TP1: TokenSnare

Seguridad de la información

2do cuatrimestre 2025

Ciberseguros

Nombre	LU	Mail
Juan Begalli	139/22	juanbegalli@gmail.com
Francisco Cueto	223/22	francue3@gmail.com
Santiago Rivas	415/22	santiagorivas0203@gmail.com
Mateo Schiro	657/22	mateo.schiro8@gmail.com

Contenidos

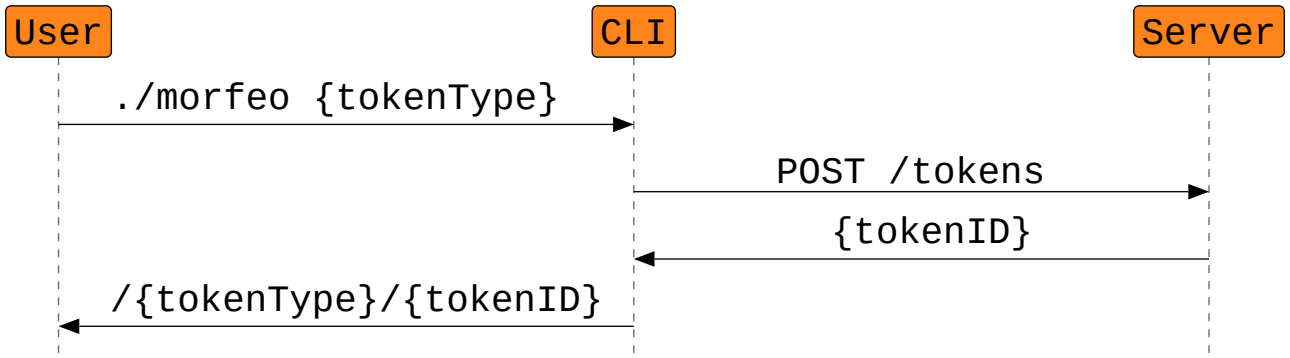
1) Introducción teórica

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do.

2) Funcionamiento

La herramienta **morfeo** cuenta con dos partes: la **CLI** (*command line interface*) y el **server**. La **cli** es la encargada de procesar los comandos del usuario, comunicarse con el server y crear los tokens. El **server** es el encargado de la identificación de los tokens, de detectar las activaciones de los mismos y dar las alertas.

Se incluye a continuación un diagrama que representa el flujo de funcionamiento de la herramienta.



Explicación:

1. El usuario ejecuta `./morfeo {tokenType}`, donde `{tokenType}` es uno de los formatos disponibles (`qr`, `bin`, `css`, etc), con las flags correspondientes del formato escogido.
2. La CLI se comunica con el server, mandando un **POST** a `tokens`, indicando la creación de un nuevo token.
3. El server se comunica con la base de datos, almacena la información mandada por la CLI, y devuelve el `tokenID` generado por la misma.
4. La CLI toma el `tokenID` recibido y construye la URL correspondiente, juntándolo con el tipo de token solicitado.

Notar que en el paso 2, al comunicarse con el server, no se indica qué tipo de token se está creando. No existe distinción desde el lado del server en los distintos tipos de tokens. Esto facilita el manejo de las activaciones, pero limita a que todos los tokens generados tengan el mismo método de *call-home*: una solicitud **GET** al servidor.

Cuando un token es activado, la solicitud **GET** es mandada a una URL con la forma `/{tokenType}/{tokenID}`. La distinción del tipo de token en la URL permite que el handler ejecute los pasos adicionales (además de la alerta) de los tokens que así lo requieran (por ejemplo, la redirección en el código QR).

3) Implementación

3.1) CLI

La **CLI** se encuentra implementada en Golang, y fue utilizada una librería de creación de **CLIs** llamada Cobra. La misma permite definir y agregar fácilmente nuevos comandos, además de realizar el *parsing* de las flags recibidas.

Todos los tokens toman como entrada necesaria dos flags: **msg** y **chat**. La flag **msg** define qué mensaje será mandado cuando se realice la alerta de activación del token, y la flag **chat** es el ID del chat de Telegram donde será mandada la alerta. Cada comando puede tomar también otras flags más específicas de su funcionamiento.

