Qiling Framework: MBR Emulation

November, 2020

Agenda

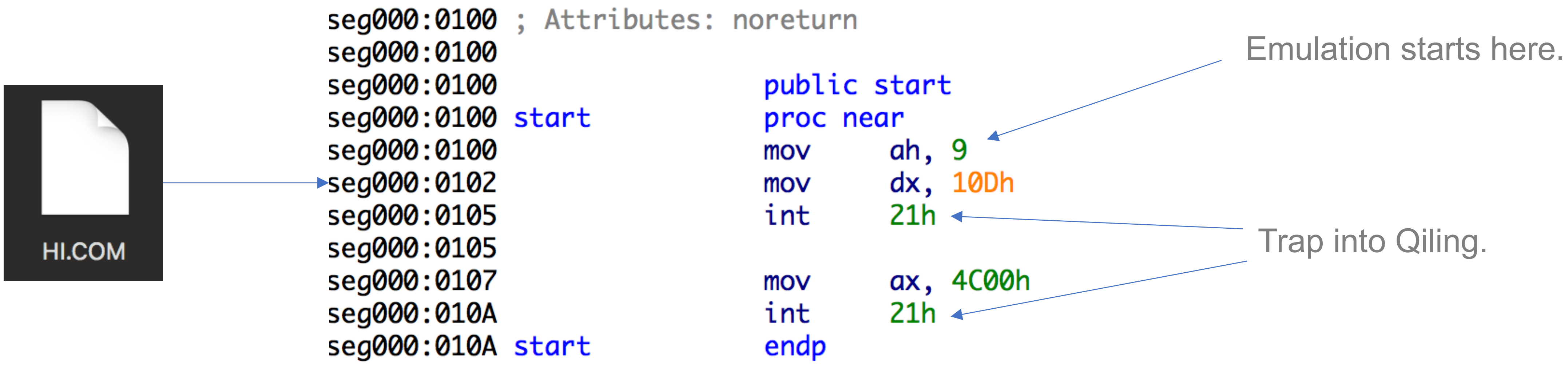
Agenda

- 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.

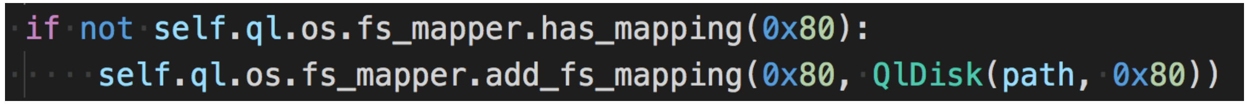
Trap into Qiling.

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

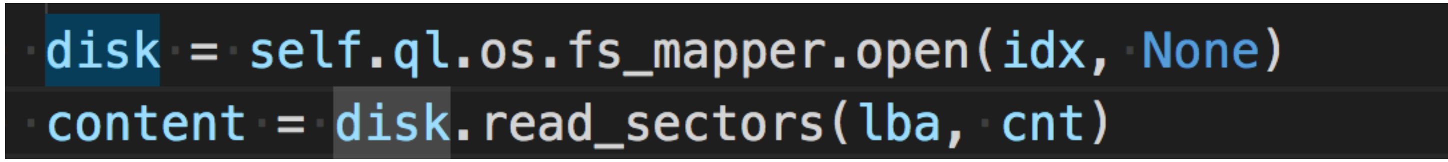
- The place where we implement traditional interrupts.
- Example: INT 13h, ah=42h, read disk sectors.
  - Implemented with fs_mapper API.
  - Map any object which implements FsMappedObject interface to an emulated device/path.
  - QlDisk is inherited from FsMappedObject with CHS and LBA support.
  -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



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

Loader

OS

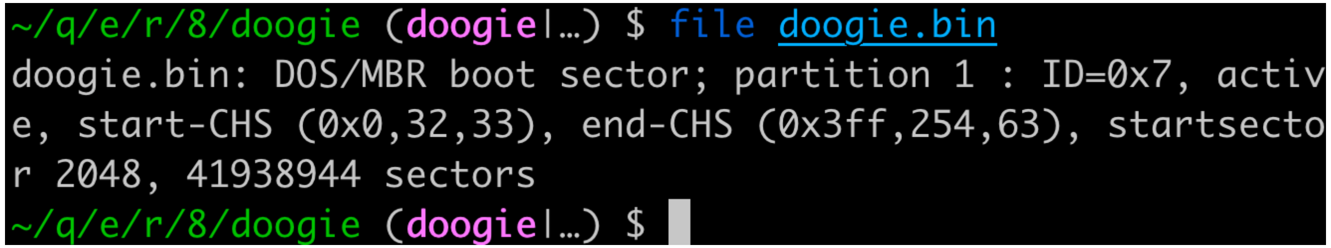


```
disk = self.ql.os.fs_mapper.open(idx, None)
content = disk.read_sectors(lba, cnt)
```

