Exploitations of Uninitialized Uses on macOS Sierra

Zhenquan Xu, Gongshen Liu, Tielei Wang, Hao Xu
Agenda

- Background & Introduction
- Vulnerability analysis & Exploitation
- Conclusion
Agenda

* Background & Introduction
* Vulnerability analysis & Exploitation
* Conclusion
Great tools have been developed to eliminate uninitialized use vulnerabilities:

- **Unisan**: prevents kernel data leakage
- **STACKLEAK**: prevents kernel stack data leakage
- **Split Kernel**: same as STACKLEAK
PwnFest 2016

- Team up with Lokihardt for pwning Safari on macOS Sierra
- Remote code execution in Safari by Lokihardt
- Kernel code execution in Safari by us
Fixed in macOS Sierra 10.12.3

- Released January 23, 2017
- CVE-2017-2357
- CVE-2017-2358
Mitigations

- kASLR : kernel Address Space Layout Randomization
- \( W^X \) : Write XOR eXecute
- SMEP : Supervisor Mode Execution Protection
- SMAP : Supervisor Mode Access Protection
Agenda

- Background & Introduction
- Vulnerability analysis & Exploitation
- Conclusion
The vulnerability lies in IOAudioFamily (<= 204.4)

Source code: https://opensource.apple.com/source/IOAudioFamily/IOAudioFamily-204.4/

The vulnerability is caused by partial initialization
Programs in the user space can register a notification port by calling IOConnectSetNotificationPort() on an IOAudioControlUserClient object. Notification messages will be sent to the port when certain audio events occur afterwards. IOAudioControlUserClient::registerNotificationPort() will be eventually invoked in the kernel.
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;
    ...
}

typedef struct __IOAudioNotificationMessage
{
    mach_msg_header_t messageHeader;
    guint32 type;
    guint32 ref;
    guint32 *sender;
} IOAudioNotificationMessage;

notificationMessage is not zeroed out and type and sender is not initialized
Analysis

- IOAudioControlUserClient::sendChangeNotification() sends notification messages to userspace programs

```c
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);
        }
    }
}
```
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);
        }
    }
}

• **sender** is uninitialized (**notificationMessage** is partial initialized)

• the kernel sends a message with 8 bytes kernel heap data to the user space
Exploitation

• `notificationMessage` is allocated via `IOMallocAligned()` which allocates a header structure to store metadata.

• `notificationMessage` is allocated in `kalloc.80` zone

• `sender` is at +0x38
Exploitation

- Challenge 1: How to leak critical information such as kASLR slide
- Challenge 2: How to make the exploitation stable
Exploitation

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;
    ...}

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

The OSSerialize objects are allocated in kalloc.80
The member at +0x38 is a function pointer
editor is set to a kernel function in
IORegistryEntryCreateCFProperties()
Exploitation

```
OSSerialize
OSSerialize
OSSerialize
...
OSSerialize
```

function pointers
Exploitation

function pointers

freed OSSerialize

freed OSSerialize

freed OSSerialize

\vdots

\vdots

freed OSSerialize
Exploitation

notificationMessage

notificationMessage

function pointers
Exploitation

- OSSerialize is allocated and deallocated within a function call (i.e. IORegistryEntryCreateCFProperties())
- We developed a new technique called “flashing memory”
- Allocate and deallocate OSSerialize objects very frequently
CVE-2017-2358

- The vulnerability lies in AMDRadeonX\textsubscript{x000}.kext (\textit{x may vary on different platforms})

- The vulnerability is caused by an uninitialized stack variable

- Accelerator is one of the devices that we can directly access in the WebContent sandbox
Local variables $v46$ and $v47$ are not initialized

$v46$ and $v47$ are passed to LookupResource()

A vcall is invoked blindly on $v46$ and $v47$ then
Analysis

- *a2* is treated as an index into a resource array
- *a3* is set only when *a2* is valid and the corresponding resource object exists (_RAX != 0)

```c
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 { prefetch0 byte ptr [rax] }
    return 1;
}
```
Analysis

- $a_2$ is user’s structure input
- We can supply an invalid id in the structure input
- PANIC

```c
__int64 __fastcall AMDRadeonX4000_AMDAccelShared2::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))();
    ...
}
Exploitation

- The uninitialized value on kernel stack is random so we need to control the stack first

- A function (selector 7333) in the AGPM userclient can be used to taint the stack

- We are able to copy at most 4096 bytes of controlled, non-zero data onto kernel stack

```c
case 7333:
kprintf("kAGPMSetPlimit plimit = %llu type = %s\n", *a2->scalarInput, a2->structureInput);
v16 = (char *)&v22 - ((a2->structureInputSize + 1 + 15LL) & 0xFFFFFFFFFFFFFFFFULL);
strcpy(v16, (const char *)a2->structureInput, a2->structureInputSize);
```
Exploitation

0xabcdabcdabcdabcd
0xabcdabcdabcdabcd
0xabcdabcdabcdabcd
0xabcdabcdabcdabcd

Stack
used by
other functions

Stack grows
up to 4KB

0xabcdabcdabcdabcd
0xabcdabcdabcdabcd
0xabcdabcdabcdabcd
0xabcdabcdabcdabcd

0xabcdabcdabcdabcd : tainted value
Exploitation

Stack used by SurfaceCopy()

\[ \text{stack} \]

\[ \text{0abcdabcdabcdabcd} \]
\[ \text{0abcdabcdabcdabcd} \]
\[ \text{0abcdabcdabcdabcd} \]
\[ \text{0abcdabcdabcdabcd} \]

\[ \vdots \]

Stack grows

\[ \text{0abcdabcdabcdabcd} \text{: tainted value} \]
\[ \text{0abcdabcdabcdabcd} \text{: v46 & v47} \]

Stack used by other functions
Exploitation

- What value should $0x\text{abcdabcdabcdabcd}$ be?
- We set $0x\text{abcdabcdabcdabcd} = \text{the address of the object we fake on the heap}$
Exploitation

- How can we know the address of our fake objects?
- Spray several GBs of data on the heap
  - Heap randomization is weak in the kernel
  - User-controlled data at fixed, high address (we chose the address 0xffffffff8060010110)
Exploitation

- zone_map_min_address(with kslide)
- zone_map_min_address(without kslide)
Exploitation

high: 0xffffffff80af090000
low: 0xffffffff80195a7000
Exploitation

Stack used by SurfaceCopy()

Stack grows

Stack used by other functions
Exploitation

0xffffffff8060010110

+0x800

4K

vm_map_copy header

Padding

Fake vtable ptr

Fake object

ROP stack
Exploitation

- Save registers
- Pivot the stack
- Disable SMEP & SMAP
- Return to SHELLCODE in the user space
- Re-enable SMEP & SMAP
- Call _thread_exception_return() to exit
Agenda

- Background & Introduction
- Vulnerability analysis & Exploitation
- Conclusion
Conclusion

- New mitigations make exploitation more challenging but also encourage us to discover new exploit techniques
- Keep initialization in our mind while coding
- More effective tools for detecting uninitialized uses are still necessary
Credits

- Ian Beer
- qwertyoruiop
- Lokihardt
- INT 80 & windknown
- ...

SHANGHAI JIAO TONG UNIVERSITY
Thank You For Your Attention
Q&A