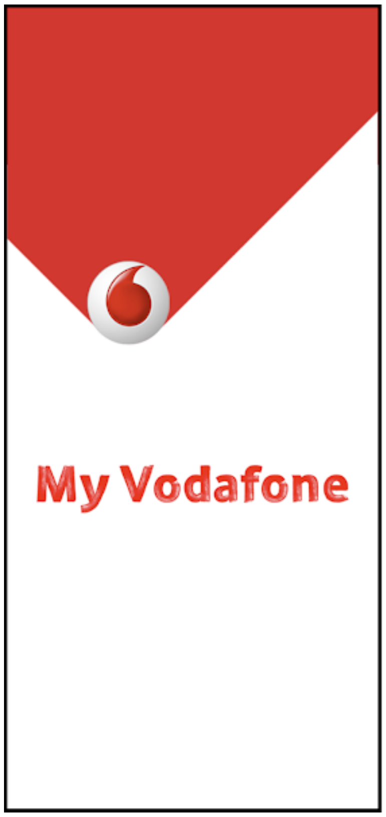
The curious tale of a fake Carrier.app

googleprojectzero.blogspot.com/2022/06/curious-case-carrier-app.html

Posted by Ian Beer, Google Project Zero

NOTE: This issue was CVE-2021-30983 was fixed in iOS 15.2 in December 2021.

Towards the end of 2021 Google's Threat Analysis Group (TAG) shared an iPhone app with me:



App splash screen showing the Vodafone carrier logo and the text "My Vodafone" (not the legitimate Vodadone app)

Although this looks like the real My Vodafone carrier app available in the App Store, it didn't come from the App Store and is not the real application from Vodafone. TAG suspects that a target receives a link to this app in an SMS, after the attacker asks the carrier to disable the target's mobile data connection. The SMS claims that in order to restore mobile data connectivity, the target must install the carrier app and includes a link to download and install this fake app.

This sideloading works because the app is signed with an enterprise certificate, which can be purchased for $299 via the Apple Enterprise developer program. This program allows an eligible enterprise to obtain an Apple-signed embedded.mobileprovision file with the ProvisionsAllDevices key set. An app signed with the developer certificate embedded within that mobileprovision file can be sideloaded on any iPhone, bypassing Apple's App Store review process. While we understand that the Enterprise developer program is designed for companies to push "trusted apps" to their staff's iOS devices, in this case, it appears that it was being used to sideload this fake carrier app.

In collaboration with Project Zero, TAG has published an additional post with more details around the targeting and the actor. The rest of this blogpost is dedicated to the technical analysis of the app and the exploits contained therein.

App structure

The app is broken up into multiple frameworks. InjectionKit.framework is a generic privilege escalation exploit wrapper, exposing the primitives you'd expect (kernel memory access, entitlement injection, amfid bypasses) as well as higher-level operations like app installation, file creation and so on.

Agent.framework is partially obfuscated but, as the name suggests, seems to be a basic agent able to find and exfiltrate interesting files from the device like the Whatsapp messages database.

Six privilege escalation exploits are bundled with this app. Five are well-known, publicly available N-day exploits for older iOS versions. The sixth is not like those others at all.

This blog post is the story of the last exploit and the month-long journey to understand it.

Something's missing? Or am I missing something?

Although all the exploits were different, five of them shared a common high-level structure. An initial phase where the kernel heap was manipulated to control object placement. Then the triggering of a kernel vulnerability followed by well-known steps to turn that into something useful, perhaps by disclosing kernel memory then building an arbitrary kernel memory write primitive.

The sixth exploit didn't have anything like that.

Perhaps it could be triggering a kernel logic bug like Linuz Henze's Fugu14 exploit, or a very bad memory safety issue which gave fairly direct kernel memory access. But neither of those seemed very plausible either. It looked, quite simply, like an iOS kernel exploit from a decade ago, except one which was first quite carefully checking that it was only running on an iPhone 12 or 13.

It contained log messages like:

```
  printf("Failed to prepare fake vtable: 0x%08x", ret);
```

which seemed to happen far earlier than the exploit could possibly have defeated mitigations like KASLR and PAC.

Shortly after that was this log message:

```
  printf("Waiting for R/W primitives...");
```

Why would you need to wait?

Then shortly after that:

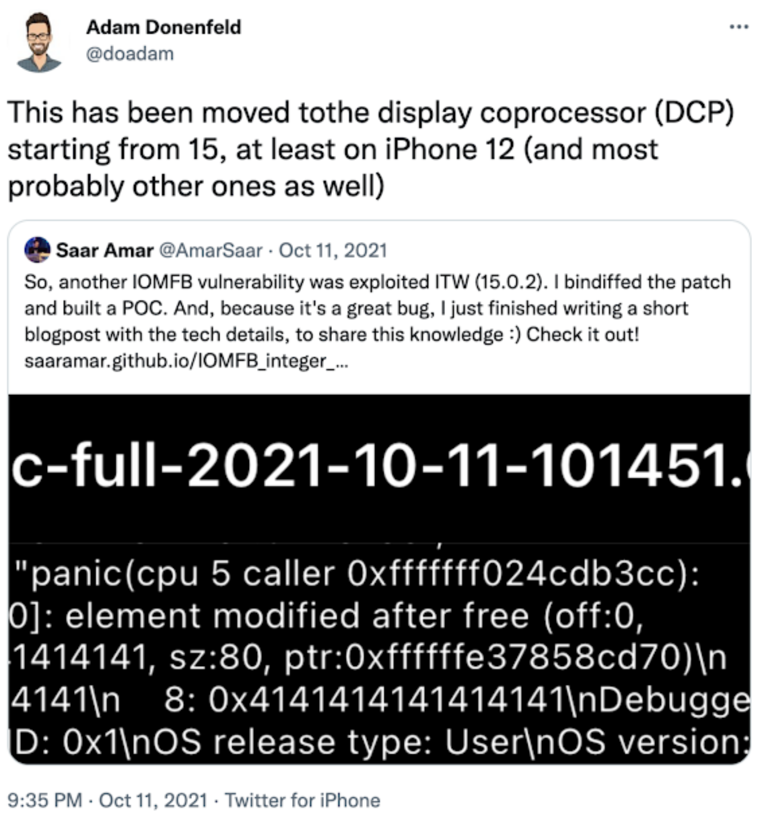
```
  printf("Memory read/write and callfunc primitives ready!");
```