Independientemente del tipo de token a crear, cada comando recibe los flags correspondientes y utiliza la siguiente función de creación (se omitió el manejo de errores para ahorrar espacio):

```
func CreateToken(msg string, extra string, chat string) string {
  data := types.UserInput{
    Msg:   msg,
    Extra: extra,
    Chat:  chat,
  }

  body, err := json.Marshal(data)
  resp, err := http.Post(serverURL+"/tokens",
                         "application/json",
                         bytes.NewBuffer(body))
  defer resp.Body.Close()

  respBytes, err := io.ReadAll(resp.Body)
  tokenID := string(respBytes)
  return tokenID
}
```

Esta función toma tres parámetros y crea un struct de tipo `UserInput`. Este struct contiene toda la información que será almacenada del token a crear. El campo `Msg` almacena el mensaje a mostrar, el campo `Extra` almacena aquella información más específica que necesitan algunos tokens para funcionar (se encuentra vacío para los otros), y el campo `Chat`, con el ID del chat de Telegram por al cual mandar el aviso.

Luego, transforma esa información a formato JSON, y lo manda en el cuerpo del pedido al server. El server se encarga del registro del token y el almacenamiento de los datos, y devuelve el `tokenID` resultante, que esta función devuelve al comando para que continúe con la creación del token.

Esta arquitectura permite que agregar un nuevo formato de honeytoken sea tan simple como agregar un comando nuevo, y que el mismo utilice la función `createToken` (además de definir el correspondiente handler en el server). Se detallan a continuación el funcionamiento e implementación de los distintos tipos de tokens disponibles.

3.1.1) QR

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do.

3.1.2) Binario

La idea para crear un honeytoken a partir de un binario compilado es crear un nuevo binario que actúe de *wrapper* del binario original. Es decir, que el binario resultante realice la alerta al servidor, y luego simule el comportamiento del binario original.

La función de creación de estos tokens tiene la siguiente forma:

```
func generateBinaryWrapper(cmd *cobra.Command, args []string) {

  tokenID := CreateToken(msg, "", chat)

  data, err := os.ReadFile(in)
  b64 := base64.StdEncoding.EncodeToString(data)

  code := strings.ReplaceAll(wrapperTemplate, "{{B64}}", b64)
  code = strings.ReplaceAll(code, "{{Endpoint}}",
                            serverURL+"/bins/"+tokenID)

  os.WriteFile("tmp.go", []byte(code), 0644)

  outCmd := exec.Command("go", "build", "-o", out, "tmp.go")
  outCmd.Stdout = os.Stdout
  outCmd.Stderr = os.Stderr
  outCmd.Run()

  os.Remove("tmp.go")
}
```

Primero, llama a la función de creación de tokens mencionada anteriormente, y consigue el tokenID del nuevo token.

Luego, lee el binario compilado que fue pasado como flag, y lo codifica en base64. Luego, toma el contenido de `wrapperTemplate` y le "inyecta" el binario codificado y la URL a la que debe dar aviso.

Finalmente, crea un archivo llamado `tmp.go` con los contenidos de dicho template (*wrapper* + binario original), crea un comando que lo compila, lo ejecuta, y remueve el archivo fuente. Este binario compilado (llamado `tmp` a menos que se utilice la flag *out*) es el honeytoken final.

El template que se compila para crear el binario final tiene la siguiente forma:

```
const encoded = "{{B64}}"
const endpoint = "{{Endpoint}}"

func sendAlert() {
  client := http.Client{
      Timeout: 2 * time.Second,
  }
  client.Get(endpoint)
}

func main() {

  sendAlert()

  data, _ := base64.StdEncoding.DecodeString(encoded)
  tmpDir, _ := os.MkdirTemp("", "honey-*")
  real := filepath.Join(tmpDir, "realbin")
  os.WriteFile(real, data, 0755)

  cmd := exec.Command(real, os.Args[1:]...)
  cmd.Stdout = os.Stdout
  cmd.Stderr = os.Stderr
  cmd.Stdin  = os.Stdin

  err := cmd.Run()
  if err != nil {
      if e, ok := err.(*exec.ExitError); ok {
          os.Exit(e.ExitCode())
      }
      panic(err)
```

```
  }
}
```

Lo primero que hace al ser ejecutado es mandar la alerta al servidor, a la URL inyectada por el generador. Con el objetivo de pasar más desapercibido, tiene un timeout de 2 segundos (por si el servidor no contesta).

