SecItem: Fundamentals

This thread has been locked by a moderator.

274

I regularly help developers with keychain problems, both here on DevForums and for my Day Job™ in DTS. Many of these problems are caused by a fundamental misunderstanding of how the keychain works. This post is my attempt to explain that. I wrote it primarily so that Future Quinn™ can direct folks here rather than explain everything from scratch (-:

If you have questions or comments about any of this, put them in a new thread and apply the *Security* tag so that I see it.

Share and Enjoy
—
Quinn "The Eskimo!" @ Developer Technical Support @ Apple
`let myEmail = "eskimo" + "1" + "@" + "apple.com"`

SecItem: Fundamentals

or *How I Learned to Stop Worrying and Love the SecItem API*

The SecItem API seems very simple. After all, it only has four function calls, how hard can it be? In reality, things are not that easy. Various factors contribute to making this API much trickier than it might seem at first glance.

This post explains the fundamental underpinnings of the keychain. For information about specific issues, see its companion post, SecItem Pitfalls and Best Practices.

Keychain Documentation

Your basic starting point should be Keychain Items.

If your code runs on the Mac, also read TN3137 On Mac keychain APIs and implementations.

Read the doc comments in `<Security/SecItem.h>`. In many cases those doc comments contain critical tidbits.

When you read keychain documentation [1] and doc comments, keep in mind that statements specific to iOS typically apply to iPadOS, tvOS, and watchOS as well (r. 102786959). Also, they typically apply to macOS *when you target the data protection keychain*. Conversely, statements specific to macOS may not apply when you target the data protection keychain.

[1] Except TN3137, which is very clear about this (-:

Caveat Mac Developer

macOS supports two different implementations: the original file-based keychain and the iOS-style data protection keychain. If you're able to use the data protection keychain, do so. It'll make your life easier.

TN3137 On Mac keychain APIs and implementations explains this distinction in depth.

The Four Freedoms^H^H^H^H^H^H^H Functions

The SecItem API contains just four functions:

- `SecItemAdd(_:_:)`
- `SecItemCopyMatching(_:_:)`
- `SecItemUpdate(_:_:)`
- `SecItemDelete(_:)`

These directly map to standard SQL database operations:

- `SecItemAdd(_:_:)` maps to `INSERT`.
- `SecItemCopyMatching(_:_:)` maps to `SELECT`.
- `SecItemUpdate(_:_:)` maps to `UPDATE`.
- `SecItemDelete(_:)` maps to `DELETE`.

You can think of each keychain item class (generic password, certificate, and so on) as a separate SQL table within the database. The rows of that table are the individual keychain items for that class and the columns are the attributes of those items.

Note Except for the digital identity class, `kSecClassIdentity`, where the values are split across the certificate and key tables. See *Digital Identities Aren't Real* in SecItem Pitfalls and Best Practices.

This is not an accident. The data protection keychain is actually implemented as an SQLite database. If you're curious about its structure, examine it on the Mac by pointing your favourite SQLite inspection tool — for example, the `sqlite3` command-line tool — at the keychain database in `~/Library/Keychains/UUU/keychain-2.db`, where `UUU` is a UUID.

WARNING Do not depend on the location and structure of this file. These have changed in the past and are likely to change again in the future. If you embed knowledge of them into a shipping product, it's likely that your product will have binary compatibility problems at some point in the future. The only reason I'm mentioning them here is because I find it helpful to poke around in the file to get a better understanding of how the API works.

Uniqueness

A critical part of the keychain model is uniqueness. How does the keychain determine if item A is the same as item B? It turns out that this is class dependent. For each keychain item class there is a set of attributes that form the uniqueness constraint for items of that class. That is, if you try to add item A where all of its attributes are the same as item B, the add fails with `errSecDuplicateItem`. For more information, see the `errSecDuplicateItem` page. It has lists of attributes that make up this uniqueness constraint, one for each class.

These uniqueness constraints are a major source of confusion, as discussed in the *Queries and the Uniqueness Constraints* section of SecItem Pitfalls and Best Practices.

Parameter Blocks Understanding

The SecItem API is a classic 'parameter block' API. All of its inputs are dictionaries, and you have to know which properties to set in each dictionary to achieve your desired result. Likewise for when you read properties in output dictionaries.

There are five different **property groups**:

- The **item class property**, `kSecClass`, determines the class of item you're operating on: `kSecClassGenericPassword`, `kSecClassCertificate`, and so on.
- The **item attribute properties**, like `kSecAttrAccessGroup`, map directly to keychain item attributes.
- The **search properties**, like `kSecMatchLimit`, control how the system runs a query.
- The **return type properties**, like `kSecReturnAttributes`, determine what values the query returns.
- The **value type properties**, like `kSecValueRef` perform multiple duties, as explained below.

There are other properties that perform a variety of specific functions. For example, `kSecUseDataProtectionKeychain` tells macOS to use the data protection keychain instead of the file-based keychain. These properties are hard to describe in general; for the details, see the documentation for each such property.

Inputs

Each of the four SecItem functions take dictionary input parameters of the same type, `CFDictionary`, but these dictionaries are not the same. Different dictionaries support different property groups:

- The first parameter of `SecItemAdd(_:_:)` is an **add dictionary**. It supports all property groups except the search properties.
- The first parameter of `SecItemCopyMatching(_:_:)` is a **query and return dictionary**. It supports all property groups.
- The first parameter of `SecItemUpdate(_:_:)` is a **pure query dictionary**. It supports all property groups except the return type properties.
- Likewise for the only parameter of `SecItemDelete(_:)`.
- The second parameter of `SecItemUpdate(_:_:)` is an **update dictionary**. It supports the item attribute and value type property groups.

Outputs

Two of the SecItem functions, `SecItemAdd(_:_:)` and `SecItemCopyMatching(_:_:)`, return values. These output parameters are of type `CFTypeRef` because the type of value you get back depends on the return type properties you supply in the input dictionary:

- If you supply a single return type property, except `kSecReturnAttributes`, you get back a value appropriate for that return type.
- If you supply multiple return type properties or `kSecReturnAttributes`, you get back a dictionary. This supports the item attribute and value type property groups. To get a non-attribute value from this dictionary, use the value type property that corresponds to its return type property. For example, if you set `kSecReturnPersistentRef` in the input dictionary, use `kSecValuePersistentRef` to get the persistent reference from the output dictionary.

In the single item case, the type of value you get back depends on the return type property and the keychain item class:

- For `kSecReturnData` you get back the keychain item's data. This makes most sense for password items, where the data holds the password. It also works for certificate items, where you get back the DER-encoded certificate. Using this for key items is kinda sketchy. If you want to export a key, called `SecKeyCopyExternalRepresentation`. Using this for digital identity items is nonsensical.
- For `kSecReturnRef` you get back an object reference. This only works for keychain item classes that have an object representation, namely certificates, keys, and digital identities. You get back a `SecCertificate`, a `SecKey`, or a `SecIdentity`, respectively.
- For `kSecReturnPersistentRef` you get back a data value that holds the persistent reference.

Value Type Subtleties

There are three properties in the value type property group:

- `kSecValueData`
- `kSecValueRef`
- `kSecValuePersistentRef`

Their semantics vary based on the dictionary type.

For `kSecValueData`:

- In an add dictionary, this is the value of the item to add. For example, when adding a generic password item (`kSecClassGenericPassword`), the value of this key is a `Data` value containing the password.
- This is not supported in a query dictionary.
- In an update dictionary, this is the new value for the item.

For `kSecValueRef`:

- In add and query dictionaries, the system infers the class property and attribute properties from the supplied object. For example, if you supply a certificate object (`SecCertificate`, created using `SecCertificateCreateWithData`), the system will infer a `kSecClass` value of `kSecClassCertificate` and various attribute values, like `kSecAttrSerialNumber`, from that certificate object.
- This is not supported in an update dictionary.

For `kSecValuePersistentRef`:

- For query dictionaries, this uniquely identifies the item to operate on.
- This is not supported in add and update dictionaries.

Revision History

- **2022-02-22** Made minor editorial changes.

Security

Reply

Posted 3 months ago by eskimo

Add a Comment