Up to that point the exploit made only four IOConnectCallMethod calls and there were no other obvious attempts at heap manipulation. But there was another log message which started to shed some light:

```
  printf("Unexpected data read from DCP: 0x%08x", v49);
```

DCP?

In October 2021 Adam Donenfeld tweeted this:


Adam Donenfeld
@doadam

This has been moved tothe display coprocessor (DCP) starting from 15, at least on iPhone 12 (and most probably other ones as well)

Saar Amar @AmarSaar · Oct 11, 2021
So, another IOMFB vulnerability was exploited ITW (15.0.2). I bindiffed the patch and built a POC. And, because it's a great bug, I just finished writing a short blogpost with the tech details, to share this knowledge :) Check it out!
saaramar.github.io/IOMFB_integer_...

```
c-full-2021-10-11-101451.
"panic(cpu 5 caller 0xfffffff024cdb3cc):
0]: element modified after free (off:0,
1414141, sz:80, ptr:0xffffffe37858cd70)\n
4141\n    8: 0x4141414141414141\nDebugge
D: 0x1\nOS release type: User\nOS version:
```

9:35 PM · Oct 11, 2021 · Twitter for iPhone

DCP is the "Display Co-Processor" which ships with iPhone 12 and above and all M1 Macs.

There's little public information about the DCP; the most comprehensive comes from the Asahi linux project which is porting linux to M1 Macs. In their August 2021 and September 2021 updates they discussed their DCP reverse-engineering efforts and the open-source DCP client written by @alyssarzg. Asahi describe the DCP like this:

On most mobile SoCs, the display controller is just a piece of hardware with simple registers. While this is true on the M1 as well, Apple decided to give it a twist. They added a coprocessor to the display engine (called DCP), which runs its own firmware (initialized by the system bootloader), and moved most of the display driver into the coprocessor. But instead of doing it at a natural driver boundary… they took half of their macOS C++ driver, moved it into the DCP, and created a remote procedure call interface so that each half can call methods on C++ objects on the other CPU!

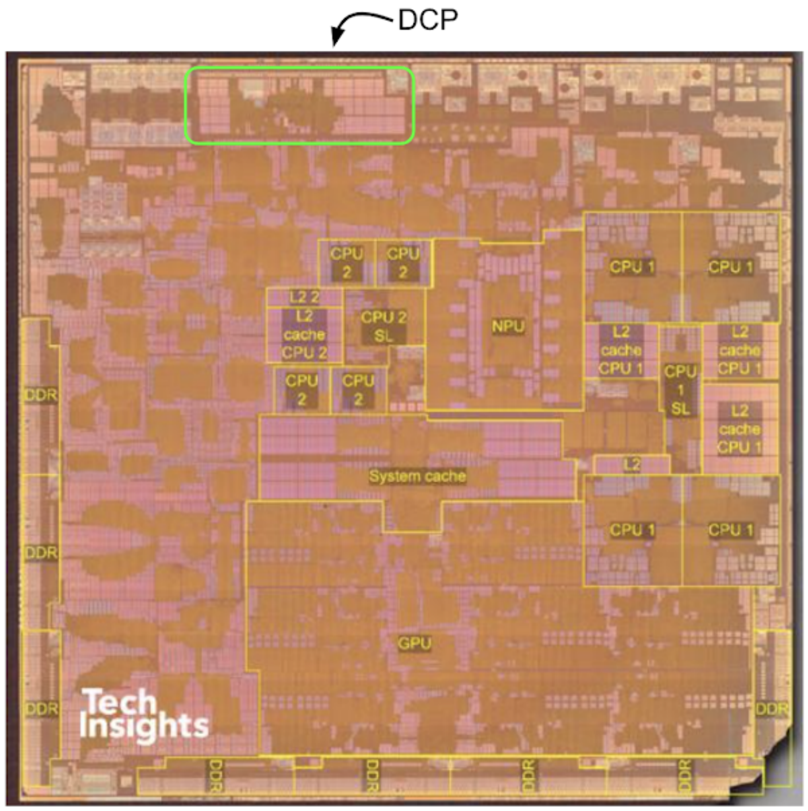
https://asahilinux.org/2021/08/progress-report-august-2021/

The Asahi linux project reverse-engineered the API to talk to the DCP but they are restricted to using Apple's DCP firmware (loaded by iBoot) - they can't use a custom DCP firmware. Consequently their documentation is limited to the DCP RPC API with few details of the DCP internals.

Setting the stage

Before diving into DCP internals it's worth stepping back a little. What even is a co-processor in a modern, highly integrated SoC (System-on-a-Chip) and what might the consequences of compromising it be?

Years ago a co-processor would likely have been a physically separate chip. Nowadays a large number of these co-processors are integrated along with their interconnects directly onto a single die, even if they remain fairly independent systems. We can see in this M1 die shot from Tech Insights that the CPU cores in the middle and right hand side take up only around 10% of the die:



M1 die-shot from techinsights.com with possible location of DCP added

https://www.techinsights.com/blog/two-new-apple-socs-two-market-events-apple-a14-and-m1

Companies like SystemPlus perform very thorough analysis of these dies. Based on their analysis the DCP is likely the rectangular region indicated on this M1 die. It takes up around the same amount of space as the four high-efficiency cores seen in the centre, though it seems to be mostly SRAM.

With just this low-resolution image it's not really possible to say much more about the functionality or capabilities of the DCP and what level of system access it has. To answer those questions we'll need to take a look at the firmware.

My kingdom for a .dSYM!

The first step is to get the DCP firmware image. iPhones (and now M1 macs) use .ipsw files for system images. An .ipsw is really just a .zip archive and the Firmware/ folder in the extracted .zip contains all the firmware for the co-processors, modems etc. The DCP firmware is this file:

```
Firmware/dcp/iphone13dcp.im4p
```

The im4p in this case is just a 25 byte header which we can discard:

```
$ dd if=iphone13dcp.im4p of=iphone13dcp bs=25 skip=1

$ file iphone13dcp

iphone13dcp: Mach-O 64-bit preload executable arm64
```

It's a Mach-O! Running nm -a to list all symbols shows that the binary has been fully stripped:

```
$ nm -a iphone13dcp
```

```
iphone13dcp: no symbols
```

Function names make understanding code significantly easier. From looking at the handful of strings in the exploit some of them looked like they might be referencing symbols in a DCP firmware image ("M3_CA_ResponseLUT read: 0x%08x" for example) so I thought perhaps there might be a DCP firmware image where the symbols hadn't been stripped.

Since the firmware images are distributed as .zip files and Apple's servers support range requests with a bit of python and the partialzip tool we can relatively easily and quickly get every beta and release DCP firmware. I checked over 300 distinct images; every single one was stripped.

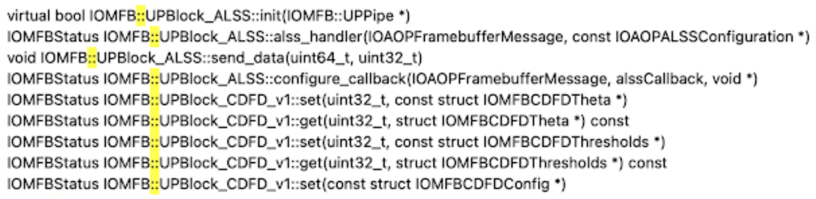
Guess we'll have to do this the hard way!

Day 1; Instruction 1

```
$ otool -h raw_fw/iphone13dcp

raw_fw/iphone13dcp:

Mach header

magic      cputype   cpusubtype caps filetype ncmds sizeofcmds flags

0xfeedfacf 0x100000C 0          0x00 5        5     2240       0x00000001
```

That cputype is plain arm64 (ArmV8) without pointer authentication support. The binary is fairly large (3.7MB) and IDA's autoanalysis detects over 7000 functions.

With any brand new binary I usually start with a brief look through the function names and the strings. The binary is stripped so there are no function name symbols but there are plenty of C++ function names as strings:


```
virtual bool IOMFB::UPBlock_ALSS::init(IOMFB::UPPipe *)
IOMFBStatus IOMFB::UPBlock_ALSS::alss_handler(IOAOPFramebufferMessage, const IOAOPALSSConfiguration *)
void IOMFB::UPBlock_ALSS::send_data(uint64_t, uint32_t)
IOMFBStatus IOMFB::UPBlock_ALSS::configure_callback(IOAOPFramebufferMessage, alssCallback, void *)
IOMFBStatus IOMFB::UPBlock_CDFD_v1::set(uint32_t, const struct IOMFBCDFDTheta *)
IOMFBStatus IOMFB::UPBlock_CDFD_v1::get(uint32_t, struct IOMFBCDFDTheta *) const
IOMFBStatus IOMFB::UPBlock_CDFD_v1::set(uint32_t, const struct IOMFBCDFDThresholds *)
IOMFBStatus IOMFB::UPBlock_CDFD_v1::get(uint32_t, struct IOMFBCDFDThresholds *) const
IOMFBStatus IOMFB::UPBlock_CDFD_v1::set(const struct IOMFBCDFDConfig *)
```

The cross-references to those strings look like this:

```
log(0x40000001LL,
    "UPBlock_ALSS.cpp",
    341,
    "%s: capture buffer exhausted, aborting capture\n",
    "void IOMFB::UPBlock_ALSS::send_data(uint64_t, uint32_t)");
```

This is almost certainly a logging macro which expands __FILE__, __LINE__ and __PRETTY_FUNCTION__. This allows us to start renaming functions and finding vtable pointers.

Object structure

From the Asahi linux blog posts we know that the DCP is using an Apple-proprietary RTOS called RTKit for which there is very little public information. There are some strings in the binary with the exact version:

```
ADD  X8, X8, #aLocalIphone13d@PAGEOFF ; "local-iphone13dcp.release"

ADD  X9, X9, #aRtkitIos182640@PAGEOFF ; "RTKit_iOS-1826.40.9.debug"
```

The code appears to be predominantly C++. There appear to be multiple C++ object hierarchies; those involved with this vulnerability look a bit like IOKit C++ objects. Their common base class looks like this:

```
struct __cppobj RTKIT_RC_RTTI_BASE
{
  RTKIT_RC_RTTI_BASE_vtbl *__vftable /*VFT*/;
  uint32_t refcnt;
  uint32_t typeid;
};
```

(These structure definitions are in the format IDA uses for C++-like objects)

The RTKit base class has a vtable pointer, a reference count and a four-byte Run Time Type Information (RTTI) field - a 4-byte ASCII identifier like BLHA, WOLO, MMAP, UNPI, OSST, OSBO and so on. These identifiers look a bit cryptic but they're quite descriptive once you figure them out (and I'll describe the relevant ones as we encounter them.)

