KeenLab iOS Jailbreak Internals: Userland Read-Only Memory Can Be Dangerous

Liang Chen(chenliang0817@hotmail.com)

Tencent Keen Security Lab

Abstract. Modern operating systems nowadays implement read-only memory mappings at their CPU architecture level, preventing common security attacks. By mapping memories as read-only, the memory owner process can usually trust the memory content, eliminating the need for boundary check, TOCTTOU(Time of check to time of use) consideration etc., with the assumption of other processes not being able to mutate read-only shared mappings in their own virtual spaces.

However, the assumption is not always correct. In the past few years, several logical issues were addressed by security community, most of which were caused by operating systems incorrectly allowing to remap the read-only memories as writable without marking them COW(copy-on-write). With operating system evolves, such issues are rare though. On the other handwith stronger and more abundant features provided by peripheral components attached to the mobile device, DMA(direct-memory-access) technology enables the ability for fast data transfer between the host and peripheral devices. In the middle of 2017, Gal Beniamini of Google Project Zero team utilized DMA to successfully achieve device-to-host attack on both Nexus 6p and iPhone 7. Unfortunately, DMA related interfaces are not exposed to userland applications directly.

With months of research, we found an exception case on iOS device, that is, Apple Graphics. At MOSEC conference in 2017, we demonstrated jailbreak for iOS 10.3.2 and iOS 11 beta 2, the latest version at that time, on iPhone 6s and iPhone 7. Details of the demonstration have never been published yet. In this talk, we will introduce indirect DMA features exposed to iOS userland, mechanism of Apple Graphics which involves read-only memories in userland, and how we exploit the flaws across several Apple Graphics components to achieve Jailbreak within iOS application sandbox.

1 Introduction

In the first part of this paper, we introduce the concepts essential to our bugs, which includes:
- Indirect DMA features exposed to iOS userland
- The implementation of IOMMU memory protection
- Notification mechanism between GPU and Apple Graphics driver

The next part will cover the details of two bugs: one in DMA mapping with host virtual memory, and another out-of-bound write issue caused by potentially untrusted userland read-only memory.

Lastly we talk about how we combine two flaws across different Apple Graphics components to achieve reliable kernel code execution from iOS application sandbox.

2 Operating system memory protection overview

Modern operating systems nowadays implement read-only memory mappings at their CPU architecture level, where specific bits are reserved for read/write/execute protections in their page table entries, and thus MMU(memory management unit) can prevent unexpected operations performed to the protected memories. On AArch64 architecture, for example, the kernel memory translation entry base addresss is stored in system register TTBR1_EL1, while each process has its own TTBR0_EL1 registry, indicating each of their address translation entry base. Access permission of each virtual page can be obtained by querying the page table and get the attribute information within the block entry[1]. For 64bit iOS systems, XNU code[2] indicates that the page table format complies with this specification.

2.1 Security mitigation

This mechanism has helped to mitigate known old school attacks. For example, by overwriting code segment contents to execute patched instructions without remapping RX page to RW is no longer possible. In another case, high privileged processes or the kernel can share memory to other low privileged process, where performance on inter-process communication is significantly improved comparing with message sending approach. By mapping memories as read-only, high privileged processes can usually trust the memory content created by itself, eliminating unnecessary security considerations such as boundary check, TOCTTOU(Time of check to time of use) issue, with the assumption of other processes not being able to mutate read-only shared mappings in their own virtual spaces.

2.2 Known vulnerabilities

In the past few years, several logical issues[3][4] were addressed by security community, most of which were caused by operating systems incorrectly allowing to remap the read-only memories as writable without marking them COW(copy-on-write). In a normal case, when user tries to remap those read-only shared memory to read/write, system should create a COW mapping so that the original physical page should never get mutated. The iOS world, when calling the mach API mach_vm_protect to change the read-only shared mapping, the XNU code handles the request correctly by checking the current memory's max_protection value and denying the attempt to set access permission higher:

```
kern_return_t
vm_map_protect(
        vm_map_t        map,
        vm_map_offset_t start,
        vm_map_offset_t end,
        vm_prot_t       new_prot,
        boolean_t       set_max)
{
...
                new_max = current->max_protection;
                if ((new_prot & new_max) != new_prot) {
                        vm_map_unlock(map);
                        return(KERN_PROTECTION_FAILURE);
                }
...
}
```

Listing 1. XNU checks max_protection

If VM_PROT_COPY is set, then mach_vm_protect will allow remapping to read-/write, but the remapped page is COW:

```
kern_return_t
vm_map_protect(
        vm_map_t        map,
        vm_map_offset_t start,
        vm_map_offset_t end,
        vm_prot_t       new_prot,
        boolean_t       set_max)
{
...
        if (new_prot & VM_PROT_COPY) {
                vm_map_offset_t         new_start;
                vm_prot_t       cur_prot, max_prot;
                vm_map_kernel_flags_t   kflags;

                /* LP64todo - see below */
                if (start >= map->max_offset) {
                        return KERN_INVALID_ADDRESS;
        }

        kflags = VM_MAP_KERNEL_FLAGS_NONE;
        kflags.vmkf_remap_prot_copy = TRUE;
        new_start = start;
        kr = vm_map_remap(map,
          &new_start,
          end - start,
          0, /* mask */
          VM_FLAGS_FIXED | VM_FLAGS_OVERWRITE,
          kflags,
          0,
          map,
          start,
          TRUE, /* copy-on-write remapping! */
          &cur_prot,
          &max_prot,
          VM_INHERIT_DEFAULT);
...
}
```

Listing 2. VM_PROT_COPY and COW

So far, we haven't seen any publicly reported bugs breaking the read-only mappings on iOS systems. Furthermore those known issues on other operating systems usually lead userland privilege escalation such as sandbox bypasses, elevating to userland root, etc. By breaking read-only userland memories, achieving kernel mode code execution is still hard, given the fact that kernel still treat userland shared memory less trustable.

3 iOS DMA features

With stronger and more abundant features provided by peripheral components attached to the mobile device, DMA(direct-memory-access) technology enables the ability for fast data transfer between the host and peripheral devices. DMA transfer avoids the involvement of CPU MMU, so that the memory protection attributes set on page tables are ignored. In theory, DMA transfer can ignore all access permission bits of a specific memory page, no matter whether the memory is mapped in kernelland, or the userland. In reality, this is not the case. IOMMU(inputoutput memory management unit)[5] is introduced to connect a direct-memory-accesscapable (DMA-capable) I/O bus to the main memory. With 64bit cellphones becoming more and more popular, there is need for IOMMUs to be compatible with 32bit peripheral devices on the SoC. On those devices, IOMMU is responsible to map 64bit physical addresses into 32bit device addresses. For 64bit iOS devices, DART(Device Address Resolution Table) is responsible to perform the address translation. In the design of IOMMU, security is taken into consideration.

3.1 Host-to-device DMA and device-to-host DMA

For 64bit iOS devices, peripheral devices such as Broadcom WIFI module, implement DMA transfers[6]. In the middle 2017, Gal Beniamini of Google Project Zero team leverages DMA features on iOS to achieve firmware to host attack.

In his research, he found a buffer in the AP kernel which is also mapped into the IO space, but the map in the firmware is writeable where it shouldn't be. However, the WIFI driver in the AP kernel still trust that buffer because the AP assumes the WIFI firmware is trustable. As a result, the AP kernel doesn't perform necessary boundary check on that buffer. The malicious WIFI firmware can craft this buffer and perform a DMA transfer into the host to cause a OOB write within iOS kernel.

Gal's research is very creative because he proposed new device-to-host attack model using the DMA feature. However, such attack scenario has to been limited in a short distance case because the hacker needs to compromise the iOS WIFI stack first by transferring low level WIFI packets.

Is it possible to make it longer distance? For example, if we could leverage DMA feature in iOS userland and achieve privilege escalation to kernel, then by combining with a browser exploit, long distance remote jailbreak can be done.

However, DMA features are kind of low level implementation which is mostly performed at kernel level. Such feature is never exposed directly to the userland.

3.2 Indirect userland DMA

Although DMA is not exposed to userland directly, there might be indirect interface. For example, Apple uses hardware JPEG engine to accelerate the encoding and decoding process, also IOSurface transform is done by hardware device whose kernel driver is called AppleM2ScalerCSC(Apple M2 Scaler and Color Space Converter Driver). With the help of those hardware, graphics processing becomes super fast.

Back in 2016, Italian researcher Luca Todesco released the proof-of-concept code for code sign bypass of iOS 9.3.5[7]. From his PoC, code signing is bypassed by remapping the userland code page into read/write, followed by a DMA transfer to patch the code, and remapping the page back to read/execute. It is obvious that DMA transfer behaves differently with calling memcpy. The bug was fixed by Apple after iOS 10 release.

The case indicates that performing DMA transfer in userland and its behavior is not well studied, thus caught our eye to research deeply.

IOSurface IOSurface object represents a userland buffer which is shared with the kernel. The userland applications can keep a reference handle of self-created IOSurface and use it as other driver's parameter. The kernelland drivers use IOSurface object to access the userland buffer, performing actions to process the buffer.

We can create IOSurface by calling userland API IOSurfaceCreate, supplying a CFDictionary object as parameter to specify the properties of the IOSurface. We can either specify the size of the buffer we want to create, and the kernel driver will create a memory descriptor and map the userland memory. Another option is to specify an existing userland buffer with its size. In the latter case, kernel will create an IOMemoryDescriptor object to associate with the newly created IOSurface object. Here is the code piece:

```
__int64 __fastcall IOSurface::allocate(IOSurface *this)
{
...
  if ( this->m_IOSurfaceAddress )
  {
    v24 = this->m_IOSurfaceAddress;
    v23 = (unsigned int)this->m_IOSurfaceAllocSize;
    v22 = 9;
    v2 = get_task_map_stub(this->m_task);
    if ( (unsigned int)mach_vm_region_stub(v2,
        &v24, &v23, 9LL, &v21, &v22, 0LL) || v21 & 4 )
      return 0xE00002C8LL;
    v3 = IOMemoryDescriptor::withAddressRange(
           v1->m_IOSurfaceAddressAligned,
           (unsigned int)v1->m_IOSurfaceMappedSize,
           (v21 >> 1) & 1 | 0x110002,
           v1->m_task);
    v4 = (__int64 *)&v1->m_IOMemoryDescriptor;
    v1->m_IOMemoryDescriptor = v3;
    v1->b_allocated = 1;
    }
...
}
```

Listing 3. IOSurface creation with existing userland buffer

In the above code logic, it obtains the permission property of the user-land buffer, and use it to specify ”options” parameter(third parameter) in IOMemoryDescriptor::withAddressRange. In read-only buffer case, the ”options” parameter has kIODirectionOut bit set, while read/write buffer will have kIODirectionOut and kIODirectionIn both set.

IOSurfaceAccelerator IOSurfaceAccelerator is a userland framework on iOS platform only(Not exist on macOS). By reversing its code, we found the two most important interface: IOSurfaceAcceleratorCreate and IOSurfaceAcceleratorTransferSurface. IOSurfaceAcceleratorCreate is responsible for creating a IOKit userclient connection representing an IOSurfaceAcceleratorClient object in the kernel. IOSurfaceAcceleratorTransferSurface takes two IOSurface handles, one for source and another for destination, along with a dictionary supplying the transfering parameters. The kernel will hand the request over to the hardware for fast IOSurface transfer with specified parameter. For example, in a screen snapshot scenario[8], we can obtain the IOSurface handle of the system framebuffer, and transfer it to user owned IOSurface. We can apply transfer parameters such as color filtering, border filling, etc. Kernel will offload all those complex computation to the hardware device.

Is it possible to abuse this amazing userland indirect DMA feature? Let's look into the kernel driver's logic to better understand the internal details.

3.3 DMA in AppleM2ScalerCSC driver

IOSurfaceAcceleratorClient object is created and maintained in AppleM2ScalerCSC, its overall architecture is illustrated by the following graph:

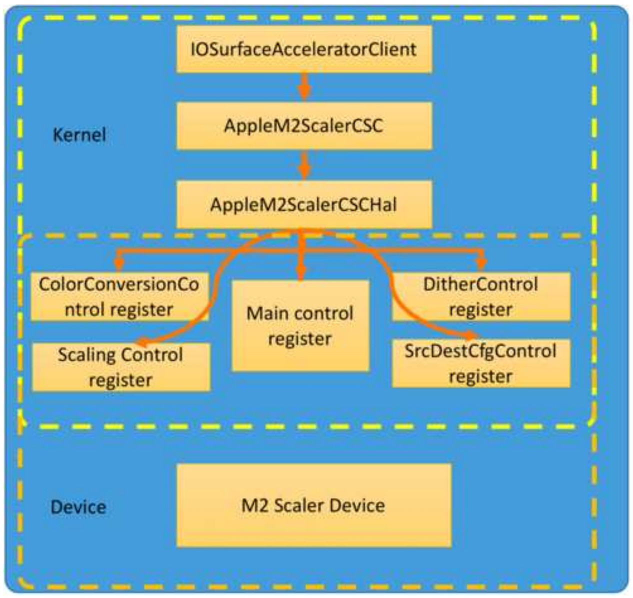


Fig. 1. AppleM2ScalerCSC architecture

IOSurfaceAcceleratorClient talks to AppleM2ScalerCSCDriver which is the upper level driver of AppleM2ScalerCSC. During AppleM2ScalerCSCDriver initializing procedure, low level driver object AppleM2ScalerCSCHal is created, responsible for handling device dependent stuff and provide device independent interface to AppleM2ScalerCSCDriver. AppleM2ScalerCSCHal will create kernel objects such as ColorConversionControl object, SrcDestCfgControl object, ScalingControl object, DitherControl, ColorManager object, also it maps a device memory into kernel virtual space, representing the device's key registers. The device memory is wrapped by those 4 objects, each representing a specific feature category.

When userland application calls IOSurfaceAcceleratorTransferSurface, IOSurfaceAcceleratorClient method 1 will be reached. Here is the handling logic in the function IOSurfaceAcceleratorClient::user_transform_surface:

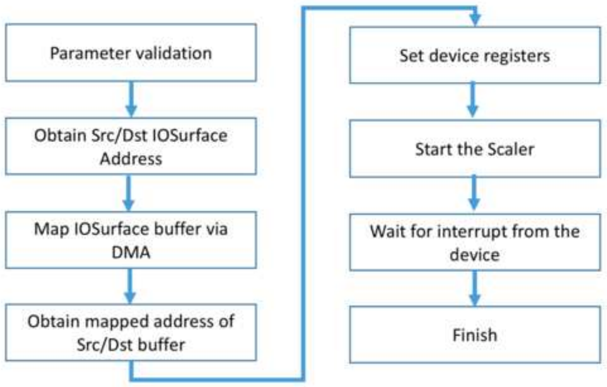

Parameter validation
Obtain Src/Dst IOSurface Address
Map IOSurface buffer via DMA
Obtain mapped address of Src/Dst buffer
Set device registers
Start the Scaler
Wait for interrupt from the device
Finish

Fig. 2. user_transform_surface handling logic

During the handling process, we are interested in the DMA mapping part. This is done by function AppleM2ScalerCSCDriver::setPreparedMemoryDescriptor, here is the code piece:

```
AppleM2ScalerCSCDriver::setPreparedMemoryDescriptor
(AppleM2ScalerCSCDriver *this, QWORD a2, void *a3,
IOMemoryDescriptor *descriptor)
{
          ...
      v8 = IODMACommand::withSpecification(
                          (__int64)OutputLittle32,
                          0x20LL,
                          0LL,
                          0LL,
                          0LL,
                          1LL,
                          (__int64)v6->m_IOMapper,
                          0LL);
      ...
      v12 = IODMACommand::setMemoryDescriptor(
               v8,
               descriptor,
               1LL);
```

```
    ...
}
```

Listing 4. map userland buffer to device

Here descriptor is directly obtained from the IOSurface, which represents the userland buffer. m_IOMapper is an instance of IODARTMapper which is independent with other devices, we could find it from iPhone7's ioreg output:

```
+-o dart-scaler@7908000  <class AppleARMIODevice>
 +-o AppleS5L8960XDART  <class AppleS5L8960XDART>
   +-o mapper-scaler@0  <class IODARTMapperNub>
     +-o IODARTMapper  <class IODARTMapper>
```

Listing 5. IODARTMapper

After the mapping, the next step is to obtain the device memory visible to IOMMU:

```
AppleM2ScalerCSCDriver::mapDescriptorMemory(
AppleM2ScalerCSCDriver *this, __int64 a2, void *a3,
IOMemoryDescriptor *a4, unsigned __int64 a5,
unsigned __int64 a6)
{
...
  if ( v8->m_IOMapper )
  {
    v13 = IODMACommand::genIOVMSegments(
                (IODMACommand *)v12,
                (unsigned __int64 *)&v32,
                &v33,
                (unsigned int *)&v31);
      v20 = IOSurface::getPlaneOffset(v10->pBuffer, 0);
      v10->PA0 = v20 + v33;
    }
  }
...
}
```

Listing 6. map userland buffer to device

The obtained 32bit address will then be set into the SrcDestCfgControl object, and synchronized with the specific field in device virtual memory:

```
M2ScalerSrcDestCfgControlMSR::setAddresses
(M2ScalerSrcDestCfgControlMSR *result,
int destOrSrc, int PA)
{
  __int64 v4; // x8@1

  v4 = result->deviceAddrVirtual;
  if ( destOrSrc )
  {
    *(_DWORD *)(v4 + 516) = PA;
  }
  else
  {
    *(_DWORD *)(v4 + 260) = PA;
  }
  return result;
}
```

Listing 7. set device register

Other registers relating to transform parameters are handling in a similar manner. After all registers are set, the scaler is started by setting the control register:

```
AppleM2ScalerCSCHalMSR::startScaler
(AppleM2ScalerCSCHalMSR6 *this, Request *a2)
{
...
  v2 = (AppleM2ScalerCSCHal *)this;
  *(_DWORD *)&this->gap8[4] = 0;
  v3 = *(_QWORD *)&this->gap64[4];

  if ( *(_DWORD *)(v3 + 72) )
          v5 = 0;
  else
          v5 = -2;
  *(_DWORD *)(this->deviceAddrVirtual + 152) = v5;
  *(_DWORD *)(this->deviceAddrVirtual + 128) = 1;
...
}
```

Listing 8. start the scaler

Finally the code starts waiting on an event. When scaler finished the processing, an interrupt will be issued and the event will be signaled. That indicates the source IOSurface has been transferred to the destination IOSurface. In the case we don't specify any transform options in the parameter, scaler simply performed a memory copy via DMA transfer.

At this stage, we understand the internals of scaler DMA feature that is indirectly exposed to the userland. In the next section we explore the IOMMU memory protection feature on iOS.

4 The implementation of IOMMU memory protection

Unlike CPU MMU, page table specification of IOMMU is not well standardized. As we mentioned in the previous section, DART on iOS has taken security into consideration, so the IOMMU on iPhone devices should support memory protect protection. Gal Beniamini has reversed the logic in DART module and explained the page table specification on 64bit iOS devices. The following graph is from his blog showing the structure of IOMMU page table:

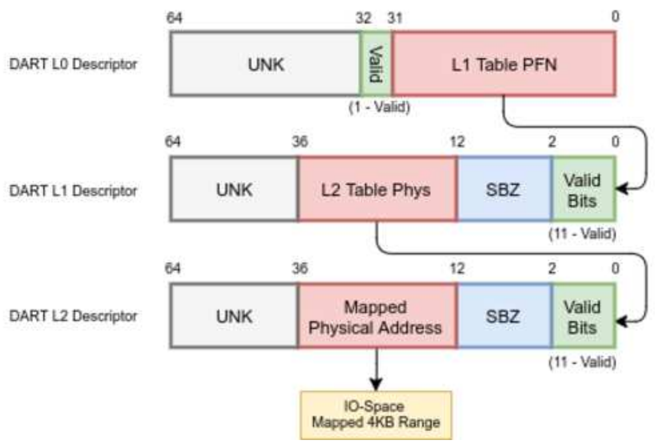


Fig. 3. dart desc by Gal[6]

However in his blog he didn't mention whether the IOMMU supports memory protection or not. To find it out, we reversed IODARTFamily and AppleS5L8960XDART module of the iOS 11.

The entry point for mapping memory in I/O space is IODARTMapper::iovmMapMemory, where the fifth parameter is mapOptions:

```
IODARTMapper::iovmMapMemory(IODARTMapper *this,
 IOMemoryDescriptor *a2, QWORD a3, QWORD length,
 unsigned int mapOptions, const IODMAMapSpecification *a6,
 IODMACommand *a7, const IODMAMapPageList *a8,
  unsigned __int64 *a9, unsigned __int64 *a10)
{
...
  direction = g_array[mapOptions - 1]; //g_array[] =[2,1,3]
  IODARTMapper::_iovmAlloc(
          v14,
          (unsigned int)(*(_DWORD *)&v14->gapF8[28] * v20),
          0LL,
        &v28,
        direction,
        a6) )
...
}
```

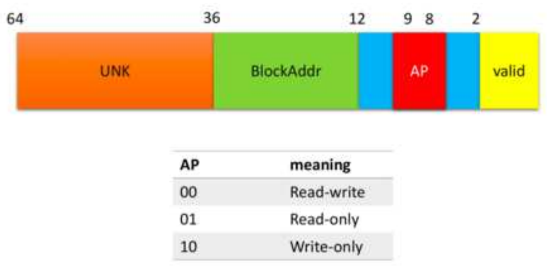
Listing 9. IODARTMapper::iovmMapMemory logic

The last 3 bits in mapOptions is translated to direction value. Read-only mapping has direction value 2, write-only mapping has value 1, and read/write mapping has the value 3. The direction variable flows and finally reached the function in AppleS5L8960XDART, the low level implementation of DART:

```
AppleS5L8960XDART::setTranslation(AppleS5L8960XDART *a1,
 __int64 a2, int a3, __int64 a4, int a5,
 int blockAddr, __int64 a7, unsigned int direction)
{
  v8 = a7;
  v11 = a1;
  v12 = blockAddr & 0xFFFFFF;
  if (direction == 2) //read-only mapping
  {
   APbits = this->apSuportMask & 0x80;
  }
  else if (direction == 3) //read/write mapping
  {
   APbits = 0;
  }
  else if (direction == 1) //write-only mapping
  {
        dartBits = (g_dartVersion << 6) & 0x100;
        APbits = this->apSuportMask & dartBits;
  }
  attrBits = ((v12 & 0xFFFFFF) << 12)|APbits|2);
  AppleS5L8960XDART::setL3TTE(
    (__int64)v11,
    v10,
    v9,
    v8 & 0xFFFFFFF000000FFFLL | attrBits;
}
```

Listing 10. AP in TTE

The code above clearly shows the AP bits logic. On a device later than iPhone 5s, the apSupport mask is always 0x180, indicating the 8th and 9th bit in the TTE are AP related bits. We can get the following TTE structure relating to AP bits:



AP	meaning
00	Read-write
01	Read-only
10	Write-only

Fig. 4. TTE structure

Interestingly, the support for write-only mapping on IOMMU depends on the DART version. On iPhone 7 devices, g_dartVersion is valued 4 so that write-only option is supported. It is possible that on iPhones of legacy models whose DART version is below 4 will map write-only as read/write.

Based on the research above, it is obvious that DART on all 64bit iPhone devices supports read-only mappings. That is not good news for hackers as it seems impossible to utilize DMA to transfer userland read-only mappings because of the AP-aware IOMMU on iPhone.

5 GPU notification mechanism

On iPhone7 device, Apple Graphics provides with 128 channels for concurrent processing. Those channels can be divided into three categories: TA channel, 3D channel, and CL channel. Userland applications utilize frameworks such as OpenGL, which wrap drawing instructions along with other information and send to the kernel for handling. Kernel parses the instructions with other information, put them into the GPU's mapped device memory of specific channel, and wait for GPU to finish the execution. Because GPU can handle multiple instructions concurrently, a well-designed notification mechanism is necessary for GPU to synchronize the status of instruction processing. The overall architecture for the notification is shown below:

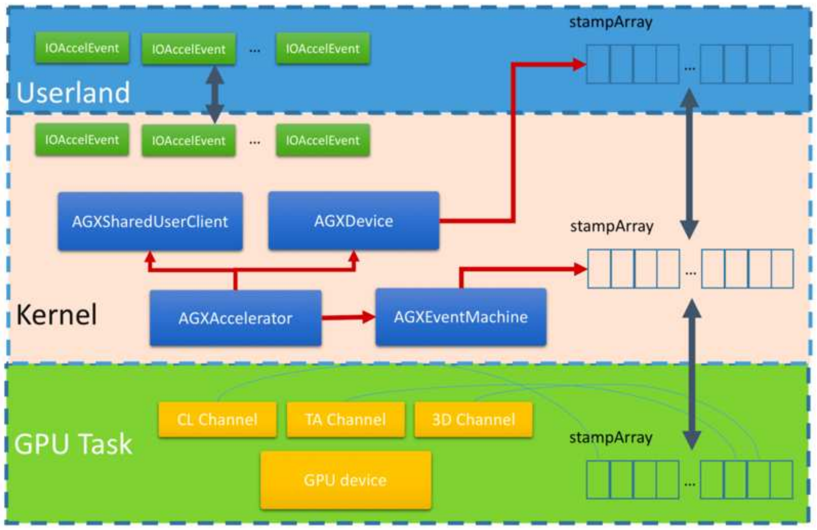


Fig. 5. GPU notify

From the above graph, we found not only the kernel, but userland application also seeks for a better way of notifying the status of instruction handling from GPU. As a result, stampArray and IOAccelEvent objects are introduced.

When kernel starts the GPU, a device memory indicating the stamp status array is mapped to kernel virtual space by AGXAccelerator object. The memory is a representation of uint32 array of 128 elements, each showing the last complete instruction's stamp of that channel. AGXAccelerator then creates AGXEventMachine object which constructs a kernel stamp address array of 128 elements, each of which is the address of stamp in that mapped buffer of that channel. Here is the logic:

```
IOAccelEventMachine2::setStampBaseAddress
(IOAccelEventMachineFast2 *this, unsigned int *a2)
{

   result = (void **)IOMalloc_stub(8 * v3);
   v2->m_stampAddressArray = result;
   if ( v2->m_numOfStamps_InitTo0x80 < 1 )
     return result;
  v5 = 0LL;
  do
  {
    v6 = v2->m_stampBaseAddress;//mapped buffer
    v2->m_stampAddressArray[v5++] = v6 + 4 * v5;
  }
  while ( v5 < v2->m_numOfStamps);
}
```

Listing 11. stamp address array initialization

The stamp value of each channel is incremental upon each of the instruction processing completeness. Before CPU process each drawing instruction, the kernel sets the content in an IOAccelEvent object, each of which contains 8 sub events. Each sub event structure is 8 bytes in size, where the lower 4 bytes represents the channel index ready to put into, and the higher 4 bytes is the expected stamp value when this instruction execution is finished. After GPU finished handling, the stamp value in its stamp array will be updated into the expected stamp value set on the IOAccelEvent, thus the kernel can wait for the event in an easy manner:

```
IOAccelEventMachine2::waitForStamp
(IOAccelEventMachineFast2 *this, __int64 a2)
{
  v8 = (unsigned int *)this->m_stampAddressArray[a2];
  do
  {
    v15 = *v8;
  }
  while ( (signed int)(expectedStamp - v15) > 0 );
}
```

Listing 12. waitForStamp logic

The above PoC code illustrates how kernel wait for a drawing instruction to complete. In the real case, code is more complex. For example, AGXEventMachine maintains a copy of the latest stamp value in each channel, when an expected stamp value matches kernel will also update that structure. Also kernel uses a better way other when ”while loop” to wait for the update of the device buffer.

Last but not the least, the stamp array is also mapped as read-only to the userland, some of the IOAccelEvent objects will be mapped to userland read-only as well. This allows the userland applications perform the check for status of each drawing instruction, without communicating to the kernel.

6 The vulnerabilities

Now that the DMA feature exposed to iOS userland, the IOMMU memory protection, and the GPU notification mechanism are all well explained, it looks like everything operates flawless at this stage. In this section, we will cover two bug details: one in DMA mapping with host virtual memory, and another out-of-bound write issue caused by potentially untrusted userland read-only memory. With two bug combined together, the whole well designed architecture becomes vulnerable.

6.1 The DMA mapping vulnerability

In the early section, we show the code logic of performing a read-only DMA mapping, however on all versions of iOS 10 and early beta version of iOS 11, DART ignores the permission bits that is passed from the upper XNU layer:

```
IODARTMapper::_iovmAlloc(IODARTMapper *this,
unsigned int a2, __int64 a3,
_DWORD *a4, int mapOptions)
{
...
    IODARTMapper::_iovmInsert(this, v17, v6+v18,
     *(unsigned int *)&v7->gapF8[56]) );
...
}
```

Listing 13. calling _iovmInsert without mapOptions

The mapOptions parameter is not used in IODARTMapper::_iovmInsert. Later, the code reaches DART's low level implementation: AppleS5L8960XDART.

```
AppleS5L8960XDART::setTranslation(AppleS5L8960XDART *this,
unsigned int a2, unsigned int a3, int a4,
 __int64 a5)
{
  tte = a5 & 0xFFFFFFF000000FFFLL |
        ((*(_QWORD *)&a4 & 0xFFFFFFLL) << 12) | 2;
  return AppleS5L8960XDART::setL3TTE(this, a2, a3, tte);
}
```

Listing 14. AppleS5L8960XDART::setTranslation in iOS 10

It is obvious that the implementation of DART in iOS 10 set the 8th and 9th bit of the TTE to 0, indicating a read/write mapping.

Because of this buggy implementation, we are able to perform DMA transfer in userland with the help of AppleM2Scaler driver, reachable from container sandbox, overwriting arbitrary userland read-only mappings in the specific process.

6.2 The out-of-bound write vulnerability

At the first glance, the above DMA mapping bug can only achieve userland sandbox bypass or privilege escalation to a higher account's context. However by exploring rich functionality in Apple Graphics stack, it is totally different case. IOAccelResource is similar in functionality as IOSurface object, except that IOAccelResource represents a shared userland buffer which would be mapped into GPU task. Like IOSurface, we can create IOAccelResource by specifying an existing userland buffer, by specifying the size, or even by providing an existing IOSurface handle. Method 0 of IOAccelSharedUserClient is for IOAccelResource creation. As part of IOAccelResource initialization process, a shared mapping will be created:

```
IOAccelResource2::init(IOAccelResource2 *this,
 IOGraphicsAccelerator2 *a2, IOAccelShared2 *a3,
 char a4)
{
...
        IOAccelSharedNamespace2::mapClientSharedForId(
         this->m_IOAccelSharedNamespace2,
         HIDWORD(v22),
         &this->m_IOAccelClientSharedRO,
         &this->m_IOAccelClientSharedRW) )
...
}
```

Listing 15. IOAccelResource2 initialization

In IOAccelSharedNamespace2::mapClientSharedForId, a shared memory is created, resource->m_IOAccelClientSharedRO field is assigned to the kernel address of the shared memory with offset appended.

```
IOAccelClientSharedMachine::mapClientSharedForId
(IOAccelClientSharedMachine *this, unsigned int a2,
void **a3)
{
  for ( i = &this->headStructPointer; ; v6 = i )
  {
    i = i->nextStruct;
    if ( i )
      goto LABEL_13;
    i = IOMalloc_stub(0x38LL);
    v15=IOGraphicsAccelerator2::
    createBufferMemoryDescriptorInTaskWithOptions(
            v5->m_IOGraphicsAccelerator2,
            0LL,
            0x10023u,
            calcSize,
            *v9);
    i->m_IOBufferMemoryDescriptor = v15;
    v20 = IOMemoryDescriptor::createMappingInTask(
           (IOMemoryDescriptor *)v15,
           (task *)v5->m_task,
           0LL,
           v5->mapOption | 1u,
           0LL,
          0LL);
    i->m_IOMemoryMapUser = v20;
    i->m_addressUser =IOMemoryMap::getVirtualAddress(v20);
    v21 =  IOMemoryDescriptor::map(
              i->m_IOBufferMemoryDescriptor,
              1u);
    i->m_IOMemoryMapKernel = v21;
    i->m_addressKernel=IOMemoryMap::getVirtualAddress(v21);
    i->nextStruct = 0LL;
    i->field_30 = v5->some_size2;
    *v3 = (i->m_addressKernel + v5->some_size * v4);
    return 1LL;
    ...
}
```

Listing 16. Map the memory in userland

The userland mapping, however, is read-only. At the end of IOAccelResource creation process, the userland address of IOAccelClientSharedRO structure will be returned to userland application.

```
IOAccelClientSharedMachine::getClientSharedAddressForId
(IOAccelClientSharedMachine *result, unsigned int a2,
unsigned __int64 *a3)
{
  clientSharedMachine_struct0x38 *v3; // x8@1
  unsigned __int64 v4; // x10@2
  v3 = result->headStructPointer;
  *a3 = v3->m_addressUser + result->some_size * a2;
}
```

Listing 17. IOAccelClientSharedRO address returned to userland

IOAccelClientSharedRO contains an 4-element IOAccelEvent array, along with its resource ID and resource type.

```
struct IOAccelClientSharedRO
{
  IOAccelEvent m_arrAccelEvent[4];
  int m_resId;
  int m_type;
};
```

Listing 18. IOAccelClientSharedRO definition

Userland application can delete the IOAccelResource by calling method 1 of IOAccelSharedUserClient, with resource ID provided. In the process of resource deletion logic, with specific options and resource type(we can specify during creation) IOAccelEventMachineFast2::testEvent will be called to check if GPU has finished processing specific drawing instruction.

```
IOAccelSharedUserClient2::delete_resource
(IOAccelSharedUserClient2 *this, unsigned int a2)
{
...
  if ( IOAccelNamespace::lookupId(
          v3->m_IOAccelShared2->m_SharedNamespace2Resource,
          a2,
          (void **)&v9) & 1 )
  {
    v6 = v9;
    if ( HIBYTE(v9->someOptions) & 1 &&
    (unsigned __int8)*(_WORD *)&v9->m_type != 0x82 )
    {
      if ( IOAccelEventMachineFast2::testEvent(
     v3->m_IOGraphicsAccelerator2->m_EventMachineFast2,
     (IOAccelEvent *)v9->m_IOAccelClientSharedRO +
     (*(unsigned __int16 *)&v9->m_type >> 8)) )
      {
        v4 = 0LL;
      }
...
}
```

Listing 19. calling IOAccelEventMachineFast2::testEvent

Normally only kernel can change the content of m_IOAccelClientSharedRO, and the mapping in userland is read-only. Because of that, kernel fully trusts m_IOAccelClientSharedRO, eliminating the boundary check:

```
IOAccelEventMachineFast2::testEventUnlocked
(IOAccelEventMachineFast2 *this, IOAccelEvent *a2)
{
  while ( 1 )
  {
    v3 = *((_QWORD *)&a2->m_channelIndex + v2);
    if ( (_DWORD)v3 != -1 )
    {
      highDword = (v3 >> 32) & 0xFFFFFFFF;
      lowDword = (signed int)v3;
      v6 = ((char *)this + 24 * v3);
      v8=v6->m_inlineArrayA0x18Struct[0].lastSyncedStamp;
      v7=&v6->m_inlineArrayA0x18Struct[0].lastSyncedStamp;
      if ( (signed int)highDword - v8 >= 1 )
      {
        v9 = *this->m_stampAddressArray[lowDword];
        *v7 = v9;
        if ( (signed int)highDword - v9 > 0 )
```

```
          break;
        }
      }
      if ( ++v2 >= 8 )
        return 1LL;
    }
    return 0LL;
}
```

Listing 20. no boundary check in IOAccelEventMachineFast2::testEventUnlocked

With the help of the DMA bug, it is feasible to perform DMA transfer in userland, getting m_channelIndex and the expected stamp value modified. Since on iPhone 7 there are only 128 channels, both m_inlineArrayA0x18Struct and m_stampAddressArray are arrays of 128 elements. With m_channelIndex changed to arbitrary value, we caused out-of-boundary read on m_stampAddressArray, and out-of-boundary write on m_inlineArrayA0x18Struct.

7 Exploitation

Exploitability of those two bugs depend on whether we can control the content for for both m_inlineArrayA0x18Struct and m_stampAddressArray with the same out-of-bound index, after which we can achieve arbitrary memory read and limited memory write. This looks a little bit challenging because of the two facts:
- Both arrays are created in very early stage of iOS boot process, it is hard to make proper memory layout around the address of them by our userland applications.
- Size of element of each array is different. The larger index we specify, the longer span in address of the referenced array element.
In this section, we will describe the approach we use to solve the above challenges, leaking kernel module address to bypass KASLR, and execute arbitrary code right after.

7.1 Craft memory layout

Kernel heap memory starts at a relatively low address. Heap grows linearly with more and more memory allocated. The start address of heap differs within tens of megabytes upon each boot. Since both m_inlineArrayA0x18Struct and m_stampAddressArray are allocated at early stage of boot, their addresses are closed with each other. Although it is relatively impossible to control the content around the two arrays, we can spray the kernel memory from userland applications. A widely used technique for kernel memory spraying is to send OOL message to a port created by our own task without receiving it[9], in which case kernel will create a vm_map_copy_t structure whose size and data (except for the structure header part) are both controlled by the user.

Based on our test, user application can spray around 350MB OOL data on an iPhone 7 device. By specifying a large channelIndex value, it is possible to make the out-of-bound elements of both m_inlineArrayA0x18Struct and m_stampAddressArray arrays fallen into our sprayed vm_map_copy_t structures. In real case, there around 50MB of heap memory allocated after m_inlineArrayA0x18Struct and m_stampAddressArray arrays' creation, so we need to choose a index which meets the condition below:

$$index * 24 < 350MB + 50MB \tag{1}$$

And:

$$index * 8 > 50MB \tag{2}$$

The index then should be in range of [0x640000, 0x10AAAAA] to meet both conditions above.

The next problem we need to resolve is the offset at which we write or read the address within a page. Thanks to mechanism of XNU zone allocator, the base address of m_inlineArrayA0x18Struct array is always at offset 0xF0 or 0x20F0 of a 0x4000 page. This is because the size of AGXEventMachine is 0x1D18, allocated in kalloc.8192 zone, and m_inlineArrayA0x18Struct is an inlined array with AGXEventMachine structure at offset 0xF0.

Similarly, m_stampAddressArray is allocated with 0x200, falling in kalloc.512 zone. The address offset within a 0x4000 page can be all values dividable by 0x200.

For m_stampAddressArray, it is easy to set index to reach arbitrary 8-byte aligned offset with a page because the element in that array is 8 byte in size (the channel address in the device virtual memory). For m_inlineArrayA0x18Struct, since element size is 24 bytes, it is a little bit tricky to out-of-bound access to arbitrary offset.