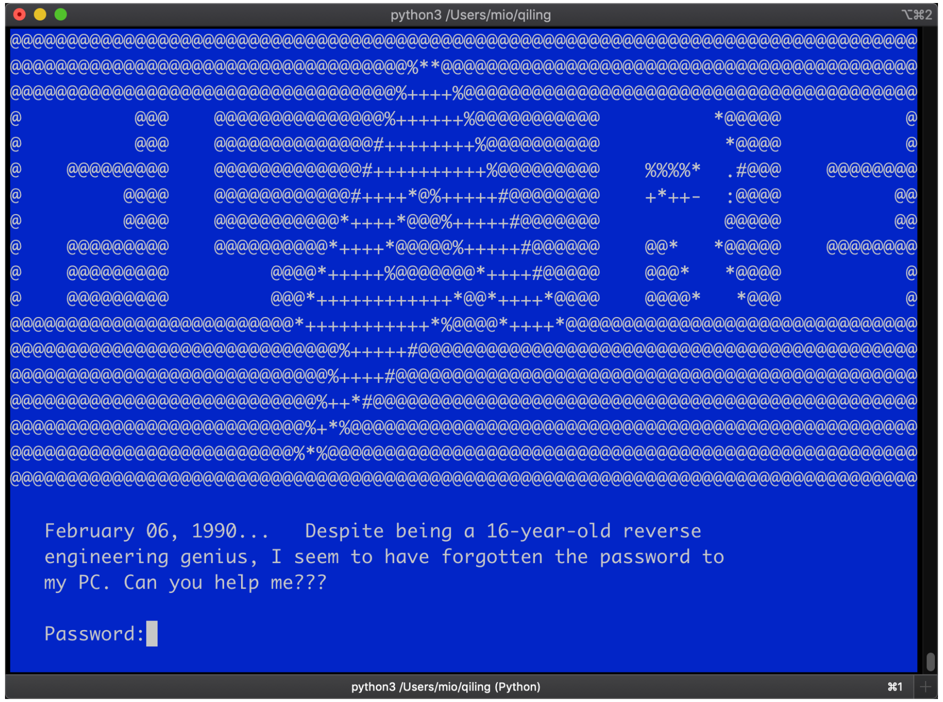
Solve a CTF Challenge

Sample Analysis

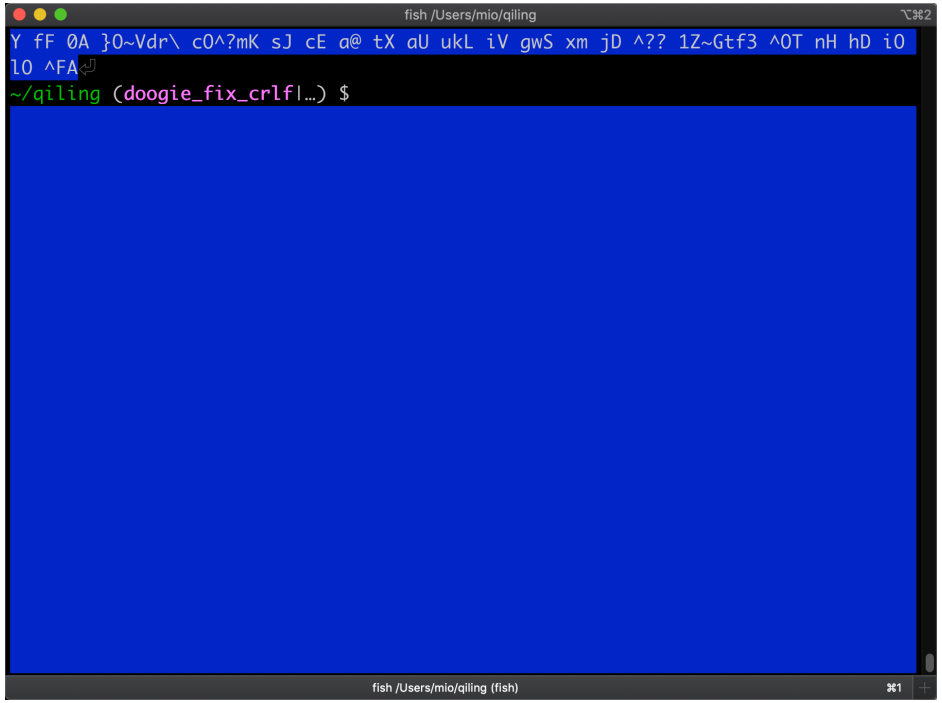
- Sample:
  - Flare-On 5 (2018) Challenge 8 – doogie
  - examples/rootfs/8086/doogie/doogie.bin
- MBR file
- Quick look by qltool.
  - python3 qltool run -f examples/rootfs/8086/doogie/doogie.bin --rootfs examples/rootfs/8086/ --console False
- Try some inputs, but only get gibberish.
- Tips: Feburary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



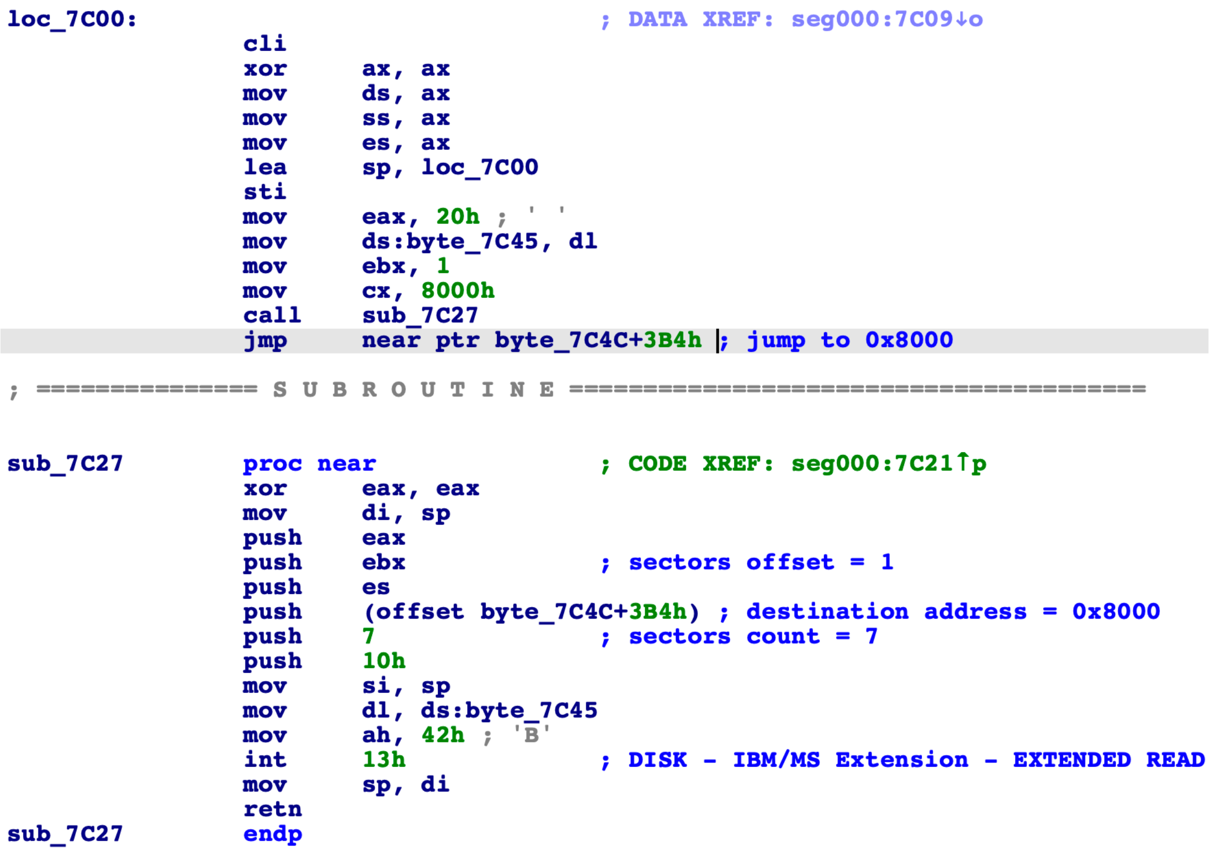
```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@%**@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@%++++%@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@          @@@    @@@@@@@@@@@@@@@%++++++%@@@@@@@@@@@          *@@@@@           @
@          @@@    @@@@@@@@@@@@@@#+++++++++%@@@@@@@@@           *@@@@           @
@    @@@@@@@@@    @@@@@@@@@@@@@#++++++++++%@@@@@@@@@    %%%%*  .#@@@    @@@@@@@@
@         @@@@    @@@@@@@@@@@@#+++++*@%+++++#@@@@@@@    +*++-  :@@@@          @@
@         @@@@    @@@@@@@@@@@*++++*@@@%+++++#@@@@@@@           @@@@@          @@
@    @@@@@@@@@    @@@@@@@@@@*++++*@@@@@%+++++#@@@@@@    @@*   *@@@@@    @@@@@@@@
@    @@@@@@@@@         @@@@*+++++%@@@@@@@*++++#@@@@@    @@@*   *@@@@           @
@    @@@@@@@@@         @@@*++++++++++++*@@*++++*@@@@    @@@@*   *@@@           @
@@@@@@@@@@@@@@@@@@@@@@@@@*+++++++++++*%@@@@*++++*@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@%+++++#@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@%++++#@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@%++*#@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@%+*%@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@%*%@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@


   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???

   Password:
```



