Passwords

CHRIS "MAC" MCENIRY

Chris "Mac" McEniry is a practicing sysadmin responsible for running a large e-commerce and gaming service. He's been working and developing in an operational capacity for 15 years. In his free time, he builds tools and thinks about efficiency. cmceniry@mit.edu

Passwords. Everybody hates them, but everyone still uses them.

While there are pushes for certificates, OTP, and other forms of authentications, the password is still king. In addition to the ubiquity of passwords, good practice dictates that we use different passwords for every account silo, and (controversially) we are supposed to change them often.

Most corporate environments use a single account silo, and the use of single sign-on systems has the promise of not needing to authenticate regularly. However, most of the time, these systems are more consistent sign-on instead of single sign-on, so you end up typing your password over and over again. On the plus side, this is convenient in quickly updating muscle memory following password changes.

The above reasons have given rise to the heavy use of personal password storage. There are online services which provide this in bulk. Most operating systems provide some form of personal password storage: Windows has credential manager, OSX has Keychain, and Linux has several depending on the distribution that you're working on. Even the mobile OSes have some form of native secrets manager that is now being opened up to the applications.

In this exercise, we're going to examine Go libraries for OSX password manager, a Windows password manager, and one that is cross-platform. We'll be looking at how to interact with them and how they store the passwords via the libraries.

The code for these examples can be found at https://github.com/cmceniry/login in the passwords directory. This code is using `dep` for dependency management, but this should work with Go modules as well. After downloading the code, change into the example's directory (`keychain`, `credmgr`, `keyring`) and run the example with `go run main.go`.

Caveats

The built-in password storage mechanisms on most OSes authenticate the application running as well as the user. With some storage types, you can grant permissions for the application to bypass the user authentication. The target of this grant depends on the OS—e.g., it can be the application binary, the user + binary, the name or file path to the binary, etc.

Unfortunately, this does not play as well with `go run` since that produces a different binary and identifier every time you run it.

Because of that, I recommend that you do not apply any "AllowAlways" rules for any of the runs, and that you only use test examples for these runs. At the end, you should clean up the examples that are created. In a production situation, once your binary has been built and distributed, then and only then, should you decide whether "AllowAlways" is worth the risk.

Some of the libraries attempt to simplify the overall interface to the native password stores. This simplification sometimes creates incomplete maps. In addition, the multiple libraries do not map the fields consistently, and the ones that support multiple native implementations may map each of those differently. This does make it a challenge to understand which is the correct invocation for each library and native store. Be sure to double check the documentation.

I've attempted to make it obvious in these examples, but you will see that that is not always an easy prospect.

Most of the libraries focus on the password or generic (`[]byte`) secret. Many of the native stores support additional typing for their secrets, but most of these are not supported by the libraries. Accordingly, we're going to focus on generic passwords in these exercises.

OSX Keychain

There are multiple implementations that interact with the OSX Keychain. We're going to explore the `go-keychain` library from `keybase`. This will have the import line (with alias to avoid naming issues):

```
keychain "github.com/keybase/go-keychain"
```

Keychain stores "Items" which are a password blob combined with metadata.

As mentioned, Keychain is capable of storing multiple Item types inside of it; however, the library support, and hence our focus, is limited to passwords, specifically "application password" (term inside of Keychain) or `GenericPassword` (term inside of the library).

`go-keychain` supports four pieces of metadata: the "Name" or "Label," the "Account," the "Service," and the "Access Group." The Name is what this Item shows up under in Keychain itself, and the library refers to this name as the Label. The Account is a string for the username associated with the password. The Service is a string for where (e.g., the URL) you want to use the password (the underlying Keychain field is literally called "Where"). You can have multiple Items with the same Name as long as the Account is different. Since the other libraries do not support a distinction between the "Name"/"Label" and the "Service" fields, we're going to set them to be the same thing.

The Access Group is a way of collecting multiple applications and multiple passwords and administering their access together. This is not used by other libraries and other OSes, so we will not use it here.

To create a simple password, we pass a `NewGenericPassword` to the `AddItem` library func:

keychain/main.go: create.

```
err := keychain.AddItem(
   keychain.NewGenericPassword(
      ";login example,"
      "falken,"
      ";login example,"
      []byte("joshua"),
      ","
   ),
)
```

Since we have to ensure that the Name and Account are unique, it is possible to encounter a duplicate. For this simple example, we're going check our errors for that and ignore just that error while responding to any other errors.

keychain/main.go: duperr.

```
if err != nil && err != keychain.ErrorDuplicateItem {
```

If all goes well, we've stored it into the password store, and now we need to retrieve it. There are two ways to retrieve the secret: the get helper, and a full query.

For just getting a password with known location, there's a convenient `GetGenericPassword` func that will grab it for us, and we can print it out. To use it, we have to identify the metadata for it: Name/Label, Account, Service, and Access Group. Since this function is general purpose, we have to fill in all four fields, even with empty strings, to match the signature.

keychain/main.go: getgeneric.

```
item, err := keychain.GetGenericPassword(
   ";login example,"
   "falken,"
   ";login example,"
   ","
)
...
fmt.Println(string(item))
```

The second method is to `query` for it. The `query` is useful when looking for multiple Items. We set the parameters that we need to match on, indicate that we're looking for one or multiple answers, ask for it to return the data rather than just the metadata, and then perform our actual query. Since this can return multiple responses (though in this case only one will return), we still want to iterate over the results.

keychain/main.go: query.

```
query := keychain.NewItem()
query.SetSecClass(keychain.SecClassGenericPassword)
query.SetLabel(";login example")
query.SetAccount("falken")
query.SetService(";login example")
query.SetMatchLimit(keychain.MatchLimitOne)
query.SetReturnData(true)
results, err := keychain.QueryItem(query)
...
for _, i := range results {
   fmt.Println(string(i.Data))
}
```

This works well if we're looking for a specific password by account. Try removing the `SetService` or `SetLabel` calls for the query to see what is returned.

