Exploitations of Uninitialized Uses on macOS Sierra

Zhenquan Xu, Gongshen Liu
School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University

Tielei Wang, Hao Xu
PWNZEN InfoTech Co., LTD.

Abstract

An uninitialized use refers to a common coding mistake where programmers directly use variables on the stack or the heap before they are initialized. Uninitialized uses, although simple, can lead to severe security consequences. In this paper, we will share our experience in gaining arbitrary kernel code execution in the latest macOS Sierra by exploiting two uninitialized use vulnerabilities for Pwnfest 2016. Specifically, we first analyze the attack surface of the XNU kernel and mitigation techniques, and then study common types of uninitialized uses and potential threats. Then we elaborate on the vulnerabilities and exploitation techniques. Lastly, we summarize the whole exploitation and discuss the reliability of the exploitation.

1 Introduction

Although not directly leading to memory corruptions, uninitialized use has become a kind of severe security vulnerability. The uninitialized use may result in information leaks or control of the instruction pointer, in the case that attackers can effectively control memory layout and usage by using advanced exploitation techniques such as stack based or heap based spraying.

Many researchers have proposed different methods or systems to detect, eliminate, or mitigate uninitialized uses. For example, MemorySanitizer [21] and kmemcheck [19] perform checks on each memory read and write operation to detect uninitialized reads but they incur significant overhead. STACKLEAK [22], proposed by PaX, clears the kernel stack when returning to the user space but it does not prevent uninitialized use in the kernel heap. Kangjie Lu's UniSan [17], which is effective in detecting stack based and heap based uninitialized uses and has slight overhead, requires rebuilding the source code thus it can only be applied to open-source projects. In short, such methods or systems are still hard to be applied to entire modern operating systems. In this paper, we will share our experience of gaining arbitrary kernel code execution in the latest macOS Sierra by exploiting two uninitialized use vulnerabilities in Pwnfest 2016.

macOS Sierra is the thirteenth major release in macOS, Apple Inc.'s desktop and server operating system for Macintosh computers. It was released to end users on September 20, 2016. Pwnfest is a security contest held on November 10-11, 2016, in Seoul, South Korea that aims to improve the security of current popular operating systems by awarding hackers that are able to hack the latest operating systems and browsers. There are eight main targets (listing in Table-1) for hackers in Pwnfest 2016.

Table 1: Targets in Pwnfest 2016

Target
Microsoft Edge + Windows 10 x64 RS1
Microsoft Hyper-V + Windows Server 2016
Google Chrome + Windows 10 x64 RS1
Android 7.0 + Google Pixel
Adobe Flash + Microsoft Edge + Windows 10 x64 RS1
Apple Safari + macOS Sierra
Apple iOS 10 + iPhone 7 Plus
VMWare Workstation Pro 12 + Windows 10 x64 RS1

In Pwnfest 2016, our target was Safari on macOS Sierra. We first exploited an info-leak vulnerability (CVE-2017-2355) and a UAF vulnerability (CVE-2017-2356) to gain remote code execution in the context of Safari, when Safari processed a crafted web page. However, to win the contest and get the bonus, we needed to escape the Safari sandbox and gain the root privilege. To achieve this, we exploited two uninitialized value vulnerabilities in the kernel. This paper will focus on the kernel exploitation. Specifically, the contribution of this

research includes:

- A detailed demonstration and analysis of exploiting uninitialized value bugs to bypass modern kernel mitigations such as kernel ALSR and SMAP/SMEP on macOS Sierra.
- State-of-the-art exploitation techniques for exploiting macOS Sierra kernel.
- A systematic review of attack surfaces of macOS Sierra.

The rest of the paper is organized as follows. We review Safari, XNU kernel and the mitigations systematically in §2. Then we analyze the common types of uninitialized use and the potential threats in §3. We analyze and exploit the vulnerabilities in detail in §4. We summarize our exploitation and make discussion in §5. Related work is summarized in §6. Lastly, we conclude in §7.

2 An Analysis of macOS Sierra

2.1 Safari Browser & Sandbox

Safari is developed by Apple Inc. and it is the default browser on macOS Sierra. In this part, the process model of Safari browser will be first introduced and then the sandbox will be discussed. The Safari browser is composed of several separated processes. Safari employs an isolated process model and obeys least privilege principles which means each of its components can only access limited system resources which it requires. Specifically, Safari can be divided into four parts [11]:

- WebProcess, which is also called WebContent, is responsible for parsing HTML files, rendering DOM objects and drawing layouts for a webpage. It deals with Javascript and other active web contents as well.
- NetworkProcess is responsible for network communication of a browser like loading pages, loading resources (pictures, audios, videos) and posting requests.
- PluginProcesses is responsible for managing plugins of a browser like Adobe Flash.
- UIProcess is the parent process of all the other processes mentioned before. It is responsible for dispatching events and messages between other processes.

macOS uses the sandbox mechanism [7] to minimize the damage to the system and user data if an app becomes compromised. The kernel implements a Mandatory Access Control (MAC) sandbox model. When a sandboxed process tries to access some system resource, the kernel will consult the app's sandbox file to determine whether to allow or to deny this operation.

Safari is also partially sandboxed. The WebProcess, the NetworkProcess and the PluginProcesses are sandboxed but the UIProcess is not sandboxed. After exploiting some vulnerabilities in WebCore or JavaScriptCore, attackers usually gain arbitrary code execution in the sandboxed WebContent process (aka WebProcess). It is still quite restricted to do something further so sandbox escape is necessary.

Commonly, two different paths are available. The first one is attacking system services which are not sandboxed and accessible in the WebProcess sandbox. Target services could be WindowServer, fontd, launchd, etc.. Exploiting vulnerabilities in these services help escape the sandbox and gain root privileges (if the target service is running as root). The other choice is exploiting kernel vulnerabilities directly. Attackers can gain kernel privilege and break out of the sandbox at the same time by exploiting kernel vulnerabilities. However, this way is much harder than the first one because of the sandbox. Only a small number of user clients are accessible from a WebProcess sandbox (Table 2). Nevertheless, in Pwnfest 2016, the second way was used and two vulnerabilities were exploited to gain the kernel privileges.

Table 2: User clients allowed to be opened in WebProcess

#	User Client Name	KEXT Name
1	AppleUpstreamUserClient	AppleUpstreamUserClient.kext
2	AppleMGPUPowerControlClient	AppleGraphicsControl.kext
3	RootDomainUserClient	System.kext
4	IOAudioControlUserClient	IOAudioFamily.kext
5	IOAudioEngineUserClient	IOAudioFamily.kext
6	IOAccelerator	IOGraphicsFamily.kext
7	IOAccelerationUserClient	IOGraphicsFamily.kext
8	IOSurfaceRootUserClient	IOSurface.kext
9	IOSurfaceSendRight	IOSurface.kext
10	IOFramebufferSharedUserClient	IOGraphicsFamily.kext
11	AppleSNBFBUserClient	AppleIntelSNBGraphicsFB.kext
12	IOHIDParamUserClient	IOHIDFamily.kext
13	AppleGraphicsControlClient	AppleGraphicsControl.kext
14	AppleGraphicsPolicyClient	AppleGraphicsControl.kext
15	AGPMClient	AppleGraphicsPowerManagement.kext

2.2 XNU attack surface

XNU [6] is a computer operating system developed at Apple Inc. and is used widely as the kernel for macOS, iOS, tvOS, and watchOS operating systems. XNU is a recursive abbreviation of "XNU is Not Unix". In this part, the attack surface of XNU will be detailedly discussed.

On the top view of XNU, XNU is a hybrid kernel which contains features of both monolithic kernels and microkernels. XNU can be considered as a mixture of the Mach kernel and the BSD kernel. XNU's BSD component uses FreeBSD as the primary codebase and it is responsible for process management, basic security policies, BSD system calls, network stack, filesystems, etc.. XNU's Mach component is based on Mach 3.0 developed by CMU in the middle 1980s. Mach is responsible for multitasking, memory management, process communication and so on [16]. Besides BSD and Mach, XNU contains a special driver framework called I/O Kit. The I/O Kit [9] is a collection of system frameworks, libraries, tools, and other resources for creating device drivers in OS X. It is implemented in a restricted form of C++ which omits features like multiple inheritance and exception handling, which are unsuitable for a multithreaded kernel. It also provides user space programs with the capability of communicating with the drivers in the kernel.

Basically, any communication channels between the user space and the kernel can be considered as an attack surface. According to the XNU architecture mentioned above, POSIX/BSD system calls, ioctl, file system and IOKit are all large attack surface for attackers. Table 3 lists the vulnerabilities used by famous jailbreak tools.

The IOKit driver framework deserves further discussion here. The IOKit is an ideal attack surface due to the following reasons:

- Some of the device drivers are developed by third-party companies, like the drivers for AMD graphic cards and the drivers for NVIDIA graphic cards. The code quality of these drivers cannot be guaranteed and these device drivers are more likely to contain security flaws than those drivers developed by Apple.
- There are hundreds of device drivers in macOS so there is much more code involved than the XNU kernel itself. Moreover, a majority of these drivers are not open source and are rarely audited by researchers.
- The IOKit provides dozens of API in the user space to communicate with drivers in the kernel and each device driver may have dozens methods for user space programs to call, which leaves a large attack surface for attackers.

To start a communication with an IOKit driver in the kernel, the following three steps are usually taken:

1. Get the name of the service that the driver corresponds to. The name can be obtained from the output of "ioreg" command in a terminal. Then pass the name to `IOServiceMatching()`, which is an API provided by the IOKit framework, to create a matching dictionary for the next step.
2. Pass the matching dictionary in step 1 to `IOServiceGetMatchingServices()` to get an iterator of all the services which match the name in the matching dictionary. Objects in the iterator are `IOService` ports, which correspond to the devices' instances in the kernel.
3. Iterate over the iterator in step 2 and call `IOServiceOpen()` on each port. `IOServiceOpen()` also returns a port corresponding to an `IOUserClient` instance. This port is necessary for the following communication. `IOServiceOpen()` has an integer parameter named `type` and the driver creates different `IOUserClient` instances according to the `type` parameter.

`IOService` class is the subclass to provide driver functionality and `IOUserClient` class is the subclass to provide user space interfaces. After obtaining a port of an `IOService` instance or an `IOUserClient` instance, several operations can be performed:

- Set properties. Properties, which are key-value pairs, can be associated with an `IOService` instance or an `IOUserClient` instance. Drivers rely on these properties to function properly. User space program can call `IORegistryEntrySetCFProperty()` to set a value for a property. It is dangerous if a user space program can modify any properties of a driver. A driver can override the `::setProperty()` to allow a user space program to set

Table 3: Vulnerabilities used in jailbreaks

Tool Name	Version	Attack Surface	Description
limera1n/greenpois0n	iOS 4.1	ioctl	DIOCADDRULE ioctl handler improper initialization
JailbreakMe 3	iOS 4.2.x	file system	HFS legacy volume name stack buffer overflow
Corona	iOS 5.0	file system	HFS heap overflow
p0sixspwn	iOS 6.1.3	POSIX System Calls	posix_spawn improperly checks file action data
evasi0n7	iOS 7.0.x	ioctl	ptmx_get_ioctl out-of-bounds memory access

properties. To ensure safety, privilege checks are necessary to prevent unprivileged user space programs from modifying sensitive properties. CVE-2016-1825 [10] showed the severity of lacking privilege checks. The vulnerability lied in `IOHIDevice` driver which overrode the `::setProperty()` but did not check the privilege. Thus an unprivileged user space program could set arbitrary properties on an `IOHIDevice` instance. If an attacker set the `IOUserClientClass` key with the value of `IOPCIDiagnosticsClient` and then call `IOServiceOpen()` on this `IOHIDevice` instance, the attacker would obtain an `IOPCIDiagnosticsClient` instance which would give the attacker arbitrary read/write capability over the kernel map.

- Call methods. Drivers provide several methods for user space programs to exchange data. These methods are called external methods. A driver may have several user clients and each user client may have several external methods. Each method takes nine parameters. The first parameter is a unique integer named `selector` which is an ID of this method in a driver. The following four parameters are four types of input/output, they are `scalarInput` which refers to integer inputs, `structureInput` which refers to binary value inputs, `scalarOutput` which refers to integer outputs and `structureOutput` refers to binary value outputs. The last four parameters are four integers to specify the length of scalar/structure inputs/outputs. User space programs can invoke these external methods via `IOConnectCallMethod()`. This API takes a port of a user client, a selector, four inputs/outputs and four lengths. The kernel checks whether these parameters are valid and the lengths match the lengths pre-defined in the driver's binary file. Then these parameters are passed to the external method according to the `selector` if all checks are passed. If the kernel detects that invalid parameters are supplied, `IOConnectCallMethod()` returns fail immediately.

- Shared memory. User space programs can ask the user clients for shared memory via `IOConnectMapMemory()`. The driver should override its `::clientMemoryForType()` and manages its shared memory itself. In the past many drivers have problems dealing with shared memory and the most common type is race condition.

- Notification. A User space program can register a notification port for a user client instance via `IOConnectSetNotificationPort()` and receives notification messages on this port later when a certain event happens . The first vulnerability used in Pwnfest 2016 is related to this notification mechanism and will be further discussed in §4.

2.3 Mitigations

Modern operating systems have spent great effort on hardening their kernels and have implemented lots of mitigations that make the vulnerabilities much harder to exploit or even make some kinds of vulnerabilities not exploitable. In this part, some general mitigations will first be discussed and then new mitigations added in macOS Sierra will be discussed in detail.

KASLR. Address Space Layout Randomization [1] (ASLR) which is a computer security technique involved in preventing exploitation of memory corruption vulnerabilities. KASLR means that this randomization is applied in the kernel space. It is first introduced in OS X Mountain Lion (Mac OS X 10.8). However, KASLR is a little bit weaker than user space ASLR because the kernel and the kernel extensions share the same slide while user space ASLR ensures that each module (the main program and the dynamic libraries) has a unique slide which makes it harder for attackers to determine the target address to jump/call.

W^X. Write XOR eXecute [5] (W^X) is the name of a security feature in macOS since OS X 10.4. It is a memory protection policy which enforces each memory page to be either writable or executable but not both. This policy applies to both user space and kernel space memory pages including the stack and the heap.

SMEP. Supervisor Mode Execution Protection [3] (SMEP) is a CPU-level feature which prevents the kernel from executing code in user space. The 20th bit of the CPU's control register CR4 indicates the state of SMEP. If this bit is set, execution of code in a higher ring generates a fault.

SMAP. Supervisor Mode Access Protection [2] (SMAP) is another CPU-level feature which prevents the kernel from accessing data in the user space. It is implemented to complement SMEP which only prevents X (execute) and extends the protection to RW (read & write). SMAP is indicated by the 21st bit of CR4. SMAP is only available for CPUs newer than Intel's Broadwell microarchitecture. In other words old macs are not protected by SMAP.

SIP. System Integrity Protection [4] (SIP or rootless) is a security feature introduced in OS X El Capitan. It protects system files and directories that are flagged for protection. Even a process with root privilege cannot modify the files/directories protected by SIP unless the process holds a special entitlement named com.apple.rootless.install which is only distributed by Apple and is only contained in few products developed by Apple.

New mitigations - hardened heap. First of all, all zones in the kernel heap use metadata now. A page metadata struct represents a virtual page (4KB). According to the definition of `zone_page_metadata` (Listing 1), different page metadata can be linked together by `pages`. Each virtual page maintains a freelist and `freelist_offset` specifies the head of the free list in this page. `free_count` indicates how many free elements are available in this page. `zindex` indicates what zone this page belongs to. `real_metadata_offset` and `page_count` are used in multipage allocation. By mapping a virtual page to a page metadata struct the kernel is able to track all the memory it allocates and eliminates the free-to-wrong-zone attack [23] (Figure 1), which can now be detected by comparing the `zindex` in a page's metadata with the zone the memory is about to free to. Secondly, the zone allocator now poisons freed memory. The freed memory in `kalloc` zones whose size is smaller than a CPU-cache-related value will be poisoned to `0xdeadbeefdeadbeef`. Zones of larger sizes will be poisoned periodically and the period is related to zone's size. This change in the zone allocator makes uninitialized use vulnerabilities impossible to exploit in small zones and harder to exploit in large zones. In OS X EI Capitan, the first 8 bytes of a free element is a free pointer. It points to the next free element in the same zone. In macOS Sierra, this pointer is xor-ed with the `zp_nopoison_cookie` and thus it always points to an invalid address. This change eliminates the point-to-free-vtable attack (Figure 2), which is a useful and stable exploitation technique in exploiting UAF vulnerabilities, especially UAFs caused by race condition. This attack takes advantages of the fact that the virtual table lies at the same position as the free pointer. By putting the fake virtual table in the next free element, the execution flow will be hijacked if a virtual call is invoked on the freed object.

Listing 1: Definition of struct page_metadata

```
struct zone_page_metadata {
    queue_chain_t pages;
    union {
        uint32_t freelist_offset;
        uint32_t real_metadata_offset;
    };
    uint16_t free_count;
    uint8_t zindex;
    uint8_t page_count;
};
```

3 An Analysis of Uninitialized Use

Uninitialized use is a common programming mistake and some types of uninitialized use are really hard to discover. In this part, the common causes and the potential threats will be discussed in detail.

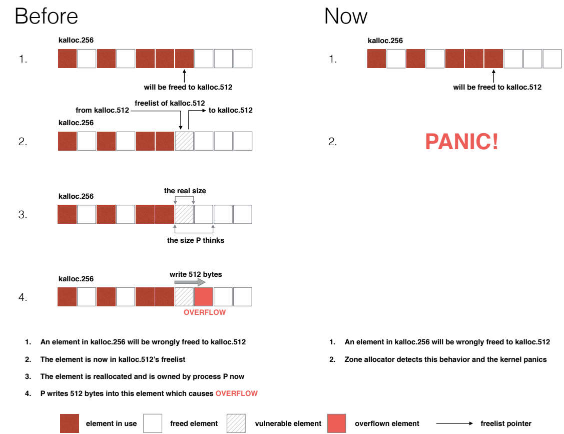

Figure 1: Free-to-wrong-zone attack

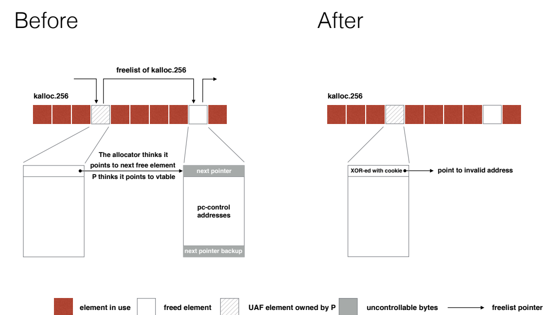

Figure 2: Point-to-fake-vtable attack

3.1 Common Types

There are five types of coding mistakes that will cause uninitialized use:

1. Missing field initialization or partial initialization. In Listing 2, the field `a` is not initialized and then `v` is `memcpy-ed` into `out_buffer`. If this is a function in the kernel space and `out_buffer` is eventually sent to the user space, this causes 8-byte uninitialized kernel heap data leaking to the user space. Actually, the first vulnerability used in our exploitation is of this type.

2. Uninitialized stack variables. This type of uninitialized use refers to a situation that a programmer allocates a stack variable but does not initialize it before using this stack variable (Listing 3). The second vulnerability used in our exploitation is of this type.

Listing 2: Partial initialization

```
void vul_f(char* out_buffer){
    typedef struct vul{
        uint64_t a;
        uint32_t b;
        uint32_t c;
    }vul_t;

    vul_t* v = malloc(sizeof(vul_t));
    v->b = 0x1234;
    v->c = 0x5678;
    memcpy(out_buffer, v, sizeof(vul_t));
}
```

Listing 3: Uninitialized stack variables

```
uint64_t vul_f(){
    uint64_t local_vul;
    not_always_initialize(&local_vul);
    return local_vul;
}
```

3. Buffer partial copy. In Listing 4, a buffer `vul_buffer` is allocated from the heap with a fixed size `MAX_LABEL_LENGTH`. However, the memcpy copies `length` bytes of data into `vul_buffer` and if `length` is smaller than `MAX_LABEL_LENGTH`, this will leave some bytes at the end of `vul_buffer` uninitialized.

Listing 4: Buffer partial copy

```
char* vul_f(char* short_buffer, int length){
#define MAX_LENGTH 0x20
    char* vul_buffer = malloc(MAX_LENGTH);
    memcpy(vul_buffer, short_buffer, length);
    return vul_buffer;
}
```

4. Buffer over copy. In Listing 5, a buffer `over_copy_buffer` is allocated on the stack with a fixed size `0x200`. The function `init` initialize `over_copy_buffer` and return the actual length of `over_copy_buffer` in `length`. This `length` is supposed to be used in `memcpy` but the programmer makes a mistake and supplies `memcpy` with a fixed size `0x200`. This also copies some uninitialized values on the stack which causes an info leak.

Listing 5: Buffer over copy

```
void vul_f(char* out_buffer){
    // out_buffer can hold up to 0x200 bytes
    char over_copy_buffer[0x200];
    int length = init(over_copy_buffer);
    // should use length instead of 0x200
    memcpy(out_buffer, over_copy_buffer, 0x200);
}
```

5. Struct padding bytes. This is a kind of uninitialized use which is not caused by coding mistakes but a complier feature. Generally, it is much faster to access an aligned memory address by the processor than an unaligned address. For better performance, the complier tends to add padding bytes within a structure to properly align its fields. In Listing 6, the complier pads 7 bytes at the end of field `b` so that the whole structure `vul` is aligned to 8-bytes (on x86_64). Although the following code initialized all the fields of `vul`, however the last 7 bytes will never be initialized. All 16 bytes are `memcpy`-ed into `out_buffer` thus causing a leak. This type of uninitialized use is really hard to notice by the programmers since the padding bytes are added by the complier and the programmers are not aware of the existence of these padding bytes.

Listing 6: Struct padding bytes

```
void vul_f(char* out_buffer){
    typedef struct vul{
        uint64_t a;
        boolean_t b;
    }vul_t;

    vul_t* v = malloc(sizeof(vul_t));
    v->a = 0x1234;
    v->b = true;
    memcpy(out_buffer, v, sizeof(vul_t));
}
```

4 Exploitation

Uninitialized use issue is actually small negligence of programmers but it may cause really bad consequences. Some types of uninitialized use may cause info leaks which defeat (K)ASLR while some types may lead to code execution. In this section, we will show you how to exploit the macOS kernel with two uninitialized use vulnerabilities. In our exploitation, two vulnerabilities were exploited - one for an info leak and another for code execution. Both could be classified as uninitialized use vulnerabilities. We turned the first vulnerability into an info leak since the uninitialized data can be retrieved by a user space program. We first found an appropriate object to taint the kernel heap with some sensitive information (function pointer). Then we figured out a way to trigger the vulnerability inside the WebProcess sandbox and retrieved the function pointer. We subtracted the base address in the kernel binary from this pointer and obtained the kernel slide. We used the second vulnerability to execute arbitrary code in the kernel. We first figured out a way to taint the kernel stack with an address of a object we faked on the kernel heap. Then we faked a virtual table in the fake object and hijacked the execution flow when a virtual call was invoked. Next, we used a small chain of ROP gadgets to disable SMEP and SMAP. Finally, we jumped to the shellcode in the user space, escalated the privilege to ROOT, did some fix-up and then exited.

4.1 Part I CVE-2017-2357

In this part, we show how we exploited an uninitialized use vulnerability caused by partial initialization by turning it into an info leak and using it to bypass KASLR. The vulnerability was in IOAudioFamily ($\leq$204.4) [8], whose source code is available on Apple's OpenSource website. The driver for IOAudioFamily is reachable from the WebProcess sandbox.

4.1.1 Vulnerability Analysis

According to §2, IOKit provides a notification mechanism for the drivers in the kernel to send notification messages to the ports registered to them. IOAudioFamily, which is a base family for audio drivers, supports this notification mechanism as well. `IOAudioControlUserClient`, which is a user client of IOAudioFamily, allows user space programs to register a notification port via `IOConnectSetNotificationPort()`. `IOAudioControlUserClient` overrides `::registerNotificationPort()` to implement its own mechanism. Listing 7 is a code snippet of ::registerNotificationPort():

Listing 7: A snippet of ::registerNotificationPort()

```
IOReturn IOAudioControlUserClient::
     registerNotificationPort(mach_port_t port,
     UInt32 type, UInt32 refCon)
{
    ...
    if (notificationMessage == 0) {
        notificationMessage = (
             IOAudioNotificationMessage *)
IOMallocAligned(sizeof(IOAudioNotificationMessage),
     sizeof (IOAudioNotificationMessage *));
        if (!notificationMessage) {
            return kIOReturnNoMemory;
        }
    }
    notificationMessage->messageHeader.msgh_bits =
         MACH_MSGH_BITS(MACH_MSG_TYPE_COPY_SEND, 0);
    notificationMessage->messageHeader.msgh_size =
         sizeof(IOAudioNotificationMessage);
    notificationMessage->messageHeader.
         msgh_remote_port = port;
    notificationMessage->messageHeader.
         msgh_local_port = MACH_PORT_NULL;
    notificationMessage->messageHeader.msgh_reserved
         = 0;
    notificationMessage->messageHeader.msgh_id = 0;
    notificationMessage->ref = refCon;
    ...
}
```

The function first checked if `notificationMessage` existed and allocated one if not via `IOMallocAligned()`. However, `IOMallocAligned()` never zeroes out the memory it allocates. Then the function set some fields of `notificationMessage` and returned. It seems clear that this function was designed to prepare a notification message - it set the message's bits, size, destination port. Moreover, `IOAudioControlUserClient` would reuse this `notificationMessage` during its life cycle since this `notificationMessage` would only be allocated the first time `registerNotificationPort()` is called . Listing 8 shows the definition of `IOAudioNotificationMessage`.

Listing 8: Definition of IOAudioNotificationMessage

```
typedef struct _IOAudioNotificationMessage
{
    mach_msg_header_t messageHeader;
    UInt32  type;
    UInt32  ref;
    void *  sender;
} IOAudioNotificationMessage;
```

According to the definition, an `IOAudioNotificationMessage` object contains a mach_msg header `messageHeader`, a 4 bytes uint `type`, a 4 bytes uint `ref` and an 8 bytes pointer `sender`. `registerNotificationPort()` only initialized `messageHeader` & `ref` and left `type` & `sender` uninitialized.

When the system audio volume changes, `IOAudioControlUserClient` will send a notification message to the port registered before via `sendChangeNotification()` (Listing 9). `sendChangeNotification()` set the `type` field of `notificationMessage` and then sent this `notificationMessage` to user space via `mach_msg_send_from_kernel()`.

Listing 9: A snippet of sendChangeNotification

```
void IOAudioControlUserClient::sendChangeNotification
     (UInt32 notificationType)
{
    if (notificationMessage) {
        kern_return_t kr;

   notificationMessage->type = notificationType;
        kr = mach_msg_send_from_kernel(&
             notificationMessage->messageHeader,
             notificationMessage->messageHeader.
             msgh_size);
        if ((kr != MACH_MSG_SUCCESS) && (kr !=
             MACH_SEND_TIMED_OUT)) {
            IOLog("IOAudioControlUserClient:
                 sendRangeChangeNotification() failed
                 - msg_send returned: \%d\n", kr);
        }
    }
}
```

Note that `sender` never got initialized since `notificationMessage` was allocated and the message was then sent to the user space, which would leak 8 bytes of kernel heap data to user space.

To recap, open an `IOAudioControlUserClient` via `IOServiceOpen()`, register a notification port via `IOConnectSetNotificationPort()`, change the system audio volume (or other actions that can fire the audio

event) and our user space program will retrieve a message with 8 bytes of kernel heap data.

4.1.2 Exploiting the Vulnerability

Exploiting this vulnerability is not trivial. First, after a complete manual inspection, we confirmed that `sender`, which could be totally under control, had no effects on kernel execution. In other words, we could not hijack kernel execution through `sender`. Second, although we could achieve a stable leak through this vulnerability, the leaked information is limited. We still need to transform a limited leak to a KASLR bypass.

Collect Basic Info of the Vulnerable Object. First, we have to find out the size of `IOAudioNotificationMessage`. Note that `notificationMessage` is allocated via `IOMallocAligned()` which will align the memory according to the second parameter of `IOMallocAligned()`. In this case, the alignment is the size of a pointer, which is 8 bytes. Moreover, `IOMallocAligned()` allocates an extra `0x10` bytes (in debug version of kernel it is `0x18` bytes) to store some metadata of this allocation (the allocation size and the allocation address). Furthermore, the address returned by `IOMallocAligned()` is `0x10` bytes away from the real allocation address to jump over the metadata header. The size of `IOAudioNotificationMessage` is `0x30` and taking the header and the alignment into consideration, `IOAudioNotificationMessage` is eventually allocated in `kalloc.80` and the offset of `sender` is `0x38` (Figure 3).

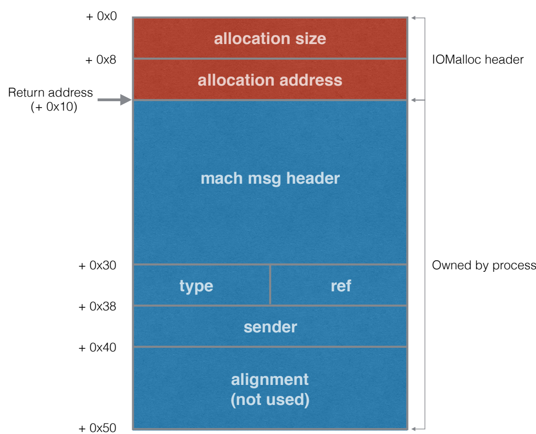

Figure 3: IOAudioNotificationMessage layout

Find Candidate Object and Leak Sensitive Information. In order to get some data we are interested in, we have to taint the kernel heap before `notificationMessage` is allocated. In XUN, the heap memory is divided into different zones by size. Objects are allocated into different zones according to their sizes. The `kalloc.80` zone holds objects whose size is bigger than `0x40` bytes and smaller than or equal to `0x50` bytes. We developed a small tool to extract the sizes of IOKit objects with the help of IDA pro. We call an object a candidate object if its size meets the requirement. We found 93 candidate objects in the kernel and kernel extensions (KEXT).

We call an object a target object if it meets following requirements:

- It must contains an interesting value at the offset of 0x38 like a heap address, an address in the TEXT segment or something else.
- It must be reachable from a WebProcess sandbox.

For the first requirement, an address in the TEXT segment would be the best choice since this directly gives us the kernel slide. A heap address is also a good choice if we have an arbitrary-read primitive. For the second requirement, since the kernel is reachable from any sandbox, we first search for target objects in the kernel. For an additional requirement, if the object is sprayable, the stability and success rate will be greatly improved.

We finally found a target object called `OSSerialize` which is used when the kernel serializes data. Its size is `0x50` and the member at `+0x38` is an `Editor` called `editor` (Listing 10). According to the definition of Editor (Listing 11), `editor` is actually a function pointer. In `IORegistryEntryCreateCFProperties()`, an `OSSerialize` object is first allocated and then the `editor` is set to the address of `GetPropertiesEditor()`, which is another function in the kernel. When `IORegistryEntryCreateCFProperties()` exits, the `OSSerialize` object is then deallocated. If we can get `notificationMessage` to reuse this freed memory and we will retrieve the address of `GetPropertiesEditor()`. By subtracting the base address of `GetPropertiesEditor()` in the kernel binary from the leaked address we can obtain the kernel slide.

Listing 10: Definition of OSSerialize

```
class OSSerialize : public OSObject
{
    ...
private:
    char * data;
    unsigned int length;
    unsigned int capacity;
    unsigned int capacityIncrement;
    OSArray * tags;
    bool binary;
    bool endCollection;
    Editor editor;
    void * editRef;
    ...
}
```

Listing 11: Definition of Editor

```
typedef const OSMetaClassBase * (*Editor)(
          void* reference,
          OSSerialize* s,
          OSCollection* container,
          const OSSymbol* name,
          const OSMetaClassBase* value);
```

Make the Memory Flashing. However, `OSSerialize` is not perfect because we cannot control when to free the `OSSerialize` object. The object is allocated and deallocated within a single function call. During the interval between the deallocation and reallocation, the freed memory might be used by other objects since `kalloc.80` is an active zone. Moreover, we cannot spare large numbers of `OSSerialize` objects for the same reason.

All these disadvantages make the exploitation unstable. However, we develop a technique named "flashing". It keeps allocating and deallocating the `OSSerialize` objects by keep calling `IORegistryEntryCreateCFProperties()` in several threads. This makes some memory regions be allocated and deallocated quite frequently and `OSSerialize` objects look like flashing on the heap.

By flashing the heap, as long as `notificationMessage` falls into these flashing regions, we are able to leak the function pointer stably.

Trigger the Vulnerability in Sandbox. As we mentioned before, we have to fire an audio event to trigger the vulnerability by changing the system volume or some other operations. We can achieve this by pressing the "volume up" or the "volume down" button on our keyboard. However in a security contest like Pwnfest or pwn2own, the participants are not allowed to interact with the PC when the exploitation is running. So we need to do this in a programmatically manner.

We can also trigger the vulnerability by setting a value for a property called `IOAudioControlValue` via `IORegistryEntrySetCFProperties()`. However, the WebProcess sandbox does not allow to set this property. By greping `IOAudioControlValue`, we found `coreaudiod`, which is a daemon service located in `/usr/sbin`, can set this property. `coreaudiod` is also responsible for mach service `com.apple.audio.coreaudiod`. Luckily, according to the WebProcess's sandbox file, WebProcess is allowed to communicate with this service. As a result, we can instruct `coreaudiod` to change the volume and trigger the notification event.

A One-shot-more-kill Way. As we discussed before, an `IOAudioControlUserClient` object can only hold one `notificationMessage` and this means a single user client instance can leak only once.

We use a small trick to leak several times but only trigger the vulnerability once. We first create many

```
;; Various services required by AppKit and other frameworks
(allow mach-lookup
       (global-name "com.apple.DiskArbitration.diskarbitrationd")
       (global-name "com.apple.FileCoordination")
       (global-name "com.apple.FontObjectsServer")


       (global-name "com.apple.PowerManagement.control")
       (global-name "com.apple.SystemConfiguration.configd")
       (global-name "com.apple.SystemConfiguration.PPPController")
       (global-name "com.apple.audio.SystemSoundServer-OSX")
       (global-name "com.apple.audio.VDCAssistant")
       (global-name "com.apple.audio.audiohald")
       (global-name "com.apple.audio.coreaudiod")
```

Figure 4: WebProcess can talk to coreaudiod

`IOAudioControlUserClient` objects through `IOServiceOpen()`. Then we register different notification ports for each `IOAudioControlUserClient` instance via `IOConnectSetNotificationPort()`. Each `IOAudioControlUserClient` allocates a `notificationMessage` with 8 bytes uninitialized data. If we trigger the vulnerability, each `IOAudioControlUserClient` will send a notification message to our user space program with 8 bytes leak data. As a result, we can leak several times by triggering the vulnerability once.

Chain All Pieces Together. To chain up, we use the following four steps to exploit this vulnerability and bypass KASLR:

1. Create 8 separate threads which keep calling `IORegistryEntryCreateCFProperties()`.

2. Create 32 `IOAudioControlUserClient` instances and register a unique port for each user clients instance.

3. Trigger the vulnerability by communicating with `coreaudiod` service.

4. Retrieve the leak data and treat the most common address as the address of `GetPropertiesEditor()`.

5. Subtract the base address of `GetPropertiesEditor()` from the leaked address and obtain kernel slide.

4.2 Part II CVE-2017-2358

In this part, we will show how to exploit an uninitialized use vulnerability caused by uninitialized stack variable to gain arbitrary code execution. The vulnerability lied in kernel extension `AMDRadeonXx000` (x may vary on different platforms). The extension is the driver for AMD's graphic card and it is closed-source. The driver is also reachable from the WebProcess sandbox. We take `AMDRadeonX4000` as an example while other versions are almost the same.

4.2.1 Vulnerability Analysis

As we mentioned in §2, drivers provide external methods for user space programs to call. `AMDRadeonX4000` also implements several external methods. The vulnerability existed in the external method `AMDRadeonX4000_AMDSIGLContext::SurfaceCopy()` with a `selector` of 0x201. This external method eventually invoked `AMDRadeonX4000_AMDAccelShared::SurfaceCopy()`, in which a local stack variable was not initialized before it was used.

In Listing 12 (irrelevant code is omitted), two local variables, `v46` and `v47`, were declared but they were not initialized. Then their memory addresses were passed to `IOAccelShared2::lookupResource()`. The following check ensured `v46` and `v47` are both non-zero and two virtual calls were invoked on `v46` and `v47` afterwards. `SurfaceCopy()` assumed that `lookupResource()` would always initialize `v46` and `v47`. `lookupId()` (Listing 13), which actually implements `lookupResource()`, took 2 arguments `a2` and `a3`. `a2` was a resource id and `a3` was a stack address to store the resource object. `lookupId()` first checked if the required resource id was smaller than the maximum id. It would returned 0 immediately if `a2` was invalid. Then `a2` was treated as an index into a resource array and `_RAX` was retrieved from this array. If `_RAX` was not zero, it was treated as a valid resource object and it was stored in `*a3`. If `_RAX` was zero the function returned 0 immediately and `*a3` would not be set. In short, `lookupResource()` would initialize the resource object only if a valid resource id was supplied. However, in `SurfaceCopy()`, we can control the id passed to `lookupResource()` since `a2` is user's structure input. As a result, if we supply a structure input which contains an invalid resource id, `v46` and `v47` will never be initialized and invoking a virtual call on `v47` or `v46` causes a panic.

4.2.2 Exploit the Vulnerability

Control the Stack. The first thing we should think about is how to put the stack in a control state because generally the values on the stack are almost random - they are "junk" values left by previous function calls. In order to control pc, we have to first control the value of the uninitialized stack variable.

We found an external method (selector 7333) in `AGPMClient` that could help us control the stack. According to Listing 14, it copies user-controlled bytes (specified by `a2->structureInputSize` and at most 4096 bytes) of user-controlled non-zero data (specified by `a2->structureInput`) onto the stack. By calling this function, we can easily control the value of the uninitialized stack variable.

Listing 12: A snippet of SurfaceCopy() (decompiled by IDA Pro)

```
__int64 __fastcall AMDRadeonX4000_AMDAccelShared::
    SurfaceCopy(IOAccelShared2 *this, __int64 a2,
    __int64 a3, __int64 a4)
{
  // local variable declaration
  ...
  IOAccelResource2 *v46; // [rsp+38h] [rbp-88h]@9
  void *v47; // [rsp+40h] [rbp-80h]@9
  ...

  // code
  ...
  IOAccelShared2::lookupResource(this, *(_DWORD *)(a2
       + 8), &v47);
  IOAccelShared2::lookupResource(this, *(_DWORD *)(a2
       + 4), (void **)&v46);
  v6 = -536870206;
  if ( !v47 || !v46 )
    goto LABEL_41;
  v12 = (*(__int64 (**)(void))(*(_QWORD *)v47 + 368LL
      ))();
  v13 = (*(__int64 (**)(void))(*(_QWORD *)v46 + 368LL
      ))();
  ...
}
```

Listing 13: A snippet of lookupId() (decompiled by IDA Pro)

```
char __fastcall IOAccelNamespace::lookupId(
    IOAccelNamespace *this, unsigned int a2, void **
    a3)
{
  if ( *((_DWORD *)this + 6) <= a2 )
    return 0;
  _RAX = *(void **)(*((_QWORD *)this + 2) + 8LL * a2)
      ;
  if ( !_RAX )
    return 0;
  *a3 = _RAX;
  __asm { prefetcht0 byte ptr [rax] }
  return 1;
}
```

Listing 14: A snippet of the method in AGPMClient (F5 code generated by IDA Pro)

```
case 7333:
   kprintf("kAGPMSetPlimit plimit = %llu type = %s\n
        ", *a2-> scalaInput, a2-> structureInput);
   v14 = (char *)&v19 - ((a2->structureInputSize + 1
         + 15LL) & 0xFFFFFFFFFFFFFFF0LL);
   strncpy(v14, (const char *)a2->structureInput, a2
        ->structureInputSize);
```

The next question is which value should we "initialize" for the uninitialized stack variable. Generally we have two choices. One choice is a pointer that points to an object we fake on the heap. Another is a pointer points to a real object on the heap. The aim of the second choice is to hopefully build an arbitrary read/write primitive by turning this vulnerability into a type confusion. However, `SurfaceCopy()` is a really complicated

function and it is hard to find an candidate object that can survive the following several virtual calls. So we chose the first choice. We fake an object and a virtual table on the heap and taint the stack to make the uninitialized stack variable point to this fake object.

Fake Object(s). Assuming that we fake an object in the kernel heap, how can we know the exact address of our fake object since the kernel heap address is also randomized during each boot. We also have two choices here.

The first choice is to reuse the info leak vulnerability to leak the address of our fake object. The info leak vulnerability gives us the capability of reading 8-bytes from an object in `kalloc.80` at the offset of `0x38`. So we have to find an object which locates in `kalloc.80` and contains a buffer pointer at the offset of `0x38`. Also, the user space program should be able to control the content of the buffer. It is quite hard to find such a target object. Moreover, we need to first allocate large numbers of such target object which contains our target buffer and then deallocate them to taint the heap (target buffers will be deallocated automatically at the same time). Then we have to get `notificationMessage` structures to reuse this tainted memory and trigger an audio event to leak some heap addresses. Finally, we deallocate these `notificationMessage` structures and reallocate target objects and buffers to cover those leaked heap addresses. There are twice free-refill operations involved which will definitely make the exploitation unstable.

The second choice is leveraging the weakness of heap randomization. First of all, the zone starts at `zone_map_min_address` which is heavily dependent on the kernel slide (Table 6, Figure 6). The kernel slide is at most `0x20000000` (`512 MB`). Secondly, the heap allocation always starts from low address to high address and the allocation at high addresses tend to be linear (Figure 5). Thus if we first spray `512MB * 2` (`512 MB` for defragment and `512 MB` for compensation for KASLR) to raise the heap to a relatively high address then the following allocation is much easier to predict. We choose to spray `2 GB` of fake objects to ensure our fake objects always cover a relatively high heap address (we choose a fixed high address `0xffffff8060010110`).

We use `vm_map_copy` struct to spray in the kernel heap since it is following advantages:

- It is very fast. It takes less than one second to spray `4 GB` data on the heap.
- It has small side effects because it creates no additional objects.
- We can control the size of the spray data (up to `0x1000` bytes) and the content of the data as well.

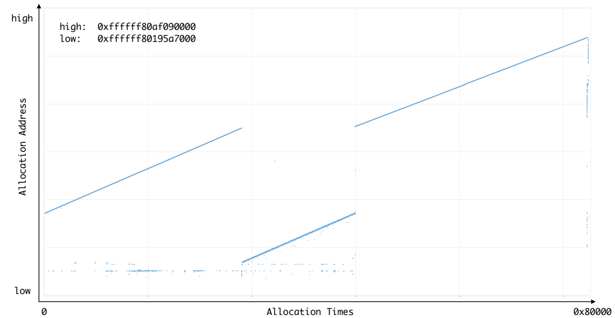

Figure 5: Linearity of heap allocations

The only restriction of `vm_map_copy` is that we cannot control the first `0x18` bytes of the data since it is the header struct used by `vm_map_copy`. However, we can adjust the value on the stack and jump over this header.

We put our fake object at the offset of `0x110` to jump over the uncontrollable header and set the value on the stack to be `0xffffff8060010110`. The first 8 bytes of our fake object is a virtual table pointer. We point it to the offset of `0x800`, where we put our ROP [20] stack.

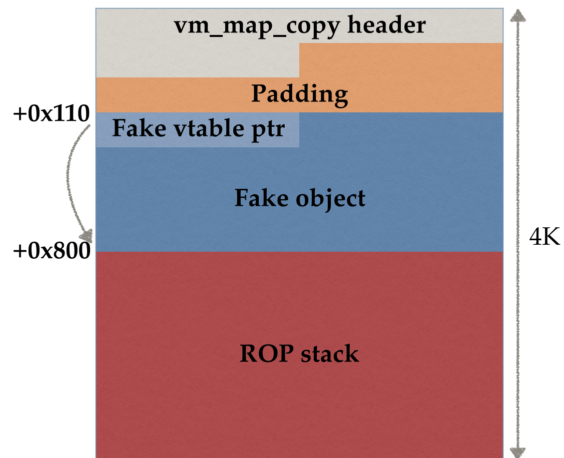

Figure 7: Layout of our fake object

The ROP Chain. In the ROP chain, we first use some JOPs [12] to store several important registers and then call a stack pivot. The ROP chain disables SMEP & SMAP and then returns to the shellcode in the user space. After the shellcode returns, the rest of ROP chain re-enables SMEP & SMAP and returns to `_thread_exception_return()` to get back to user mode. The whole exploitation finishes.

By clearing the 20th and 21st bits of Control Register 4 (CR4) we are able to disable SMEP & SMAP and

Table 4: Zone start address vs. kslide

#	zone_map_min_address (with kslide)	kslide	zone_map_min_address (without kslide)
1	0xffffff8012b0f000	0xcc00000	0xffffff8005f0f000
2	0xffffff800a495000	0x4600000	0xffffff8005e95000
3	0xffffff801eeeb000	0x19000000	0xffffff8005eeb000
4	0xffffff801cb6c000	0x16c00000	0xffffff8005f6c000
5	0xffffff80189cc000	0x12a00000	0xffffff8005fcc000
6	0xffffff80210d8000	0x1b200000	0xffffff8005ed8000
7	0xffffff80116be000	0xb800000	0xffffff8005ebe000
8	0xffffff8024429000	0x1e400000	0xffffff8006029000
9	0xffffff80229d4000	0x1ca00000	0xffffff8005fd4000
10	0xffffff800aaab000	0x4c00000	0xffffff8005eab000
11	0xffffff801721f000	0x11200000	0xffffff800601f000
12	0xffffff80149b2000	0xea00000	0xffffff8005fb2000
13	0xffffff801beae000	0x16000000	0xffffff8005eae000
14	0xffffff8020078000	0x1a000000	0xffffff8006078000
15	0xffffff800e5fd000	0x8600000	0xffffff8005ffd000
16	0xffffff801ad63000	0x14e00000	0xffffff8005f63000
17	0xffffff800aeea000	0x5000000	0xffffff8005eea000
18	0xffffff800c6b5000	0x6800000	0xffffff8005eb5000
19	0xffffff801f741000	0x19800000	0xffffff8005f41000
20	0xffffff80158b6000	0xfa00000	0xffffff8005eb6000

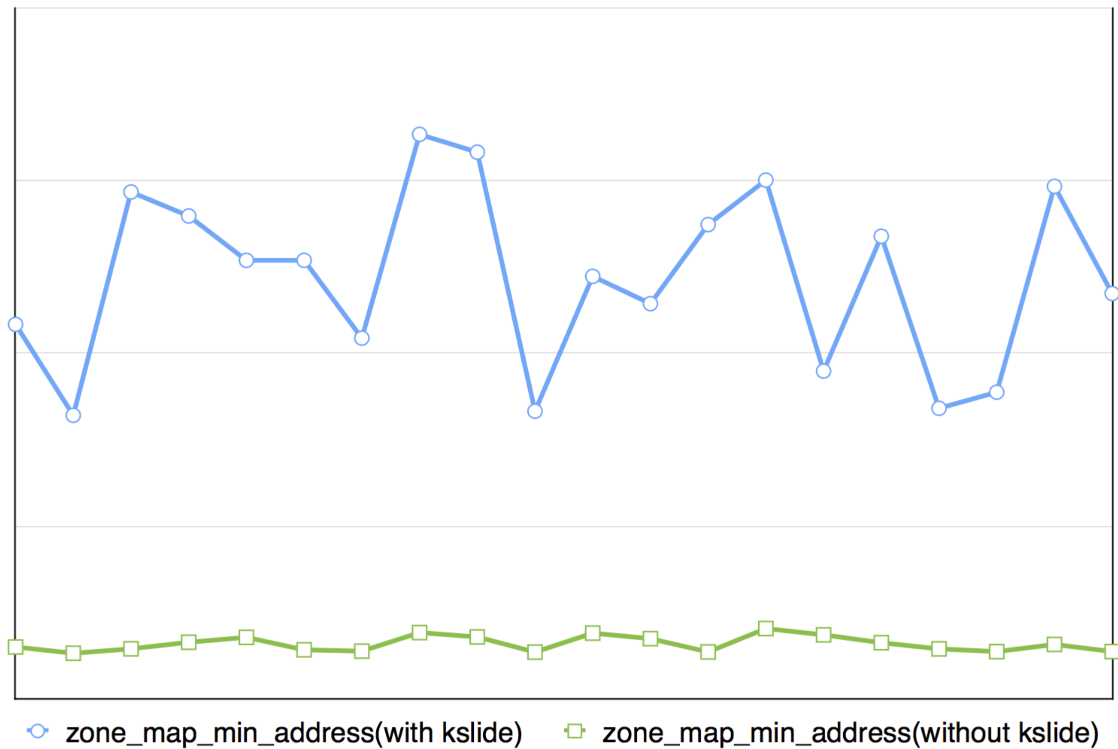

Figure 6: Zone start address vs. kslide

then jump to our shellcode in the user space. There are gadgets in the kernel which help us manipulate CR4:

- read CR4: `mov rax, cr4 ,..., ret`
- Write CR4: `mov cr4, rax ,..., ret`

5 Discussion

The whole exploitation takes the following seven steps:

1. Taint the kernel heap by allocating large numbers of the target objects and then deallocating them all. The target object is allocated in `kalloc.80` and contains a function pointer at the offset `0x38`
2. Create many user client instances and register different ports for them to allocate the vulnerable `notificationMessage` structures which reuse the tainted heap memory.
3. Trigger an audio event by instructing the `coreaudiod` service to set the system volume and retrieve an address of a kernel function which helps bypass KASLR.
4. Spray our fake objects which contain a fake virtual table and a ROP stack in the kernel heap and set up the shellcode in the user space to prepare for the second part of our exploitation.
5. Taint the kernel stack by calling an external method in `AGPMClient`.
6. Trigger the second vulnerability by calling the vulnerable external method in `AMDRadeonX4000_AMDSIGLContext` user client.
7. Escalate the privilege and do some cleanup in the shellcode.

There are two special notes for this exploitation. The first one is what we call "common stack hazard". In our tests, we found that the stack used in kernel mode varied from time to time. Sometimes the kernel might use a common stack which might have been used by other kernel threads before while sometimes the kernel might use a totally clear stack. If we enter the kernel mode with a clear stack in step 6 the kernel will panic immediately. We used an eclectic approach to avoid these bad cases by tainting the kernel stack several times in step 5 to cover as many kernel stacks as possible. The second one is about the cleanup. The `SurfaceCopy()` holds a lock of AMD driver and we have to drop it in our shellcode. Otherwise, the screen will freeze after the exploitation returns since the other functions in AMD driver are all blocked by the lock.

