Receipt Validation Programming Guide

Developer

Contents

Figures, Tables, and Listings

About Receipt Validation

Note: This book was previously titled *Validating Mac App Store Receipts*.

The receipt for an application or in-app purchase is a record of the sale of the application and of any in-app purchases made from within the application. You can add receipt validation code to your application to prevent unauthorized copies of your application from running. Refer to the license agreement and the review guidelines for specific information about what your application may and may not do to implement copy protection.

Receipt validation requires an understanding of cryptography and a variety of secure coding techniques. It's important that you employ a solution that is unique to your application.

At a Glance

There are two ways to validate receipts: locally and with the App Store. Compare both approaches and determine which is a better fit for your app and your infrastructure. You can also choose to implement both approaches.

Validating Receipts Locally

Validating locally requires code to read and validate a PKCS #7 signature, and code to parse and validate the signed payload.

Relevant Chapters: Validating Receipts Locally (page 6), Receipt Fields (page 23)

Validating Receipts With the App Store

Validating with the App Store requires a secure connection between your app and your server, and code on your server to to validate the receipt with the App Store.

Relevant Chapters: Validating Receipts With the App Store (page 19), Receipt Fields (page 23)

Validating Receipts Locally

Perform receipt validation immediately after your app is launched, before displaying any user interface or spawning any child processes. Implement this check in the `main` function, before the `NSApplicationMain` function is called. For additional security, you may repeat this check periodically while your application is running.

Locate and Parse the Receipt

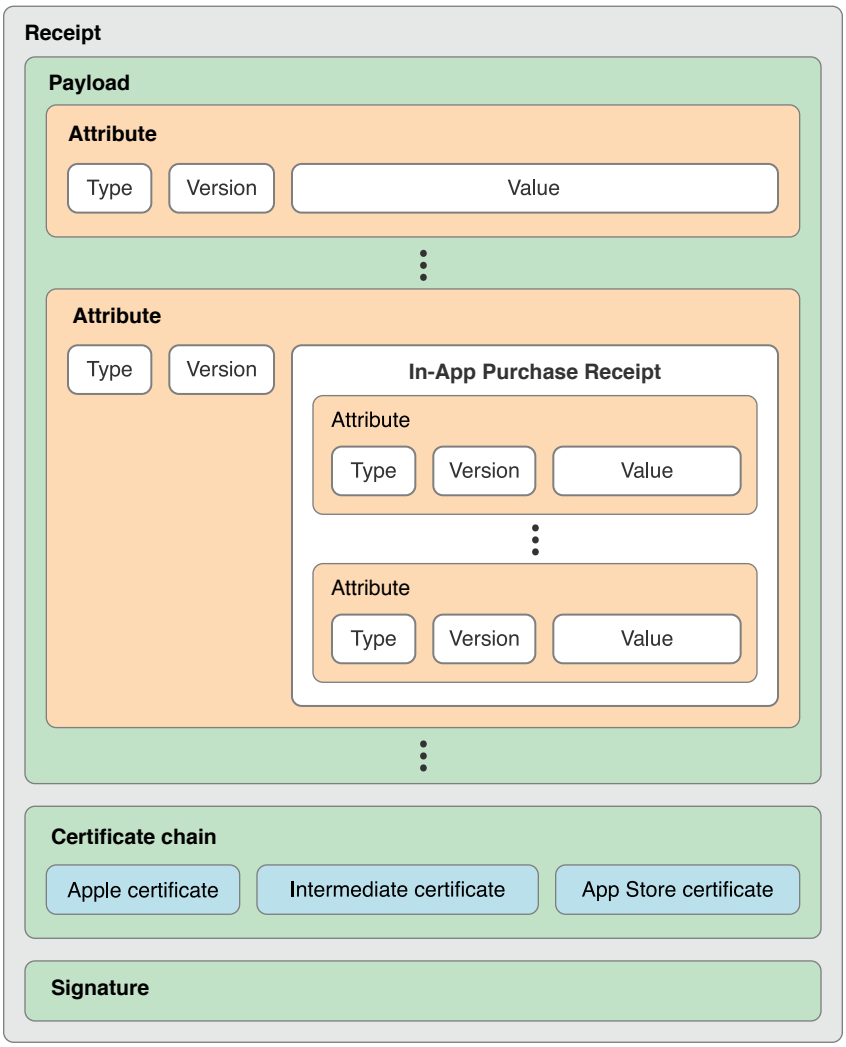
When an application is installed from the App Store, it contains an application receipt that is cryptographically signed, ensuring that only Apple can create valid receipts. The receipt is stored inside the application bundle. Call the `appStoreReceiptURL` method of the `NSBundle` class to locate the receipt.

