科恩实验室
KEEN SECURITY LAB

KeenLab iOS Jailbreak Internals:

Userland Read-Only Memory can be Dangerous

Liang Chen (@chenliang0817)

About me

- Security researcher & team leader at Tencent Keen Security Lab
- Browser vulnerability research (Safari, Chrome, IE etc.)
- Apple vulnerability research (Sandbox, Kernel, etc.)

Agenda

- Operating system memory protection overview
- iOS DMA features
- Implementation of iOS IOMMU protection
- iOS GPU notification mechanism
- The vulnerabilities
- Exploitation & Demo
- Conclusion

PART I: Operating system memory protection overview

Memory protection

- Modern OS implements memory protection at hardware level
    - Mitigate known attacks, makes exploitation harder

- Different levels
    - Translation Block Entry properties at MMU (Memory Management Unit) level
        - NX, PXN, AP, etc.
    - KPP, AMCC, etc.

Userland read-only memory mappings

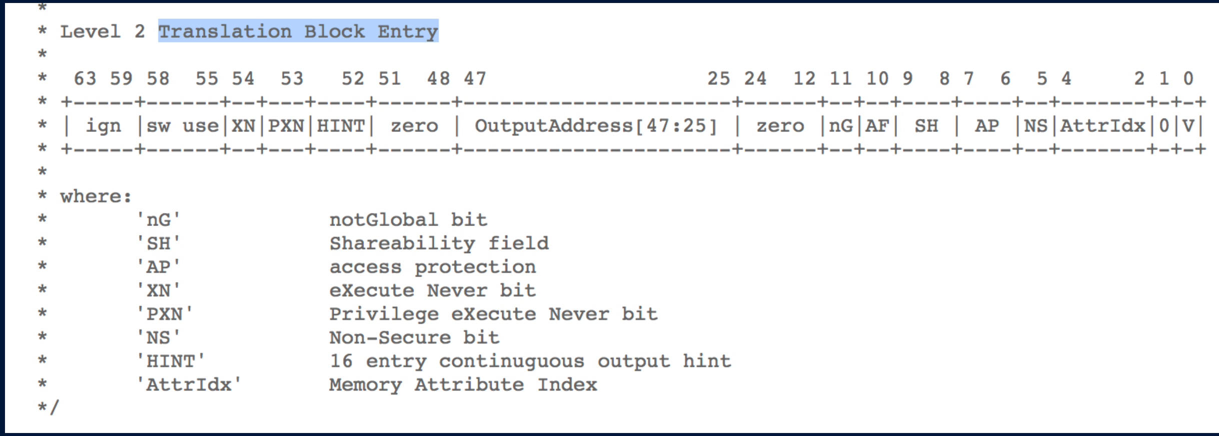
- Old-school approach to protect userland memory
  - Easy but effective
- Scenarios on iOS
  - Executable memory is read-only, preventing from overwriting the code at early stage of exploitation
  - Sharing memory between processes
  - Sharing memory between kernel and process

Userland memory sharing in iOS

- Make inter-process or user-to-kernel communication more efficient

- Low privileged process only needs a read-only mapping of the same physical memory
  - So that server process/kernel can make sure the content of the memory is trustable
  - Eliminating specific security consideration (e.g boundary check, TOCTTOU issues, etc)

- Implementation
  - Access protection bits at Translation Block Entry
  - System MMU can identify those bits



```
 *
 * Level 2 Translation Block Entry
 *
 *  63 59 58  55 54  53   52 51  48 47                  25 24   12 11 10 9  8 7  6  5 4     2 1 0
 * +-----+------+--+---+----+------+----------------------+------+--+--+----+----+--+-------+-+-+
 * | ign |sw use|XN|PXN|HINT| zero | OutputAddress[47:25] | zero |nG|AF| SH | AP |NS|AttrIdx|0|V|
 * +-----+------+--+---+----+------+----------------------+------+--+--+----+----+--+-------+-+-+
 *
 * where:
 *       'nG'           notGlobal bit
 *       'SH'           Shareability field
 *       'AP'           access protection
 *       'XN'           eXecute Never bit
 *       'PXN'          Privilege eXecute Never bit
 *       'NS'           Non-Secure bit
 *       'HINT'         16 entry continuguous output hint
 *       'AttrIdx'      Memory Attribute Index
 */
```

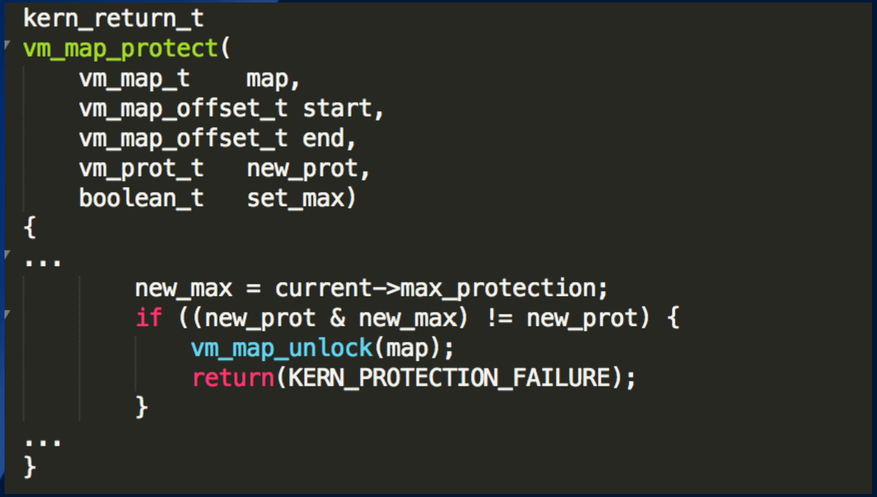
Breaking the trust boundary

- By remapping read-only memory as writable
  - But without marking them COW(copy-on-write)

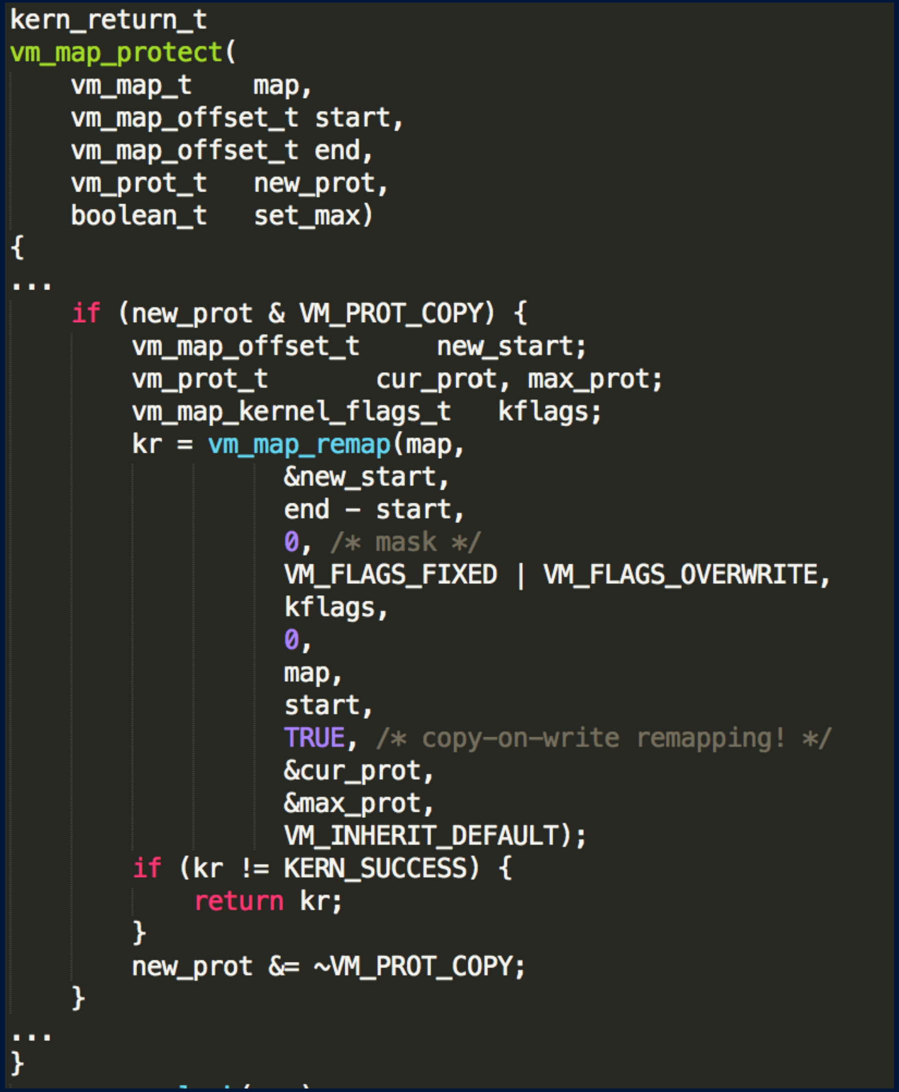
- Can lead to privilege escalation
  - Sandbox bypass in process-to-process memory sharing scenario
  - Or even kernel code execution in process-to-kernel memory sharing scenario

Breaking the trust boundary

- How iOS prevent writable remapping on read-only memory?
  - By preventing users from setting new_prot higher than max_prot in vm_map_protect
  - The only exception is to mark the remapping as 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)
{
...
        new_max = current->max_protection;
        if ((new_prot & new_max) != new_prot) {
            vm_map_unlock(map);
            return(KERN_PROTECTION_FAILURE);
        }
...
}
```



