PANGU TEAM

Search

Don’t place a port in shared memory

int80 November 14, 2020 No Comments

Motivation

Today I read a very interesting blogpost by Brandon Azad from Google Project 0 (https://googleprojectzero.blogspot.com/2020/11/oops-i-missed-it-again.html) , which disclosed an incorrect bound-check in the function `H11ANEInDirectPathClient::externalMethod` , allowing to invoke (more privileged) `H11ANEInUserClient` ’s external methods in the context of (less privileged) `H11ANEInDirectPathClient` , eventually leading to type confusions or other security issues.

It is very likely a copy-paste bug. Similar to Brandon, I also missed the vulnerability! It’s kind of unacceptable. Following Brandon’s blogpost, we will share another bug in the `H11ANEIn` driver.

H11ANEIn

The `H11ANEIn` driver was first introduced in iOS 12 for A12 devices (e.g., iPhone XS and XS Max). I think that “ANE” stands for “Apple Neural Engine”.

The `H11ANEIn` driver offers two types of IOUserClients, `H11ANEInDirectPathClient` and `H11ANEInUserClient` . In particular, `H11ANEInUserClient` is more powerful than `H11ANEInDirectPathClient` , as `H11ANEInUserClient` has more external methods than `H11ANEInDirectPathClient` . However, creating `H11ANEInUserClient` requires a special

entitlement, `com.apple.ane.iokit-user-access`. Only a few executables on iOS have the entitlements, such as `aned` and `mediaserverd`.

H11ANEInDirectPathClient

Fortunately, the container sandbox allows to open the `H11ANEInDirectPathClient` IOUserClient. So third-party apps can open and communicate with `H11ANEInDirectPathClient` IOUserClients. Looking at the function `H11ANEInDirectPathClient::externalMethod`, (Ok, please forget the incorrect bound-check here), we can find that `H11ANEInDirectPathClient` actually has the following interfaces:

- _ANE_DeviceOpen
- _ANE_DeviceClose
- _ANE_ProgramSendRequest

Now let’s take a look at the interface `_ANE_ProgramSendRequest`. Function `_ANE_ProgramSendRequest` accepts a structural input with length 16.

```
DCQ H11ANEInDirectPathClient___ANE_ProgramSendRequest
DCD 0
DCD 0x10
DCD 0
DCD 0
```

In fact, at the very beginning of function _ANE_ProgramSendRequest, it would treat the structural input as two uint64 values, and pass them into `IOMemoryDescriptor::withAddressRange()`;

```
__int64 __fastcall H11ANEInDirectPathClient::_ANE_ProgramSendRequest(__int64 H11ANEInDirectPathClient, __int64 a2, IOExternalMethodArguments *a3)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v3 = 0xE00002BDLL;
  structureInput = (uint64_t)a3->structureInput;
  if ( *(_QWORD *)(structureInput + 8) != 2704LL )
    return 0xE00002C2LL;
  IOMemoryDescriptor = IOMemoryDescriptor::withAddressRange(
                         *(_QWORD *)structureInput,
                         2704LL,
                         3LL,
                         *(_QWORD *)(H11ANEInDirectPathClient + 232));
```

Clearly, the first value is a userspace address, and the second value is the length of the userspace buffer. After creating the `IOMemoryDescriptor`, `_ANE_ProgramSendRequest` continues to call the `prepare` and `map` functions, and map the memory into the kernel space. In other words, `_ANE_ProgramSendRequest` creates a shared memory now.

```
if ( IOMemoryDescriptor )
{
  if ( !(*(unsigned int (__fastcall **)(__int64, __int64))(*(_QWORD *)IOMemoryDescriptor + 216LL))(// prepare
          IOMemoryDescriptor,
          3LL) )
  {
    IOMemoryMap = (*(__int64 (__fastcall **)(__int64, _QWORD))(*(_QWORD *)IOMemoryDescriptor + 232LL))(
                    IOMemoryDescriptor,
                    0LL);                       // map
    if ( IOMemoryMap )
    {
      sharedmem = (*(__int64 (**)(void))(*(_QWORD *)IOMemoryMap + 120LL))();// getvirtualaddress
      if ( sharedmem )
```

More importantly, the `_ANE_ProgramSendRequest` interface should be called through the `IOConnectCallAsyncMethod` function. `IOConnectCallAsyncMethod` allows the IOKit extension

to asynchronously process requests from the users-pace. By supplying a notification/wakeup port, a user-space program will receive a notification message when the request is done. It implies that the kernel or the extension should store the wakeup port somewhere so that the kernel or the extension can send the notification later.

