
Tencent 腾讯
腾讯安全
CSS
互联网安全领袖峰会
Cyber Security Summit
如何批量挖掘macOS/iOS内核信息泄漏漏洞
王伟波

议题内容

- 业界挖掘信息泄露漏洞的工具
- xnu内核信息泄漏的几种方式
- 如何实现批量化发掘?
- 漏洞分析
- 目前存在的问题

业界挖掘信息泄露漏洞的工具

fuzz信息泄漏的工具对比

Name	指令执行方式	指令执行速度	平台
Bochspwn Reloaded	翻译执行	慢	Windows/Linux
Digtool	硬件虚拟化	快	Windows
N/A	N/A	N/A	macOS

- Bochs is a full IA-32 and AMD64 PC emulator.
  - CPU plus all basic peripherals, i.e. a whole emulated computer.
- Written in C++.
- Supports all latest CPUs and their advanced features
  - SSE2, SSE3, SSSE3, SSE4, SSE5, AVX, both SVM & VT-x etc.
- Correctly hosts all common operating systems.
- Provides extensive instrumentation API.
- A-W-E-S-O-M-E!

Bochs instrumentation callbacks

```
BX_INSTR_INIT_ENV                BX_INSTR_INTERRUPT
BX_INSTR_EXIT_ENV                BX_INSTR_HWINTERRUPT
BX_INSTR_INITIALIZE              BX_INSTR_CLFLUSH
BX_INSTR_EXIT                    BX_INSTR_CACHE_CNTRL
BX_INSTR_RESET                   BX_INSTR_TLB_CNTRL
BX_INSTR_HLT                     BX_INSTR_PREFETCH_HINT
BX_INSTR_MWAIT                   BX_INSTR_BEFORE_EXECUTION
BX_INSTR_DEBUG_PROMPT            BX_INSTR_AFTER_EXECUTION
BX_INSTR_DEBUG_CMD               BX_INSTR_REPEAT_ITERATION
BX_INSTR_CNEAR_BRANCH_TAKEN      BX_INSTR_LIN_ACCESS
BX_INSTR_CNEAR_BRANCH_NOT_TAKEN  BX_INSTR_PHY_ACCESS
BX_INSTR_UCNEAR_BRANCH           BX_INSTR_INP
BX_INSTR_FAR_BRANCH              BX_INSTR_INP2
BX_INSTR_OPCODE                  BX_INSTR_OUTP
BX_INSTR_EXCEPTION               BX_INSTR_WRMSR
```

Software emulators (bochs)

Pros

- Full access to the CPU logic
  - including the ability to change
  - 100% control over the execution environment.
- Ease of development.
- Ease of debugging.

Cons

- Extremely, painfully slow.
- Even slower with additional instrumentation running.
- Limited to virtual (emulated) hardware.

BochsPwn

栈内存:
1.监测每一次创建内核栈帧，注册回调
2.再回调里污染栈内存
3.当写数据到用户态的时候检测是否存在污染特征。

Reference:
https://j00ru.vexillium.org/papers/2018/bochspwn_reloaded.pdf

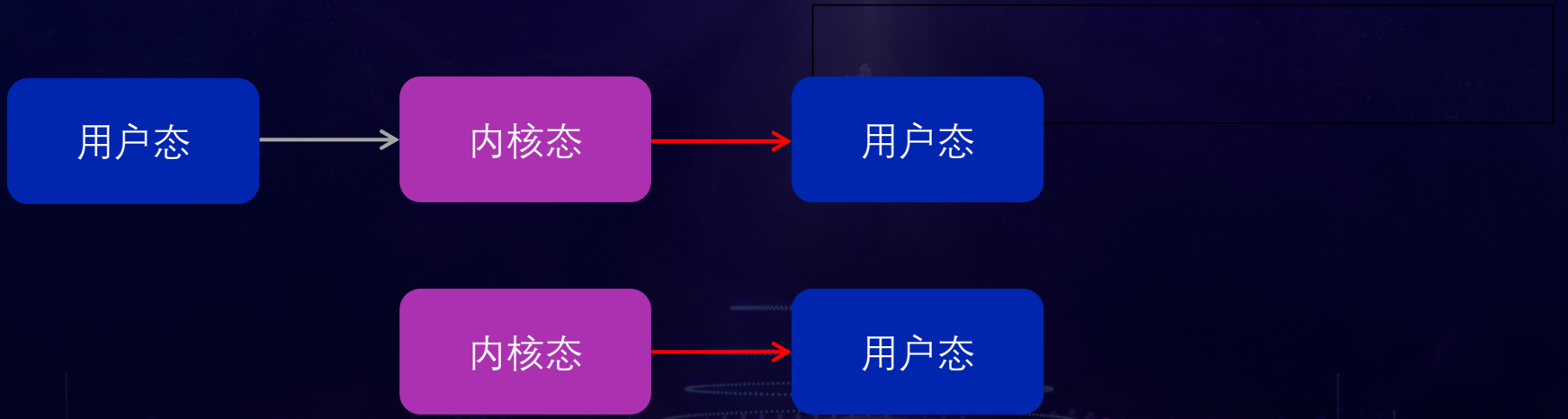
堆内存:
1.监测每一次堆内存的申请，
2.当调用free时，清空污染特征

```
1   void bx_instr_before_execution(CPU cpu, instruction i) {
2     if (!cpu.protected_mode ||
3         !os::is_kernel_address(cpu.eip) ||
4         !os::is_kernel_address(cpu.esp)) {
5       return;
6     }
7
8     if (i.opcode == SUB || i.opcode == ADD || i.opcode == AND) {
9       if (i.op[0] == ESP) {
10        globals::esp_changed = true;
11        globals::esp_value = cpu.esp;
12      }
13    }
14  }
15
16  void bx_instr_after_execution(CPU cpu, instruction i) {
17    if (globals::esp_changed && cpu.esp < globals::esp_value) {
18      set_taint(/*from=  */cpu.esp,
19                /*to=    */globals::esp_value - 1,
20                /*origin=*/cpu.eip);
21    }
22
23    globals::esp_changed = false;
24  }
```

Listing 7: Pseudo-code of the stack tainting logic

xnu内核信息泄漏的几种方式

- 栈信息泄漏
- 堆信息泄漏


用户态
内核态
用户态
内核态
用户态

写数据到用户态

- IPC：
  包括 XPC, NSXPC, mach_msg 等
- IOKit:
  IOConnectCallMethod/IOConnectMapMemory/IODataQueue
- IOCTLs:
  一种特殊的系统调用
- SYSCALL：
  *NIX 系统调用
- kernel notifications & kernel control:
  内核主动的发送数据到用户态

通过copyout/copyoutphys最终完成写数据到用户层

```
case COPYIN:
        error = _bcopy((const void *) user_addr,
            kernel_addr,
            nbytes);
    break;

case COPYOUT:
        error = _bcopy(kernel_addr,
            (void *) user_addr,
            nbytes);
    break;

case COPYINPHYS:
        error = _bcopy((const void *) user_addr,
            PHYSMAP_PTOV(kernel_addr),
            nbytes);
    break;

case COPYOUTPHYS:
        error = _bcopy((const void *) PHYSMAP_PTOV(kernel_addr),
            (void *) user_addr,
            nbytes);
    break;
```

case 1：对象未初始化或者部分初始化

```
void func1(uint8_t* buffer)
{
    struct Vul_obj{
    uint64_t a;
    uint64_t b;
    uint64_t c;
    };
    struct Vul_obj vo;
    vo.a=1;
    vo.c=2;
   copyout(&vo,buffer,sizeof(struct Vul_ob));
}
```

case 2：类型转换导致变量部分未初始化

```
void func2(uint64_t* outptr)
{
    uint32_t value=1;
    *(uint32_t*)outptr=value;
}
```

case 3：字符串数组未初始化

```
void func3(char* userptr)
{
    char strbuf[0x20];
    strcpy(strbuf,"css2019");
    copyout(strbuf,userptr,sizeof(strbuf));
}
```

case 4：编译器结构体8字节对齐

```
struct Vul
{
    uint32_t a;
    uint64_t b;
};

struct  Vul v;
v->a=0;
v->b=0x4141414141414141;
```

a	00 00 00 00
	?? ?? ?? ??
b	41 41 41 41
	41 41 41 41

如何来批量化发现此类漏洞?

怎样在macOS平台上实现?
--QemuPwn的实现

基本思路:

1.污染内核申请的内存(栈，堆内存)
2.在返回用户空间的时候，检查是否存在污染数据
3.若有，记录下当前内核调用栈信息，记录此时用户态程序调用栈信息，等其他信息
4.根据记录下来的信息，还原漏洞触发过程，完成poc

QemuPWN:

1.QEMU: https://github.com/qemu/qemu
2.OSX-KVM: https://github.com/kholia/OSX-KVM

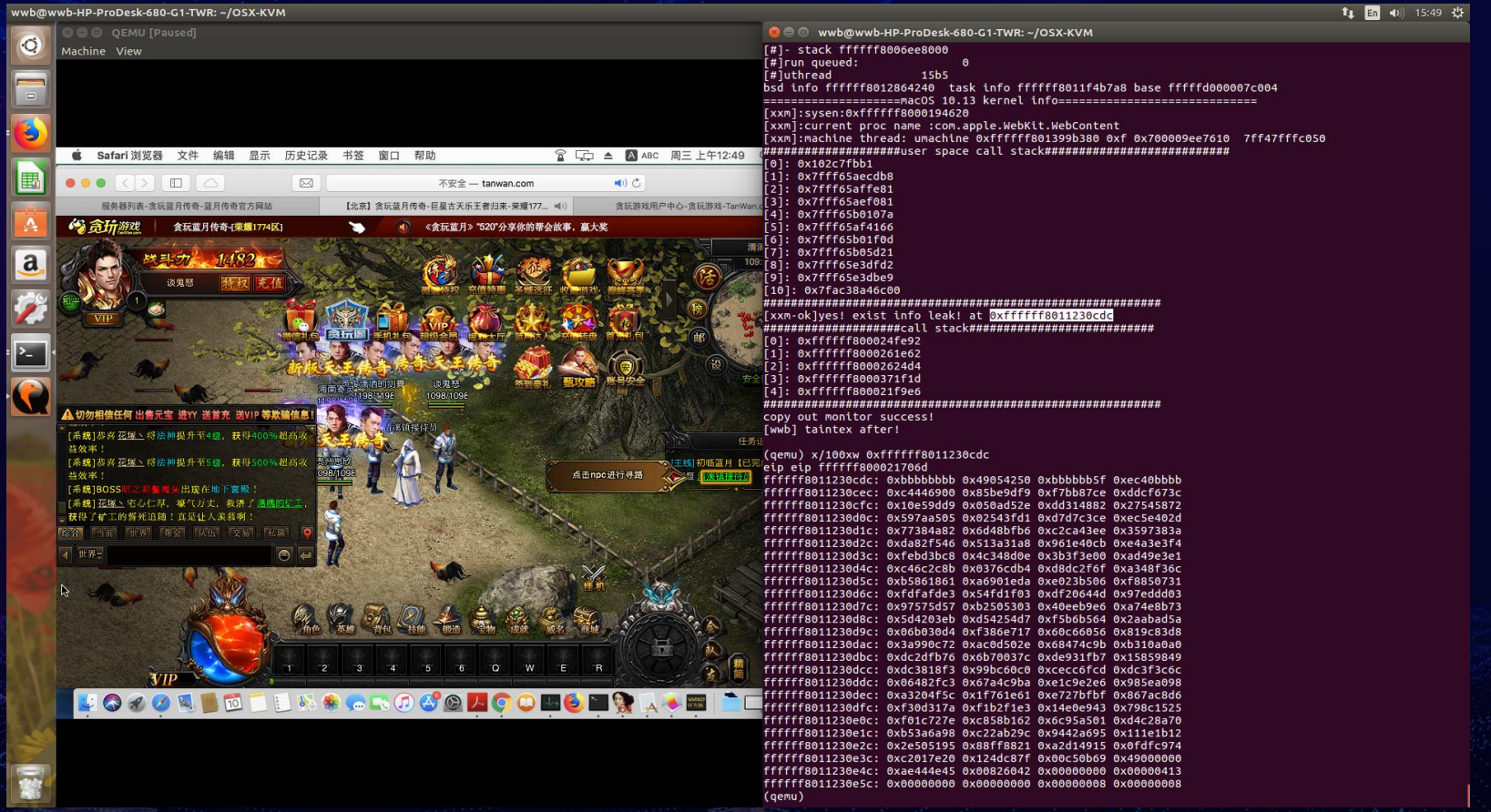
优点:

- 可以监控内存读写情况
- linux平台即可运行，可以批量化运行实例
- 可以很方便的获取任何想获取的cpu,内存等信息
- 可以直接调试内核

QemuPWN:


cmd_start_taint
污染内核栈空间
污染堆内存申请
QEMU-macOS
copyout 检查
调试界面
日志输出

- 玩一把游戏就能发现一个0day
- 甚至开机就能发现几个0day ☺



QEMU:

- Human Monitor Interface:
  The HMP is the simple interactive monitor on QEMU, designed primarily for debugging and simple human use.

- 利用HMP界面完成所需要的功能
- 利用cpu_memory_rw_debug写入hook代码，实现内存污染，污染检查

```
    ETEXI

        {
            .name       = "rm",
            .args_type  = "addr:l",
            .params     = "addr ",
            .help       = "rm bp",
            .cmd        = hmp_rmbp,
        },

    STEXI
```

- 获取xnu内核栈信息:

```
mov    rax,QWORD PTR gs:0x0
mov    rcx,QWORD PTR [rax+0x28]
mov    rdx,QWORD PTR [rax+0x30]
```

- 实现类似windows 内核函数 IoGetStackLimits的功能

```
typedef struct cpu_data
{
    struct pal_cpu_data cpu_pal_data;       /* PAL-specific data */
#define                 cpu_pd cpu_pal_data /* convenience alias */
    struct cpu_data     *cpu_this;          /* pointer to myself */
    thread_t            cpu_active_thread;
    thread_t            cpu_nthread;
    volatile int            cpu_preemption_level;
    int             cpu_number;         /* Logical CPU */
    void                *cpu_int_state;     /* interrupt state */
    vm_offset_t         cpu_active_stack;   /* kernel stack base */
    vm_offset_t         cpu_kernel_stack;   /* kernel stack top */
    vm_offset_t         cpu_int_stack_top;
    int             cpu_interrupt_level;
    volatile int            cpu_signals;        /* IPI events */
    volatile int            cpu_prior_signals;  /* Last set of events,
                                                 * debugging
                                                 */
```

如何获取系统运行时信息?

通过解析一下结构，获取进程名,内核堆栈，线程信息，调用栈信息等

GDT(gs:0)
\\ ===>cpu_data
+8 \\ ===>thread
+ 824\\===>task
+912 \\===>bsd_info(struct proc)

macOS 10.13

栈污染过程:

hook系统调用最开的地方(hndl_syscall函数里)

- mach_scall64: Mach陷阱,直接调用XNU的mach核心接口
- unix_scall64: POSIX/BSD 系统调用,XNU的BSD API接口
- mdep_scall64: 机器相关的调用
- diag_scall64:诊断调用，用于底层的内存诊断

```
        mov     eax, [r15+88h]
        mov     edx, eax
        and     edx, 0FF000000h
        cmp     edx, 1000000h
        jz      _hndl_mach_scall64
        cmp     edx, 2000000h
        jz      _hndl_unix_scall64
        cmp     edx, 3000000h
        jz      _hndl_mdep_scall64
        cmp     edx, 4000000h
        jz      _hndl_diag_scall64
        sti
        mov     rdi, 7
```

堆内存污染:

在函数 kalloc_canblock函数返回之前执行内存污染操作

栈污染代码:

```
.data:00000000 418b8788000000      mov eax,DWORD PTR [r15+0x88]                ;
.data:00000007 3d00000c01          cmp eax,0x10c0000
.data:0000000c 7454                je loc_00000062
.data:0000000e 3d01000c01          cmp eax,0x10c0001
.data:00000013 744d                je loc_00000062
.data:00000015 51                  push rcx
.data:00000016 4150                push r8
.data:00000018 4d31c0              xor r8,r8                                  ;
.data:0000001b 65488b042500000000  mov rax,QWORD PTR gs:0x0                   ;
.data:00000024 488b4828            mov rcx,QWORD PTR [rax+0x28]
.data:00000028 488b5030            mov rdx,QWORD PTR [rax+0x30]
.data:0000002c 48b800000000ffffff7fmovabs rax,0xffffff7f00000000
.data:00000036 4889ce              mov rsi,rcx
.data:00000039 4883ea10            sub rdx,0x10
.data:0000003d 4839c1              cmp rcx,rax
.data:00000040 7c1f                jl loc_00000061
.data:00000042 4839c2              cmp rdx,rax
.data:00000045 7c1a                jl loc_00000061
.data:00000047 4839ca              cmp rdx,rcx
.data:0000004a 7c15                jl loc_00000061
.data:0000004c 4829ca              sub rdx,rcx
.data:0000004f 4989d0              mov r8,rdx
.data:00000052 4831d2              xor rdx,rdx
.data:00000055
.data:00000055                   loc_00000055:
.data:00000055 c60411bb            mov BYTE PTR [rcx+rdx*1],0xbb
.data:00000059 48ffc2              inc rdx
.data:0000005c 4939d0              cmp r8,rdx
.data:0000005f 7ff4                jg loc_00000055
.data:00000061
.data:00000061                   loc_00000061:
.data:00000061 90                  nop
.data:00000062
.data:00000062                   loc_00000062:
.data:00000062 4158                pop r8
.data:00000064 59                  pop rcx
.data:00000065 418b8788000000      mov eax,DWORD PTR [r15+0x88]
.data:0000006c 89c2                mov edx,eax
.data:0000006e 81e2000000ff        and edx,0xff000000
.data:00000074 48be58f9210080ffffffmovabs rsi,0xffffff800021f958
.data:0000007e ffe6                jmp rsi
```

获取栈空间

污染循环

copyout检查:

1.过滤不想要的进程名
2.检查目的地址是否属于用户空间
3.检测是否存在被污染的字节序列(0xbbbbbbbb)
4.存在则执行int3
5.qemu检测到异常，跳出kvm执行流程，记录当前task信息，寄存器信息，内核调用栈，用户态调用栈信息等到日志中，如果是调试模式，则暂停执行，等待调试.
6.恢复cpu状态，继续返回步骤一执行

```
.data:00000000 654c8b342500000000 mov r14,QWORD PTR gs:0x0
.data:00000009 4d8b7608           mov r14,QWORD PTR [r14+0x8]
.data:0000000d 4d8bb638030000     mov r14,QWORD PTR [r14+0x338]
.data:00000014 4d8bb690030000     mov r14,QWORD PTR [r14+0x390]
.data:0000001b 458bb6f1020000     mov r14d,DWORD PTR [r14+0x2f1]
.data:00000022 4181fe616d6669     cmp r14d,0x69666d61
.data:00000029 7447               je loc_00000072
.data:0000002b 4181fe7379736d     cmp r14d,0x6d737973
.data:00000032 743e               je loc_00000072
.data:00000034 4981ff00400000     cmp r15,0x4000
.data:0000003b 7735               ja loc_00000072
.data:0000003d 4983ff04           cmp r15,0x4
.data:00000041 722f               jb loc_00000072
.data:00000043 4983ef04           sub r15,0x4
.data:00000047 49be00000007ffffff movabs r14,0xffffff7f00000000
.data:00000051 4d39f0             cmp r8,r14
.data:00000054 7218               jb loc_0000006e
.data:00000056 4d31f6             xor r14,r14
.data:00000059 eb08               jmp loc_00000063
.data:0000005b
.data:0000005b                 loc_0000005b:
.data:0000005b 49ffc6             inc r14
.data:0000005e 4d39fe             cmp r14,r15
.data:00000061 730b               jae loc_0000006e
.data:00000063
.data:00000063                 loc_00000063:
.data:00000063 43813c30bbbbbbbb   cmp DWORD PTR [r8+r14*1],0xbbbbbbbb
.data:0000006b 75ee               jne loc_0000005b
.data:0000006d cc                 int3
.data:0000006e
.data:0000006e                 loc_0000006e:
.data:0000006e 4983c704           add r15,0x4
.data:00000072
.data:00000072                 loc_00000072:
.data:00000072 49beb7e5370080ffffff movabs r14,0xffffff800037e5b7
.data:0000007c 41ffe6             jmp r14
```

查找进程名

循环搜索污染特征

- kvm_cpu_exec执行客户机代码，当有异常/中断发生时，跳出正常流出，qemu来接管处理异常
- getinfoleak_info：获取信息泄漏详细信息
- vm_start/qmp_cont 恢复cpu状态，继续执行

```
    cpu->created = true;
    qemu_cond_signal(&qemu_cpu_cond);
        do {
        if (cpu_can_run(cpu)) {
            r = kvm_cpu_exec(cpu);
            if (r == EXCP_DEBUG) {

                printf("[wwb]EXCP_DEBUG is called\n");
                cpu_handle_guest_debug(cpu);
                getinfoleak_info(cpu);
                printf("[wwb] taintex after!\n");
                Error *err = NULL;
                vm_start();
                qmp_cont(&err);
                //hmp_handle_error(mon, &err);

              //  mon_get_cpu();
             //  kvm_remove_all_breakpoints(cpu);
            }
            else if(r== EXCP_HALTED)
            {
                print_callstack(cpu);
                printf("[wwb]THE OS is halt!\n");
            }
            else
            {
               // printf("[wwb] other exception %x\n",r);

            }
        }

        qemu_wait_io_event(cpu);
    } while (!cpu->unplug || cpu_can_run(cpu));
```

qemu_kvm_cpu_thread_fn

漏洞分析

1.CVE—2019-8540:类型转换高位未初始化，导致xnu内核栈信息泄漏

```
1644  IOReturn IOUserClient::exportObjectToClient(task_t task,
1645              OSObject *obj, io_object_t *clientObj)
1646  {
1647      mach_port_name_t    name;
1648
1649      name = IOMachPort::makeSendRightForTask( task, obj, IKOT_IOKIT_OBJECT );
1650
1651      *(mach_port_name_t *)clientObj = name;
1652
1653      if (obj) obj->release();
1654
1655      return kIOReturnSuccess;
1656  }
1657
```

2.CVE—2019-6207: xnu堆内存未初始化漏洞

- sysctl_dumpentry函数，用来dump内核路由表的
- 在初始rt_msghdr对象的时候，忘记了初始化rtm_inits
- 这个漏洞可以用来泄漏 mach port address

ref:
http://iosre.com/t/cve-2019-6207-port-address-low-4-bytes/15029

```
struct rt_msghdr {
    u_short rtm_msglen; /* to skip over non-understood
    u_char  rtm_version;    /* future binary compatibi
    u_char  rtm_type;   /* message type */
    u_short rtm_index;  /* index for associated ifp */
    int rtm_flags;  /* flags, incl. kern & message, e.
    int rtm_addrs;  /* bitmask identifying sockaddrs i
    pid_t   rtm_pid;    /* identify sender */
    int rtm_seq;    /* for sender to identify action *
    int rtm_errno;  /* why failed */
    int rtm_use;    /* from rtentry */
    u_int32_t rtm_inits;    /* which metrics we are in
    struct rt_metrics rtm_rmx; /* metrics themselves *
};
```

3.CVE—2019-8627

ALF.kext:
com.apple.nke.applicationfirewall

- 主动泄漏内核地址到用户态😂
- 通过注册用户空间内核通知事件，即可捕获泄漏信息

```
v5 = (__int64 *)_MALLOC(v4, 80LL, 4LL);
sockwall_lock(v4);
if ( !v5 )
{
  printf("AFL: error, cannot allocate updaterule message\n", 80LL);
  return 12LL;
}
*(_BYTE *)(a1 + 268) |= 0x80u;
bzero(v5, v4);
*(_WORD *)v5 = v3;
*((_WORD *)v5 + 1) = 8;
*((_DWORD *)v5 + 1) = *(_DWORD *)(a1 + 16);
v5[4] = a1;
v5[3] = (__int64)v5;
v5[5] = *(_QWORD *)(a1 + 288);
if ( v2 )
{
  *((_BYTE *)v5 + 8) |= 0x10u;
  __strlcpy_chk(v5 + 6, *(_QWORD *)(a1 + 240), 2048LL, 2048LL);
  v5[262] = 0LL;
  v6 = off_8048;
  v5[263] = (__int64)off_8048;
  *v6 = (__int64)v5;
  off_8048 = v5 + 262;
  v7 = v5;
}
else
{
  *(__int64 *)((char *)v5 + 52) = 0LL;        // here
  v9 = off_8058;
  *(__int64 *)((char *)v5 + 60) = (__int64)off_8058;
  *v9 = (__int64)v5;
  v7 = v5;
  off_8058 = (__int64 *)((char *)v5 + 52);
}
v10 = *((unsigned __int16 *)v7 + 8);
v11 = *((unsigned __int16 *)v7 + 9);
v12 = *((unsigned __int16 *)v7 + 10);
*((_BYTE *)v7 + 48);
fwlog(
  256,
  6,
  "sockwall_cntl_updaterules: pid=%d flag=0x%x perm %04x %04x %04x %04x %04x  %s",
  *((unsigned int *)v7 + 1),
  *((unsigned int *)v7 + 2),
  *((unsigned __int16 *)v7 + 6),
  *((_WORD *)v7 + 7));
if ( !sws_head )
  return 2LL;
result = ctl_enqueuedata(*(_QWORD *)(sws_head + 16), *(unsigned int *)(sws_head + 24), v5, v3, 2LL);
if ( !(_WORD)result )
```

修补代码:

- 在进程连接驱动的时候， 加入”com.apple.private.alf” 的entitlement检查

```
__int64 __usercall sockwall_cntl_connect@<rax>(__int64 a1@<rdi>, __int64 a2
{
  __int64 *v4; // r12
  __int64 v5; // r15
  unsigned int v6; // eax
  unsigned int v7; // ebx
  unsigned int v8; // eax
  const char *v9; // rdi
  __int64 v10; // rax
  __int64 v11; // r14
  __int64 **v12; // rax
  __int64 v14; // rax
  __int64 v15; // [rsp-38h] [rbp-38h]

  v15 = a4;
  v4 = a3;
  v5 = a1;
  v6 = checkentitlement(a1);
  if ( !v6 )
  {
    if ( *(_DWORD *)(a2 + 8) == 666 )
```

```
1  __int64 __fastcall checkentitlement(__int64 a1)
2  {
3    unsigned int v1; // ebx
4    __int64 v2; // rax
5
6    v1 = 0;
7    if ( current_task(a1) )
8    {
9      v2 = current_task(a1);
10     if ( !(unsigned int)IOTaskHasEntitlement(v2, "com.apple.private.alf") )
11     {
12       printf("ALF: client has NO entitlement");
13       v1 = 1;
14     }
15   }
16   return v1;
17 }
```

目前存在的问题

目前存在的问题:

1.系统调用参数没有变异
2.符号没有写入日志，导致分析需要对照ida
3.不能检测double fetch类型漏洞
4.qemu环境，有些内核扩展并没有加载
5.没有覆盖所有数据交互的过程


互联网安全领袖峰会
Cyber Security Summit
Tencent 腾讯 | 腾讯安全
THANKS!