```
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;
        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);
        if (kr != KERN_SUCCESS) {
            return kr;
        }
        new_prot &= ~VM_PROT_COPY;
    }
...
}
```

Breaking the trust boundary

- Historical issues
  - Read-only SharedMemory descriptors on Android are writable: https://bugs.chromium.org/p/project-zero/issues/detail?id=1449
  - Did the Man With No Name Feel Insecure?: https://googleprojectzero.blogspot.com/2014/10/did-man-with-no-name-feel-insecure.html
- On iOS, there is no such known issues
  - Seems everything is fine up till now

PART II: iOS DMA features

DMA overview

- DMA(direct-memory-access) technology enables the ability for fast data transfer between the host and peripheral devices
  - With stronger and more abundant features provided by modern peripheral devices attached to the mobile devices

- DMA transfer does NOT involve CPU
  - So access protection bits on Translation Block Entry is ignored

DMA overview

- DMA transfer use physical address, and no memory protection any more?
  - Not the case, why?
- Modern phones are 64-bit, while many of their peripheral devices remain 32-bit
  - Address translation is needed
- And, we need memory protection for DMA transfer also

IOMMU(input/output memory management unit) and DART

- IOMMU is introduced to connect a direct-memory-access capable I/O bus to the main memory
- There is need 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

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

- For 64bit iOS devices, peripheral devices such as Broadcom WIFI module, implement DMA transfers

- Gal Beniamini of Google Project Zero team leverages DMA features on iOS to achieve firmware to host attack in middle 2017
  - Compromised WIFI firmware is able to mutate writable DMA mapping
  - Kernel trust that DMA mapping , lack of boundary check, leading to code execution

- Limitation of device-to-host attack
  - Short distance
    - Have to compromise WIFI stack first (Attacker and victim in same WIFI environment)

Long distance remote attack?

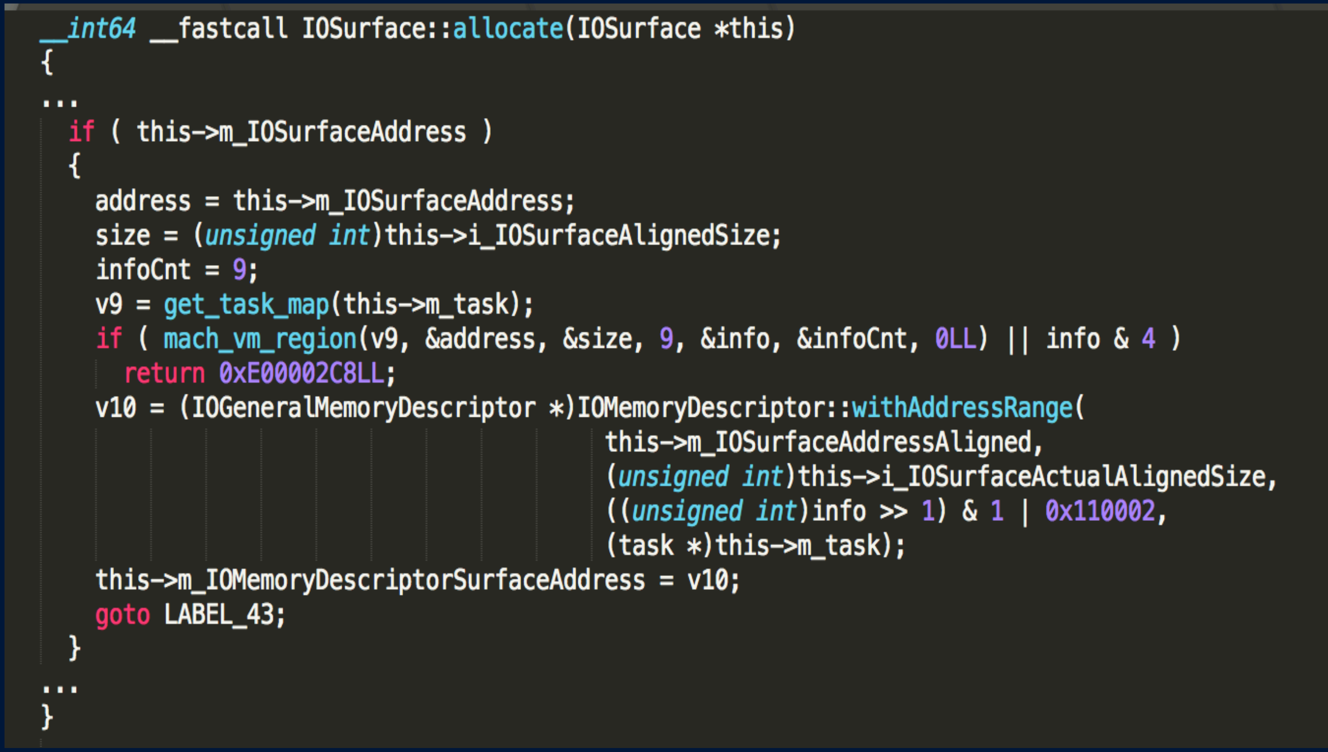
- Browser exploit + kernel privilege escalation
  - By using DMA related vulnerabilities? What?
- Looks like a crazy idea
  - DMA features are kind of low level implementation which is mostly performed at kernel level.
- Such feature is never exposed directly to the userland.
  - Is there any DMA attack surface in userland?

Indirect userland DMA

- There might be indirect DMA interface at userland
- Hardware JPEG engine to accelerate the encoding and decoding process
- IOSurface transform is done by hardware device
- Etc.

IOSurface and IOSurfaceAccelerator

- IOSurface object represents a userland buffer which is shared with the kernel.
- Users can create IOSurface by providing existing userland memory address and its length
- Kernel creates IOMemoryDescriptor and map userland memory to kernel
  - Option kIODirectionOut is set for read-only userland mappings
  - Option kIODirectionOut and kIODirectionIn are both set for read-write userland mappings



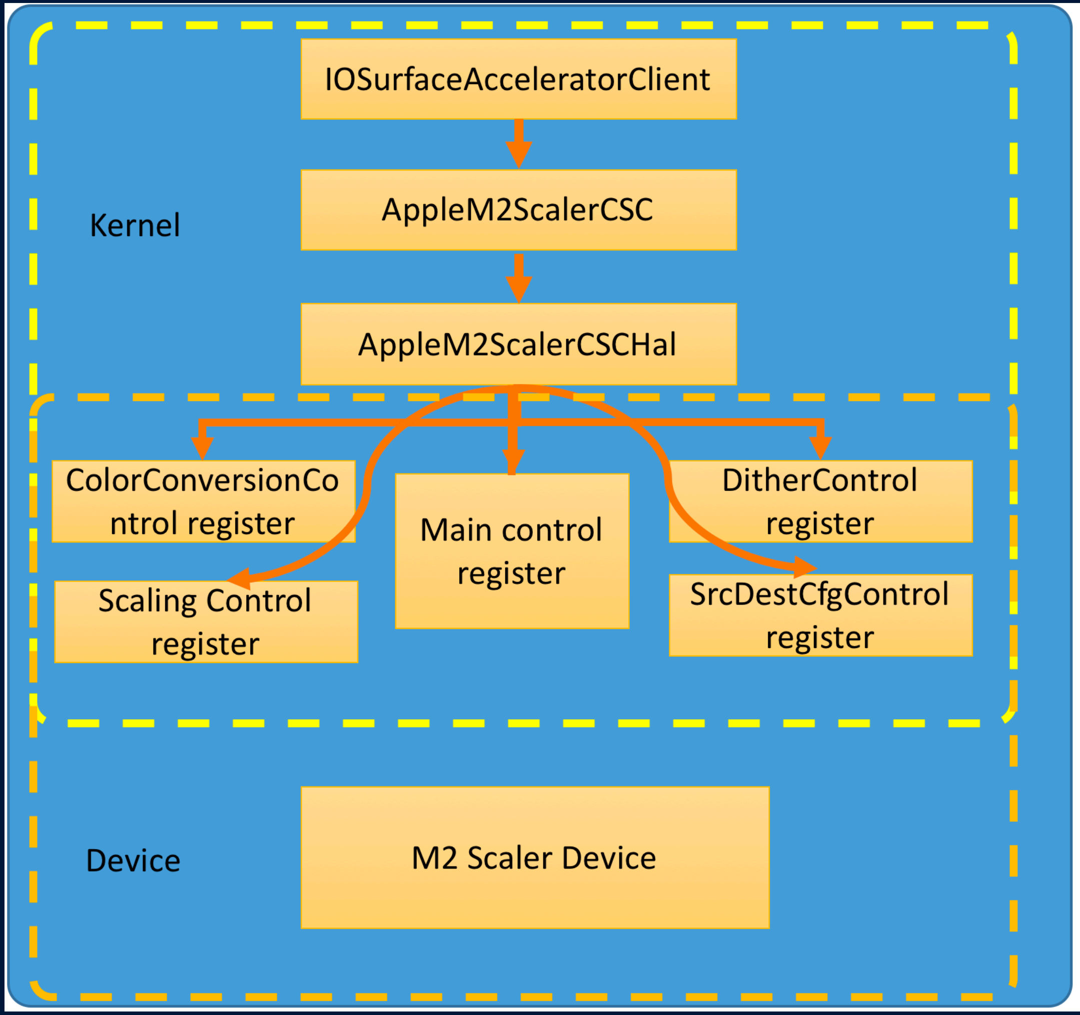
```
__int64 __fastcall IOSurface::allocate(IOSurface *this)
{
...
  if ( this->m_IOSurfaceAddress )
  {
    address = this->m_IOSurfaceAddress;
    size = (unsigned int)this->i_IOSurfaceAlignedSize;
    infoCnt = 9;
    v9 = get_task_map(this->m_task);
    if ( mach_vm_region(v9, &address, &size, 9, &info, &infoCnt, 0LL) || info & 4 )
      return 0xE00002C8LL;
    v10 = (IOGeneralMemoryDescriptor *)IOMemoryDescriptor::withAddressRange(
                                         this->m_IOSurfaceAddressAligned,
                                         (unsigned int)this->i_IOSurfaceActualAlignedSize,
                                         ((unsigned int)info >> 1) & 1 | 0x110002,
                                         (task *)this->m_task);
    this->m_IOMemoryDescriptorSurfaceAddress = v10;
    goto LABEL_43;
  }
...
}
```

IOSurface and IOSurfaceAccelerator

- IOSurfaceAccelerator is a userland framework on iOS platform only
- Two most important interfaces: IOSurfaceAcceleratorCreate and IOSurfaceAcceleratorTransferSurface
    - IOSurfaceAcceleratorCreate is responsible for creating a IOKit userclient connection representing an IOSurfaceAcceleratorClient kernel object
    - IOSurfaceAcceleratorTransferSurface takes two IOSurface handles, one for source and another for destination, along with a dictionary supplying the transferring parameters
        - Transfer the surface from src to dst, with color filtering, border filling etc, processed directly by the kernel
- Typical scenario of using IOSurfaceAccelerator: Screen Snapshot

Low level implementation of IOSurfaceAccelerator

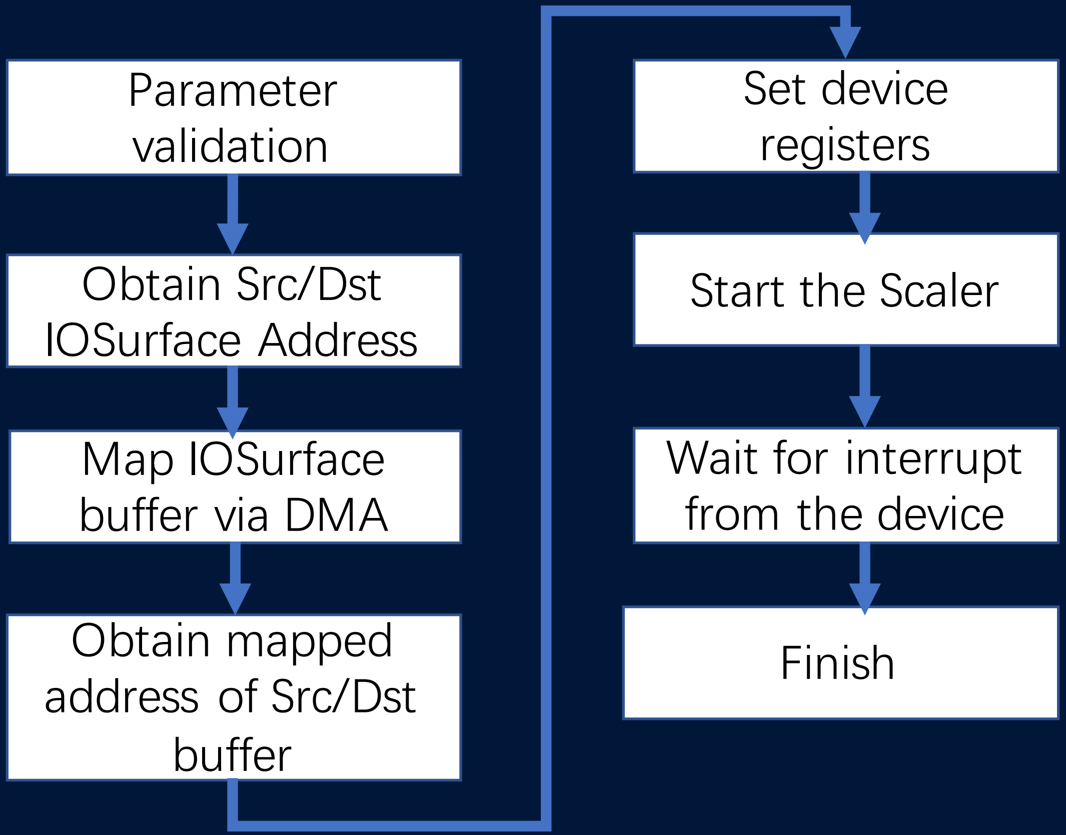
- AppleM2ScalerCSC driver architecture
  - IOSurfaceAcceleratorClient talks to AppleM2ScalerCSC which is the core driver
  - Low level driver object AppleM2ScalerCSCHal is for handling device dependent stuff and provide device independent interface to AppleM2ScalerCSCDriver
  - AppleM2ScalerCSCHal creates ColorConversionControl, SrcDestCfgControl, ScalingControl, DitherControl, ColorManager,etc.
  - AppleM2ScalerCSCHal maps a device memory into kernel virtual space, representing the device's key registers.



IOSurfaceAcceleratorTransferSurface Internals

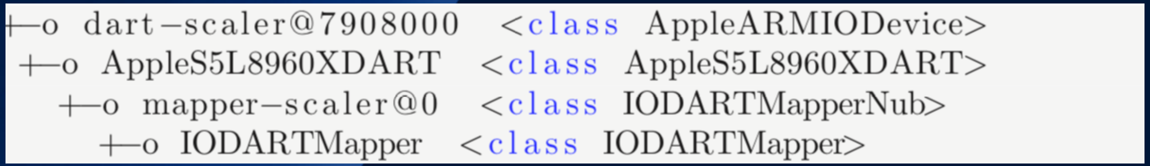
- IOSurfaceAcceleratorTransferSurface will reach kernel function IOSurfaceAcceleratorClient::user_transform_surface

- Overall procedure of user_transform_surface
  - We are interested in the DMA mapping part


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

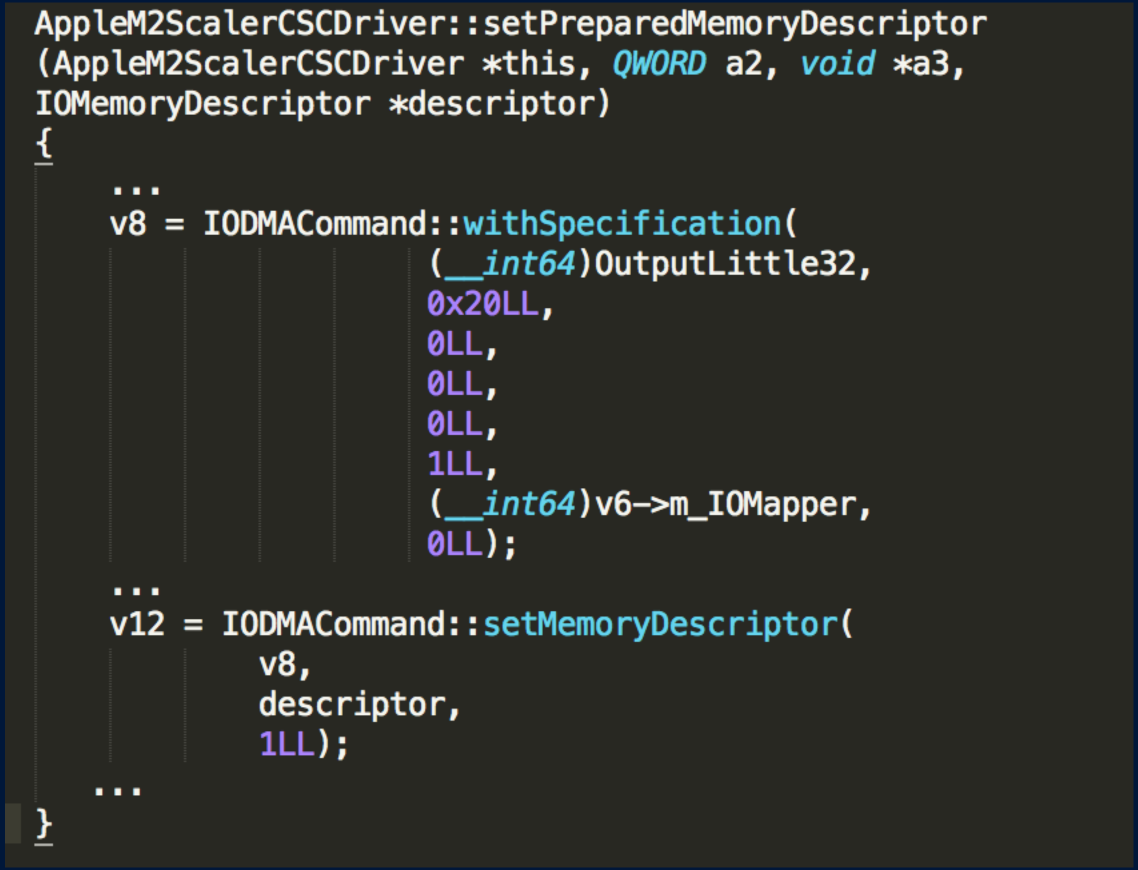
Map IOSurface buffer via DMA

- An IODMACommand object is created with a special IOMapper
  - m_IOMapper is an instance of IODARTMapper which is independent between devices



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

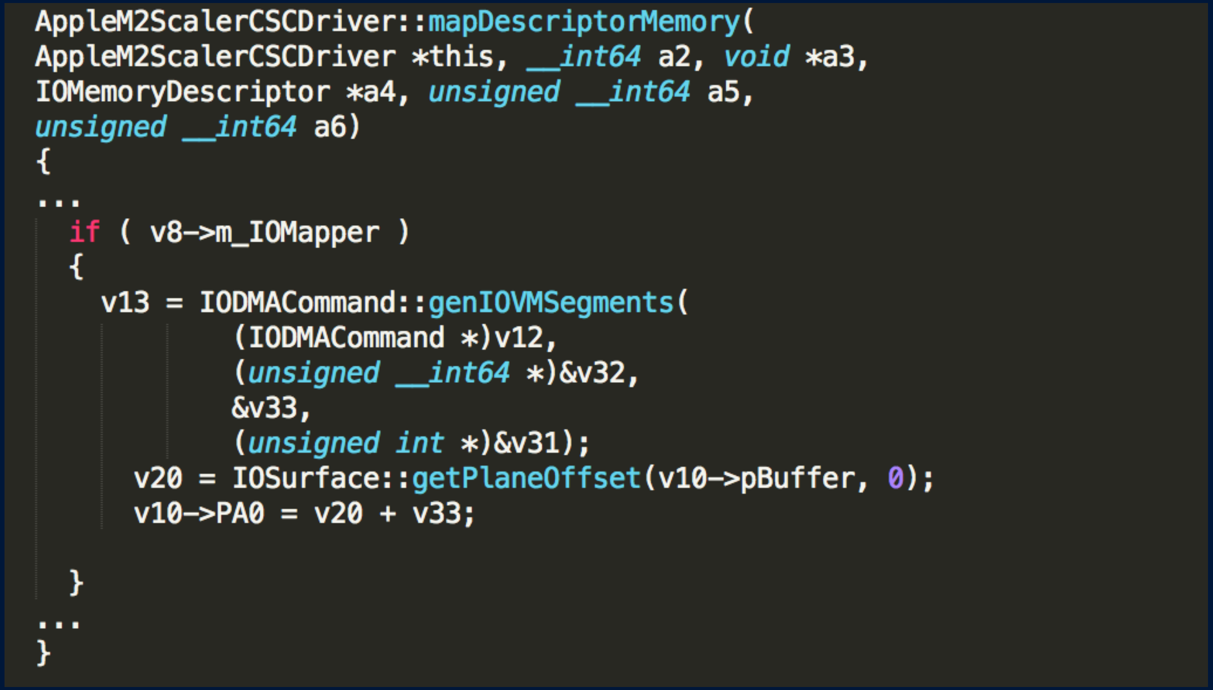
- Then the IOSurface memory descriptor is bond to the IODMACommand