However with congruence theory, it can be easily achieved. Apparently the least common multiple number of 24 and 0x4000 is 0xc000, so:

$$0xc000 \equiv 0(mod0x4000) \tag{3}$$

So with arbitray integer n:

$$n * 0x800 * 24 \equiv 0(mod0x4000) \tag{4}$$

Also becase:

$$0x4000 \equiv 16(mod24) \tag{5}$$

With 0x4000 / 24 * 0xF0 / 16 = 0x27f6, we can get:

$$0xF0 + 0x27f6 * 24 + n * 0x800 * 24 \equiv 0(mod0x4000) \tag{6}$$

Thus, with arbitrary integer n, we can out-of-bound write to the first 8 bytes in a sprayed page given:

$$index = 0x27f6 + n * 0x800 \quad (7)$$

To reach arbitrary offset aligned by 8 bytes, we just ensure:

$$index = 0x27f6 + 0x2ab * m/8 + n * 0x800 \quad (8)$$

Where m is the offset in the page where you want to write the value.

Here we choose index value as 0x9e6185, which is within the range of [0x640000, 0x10AAAAA]. Also we choose to spray the size of 0x2000-0x18(0x18 is the size of vm_map_copy_t header), and use a guessed address value of 0xffffffe00a5d4030 which is within the range of our sprayed data. Eventually we reach the following situation:

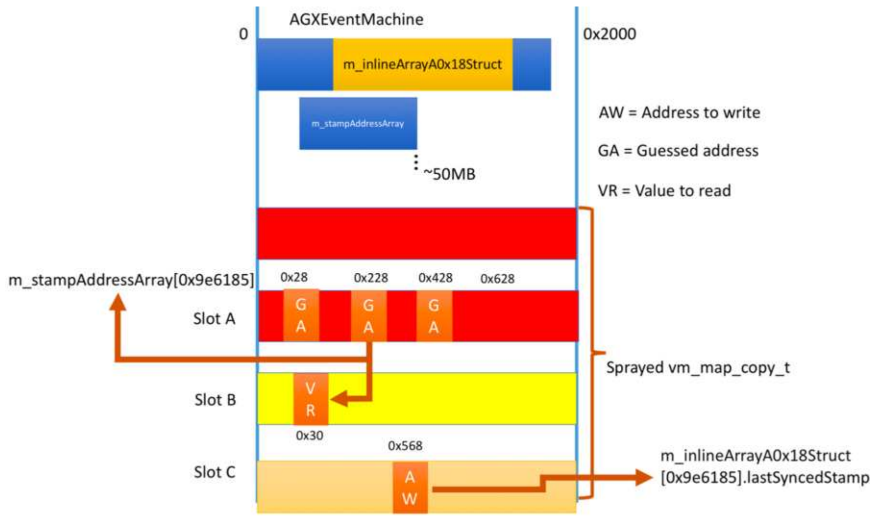


Fig. 6. initial memory layout

To simpify our explaination, we call the referenced page of m_stampAddressArray[0x9e6185] slot A, referenced page of the guessed address slot B, referenced page of m_inlineArrayA0x18Struct[0x9e6185].lastSyncedStamp slot C.

At this stage, we craft the memory layout to ensure both out-of-bound element of m_stampAddressArray[index] and m_inlineArrayA0x18Struct[index] fallen in our controlled data, and able to write AW(address to write) to VR(value to read).

7.2 Bypass KASLR

Because userland application can read the vm_map_copy_t data by receiving the OOL message on the specific port we created, we are able to find out what value we have written on which port. To implement this, we create around 0xA000 port, each associates with one OOL data of 0x2000-0x18. On each OOL data buffer, we specify an index value range from 0 to 0xA000, indicating the identity of that OOL, at its 0x30-0x18 offset.

By receiving the message of each port, we can find out whether the OOL data is allocated at slot C by checking its value on 0x568-0x18 offset. If it is not, we quickly create another port and send another OOL message with guessed address value changed to 0xfffffffe00a5d4000. If we find the correct index allocated on slot C, we also obtain the index of OOL data which is allocated on slot B, whose address is 0xfffffffe00a5d4000.

After that, we think of refill in another object with vtable embed in its offset 0 at slot B. By looking for all object candidate which can be created within container sandbox, we find AGXGLContext object falls into kalloc.8192 kalloc zone, same as the sprayed OOL data. We receive the message around the index(10+-) of slot B to ensure the vm_map_copy_t structure in slot B is freed, and quickly allocate 30 AGXGLContext object to fill in.

After that, the memory layout is changed to the following:

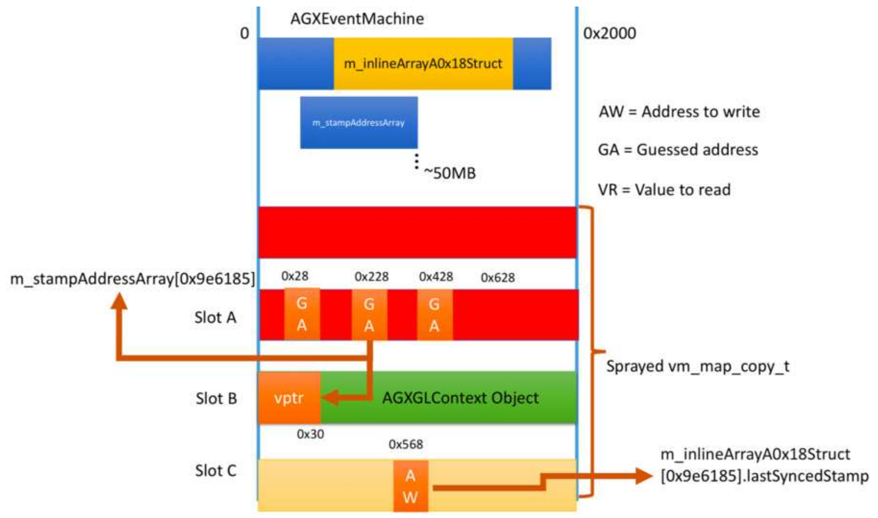


Fig. 7. memory layout to bypass KASLR

By triggering the OOB write bug again, and receive the port with slot C index, we can obtain the lower 32bit of AGXGLContext vtable. Since all iOS kernal extensions and the kernel itself share the same kslide, by leaking AGXGLContext vtable address we fully bypass KASLR.

7.3 Code execution

To get code execution, we need to free the slot C, and fill in an object where its 0x568 offset represents important object. The reason we choose out-of-bound index as 0x9e6185 is because address of m_inlineArrayA0x18Struct[0x9e6185].lastSyncedStamp is at offset 0x568 of slot C, and coincidently the offset 0x568 in AGXGLContext is set to AGXAccelerator object. By calling its method 0, we reached IOAccelGLContext2::context_finish:

```
IOAccelGLContext2::context_finish(__int64 this)
{
...
  v2 = (*(**(_QWORD **)(*(_QWORD *)(this + 0x568)
           + 776LL)
           + 208LL))(
          *(*(_QWORD *)(this + 0x568) + 776LL),
          (IOAccelEvent *)(this + 1416));
...
}
```

Listing 21. IOAccelGLContext2::context_finish

With AGXAccelerator address within AGXGLContext changed to the address pointing our controlled data, we are able to perform code execution and obtain task_for_pid 0.

So the final layout before the third OOB trigger is shown below:

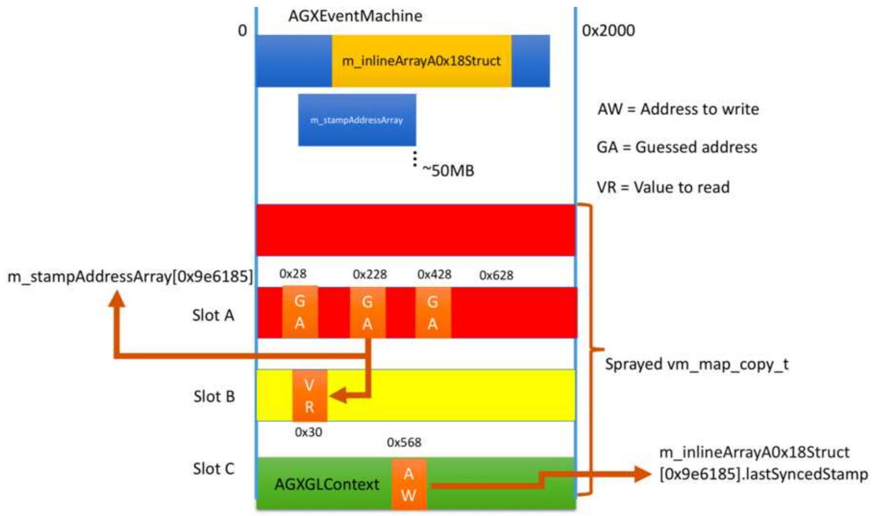


Fig. 8. memory layout for code execution

7.4 post exploitation

After tfp 0 is obtained, it is still far away from a full jailbreak. Post exploitation work includes KPP/AMCC bypass, patching the kernel, installing Cydia, etc. We have no plan to discuss those work in this paper. If we get the submission accepted we will show a demo during the speech.

To wrap up this section, we conclude our exploit strategy as below:

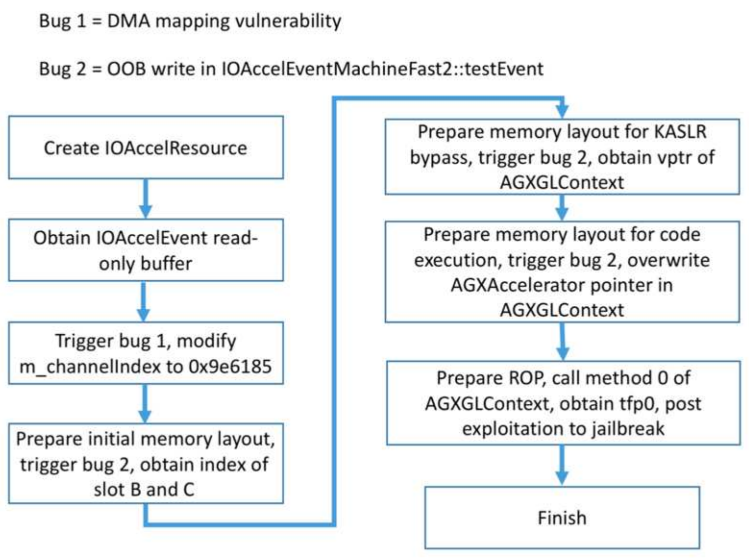

Bug 1 = DMA mapping vulnerability
Bug 2 = OOB write in IOAccelEventMachineFast2::testEvent
Create IOAccelResource
Obtain IOAccelEvent read-only buffer
Trigger bug 1, modify m_channelIndex to 0x9e6185
Prepare initial memory layout, trigger bug 2, obtain index of slot B and C
Prepare memory layout for KASLR bypass, trigger bug 2, obtain vptr of AGXGLContext
Prepare memory layout for code execution, trigger bug 2, overwrite AGXAccelerator pointer in AGXGLContext
Prepare ROP, call method 0 of AGXGLContext, obtain tfp0, post exploitation to jailbreak
Finish

Fig. 9. overall exploit strategy

8 Conclusion

With the first release of iOS 11, Apple fixed the DMA mapping bug by adding implementation of read-only mapping at its DART code. The other OOB write bug, however, remains unfixed at the time we wrote this paper(iOS 11.4).

The DMA mapping bug is a good example of how security can be broken by bad implementation with good hardware design. Along with the DMA mapping bug, we make possible a complex exploit chain to achieve jailbreak within userland applications. On the other hand, it still remains a problem whether the userland read-only memory should be trusted by the kernel. After all, userland read-only memory can be dangerous. It is possible that another bug found in the future which can subvert the read-only attribute of userland memory, thus make the whole exploit chain back again, and we never know.

References

1. Translation tables in ARMv8-A, https://developer.arm.com/products/architecture/a-profile/docs/100940/latest/translation-tables-in-armv8-a
2. iOS page table format, https://opensource.apple.com/source/xnu/xnu-4570.41.2/osfmk/arm64/proc_reg.h.auto.html
3. Read-only SharedMemory descriptors on Android are writable, https://bugs.chromium.org/p/project-zero/issues/detail?id=1449

4. Did the Man With No Name Feel Insecure?, https://googleprojectzero.blogspot.com/2014/10/did-man-with-no-name-feel-insecure.html
5. Inputoutput memory management unit, https://en.wikipedia.org/wiki/Input
6. Over The Air - Vol. 2, Pt. 3: Exploiting The Wi-Fi Stack on Apple Devices, https://googleprojectzero.blogspot.com/2017/10/over-air-vol-2-pt-3-exploiting-wi-fi.html
7. Code sign bypass in iOS 9.3.5 by Luca Todesco, https://github.com/kpwn/935csbypass
8. IOS Private API - Capture screenshot in background , https://stackoverflow.com/questions/16463402/iosurface-ios-private-api-capture-screenshot-in-background
9. From USR to SVC: Dissecting the 'evasi0n' Kernel Exploit, http://blog.azimuthsecurity.com/2013/02/from-usr-to-svc-dissecting-evasi0n.html