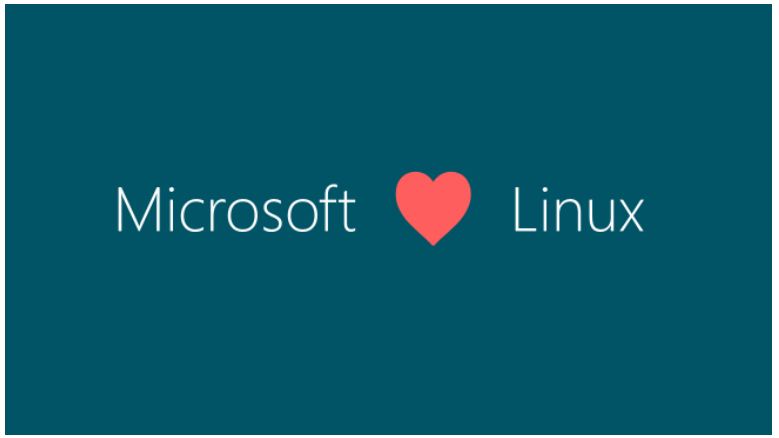
Releases 10

Version 1.1.3 Latest on Sep 30

+ 9 releases

Demo Setup

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



- Required OS
  - Ubuntu 18.04 / 20.04
  - WSL2

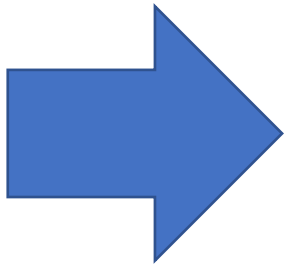
- Install Qiling Framework
  - sudo apt-get update
  - sudo apt-get upgrade
  - sudo apt install python3-pip git cmake build-essential libtool-bin python3-dev automake flex bison libglib2.0-dev libpixman-1-dev clang python3-setuptools
  - git clone https://github.com/qilingframework/qiling.git
  - pip3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

OLLVM De-flatten

De-flatten: Introduction

- Reserve all original basic blocks and transform control flow.
  - Utilize a big while-switch loop to hide real control flow.
  - The example here introduces a variable b.
- With more complex code, the number of cases increases dramatically.

```
int main(int argc, char** argv) {
  int a = atoi(argv[1]);
  if(a == 0)
    return 1;
  else
    return 10;
  return 0;
}
```

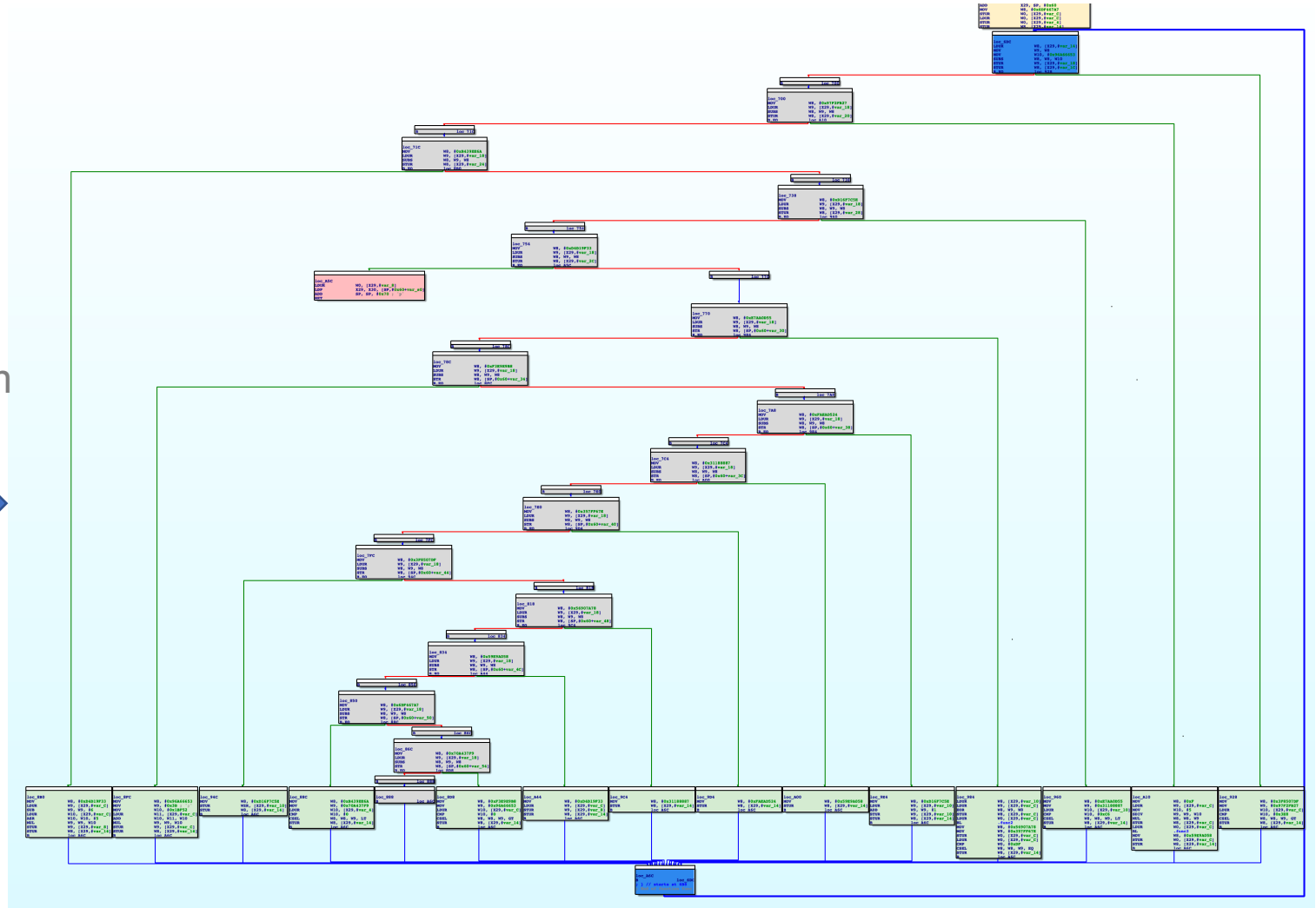


```
int main(int argc, char** argv) {
  int a = atoi(argv[1]);
  int b = 0;
  while(1) {
    switch(b) {
      case 0:
        if(a == 0)
          b = 1;
        else
          b = 2;
        break;
      case 1:
        return 1;
      case 2:
        return 10;
      default:
        break;
    }
  }
  return 0;
}
```

