Article

Validating receipts on the device

Verify the contents of app receipts by decoding and parsing the receipt on the device.

Overview

When users install apps from the App Store, the app contains a cryptographically signed receipt that Apple creates and stores inside the app bundle, which you can then validate.

> **Note**
>
> The receipt isn't necessary if you use `AppTransaction` to validate the app download, or `Transaction` to validate in-app purchases. Only use the receipt if your app uses the Original API for In-App Purchase, or needs the receipt to validate the app download because it can't use `AppTransaction`.

Validating the receipt locally requires you to develop or use code to read and decode the receipt as a PKCS #7 container, as defined by RFC 2315. The App Store encodes the payload of the container using Abstract Syntax Notation One (ASN.1), as defined by ITU-T X.690. The payload contains a set of receipt attributes. Each receipt attribute contains a type, a version, and a value.

The App Store defines the structure of the payload with the following ASN.1 notation:

```
ReceiptModule DEFINITIONS ::=
BEGIN

ReceiptAttribute ::= SEQUENCE {
    type    INTEGER,
    version INTEGER,
    value   OCTET STRING
```

```
}

Payload ::= SET OF ReceiptAttribute

END
```

Validate the receipt

In macOS and Mac apps built with Mac Catalyst, implement receipt validation in the main function, before the app calls `NSApplicationMain(_:_:)`.

To validate the app receipt, perform the following tests in order:

1. Locate and load the app receipt from the app's bundle. The class `Bundle` provides the location of the receipt with the property `appStoreReceiptURL`.
2. Decode the app receipt as a PKCS #7 container and verify that the chain of trust for the container's signature traces back to the Apple Inc. Root certificate, available from Apple PKI. Use the `receipt_creation_date`, identified as ASN.1 Field Type 12 when validating the receipt signature.
3. Verify that the bundle identifier, identified as ASN.1 Field Type 2, matches your app's bundle identifier.
4. Verify that the version identifier string, identified as ASN.1 Field Type 3, matches the version string in your app's bundle.
5. Compute a SHA-1 hash for the device that installs the app and verify that it matches the receipt's hash, identified as ASN.1 Field Type 5.

The validation passes if all of the tests pass. If any test fails, the validation fails.

For information about the keys in a receipt, see Receipt Fields.

Verify the certificate chain of trust

Decode the app receipt as a PKCS #7 container and verify that the chain of trust for the container's signature traces back to the Apple Inc. Root certificate, available from Apple PKI.

> **Tip**
>
> Don't hardcode intermediate certificates in your app. Ensure that your code supports certificates that use SHA-256 and SHA-1 signing algorithms.

Make sure your app uses the date from the `receipt_creation_date` field, identified as ASN.1 Field Type 12, to validate the receipt's signature. Many cryptographic libraries default to using the device's current time and date when validating a PKCS #7 package, but this may not produce the correct results when validating a receipt's signature. For example, if the receipt was signed with a valid certificate, but the certificate has since expired, using the device's current date incorrectly returns an invalid result.

Compute the SHA-1 hash

Compute the SHA-1 hash to match the local device with the device hash inside the App Store reciept. When computing the SHA-1 hash, use the platform-specific data source. The source of bytes for each platform is:

- watchOS: Use the raw bytes from the `uuid` property of the `UUID` that `identifierForVendor` provides.
- iOS, iPadOS, tvOS, and iOS apps running on a Mac with Apple silicon: Use the raw bytes from the `uuid` property of the `UUID` that `identifierForVendor` provides.
- macOS and apps built with Mac Catalyst: Use the data that returns from `copy_mac_address` from the example code below.

The following two code examples illustrate how to retrieve an identifier in macOS, as the `copy_mac_address` function shows, for validating an App Store receipt.

