3
CODE SIGNING

In this chapter, we'll write code that can extract code signing information from distribution file formats that malware often abuses, such as disk images and packages. Then we'll turn our attention to the code signing information of on-disk Mach-O binaries and running processes. For each, I'll show you how to programmatically validate the code signing information and detect any revocations.

The behavior-based heuristics covered throughout this book are a powerful approach to detecting malware. But the approach comes with a downside: *false positives*, which occur when code incorrectly flags something as suspicious.

One way to reduce false positives is by examining an item's code signing information. Apple's support of cryptographic code signing is unparalleled, and as malware detectors, we can leverage it in a variety of ways, most notably to confirm that items come from known, trusted sources and that these items haven't been tampered with.

On the flip side, we should closely scrutinize any unsigned or non-notarized item. For example, malware is often either wholly unsigned or signed in an ad hoc manner, meaning with a self-signed or untrusted certificate. While threat actors may occasionally sign their malware with fraudulently obtained or stolen developer certificates, it's rare for Apple to have notarized the malware as well. Moreover, Apple is often quick to revoke the signing certificate or notarization ticket when it makes a mistake.

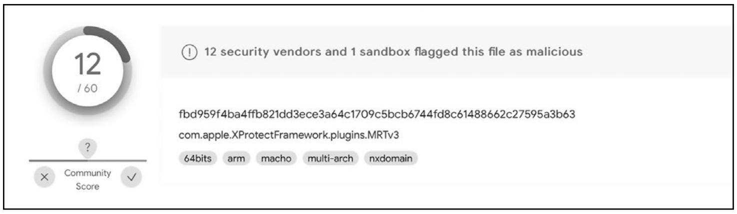
You can find the majority of code snippets presented in this chapter in the *checkSignature* project, available in the book's GitHub repository.

The Importance of Code Signing in Malware Detection

As an example of why code signing is useful for malware detection, imagine that you develop a heuristic to monitor the filesystem for persistent items (a reasonable approach to detecting malware, as the vast majority of Mac malware will persist on an infected host). Say your heuristic triggers when the *com.microsoft.update.agent.plist* property list is persisted as a launch agent. This property list references an application named *MicrosoftAutoUpdate.app*, which the operating system will now start automatically each time the user logs in.

If your detection capabilities don't take into account the code signing information of the persisted item, you might generate an alert for what is actually a totally benign persistence event. The question, therefore, becomes: Is this really a Microsoft updater, or is it malware masquerading as such? By checking the application's code signing signature, you should be able to answer this question conclusively; if Microsoft has indeed signed the item, you can ignore the persistence event, but if not, the item warrants a much closer look.

Unfortunately, existing malware detection products may fail to adequately take code signing information into account. For example, consider Apple's Malware Removal Tool (MRT), a built-in malware detection tool found in certain versions of macOS. This platform binary is, of course, signed by Apple proper. Yet many antivirus engines have, at one point or another, flagged an MRT binary, *com.apple.XProtectFramework.plugins.MRTv3*, as malicious because their antivirus signatures naively matched MRT's own embedded viral signatures (Figure 3-1).


12 security vendors and 1 sandbox flagged this file as malicious

fbd959f4ba4ffb821dd3ece3a64c1709c5bcb6744fd8c61488662c27595a3b63

com.apple.XProtectFramework.plugins.MRTv3

64bits arm macho multi-arch nxdomain

Figure 3-1: Apple's Malicious Removal Tool flagged as malicious

A rather hilarious false positive indeed. Joking aside, products that incorrectly classify legitimate items as malware may alert the user, causing consternation, or worse, may break legitimate functionality by quarantining the item. While third-party security products luckily can't delete system components such as MRT, Apple has been known to inadvertently block its own components, disrupting system operations.[1] In both cases, the detection logic could have simply checked the item's code signing information to see that it belonged to a trusted source.

Code signing information can do more than just reduce false positives. For example, security tools should allow trusted or user-approved items to perform actions that might otherwise trigger an alert. Consider the case of a simple firewall that generates a notification whenever an untrusted item attempts to access the network. To distinguish between trusted and untrusted items, the firewall can check the items' code signing signatures. Creating firewall rules based on code signing information has a few benefits:

- If malware attempts to bypass the firewall by modifying a legitimate item, code signing checks will detect this tampering.
- If an approved item moves to another location on the filesystem, the rule will still match, as it isn't tied to the item's path or specific location.

Hopefully, these brief examples have already shown you the value of inspecting the code signing information. For good measure, let's list a few other ways that code signing information can help us programmatically detect malicious code:

Detecting notarization Recent versions of macOS require all downloaded software to be signed in order to run. As such, most malware is now signed, often with an ad hoc certificate or fraudulent developer ID. However, malware is rarely notarized, because notarization requires submitting an item to Apple, which scans it, then issues a notarization ticket if the item doesn't appear to be malicious.[2] On the few occasions that Apple has inadvertently notarized malware, it has quickly detected the misstep and revoked the notarization.[3] These blunders are exceedingly rare, and notarized items are most likely benign. Using code signing, you can quickly determine whether an item is notarized, providing a reliable indication that Apple doesn't consider it to be malware.

Detecting revocations If Apple has revoked an item's code signing certificate or notarization ticket, it means they have determined that the item should no longer be distributed and run. Although revocation sometimes happens for benign reasons, it's often because Apple deemed the item malicious. This chapter explains how to programmatically detect revocations.[4]

Linking items to known adversaries Code signing information that researchers have attributed to malicious adversaries, such as team identifiers, can later identify other malware specimens created by the same authors.

When detecting malware, you're generally interested in the following code signing information for an item:

- The general status of the information, signing certificate, and notarization ticket. Is the item fully signed and notarized, and are the signing certificate and notarization ticket still in good standing?
- The code signing authorities describing the chain of signers, as they can provide insight into the origin and trustworthiness of the signed item.
- The item's optional team identifier, which specifies the team or company that created the signed item. If the team identifier belongs to a reputable company, you can generally trust the signed item.

This chapter won't cover code signing internals. Rather, it focuses on higher-level concepts, as well as the APIs used to extract code signing information.[5]

Keep in mind, however, that not everything on macOS is signed, nor is it signed in the same way. Most notably, developers can't sign stand-alone scripts (one of the reasons Apple is desperately trying to deprecate them). Nor is the macOS kernel signed per se. Instead, the boot process uses a cryptographic hash to verify that it remains pristine.

While developers can and should sign distribution media such as disk images, packages, and zip archives, as well as applications and stand-alone binaries, the tools and APIs that extract the code signing information are often specific to the file type. For example, Apple's `codesign` utility and code signing services APIs work on disk images, applications, and binaries, but not on packages, whose information you can examine with the `pkgutil` utility or the private `PackageKit` APIs.

