Optimized Fuzzing IOKit in iOS

Lei Long, Peng Xiao
Mobile Security of Alibaba
longlei.ll@alibaba-inc.com, xiaopeng.xp@alibaba-inc.com

Abstract：

In recent years, lots of kernel vulnerabilities in iOS are revealed, among which more than 50% lie in kernel extensions and the percentage is increasing year by year. We believe that there are still lots of unrevealed vulnerabilities in kernel extensions of iOS system. Fuzzing is the most common way of exploiting vulnerabilities, and IOKit is an ideal target in kernel extensions for fuzzing . The interfaces in IOKit use specific structures, such as IOExternalMethod, IOExternalMethodDispatch, to check the input parameters in various ways. Purely random inputs when fuzzing IOKit can hardly pass the interfaces' parameter checking, so that most of fuzzing data cannot reach the kernel IOUserClient subclass at all. Thus such kind of fuzzing is inefficient.

One way to improve such a blind fuzzing method is to use the static information exported by sMethod symbols which can be dumped by a static analysis tools such as IDA. However it may not always be available because in iOS 6 there are just a little amount of drivers that reserve sMethod symbols, and since iOS 7 all the symbols of IOExternalMethod and IOExternalMethodDispatch are totally removed from the kernel by Apple. This means that the static information of sMethod symbols is not available since iOS 7.

In this paper we will introduce an approach to resolve the symbols and parameter information dynamically based on a kernel patch to read and write memories. In the approach we can exploit quite a lot of useful information, including not only the standard parameters of IOKit interfaces, but also other supplementary data . We have also built a fuzzing framework, which uses the resolved information and generates the random inputs which can pass the basic parameter checking by IOKit interfaces. Therefore the fuzzing can be done efficiently. Although the inputs pass the parameter checking by IOKit interface, they still might lead to kernel panic in the corresponding driver or somewhere else. The fuzzing framework then collects the panic logs for further analysis.

At the end of this paper, we also present the information of IOKit interfaces exported by our approach, and several typical vulnerabilities found by our fuzzing framework.

1. Information Export

Class IOUserClient is the most important base class to provide IOKit interfaces from kernel mode to user mode, which is overridden by the kernel drivers to provide interfaces functionalities. In this chapter, the information of all OSObject subclasses in the kext (kernel extension) can be dynamically exported, definitely including IOUserClient subclasses which is used in fuzzing IOKit. The export information includes:

(1) the basic information of IOUserClient subclasses, such as class name, class size, vtable(virtual methods table) address, inherited relationships, virtual method overridden symbols, etc.
(2) the key parameters to open the services of a specific IOUserClient subclass, such as service name and open type.
(3) the IOExternalMethodDispatch table, which is used to check the parameters when Function externalMethod() is overridden by the kernel IOUserClient subclass to obtain the input from user mode.
(4) the IOExternalMethod table, which is used to check the parameters when Function getTargetAndMethodForIndex() or getExternalMethodForIndex() is overridden by the kernel IOUserClient subclass to obtain the input from user mode.

1.1 Basic Information

How to export the basic information of all OSObject subclasses in the kext is detailed in this section step by step:

(1) Through kernel memory read operations, obtain the __DATA__const address and Bundle Info of all kexts.
(2) Obtain the class information and vtable of each OSObject subclass from its corresponding __DATA__const memory space.
(3) Obtain virtual method symbols of OSObject subclasses in the kext, by symbolization of all classes' virtual methods in the kernel.

1.1.1 Kexts' __DATA __const

It is discovered that vtables of all the OSObject subclasses are stored in the __DATA__const memory space in the kexts. Firstly, locate the __DATA__const addresses of all kexts.

(1) Based on tfp0 patch in iOS, the kernel memory can be read. Function task_for_pid() is used to obtain the kernel memory-read capability, as shown in TextBlock 1.

(2) In CRReadAtAddress() in our project, Function vm_read_overwrite() is used to read 256 bytes memory space each time, and read the left-size memory space less than 256 bytes at the last time, as shown in TextBlock 1.

```
void CRReadAtAddress(vm_address_t _address,vm_size_t _size,void
**buffer) {
        ……
        kern_return_t kr = task_for_pid(mach_port_t sel, int curPid, &ker-
nel_task);
        ……
        if(_size > 256) {
           …
            kern_return_t ret = vm_read_overwrite(…, leftSize, …);
            …
        } else {
            kern_return_t ret = vm_read_overwrite(kernel_task, address_-
long, _size, (vm_address_t)*buffer, &outsize);
            …
       }
   }
```

TextBlock 1

(3) Traverse the kernel_slider by slider_byte from 256 to 1, and if the memory pointed by the current kernel_slider is equal to 0xfeedface, where 0xfeedface is the magic code of Mach-O in iOS, and this Mach-O header includes a key-value pair (segname, __PRELINK_TEXT), then the kernelcache Mach-O address in the kext is obtained, as shown in TextBlock 2.

```
(1)kernel_slider=0x01000000+slider_byte*0x00200000,
slider_byte={256,…,1}.
(2)*kernel_slider=0xfeedface.
(3)Mach-O segname=__PRELINK_TEXT.
```

TextBlock 2

(4) Resolve the kernelcache Mach-O header, and locate its section with segname=__PRELINK_TEXT, then the value of addr in this section is the kext Mach-O 0xfeedface address, as shown in TextBlock 3. Each LoadCommand may have one or more sections, while there is only 1 section shown in TextBlock 3. Kexts are stored in the memory which is pointed by each __PRELINK_TEXT addr, in the form of continuous Mach-O files, and addr in this section is the first kext Mach-O 0xfeedface address.

```
Load command 4
        cmd LC_SEGMENT
        cmdsize 124
        segname __PRELINK_TEXT
        vmaddr 0x8044f000
        vmsize 0x00a9f000
        fileoff 4218880
        filesize 11137024
        maxprot 0x00000003
        initprot 0x00000003
        nsects 1
        flags 0x0
Section
        sectname __text
        segname __PRELINK_TEXT
        addr 0x8044f000
        size 0x00a9f000
        offset 4218880
        align 2^0 (1)
        reloff 0
        nreloc 0
        flags 0x00000000
        reserved1 0
        reserved2 0
```

TextBlock 3

(5) After obtaining all 0xfeedface addresses, i.e. addr in TextBlock 3, all kexts' Mach-O header can be resolved, and then __DATA__const of kexts can be obtained:

Traversing all kexts' Mach-O header sections, if a section's sectname=__const and segname=__DATA, then the addr and size in this section is the addr and size of the kext' __DATA__const, as shown in TextBlock 4.

```
Section
  sectname __const
   segname __DATA
      addr 0x80383000
      size 0x0000e630
    offset 3678208
     align 2^12 (4096)
    reloff 0
    nreloc 0
     flags 0x00000000
 reserved1 0
 reserved2 0
```

TextBlock 4

1.1.2 OSObject Subclasses' Vtable

In the kernel, the vtable of each OSObject subclass is stored in the memory space of its Mach-O __DATA__const, including kernelcache main Mach-O and kext Mach-O. Class OSObject is inherited from base class OSMetaClassBase, and OSMetaClassBase owns the runtime information in the form of OSMetaClass object, including class name, class size, super class, etc. So we can use Function OSMetaClassBase::getMetaClass() to return the OSMetaClass object, and resolve the information of its corresponding class.