De-flatten: Sample

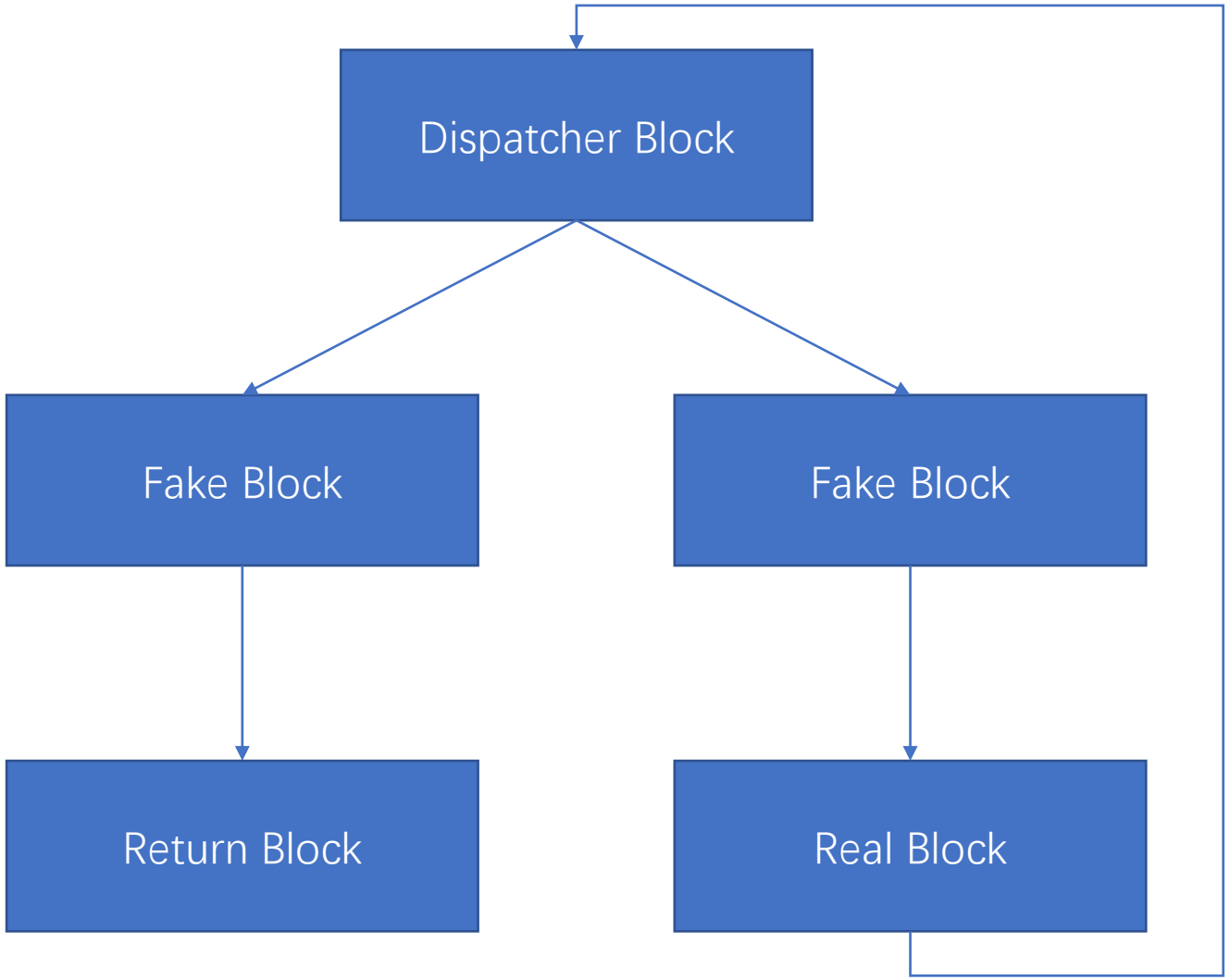
```
int func3(int v){
        return v+v/2+v*v;
}

int func2(int v){
        return v*v*4;
}

int func(int v){
        if(v<0)
                return (v - 6)*(v>>5);
        if(v>0)
                v = 59 * (v + 114514);
        if(v > 1000){
                for(int i =0;i<198;i++){
                        v ^= i;
                        v = func2(v);
                        if(v == 223)
                                break;
                }
        }else{
                v = 15 * (v / 5);
                v = func3(v);
        }
        return v;
}

int main(){
        func(1);
}
```