```
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);
   ...
}
```

Obtain the IOSurface address in IOSpace

- After the mapping, the next step is to obtain the device memory visible to IOMMU
  - v33 here represents the 32bit address in IOSpace

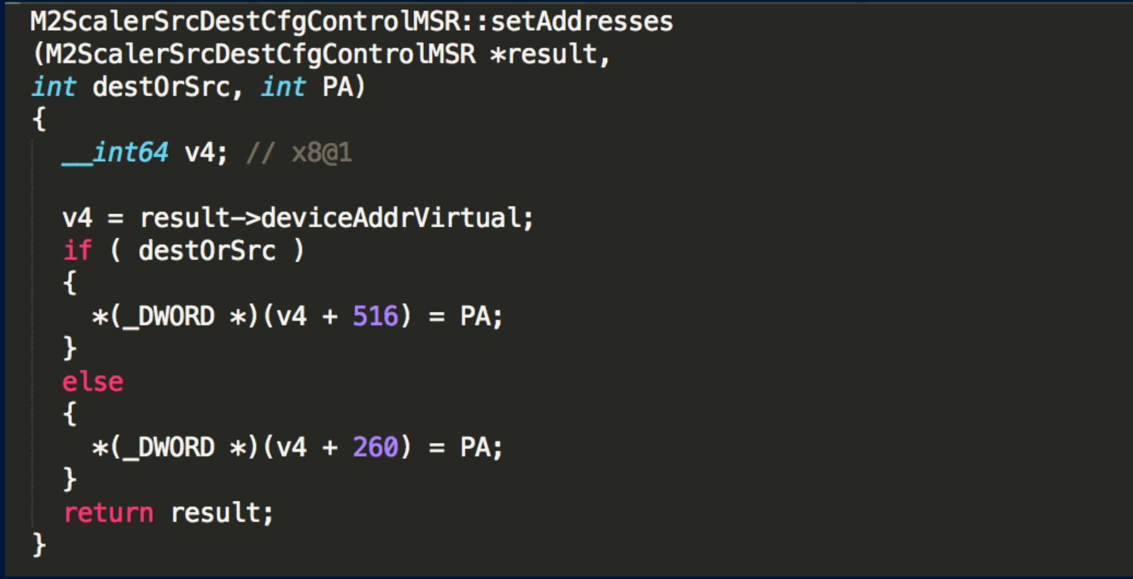


```
AppleM2ScalerCSCDriver::mapDescriptorMemory(
AppleM2ScalerCSCDriver *this, __int64 a2, void *a3,
IOMemoryDescriptor *a4, unsigned __int64 a5,
unsigned __int64 a6)
{
...
  if ( v8->m_IOMapper )
  {
    v13 = IODMACommand::genIOVMSegments(
                (IODMACommand *)v12,
                (unsigned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

  }
...
}
```

SrcDestCfgControl object

- SrcDestCfgControl object is used to specify the address of src and dst address of IOSurface in IOSpace
- After IOSurface's IOSpace address is decided, SrcDestCfgControl object is set
  - v4 represents the kernel virtual mapping of Scaler's device memory
- Other registers relating to transform parameters are handling in a similar manner

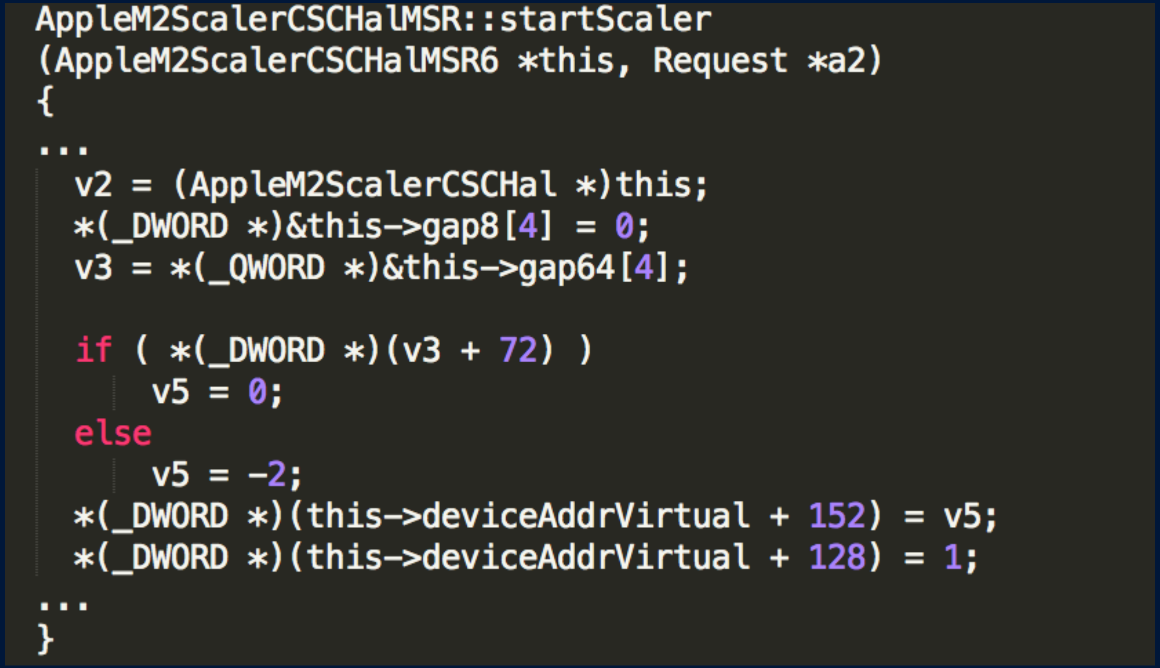


```
M2ScalerSrcDestCfgControlMSR::setAddresses
(M2ScalerSrcDestCfgControlMSR *result,
int destOrSrc, int PA)
{
  __int64 v4; // x8@1

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

Start the scaler

- After all registers are set, the scaler is started by setting the control register
  - (deviceAddrVirtual + 128) is the "powerOn" register of the scaler
- The scaler device starts its work immediately
- 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 per-formed a memory copy via DMA transfer.



```
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;
...
}
```

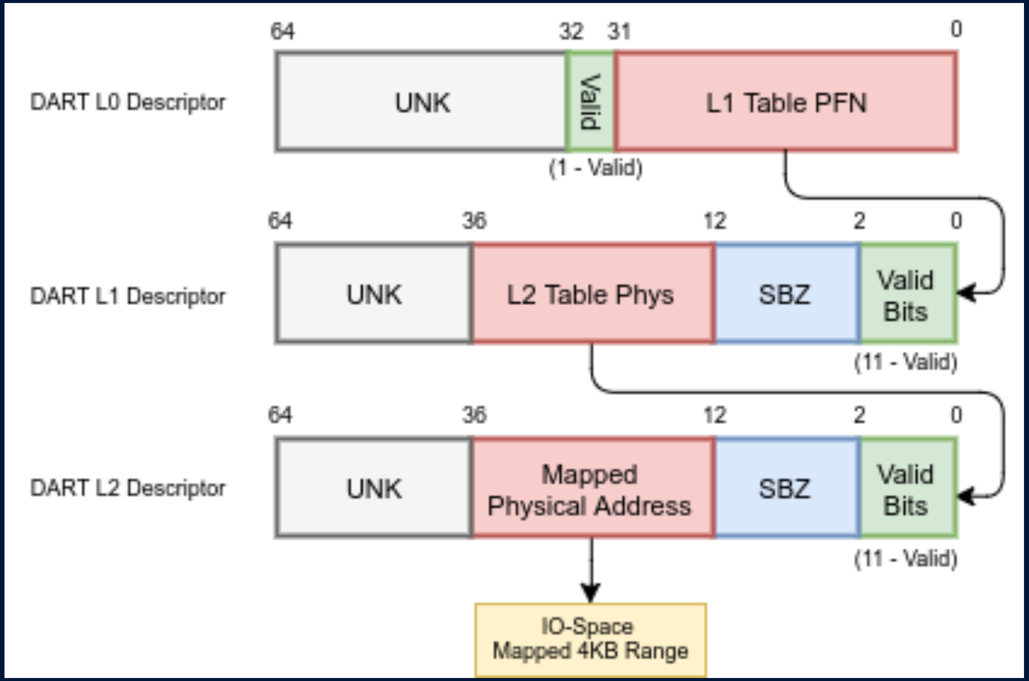
PART III: The implementation of IOMMU memory protection

Page table specification of IOMMU

- Unlike CPU MMU, page table specification of IOMMU is not well standardized

- On iOS DART is responsible for IOMMU page table management
  - So by reversing iOS DART code, we can understand the page table better

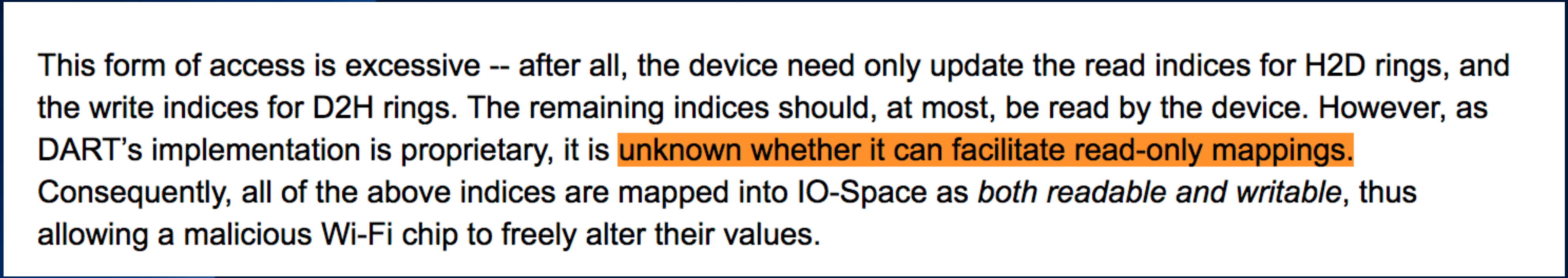
- Gal Beniamini has reversed the logic in DART module and explained the page table specification on 64bit iOS devices.
  - Similar as CPU MMU page table



Referenced from: https://googleprojectzero.blogspot.com/2017/10/over-air-vol-2-pt-3-exploiting-wi-fi.html

IOMMU memory protection

- In Gal’s blog he didn’t mention whether the IOMMU supports memory protection or not.
  - It remains unclear if IOMMU memory protection exists or not

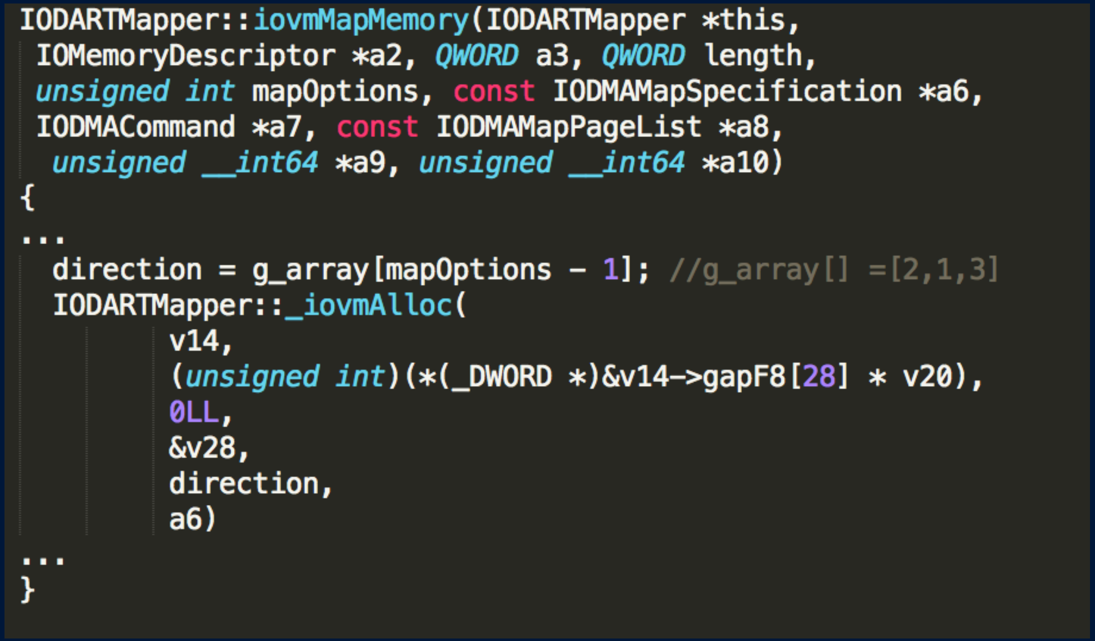


This form of access is excessive -- after all, the device need only update the read indices for H2D rings, and the write indices for D2H rings. The remaining indices should, at most, be read by the device. However, as DART’s implementation is proprietary, it is unknown whether it can facilitate read-only mappings. Consequently, all of the above indices are mapped into IO-Space as *both readable and writable*, thus allowing a malicious Wi-Fi chip to freely alter their values.

  - However we can get the answer by reversing DART implementation in iOS 11

IOMMU memory protection

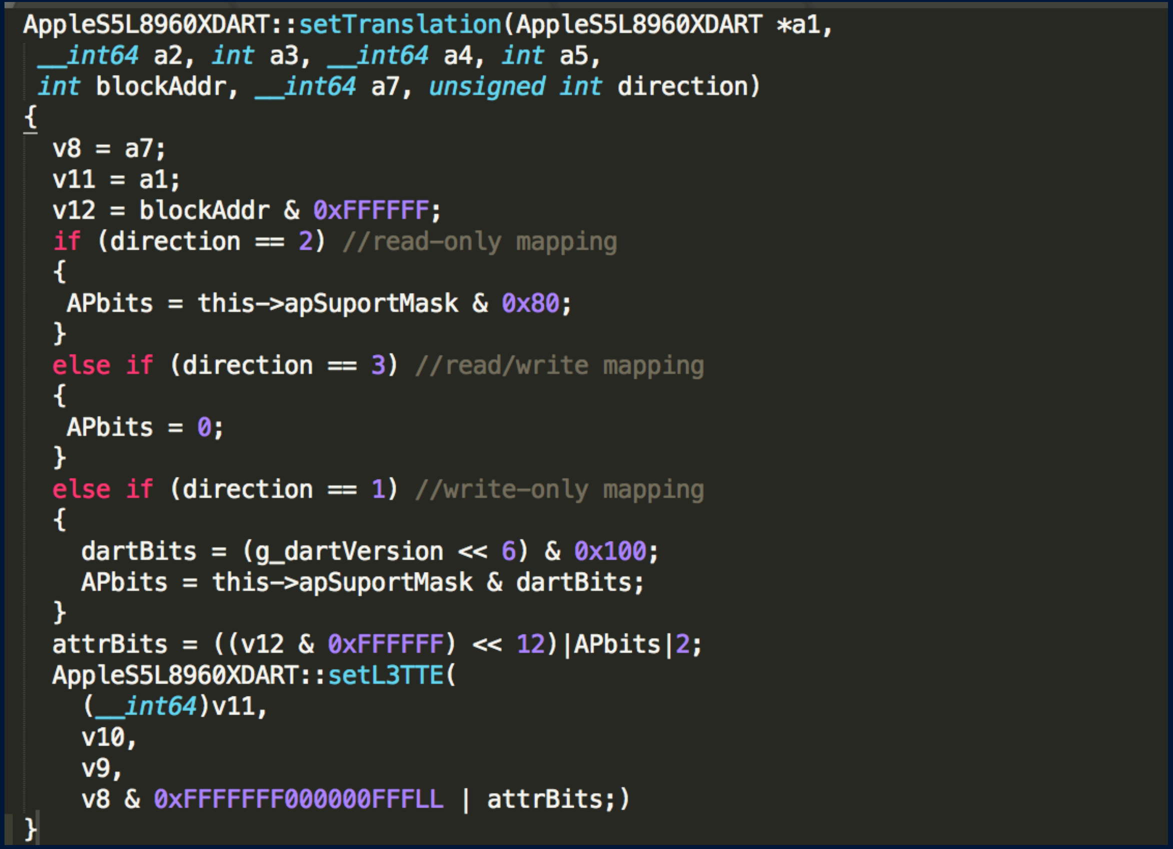
- The entry point for mapping memory in IOSpace: IODARTMapper::iovmMapMemory
  - mapOptions parameter has included the memory's protection bits in virtual space
  - The last 3 bits in mapOptions is translated to direction value
    - Read-only mapping has direction value 2
    - Write-only mapping has value 1
    - Read/write mapping has the value 3.



```
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)
...
}
```

IOMMU memory protection

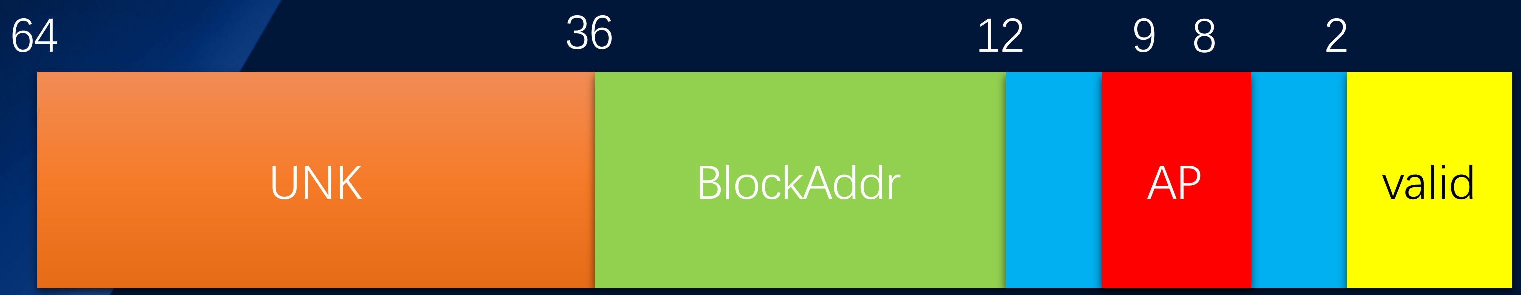
- The direction variable flows and finally reached the function in AppleS5L8960XDART, the low level implementation of DART
  - On a device later than iPhone 5s, the apSupport mask is always 0x180
  - Indicating the 8th and 9th bit in the TTE are AP related bits



```
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;)
}
```

IOMMU memory protection

- Finally we got the access protection specification in TTE
  - IOMMU supports memory protection



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

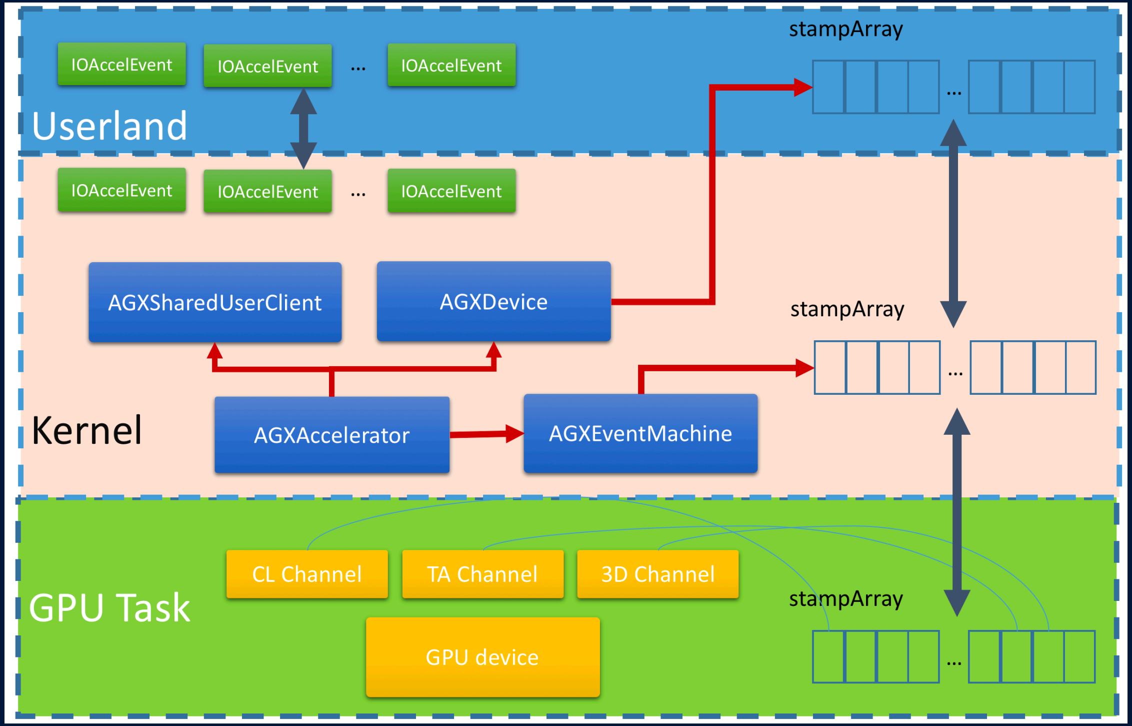
PART IV: GPU notification mechanism

Apple Graphics workflow

- On iPhone7 device, Apple Graphics provides with 128 channels for concurrent processing.
- Three channel categories: CL channel, GL channel and TA channel
- Kernel wraps drawing instructions from userland and put them into those channels
- Kernel then waits for the GPU to finish processing
- A well-designed notification mechanism is necessary for GPU to synchronize the status of instruction processing
  - Considering the concurrency of GPU handling

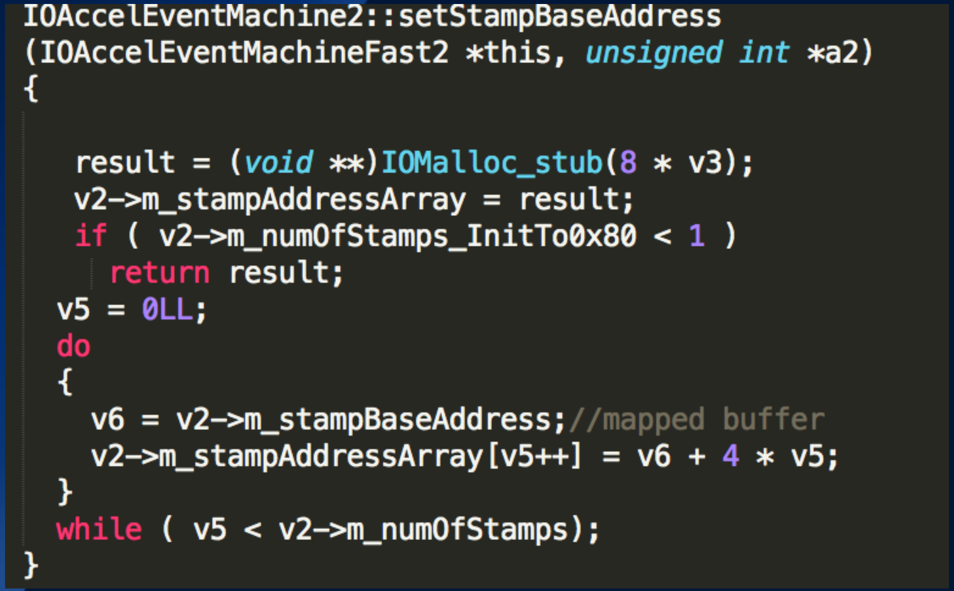
GPU notification architecture

- GPU task owns a stampArray representing the stamp status of each channel
  - The memory is a representation of uint32 array of 128 elements, each showing the last complete instruction's stamp of that channel
- stampArray memory is mapped to kernel and userland also (userland mapping is read-only)



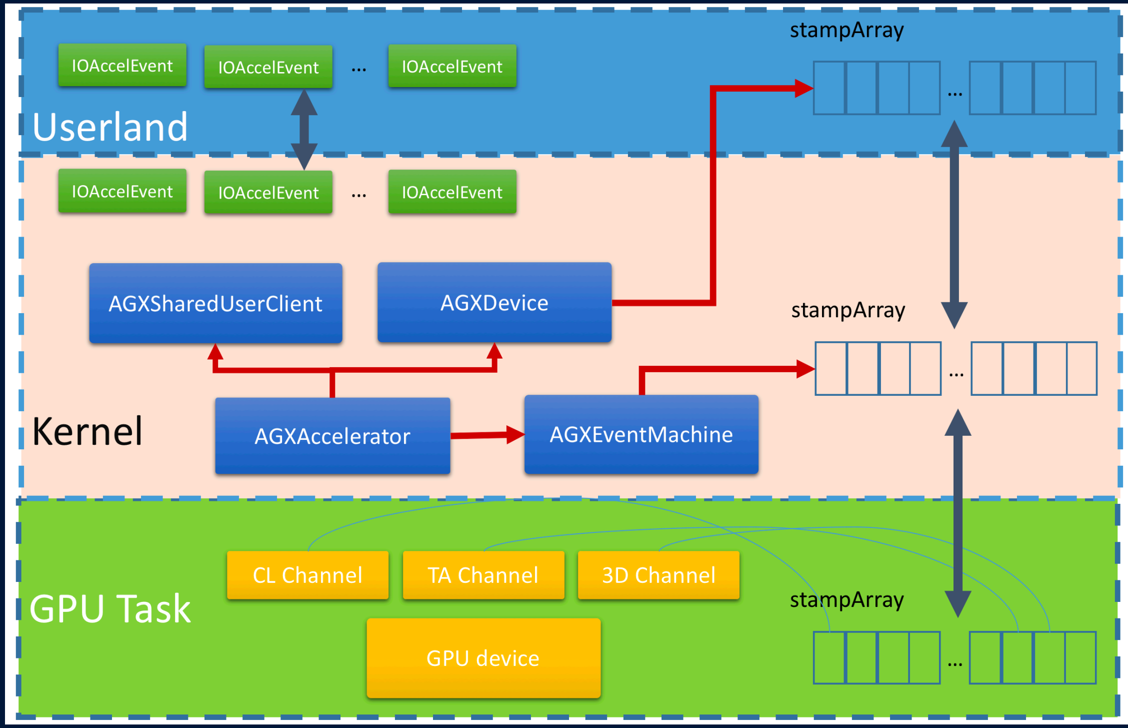
Stamp address array

- Kernel also maintains a stamp address array of 128 elements
  - Each element represents the virtual address of that mapped stamp status kernel address



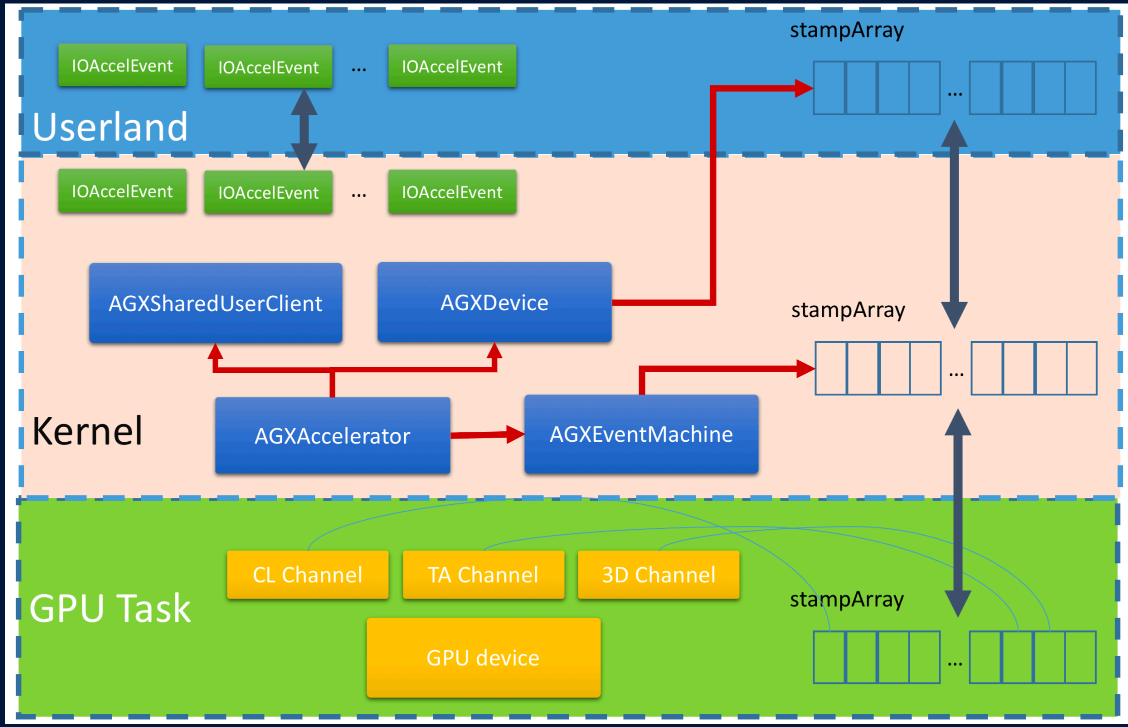
```
IOAccelEventMachine2::setStampBaseAddress
(IOAccelEventMachineFast2 *this, unsigned int *a2)
{

   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);
}
```



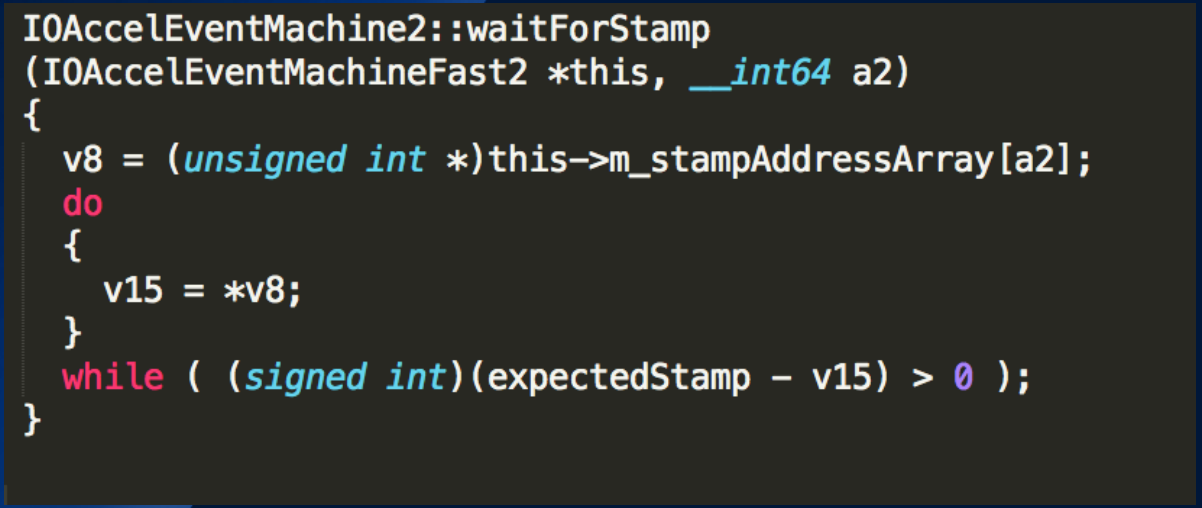
IOAccelEvent object

- The stamp value of each channel is incremental upon each of the instruction processing completeness.
- IOAccelEvent represents the expected stamp value in specific channel of one or one group of drawing instructions
  - One IOAccelEvent contains 8 sub events
  - One sub event is 8 bytes in size
    - Lower 4 bytes represents the channel index
    - Higher 4 bytes represents the expected stamp value of that drawing instruction

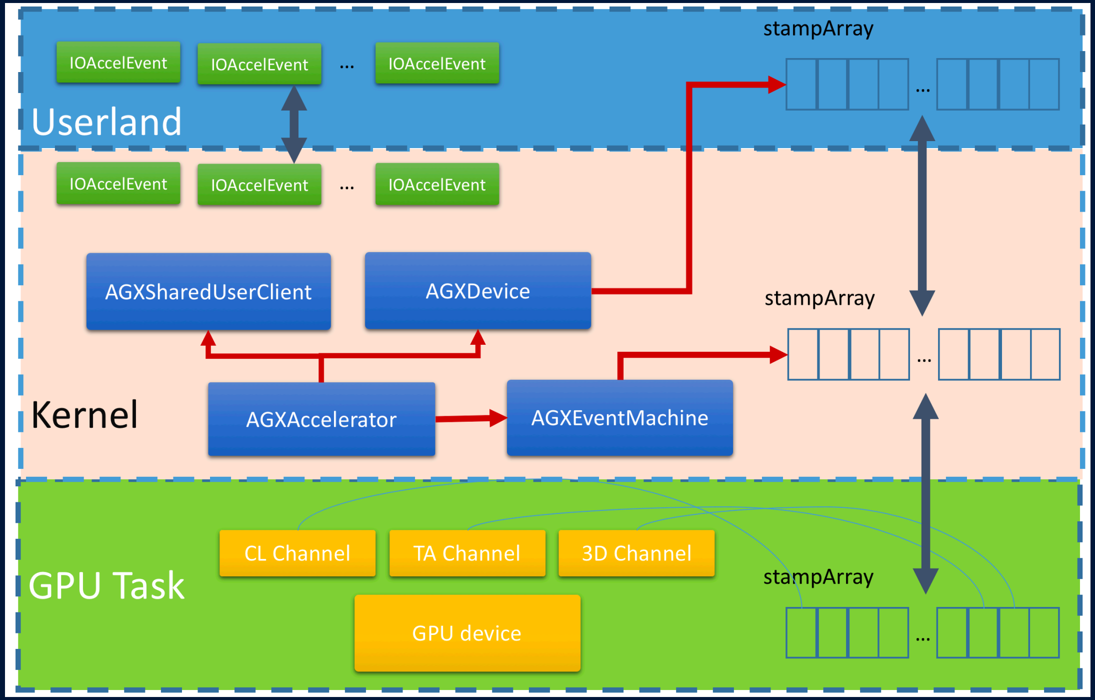


IOAccelEvent object

- By comparing the expectStamp value with the value in the stamp array, kernel can decide whether an IOAccelEvent has been finished or not

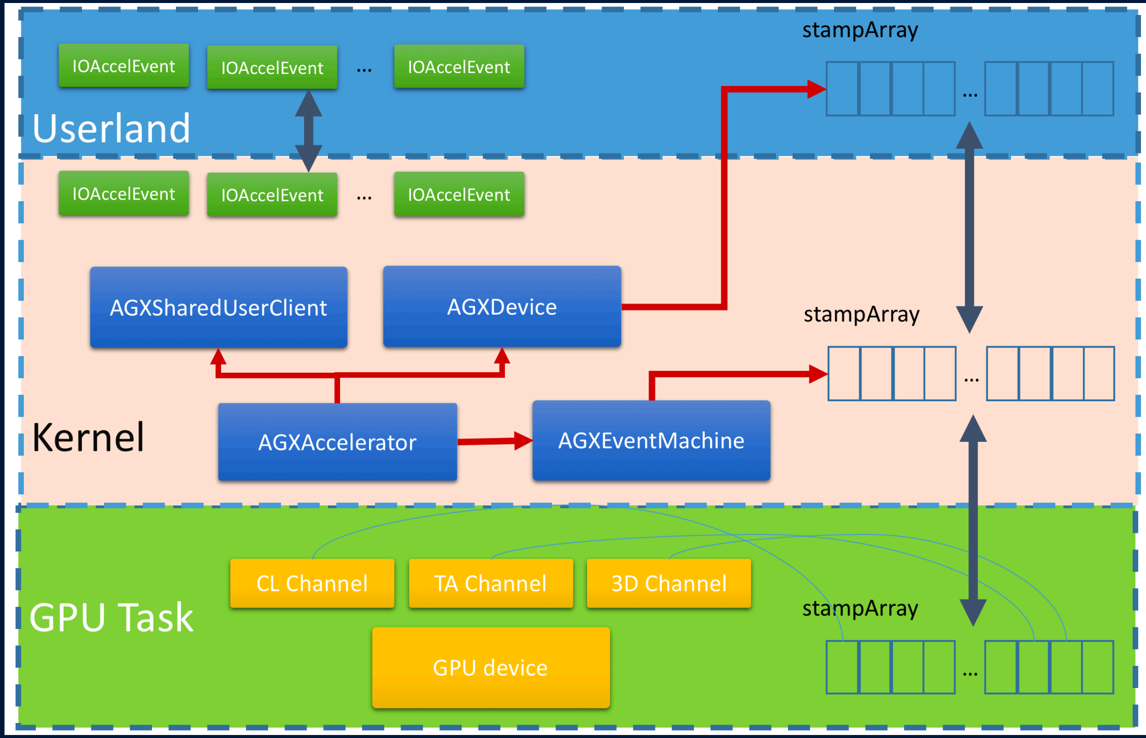


```
IOAccelEventMachine2::waitForStamp
(IOAccelEventMachineFast2 *this, __int64 a2)
{
  v8 = (unsigned int *)this->m_stampAddressArray[a2];
  do
  {
    v15 = *v8;
  }
  while ( (signed int)(expectedStamp - v15) > 0 );
}
```



IOAccelEvent object

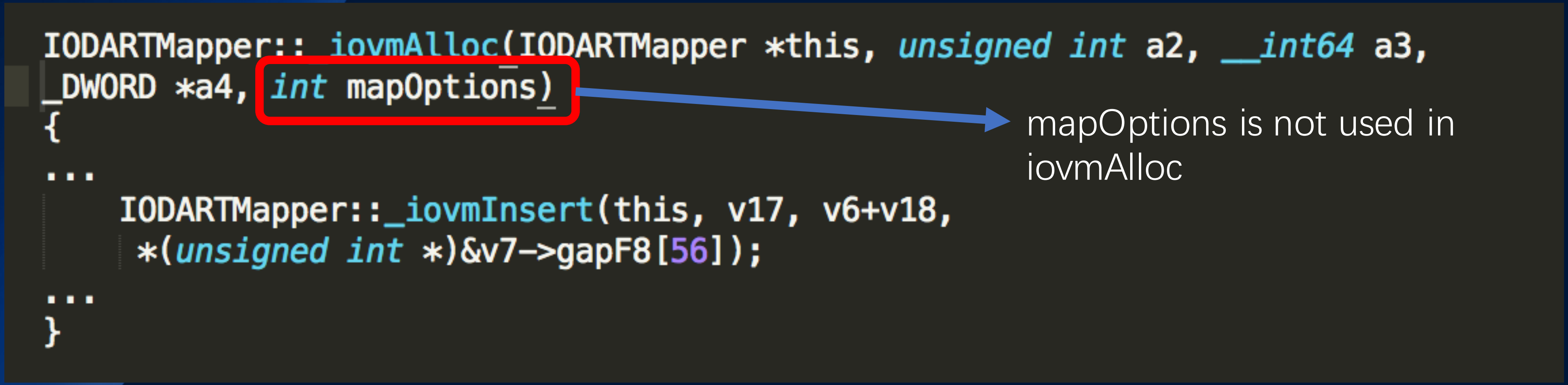
- To improve performance, some IOAccelEvent objects are mapped into userland as read-only
  - Make userland apps understand the status of an event without asking the kernel



PART V: The vulnerabilities

1. The DMA mapping vulnerability

- On iOS 10 and early beta of iOS 11, the mapOptions of the virtual memory is simply ignored by the DART mapper



