2019 LONDON 2 – 4 October 2019

PLAY FUZZING MACHINE – HUNTING IOS/MACOS KERNEL VULNERABILITIES AUTOMATICALLY AND SMARTLY

Lilang Wu & Moony Li
Trend Micro, China

{574407955, 411527096}@qq.com

ABSTRACT

As we all know, *Apple*'s *iOS* and *MacOS* systems have gained much popularity with the huge success of the *iPhone* and the *MacPro*. System security vulnerabilities in *iOS* and *MacOS* have been developed and abused by hackers, and have also begun to attract more attention from security researchers.

The more you know about your enemy, the easier it is to defeat him. But how? Since *iOS 10*, *Apple* has released the unpacked/decrypted kernel cache (*.ipsw), but the system source code, in particular the kernel and driver part, remain close-sourced. What is more, symbol info in the binary (kernel cache) has been greatly removed, which makes reverse engineering more difficult.

A challenge means a chance. The truth in security research is that the more attack interface you expose, and the more implementation you do, the greater the probability of finding a zero-day vulnerability. The relatively good news is that, in every *iOS*/*MacOS* system update or new hardware release (e.g. the touch bar in *MacPro*), there is always a lot of change in interface and implementation code (e.g. more selectors exposed in the driver service via IOUserClient).

Hence, we plan to expose the typical workflow and thinking in order to explore and analyse the new (kernel) attack interface using reverse engineering and dynamic analysis. We will not only share the relative tool chain used to explore the attack interfaces but also the public kernel vulnerability finding system that is based on enhanced passive fuzzing. Finally, we will describe some of the vulnerabilities we have found using these methodologies.

1. SOLUTION OVERVIEW

The basic work flow can be separated into the three parts where the architecture or implementation of the *iOS*/*OSX* system kernel has potentially changed: identification of attack interfaces using reverse analysis (static analysis), use of dynamic analysis for the kernel attack interface, and use of a passive kernel fuzzing system, accordingly.

Typically, the occurrence of a change in kernel code would coincide with an *iOS*/*OSX* version being released following a public security bulletin(s), *Apple* adding new hardware equipment (e.g. touch bar equipment for *MacOS*, A12 CPU update for *iPhone*, new graphic I/O devices, etc.), and so on.

After obtaining the whole kernel cache binary, the first step would normally be to identify the modification or new attack interface using reverse engineering. Since we focus on the kernel part in

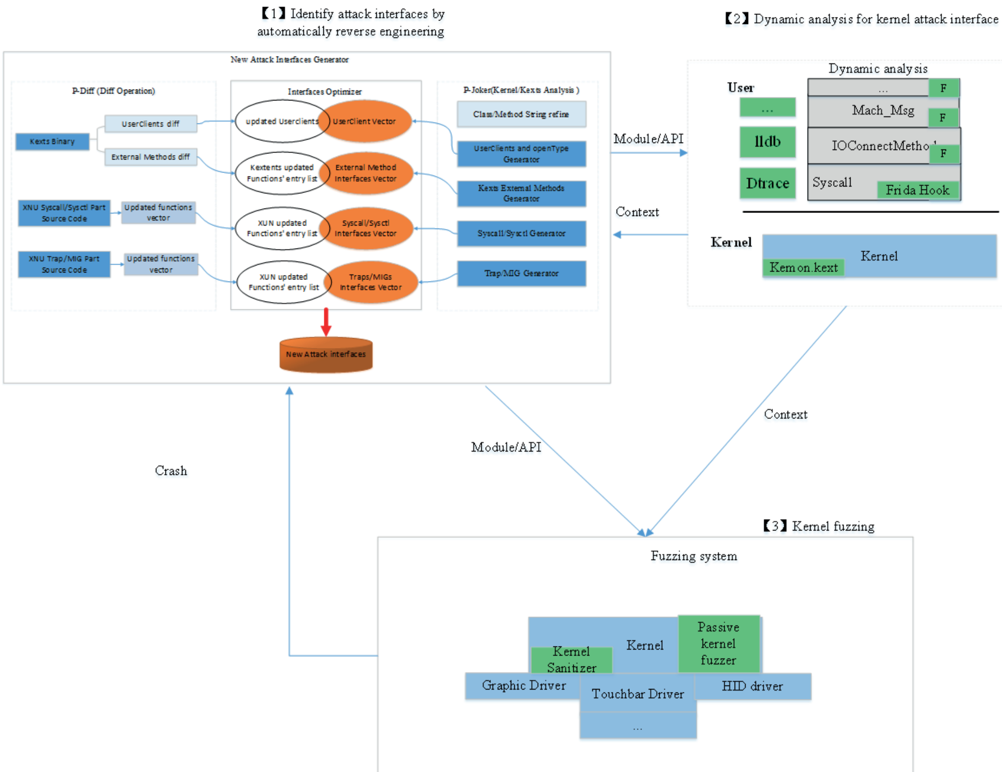


Figure 1: Basic architecture of the solution.

this paper, we should disassemble the Mach-O code in order to identify the driver extension module, classes, external methods, metadata, user clients, system call and other data structures for the kernel attack interface. Moreover, after a XUN and kexts diff and call graphic analysis, the entries list vector for those updated functions, which is used in the following steps, can be obtained quickly and exactly.

As part of our research, we would like to know not only the module/API information but also the context at runtime when these APIs are called. As the second step of the whole work flow, typically, dynamic analysis would try to get the call stack (with function name symbol and argument value) both in user mode and kernel mode, and determine how an object related to the kernel API is created (e.g. how the service is opened for IoConnectCallMethod), and so on.

Finally, as the third step, we would carry out (passive) fuzzing of these kernel attack interfaces to hunt for zero-day vulnerabilities. The key to fuzzing system design is to touch as many possible execution paths (or as much code coverage) as possible, and catch the first spot where a vulnerability is triggered. Hence, we try to hook the attack interface in the kernel and fuzz the data passed through from user mode directly to try to 'touch' more execution paths after the restriction check. A kernel address sanitizer mechanism is also introduced in order to catch the point of heap/stack overflow, for example.

As we can see, the static analysis would identify a new attack interface and dynamic analysis would help to trigger it, while the fuzzing system used on the identified attack interface would hunt for new vulnerabilities (kernel panic/crash) to help find more new attack interfaces.

2. REVERSE ENGINEERING AUTOMATICALLY FOR KERNEL ATTACK INTERFACE

Figure 2 show the reverse engineering solution for analysis of a *macOS*/*iOS* system. The new attack interfaces generator consists mainly of three parts, the work flow of which can be summarized in the following steps:

- The first step is kernel/kexts analysis, which can get all the attack interfaces from a newly released version including UserClient vector, external method interfaces vector, syscall/sysctl interfaces vector and traps/MIGs vector.
- The second step is to diff the XUN project and kexts between two neighbouring versions, which can get the entry list of the updated functions.
- The third step is to filter out those entries which cannot be accessed from user space and save the remaining entries which can be accessed directly.
- Finally, these interfaces are collected and saved into a fuzzing corpus database.

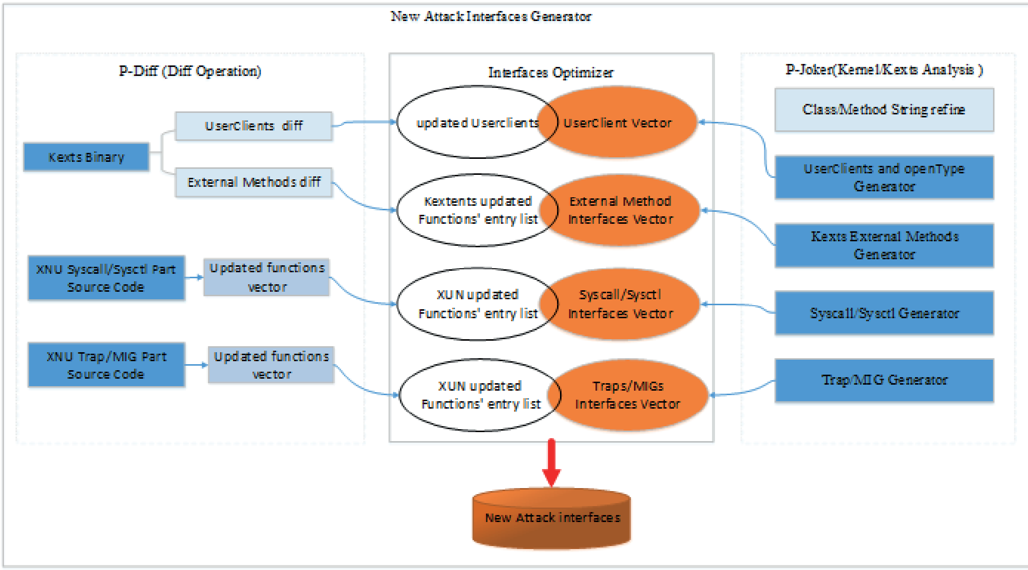


Figure 2: The reverse engineering solution for kernel and kextentions.

For the *iOS* or *macOS* kernel and kexts analysis, there are already many open-source tools available in *GitHub*, as shown in Table 1, but there are still some features that are not available for automatic analysis of the kernel or each kext. We introduce p-joker, which is an automatic tool used to analyse the *macOS* and *iOS* core module, the most powerful function of which is to get the service connection types and corresponding UserClients' external dispatch methods automatically.

Tools	Symbolicate C++ method tables	Class hierarchy	Struct class	UserClients and open type	Dispatch method
iokit-utils	N	Y	N	Y (using IOServiceOpen)	N
ida_kernelcache	Y	Y	Y	N	N
ioskextdump	Y	Y	N	N	Y(part)
Ryuk	Y	N	Y	N	N
p-joker	Y	Y	Y	Y	Y

Table 1: Comparison of open-source tools.

For the XUN project and kexts implementation diff, we introduce an *IDA Pro* script, p-diff, which can diff those non-open-source kexts and get a list of changed functions automatically. Then, it analyses the related kext and gets the call graphic for each updated function, and their entry list can be surmised from these calling sequences. In the end, p-diff will reserve those interfaces that can be accessed within a sandbox or by some user-mode privilege process.

2.1 Kernel and kexts automatic analysis

Figure 3 is an overview of p-joker implementation. Later, we will introduce how to get the service connection type and how to get the dispatch method. As mentioned before, there are many open-source tools available, so we will simply introduce those with overlapped content.

2.1.1 Class/method names refined through inheritance

The string symbol was stripped from the release version of the system so, for *macOS*, we can analyse the kernel and driver contained in the Kernel Debug Kit instead of the release one. However, there are no debug kits available for *iOS*. So we should refine the method names for each OSObject class, including the instance method table and meta method table. We know all the *Apple* drivers are implemented in C++, all of the services are inherited from the IOService class. So, through the inherited mechanism, many method names can be inferred from their parent class. Figure 4 shows the IOMobileFramebuffer class and the method table of its parent classes.

2.1.2 Methodology for finding IOKit service connection types and their user client vector

The IOKit in the kernel part delivers many MIG interfaces to user space in order to operate different drivers. When one user client is spawned, it should open the corresponding service first. In the user space, the IOServiceOpen function is responsible for spawning the driver's proxy client according to the given properties dictionary. Figure 5 shows the process to open a service through a MIG interface.

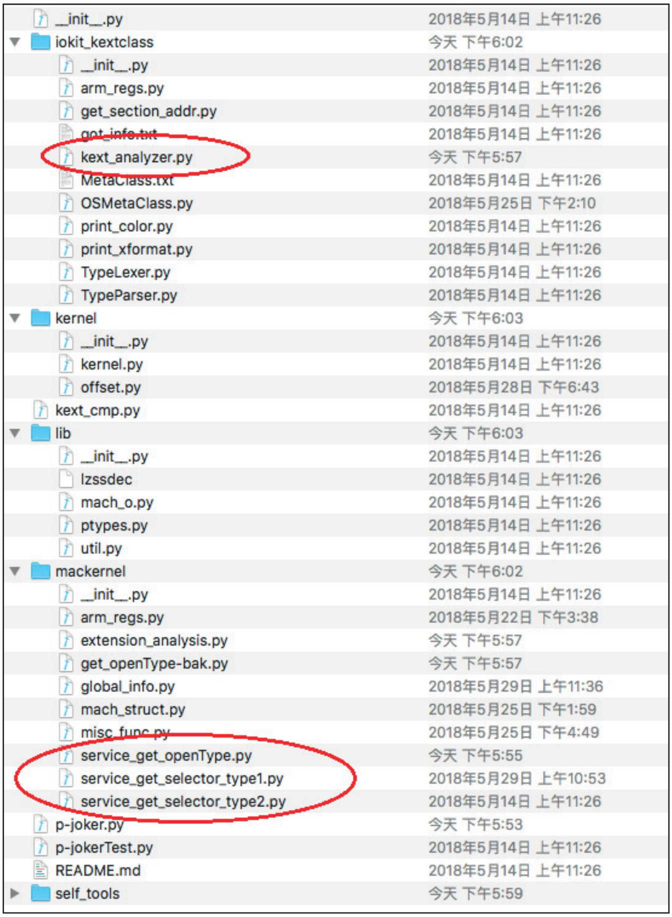


Figure 3: Overview of the implementation of p-joker.