Note: In OS X, if the `appStoreReceiptURL` method is not available (on older systems), you can fall back to a hardcoded path. The receipt's path is `/Contents/_MASReceipt/receipt` inside the app bundle.

In iOS, if the `appStoreReceiptURL` method is not available (on older systems), you can fall back to validating the `transactionReceipt` property of an `SKPaymentTransaction` object with the App Store. For details, see Validating Receipts With the App Store (page 19).

The receipt is a binary file with the structure shown in Figure 1-1.

Figure 1-1 Structure of a receipt



The outermost portion (labeled *Receipt* in the figure) is a PKCS #7 container, as defined by RFC 2315, with its payload encoded using ASN.1 (Abstract Syntax Notation One), as defined by ITU-T X.690. The payload is composed of a set of *receipt attributes*. Each receipt attribute contains a type, a version, and a value.

The structure of the payload is defined using ASN.1 notation in Listing 1-1. You can use this definition with the `asn1c` tool to generate data type declarations and functions for decoding the payload, rather than writing that part of your code by hand. You may need to install `asn1c` first; it is available through MacPorts and SourceForge.

For information about keys found in a receipt, see Receipt Fields (page 23).

To generate the code, save the payload description shown in Listing 1-1 to a file and, in Terminal, run the following command:

```
asn1c -fnative-types filename
```

After the `asn1c` tool finishes generating files in the current directory, add the files it generated to your Xcode project.

Listing 1-1 ASN.1 definition of the payload format

```
ReceiptModule DEFINITIONS ::=
BEGIN

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

Payload ::= SET OF ReceiptAttribute

END
```

Compute the Hash of the GUID

In OS X, use the method described in Get the GUID in OS X (page 13) to fetch the computer's GUID.

In iOS, use the value returned by the `identifierForVendor` property of `UIDevice` as the computer's GUID.

To compute the hash, first concatenate the GUID value with the opaque value (the attribute of type 4) and the bundle identifier. Use the raw bytes from the receipt without performing any UTF-8 string interpretation or normalization. Then compute the SHA-1 hash of this concatenated series of bytes.

Validate the Receipt

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

1. Locate the receipt.

   If no receipt is present, validation fails.

2. Verify that the receipt is properly signed by Apple.

   If it is not signed by Apple, validation fails.

3. Verify that the bundle identifier in the receipt matches a hard-coded constant containing the `CFBundleIdentifier` value you expect in the `Info.plist` file.

   If they do not match, validation fails.

4. Verify that the version identifier string in the receipt matches a hard-coded constant containing the `CFBundleShortVersionString` value you expect in the `Info.plist` file.

   If they do not match, validation fails.

5. Compute the hash of the GUID as described in Compute the Hash of the GUID (page 8).

   If the result does not match the hash in the receipt, validation fails.

If all of the tests pass, validation passes.

> **Note:** Bundle identifiers and version identifier strings are UTF-8 strings, not just a series of bytes. Make sure you code your comparison logic accordingly.

If your app supports the Volume Purchase Program, check the receipt's expiration date.

Respond to Receipt Validation Failure

Validation can fail for a variety of reasons. For example, when users copy your application from one Mac to another, the GUID no longer matches, causing receipt validation to fail.

Exit If Validation Fails in OS X

If validation fails in OS X, call `exit` with a status of `173`. This exit status notifies the system that your application has determined that its receipt is invalid. At this point, the system attempts to obtain a valid receipt and may prompt for the user's iTunes credentials.

If the system successfully obtains a valid receipt, it relaunches the application. Otherwise, it displays an error message to the user, explaining the problem.

Do not display any error message to the user if validation fails. The system is responsible for trying to obtain a valid receipt or informing the user that the receipt is not valid.

Refresh the Receipt If Validation Fails in iOS

If validation fails in iOS, use the `SKReceiptRefreshRequest` class to refresh the receipt.

Do not try to terminate the app. At your option, you may give the user a grace period or restrict functionality inside your app.

Set a Minimum System Version for Mac Apps

Include the `LSMinimumSystemVersion` key with a value of 10.6.6 or greater in your application's `Info.plist` file. If receipt validation fails on versions of OS X earlier than 10.6.6, your application quits immediately after launch with no explanation to the user. Earlier versions of OS X do not interpret the exit status of `173`, so they do not try to obtain a valid receipt or display any error message.

Don't Localize Your Version Number

If your application is localized, the `CFBundleShortVersionString` key should not appear in any of your application's `InfoPlist.strings` files. The unlocalized value from your `Info.plist` file is stored in the receipt—attempting to localize the value for this key can cause receipt validation to fail.

Protect Your Validation Check

An attacker may try to circumvent the validation code by patching your application binary or altering the basic operating system routines that the validation code depends upon. Resilience against these types of attacks requires a variety of coding techniques, including the following:

- Inline the code for cryptographic checks instead of using the APIs provided by the system.

- Avoid simple code constructions that provide a trivial target for patching the application binary.

  For example, avoid writing code like the following:

```
if (failedValidation) {
   exit(173);
}
```

- Implement code robustness techniques, such as obfuscation.

  If multiple applications use the same code for performing validation, this common code signature can be targeted by tools that patch application binaries.

- Ensure that, even if the `exit` function fails to terminate your application, your application stops running.

Test During the Development Process

In order to test your main application during the development process, you need a valid receipt so that your application launches. To set this up, do the following:

1. Make sure you have Internet access so you can connect to Apple's servers.
2. Launch your application by double-clicking on it (or in some way cause Launch Services to launch it).

After you launch your application, the following occurs:

- Your application fails to validate its receipt because there is no receipt present, and it exits with a status of `173`.
- The system interprets the exit status and attempts to obtain a valid receipt. Assuming your application signing certificate is valid, the system installs a valid receipt for the application. The system may prompt you for your iTunes credentials.
- The system relaunches your application, and your application successfully validates the receipt.

With this development receipt installed, you can launch your application by any method—for example, with `gdb` or the Xcode debugger.

Validate In-App Purchases

To validate an in-app purchase, your application performs the following tests, in order:

1. Parse and validate the application's receipt, as described in the previous sections.

If the receipt is not valid, none of the in-app purchases are valid.

2. Parse the in-app purchase receipts (the values for the attributes of type 17).

   Each in-app purchase receipt consists of a set of attributes, like the application's receipt does. The structure for these receipts is defined in Listing 1-2 (page 12). As when parsing the receipt, you can generate some of your code from the ASN.1 description using the `asn1c` tool. Ignore all attributes with types that do not appear in the table—they are reserved for use by the system and their contents may change at any time.

   For information about the fields in a receipt, see Receipt Fields (page 23).

3. Examine the product identifier for each in-app purchase receipt and enable the corresponding functionality or content in your app. For information about how to calculate a subscription's active period, see Working with Subscriptions.

If validation of an in-app purchase receipt fails, your application simply does not enable the functionality or content.