The base type has the following associated vtable:

```
struct /*VFT*/ RTKIT_RC_RTTI_BASE_vtbl
{
  void (__cdecl *take_ref)(RTKIT_RC_RTTI_BASE *this);
  void (__cdecl *drop_ref)(RTKIT_RC_RTTI_BASE *this);
  void (__cdecl *take_global_type_ref)(RTKIT_RC_RTTI_BASE *this);
  void (__cdecl *drop_global_type_ref)(RTKIT_RC_RTTI_BASE *this);
  void (__cdecl *getClassName)(RTKIT_RC_RTTI_BASE *this);
  void (__cdecl *dtor_a)(RTKIT_RC_RTTI_BASE *this);
  void (__cdecl *unk)(RTKIT_RC_RTTI_BASE *this);
};
```

Exploit flow

The exploit running in the app starts by opening an IOKit user client for the AppleCLCD2 service. AppleCLCD seems to be the application processor of IOMobileFrameBuffer and AppleCLCD2 the DCP version.

The exploit only calls 3 different external method selectors on the AppleCLCD2 user client: 68, 78 and 79.

The one with the largest and most interesting-looking input is 78, which corresponds to this user client method in the kernel driver:

```
IOReturn
IOMobileFramebufferUserClient::s_set_block(
  IOMobileFramebufferUserClient *this,
  void *reference,
  IOExternalMethodArguments *args)
{
  const unsigned __int64 *extra_args;
  u8 *structureInput;
  structureInput = args->structureInput;
  if ( structureInput && args->scalarInputCount >= 2 )
  {
    if ( args->scalarInputCount == 2 )
      extra_args = 0LL;
    else
      extra_args = args->scalarInput + 2;
    return this->framebuffer_ap->set_block_dcp(
             this->task,
             args->scalarInput[0],
             args->scalarInput[1],
             extra_args,
             args->scalarInputCount - 2,
             structureInput,
             args->structureInputSize);
  } else {
    return 0xE00002C2;
  }
}
```

this unpacks the IOConnectCallMethod arguments and passes them to:

```
IOMobileFramebufferAP::set_block_dcp(
        IOMobileFramebufferAP *this,
        task *task,
        int first_scalar_input,
        int second_scalar_input,
        const unsigned __int64 *pointer_to_remaining_scalar_inputs,
        unsigned int scalar_input_count_minus_2,
        const unsigned __int8 *struct_input,
        unsigned __int64 struct_input_size)
```

This method uses some autogenerated code to serialise the external method arguments into a buffer like this:

```
arg_struct:
{
  struct task* task
  u64 scalar_input_0
  u64 scalar_input_1
  u64[] remaining_scalar_inputs
  u64 cntExtraScalars
  u8[] structInput
  u64 CntStructInput
}
```

which is then passed to UnifiedPipeline2::rpc along with a 4-byte ASCII method identifier ('A435' here):

```
UnifiedPipeline2::rpc(
      'A435',
      arg_struct,
      0x105Cu,
      &retval_buf,
      4u);
```

UnifiedPipeline2::rpc calls DCPLink::rpc which calls AppleDCPLinkService::rpc to perform one more level of serialisation which packs the method identifier and a "stream identifier" together with the arg_struct shown above.

AppleDCPLinkService::rpc then calls rpc_caller_gated to allocate space in a shared memory buffer, copy the buffer into there then signal to the DCP that a message is available.

Effectively the implementation of the IOMobileFramebuffer user client has been moved on to the DCP and the external method interface is now a proxy shim, via shared memory, to the actual implementations of the external methods which run on the DCP.

Exploit flow: the other side

The next challenge is to find where the messages start being processed on the DCP. Looking through the log strings there's a function which is clearly called rpc_callee_gated - quite likely that's the receive side of the function rpc_caller_gated we saw earlier.

rpc_callee_gated unpacks the wire format then has an enormous switch statement which maps all the 4-letter RPC codes to function pointers:

```
    switch ( rpc_id )
    {
      case 'A000':
        goto LABEL_146;
      case 'A001':
        handler_fptr = callback_handler_A001;
        break;
      case 'A002':
        handler_fptr = callback_handler_A002;
        break;
      case 'A003':
        handler_fptr = callback_handler_A003;
        break;
      case 'A004':
        handler_fptr = callback_handler_A004;
        break;
      case 'A005':
        handler_fptr = callback_handler_A005;
        break;
```

At the the bottom of this switch statement is the invocation of the callback handler:

```
ret = handler_fptr(
          meta,
          in_struct_ptr,
          in_struct_size,
          out_struct_ptr,
          out_struct_size);
```

in_struct_ptr points to a copy of the serialised IOConnectCallMethod arguments we saw being serialized earlier on the application processor:

```
arg_struct:
{
  struct task* task
  u64 scalar_input_0
  u64 scalar_input_1
  u64[] remaining_scalar_inputs
  u32 cntExtraScalars
  u8[] structInput
  u64 cntStructInput
}
```

The callback unpacks that buffer and calls a C++ virtual function:

```
unsigned int
callback_handler_A435(
  u8* meta,
  void *args,
  uint32_t args_size,
  void *out_struct_ptr,
  uint32_t out_struct_size
{
  int64 instance_id;
  uint64_t instance;
  int err;
  int retval;
  unsigned int result;
  instance_id = meta->instance_id;
  instance =
    global_instance_table[instance_id].IOMobileFramebufferType;
  if ( !instance ) {
    log_fatal(
      "IOMFB: %s: no instance for instance ID: %u\n",
      "static T *IOMFB::InstanceTracker::instance"
        "(IOMFB::InstanceTracker::tracked_entity_t, uint32_t)"
        " [T = IOMobileFramebuffer]",
      instance_id);
  }
  err = (instance-16)->vtable_0x378( // virtual call
         (instance-16),
         args->task,
         args->scalar_input_0,
         args->scalar_input_1,
         args->remaining_scalar_inputs,
         args->cntExtraScalars,
         args->structInput,
         args->cntStructInput);
  retval = convert_error(err);
  result = 0;
  *(_DWORD *)out_struct_ptr = retval;
  return result;
}
```

