www.groundx.io/demigod

Demigod: The Art of Emulating Kernel Rootkits

Blackhat USA
August 5th, 2020

NGUYEN Anh Quynh
NGUYEN Hong Quang
DO Minh Tuan

About us

NGUYEN Anh Quynh

- Nanyang Technological University, Singapore. PhD in Computer Science
- Operating System, Virtualization, Binary analysis, etc
- Frequent speaker of Blackhat USA/EU/Asia, Defcon, Recon, Syscan, HackInTheBox, etc
- Founder of few open source reverse engineering frameworks: Capstone, Unicorn & Keystone.
- Contact: **aquynh @ gmail.com**

About us

NGUYEN Hong Quang

- Security Researcher at Viettel Cyber Security (VCS)
- Interested in Fuzzing and Exploits
- I would like to thank my wife & my children. Without them, I would finish this research few months earlier ;-)
- Contact: **quangnh89** @ **gmail.com**

About us

DO Minh Tuan

- Security Researcher at CyStack., JSC
- Interested in Fuzzing and Exploits
- BabyPhD CTF team's member
- Speaker at Blackhat USA/Asia, T2, XCon, etc
- Contact: **tuanit96** @ **gmail.com**

Agenda

- Background & Motivation
- Design & Implementation
  - Cross-platform framework
  - Windows
  - MacOS
  - Linux
- Demos
- More applications of Demigod
- Conclusion

Background & Motivations

Kernel Rootkit

- Computer system = Userland (ring 3) + OS Kernel (ring 0)
  - Kernel operates at lowest level, with full control on whole system
- Malware run inside kernel
  - God-mode code, with full power
  - Can fool all monitoring & defensive mechanisms
  - Very hard to detect & kill
    - Can be creative with lots of tricks to evade detection
    - Can even kill or fool any anti-malware tools
  - Mostly come in the form of kernel modules/drivers
    - Easy to build & maintain + high compatibility
    - Windows: .SYS file
    - MacOS: .KEXT file
    - Linux LKM: .KO file

Analyze Kernel Rootkit

- Goal: analyze rootkits in kernel modules/drivers
  - .SYS, .KEXT, .KO
- Dynamic analysis is troublesome
  - Need 2 machines (physical or virtual)
    - One machine to load kernel rootkits
    - Another machine to run analysis tools
      - Remote debugger
  - Crash machine
  - Low visibility when kernel run at very low level
  - Few tools built for kernel-level analysis
- Static analysis
  - Packed code?
  - Inaccurate analysis
    - Real kernel setting depends on config
      - Kernel modules are blind
    - Missing API

Better Tool for Kernel Rootkits?

- Analyze ring 0 code in ring 3?
  - No more machine crash
  - No more headache of setting up separate machines, or virtual machines
  - No more risk with kernel level malware
  - Good visibility
  - Existing tools can be reused
    - Monitor, trace, debug, etc
- Cross-platform-architecture analysis
  - Analyze Windows kernel rootkits on Linux - and vice versa
  - Analyze Arm rootkits on X86 would be fun ;-)
- High-level language scripting is possible - such as Python
  - So it is quick & efficient to build new analysis tools

Emulator Approach

- Emulate OS kernel in software, so kernel modules/drivers can be run inside
  - Emulator runs in ring 3
    - No more machine crash
    - No more headache of setting up separate machines, or virtual machines
    - No more risk with kernel level malware
    - Existing tools can be reused
  - Excellent visibility since we can all execution code
    - Instrumentation at various levels
- Cross-platform-architecture analysis
  - Cross-platform-architecture emulator
- High-level language scripting is possible - such as Python
  - Emulator as framework
  - Python-based emulator ;-)

Design of Demigod

Qiling Emulator (qiling.io)

- Emulator - for userland code only
  - Cross-platforms: Windows, MacOS, Linux
  - Cross-architectures: X86, Arm, Arm64, Mips
  - Multiple executable formats: PE, MachO, ELF
  - Python-based framework
  - Instrumentation at various levels
  - Emulate system API (userland)
  - Enable debugging (with GDB server)
- Cannot handle kernel code → so cannot emulate kernel rootkis
  - Loaders for kernel modules are different from userland binaries
    - Different format structures
  - Do not emulate kernel components
  - Do not emulate kernel API
  - Kernel module has different execution scheme
    - Load vs Serve request from userland

Emulator for Kernel Rootkits - 1

- Build on top of Qiling!
  - Reuse the excellent frameworks with many ready features
    - Loader + little modification
    - Symbol relocation
    - Imported functions
  - Emulate kernel components
  - Emulate kernel API by hooking & emulate its semantics
    - Sometimes we can forward to kernel code (PASS-THRU)
  - Instrument memory access and code execution for dynamic analysis
    - System API hooking
    - Automatic alloc/map missing memory to enable continuous execution
  - Facility to assist dynamic analysis
    - Reuse Qiling GDB server for debugging

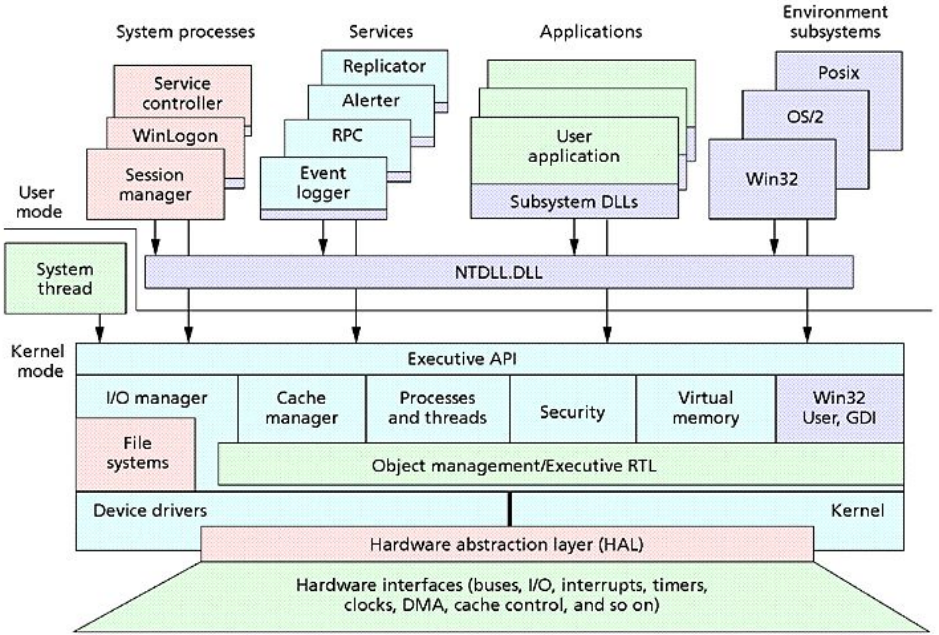
Emulator for Kernel Rootkits - 2

- Userland code is typically just 1 phase emulation
- Kernel driver is 2-phase scheme
  - Driver needs to be loaded first, then it stays in kernel to serve user request
  - Emulate driver entry
    - Locate entry to emulate
      - Windows: DriverEntry
      - MacOS: realmain
      - Linux: module_init
    - During emulation, extract code paths for next phase
      - Rootkit syscalls
      - IOCTL callbacks
      - Other registered callbacks
  - Emulating selected code paths
    - User choose which code path to emulate
      - Input provided by user

Windows

Windows Overview

- Windows kernel
  - Kernel image: ntoskrnl.exe
    - cache manager
    - executive
    - kernel
    - security reference monitor
    - memory manager
    - Scheduler
  - Driver: .SYS
    - Portable Executable (PE) format
    - Subsystem: native
- Goal: emulate kernel rootkit in .SYS driver



Load .SYS File

- PE loader for .SYS file
  - Parse all sections of PE file and map all sections into emulator
  - Resolve Import Address Table and load all dependent DLLs into emulator.
  - Relocation
- Setup CPU context
  - Common registers (RSP, ...)
  - Control registers (CR4, CR8...)
- Initialize Windows structures
  - EPROCESS, DRIVER_OBJECT, KUSER_SHARED_DATA, ..
  - Setup IMAGE_LOAD_CONFIG_DIRECTORY: SecurityCookie value
- Locate initialization function
  - NTSTATUS NTAPI DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath);

Windows Internal Structures

- DRIVER_OBJECT
  - represent image of a loaded kernel-mode driver.
- IRP
  - represent an I/O request packet.
- KUSER_SHARED_DATA
  - data structure which is shared between user mode and kernel mode.
- IO_STACK_LOCATION
  - define an I/O stack location
- MDL
  - partially opaque structure that represents a memory descriptor list (MDL).