Listing 1-2 ASN.1 definition of the in-app purchase receipt format

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

InAppReceipt ::= SET OF InAppAttribute
```

The attributes for the original transaction identifier and original transaction date are used when a purchase is redownloaded. The redownloaded purchase is given a new transaction identifier, but it contains the identifier and date of the original purchase.

Implementation Tips

This section contains several code listings for your reference as you implement receipt validation.

Get the GUID in OS X

In OS X, follow the model in Listing 1-3 (or even use this exact same code), so that the method you use to get the GUID in your validation code is exactly the same as the method used when the application's receipt was created.

Listing 1-3 Get the computer's GUID

```
#import <IOKit/IOKitLib.h>
#import <Foundation/Foundation.h>

// Returns a CFData object, containing the computer's GUID.
CFDataRef copy_mac_address(void)
{
    kern_return_t             kernResult;
    mach_port_t               master_port;
    CFMutableDictionaryRef    matchingDict;
    io_iterator_t             iterator;
    io_object_t               service;
    CFDataRef                 macAddress = nil;

    kernResult = IOMasterPort(MACH_PORT_NULL, &master_port);
    if (kernResult != KERN_SUCCESS) {
        printf("IOMasterPort returned %d\n", kernResult);
        return nil;
    }

    matchingDict = IOBSDNameMatching(master_port, 0, "en0");
    if (!matchingDict) {
        printf("IOBSDNameMatching returned empty dictionary\n");
        return nil;
    }

   kernResult = IOServiceGetMatchingServices(master_port, matchingDict, &iterator);
    if (kernResult != KERN_SUCCESS) {
        printf("IOServiceGetMatchingServices returned %d\n", kernResult);
        return nil;
```

```
    }

    while((service = IOIteratorNext(iterator)) != 0) {
        io_object_t parentService;

        kernResult = IORegistryEntryGetParentEntry(service, kIOServicePlane,
                &parentService);
        if (kernResult == KERN_SUCCESS) {
            if (macAddress) CFRelease(macAddress);

          macAddress = (CFDataRef) IORegistryEntryCreateCFProperty(parentService,
                    CFSTR("IOMACAddress"), kCFAllocatorDefault, 0);
            IOObjectRelease(parentService);
        } else {
            printf("IORegistryEntryGetParentEntry returned %d\n", kernResult);
        }

        IOObjectRelease(service);
    }
    IOObjectRelease(iterator);

    return macAddress;
}
```

Parse the Receipt and Verify Its Signature

Use the following code listings as an outline of one possible implementation of receipt validation using OpenSSL and `asn1c`. These listings are provided to guide you as you write your own code, by highlighting relevant APIs and data structures, not to use as a copy-and-paste solution.

If you use OpenSSL, statically link your binary against it. Dynamic linking against OpenSSL is deprecated and results in build warnings.

Make sure your code does the following as outlined in the listings:

1. Verify the signature (Listing 1-4 (page 15)).
2. Parse the payload (Listing 1-5 (page 16)).

3. Extract the receipt attributes (Listing 1-6 (page 16)).
4. Compute the hash of the GUID (Listing 1-7 (page 17)).

Listing 1-4 Verify the signature using OpenSSL

```
/* The PKCS #7 container (the receipt) and the output of the verification. */
BIO *b_p7;
PKCS7 *p7;

/* The Apple root certificate, as raw data and in its OpenSSL representation. */
BIO *b_x509;
X509 *Apple;

/* The root certificate for chain-of-trust verification. */
X509_STORE *store = X509_STORE_new();

/* ... Initialize both BIO variables using BIO_new_mem_buf() with a buffer and its
 size ... */

/* Initialize b_out as an output BIO to hold the receipt payload extracted during
 signature verification. */
BIO *b_out = BIO_new(BIO_s_mem());

/* Capture the content of the receipt file and populate the p7 variable with the
PKCS #7 container. */
p7 = d2i_PKCS7_bio(b_p7, NULL);

/* ... Load the Apple root certificate into b_X509 ... */

/* Initialize b_x509 as an input BIO with a value of the Apple root certificate and
 load it into X509 data structure. Then add the Apple root certificate to the
structure. */
Apple = d2i_X509_bio(b_x509, NULL);
X509_STORE_add_cert(store, Apple);

/* Verify the signature. If the verification is correct, b_out will contain the
PKCS #7 payload and rc will be 1. */
int rc = PKCS7_verify(p7, NULL, store, NULL, b_out, 0);
```

```
/* For additional security, you may verify the fingerprint of the root certificate
 and verify the OIDs of the intermediate certificate and signing certificate. The
 OID in the certificate policies extension of the intermediate certificate is (1
2 840 113635 100 5 6 1), and the marker OID of the signing certificate is (1 2 840
 113635 100 6 11 1). */
```

Listing 1-5 Parse the payload using asn1c

```
#include "Payload.h" /* This header file is generated by asn1c. */


/* The receipt payload and its size. */

void *pld = NULL;

size_t pld_sz;


/* Variables used to parse the payload. Both data types are declared in Payload.h.
 */

Payload_t *payload = NULL;

asn_dec_rval_t rval;


/* ... Load the payload from the receipt file into pld and set pld_sz to the payload
 size ... */


/* Parse the buffer using the decoder function generated by asn1c.  The payload
variable will contain the receipt attributes. */

rval = asn_DEF_Payload.ber_decoder(NULL, &asn_DEF_Payload, (void **)&payload, pld,
 pld_sz, 0);
```

Listing 1-6 Extract the receipt attributes

```
/* Variables used to store the receipt attributes. */

OCTET_STRING_t *bundle_id = NULL;

OCTET_STRING_t *bundle_version = NULL;

OCTET_STRING_t *opaque = NULL;

OCTET_STRING_t *hash = NULL;


/* Iterate over the receipt attributes, saving the values needed to compute the
GUID hash. */
```

```
size_t i;
for (i = 0; i < payload->list.count; i++) {
    ReceiptAttribute_t *entry;

    entry = payload->list.array[i];

    switch (entry->type) {
        case 2:
            bundle_id = &entry->value;
            break;
        case 3:
            bundle_version = &entry->value;
            break;
        case 4:
            opaque = &entry->value;
            break;
        case 5:
            hash = &entry->value;
            break;
        }
    }
```

Listing 1-7 Compute the hash of the GUID

```
/* The GUID returned by copy_mac_address() is a CFDataRef.  Use CFDataGetBytePtr()
 and CFDataGetLength() to get a pointer to the bytes that make up the GUID and to
 get its length. */
UInt8 *guid = NULL;
size_t guid_sz;

/* Declare and initialize an EVP context for OpenSSL. */
EVP_MD_CTX evp_ctx;
EVP_MD_CTX_init(&evp_ctx);

/* A buffer for result of the hash computation. */
UInt8 digest[20];
```

```
/* Set up the EVP context to compute a SHA-1 digest. */
EVP_DigestInit_ex(&evp_ctx, EVP_sha1(), NULL);

/* Concatenate the pieces to be hashed.  They must be concatenated in this order.
 */
EVP_DigestUpdate(&evp_ctx, guid, guid_sz);
EVP_DigestUpdate(&evp_ctx, opaque->buf, opaque->size);
EVP_DigestUpdate(&evp_ctx, bundle_id->buf, bundle_id->size);

/* Compute the hash, saving the result into the digest variable. */
EVP_DigestFinal_ex(&evp_ctx, digest, NULL);
```

Validating Receipts With the App Store

Note: There is a vulnerability in iOS 5.1 and earlier related to receipt validation with the app store directly from a device, without using a server. For more details and a mitigation strategy, see *In-App Purchase Receipt Validation on iOS*.

Use a trusted server to communicate with the App Store. Using your own server lets you design your app to recognize and trust only your server, and lets you ensure that your server connects with the App Store server. It is not possible to build a trusted connection between a user's device and the App Store directly because you don't control either end of that connection.

Communication with the App Store is structured as JSON dictionaries, as defined in RFC 4627. Binary data is base64 encoded, as defined in RFC 4648.

Read the Receipt Data

To retrieve the receipt data, use the `appStoreReceiptURL` method of `NSBundle` to locate the app's receipt, and then read the entire file. If the `appStoreReceiptURL` method is not available, you can fall back to the value of a transaction's `transactionReceipt` property for backward compatibility. Then send this data to your server—as with all interactions with your server, the details are your responsibility.

