Introduction à Go – TD6: paman

Franklin "Snaipe" Mathieu

2023

Durant ce TD, nous allons implémenter, progressivement, un CLI de gestion de mot de passe. Le coffre-fort des mots de passe serait chiffré, et stocké dans différents backends, en fonction de la configuration.

L'interface proposée de cet outil serait:

- `paman get <id>` – récupère le mot de passe ayant l'identifiant `<id>`, et l'affiche sur la sortie standard.
- `paman set <id>` – lis un mot de passe sur l'entrée standard, et l'assigne à (ou crée) l'identifiant donné.
- `paman generate <id> [length] [--chars <set>]` – génère un mot de passe aléatoire de longueur `length` et l'assigne à (ou crée) l'identifiant donné. `--chars set`, si spécifié, représente les caractères autorisés pour générer le mot de passe; par exemple a-zA-Z0-9+*- autorise tous les caractères alphanumériques, plus (+), étoile (*), et tiret (-). Le tiret n'est spécifiable qu'en début ou fin de set, sinon il est utilisé en tant que séparateur d'intervalle.

1 Les bases

1.1 Les fichiers de base

Comme d'habitude, créez un nouveau dossier pour votre projet, et créez le module associé:

```
$ go mod init example.com/paman
$ go get github.com/alecthomas/kong
```

Crééz un dossier `cmd/paman`, et ajoutez un fichier `cmd/paman/main.go`:

```go
package main

import (
	"fmt"
	"os"

	"github.com/alecthomas/kong"
)

func main() {
	var cli struct {
		Get      GetCmd      `cmd help:"retrieve a password"`
		Set      SetCmd      `cmd help:"assign a password"`
		Generate GenerateCmd `cmd help:"generate a password"`
	}
	ctx := kong.Parse(&cli)

	if err := ctx.Run(); err != nil {
		fmt.Fprintf(os.Stderr, "%s: %s\n", os.Args[0], err)
		os.Exit(1)
	}
}
```

Un fichier `cmd/paman/get.go`:

```
package main

type GetCmd struct {
        ID string `arg`
}

func (cmd *GetCmd) Run() error {
        return nil
}
```

Un fichier `cmd/paman/set.go`:

```
package main

type SetCmd struct {
        ID string `arg`
}

func (cmd *SetCmd) Run() error {
        return nil
}
```

Et un fichier `cmd/paman/generate.go`:

```
package main

type GenerateCmd struct {
        Chars string `default:"a-zA-Z0-9!-/:-@[-"`

        ID     string `arg`
        Length int    `arg`
}

func (cmd *GenerateCmd) Run() error {
        return nil
}
```

1.2 Coffre-fort

Notre gestionnaire de mot de passe a besoin d'une implémentation de coffre fort pour pouvoir stocker nos mots de passe. Faisons un nouveau paquet nommé `vault` pour représenter cette implémentation.

Le paquet `vault` contiendra un type `Vault` contenant toutes les informations que nous souhaitons stocker dans notre coffre fort, ainsi qu'une interface `Store` représentant les opérations Load et Store pour respectivement charger et sauvegarder un `Vault`.

L'idée de l'interface Store est qu'il devient possible d'implémenter différents types de backends dans lequel serialiser notre coffre fort.

`Vault` implémenterait des méthodes pour en sérialiser et chiffrer le contenu via AES, et invesement déchiffrer et déserialiser.

L'utilisation basique proposée du paquet vault permettrait l'écriture de fonctions telles que:

```
func RotatePassword(store vault.Store, id string) (old, new string, err error) {
        var data vault.Vault
        if err := store.Load(&data); err != nil {
                return "", "", err
        }

        old = data.Passwords[id]
        new = GeneratePassword()
        data.Passwords[id] = new

        if err := store.Store(&data); err != nil {
                return "", "", err
        }
        return old, new, nil
}
```

Commençons par créer `vault/vault.go`:

```
package vault

import (
        "crypto/cipher"
        "io"
)

type Entry struct {
        Password string

        // Potentiellement d'autres champs plus tard
}

type Vault struct {
        Entries map[string]Entry
}

func (v *Vault) Marshal(out io.Writer, block cipher.Block) error {
        // FIXME:
        // 1. utilisez encoding/json pour transformer Vault en texte à chiffrer
        // 2. utilisez block pour chiffrer le texte
        // 3. écrivez le texte dans out
        //
        // NOTE: l'utilisation de cipher.Block n'est pas triviale; ne tentez pas
        // de le deviner par vous même. Référez vous à l'exemple dans la bibliothèque
        // standard: https://pkg.go.dev/crypto/cipher#NewCFBEncrypter
}

func (v *Vault) Unmarshal(in io.Reader, block cipher.Block) error {
        // FIXME:
        // 1. lisez le document chiffré depuis le Reader
        // 2. utilisez block pour déchiffrer le document
        // 3. utilisez encoding/json pour décoder le texte en Vault
        //
        // NOTE: l'utilisation de cipher.Block n'est pas triviale; ne tentez pas
        // de le deviner par vous même. Référez vous à l'exemple dans la bibliothèque
        // standard: https://pkg.go.dev/crypto/cipher#NewCFBDecrypter
}
```

Ajoutons immédiatement un test dans `vault/vault_test.go`:

```go
package vault

import (
	"bytes"
	"crypto/aes"
	"encoding/hex"
	"testing"
	"maps"
)

func TestVaultMarshaling(t *testing.T) {
	cases := []struct{
		Name  string
		Vault Vault
	}{
		{
			Name:  "Empty",
			Vault: Vault{},
		},
		{
			Name:  "Basic",
			Vault: Vault{
				Entries: map[string]Entry{
					"google": Entry{
						Password: "my google password",
					},
					"netflix": Entry{
						Password: "shared among friends",
					},
				},
			},
		},
	}

	testKey, _ := hex.DecodeString("6368616e676520746869732070617373")

	block, err := aes.NewCipher(testKey)
	if err != nil {
		t.Fatal(err)
	}

	for _, tcase := range cases {
		t.Run(tcase.Name, func(t *testing.T) {
			// On serialise le Vault du test
			var encrypted bytes.Buffer
			if err := tcase.Vault.Marshal(&encrypted, block); err != nil {
				t.Fatal(err)
			}

			// On le déserialise dans un nouveau Vault
			var actual Vault
			if err := actual.Unmarshal(&encrypted, block); err != nil {
				t.Fatal(err)
			}

			// Le résultat devrait être egal au Vault initial
			if !maps.Equal(tcase.Vault.Entries, actual.Entries) {
				t.Fatalf("vaults are not equal: expected %v, got %v",
					tcase.Vault, actual)
			}
		})
	}
}
```

Le teste valide qu'un Vault reste inchangé si on le serialise puis déserialise simplement.

1.3 File store

Nous avons un Vault, mais in nous manque un backend de stockage pour le serialiser. Pour l'instant, et pour nos tests locaux, implémentons un vault.Store qui utilise un fichier local.

Déjà, nous devons définir notre interface Store dont nous avons parlé précédemment. Dans un fichier `vault/store.go`:

```
package vault

type Store interface {
        Load(*Vault) error
        Store(*Vault) error
}
```

Ajoutons plus bas une implémentation nommée FileStore:

```
type FileStore struct {
        Block cipher.Block
        Path  string
}

func (store *FileStore) Load(vault *Vault) error {
        // FIXME
}

func (store *FileStore) Store(vault *Vault) error {
        // FIXME
}
```

FileStore devrait être relativement simple à implémenter: il suffit principalement d'utiliser os.Open/os.Create ainsi que les méthodes de Vault précédemment implémentées. L'utilisateur de l'API sera responsable de renseigner le chemin di fichier dans Path et l'algorithme de chiffrage dans Block.

1.4 De nouveau dans le CLI

Nous avons une implémentation fonctionnelle d'un coffre-fort, ainsi que d'un backend de stockage se basant sur un fichier local. Nous pouvons commencer à implémenter la logique du CLI, mais nous avons quelques prérequis à régler:

- Nous devons initialiser un Block AES, mais celui-ci demande une clé, comme vous avez probablement pû le voir dans le test de Vault.
- Nous devons communiquer un vault.Store dans nos sous-commandes de CLI.

Attaquons chaque problème en son temps. Déjà, nous devons passer un Store à nos commandes. Kong nous permet de faire ça via les fonctions Bind et BindTo; changeons le main:

```
 func main() {
         var cli struct {
                 Get      GetCmd      `cmd help:"retrieve a password"`
                 Set      SetCmd      `cmd help:"assign a password"`
                 Generate GenerateCmd `cmd help:"generate a password"`
         }
         ctx := kong.Parse(&cli)

+        store := &vault.FileStore{
+                Block: block, // sera défini plus tard
+                Path:  "vault.dat",
+        }
+        ctx.BindTo(store, (*vault.Store)(nil))

         if err := ctx.Run(); err != nil {
                 fmt.Fprintf(os.Stderr, "%s: %s\n", os.Args[0], err)
                 os.Exit(1)
         }
 }
```

Notre FileStore a maintenant été bind en argument à un vault.Store; ce que ça veut dire, c'est que vous pouvez modifier vos sous-commandes tel que:

```
-func (cmd *GetCmd) Run() error {
+func (cmd *GetCmd) Run(store vault.Store) error {
         return nil
 }
```

Ce faisant, vous avez maintenant accès, depuis vos sous-commandes, à un vault.Store.

Repassons dans le main; nous devons maintenant définir `block`:

```
 func main() {
         var cli struct {
                 Get      GetCmd      `cmd help:"retrieve a password"`
                 Set      SetCmd      `cmd help:"assign a password"`
                 Generate GenerateCmd `cmd help:"generate a password"`
         }
         ctx := kong.Parse(&cli)

+        block, err := aes.NewCipher(key) // key défini plus tard
+        if err != nil {
+                fmt.Fprintf(os.Stderr, "%s: %s\n", os.Args[0], err)
+                os.Exit(1)
+        }

         store := &vault.FileStore{
                 Block: block,
                 Path:  "vault.dat",
         }
         ctx.BindTo(store, (*vault.Store)(nil))

         if err := ctx.Run(); err != nil {
                 fmt.Fprintf(os.Stderr, "%s: %s\n", os.Args[0], err)
                 os.Exit(1)
         }
 }
```

Finalement, nous avons besoin d'une clé. Le paquet `crypto/aes` nous dit que la clé doit être de 16, 24, ou 32 octets fixes.

Ce n'est donc pas un mot de passe; il nous faudrait un moyen de convertir un mot de passe en clé valide AES. Heureusement, il existe des fonctions cryptographiques pour faire ceci (appellées Key Derivation Functions).

Pour ce projet, nous allons utiliser PBKDF2. Nous allons demander un mot de passe à l'utilisateur pour chiffrer et déchiffrer le Vault, le convertir en une clé AES de 32 octets avec PBKDF2, puis l'utiliser dans aes.NewCipher.

Pour faire ceci, vous aurez besoin de deux bibliothèques supplémentaires: `golang.org/x/term` pour demander un mot de passe en ligne de commande, et `golang.org/x/crypto` pour PBKDF2.

```
 func main() {
         var cli struct {
                 Get      GetCmd      `cmd help:"retrieve a password"`
                 Set      SetCmd      `cmd help:"assign a password"`
                 Generate GenerateCmd `cmd help:"generate a password"`
         }
         ctx := kong.Parse(&cli)

+        fmt.Fprint(os.Stderr, "please enter vault password: ")
+        password, err := term.ReadPassword(0)
+        if err != nil {
+                fmt.Fprintf(os.Stderr, "%s: %s\n", os.Args[0], err)
+                os.Exit(1)
+        }
+
+        // NOTE: dans la vraie vie, salt devrait être généré aléatoirement pour
+        // chaque utilisateur, et stocké quelque part.
+        salt := []byte(os.Getenv("USER"))
+
+        // NOTE: le nombre de rounds de PBKDF2 choisi dépends du délai acceptable
+        // de l'opération pour votre application
+        const rounds = 1000
+
+        key := pbkdf2.Key(password, salt, rounds, 32, sha256.New)

         block, err := aes.NewCipher(key)
         if err != nil {
                 fmt.Fprintf(os.Stderr, "%s: %s\n", os.Args[0], err)
                 os.Exit(1)
         }

         store := &vault.FileStore{
                 Block: block,
                 Path:  "vault.dat",
         }
         ctx.BindTo(store, (*vault.Store)(nil))

         if err := ctx.Run(); err != nil {
                 fmt.Fprintf(os.Stderr, "%s: %s\n", os.Args[0], err)
                 os.Exit(1)
         }
 }
```

Vous avez maintenant toutes les pièces pour implémenter une première version des sous-commandes GetCmd, SetCmd et GenerateCmd.

Bonus: demander un mot de passe à chaque fois que le programme est invoqué est fatiguant pour l'utilisateur. Utilisez `github.com/zalando/go-keyring` pour stocker et récupérer le mot de passe du coffre pour l'utilisateur actuel.

2 Store distant

Le stockage du vault dans un fichier local est utile pour nos tests, mais peu pratique dans la vraie vie. Il serait utile de pouvoir stocker le coffre dans un système distant.

Nous allons implémenter un vault.Store qui se connecte à une instance MongoDB.

2.1 Lancer mongodb

Pour cette exercice, nous allons lancer une instance locale de mongodb via docker:

```
$ # Pensez à installer et démarrer docker
$ sudo apt install -y docker.io
$ sudo systemctl start docker.socket

$ # Si vous êtes sur la machine de TP, executez ceci
$ sudo chmod 777 /var/run/docker.sock

$ # démarre mongodb et expose son port 27017
$ docker run --name mongo \
>   -e MONGO_INITDB_ROOT_USERNAME=root \
>   -e MONGO_INITDB_ROOT_PASSWORD=1234 \
>   -p 27017:27017 \
>   -d mongo

$ # Pour stopper le conteneur mongo
$ # docker stop mongo && docker rm mongo
```

Nous allons aussi ajouter la bibliothèque Go mongodb:

```
$ # Driver mongo officiel
$ go get go.mongodb.org/mongo-driver/mongo
```

Pour vous connecter à mongodb depuis Go:

```
import (
        "go.mongodb.org/mongo-driver/bson"
        "go.mongodb.org/mongo-driver/mongo"
        "go.mongodb.org/mongo-driver/mongo/options"
)
...

ctx := context.Background()
uri := "mongodb://root:1234@localhost:27017/"

client, err := mongo.Connect(ctx, options.Client().ApplyURI(uri))
if err != nil {
        return err
}

defer func() {
        // Attendre 5 minutes maximum que le client se déconnecte.
        ctx, stop := context.WithTimeout(context.Background(), 5 * time.Minute)
        defer stop()

        if err := client.Disconnect(ctx); err != nil {
                fmt.Fprintln(os.Stderr, err)
        }
}()

// Utiliser le client mongo
```

Vous allez avoir besoin de deux choses: savoir comment trouver un document, et savoir comment en écrire un.

2.2 Trouver un document

Nous allons modéliser un schéma mongodb d'exemple; nous avons une base de donnée `addressbook`, contenant une collection `profiles` stockant des profils utilisateurs.

Un profil est ici représenté par:

```
type Profile struct {
        // import "go.mongodb.org/mongo-driver/bson/primitive"
        ID    primitive.ObjectID `bson:"_id"`
        User  string
        Email string
}
```

C'est à dire qu'un utilisateur est identifié de manière unique par un ID, et possède un username et une adresse email.

Pour trouver le premier profil d'un utilisateur dont l'username est "robpike":

```
ctx := context.Background()
coll := client.Database("addressbook").Collection("profiles")

// import "go.mongodb.org/mongo-driver/bson"
filter := bson.D{{"user", "robpike"}}

var profile Profile
if err := coll.FindOne(ctx, filter).Decode(&result); err != nil {
        return err
}
```

2.3 Insérer un document

En reprenant le même schéma plus haut, il est possible d'insérer un document dans la base de donnée:

```
coll := client.Database("addressbook").Collection("profiles")
profile := Profile{Name: "robpike", Email: "r@google.com"}

result, err := coll.InsertOne(ctx, profile)
```

Notez que ceci insère un nouveau document; pour mettre à jour un document existant, vous devrez utiliser `coll.UpdateOne`.

2.4 Implémentons un vault.Storer

Nous avons maintenant tous les éléments pour implémenter notre vault.Storer.

Faisons un paquet `mongostore`, et un fichier `mongostore/storer.go`:

```go
package mongostore

import (
        "example.com/paman/vault"
)

type Store struct {
        // FIXME
}

func NewStorer(/* FIXME ... */) *Storer {
        // FIXME
        return nil
}

func (s *Store) Load(v *vault.Vault) error {
        // FIXME
        return nil
}

func (s *Store) Store(v *vault.Vault) error {
        // FIXME
        return nil
}

// NOTE: ceci permet de s'assurer que notre type *Store implémente
// l'interface vault.Store
var _ vault.Store = (*Store)(nil)
```

C'est maintenant à vous d'implémenter le store. Vous devrez aussi changer votre main pour l'utiliser au lieu d'un FileStore.

3 Configuration

Avoir changé le main pour passer d'un vault.FileStore à un mongostore.Store ne semble pas idéal. Il serait judicieux de proposer à l'utilisateur un moyen de configurer quel store il veut utiliser.

Nous allons utiliser TOML pour représenter notre configuration. Le format proposé serait:

```toml
default_store = "mongo"

[stores.local]
type      = "file"
file.path = "/home/tp/.vault.dat"

[stores.mongo]
type       = "mongo"
mongo.uri  = "mongo://root:1234@localhost:27017/"
```

L'idée serait que le fichier de configuration déclarerait un certain nombre de stores utilisables par paman, et `default_store` définirait quel store utiliser par défaut. On pourrait même ajouter un flag `--store=local` à la commande pour selectionner un autre store.

Il existe une bibliothèque populaire pour parser du TOML sur `github.com/BurntSushi/toml`.

Son utilisation est similaire à `encoding/json`; vous définissez une struct avec vos champs de configuration, et vous utilisez `toml.Decode` pour décoder votre configuration depuis un fichier.

La structure correspondant à la configuration plus haut serait:

```
type Config struct {
        DefaultStore string `toml:"default_store"`

        Stores map[string]struct{
                Type string

                File struct {
                        Path string
                }

                Mongo struct {
                        URI string
                }
        }
}
```

Changez votre main pour décoder la configuration depuis `$HOME/.config/paman.toml`, puis selectionner et instancier un vault.Store correspondant au DefaultStore configuré, ou au store spécifié par `--store`.