Let's consider how to manually and programmatically extract and validate code signing information, starting with distribution media.

Disk Images

Both legitimate developers and malware authors often distribute their code as disk images, which have the *.dmg* extension. Most disk images containing malware are unsigned, and if you encounter an unsigned *.dmg*, you should at the very least check whether the items it contains are signed and notarized. The presence of code signing information doesn't mean a disk image is benign, however; nothing stops malware authors from leveraging cryptographic signatures. When you encounter a signed disk image, use its code signing information to identify the creator.

Manually Verifying Signatures

You can manually verify the signature of a disk image with macOS's built-in `codesign` utility. Execute it with the `--verify` command line option (or `-v` for short) and the path of a *.dmg* file.

In the following example, `codesign` identifies a validly signed disk image containing LuLu, legitimate software from Objective-See. When it encounters validly signed images, the tool won't output anything by default; hence, we use the -dvv option to display verbose output:

```
% codesign --verify LuLu_2.6.0.dmg

% codesign --verify -dvv LuLu_2.6.0.dmg
Executable=/Users/Patrick/Downloads/LuLu_2.6.0.dmg
Identifier=LuLu
Format=disk image
...
Authority=Developer ID Application: Objective-See, LLC (VBG97UB4TA)
Authority=Developer ID Certification Authority
Authority=Apple Root CA
```

The verbose output shows information about the disk image, such as its path, identifier, and format, as well as its code signing status, including the certificate authority chain. From the certificate authority chain, you can see the package has been signed with an Apple Developer ID belonging to Objective-See.

If a disk image isn't signed, the utility will display a `code object is not signed at all` message. Many software items, including most of the malware specimens distributed via disk images, fall into this category; the authors may have signed the software or malware but not its distribution media. For example, take a look at the EvilQuest malware. Distributed via disk images, it contains packages of trojanized applications:

```
% codesign --verify "EvilQuest/Mixed In Key 8.dmg"
EvilQuest/Mixed In Key 8.dmg: code object is not signed at all
```

Lastly, if Apple has revoked a disk image's signature, `codesign` will display `CSSMERR_TP_CERT_REVOKED`. You can see an example of this in the disk image used to distribute the CreativeUpdate malware:

```
% codesign --verify "CreativeUpdate/Firefox 58.0.2.dmg"
CreativeUpdate/Firefox 58.0.2.dmg: CSSMERR_TP_CERT_REVOKED
```

The malware's signature is no longer valid.

Extracting Code Signing Information

Let's programmatically extract and verify the code signing information of a disk image using Apple's code signing services (`Sec*`) APIs.[6] In the chapter's *checkSignature* project, you'll find a function named `checkItem` that takes the path to an item to verify, such as a disk image, and returns a dictionary containing the results of the verification. For validly signed items, it also returns information such as the code signing authorities, if any.

For the sake of brevity, I've omitted basic sanity and error checks from most of the code snippets in this book. However, when it comes to code signing, which provides the means to make crucial decisions about the trustworthiness of items, it's imperative that the code handle errors appropriately.

Without resilient error-handling mechanisms, the code might inadvertently trust a malicious item masquerading as something benign! Thus, in this chapter, the code snippets don't omit such important error checks.

The first step to extracting the code signing information of any item is to obtain what is referred to as a *code object* reference that you can then pass to all subsequent code signing API calls. For on-disk items such as disk images, you'll obtain a static code object of type `SecStaticCodeRef`.[7] For running processes, you'll instead obtain a dynamic code object of type `SecCodeRef`.[8]

To obtain a static code reference from a disk image, invoke the `SecStaticCodeCreateWithPath` API with a path to the specified disk image, optional flags, and an out pointer. Once the function returns, this out pointer will contain a `SecStaticCode` object for use in subsequent API calls (Listing 3-1).[9] Note that you should free this pointer using `CFRelease` once you're done with it.

```
NSMutableDictionary* checkImage(NSString* item) {
    SecStaticCodeRef codeRef = NULL;
    NSMutableDictionary* signingInfo = [NSMutableDictionary dictionary];

  ❶ CFURLRef itemURL = (__bridge CFURLRef)([NSURL fileURLWithPath:item]);

  ❷ OSStatus status = SecStaticCodeCreateWithPath(itemURL, kSecCSDefaultFlags, &codeRef);
  ❸ if(errSecSuccess != status) {
        goto bail;
    }
    ...

bail:
    if(nil != codeRef) {
        CFRelease(codeRef);
    }
    return signingInfo;
}
```

Listing 3-1: Obtaining a static code object for a disk image

After initializing a URL object containing the path of the disk image we're to check ❶, we invoke the `SecStaticCodeCreateWithPath` API ❷. If this function fails, it will return a nonzero value ❸. If `Sec*` APIs succeed, they return zero, which maps to the preferred `errSecSuccess` constant. I discuss the error codes that the `Sec*` APIs may return in "Code Signing Error Codes" on page 97. They're also detailed in Apple's "Code Signing Services Result Codes" documentation.[10] Also note that when we are done with the code reference, we must release it via `CFRelease`.

In this and subsequent code snippets, you'll see the use of *bridging*, a mechanism to cast Objective-C objects in a toll-free manner into (and out of) the Core Foundation objects used by Apple's code signing APIs. For example, in Listing 3-1, the `SecStaticCodeCreateWithPath` API expects a `CFURLRef` as its first argument. After converting the path of the disk image to an `NSURL` object, we bridge it to a `CFURLRef` using `(__bridge CFURLRef)`. You can read more about bridging in Apple's "Core Foundation Design Concepts."[11]

Once we've created a static code object for the disk image, we can invoke the SecStaticCodeCheckValidity API with the just-created SecStaticCode object to check its validity, saving the result of the call so we can return it to the caller (Listing 3-2).

```
...
#define KEY_SIGNATURE_STATUS @"signatureStatus"

status = SecStaticCodeCheckValidity(codeRef, kSecCSEnforceRevocationChecks, NULL);
signingInfo[KEY_SIGNATURE_STATUS] = [NSNumber numberWithInt:status];
if(errSecSuccess != status) {
    goto bail;
}
```

Listing 3-2: Checking a disk image's code signing validity

You'll normally see this API invoked with the kSecCSDefaultFlags constant, which contains a default set of flags, but to perform certificate revocation checks as part of the validation, you need to pass in kSecCSEnforceRevocationChecks.

Next, we check that the invocation succeeded. If we fail to perform this validation, malicious code may be able to subvert code signing checks.[12] If the API fails, for example, with errSecCSUnsigned, you'll likely want to abort the extraction of any further code signing information, which either won't be present (in the case of unsigned items) or won't be trustworthy.

Once we've determined the validity of the disk image's code signing status, we can extract its code signing information via the SecCodeCopySigningInformation API. We pass this API the SecStaticCode object, the kSecCSSigningInformation flag, and an out pointer to a dictionary to populate with the disk image's code signing details (Listing 3-3).

```
CFDictionaryRef signingDetails = NULL;

status = SecCodeCopySigningInformation(codeRef,
kSecCSSigningInformation, &signingDetails);
if(errSecSuccess != status) {
    goto bail;
}
```

Listing 3-3: Extracting code signing information

Now we can extract stored details from the dictionary, such as the certificate authority chain, using the key kSecCodeInfoCertificates (Listing 3-4).

```
#define KEY_SIGNING_AUTHORITIES @"signatureAuthorities"

signingInfo[KEY_SIGNING_AUTHORITIES] = ((__bridge NSDictionary*)signingDetails)
[(__bridge NSString*)kSecCodeInfoCertificates];
```

Listing 3-4: Extracting the certificate authority chain

If the item has an ad hoc signature, it won't have an entry under the kSecCodeInfoCertificates key in its code signing dictionary. Another way to identify ad hoc signatures is to check the kSecCodeInfoFlags key, which contains the item's code signing flags. For ad hoc signatures, we'll find the second least significant bit (2) set in the flag, which, after consulting Apple's *cs_blobs.h* header file, we see maps to the constant CS_ADHOC.

It's rare to see disk images signed in an ad hoc manner, as they don't require a signature to begin with, but because apps and binaries must be signed to run, you'll commonly see malware signed in this way. We can extract the code signing flags in the manner shown in Listing 3-5.

```
#define KEY_SIGNING_FLAGS @"flags"

signingInfo[KEY_SIGNING_FLAGS] = [(__bridge NSDictionary*)signingDetails
objectForKey:(__bridge NSString*)kSecCodeInfoFlags];
```

Listing 3-5: Extracting an item's code signing flags

We could then check these extracted flags for the value indicating an ad hoc signature (Listing 3-6).

```
if([results[KEY_SIGNING_FLAGS] intValue] & CS_ADHOC) {
    // Code here will run only if item is signed in an ad hoc manner.
}
```

Listing 3-6: Verifying code signing flags

The dictionary stores these flags in a number object, so we must first convert them to an integer and then perform a bitwise AND operation (&) to check for the bits specified by CS_ADHOC.

When we're finished with the CFDictionaryRef dictionary, we must free it via CFRelease.

Extracting Notarization Information

To extract the notarization status of the disk images, we can use the SecRequirementCreateWithString API, which lets us create a requirement to which an item must conform. In Listing 3-7, we create a requirement with the string "notarized".

```
static SecRequirementRef requirement = NULL;
SecRequirementCreateWithString(CFSTR("notarized"), kSecCSDefaultFlags, &requirement);
```

Listing 3-7: Initializing a requirement reference string

The API generates an object by compiling the code requirement string we pass to it, allowing us to use the requirement multiple times.[13] If you're performing a one-time requirement check, you can skip the compilation step and instead use the SecTaskValidateForRequirement API, which takes a string-based requirement to validate as a second argument.

Now we can call the SecStaticCodeCheckValidity API, passing it the SecStaticCode object, as well as the requirement reference (Listing 3-8).

```
if(errSecSuccess == SecStaticCodeCheckValidity(codeRef, kSecCSDefaultFlags, requirement)) {
    // Code placed here will run only if the item is notarized.
}
```

Listing 3-8: Checking a notarization requirement

If the API returns `errSecSuccess`, we know that the item conforms to the requirement we passed in. In our case, this means the disk image is indeed notarized. You can read more about requirements, including useful requirement strings, in Apple's informative "Code Signing Requirement Language" document.[14]

If the notarization validation fails, we should check whether Apple has revoked the item's notarization ticket, even if the item is validly signed. This nuanced case presents a huge red flag; for an example, see the discussion of the 3CX supply chain attack in "On-Disk Applications and Executables" on page 93.

Although I've asked for one,[15] Apple has not approved any method of determining whether an item's notarization ticket has been revoked. However, two undocumented APIs, `SecAssessmentCreate` and `SecAssessmentTicketLookup`, can provide this information. In Listing 3-9, we invoke `SecAssessmentCreate` to check whether an item that has passed other code signing checks has had its notarization ticket revoked.

```
❶ SecAssessmentRef secAssessment = SecAssessmentCreate(itemURL,
  kSecAssessmentDefaultFlags, (__bridge CFDictionaryRef)(@{}), &error);
❷ if(NULL == secAssessment) {
      if( (CSSMERR_TP_CERT_REVOKED == CFErrorGetCode(error)) ||
          (errSecCSRevokedNotarization == CFErrorGetCode(error)) ) {
          signingInfo[KEY_SIGNING_NOTARIZED] =
          [NSNumber numberWithInteger:errSecCSRevokedNotarization];
      }
  }
❸ if(NULL != secAssessment) {
      CFRelease(secAssessment);
  }
```

Listing 3-9: Checking whether a notarization ticket has been revoked

We pass the function the path to the item, such as a disk image; the default assessment flags; an empty but non-`NULL` dictionary; and an out pointer to an error variable ❶.

If Apple has revoked either the notarization ticket or the certificate, the function will set an error to `CSSMERR_TP_CERT_REVOKED` or `errSecCSRevokedNotarization`. The name of the first error is a bit nuanced, as it can return items with valid certificates but revoked notarization tickets, which is what we're interested in here.

If we receive a `NULL` assessment and either of these error codes ❷, we know something has been revoked. Moreover, because we've already validated the code signing certificates, we know that the revocation refers to the notarization ticket. Once we're done with the assessment, we make sure to free it if it's not `NULL` ❸.

Running the Tool

Let's compile the *checkSignature* project and run it against the disk images mentioned earlier in this section:

```
% ./checkSignature LuLu_2.6.0.dmg
Checking: LuLu_2.6.0.dmg
Status: signed
Is notarized: no

Signing auths: (
    "<cert(0x11100a800) s: Developer ID Application: Objective-See, LLC (VBG97UB4TA)
    i: Developer ID Certification Authority>",
    "<cert(0x111808200) s: Developer ID Certification Authority i: Apple Root CA>",
    "<cert(0x111808a00) s: Apple Root CA i: Apple Root CA>"
)
```

As expected, the code reports that LuLu's disk image is signed, though it isn't notarized. The code also extracts the chain of its code signing authorities, which include its developer ID application and its developer ID certification authority. (When detecting malware, you may want to ignore disk images signed via trusted developer IDs unless you're interested in detecting supply chain attacks.)