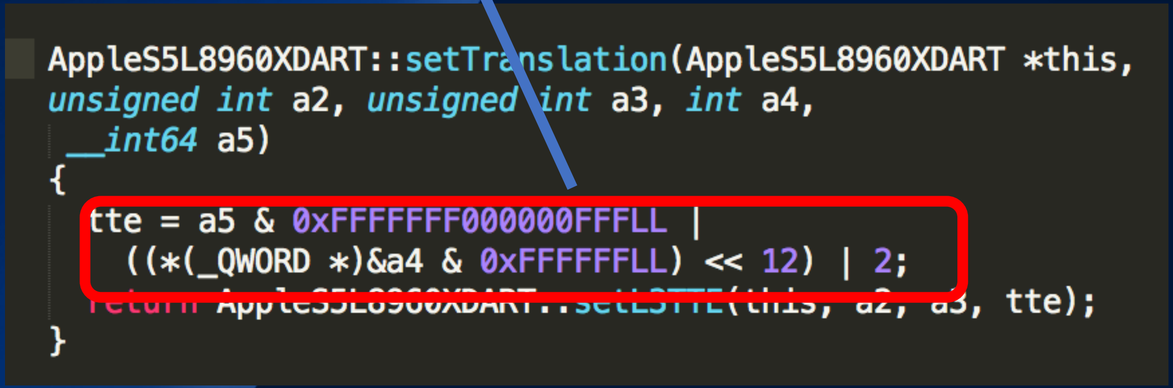
```
IODARTMapper:: iovmAlloc(IODARTMapper *this, unsigned int a2, __int64 a3,
_DWORD *a4, int mapOptions)
{
...
    IODARTMapper::_iovmInsert(this, v17, v6+v18,
     *(unsigned int *)&v7->gapF8[56]);
...
}
```

mapOptions is not used in iovmAlloc

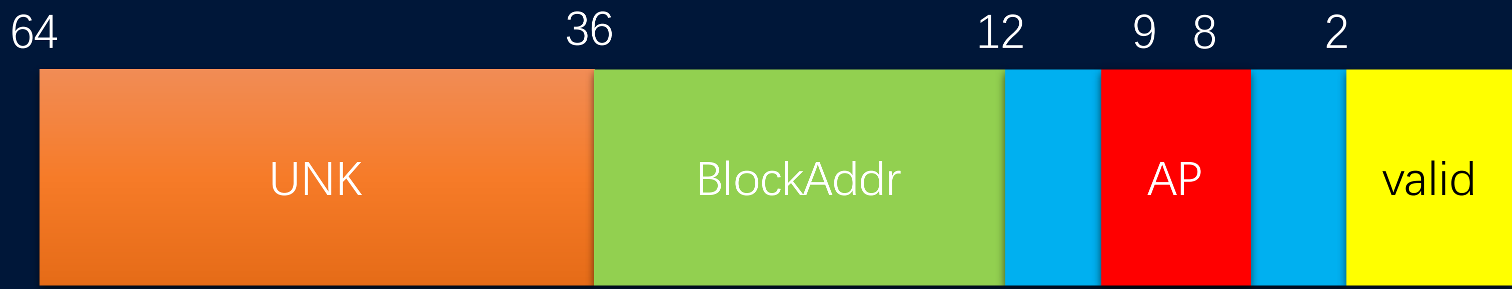
1. The DMA mapping vulnerability

- Later on, AppleS5L8960XDART::setL3TTE is reached, which 8th and 9th bit of the TTE set to 0
  - Indicating read-write mapping

8th and 9th bit set to 0



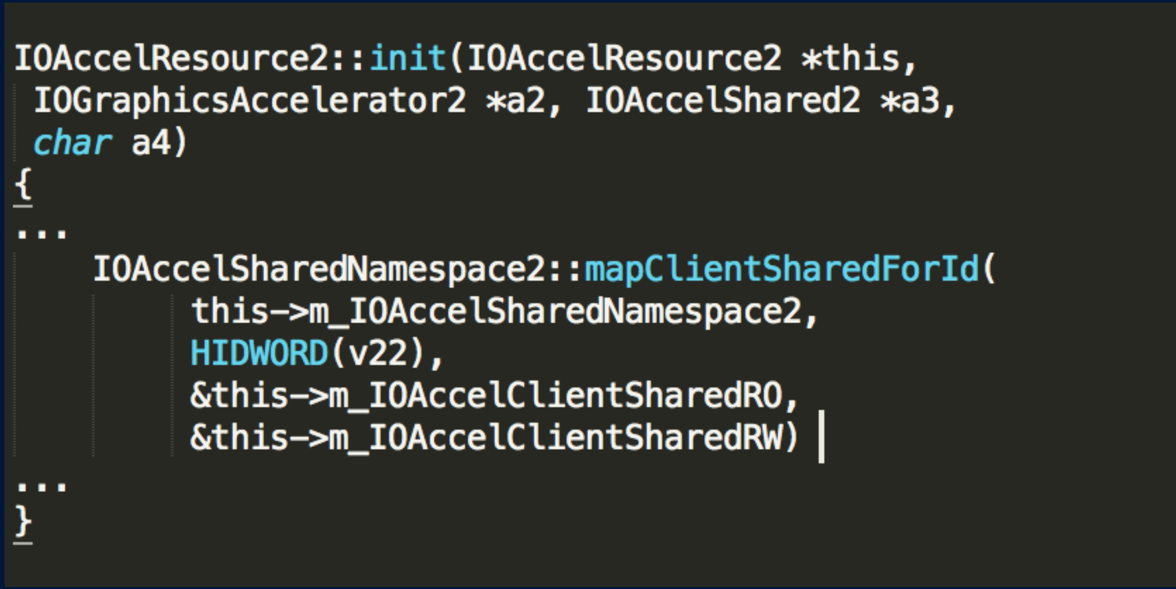
```
AppleS5L8960XDART::setTranslation(AppleS5L8960XDART *this,
unsigned int a2, unsigned int a3, int a4,
  __int64 a5)
{
  tte = a5 & 0xFFFFFFF000000FFFLL |
    ((*(_QWORD *)&a4 & 0xFFFFFFLL) << 12) | 2;
  return AppleS5L8960XDART::setL3TTE(this, a2, a3, tte);
}
```



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

2. The out-of-bound write vulnerability

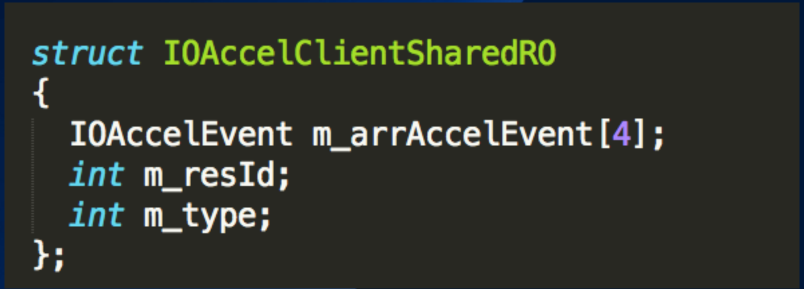
- IOAccelResource is similar in functionality as IOSurface object, except that IOAccelResource represents a shared userland buffer which would be mapped into GPU task.
- Like IOSurface, we can create IOAccelResource by specifying an existing userland buffer, by specifying the size, or even by providing an existing IOSurface handle.
- 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)
...
}
```

2. The out-of-bound write vulnerability

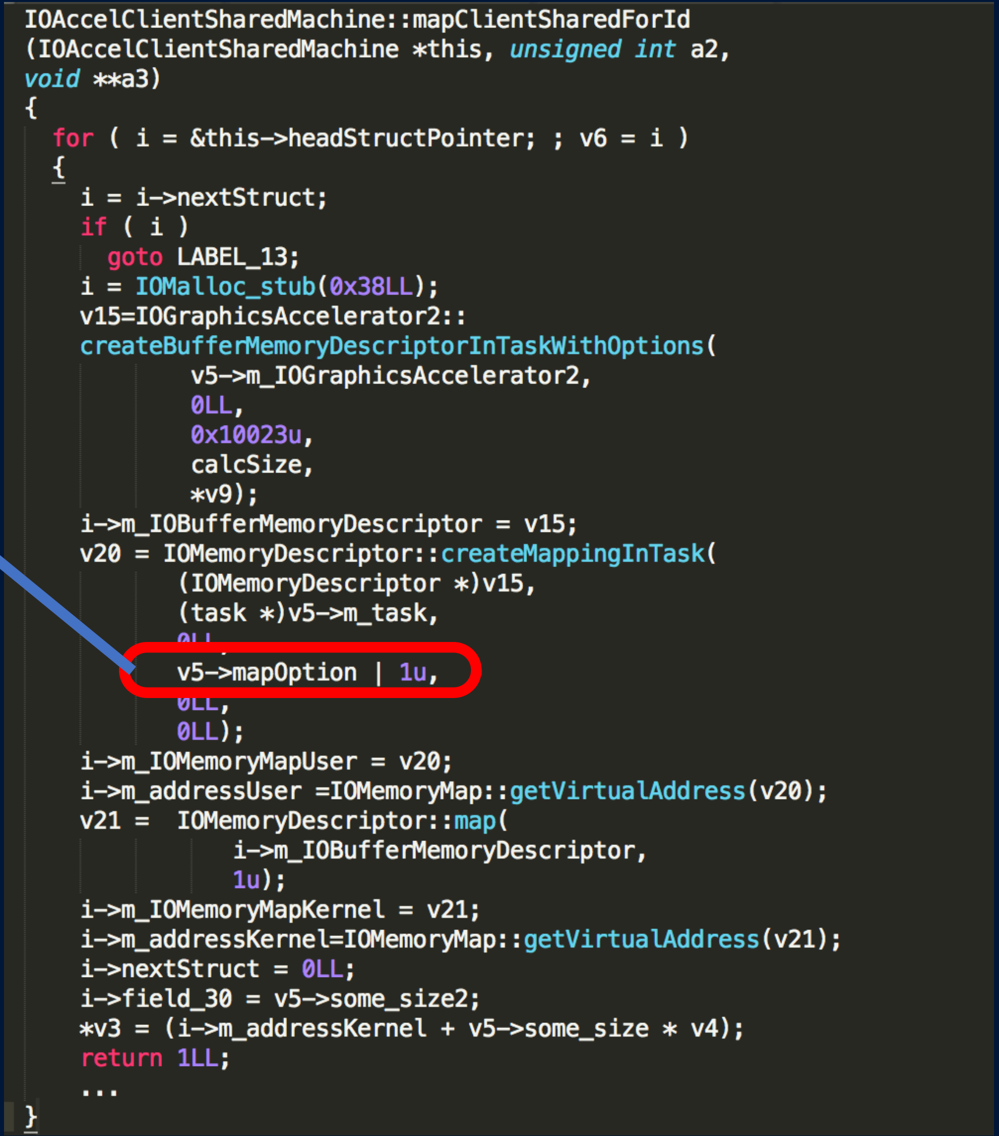
- What is IOAccelClientSharedRO?
  - Contains IOAccelEvent array with 4 elements
  - With resource id and type information



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

- m_IOAccelClientSharedRO is mapped to both userland and the kernel
  - userland mapping is read-only
- The userland mapped address is returned to user

Read-only mapping

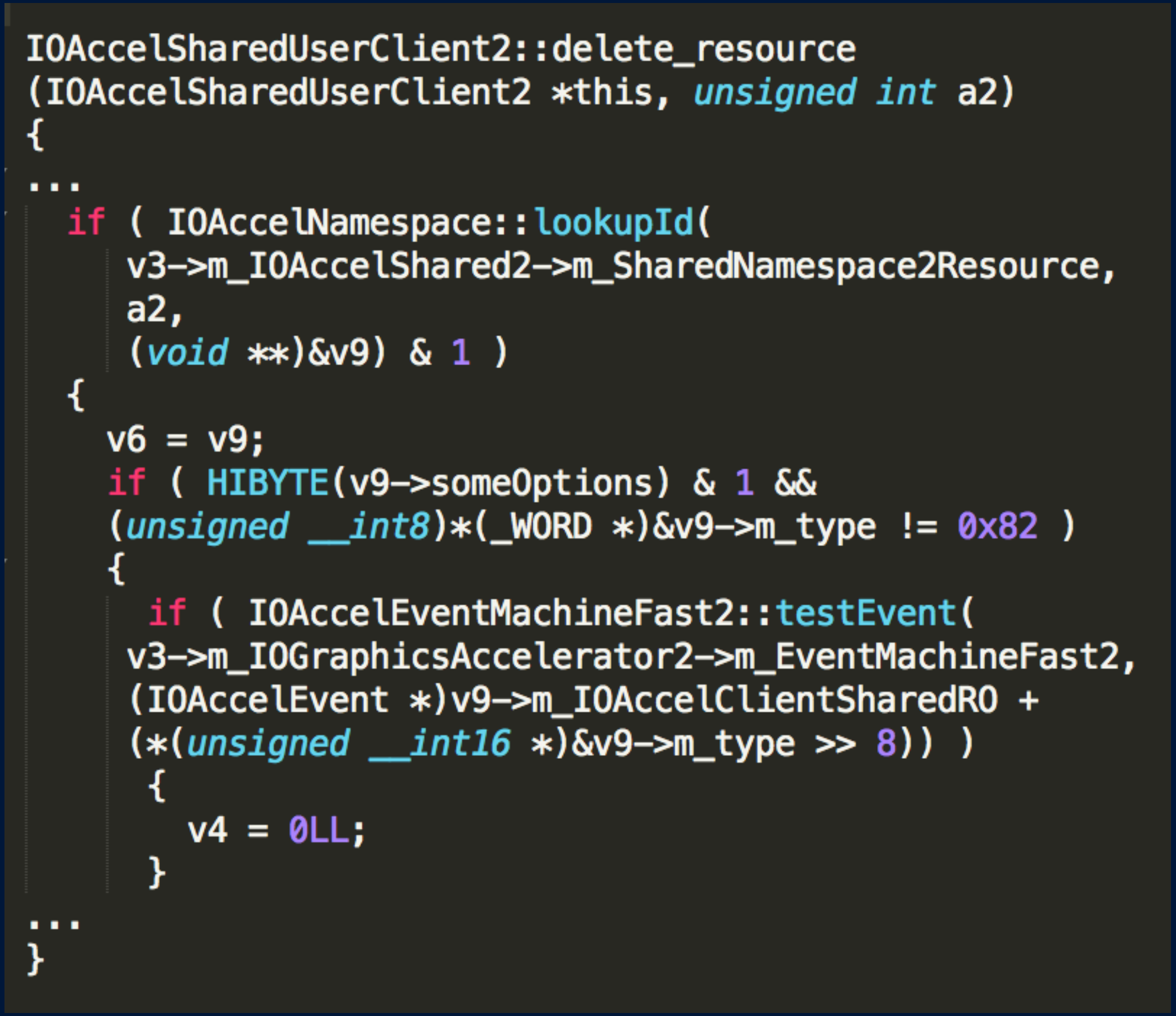


