

Campus Internacional
CIBERSEGURIDAD

ENIIT
INNOVAT BUSINESS SCHOOL



MÁSTER EN ANÁLISIS DE MALWARE, REVERSING Y BUG HUNTING



Universidad Católica de Murcia
ENIIT - Campus Internacional de Ciberseguridad

Estudio práctico de técnicas de ofuscación y contramedidas aplicables

PRESENTA

María San José Seco
@drkrysSrng/freyja

PROFESOR

David García

ASIGNATURA

Trabajo de Fin de Máster

3 de septiembre de 2023

Índice

1. Introducción	4
1.1. En qué consiste la ofuscación	4
2. Diagrama evolutivo del malware	5
3. Tipos de Ofuscación de malware	6
3.1. Packing	6
3.2. Inserción de Dead-Code o código basura	6
3.3. XOR	6
3.4. Reasignamiento de registros	7
3.5. Sustitución de Instrucciones	7
3.6. Base64	7
3.7. Transposición de Código	7
3.7.1. Reordenamiento de subrutinas	7
3.8. Integración de Código	8
3.9. Expresiones MBA	10
3.10. Expresiones Opacas	10
4. Encriptación, Compresión y Metamorfismo	12
4.1. Oligomorfismo	12
4.2. Polimorfismo	13
4.3. Metamorfismo	14
4.4. Motor Metamórfico	14
4.4.1. Pasos del Motor Metamórfico para desencriptar	15
4.5. Encriptación	15
4.6. Compresión	16
5. Análisis de Entropía	18
6. Importancia de las amenazas de Javascript en Windows	20
6.1. WJworm	20
6.2. WSHRat	20
6.3. STRRAT	20
6.4. BlackByte Ransomware	21
6.5. Carbanak/FIN7 JavaScript Backdoor	21
7. Pasos para Desofuscar un Código Malware en JavaScript	22
7.1. JavaScript Beautifully	22
7.2. IIFE: Immediately Infoked Function Expression	22
7.3. Expresiones, operador coma, parseInt() y toString()	23
7.4. Consiguiendo el constructor de la función	26
7.5. Operadores Lógicos	27
7.6. Ofuscación de caracteres y Strings	29
7.6.1. Conjunto de enteros evaluados con eval	29
7.6.2. Caracteres en Unicode	30
7.6.3. Caracteres en Hexadecimal	30
7.6.4. Caracteres de URL	30
7.7. Base64	31



8. Freyja Deobfuscation Tool	32
8.1. Funcionamiento	32
8.2. Entropía	32
8.2.1. Línea por Línea	32
8.2.2. Fichero completo	33
8.2.3. Código fuente	33
8.3. Desofuscación	34
8.3.1. Ejemplo de Nivel 1: Beautify	35
8.3.2. Ejemplo de Nivel 2: Parsear Hexadecimal	36
8.3.3. Ejemplo de Nivel 3: Parsear Unicodes	37
8.3.4. Ejemplo de Nivel 4 y 5: Reemplazar la función <code>toString</code>	38
8.3.5. Ejemplo de Nivel 6: Parsear eval si tiene una lista de números	39
8.3.6. Ejemplo de Nivel 7: Parsear la función <code>unescape</code>	40
8.3.7. Ejemplo de Nivel 8: Parsear conjuntos de caracteres	41
8.3.8. Ejemplo de Nivel 9: Parsear la función <code>parseInt</code>	43
8.3.9. Ejemplo de Nivel 10: Parsear las concatenaciones de caracteres	43
8.4. Búsquedas de Base64	44
9. Bibliografía	50



1. Introducción

1.1. En qué consiste la ofuscación

Tanto para proteger la propiedad intelectual o intercambiar secretos, además de prevenir la ingeniería inversa de una aplicación software, el código fuente se suele ofuscar.

Una forma de ofuscación es cambiar parte del código fuente, para dificultar su lectura, comprensión y su análisis. Un ofuscador es una herramienta que convierte el código fuente de un programa en otro código distinto que hace lo mismo pero de una forma mucho más difícil de leer y entender.

Además, también es una de las muchas formas que tiene el malware para evadir el análisis estático o para evitar ser detectado por los métodos tradicionales anti-malware que se basan en hashes o firmas y strings para su detección.