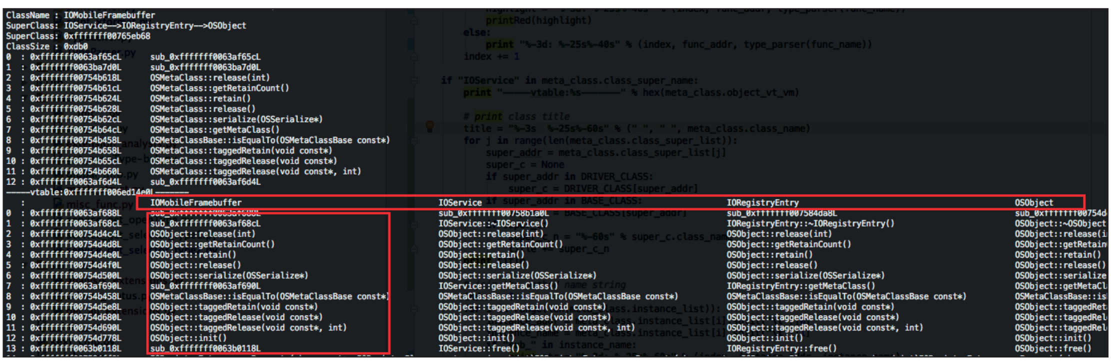


Figure 4: Refining the method table for the IOMobileFramebuffer class.

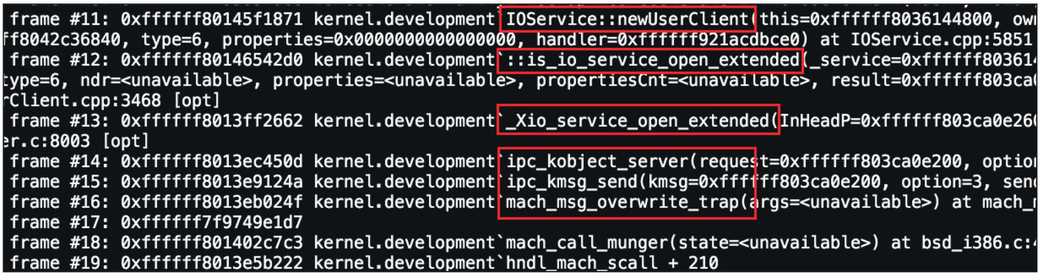


Figure 5: Call graphic of process to open a service in user space.

During the process to open a service, the system will first call the parent newUserClient method. However, because the IOService class is an abstract class and most of its functions are virtual ones, it will call the subservice that implements it. Figure 6 is an example of spawning an IOFramebuffer user client with connection type 0.

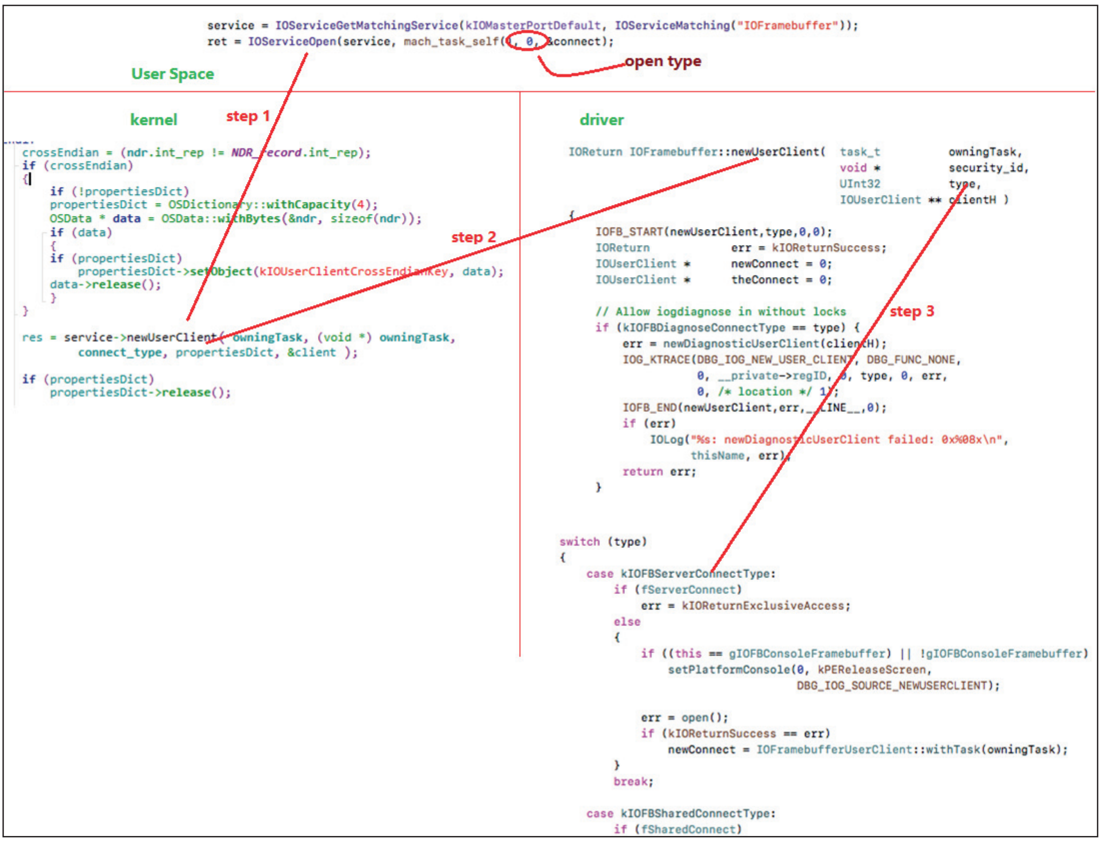


Figure 6: Spawning an IOFramebuffer user client with connection type 1.

So, the steps to find the connection tuple as shown in Figure 6 are the following:

- Locate the newUserClient function address in the driver.
- Enumerate the connection types.
- Analyse the instructions to get the corresponding user client for each connection type.

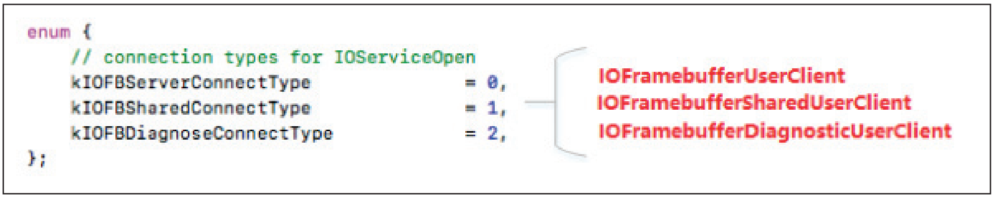


Figure 7: Connection types for IOServiceOpen.

2.1.3 Methodology for finding user client external methods automatically

IOUserClient is a subclass of IOService, which provides a basis for communication between client applications and I/O kit objects. Figure 8 shows the process of executing an external method.

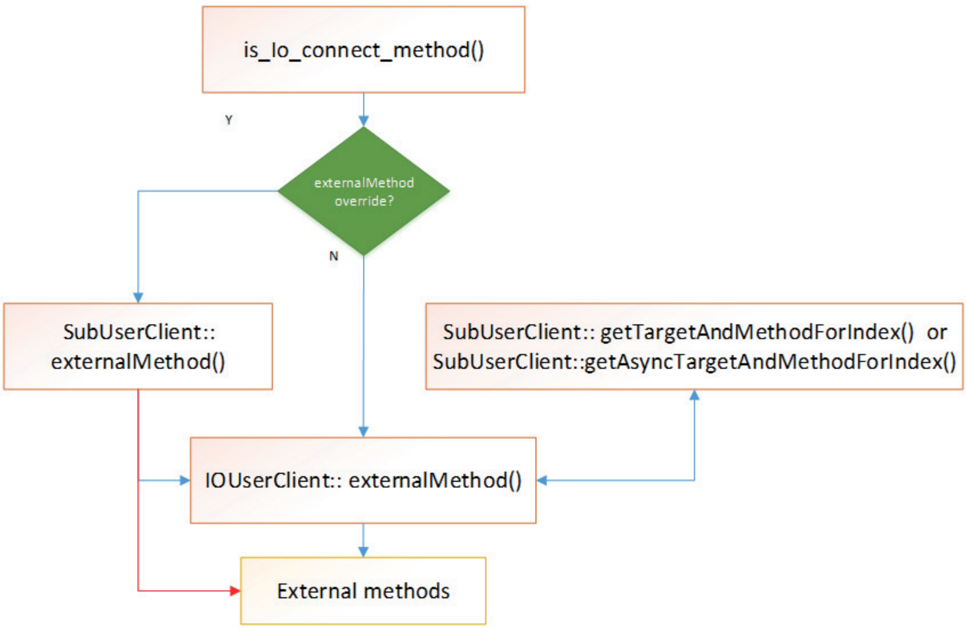


Figure 8: 'ExternalMethod' function workflow.

Clients use the 'IOUserClient::externalMethod' function to execute external methods. All the external methods are implemented within drivers and can be indexed by selectors. Drivers also define the methods' input and output conditions, which are used to check the user-mode input or output in coarse-grained user-mode input. For researchers, it is important to find the corresponding selector and conditions for these external methods.

2.1.3.1 Defined as global or local constant array

Generally, subservices will define this information in the externalMethod override function as a dispatch table that is usually a static constant array. If subservices do not override the externalMethod function, it will be defined in the getTargetAndMethodForIndex function or the getAsyncTargetAndMethodForIndex function, and the dispatch structure will be a little different, as shown in Figure 9.

```
IOReturn IOFramebufferUserClient::externalMethod( uint32_t selector, IOExternalMethodArguments * args,
                                                IOExternalMethodDispatch * dispatch, OSObject * target, void * reference )
{
    IOFBUC_START(externalMethod,selector,0,0);
    IOReturn     ret = kIOReturnSuccess;

    static const IOExternalMethodDispatch methodTemplate[] =
    {
        /*[0]*/  { (IOExternalMethodAction) &IOFramebuffer::extCreateSharedCursor,
            3, 0, 0, 0 },
        /*[1]*/  { (IOExternalMethodAction) &IOFramebuffer::extGetPixelInformation,
            3, 0, 0, sizeof(IOPixelInformation) },
        /*[2]*/  { (IOExternalMethodAction) &IOFramebuffer::extGetCurrentDisplayMode,
            0, 0, 2, 0 },
        /*[3]*/  { (IOExternalMethodAction) &IOFramebuffer::extSetStartupDisplayMode,
            2, 0, 0, 0 },
        /*[4]*/  { (IOExternalMethodAction) &IOFramebuffer::extSetDisplayMode,
            2, 0, 0, 0 },
        /*[5]*/  { (IOExternalMethodAction) &IOFramebuffer::extGetInformationForDisplayMode,
            1, 0, 0, kIOUCVariableStructureSize },

IOExternalMethod * IOHIDParamUserClient::getTargetAndMethodForIndex(
                        IOService ** targetP, UInt32 index )
{
    // get the same library function to work for param & server connects
    static const IOExternalMethod methodTemplate[] = {
        /* 0 */  { NULL, NULL, kIOUCScalarIScalarO, 1, 0 },
        /* 1 */  { NULL, NULL, kIOUCScalarIScalarO, 1, 0 },
        /* 2 */  { NULL, NULL, kIOUCScalarIScalarO, 1, 0 },
        /* 3 */  { NULL, (IOMethod) &IOHIDParamUserClient::extPostEvent, kIOUCStructIStructO, sizeof( struct evioLLE
        /* 4 */  { NULL, (IOMethod) &IOHIDSystem::extSetMouseLocation, kIOUCStructIStructO, 0xffffffff, 0 },
        /* 5 */  { NULL, (IOMethod) &IOHIDSystem::extGetStateForSelector, kIOUCScalarIScalarO, 1, 1 },
        /* 6 */  { NULL, (IOMethod) &IOHIDSystem::extSetStateForSelector, kIOUCScalarIScalarO, 2, 0 },
        /* 7 */  { NULL, (IOMethod) &IOHIDSystem::extRegisterVirtualDisplay, kIOUCScalarIScalarO, 0, 1 },
        /* 8 */  { NULL, (IOMethod) &IOHIDSystem::extUnregisterVirtualDisplay, kIOUCScalarIScalarO, 1, 0 },
        /* 9 */  { NULL, (IOMethod) &IOHIDSystem::extSetVirtualDisplayBounds, kIOUCScalarIScalarO, 5, 0 },
        /* 10 */ { NULL, (IOMethod) &IOHIDParamUserClient::extGetUserHidActivityState, kIOUCScalarIScalarO, 0, 1 },
        /* 11 */ { NULL, (IOMethod) &IOHIDSystem::setContinuousCursorEnable, kIOUCScalarIScalarO, 1, 0 },
```

Figure 9: Defined as local constant array in functions.

But there are also many SubUserClients declared as global arrays, as shown in Figure 10.

Regardless of whether the dispatch table is defined as a local array or a global array, the methodology to find the array is as follows:

1. Locate the starting address for each constant array in the symbol table.
2. Parse the contents according to the IOExternalMethodDispatch or IOExternalMethod structure from the starting address.

```
const IOExternalMethodDispatch IOHIDLibUserClient::
sMethods[kIOHIDLibUserClientNumCommands] = {
    { //    kIOHIDLibUserClientDeviceIsValid
    (IOExternalMethodAction) &IOHIDLibUserClient::_deviceIsValid,
    0, 0,
    2, 0
    },
    { //    kIOHIDLibUserClientOpen
    (IOExternalMethodAction) &IOHIDLibUserClient::_open,
    1, 0,
    0, 0
    },
    { //    kIOHIDLibUserClientClose
    (IOExternalMethodAction) &IOHIDLibUserClient::_close,
    0, 0,
    0, 0
    },
    { //    kIOHIDLibUserClientCreateQueue
```

Figure 10: Defined as global constant array in classes.

2.1.3.2 Defined in the code logic

Some drivers implement the external dispatch method using code logic instead of a constant array, as shown in Figure 11. Therefore, the methodology to find this kind of dispatch is as follows:

1. Locate the address of the override externalMethod/getTarget…/getAsyncTarget... function.
2. Analyse assembly instructions to get the selector and external methods.

```
IOReturn IOAudioEngineUserClient::externalMethod ( uint32_t selector, IOExternalMethodArguments * arguments,
    IOExternalMethodDispatch * dispatch, OSObject * target, void * reference)
{
    IOReturn result = kIOReturnBadArgument;
    audioDebugIOLog(3, "+ IOAudioEngineUserClient::externalMethod, selector=0x%x,   arg0 0x%llX, arg1 0x%llx, arg2 0x%llx arg3 0x%llx \n",
                    selector, arguments->scalarInput[0], arguments->scalarInput[1], arguments->scalarInput[2], arguments->scalarInput[3]);
    audioDebugIOLog(3, "  scalarInputCount=0x%x  structureInputSize 0x%x, scalarOutputCount 0x%x, structureOutputSize 0x%x \n",
                    arguments->scalarInputCount, arguments->structureInputSize, arguments->scalarOutputCount, arguments->structureOutputSize );

    // require entitlement for all external methods
    OSObject *entitlement = copyClientEntitlement(current_task(), kDriverHelper_DriverHostEntitlement);
    bool entitled = false;
    if ((entitlement != NULL) && (entitlement == kOSBooleanTrue)) {
        entitled = true;
    }
    require_action_string(entitled, Exit, result = kIOReturnNotPrivileged, "not entitled");

    // Dispatch the method call
    switch (selector)
    {
    case kIOAudioEngineCallRegisterClientBuffer:
        if (arguments != 0)
        {
            if ( arguments->scalarInputCount >= 4 )     //  <rdar://9204853>
            {
                result = registerBuffer64((IOAudioStream *)arguments->scalarInput[0], (mach_vm_address_t)arguments->scalarInput[1],
                        (UInt32)arguments->scalarInput[2], (UInt32)arguments->scalarInput[3] );
            }
            else
            {
                audioDebugIOLog(3, "  kIOAudioEngineCallRegisterClientBuffer: invalid input argument count %d. Need at least 4.\n",
                    arguments->scalarInputCount);
            }
        }
        break;
```

Figure 11: Defined in switch case.

2.1.4 Implementation for automatic tools

Depending on the methodology introduced previously, p-joker typically contains two implementation technologies. One emulates the execution of assembly instructions in order to get connection types and external methods implemented by code logic, while the other parses the symbol table in the driver Mach-O file in order to get the dispatch table defined as a constant array.

2.1.4.1 Analysis of assembly instructions

a. Start from scratch

To analyse or emulate all instructions from scratch is hard, however, the functions we care about use only a small instruction set. Figure 12 shows the assembly of the override newUserClient function.

To improve the accuracy, the key point is to ensure the correctness of emulation for control flow and data flow. For control flow, as many instructions as possible should be emulated. For data flow, the integrity and accuracy of registers' data transfer should be ensured. From Listing 1, it can be seen that connection type is the fourth argument, so during the instruction analysis process, it's important to analyse the control flow depending on ecx/rcx and related registers.

```
AppleHDAEngine::newUserClient(AppleHDAEngine *this, task *a2, void *a3, int
a4, IOUserClient **a5)
```

Listing 1: The AppleHDAEngine::newUserClient function.

```
                                        ; CODE XREF: AppleHDAEngine::newUserClient(
mov     r12d, 0E00002C2h
cmp     r14d, 3
jz      short loc_2C235
cmp     r14d, 2
jz      loc_2C2B4
test    r14d, r14d
jnz     loc_2C354
mov     rax, cs:off_920E0
xor     ecx, ecx
mov     rdi, r13
mov     rsi, r15
mov     rdx, [rbp+var_40]
mov     r8, [rbp+var_30]
call    qword ptr [rax+788h]
```

Figure 12: Assembly of AppleHDAEngine::newUserClient function.

b. Angr or miasm

As an alternative to starting from scratch, there are some excellent binary analysis tools, such as angr [1] and miasm [2], both of which can emulate the x64 or ARM code. Take miasm for example: not only can it emulate the execution of assembly instructions but it also monitors the value in each register.

Figure 13 shows the control flow that can be obtained using the miasm tool, after which the corresponding block can be parsed, with the result shown in Figure 14.

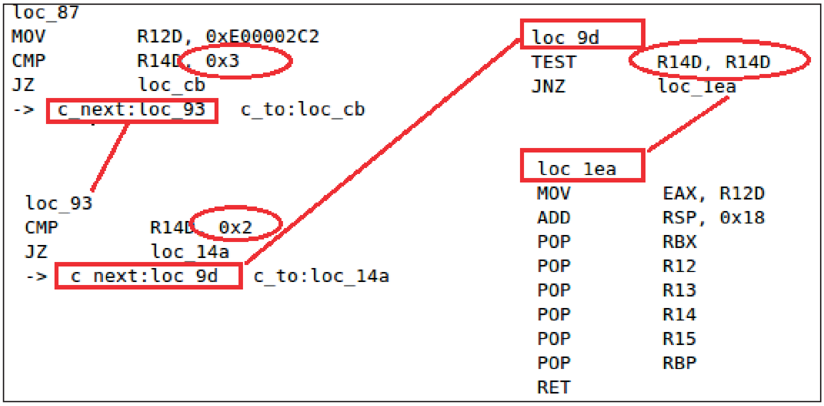


Figure 13: Control flow of the AppleHDAEngine::newUserClient function obtained using miasm.

```
------------------------------------------------------
index  CanOpen TOpenType       ServiceName                              extends
 4      True      0       AppleHDAEngineOutput                          IOAudioEngine::gMetaClass-->AppleHDAEngine-->AppleHDAEngineOutput
 86     True      0       AppleHDAEngine                                IOAudioEngine::gMetaClass-->AppleHDAEngine

------------------------------------------------------
ServiceName       OpenType       UserClient
AppleHDAEngine    0x3           AppleHDAEngineUserClient::metaClass
AppleHDAEngine    0x2           DspFuncUserClient::Create(IOAudioEngine*, task*)
```

Figure 14: User clients and connection types in the AppleHDA driver.

2.1.4.2 Parse the kexts Mach-O file

As we know, the constant variables are saved in the symbol table. So, it's very convenient to parse this table to get the address for each constant array.

```
String Table Index      __ZZN23IOFramebufferUserClient14externalMethodEjP25IOExternalMethodArgumentsP24IOExternalMethodDispatchP8OSObjectPvE14methodTemplate
Type
0E                      N_SECT
Section Index           7 (__DATA,__const)
Description
Value                   205360 ($+41072)

000627A0  0000C05F          String Table Index   __ZN18IOHIDLibUserClient8sMethodsE
000627A4  0F                Type
                            0E                   N_SECT
                            01                   N_EXT
000627A5  08                Section Index        8 (__DATA,__const)
000627A6  0000              Description
000627A8  0000000000042F10  Value                274192 ($+11216)
```

Figure 15: Constant variables in the symbol table.

After getting the address, the binary contents can simply be parsed with the IOExternalMethodDispatch or IOExternalMethod structure. The results are shown in Figure 16.

In the end, all the user clients with their connection types and external method dispatches can be obtained through these steps, and saved as interface vectors.

```
0x32230 – IOFramebufferUserClient::externalMethod(unsigned int, IOExternalMethodArguments*, IOExternalMethodDispatch*, OSObject*, void*)::methodTemplate  confidence: 100%
selector       cSIC          cSIS          cSOC          cSOS       func_name
  0             0x3           0x0           0x0           0x0       IOFramebuffer::extCreateSharedCursor(OSObject*, void*, IOExternalMethodArguments*)
  1             0x3           0x0           0x0          0xac       IOFramebuffer::extGetPixelInformation(OSObject*, void*, IOExternalMethodArguments*)
  2             0x0           0x0           0x2           0x0       IOFramebuffer::extGetCurrentDisplayMode(OSObject*, void*, IOExternalMethodArguments*)
  3             0x2           0x0           0x0           0x0       IOFramebuffer::extSetStartupDisplayMode(OSObject*, void*, IOExternalMethodArguments*)
  4             0x2           0x0           0x0           0x0       IOFramebuffer::extSetDisplayMode(OSObject*, void*, IOExternalMethodArguments*)
  5             0x1           0x0           0x0    0xffffffff       IOFramebuffer::extGetInformationForDisplayMode(OSObject*, void*, IOExternalMethodArguments*)
  6             0x0           0x0           0x1           0x0       IOFramebuffer::extGetDisplayModeCount(OSObject*, void*, IOExternalMethodArguments*)
  7             0x0           0x0           0x0    0xffffffff       IOFramebuffer::extGetDisplayModes(OSObject*, void*, IOExternalMethodArguments*)
  8             0x1           0x0           0x1           0x0       IOFramebuffer::extGetVRAMMapOffset(OSObject*, void*, IOExternalMethodArguments*)
  9             0x0    0xffffffff           0x0           0x0       IOFramebuffer::extSetBounds(OSObject*, void*, IOExternalMethodArguments*)
 10             0x3           0x0           0x0           0x0       IOFramebuffer::extSetNewCursor(OSObject*, void*, IOExternalMethodArguments*)
 11             0x5    0xffffffff           0x0           0x0       IOFramebuffer::extSetGammaTable(OSObject*, void*, IOExternalMethodArguments*)
 12             0x1           0x0           0x0           0x0       IOFramebuffer::extSetCursorVisible(OSObject*, void*, IOExternalMethodArguments*)
 13             0x2           0x0           0x0           0x0       IOFramebuffer::extSetCursorPosition(OSObject*, void*, IOExternalMethodArguments*)
 14      0xffffffff           0x0           0x0           0x0       IOFramebuffer::extAcknowledgeNotification(OSObject*, void*, IOExternalMethodArguments*)
 15             0x1    0xffffffff           0x0           0x0       IOFramebuffer::extSetColorConvertTable(OSObject*, void*, IOExternalMethodArguments*)
 16             0x2    0xffffffff           0x0           0x0       IOFramebuffer::extSetCLUTWithEntries(OSObject*, void*, IOExternalMethodArguments*)
 17             0x0    0xffffffff           0x0    0xffffffff       IOFramebuffer::extValidateDetailedTiming(OSObject*, void*, IOExternalMethodArguments*)
 18             0x1           0x0           0x1           0x0       IOFramebuffer::extGetAttribute(OSObject*, void*, IOExternalMethodArguments*)
 19             0x2           0x0           0x0           0x0       IOFramebuffer::extSetAttribute(OSObject*, void*, IOExternalMethodArguments*)
 20             0x3    0xffffffff           0x0           0x0       IOFramebuffer::extSetHibernateGammaTable(OSObject*, void*, IOExternalMethodArguments*)
```

Figure 16: External method dispatch of IOFramebufferUserClient.

2.1.5 Best practice

2.1.5.1 Assembly instruction set emulation

Figure 17 shows the implementation of the mov, cmp, je, jz and test instructions operation. P-joker implements an API set to operate the register. For emulating more operations, the x64 or ARM architecture reference manual [3] can be referenced.

2.1.5.2 Parse binary contents according different structure

Figure 18 shows the implementation used to read contents from a Mach-O binary using the IOExternalMethodDispatch or IOExternal structure.

2.2 Kernel and kexts diff analysis

Apple has open-sourced its XNU project for both *macOS* and *iOS* systems, as well as part of the drivers. For closed-source drivers, automatic reverse engineering methods have already been introduced, therefore all the attack interfaces can be obtained.

However, in order to find the newest introduced attack interface, researchers need to know which functions have been updated and which services or syscalls are newly added. Once that information has been obtained they also need to know how to access or call these updated or newly added attack interfaces. The following sections will introduce how to find the newly added attack interfaces and list their entry points.

2.2.1 Kernel diff methodology

MIG interface, syscall, sysctl and traps are implemented in the system kernel. Listing 2 shows their source code in the XNU project.

1. First, coarse-grained parse and diff these files in order to get the new interfaces. These interfaces include the newly added ones and ones in which arguments have changed. Alternatively, the corresponding information can be obtained directly from the kernel binary diff – there are already many excellent tools available, such as joker [4].

```
xr_m.set_actual_value_by_regN(x86_const.X86_REG_RIP, address + insn.size)

if mnemonic in ["mov"]:
    seg_num = insn.op_count(CS_OP_REG)
    if seg_num == 2:
        f_reg = get_first_reg(insn)
        s_reg = get_second_reg(insn)
        if s_reg in [x86_const.X86_REG_RCX, x86_const.X86_REG_ECX]:
            subs_rcx.append(f_reg)

if mnemonic in ["cmp"]:
    seg_num = insn.op_count(CS_OP_REG)
    imm_num = insn.op_count(CS_OP_IMM)
    if seg_num == 1 and imm_num == 1:
        f_reg = get_first_reg(insn)
        if f_reg in subs_rcx:
            open_Type = get_single_IMM(insn)
            couple_switch = 1

if couple_switch and mnemonic in ["je", "jz"]:
    imm_num = insn.op_count(CS_OP_IMM)
    if imm_num == 1:
        jump_vm = get_single_IMM(insn)
        jump_f = k_header.get_f_from_vm(text_fileaddr, text_vmaddr, int(jump_vm, 16))
        userclient = analysis_uc_func(k_header, int(jump_vm, 16), jump_f)
        if userclient:
            meta_class.openType[open_Type] = userclient
    couple_switch = 0

if mnemonic in ["test"]:
    seg_num = insn.op_count(CS_OP_REG)
    if seg_num == 2:
        f_reg = get_first_reg(insn)
        s_reg = get_second_reg(insn)
```

Figure 17: Code snippet for emulating instruction execution from scratch.

```
if meta_method_n == "externalMethod":
    meta_class.is_ioemd = True

    while True:
        func_addr = k_header.memcpy(start_f, 8)
        func_name = ""
        if func_addr in EXT_RELOCATIONS:
            func_name  = EXT_RELOCATIONS[func_addr]
        elif func_addr in STRING_TAB:
            func_name = STRING_TAB[func_addr]
        if start_f > check_boundary:
            break

        if func_name and "::gMetaClass" not in demangle(func_name):
            io_external_md = IOExternalMethodDispatch()
            io_external_md.function_addr = func_addr
            io_external_md.function_name = func_name
            io_external_md.checkScalarInputCount    = hex(k_header.memcpy(start_f + 8, 4))
            io_external_md.checkStructureInputSize  = hex(k_header.memcpy(start_f + 12, 4))
            io_external_md.checkScalarOutputCount   = hex(k_header.memcpy(start_f + 16, 4))
            io_external_md.checkStructureOutputSize = hex(k_header.memcpy(start_f + 20, 4))
            meta_class.IOExternalMethodDispatch.append(io_external_md)
            start_f += 24
        else:
            break

elif meta_method_n == "getTargetAndMethodForIndex":
    meta_class.is_ioem = True

    while True:
        # check the end of IOExternalMethod array
        # two situations: 1) another metaclass startup; 2) start_f out of const fields.
        check_addr = k_header.memcpy(start_f, 8)
        start_vm = k_header.get_vm_from_f(check_c_f, check_c_vm, start_f)
```

Figure 18: Parsing the binary contents using the corresponding structure.

```
xnu-4570.71.2/
|-- bsd
|   `-- kern
|       |-- kern_sysctl.c                //sysctl
|       `-- syscalls.master              //syscall
`-- osfmk
    |-- device
    |   `-- device.defs                //mig
    |-- kern
    |   `-- syscall_sw.c               //traps
    `-- mach
        `-- mach_traps.h              //traps
```

Listing 2: Related implementation files in XNU source code.

2. Next, diff all the files in the XNU project and tag those that have been changed or newly added. Ignore unrelated files, such as test code files and deleted files, take BSD VFS.

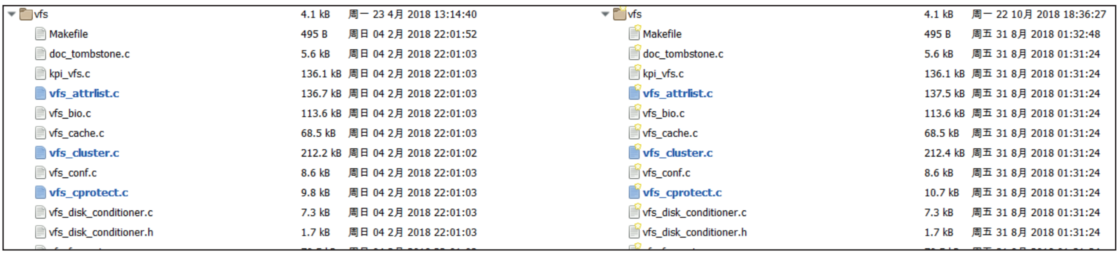


Figure 19: Difference between the BSD VFS folders.

3. Next, compile statistics for changed or newly added functions in those files, and the related function name list.

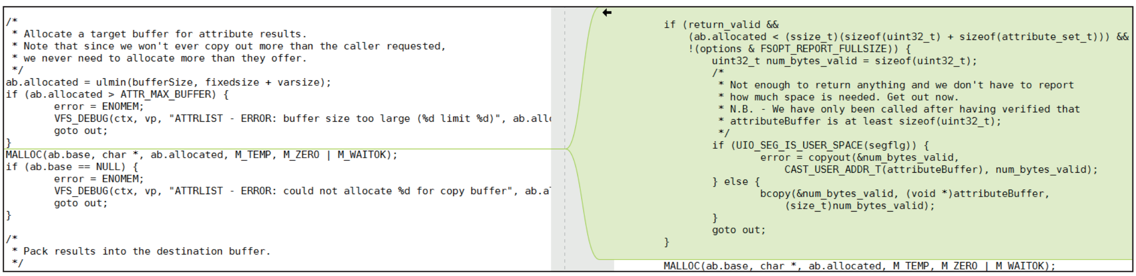


Figure 20: Changed function 'getvolattrlist ()' in the vfs_attrlist.c file.

4. Next, construct the calling sequence for the changed functions and get the entry functions. This step in p-diff is implemented through an *IDA Pro* script. There are two functions, 'CodeRefsTo(ea, flow)' and 'CodeRefsFrom(ea, flow)', available in the idautils.py file [5].

```
[P-Diff calling sequence 0]: _getvolattrlist() <- _getattrlist_internal() <- _getattrlistat_internal() <- _getattrlistat()
[P-Diff calling sequence 1]: _getvolattrlist() <- _getattrlist_internal() <- _fgetattrlist()
[P-Diff calling sequence 2]: _getvolattrlist() <- _getattrlist_internal() <- _readdirattr() <- _getattrlistbulk()
[P-Diff calling sequence 3]: _getvolattrlist() <- _getattrlist_internal() <- _getattrlistat_internal() <- _getattrlist()
```

Figure 21: Calling sequence of the 'getvolattrlist ()' function.

5. Finally, list the entry functions for each calling sequence, and check if they are interfaces that are exposed to user mode. Table 2 shows the syscalls that can be called from user space. Together with the newly added and changed interfaces, these are the new attack interfaces which should be the main fuzz point.

`220 AUE_GETATTRLIST ALL { int getattrlist(const char *path, struct attrlist *alist, void *attributeBuffer, size_t bufferSize, u_long options) NO_SYSCALL_STUB; }`
`461 AUE_GETATTRLISTBULK ALL { int getattrlistbulk(int dirfd, struct attrlist *alist, void *attributeBuffer, size_t bufferSize, uint64_t options); }`
`228 AUE_FGETATTRLIST ALL { int fgetattrlist(int fd, struct attrlist *alist, void *attributeBuffer, size_t bufferSize, u_long options); }`
`476 AUE_GETATTRLISTAT ALL { int getattrlistat(int fd, const char *path, struct attrlist *alist, void *attributeBuffer, size_t bufferSize, u_long options); }`

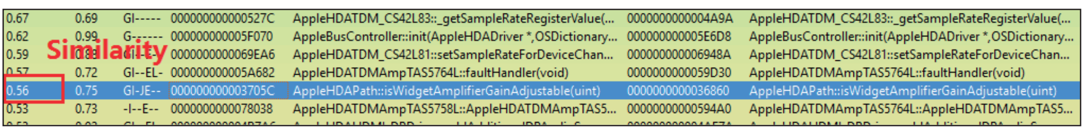
Table 2: Syscalls obtained from the calling sequence in the fourth step.

In fact, the example we mentioned is the patch for CVE-2018-4243, which was found by Ian Beer. However, using this methodology, researchers can find the newest attack interface quickly. After that, they can update their fuzz corpus accordingly and discover the potential vulnerabilities introduced by the newly added code.

2.2.2 Driver diff methodology

Nearly all the drivers on *macOS* are closed source, however, the methodology is similar. The only difference is that all the diff operations would be based on binary instead of source code. For binary diff, an *IDA Pro* script can be used with the BinDiff [6] plug-in.

All drivers on *macOS* can be found in the '/System/Library/Extensions' folder. A comparison of the same driver's binary can be made for different versions. But for *iOS*, all the drivers are pre-compiled within the kernelcache file. Researchers should split all drivers first using p-joker or an existing tool such as joker. However, the format for *iOS* kernelcache has changed, leading to the existing tools no longer working. Luckily, p-joker supports the new format – we will push the newest p-joker version to *GitHub* soon.



0.67	0.69	GI-----	000000000000527C	AppleHDATDM_CS42L83::_getSampleRateRegisterValue(...	0000000000004A9A	AppleHDATDM_CS42L83::_getSampleRateRegisterValue(...
0.62	0.99	G------	000000000005F070	AppleBusController::init(AppleHDADriver *,OSDictionary...	000000000005E6D8	AppleBusController::init(AppleHDADriver *,OSDictionary...
0.59	[illegible]	[illegible]	0000000000069EA6	AppleHDATDM_CS42L81::setSampleRateForDeviceChan...	000000000006948A	AppleHDATDM_CS42L81::setSampleRateForDeviceChan...
0.57	0.72	GI--EL-	000000000005A682	AppleHDATDMAmpTAS5764L::faultHandler(void)	0000000000059D30	AppleHDATDMAmpTAS5764L::faultHandler(void)
0.56	0.75	GI-JE--	000000000003705C	AppleHDAPath::isWidgetAmplifierGainAdjustable(uint)	0000000000036860	AppleHDAPath::isWidgetAmplifierGainAdjustable(uint)
0.53	0.73	-I--E--	0000000000078038	AppleHDATDMAmpTAS5758L::AppleHDATDMAmpTAS5...	00000000000594A0	AppleHDATDMAmpTAS5764L::AppleHDATDMAmpTAS5...

Figure 22: Diff results using an IDA Pro script.

The main purpose of driver diff is to find selectors for the newly added external methods and the external method entries for changed functions. This way, we only get those interfaces that can fuzz the updated code effectively and in a timely way.

```
P-Diff: entry functions

        AppleHDAEngine::createVolumeAndMuteControlsForActivePathSet()
        AppleHDAEngine::handlePowerStateChange()
        AppleHDAEngine::resetDSPToPlist()
        AppleHDAEngineInput::init(IOService, AppleHDACodec, OSDictionary, OSArray)
        AppleHDAEngineInput::initAudioStream()
        AppleHDAEngineInput::performFormatChange(IOAudioStream, _IOAudioStreamFormat, _IOAudioSampleRate)
        AppleHDAEngineInput::protectedChangePathSet(IOAudioControl)
        AppleHDAEngineOutput::init(IOService, AppleHDACodec, OSDictionary, OSArray)
        AppleHDAEngineOutput::protectedChangePathSet(IOAudioControl)
        AppleHDAEngineUserClient::getHardwareVolume()
        AppleHDAEngineUserClient::getStateAction(UserClientData)
        AppleHDAEngineUserClient::setHardwareVolume()
        AppleHDAEngineUserClient::setStateAction(UserClientData)
```

Figure 23: Find the entry functions list using p-diff.

2.2.3 Best practice

2.2.3.1 Find the user-mode entry for updated functions

After the calling sequences have been obtained, the main job is to find the user-mode entry that can call them. Figure 24 shows a code snippet of p-diff implementation.

```
def GetRefToSet(to_func):

    for r in CodeRefsTo(to_func,1):

        if GetFunctionName(to_func) in call_array.keys():
            call_array[GetFunctionName(to_func)].append(GetFunctionName(r))
            if GetFunctionName(r) not in call_array.keys():
                call_array[GetFunctionName(r)] = []
        else:
            call_array[GetFunctionName(to_func)] = [GetFunctionName(r)]
            call_array[GetFunctionName(r)] = []
        GetRefToSet(LocByName(GetFunctionName(r)))

def print_callgraphic(xrefto_name):
    for key, value in call_array.iteritems():
        if len(value) == 0:
            #entry_func.append(cxxfilt.demangle(value[1:]))
            entry_func.append(type_parser(key))
            call_paths.append(find_path(call_array, xrefto_name, key))
    return call_paths
```

Figure 24: Code snippet of p-diff implementation.

2.2.3.2 New attack interfaces introduced by iOS 12

Some kextensions will have been removed or added in the newest version release. Table 3 shows the updated kernel extensions list.

Drivers	Status
com.apple.driver.ApplePinotLCD	Newly added
com.apple.AppleARM64ErrorHandler	Newly added
com.apple.drivers.AppleS7002SPU	Newly added
com.apple.driver.AppleSMCWirelessCharger	Newly added
com.apple.driver.usb.AppleUSBHub	Newly added
com.apple.AppleSMC_Embedded	Changed
com.apple.AGXFirmwareKextG10P	Changed
com.apple.kext.CoreTrust	Newly added
com.apple.nke.lttp	Newly added
com.apple.iokit.IOUSBHostFamily	Changed
com.apple.driver.BCMWLANFirmware4357_Hashstore	Changed
com.apple.Libm.kext	Changed
com.company.driver.modulename	Newly added
com.apple.AppleHapticsSupportCallan	Newly added
com.apple.driver.AppleCredentialManager	Newly added
com.apple.security.AppleImage4	Newly added
com.apple.AGXFirmwareKextG5P	Changed
com.apple.file-systems.hfs.kext	Changed
com.apple.driver.AppleAVE	Changed
com.apple.iokit.IOReporting	Newly added
com.apple.driver.AppleEmbeddedAudioLibs	Newly added
com.apple.driver.AOPTouchKext	Newly added
com.apple.drivers.AppleS7002SPUSphere	Newly added

Table 3: Updated kextensions in iOS 12.0.1.

2.2.3.3 New attack interfaces introduced by macOS 10.14 AppleHDA.kext

Due to there being lots of drivers, here we only take the AppleHDA.kext driver as an example. Table 4 shows its external method dispatch details. Table 5 shows the updated functions from *macOS* 10.13.6 to *macOS* 10.14, as well as their entry interfaces in user-mode.

Selector	Function name	Scalar InputCount	Structure InputSize	Scalar OutputCount	Structure OutputSize
0	getState	2	0	0	0xfff
1	setState	2	0xfff	0	0
2	resetDSPToPropertyList	0	0	0	0
3	isPortPresent	1	0	1	0
4	getHardwareVolume	0	0	6	0
5	setHardwareVolume	1	0	0	0
6	getActiveSpatialChannels	0	0	0x10	0
7	getAudioSnoopEnabled	0	0	3	0
8	setAudioSnoopEnabled	3	0	0	0
9	setSpatialChannelMute	2	0	0	0

Table 4: External method dispatch details for AppleHDA.kext in macOS 10.14.

```
P-Diff: entry functions for function
AppleHDAEngine::resetVolumeFromVolumeCacheForAppleHDAPathSet(AppleHDAPathSet)
        AppleHDAEngineUserClient::setStateAction(UserClientData)
```

```
P-Diff: entry functions for function
AppleHDAEngine::resetSoftwareVolumeFromVolumeCacheForAppleHDAPathSet(
AppleHDAPathSet)
        AppleHDAEngineUserClient::setStateAction(UserClientData)
```

```
P-Diff: entry functions for function
AppleHDAPath::isWidgetAmplifierMuteCapable()
        AppleHDAEngineUserClient::setSpatialChannelMute()
        AppleHDAEngineUserClient::setStateAction(UserClientData)
```

```
P-Diff: entry functions for function
AppleHDAPath::isWidgetAmplifierGainAdjustable()
        AppleHDAEngineUserClient::getHardwareVolume()
        AppleHDAEngineUserClient::getStateAction(UserClientData)
        AppleHDAEngineUserClient::setHardwareVolume()
        AppleHDAEngineUserClient::setStateAction(UserClientData)
```

```
P-Diff: entry functions for function AppleHDAPath::getWidgetAmplifierGainRange()
        AppleHDAEngineUserClient::getHardwareVolume()
        AppleHDAEngineUserClient::getStateAction(UserClientData)
```

```
P-Diff: entry functions for function
AppleHDAPathSet::isAmplifierGainAdjustable()
        AppleHDAEngineUserClient::getHardwareVolume()
        AppleHDAEngineUserClient::getStateAction(UserClientData)
        AppleHDAEngineUserClient::setHardwareVolume()
        AppleHDAEngineUserClient::setStateAction(UserClientData)
```

```
P-Diff: entry functions for function AppleHDAPathSet::isAmplifierMuteCapable()
        AppleHDAEngineUserClient::setSpatialChannelMute()
        AppleHDAEngineUserClient::setStateAction(UserClientData)
```

Table 5: User-mode entry points for changed functions in AppleHDA.kext.

3. DYNAMIC ANALYSIS FOR KERNEL ATTACK INTERFACE

As we have mentioned before, the key methodology for dynamic analysis is to get the runtime context of the attack interface API in order to help trigger, fuzz or even reproduce the potential vulnerability.

As our best practice, we would choose Frida to control and trace user-mode context and Dtrace to trace the kernel counterpart. As a manual alternative, debugging (via lldb) both the user and kernel is reasonable.

Table 6 shows a basic comparison of different typical dynamic traces according to difference dimensions.

	User trace	Kernel trace	Embedded in OS	Any privilege?	Support script?	Performance	Platform
Frida	Yes	No	No	Root or Repack	Yes	Middle	iOS/OSX
Dtrace	No	Yes	Yes	Root	Yes	High	OSX
lldb	Yes	Yes	Yes	Root	Yes	Low	iOS/OSX
Kernel hook	---	Yes	No	Root	No	Middle	OSX

Table 6: Comparison of dynamic traces.

3.1 Frida hook in user mode

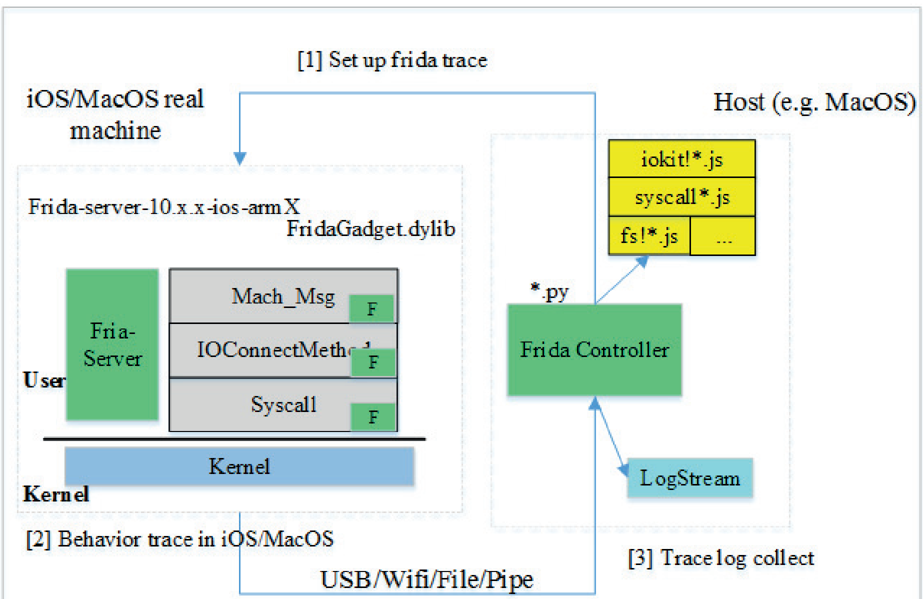


Figure 25: Frida hook in user mode.

Frida is one of the most popular dynamic instrumentation toolkits on many platforms including *MacOS* and *iOS*. One of its advantages is that it allows you to peek at and control every function using well documented JavaScript APIs which include the pre and post event handling. Typically, the retrieved runtime information includes call stack backtrace, thread context, return value, and any other you define. Take the xpc_connection_send_message API context for example, as shown in Listing 3.


```
{"time":"2017-09-18T10:38:32.807Z",
"txnType":"moony?",
"lib":"libxpc.dylib",
"method":"xpc_connection_send_message",
"artifact":[{
    "name":"connection",
    "value":"0x1658d090","argSeq":0}, {"name":"connectioninfo","value":"\
tconnection=0x1658d090\tconnectionName=\tconnectionPid=2312\
tconnectionProcName=Preferences","argSeq":0},
    {"name":"retval","value":374477440,"argSeq":-1}
]}
```


Listing 3: Xpc_connection_send_message API.

As the basic steps to use Frida, first launch the Frida server under root privilege or repack the Frida gadget in the target application, then you can develop your own JavaScript code for hooking any API you want in the Frida controller. The trace log generated by the Frida server is sent to the Frida controller for further analysis via USB or network.

3.2 Dtrace in kernel mode

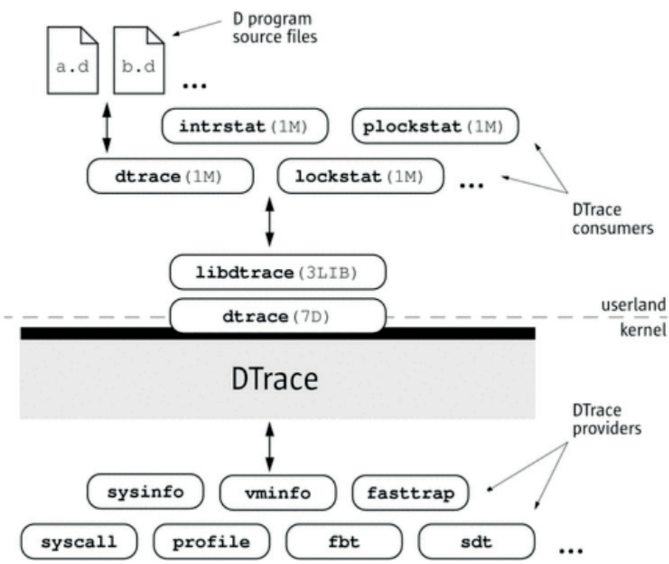


Figure 26: Dtrace architecture. (Source: Solaris Dynamic Tracing Guide.)

Dtrace has one the best tracing designs, with high performance and usability in Unix-like systems including *OSX* (unfortunately, Dtrace is not officially supported on *iOS* devices), as can be seen from the architecture shown in Figure 26.

Dtrace also provides multiple probes embedded in the kernel with categories such as sysinfo, syscall, fbt, sdt and so on. The typical system call, IOKit, mach msg, network, disk and file are almost all covered by Dtrace probes. Figure 27 shows the Dtrace providers list.

What is more, D language (*.d) in script provides fruitful APIs to intercept pre or post event (e.g. BEGIN,END) and keywords related to runtime process context (e.g. PID, timestamp, filename, exe name and so on). Thus you can see the code pieces for the IOFile probe. Figure 28 shows the Dtrace script for file probe.

3.3 Misc

Another useful dynamic trace tool is lldb embedded in *OSX* and *iOS* systems. Besides typical debugging utilities such as single step, break point, memory read/write operations and thread info, lldb can debug any user-mode service or process and even the whole *OSX* kernel. Lldb also supports python script to wrap the typical lldb-related objects (e.g. thread, process, module, memory, lldb attach) and operation with good documentation. In fact, we not only traced the API sequences we are interested in dynamically, but we also fuzzed the kernel to reproduce lots of kernel crashes, details of which will be announced in another paper.

4. ENHANCED PASSIVE FUZZING SOLUTION

We want some kind of enhanced *OSX*/*iOS* kernel fuzzing system, which is currently under development since we know the technical details of the kernel interface APIs from the viewpoint of static and dynamic analysis.

As the key methodology for fuzzing, we would like to touch as much of the execution path (code coverage) as possible and also catch the first spot of the kernel crash.

The first step of fuzzing is to try to generate a fruitful corpus of kernel interface APIs and call the kernel from the user agent. Besides blind fuzzing using tools like Trinity, we recommend using normal programs which have more opportunities to interact with kernels than the agent. For example, playing 3D games that use openGL or graphics drivers, operating peripheral devices (e.g. Wi-Fi, Bluetooth management), and so on. These kinds of real kernel API call could eventually touch ‘deeper’ kernel code execution paths because the legal input parameters have already bypassed most trick kernel checks.

As the second step, the passive fuzzer intercepts the typical API in the kernel counterpart usually as an inline hook for pre and post event handling. By fuzzing the input data of the API parameter (usually as buffer content of an argument, or kernel/user shared memory) in special time strategy, you could probably get plenty of kernel crashes.

The kernel sanitizer mechanism (such as KASAN, kernel address sanitizer) could be useful for improving the quantity and quality of fuzzing. Without a sanitizer mechanism, a crash caused by a memory corruption vulnerability may be handled or dismissed by the kernel code itself. What is more, there would exist instruction sequence disorder between the root cause point and the final instruction pointed to by RIP in one crash, which could cost more analysis effort for researchers.

```
sh-3.2# dtrace -l |more
   ID   PROVIDER            MODULE                          FUNCTION NAME
    1     dtrace                                                     BEGIN
    2     dtrace                                                     END
    3     dtrace                                                     ERROR
    4   lockstat       mach_kernel                      lck_mtx_lock adaptive-acquire
    5   lockstat       mach_kernel                      lck_mtx_lock adaptive-spin
    6   lockstat       mach_kernel                      lck_mtx_lock adaptive-block
    7   lockstat       mach_kernel                  lck_mtx_try_lock adaptive-acquire
    8   lockstat       mach_kernel             lck_mtx_try_spin_lock adaptive-acquire
    9   lockstat       mach_kernel                    lck_mtx_unlock adaptive-release
   10   lockstat       mach_kernel                  lck_mtx_ext_lock adaptive-acquire
   11   lockstat       mach_kernel                  lck_mtx_ext_lock adaptive-spin
   12   lockstat       mach_kernel                  lck_mtx_ext_lock adaptive-block
   13   lockstat       mach_kernel                lck_mtx_ext_unlock adaptive-release
   14   lockstat       mach_kernel                 lck_mtx_lock_spin adaptive-acquire
   15   lockstat       mach_kernel                lck_rw_lock_shared rw-acquire
   16   lockstat       mach_kernel                lck_rw_lock_shared rw-block
   17   lockstat       mach_kernel                lck_rw_lock_shared rw-spin
   18   lockstat       mach_kernel             lck_rw_lock_exclusive rw-acquire
   19   lockstat       mach_kernel             lck_rw_lock_exclusive rw-block
   20   lockstat       mach_kernel             lck_rw_lock_exclusive rw-spin
   21   lockstat       mach_kernel                       lck_rw_done rw-release
   22   lockstat       mach_kernel            lck_rw_try_lock_shared rw-acquire
   23   lockstat       mach_kernel         lck_rw_try_lock_exclusive rw-acquire
   24   lockstat       mach_kernel        lck_rw_shared_to_exclusive rw-upgrade
   25   lockstat       mach_kernel        lck_rw_shared_to_exclusive rw-spin
   26   lockstat       mach_kernel        lck_rw_shared_to_exclusive rw-block
   27   lockstat       mach_kernel        lck_rw_exclusive_to_shared rw-downgrade
   28   lockstat       mach_kernel                     lck_spin_lock spin-acquire
   29   lockstat       mach_kernel                     lck_spin_lock spin-spin
   30   lockstat       mach_kernel                   lck_spin_unlock spin-release
   31    profile                                                     profile-97
   32    profile                                                     profile-199
   33    profile                                                     profile-499
   34    profile                                                     profile-997
   35    profile                                                     profile-1999
   36    profile                                                     profile-4001
   37    profile                                                     profile-4999
   38    profile                                                     tick-1
   39    profile                                                     tick-10
   40    profile                                                     tick-100
```

Figure 27: Dtrace providers list.

```
dtrace:::BEGIN
{
    printf("Tracing... Hit Ctrl-C to end.\n");
}

/* save time at start */
io:::wait-start
{
    self->start = timestamp;
}

/* process event */
io:::wait-done
/self->start/
{
    /*
     * wait-done is used as we are measing wait times. It also
     * is triggered when the correct thread is on the CPU, obviating
     * the need to link process details to the start event.
     */
    this->elapsed = timestamp - self->start;
    @files[pid, execname, args[2]->fi_pathname] = sum(this->elapsed);
    self->start = 0;
}

/* print report */
dtrace:::END
{
    normalize(@files, 1000);
    printf("%6s %-12s %8s %s\n", "PID", "CMD", "TIME", "FILE");
    printa("%6d %-12.12s %@8d %s\n", @files);
```

Figure 28: Dtrace script for file probe.

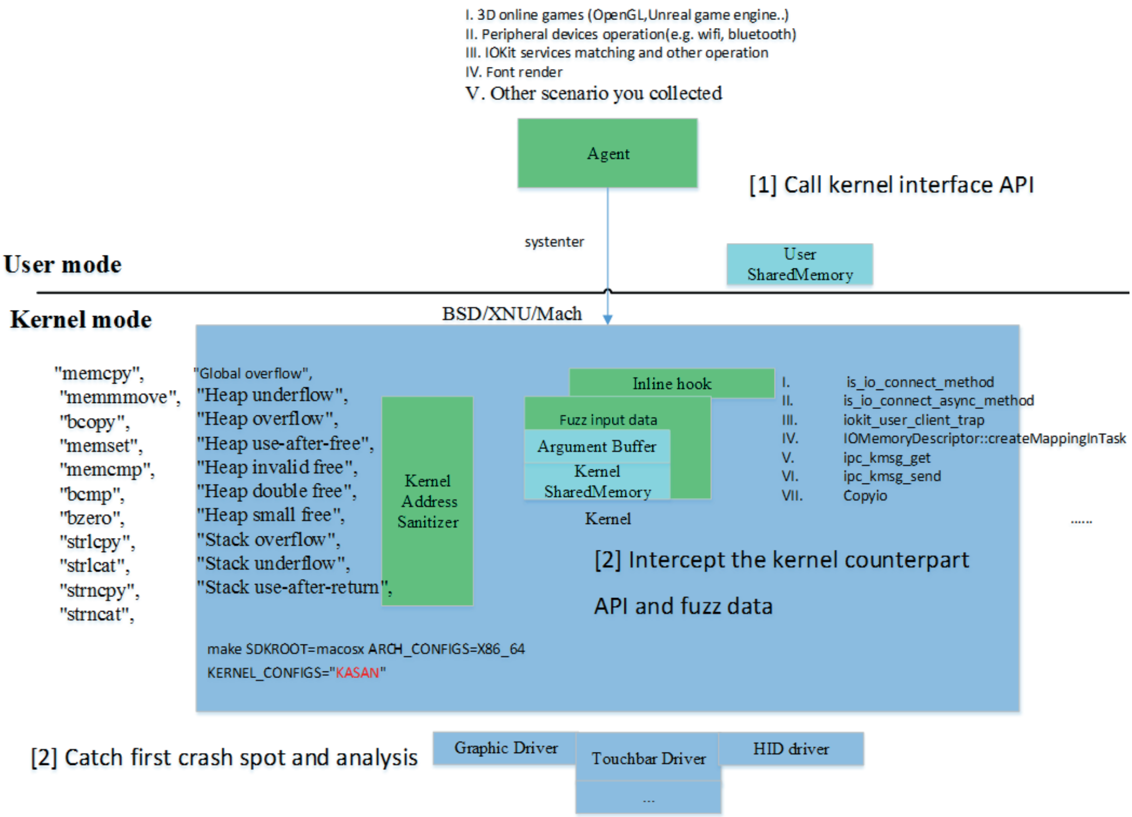


Figure 29: Enhanced kernel fuzz.

4.1 KASAN in iOS/OSX kernel

KASAN (kernel address sanitizer) is one of the standard sanitizers supported by popular operating systems including *OSX* which could help catch the crash spot so as to identify the root cause of the vulnerability and reproduce it much more easily.

What is lucky is that the KASAN feature is officially supported in XNU building, so you could build your own XNU kernel like this and replace the original in /System/Library/Kernels/:

```
make SDKROOT=macosx ARCH_CONFIGS=X86_64 KERNEL_CONFIGS="KASAN"
```

In fact, as the implementation of KASAN in XUN uses extra memory for tracing, it would guard any memory address allocating, freeing and referencing (e.g. memcpy, memcpy, bcopy) in the source code at instruction level such as variable in stack, heap and so on. In this way, the typical memory error such as buffer (stack, heap) overflow and UAF (use after free) in XUN could be caught in the first crash spot. Figure 30 shows the code pieces in __Xio_connect_method in the kernel.

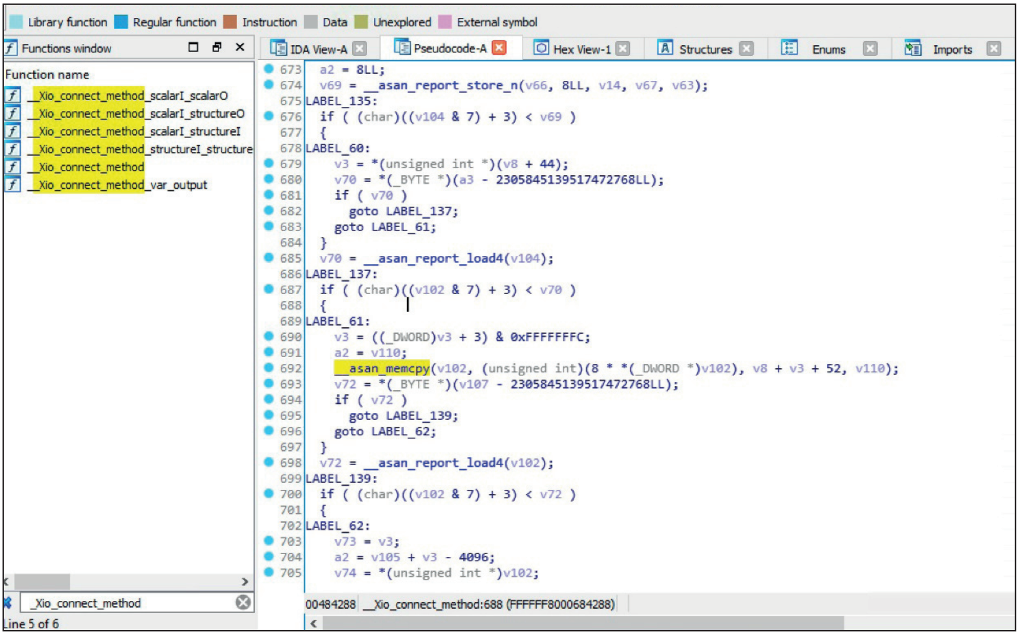


Figure 30: KASAN in __Xio_connect_method.

4.2 Inline hook and fuzz in kernel

As we have mentioned, we want to touch as much of the execution path (code coverage) as possible. In our experience, the typical kernel API could be one of the best hooking points, which contains IOKit control, memory share, mach msg method and system call.

When the CPU executes instructions to this kind of kernel API, many routine checks (e.g. send correct message id to the correct user client) have been made, which could reduce the useless blind fuzzing corpus and save fuzzing time.

Besides simple tampering with the input data, we could introduce a more advanced fuzzing method at this point. As part of our further research, we could locate our agent in kernel mode towards the kernel API and support code coverage feedback by static or dynamic instrumentation. You could imagine it as syzkaller or AFL in kernel mode.

4.3 Future plan

4.3.1 Syzkaller-like fuzzing in kernel mode

As we know, the runtime environment in the kernel would be complex. There would exist much environment preparation or initialization (e.g. open the correct service, initialize the target devices and send the correct mach message id) before a special kernel API (e.g. IOConnectionCallMethod) could work properly. So why don't we intercept the kernel API at the proper time under the proper state and fuzz it like AFL does, directly in kernel mode?

4.3.2 Porting KASAN/KMSAN for a closed-source driver

In fact, porting a kernel sanitizer mechanism to a closed-source driver on *iOS*/*OSX* is possible if we want expand the memory guard for the whole kernel mode. Every kernel module (including driver) would utilize the memory management service provided by the kernel via API (e.g. kmem_alloc, bcopy). Modifying the memory API to asan_* in the import table in the driver module, or patching the code with memory management in the driver to support the kernel sanitizer could be investigated in further research.

5. HUNT FOR 0-DAY VULNERABILITIES

5.1 CVE-2018-4462 – an integer overflow vulnerability which can leak kernel information found in the AMDFramebuffer driver

Figure 31 is a backtrace of the crash point. The extGetPixelInformation() method is one of the IOFramebufferUserClient methods whose selector is 1. This method takes three scalar data inputs, which are displayMode, depth and aperture, and returns the pixel information.

```
(lldb) bt
* thread #1, stop reason = signal SIGSTOP
  * frame #0: 0xffffff7f8d91e324 AMDFramebuffer`AMDFramebuffer::getPixelInformationFromTiming(AtiDetailedTimingInformation const&, IOPixelInformation*, int, int) + 388
    frame #1: 0xffffff7f8d91e180 AMDFramebuffer`AMDFramebuffer::getPixelInformation(int, int, IOPixelInformation*) + 112
    frame #2: 0xffffff7f8d91e0a5 AMDFramebuffer`AMDFramebuffer::getPixelInformation(int, int, int, IOPixelInformation*) + 101
    frame #3: 0xffffff7f8b42223d IOGraphicsFamily`IOFramebuffer::extGetPixelInformation(target=0xffffff869e59f000, reference=<unavailable>, args=<unavailable>) at IOFram
    frame #4: 0xffffff800aa4c478 kernel.development`IOUserClient::externalMethod(this=<unavailable>, selector=<unavailable>, args=0xffffffa756c8b988, dispatch=0xffffff7f
0000) at IOUserClient.cpp:5335 [opt]
    frame #5: 0xffffff7f8b437d0b IOGraphicsFamily`IOFramebufferUserClient::externalMethod(this=0xffffff80b8810800, selector=1, args=0xffffffa756c8b988, dispatch=<unavail
amebufferUserClient.cpp:380 [opt]
    frame #6: 0xffffff800aa553cf kernel.development`::is_io_connect_method(connection=0xffffff80b8810800, selector=1, scalar_input=<unavailable>, scalar_inputCnt=<unavai
put=0, ool_input_size=0, inband_output="", inband_outputCnt=0xffffff80ac2e2e0c, scalar_output=0xffffffa756c8bcb0, scalar_outputCnt=0xffffffa756c8bcac, ool_output=0, ool_
]
    frame #7: 0xffffff7f8e6c854b pasive_kernel_fuzz`trampline_is_io_connect_method(connection=0xffffff80b8810800, selector=1, scalar_input=0xffffff80b8b81e10, scalar_inp
_input_size=0, inband_output="", inband_outputCnt=0xffffff80ac2e2e0c, scalar_output=0xffffffa756c8bcb0, scalar_outputCnt=0xffffffa756c8bcac, ool_output=0, ool_output_siz
    frame #8: 0xffffff800a3f2bd4 kernel.development`_Xio_connect_method(InHeadP=<unavailable>, OutHeadP=0xffffff80ac2e2de0) at device_server.c:8379 [opt]
    frame #9: 0xffffff800a2c450d kernel.development`ipc_kobject_server(request=0xffffff80b8b81d70, option=<unavailable>) at ipc_kobject.c:359 [opt]
    frame #10: 0xffffff800a29124a kernel.development`ipc_kmsg_send(kmsg=0xffffff80b8b81d70, option=3, send_timeout=0) at ipc_kmsg.c:1822 [opt]
    frame #11: 0xffffff800a2b024f kernel.development`mach_msg_overwrite_trap(args=<unavailable>) at mach_msg.c:546 [opt]
    frame #12: 0xffffff7f8e6d81d7 pasive_kernel_fuzz`trampline_mach_msg_overwrite_trap(args=0xffffffa756c8bf08) at mach_msg_overwrite_trap_trampline.c:131
    frame #13: 0xffffff800a42cb09 kernel.development`mach_call_munger64(state=0xffffff80ac13de20) at bsd_i386.c:573 [opt]
    frame #14: 0xffffff800a25b466 kernel.development`hndl_mach_scall64 + 22
```

Figure 31: Backtrace of the integer overflow.

5.1.1 Root cause

Figure 32 is an assembly snippet of the crash point which is in the method AMDFramebuffer::getPixelInformationFromTiming(AtiDetailedTimingInformation const&, IOPixelInformation*, int, int). From the code snippet and the debug info, we can see that the register 'rdi = 0xffffffff2000001' is so big it is out of boundary. And after this buffer read operation, this function use the 'Utilities::str_copy' function to copy 'sizeof(IOPixelInformation*)' bits of pixel information to the caller, so it can leak the kernel information to the user-mode process.

