SecItem: Pitfalls and Best Practices

This thread has been locked by a moderator.

806

If you’re on macOS and targeting the file-based keychain, kSecMatchLimitAll always defaults to kSecMatchLimitOne

I regularly help developers with keychain problems, both here on DevForums and for my Day Job™ in DTS. Over the years I’ve learnt a lot about the API, including many pitfalls and best practices. This post is my attempt to collect that experience in one place.

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: Pitfalls and Best Practices

It’s just four functions, how hard can it be?

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 some of the keychain’s pitfalls and then goes on to explain various best practices. Before reading this, make sure you understand the fundamentals by reading its companion post, SecItem: Fundamentals.

Pitfalls

Lets start with some common pitfalls.

Queries and Uniqueness Constraints

The relationship between query dictionaries and uniqueness constraints is a major source of problems with the keychain API. Consider code like this:

```
var copyResult: CFTypeRef? = nil
let query = [
    kSecClass: kSecClassGenericPassword,
    kSecAttrService: "AYS",
    kSecAttrAccount: "mrgumby",
    kSecAttrGeneric: Data("SecItemHints".utf8),
] as NSMutableDictionary
let err = SecItemCopyMatching(query, &copyResult)
if err == errSecItemNotFound {
    query[kSecValueData] = Data("opendoor".utf8)
    let err2 = SecItemAdd(query, nil)
    if err2 == errSecDuplicateItem {
        fatalError("… can you get here? …")
    }
}
```

Can you get to the fatal error?

At first glance this might not seem possible because you’ve run your query and it’s returned `errSecItemNotFound`. However, the fatal error *is* possible because the query contains an attribute, `kSecAttrGeneric`, that does not contribute to the uniqueness. If the keychain contains a generic password whose service (`kSecAttrService`) and account (`kSecAttrAccount`) attributes match those supplied but who’s generic (`kSecAttrGeneric`) attribute does not, the `SecItemCopyMatching` calls will return `errSecItemNotFound`. However, for a generic password item, of the attributes shown here, only the service and account attributes are included in the uniqueness constraint. If you try to add an item where those attributes match an existing item, the add will fail with `errSecDuplicateItem` even though the value of the generic attribute is different.

The take-home point is that that you should study the attributes that contribute to uniqueness and use them in a way that’s aligned with your view of uniqueness. See the *Uniqueness* section of SecItem: Fundamentals for a link to the relevant documentation.

Erroneous Attributes

Each keychain item class supports its own specific set of attributes. For information about the attributes supported by a given class, see SecItem: Fundamentals.

I regularly see folks use attributes that aren’t supported by the class they’re working with. For example, the `kSecAttrApplicationTag` attribute is only supported for key items (`kSecClassKey`). Using it with a certificate item (`kSecClassCertificate`) will cause, at best, a runtime error and, at worst, mysterious bugs.

This is an easy mistake to make because:

- The ‘parameter block’ nature of the SecItem API means that the compiler won’t complain if you use an erroneous attribute.
- On macOS, the shim that connects to the file-based keychain ignores unsupported attributes.

Imagine you want to store a certificate for a particular user. You might write code like this:

```
let err = SecItemAdd([
    kSecClass: kSecClassCertificate,
    kSecAttrApplicationTag: Data(name.utf8),
    kSecValueRef: cert,
] as NSDictionary, nil)
```

The goal is to store the user’s name in the `kSecAttrApplicationTag` attribute so that you can get back their certificate with code like this:

```
let err = SecItemCopyMatching([
kSecClass: kSecClassCertificate,
kSecAttrApplicationTag: Data(name.utf8),
kSecReturnRef: true,
] as NSDictionary, &copyResult)
```

On iOS, and with the data protection keychain on macOS, both calls will fail with `errSecNoSuchAttr`. That makes sense, because the `kSecAttrApplicationTag` attribute is not supported for certificate items. Unfortunately, the macOS shim that connects the SecItem API to the file-based keychain ignores extraneous attributes. This results in some *very* bad behaviour:

- `SecItemAdd` works, ignoring `kSecAttrApplicationTag`.
- `SecItemCopyMatching` ignores `kSecAttrApplicationTag`, returning the first certificate that it finds.

If you only test with a single user, everything seems to work. But, later on, when you try your code with multiple users, you might get back the wrong result depending on the which certificate the `SecItemCopyMatching` call happens to discover first.

Ouch!

Context Matters

Some properties change behaviour based on the context. The value type properties are the biggest offender here, as discussed in the *Value Type Subtleties* section of SecItem: Fundamentals. However, there are others.

The one that’s bitten me is `kSecMatchLimit`:

- In a query and return dictionary its default value is `kSecMatchLimitOne`. If you don’t supply a value for `kSecMatchLimit`, `SecItemCopyMatching` returns at most one item that matches your query.
- In a pure query dictionary its default value is `kSecMatchLimitAll`. For example, if you don’t supply a value for `kSecMatchLimit`, `SecItemDelete` will delete *all* items that match your query. This is a lesson that, once learnt, is never forgotten!

Note Although this only applies to the data-protection keychain. If you’re on macOS and targeting the file-based keychain, `kSecMatchLimit` always defaults to `kSecMatchLimitOne` (r. 105800863). Fun times!

Digital Identities Aren’t Real

A digital identity is the combination of a certificate and the private key that matches the public key within that certificate. The SecItem API has a digital identity keychain item class, namely `kSecClassIdentity`. However, the keychain does not store digital identities. When you add a digital identity to the keychain, the system stores its components, the certificate and the private key, separately, using `kSecClassCertificate` and `kSecClassKey` respectively.

This has a number of non-obvious effects:

- Adding a certificate can ‘add’ a digital identity. If the new certificate happens to match a private key that’s already in the keychain, the keychain treats that pair as a digital identity.
- Likewise when you add a private key.
- Similarly, removing a certificate or private key can ‘remove’ a digital identity.
- Adding a digital identity will either add a private key, or a certificate, or both, depending on what’s already in the keychain.
- Removing a digital identity removes its certificate. It might also remove the private key, depending on whether that private key is used by a different digital identity.

Keys Aren’t Stored in the Secure Enclave

Apple platforms let you protect a key with the Secure Enclave (SE). The key is then *hardware bound*. It can only be used by that specific SE [1].

Earlier versions of the Protecting keys with the Secure Enclave article implied that SE-protected keys were stored in the SE itself. This is not true, and it’s caused a lot of confusion. For example, I once asked the keychain team “How much space does the SE have available to store keys?”, a question that’s complete nonsense once you understand how this works.

In reality, SE-protected keys are stored in the standard keychain database alongside all your other keychain items. The difference is that the key is wrapped in such a way that only the SE can use it. So, the key is *protected by* the SE, not *stored in* the SE.

A while back we updated the docs to clarify this point but the confusion persists.

[1] Technically it’s that specific iteration of that specific SE. If you erase the device then the key material needed to use the key is erased and so the key becomes permanently useless. This is the sort of thing you’ll find explained in Apple Platform Security.

Careful With that Shim, Mac Developer

As explained in TN3137 On Mac keychain APIs and implementations, macOS has a shim that connects the SecItem API to either the data protection keychain or the file-based keychain depending on the nature of the request. That shim has limitations. Some of those are architectural but others are simply bugs in the shim. For some great examples, see the *Investigating Complex Attributes* section below.

The best way to avoid problems like this is to target the data protection keychain. If you can’t do that, try to avoid exploring the outer reaches of the SecItem API. If you encounter a case that doesn’t make sense, try that same case with the data protection keychain. If it works there but fails with the file-based keychain, please do file a bug against the shim. It’ll be in good company.

Add-only Attributes

Some attributes can only be set when you add an item. These attributes are usually associated with the scope of the item. For example, to protect an item with the Secure Enclave, supply the `kSecAttrAccessControl` attribute to the `SecItemAdd` call. Once you do that, however, you can’t change the attribute. Calling `SecItemUpdate` with a new `kSecAttrAccessControl` won’t work.

Best Practices

With the pitfalls out of the way, let’s talk about best practices.

Less Painful Dictionaries

I look at a lot of keychain code and it’s amazing how much of it is *way* more painful than it needs to be. The biggest offender here is the dictionaries. Here are two tips to minimise the pain.

First, don’t use `CFDictionary`. It’s seriously ugly. While the SecItem API is defined in terms of `CFDictionary`, you don’t have to work with `CFDictionary` directly. Rather, use `NSDictionary` and take advantage of the toll-free bridge.

For example, consider this `CFDictionary` code:

```
CFTypeRef keys[4] = {
    kSecClass,
    kSecAttrService,
    kSecMatchLimit,
    kSecReturnAttributes,
};
static const int kTen = 10;
CFNumberRef ten = CFNumberCreate(NULL, kCFNumberIntType, &kTen);
CFAutorelease(ten);
CFTypeRef values[4] = {
    kSecClassGenericPassword,
    CFSTR("AYS"),
    ten,
    kCFBooleanTrue,
};
CFDictionaryRef query = CFDictionaryCreate(
    NULL,
    keys,
    values,
    4,
    &kCFTypeDictionaryKeyCallBacks,
    &kCFTypeDictionaryValueCallBacks
);
```

Note This might seem rather extreme but I’ve literally seen code like this, and worse, while helping developers.

Contrast this to the equivalent `NSDictionary` code:

```
NSDictionary * query = @{
    (__bridge NSString *) kSecClass: (__bridge NSString *) kSecClassGenericPassword,
    (__bridge NSString *) kSecAttrService: @"AYS",
    (__bridge NSString *) kSecMatchLimit: @10,
    (__bridge NSString *) kSecReturnAttributes: @YES,
};
```

Wow, that’s so much better.

Second, if you’re working in Swift, take advantage of its awesome ability to create `NSDictionary` values from Swift dictionary literals. Here’s the equivalent code in Swift:

```
let query = [
    kSecClass: kSecClassGenericPassword,
    kSecAttrService: "AYS",
    kSecMatchLimit: 10,
    kSecReturnAttributes: true,
] as NSDictionary
```

Nice!

Avoid Reusing Dictionaries

I regularly see folks reuse dictionaries for different SecItem calls. For example, they might have code like this:

```
var copyResult: CFTypeRef? = nil
let dict = [
    kSecClass: kSecClassGenericPassword,
    kSecAttrService: "AYS",
    kSecAttrAccount: "mrgumby",
    kSecReturnData: true,
] as NSMutableDictionary
var err = SecItemCopyMatching(dict, &copyResult)
if err == errSecItemNotFound {
    dict[kSecValueData] = Data("opendoor".utf8)
    err = SecItemAdd(dict, nil)
}
```

This specific example will work, but it’s easy to spot the logic error. `kSecReturnData` is a return type property and it makes no sense to pass it to a `SecItemAdd` call whose second parameter is `nil`.

I’m not sure why folks do this. I think it’s because they think that constructing dictionaries is expensive. Regardless, this pattern can lead to all sorts of weird problems. For example, it’s the leading cause of the issue described in the *Queries and the Uniqueness Constraints* section, above.

My advice is that you use a new dictionary for each call. That prevents state from one call accidentally leaking into a subsequent call. For example, I’d rewrite the above as:

```
var copyResult: CFTypeRef? = nil
let query = [
    kSecClass: kSecClassGenericPassword,
    kSecAttrService: "AYS",
    kSecAttrAccount: "mrgumby",
    kSecReturnData: true,
] as NSMutableDictionary
var err = SecItemCopyMatching(query, &copyResult)
if err == errSecItemNotFound {
    let add = [
        kSecClass: kSecClassGenericPassword,
        kSecAttrService: "AYS",
        kSecAttrAccount: "mrgumby",
        kSecValueData: Data("opendoor".utf8),
    ] as NSMutableDictionary
    err = SecItemAdd(add, nil)
}
```

It’s a bit longer, but it’s much easier to track the flow. And if you want to eliminate the repetition, use a helper function:

```
func makeDict() -> NSMutableDictionary {
    [
        kSecClass: kSecClassGenericPassword,
        kSecAttrService: "AYS",
        kSecAttrAccount: "mrgumby",
    ] as NSMutableDictionary
}

var copyResult: CFTypeRef? = nil
let query = makeDict()
query[kSecReturnData] = true
var err = SecItemCopyMatching(query, &copyResult)
if err == errSecItemNotFound {
    let add = makeDict()
    query[kSecValueData] = Data("opendoor".utf8)
    err = SecItemAdd(add, nil)
}
```

Think Before Wrapping

A lot of folks look at the SecItem API and immediately reach for a wrapper library. A keychain wrapper library might seem like a good idea but there are some serious downsides:

- It adds another dependency to your project.
- Different subsystems within your project may use different wrappers.
- The wrapper can obscure the underlying API. Indeed, its entire raison d’être *is* to obscure the underlying API. This is problematic if things go wrong. I regularly talk to folks with hard-to-debug keychain problems and the conversation goes something like this:

  Quinn: What attributes do you use in the query dictionary?

  J R Developer: What’s a query dictionary?

  Quinn: OK, so what error are you getting back?

  J R Developer: It throws `WrapperKeychainFailedError`.

  That’s not helpful)-:

  If you do use a wrapper, make sure it has diagnostic support that includes the values passed to and from the SecItem API. Also make sure that, when it fails, it returns an error that includes the underlying keychain error code. These benefits will be particularly useful if you encounter a keychain problem that only shows up in the field.
- Wrappers must choose whether to be general or specific. A general wrapper may be harder to understand than the equivalent SecItem calls, and it’ll certainly contain a lot of complex code. On the other hand, a specific wrapper may have a model of the keychain that doesn’t align with your requirements.