Los antivirus tradicionales hacen sus análisis para detectar el malware, normalmente comparando el hash del fichero que están analizando con los hashes que tienen en sus Base de Datos, en el caso de un analista, sería por ejemplo, compararlo con los hashes en las bases de VirusTotal [8].

Otro método de análisis de código suele basarse en analizar strings. Consiste en buscar y analizar strings legibles de texto (en ASCII o Unicode) en un fichero que puede que no tenga caracteres legibles, porque sea un binario, por ejemplo. Esos strings de texto, pueden mostrarnos nombres de ficheros, IPs, URL, peticiones HTTP, claves de registro u otras cadenas que nos puedan indicar cómo funciona dicho malware.

Herramientas como YARA permiten realizar búsquedas programáticas sobre texto con el fin de realizar detecciones basadas en reglas. En el siguiente ejemplo, tenemos una regla YARA que busca tres patrones de texto[3]:

```
rule silent_banker : banker
{
    meta:
        description = "This is just an example"
        threat_level = 3
        in_the_wild = true

    strings:
        $a = {6A 40 68 00 30 00 00 6A 14 8D 91}
        $b = {8D 4D B0 2B C1 83 C0 27 99 6A 4E 59 F7 F9}
        $c = "UVODFRYSIHLNWPEJX0ZAKCBGMT"

    condition:
        $a or $b or $c
}
```

Figura 1: Ejemplo de reglas Yara para buscar tres Strings



2. Diagrama evolutivo del malware

Fuente: *Metamorphic Malware and Obfuscation -A Survey of Techniques, Variants and Generation Kits* [5]