NDK + ollvm



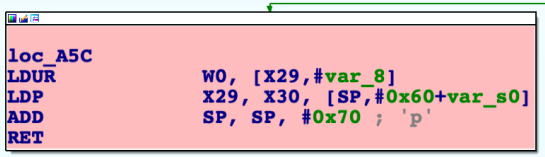
De-flatten: Identification

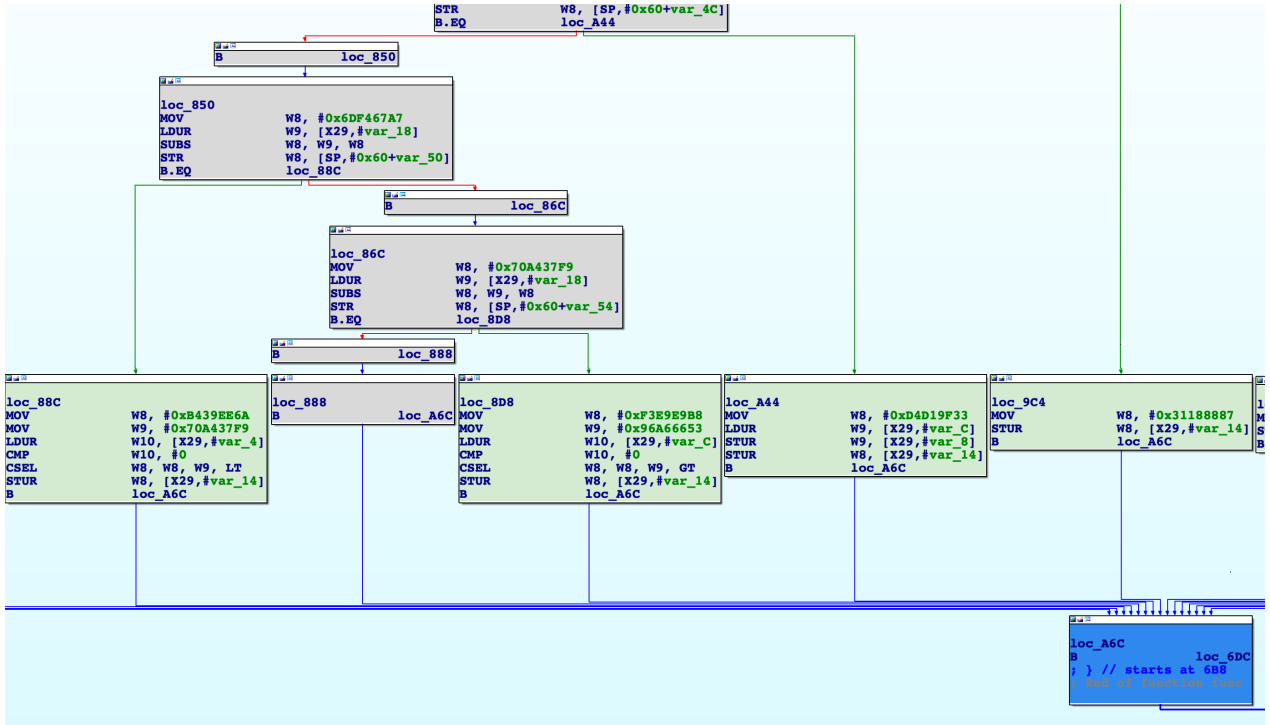
- Four types of blocks
- Real Blocks
  - Only contain original logic
  - May contain zero or only one conditional statement.
- Fake Blocks
  - Only responsible for switch-case implementation.
- Return Blocks
  - Return the function.
- Dispatcher Blocks
  - Equivalent to switch statement.
  - Decide the following control flow.
  - The core of the identification.