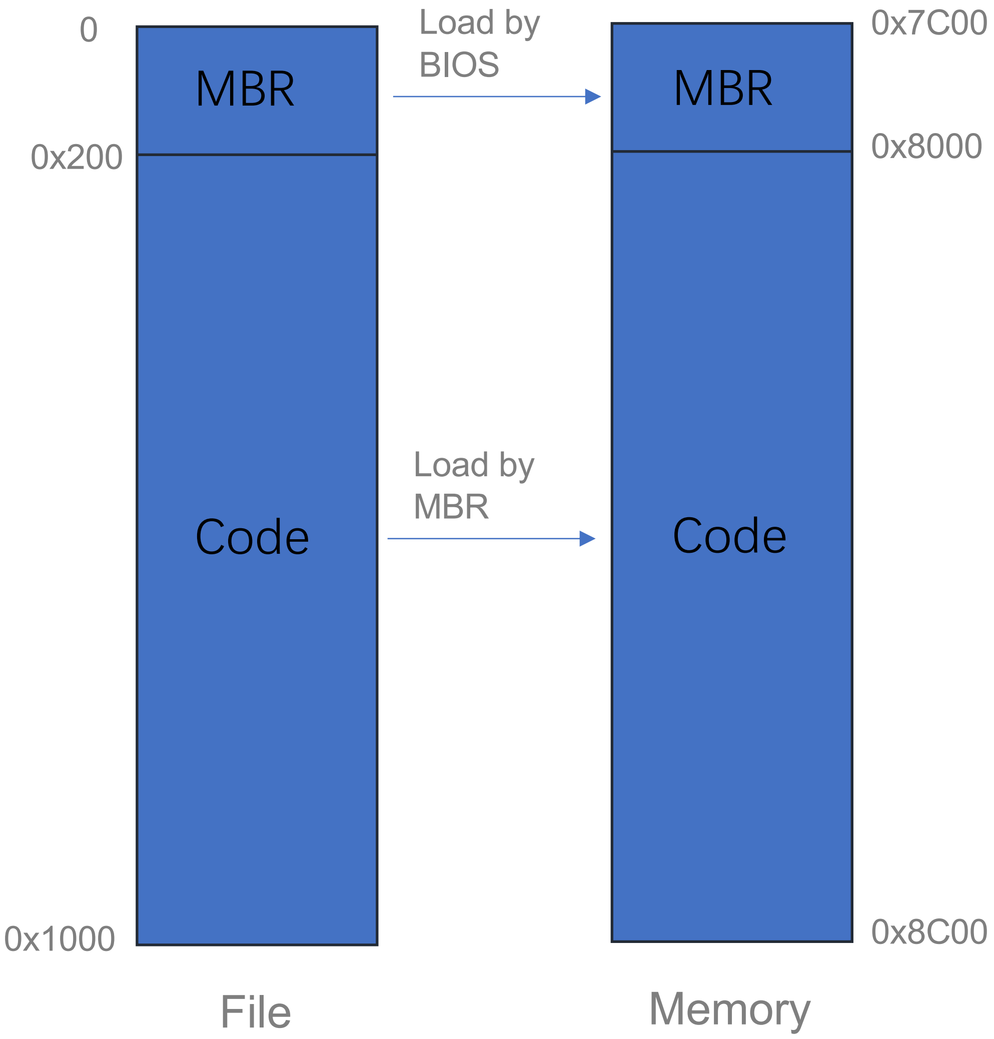
```
Y fF 0A }O~Vdr\ cO^?mK sJ cE a@ tX aU ukL iV gwS xm jD ^?? 1Z~Gtf3 ^OT nH hD iO
lO ^FA⏎
~/qiling (doogie_fix_crlf|…) $
```

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

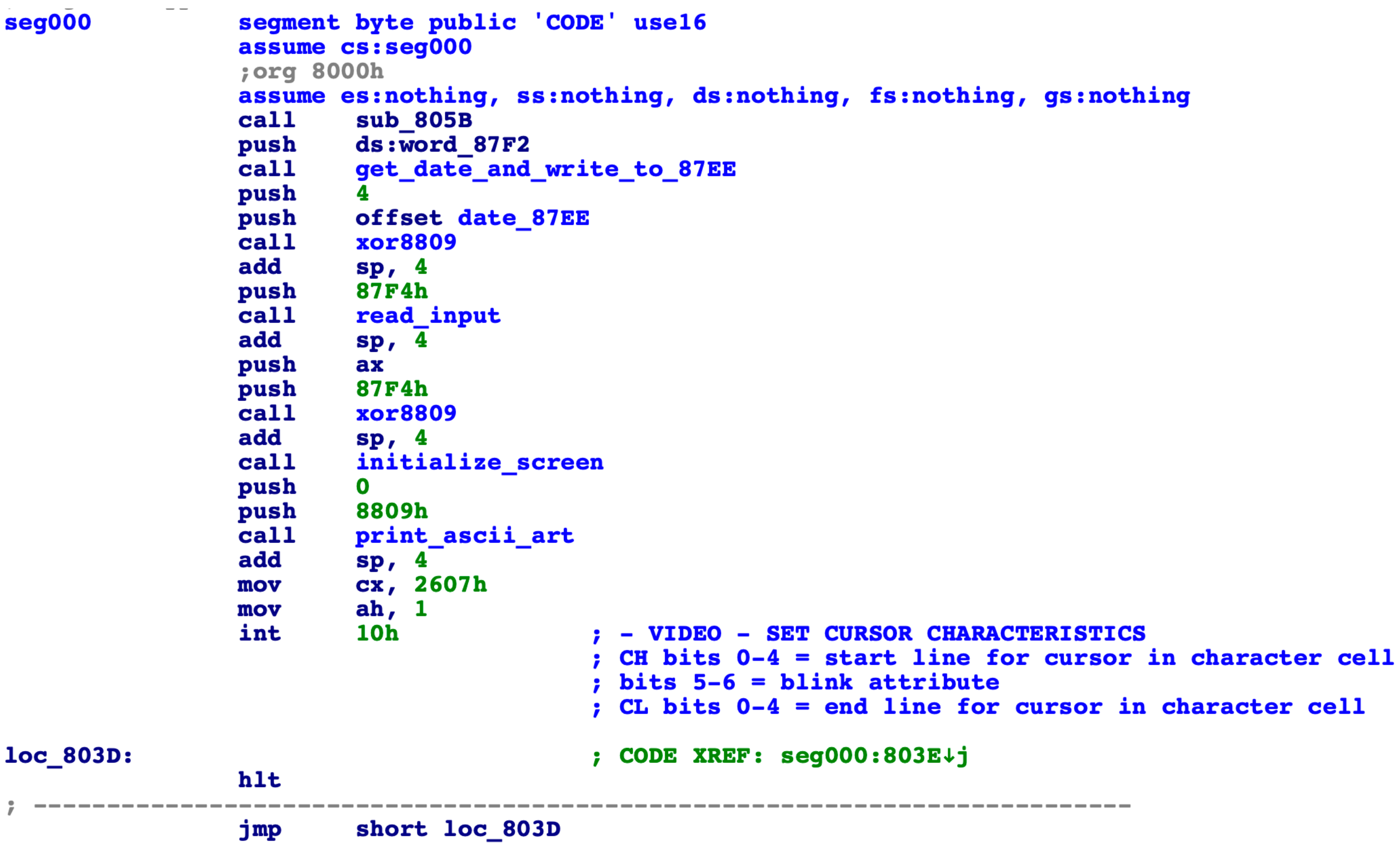


```
loc_7C00:                               ; DATA XREF: seg000:7C09↓o
                cli
                xor     ax, ax
                mov     ds, ax
                mov     ss, ax
                mov     es, ax