The challenge here is to figure out where that virtual call goes. The object is being looked up in a global table based on the instance id. We can't just set a breakpoint and whilst emulating the firmware is probably possible that would likely be a long project in itself. I took a hackier approach: we know that the vtable needs to be at least 0x380 bytes large so just go through all those vtables, decompile them and see if the prototypes look reasonable!

There's one clear match in the vtable for the UNPI type:

```
UNPI::set_block(
    UNPI* this,
    struct task* caller_task_ptr,
    unsigned int first_scalar_input,
    int second_scalar_input,
    int *remaining_scalar_inputs,
    uint32_t cnt_remaining_scalar_inputs,
    uint8_t *structure_input_buffer,
    uint64_t structure_input_size)
```

Here's my reversed implementation of UNPI::set_block

```
UNPI::set_block(
        UNPI* this,
        struct task* caller_task_ptr,
        unsigned int first_scalar_input,
        int second_scalar_input,
        int *remaining_scalar_inputs,
        uint32_t cnt_remaining_scalar_inputs,
        uint8_t *structure_input_buffer,
        uint64_t structure_input_size)
{
  struct block_handler_holder *holder;
  struct metadispatcher metadisp;
  if ( second_scalar_input )
    return 0x80000001LL;
  holder = this->UPPipeDCP_H13P->block_handler_holders;
  if ( !holder )
    return 0x8000000BLL;
  metadisp.address_of_some_zerofill_static_buffer = &unk_3B8D18;
  metadisp.handlers_holder = holder;
  metadisp.structure_input_buffer = structure_input_buffer;
  metadisp.structure_input_size = structure_input_size;
  metadisp.remaining_scalar_inputs = remaining_scalar_inputs;
  metadisp.cnt_remaining_sclar_input = cnt_remaining_scalar_inputs;
  metadisp.some_flags = 0x40000000LL;
  metadisp.dispatcher_fptr = a_dispatcher;
  metadisp.offset_of_something_which_looks_serialization_related = &off_1C1308;
  return metadispatch(holder, first_scalar_input, 1, caller_task_ptr, structure_input_buffer, &metadisp, 0);
}
```

This method wraps up the arguments into another structure I've called metadispatcher:

```
struct __attribute__((aligned(8))) metadispatcher
{
 uint64_t address_of_some_zerofill_static_buffer;
 uint64_t some_flags;
 __int64 (__fastcall *dispatcher_fptr)(struct metadispatcher *, struct BlockHandler *, __int64, _QWORD);
 uint64_t offset_of_something_which_looks_serialization_related;
 struct block_handler_holder *handlers_holder;
 uint64_t structure_input_buffer;
 uint64_t structure_input_size;
 uint64_t remaining_scalar_inputs;
 uint32_t cnt_remaining_sclar_input;
};
```

That metadispatcher object is then passed to this method:

```
return metadispatch(holder, first_scalar_input, 1, caller_task_ptr, structure_input_buffer, &metadisp, 0);
```

In there we reach this code:

```
  block_type_handler = lookup_a_handler_for_block_type_and_subtype(
                         a1,
                         first_scalar_input, // block_type
                         a3);                // subtype
```

The exploit calls this set_block external method twice, passing two different values for first_scalar_input, 7 and 19. Here we can see that those correspond to looking up two different block handler objects here.

The lookup function searches a linked list of block handler structures; the head of the list is stored at offset 0x1448 in the UPPipeDCP_H13P object and registered dynamically by a method I've named add_handler_for_block_type:

```
add_handler_for_block_type(struct block_handler_holder *handler_list,
                           struct BlockHandler *handler)
```

The logging code tells us that this is in a file called IOMFBBlockManager.cpp. IDA finds 44 cross-references to this method, indicating that there are probably that many different block handlers. The structure of each registered block handler is something like this:

```
struct __cppobj BlockHandler : RTKIT_RC_RTTI_BASE
{
  uint64_t field_16;
  struct handler_inner_types_entry *inner_types_array;
  uint32_t n_inner_types_array_entries;
  uint32_t field_36;
  uint8_t can_run_without_commandgate;
  uint32_t block_type;
  uint64_t list_link;
  uint64_t list_other_link;
  uint32_t some_other_type_field;
  uint32_t some_other_type_field2;
  uint32_t expected_structure_io_size;
  uint32_t field_76;
  uint64_t getBlock_Impl;
  uint64_t setBlock_Impl;
  uint64_t field_96;
  uint64_t back_ptr_to_UPPipeDCP_H13P;
};
```

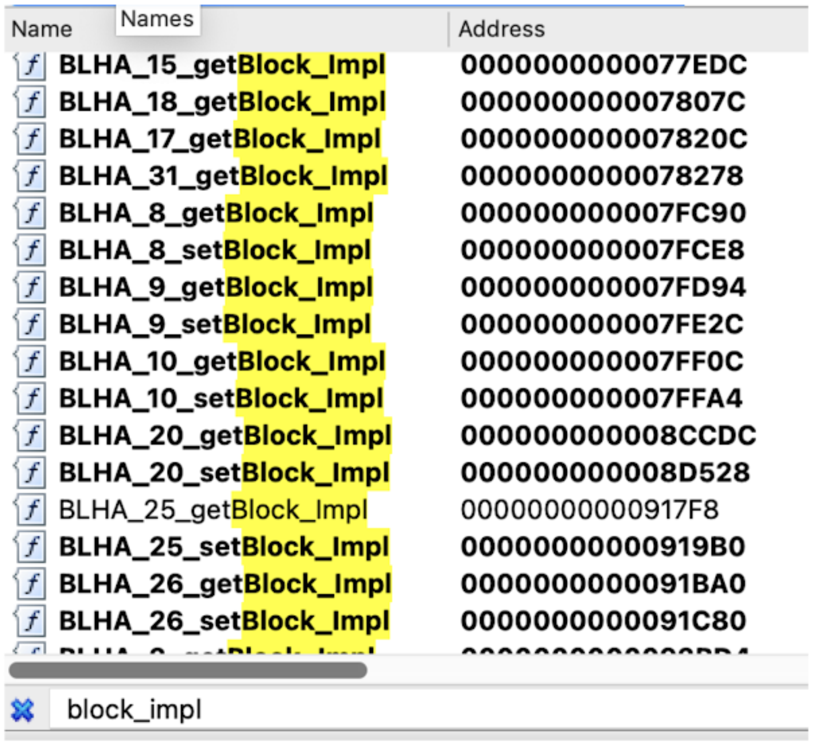
The RTTI type is BLHA (BLock HAndler.)

For example, here's the codepath which builds and registers block handler type 24:

```
BLHA_24 = (struct BlockHandler *)CXXnew(112LL);
BLHA_24->__vftable = (BlockHandler_vtbl *)BLHA_super_vtable;
BLHA_24->block_type = 24;
BLHA_24->refcnt = 1;
BLHA_24->can_run_without_commandgate = 0;
BLHA_24->some_other_type_field = 0LL;
BLHA_24->expected_structure_io_size = 0xD20;
typeid = typeid_BLHA();
BLHA_24->typeid = typeid;
modify_typeid_ref(typeid, 1);
BLHA_24->__vftable = vtable_BLHA_subclass_type_24;
BLHA_24->inner_types_array = 0LL;
BLHA_24->n_inner_types_array_entries = 0;
BLHA_24->getBlock_Impl = BLHA_24_getBlock_Impl;
BLHA_24->setBlock_Impl = BLHA_24_setBlock_Impl;
BLHA_24->field_96 = 0LL;
BLHA_24->back_ptr_to_UPPipeDCP_H13P = a1;
add_handler_for_block_type(list_holder, BLHA_24);
```

Each block handler optionally has getBlock_Impl and setBlock_Impl function pointers which appear to implement the actual setting and getting operations.

We can go through all the callsites which add block handlers; tell IDA the type of the arguments and name all the getBlock and setBlock implementations:



Name	Address
BLHA_15_getBlock_Impl	0000000000077EDC
BLHA_18_getBlock_Impl	000000000007807C
BLHA_17_getBlock_Impl	000000000007820C
BLHA_31_getBlock_Impl	0000000000078278
BLHA_8_getBlock_Impl	000000000007FC90
BLHA_8_setBlock_Impl	000000000007FCE8
BLHA_9_getBlock_Impl	000000000007FD94
BLHA_9_setBlock_Impl	000000000007FE2C
BLHA_10_getBlock_Impl	000000000007FF0C
BLHA_10_setBlock_Impl	000000000007FFA4
BLHA_20_getBlock_Impl	000000000008CCDC
BLHA_20_setBlock_Impl	000000000008D528
BLHA_25_getBlock_Impl	00000000000917F8
BLHA_25_setBlock_Impl	00000000000919B0
BLHA_26_getBlock_Impl	0000000000091BA0
BLHA_26_setBlock_Impl	0000000000091C80

You can perhaps see where this is going: that's looking like really quite a lot of reachable attack surface! Each of those setBlock_Impl functions is reachable by passing a different value for the first scalar argument to IOConnectCallMethod 78.

There's a little bit more reversing though to figure out how exactly to get controlled bytes to those setBlock_Impl functions:

Memory Mapping

The raw "block" input to each of those setBlock_Impl methods isn't passed inline in the IOConnectCallMethod structure input. There's another level of indirection: each individual block handler structure has an array of supported "subtypes" which contains metadata detailing where to find the (userspace) pointer to that subtype's input data in the IOConnectCallMethod structure input. The first dword in the structure input is the id of this subtype - in this case for the block handler type 19 the metadata array has a single entry:

```
<2, 0, 0x5F8, 0x600>
```

The first value (2) is the subtype id and 0x5f8 and 0x600 tell the DCP from what offset in the structure input data to read a pointer and size from. The DCP then requests a memory mapping from the AP for that memory from the calling task:

```
  return wrap_MemoryDescriptor::withAddressRange(
    *(void*)(structure_input_buffer + addr_offset),
    *(unsigned int *)(structure_input_buffer + size_offset),
    caller_task_ptr);
```

We saw earlier that the AP sends the DCP the struct task pointer of the calling task; when the DCP requests a memory mapping from a user task it sends those raw task struct pointers back to the AP such that the kernel can perform the mapping from the correct task. The memory mapping is abstracted as an MDES object on the DCP side; the implementation of the mapping involves the DCP making an RPC to the AP:

```
  make_link_call('D453', &req, 0x20, &resp, 0x14);
```

which corresponds to a call to this method on the AP side:

```
  IOMFB::MemDescRelay::withAddressRange(unsigned long long, unsigned long long, unsigned int, task*, unsigned long*, unsigned long long*)
```