Credential Manager

Next, we're going to interface with Window's Credential Manager using the `wincred` library from GitHub user `danieljoos`.

It has the import path:

```
"github.com/danieljoos/wincred"
```

Like Keychain, Credential Manager maintains some metadata for its secrets. It requires a "TargetName," which has to be unique, and allows for an optional Username field. For this example, we're going to limit it to just the TargetName and secret itself.

The creation of new passwords is relatively straightforward. We create a new record by its name with `NewGenericCredential`, assign the password itself and the optional metadata, and then write that to the store.

credmgr/main.go: create.

```
cred := wincred.NewGenericCredential("loginExample")
cred.UserName = "falken"
cred.CredentialBlob = []byte("joshua")
err := cred.Write()
```

In the event of duplicates, Credential Manager will overwrite what is there.

Now we can retrieve that password out of the Credential Manager. Since we're fetching by TargetName, it's a simple `get` command followed by print.

credmgr/main.go: get.

```
cred, err := wincred.GetGenericCredential("loginExample")
...
fmt.Println(cred.UserName)
fmt.Println(strings(cred.CredentialBlob))
```

The `wincred` library and underlying interface to Window's Credential Manager is quite a bit more intuitive than the Keychain interface, but it also does not allow for more complex cases that use duplicate metadata values for records (e.g., retaining multiple versions).

Cross-Platform

Now, let's combine those and use a common library to try to make it cross-platform. To be specific, our definition of cross-platform use is to be able to use the same binary/code across multiple OSes; it is not about moving the password data across multiple OSes. Directly porting over the password data is complicated and needs some transformation since Keychain and Credential Manager and other native stores have different semantics.

For cross-platform usage, we're going to look at the `keyring` library by `99designs`. It supports storing secrets in multiple backends: Keychain, Credential Manager, the Gnome secrets service, KDE Wallet, and others. It has the import path:

```
"github.com/99designs/keyring"
```

Many of the overall options for `go-keychain` and `wincred` are stripped out from `keyring`. With it, you can specify a container, the "ServiceName," and a "Key" for a specific entry. In the underlying password store, these two fields may be mirrored onto other fields (e.g., when backing with Keychain, it sets Label and Service to both be the ServiceName), but `keyring` only allows for these two fields.

For creation and retrieving, we start by opening (read declaring) our password container by its ServiceName:

keyring/main.go: open.

```
kr, err := keyring.Open(keyring.Config{
   ServiceName: ";login example,"
})
```

After that, we are able to commit it to the password store with its value. We must also identify the unique Key inside of this ServiceName for this password. Note that unlike some of the native libraries where this is a "construct then write" method, the `keyring` library does this as one command (again, different semantics).

keyring/main.go:create.

```
_ = kr.Set(keyring.Item{
   Key:  "falken,"
   Data: []byte("joshua"),
})
```

Much like `wincred`, `keyring` overwrites existing values instead of indicating a duplicate Item.

Now that we have the data stored, we can pull it out and display it. As with the single function to write the Item, the `keyring` library uses a single function to retrieve data.

keyring/main.go: get.

```
p, err := kr.Get("falken")
if err != nil {
          panic(err)
}
fmt.Println(string(p.Data))
```

Library Compatibility

Making it so that we can use `keyring` and either `go-keychain` or `wincred` together requires us to match up the appropriate metadata.

For `go-keychain`, this is a matter of matching the `keyring` ServiceName with both the `go-keychain` Label and Service and the `keyring` Key with the `go-keychain` Account.

For `wincred`, we must match the `keyring` ServiceName with the `wincred` TargetName which currently has to include some additional markup. As of this writing, the `keyring` library will append `aws-vault:` to the ServiceName and Key to store as the TargetName in `wincred`. (The `keyring` library was originally named `aws-vault` and so retains a few vestiges of that. There is an open issue for this.)

Conclusion

Password management is hard.

The current level of cross-platform compatibility is low. There is no standard for password storage—particularly the identity and metadata for finding the passwords. There are significant inconsistencies in the native password stores as well as inconsistencies in the way that the libraries map to those password stores. Applications can be ported across multiple OSes, but the passwords saved are hit or miss if you try to use them across multiple applications. It's best right now to pick one library and stick with it.

When considering the user, it's a constant tradeoff between security and ease of use. You have to decide for yourself what is your level of risk. If storing passwords in the native password stores encourages other secure activities (e.g., shorter sessions, not using static API keys, etc.), then this may be right up your alley. If it's a matter of storing the passwords in the native passwords stores versus your own securely wrapped (encrypted, permissions, etc.) files, it's better to use the former as a significant amount of engineering effort has been put in place that makes the native password stores probably more secure than what most of us would do otherwise.

Despite the issues, I hope that this column has given you some insight into how to handle these in reasonable ways and the confidence to do so. Anything that we can do to improve the state of password management by making it easier on the user and more secure is a boon for our field. Native password stores are just one of those methods, and I encourage you to use them. Good luck and be safe!