Emulating Driver Initialization

- Setup arguments of DriverEntry():
  - 1st argument: A pointer to a DRIVER_OBJECT structure (driver object)
  - 2nd argument: A pointer to a UNICODE_STRING structure (path to driver registry).
- Emulate from DriverEntry
  - Emulate Windows API functions
    - CreateIoDevice
    - CreateSymbolicLink

Hook Windows Kernel APIs

- Emulate some Windows APIs necessary for driver to work
  - RtlInitUnicodeString
  - IoCreateDevice, IofCompleteRequest
  - Memory management:
    - ExAllocatePool / ExFreePool
    - ExAllocatePoolWithTag / ExFreePoolWithTag
  - MmGetSystemRoutineAddress
  - Runtime functions (strcpy, wcscpy, ...)
  - Registry functions: NtOpenKey, ...
  - Thread functions: PsCreateSystemThread
  - Object functions: ObfDereferenceObject, ObOpenObjectByPointer

```
# void IofCompleteRequest(
#   PIRP  Irp,
#   CCHAR PriorityBoost
# );
@winapi(cc=STDCALL,
        params={
            "Irp": POINTER,
            "PriorityBoost": CCHAR
            })
def hook_IofCompleteRequest(ql, address, params):
    return None
```

Emulate Code Path

- Userspace communicates with driver via IOCTL code (DeviceIoControl) or ReadFile/WriteFile
  - Create structures in memory
    - IRP
    - IO_STACK_LOCATION, IO_STACK_LOCATION_PARAM
  - Fill all structures with appropriate data
  - Setup arguments for callbacks:
    - IRP_MJ_DEVICE_CONTROL
    - IRP_MJ_READ/IRP_MJ_WRITE
  - Start emulate callbacks
  - On return, determine NTSTATUS value & return data.
    - Retrieve IRP structure from emulator memory

Windows I/O methods

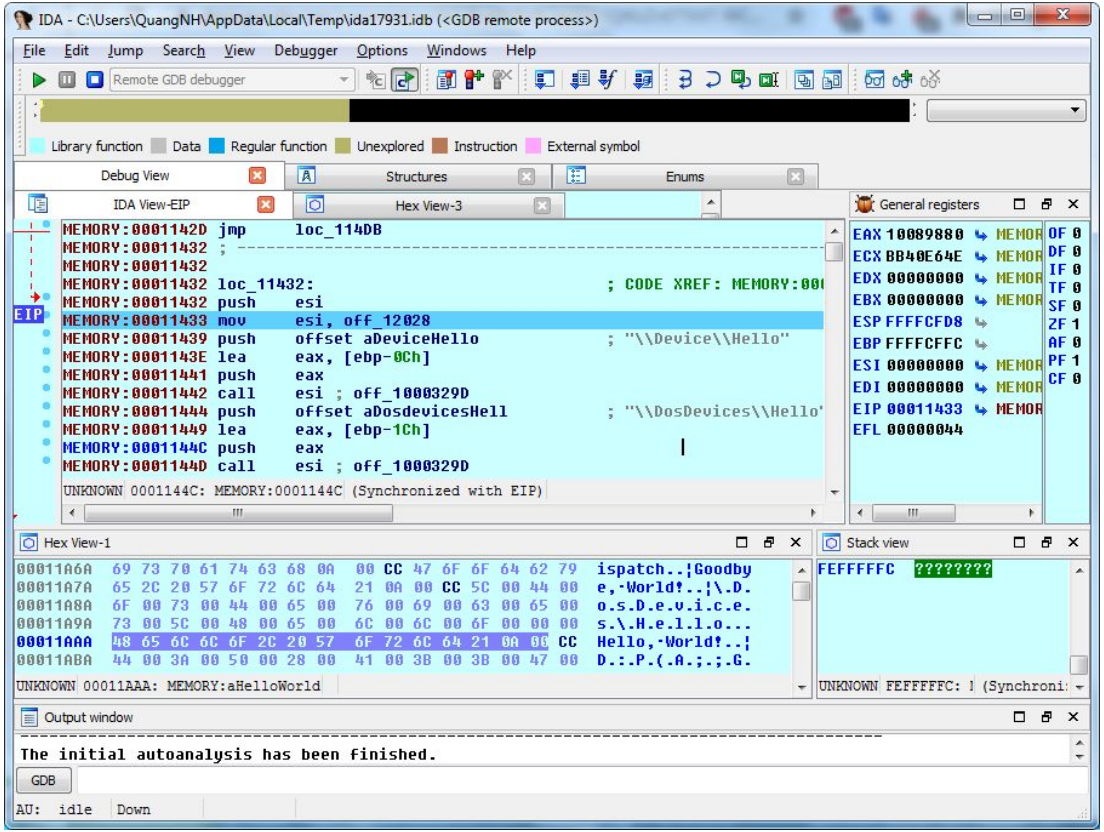
- BUFFERED_IO
  - Create a large enough buffer and assign address of buffer to irp.AssociatedIrp.SystemBuffer
- DIRECT_IO
  - Create a buffer to store data
  - Create a MDL structure
    - MappedSystemVa == StartVa == address of buffer
    - ByteCount = size of buffer
- NEITHER_IO
  - UserBuffer field of IRP structure points to address of buffer

Demigod Tools

- **dm-unpack.py**: A tool to unpack kernel mode packer
  - Dump PE file from emulator environment to file
  - Add new section to store Import Address Table (IAT)
  - Rebuild Import Address Table
    - Find all call/jmp to possible IAT function pointers
    - Verify pointers by checking export address table of other modules
- **dm-ioctl-reverse.py**: a tool to help when working with IOCTL codes
  - Bruteforce emulation to find IOCTL from binary
  - Check and count all basic blocks

Demo: Debug Windows driver

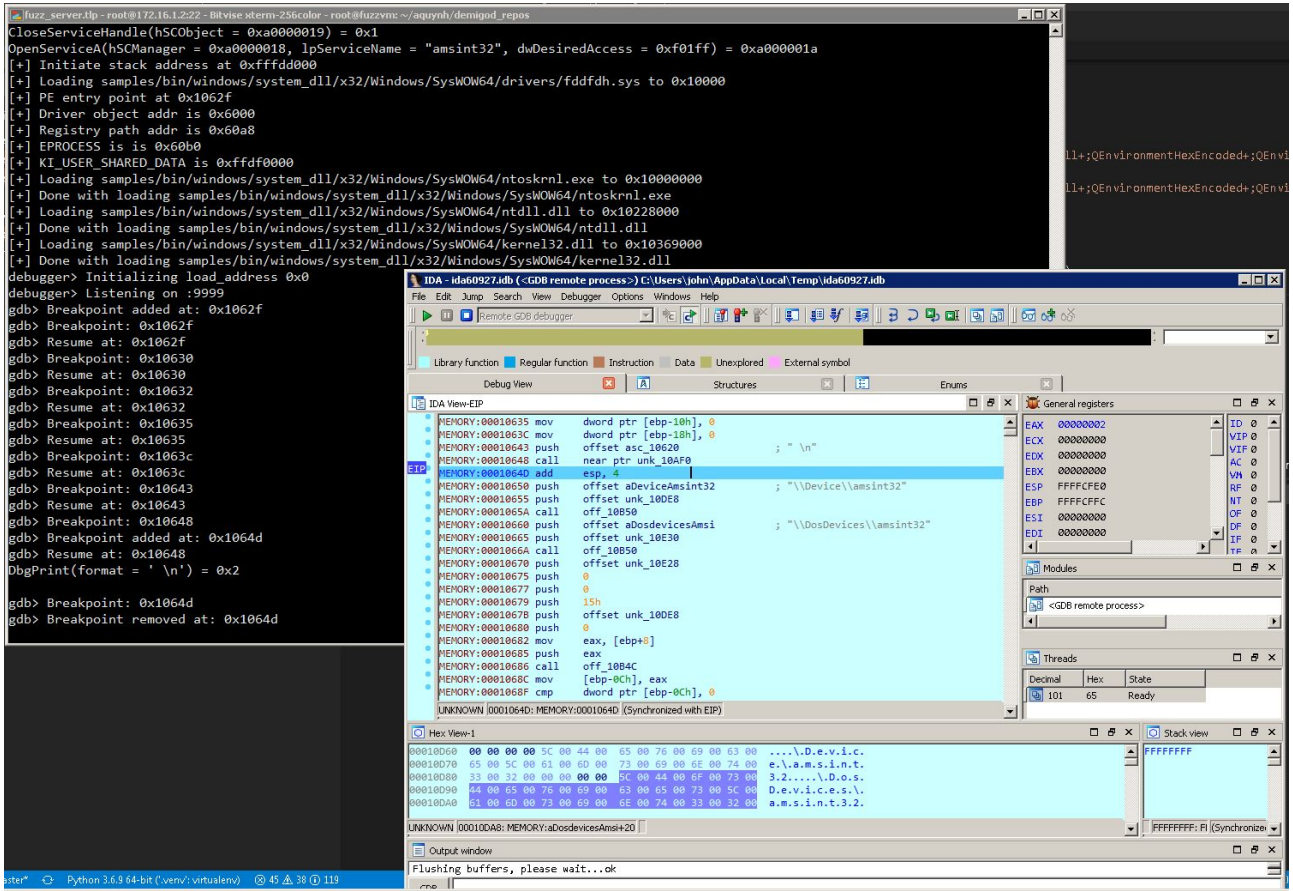
- Setup:
  - Enable gdb mode
    - ql.debugger = True
    - ql.debugger= ":9999"
- Execute script
- Connect IDA to port 9999 & trace
- Demo: https://youtu.be/5Tzc0rMfYSg



Sality malware

- Sality is a polymorphic file infector, targeting Windows executable files
- Communicate over a peer-to-peer (P2P) network to form a botnet
- Rootkit module
  - Terminate processes via NtTerminateProcess
  - Block access to some anti-virus resources
- Sample available
  - Fae6b86d04f93bcad7736d88beff90f1278c3e3d786a25fc45c9ef2ee375df0f
- Demo: https://youtu.be/ID4cZpWnwkw

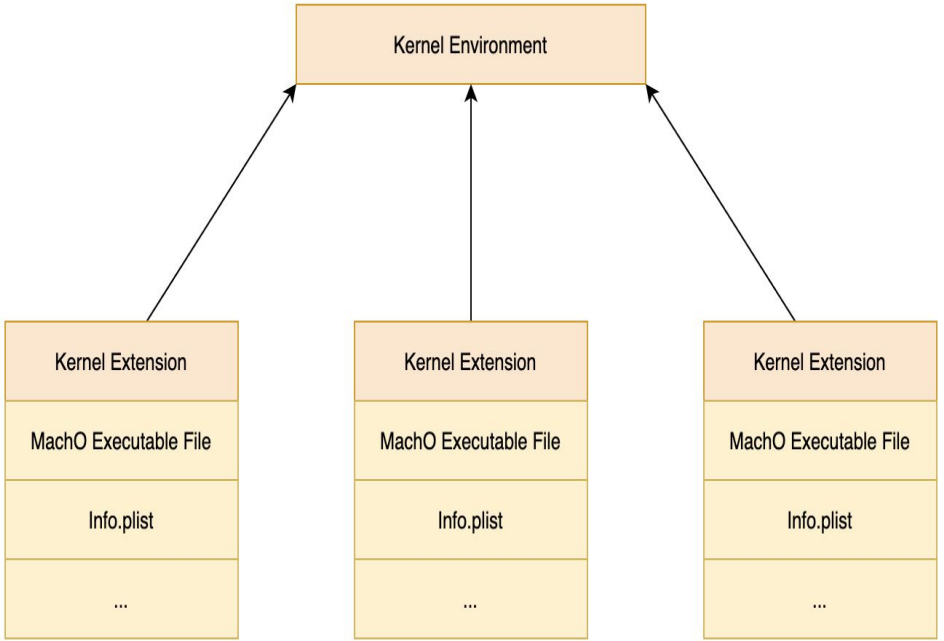
Demo: Debug Sality rootkit



MacOS

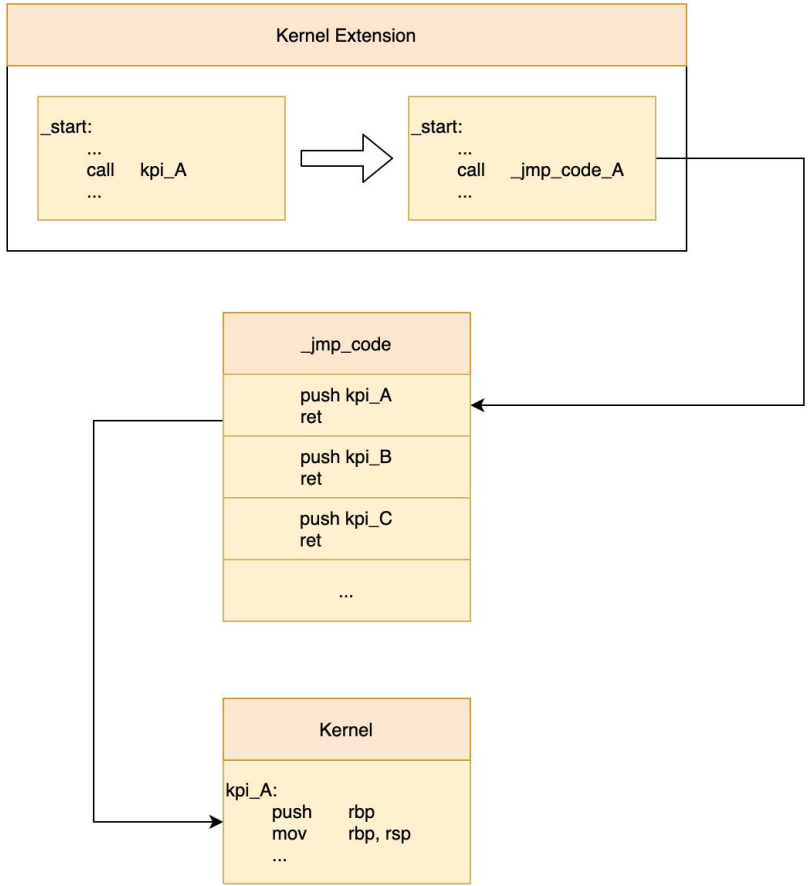
MacOS Overview

- XNU, hybrid kernel
  - Mach-O x86_64 executable file
  - Kernel Programming Interface (KPI)
  - All implementation code of KPIs are inside
- Kernel extension (KEXT):
  - Born to be a bundle
  - Load in on-demand by kernel
  - Extra attributes come from plist file
- Goal: emulate kernel rootkit in KEXT



Load KEXT

- Load all SEGMENT64s of kernel & KEXT to emulator
- Kernel extension (KEXT)
  - Resolve local relocations
  - Resolve dynamic symbols:
    - Create jmp code section
    - Get symbol address from loaded kernel
  - Retrieve initial methods/functions addresses
    - IOKit: ***::start()***
    - Generic driver: ***__realmain***

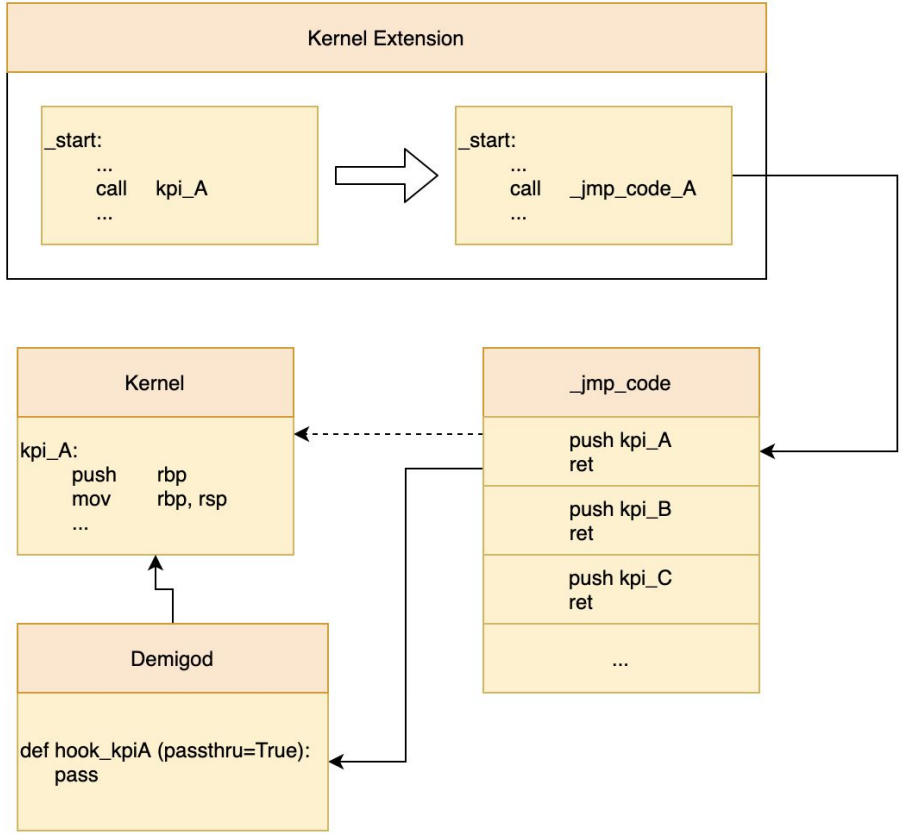


Emulating KEXT Initialization

- Initialize kernel environment
  - Setup MAC policy list
    - Allocate new object in emulator
    - Fill the address on loaded ***mac_policy_list*** symbol on kernel space
  - Create some default processes
    - Allocate new objects in emulator
    - Link them by LIST/SLIST structure
    - Fill the addresses on loaded ***allproc*** symbol on kernel space
  - Make some vnodes, current credential, ...
- Run emulation for pre-process methods / functions
  - IOKit: ***attach()***, ***probe()***, ...
  - Generic driver: ***kmod_info,*** ...
- Emulate from entry point of driver

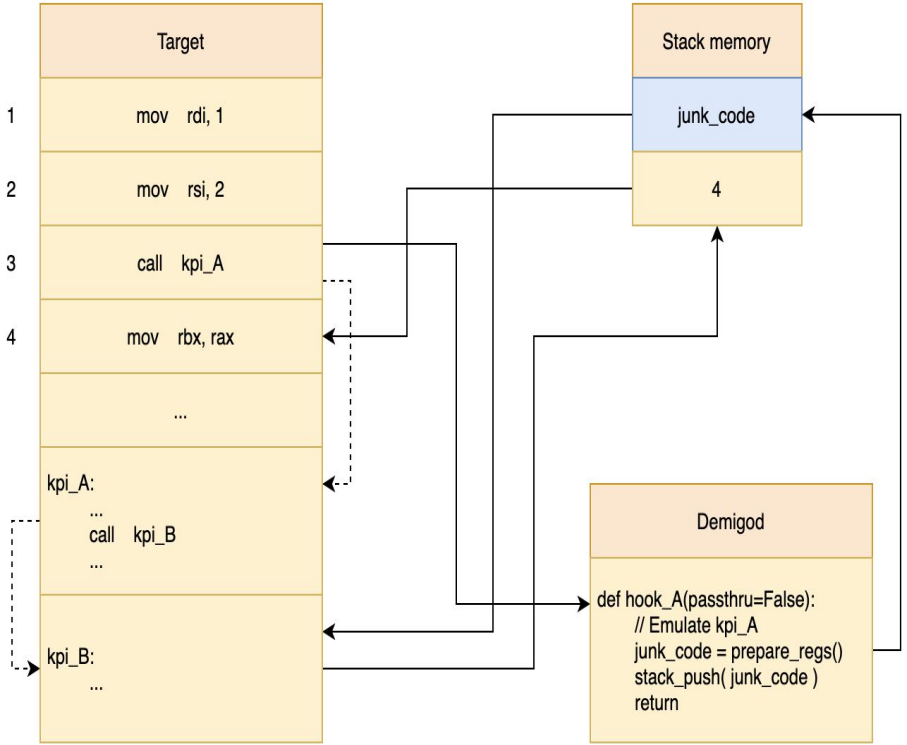
Instrument KPI

- Map every KPIs to user-defined hook methods
  - Simplify features
  - Use native functions
- Interact with real environment
  - ***getattrlistbulks()***: retrieve attributes from a path, then pack the result by calling ***vfs_attr_pack()*** KPI
- Syscall emulation
  - Get ***sysent*** from kernel
  - Assign arguments to registers & run corresponding syscall entry address



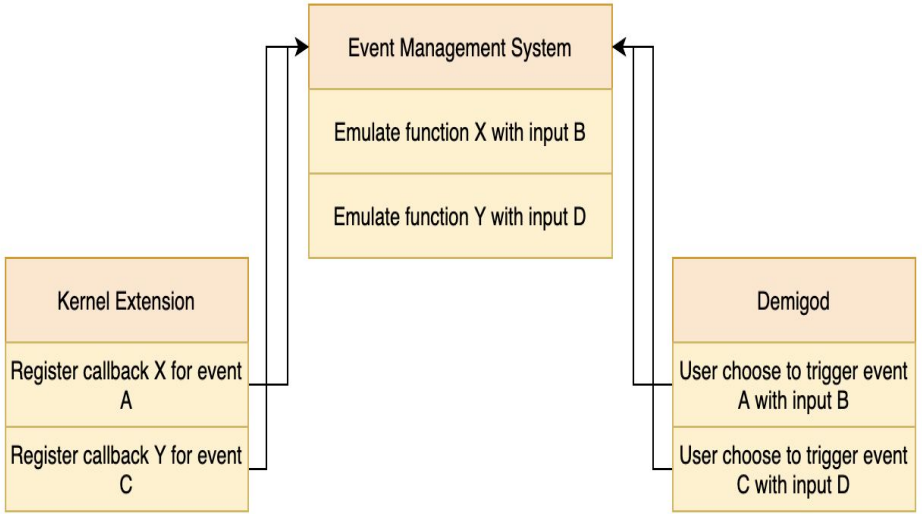
Nested Emulation

- Call to a native KPI from a hooked KPI
  - Need to save current context to emulate new code path
  - Solution
    - Construct junk code
    - Push address of junk code to stack
    - Return
- The junk code has 3 missions
  - Prepare arguments (registers / stack)
  - Add RSP to clear arguments if necessary
  - Jump to address of that KPI



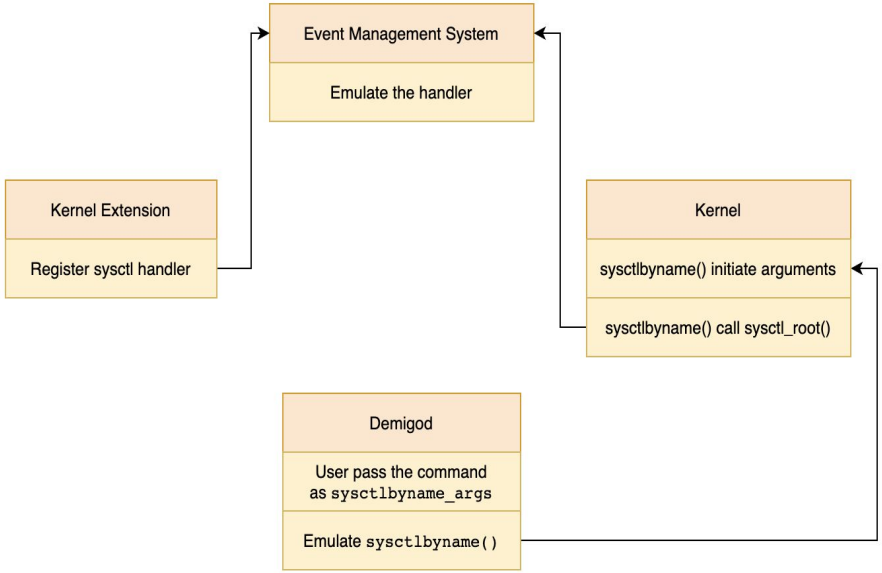
Emulate Code Path

- User “talks” to driver through callbacks
- Solution
  - Event Management System (EMS)
  - Hook KPIs to collect callback functions
  - Emulate callbacks by user trigger specific events
- From Asynchronous system to Synchronous system
  - Load driver
  - EMS collect callbacks from driver
  - User choose to trigger an event with customzed inputs



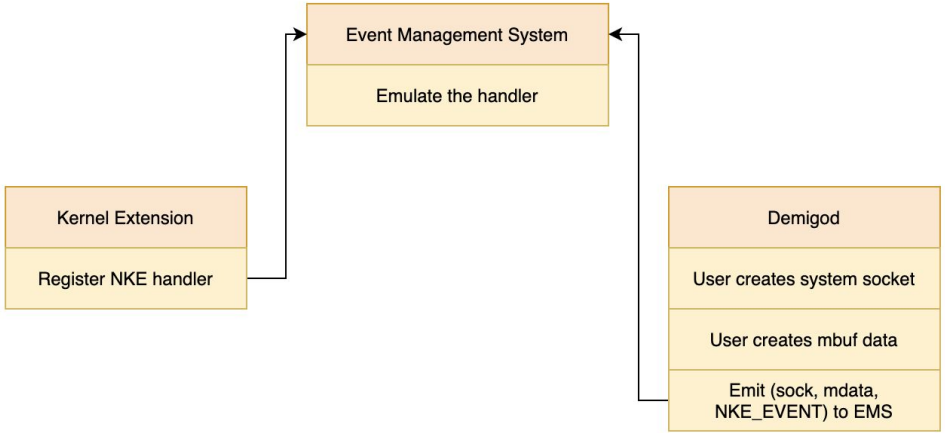
Event Management System - SYSCTL

- Hook ***sysctl_register_oid()*** to register SYSCTL events
  - Retrieve name of handler from struct
  - Retrieve callback & commit to EMS
- Hook ***sysctl_unregister_oid()*** to unregister events from EMS
- Hook ***sysctl_root()*** to trigger event from EMS
- User pass input as a ***sysctlbyname_args*** object to ***sysctlbyname()***
  - ***sysctlbyname()*** converts input to some structs
  - ***sysctl_root()*** take structs & trigger event



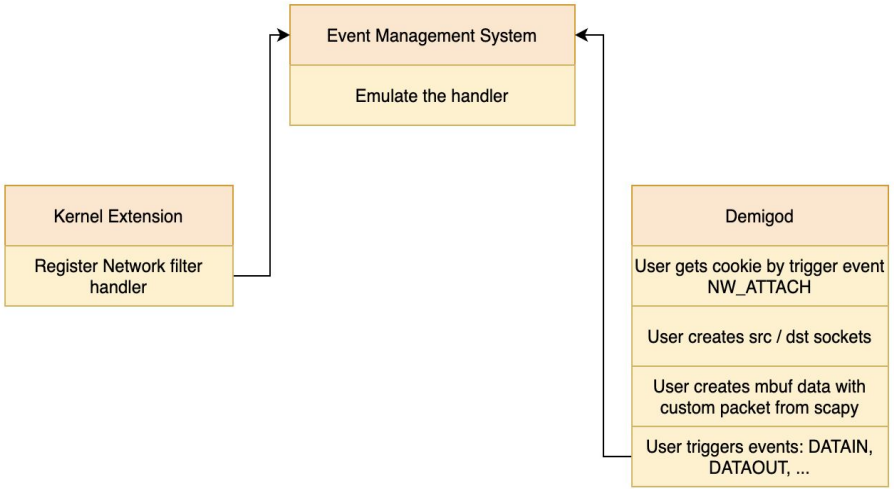
Event Management System - NKE

- Hook ***ctl_register()*** to register NKE events
  - Retrieve name of ctl instance from struct
  - Check if callbacks are valid, then commit to EMS
  - Store address of struct to ***ctl_ref***
- Hook ***ctl_deregister()*** to unregister events from EMS
  - Get name from struct stored in ***ctl_ref***
- User creates socket object and ***mbuf*** data to trigger NKE event



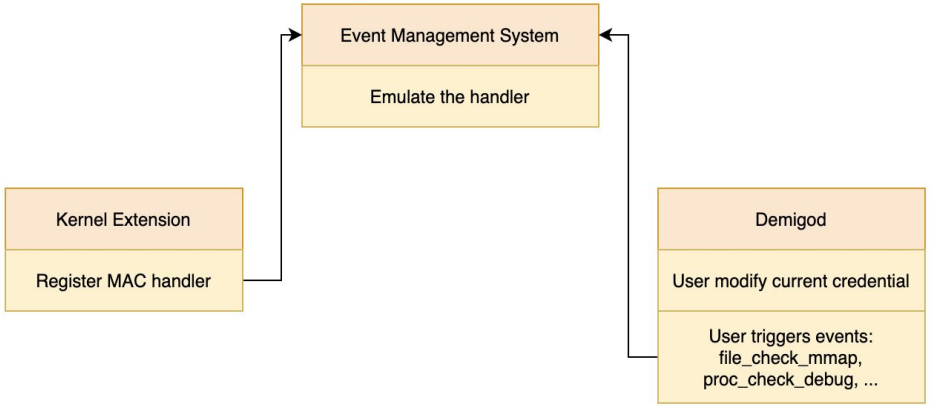
Event Management System - Network Filter

- Hook ***sflt_register()*** or ***ipf_addv4()*** to register NETWORK events
  - Retrieve name from struct
  - Retrieve callback & commit to EMS
- Hook ***sflt_unregister()*** to deregister the events from EMS
- User creates a fake source and destination of transition
  - Need to get cookie before create connection
- Use ***scapy*** to create packet with custom layers (Ether / IP / TCP) & pack to ***mbuf_t***, then trigger the event.



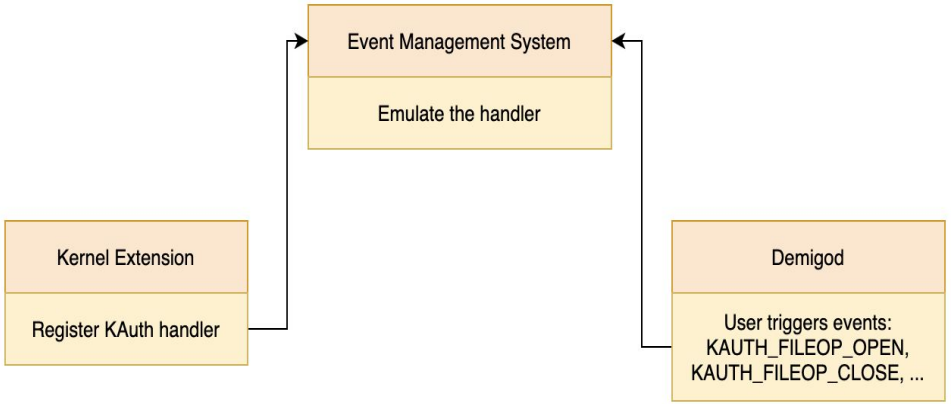
Event Management System - MAC policy

- Hook ***mac_policy_register()*** to register MAC events
  - Retrieve name from struct
  - Check if callbacks are valid, then commit to EMS
  - Update ***mac_policy_list*** symbol in kernel space
  - Trigger event ***mpo_policy_init*** & ***mpo_policy_initbsd***
- Hook ***mac_policy_unregister()*** to unregister events from EMS
- User triggers some events



Event Management System - KAuth

- Hook ***kauth_listen_scope()*** to KAuth events
  - Retrieve identifier from struct
  - Retrieve callback and commit to EMS
- Hook ***kauth_unlisten_scope()*** to unregister events from EMS
- User triggers some events ***KAUTH_FILEOP_OPEN***, ***KAUTH_FILEOP_CLOSE***, ...

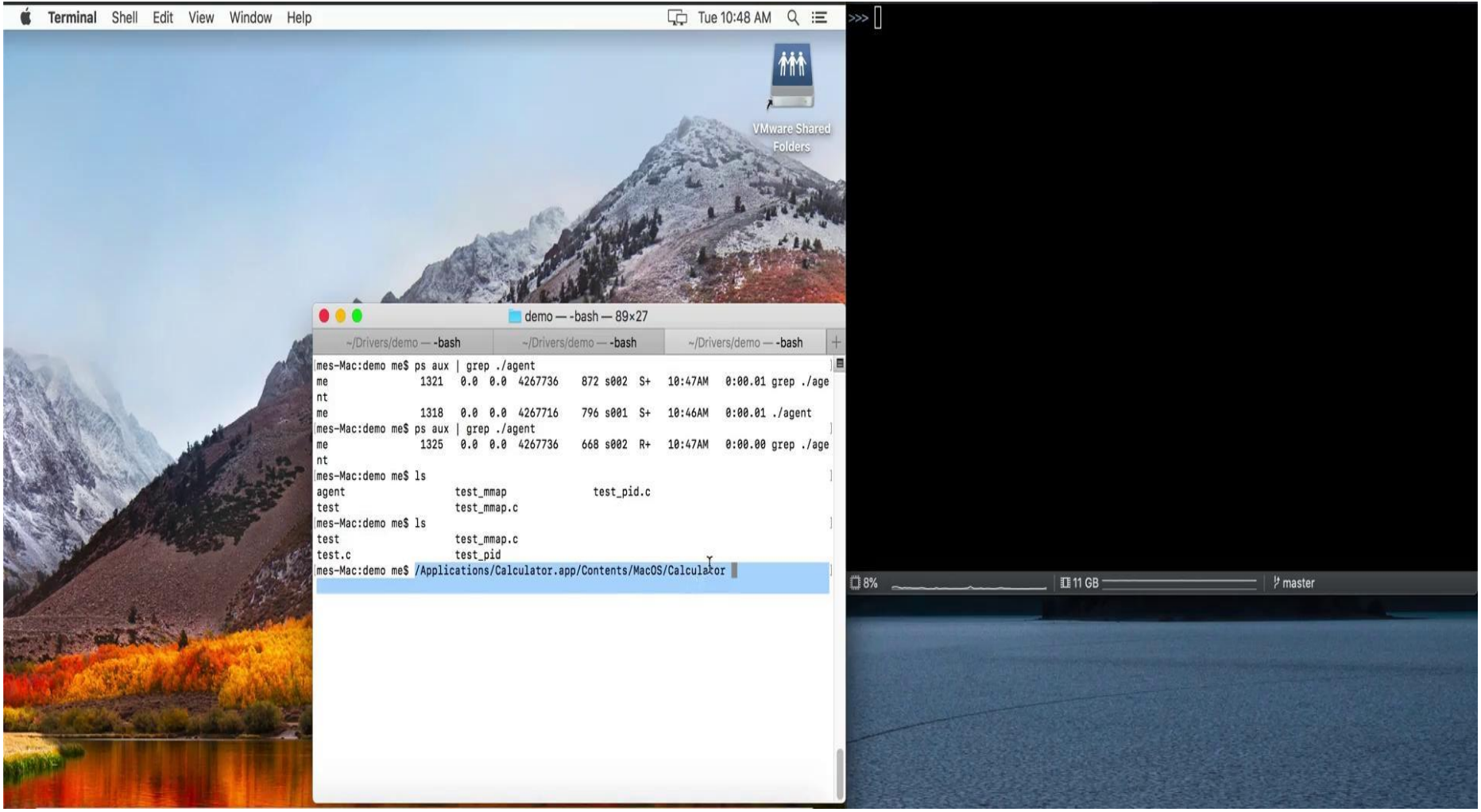


Rubilyn Rootkit

- Released on full disclosure in 2012
  - Grant root access to pid
  - Hide files / folders by hooking ***getdirentries64()***
  - Hide a process
  - Hide an user from ‘who’ / ‘w’
  - Hide a network port from netstat
  - Sysctl interface for userland control
  - Execute a binary through ICMP ping

Rubilyn Rootkit: new Generation

- Problems
  - Failed to load on MacOS High Sierra and later.
  - Hide files / folders by hooking ***getdirentries64()***
    - "ls" applications now uses ***getattrlistbulk()***
    - Interrupt Descriptor Table (IDT) is located on another page
    - Syscall Table memory is read-only
  - Execute a binary through ICMP ping → Data structure now is different
- Solution
  - Use ***saved RIP*** and ***step back*** to find kernel base
  - Scan syscall handlers to find ***sysent***
  - Modify ***cr0*** register to overwrite entries in ***sysent***
  - Re-construct ICMP packet based on new structures
- Demo: https://youtu.be/k5vfZUTX3sM


Terminal Shell Edit View Window Help
Tue 10:48 AM
VMware Shared Folders
demo — -bash — 89×27
~/Drivers/demo — -bash
~/Drivers/demo — -bash
~/Drivers/demo — -bash
mes-Mac:demo me$ ps aux | grep ./agent
me 1321 0.0 0.0 4267736 872 s002 S+ 10:47AM 0:00.01 grep ./agent
me 1318 0.0 0.0 4267716 796 s001 S+ 10:46AM 0:00.01 ./agent
mes-Mac:demo me$ ps aux | grep ./agent
me 1325 0.0 0.0 4267736 668 s002 R+ 10:47AM 0:00.00 grep ./agent
mes-Mac:demo me$ ls
agent test_mmap test_pid.c
test test_mmap.c
mes-Mac:demo me$ ls
test test_mmap.c
test.c test_pid
mes-Mac:demo me$ /Applications/Calculator.app/Contents/MacOS/Calculator
>>>
8%
11 GB
master

Emulate Rubilyn

- Load the rootkit and emulate its initial functions from ***__realmain*** symbol.
- Register sysctl handlers on Event Management System
- Build some network, process objects, ...
- User pass inputs to trigger code path of rootkit's features
- Demo: https://youtu.be/99QiLnmjBYE

```
_ipf_addv4(filter = 0xffffff7000013ec0, filter_ref = 0xffffff7000015ca0) = 0x0
********************************************************************************
[ demigod ] Get root for process:
_sysctlbyname(p = 0x500000540, uap = 0x500001788) (PASSTHRU)
_kalloc_canblock(psize = 0x7ffd099ffed8, canblock = "True", site = 0xffffff8000c9b688) = 0x5000017b8
_copyio(copy_type = 0x0, user_addr = 0x500001568, kernel_addr = 0x5000017b8, nbytes = 0x11, lencopied = 0x0, use_kernel_map = 0x0) = 0x0
_bzero(dst0 = 0x7ffd099fff28, length = 0x50) (PASSTHRU)
_sysctl_root(from_kernel = "False", string_is_canonical = "True", namestring = 0x5000017b8, namestringlen = 0x12, name = 0x7ffd099fff90, namelen = 0xc, req = 0x7ffd099ffed8) = 0x0
_sysctl_handle_int(oidp = 0xffffff7000013c90, arg1 = 0xffffff7000015c90, arg2 = 0x0, req = 0x7ffd099ffed8) (PASSTHRU)
_sysctl_io_number(req = 0x7ffd099ffed8, bigValue = 0x0, valueSize = 0x4, pValue = 0xffffff7000015c90, changed = 0x0) (PASSTHRU)
_sysctl_old_user(req = 0x7ffd099ffed8, p = 0x7ffd099ffe48, l = 0x4) (PASSTHRU)
_sysctl_new_user(req = 0x7ffd099ffed8, p = 0x7ffd099ffe48, l = 0x8) (PASSTHRU)
_copyio(copy_type = 0x0, user_addr = 0x500001580, kernel_addr = 0x7ffd099ffe48, nbytes = 0x8, lencopied = 0x0, use_kernel_map = 0x0) = 0x0
_proc_find(pid = 0x539) = 0x500001048
_lck_mtx_lock(lock = 0x5000010a0) = 0x1
_kauth_cred_setuidgid(cred = 0x5000014d8, uid = 0x0, gid = 0x0) = 0x500001818
_lck_mtx_unlock(lock = 0x5000010a0) = 0x1
_kfree_addr(addr = 0x5000017b8)
tuanit96@MacBook-Pro-2:~/Projects/demigod <master> ✘
$ python3 bh_usa_demo/macos/SuperRootkit.py --file samples/bin/macos/SuperRootkit.kext --phide 1337                    [12:07:05]
[+] Parsing kernel: qiling/examples/rootfs/x8664_macos/System/Library/Kernels/kernel.development
[+] Memory for external static symbol is created at 0xffffff700001b000 with size 0x1000
[+] Kernel loaded at 0xffffff8000200000
[+] Found entry point: 0xffffff7000007780
[+] Found exit point: 0xffffff7000007990
_printf(format = "[ SuperRootkit ] Size of proc: %lu") = 0x0
_printf(format = "[ SuperRootkit ] Kernel base = 0x%llx") = 0x0
_sysctl_register_oid(oidp = 0xffffff7000013c40)
_sysctl_register_oid(oidp = 0xffffff7000013c90)
_sysctl_register_oid(oidp = 0xffffff7000013ce0)
_sysctl_register_oid(oidp = 0xffffff7000013d30)
_sysctl_register_oid(oidp = 0xffffff7000013d80)
_sysctl_register_oid(oidp = 0xffffff7000013dd0)
_sysctl_register_oid(oidp = 0xffffff7000013e20)
_sysctl_register_oid(oidp = 0xffffff7000013e70)
_ipf_addv4(filter = 0xffffff7000013ec0, filter_ref = 0xffffff7000015ca0) = 0x0
********************************************************************************
[ demigod ] Hide process:
_sysctlbyname(p = 0x500000540, uap = 0x500001788) (PASSTHRU)
_kalloc_canblock(psize = 0x7ffd099ffed8, canblock = "True", site = 0xffffff8000c9b688) = 0x5000017b8
_copyio(copy_type = 0x0, user_addr = 0x500001568, kernel_addr = 0x5000017b8, nbytes = 0x12, lencopied = 0x0, use_kernel_map = 0x0) = 0x0
_bzero(dst0 = 0x7ffd099fff28, length = 0x50) (PASSTHRU)
_sysctl_root(from_kernel = "False", string_is_canonical = "True", namestring = 0x5000017b8, namestringlen = 0x13, name = 0x7ffd099fff90, namelen = 0xc, req = 0x7ffd099ffed8) = 0x0
_sysctl_handle_int(oidp = 0xffffff7000013ce0, arg1 = 0xffffff7000015c94, arg2 = 0x0, req = 0x7ffd099ffed8) (PASSTHRU)
_sysctl_io_number(req = 0x7ffd099ffed8, bigValue = 0x0, valueSize = 0x4, pValue = 0xffffff7000015c94, changed = 0x0) (PASSTHRU)
_sysctl_old_user(req = 0x7ffd099ffed8, p = 0x7ffd099ffe48, l = 0x4) (PASSTHRU)
_sysctl_new_user(req = 0x7ffd099ffed8, p = 0x7ffd099ffe48, l = 0x8) (PASSTHRU)
_copyio(copy_type = 0x0, user_addr = 0x500001580, kernel_addr = 0x7ffd099ffe48, nbytes = 0x8, lencopied = 0x0, use_kernel_map = 0x0) = 0x0
_proc_list_lock()
_proc_list_unlock()
_kfree_addr(addr = 0x5000017b8)
tuanit96@MacBook-Pro-2:~/Projects/demigod <master> ✘
$                                                                                                                    [12:07:21]
```

Analyze Rubilyn: Debug

- Enable gdb server on emulator
  - Use ***--gdb ":<port>"*** option
- Connect gdb remotely to gdb server
  - ***target remote <address>:<port>***
- Debug Rubilyn with gdb
  - Set breakpoints: ***b *<address>***
  - Step next, step inside: ***si, ni, ...***
  - Continue: ***c***
- Demo: https://youtu.be/vSJjF9IG6Ck

python3 (Python)

```
[+] 0xffffff7000007d84 0f 86 44 00 00 00        jbe 0xffffff7000007dce
[+] 0xffffff7000007d8a 48 8b 45 d8              mov rax, qword ptr [rbp - 0x28]
[+] 0xffffff7000007d8e 48 05 08 00 00 00        add rax, 8
[+] 0xffffff7000007d94 48 89 45 c8              mov qword ptr [rbp - 0x38], rax
[+] 0xffffff7000007d98 48 8b 7d c8              mov rdi, qword ptr [rbp - 0x38]
[+] 0xffffff7000007d9c 48 8d 35 ea 51 00 00     lea rsi, [rip + 0x51ea]
[+] 0xffffff7000007da3 ba 07 00 00 00           mov edx, 7
[+] 0xffffff7000007da8 e8 cf 33 01 00           call 0xffffff700001b17c
[+] 0xffffff700001b17c 48 83 ec 08              sub rsp, 8
[+] 0xffffff700001b180 c7 04 24 f0 cf 48 00     mov dword ptr [rsp], 0x48cff0
[+] 0xffffff700001b187 c7 44 24 04 80 ff ff ff  mov dword ptr [rsp + 4], 0xffffff80
[+] 0xffffff700001b18f c3                       ret
[+] 0xffffff800048cff0 55                       push rbp
_strncmp(s1 = "demigod", s2 = "demigod", n = 0x7) (PASSTHRU)
[+] 0xffffff800048cff1 48 89 e5                 mov rbp, rsp
[+] 0xffffff800048cff4 31 c0                    xor eax, eax
[+] 0xffffff800048cff6 48 85 d2                 test rdx, rdx
[+] 0xffffff800048cff9 75 13                    jne 0xffffff800048d00e
[+] 0xffffff800048d00e 0f be 0f                 movsx ecx, byte ptr [rdi]
[+] 0xffffff800048d011 44 0f be 06              movsx r8d, byte ptr [rsi]
[+] 0xffffff800048d015 44 38 c1                 cmp cl, r8b
[+] 0xffffff800048d018 75 06                    jne 0xffffff800048d020
[+] 0xffffff800048d01a 84 c9                    test cl, cl
[+] 0xffffff800048d01c 75 e2                    jne 0xffffff800048d000
[+] 0xffffff800048d000 48 ff ca                 dec rdx
[+] 0xffffff800048d003 48 ff c7                 inc rdi
[+] 0xffffff800048d006 48 ff c6                 inc rsi
[+] 0xffffff800048d009 48 85 d2                 test rdx, rdx
[+] 0xffffff800048d00c 74 10                    je 0xffffff800048d01e
[+] 0xffffff800048d00e 0f be 0f                 movsx ecx, byte ptr [rdi]
[+] 0xffffff800048d011 44 0f be 06              movsx r8d, byte ptr [rsi]
[+] 0xffffff800048d015 44 38 c1                 cmp cl, r8b
[+] 0xffffff800048d018 75 06                    jne 0xffffff800048d020
[+] 0xffffff800048d01a 84 c9                    test cl, cl
[+] 0xffffff800048d01c 75 e2                    jne 0xffffff800048d000
[+] 0xffffff800048d000 48 ff ca                 dec rdx
[+] 0xffffff800048d003 48 ff c7                 inc rdi
[+] 0xffffff800048d006 48 ff c6                 inc rsi
[+] 0xffffff800048d009 48 85 d2                 test rdx, rdx
[+] 0xffffff800048d00c 74 10                    je 0xffffff800048d01e
[+] 0xffffff800048d00e 0f be 0f                 movsx ecx, byte ptr [rdi]
[+] 0xffffff800048d011 44 0f be 06              movsx r8d, byte ptr [rsi]
[+] 0xffffff800048d015 44 38 c1                 cmp cl, r8b
[+] 0xffffff800048d018 75 06                    jne 0xffffff800048d020
[+] 0xffffff800048d01a 84 c9                    test cl, cl
[+] 0xffffff800048d01c 75 e2                    jne 0xffffff800048d000
[+] 0xffffff800048d000 48 ff ca                 dec rdx
[+] 0xffffff800048d003 48 ff c7                 inc rdi
[+] 0xffffff800048d006 48 ff c6                 inc rsi
[+] 0xffffff800048d009 48 85 d2                 test rdx, rdx
[+] 0xffffff800048d00c 74 10                    je 0xffffff800048d01e
[+] 0xffffff800048d00e 0f be 0f                 movsx ecx, byte ptr [rdi]
```

gdb (ssh)

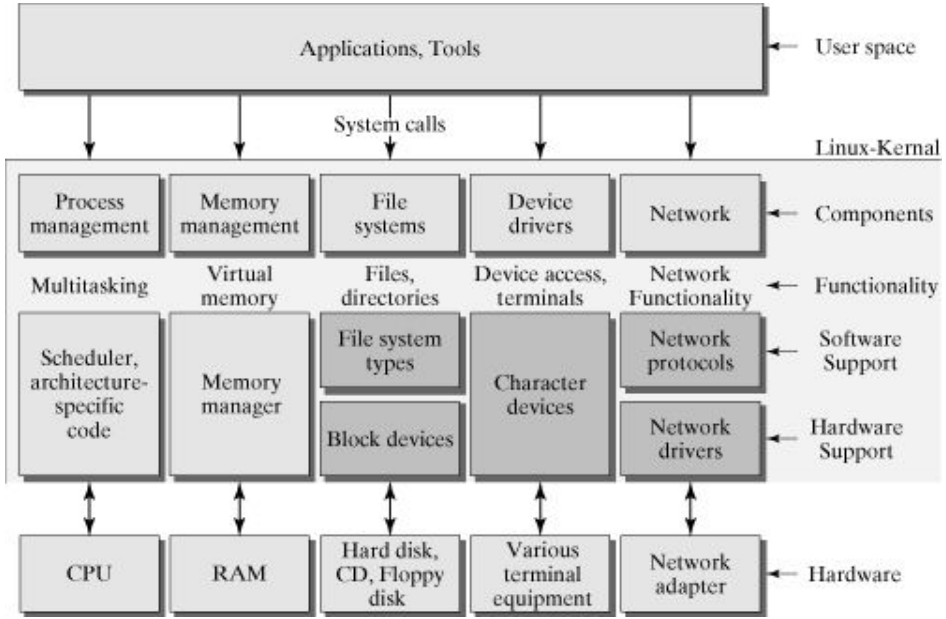
```
   0xffffff7000007780      push   %rbp
   0xffffff7000007781      mov    %rsp,%rbp
   0xffffff7000007784      sub    $0x40,%rsp
   0xffffff7000007788      mov    %rdi,-0x10(%rbp)
   0xffffff700000778c      mov    %rsi,-0x18(%rbp)
   0xffffff7000007790      mov    %rbp,%rax
   0xffffff7000007793      mov    %rax,-0x20(%rbp)
   0xffffff7000007797      movabs $0xffffff8000200000,%rax
   0xffffff70000077a1      mov    %rax,-0x28(%rbp)
   0xffffff70000077a5      lea    0x5798(%rip),%rdi        # 0xffffff700000cf44
   0xffffff70000077ac      mov    $0x3e0,%esi
   0xffffff70000077b1      mov    $0x0,%al
B+>0xffffff70000077b3      callq  0xffffff700001b140
   0xffffff70000077b8      mov    -0x28(%rbp),%rsi
   0xffffff70000077bc      lea    0x57a4(%rip),%rdi        # 0xffffff700000cf67
   0xffffff70000077c3      mov    %eax,-0x34(%rbp)
   0xffffff70000077c6      mov    $0x0,%al
   0xffffff70000077c8      callq  0xffffff700001b140
   0xffffff70000077cd      cmpq   $0x0,-0x28(%rbp)
   0xffffff70000077d2      je     0xffffff700000793e
   0xffffff70000077d8      movabs $0xffffff800084b3a0,%rax
   0xffffff70000077e2      mov    %rax,0xdc57(%rip)        # 0xffffff7000015440
   0xffffff70000077e9      movabs $0xffffff800087cbf0,%rax
   0xffffff70000077f3      mov    %rax,0xdc4e(%rip)        # 0xffffff7000015448
   0xffffff70000077fa      movabs $0xffffff800087cc10,%rax
   0xffffff7000007804      mov    %rax,0xdc45(%rip)        # 0xffffff7000015450
   0xffffff700000780b      movabs $0xffffff8000e5ff70,%rax
   0xffffff7000007815      mov    %rax,0xdc3c(%rip)        # 0xffffff7000015458
   0xffffff700000781c      movabs $0xffffff800033f200,%rax
   0xffffff7000007826      mov    %rax,0xdc33(%rip)        # 0xffffff7000015460
   0xffffff700000782d      movabs $0xffffff8000ecf920,%rax
   0xffffff7000007837      mov    %rax,0xdc2a(%rip)        # 0xffffff7000015468

remote Thread 42000.1996 In:                                   L??   PC: 0xffffff70000077b3
(gdb) si
0xffffff7000007781 in ?? ()
(gdb) si
0xffffff7000007784 in ?? ()
(gdb) si
0xffffff7000007788 in ?? ()
(gdb) si
0xffffff700000778c in ?? ()
(gdb) b *0xffffff70000077b3
Breakpoint 1 at 0xffffff70000077b3
(gdb) c
Continuing.

Breakpoint 1, 0xffffff70000077b3 in ?? ()
(gdb)
```

7% 11 GB 0.0 kB↓ 0.0 kB↑ master +

Linux

Linux Overview

- Linux kernel
  - System calls play key roles to provide services
  - Also interface with userland via IOCTL
  - Various callbacks indirectly triggered by userland
    - File operations, etc
  - Kernel module in LKM: .KO
    - ELF format, but all in sections
      - No program headers
- Goal: emulate kernel rootkit in .KO driver



Load LKM File (.KO)

- ELF loader for .KO file
  - Parse all sections of ELF file and map all sections into emulator
  - Resolve external API to provide our own implementation.
  - Relocate external functions and symbols (data)
- Setup CPU context
  - GS segment (current_task)
- Initialize Linux syscall table
  - Write address of our own syscall implementation into syscall table
  - Hook syscalls to execute our syscall implementation
- Locate initialization function
  - **init_module** symbol in SYMTAB section

Linux Internal Structures

- task_struct
  - Information for userland tasks.
- file_operations
  - Callback to file operations (read/write/open/close/etc)
- module
  - Information about LKM, with links to all modules
- user_namespace
  - Access to user credentials
- file_struct
  - Files accessed by process
- linux_direntXX
  - Directory information

Emulating LKM Initialization

- Setup optional arguments for **init_module** entry
  - Ignored when LKM is loaded without arguments
  - Similar to input arguments for **sys_init_module**
    - Buffer point to LKM arguments & total length
- Emulate from **init_module**
  - Stop emulation when reaching RET

Hook Linux Kernel APIs

- Emulate some APIs necessary for LKM to work
  - __fentry__
  - __x86_indirect_thunk_rax
  - __stack_chk_fail
  - _copy_to_user, _copy_from_user
  - kmalloc, kfree
  - Device management
    - misc_register, misc_deregister
    - register_chrdev
    - device_create
    -
  - Syscalls
    - sys_read, sys_write, sys_open
    - prepare_creds, commit_creds
    - etc

```
# void IofCompleteRequest(
#   PIRP  Irp,
#   CCHAR PriorityBoost
# );
@winapi(cc=STDCALL,
        params={
            "Irp": POINTER,
            "PriorityBoost": CCHAR
            })
def hook_IofCompleteRequest(ql, address, params):
    return None
```

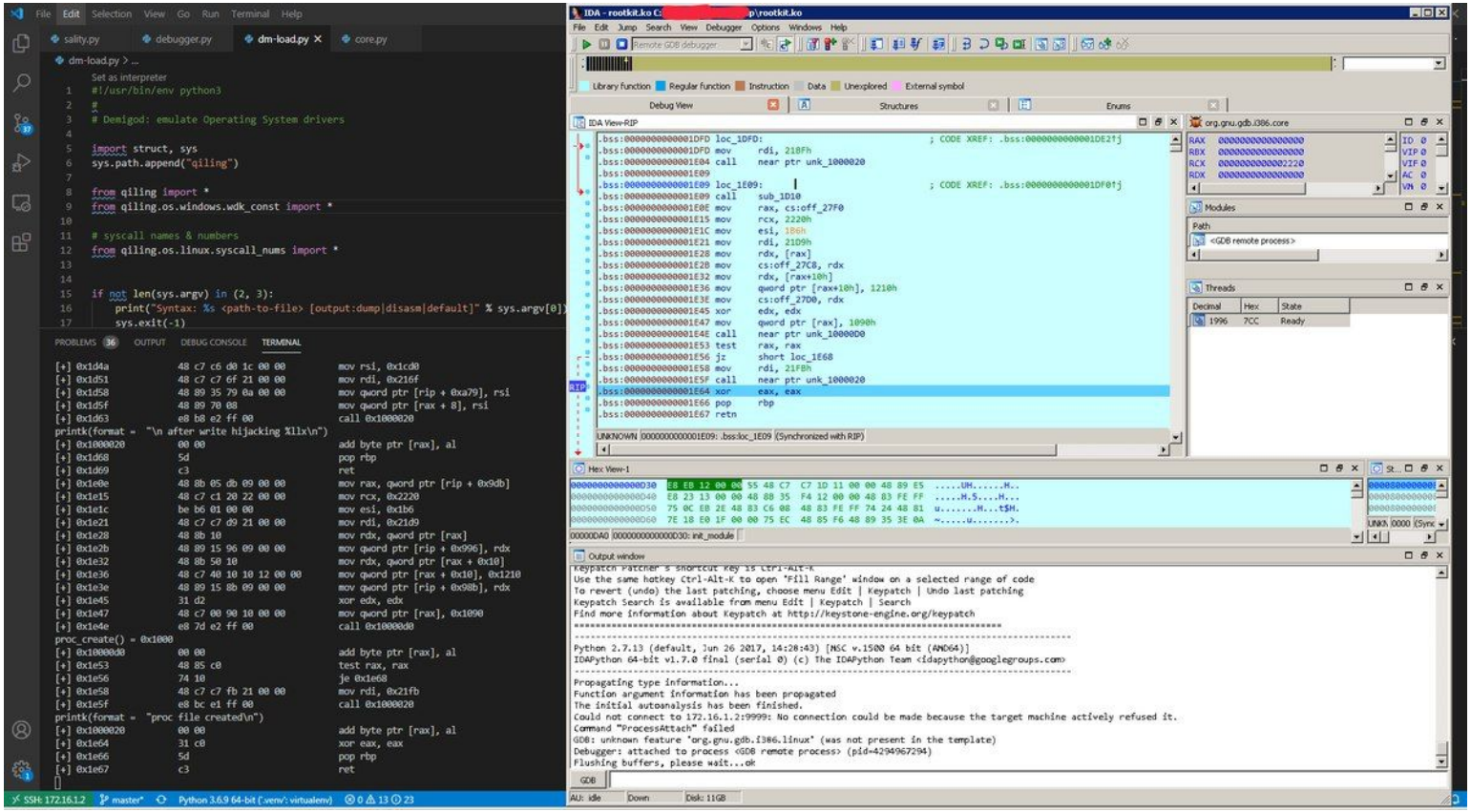
Emulate Code Path

- Extract syscalls or callbacks from LKM loading phase
- Pass input arguments provided by users via registers
  - SYSV ABI
- Userspace communicates with LKM via syscalls or IOCTL
  - READ, WRITE, OPEN syscalls
  - IOCTL syscalls
- Other special interfaces
  - **proc_file_fops** for /proc access
    - Emulate **.read**, **.write**
  - **nf_hook_ops** for netfilter access (monitor network packets).
    - Emulate **.hook**
  - **notifier_block** to access to keystrokes (keylogger)
    - Emulate **.notifier_call**

Demo: Linux m0hamed Rootkit

- Load in as LKM (rootkit.ko)
- A proof-of-concept kernel rootkit, with typical rootkit behaviors
  - Hide kernel modules (and itself)
  - Hide files
  - Hide network ports
  - Grant root access to current process (backdoor)
  - Overwrite syscalls: READ, WRITE, OPEN, CLOSE
  - Interface with userland customized file in /proc
- Code available
  - https://github.com/m0hamed/lkm-rootkit
- Demo
  - Load rootkit & initialize it
  - Retrieve address of rootkit's WRITE syscall
  - Emulate rootkit WRITE syscall directly, with selected input
  - Trace execution with debugger of IDA

Demo: Debug m0hamed Rootkit



More Applications of Demigod

- Debug kernel rootkits in ring 3, within safety sandbox
  - IDA Pro, GDB client, Binary Ninja etc
- Auto-unpacker for kernel code
  - Trace kernel code to detect the time code is unpacked, then dump & auto rebuild binary
- IOCTL analyzer
  - Auto-discover IOCTL code & callbacks using random inputs
- Auto-summarize rootkit behaviors to produce high-level reports
  - Auto-discover different rootkit code paths, and summarize its behavior according to rules
  - Demigod enable save execution on checkpoints, taking snapshots, etc
- Some other exciting private ideas ;-)

Status & Future work

- Supported Windows, MacOS & Linux kernel rootkits
- Still need to test & improve to support more rootkits
  - More kernel API
  - More syscalls
  - More kernel subsystems
  - Emulating enough features of real OS is hard & long-term project
- Support more platforms?
  - iOS, Android, etc
- To be merged into Qiling emulator soon
  - Watch out **https://groundx.io/demigod** for more announcement

Conclusions

- Demigod is a framework to emulate kernel rootkits
  - Enable analyzing ring 0 code from ring 3, in safety sandbox
    - Monitor, trace, debug, etc
  - Built on top of the excellent **Qiling.io** emulator
  - Cross-platforms: Windows, MacOS & Linux
  - Cross-architectures: X86, Arm, Arm64, Mips
- Enable advanced binary analysis of kernel code
  - Python-based framework for tool building
  - Instrumentation at various levels
- Release to be announced on groundx.io/demigod
  - To be merged into **Qiling.io** very soon

Q & A

Demigod: The Art of Emulating Kernel Rootkits

NGUYEN Anh Quynh <aquynh @ gmail.com>

NGUYEN Hong Quang <quangnh89 @ gmail.com>

DO Minh Tuan <tuanit96 @ gmail.com>