Now let's run the code against the EvilQuest malware. As you'll see, the code matches the results from Apple's `codesign` utility, indicating that the disk image is unsigned:

```
% ./checkSignature "EvilQuest/Mixed In Key 8.dmg"
Checking: Mixed In Key 8.dmg
Status: unsigned
```

Finally, we run the code against the CreativeUpdate malware, whose code signing certificate has been revoked:

```
% ./checkSignature "CreativeUpdate/Firefox 58.0.2.dmg"
Checking: Firefox 58.0.2.dmg
Status: revoked
```

Now that we can programmatically extract and validate code signing information from disk images, let's do the same for packages, which unfortunately require a completely different approach.

Packages

You can manually verify the signature of a package (*.pkg*) with the built-in `pkgutil` utility. Execute it with the `--check-signature` command line option, followed by the path of the *.pkg* file you'd like to verify. The utility should display the result of the check in a line prefixed with `Status`:

```
% pkgutil --check-signature GoogleChrome.pkg
Package "GoogleChrome.pkg":
   Status: signed by a developer certificate issued by Apple for distribution
   Notarization: trusted by the Apple notary service
   Signed with a trusted timestamp on: 05-15 20:46:50 +0000
   Certificate Chain:
    1. Developer ID Installer: Google LLC (EQHXZ8M8AV)
       Expires: 2027-02-01 22:12:15 +0000
       SHA256 Fingerprint:
           40 02 6A 12 12 38 F4 E0 3F 7B CE 86 FA 5A 22 2B DA 7A 3A 20 70 FF
           28 0D 86 AA 4E 02 56 C5 B2 B4
       ------------------------------------------------------------------------
    2. Developer ID Certification Authority
       Expires: 2027-02-01 22:12:15 +0000
       SHA256 Fingerprint:
           7A FC 9D 01 A6 2F 03 A2 DE 96 37 93 6D 4A FE 68 09 0D 2D E1 8D 03
           F2 9C 88 CF B0 B1 BA 63 58 7F
       ------------------------------------------------------------------------
    3. Apple Root CA
       Expires: 2035-02-09 21:40:36 +0000
       SHA256 Fingerprint:
           B0 B1 73 0E CB C7 FF 45 05 14 2C 49 F1 29 5E 6E DA 6B CA ED 7E 2C
           68 C5 BE 91 B5 A1 10 01 F0 24
```

The results show that pkgutil has verified that the package, a Google Chrome installer, is signed and notarized. The tool also displayed the certificate authority chain, which indicates that the package was signed via an Apple Developer ID belonging to Google.

Note that you can't use the codesign utility to check the code signature of packages, as *.pkg* files use a different mechanism for storing code signing information that codesign doesn't understand. For example, when run against the same package, it detects no signature:

```
% codesign --verify -dvv GoogleChrome.pkg
GoogleChrome.pkg: code object is not signed at all
```

If a package isn't signed, pkgutil will display a Status: no signature message. Most malware distributed via packages, including EvilQuest, falls into this category. These disk images contain a malicious package, and once the disk image is mounted, we can use pkgutil to show that this package is unsigned:

```
% pkgutil --check-signature "EvilQuest/Mixed In Key 8.pkg"
Package "Mixed In Key 8.pkg":
   Status: no signature
```

Finally, if a package was signed but Apple has revoked its code signing certificate, pkgutil will display Status: revoked signature but will still show the certificate chain. We find an example of this behavior in a package used to distribute the KeySteal malware:

```
% pkgutil --check-signature KeySteal/archive.pkg
Package "archive.pkg":
   Status: revoked signature
   Signed with a trusted timestamp on: 10-18 12:58:45 +0000
   Certificate Chain:
    1. Developer ID Installer: fenghua he (32W7BZNTSV)
       Expires: 2027-02-01 22:12:15 +0000
       SHA256 Fingerprint:
           EC 7C 85 1D B0 A0 8C ED 45 31 6B 8E 9D 7D 34 0F 45 B8 4E CE 9D 9C
           97 DB 2F 63 57 C2 D9 71 0C 4E
       ------------------------------------------------------------------------
    2. Developer ID Certification Authority
       Expires: 2027-02-01 22:12:15 +0000
       SHA256 Fingerprint:
           7A FC 9D 01 A6 2F 03 A2 DE 96 37 93 6D 4A FE 68 09 0D 2D E1 8D 03
           F2 9C 88 CF B0 B1 BA 63 58 7F
       ------------------------------------------------------------------------
    3. Apple Root CA
       Expires: 2035-02-09 21:40:36 +0000
       SHA256 Fingerprint:
           B0 B1 73 0E CB C7 FF 45 05 14 2C 49 F1 29 5E 6E DA 6B CA ED 7E 2C
           68 C5 BE 91 B5 A1 10 01 F0 24
```

Apple has revoked the signature. In addition, the revoked code signing identifier, fenghua he (32W7BZNTSV), may help you find other malware signed by the same malware author.

Reverse Engineering pkgutil

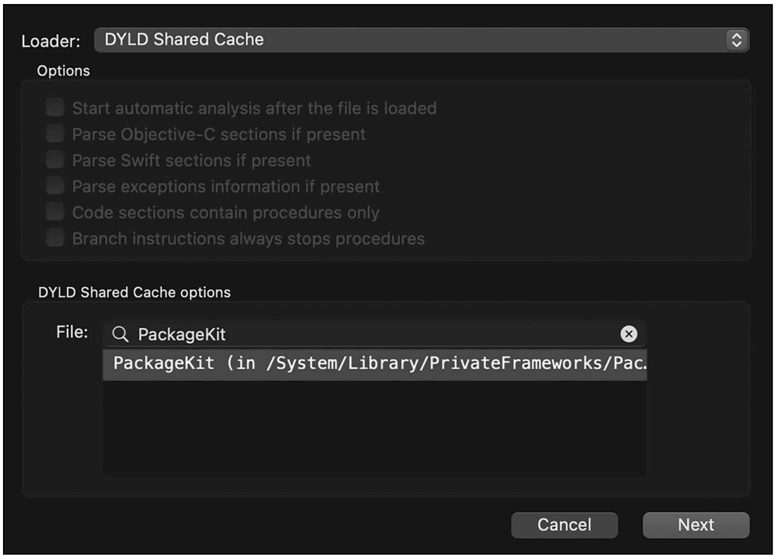
Now, you may be wondering how to programmatically check the signatures of packages. This is a good question, as there are currently no public APIs for verifying a package! Thanks, Cupertino.

Luckily, a quick reverse engineering session of the `pkgutil` binary reveals exactly how it checks the signature of packages. To begin, we can see that `pkgutil` is linked against the private *PackageKit* framework:

```
% otool -L /usr/sbin/pkgutil
/usr/sbin/pkgutil:
...
/System/Library/PrivateFrameworks/PackageKit.framework/Versions/A/PackageKit
...
```

The name of this framework suggests that it likely contains relevant APIs. Traditionally found in the */System/Library/PrivateFrameworks/* directory, the framework lives in the shared *dyld cache*, a prelinked shared file containing commonly used libraries, on recent versions of macOS.[16] Its name and location depend on the version of macOS and the architecture of the system but might look something like *dyld_shared_cache_arm64e* and */System/Volumes/Preboot/Cryptexes/OS/System/Library/dyld/*, respectively.

We must extract the *PackageKit* framework from the *dyld* cache before we can reverse engineer it. A tool such as Hopper, shown in Figure 3-2, can extract frameworks from the cache.


Loader: DYLD Shared Cache

Options

Start automatic analysis after the file is loaded
Parse Objective-C sections if present
Parse Swift sections if present
Parse exceptions information if present
Code sections contain procedures only
Branch instructions always stops procedures

DYLD Shared Cache options

File: PackageKit

PackageKit (in /System/Library/PrivateFrameworks/Pac.

Figure 3-2: Extracting the PackageKit *framework from the* dyld *cache*

If you prefer to use a command line tool to extract libraries, one good option is the *dyld-shared-cache-extractor*.[17] After installing this tool, you can execute it with the path of the *dyld* cache and an output directory, which we specify here as */tmp/libraries*:

```
% dyld-shared-cache-extractor /System/Volumes/Preboot/Cryptexes/OS/System/
Library/dyld/dyld_shared_cache_arm64e /tmp/libraries
```

Once the tool has extracted all of the libraries from the cache, you'll find the *PackageKit* framework at */tmp/libraries/System/Library/Private Frameworks/PackageKit.framework*.

Now we can load the framework into a disassembler to gain insight into its APIs and internals. For example, we find a class named `PKArchive` that contains useful methods, such as `archiveWithPath:` and `verifyReturningError:`, among others:

```
@interface PKArchive : NSObject
    +(id)archiveWithPath:(id)arg1;
    +(id)_allArchiveClasses;
    -(BOOL)closeArchive;
    -(BOOL)fileExistsAtPath:(id)arg1;
    -(BOOL)verifyReturningError:(id*)arg1;
    ...
@end
```

I won't cover the full details of reverse engineering the *PackageKit* framework here, but you can learn more about the process online.[18] You can also find the entirety of my package verification source code in my What's Your Sign utility's *Package.h/Package.m* file.[19]

Accessing Framework Functions

To use the methods we've discovered in our *checkSignature* project, we'll need a header file containing the private class definitions from the *PackageKit* framework. This will allow us to invoke them directly from our code. In the past, tools such as `class-dump` could easily create such header files,[20] but this approach isn't fully compatible with newer Apple Silicon binaries. Instead, you can manually extract these class definitions from a disassembler or by using `otool`. Listing 3-10 shows the extracted definitions.

```
@interface PKArchive : NSObject
    +(id)archiveWithPath:(id)arg1;
    +(id)_allArchiveClasses;
    -(BOOL)closeArchive;
    -(BOOL)fileExistsAtPath:(id)arg1;
    -(BOOL)verifyReturningError:(id*)arg1;
    ...

    @property(readonly) NSString* archiveDigest;
    @property(readonly) NSString* archivePath;
    @property(readonly) NSDate* archiveSignatureDate;
    @property(readonly) NSArray* archiveSignatures;
@end

@interface PKArchiveSignature : NSObject
{
    struct __SecTrust* _verifyTrustRef;
}

    -(struct __SecTrust*)verificationTrustRef;
    -(BOOL)verifySignedDataReturningError:(id *)arg1;
    -(BOOL)verifySignedData;
    ...

    @property(readonly) NSString* algorithmType;
    @property(readonly) NSArray* certificateRefs;
@end
...
```

Listing 3-10: The PackageKit *framework's extracted class and method definitions*

Now we can write code to use these classes, invoking their methods to programmatically verify packages of our choosing. We'll do this in a function we name `checkPackage`. As its only argument, it takes a path to the package to verify and returns a dictionary containing the results of verification, plus other code signing information, such as the package's code signing authorities. The function starts by loading the required *PackageKit* framework (Listing 3-11).

```
#define PACKAGE_KIT @"/System/Library/PrivateFrameworks/PackageKit.framework" ❶

NSMutableDictionary* checkPackage(NSString* package) {
    NSBundle* packageKit = [NSBundle bundleWithPath:PACKAGE_KIT]; ❷
    [packageKit load];
    ...
}
```

Listing 3-11: Loading the PackageKit *framework*

First, we define the path to the *PackageKit* framework ❶. We then load the framework with the `NSBundle` class's `bundleWithPath:` and `load` methods so that we can dynamically resolve and invoke the framework's methods ❷.

Due to its introspective nature, the Objective-C programming language makes it easy to use private classes and invoke private methods. To access a private class, use the `NSClassFromString` function. For example, Listing 3-12 shows how to dynamically obtain the class object for the `PKArchive` class.

```
Class PKArchiveCls = NSClassFromString(@"PKArchive");
```

Listing 3-12: Obtaining the `PKArchive` class object

Reverse engineering `pkgutil` revealed that it instantiates an archive object (`PKXARArchive`) using the `PKArchive` class's `archiveWithPath:` method, along with the path of the package to validate. In Listing 3-13, our code does the same.

```
PKXARArchive* archive = [PKArchiveCls archiveWithPath:package];
```

Listing 3-13: Instantiating an archive object

When dealing with private classes such as the `PKArchive` class, note that it's wise to invoke the `respondsToSelector:` method before invoking its methods. The `respondsToSelector:` method will return a Boolean value that tells you whether you can safely invoke the method on the class or class instance.[21] If you skip this step and an object doesn't respond to a method, it will crash your program with an `unrecognized selector sent to class` exception.

The following code checks to make sure the `PKArchive` class implements the `archiveWithPath:` method (Listing 3-14).

```
if(YES != [PKArchiveCls respondsToSelector:@selector(archiveWithPath:)]) {
    goto bail;
}
```

Listing 3-14: Checking for a method

Now we're ready to perform some basic package validation.

Validating the Package

Again, we mimic pkgutil by using the PKXARArchive class's verifyReturningError: method (Listing 3-15).

```
NSError* error = nil;
if(YES != [archive verifyReturningError:&error]) {
    goto bail;
}
```

Listing 3-15: Performing basic package validation

Once the package has passed basic verification checks, we can check its signature, which we find in the archive's archiveSignatures instance variable. This variable is an array holding pointers to PKArchiveSignature objects. A signed package will have at least one signature (Listing 3-16).

```
❶ NSArray* signatures = archive.archiveSignatures;
  if(0 == signatures.count) {
      goto bail;
  }

  PKArchiveSignature* signature = signatures.firstObject;
❷ if(YES != [signature verifySignedDataReturningError:&error]) {
      goto bail;
  }
```

Listing 3-16: Verifying a package's leaf signature

After ensuring that the package has at least one signature ❶, we verify the first, or *leaf*, signature, using the PKArchiveSignature class's verifySigned DataReturningError: method ❷. Additionally, we evaluate the trust of this signature (Listing 3-17).

```
  Class PKTrustCls = NSClassFromString(@"PKTrust");

  struct __SecTrust* trustRef = [signature verificationTrustRef];

❶ PKTrust* pkTrust = [[PKTrustCls alloc] initWithSecTrust:trustRef
  usingAppleRoot:YES signatureDate:archive.archiveSignatureDate];

❷ if(YES != [pkTrust evaluateTrustReturningError:&error]) {
      goto bail;
  }
```

Listing 3-17: Evaluating the trust of a signature

We instantiate a PKTrust object with the signature ❶ and then invoke the PKTrust class's evaluateTrustReturningError: method ❷. If verification TrustRef returns nil, we can validate the package via certificates by using the PKTrust class's initWithCertificates:usingAppleRoot:signatureDate: method. See this chapter's *checkSignature* project code for more details. If the signature and signature trust verifications pass, we have a validly signed package.

You could also extract the signature's certificates, which would allow you to perform actions like checking the name of each signing authority. You can access these certificates through the PKArchiveSignature object's certificateRefs instance variable, which is an array of SecCertificateRef objects, and extract their information with the SecCertificate* APIs.

Checking Package Notarization

I'll wrap up this section by showing how to determine whether Apple has notarized a package. Recall that pkgutil leverages the private *PackageKit* framework to validate packages. However, reverse engineering revealed that the package notarization checks aren't implemented in that framework with the rest of the checks, but rather directly in the pkgutil binary.

To check the notarization status of a package, pkgutil invokes the SecAssessmentTicketLookup API. Though this API is undocumented, we find its declaration in Apple's *SecAssessment.h* header file. Listing 3-18 mimics pkgutil's approach. Given a validated PKArchiveSignature object from a package, it determines whether the package has been notarized.

```
#import <CommonCrypto/CommonDigest.h>

typedef uint64_t SecAssessmentTicketFlags;
enum {
    kSecAssessmentTicketFlagDefault = 0,
    kSecAssessmentTicketFlagForceOnlineCheck = 1 << 0,
    kSecAssessmentTicketFlagLegacyListCheck = 1 << 1,
};

Boolean SecAssessmentTicketLookup(CFDataRef hash, SecCSDigestAlgorithm
hashType, SecAssessmentTicketFlags flags, double* date, CFErrorRef* errors);

BOOL isPackageNotarized(PKArchiveSignature* signature) {
    CFErrorRef error = NULL;
    BOOL isItemNotarized = NO;
    double notarizationDate = 0;

    SecCSDigestAlgorithm hashType = kSecCodeSignatureHashSHA1;

  ❶ NSData* hash = [signature signedDataReturningAlgorithm:0x0];
    if(CC_SHA1_DIGEST_LENGTH == hash.length) {
        hashType = kSecCodeSignatureHashSHA1;
    } else if(CC_SHA256_DIGEST_LENGTH == hash.length) {
        hashType = kSecCodeSignatureHashSHA256;
    }

  ❷ if(YES == SecAssessmentTicketLookup((__bridge CFDataRef)(hash), hashType,
    kSecAssessmentTicketFlagDefault, &notarizationDate, &error)) {
        isItemNotarized = YES;
  ❸ } else if(YES == SecAssessmentTicketLookup((__bridge CFDataRef)(hash),
    hashType, kSecAssessmentTicketFlagForceOnlineCheck, &notarizationDate,
    &error)) {
        isItemNotarized = YES;
    }

    return isItemNotarized;
}
```

Listing 3-18: A package notarization check

We declare various variables, most of which we'll need for the `SecAssessmentTicketLookup` API call. We then invoke the signature's `signedDataReturningAlgorithm:` method, which returns a data object containing a hash ❶.

Next, we make the first call to `SecAssessmentTicketLookup` ❷, passing it the hash and hash type, which will be either SHA-1 or SHA-256, represented by the `kSecCodeSignatureHashSHA1` and `kSecCodeSignatureHashSHA256` constants, respectively. We also pass in the assessment flags and an out pointer that will receive the date of the notarization if the package is notarized. The last argument is an optional out pointer to an error variable.

Mimicking the `pkgutil` binary, we first invoke the API with the assessment flags set to `kSecAssessmentTicketFlagDefault`. If this call fails to determine whether the package is notarized, we invoke the API again, this time with the flag set to `kSecAssessmentTicketFlagForceOnlineCheck` ❸. You can find these and other flag values in the *SecAssessment.h* header file.

If either API invocation returns a nonzero value, the package is notarized, and the Apple notary service trusts it. Because we mimicked `pkgutil`, however, our code doesn't specify whether a non-notarized package has had its notarization ticket revoked. Given an item's code signing hash and hash type, we could implement such a check in the manner shown in Listing 3-19.

```
CFErrorRef error = NULL;

if(YES != SecAssessmentTicketLookup(hash, hashType,
kSecAssessmentTicketFlagForceOnlineCheck, NULL, &error)) {
    if(EACCES == CFErrorGetCode(error)) {
        // Code placed here will run if the item's notarization ticket has been revoked.
    }
}
```

Listing 3-19: Checking for revoked notarization tickets

The `SecAssessmentTicketLookup` API will set its error variable to the value `EACCES` if the item's notarization ticket has been revoked.[22]

Running the Tool

Let's run the *checkSignature* tool against the packages mentioned earlier in this chapter:

```
% ./checkSignature GoogleChrome.pkg
Checking: GoogleChrome.pkg
```

```
Status: signed
Notarized: yes
Signing authorities (
    "<cert(0x11ee0ac30) s: Developer ID Installer: Google LLC (EQHXZ8M8AV)
    i: Developer ID Certification Authority>",
    "<cert(0x11ee08360) s: Developer ID Certification Authority i: Apple Root CA>",
    "<cert(0x11ee07820) s: Apple Root CA i: Apple Root CA>"
)

% ./checkSignature "EvilQuest/Mixed In Key 8.pkg"
Checking: Mixed In Key 8.pkg

Status: unsigned

% ./checkSignature KeySteal/archive.pkg
Checking: archive.pkg

Status: certificate revoked

Signing authorities: (
    "<cert(0x151406100) s: Developer ID Installer: fenghua he (32W7BZNTSV)
    i: Developer ID Certification Authority>",
    "<cert(0x151406380) s: Developer ID Certification Authority i: Apple Root CA>",
    "<cert(0x1514082b0) s: Apple Root CA i: Apple Root CA>"
)
```

The output matches the results of Apple's `pkgutil`. Our code accurately identifies the first package as validly signed and notarized; the second, containing the EvilQuest malware, as unsigned; and the last, containing the KeySteal malware, as revoked.