```
// Load the receipt from the app bundle.
NSURL *receiptURL = [[NSBundle mainBundle] appStoreReceiptURL];
NSData *receipt = [NSData dataWithContentsOfURL:receiptURL];
if (!receipt) { /* No local receipt -- handle the error. */ }

/* ... Send the receipt data to your server ... */
```

Send the Receipt Data to the App Store

On your server, create a JSON object with the following keys:

Key	Value
`receipt-data`	The base64 encoded receipt data.
`password`	*Only used for receipts that contain auto-renewable subscriptions.* Your app's shared secret (a hexadecimal string).

Submit this JSON object as the payload of an HTTP POST request. In the test environment, use `https://sandbox.itunes.apple.com/verifyReceipt` as the URL. In production, use `https://buy.itunes.apple.com/verifyReceipt` as the URL.

```
NSData *receipt; // Sent to the server by the device

// Create the JSON object that describes the request
NSError *error;
NSDictionary *requestContents = @{
    @"receipt-data": [receipt base64EncodedStringWithOptions:0]
};
NSData *requestData = [NSJSONSerialization dataWithJSONObject:requestContents
                                                       options:0
                                                         error:&error];

if (!requestData) { /* ... Handle error ... */ }

// Create a POST request with the receipt data.
NSURL *storeURL = [NSURL
URLWithString:@"https://buy.itunes.apple.com/verifyReceipt"];
NSMutableURLRequest *storeRequest = [NSMutableURLRequest requestWithURL:storeURL];
[storeRequest setHTTPMethod:@"POST"];
[storeRequest setHTTPBody:requestData];

// Make a connection to the iTunes Store on a background queue.
NSOperationQueue *queue = [[NSOperationQueue alloc] init];
[NSURLConnection sendAsynchronousRequest:storeRequest queue:queue
        completionHandler:^(NSURLResponse *response, NSData *data, NSError
*connectionError) {
```

```
    if (connectionError) {
        /* ... Handle error ... */
    } else {
        NSError *error;
        NSDictionary *jsonResponse = [NSJSONSerialization JSONObjectWithData:data
 options:0 error:&error];
        if (!jsonResponse) { /* ... Handle error ...*/ }
        /* ... Send a response back to the device ... */
    }
}];
```

Parse the Response

The response's payload is a JSON object that contains the following keys and values:

Key	Value
`status`	Either `0` if the receipt is valid, or one of the error codes listed in Table 2-1 (page 22). For iOS 6 style transaction receipts, the status code reflects the status of the specific transaction's receipt. For iOS 7 style app receipts, the status code is reflects the status of the app receipt as a whole. For example, if you send a valid app receipt that contains an expired subscription, the response is `0` because the receipt as a whole is valid.
`receipt`	A JSON representation of the receipt that was sent for verification. For information about keys found in a receipt, see Receipt Fields (page 23).
`latest_receipt`	*Only returned for iOS 6 style transaction receipts for auto-renewable subscriptions.* The base-64 encoded transaction receipt for the most recent renewal.
`latest_receipt_info`	*Only returned for iOS 6 style transaction receipts for auto-renewable subscriptions.* The JSON representation of the receipt for the most recent renewal.

Table 2-1 Status codes

Status Code	Description
21000	The App Store could not read the JSON object you provided.
21002	The data in the `receipt-data` property was malformed or missing.
21003	The receipt could not be authenticated.
21004	The shared secret you provided does not match the shared secret on file for your account. *Only returned for iOS 6 style transaction receipts for auto-renewable subscriptions.*
21005	The receipt server is not currently available.
21006	This receipt is valid but the subscription has expired. When this status code is returned to your server, the receipt data is also decoded and returned as part of the response. *Only returned for iOS 6 style transaction receipts for auto-renewable subscriptions.*
21007	This receipt is from the test environment, but it was sent to the production environment for verification. Send it to the test environment instead.
21008	This receipt is from the production environment, but it was sent to the test environment for verification. Send it to the production environment instead.

The values of the `latest_receipt` and `latest_receipt_info` keys are useful when checking whether an auto-renewable subscription is currently active. By providing any transaction receipt for the subscription and checking these values, you can get information about the currently-active subscription period. If the receipt being validated is for the latest renewal, the value for `latest_receipt` is the same as `receipt-data` (in the request) and the value for `latest_receipt_info` is the same as `receipt`.

Receipt Fields

Receipts are made up of a number of fields. Some fields are only available locally, in the ASN.1 form of the receipt, or only when validating with the App Store, in the JSON form of the receipt. Keys not documented below are reserved for use by Apple and must be ignored by your app.

App Receipt Fields

Bundle Identifier

The app's bundle identifier.

ASN.1 Field Type 2

ASN.1 Field Value UTF8STRING

JSON Field Name bundle_id

JSON Field Value string

This corresponds to the value of `CFBundleIdentifier` in the `Info.plist` file.

App Version

The app's version number.

ASN.1 Field Type 3

ASN.1 Field Value UTF8STRING

JSON Field Name application_version

JSON Field Value string

This corresponds to the value of `CFBundleVersion` (in iOS) or `CFBundleShortVersionString` (in OS X) in the `Info.plist`.

Opaque Value

An opaque value used, with other data, to compute the SHA-1 hash during validation.

ASN.1 Field Type 4

ASN.1 Field Value A series of bytes

JSON Field Name (none)

JSON Field Value (none)

SHA-1 Hash

A SHA-1 hash, used to validate the receipt.

ASN.1 Field Type 5

ASN.1 Field Value 20-byte SHA-1 digest

JSON Field Name (none)

JSON Field Value (none)

In-App Purchase Receipt

The receipt for an in-app purchase.

ASN.1 Field Type 17

ASN.1 Field Value SET of in-app purchase receipt attributes

JSON Field Name in_app

JSON Field Value array of in-app purchase receipts

In the JSON file, the value of this key is an array containing all in-app purchase receipts. In the ASN.1 file, there are multiple fields that all have type 17, each of which contains a single in-app purchase receipt.

The in-app purchase receipt for a consumable product is added to the receipt when the purchase is made. It is kept in the receipt until your app finishes that transaction. After that point, it is removed from the receipt the next time the receipt is updated—for example, when the user makes another purchase or if your app explicitly refreshes the receipt.

The in-app purchase receipt for a non-consumable product, auto-renewable subscription, non-renewing subscription, or free subscription remains in the receipt indefinitely.

Original Application Version

The version of the app that was originally purchased.

ASN.1 Field Type 19

ASN.1 Field Value UTF8STRING

JSON Field Name original_application_version

JSON Field Value string

This corresponds to the value of `CFBundleVersion` (in iOS) or `CFBundleShortVersionString` (in OS X) in the `Info.plist` file when the purchase was originally made.

In the sandbox environment, the value of this field is always "1.0".

Receipts prior to June 20, 2013 omit this field. It is populated on all new receipts, regardless of OS version. If you need the field but it is missing, manually refresh the receipt using the `SKReceiptRefreshRequest` class.