Luego, decodifica el binario original compilado, crea un directorio temporal en `/tmp` con un sufijo aleatorio, y crea un archivo ahí dentro con los datos del binario.

Finalmente, crea un comando de ejecución del archivo recién creado, le pasa los mismos *file descriptors* de entrada, salida y error, y lo ejecuta. Además, en caso de que el binario original finalice con algún código de error, el binario wrapper finaliza de la misma forma.

3.1.3) PDF

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do.

3.1.4) IMG

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do.

3.1.5) CSS

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do.

3.2) Server

El **server** también se encuentra implementado en Golang, y fue utilizado un framework de manejo de peticiones **HTTP** y creación de aplicaciones web llamado Gin. Además, se utiliza una base de datos online llamada MongoDB Atlas.

La función que inicia el server es la siguiente (omitiendo manejo de errores):

```
func StartServer() {
  r := gin.Default()

  r.GET("/", func(c *gin.Context) {
    c.Data(200, "text/html; charset=utf-8", []byte(morfeoString))
  })

  mongoURL := [...]
  client, err := mongo.Connect(context.Background(),
                               options.Client().ApplyURI(mongoURL))

  collection := client.Database("fcen").Collection("tokens")
  tokenController := handlers.NewTokenController(collection)

  // Para que el controller esté disponible en los handlers
  r.Use(func(c *gin.Context) {
    c.Set("tokenController", tokenController)
    c.Next()
  })

  r.POST("/tokens", handleNewToken)

  handlers.HandleQRs(r)
  handlers.HandleIMGs(r)
  handlers.HandleCSS(r)
  handlers.HandlePDFs(r)
  handlers.HandleBINs(r)

  port := os.Getenv("PORT")
  if port == "" {
    port = "8000"
  }

  r.Run(":" + port)
}
```

En ella, primero se define un **router**, donde se van a definir los endpoints del server y sus respectivos handlers.

El **GET** a `/` devuelve **`morfeoString`**, que no es más que un simple HTML que muestra el nombre del sistema. Debido a que la plataforma que fue utilizada para hacer el deploy "duerme" a las aplicaciones que no son utilizadas por un tiempo, este endpoint es utilizado para "despertar" a la aplicación.

Se construye luego la URL de conexión a la base de datos usando variables de ambiente (omitida por espacio), y se realiza la conexión. Se obtiene la colección de los tokens, y para facilitar el uso de la misma se crea un `tokenController`, que es guardado en el contexto para que esté disponible en todos los handlers.

Luego, se agrega en el **POST** a `/tokens` la función de registro de los mismos, `handleNewToken`. Esta función simplemente recupera los datos recibidos en el cuerpo del pedido, introduce un nuevo documento con los mismos en la base de datos y envía en la respuesta el `tokenID` generado.

Finalmente, se definen los handlers para las alertas de los tokens, y se levanta el server.

3.2.1) Handlers

Todos los handlers siguen una estructura como la siguiente, definiendo cada uno en su caso respuestas distintas o chequeos adicionales:

```
func HandleTokenType(r *gin.Engine) {
  r.GET("/tokenType/:tokenID", func(c *gin.Context) {
    tokenID := c.Param("tokenID")

    controller := c.MustGet("tokenController").(*TokenController)
    token, err := controller.GetToken(tokenID)

    chat := token.Chat
    alertText := "Fue activado el token " +
                 strings.ToLower(token.Msg) +
                 " desde la IP: " + c.ClientIP()
    Alert(alertText, chat)
  })
}
```

En ellos, se obtiene el `tokenID` de la URL, se recupera el `tokenController` del contexto, y se lo utiliza para obtener la información del token correspondiente. Luego, se llama al método `Alert`, que recibe el mensaje y se encarga de hacer la alerta (en este caso, mediante un mensaje de Telegram al ID guardado).

4) Y el manual

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do.