The vulnerability and exploit

The vulnerability is that, `_ANE_ProgramSendRequest` records the notification context, including the port pointer, in the shared memory!

```
if ( sharedmem )
{
  v13 = sharedmem + 2616;
  v14 = (uint64_t)a3->asyncReference;
  portptr = *(_OWORD *)v14;
  v16 = *(_OWORD *)(v14 + 16);
  v17 = *(_OWORD *)(v14 + 48);
  *(_OWORD *)(v13 + 32) = *(_OWORD *)(v14 + 32);
  *(_OWORD *)(v13 + 48) = v17;
  *(_OWORD *)v13 = portptr;
  *(_OWORD *)(v13 + 16) = v16;
  IOUserClient::setAsyncReference64(
    (__int64 *)v13,
    *(_QWORD *)&a3->asyncWakePort,
    *(_QWORD *)(sharedmem + 2680),
    *(_QWORD *)(sharedmem + 2688));
  if ( *(_BYTE *)(H11ANEInDirectPathClient + 225) )
    v3 = H11ANEIn::ANE_ProgramSendRequest(
           *(_QWORD *)(H11ANEInDirectPathClient + 216),
           sharedmem,
           H11ANEInDirectPathClient,
```

As we can see above, before invoking `H11ANEIn::ANE_ProgramSendRequest`, the wakeup port is stored at the shared memory at offset 2616 (note that the offsets vary across iOS kernel versions).

It’s quite straightforward to convert the vulnerability into an info leak. We can easily get a port pointer in the shared memory after we trigger the execution of `H11ANEInDirectPathClient::_ANE_ProgramSendRequest`.

Beyond the info leak, you may have already realized that there are a lot of chances to further exploit the vulnerability, especially for the readers who can still remember the vulnerability exploited by the very first Pangu jailbreak tool (for iOS 7.4.1).

Just a quick recap. At that time, `IOSharedDataQueue` stores a mach message header in shared memory. We achieve the kernel exploit by crafting the mach message header (https://googleprojectzero.blogspot.com/2018/10/deja-xnu.html).

Similar here, we can modify the port pointer in the shared memory so that when a notification is to be sent to a fake (arbitrary) port. This blog won’t go to details of the exploit. There are so many different ways to accomplish a kernel exploit when you have a full control to a port pointer.

Things I haven’t mentioned

It wouldn’t be that easy to reproduce the vulnerability, because you will find that `H11ANEIn::ANE_ProgramSendRequest` will not deliver a notification to the user-space app unless you have a properly-filled request.

First, `H11ANEInDirectPathClient::ANE_DeviceOpen` needs to be executed before `H11ANEIn::ANE_ProgramSendRequest` . Second, `H11ANEIn::ANE_ProgramSendRequest` will eventually go to the function `H11ANEIn::ANE_ProgramSendRequest_gated` .

Both `H11ANEInDirectPathClient::ANE_DeviceOpen` and `H11ANEIn::ANE_ProgramSendRequest_gated` have a common operation: looking up a program buffer according to a program handle id. This program handle id comes from the user input. Without a correct program handle id, `H11ANEIn::ANE_ProgramSendRequest` would simply return. No notification would be sent.

So what is the program buffer and the program handle id? Remember that there is another more privileged IOUserClient, `H11ANEInUserClient` . `H11ANEInUserClien` t has the interfaces to create and destroy ane programs and get the program handle ids:

- H11ANEInUserClient::_ANE_ProgramCreate
- H11ANEInUserClient:: _ANE_ProgramCreateInstance
- H11ANEInUserClient::_ANE_ProgramDestroy

However, container apps cannot create `H11ANEInUserClient` , what can we do?

I had to spend some time on machine learning. It’s true. I learned how to use the Core ML framework (https://developer.apple.com/documentation/coreml), and how to integrate machine learning models into my app. I tried all the models on the page https://developer.apple.com/machine-learning/models/ .

Finally, I figured out that our app can talk to `aned` , and then ask `aned` to create a `H11ANEInUserClient` and create an ane program. After `aned` returns a program handle id to my app, we can continue to develop our exploit. So, if you want to reproduce the vulnerability, learn machine learning first!

Thank you for your time.

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment