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 3
2) Funcionamiento 4
3) Implementación 5
    3.1) CLI 5
        3.1.1) QR 6
        3.1.2) Binario 6
        3.1.3) PDF 7
        3.1.4) IMG 7
        3.1.5) CSS 8
    3.2) Server 10
        3.2.1) Handlers 11
4) Manual de uso 12

1) Introducción teórica

El objetivo de este trabajo es la creación de un sistema funcional para el armado y alerta de **honeytokens** (señuelo digital que funciona como sistema de alarma; al accederlo, se delata la presencia de una actividad no autorizada en el sistema y una posible brecha de seguridad).

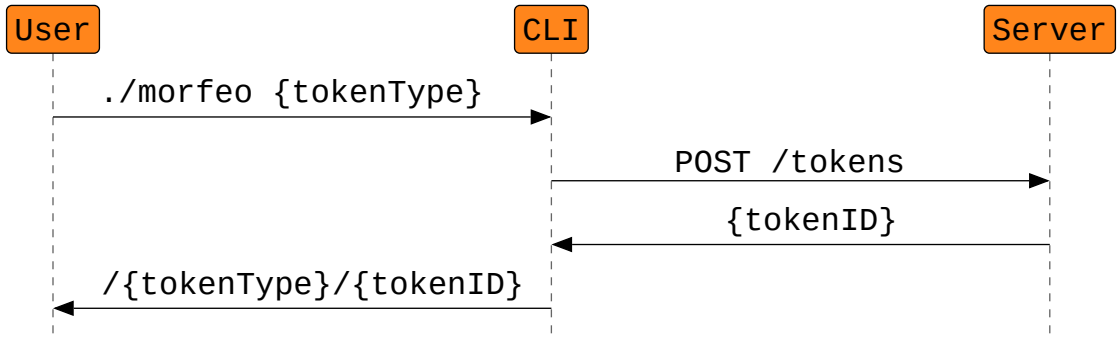
Existen muchos tipos de **honeytokens**, pueden ser archivos PDF, Word/Excel, binarios, Direcciones de correo, Cuentas de usuario falsas entre otras. En este trabajo, investigamos los mecanismos *Call Home*, inspirados en herramientas como *CanaryTokens*, para desarrollar **honeytokens** en los siguientes formatos: QR, Binarios, PDF, IMG y CSS.

- QR: se encapsula una URL configurada para actuar como nuestro canary token. Al ser escaneado por un atacante, se genera una solicitud hacia el servidor que controla el investigador. Al recibir esta solicitud en un endpoint, se constituye el evento de callback y la confirmación inmediata de la intrusión o el acceso no autorizado. El servidor luego activa la alerta de seguridad sin necesidad de interactuar o exponer datos sensibles y se redirige al usuario a otra URL con un codigo 302.
- Binario: como el caso anterior, se hace un callback a nuestro endpoint controlado pero se activa de otra manera. En este **honeytoken** hacemos un wrapper de nuestro codigo de alerta sobre un programa legitimo. Cuando se ejecuta el binario original se activa nuestra alerta que llega al servidor mediante el pedido HTTP mientras que el programa original corre sin problema.
- PDF: en este tipo de **honeytoken** la activacion de la alarma se efectua cuando se abre el documento en cuestion. Una de las particularidades es que hay variantes respecto a como puede hacerse esto, ya sea con codigo JavaScript embebebido que realiza el pedido HTTP o con la busqueda a un recurso externo (como una referencia a una imagen que apunta a una URL alojada nuestro servidor). El pedido al servidor se hace con el lector, que ejecuta el script de inicio, y el codigo JS incrustado realiza la petición HTTP o este interpreta la referencia externa, intentando descargar la imagen para mostrar el PDF y termina enviando la solicitud al servidor.
- CSS:

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

En primer lugar, el comando acepta los siguientes flags:

– `msg` (obligatorio): Identificador único del honeytoken

– `chat` (obligatorio): ID del chat de Telegram donde se recibirá la alerta al activarse

– `in` (opcional): Path a una imagen existente que se desee utilizar para generar el honeytoken

– `out` (opcional): Ruta del archivo HTML de salida (por defecto: honeytoken_image.html)

Cuando se pasa el parametro **`--in`**, el programa:

1. Se fija si existe la imagen

2. Extrae las dimensiones de la imagen usando `image.DecodeConfig()`
3. Genera un archivo HTML que contiene:
    - La imagen original visible
    - Un pixel invisible de 1x1 que apunta a la URL indicada

Ademas, se genera un archivo SVG con estructura similar:

- Elemento `<image>` principal con la imagen original
- Elemento `<image>` de 1x1
- ViewBox configurado según las dimensiones originales

Si no se proporciona imagen de entrada, el sistema crea:

- HTML con únicamente un pixel invisible sobre un fondo blanco
- SVG de 1x1 píxel conteniendo solo el honeytoken

Ambos formatos utilizan la técnica de *tracking pixel*: cuando se abre con algun editor de imagenes (no todos) el HTML o SVG, automáticamente realiza una petición HTTP GET a la URL embebida en la imagen

3.1.5) CSS

El canary token de CSS es particular por que no te avisa cuando el archivo es usado sino cuando tu pagina web fue clonada, es por ello que en este caso a las flags ya mencionadas se le agregan `in`, `out` y `dominio`. Las primeras dos flags indican cual es el arhivo CSS que se desea tokenizar y el nombre del token final (de forma predeterminada crea un archivo con el mismo nombre pero que arranca con new_). Por su parte la flag `dominio` indica cual es el dominio de la pagina del usuario. El funcionamiento del mismo consta de insertar al final del archivo CSS lo siguiente:

```
body {
    background: url(https://morfeo-c8s3.onrender.com/fondo/{token_ID}) !
important;
}
```

Esta instruccion de CSS tendra el efecto de hacer un pedido GET a nuestro servidor en busca de una imagen, en este pedido los buscadores agregan un header llamado Referer el cual indica desde que dominio se hizo el pedido. Por otra parte la flag !important le indica al buscador que no se debe saltear este background, asegurando que siempre se pida.

Por su parte nuestro servidor al recibir el pedido GET revisara el campo Referer y comparara su contenido con el dominio del token, luego si el campo esta vacio se alerta al usuario indicando que es posible que se clonase la pagina y si el campo no coincide con el dominio de la pagina original se alerta que la pagina fue clonada indicando el dominio de la pagina clonada.

El token funciona pero se le podrian realizar algunas mejoras:

- Compatibilidad con otros formatos: En desarollo web se suelen usar otros lenguajes que luego son traducidos a CSS, como es el caso de SASS. Una posible mejora seria hacer que

el token sea compatible con dichos formatos para que no se tenga que volver a crear el token cada ves que se recompile el formato de alto nivel.

- Dificultar la busqueda: Podriamos tomar algunas medidas para dificultar encontrar el token, por ejemplo se podria colocar en un lugar aleatorio del codigo CSS (no cambia mucho pero es algo), usar clases ya armadas es decir no crear una nueva definicio de body{} si ya existe una, etc.
- Reducir Alertas: Podriamos hacer que si se detecta varias veces un clon desde el mismo dominio solo se alerte una ves, ademas se podria usar un timer para que pasado un tiempo se permita volver a alertar.

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) Manual de uso

Para poder usar la herramienta, es necesario clonar el repositorio y tener Golang instalado. Luego, se deben ejecutar los siguientes comandos en la raiz del repositorio:

```
$ cp env-samples .env
```

```
$ go build
```

Esto generará un ejecutable llamado `morfeo`. Luego, al ejecutar

```
$ ./morfeo
```

desde donde se encuentre, se podrá ver la salida:

```
Usage:
  morfeo [command]

Available Commands:
  bin         Genera un honeytoken a partir de un binario
  css         Genera el honeytoken de css para paginas clonadas
  help        Help about any command
  image       Genera un honeytoken de imagen
  pdf         Genera el honeytoken de pdf
  qr          Genera el honeytoken de qr

Flags:
      --chat string   Chat ID al cual enviar la alerta al ser activado
  -h, --help          help for morfeo
      --msg string    Identificador del token

Use "morfeo [command] --help" for more information about a command
```

Para poder recibir las alertas, es necesario activar el bot que las manda. Para hacer esto, se entra al chat con el mismo, y se le da `start`.

Luego, simplemente se ejecuta `./morfeo [command] [flags]`. En caso de necesitar más información sobre un comando, ejecutarlo sin flags o con la flag `--help` imprime más detalles sobre el mismo.

Algunos comandos tienen flags que otros no tienen, pero hay dos flags que todos los comandos comparten:

1. `--msg`: Un mensaje para identificar al token que está siendo creado, que será mandado en el mensaje de alerta cuando el mismo sea activado.
2. `--chat`: El **ID** del chat de Telegram al que será mandada la alerta cuando el token que está siendo creado sea activado. Para encontrarlo, se le puede escribir a este bot.

Ejemplo de uso:

Para crear un código QR que actúe como honeytoken:

```
$ ./morfeo qr --msg "de ejemplo" --chat {chatID}
```

Esto genera un QR que al ser escaneado produce la siguiente alerta mediante Telegram: