Exploiting IOSurface 0

Liang Chen@Pangu Team

Agenda

- IOSurface overview
- IOSurface 0 and exploitation techniques
- New mitigations overview (for late iOS 12 and iOS 13)
- Conclusion

IOSurface Overview

- IOSurface object represents a userland buffer which is shared with the kernel.
- Fundamental framework for both iOS and macOS
- Users can create IOSurface in userland, within container/WebContent sandbox

IOSurface Creation

- IOSurfaceRootUserClient method 0, 6, 7
  - IOSurfaceRootUserClient::s_create_surface
  - IOSurfaceRootUserClient::s_create_surface_fast_path
  - IOSurfaceRootUserClient::s_create_surface_client_mem

- IOSurfaceRootUserClient::s_create_surface requires user to provide a dictionary including key parameters of the IOSurface

- IOSurfaceRootUserClient::s_create_surface_fast_path and IOSurfaceRootUserClient::s_create_surface_client_mem are simplified version of IOSurfaceRootUserClient::s_create_surface

- In all cases, IOSurfaceRoot::createSurface will be reached to create the IOSurface object

IOSurface Creation

- Question: where is the created IOSurface stored
  - In IOSurfaceRootUserClient: Yes
    - But not all IOSurface is created by userland IOSurfaceRootUserClient
    - Also IOSurface can be looked up by other IOSurfaceRootUserClient objects
    - Needs to be stored globally
- Stored in IOCoreSurfaceRoot object
  - Global array with bitmap managed by IOCoreSurfaceRoot object
  - Expand if more IOSurface is created

IOSurface Creation

- IOSurface Id
  - Generated in function IOSurfaceRoot::alloc_surface_handle
  - Find the first available slot in the bitmap, the array index is the IOSurface Id
- The first IOSurface in iOS system should be 0?
  - Depends on the initial bitmap of the array

```c
__int64 __fastcall IOSurfaceRoot::alloc_surface_handle(IOSurfaceRoot *this, void *a2, unsigned int *a3)
{
  v_IOSurfaceHandleBitMapArrayCount = (unsigned int)this->i_IOSurfaceCurrentHandleCount >> 5;
  while ( 1 )
  {
    v_IOSurfaceHandleTotalCapability = this->i_IOSurfaceHandleTotalCapability;
    if ( v_IOSurfaceHandleBitMapArrayCount < v_IOSurfaceHandleTotalCapability >> 5 )
      break;
LABEL_6:
    if ( !(IOSurfaceRoot::alloc_handles(this) & 1) ) //if the bitmap is full
      return 0LL;
  }
  v8 = v_IOSurfaceHandleTotalCapability >> 5;
  v9 = 32 * v_IOSurfaceHandleBitMapArrayCount;
  while ( 1 )
  {
    v_bitMapValue = this->m_IOSurfaceHandleBitMap[v_IOSurfaceHandleBitMapArrayCount];
    if ( v_bitMapValue != -1 )
      break;
    ++v_IOSurfaceHandleBitMapArrayCount;
    v9 += 0x20;
    if ( v_IOSurfaceHandleBitMapArrayCount >= v8 )
      goto LABEL_6;
  }
  newHandleId = (unsigned int)(v12 + v9); // the IOSurface Id
  v14 = &this->m_IOSurfaceArray[(unsigned int)newHandleId];
  ...
  this->i_IOSurfaceCurrentHandleCount = newHandleId + 1;
  *v14 = (IOSurface *)a2;
  result = 1LL;
  this->m_IOSurfaceHandleBitMap[v_IOSurfaceHandleBitMapArrayCount] |= 1 << v12;
  *a3 = newHandleId;
  return result;
}
```

