Building Next-Gen Security Analysis Tools With Qiling Framework

HITB Lockdown, April 25th, 2020
About xwings

Hoping making the world a better place

- Lab Director / Founder
- Blockchain Research
- IoT Research

Electronic fan boy, making toys from hacker to hacker

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

Cross platform and multi architecture advanced binary emulation framework

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

2005, HITB CTF, Malaysia, First Place /w 20+ Intl. Team
2010, Hack In The Box, Malaysia, Speaker
2012, Codegate, Korean, Speaker
2015, VXRL, Hong Kong, Speaker
2015, HITCON Pre Qual, Taiwan, Top 10 /w 4K+ Intl. Team
2016, Codegate PreQual, Korean, Top 5 /w 3K+ Intl. Team
2016, Qcon, Beijing, Speaker
2016, Kcon, Beijing, Speaker
2017, Kcon, Beijing, Trainer

2018, KCON, Beijing, Trainer
2018, Brucon, Brussel, Speaker
2018, H2HC, San Paolo, Brazil, Speaker
2018, HITB, Beijing/Dubai, Speaker
2018, beVX, Hong Kong, Speaker
2019, Defcon 27, Las Vegas, Speaker
2019, HITCON, Taiwan, Speaker
2019, Zeronight, Russia, Speaker

MacOS SMC, Buffer Overflow, suid
GDB, PE File Parser Buffer Overflow
Metasploit Module, Snort Back Oriffice
Linux ASLR bypass, Return to EDX
About Dlv3/w1tcher/Null/Sp1ke/

Rest of the team members are from theshepherdlab, Dubhe CTF team & community
About NGUYEN Anh Quynh

- Nanyang Technological University, Singapore
- PhD in Computer Science
- Operating System, Virtual Machine, Binary analysis, etc
- Usenix, ACM, IEEE, LNCS, etc
- Blackhat USA/EU/Asia, DEFCON, Recon, HackInTheBox, Syscan, etc
- Capstone disassembler: http://capstone-engine.org
- Unicorn emulator: http:// unicorn-engine.org
- Keystone assembler: http://keystone-engine.org
Agenda

- Motivation
- Shellcode emulation
- Qiling framework
  - Design & implementation
- Build dynamic analysis tools on top of Qiling Framework
- Demo
- Conclusion
Unicorn Emulator framework

- Multi-architectures: Arm, Arm64, M68K, Mips, Sparc, & X86 (include X86_64)
- Native support for Windows & *nix (with Mac OSX, Linux, *BSD & Solaris confirmed)
- Clean/simple/lightweight/intuitive architecture-neutral API
- Implemented in pure C language, with multiple bindings
- High performance by using Just-In-Time compiler technique
- Support fine-grained instrumentation at various levels

Limitation

- Just emulator for low level instructions + memory access
- No higher level concepts of Operating System
  - File format
  - Library
  - Filesystem
  - Systemcall
  - OS structures
How Qiling Got Started
Everything From Executing Shellcode

- Smash Input
- Program Crash
- Craft Payload
- Control Execution Flow
- **Payload** Execution
- Full Control
Traditional Shellcode vs Modern Payload

- More Complex
- Harder to detect
- Designed to bypass detection
- Detection can be
- Network
- System/OS level
Possible Solution(s)

usercorn

Building

Usercorn depends on Go 1.6 or newer, as well as the latest unstable versions of Capstone, Unicorn, and Keystone.

```bash
make deps (requires 'make') will attempt to install all of the above dependencies into the source tree under 'deps/'.
make will update Go packages and build usercorn.
```

Example Commands

- `usercorn run bina/x86.linux.elf`
- `usercorn run bina/x86.64.linux.elf`
- `usercorn run bina/x86.darwin.macho`
- `usercorn run bina/x86_32.darwin.macho`
- `usercorn run bina/x86.linux.x86`
- `usercorn run bina/mipsel.linux.elf`
- `usercorn run -trace bina/x86.linux.elf`
- `usercorn run -trace -to trace.uc bina/x86.linux.elf`
- `usercorn trace -pretty trace.uc`
- `usercorn run -repl bina/x86.linux.elf`

What.

- Usercorn is an analysis and emulator framework, with a base similar to qemu-user.
- It can run arbitrary binaries on a different host kernel, unlike qemu-user.
- While recording full system state at every instruction.
- To a serializable compact format capable of rewinding and re-execution.
- It’s useful out of the box for debugging and dynamic analysis.
- With an arch-neutral powerful lua-based scripting language and debugger.
- It’s also easy to extend and use to build your own tools.

Usercorn could be used to emulate 16-bit DOS, 32-bit and 64-bit ARM/MIPS/x86/SPARC binaries for Linux, Darwin, BSD, DECREE, and even operating systems like Redux.

Right now, x86_64 linux and DECREE are the best supported guests.

- Very good project!
- Mostly *nix based only
- Limited OS Support
- Go and Lua is not hacker’s friendly
- Syscall forwarding
What is Required

Debugger or Disassembler

*BSD  Linux  MacOS  Windows

MIPS  ARM  AARCH64  X86
Why Unicorn

More Emulate = Higher Chances Being Detected
Making A Good “Hackable Shellcode Emulator”

You Need to Be a ASSEMBLER
Each Good for Different ARCH
Each Good for Different Platform
Only Able to Use in Limited Platform
Steep Leaving Curve

Too Complicated to Pick One
Too Debugger Oriented
Limited Option have with Assembler and Debugger
Normally only a Helping Script / IDAPython
Limited Function
Qiling{JiuWei}

- asm
- binary
- hex as argv
- hex as file

Pre-processing

INJECT

APP OS

APP OS

APP OS

Linux/MacOS/Windows/BSD

emu

APP OS

APP OS

APP OS

arm/x86/mips

emu

Execution /w Kernel emulation

dump

executive

INJECT

Execution /w Kernel emulation

Lightweight, Automated, High Performance and Scalable Platform
In Action
AARCH64 Reverse TCP Shellcode
Linux x86_32 input as ASM

Debug and Quiet Mode with HEX, Binary and ASM Input
Running a Windows Shellcode

Calling calc.exe
Qiling Framework

The ACTUAL TALK
Features

- Cross platform: Windows, MacOS, Linux, BSD
- Cross architecture: X86, X86_64, Arm, Arm64, Mips
- Multiple file formats: PE, MachO, ELF
- 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
- Full GDB/IDA/r2 Support
- OS profiling support
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 (No *NIX)
- 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

WSL/2
- Limited ARCH Support
- Only Linux and run in Windows
- Not Sandboxed, It linked to /mnt/c
- No Instrumentation (maybe)

Zelos
- Very good project!
- It’s a Framework!
- Linux based only (No Windows)
- Incomplete support for Linux multi arch
Syscall Forwarding
User Mode Emulation

- qemu-usermode
  - Over Emulate
  - 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**
How Qiling Works
How Does It Work

Base OS can be Windows/Linux/bsd or OSX
And not limited to ARCH
OS Adventure
class ELFParser:
    def __init__(self, path, ql):
        self.path = path
        self.ql = ql
        with open(path, "rb") as f:
            self.elfdata = f.read()
        self.elfid = self.getelfid()
        if self.elfid[0:4] == b'\x7fELF':
            ql.print(">>> ERROR: NOT a ELF")
            exit()
        if self.elfid[0x4] == 1: # 32 bit
            self.is32bit = True
        else:
            self.is32bit = False
        if self.elfid[0x4] == 2: # 64 bit
            self.is64bit = True
        else:
            self.is64bit = False
        if self.elfid[0x5] == 1: # little endian
            self.endian = 1
        elif self.elfid[0x5] == 2: # big endian
            self.endian = 2

class PE32:
    def __init__(self, path, ql, path=":"
        self.ql = ql
        self.uc = ql.uc
        self.path = path
        self.PE_IMAGE_BASE = 0
        self.PE_IMAGE_SIZE = 0
        self.PE_ENTRY_POINT = 0
        self.sizeOfStackReserve = 0
        self.dlls = {}
        self.import_symbols = {}
        self.import_address_table = {}
        self.cmdline = ""
        self/filepath = ""

        def loadX64Shellcode(self, dlls):
            self.initEBX()
            self.initEDX()
            self.initData()
            for each in dlls:
                self.loadDLL(each)

        def loadPE32(self):
            self.pe = profile.PE(self.path, fast_load=True)

            # for simplicity, no image base relocation
            self.ql.PE_IMAGE_BASE = self.pe.PE_IMAGE_BASE = self.pe.OPTIONAL_HEADER.ImageBase
            self.ql.PE_IMAGE_SIZE = self.pe.OPTIONAL_HEADER.SizeOfImage
            self.ql.entry_point = self.pe.PE_ENTRY_POINT = self.pe.PE_IMAGE_BASE + self.pe.OPTIONAL_HEADER.AddressOfEntryPoint
            self.sizeOfStackReserve = self.pe.OPTIONAL_HEADER.SizeOfStackReserve
            self.ql.print(">>> loading %s to 0x%x" % (self.path, self.pe.PE_IMAGE_BASE))
Posix Series - Syscall Emulator

```python
def q1_syscall_read(ql, uc, read_fd, read_buf, read_len, null0, null1, null2):
    path = (ql_read_string(ql, uc, read_buf))
    if read_fd < 256 and ql.file_des[read_fd] != -1:
        try:
            if isinstance(ql.file_des[read_fd], socket.socket):
                data = ql.file_des[read_fd].recv(read_len)
            else:
                data = ql.file_des[read_fd].read(read_len)
            mem_write(read_buf, data)
            ql.outprintf(\"--- Read Completed %d % path\")
            return len(data)
        except:
            return -1
    return -1
ql.outprintf(\"read_fd, size, size\") - % (read_fd, read_buf, read_len, return)
ql_definesyscall_return(ql, uc, return)

def q1_syscall_sendfile(ql, uc, send_fd, send_start, send_end, send_filename, null0, null1, null2):
    send_fd = (ql.file_des[send_fd])
    send_start = (send_start)
    send_end = (send_end)
    send_filename = (send_filename)
    return send_fd
ql_definesyscall_return(ql, uc, return)

def q1_syscall_brk(ql, uc, brk_input, null0, null1, null2, null3, null4):
    brkput = (int(brk_input & 0xffff) / 0x10000 * 0x10000 - brk_address))
    return brk_address
ql_definesyscall_return(ql, uc, return)
ql.outprintf(\"brk_input, brk_address\") - % (ql, return)

def q1_syscall_mprotect(ql, uc, mprotect_start, mprotect_len, mprotect_prot, null0, null1, null2):
    return 0
ql.outprintf(\"mprotect_start, mprotect_len, mprotect_prot, null0, null1, null2\") - % (mprotect_start, mprotect_len, mprotect_prot, return)
ql_definesyscall_return(ql, uc, return)
```

Syscall almost the same for OSX/Linux/*BSD

Kernel Programming 101

Emulate Syscall

Skip/Forward or Emulate Code

Prepare Execution Report

Syscall Implementation
Windows Emulator 0x1

```python
def set_gs_msr(uc, SEGMENT_ADDR, SEGMENT_SIZE):
    uc.mem_map(SEGMENT_ADDR, SEGMENT_SIZE)
    uc.msr_write(GSMSR, SEGMENT_ADDR)
```

```python
def init_TEB_PEB(uc):
    print('>> TEB Addr is: ', hex(config4.GS_LAST_BASE))
    TEB_SIZE = len(TEB8(), bytes())
    tdep_data = TEB_base = config4.GS_LAST_BASE, PEB_Address = config4.GS_LAST_BASE + TEB_SIZE
    uc.mem_write(config4.GS_LAST_BASE, tdep_data, tdep_size)
    config4.GS_LAST_BASE = TEB_SIZE
    data = tdep_data, tdep_size

    print('>> PEB Addr is: ', hex(config4.GS_LAST_BASE))
    PEB_SIZE = len(PEB8(), bytes())
    pdep_data = PEB_base = config4.GS_LAST_BASE, _ldrAddress = config4.GS_LAST_BASE + PEB_SIZE
    uc.mem_write(config4.GS_LAST_BASE, pdep_data, pdep_size)
    config4.GS_LAST_BASE = PEB_SIZE
    _ldr_data = _ldr_data = config4.GS_LAST_BASE,
    _ldr_data = config4.GS_LAST_BASE,
    `Int.loadOrderModuleList = {'Path': config4.GS_LAST_BASE + 0x39, 'Blk': config4.GS_LAST_BASE + 0x39, 'Name': config4.GS_LAST_BASE + 0x39, 'Init': config4.GS_LAST_BASE + 0x39}
    uc.mem_write(config4.GS_LAST_BASE, _ldr_data, _ldr_size, _ldr_size)
```
Sample Code on How To Execute X86_32/64bit Windows Shellcode
CPU Adventure
X86 32/64 Series

X86 32/64bit GDT For Linux

```
ql_x86_setup_gdt_segment_ds(ql, ql.uc)
ql_x86_setup_gdt_segment_cs(ql, ql.uc)
ql_x86_setup_gdt_segment_ss(ql, ql.uc)
```

X86 32bit GDT For Windows

```
# New set GDT Share with Linux
ql_x86_setup_gdt_segment_fs(ql, ql.uc, ql.FS_SEGMENT_ADDR, ql.FS_SEGMENT_SIZE)
ql_x86_setup_gdt_segment_gs(ql, ql.uc, ql.GS_SEGMENT_ADDR, ql.GS_SEGMENT_SIZE)
ql_x86_setup_gdt_segment_ds(ql, ql.uc)
ql_x86_setup_gdt_segment_cs(ql, ql.uc)
ql_x86_setup_gdt_segment_ss(ql, ql.uc)
```

X86 64bit GDT For Windows

```
def set_pe64_gdt(ql):
    # uc.mem_map(GS_SEGMENT_ADDR, GS_SEGMENT_SIZE)
    # setup_gdt_segment_uc, GDT_ADDR, GDT_LIMIT, UC_X86_REG
    # setup_gdt_segment(ql, GDT_ADDR, GDT_LIMIT, UC_X86_REG)
    # GSMSR = 0x00000101
    ql.uc.mem_map(ql.GS_SEGMENT_ADDR, ql.GS_SEGMENT_SIZE)
    ql.uc.msr_write(GSMSR, ql.GS_SEGMENT_ADDR)
```

Setup segments GDT and Set Thread Area
ARM/64 Series

```python
def ql_arm_init_kernel_get_tls(uc):
    uc.mem_map(0xFFFF0000, 0x1000)
    sc = 'adr r0, data; ldr r0, [r0]; mov pc, lr; data: ascii \
x0\x00'
```

```python
def ql_arm64_enable_vfp(uc):
    ARM64FP = uc.reg_read(UC_ARM64_REG_CPACR_EL1)
    ARM64FP |= 0x300000
    uc.reg_write(UC_ARM64_REG_CPACR_EL1, ARM64FP)
```

```
main:
    mcr p15, 0, r0, c13, c0, 3
    adr r1, ret_to
    add r1, r1, #1
    bx r1
```
MIPS32EL Series

MIPS Comes with CO Processor

Configuration needed for CO Processor

Unicorn does not support Floating Point

Patch Unicorn to Support CO Processors

Custom Binary Injected for Set Thread Area
Applications of Qiling + Demo
Build dynamic analysis tools - Basic

- Let Qiling load the binary into memory (loading + dynamic linking)
- Syscall & system API logging available, provided by default

```python
from qiling import *

def run_sandbox(path, rootfs, ostype, output):
    ql = Qiling(path, rootfs, ostype = ostype, output = output)
    ql.run()

if __name__ == "__main__":
    run_sandbox(["rootfs/arm_linux/bin/arm32-hello-static"], "rootfs/arm_linux", "linux", "debug")
```
Build dynamic analysis tools – Basic ++

- Let Qiling loads the binary (loading + dynamic linking)
- Syscall & system API logging available, provided by default
- Program callbacks with Qiling hook capabilities: hook memory access, hook address range
- Repeat in a loop: run() → analysis → resume()

```python
from unicorn import *
from capstone import *
from qiling import *

md = Cs(CS_ARCH_X86, CS_MODE_64)

def print_asm(ql, address, size):
    buf = ql.uc.mem_read(address, size)
    for i in md.disasm(buf, address):
        print("{:0%x:\t%s\t%s"%(i.address, i.mnemonic, i.op_str))

if __name__ == "__main__":
    ql = Qiling(["rootfs/x8664_linux/bin/x8664_hello"], "rootfs/x8664_linux")
    ql.hook_code(print_asm)
    ql.run()
```
Firmware analysis

- Emulation offers a chance to move analysis to a much more powerful platform
- Emulate a single binary is better than whole firmware
  - Hardware emulation is tough without hardware specs
  - Series of different firmware can share the same target binary
- Challenges
  - Dump firmware, or extract firmware from binary blob
  - Extract the target binary
  - NVRAM emulation
  - Dependency libraries
  - Presence of other devices: wireless interface
Guided fuzzer – cross platform/architecture

- Cross platform/architecture: Windows, MacOS, Linux, BSD on X86, Arm, Arm64, Mips
- https://github.com/qilingframework/qiling/tree/dev/examples/fuzzing
Malware analysis

- Analyze malware in Qiling sandbox
- Cross-platform analysis
- API logging available to summarize malware behaviour at API level
- Randomization Wrapper
- Optional GDB compatible debugger

```python
[PATH]
username = Qiling
cdrive = C:\
windir = %cdrive%\windows\
userdir = %cdrive%\users\
appdata = %userdir%\%\%\username\AppData\
userhome = %cdrive%\users\%\%\username\Temp\%
```

```python
[SYSTEM]
# Major Minor ProductType
majorVersion = 10
minorVersion = 0
productType = 1
language = 1093
VER_SERVICEPACKMAJOR = 0
computer_name = qilingpc
permission = root
```

```python
[CREATE]
userdata = %userdir%\%\%\username\AppData\&
userhome = %userdir%\%\%\username\Temp\%
```

```python
[REGISTRY]
registry_diff = registry_diff.json
```

```python
[VOLUME]
path = C:
serial_number = 3224810732
type = NTFS
sectors_per_cluster = 10
bytes_per_sector = 512
number_of_free_clusters = 12345
number_of_clusters = 65536
```

```python
[NETWORK]
dns_response_ip = 10.20.30.40
```
Debugger – GDB / IDAPro/ r2
Gandcrab

- Ransomware
- Use evasion and privilege escalation techniques
- Steal and store information from the target computer
- Tries to encrypt everything
Al-Khaser

- PoC Malware with good intentions
- Used to stress an anti-malware sandbox
- Performs many malware common tricks (Anti-debugging, Anti-injection, Anti-dumping and more)
- Emulated only partially

```
[al-khaser version 0.67]
--------------------------[Debugger Detection]--------------------------
[*] Checking IsDebuggerPresent API () [=] GOOD
[*] Checking PEB.BeingDebugged [=] GOOD
[*] Checking CheckRemoteDebuggerPresent API () [=] GOOD
[*] Checking PEB.NtGlobalFlag [=] GOOD
[*] Checking ProcessHeap.Flags [=] GOOD
[*] Checking ProcessHeap.ForceFlags [=] GOOD
[*] Checking NtQueryInformationProcess with ProcessDebugPort [=] GOOD
[*] Checking NtQueryInformationProcess with ProcessDebugFlags [=] GOOD
[*] Checking NtQueryInformationProcess with ProcessDebugObject [=] GOOD
[*] Checking NtSetInformationThread with ThreadHideFromDebugger [=] GOOD
[*] Checking CloseHandle with an invalid handle [=] GOOD
[*] Checking UnhandledExcepFilterTest [=] GOOD
[*] Checking OutputDebugString [=] GOOD
[*] Checking Hardware Breakpoints [=] GOOD
[*] Checking Software Breakpoints [=] GOOD
[*] Checking Interrupt 0x2d [=] BAD

We are done for now
```
One More Thing
Qiling is a Python-based lightweight emulator framework

- Built-in shellcode emulator
- Emulate Operating System to support full binary
- Well maintained by a good team of researchers
- Version 1.0 released TODAY

Come more exciting binary analysis tools built on top of Qiling!
Call for sponsor for development of Unicorn 2

- Current Unicorn is based on Qemu 2.1.2, from 2015
- Planning for Unicorn 2, based on new Qemu (5+)
- Some new exciting APIs in planning

NGUYEN Anh Quynh, aquynh -at- gmail.com, @unicorn_engine