In Class OSMetaClassBase, Function getMetaClass() is a virtual method as shown in TextBlock 5, and OSObject subclasses override it to return the corresponding OSMetaClass object. If the implementations of overriding getMetaClass() in all OSObject subclasses are the same, a stationary instruction parsing algorithm can be used to return the OSMetaClass object address, so does it in iOS.

```
class OSMetaClassBase
        …
        virtual const OSMetaClass * getMetaClass() const = 0;
        …
}
```

TextBlock 5

TextBlock 6 is the annotations in OSMetaClass.h in XNU 10.10, which defines that all OSObject subclasses in the kext must use one of these two macros declared here to implement OSMetaClass runtime. The macro OSDeclareCommonStructors is used for class declaration, while the

```
 * While kernel extensions rarey interact directly with OSMetaClass at
run time,
 * they must register their classes with the metaclass system
 * using the macros declared here.
 * The class declaration should use one of these two macros
 * before its first member function declaration:
 * <ul>
 * <li><code>@link OSDeclareDefaultStructors OSDeclareDefaultStruc-
tors@/link</code> -
 *     for classes with no abstract member function declarations</li>
 * <li><code>@link OSDeclareAbstractStructors OSDeclareAbstract-
Structors@/link</code> -
 *     for classes with at least one abstract member function
declaration</li>
 * <li><code>@link OSDeclareFinalStructors OSDeclareFinalStructors@/
link</code> -
 *     for classes that should not be subclassable by another kext</li>
 * </ul>
```

TextBlock 6

macro OSDefineMetaClassWithInit is used for class initialization, as shown in TextBlock 7, where Function getMetaClass() is defined here to return the address of OSMetaClass object.

```
#define OSDeclareCommonStructors(className)                    \
    private:                                                   \
    static const OSMetaClass * const superClass;               \
    public:                                                    \
    static const OSMetaClass * const metaClass;                \
        static class MetaClass : public OSMetaClass {          \
        public:                                                \
            MetaClass();                                        \
            virtual OSObject *alloc() const;                    \
        } gMetaClass;                                           \
        friend class className ::MetaClass;                     \
        virtual const OSMetaClass * getMetaClass() const;       \
    protected:                                                 \
    className (const OSMetaClass *);                            \
    virtual ~ className ()

#define OSDefineMetaClassWithInit(className, superclassName, init)
\
    /* Class global data */                                        \
    className ::MetaClass className ::gMetaClass;                          \
    const OSMetaClass * const className ::metaClass =
\
        & className ::gMetaClass;                                   \
    const OSMetaClass * const className ::superClass =
\
        & superclassName ::gMetaClass;                               \
    /* Class member functions */                                    \
    className :: className(const OSMetaClass *meta)
\
        : superclassName (meta) { }                                 \
    className ::~ className() { }                                   \
    const OSMetaClass * className ::getMetaClass() const
\
        { return &gMetaClass; }                                     \
    /* The ::MetaClass constructor */                               \
    className ::MetaClass::MetaClass()                              \
        : OSMetaClass(#className, className::superClass, sizeof(class-
Name))    \
        { init; }
```

TextBlock 7

The ARM instructions of getMetaClass() are shown in TextBlock 8, and TextBlock 9 gives out the pseudocode to calculate the address of OSMetaClass object in 32-bit devices.

```
address1: LDR  R0, =(immediate)
address2: ADD R0, PC
address3: BX    LR
```
TextBlock 8

```
address = (address1+ immediate);
address += (4 - (address % 4));
address = KernelRead4Byte(address);
address = address2 + address + 4;
```
TextBlock 9

The memory layout of the returned OSMetaClass object gMetaClass is shown in Table 1, where we can get superClassLink(+8), className(+12) and classSize(+16) from the offsets. And className is an OSSymbol structure, whose memory layout is shown in Table 2 and we can get className length(+12) and strings(+16) from the offsets.

Table 1 gMetaClass Layout

OSMetaClass(gMetaClass)	Offset
…	…
const OSMetaClass *superClassLink	gMetaClass + 8
const OSSymbol *className	gMetaClass + 12
unsigned int classSize	gMetaClass + 16
…	…

Table 2 className Layout

OSSymbol from OSString(className)	Offset
…	…
unsigned int length	className+12
char * string	className+16
…	…

It is discovered that the vtable of each OSObject subclass is stored in the memory space of its Mach-O __DATA__const Section, then its address range can be taken out from the steps in Sec-

tion 1.1.1. After disassembling, it is discovered that Function getMetaClass() stands in the 8th order in the whole vtable if it exists. Thus, we can determine that an address points to a getMetaClass() function, if it meets these requirements:

(1) The vtable contains at least 14 continuous adjacent addresses (N>=13 as shown in Table 3), and these virtual methods' addresses should be in the range (XNU_TEXT_StartAddress, XNU_TEXT_EndAddress) or (KEXT_TEXT_StartAddress, KEXT_TEXT_EndAddress).

Table 3 Vtable Layout

virtual method 0
virtual method 1
virtual method 2
virtual method 3
….
virtual method N-3
virtual method N-2
virtual method N-1
virtual method N

(2) Take out the 8th (N=7) address, and check the memory which it points to. If the memory structure are the same as ARM instructions listed in TextBlock 9, then it is truly an implementation of Function getMetaClass(), and we can obtain the returned OSMetaClass object gMetaClass from TextBlock 10 in 32-bit devices.

Thus, we check all addresses in the __DATA__const Section byte by byte whether it meets the two requirements above or not. If so, gMetaClass can be located, and the runtime information and vtable addresses of the inheriting OSObject subclass can be exported.

Though this measure, we can get almost all OSObject subclasses' information, definitely including IOUserClient subclasses which will be used in subsequent IOKit fuzzing framework.

1.1.3 Virtual Methods Symbolization

In this section, we will obtain virtual method symbols of all OSObject subclasses in the kext, by resolving the symbolization of all classes' virtual methods in the kernel. After symbolization resolving, we can:

(1) locate the target method in disassembling and analyzing;

(2) get the the connection between clients and services, since each client may offer multiply services and different clients offer different services.

Storing the symbolization of all classes' virtual methods in the kernel, File kerenlcache is encrypted and stored in /System/Library/Caches/com.apple.kernelcaches/kernelcache, taking iPhone 4s for instance here. In some mobile devices, the secret key of this encrypted file has been totally cracked and published, also taking iPhone 4s 8.2b4 for instance as shown in TextBlock 10.

http://theiphonewiki.com/wiki/Firmware_Keys:

kernelcache.release.n94
IV: ae291ecd536ab102e6975a730f065f2f
Key: c45aac2036dea7bf564bd99399e6ff35b241b580afd323a7aee1b6e9162b1d4f

TextBlock 10

At first, we can export the system's kernelcache using the command "nm kernelcache". Then there are two different situations needed to be coped with: a complete kernelcache in plaintext decrypted with a published secret key; or an incomplete kernelcache because of unknown secret key.

In the situation that a complete kernelcache in plaintext can be exported:

(1) through searching __DATA__const addr in the main Mach-O, the address range of all base classes' vtables can be obtained, however some addresses in the range are not a true vtable of a base class.

(2) calculate the address_r with each address vtable_r in the vtable address range, by picking out the 8th address and turning it to a fixed constant:

address_r = KernelRead4Btye(vtable_r + sizeof(vm_address_t)*7)&0xfffffffe - kernel_slider

(3) search address_r in the address-symbol table in the plaintext kernelcache. If the paired symbol_r with address_r is found, and symbol_r contains String "getMetaClass", at the meanwhile it doesn't contain String "OSMetaClass", then vtable_r is truly a vtable of a base class.

(4) The base class name className_r corresponding with vtable_r can be deduced out from a regular expression:

className_r = symbol_r ([A-Z][A-Za-z]{3,})

(5) through traversing vtable_r in the vtable address range, we can get all true vtables and their corresponding base classes' names.

In the situation that an incomplete kernelcache is exported, it is supposed that symbols of base classes' virtual methods in the kernel are in the same arrangement sequence in different mobile devices. So we can deduce out the symbols and addresses of bass classes' virtual methods in different devices from those in iPhone 4s.

1.2 IOUserClients' Access

After exporting the information of all IOUserClient subclasses in the kernel in Section 1.1, in order to access a specific IOUserClient subclass and open its services, we still need two key parameters: serviceName and openType. In this section, how to obtain serviceNames and their corresponding openTypes of all IOUserClient subclasses is detailed.

1.2.1 Try All Service Names

All serviceNames in the kernel can be obtained by taking IOService subclasses' names out of all IOObject subclasses exported in Section 1.1.2.
And then execute codes shown in TextBlock 11, which use Function IOServiceOpen() to open services and try openType from 0 to 0xff. When kr=0, the service is open successfully. In some situations, services of IOUserClient subclasses cannot be open via codes here, and we will cope with them in Section 1.2.3 next.

```
for(serviceName in allServiceNames) {
        for(int _type = 0 ; _type < 0xff ; _type++) {
          CFDictionaryRef matchingDict = NULL;
          kern_return_t kr;
          io_service_t server = 0;
          task_port_t io_connect_t = 0;
          matchingDict = IOServiceMatching([serviceName UTF8String]);
          server = IOServiceGetMatchingService(0,matchingDict);
          kr = IOServiceOpen (server,mach_task_self(),_type,
&io_connect_t);
        }
}
```

TextBlock 11

1.2.2 io_connect_t address

In the parameters in IOServiceOpen(), the address of io_connect_t should be provided as the 4th parameter, which is the Mach Port of this IOUserClient subclass object in the kernel. So, we need to get the actual io_connect_t address.

In iOS versions lower than 8.1.3, through invoking Function mach_port_kobject(), we can get the Mach Port address of IOUserClient subclass object, but the returned address is obfuscated by the kernel. The obfuscation function is VM_KERNEL_ASSRPERM(), as shown in TextBlock 12, which uses a global variable vm_kernel_addperm.

```
kern_return_t
mach_port_kobject(
        ipc_space_t                     space,
        mach_port_name_t                name,
        natural_t               *typep,
        mach_vm_address_t               *addrp) {
          ..................
          *addrp = VM_KERNEL_ADDRPERM(VM_KERNEL_UNSLIDE(kaddr));
          ..................
}

#define VM_KERNEL_ADDRPERM(_v) \
        (((vm_offset_t)(_v) == 0) ? \
        (vm_offset_t)(0) :          \
        (vm_offset_t)(_v) + vm_kernel_addrperm)
```

TextBlock 12

But after CVE-2014-4496 patching in iOS 8.1.3, the mach_port_kobject() interface is disabled, then we need to find another effective way to get the Mach Port address of IOUserClient subclass object. Here we do find another interface mach_port_space_info() to implement this function in 32-bit devices, through a well-crafted method as detailed in TextBlock 13. Also, the returned obaddress is obfuscated with vm_kernel_addperm.

```
vm_address_t cr_mach_port_kobject(vm_address_t portname) {
    ipc_info_space_t info;
    ipc_info_name_array_t table = 0;
    mach_msg_type_number_t tableCount = 0;
    ipc_info_tree_name_array_t tree = 0;
    mach_msg_type_number_t treeCount = 0;
    vm_address_t obaddress = 0;
    mach_port_space_info(mach_task_self(), &info, &table, &tableCount,
&tree, &treeCount);
    for( int index = 0 ; index < tableCount ; index++ ) {
        ipc_info_name_t info = table[index];
        if(portname == info.iin_name) {
            obaddress = info.iin_object;
        }
    }
//obaddress is the address of structure ipc_port. By adding offset
//0x44,we can get ipc_kobject_t kobject in 32-bit devices.
    return CRReadAtAddress(obaddress+0x44);
}
```

TextBlock 13

vm_kernel_addperm is a global static variable in the kernel, which is assigned in the device booting and unreadable and unwritable in user mode. Thus, we need to find a kernel code segment which uses Function VM_KERNEL_ADDRPERM() and has a unique characteristic to be easily located in the kernel. Here Function IOGeneralMemoryDescriptor::wireVirtual() selected, whose code snippet is shown in TextBlock 14, by reasons that:

(1) String “IOMemoryDescriptor 0x%lx prepared read only” is a unique characteristic, and only appears in this function in the whole kernel.

(2) VM_KERNEL_ADDRPERM() is used in this function, i.e. vm_kernel_addperm.

```
IOReturn IOGeneralMemoryDescriptor::wireVirtual(IODirection forDirec-
tion) {
            ……
OSReportWithBacktrace("IOMemoryDescriptor 0x%lx prepared read only",
VM_KERNEL_ADDRPERM(this));
            ……}
```

TextBlock 14

Then we search the characteristic string “IOMemoryDescriptor 0x%lx prepared read only” in the IDA workspace to locate Function OSReportWithBacktrace(), and analyze the 2nd parameter VM_KERNEL_ADDRPERM(this) of this function to locate the kernel address of vm_kernel_ad-

drperm. Finally hardcode this address plus with kerenl_slider into our project, and then use memory read operation, CRReadAtAddress() as shown in TextBlock 1, to get the current value of vm_kernel_addrperm.

By now, we can get the actual io_connect_t address, as shown in TextBlock 15.

```
//before iOS 8.1.3
        mach_port_kobject(mach_task_self(), io_connect_t, &type, &kaddr);
//in or after iOS 8.1.3
        cr_mach_port_kobject(vm_address_t portname);
kaddr = kaddr - vm_kernel_addrperm;
```

TextBlock 15

1.2.3 Exception

In some situations, services of IOUserClient subclasses cannot be open via codes listed in TextBlock 11, and after carefully and deeply analyzing, there are three possibilities:

(1) This IOUserClient subclass is a base class of another IOUserClient subclass which can be open successfully.

(2) This IOUserClient subclass is never used in current iOS device, which may be a redundant component of the iOS system and used in another different device.

(3) The openType of current serviceName in this IOUserClient subclass is greater than 0xff, which is out of the traversing range in TextBlock 11.

With regard to the former two possibilities, we don't need to open services in this IOUserClient subclass. And in the last situation that openType>0xff, some openTypes are designed to a unreasonable large number by intention, to prevent from malicious call, and now we must analyze the concrete calling procedure.

When IOServiceOpen() is called to open the services of IOUserClient subclass in the user mode, in the kernel mode IOService::newUserClient() will be called to cope with it. TextBlock 16 shows the definition of IOService::newUserClient() in IOService.h in XNU 10.10.

From TextBlock 16, we can see that the 3rd parameter of newUserClient() is the openType we are searching. Using the information obtained in Section 1.1, we can locate the overridden address of newUserClient() in the corresponding IOService subclass, and the R3 register in this address is the pointer to the openType of this IOService subclass.

```
/*! @function newUserClient
    @abstract Creates a connection for a non kernel client.
    @discussion A non kernel client may request a connec-
tion be opened via the @link //apple_ref/c/func/IOSer-
viceOpen IOServiceOpen@/link library function, which will
call this method in an IOService object. The rules and ca-
pabilities of user level clients are family dependent, and
use the functions of the IOUserClient class for support.
IOService's implementation returns <code>kIOReturnUnsup-
ported</code>, so any family supporting user clients must
implement this method.
     @param owningTask The Mach task of the client thread
in the process of opening the user client. Note that in Mac
OS X, each process is based on a Mach task and one or more
Mach threads. For more information on the composition of a
Mach task and its relationship with Mach threads, see
{@linkdoc //apple_ref/doc/uid/TP30000905-CH209-TPXREF103
"Tasks and Threads"}.
    @param securityID A token representing the access level
for the task.
    @param type A constant specifying the type of connec-
tion to be created, specified by the caller of @link //ap-
ple_ref/c/func/IOServiceOpen IOServiceOpen@/link and inter-
preted only by the family.
    @param handler An instance of an IOUserClient object to
represent the connection, which will be released when the
connection is closed, or zero if the connection was not
opened.
    @param properties A dictionary of additional properties
for the connection.
    @result A return code to be passed back to the caller
of <code>IOServiceOpen</code>. */

    virtual IOReturn newUserClient( task_t owningTask, void
* securityID, UInt32 type, OSDictionary * properties,
    IOUserClient ** handler );

    virtual IOReturn newUserClient( task_t owningTask, void
* securityID, UInt32 type, IOUserClient ** handler );
```

TextBlock 16

TextBlock 17 lists some openTypes exported by our team, which is still correct in the newest iOS system.

```
0xff000001 IONetwork

0x44504456 IODPDevice to IODPDeviceUserClient

0x44505356 IODPService to IODPServiceUserClient

0x44504354 IODPController to IODPControllerUserClient

0x64506950 CCDataPipe to CCDataPipeUserClient

0x6C506950 CCLogPipe to CCLogPipeUserClient

0x57694669 AppleBCMWLANCore to AppleBCMWLANUserClient
```

TextBlock 17

1.3 IOExternalMethodDispatch

After successfully opening the services of IOUserClient subclass, IO methods can be called: when Function IOConnectCallMethod() is used in the user mode, Function IOUserClient::externalMethod() will be overridden in the kernel mode.

In the kernel there are 5 parameters in IOUserClient::externalMethod(), as shown in TextBlock 18. When it is called, only a selector and an encapsulation “arguments” are delivered from IOConnectCallMethod() in the user mode. The 3rd parameter IOExternalMethodDispatch * dispatch = 0, and at the meanwhile, the IOUserClient subclass which offers the service will override externalMethod() to supply the dispatch to IOUserClient::externalMethod() in the kernel.

```
virtual IOReturn externalMethod(
        uint32_t selector,
        IOExternalMethodArguments * arguments,
        IOExternalMethodDispatch * dispatch = 0,
        OSObject * target = 0,
        void * reference = 0 );
```

TextBlock 18

IOExternalMethodDispatch structure is shown in TextBlock 19, which is crucial to check the input from IOConnectCallMethod() in the user mode, i.e. “arguments” in TextBlock 18. In this structure, checkScalarInputCount is the length of input 64-bit integer array, checkStructureInputSize is the bytes length of input structure buffer, checkScalarOutputCount is the length of output 64-bit integer array, and checkStructureOutputSize is the bytes length of output structure buffer.

```
struct IOExternalMethodDispatch
{
    IOExternalMethodAction function;
    uint32_t                checkScalarInputCount;
    uint32_t                checkStructureInputSize;
    uint32_t                checkScalarOutputCount;
    uint32_t                checkStructureOutputSize;
}
```

TextBlock 19

In the input check of IOUserClient::externalMethod(), arguments are derived from IOConnectCallMethod() in the user mode, whose length must be equal to the defined length of the corresponding input type in IOExternalMethodDispatch. If failed the check, a 0xe00002c2 error will occur, and the input from user mode cannot reach the opened service of this IOUserClient subclass object in the kernel. There is an exception here, that if one of the defined check length in IOExternalMethodDispatch is equal to 0xffffffff, then the corresponding input can be of any length.

Obtaining the IOExternalMethodDispatchs of IOUserClient subclasses are crucial to fuzz IOKit in iOS, which can avoid a lot of useless fuzzing with unavailable inputs and greatly enhance its accuracy and efficiency. Next, two measures to obtain the IOExternalMethodDispatch tables in the kernel are presented. In our project, Measure One in Section 1.3.1 will be carried out first, which can cover most of IOExternalMethodDispatch tables; and if failed, then Measure Two in Section 1.3.2 will be carried out to cover the left.

1.3.1 Measure One

It is discovered that most IOExternalMethodDispatch tables store in a regular memory layout in the kernel as shown in TextBlock 20, because they are coded as global static variables in the same source code of a IOUserClient subclass.

Kext Mach-O __DATA__const and client vtable start address can be obtained in Section 1.1, and the following is client virtual methods block starting from client vtable start address. Then, it is client metaClass virtual methods block, which is separated from client virtual methods block by multiline 0s, and client metaClass vtable start address can be calculated by adding 1 to the last 0's address between the two blocks. And next, it is the variables block after client metaClass virtual methods block, which is also separated by multiline 0s from the former block, including IOExternalMethodDispatch block we are looking for.

```
            0
client virtual method 0              ——(client vtable start address)
client virtual method 1              ——>
client virtual method 2              ——client virtual methods block
……                                   ——>
client virtual method N              ——(client vtable end address)
            0
……                                   ——(all 0, at least 4 lines)
client metaClass virtual method 0    ——(client metaClass vtable start address)
client metaClass virtual method 1    ——>
client metaClass virtual method 2    ——client metaClass virtual methods block
            ……                       ——>
client metaClass virtual method N    ——(client metaClass vtable end address)
            0
……                                   —— variables block
function-0
checkScalarInputCount-0              ——>
checkStructureInputSize-0            ——>
checkScalarOutputCount-0             —— (IOExternalMethodDispatch 0)
checkStructureOutputSize-0           ——>
……                                   ——IOExternalMethodDispatch block
function-N
checkScalarInputCount-N              ——>
checkStructureInputSize-N            ——>
checkScalarOutputCount-N             —— (IOExternalMethodDispatch N)
checkStructureOutputSize-N           ——>
```

TextBlock 20

Here we taken meta_vtable_end as client metaClass vtable end address and variable block start address, which is the first 0's address between the two blocks. Starting from meta_vtable_end, execute the pseudocode listed in TextBlock 21 to match all IOExternalMethodDispatch addresses. According to TextBlock 19, the matching rules of IOExternalMethodDispatch address are:

(1) the first address points to an actual function in the kernel, i.e, x∈(KEXT_TEXT_StartAddress, KEXT_TEXT_EndAddress).

(2) the following 4 addresses point to a unit32_t type or a 0xffffffff.

```
//Match start position
dispatch_start_addr = 0
size_t addSize = sizeof(vm_address_t);
for( start = meta_vtable_end ; start < (meta_vtable_end + 50); start++) {
        //selector 0
        check_func_0       = KernelRead4Byte(start);
        check_scalar_i_0   = KernelRead4Byte(start+ addSize);
        check_struct_i_0   = KernelRead4Byte(start+ addSize*2);
        check_scalar_o_0   = KernelRead4Byte(start+ addSize*3);
        check_struct_o_0   = KernelRead4Byte(start+ addSize*4);
        //selector 1
        check_func_1       = KernelRead4Byte(start+ addSize*5);
        check_scalar_i_1   = KernelRead4Byte(start+ addSize*6);
        check_struct_i_1   = KernelRead4Byte(start+ addSize*7);
        check_scalar_o_1   = KernelRead4Byte(start+ addSize*8);
        check_struct_o_1   = KernelRead4Byte(start+ addSize*9);

        if(
        ((check_func_0 > KEXT_TEXT_StartAddress)
                  && (check_func_0 < KEXT_TEXT_EndAddress))         &&
        (check_scalar_i_0 < 0xffff  ||  check_scalar_i_0 == 0xffffffff)   &&
        (check_struct_i_0 < 0xffff  ||  check_struct_i_0 == 0xffffffff)   &&
        (check_scalar_o_0 < 0xffff ||  check_scalar_o_0 == 0xffffffff)  &&
        (check_scalar_i_0 < 0xffff  ||  check_scalar_i_0 == 0xffffffff)   &&
        ((check_func_1 > KEXT_TEXT_StartAddress)
                  && (check_func_1 < KEXT_TEXT_EndAddress))         &&
        (check_scalar_i_1 < 0xffff   ||  check_scalar_i_1 == 0xffffffff)  &&
        (check_struct_i_1 < 0xffff   ||  check_struct_i_1 == 0xffffffff)  &&
        (check_scalar_o_1 < 0xffff  ||  check_scalar_o_1 == 0xffffffff) &&
        (check_struct_o_1 < 0xffff  ||  check_struct_o_1 == 0xffffffff)
        ) {
           dispatch_start_addr = start;
            break;
        }
}
```

TextBlock 21

After finding the first matched IOExternalMethodDispatch address, match the following memory space to find more IOExternalMethodDispatchs until the matching rules are not fulfilled, because all the IOExternalMethodDispatchs are stored in a continuous memory space.

If the first matched IOExternalMethodDispatch address doesn't exist, which means IOExternal-MethodDispatchs aren't stored as global static variables in this IOUserClient subclass, then Measure One fails, and Measure Two is carried out in our project, as detailed in the next section.

1.3.2 Measure Two

Firstly, by locating all LDR Literal instructions when externalMethod() is overridden in the kernel, we can resolve the addresses of their corresponding instruction (ADD Rn, PC). Then check these addresses whether they match the 2 rules of IOExternalMethodDispatch address listed in Section 1.3.1 above. If matching successfully, then it's an address of a valid IOExternalMethod-Dispatch. By this measure, we can obtain those IOExternalMethodDispatchs which are directly used in externalMethod().

However, Measure Two covers only a small-scale IOExternalMethodDispatchs, because a lot of IOExternalMethodDispatchs aren't used directly in overridden externMethod(). If IOExternal-MethodDispatch isn't stored as a global static variable in the source code of IOUserClient sub-class, and it isn't used directly in overridden externMethod(), then the two measures presented in this paper will both fail. In fact, Measure One can cover most IOExternalMethodDispatchs in the kernel, and Measure Two is only carried out as a supplementary when Measure One fails.

1.4 IOExternalMethod

Differently from Section 1.3, there is a mount of IOUserClient subclasses which don't override Function externalMethod(). Instead, they will select Function getTargetAndMethodForIndex() or Function getExternalMethodForIndex() to override and return a data structure IOExternalMethod to IOUserClient::externalMethod() in the kernel. Here IOExternalMethod is a similar structure as IOExternalMethodDispatch to check the input/output parameters, as shown in TextBlock 22.

```
struct IOExternalMethod {
    IOService *             object;
    vm_address_t*           func;
    int                     vflag
    IOOptionBits            flags;
    IOByteCount             count0;
    IOByteCount             count1;
}

IOExternalMethod * IOUserClient::
getTargetAndMethodForIndex(IOService **targetP, UInt32 index)
```

TextBlock 22

In IOExternalMethod structure, the field vflag is a flagging integer: if vflag=0, func is the function address, while if vflag=1, func is the object's vtable offset. The fields flags, count0 and count1 work together to define the length of input/output integer array or structure buffer, which is defined in IOUserClient.h in XNU 10.10.
The most easy and reliable approach to obtain IOExternalMethod is to directly invoke Function getTargetAndMethodForIndex() in the kernel.

1.4.1 Arbitrary Kernel Function Call

According to Ref. [1] "Tales from iOS 6 Exploitation and iOS 7 Security Changes", an approach to implement arbitrary kernel function call was presented in HITB2013. Here we will introduce its details in this section, and construct a getTargetAndMethodForIndex() call in the next section.
In the kernel, the APIs offered by IOKit to the user mode mainly are IOConnectTrap0(), IOConnectTrap1(), IOConnectTrap2(), IOConnectTrap3(), IOConnectTrap4(), IOConnectTrap5() and IOConnectTrap6(), where the digit in the API name represents the amount of explicit parameters in the calling functions, as shown in TextBlock 23.
The corresponding codes in the kernel start from Function iokit_user_client_trap(), as listed in TextBlock 24. If the IOExternalTrap address which is returned from Function getExternalTrapForIndex() points to an controllable IOExternalTrap object, then functions in arbitrary kernel address with a maximum 7 parameters (target, args->p1, args->p2, args->p3, args->p4, args->p5, args->p6) can be called by controlling the returned IOExternalTrap object address, where target is the IOService* object in the IOExternalTrap structure, as shown in TextBlock 25.

```
struct IOExternalTrap {
    IOService *         object;
    IOTrap              func;
};
```

TextBlock 25

In the kernel, IOUserClient::getExternalTrapForIndex() is a virtual method which returns NULL as shown in TextBlock 24. But in a jail-broken device, its __DATA__const can be written. Thus, we rewrite the address of getExternalTrapForIndex() to the address of the gadget shown in TextBlock 26 plus with 1, i.e. the address of code {0x8 0x46 0x70 0x47} in the kernel executable addresses plus with 1. By doing so, the implementation of getExternalTrapForIndex() is changed to that in TextBlock 27, where index is an input from IOConnectTrap() and points to a controllable IOExternalTrap object. Now, arbitrary kernel function call is implemented.

```
kern_return_t
IOConnectTrap0(io_connect_t    connect,
               uint32_t        index );

kern_return_t
IOConnectTrap1(io_connect_t    connect,
               uint32_t        index,
               uintptr_t       p1 );

kern_return_t
IOConnectTrap2(io_connect_t    connect,
               uint32_t        index,
               uintptr_t       p1,
               uintptr_t       p2);

kern_return_t
IOConnectTrap3(io_connect_t    connect,
               uint32_t        index,
               uintptr_t       p1,
               uintptr_t       p2,
               uintptr_t       p3);

kern_return_t
IOConnectTrap4(io_connect_t    connect,
               uint32_t        index,
               uintptr_t       p1,
               uintptr_t       p2,
               uintptr_t       p3,
               uintptr_t       p4);

kern_return_t
IOConnectTrap5(io_connect_t    connect,
               uint32_t        index,
               uintptr_t       p1,
               uintptr_t       p2,
               uintptr_t       p3,
               uintptr_t       p4,
               uintptr_t       p5);

kern_return_t
IOConnectTrap6(io_connect_t    connect,
               uint32_t        index,
               uintptr_t       p1,
               uintptr_t       p2,
               uintptr_t       p3,
               uintptr_t       p4,
               uintptr_t       p5,
               uintptr_t       p6);
```

```
kern_return_t iokit_user_client_trap(struct iokit_user_client_trap_args
*args)
{
    kern_return_t result = kIOReturnBadArgument;
    IOUserClient *userClient;
   if ((userClient = OSDynamicCast(IOUserClient,
        iokit_lookup_connect_ref_current_task((OSObject *)(args->user-
ClientRef))))) {
        IOExternalTrap *trap;
        IOService *target = NULL;
        trap = userClient->getTargetAndTrapForIndex(&target, args->in-
dex);
        if (trap && target) {
          IOTrap func;
          func = trap->func;
          if (func) {
              result = (target->*func)(args->p1, args->p2, args->p3,
args->p4, args->p5, args->p6);
           }
        }
       userClient->release();
    }
    return result;
}

IOExternalTrap * IOUserClient::
getTargetAndTrapForIndex(IOService ** targetP, UInt32 index)
{
     IOExternalTrap *trap = getExternalTrapForIndex(index);

     if (trap) {
           *targetP = trap->object;
     }
     return trap;
}

IOExternalTrap * IOUserClient::
getExternalTrapForIndex(UInt32 index)
{
       return NULL;
}
```

TextBlock 24

```
gadget
address1:  MOV R0 R1
address2:  BX    LR
```

TextBlock 26

```
IOExternalTrap * IOUserClient::
getExternalTrapForIndex(UInt32 index)
{
       return index;
}
```

TextBlock 27

1.4.2 getTargetAndMethodForIndex() call

Based on the measure of arbitrary kernel function call, a specific getTargetAndMethodForIndex() call is constructed in this section, whose structure is shown in Figure 1.