I recommend that you think twice before using a keychain wrapper. Personally I find the SecItem API relatively easy to call, assuming that:

- I use the techniques shown in *Less Painful Dictionaries*, above, to avoid having to deal with `CFDictionary`.
- I use my `secCall(…)` helpers to simplify error handling. For the code, see Calling Security Framework from Swift.

If you’re not prepared to take the SecItem API neat, consider writing your own wrapper, one that’s tightly focused on the requirements of your project. For example, in my VPN apps I use the wrapper from this post, which does exactly what I need in about 100 lines of code.

Prefer to Update

Of the four SecItem functions, `SecItemUpdate` is the most neglected. Rather than calling `SecItemUpdate` I regularly see folks delete and then re-add the item. This is a shame because `SecItemUpdate` has some important benefits:

- It preserves persistent references. If you delete and then re-add the item, you get a new item with a new persistent reference.
- It’s well aligned with the fundamental database nature of the keychain. It forces you to think about which attributes uniquely identify your item and which items can be updated without changing the item’s identity.

Understand These Key Attributes

Key items have a number of attributes that are similarly named, and it’s important to keep them straight. I created a cheat sheet for this, namely, SecItem attributes for keys. You wouldn’t believe how often I consult this!

Investigating Complex Attributes

Some attributes have values where the format is not obvious. For example, the `kSecAttrIssuer` attributed is documented as:

> The corresponding value is of type `CFData` and contains the X.500 issuer name of a certificate.

What exactly does that mean? If I want to search the keychain for all certificates issued by a specific certificate authority, what value should I supply?

One way to figure this out is to add a certificate to the keychain, read the attributes back, and then dump the `kSecAttrIssuer` value. For example:

```
let cert: SecCertificate = …
let attrs = try secCall { SecItemAdd([
    kSecValueRef: cert,
    kSecReturnAttributes: true,
] as NSDictionary, $0) } as! [String: Any]
let issuer = attrs[kSecAttrIssuer as String] as! NSData
print((issuer as NSData).debugDescription)
// prints: <3110300e 06035504 030c074d 6f757365 4341310b 30090603 55040613 024742>
```

Those bytes represent the *contents* of a X.509 `Name` ASN.1 structure with DER encoding. This is without the outer `SEQUENCE` element, so if you dump it as ASN.1 you’ll get a nice dump of the first `SET` and then a warning about extra stuff at the end of the file:

```
% xxd issuer.asn1
00000000: 3110 300e 0603 5504 030c 074d 6f75 7365  1.0...U....Mouse
00000010: 4341 310b 3009 0603 5504 0613 0247 42    CA1.0...U....GB
% dumpasn1 -p issuer.asn1
SET {
  SEQUENCE {
    OBJECT IDENTIFIER commonName (2 5 4 3)
    UTF8String 'MouseCA'
    }
  }
Warning: Further data follows ASN.1 data at position 18.
```

Note For details on the `Name` structure, see section 4.1.2.4 of RFC 5280.

Amusingly, if you run the same test against the file-based keychain you’ll… crash. OK, that’s not amusing. It turns out that the code above doesn’t work when targeting the file-based keychain because `SecItemAdd` doesn’t return a dictionary but rather an array of dictionaries (r. 21111543). Once you get past that, however, you’ll see it print:

```
<301f3110 300e0603 5504030c 074d6f75 73654341 310b3009 06035504 06130247 42>
```

Which is different! Dumping it as ASN.1 shows that it’s the full `Name` structure, including the outer `SEQUENCE` element:

```
% xxd issuer-file-based.asn1
00000000: 301f 3110 300e 0603 5504 030c 074d 6f75  0.1.0...U....Mou
00000010: 7365 4341 310b 3009 0603 5504 0613 0247  seCA1.0...U....G
00000020: 42                                       B
% dumpasn1 -p issuer-file-based.asn1
SEQUENCE {
  SET {
    SEQUENCE {
      OBJECT IDENTIFIER commonName (2 5 4 3)
      UTF8String 'MouseCA'
      }
    }
  SET {
    SEQUENCE {
      OBJECT IDENTIFIER countryName (2 5 4 6)
      PrintableString 'GB'
      }
    }
  }
```

This difference in behaviour between the data protection and file-based keychains is a known bug (r. 26391756) but in this case it’s handy because the file-based keychain behaviour makes it easier to understand the data protection keychain behaviour.

Revision History

- **2023-09-22** Made minor editorial changes.
- **2023-09-12** Fixed various bugs in the revision history. Added the *Erroneous Attributes* section.
- **2023-02-22** Fixed the link to the `VPNKeychain` post. Corrected the name of the *Context Matters* section. Added the *Investigating Complex Attributes* section.
- **2023-01-28** First posted.

Security

Reply

Posted 8 months ago by eskimo

Add a Comment