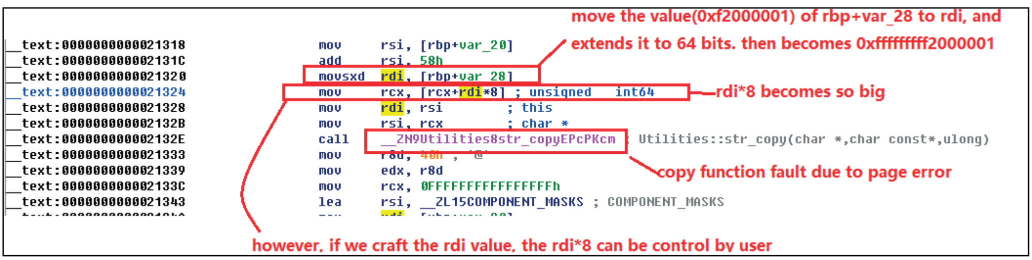


Figure 32: Code snippet for the crash point.

5.2 Untrusted pointer de-reference issue found in IntelAccelerator

Figure 33 shows the backtrace of this untrusted pointer de-reference issue. This vulnerability can be triggered on *Mac mini*. 'IntelAccelerator' is a service delivered by the *Intel* graphics driver, it can be opened from a user-mode process.

```
(lldb) bt
* thread #1, stop reason = signal SIGSTOP
  * frame #0: 0xffffff8013da405b kernel.development`memcpy + 11
    frame #1: 0xffffff7f964b18c7 AppleIntelHD5000Graphics`IntelAccelerator::newGTT(unsigned int**, bool, IGAccelTask&) + 173
    frame #2: 0xffffff7f9649a330 AppleIntelHD5000Graphics`IntelPPGTT::init(IntelAccelerator&, bool, IGAccelTask&) + 24
    frame #3: 0xffffff7f964a2c88 AppleIntelHD5000Graphics`IGAccelTask::prepare(IntelAccelerator&) + 38
    frame #4: 0xffffff7f964b17c7 AppleIntelHD5000Graphics`IntelAccelerator::createUserGPUTask() + 219
    frame #5: 0xffffff7f95083cca IOAcceleratorFamily2`IOAccelShared2::init(IOGraphicsAccelerator2*, task*) + 58
    frame #6: 0xffffff7f9509848d IOAcceleratorFamily2`IOGraphicsAccelerator2::createShared(task*) + 51
    frame #7: 0xffffff7f95087051 IOAcceleratorFamily2`IOAccelSharedUserClient2::sharedStart() + 43
    frame #8: 0xffffff7f964933a6 AppleIntelHD5000Graphics`IGAccelSharedUserClient::sharedStart() + 22
    frame #9: 0xffffff7f95085254 IOAcceleratorFamily2`IOAccelSharedUserClient2::start(IOService*) + 156
    frame #10: 0xffffff7f95097db0 IOAcceleratorFamily2`IOGraphicsAccelerator2::newUserClient(task*, void*, unsigned int, IOUserClient**) + 1088
    frame #11: 0xffffff80145f1871 kernel.development`IOService::newUserClient(this=0xffffff8036144800, owningTask=0xffffff8042c36840, securityID=0xf
fffff8042c36840, type=6, properties=0x0000000000000000, handler=0xffffff921acdbce0) at IOService.cpp:5851 [opt]
    frame #12: 0xffffff80146542d0 kernel.development`::is_io_service_open_extended(_service=0xffffff8036144800, owningTask=0xffffff8042c36840, conne
ct_type=6, ndr=<unavailable>, properties=<unavailable>, propertiesCnt=<unavailable>, result=0xffffff803ca0e0b8, connection=0xffffff921acdbd30) at IO
UserClient.cpp:3468 [opt]
    frame #13: 0xffffff8013ff2662 kernel.development`_Xio_service_open_extended(InHeadP=0xffffff803ca0e260, OutHeadP=0xffffff803ca0e07c) at device_s
erver.c:8003 [opt]
    frame #14: 0xffffff8013ec450d kernel.development`ipc_kobject_server(request=0xffffff803ca0e200, option=<unavailable>) at ipc_kobject.c:359 [opt]
    frame #15: 0xffffff8013e9124a kernel.development`ipc_kmsg_send(kmsg=0xffffff803ca0e200, option=3, send_timeout=0) at ipc_kmsg.c:1822 [opt]
    frame #16: 0xffffff8013eb024f kernel.development`mach_msg_overwrite_trap(args=<unavailable>) at mach_msg.c:546 [opt]
    frame #17: 0xffffff7f9749e1d7
    frame #18: 0xffffff801402c7c3 kernel.development`mach_call_munger(state=<unavailable>) at bsd_i386.c:481 [opt]
    frame #19: 0xffffff8013e5b222 kernel.development`hndl_mach_scall + 210
(lldb) dis
kernel.development`memcpy:
    0xffffff8013da4050 <+0>:  movq   %rdi, %rax
    0xffffff8013da4053 <+3>:  movq   %rdx, %rcx
    0xffffff8013da4056 <+6>:  shrq   $0x3, %rcx
    0xffffff8013da405a <+10>: cld
->  0xffffff8013da405b <+11>: rep    movsq      (%rsi), %es:(%rdi)
    0xffffff8013da405e <+14>: movq   %rdx, %rcx
    0xffffff8013da4061 <+17>: andq   $0x7, %rcx
    0xffffff8013da4065 <+21>: rep    movsb      (%rsi), %es:(%rdi)
    0xffffff8013da4067 <+23>: retq
    0xffffff8013da4068 <+24>: nopl   (%rax,%rax)
(lldb) register read $rdi
     rdi = 0x0000000000000000
```

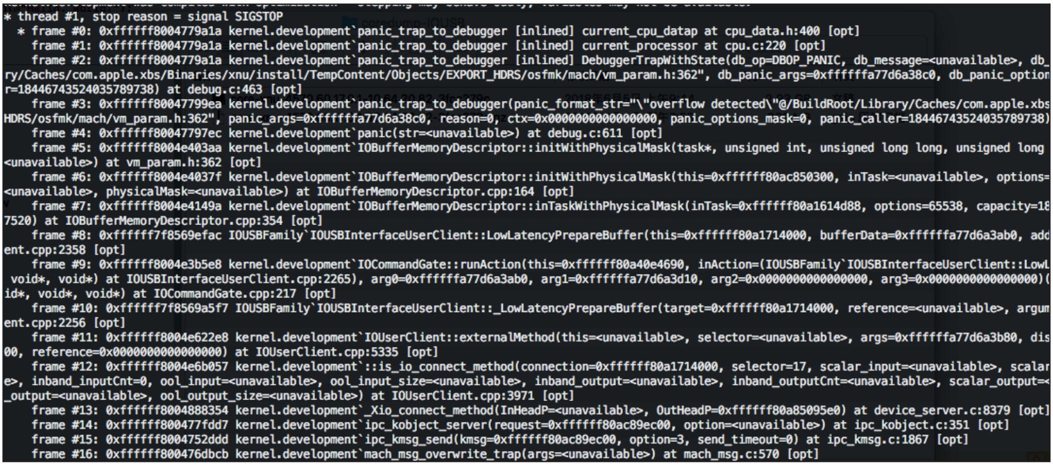
Figure 33: Crash info for the intelAccelerator driver NULL PAGE read operation.

5.2.1 Root cause

Figure 34 shows the arguments list for the 'newUserClient' function when this service is opened. When connection type is 6 and the properties are NULL, this vulnerability will be triggered.

```
(lldb) f 11
frame #11: 0xffffff80145f1871 kernel.development`IOService::newUserClient(this=0xffffff8036144800, owningTa
f8042c36840, type=6, properties=0x0000000000000000, handler=0xffffff921acdbce0) at IOService.cpp:5851 [opt]
(lldb) fr v
(IOService *) this = 0xffffff8036144800
(task_t) owningTask = 0xffffff8042c36840
(void *) securityID = 0xffffff8042c36840
(UInt32) type = 6
(OSDictionary *) properties = 0x0000000000000000
(IOUserClient **) handler = 0xffffff921acdbce0
(OSObject *) prop = <variable not available>

(const OSSymbol *) userClientClass = <variable not available>

(OSObject *) temp = <variable not available>

(IOUserClient *) client = <variable not available>

(lldb)
```

Figure 34: Arguments list for newUserClient function when this bug is triggered.



```
* thread #1, stop reason = signal SIGSTOP
  * frame #0: 0xffffff8004779a1a kernel.development`panic_trap_to_debugger [inlined] current_cpu_datap at cpu_data.h:400 [opt]
    frame #1: 0xffffff8004779a1a kernel.development`panic_trap_to_debugger [inlined] current_processor at cpu.c:220 [opt]
    frame #2: 0xffffff8004779a1a kernel.development`panic_trap_to_debugger [inlined] DebuggerTrapWithState(db_op=DBOP_PANIC, db_message=<unavailable>, db_
ry/Caches/com.apple.xbs/Binaries/xnu/install/TempContent/Objects/EXPORT_HDRS/osfmk/mach/vm_param.h:362", db_panic_args=0xffffffa77d6a38c0, db_panic_option
r=18446743524035789738) at debug.c:463 [opt]
    frame #3: 0xffffff80047799ea kernel.development`panic_trap_to_debugger(panic_format_str="\"overflow detected\"@/BuildRoot/Library/Caches/com.apple.xbs
HDRS/osfmk/mach/vm_param.h:362", panic_args=0xffffffa77d6a38c0, reason=0, ctx=0x0000000000000000, panic_options_mask=0, panic_caller=18446743524035789738)
    frame #4: 0xffffff80047797ec kernel.development`panic(str=<unavailable>) at debug.c:611 [opt]
    frame #5: 0xffffff8004e403aa kernel.development`IOBufferMemoryDescriptor::initWithPhysicalMask(task*, unsigned int, unsigned long long, unsigned long
<unavailable>) at vm_param.h:362 [opt]
    frame #6: 0xffffff8004e4037f kernel.development`IOBufferMemoryDescriptor::initWithPhysicalMask(this=0xffffff80ac850300, inTask=<unavailable>, options=
<unavailable>, physicalMask=<unavailable>) at IOBufferMemoryDescriptor.cpp:164 [opt]
    frame #7: 0xffffff8004e4149a kernel.development`IOBufferMemoryDescriptor::inTaskWithPhysicalMask(inTask=0xffffff80a1614d88, options=65538, capacity=18
7520) at IOBufferMemoryDescriptor.cpp:354 [opt]
    frame #8: 0xffffff7f8569efac IOUSBFamily`IOUSBInterfaceUserClient::LowLatencyPrepareBuffer(this=0xffffff80a1714000, bufferData=0xffffffa77d6a3ab0, add
ent.cpp:2358 [opt]
    frame #9: 0xffffff8004e3b5e8 kernel.development`IOCommandGate::runAction(this=0xffffff80a40e4690, inAction=(IOUSBFamily`IOUSBInterfaceUserClient::LowL
 void*, void*) at IOUSBInterfaceUserClient.cpp:2265), arg0=0xffffffa77d6a3ab0, arg1=0xffffffa77d6a3d10, arg2=0x0000000000000000, arg3=0x0000000000000000)(
id*, void*, void*) at IOCommandGate.cpp:217 [opt]
    frame #10: 0xffffff7f8569a5f7 IOUSBFamily`IOUSBInterfaceUserClient::_LowLatencyPrepareBuffer(target=0xffffff80a1714000, reference=<unavailable>, argum
ent.cpp:2256 [opt]
    frame #11: 0xffffff8004e622e8 kernel.development`IOUserClient::externalMethod(this=<unavailable>, selector=<unavailable>, args=0xffffffa77d6a3b80, dis
00, reference=0x0000000000000000) at IOUserClient.cpp:5335 [opt]
    frame #12: 0xffffff8004e6b057 kernel.development`::is_io_connect_method(connection=0xffffff80a1714000, selector=17, scalar_input=<unavailable>, scalar
e>, inband_inputCnt=0, ool_input=<unavailable>, ool_input_size=<unavailable>, inband_output=<unavailable>, inband_outputCnt=<unavailable>, scalar_output=<
_output=<unavailable>, ool_output_size=<unavailable>) at IOUserClient.cpp:3971 [opt]
    frame #13: 0xffffff8004888354 kernel.development`_Xio_connect_method(InHeadP=<unavailable>, OutHeadP=0xffffff80a85095e0) at device_server.c:8379 [opt]
    frame #14: 0xffffff800477fdd7 kernel.development`ipc_kobject_server(request=0xffffff80ac89ec00, option=<unavailable>) at ipc_kobject.c:351 [opt]
    frame #15: 0xffffff8004752ddd kernel.development`ipc_kmsg_send(kmsg=0xffffff80ac89ec00, option=3, send_timeout=0) at ipc_kmsg.c:1867 [opt]
    frame #16: 0xffffff800476dbcb kernel.development`mach_msg_overwrite_trap(args=<unavailable>) at mach_msg.c:570 [opt]
```

Figure 35: Backtrace for the crash point.

5.3 OverFlow issue due to no boundary check in IOUSBFamily extension

The IOUSBFamily driver provides an external method for a user-mode process which is called IOUSBFamily`IOUSBInterfaceUserClient::LowLatencyPrepareBuffer. This function can be called by IOConnectCallMethod with selector 17. Figure 35 shows the backtrace of the crash point.

5.3.1 Root cause

A capacity argument is needed for this function. The IOUSBInterfaceUserClient::_ LowLatencyPrepareBuffer function will copy the input_scalar[0~4] data for the IOUSBFamily`IOUSBInterfaceUserClient::LowLatencyPrepareBuffer function directly. But the input scalar content is transferred from user space, so we can control the capacity to a large degree to trigger this bug.

5.4 Divide zero issue found in AMDRadeonX4000_AMDAccelResource class

IOAccelCommandQueue is used to process the graphic accelerator command information for 3D rendering. This vulnerability occurred in selector 1 whose function name is 'IOAccelCommandQueue::s_submit_command_buffers' with open type 9. When an AMDRadeonX4000 driver processes these command, it will prepare the AMDAccelResource first. However, there are many divide operations in this process, and a lack of zero checking.