In the following Swift code, the `io_service` function uses IOKit to retrieve network interfaces as an optional IOKit object. The `copy_mac_address` function looks up an appropriate network interface and returns the hardware address from the IOKit object as optional `CFData`.

```
import IOKit
import Foundation

// Returns an object with a +1 retain count; the caller needs to release.
func io_service(named name: String, wantBuiltIn: Bool) -> io_service_t? {
    let default_port = kIOMasterPortDefault
    var iterator = io_iterator_t()
    defer {
        if iterator != IO_OBJECT_NULL {
            IOObjectRelease(iterator)
        }
    }

    guard let matchingDict = IOBSDNameMatching(default_port, 0, name),
        IOServiceGetMatchingServices(default_port,
```

```
                                    matchingDict as CFDictionary,
                                    &iterator) == KERN_SUCCESS,
        iterator != IO_OBJECT_NULL
    else {
        return nil
    }

    var candidate = IOIteratorNext(iterator)
    while candidate != IO_OBJECT_NULL {
        if let cftype = IORegistryEntryCreateCFProperty(candidate,
                                                        "IOBuiltin" as CFString,
                                                        kCFAllocatorDefault,
                                                        0) {
            let isBuiltIn = cftype.takeRetainedValue() as! CFBoolean
            if wantBuiltIn == CFBooleanGetValue(isBuiltIn) {
                return candidate
            }
        }

        IOObjectRelease(candidate)
        candidate = IOIteratorNext(iterator)
    }

    return nil
}

func copy_mac_address() -> CFData? {
    // Prefer built-in network interfaces.
    // For example, an external Ethernet adaptor can displace
    // the built-in Wi-Fi as en0.
    guard let service = io_service(named: "en0", wantBuiltIn: true)
            ?? io_service(named: "en1", wantBuiltIn: true)
            ?? io_service(named: "en0", wantBuiltIn: false)
        else { return nil }
    defer { IOObjectRelease(service) }

    if let cftype = IORegistryEntrySearchCFProperty(
        service,
        kIOServicePlane,
        "IOMACAddress" as CFString,
        kCFAllocatorDefault,
        IOOptionBits(kIORegistryIterateRecursively | kIORegistryIterateParents)) {
            return (cftype as! CFData)
```

```
    }

    return nil
}
```

The following Objective-C code works in the same fashion. This example uses IOKit to look up the relevant network interface, and returns the bytes that identify the built-in network interface:

```
#import <Foundation/Foundation.h>
#import <IOKit/network/IONetworkLib.h>

io_service_t io_service(const char *name, BOOL wantBuiltIn) {
    io_iterator_t iterator = IO_OBJECT_NULL;
    mach_port_t default_port = kIOMasterPortDefault;
    io_service_t service = IO_OBJECT_NULL;

    if (KERN_SUCCESS != IOMasterPort(MACH_PORT_NULL, &default_port)) {
        return IO_OBJECT_NULL;
    }

    CFMutableDictionaryRef matchingDict = IOBSDNameMatching(default_port,
                                                            0,
                                                            name);
    if (matchingDict == NULL) {
        return IO_OBJECT_NULL;
    }

    if (KERN_SUCCESS != IOServiceGetMatchingServices(default_port,
                                                     matchingDict,
                                                     &iterator)) {
        return IO_OBJECT_NULL;
    }

    if (iterator != IO_OBJECT_NULL) {
        io_service_t candidate = IOIteratorNext(iterator);
        while (candidate != IO_OBJECT_NULL) {
            CFTypeRef isBuiltIn =
            IORegistryEntryCreateCFProperty(candidate,
                                            CFSTR(kIOBuiltin),
                                            kCFAllocatorDefault,
                                            0);
            if (isBuiltIn != NULL && CFGetTypeID(isBuiltIn) == CFBooleanGetTypeID())
                if (wantBuiltIn == CFBooleanGetValue(isBuiltIn)) {
```

```
                    service = candidate;
                    break;
                }
            }

            IOObjectRelease(candidate);
            candidate = IOIteratorNext(iterator);
        }
        IOObjectRelease(iterator);
    }

    return service;
}

CFDataRef copy_mac_address() {
    CFDataRef macAddress = NULL;
    io_service_t service = io_service("en0", true);

    if (service == IO_OBJECT_NULL) {
        service = io_service("en1", true);
    }

    if (service == IO_OBJECT_NULL) {
        service = io_service("en0", false);
    }

    if (service != IO_OBJECT_NULL) {
        CFTypeRef property =
        IORegistryEntrySearchCFProperty(service,
                                        kIOServicePlane,
                                        CFSTR(kIOMACAddress),
                                        kCFAllocatorDefault,
                                        kIORegistryIterateRecursively | kIORegistryI
        if (property != NULL) {
            if (CFGetTypeID(property) == CFDataGetTypeID()) {
                macAddress = property;
            }
            else {
                CFRelease(property);
            }
        }

        IOObjectRelease(service);
```

```
    }

    return macAddress;
}
```

Respond to validation failures

If your app receipt validation fails, respond to that failure as follows:

- Don't try to terminate the app. Without a validated receipt, assume the user doesn't have access to premium content. Provide a user interface to gracefully handle this case and inform the user what they can do to get full access to your app's features.
- If the app receipt is missing or corrupt, use the `SKReceiptRefreshRequest` object to refresh the app receipt.
- In the sandbox environment, if the app receipt is missing, assume the tester is a new customer and doesn't have access to premium content.

> **Note**
>
> For apps in iOS and iPadOS running in the sandbox environment and in StoreKit testing in Xcode, the app receipt is present only after the tester makes their first in-app purchase. The app receipt is always present in TestFlight on devices running macOS.