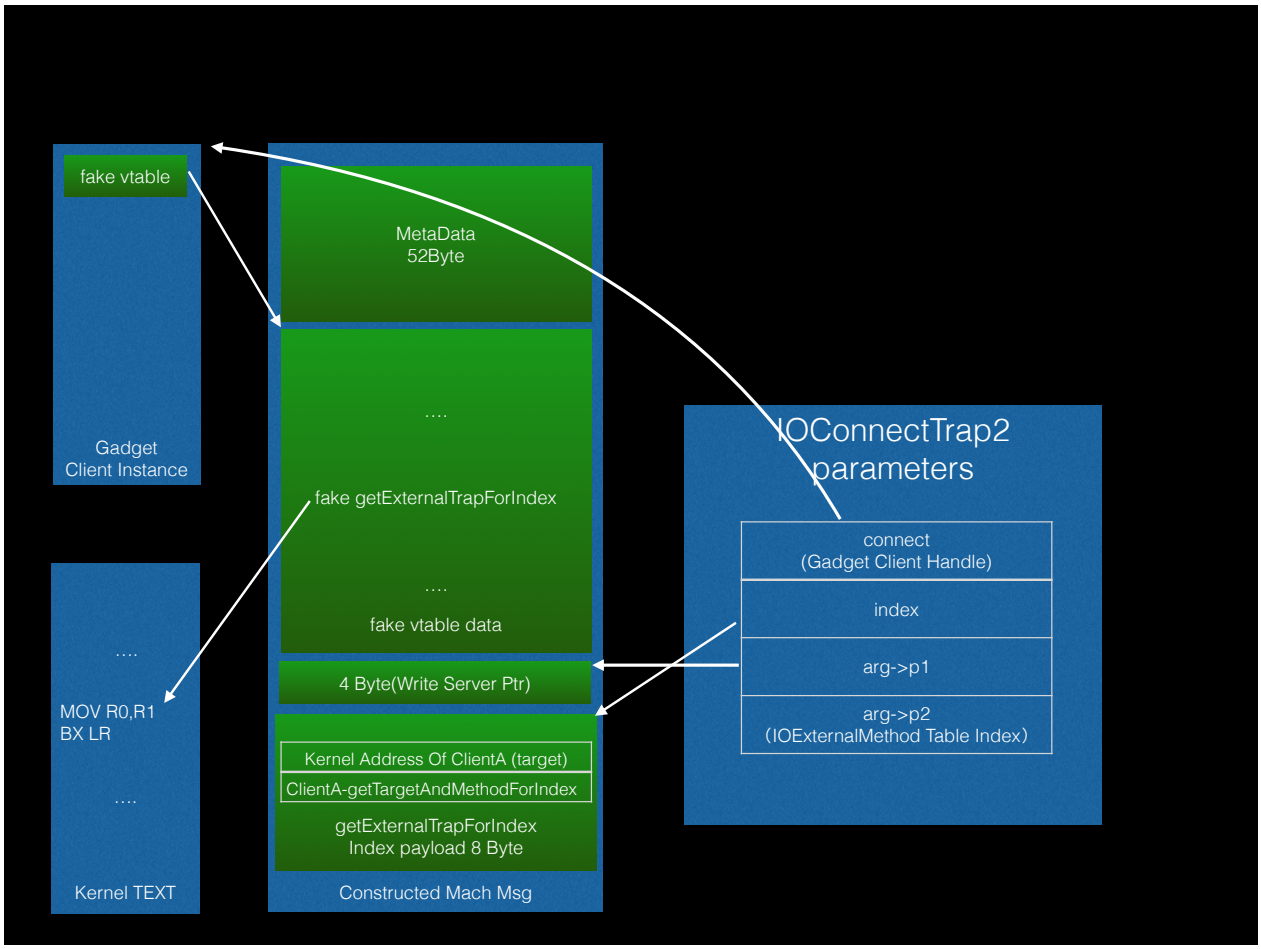

Figure 1 getTargetAndMethodForIndex() call

Function getTargetAndMethodForIndex() is defined in IOUserClient.h in XNU 10.10, as shown in TextBlock 28.

```
virtual IOExternalMethod *
        getTargetAndMethodForIndex( IOService ** targetP, UInt32 index );
```

TextBlock 28

To construct a getTargetAndMethodForIndex() call, the selected API should be IOConnectTrap2(), as shown in Figure 1, and its four parameters are set as follows:

(1) connect, pointed to a Gadget Client Instance. In this instance, there is a fake vtable address, pointed to a fake vtable data containing a fake getExternalTrapForIndex() function's address, which is detailed in Section 1.4.1.

(2) index, pointed to an IOExternalTrap structure. and the fields in IOExternalTrap structure is constructed as Table 4. It is a getExternalTrapForIndex()'s Index payload in 8 bytes.

Table 4 IOExternalTrap Construction

Fields	Values
IOService * object	Kernel Address of ClientA (target)
`IOTrap func;`	ClientA->getTargetAndMethodForIndex() address

(3) arg->p1, pointed to an writable address not in use.

(4) arg->p2, which is IOExternalMethod table index.

By invoke IOConnectTrap2(), we can invoke ClientA->getTargetAndMethodForIndex() actually to get IOExternalMethod from arg->p2, i.e, IOExternalMethod table index.

2. Fuzzing

As a result of Chapter 1, all information we get about IOUserClient subclasses can be used to generate a much more accurate and effective input in our IOKit fuzzing framework, which brings a great optimized enhancement in the fuzzing effect.

2.1 IOKit Fuzzing Framework

The IOKit fuzzing framework is constructed on the base of iOS LaunchDeamon, which can implement automatic fuzzing, restore the parameters as accurate as possible when panic occurs, and keep its own running sustainable and efficient.

The main components of the framework is shown in Figure 1.

Panic Log Collector: each Client-Selector pair in the current fuzzing process will be recorded in the logs. And when iOS LaunchDeamon starts after rebooting, this Collector will collect the last panic in /Library/Logs/CrashReporter/Panics/ directory and record its corresponding Client-Selector pairs in the Log, and then empty the /Library/Logs/CrashReporter/Panics/ directory.