```
IOAccelClientSharedMachine::mapClientSharedForId
(IOAccelClientSharedMachine *this, unsigned int a2,
void **a3)
{
  for ( i = &this->headStructPointer; ; v6 = i )
  {
    i = i->nextStruct;
    if ( i )
      goto LABEL_13;
    i = IOMalloc_stub(0x38LL);
    v15=IOGraphicsAccelerator2::
    createBufferMemoryDescriptorInTaskWithOptions(
            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;
    ...
}
```

2. The out-of-bound write vulnerability

- Userland application can delete the IOAccelResource by calling method 1 of IOAccelSharedUserClient

- IOAccelEventMachineFast2::testEvent will be called if the IOAccelResource is created with specific option

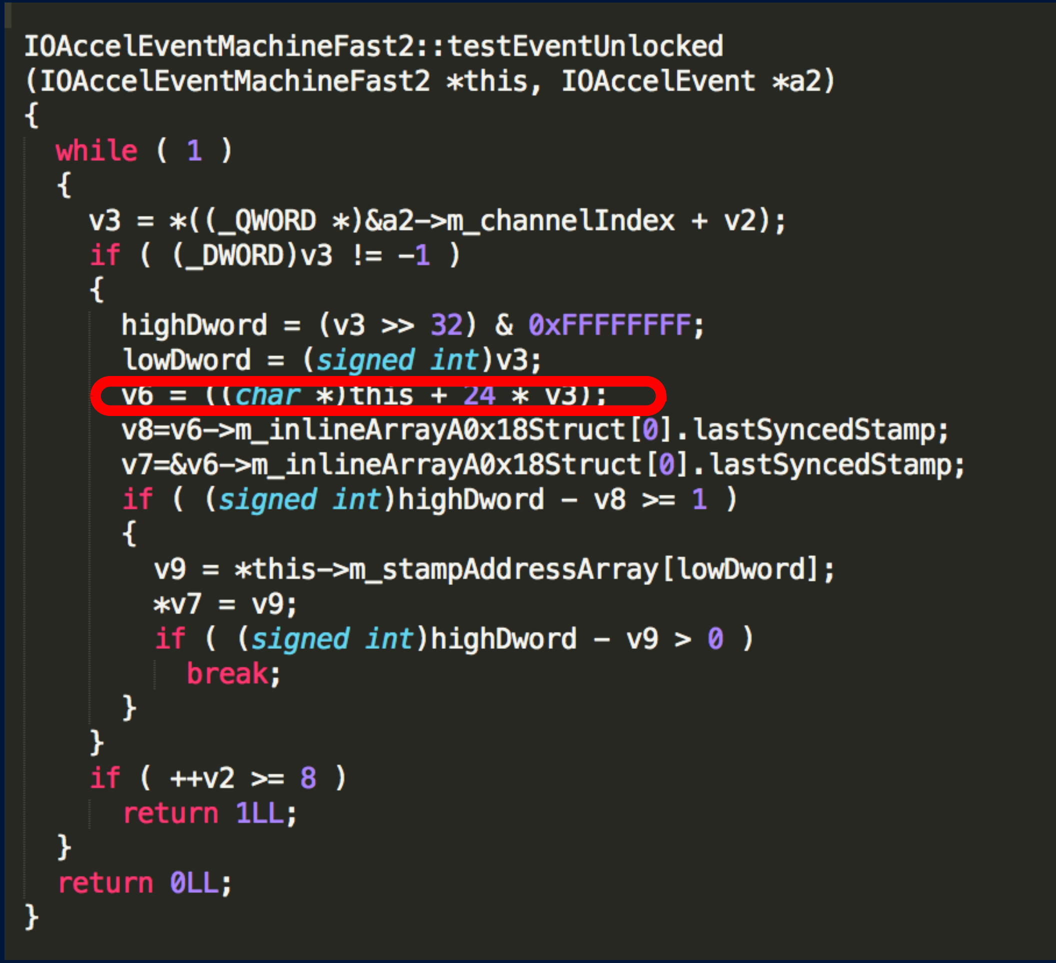


```
IOAccelSharedUserClient2::delete_resource
(IOAccelSharedUserClient2 *this, unsigned int a2)
{
...
  if ( IOAccelNamespace::lookupId(
     v3->m_IOAccelShared2->m_SharedNamespace2Resource,
     a2,
     (void **)&v9) & 1 )
  {
    v6 = v9;
    if ( HIBYTE(v9->someOptions) & 1 &&
    (unsigned __int8)*(_WORD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

2. The out-of-bound write vulnerability

- In IOAccelEventMachineFast2::testEvent, it checks whether the IOAccelEvent in IOAccelClientSharedRO has been completed

- No boundary check is performed at channel index
  - Since the IOAccelEvent is created and managed only by the kernel while the userland mapping is read-only
  - Kernel trust the index value



```
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;
}
```

2. The out-of-bound write vulnerability

- However with the DMA mapping bug, the trust boundary is broken
- Channel index and expected stamp value can be mutated by the userland app
- 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

PART VI: Exploitation & Demo

Exploitability

- 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
  - Then we can perform arbitrary memory read and write
- Looks like a hard task, because
  - Both arrays are created in very early stage of iOS boot process, impossible to put controlled contents right after them
  - Size of element of each array is different. The larger index we specify, the longer span in address of the referenced array element.

Craft memory layout

- Some facts:
  - 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
  - Addresses of m_inlineArrayA0x18Struct and m_stampAddressArray are closed with each other

- It means
  - We can use a relative large channel index
  - Along with kernel spray techniques, we might be able to ensure content of the OOB value of m_inlineArrayA0x18Struct[channelIndex] and m_stampAddressArray[channelIndex] are both under our control

Feasibility of memory layouting

- Kernel heap spray
  - Quite some known techniques: e.g. sending large OOL message

- On iPhone 7, we can spray around 350MB kernel memory within container sandbox app

- After m_inlineArrayA0x18Struct and m_stampAddressArray are created, around 50MB extra kernel memory is allocated (As a part of boot process)

Feasibility of memory layouting

- m_inlineArrayA0x18Struct element size is 24 bytes
- m_stampAddressArray element size is 8 bytes
- So if:
  - $index * 24 < 350MB + 50MB$, and
  - $index * 8 > 50MB$
- With index value in range of [0x640000, 0x10AAAAA], the out-of-bound elements of both m_inlineArrayA0x18Struct and m_stampAddressArray arrays can be fallen into our sprayed data

Arbitrary read and write?

- Arbitrary memory read is not a problem
  - Because m_stampAddressArray element size is 8 bytes
  - You can reach every dword within each page (higher 4 bytes of a QWORD cannot be read though, but usually not necessary)

- Arbitrary memory write is more tricky
  - Because m_inlineArrayA0x18Struct element size is 24 bytes, only one of its DWORD can be written

Write at specific offset 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
  - 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.
- So how to write arbitrary page offset by OOB write on m_inlineArrayA0x18Struct array
  - For example, a vtable is usually at offset 0 of the page if the object size is large enough
- The problem can be solved by congruence theory in mathematics

Read at arbitrary offset within a page

- Because:
  - $0xc000 \equiv 0(mod0x4000)$
- So with arbitrary integer n:
  - $n * 0x800 * 24 \equiv 0(mod0x4000)$
- With 0x4000 / 24 * 0xF0 / 16 = 0x27f6, we can get:
  - $0xF0 + 0x27f6 * 24 + n * 0x800 * 24 \equiv 0(mod0x4000)$
- With arbitrary integer n, we can out-of-bound write to the first 8 bytes in a sprayed page given:
  - $index = 0x27f6 + n * 0x800$
- To reach arbitrary offset within page, with m be the offset of the page, we just ensure index is:
  - $index = 0x27f6 + 0x2ab * m/8 + n * 0x800$

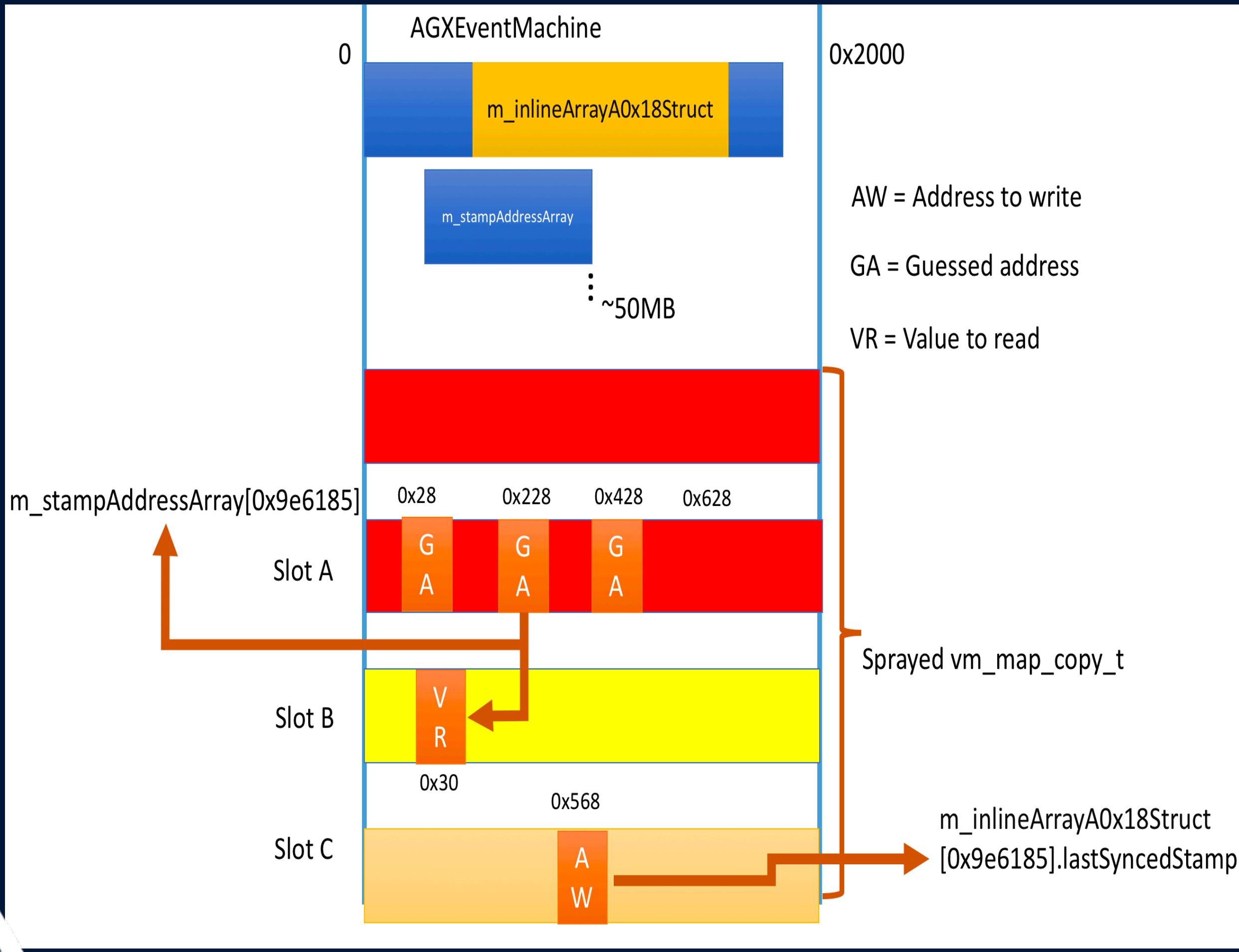
Choose the best channelIndex value

- With $index = 0x27f6 + 0x2ab * m/8 + n * 0x800$, plus the range of [0x640000, 0x10AAAAA]

- We choose the value of 0x9e6185 (can be other values as well)
  - This value can reach offset 0x568 of a 0x4000 page by writing m_inlineArrayA0x18Struct out-of-bound
  - And reach offset 0x28/0x228/0x448, etc... of a 0x4000 page by reading m_stampAddressArray out-of-bound

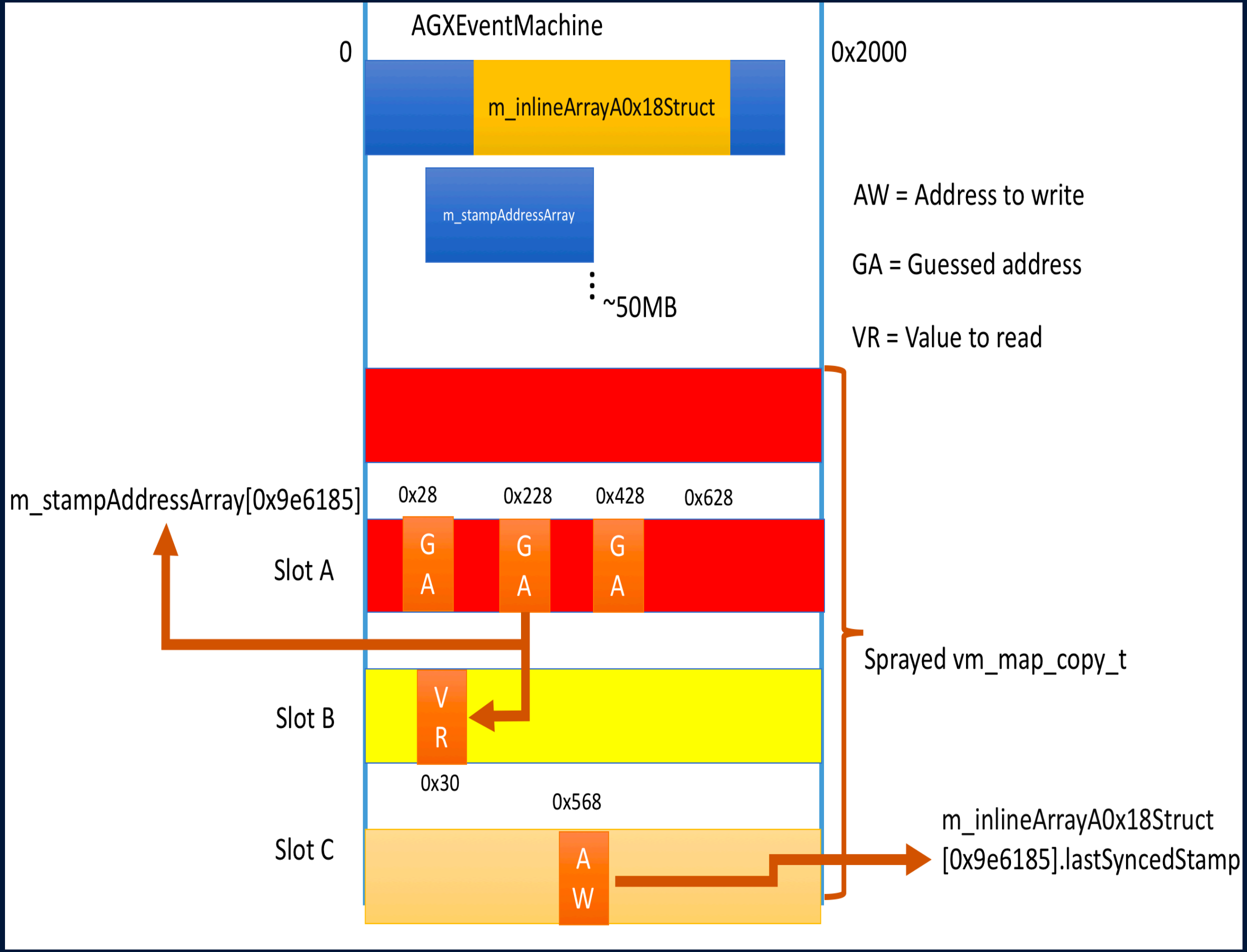
First attempt to exploit

- Spray around 350MB OOL messages of size 0x2000
- Make sure
  - Offset 0x28, 0x228,0x428,0x628… of each page is filled with a guessed value(GA), in real case we use 0xffffffe00a5d4030
  - Each QWORD in page offset 0x30 is filled with a value to read(VR)
  - Each QWORD in page offset 0x568(AW field) is filled with a specific value(make sure VR is different with this value)



