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

El objetivo de este trabajo es la creación de un sistema funcional para el armado y alerta de **honeytokens**. Los honeytokens son artefactos digitales deliberadamente falsos que no tienen uso legítimo, pero están diseñados para parecer reales. Su propósito es detectar accesos no autorizados, exfiltración de datos o movimiento lateral: cualquier interacción con un honeytoken es por definición sospechosa, y genera una señal de alerta.

Se utilizan tanto en prevención y detección de intrusiones como en análisis forense, ya que permiten identificar vectores de ataque, superficies expuestas y comportamientos del adversario sin interferir con sistemas productivos.

Existen muchos tipos de honeytokens, pueden ser archivos PDF, Word/Excel, binarios, direcciones de correo, cuentas de usuario falsas, registros DNS, entradas en bases de datos, entre otras. En este trabajo, desarrolamos una herramienta capaz de crear honeytokens en los siguientes formatos: QR, Binarios, PDF, IMG y CSS.

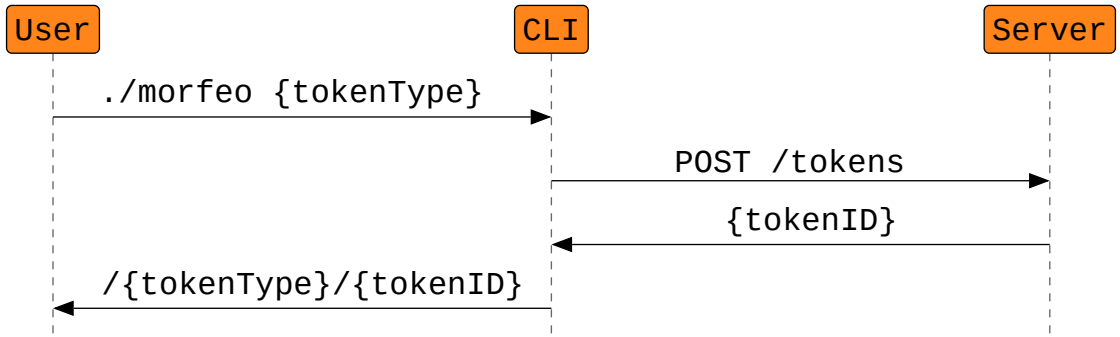
Todos los tokens desarrollados en este trabajo comparten el mismo mecanismo de *Call Home*: un pedido HTTP/HTTPS a un endpoint controlado en nuestro servidor, desde el cual podemos detectar el pedido y realizar la alerta. Esto lo logramos mediante un pedido directo, o el uso de un recurso externo que está alojado en nuestro servidor.

Otros mecanismos de activación pueden ser: DNS callbacks, donde el uso del token provoca una resolución DNS hacia un dominio controlado; SMTP callbacks, típicos en credenciales o direcciones que generan un envío de correo; integraciones con servicios cloud (por ejemplo, accesos a buckets, funciones o APIs que registran eventos); entre otros.

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 honeytoken es activado, la solicitud **GET** es mandada a una URL de 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

El honeytoken del QR es uno de los más sencillos, y no requiere muchos parametros para funcionar. La idea de este token es detectar el escaneo de un código QR.

Luego de la creación del token mediante la función `createToken`, se arma una URL apuntando a nuestro servidor, que es la que se codifica en el código QR (mediante una libreria llamada *"barcode"*). Luego, una vez que alguien entra a esa URL, el servidor detecta el pedido y genera la alerta.

Para disfrazar el uso del token, el mismo tiene una URL a la cual la persona que lo escanea es redirigido. Por default, esta URL es a Google, pero la misma puede configurarse mediante la flag `--redirect`.

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

La idea de este honeytoken es insertar en un documento PDF código JavaScript para alertar cuando dicho documento es abierto por un tercero. Se basa en la capacidad de los documentos PDF de ejecutar código JavaScript al momento de su apertura, especialmente en Adobe Acrobat Reader. La función que crea estos tokens es la siguiente:

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

  tokenID := CreateToken(msg, "", chat)
  url := serverURL + "/pdf/" + tokenID

  injectedCode := fmt.Sprintf(`
        var cURL = "%s";
        app.launchURL(cURL, true);
    `, url)

  f, err := os.Open(in)
  defer f.Close()

  reader, err := model.NewPdfReader(f)
  nPages, err := reader.GetNumPages()
  writer := model.NewPdfWriter()

  for i := 1; i <= nPages; i++ {
    page, err := reader.GetPage(i)
    err = writer.AddPage(page)
  }

  dict := core.MakeDict()
  dict.Set("S", core.MakeName("JavaScript"))
  dict.Set("JS", core.MakeString(injectedCode))
  err = writer.SetOpenAction(dict)

  fOut, err := os.Create(out)
  defer fOut.Close()

  writer.Write(fOut)
}
```

Primero se genera un tokenID y se construye la URL de activación que apunta al servidor. Luego, se genera el código a inyectar (`injectedCode`) que usará la función `app.launchURL()` para intentar abrir la URL de activación:

```
app.launchURL('https://{serverURL}/bins/{tokenID}', true);
```

Se lee el archivo de entrada, y se copian todas las páginas del PDF original a uno nuevo sin modificar su contenido. En PDF, el mecanismo para ejecutar código al abrir es la **OpenAction**. Se crea un diccionario de acción y se configura para ser de tipo JavaScript con `/S /JavaScript`. El código generado se establece como el valor de la clave `/JS`.

Finalmente, el diccionario se establece como el OpenAction del documento utilizando `writer.SetOpenAction(dict)` y se guarda el contenido del PDF modificado (páginas originales + el OpenAction inyectado) en el archivo de salida `out`.

Cuando el PDF se abre en un visor compatible, se ejecuta el JavaScript que realiza una petición GET a la URL de nuestro servidor, y se realiza la alerta. Además, como este código JavaScript abre un navegador, también se realiza una redirección a la página de Adobe, para esconder el token.

3.1.4) IMG

La idea de este honeytoken es insertar en un archivo HTML o SVG un recurso externo (en este caso, una imagen) que se encuentra ubicado en nuestro servidor. Luego, cuando el archivo es abierto y el recurso es cargado, el servidor detecta esa apertura y realiza la alerta.

Cuando se pasa el parametro **`--in`**, el programa se fija si existe la imagen, extrae las dimensiones de la imagen usando `image.DecodeConfig()`, y genera un archivo HTML que contiene la imagen original visible y un pixel invisible de 1x1 que apunta a la URL indicada.

Ademas, se genera un archivo SVG con estructura similar: un elemento `<image>` principal con la imagen original, un elemento `<image>` de 1x1, y un ViewBox configurado según las dimensiones originales.

Si no se proporciona imagen de entrada, el sistema crea un HTML con únicamente un pixel invisible sobre un fondo blanco, y un 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 honey token de CSS es particular, ya que no detecta cuando el archivo es utilizado, sino que detecta cuando una página web fue clonada. Para ello, este token recibe tres flags adicionales: `in`, `out` y `dominio`. Las primeras dos flags indican cuál es el arhivo CSS que se desea tokenizar y el nombre del token final (de forma predeterminada crea al archivo con el mismo nombre que el original). Por su parte la flag `dominio` indica cual es el dominio de la pagina del usuario.

El mismo funciona insertando al final del archivo CSS lo siguiente:

```
body {
  background: url({serverURL}/fondo/{token_ID}) !important;
}
```

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

Cuando el servidor recibe el pedido, revisa el campo Referer y compara su contenido con el dominio del token. Si el campo está vacío, 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, y se indica el dominio de la pagina clonada.

Algunas mejoras que podrían realizarse son:

- Compatibilidad con otros formatos que luego son traducidos a CSS, como SASS.
- Tomar algunas medidas para dificultar la busqueda del token, como colocarlo en un lugar aleatorio del codigo CSS, usar los styles ya armados de haberlos si ya existe una, etc.

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

Hay dos maneras de utilizar la herramienta. La primer manera es compilando el binario manualmente a partir del código fuente. La segunda manera es mediante un contenedor de docker que la compila, y utilizar los comandos dentro del contenedor.

En ambos casos, luego de clonado el repositorio debe ejecutarse el siguiente comando:

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

para tener disponibles las variables de ambiente. Además, para poder recibir las alertas es necesario activar el bot que las manda. Para hacer esto simplemente se debe entrar al chat con el mismo, y darle `start`.

4.1) Compilando el código fuente

Teniendo Golang instalado, y estando en la rama main, se ejecuta en la raiz del repositorio:

```
$ 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
```

Luego, 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 obligatorias que todos los comandos comparten:

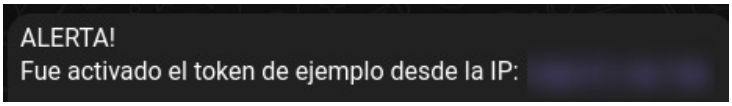
1. `--msg`: Un mensaje para identificar al token que está siendo creado, que será mandado en el mensaje de alerta cuando el mismo sea activado.
2. `--chat`: El **ID** del chat de Telegram al que será mandada la alerta cuando el token que está siendo creado sea activado. Para encontrarlo, se le puede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



4.2) Utilizando un contenedor de Docker

Si se desea en su lugar utilizar un contenedor de Docker, deben seguirse los siguientes pasos. Primero, pararse en la rama DockerTest, y crear los directorios que serán utilizados como entrada y salida con el contenedor:

```
$ mkdir tokens input tmp
```

Luego, deben completarse las variables del `UID` y `GID` del `.env` con los valores devueltos por los siguientes comandos, respectivamente: `id -u` e `id -g`. Una manera simple de realizar este paso es con el siguiente comando:

```
$ sed -e "s/^UID=.*/UID=\"$(id -u)\"/" \
      -e "s/^GID=.*/GID=\"$(id -g)\"/" \
      env-sample > .env
```

Luego, se realiza un build del contenedor:

```
$ docker compose build
```

Finalmente, cada vez que se desee utilizar la herramienta se ejecuta:

```
$ docker compose run --rm morfeo-cli
```

Esto abre una terminal en el directorio `/app` del contenedor. En él, se encuentra el binario `morfeo` ya compilado, y dos directorios más: `/app/input` y `/app/output`. Estos directorios se encuentran linkeados a los directorios creados previamente.

Para pasarle los archivos de entrada, los mismos deben ubicarse en el directorio `./input` del host, y los mismos aparecerán en `/app/input` del contenedor. De forma inversa, la herramienta generará los archivos en `/app/output`, y los mismos podrán ser encontrados en `./tokens` del host.

Luego, la utilización de la herramienta es similar a la mencionada previamente.