These vulnerabilities were found in the latest *MacOS* (10.14.3) system. Listing 4 shows the backtrace of this bug.

5.4.1 Root cause

Listing 5 shows an assembly code snippet of this vulnerable function. The r12d register is initialized with zero. It will be assigned a new value within the omitted code if it meets some condition, but this is not certain, therefore, it will result in a divide zero bug in the place b.

```
(lldb) bt
* thread #1, stop reason = signal SIGSTOP
    frame #0: 0xffffff7f88b04941 AMDRadeonX4000'BltMgr::HwlOptimizeBufferBltRects(BltInfo*, 
unsigned int) + 879
    frame #1: 0xffffff7f88b1c474 AMDRadeonX4000'SiBltMgr::Adjust3dBltInfo(BltInfo*) + 662
    frame #2: 0xffffff7f88b1bfe2 AMDRadeonX4000'SiBltMgr::Execute3dBlt(BltInfo*) + 76
    frame #3: 0xffffff7f88b04241 AMDRadeonX4000'BltMgr::Memset(BltDevice*, _UBM_
MEMSETINFO*) + 753
    frame #4: 0xffffff7f88a75506 AMDRadeonX4000'AMDRadeonX4000_AMDBltMgr::Memset(_UBM_
MEMSETINFO*, _UBM_E_RETURNCODE*) + 28
    frame #5: 0xffffff7f88a73d29 AMDRadeonX4000'AMDRadeonX4000_
AMDAtomicBlitManager::doMemset(_UBM_MEMSETINFO*, ABM_OPTIONS const*) + 263
    frame #6: 0xffffff7f88a56478 AMDRadeonX4000'AMDRadeonX4000_
AMDAccelResource::initFillRegions() + 390
    frame #7: 0xffffff7f88a5c9ac AMDRadeonX4000'AMDRadeonX4000_AMDAccelResource::prepare() 
+ 108
    frame #8: 0xffffff7f889b7c3e IOAcceleratorFamily2'IOAccelSegmentResourceList::prepare() 
+ 48
    frame #9: 0xffffff7f889cbe94 
IOAcceleratorFamily2'IOAccelCommandQueue::coalesceSegment(IOAccelCommandQueueSegment*, 
unsigned int*, IOAccelSegmentResourceList*, IOAccelKernelCommand const*, 
IOAccelKernelCommand const*) + 78
    frame #10: 0xffffff7f889cc1ce 
IOAcceleratorFamily2'IOAccelCommandQueue::processCommandBuffer(unsigned int, unsigned 
int) + 666
    frame #11: 0xffffff7f889cd188 IOAcceleratorFamily2'IOAccelCommandQueue::process_
command_buffer(unsigned int, unsigned int) + 924
    frame #12: 0xffffff7f889cb4c0 IOAcceleratorFamily2'IOAccelCommandQueue::submit_
command_buffer(unsigned int, unsigned int, unsigned long long, unsigned long long) + 252
    frame #13: 0xffffff7f889cb2b9 IOAcceleratorFamily2'IOAccelCommandQueue::submit_
command_buffers(IOAccelCommandQueueSubmitArgs const*) + 827
    frame #14: 0xffffff7f889ca2f4 IOAcceleratorFamily2'IOAccelCommandQueue::s_submit_
command_buffers(IOAccelCommandQueue*, void*, IOExternalMethodArguments*) + 250
    frame #15: 0xffffff8006224978 kernel.
development'IOUserClient::externalMethod(this=<unavailable>, selector=<unavailable>, 
args=0xffffffa753d3b9b8, dispatch=0xffffff7f889fac68, target=<unavailable>, 
reference=<unavailable>) at IOUserClient.cpp:5689 [opt]
  * frame #16: 0xffffff800622da02 kernel.development'::is_io_connect_
method(connection=<unavailable>, selector=1, scalar_input=<unavailable>, scalar_
inputCnt=<unavailable>, inband_input=<unavailable>, inband_inputCnt=32, ool_input=0, 
ool_input_size=0, inband_output="", inband_outputCnt=0xffffff80b137560c, scalar_
output=0xffffffa753d3bce0, scalar_outputCnt=0xffffffa753d3bcdc, ool_output=0, ool_output_
size=0xffffff80b1ee3138) at IOUserClient.cpp:4304 [opt]
    frame #17: 0xffffff7f8c504e32
    frame #18: 0xffffff8005bbc386 kernel.development'_Xio_connect_
method(InHeadP=<unavailable>, OutHeadP=0xffffffa753d3bce0) at device_server.c:8379 [opt]
    frame #19: 0xffffff8005a948fd kernel.development'ipc_kobject_
server(request=0xffffff80b1ee3050, option=3) at ipc_kobject.c:361 [opt]
    frame #20: 0xffffff8005a6088e kernel.development'ipc_kmsg_send(kmsg=0xffffff80b1ee3050, 
option=3, send_timeout=0) at ipc_kmsg.c:1868 [opt]
    frame #21: 0xffffff8005a800e3 kernel.development'mach_msg_overwrite_
trap(args=<unavailable>) at mach_msg.c:553 [opt]
    frame #22: 0xffffff8005bf702b kernel.development'mach_call_
   munger64(state=0xffffff80acb77100) at bsd_i386.c:580 [opt]    frame #23: 
   0xffffff8005a2a476 kernel.development'hndl_mach_scall64 + 22
```

Listing 4: Backtrace of the bug.

```
__text:00000000000BB75B                 div     esi
__text:00000000000BB75D                 mov     r14d, 0
__text:00000000000BB763                 mov     r12d, 0           -----init
r12d with 0                                 --(a)
__text:00000000000BB769                 test    edx, edx
  …..
  -----omitted code ----
  …..
__text:00000000000BB93C loc_BB93C:                                ; CODE XREF:
BltMgr::HwlOptimizeBufferBltRects(BltInfo *,uint)+3E1j
__text:00000000000BB93C                 xor     edx, edx
__text:00000000000BB93E                 mov     eax, r13d
__text:00000000000BB941                 div     r12d                 -----r12d
is not always nonzero              ---(b)
__text:00000000000BB944                 cmp     eax, r14d
__text:00000000000BB947                 jbe     short loc_BB95B
__text:00000000000BB949                 mov     dword ptr [rsi+rbx-0Ch], 0
__text:00000000000BB951                 mov     [rsi+rbx-4], r12d
__text:00000000000BB956                 mov     eax, r14d
__text:00000000000BB959                 jmp     short loc_BB97C
```

Listing 5: Asm code snippet of the BltMgr::HwlOptimizeBufferBltRects function.

5.5 OOB read in AMDRadeonX4000 extension

AMDRadeonX4000_AMDAccelResource is used to process the graphic accelerator resource information for 3D rendering. This vulnerability occurred in selector 0, whose function name is 'IOAccelSharedUserClient2::s_new_resource' with open type 6. This vulnerability was found in the latest *MacOS* (10.14.3) system.

Listing 6 shows the backtrace of this OOB bug.

5.5.1 Root cause

As shown in Listing 7, the register of rax is the address of the buffer which is created from the IOMalloc function. The r15 register points to the structureInput buffer which is controlled by user mode. The ecx register stores the length of the IOMalloc buffer. The rdx register is used as an index to copy the structureInput buffer content to the IOMalloc buffer. However, here, ecx is obtained directly from user mode which is structureInput at offset 62 dword. So, if we set ecx to a high value, it will read overflow from the structureInput buffer.

REFERENCES

[1] Angr. https://github.com/angr/angr.

[2] Miasm. https://github.com/cea-sec/miasm.

[3] ARM Architecture Reference Manual. https://cs.nyu.edu/courses/spring18/CSCI-GA.2130-001/ARM/arm_arm.pdf.

[4] Joker. http://www.newosxbook.com/tools/joker.html.

```
* thread #1, stop reason = signal SIGSTOP
  * frame #0: 0xffffff7fa00965d3 AMDRadeonX4000'AMDRadeonX4000_
AMDAccelResource::initialize(IOAccelNewResourceArgs*, unsigned long long) +
1525
  frame #1: 0xffffff7f9fea346b IOAcceleratorFamily2'IOAccelSharedUserClient2::new_
resource(IOAccelNewResourceArgs*, IOAccelNewResourceReturnData*, unsigned long
long, unsigned int*) + 1893
  frame #2: 0xffffff7f9fea4a41 IOAcceleratorFamily2'IOAccelSharedUserClient2::s_
new_resource(IOAccelSharedUserClient2*, void*, IOExternalMethodArguments*) +
151
  frame #3: 0xffffff801d625ab8 kernel.
development'IOUserClient::externalMethod(this=<unavailable>,
selector=<unavailable>, args=0xffffff83dd4b3b58, dispatch=0xffffff7f9fee8260,
target=0xffffff80854fd780, reference=0x0000000000000000) at IOUserClient.cpp:5358
[opt]
  frame #4: 0xffffff7f9fea4d98
IOAcceleratorFamily2'IOAccelSharedUserClient2::externalMethod(unsigned int,
IOExternalMethodArguments*, IOExternalMethodDispatch*, OSObject*, void*) + 120
  frame #5: 0xffffff801d62eb7f kernel.development'::is_io_connect_
method(connection=0xffffff80854fd780, selector=0, scalar_input=<unavailable>,
scalar_inputCnt=<unavailable>, inband_input=<unavailable>, inband_
inputCnt=2424, ool_input=0, ool_input_size=0, inband_output="", inband_
outputCnt=0xffffff806ba03e0c, scalar_output=0xffffff83dd4b3ce0, scalar_
outputCnt=0xffffff83dd4b3cdc, ool_output=0, ool_output_size=0xffffff8085919d5c) at
IOUserClient.cpp:3994 [opt]
  frame #6: 0xffffff801cfbbce4 kernel.development'_Xio_connect_
method(InHeadP=<unavailable>, OutHeadP=0xffffff806ba03de0) at device_
server.c:8379 [opt]
  frame #7: 0xffffff801ce8d27d kernel.development'ipc_kobject_
server(request=0xffffff8085919000, option=<unavailable>) at ipc_kobject.c:359
[opt]
  frame #8: 0xffffff801ce59465 kernel.development'ipc_kmsg_
send(kmsg=0xffffff8085919000, option=3, send_timeout=0) at ipc_kmsg.c:1832 [opt]
  frame #9: 0xffffff801ce78a75 kernel.development'mach_msg_overwrite_
trap(args=<unavailable>) at mach_msg.c:549 [opt]
  frame #10: 0xffffff801cff6323 kernel.development'mach_call_
munger64(state=0xffffff806ca9c480) at bsd_i386.c:573 [opt]
  frame #11: 0xffffff801ce23486 kernel.development'hndl_mach_scall64 + 22
```

Listing 6: The backtrace of this OOB bug.

[5] CodeRefsFrom(ea, flow). https://www.hex-rays.com/products/ida/support/idapython_docs/idautils-module.html#CodeRefsFrom.

[6] BinDiff. https://www.zynamics.com/bindiff.html.

[7] Meld. http://meldmerge.org/.

[8] iOS Security Guide. https://images.apple.com/business/docs/iOS_Security_Guide.pdf.

```
__text:000000000000E58E loc_E58E:            ; CODE XREF: AMDRadeonX4000_
AMDAccelResource::initialize(IOAccelNewResourceArgs *,ulong long)+58Dj
__text:000000000000E58E                 mov     ecx, [r15+0F8h]
__text:000000000000E595                 test    rcx, rcx
__text:000000000000E598                 jz      short loc_E603
__text:000000000000E59A                 shl     rcx, 3
__text:000000000000E59E                 lea     rdi, [rcx+rcx*2]
__text:000000000000E5A2                 call    _IOMalloc
__text:000000000000E5A7                 mov     [r12+178h], rax  --- rax==
buffer address which create by IOMalloc
__text:000000000000E5AF                 test    rax, rax
__text:000000000000E5B2                 jz      short loc_E62A
__text:000000000000E5B4                 or      byte ptr [r12+186h], 8
__text:000000000000E5BD                 mov     ecx, [r15+0F8h]
--------r15==structureInput, ecx=( (uint32_t*) structureInput+62)
__text:000000000000E5C4                 mov     [r12+180h], ecx
__text:000000000000E5CC                 test    rcx, rcx
__text:000000000000E5CF                 jz      short loc_E639
__text:000000000000E5D1                 xor     edx, edx
__text:000000000000E5D3
__text:000000000000E5D3 loc_E5D3:          ; CODE XREF: AMDRadeonX4000_
AMDAccelResource::initialize(IOAccelNewResourceArgs *,ulong long)+621j
__text:000000000000E5D3                 mov     rsi, [r15+rdx+98h]  ---- mov
structureInput+rdx+0x98 to rsi
__text:000000000000E5DB                 mov     [rax+rdx], rsi  ----mov rsi
to rax+rdx, rax== buffer address which create by IOMalloc
__text:000000000000E5DF                 mov     rsi, [r15+rdx+0A0h]
__text:000000000000E5E7                 mov     [rax+rdx+8], rsi
__text:000000000000E5EC                 mov     esi, [r15+rdx+0A8h]
__text:000000000000E5F4                 mov     [rax+rdx+10h], esi
__text:000000000000E5F8                 add     rdx, 18h
__text:000000000000E5FC                 dec     rcx
__text:000000000000E5FF                 jnz     short loc_E5D3
```

Listing 7: Asm code snippet of AMDRadeonX4000_AMDAccelResource::initialize.