De-flatten: Identification

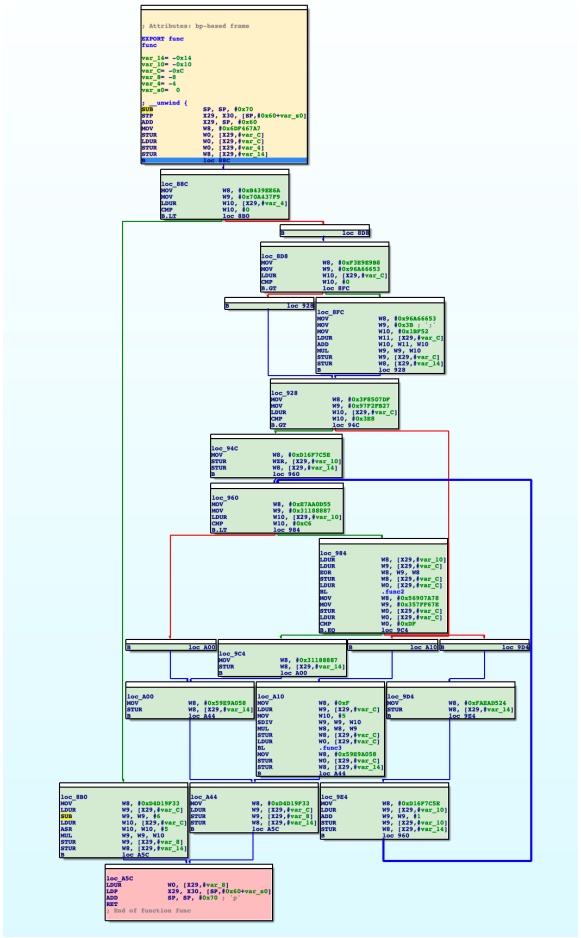
- The successor of all real blocks is the dispatcher block.
  - Locate dispatcher blocks by reference counts.
- The blocks with more than one instructions whose successor is the dispatcher can be identified as real blocks.
- The blocks which has any return statement like retn is considered as return Blocks.
- All other blocks are fake blocks.
- We use different colors to identify the blocks.
  - Users can also adjust the result manually.





De-flatten: Qiling Emulation

- Restore original control flow.
- Partial Execution && Snapshot API
  - Save snapshot and start from any real block.
  - Execute each branch for conditional instruction.
  - Stop when meeting another real/return block.
  - Record control flow and restore snapshot.
  - Iterate all real blocks.
- Patch with Keystone
  - Only real blocks and return blocks are reserved.



```
__int64 __fastcall func(int a1)
{
  int i; // [xsp+50h] [xbp-10h]
  int v3; // [xsp+54h] [xbp-Ch]
  unsigned int v4; // [xsp+58h] [xbp-8h]

  v3 = a1;
  if ( a1 >= 0 )
  {
    if ( a1 > 0 )
      v3 = 59 * (a1 + 114514);
    if ( v3 > 1000 )
    {
      for ( i = 0; i < 198; ++i )
      {
        v3 = func2(v3 ^ (unsigned int)i);
        if ( v3 == 223 )
          break;
      }
    }
    else
    {
      v3 = func3((unsigned int)(15 * (v3 / 5)));
    }
    v4 = v3;
  }
  else
  {
    v4 = (a1 - 6) * (a1 >> 5);
  }
  return v4;
}
```

De-flatten: Intermediate Representation

- Some implementation details
  - Conditional instruction differs.
  - cmov, csel, ittq…
  - External function calls.
- Solution：IR
  - IDA 7.1 microcode
  - Match microcode pattern and identify conditional jumps to avoid hard-code instructions.
  - Identify external calls with microcode and skip such calls.

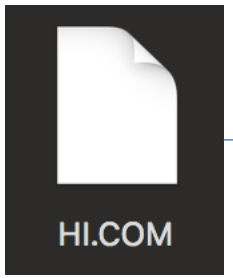
```
.text:000000000000088C loc_88C                                 ; CODE XREF: func+1AC↑j
.text:000000000000088C                 MOV     W8, #0xB439EE6A
.text:0000000000000894                 MOV     W9, #0x70A437F9
.text:000000000000089C                 LDUR    W10, [X29,#var_4]
.text:00000000000008A0                 CMP     W10, #0
.text:00000000000008A4                 CSEL    W8, W8, W9, LT
.text:00000000000008A8                 STUR    W8, [X29,#var_14]
.text:00000000000008AC                 B       loc_A6C
```

```
====================0x86c==================== id=16 0x8a4
0x8a4:   jge      %var_4.4  {1}          ,                    #0.4
>>>>>>>>>>>>>>>>>>>0x884<<<<<<<<<<<<<<<<<<<
====================0x8a4==================== id=17 0x8a4
0x8a4:   mov        #0xB439EE6A.4                ,           w8.4
0x8a4:   goto     @19
>>>>>>>>>>>>>>>>>>>0x8a8<<<<<<<<<<<<<<<<<<<
====================0x8a4==================== id=18 0x8a8
0x8a4:   mov        #0x70A437F9.4                ,           w8.4
```