IOSurface Creation

- The first IOSurface Id
  - Initialized in IOSurfaceRoot::start
  - Initial capacity is set to 0x200 and the first DWORD of the bitmap is set to 1
- First IOSurface Id is 1
- IOSurface 0 does not exist

```
__int64 __fastcall IOSurfaceRoot::start(IOSurfaceRoot *this, IOService *a2)
{
  ...
      v2->m_ArrayIOSurfaceHandle = 0LL;
      v2->m_IOSurfaceHandleBitMap = 0LL;
      v2->i_IOSurfaceHandleTotalCapability = 0;
      v2->i_IOSurfaceCurrentHandleCount = 0;
      IOSurfaceRoot::alloc_handles(v2);
...
}
```

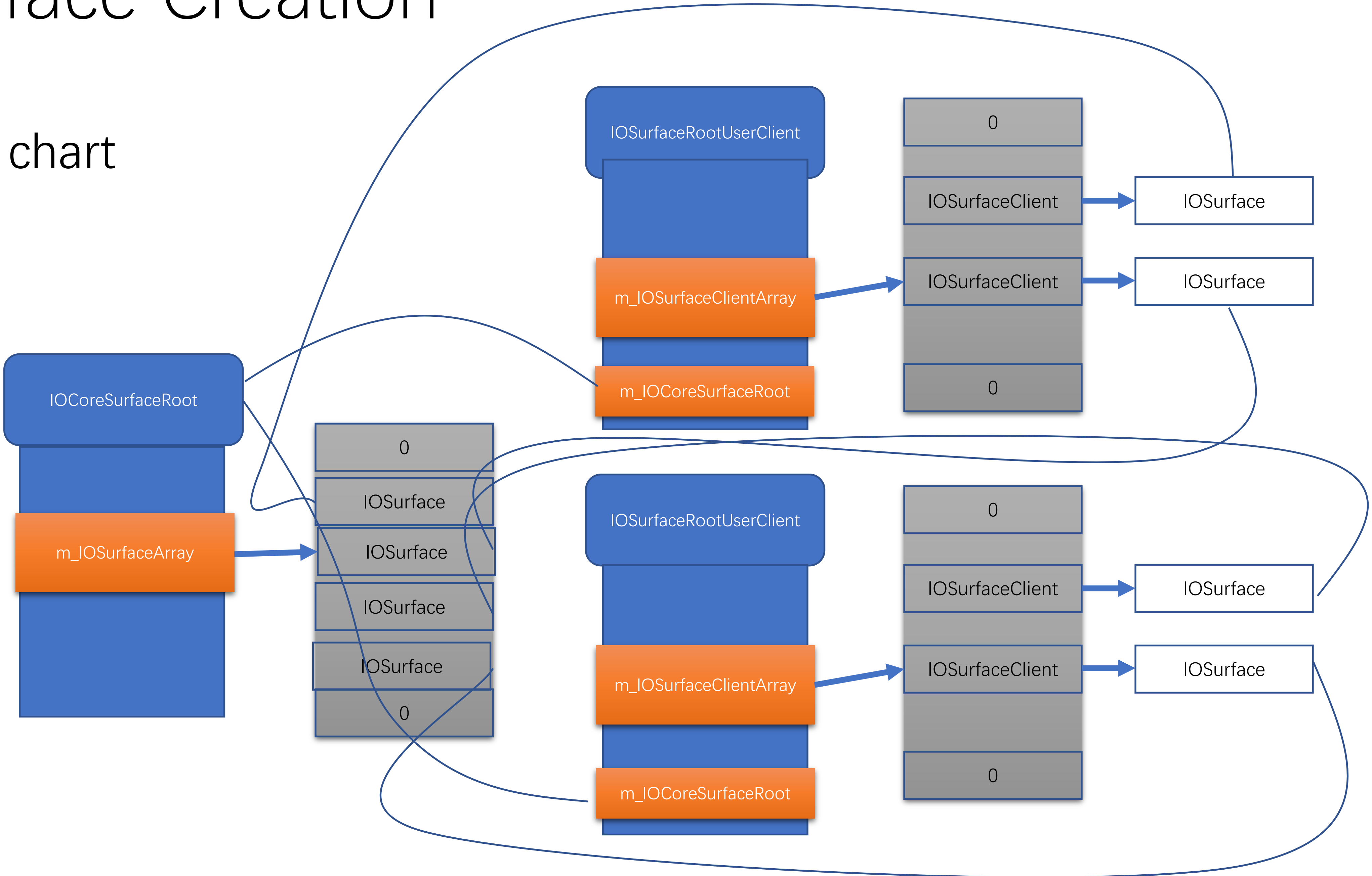
```
__int64 __fastcall IOSurfaceRoot::alloc_handles(IOSurfaceRoot *this)
{
  v2 = (unsigned int)this->i_IOSurfaceHandleTotalCapability;
  if ( (_DWORD)v2 )
  {
    if ( (unsigned int)v2 >> 14 )
      return 0LL;
    v3 = 2 * v2;
  }
  else
  {
    v3 = 0x200; //initial capacity is 0x200
  }
  v4 = this->m_IOSurfaceArray;
  v5 = this->m_IOSurfaceHandleBitMap;
  v6 = (v3 >> 3) + 8LL * v3;
  newIOSurfaceArray = (IOSurface **)IOMalloc(v6);
  this->m_IOSurfaceArray = newIOSurfaceArray;
  if ( newIOSurfaceArray )
  {
    this->i_IOSurfaceHandleTotalCapability = v3;
    this->m_IOSurfaceHandleBitMap = (int *)&newIOSurfaceArray[v3];
    memset(newIOSurfaceArray, 0, v6);
    if ( v4 )
    {
      memmove(this->m_IOSurfaceArray, v4, 8 * v2);
      memmove(this->m_IOSurfaceHandleBitMap, v5, (unsigned int)v2 >> 3);
      IOFree((__int64)v4, ((unsigned int)v2 >> 3) + 8 * v2);
      result = 1LL;
    }
    else
    {
      result = 1LL;
      *this->m_IOSurfaceHandleBitMap = 1; // the first 4 bytes of the bitmap is set to 1 by default
    }
  }
...
  return result;
}
```

IOSurface Creation

- IOSurfaceClient
  - When IOSurface is created by the user (Using IOSurfaceRootUserClient API), IOSurfaceClient is created and associated with IOSurface object
- IOSurfaceClientArray
  - An array to store IOSurfaceClient, array index is the IOSurface Id
  - Array element is assigned when either user creates IOSurface, or lookup an IOSurface
  - Each IOSurfaceRootUserClient owns an IOSurfaceClientArray

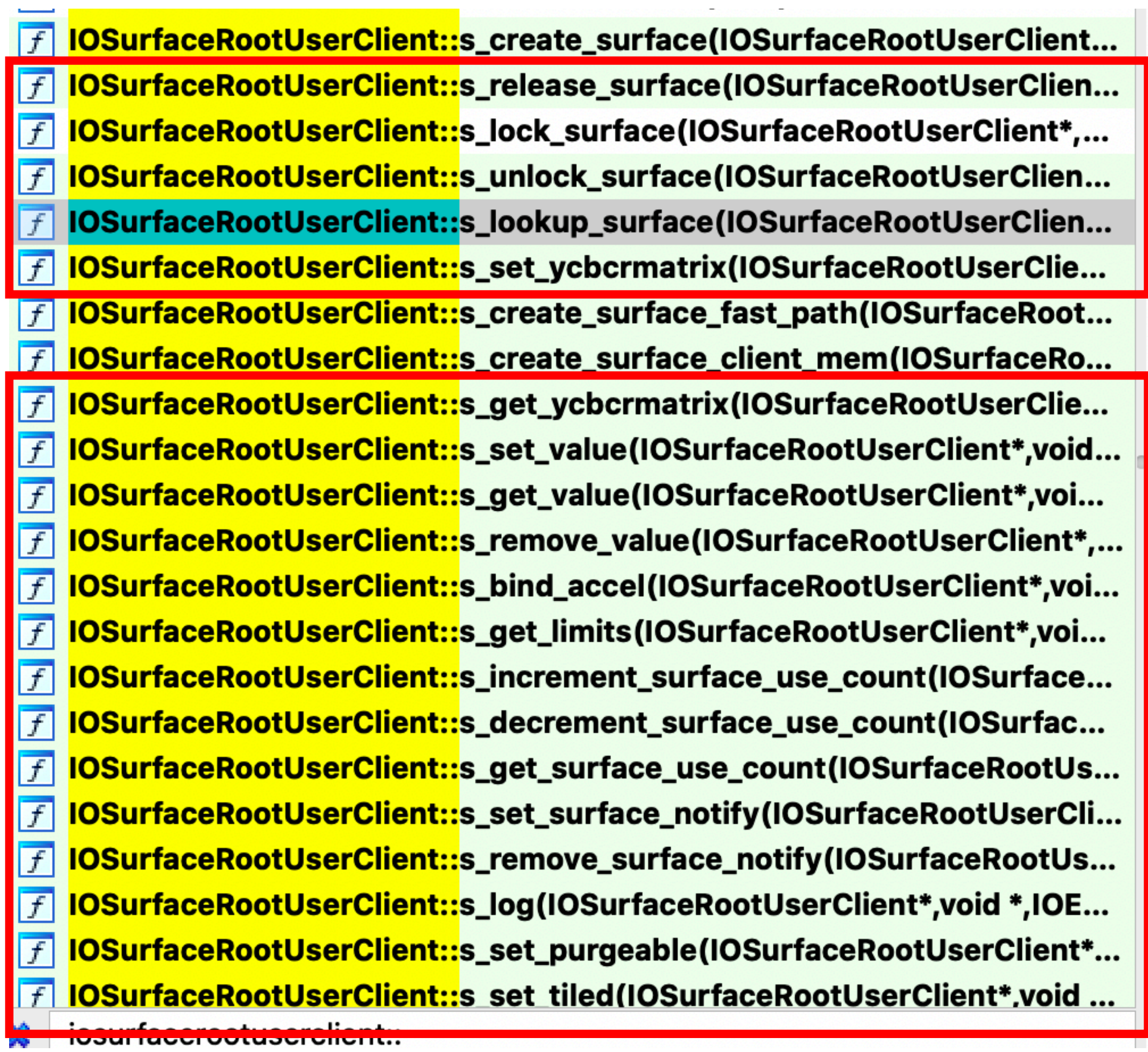
IOSurface Creation

- Overall chart



IOSurface API

- Kernel exposes several IOSurface APIs to user
- Most of them will require IOSurface Id as input (except for creation related APIs)



IOSurface API

- Directly dereference IOSurfaceClientArray[id], without checking id == 0 or not

- It will call IOSurfaceClient->m_IOSurface vtable method

```
__int64 __fastcall IOSurfaceRootUserClient::set_ycbcrmatrix
(IOSurfaceRootUserClient *this, unsigned int a2, unsigned int a3)
{
  lck_mtx_lock(this->m_lock);
  if ( v5->i_surfaceClientCapacity > v4 )
  {
    v7 = v5->m_IOSurfaceClientArrayPointer[v4];
    if ( v7 )
      v6 = IOSurfaceClient::setYCbCrMatrix(v7, v3);
  }
  lck_mtx_unlock(v5->m_lock);
  return v6;
}
```

```
__int64 __fastcall IOSurfaceClient::setYCbCrMatrix(__int64 this, unsigned int a2)
{
  return ((**(_QWORD **)(this + 0x40) + 0x120LL))(
           *(IOSurface **)(this + 0x40),
           a2);
}
```

IOSurface 0 exploitation

- Is it a problem?
  - Not a bug definitely, it is by design

- Good for exploitation
  - When we have heap overflow bugs
  - The first element in IOSurfaceClientArray can be overflowed to
    - By default, IOSurfaceClientArray is in kalloc.4096. But our buggy object can be in any zone.
  - Especially useful when the overflowed content is a c++ object
    - Type confusion

IOSurface 0 exploitation

- Given the first element in IOSurfaceClient Array is overflowed
- An easy way to probe which IOSurfaceClient Array has been overflowed
  - By calling IOSurface APIs with IOSurface Id 0

