DER Entitlements: The (Brief) Return of the Psychic Paper

Posted by Ivan Fratric, Project Zero

Note: The vulnerability discussed here, CVE-2022-42855, was fixed in iOS 15.7.2 and macOS Monterey 12.6.2. While the vulnerability did not appear to be exploitable on iOS 16 and macOS Ventura, iOS 16.2 and macOS Ventura 13.1 nevertheless shipped hardening changes related to it.

Last year, I spent a lot of time researching the security of applications built on top of XMPP, an instant messaging protocol based on XML. More specifically, my research focused on how subtle quirks in XML parsing can be used to undermine the security of such applications. (If you are interested in learning more about that research, I did a talk on it at Black Hat USA 2022. The slides and the recording can be found here and here).

At some point, when a part of my research was published, people pointed out other examples (unrelated to XMPP) where quirks in XML parsing led to security vulnerabilities. One of those examples was a vulnerability dubbed Psychic Paper, a really neat vulnerability in the way Apple operating system checks what *entitlements* an application has.

Entitlements are one of the core security concepts of Apple's operating systems. As Apple's documentation explains, "An entitlement is a right or privilege that grants an executable particular capabilities." For example, an application on an Apple operating system can't debug another application without possessing proper entitlements, even if those two applications run as the same user. Even applications running as root can't perform all actions (such as accessing some of the kernel APIs) without appropriate entitlements.

Psychic Paper was a vulnerability in the way entitlements were checked. Entitlements were stored inside the application's signature blob in the XML format, so naturally the operating system needed to parse those at some point using an XML parser. The problem was that the OS didn't have a single parser for this, but rather a staggering four parsers that were used in different places in the operating system. One parser was used for the initial check that the application only has permitted entitlements, and a different parser was later used when checking whether the application has an entitlement to perform a specific action.

When giving my talk on XMPP, I gave a challenge to the audience: Find me two different XML parsers that always, for every input, result in the same output. The reason why that is difficult is because XML, although intended to be a simple format, in reality is anything but simple. So it shouldn't come as a surprise that a way was found for one of Apple's XML parsers to return one set of entitlements and another parser to see a different set of entitlements when parsing the same entitlements blob.

The fix for the Psychic Paper bug: originally, the problem occurred because Apple had four XML parsers in the OS, so, surprisingly, the fix was to add a fifth one.

So, after my XMPP research, when I learned about the Psychic Paper bug, I decided to take a look at these XML parsers and see if I can somehow find another way to trigger the bug even after the fix. After playing with various Apple XML parsers, I had an XML snippet I wanted to try out. However when I actually tried to use it in an application, I discovered that the system for checking entitlements behaved completely differently than I thought. This was because of...

DER Entitlements

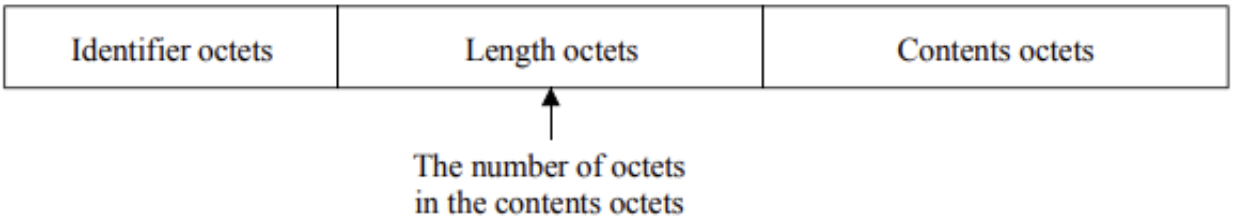
According to Apple developer documentation, "Starting in iOS 15, iPadOS 15, tvOS 15, and watchOS 8, the system checks for a new, more secure

signature format that uses Distinguished Encoding Rules, or DER, to embed entitlements into your app's signature". As another Apple article boldly proclaims, "The future is DER".

So, what is DER?

Unlike the previous text-based XML format, DER is a binary format, also commonly used in digital certificates. The format is specified in the X.690 standard.

DER follows relatively simple type-length-data encoding rules. An image from the specification illustrates that:



The Identifier field encodes the object type, which can be a primitive (e.g. a string or a boolean), but also a *constructed* type (an object containing other objects, e.g. an array/sequence). The length field encodes the number of bytes in the content. Length can be encoded differently, depending on the length of content (e.g. if content is smaller than 128 bytes, then the length field only takes a single byte). Length field is followed by the content itself. In case of constructed types, the content is also encoded using the same encoding rules.