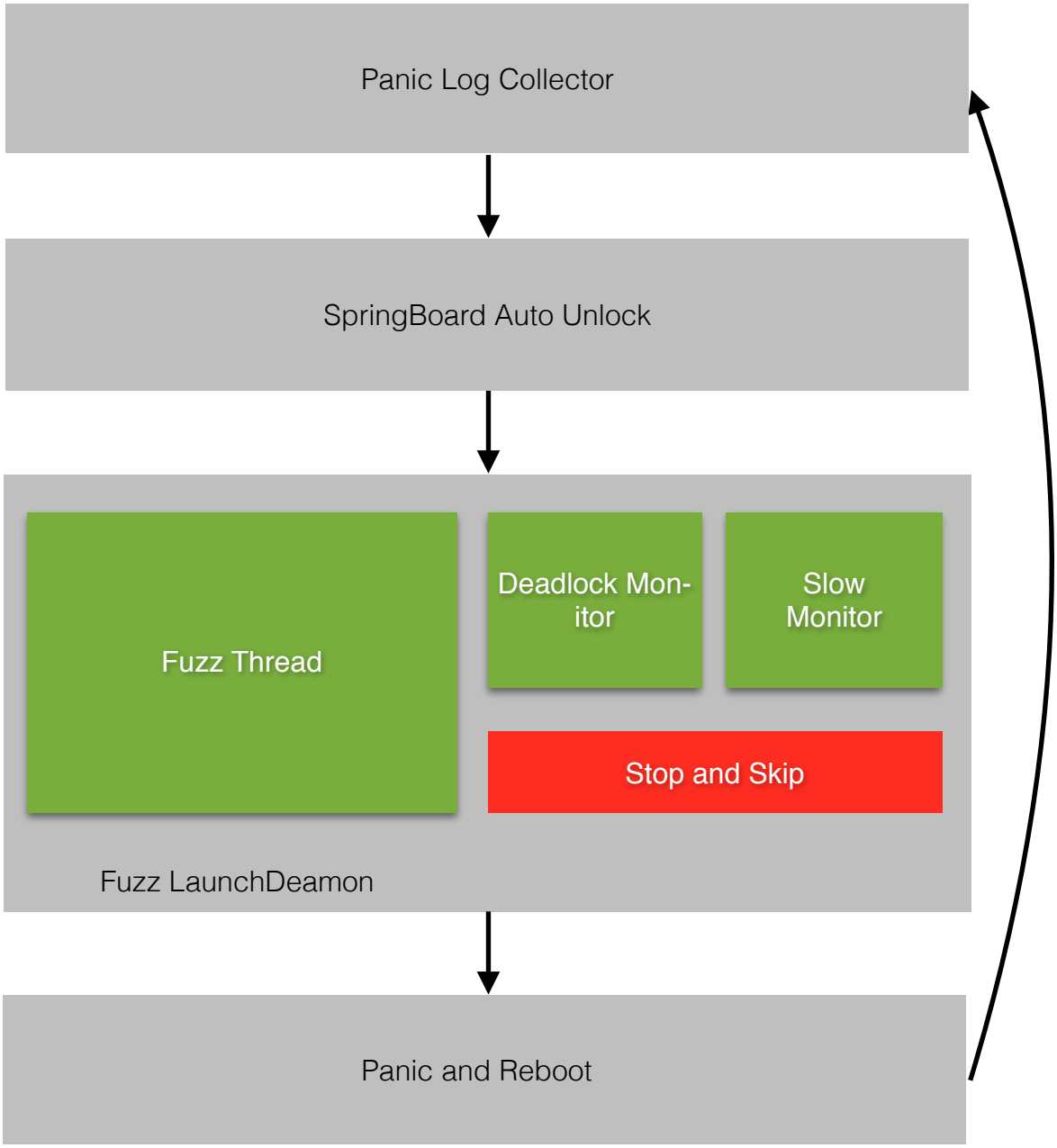

Figure 1 Framework Infrastructure

SpringBoard Auto Unlock: the iPhone will enter the automatic sleep mode in a little while after rebooting, so a small Tweak is running to unlock the device automatically, whose codes are shown in TextBlock 29.

```
[[objc_getClass("SBBacklightController") sharedInstance] turnOnScreenFullyWithBacklightSource:0];
[[objc_getClass("SBLockScreenManager") sharedInstance] unlockUIFromSource:0 withOptions:nil];
```

TextBlock 29

Fuzz LaunchDaemon is the main body, which consists of :

(1) Fuzz Thread: the fuzzing algorithms will be detailed in the next Section 2.2.

(2) Deadlock Monitor: when fuzzing some IOUserClient interfaces, the kernel may enter a deadlock state. This Monitor will monitor the times of fuzzing a specific IOUserClient interface. If the times keeps unchanged in a long period, then the Monitor will reboot the device and record the deadlock times of this interface. And if the deadlock times exceed a defined threshold, this IOUserClient interface will not be fuzzed any more.

(3) Slow Monitor: when fuzzing some IOUserClient interfaces, the kernel suffers a great consumption, which brings a low speed and effectiveness in the whole fuzzing framework. This Monitor will reboot the device if its consumption is huge.

(4) Stop and Skip: stop the fuzz thread and reboot the device. It will be called by Deadlock Monitor and Slow Monitor.

Panic and Reboot: when a panic occurs, the device will reboot, and the fuzzing application will be called back to Panic Log Collector.

2.2 Fuzz Thread

This section contains two mechanisms in fuzz thread in the framework, including sequential fuzzing and shuffle fuzzing.

2.2.1 Sequential fuzzing

Select a IOUserClient subclass Client, and traversing all its interfaces' Selector in their arrangement sequence. Skip those interfaces' Selector with no arguments, and fuzz other interfaces' Selector with input generated in Section 2.2.1. The times of fuzzing each interface is defined to 0xffffff in our framework.

By so, we can get as much parameter information as we can after a panic, such as Client, Service, Selector and so on. However, since some interfaces have a dependency relationship with others, thus it has an incomplete coverage, e.g., read() interface Selector arranged behind close() interface Selector will not be effectively fuzzed.

TextBlock 30 lists the statistics result of panics in different types in iOS712-4 exploited by our team, where different backtrace-sliders denote different panics instead of the actual panic times:

```
panics in iOS712-4:
        null pointer dereference: 28
        invalid address read: 7
        invalid address write: 1
        invalid address execute:0
```

TextBlock 30

TextBlock 31 lists the statistics result of panic types in iOS812-4S exploited by our team, where different backtrace-sliders denote different panics instead of the actual panic times:

```
panics in iOS812-4S:
        null pointer dereference: 7
        invalid address read: 1
        invalid address write: 1
        invalid address execute: 1
```

TextBlock 31

We can see that iOS 8 has patched a lot of potential vulnerabilities, and also brought about new undiscovered ones.

2.2.2 Shuffle fuzzing

To avoid dependency relationship between interfaces in the same IOUserClient subclass, shuffle fuzzing is imported in our project. It breaks down the sequence of the interfaces' Selector arranging in the same IOUserClient subclass, and selects a random Selector to fuzz this subclass each time. If the current selected Selector interface has no arguments, skip it; else, try to fuzz this Client-Selector pair with generated input for at most 500 times, which will terminate if it returns success or its maximum times 500 is exceeded.

Shuffle fuzzing can effectively expand the fuzzing coverage, however, panics in shuffle fuzzing cannot record the current Selector number, which can only be restored by back trace in the panic logs.

TextBlock 32 lists the statistics result of panic types in iOS812-4S, where different backtrace-sliders denote different panics instead of the actual panic times:

```
panics in iOS812-4S:
        null pointer dereference: 11
        invalid address read: 3
        invalid address write: 2
        invalid address execute: 2
```

TextBlock 32

Reference

[1] Stefan Esser, “Tales from iOS 6 Exploitation and iOS 7 Security Changes”, Hack In The Box 2013 - HITB Security Conference, Malaysia, Oct. 16 - 17, 2013.