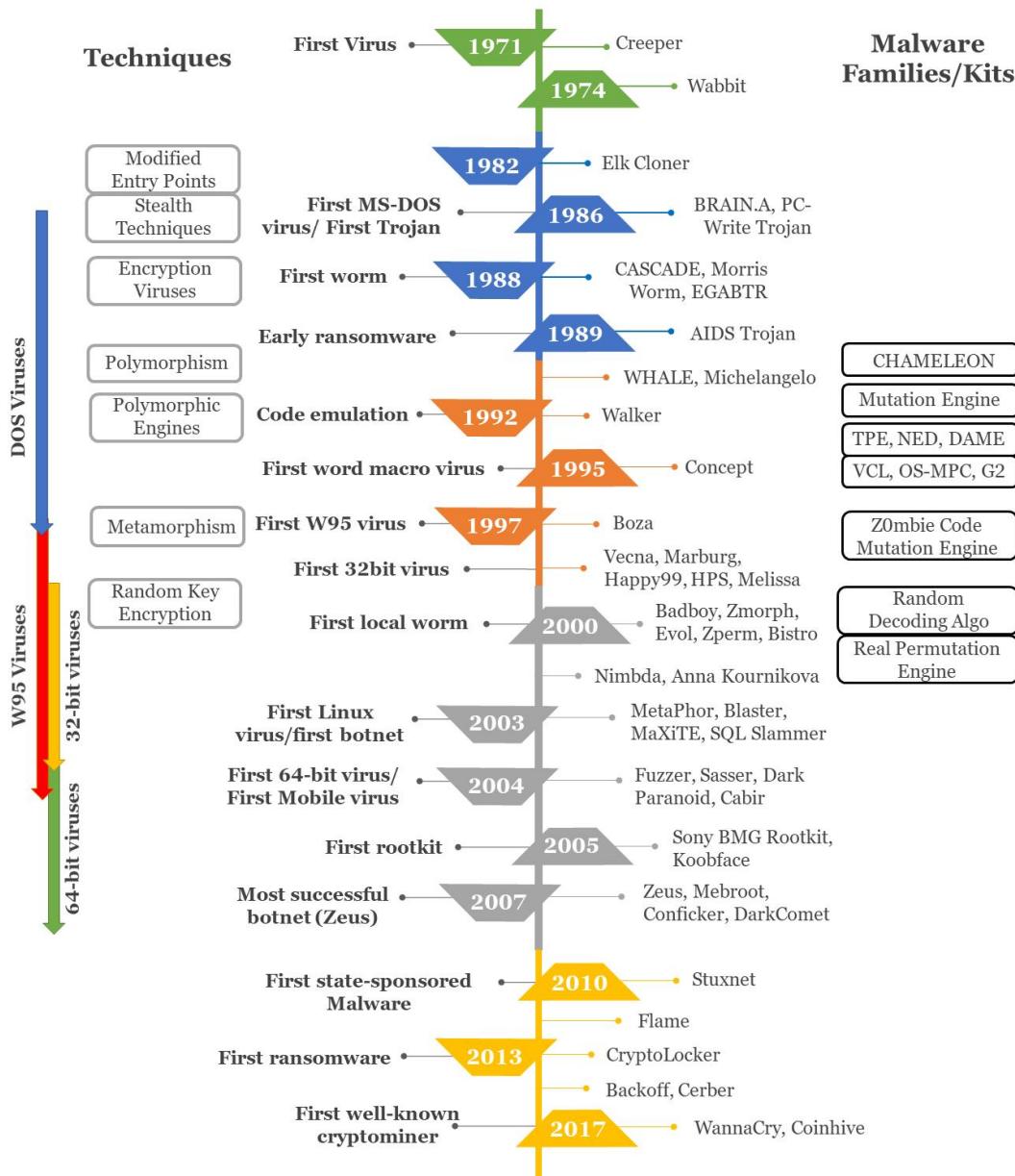


Figura 2: Diagrama histórico de las variantes del malware, técnicas y motores de mutación.

3. Tipos de Ofuscación de malware

3.1. Packing

Se comprime el fichero para que tenga una firma distinta al original. Además de eso, los strings también serán incomprensibles. También puede incluso comprimirse el binario con software para hacerlo más pequeño. El ejecutable comprimido se empaca dentro del código requerido para descomprimirlo en tiempo de ejecución, luego la ejecución se hace en memoria. Este tipo de ofuscación lo utilizan malware como Redline Stealer [9] y Hancitor [10]

Sin embargo, el hecho de que un programa esté empaquetado, ya es sospechoso, por lo que el malware moderno cada vez utiliza menos este sistema para pasar desapercibido.

3.2. Inserción de Dead-Code o código basura

Es una forma simple de cambiar la apariencia del programa y su funcionalidad. **NOP** es un ejemplo de comando en ensamblador. El código original se ofusca fácilmente insertando instrucciones **NOP**. Los análisis basados en firma podrían detectarlo simplemente eliminando esta instrucción.¹

NOP no hace nada. La ejecución continúa en la siguiente instrucción. Ningún registro ni flag es afectado por esta instrucción, simplemente se genera un delay en la ejecución o una reserva de espacio en memoria.

También, podemos ver encadenamiento de instrucciones como if-else o inserción de código basura que no afecta al funcionamiento del programa.

```

1 ; Before obfuscation
2 xor eax, eax
3 move eax, 0x2D
4 mov ecx, 0xA
5 ; After obfuscation
6 xor eax, eax
7 move eax, 0x2D
8 nop
9 nop
10 mov ecx, 0xA

```

Código 1: Ejemplo de Inserción de Dead-Code

3.3. XOR

Este método es bastante popular para ocultar datos y que no puedan ser analizados. Lo que hace es intercambiar los contenidos de dos variables dentro del código, como por ejemplo:

```

1 XOR EBX, EAX
2 XOR EAX, EBX
3 XOR EBX, EAX

```

Código 2: Ejemplo de XOR

XOR es una forma muy sencilla de ofuscación y encriptación ya que es muy fácil volver al valor original o recuperar una variable ofuscada ya que cualquier valor *xoreado* consigo mismo da 0. Veamos la tabla de operación de XOR:

Operación XOR		
Valor 1	Valor 2	Resultado
0	0	0
0	1	1
1	0	1
1	1	0

¹NOP significa en x86 o en intel *no operation*



3.4. Reasignamiento de registros

Es otra técnica simple que cambia registros por cada generación mientras el código del programa y su comportamiento siguen igual. En el siguiente código, se cambian los registros al inicio y se vuelven a cambiar al final, dejando la funcionalidad como estaba. El malware W95/Regswap [11]

```

1 ; Before obfuscation
2 mv eax ecx
3 xor ebx, ebx
4 test eax, ebx
5 ; After obfuscation
6 mov ebx, ecx
7 xor eax, eax
8 test ebx, eax