IOSurface 0 exploitation

- The type confusion
  - In normal case, function pointer *(**(IOSurfaceArray+0x40)+0xXXX) will be called
  - The offset 0xXXX varies depend on the APIs you call
  - IOSurface vtable is big

- If you can control your overflowed object + 0x40 pointer to a c++ object whose vtable is smaller than IOSurface
  - Can call the method out of object’ s vtable
  - Usually XXX::MetaClass vtable is put right after XXX vtable

```
DCQ __ZN9IOSurface19getMemoryDescriptorEP9IOService ; IOSurface::getMemoryDescriptor(IOServi
DCQ __ZN9IOSurface12getPlaneBaseEj ; IOSurface::getPlaneBase(uint)
DCQ __ZN9IOSurface14getPlaneOffsetEj ; IOSurface::getPlaneOffset(uint)
DCQ __ZN9IOSurface19getPlaneBytesPerRowEj ; IOSurface::getPlaneBytesPerRow(uint)
DCQ __ZN9IOSurface23getPlaneBytesPerElementEj ; IOSurface::getPlaneBytesPerElement(uint)
DCQ __ZN9IOSurface20getPlaneElementWidthEj ; IOSurface::getPlaneElementWidth(uint)
DCQ __ZN9IOSurface21getPlaneElementHeightEj ; IOSurface::getPlaneElementHeight(uint)
DCQ __ZN9IOSurface13getPlaneWidthEj ; IOSurface::getPlaneWidth(uint)
DCQ __ZN9IOSurface14getPlaneHeightEj ; IOSurface::getPlaneHeight(uint)
DCQ __ZN9IOSurface12getPlaneSizeEj ; IOSurface::getPlaneSize(uint)
DCQ __ZN9IOSurface14writeDebugInfoEP12OSDictionary ; IOSurface::writeDebugInfo(OSDictionary
DCQ __ZN9IOSurface14setYCbCrMatrixEj ; IOSurface::setYCbCrMatrix(uint)
DCQ __ZN9IOSurface14getYCbCrMatrixEPj ; IOSurface::getYCbCrMatrix(uint *)
DCQ __ZN9IOSurface8setValueEPK8OSSymbolPK15OSMetaClassBase ; IOSurface::setValue(OSSymbol co
DCQ __ZN9IOSurface8setValueEPK8OSStringPK15OSMetaClassBase ; IOSurface::setValue(OSString co
DCQ __ZN9IOSurface8setValueEPKcPK15OSMetaClassBase ; IOSurface::setValue(char const*,OSMetaC
DCQ __ZN9IOSurface8getValueEPK8OSSymbol ; IOSurface::getValue(OSSymbol const*)
DCQ __ZN9IOSurface8getValueEPK8OSString ; IOSurface::getValue(OSString const*)
DCQ __ZN9IOSurface8getValueEPKc ; IOSurface::getValue(char const*)
DCQ __ZN9IOSurface9copyValueEPK8OSSymbol ; IOSurface::copyValue(OSSymbol const*)
DCQ __ZN9IOSurface9copyValueEPK8OSString ; IOSurface::copyValue(OSString const*)
DCQ __ZN9IOSurface9copyValueEPKc ; IOSurface::copyValue(char const*)
DCQ __ZN9IOSurface11removeValueEPK8OSSymbol ; IOSurface::removeValue(OSSymbol const*)
DCQ __ZN9IOSurface11removeValueEPK8OSString ; IOSurface::removeValue(OSString const*)
DCQ __ZN9IOSurface11removeValueEPKc ; IOSurface::removeValue(char const*)
DCQ __ZN9IOSurface25deviceCacheForAcceleratorEPvjj ; IOSurface::deviceCacheForAccelerator(vo
DCQ __ZN9IOSurface30deviceCacheForAcceleratorPlaneEPvjjj ; IOSurface::deviceCacheForAccelera
DCQ __ZN9IOSurface17removeDeviceCacheEP20IOSurfaceDeviceCache ; IOSurface::removeDeviceCache
DCQ __ZN9IOSurface9bindAccelEjj ; IOSurface::bindAccel(uint,uint)
DCQ __ZN9IOSurface16bindAccelOnPlaneEjjj ; IOSurface::bindAccelOnPlane(uint,uint,uint)
DCQ __ZN9IOSurface20processorDataUpdatedEbb ; IOSurface::processorDataUpdated(bool,bool)
DCQ __ZN9IOSurface28processorDataUpdatedForPlaneEbbj ; IOSurface::processorDataUpdatedForPla
DCQ __ZN9IOSurface21setCurrentDeviceCacheEP20IOSurfaceDeviceCache ; IOSurface::setCurrentDev
DCQ __ZN9IOSurface28setCurrentDeviceCacheOnPlaneEP20IOSurfaceDeviceCachej ; IOSurface::setCu
DCQ __ZN9IOSurface19increment_use_countEv ; IOSurface::increment_use_count(void)
DCQ __ZN9IOSurface19decrement_use_countEv ; IOSurface::decrement_use_count(void)
DCQ __ZN9IOSurface13get_use_countEv ; IOSurface::get_use_count(void)
```