The exploitation is very reliable. According to our test, the exploitation only failed 3 times out of 50 tries, which gives 94% success rate. There are 3 weaknesses that could cause the exploitation to fail:

- Kernel slide leakage failure. We must let `notificationMessage` reuse the heap memory tainted by us before. However, if the `notificationMessage` does not reuse the heap memory, we will get the wrong kernel function address thus the kernel slide calculated will also be incorrect and exploitation will definitely fail. We are able to reduce the possibility of this situation by allocating large numbers of `notificationMessage` objects and checking whether the calculated kernel slide is valid(it should be 2MB aligned and smaller than 0x20000000). According to our test, we never fail due to an incorrect kernel slide.
- Heap spray failure. We assume the fixed address(`0xffffff8060010110`) is always occupied by our fake objects. However, in theory, this fixed address could also be used before we spray. In this case, the exploitation would fail. According to our test, the fixed address can always be occupied by our fake objects.
- Common stack hazard. According to our test, the exploitation failed 3 times and they are all caused by "common stack". The "common stack" seems to be a feature of macOS and although we taint the kernel stack several times in step 5, there are still cases that we enter kernel mode with a clear stack. The "common stack" is the Achilles' heel of the exploitation.

For the first vulnerability, the technique used is not universal and it takes lots of time hunting for the target object which leaks a function pointer. For the second vulnerability, the technique of spraying memory and the technique of locating fake object are universal and can be used in other exploitations. The technique used to disable SMEP & SMAP is dependent on the kernel binary since it requires gadgets to manipulate CR4.

As we can see in the analysis, these two vulnerability are both caused by small coding mistakes or negligence. As a result, it is always a good programming habit to zero out the newly allocated memory and set an initial value for a local variable before it is used by others.

6 Related Work

Exploitation techniques. The main idea of exploiting uninitialized use vulnerabilities is to control the uninitialized variable. Kees Cook [13] gave a talk about exploiting an uninitialized stack variable vulnerability in detail.

He overlapped the uninitialized stack variable in the vulnerable code path by a local variable(of type char[32]), in another code path within the same function. By controlling the uninitialized stack variable, he could write an arbitrary value to an arbitrary address. Mercy [18] demonstrated the stack-reuse technique to exploit uninitialized stack vulnerabilities. Halvar Flake [14] proposed an algorithm to find out all the code paths which had overlapped stack frame within a function. By analyzing the offsets of the local variables in the overlapped stack frame this helps to quickly find the way to control the uninitialized variable.

Kernel data leak detection and prevention. PaX proposed a plugin called STACKLEAK [22] which aimed at preventing data leakage from the kernel stack by clearing the used kernel stack before return to user space. Split kernel [15] however, clears the stack frame at ever time a function is called. Both STACKLEAK and Split kernel are able to prevent data leaking from kernel but they could impose significant overhead. Another weakness is that they cannot detect nor prevent data leakage from the kernel heap. K Lu proposed UniSan [17], which is a compiler-based approach to eliminate data leaks from both kernel stack and heap. However, UniSan requires re-compiling the source code and cannot be applied to those close-source projects.

Uninitialized use vulnerabilities discovery. XB Chen [24] proposed two kinds of fuzzing strategies, active fuzzing and passive fuzzing, to fuzz iOS IOKIT bugs. Active fuzzing refers to supplying all kinds of possible test cases to IOKIT and passive fuzzing refers to hook some kernel APIs and modify the parameters passed to IOKIT. However uninitialized use vulnerabilities do not always crash the kernel hence they are far more difficult to be caught by fuzzing than code execution vulnerabilities.

7 Conclusion

This paper discusses the whole exploitation chain of a local-privilege-escalation attack caused by two uninitialized use vulnerabilities from a Safari sandbox in detail. If combined with remote code execution vulnerabilities in Safari, an attacker can easily take over the control of a latest MacBook. This paper also presents a warning for all programmers to always keep initialization in mind when coding. Furthermore, operating systems are becoming more and more hardened and new mitigations make the vulnerabilities harder and harder to exploit, which will encourage researchers to develop new exploitation techniques.

8 Acknowledgments

We thank Ian Beer, Luca, and other researchers for their wonderful write-ups and blogs. We thank our friends, teachers and family for their encouragement and support. Thanks to Shanghai Jiao Tong University, Pwnzen and Power of Community for giving us the opportunity to do this research. This research was sponsored by ShanghaiPujiangProgram (16PJ1430800).

References

[1] Address Space Layout Randomization. `https://en.wikipedia.org/wiki/Address_space_layout_randomization`.

[2] Supervisor Mode Access Prevention. `https://en.wikipedia.org/wiki/Supervisor_Mode_Access_Prevention`.

[3] Supervisor Mode Execte Prevention. `https://en.wikipedia.org/wiki/Control_register#SMEP`.

[4] System Integrity Protection. `https://en.wikipedia.org/wiki/System_Integrity_Protection`.

[5] Write XOR Execute. `https://en.wikipedia.org/wiki/W%5EX`.

[6] XNU. `https://en.wikipedia.org/wiki/XNU`.

[7] APPLE. About App Sandbox. `https://developer.apple.com/library/content/documentation/Security/Conceptual/AppSandbox-DesignGuide/AboutAppSandbox/AboutAppSandbox.html`.

[8] APPLE. IOAudioFamily. `https://opensource.apple.com/source/IOAudioFamily/IOAudioFamily-204.4/`.

[9] APPLE. IOKit Fundamentals. `https://developer.apple.com/library/content/documentation/DeviceDrivers/Conceptual/IOKitFundamentals/Features/Features.html`.

[10] AZAD, B. physmem: Accessing Physical Memory from User Space on OS X. `https://bazad.github.io/2017/01/physmem-accessing-physical-memory-os-x/#variant-cve-2016-7617`.

[11] BEER, I. pwn4fun spring 2014 safari part ii. `https://googleprojectzero.blogspot.co.id/2014/11/pwn4fun-spring-2014-safari-part-ii.html`, Novenber 2014.

[12] BLETSCH, T., JIANG, X., FREEH, V. W., AND LIANG, Z. Jump-oriented programming: A New Class of Code-reuse Attack. In *ACM Symposium on Information, Computer and Communications Security* (2011), pp. 30–40.

[13] COOK, K. Kernel Exploitation Via Uninitialized Stack. DEFCON19.

[14] HALVAR-FLAKE. Attacks on Uninitialized Local Variables. BlackHat Europe, 2006.

[15] KURMUS, A., AND ZIPPEL, R. Kernel Hardening Wars with Split Kernel.

[16] LEVIN, J. *Mac OS® X and iOS Internals TO THE APPLE'S CORE*, first edtion ed. John Wiley & Sons, Inc., 2013.

[17] LU, K., SONG, C., KIM, T., AND LEE, W. UniSan: Proactive Kernel Memory Initialization to Eliminate Data Leakages. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 920–932.

[18] MERCY. Exploiting Uninitialized Data Bugs. whitepaper, Jan. 2006.

[19] PEIRÓ, S., MUÑOZ, M., MASMANO, M., AND CRESPO, A. Detecting Stack Based Kernel Information Leaks. In *International Joint Conference SOCO'14-CISIS'14-ICEUTE'14* (2014), Springer, pp. 321–331.

[20] PRANDINI, M., AND RAMILLI, M. Return-oriented Programming. *IEEE Security & Privacy 10*, 6 (2012), 84–87.

[21] STEPANOV, E., AND SEREBRYANY, K. MemorySanitizer: Fast Detector of Uninitialized Memory Use in C++. In *Code Generation and Optimization (CGO), 2015 IEEE/ACM International Symposium on* (2015), IEEE, pp. 46–55.

[22] TEAM, P. PaX GCC Plugins Galore. `https://pax.grsecurity.net/docs/PaXTeam-H2HC13-PaX-gcc-plugins.pdf`.

[23] TODESCO, L. Attacking the XNU Kernel in El Capitan. BlackHat Europe, 2015.

[24] XIAOBO CHEN, H. X. Find Your Own iOS Kernel Bug. SyScan360, 2012.