An example from Apple developer documentation shows what DER-encoded entitlements might look like:

```
appl [ 16 ]
 INTEGER         :01
 cont [ 16 ]
  SEQUENCE
```

```
  UTF8STRING      :application-identifier
  UTF8STRING      :SKMME9E2Y8.com.example.apple-
samplecode.ProfileExplainer
 SEQUENCE
  UTF8STRING      :com.apple.developer.team-identifier
  UTF8STRING      :SKMME9E2Y8
 SEQUENCE
  UTF8STRING      :get-task-allow
  BOOLEAN         :255
 SEQUENCE
  UTF8STRING      :keychain-access-groups
  SEQUENCE
   UTF8STRING      :SKMME9E2Y8.*
   UTF8STRING      :com.apple.token
```

Each individual entitlement is a sequence which has two elements: a key and a value, e.g. “get-task-allow”:boolean(true). All entitlements are also a part of a constructed type (denoted as “cont [16]” in the listing).

DER is meant to have unique binary representation and replacing XML with DER was very likely motivated, at least in part, by preventing issues such as Psychic Paper. But does it necessarily succeed in that goal?

How entitlements are checked

To understand how entitlements are checked, it is useful to also look at the bigger picture and understand what security/integrity checks Apple operating systems perform on binaries before (and in some cases, while) running them.

Integrity information in Apple binaries is stored in a structure called the Embedded Signature Blob. This structure is a container for various other structures that play a role in integrity checking: The digital signature itself, but also entitlements and an important structure called the Code Directory. The Code Directory contains a hash of every page in the binary (up to the Signature Blob), but also other information, including the hash of the entitlements blob. The hash of the CodeDirectory is called cdHash and it is used to uniquely identify the binary. For example, it is cdHash that the digital signature actually signs. Since Code Directory contains a hash of the entitlements blob, note that any change to entitlements will lead to cdHash being different.

As also noted in the Psychic Paper writeup, there are several ways a binary might be signed:

The cdHash of the binary could be in the system's Trust Cache, which stores the hashes of system binaries.
The binary could be signed by the Apple App Store.
The binary could be signed by the developer, but in that case it must reference a "Provisioning Profile" signed by Apple.
On macOS only, a binary could also be self-signed.

We are mostly interested in the last two cases, because they are the only ones that allow us to provide custom entitlements. However, in those cases, any entitlements a binary has must either be a subset of those allowed by the provisioning profile created by Apple (in the "provisioning profile" case) or entitlements must only contain those whitelisted by the OS (in the self-signed case). That is, unless one has a vulnerability like Psychic Paper.

An entrypoint into file integrity checks is the vnode_check_signature function inside the AppleMobileFileIntegrity kernel extension. AppleMobileFileIntegrity is the kernel component of integrity checking, but there is also the userspace

demon: amfid which AppleMobileFileIntegrity uses to perform certain actions as noted further in the text.

We are not going to analyze the full behavior of vnode_check_signature. However, I will highlight the most important actions and those relevant to understanding DER entitlements workflow:

First, vnode_check_signature retrieves the entitlements in the DER format. If the currently loaded binary does not contain DER entitlements and only contains entitlements in the XML format, AppleMobileFileIntegrity calls transmuteEntitlementsInDaemon which uses amfid to convert entitlements from XML into DER format. The conversion itself is done via CFPropertyListCreateWithData (to convert XML to CFDictionary) and CESerializeCFDictionary (which converts CFDictionary to DER representation).

Checks on the signature are performed using the CoreTrust library by calling CTEvaluateAMFICodeSignatureCMS. This process has recently been documented in another vulnerability writeup and will not be examined in detail as it does not relate to entitlements directly.

Additional checks on the signature and entitlements are performed in amfid via the verify_code_directory function. This call will be analyzed in-depth later. One interesting detail about this interaction is that amfid receives the path to the executable as a parameter. In order to prevent race conditions where the file was changed between being loaded by kernel and checked by amfid, amfid returns the cdHash of the checked file. It is then verified that this cdHash matches the cdHash of the file in the kernel.

Provisioning profile information is retrieved and it is checked that the entitlements in the binary are a subset of the entitlements in the provisioning profile. This is done with the CEContextIsSubset function. This step does not appear to be present on macOS running on Intel CPUs,

however even in that case, the entitlements are still checked in amfid as will be shown below.

