Custom Binaries to Ease Onboarding Using Go

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

It recently came up that I needed to release a helper tool for our work environment. I was limited with regards to my distribution methods since much of the user base is BYOD based. Having Go in my toolbox, I knew that I could use Go to ease the distribution. One of Go's selling points is its ability to package up all of its dependencies and runtime at compile time, so that you can avoid the runtime dependencies management issues that often arise.

In this case, the executable I built required a lot of configuration information—our list of compute clusters, the authentication endpoint, and an authentication client ID (OAuth 2 based)—so I wrote up and released the documentation on how a user can configure the tool for our environment.

After several weeks with supporting users and the binary, I observed two key behaviors:

1. Every user eventually used the same configuration file, and
2. I needed to regularly update the configuration file as we built, deleted, or moved clusters.

Every time there was a change in the latter, I had to inform the users, publish a new set of documentation, and ask everyone to update. This had mixed success. The extra amount of work, the amount of internal works that were exposed to every user, and the limit of effectiveness of the updates made me look for an easier way to accomplish this.

I started to compare it to another rising situation: mobile device application management. Mobile devices operate under similar circumstances. They tend to be dominated by BYOD. Applications are distributed as large single installs that similarly embed the runtime. The one big difference that I noticed is that with mobile devices, the users are limited with some configuration items. Most configuration items are either compiled into the binary or fetched and cached on the device. Some of those configuration items include secrets such as application identifiers and client tokens.

In an attempt to make life easier, I decided to try moving the configuration around with my helper executable.

In this column, we're going to explore moving the configuration for organizational applications out of configuration files. Along the way, we're going to use this as an opportunity to pick up the AWS object storage, S3, to help us out. We're going to store our basic configuration in an S3 bucket, and we're going to provide access to that bucket by hard coding the access values into the executable.

The code for this example can be found at https://github.com/cmceniry/login/ in the "hardcode" directory. hardcode contains a customizer directory, which is our example application without any organizational-specific configuration.

For this example, you'll need:

1. An AWS account.
2. To create a bucket and upload a sample. If you are new to S3, use this guide: https://docs.aws.amazon.com/AmazonS3/latest/gsg/GetStartedWithS3.html.
3. Set up an IAM user with access keys. This user should have the AmazonS3ReadOnlyAccess policy applied to it (or a more restrictive one if you are familiar with IAM). If you are new to AWS access management, see https://docs.aws.amazon.com/IAM/latest/UserGuide/id_users_create.html#id_users_create_console.

You can also choose to use a different, off-site storage technique, but a point of this article is to learn how to use the AWS Go interface for S3.

Storing Remote Configuration

The first part is to move the bulk of configuration into an external store.

Amazon's Simple Storage Service (S3) is a household name for many working in cloud environments. It provides an authenticated and globally available storage location for static contents. We're fetching our configuration from an S3 bucket. In this example, that configuration is going to be a simple string message, but it could easily be a block of structured data to hold various values.

We're dependent on the AWS Go SDK. You can obtain this with go get github.com/aws-sdk-go/... or using a Go dependency management tool.

Since we're only accessing S3 in one place, we're going to wrap all of that in a single function. It expects four inputs: the access key, its secret key, and the bucket name and path. Instead of returning a value, it will set a global configuration—we'll return to this in the next section.

fetch.go: fetch.

```
func fetch(ak, sk, bucket, path string) error {
```

Inside of `fetch`, we start by configuring an AWS session. This is used for all of the interactions with the AWS APIs. We're going to give it our authentication keys and provide a region for the profile to use. Outside of the static credentials, the AWS SDK does not operate directly on Go types, so we'll need to wrap these with the AWS SDK types.

fetch.go: session.

```
sess, err := session.NewSession(&aws.Config{
    Region: aws.String("us-west-2"),
    Credentials: credentials.NewStaticCredentials(
        ak, sk, "",
    ),
})
```

Before we download, we need a place to download to. Since the application will be using this configuration, we want to keep this configuration in a memory buffer. Underneath the hood, S3 may download across multiple streams and data segments. This means that an ordinary buffer, specifically one that expects just to append to the end, will not work. AWS provides a buffer—aws.WriteAtBuffer—that can be written to in multiple locations at the same time so we can use that.

fetch.go: writeatbuf.

```
writeAtBuf := aws.NewWriteAtBuffer([]byte{})
```

Next we construct the downloader and run the download. The `s3manager.Downloader` is an intelligent transfer manager and capable of downloading many different objects in parallel. In this case, we're just downloading the one object, but we still funnel everything through it. When creating it, we need to tell it our AWS API session so that it has the proper authentication information. `Download` requires a destination—our `writeAtBuf` buffer—and a source—the `aws.String` wrapped bucket name and path or key name.

One thing to note: the `writeAtBuf` parameter passed into download is an `io.WriteAt` interface. This means anything that has a `WriteAt` member method can be used there. For instance, if you were downloading straight to a file, then `os.File` can be used directly since it has the `WriteAt` member method. This is an excellent example of using Go interfaces for flexibility.

fetch.go: download.

```
downloader := s3manager.NewDownloader(sess)
_, err = downloader.Download(
    writeAtBuf,
    &s3.GetObjectInput{
        Bucket: aws.String(bucket),
        Key:    aws.String(path),
    },
)
```

Once we're complete on the download, we then convert that into a string we can use in our configuration. For presentation purposes, we strip leading and trailing whitespace from our value.

fetch.go: config.

```
globalConfig = strings.TrimSpace(
    string(
        writeAtBuf.Bytes(),
    ),
)
```

Again, we stored the configuration in a `customizer`-level variable instead of returning it from the function. As we'll see next, that will help us with our custom application configuration.

Packaging the Configuration Access

Also inside of the `customizer` directory is a `main.go` containing a Main method. This is not a standard Go `main`—it is not in the main package, and it is exported. It is, however, meant to be the entry point for execution of our application. It lacks specific organizational customizations and only has variables to allow for this.

To simplify naming, it takes a `customizer.Options` type. In this example, we just mirror the four items we need to access our S3 bucket. In other situations, this could also include authentication endpoint URLs, specific DNS names, or any other generally unchanging values.

main.go: options.

```
type Options struct {
   AccessKeyID     string
   SecretAccessKey string
   BucketName      string
   BucketPath      string
}
```

This is instantiated as a `customizer`-level variable so that any function inside of `customizer` has access to it. For this same reason, we put our `globalConfig` value from the S3 bucket at the same level. This mirrors how many Go command line tools operate—especially ones that use the Standard Library `flag` or Steve Francia's `pflag` libraries.

main.go: vars.

```
var opt Options
var globalConfig string
```

Once the inputs and variables are established, we can define our pseudo-`Main`. It should be passed an `Options` parameter, which is what will be provided to make an organization-specific application build. The `customizer`-level `opt` parameter is set to the provided `Options` parameter for these values to take effect.

main.go: mainopt.

```
func Main(o Options) {
   opt = o
```

Beyond that, it behaves akin to any `main`, including parts such as command line argument parsing. Since we're pulling additional configuration from S3, we also want to ensure that we perform that as part of this `Main`.

main.go: mainfetch.

```
   err := fetch(
      opt.AccessKeyID,
      opt.SecretAccessKey,
      opt.BucketName,
      opt.BucketPath,
   )
```

Since this is an example, it does not do anything other than print the value of the retrieved configuration file from S3.

main.go: mainprint.

```
   fmt.Printf("Using Configuration: %s\n", globalConfig)
```

Creating a Custom Binary

The `customizer` library can't execute on its own. We need to call it from our own `main.main` method where we pass the specific organizational `Options` values to it.

```
package main

import "github.com/cmceniry/login/hardcode/customizer"

func main() {
   customizer.Run(
      customizer.Options{
         AccessKeyID: "appspecific1",
         SecretKeyID: "orgspecific2",
         BucketName:  "orgspecific3",
         BucketPath:  "orgspecific4",
      },
   )
}
```

Any number of these organization-specific builds can be done, and all end up being approximately the same number of lines of code (depends on the number of options). The marginal effort to create organization-specific builds is limited to ensuring that the configuration items are specified.

Considerations

While this approach certainly aids in the ease-of-use department, there are several considerations and tradeoffs to at least look at. Many exist, but here are some of the more pressing ones.

All of the configuration items in the binary or remote storage should be limited to low-risk items. Low risk is relative, but the rule of thumb is that there is not anything more disclosed than could be available to anyone inside of the organization. Typically, this limits it to coarse-level information disclosures. Conversely, if this opens to arbitrary code execution—e.g., download a binary and run it—you should ensure that the code is signed or validated. It's arguable that no secret should even be hard coded into a binary, especially one that is expected to be widely distributed in an organization. The worry is that this is a slippery slope and encourages bad practices. The balance of security and usability is a constant navigation of slippery slopes.

Any time you use hard-coded values in user applications, you need to account for the fact that you'll have multiple applications in the wild at a time. This means that you'll need to ensure that the use of these configuration items could exist at the same time.

For instance, in this example, AWS supports two API keys per user. This allows you to rotate the key, and both the old and new values are valid while you rotate it.

This is not limited to the server side. If there are validation keys, your application will need to support an array of keys so that the old and new can exist at the same time.

```
customizer.Options{
   VerifyKeys: []string{"abcd", "efgh"},
}
```

For remote configuration, your application will need to support the configuration format of the future. In practice, this means that your application will probably rely on non-strict validation of the configuration data and reasonable defaults when the configuration is unspecified on the remote storage.

You have to decide what goes in the application and what is stored in remote configuration storage. This will largely come down to a question of flexibility. If you expect something to remain largely static or static over a longer period of time, you can put it into the binary. If you expect it to change on a regular basis—at least more often than you want to release binary updates—put it into the configuration repository.

Conclusion

I long held the belief that you should not hard code anything into your binary. If you did, it was a sign that your infrastructure lacked good distribution mechanisms. Best practice was to build and distribute them separately using strong central configuration tools.

Those assumptions came from a specific perspective. That perspective was common, but, with the rise of decentralizing practices such as BYOD and remote work, it has become less so.

Sometimes you have to question your assumptions about best practices. When best practices are established, they are done so in a certain environment. If the environment of the nature of the problem has changed, then the practices need to adjust with them. We're seeing more and more environments where controlling the end device is a very different prospect than it used to be.

Don't be afraid to question the assumptions that you've held. Sometimes you'll find that you're not held to the same constraints that you used to be or that you're not enabled by the same capabilities that you used to be. When this happens, you have to adjust and come up with new best practices.