                lea     sp, loc_7C00
                sti
                mov     eax, 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

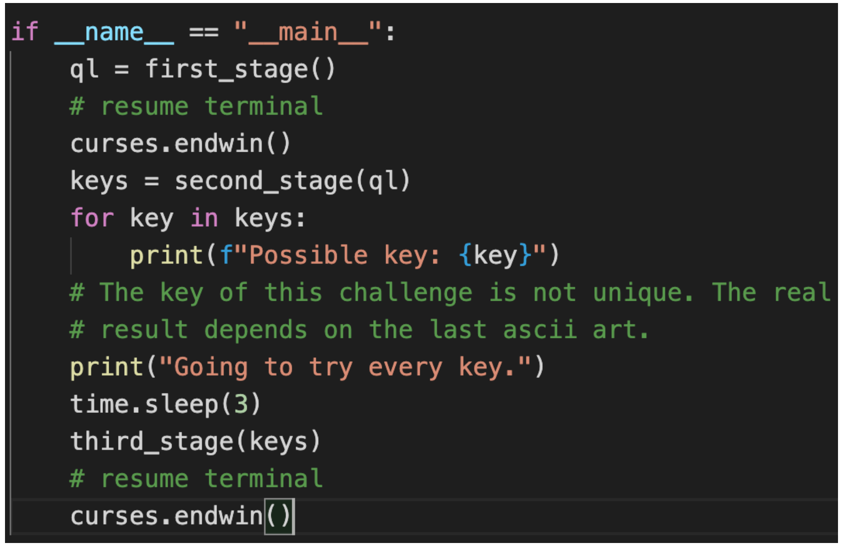
- The logic which starts from 0x8000 is pretty clear
  - Firstly, it gets current datetime by INT 1a and then xors the string at 0x8809 with that datetime
  - Then, it reads user input and xors the same string at 0x8809 with the input
  - Lastly, it initializes the screen and print the ascii art