Receipt Expiration Date

The date that the app receipt expires.

ASN.1 Field Type 21

ASN.1 Field Value IA5STRING, interpreted as an RFC 3339 date

JSON Field Name expiration_date

JSON Field Value IA5STRING, interpreted as an RFC 3339 date

This key is present only for apps purchased through the Volume Purchase Program. If this key is not present, the receipt does not expire.

When validating a receipt, compare this date to the current date to determine whether the receipt is expired. Do not try to use this date to calculate any other information, such as the time remaining before expiration.

In-App Purchase Receipt Fields

Quantity

The number of items purchased.

ASN.1 Field Type 1701

ASN.1 Field Value INTEGER

JSON Field Name quantity

JSON Field Value string, interpreted as an integer

This value corresponds to the `quantity` property of the `SKPayment` object stored in the transaction's `payment` property.

Product Identifier

The product identifier of the item that was purchased.

ASN.1 Field Type 1702

ASN.1 Field Value UTF8STRING

JSON Field Name product_id

JSON Field Value string

This value corresponds to the `productIdentifier` property of the `SKPayment` object stored in the transaction's `payment` property.

Transaction Identifier

The transaction identifier of the item that was purchased.

ASN.1 Field Type 1703

ASN.1 Field Value UTF8STRING

JSON Field Name transaction_id

JSON Field Value string

This value corresponds to the transaction's `transactionIdentifier` property.

Original Transaction Identifier

For a transaction that restores a previous transaction, the transaction identifier of the original transaction. Otherwise, identical to the transaction identifier.

ASN.1 Field Type 1705

ASN.1 Field Value UTF8STRING

JSON Field Name original_transaction_id

JSON Field Value string

This value corresponds to the original transaction's `transactionIdentifier` property.

All receipts in a chain of renewals for an auto-renewable subscription have the same value for this field.

Purchase Date

The date and time that the item was purchased.

ASN.1 Field Type 1704

ASN.1 Field Value IA5STRING, interpreted as an RFC 3339 date

JSON Field Name purchase_date

JSON Field Value string, interpreted as an RFC 3339 date

This value corresponds to the transaction's `transactionDate` property.

For a transaction that restores a previous transaction, the purchase date is the date of the restoration. Use Original Purchase Date (page 27) to get the date of the original transaction.

In an auto-renewable subscription receipt, this is always the date when the subscription was purchased or renewed, regardless of whether the transaction has been restored.

Original Purchase Date

For a transaction that restores a previous transaction, the date of the original transaction.

ASN.1 Field Type 1706

ASN.1 Field Value IA5STRING, interpreted as an RFC 3339 date

JSON Field Name original_purchase_date

JSON Field Value string, interpreted as an RFC 3339 date

This value corresponds to the original transaction's `transactionDate` property.

In an auto-renewable subscription receipt, this indicates the beginning of the subscription period, even if the subscription has been renewed.

Subscription Expiration Date

The expiration date for the subscription, expressed as the number of milliseconds since January 1, 1970, 00:00:00 GMT.

ASN.1 Field Type 1708

ASN.1 Field Value IA5STRING, interpreted as an RFC 3339 date

JSON Field Name expires_date

JSON Field Value number

This key is only present for auto-renewable subscription receipts.

Cancellation Date

For a transaction that was canceled by Apple customer support, the time and date of the cancellation.

ASN.1 Field Type 1712

ASN.1 Field Value IA5STRING, interpreted as an RFC 3339 date

JSON Field Name cancellation_date

JSON Field Value string, interpreted as an RFC 3339 date

Treat a canceled receipt the same as if no purchase had ever been made.

App Item ID

A string that the App Store uses to uniquely identify the application that created the transaction.

ASN.1 Field Type (none)

ASN.1 Field Value (none)

JSON Field Name app_item_id

JSON Field Value string

If your server supports multiple applications, you can use this value to differentiate between them.

Apps are assigned an identifier only in the production environment, so this key is not present for receipts created in the test environment.

This field is not present for Mac apps.

See also Bundle Identifier (page 23).

External Version Identifier

An arbitrary number that uniquely identifies a revision of your application.

ASN.1 Field Type (none)

ASN.1 Field Value (none)

JSON Field Name version_external_identifier

JSON Field Value string

This key is not present for receipts created in the test environment.

Web Order Line Item ID

The primary key for identifying subscription purchases.

ASN.1 Field Type 1711

ASN.1 Field Value INTEGER

JSON Field Name web_order_line_item_id

JSON Field Value string

Document Revision History

This table describes the changes to *Receipt Validation Programming Guide*.

Date	Notes
2014-11-18	Noted that non-renewing subscriptions remain in the receipt in In-App Purchase Receipt (page 24).
2014-02-11	Expanded discussion of server side receipt validation in Validating Receipts With the App Store (page 19).
2013-09-18	Added new receipt fields. Incorporated information about server-side validation. Formerly titled *Validating Mac App Store Receipts*. Incorporated content from a previous revision of *In-App Purchase Programming Guide*.
2012-01-09	Minor updates for OS X v10.7.
2011-07-07	New document that describes how an application can validate its receipt.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010