On-Disk Applications and Executables

The majority of macOS malware is distributed as applications or standalone Mach-O binaries. We can extract code signing information from an on-disk application bundle or executable binary in the same manner as for disk images: manually, via the `codesign` utility, or programmatically, via Apple's Code Signing Services APIs. However, this case presents a few important differences.

The first involves the `SecStaticCodeCheckValidity` API, which validates the item's signature. When the item isn't a disk image, we must invoke this function with the `kSecCSCheckAllArchitectures` flag (Listing 3-20).

```
SecCSFlags flags = kSecCSEnforceRevocationChecks;
if(NSOrderedSame != [item.pathExtension caseInsensitiveCompare:@"dmg"]) {
    flags |= kSecCSCheckAllArchitectures;
}
status = SecStaticCodeCheckValidity(staticCode, flags, NULL);
...
```

Listing 3-20: Checking an item's signature

This flag handles multiarchitecture items like universal binaries, which can include several embedded Mach-O binaries, potentially with different code signers. For a real-world example in which attackers abused a universal binary to bypass insufficient code signing checks, see CVE-2021-30773.[23] This flag value also enforces revocation checks, as it contains the value `kSecCSEnforceRevocationChecks`.

Earlier in this chapter, I showed you how to check whether a specified item conforms to some requirement, such as notarization. You might want to check additional requirements, such as whether Apple proper signed the item (the *anchor apple* requirement) or whether both Apple and a third-party developer ID have signed it (the *anchor apple generic* requirement). In each of these cases, your code can invoke the `SecRequirementCreateWithString` function with the requirement you wish to check and then pass this requirement to the `SecStaticCodeCheckValidity` API. To take into account universal binaries, invoke this function with a flag value that contains `kSecCSCheckAllArchitectures`.

You should also invoke the `SecAssessmentCreate` API to account for items with valid signatures but revoked notarization tickets. For a real-world example of this situation pertaining to applications, consider the 3CX supply chain attack mentioned previously. In this attack, North Korean attackers compromised the 3CX company network and build server, subverted the 3CX application with malware, signed it with the 3CX code signing certificate, and then tricked Apple into notarizing it. Not wanting to revoke 3CX's code signing certificate, which would have blocked many other legitimate 3CX apps, Apple merely revoked the subverted application's notarized ticket.

Let's run the *checkSignature* project on legitimate applications as well as malware, including the 3CX sample:

```
% ./checkSignature /Applications/LuLu.app
Checking: LuLu.app

Status: signed
Notarized: yes
Signing authorities: : (
    "<cert(0x13b814800) s: Developer ID Application: Objective-See, LLC (VBG97UB4TA)
    i: Developer ID Certification Authority>",
    "<cert(0x13b81c800) s: Developer ID Certification Authority i: Apple Root CA>",
    "<cert(0x13b81d000) s: Apple Root CA i: Apple Root CA>"
)

% ./checkSignature WindTail/Final_Presentation.app
Checking: Final_Presentation.app

Status: certificate revoked

% ./checkSignature "SmoothOperator/3CX Desktop App.app"
Checking: 3CX Desktop App.app
```

```
Status: signed
Notarized: revoked

% ./checkSignature MacMa/client
Checking: client

Status: unsigned
```

We first check Objective-See's signed and notarized LuLu application, followed by a WindTail malware specimen with a revoked certificate. Next, we test an instance of the trojanized 3CX application; our code correctly detects its revoked notarization status. Finally, we demonstrate that the MacMa malware is unsigned.

Running Processes

So far, we've examined on-disk items by obtaining static code object references. In this section, we'll check the code signing information of running processes by using dynamic code object references (`SecCodeRef`).

When applicable, you should make use of dynamic code object references for two reasons. The first is efficiency; the operating system will have already validated much of the code signing information for a dynamic instance of an item of interest to ensure conformance with runtime requirements. For us, this means we can avoid the costly file I/O operations associated with static code checks and skip certain computations.

The other reason that dynamic code references are preferable to static code references relates to possible discrepancies between an item's on-disk image and its in-memory one. For example, there is little stopping malware from changing the code signing information of its on-disk item to a benign value. (Of course, this highly anomalous behavior should itself raise a huge red flag.) On the other hand, a running item can't change its dynamic code signing information.

To check whether a running process is signed and then extract its code signing information, we first must obtain a code reference via the `SecCodeCopyGuestWithAttributes` API. Invoke it with the process's ID, or preferably, with a more secure process audit token (Listing 3-21).

```
SecCodeRef dynamicCode = NULL;

NSData* data = [NSData dataWithBytes:token length:sizeof(audit_token_t)]; ❶
NSDictionary* attributes = @{(__bridge NSString*)kSecGuestAttributeAudit:data}; ❷

status = SecCodeCopyGuestWithAttributes(NULL,
(__bridge CFDictionaryRef _Nullable)(attributes), kSecCSDefaultFlags, &dynamicCode); ❸
if(errSecSuccess != status) {
    goto bail;
}
```

Listing 3-21: Obtaining a code object reference via a process's audit token

We first convert the audit token into a data object ❶. We need this conversion so we can place the audit token in a dictionary, keyed by the string `kSecGuestAttributeAudit` ❷. We then pass this dictionary to the `SecCodeCopyGuestWithAttributes` API, along with an out pointer to populate with a code object reference ❸.

With a code object reference in hand, you can validate the process's code signing information with `SecCodeCheckValidity` or `SecCodeCheckValidityWithErrors`. Recall that for on-disk items such as universal binaries, we make use of the `kSecCSCheckAllArchitectures` flag value to validate all embedded Mach-Os; for running processes, the dynamic loader will load and execute only one embedded Mach-O, so that flag value is irrelevant and not needed.

It's essential that you validate a process's code signing information before extracting or acting upon any of it. If you don't, or if the validation fails, you won't be able to trust it. If the code signing information is valid, you can extract it via the `SecCodeCopySigningInformation` function that was already discussed.

With a code reference for a process, you can also perform other mundane but important tasks in a simple and secure manner. For example, using the `SecCodeCopyPath` API, you can retrieve the process's path (Listing 3-22).

```
CFURLRef path = NULL;
SecCodeCopyPath(dynamicCode, kSecCSDefaultFlags, &path);
```

Listing 3-22: Obtaining a process's path from a dynamic code object reference

You can also perform specific validations using requirements, as was discussed for static code object references. Using dynamic code object references, the approach is largely the same, except you'll make use of the `SecCodeCheckValidity` API to perform the validation. It is important to note that when you are done with a dynamic code reference, you should release it via `CFRelease`.

Because macOS won't allow a process to execute if either its certificate or its notarization ticket has been revoked, you don't need to perform this check yourself for running processes.