The DCP calls ::prepare and ::map on the returned MDES object (exactly like an IOMemoryDescriptor object in IOKit), gets the mapped pointer and size to pass via a few final levels of indirection to the block handler:

```
  a_descriptor_with_controlled_stuff->dispatcher_fptr(
    a_descriptor_with_controlled_stuff,
    block_type_handler,
    important_ptr,
    important_size);
```

where the dispatcher_fptr looks like this:

```
a_dispatcher(
    struct metadispatcher *disp,
    struct BlockHandler *block_handler,
    __int64 controlled_ptr,
    unsigned int controlled_size)
{
  return block_handler->BlockHandler_setBlock(
      block_handler,
      disp->structure_input_buffer,
      disp->structure_input_size,
      disp->remaining_scalar_inputs,
      disp->cnt_remaining_sclar_input,
      disp->handlers_holder->gate,
      controlled_ptr,
      controlled_size);
}
```

You can see here just how useful it is to keep making structure definitions while reversing; there are so many levels of indirection that it's pretty much impossible to keep it all in your head.

BlockHandler_setBlock is a virtual method on BLHA. This is the implementation for BLHA 19:

```
BlockHandler19::setBlock(
  struct BlockHandler *this,
  void *structure_input_buffer,
  int64 structure_input_size,
  int64 *remaining_scalar_inputs,
  unsigned int cnt_remaining_scalar_inputs,
  struct CommandGate *gate,
  void* mapped_mdesc_ptr,
  unsigned int mapped_mdesc_length)
```

This uses a Command Gate (GATI) object (like a call gate in IOKit to serialise calls) to finally get close to actually calling the setBlock_Impl function.

We need to reverse the gate_context structure to follow the controlled data through the gate:

```
struct __attribute__((aligned(8))) gate_context
{
 struct BlockHandler *the_target_this;
 uint64_t structure_input_buffer;
 void *remaining_scalar_inputs;
 uint32_t cnt_remaining_scalar_inputs;
 uint32_t field_28;
 uint64_t controlled_ptr;
 uint32_t controlled_length;
};
```

The callgate object uses that context object to finally call the BLHA setBlock handler:

```
callback_used_by_callgate_in_block_19_setBlock(
  struct UnifiedPipeline *parent_pipeline,
  struct gate_context *context,
  int64 a3,
  int64 a4,
  int64 a5)
{
  return context->the_target_this->setBlock_Impl)(
         context->the_target_this->back_ptr_to_UPPipeDCP_H13P,
         context->structure_input_buffer,
         context->remaining_scalar_inputs,
         context->cnt_remaining_scalar_inputs,
         context->controlled_ptr,
         context->controlled_length);
}
```

SetBlock_Impl

And finally we've made it through the whole callstack following the controlled data from IOConnectCallMethod in userspace on the AP to the setBlock_Impl methods on the DCP!

The prototype of the setBlock_Impl methods looks like this:

```
setBlock_Impl(struct UPPipeDCP_H13P *pipe_parent,
              void *structure_input_buffer,
              int *remaining_scalar_inputs,
              int cnt_remaining_scalar_inputs,
              void* ptr_via_memdesc,
              unsigned int len_of_memdesc_mapped_buf)
```

The exploit calls two setBlock_Impl methods; 7 and 19. 7 is fairly simple and seems to just be used to put controlled data in a known location. 19 is the buggy one. From the log strings we can tell that block type 19 handler is implemented in a file called UniformityCompensator.cpp.

Uniformity Compensation is a way to correct for inconsistencies in brightness and colour reproduction across a display panel. Block type 19 sets and gets a data structure containing this correction information. The setBlock_Impl method calls UniformityCompensator::set and reaches the following code snippet where controlled_size is a fully-controlled u32 value read from the structure input and indirect_buffer_ptr points to the mapped buffer, the contents of which are also controlled:

```
uint8_t* pages = compensator->inline_buffer; // +0x24
for (int pg_cnt = 0; pg_cnt < 3; pg_cnt++) {
  uint8_t* this_page = pages;
  for (int i = 0; i < controlled_size; i++) {
    memcpy(this_page, indirect_buffer_ptr, 4 * controlled_size);
    indirect_buffer_ptr += 4 * controlled_size;
    this_page += 0x100;
  }
  pages += 0x4000;
}
```

There's a distinct lack of bounds checking on controlled_size. Based on the structure of the code it looks like it should be restricted to be less than or equal to 64 (as that would result in the input being completely copied to the output buffer.) The compensator->inline_buffer buffer is inline in the compensator object. The structure of the code makes it look that that buffer is probably 0xc000 (three 16k pages) large. To verify this we need to find the allocation site of this compensator object.

It's read from the pipe_parent object and we know that at this point pipe_parent is a UPPipeDCP_H13P object.

There's only one write to that field, here in UPPipeDCP_H13P::setup_tunables_base_target:

```
compensator = CXXnew(0xC608LL);
...
this->compensator = compensator;
```

The compensator object is a 0xc608 byte allocation; the 0xc000 sized buffer starts at offset 0x24 so the allocation has enough space for 0xc608-0x24=0xC5E4 bytes before corrupting neighbouring objects.