The verify_code_directory function of amfid performs (among other things) the following actions:

Parses the binary and extracts all the relevant information for integrity checking. The code that does that is part of the open-source Security library and the two most relevant classes here are StaticCode and SecStaticCode. This code is also responsible for extracting the DER-encoded entitlements. Once again, if only XML-encoded entitlements are present, they get converted to DER format. This is, once again, done by the CFPropertyListCreateWithData and CESerializeCFDictionary pair of functions. Additionally, for later use, entitlements are also converted to CFDictionary representation. This conversion from DER to CFDictionary is done using the CEManagedContextFromCFData and CEQueryContextToCFDictionary function pair.

Checks that the signature signs the correct cdHash and checks the signature itself. The checking of the digital signature certificate chain isn't actually done by the amfid process. amfid calls into trustd for that.

On macOS, the entitlements contained in the binary are filtered into restricted and unrestricted ones. On macOS, the unrestricted entitlements are com.apple.application-identifier, com.apple.security.application-groups*, com.apple.private.signing-identifier and com.apple.security.*. If the binary contains any entitlements not listed above, it needs to be checked against a provisioning profile. The check here relies on the entitlements CFDictionary, extracted earlier in amfid.

Later, when the binary runs, if the operating system wants to check that the binary contains an entitlement to perform a specific action, it can do this in several ways. The "old way" of doing this is to retrieve a dictionary

representation of entitlements and query a dictionary for a specific key. However, there is also a new API for querying entitlements, where the caller first creates a CEQuery object containing the entitlement key they want to query (and, optionally, the expected value) and then performs the query by calling CEContextQuery function with the CEQuery object as a parameter. For example, the IOTaskHasEntitlement function, which takes an entitlement name and returns bool relies on the latter API.

libCoreEntitlements

You might have noticed that a lot of functions for interacting with DER entitlements start with the letters "CE", such as CEQueryContextToCFDictionary or CEContextQuery. Here, CE stands for libCoreEntitlements, which is a new library created specifically for DER-encoded entitlements. libCoreEntitlements is present both in the userspace (as libCoreEntitlements.dylib) and in the OS kernel (as a part of AppleMobileFileIntegrity kernel extension). So any vulnerability related to parsing the DER entitlement format would be located there.

"There are no vulnerabilities here", I declared in one of the Project Zero team meetings. At the time, I based this claim on the following:

There is a single library for parsing DER entitlements, unlike the four/five used for XML entitlements. While there are two versions of the library, in userspace and kernel, those appear to be the same. Thus, there is no potential for parsing differential bugs like Psychic Paper. In addition to this, binary formats are much less susceptible to such issues in the first place.

The library is quite small. I both reviewed and fuzzed the library for memory corruption vulnerabilities and didn't find anything.

It turns out I was completely wrong (as it often happens when someone declares code is bug free). But first...

Interlude: SHA1 Collisions

After my failure to find a good libCoreEntitlements bug, I turned to other ideas. One thing I noticed when digging into Apple integrity checks is that SHA1 is still supported as a valid hash type for Code Directory / cdHash. Since practical SHA1 collision attacks have been known for some time, I wanted to see if they can somehow be applied to break the entitlement check.

It should be noted that there are two kinds of SHA1 collision attacks:

Identical prefix attacks, where an attacker starts with a common file prefix and then constructs collision blocks such that SHA1(prefix + collision_blocks1) == SHA1(prefix + collision_blocks2).

Chosen prefix attacks, where the attacker starts with two different prefixes chosen by the attacker and constructs collision blocks such that SHA1(prefix1 + collision_blocks1) == SHA1(prefix2 + collision_blocks2).

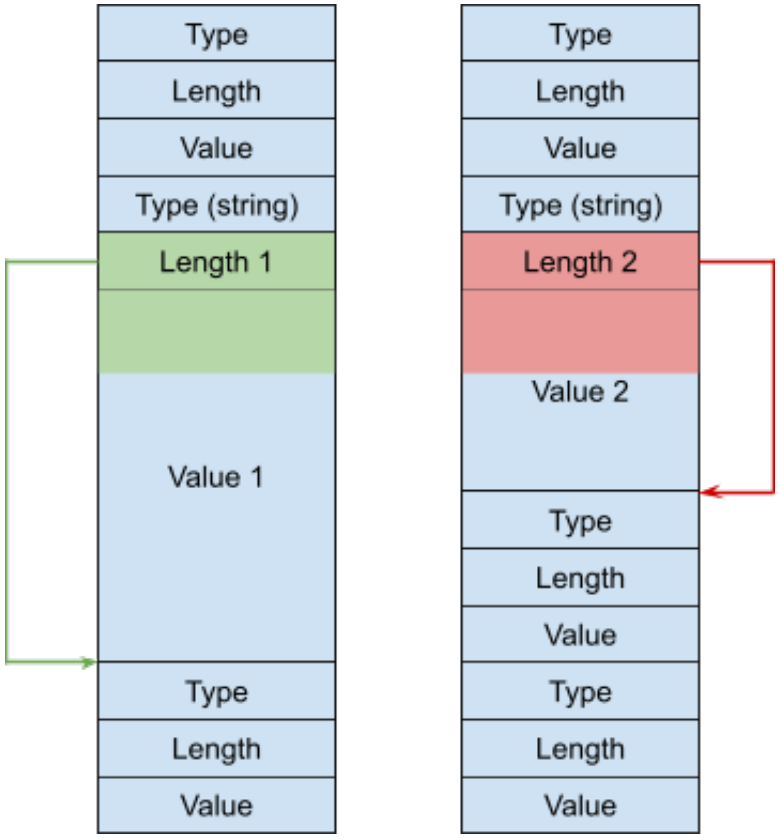
While both attacks are currently practical against SHA1, the first type is significantly cheaper than the second one. According to SHA-1 is a Shambles paper from 2020, the authors report a "cost of US$ 11k for a (identical prefix) collision, and US$ 45k for a chosen-prefix collision". So I wanted to see if an identical prefix attack on DER entitlements is possible. Special thanks to Ange Albertini for bringing me up to speed on the current state of SHA1 collision attacks.

My general idea was to

Construct two sets of entitlements that have the same SHA1 hash
Since entitlements will have the same SHA1 hash, the cdHash of the corresponding binaries will also be the same, as long as the corresponding Code Directories use SHA1 as the hash algorithm.
Exploit a race condition and switch the binary between being opened by

the kernel and being opened by amfid
If everything goes well, amfid is going to see one set of entitlements and the kernel another set of entitlements, while believing that the binary hasn't changed.

In order to construct an identical-prefix collision, my plan was to have an entitlements file that contains a long string. The string would declare a 16-bit size and the identical prefix would end right at the place where string length is stored. That way, when a collision is found, lengths of the string in two files would differ. Following the lengths would be more collision data (junk), but that would be considered string content and the parsing of the file would still succeed. Since the length of the string in two files would be different, parsing would continue from two different places in the two files. This idea is illustrated in the following image.



A (failed) idea for exploiting an identical prefix SHA1 collision against DER entitlements. The blue parts of the files are the same, while green and the

red part represent (different) collision blocks.

Except it won't work.

The reason it won't work is that every object in DER, including an object that contains other objects, has a size. Once again, hat tip to Ange Albertini who pointed out right away that this is going to be a problem.

Since each individual entitlement is stored in a separate sequence object, with our collision we could change the length of a string inside the sequence object (e.g. a value of some entitlement), but not change the length of the sequence object itself. Having a mismatch between the sequence length and the length of content inside the sequence could, depending on how the parser is implemented, lead to a parsing failure. Alternatively, if the parser was more permissive, the parser would succeed but would continue parsing after the end of the current sequence, so the collision wouldn't cause different entitlements to be seen in two different files.

Unless the sequence length is completely ignored.

So I took another look at the library, specifically at how sequence length was handled in different parts of the library, and that is when I found the actual bug.

The actual bug

libCoreEntitlements does not implement traversing DER entitlements in a single place. Instead, traversing is implemented in (at least) three different places:

Inside recursivelyValidateEntitlements, called from CEValidate which is used to validate entitlements before being loaded (e.g. CEValidate is called from CEManagedContextFromCFData which was mentioned before). Among other things, CEValidate ensures that entitlements are in

alphabetical order and there are no duplicate entitlements. der_vm_iterate, which is used when converting entitlements to dictionary (CEQueryContextToCFDictionary), but also from CEContextIsSubset. der_vm_execute_nocopy, which is used when querying entitlements using CEContextQuery API.

The actual bug is that these traversal algorithms behave slightly differently, in particular in how they handle entitlement sequence length. der_vm_iterate behaved correctly and, after processing a single key/value sequence, continued processing after the sequence end (as indicated by the sequence length). recursivelyValidateEntitlements and der_vm_execute_nocopy however completely ignore the sequence length (beyond an initial check that it is within bounds of the entitlements blob) and, after processing a single entitlement (key/value sequence), would continue processing after the end of the *value*. This difference in iterating over entitlements is illustrated in the following image.

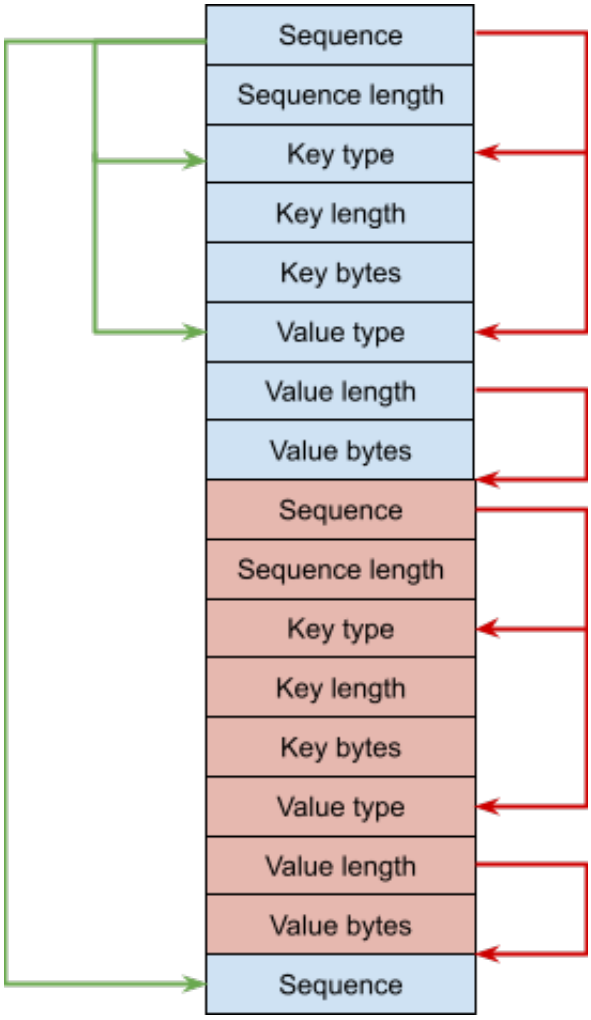


Illustration of how different algorithms within libCoreEntitlements traverse the same DER structure. der_vm_iterate is shown as green arrows on the left while der_vm_execute is shown as red arrows on the right. The red parts of the file represent a “hidden” entitlement that is only visible to the second algorithm.

This allowed hiding entitlements by embedding them as children of other entitlements, as shown below. Such entitlements would then be visible to CEValidate and CEContextQuery, but hidden from CEQueryContextToCFDictionary and CEContextIsSubset.

```
SEQUENCE
  UTF8STRING      :application-identifier
  UTF8STRING      :testapp
```

```
  SEQUENCE
    UTF8STRING      :com.apple.private.foo
    UTF8STRING      :bar
SEQUENCE
  UTF8STRING      :get-task-allow
  BOOLEAN         :255
```

Luckily (for an attacker), the functions used to perform the check that all entitlements in the binary are a subset of those in the provisioning profile wouldn't see these hidden entitlements, which would only surface when the kernel would query for a presence of a specific entitlement using the CEContextQuery API.

Since CEValidate would also see the "hidden" entitlements, these would need to be in alphabetical order together with "normal" entitlements. However, because generally allowed entitlements such as application-identifier on iOS and com.apple.application-identifier on macOS appear pretty soon in the alphabetical order, this requirement doesn't pose an obstacle for exploitation.

Exploitation

In order to try to exploit these issues, some tooling was required. Firstly, tooling to create crafted entitlements DER. For this, I created a small utility called createder.cpp which can be used for example as

./createder entitlements.der com.apple.application-identifier=testapp -- com.apple.private.security.kext-collection-management

where entitlements.der is the output DER encoded entitlements file, entitlements before "--" are "normal" entitlements and those after "--" are hidden.

Next, I needed a way to embed such crafted entitlements into a signature blob. Normally, for signing binaries, Apple's codesign utility is used. However, this utility only accepts XML entitlements as input and the user can't provide a DER blob. At some point, codesign converts the user-provided XML to DER and embeds both in the resulting binary.

I decided the easiest way to embed custom DER in a binary would be to modify codesign behavior, specifically the XML to DER conversion process. To do this, I used TinyInst, a dynamic instrumentation library I developed together with contributors. TinyInst currently supports macOS and Windows.

In order to embed custom DER, I hooked two functions from libCoreEntitlements that are used during XML to DER conversion: CESizeSerialization and CESerialize (CESerializeWithOptions on macOS 13 / iOS 16). The first one computes the size of DER while the second one outputs DER bytes.

Initially, I wrote these hooks using TinyInst's general-purpose low-level instrumentation API. However, since this process was more cumbersome than necessary, in the meantime I created an API specifically for function hooking. So instead of linking to my initial implementation that I sent with the bug report to Apple, let me instead show how the hooks would be implemented using the new TinyInst Hook API:

cehook.h

```
#include "hook.h"

class CESizeSerializationHook : public HookReplace {

public:

  CESizeSerializationHook() : HookReplace("libCoreEntitlements.dylib",

                                          "_CESizeSerialization",3) {}
```

```
protected:
  void OnFunctionEntered() override;
};
class CESerializeWithOptionsHook : public HookReplace {
public:
  CESerializeWithOptionsHook() : HookReplace("libCoreEntitlements.dylib",
                                  "_CESerializeWithOptions", 6) {}
protected:
  void OnFunctionEntered() override;
};
// the main client
class CEInst : public TinyInst {
public:
  CEInst();
protected:
  void OnModuleLoaded(void* module, char* module_name) override;
};
```

cehook.cpp

```
#include "cehook.h"
#include <fstream>
char *der;
size_t der_size;
size_t kCENoError;
```

```
// read entitlements.der into buffer
void ReadEntitlementsFile() {
  std::ifstream file("entitlements.der", std::ios::binary | std::ios::ate);
  if(file.fail()) {
    FATAL("Error reading entitlements file");
  }
  der_size = (size_t)file.tellg();
  file.seekg(0, std::ios::beg);
  der = (char *)malloc(der_size);
  file.read(der, der_size);
}
void CESizeSerializationHook::OnFunctionEntered() {
  // 3rd argument (output) is a pointer to the size
  printf("In CESizeSerialization\n");
  size_t out_addr = GetArg(2);
  RemoteWrite((void*)out_addr, &der_size, sizeof(der_size));
  SetReturnValue(kCENoError);
}
void CESerializeWithOptionsHook::OnFunctionEntered() {
  // 5th argument points to output buffer
  printf("In CESerializeWithOptions\n");
  size_t out_addr = GetArg(4);
  RemoteWrite((void*)out_addr, der, der_size);
  SetReturnValue(kCENoError);
```

```
}
CEInst::CEInst() {
  ReadEntitlementsFile();
  RegisterHook(new CESizeSerializationHook());
  RegisterHook(new CESerializeWithOptionsHook());
}
void CEInst::OnModuleLoaded(void* module, char* module_name) {
  if(!strcmp(module_name, "libCoreEntitlements.dylib")) {
    // we must get the value of kCENoError,
    // which is libCoreEntitlements return value in case no error
    size_t kCENoError_addr =
          (size_t)GetSymbolAddress(module, (char*)"_kCENoError");
    if (!kCENoError_addr) FATAL("Error resolving symbol");
    RemoteRead((void *)kCENoError_addr, &kCENoError, sizeof(kCENoError));
  }
  TinyInst::OnModuleLoaded(module, module_name);
}
```

HookReplace (base class for our 2 hooks) is a helper class used to completely replace the function implementation with the hook. CESizeSerializationHook::OnFunctionEntered() and CESerializeWithOptionsHook::OnFunctionEntered() are the bodies of the hooks. Additional code is needed to read the replacement entitlement DER and also to retrieve the value of kCENoError, which is a value libCoreEntitlements functions return on success and that we must return from our hooked versions. Note that, on macOS Monterey,

CESerializeWithOptions should be replaced with CESerialize and the 3rd argument will be the output address rather than 4th.

So, with that in place, I could sign a binary using DER provided in the entitlements.der file.