MBR Emulation

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)
```

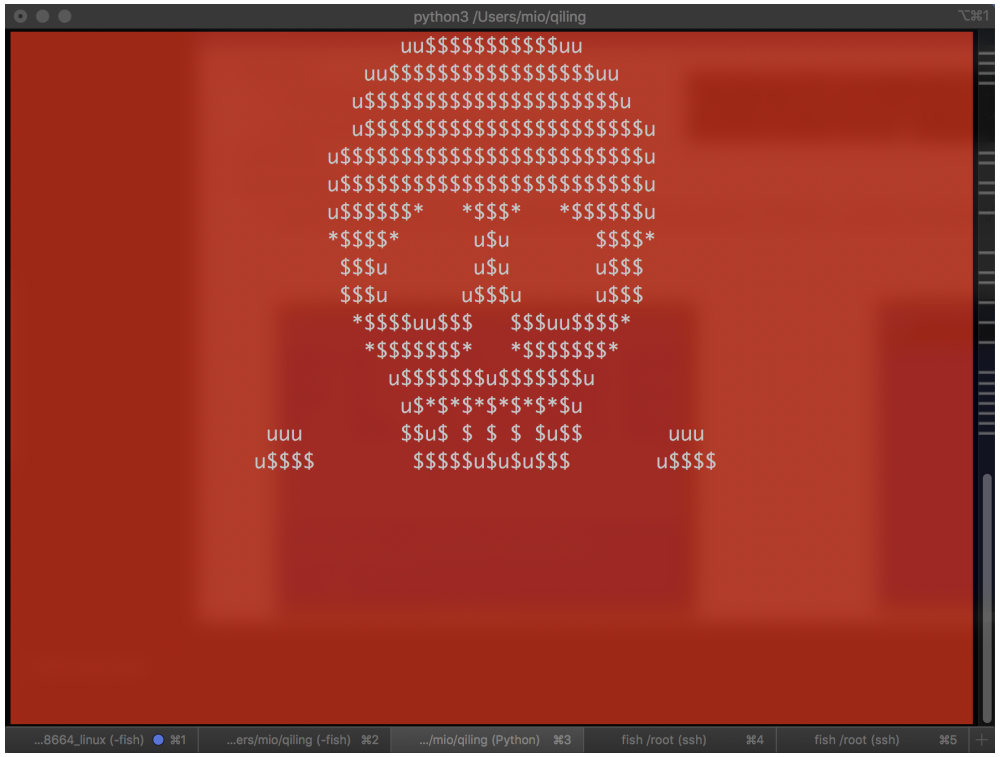
Petya

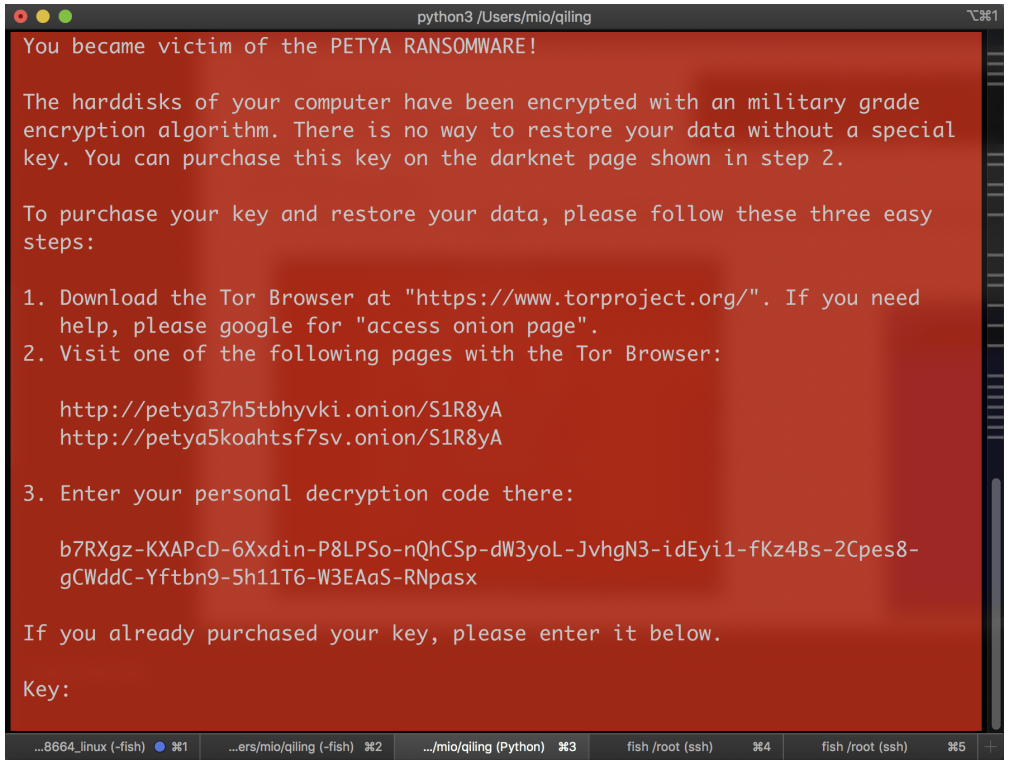
examples/rootfs/8086/petya/out_1M.raw

Sample Analysis

- Sample:
  - Petya/NotPetya
  - examples/rootfs/8086/petya/out_1M.raw
- MBR file
- Quick look by qltool.
  - python3 qltool run -f examples/rootfs/8086/petya/out_1M.raw --rootfs examples/rootfs/8086/ --console False
- Write a script.

```
~/q/e/r/8/petya (dev|✔) $ file out_1M.raw
out_1M.raw: DOS/MBR boot sector; partition 1 : ID=0x7, active, start-CHS (0x0,32,33), end-CHS (0x3ff,254,63), startsector 2048, 41938944 sectors
~/q/e/r/8/petya (dev|✔) $
```





HFS MBR

examples/rootfs/8086/dos/hfs.img

Reading Material: https://gu.mk/mbr
Writeup: https://gu.mk/hfs

Doogie

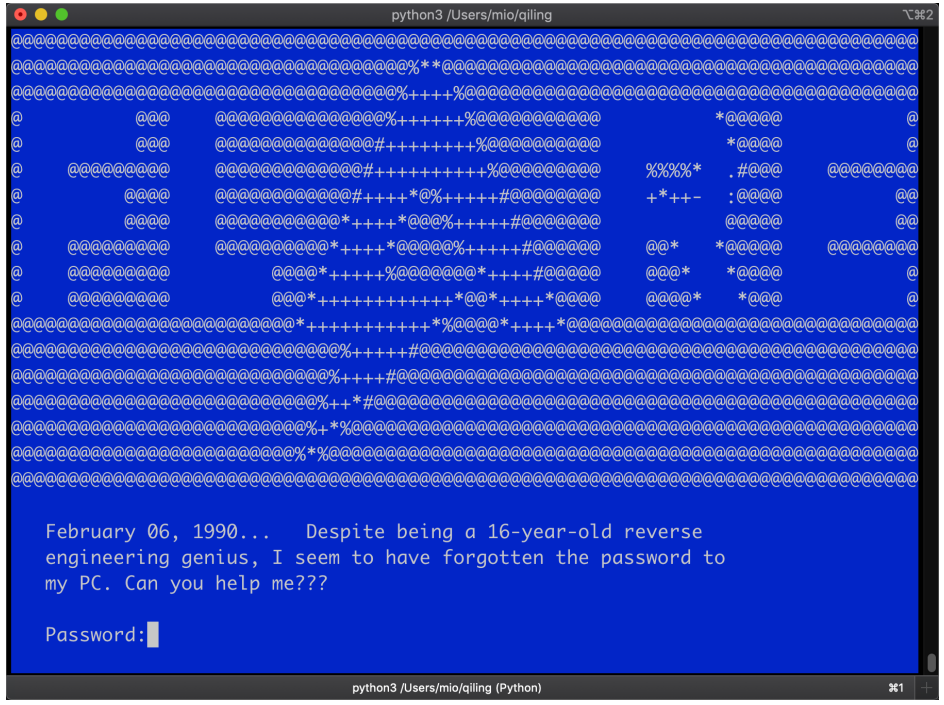
examples/rootfs/8086/doogie/doogie.bin

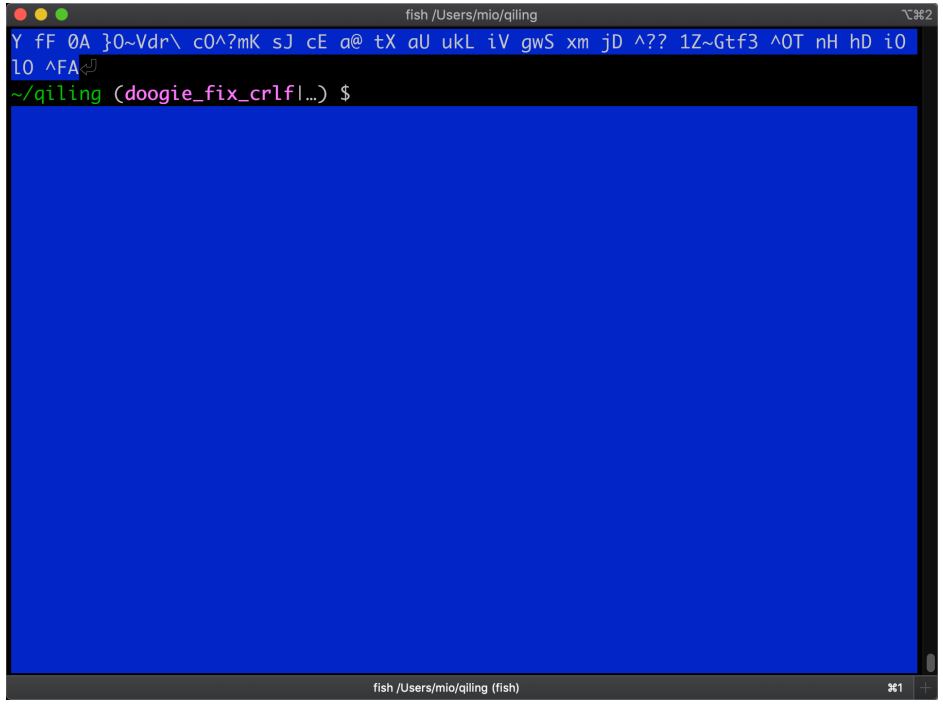
Official Writeup: https://gu.mk/doogie

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|…) $
```

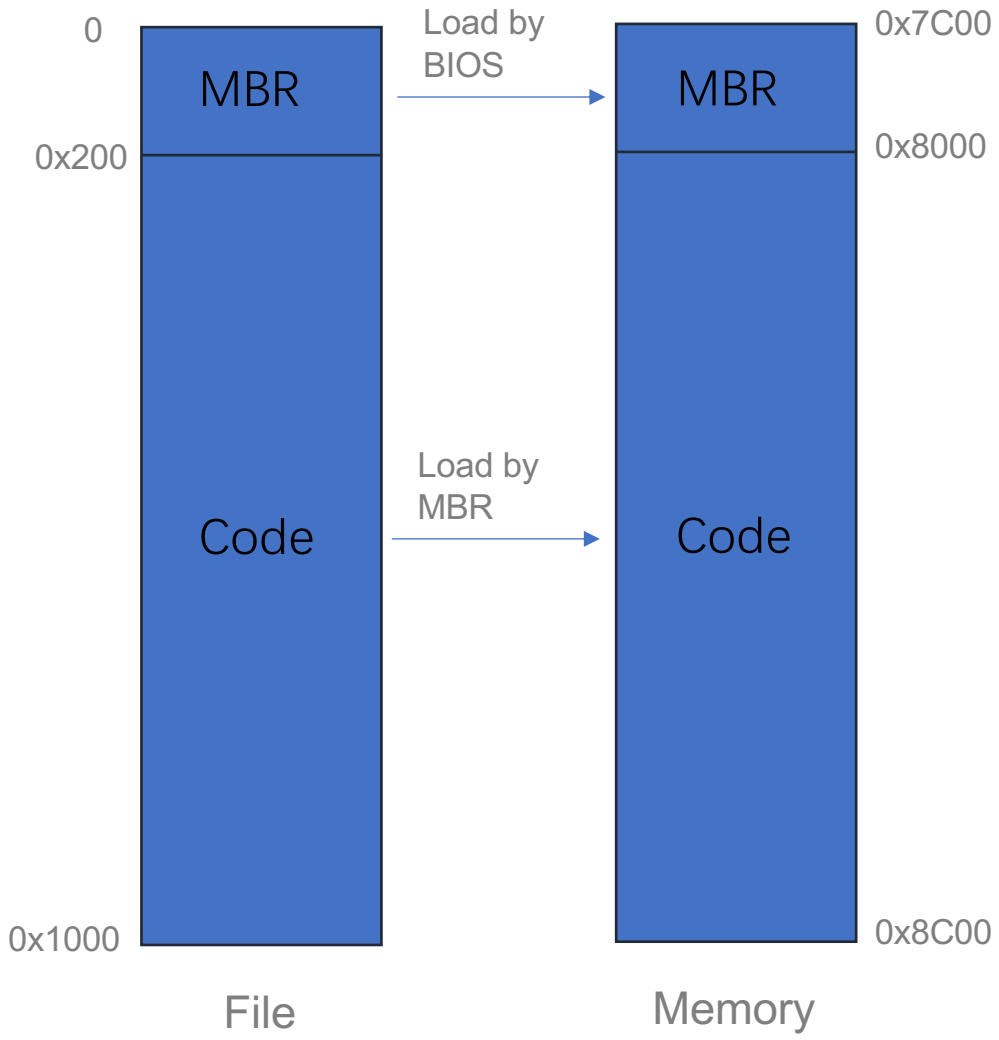




Sample Analysis: First Stage

- 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

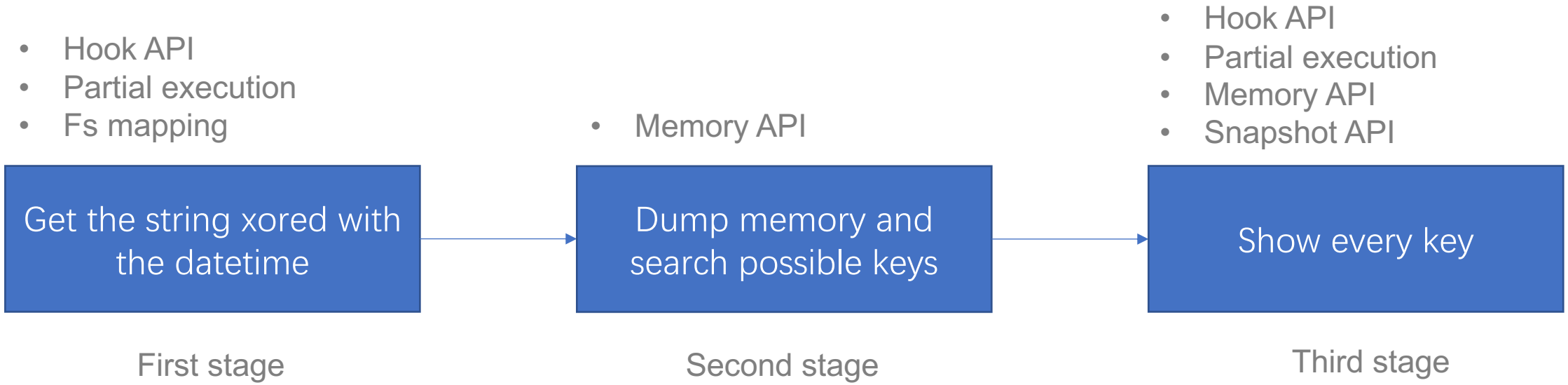
- The logic which starts from 0x8000 is pretty clear
  - Firstly, it gets current datetime by INT 1a and then xors the string at 0x8809 with that datetime
  - Then, it reads user input and xors the same string at 0x8809 with the input
  - Lastly, it initializes the screen and print the ascii art

```
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

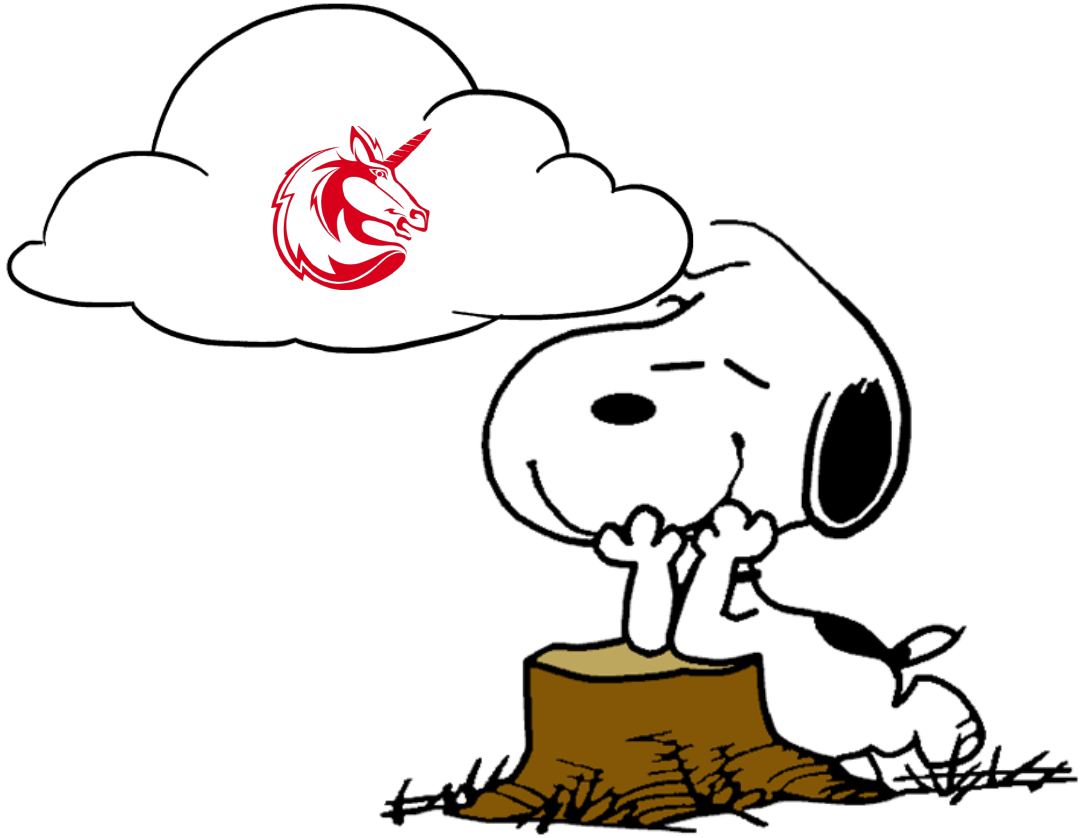
- Execute until 0x8018 and take a snapshot
- Fill in the key in the memory and Skip reading user input
- After completing one round, resume the program to previous snapshot and try next key
- One of the correct keys is: ‘ioperateonmalware’


```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)
```

Future

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



Join Us and Make Pull Request !!!