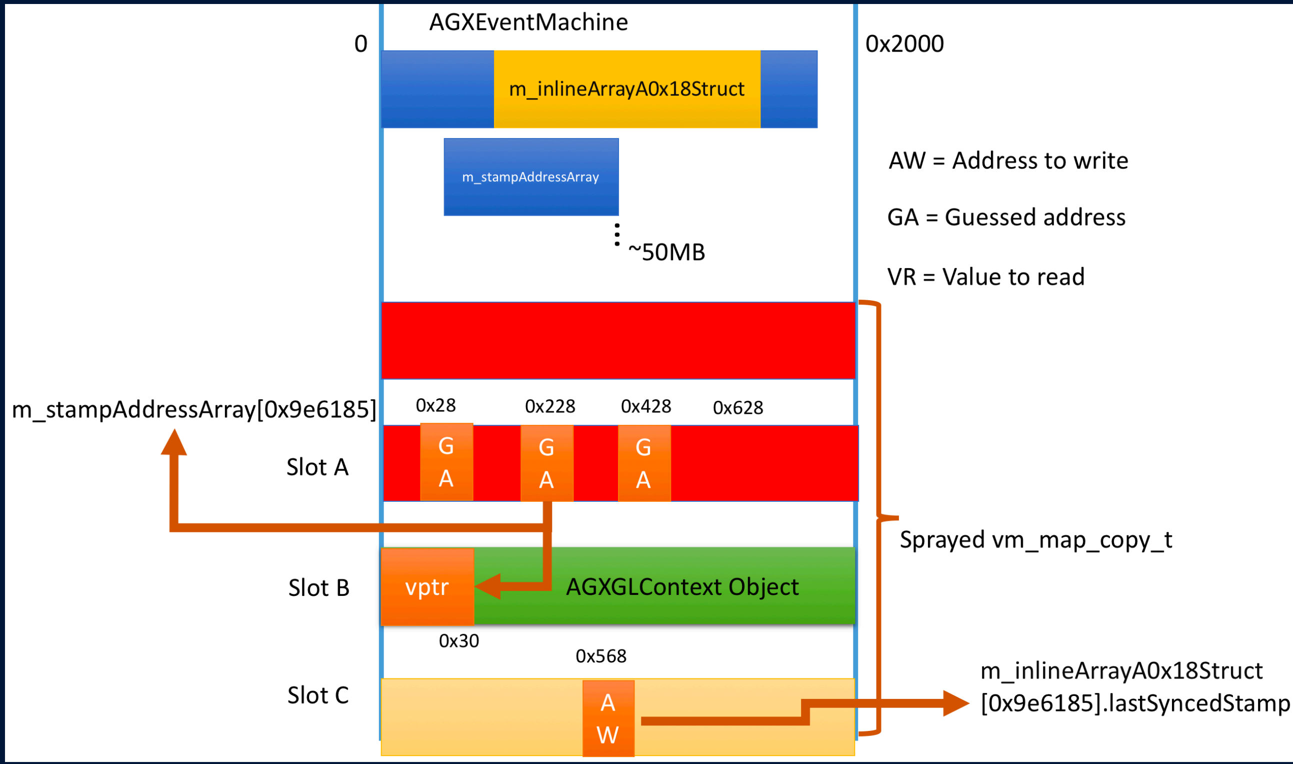
KASLR bypass

- By first attempt of the exploit, we then receive the OOL message to see which message's AW field has been changed
- We can then obtain the information of:
  - Which OOL message is allocated at slot C
  - Which OOL message is in GA address(slot B)



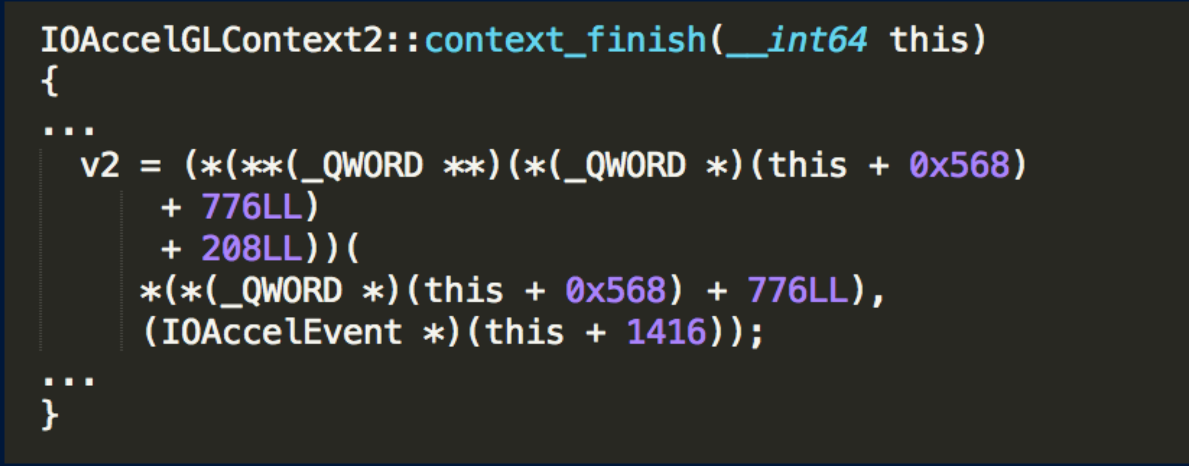
KASLR bypass

- Fill slot B with AGXGLContext object(also in kalloc.8192)
- Change GA value to 0xffffffe00a5d4000(original GA-0x30, base address of slot B, vtable of AGXGLContext object)
- Exploit the OOB bug again
- And we get the lower 4 bytes of AGXGLContext vtable by receiving OOL message in slot C

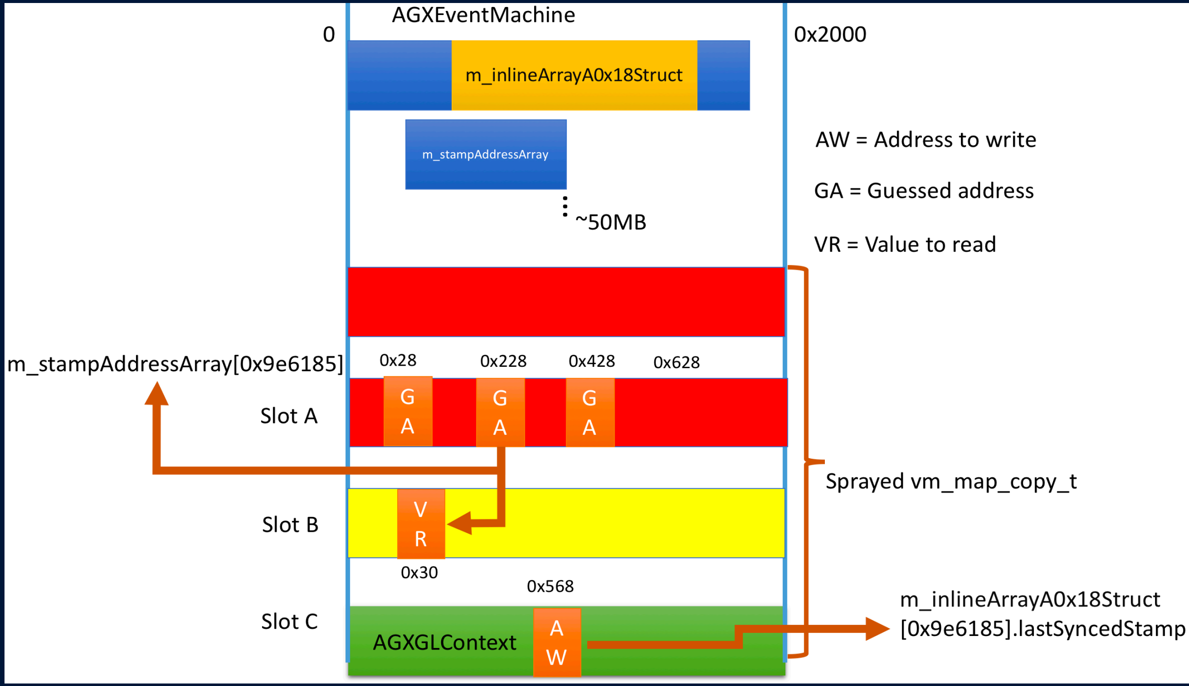


Code execution

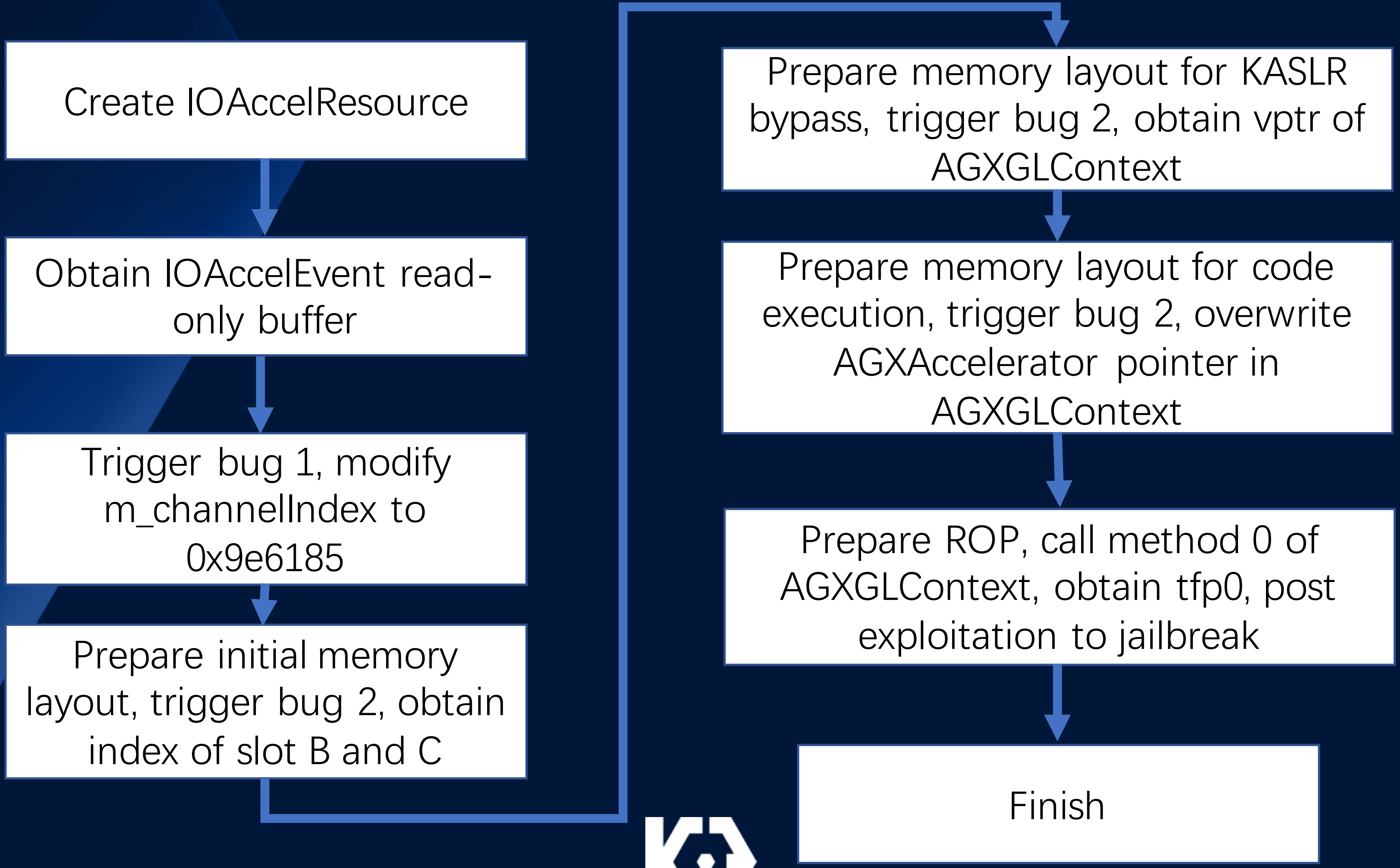
- To get code execution, we need to free the slot C, and fill in an object where its 0x568 offset represents important object
- The offset 0x568 in AGXGLContext is set to AGXAccelerator object
- Calling AGXGLContext method 0 can reach the function IOAccelGLContext2:: context_finish
- With its 0x568 field modified to arbitrary value , we get PC control



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



Overall exploit workflow


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

Post exploitation

- By running a piece of ROP gadget, we obtain task_for_pid 0
- Still far away from the jailbreak
  - Break AMFI
  - Rootfs remount
  - KPP/AMCC bypass
  - Etc.
- Most of those have public write-ups
- Not a scope of the talk

Demo

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, remains unfixed
- A good example of how security can be broken by bad implementation with good hardware design
    - We make possible a complex exploit chain to achieve jailbreak within userland applications
- Should kernel trust the userland read-only mappings?
- Last but not the least, userland read-only memory can be dangerous

Thanks & Questions