Now, with the tooling in place, the question becomes, how to exploit it. Or rather, which entitlement to target. In order to exploit the issue, we need to find an entitlement check that is ultimately made using CEContextQuery API. Unfortunately, that rules out most userspace daemons - all the ones I looked at still did entitlement checking the "old way", by first obtaining a dictionary representation of entitlements. That leaves "just" the entitlement checks done by the OS kernel.

Fortunately, a lot of checks in the kernel are made using the vulnerable API. Some of these checks are done by AppleMobileFileIntegrity itself using functions like OSEntitlements::queryEntitlementsFor, AppleMobileFileIntegrity::AMFIEntitlementGetBool or proc_has_entitlement. But there are other APIs as well. For example, the IOTaskHasEntitlement and IOCurrentTaskHasEntitlement functions that are used from over a hundred places in the kernel (according to macOS Monterey kernel cache) end up calling CEContextQuery (by first calling amfi->OSEntitlements_query()).

One other potentially vulnerable API was IOUserClient::copyClientEntitlement, but only on Apple silicon, where pmap_cs_enabled() is true. Although I didn't test this, there is some indication that in that case the CEContextQuery API or something similar to it is used. IOUserClient::copyClientEntitlement is most notably used by device drivers to check entitlements.

I attempted to exploit the issue on both macOS and iOS. On macOS, I found places where such an entitlement bypass could be used to unlock previously unreachable attack surfaces, but nothing that would

immediately provide a stand-alone exploit. In the end, to report something while looking for a better primitive, I reported that I was able to invoke functionality of the kext_request API (kernel extension API) that I normally shouldn't be able to do, even as the root user. At the time, I did my testing on macOS Monterey, versions 12.6 and 12.6.1.

On iOS, exploiting such an issue would potentially be more interesting, as it could be used as part of a jailbreak. I was experimenting with iOS 16.1 and nothing I tried seemed to work, but due to the lack of introspection on the iOS platform I couldn't figure out why at the time.

That is, until I received a reply from Apple on my macOS report, asking me if I can reproduce on macOS Ventura. macOS Ventura was just released in the same week when I sent my report to Apple and I was hesitant to upgrade to a new major version while having a working setup on macOS Monterey. But indeed, when I upgraded to Ventura, and after making sure my tooling still works, the DER issue wouldn't reproduce anymore. This was likely also the reason all of my iOS experiments were unsuccessful: iOS 16 shares the codebase with macOS Ventura.

It appears that the issue was fixed due to refactoring and not as a security issue (macOS Monterey would be updated in this case as well as Apple releases some security patches not just for the latest major version of macOS, but also for the two previous ones). Looking at the code, it appears the API libCoreEntitlements uses to do low-level DER parsing has changed somewhat - on macOS Monterey, libCoreEntitlements used mostly ccder_decode* functions to interact with DER, while on macOS ventura ccder_blob_decode* functions are used more often. The latter additionally take the parent DER structure as input and thus make processing of hierarchical DER structure somewhat less error-prone. As a consequence of this change, der_vm_execute_nocopy continues processing the next key/value sequence after the end of the current one (which is the intended behavior) and not, as before, after the end of the

value object. At this point, since I could no longer reproduce the issue on the latest OS release, I was no longer motivated to write the exploit and stopped my efforts in order to focus on other projects.

Apple addressed the issue as CVE-2022-42855 on December 13, 2022. While Apple applied the CVE to all supported versions of their operating systems, including macOS Ventura and iOS 16, on those latest OS versions the patch seems to serve as an additional validation step. Specifically, on macOS Ventura, the patch is applied to function der_decode_key_value and makes sure that the key/value sequence does not contain other elements besides just key and the value (in other words, it makes sure that the length of the sequence matches length of key + value objects).

Conclusion

It is difficult to claim that a particular code base has no vulnterabilities. While that might be possible for a *specific type* of vulnerabilities, it is difficult, and potentially impossible, to predict all types of vulnerabilities a complex codebase might have. After all, you can't reason about what you don't know.

While DER encoding is certainly a much safer choice than XML when it comes to parsing logic issues, this blog post demonstrates that such issues are still possible even with the DER format.

And what about the SHA1 collision? While the identical prefix attack shouldn't work against DER, doesn't that still leave the chosen prefix attack? Maybe, but that's something to explore another time. Or maybe (hopefully) Apple will remove SHA1 support in their signature blobs and there will be nothing to explore at all.