The structure input provided by the exploit for the block handler 19 setBlock call looks like this:

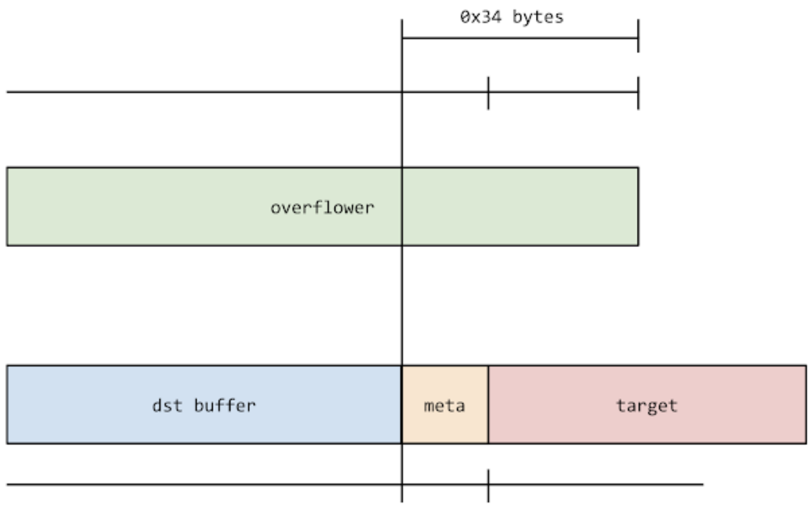
```
struct_input_for_block_handler_19[0x5F4] = 70; // controlled_size
struct_input_for_block_handler_19[0x5F8] = address;
struct_input_for_block_handler_19[0x600] = a_size;
```

This leads to a value of 70 (0x46) for controlled_size in the UniformityCompensator::set snippet shown earlier. (0x5f8 and 0x600 correspond to the offsets we saw earlier in the subtype's table: <2, 0, 0x5F8, 0x600>)

The inner loop increments the destination pointer by 0x100 each iteration so 0x46 loop iterations will write 0x4618 bytes.

The outer loop writes to three subsequent 0x4000 byte blocks so the third (final) iteration starts writing at 0x24 + 0x8000 and writes a total of 0x4618 bytes, meaning the object would need to be 0xC63C bytes; but we can see that it's only 0xc608, meaning that it will overflow the allocation size by 0x34 bytes. The RTKit malloc implementation looks like it adds 8 bytes of metadata to each allocation so the next object starts at 0xc610.

How much input is consumed? The input is fully consumed with no "rewinding" so it's 3*0x46*0x46*4 = 0xe5b0 bytes. Working backwards from the end of that buffer we know that the final 0x34 bytes of it go off the end of the 0xc608 allocation which means +0xe57c in the input buffer will be the first byte which corrupts the 8 metadata bytes and +0x8584 will be the first byte to corrupt the next object:



This matches up exactly with the overflow object which the exploit builds:

```
v24 = address + 0xE584;
v25 = *(_DWORD *)&v54[48];
v26 = *(_OWORD *)&v54[32];
v27 = *(_OWORD *)&v54[16];
*(_OWORD *)(address + 0xE584) = *(_OWORD *)v54;
*(_OWORD *)(v24 + 16) = v27;
*(_OWORD *)(v24 + 32) = v26;
*(_DWORD *)(v24 + 48) = v25;
```

The destination object seems to be allocated very early and the DCP RTKit environment appears to be very deterministic with no ASLR. Almost certainly they are attempting to corrupt a neighbouring C++ object with a fake vtable pointer.

Unfortunately for our analysis the trail goes cold here and we can't fully recreate the rest of the exploit. The bytes for the fake DCP C++ object are read from a file in the app's temporary directory (base64 encoded inside a JSON file under the exploit_struct_offsets key) and I don't have a copy of that file. But based on the flow of the rest of the exploit it's pretty clear what happens next:

sudo make me a DART mapping

The DCP, like other coprocessors on iPhone, sits behind a DART (Device Address Resolution Table.) This is like an SMMU (IOMMU in the x86 world) which forces an extra layer of physical address lookup between the DCP and physical memory. DART was covered in great detail in Gal Beniamini's Over The Air - Vol. 2, Pt. 3 blog post.

The DCP clearly needs to access lots of buffers owned by userspace tasks as well as memory managed by the kernel. To do this the DCP makes RPC calls back to the AP which modifies the DART entries accordingly. This appears to be exactly what the DCP exploit does: the D45X family of DCP->AP RPC methods appear to expose an interface for requesting arbitrary physical as well as virtual addresses to be mapped into the DCP DART.

The fake C++ object is most likely a stub which makes such calls on behalf of the exploit, allowing the exploit to read and write kernel memory.

Conclusions

Segmentation and isolation are in general a positive thing when it comes to security. However, splitting up an existing system into separate, intercommunicating parts can end up exposing unexpected code in unexpected ways.

We've had discussions within Project Zero about whether this DCP vulnerability is interesting at all. After all, if the UniformityCompensator code was going to be running on the Application Processors anyway then the Display Co-Processor didn't really introduce or cause this bug.

Whilst that's true, it's also the case that the DCP certainly made exploitation of this bug significantly easier and more reliable than it would have been on the AP. Apple has invested heavily in memory corruption mitigations over the last few years, so moving an attack surface from a "mitigation heavy" environment to a "mitigation light" one is a regression in that sense.

Another perspective is that the DCP just isn't isolated enough; perhaps the intention was to try to isolate the code on the DCP such that even if it's compromised it's limited in the effect it could have on the entire system. For example, there might be models where the DCP to AP RPC interface is much more restricted.

But again there's a tradeoff: the more restrictive the RPC API, the more the DCP code has to be refactored - a significant investment. Currently, the codebase relies on being able to map arbitrary memory and the API involves passing userspace pointers back and forth.

I've discussed in recent posts how attackers tend to be ahead of the curve. As the curve slowly shifts towards memory corruption exploitation getting more expensive, attackers are likely shifting too. We saw that in the logic-bug sandbox escape used by NSO and we see that here in this memory-corruption-based privilege escalation that side-stepped kernel mitigations by corrupting memory on a co-processor instead. Both are quite likely to continue working in some form in a post-memory tagging world. Both reveal the stunning depth of attack surface available to the motivated attacker. And both show that defensive security research still has a lot of work to do.