```
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

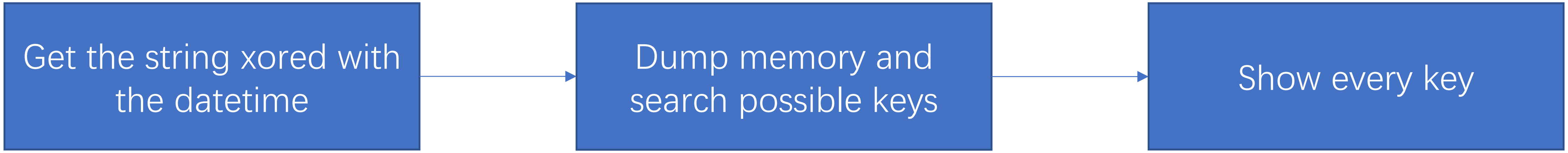


```python
if __name__ == "__main__":
    ql = first_stage()
    # resume terminal
    curses.endwin()
    keys = second_stage(ql)
    for key in keys:
        print(f"Possible key: {key}")
    # The key of this challenge is not unique. The real
    # result depends on the last ascii art.
    print("Going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

- Hook API
- Partial execution
- Fs mapping

- Memory API

- Hook API
- Partial execution
- Memory API
- Snapshot API



Get the string xored with the datetime

Dump memory and search possible keys

Show every key

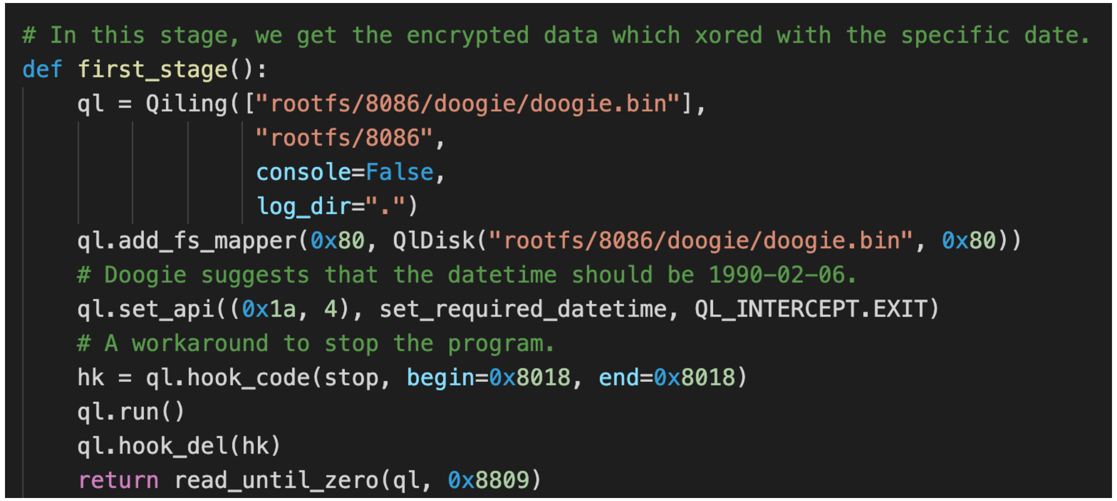
First stage

Second stage

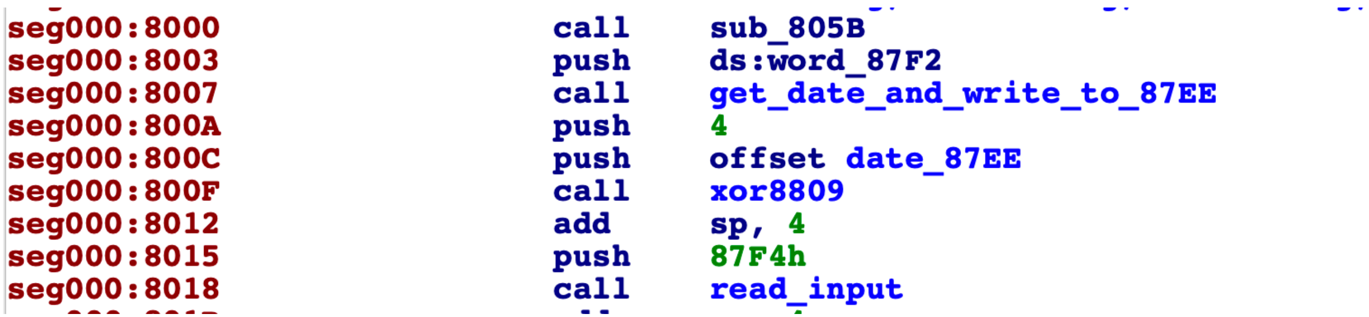
Third stage

Crack: First stage

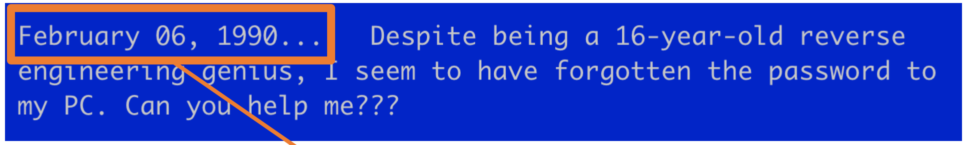
- The quick look before suggests that we have to set the datetime to Feburary 06, 1990
  - It’s extremely easy to achieve that with **ql.set_api**
- The program also read disks directly
  - Use **ql.fs_mapper** API to emulate a disk
- Execute the program until 0x8018
  - At this time, the string at 0x8809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



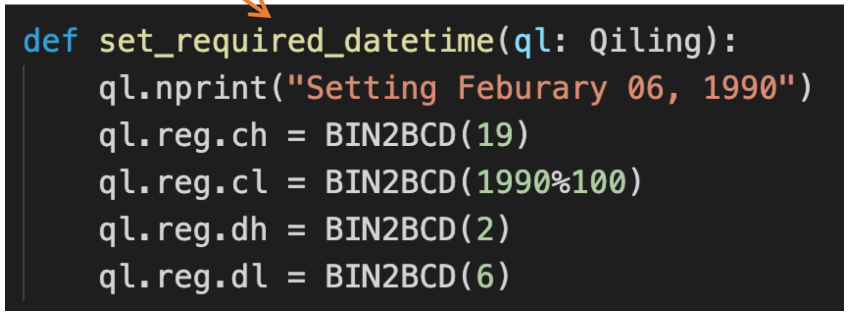
```python
# In this stage, we get the encrypted data which xored with the specific date.
def first_stage():
    ql = Qiling(["rootfs/8086/doogie/doogie.bin"],
                "rootfs/8086",
                console=False,
                log_dir=".")
    ql.add_fs_mapper(0x80, QlDisk("rootfs/8086/doogie/doogie.bin", 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
```



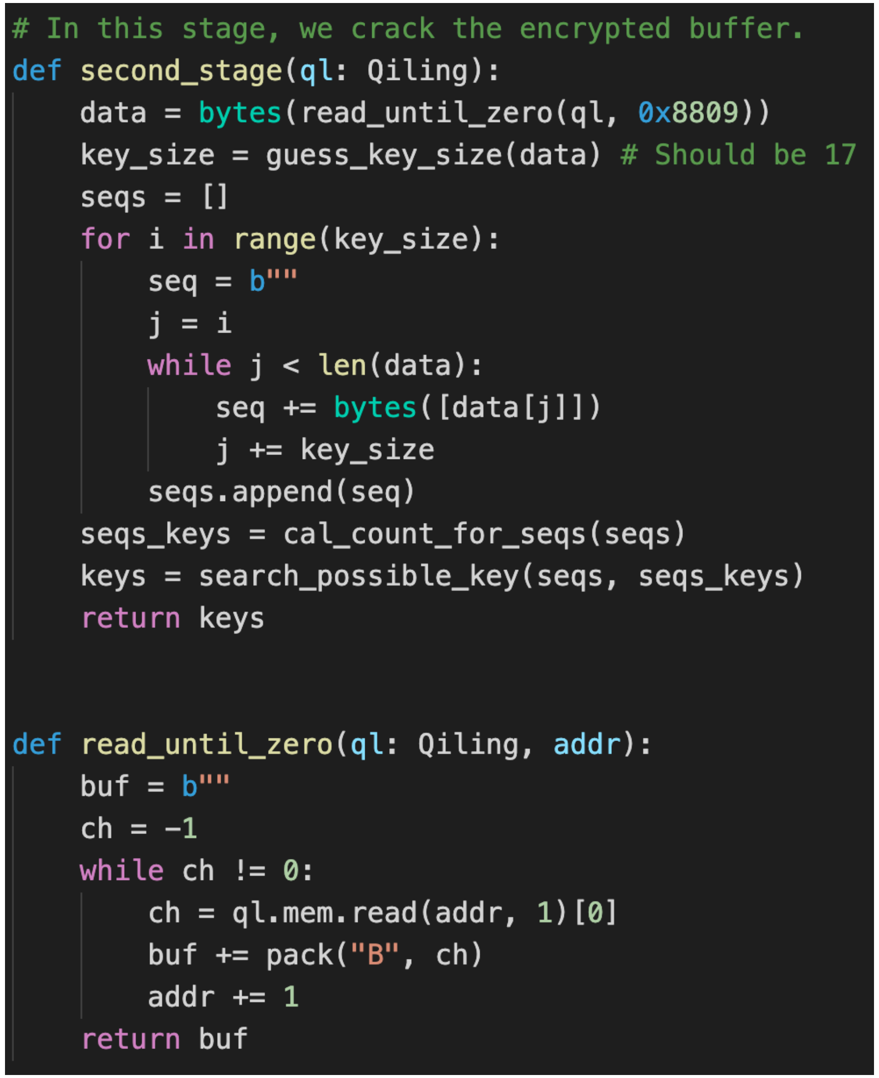
```
February 06, 1990...   Despite being a 16-year-old reverse
engineering genius, I seem to have forgotten the password to
my PC. Can you help me???
```



```python
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

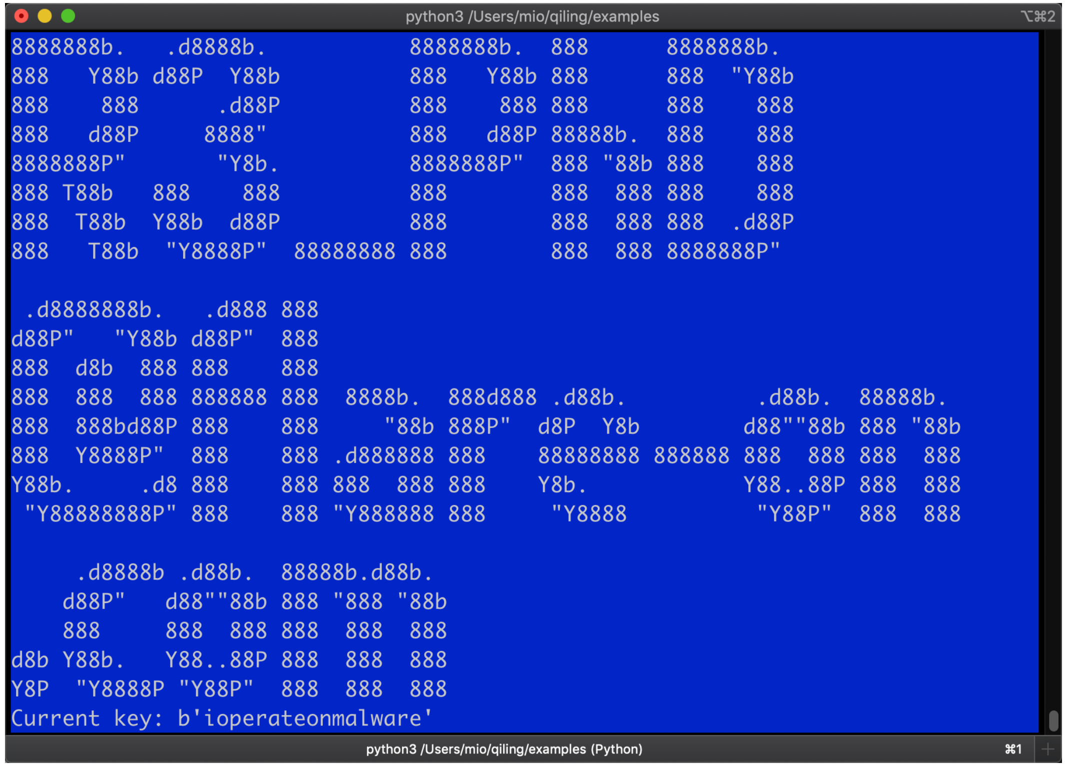


```python
# In this stage, we crack the encrypted buffer.
def second_stage(ql: Qiling):
    data = bytes(read_until_zero(ql, 0x8809))
    key_size = guess_key_size(data) # Should be 17
    seqs = []
    for i in range(key_size):
        seq = b""
        j = i
        while j < len(data):
          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

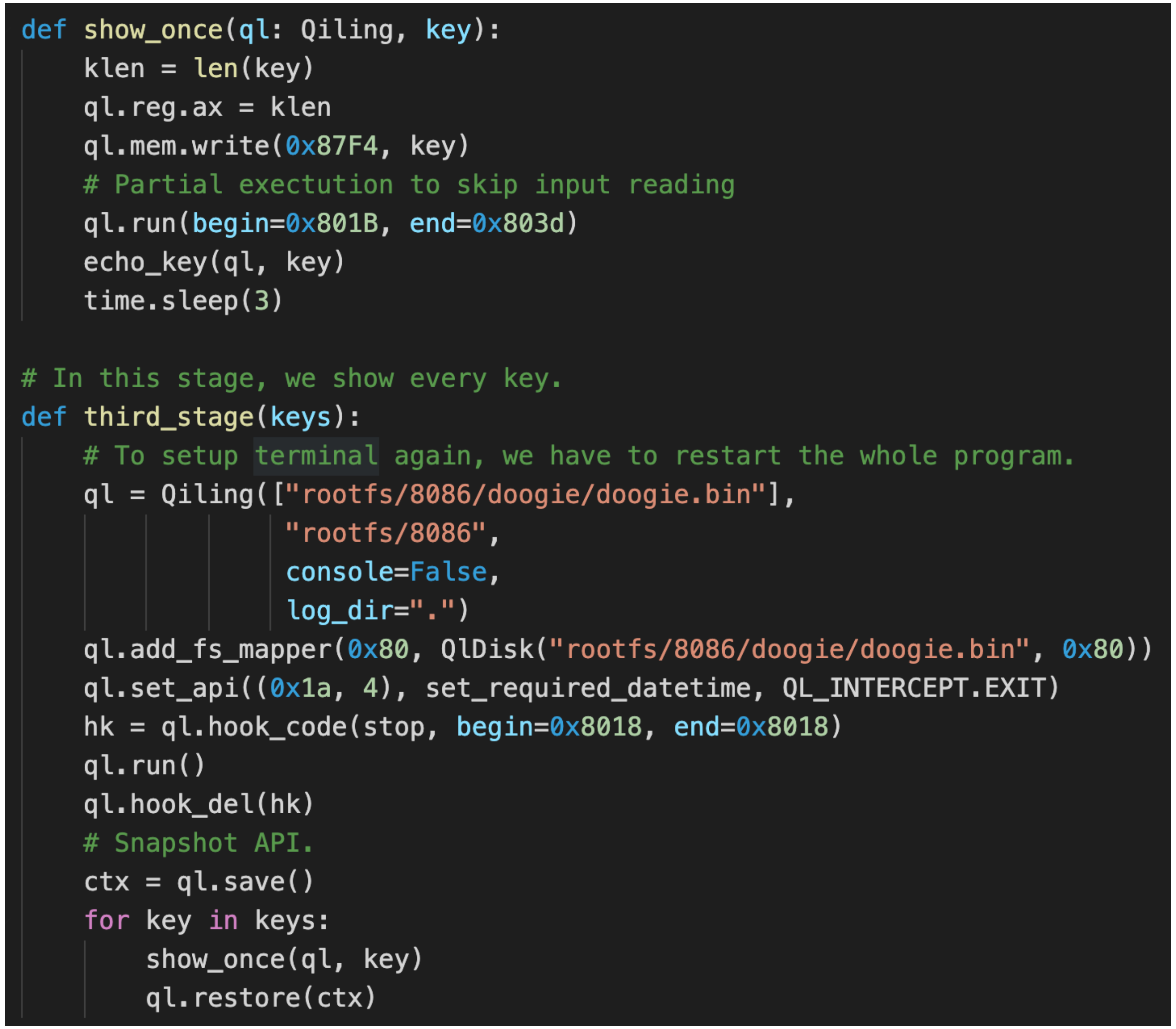
[1]: https://trustedsignal.blogspot.com/2015/06/xord-play-normalized-hamming-distance.html

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’


python3 /Users/mio/qiling/examples
⌥⌘2
python3 /Users/mio/qiling/examples (Python)
⌘1

```
8888888b.   .d8888b.           8888888b.  888      8888888b.
888   Y88b d88P  Y88b          888   Y88b 888      888  "Y88b
888    888      .d88P          888    888 888      888    888
888   d88P     8888"           888   d88P 88888b.  888    888
8888888P"       "Y8b.          8888888P"  888 "88b 888    888
888 T88b   888    888          888        888  888 888    888
888  T88b  Y88b  d88P          888        888  888 888  .d88P
888   T88b  "Y8888P" 88888888 888        888  888 8888888P"

 .d8888888b.   .d888 888
d88P"   "Y88b d88P"  888
888  d8b  888 888    888
888  888  888 888888 888  8888b.  888d888 .d88b.          .d88b.  88888b.
888  888bd88P 888    888     "88b 888P"  d8P  Y8b        d88""88b 888 "88b
888  Y8888P"  888    888 .d888888 888    88888888 888888 888  888 888  888
Y88b.     .d8 888    888 888  888 888    Y8b.            Y88..88P 888  888
 "Y88888888P" 888    888 "Y888888 888     "Y8888          "Y88P"  888  888

     .d8888b .d88b.  88888b.d88b.
    d88P"   d88""88b 888 "888 "88b
    888     888  888 888  888  888
d8b Y88b.   Y88..88P 888  888  888
Y8P  "Y8888P "Y88P"  888  888  888
Current key: b'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)
```

Q&A