IOSurface 0 exploitation

- Info leak
  - Leak kernel .TEXT address: by calling OSMetaClass::getMetaClass
  - Leak heap address: by calling OSMetaClass::release or OSMetaClass::retain
    - X0 will be set as OSMetaClass object address and returned to userland(lower 4 bytes)
- Code execution
  - When first 8 bytes of the overflowed object can be controlled, code execution is not a problem. (try to call IOSurfaceRootUserClient::s_release _surface)

```
; __int64 __fastcall OSMetaClass::retain(OSMetaClass *__hidden this)
                EXPORT __ZNK11OSMetaClass6retainEv
__ZNK11OSMetaClass6retainEv             ; DATA XREF: __const:FFFFFFF007480060↑o
                                        ; __const:FFFFFFF0074800E0↑o ...
                RET
; End of function OSMetaClass::retain(void)

; =============== S U B R O U T I N E =======================================

; __int64 __fastcall OSMetaClass::release(OSMetaClass *__hidden this)
                EXPORT __ZNK11OSMetaClass7releaseEv
__ZNK11OSMetaClass7releaseEv            ; DATA XREF: __const:FFFFFFF007480068↑o
                                        ; __const:FFFFFFF0074800E8↑o ...
                RET
; End of function OSMetaClass::release(void)
```

Case study: IOSurface 0 exploitation

- Suppose we have a bug which can overflow an IOAccelResource2 object(or IOSurfaceMemoryRegion ☺) to the first element of an IOSurfaceClientArray
  - Actually in the past there are several such known bugs ☺

- We now overflow an IOAccelResource2 object

Case study: IOSurface 0 exploitation

- Next we call IOSurfaceRootUserClient::s_set_purgeable with IOSurface Id 0

- What happened?
  - *(**(IOAccelResource2 + 0x40) + 0x230) is called
  - IOAccelResource2 + 0x40 is initialized as an AGXMemoryMap object
  - (vtable of AGXMemoryMap + 0x230) is OSMetaClass::getMetaClass !

```
__int64 __fastcall IOSurfaceRootUserClient::set_purgeable(IOSurfaceRootUserClient *this, unsigned int a2, _
{
  unsigned int *v4; // x19
  unsigned int v5; // w20
  unsigned int v6; // w22
  IOSurfaceRootUserClient *v7; // x21
  __int64 v8; // x8
  __int64 v9; // x22
  __int64 v10; // x19
  __int64 result; // x0

  v4 = a4;
  v5 = a3;
  v6 = a2;
  v7 = this;
  lck_mtx_lock(this->m_lock);
  if ( v7->i_surfaceClientCapacity > v6 && (v8 = (__int64)v7->m_IOSurfaceClientArrayPointer[v6]) != 0 )
  {
    v9 = *(_QWORD *)(v8 + 0x40);
    (*(void (__cdecl **)(OSObject *))(*(_QWORD *)v9 + 32LL))((OSObject *)v9);
    lck_mtx_unlock(v7->m_lock);
    v10 = (*(_QWORD (__cdecl **)(IOSurface *, unsigned int, unsigned int *))(*(_QWORD *)v9 + 0x230LL))(
            (IOSurface *)v9,
            v5,
            v4);
    (*(void (__cdecl **)(IOSurface *))(*(_QWORD *)v9 + 120LL))((IOSurface *)v9);
    result = v10;
  }
  else
  {
    lck_mtx_unlock(v7->m_lock);
    result = 0xE00002C2LL;
  }
  return result;
}
```

Case study: IOSurface 0 exploitation

- Next we call IOSurfaceRootUserClient::s_set_ycbcrmatrix to leak a heap address.
  - If our bug is to overflow other objects other than IOAccelResource2, similar techniques can be used, but need to call another IOSurface API
- Finally, we spray the memory , free the IOAccelResource2, fill with heap address that we can control , and achieve code execution

IOSurface 0 exploitation summary

- Principle:
  - During IOSurface creation process, IOSurface 0 can not be created
  - When calling IOSurface API with IOSurface Id 0, iOS doesn’ t treat as illegal call.

- Exploit methodology:
  - We can utilize IOSurface 0 feature to probe which memory we has been successfully overflowed
  - Various objects can be used to confused as IOSurface object and because:
    - Most c++ objects’ vtable is smaller than IOSurface
    - IOSurface has quite some APIs in vtable which can be reached directly from userland
- We can easily leak kernel .TEXT address to bypass kASLR and leak kernel heap address to better spray the memory

- And… Type confusion exploitation is my favorite. Usually can be used to bypass most of the software CFG implementation

However…

PAC is introduced in 2018

- On devices with A12 and later
- C++ each function pointer in vtable is PACed with different context
  - Strongly protected
  - For more information, check my POC 2018 talk
- PAC has well mitigated IOSurface 0 exploitation
- To successfully exploit bugs on A12 or later, vtable call related exploitation techniques should be avoided.

Enhanced kASLR

- Before iOS 12.2, kslide is just 1 byte (256 possibilities), and only affect high bits of the lower 4 bytes of the address

- Also, once we obtain any .TEXT pointer, we can obtain kernel base just by simple AND operation (regardless of iOS version)

- Now, kslide is much more complex than before.
  - Example: slide: 0x0000000008c5c000

zone_require check

- Introduced in iOS 13
- Possibly the strongest protection to stop port related exploitation
- Enforced to protect all devices including pre-a12

zone_require check

- The check is to ensure the address is in correct zone
  - E.g during the process of copyout ports to userland, zone_require is performed to check if the port address is in "ipc ports" zone
- Previous common exploit involves cross-zone attack to gc a "ipc ports" zone and fill in with kalloc content to fake tfp0 ports
  - With zone_require, it is not possible now

```c
unsigned __int64 __fastcall zone_require(unsigned __int64 result, char *a2)
{
  ...
  v2 = qword_FFFFFFF00918F330 + 24 * (((result & 0xFFFFFFFFFFFFC000LL) - zone_map_min) >> 14);
  v3 = *(_WORD *)(v2 + 22) & 0x3FF;
  if ( (_DWORD)v3 == 0x3FF )
    v3 = *(_WORD *)(v2 - *(unsigned int *)(v2 + 16) + 22) & 0x3FF;
  if ( (char *)&unk_FFFFFFF00918F348 + 328 * (unsigned int)v3 != a2 )
    panic(
      "\"Address not in expected zone for zone_require check (addr: %p, zone: %s)\"",
      (const void *)result,
      *((const char **)&unk_FFFFFFF00918F348 + 41 * v3 + 36));
  ...
}
```

zone_require check

- “ipc ports” zone cannot be freed and filled with controlled kalloc content
  - We have to rely on better memory write ability before obtaining tfp0
    - To overwrite an existing “ipc ports” object to be a fake tfp0 port
- In iOS 13.2, more zone_require check is added
  - “task” zone is also checked in critical functions
  - Seems it is hard to overwrite an existing task structure to be fake tfp0 as it will cause issues to existing tasks
- But… If we have perfect arbitrary memory write ability, why we still need tfp0?
  - We just need better bugs. For example: CVE-2019-8605

GUARD_TYPE_MACH_PORT

- Some types of mach_port cannot copyout to another process
  - For example: io_connect

- Make out of sandbox exploitation harder
  - Rely on long ROP

Refcount 0 protection

- Before iOS 13, “overflow write 0” bug can be turned into UAF bug.
  - Exploited by Ian Beer’s empty_list exploit
- IPC port refcount can be overwritten to 0
- Then call some mach_port APIs to add port refcount to 1 and then decrease to 0 again to trigger the free, while we still have a userland port reference
  - E.g by calling mach_port_set_attributes

Refcount 0 protection

- Now port refcount cannot be 0 anymore

```
v25 = v23[1];
if ( (unsigned int)(v25 - 1) > 0x7FFFFFFD )
  panic("\"%s: reference count %u is invalid\\n\"", "io_reference", (unsigned int)v23[1]);
do
```

Sandbox profile hardening

- Before iOS 13, we can replace the structure pointer for sandbox collection profile, or platform profile
  - The structure pointer is malloc-ed

```
__const:000000000000C473F                    DCB     0
__const:00000000000C4740 _sandbox_collection DCQ _sandbox_collection_profile
__const:00000000000C4740                                          ; DATA XREF: _hook_policy_init:loc_AEFB4↑o
__const:00000000000C4740                                          ; _hook_policy_init+314↑r
__const:00000000000C4748 _sandbox_collection_profile DCQ 0        ; DATA XREF: _hook_policy_init+334↑w
__const:00000000000C4748                                          ; _hook_policy_init+3C4↑r ...
__const:00000000000C4750 qword_C4750      DCQ 0                   ; DATA XREF: _hook_policy_init+3B4↑r
__const:00000000000C4750                                          ; _hook_policy_init+470↑r ...
__const:00000000000C4758                  ALIGN 0x20
__const:00000000000C4760 qword_C4760      DCQ 0                   ; DATA XREF: _hook_policy_init+4A0↑r
__const:00000000000C4768 qword_C4768      DCQ 0                   ; DATA XREF: _hook_policy_init+3C0↑r
__const:00000000000C4768                                          ; _hook_policy_init+4A8↑r
__const:00000000000C4770 word_C4770       DCW 0                   ; DATA XREF: _hook_policy_init+33C↑w
__const:00000000000C4772 word_C4772       DCW 0                   ; DATA XREF: _hook_policy_init+4C8↑r
__const:00000000000C4774 byte_C4774       DCB 0                   ; DATA XREF: _hook_policy_init+4D0↑r
__const:00000000000C4775 byte_C4775       DCB 0                   ; DATA XREF: _hook_policy_init+38C↑w
```

- Now, the structure is in kernel .const initialized before KTRR is enabled, and protected by KTRR after

Trust cache hardening

- Before A12 is introduced, trust cache element can be added by tfp0
- In A12, trust cache is put into PPL layer and protected by APRR
- Once we bypass PAC in A12 and achieve arbitrary call, we can just call pmap_load_trust_cache to add trust cache

Trust cache hardening

- Since iOS 13, more operation is put into the ppl layer function
- We have to fully bypass APRR to add trust cache

Other mitigations

- Userland GOT read-only
- Kernel ROP/JOP gadget harder to find
- Etc.

Conclusion

- After A12 and iOS 13, iOS exploit becomes more and more difficult
  - Quite some nice exploits are killed, or being killed
  - Port related exploitation is much harder

- Bugs with better quality are required
  - For example, CVE-2019-8605

- Apple cannot stop exploits such as checkm8 (Luca will talk about this tomorrow)

Thank You