Detecting False Positives

At the beginning of the chapter, I noted that various antivirus engines had incorrectly flagged components of Apple's MRT as malware. If these engines had taken the item's code signing information into account, they would have identified MRT and its components as a built-in part of macOS signed solely by Apple proper and safely ignored it.

I'll show you how to perform such a check using the APIs introduced in this chapter. Specifically, you'll make use of the *anchor apple* requirement string, which holds cryptographically true if and only if nobody but Apple has signed an item.

Let's assume we've obtained a static code reference to the binary that was incorrectly flagged as malware. In Listing 3-23, we first compile the requirement string and then pass it and the code reference to the `SecStaticCodeCheckValidity` API.

```
static SecRequirementRef requirement = NULL;
SecRequirementCreateWithString(CFSTR("anchor apple"), kSecCSDefaultFlags, &requirement);

if(errSecSuccess ==
SecStaticCodeCheckValidity(staticCodeRef, kSecCSCheckAllArchitectures, requirement)) {
    // Code placed here will run only if the item is signed by Apple alone.
}
```

Listing 3-23: Checking the validity of an item against the anchor apple *requirement*

If `SecStaticCodeCheckValidity` returns `errSecSuccess`, we know that only Apple proper has signed the item, meaning it belongs to macOS and therefore certainly isn't malware.

Code Signing Error Codes

As mentioned throughout this chapter, it's important to appropriately handle any errors you encounter when validating an item's cryptographic signature. You can find the error codes for the code signing services APIs in Apple's "Code Signing Services Result Codes" developer documentation[24] or in the *CSCommon.h* file, found at *Security.framework/Versions/A/Headers/*. These resources indicate, for example, that the error code `-66992` maps to `errSecCSRevokedNotarization`, signifying that the code has been revoked.

If perusing header files isn't your thing, consult the OSStatus website. This website provides a simple way to map any Apple API error code to its human-readable name.

Conclusion

Code signing allows us to determine where an item is from and whether the item has been modified. In this chapter, you delved into code signing APIs that can verify, extract, and validate code signing information for items such as disk images, packages, on-disk binaries, and running processes.

Understanding these APIs is imperative in the context of detecting malware, especially as heuristic-based approaches can be fraught with false positives. The information provided by code signing can drastically reduce your detection errors. When building antimalware tools, you can use code signing in a myriad of ways, including identifying core operating system components you can trust, detecting items whose certificates or notarization tickets have been revoked, and authenticating clients, such as tool modules attempting to connect to XPC interfaces (a topic covered in Chapter 11).

Notes

1. Rich Trouton, "Apple Security Update Blocks Apple Ethernet Drivers on OS X El Capitan," *Der Flounder*, February 28, 2016, *https://derflounder.wordpress.com/2016/02/28/apple-security-update-blocks-apple-ethernet-drivers-on-el-capitan/*.
2. "Notarizing macOS Software Before Distribution," Apple Developer Documentation, *https://developer.apple.com/documentation/security/notarizing_macos_software_before_distribution*.
3. Patrick Wardle, "Apple Approved Malware," Objective-See, August 30, 2020, *https://objective-see.com/blog/blog_0x4E.html*.
4. You can read more about the revocation of developer certificates in Jeff Johnson, "Developer ID Certificate Revocation," *Lapcat Software*, October 29, 2020, *https://lapcatsoftware.com/articles/revocation.html*.
5. If you're interested in the technical details of code signing, see Jonathan Levin, "Code Signing—Hashed Out," *NewOSXBook*, April 20, 2015, *http://www.newosxbook.com/articles/CodeSigning.pdf*, or "macOS Code Signing in Depth," Apple Developer Documentation, *https://developer.apple.com/library/archive/technotes/tn2206/_index.html*.
6. "Code Signing Services," Apple Developer Documentation, *https://developer.apple.com/documentation/security/code_signing_services*.
7. "SecStaticCodeRef," Apple Developer Documentation, *https://developer.apple.com/documentation/security/secstaticcoderef?language=objc*.
8. "SecCodeRef," Apple Developer Documentation, *https://developer.apple.com/documentation/security/seccoderef?language=objc*.
9. "SecStaticCodeCreateWithPath," Apple Developer Documentation, *https://developer.apple.com/documentation/security/1396899-secstaticcodecreatewithpath*.
10. "Code Signing Services Result Codes," Apple Developer Documentation, *https://developer.apple.com/documentation/security/1574088-code_signing_services_result_cod*.
11. "Core Foundation Design Concepts," Apple Developer Documentation, *https://developer.apple.com/library/archive/documentation/CoreFoundation/Conceptual/CFDesignConcepts/Articles/tollFreeBridgedTypes.html*.
12. For a real-world example, see Ilias Morad, "CVE-2020–9854: 'Unauthd,'" Objective-See, August 1, 2020, *https://objective-see.org/blog/blog_0x4D.html*, which highlighted this issue in macOS's `authd`.
13. "SecRequirementCreateWithString," Apple Developer Documentation, *https://developer.apple.com/documentation/security/1394522-secrequirementcreatewithstring*.

14. "Code Signing Requirement Language," Apple Developer Documentation, *https://developer.apple.com/library/archive/documentation/Security/Conceptual/CodeSigningGuide/RequirementLang/RequirementLang.html.*

15. Asfdadsfasdfasdfsasdafads, "Programmatically Detected If a Notarization Ticket Has Been Revoked," Apple Developer Forums, June 2023, *https://developer.apple.com/forums/thread/731675.*

16. "dyld Shared Cache Info," Apple Developer Documentation, *https://developer.apple.com/forums/thread/692383.*

17. See *https://github.com/keith/dyld-shared-cache-extractor.*

18. See, for example, Patrick Wardle, "Reversing 'pkgutil' to Verify PKGs," *Jamf,* January 22, 2019, *https://www.jamf.com/blog/reversing-pkgutil-to-verify-pkgs/.*

19. See *https://github.com/objective-see/WhatsYourSign/blob/master/WhatsYourSignExt/FinderSync/Packages.m.*

20. Steve Nygard, "Class-dump," *http://stevenygard.com/projects/class-dump/.*

21. "respondsToSelector:," Apple Developer Documentation, *https://developer.apple.com/documentation/objectivec/1418956-nsobject/1418583-respondstoselector.*

22. "Notarization," Apple Developer Documentation, *https://opensource.apple.com/source/Security/Security-59306.120.7/OSX/libsecurity_codesigning/lib/notarization.cpp.*

23. Linus Henze, "Fugu15: The Journey to Jailbreaking iOS 15.4.1," paper presented at Objective by the Sea v5, Spain, October 6, 2022, *https://objectivebythesea.org/v5/talks/OBTS_v5_lHenze.pdf.*

24. "Code Signing Services Result Codes," Apple Developer Documentation, *https://developer.apple.com/documentation/security/1574088-code_signing_services_result_cod.*