```

Código 3: Ejemplo de Reasignamiento de registro

3.5. Sustitución de Instrucciones

Hay una gran variedad de sustituciones que se pueden introducir. A continuación vemos un código en el que se sustituye la instrucción *push eax; mov eax, ebx* con *push eax; push ebx; pop eax*. Semánticamente son equivalentes pero *push* y *pop* es más lento que escribir directamente en el registro con *mov*. Esta técnica se utiliza en el malware W95/Zmist [12], y también en malware avanzados como Evol, MetaPHOR [13], Zperm [14] y Avron[15].

```

1 ; Before obfuscation
2 add eax, 05H
3 mov ebx eax
4 ; After obfuscation
5 add eax, 01H
6 add eax, 05H
7 push ebx
8 pop eax

```

Código 4: Ejemplo de sustitución de instrucciones

3.6. Base64

Es una forma de ofuscación muy conocida. Básicamente consiste en un esquema de codificación de caracteres, con el carácter = como padding. El alfabeto incluye desde la a-z, A-Z, + / y los caracteres numéricos del 0-9. La codificación funciona encadenando 3 caracteres para generar una cadena de 24 bits, que luego se divide en cuatro fragmentos de 6 bits, cada uno de los cuales se traduce a uno de los caracteres Base64. A pesar de ser una técnica trivial, es ampliamente utilizada, sobre todo para ofuscar cadenas.

3.7. Transposición de Código

Reordena la secuencia de instrucciones del código original sin afectar el comportamiento del código. Hay dos maneras:

- Cambiar las instrucciones de forma aleatoria y luego inserta ramas incondicionales o saltos para restaurar la ejecución original. No es difícil detectar este método porque el programa original se puede restaurar eliminando las ramas incondicionales o saltos.
- Crear nuevo código eligiendo y reordenando las instrucciones que son independientes entre ellas, es una técnica difícil de implementar, pero hace que la detección sea más difícil.

3.7.1. Reordenamiento de subrutinas

También conocido como Block Reordering o reordenamiento de bloques, es una técnica que reordena el flujo del proceso cambiando bloques del código que tienen subrutinas independientes. Si un código tiene n subrutinas de código, tendrá $n!$ permutaciones para reordenar dicho código.



```

1 ; Before obfuscation
2 mov eax , ebx
3 add ecx , edx
4 add eax , ecx
5
6 ; After obfuscation
7 add ecx , edx
8 mov eax , ebx
9 add eax , ecx

```

Código 5: Ejemplo de sustitución de reordenamiento de subrutina

```

1 ; Before obfuscation
2 mov eax , ecx
3 mov ebx , 10
4 mul ebx
5 add eax , 5
6 mov ecx , eax
7 ; After obfuscation
8 mov ebx , 10
9 jmp F1
10 jnk
11 F2: push edx; jnk
12 pop ecx
13 jmp F3
14 F1: mul , ebx
15 add ecx , 1; jnk
16 add ecx , 5
17 jmp F2
18 F3: mul ebx

```

Código 6: Ejemplo de Code Transposition, Subroutine Redordering and Garbage Code

3.8. Integración de Código

Fue por primera vez visto en W95/Zmist [12], le indica al código malicioso, unirse al código del programa objetivo, el malware decompila el programa e inyecta el código y lo reensambla para poder utilizarlo.

Lo que hace este tipo de malware es buscar un fichero/comando/código o binario para inyectar código propio dentro de él, como si fuese un código parásito y luego si es necesario lo recompila. Es una forma de ofuscación muy avanzada.

En el caso de Zminst, éste se inyectaba buscando instrucciones `jmp` en ficheros objetivo. Inyectando su código con una copia encriptada suya, reescribía el fichero.

A continuación vemos un ejemplo de una inyección de un programa a través de una instrucción de código `jmp` [6]



```
1 BITS 64
2
3 global _start
4
5 section .text
6 _start:
7     ; Save register state, RBX can be safely used
8     push rax
9     push rcx
10    push rdx
11    push rsi
12    push rdi
13    push r11
14
15    jmp parasite
16    message:   db  "-x-x-x-x- COMPILEPEACE : Cute little virus ^_^ -x-x-x-x-", 0xa
17
18
19 parasite:
20    ; print a message
21    ; ssize_t write(int fd, const void *buf, size_t count);
22    xor rax, rax           ; RAX = 0
23    add rax, 0x1           ; RAX = 1 [SYSCALL number]
24    mov rdi, rax           ; RDI = 1 (for STDOUT) [FD]
25    lea rsi, [rel message] ; RSI = (Address of message string) [BUF]
26    xor rdx, rdx           ; RDX = 0
27    mov dl, 0x39           ; RDX = 0x39 (57 in decimal) [COUNT]
28    syscall                ; Perform a software interrupt
29
30    ; Restoring register state
31    pop r11
32    pop rdi
33    pop rsi
34    pop rdx
35    pop rcx
36    pop rax
37
38    ; jmp to original host entry point (to be patched by kaal bhairav)
39    mov rbx, 0xFFFFFFFFAAAAAA
40    jmp rbx
```

Figura 3: Ejemplo de inyección de un programa parásito



3.9. Expresiones MBA

Las expresiones MBA o Mixed Boolean-Arithmetic sirven para ofuscar el código utilizando operadores booleanos o polinomios dejando la funcionalidad del código intacta. Es un método para preservar la semántica a través de transformaciones con simples expresiones, para dificultar el análisis del código. Esta técnica consiste en la mezcla de expresiones aritméticas y operaciones booleanas. El código binario con ofuscación MBA puede ocultar los datos y algoritmos de análisis estáticos y dinámicos, incluyendo los análisis avanzados.

```

1
2     def e1(x, y):
3         return x + y
4
5     def e2(x, y):
6         return (x + y) + 2 * (x & y)
7
8     print("Mba test e1 with 3 and 4 is:", e1(3, 4))
9     print("Mba test e2 with 3 and 4 is:", e2(3, 4))
10
11 #Resultado:
12 El resultado de e1 es: 7
13 El resultado de e2 es: 7

```

Código 7: Ejemplo de expresiones MBA equivalentes

A continuación podemos ver más ejemplos de expresiones booleanas y sus equivalentes en MBA:

Truth Value	Boolean Expr	MBA Expr
0000	0	0
0001	$x \wedge y$	$x \wedge y$
0010	$x \wedge \neg y$	$x - (x \wedge y)$
0011	x	x
0100	$\neg x \wedge y$	$y - (x \wedge y)$
0101	y	y
0110	$x \oplus y$	$x + y - 2 * (x \wedge y)$
0111	$x \vee y$	$x + y - (x \wedge y)$
1000	$\neg(x \vee y)$	$-x - y + (x \wedge y) - 1$
1001	$\neg(x \oplus y)$	$-x - y + 2 * (x \wedge y) - 1$
1010	$\neg y$	$-y - 1$
1011	$x \vee \neg y$	$-y + (x \wedge y) - 1$
1100	$\neg x$	$-x - 1$
1101	$\neg x \vee y$	$-x + (x \wedge y) - 1$
1110	$\neg(x \wedge y)$	$-(x \wedge y) - 1$
1111	-1	-1

Figura 4: Ejemplos de equivalencias en expresiones MBA

3.10. Expresiones Opacas

Normalmente se refieren a expresiones que siempre toman el valor de True o False, conocidas en tiempo de compilación pero son evaluadas en tiempo de ejecución. Aquí tenemos tres ejemplos donde añadiendo este tipo de expresiones condicionales, siempre se va a ejecutar el código que nos interese:

```

1
2     def o1(x):
3         if x % 2: # 5 cannot be divided by 2, always true

```



```

4         x = 1 << x
5         x = 2 * x + 9
6         return x
7
8     def o2(x):
9         if ((4 * x * x + 4) % 19) != 0:
10            x = 1 << x
11            x = 2 * x + 9
12            return x
13
14    def o3(x):
15        if ((4 * x * x + 4) % 19) != 0:
16            if x % 2:
17                x = 1 << x
18                x = 2 * x + 9
19                return x
20            else:
21                x *= 5
22                return x
23        else:
24            x += 85
25            return x
26
27 print("Opaque test o1: ", o1(5))
28 print("Opaque test o2: ", o2(5))
29 print("Opaque test o3: ", o3(5))
30
31 # Resultado
32 El resultado de o1 es: 73
33 El resultado de o2 es: 73
34 El resultado de o3 es: 73

```

Código 8: Ejemplo de expresiones Opacas equivalentes



4. Encriptación, Compresión y Metamorfismo

El metamorfismo y otras formas de ofuscación, son la parte principal de las amenazas que nos encontramos hoy en día. Al igual que las técnicas basadas en firma de los antivirus avanzan, también los niveles de ofuscación empleados por los actores para evadirlas.

El malware hace uso de Entry Point Obscuration (EPO) para evitar coherencia en el orden al ejecutar el archivo infectado. En este ejemplo, el encabezado apuntaría a una dirección que ejecutaría el código infeccioso que luego apuntaría al fichero host para que la ejecución del virus lo hiciera sin darse cuenta.

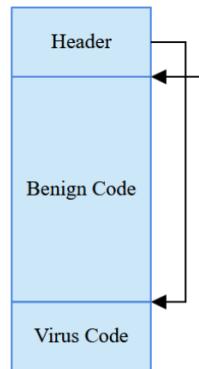


Figura 5: Entry Point Obscuration (EPO)

4.1. Oligomorfismo

Comenzó como una forma de evadir las técnicas de escaneo basadas en firma. Para evitar esto, tenemos las técnicas de escaneo tales como Wildcard y Mismatch. Teniendo en cuenta que el código de la infección suele estar añadido a un fichero, escaneos Top-and-Tail pueden ser efectivos para extraer las firmas de algunas partes del código. Además, el uso de sandboxes o emuladores sirve para ver cómo se extrae el malware y monitorear la memoria y el código que se dumpea. El primer malware Oligomórfico fue Whale DOS [16], identificado por primera vez en 1990. En la siguiente imagen, la rutina de desencriptado se utiliza para desencriptar el cuerpo y evitar la detección del malware.



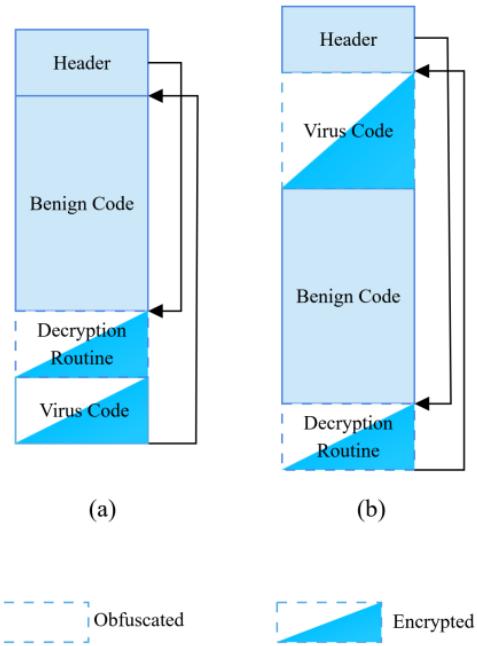


Figura 6: Malware Oligomórfico y Metamórfico

La limitación de esta técnica, es que el bucle de posibles descifradores es finito. Por ejemplo el W95/Memorial [17] tiene exactamente 96 desencriptadores para elegir. Una vez que se agota un generador oligomórfico, la única solución es introducir ofuscación en la Rutina de Desencriptado y así tendremos una forma infinita de rutinas de desencriptado dando lugar al malware de tipo Polimórfico. Dando lugar a la primera generación de malware Polimórfico como 1260 [18] y famosos generadores como Phalcon/Skism Mass-Produced Code Generator (PS-MPC) [19] y Virus Creation Lab (VCL)[20] que se siguen utilizando actualmente.

4.2. Polimorfismo

Este malware es capaz de desencriptar como ofuscar y recompilar todo en uno. El cuerpo desencriptado crea un nuevo desencriptador mutado, utilizando un algoritmo aleatorio de encriptado y luego permite al desencriptador encriptar antes de linkar las dos partes. El único problema son los emuladores, la ejecución en memoria puede ser detectada por los investigadores de malware. Las primeras generaciones de ofuscadores tenían el siguiente problema:

- El tamaño del código infeccioso era constante (Polimer.512.A y Vienna [21]).
- Adjuntando o preadjuntando al fichero objetivo, puede ser detectado por escaneo de firma.
- Segmentos de código similares entre generaciones del malware pueden detectarse por análisis de entropía.

Ejemplos de malware polimórfico:

- Luna. Uno de los primeros malware perlomórficos desarrollado por Bumblebee en España en el año 1999. Apareció en la cuarta versión de la revista 29A. [22]
- Loveletter. Uno de los primeros gusanos en alcanzar un gran número de daños.[23]
- Storm Worm Email. Responsable del 8 % de todas las infecciones gloables de malware. Utilizando como infección un archivo adjunto de un email. Su difícil detección era porque su código cambiaba cada 30 minutos aproximadamente. [24]



- CryptoWall. Es un ransomware que cifra los ficheros de la computadora de la víctima exigiendo el pago de un rescate para su descifrado. [27]
- Virlock. Se vio por primera vez en 2014, pero en septiembre de 2016 se descubrió que era capaz de propagarse a través de las redes a través de aplicaciones de colaboración y almacenamiento en la nube. No sólo cifra los ficheros sino que también los convierte en infectador de archivos binarios polimórfico. [26]
- CryptXXX. Distribuido por Angler Exploit Kit, encripta varios archivos almacenados en las unidades locales y extraíbles mediante RSA4096, generando una clave pública y otra privada. La clave privada se almacena en servidores remotos, también copia ficheros privados como cookies, datos de navegación... [28]
- CryptoLocker. Es un caballo de Troya que infecta la computadora y luego busca ficheros para cifrar, tanto discos duros, USB, unidades de red o en la nube. Sólo es para Windows. El cifrado es asimétrico. [29]
- WannaCry. Este ransomware se propaga aprovechando una vulnerabilidad en el protocolo de Windows Server Message Block (SMB). Este protocolo permite la comunicación entre máquinas Windows en red. Exploró en 2017 infectando a más de 230,000 computadoras en todo el mundo, causando daños valorados en miles de dólares. Fue difundido por EternalBlue, un exploit de Zero-Day que usa una versión antigua del protocolo SMB. Se cree que fue creado por la Agencia de Seguridad Nacional de EE.UU. (NSA) y luego obtenido por los Shadow Brokers. [30]

4.3. Metamorfismo

Este tipo de malware introdujo la idea por primera vez la idea de que no pueden existir dos generaciones de malware con firmas similares. En el gráfico anterior (b), se muestra el virus metamórfico. A diferencia del polimorfismo, el código viral está ofuscado, estando todo el virus presente en un estado ofuscado. Esto introduce la idea de que los Metamórficos son body-Polimórficos, que como resultado no tienen el cuerpo constante. Los primeros virus de tipo metamórfico fueron W95/Regswap en 1998 [11] seguido del W32/Ghost en el 2000 [31]. Este último contenía 10 submódulos, 3.6 millones de posibles variaciones para hacer un reordenamiento de subrutinas. En el caso del gráfico anterior(b), la separación entre el desencriptador y el cuerpo malicioso no es posible y la ofuscación hace que el cifrado ya no sea necesario. Además la rutina de descifrado se encuentra en el código benigno, así que no necesita desempacar para crear un cuerpo del malware constante como en el caso del polimorfismo. Uno de los generadores metamórficos más utilizados es W32/NGVCK creado en 2001 [32] creado en 2001. Este tipo de malware tiene un motor de mutación que contiene muchos subprocessos.

4.4. Motor Metamórfico

Es el responsable de la ofuscación/reconstrucción del binario para que éste funcione. Los componentes de este motor son los siguientes:

- **Disassembler:** Responsable de convertir el código binario en instrucciones de ensamblador.
- **Shrinker:** Eliminar la mayoría del código basura producido por generaciones anteriores o de otro código generado por ofuscación.
- **Permutor:** Lleva a cabo la mayoría de la ofuscación utilizando permutaciones y subrutinas, muchas veces de forma aleatoria. También hace inserción de instrucciones *jmp*
- **Expander:** Sustituye instrucciones para convertirlas en otro conjunto de instrucciones equivalente, además los registros se reasignan y las variables se vuelven a seleccionar utilizando tablas de sustitución. Código basura y código que no hace nada se añade y las funciones se alinean.
- **Assembler:** Reestructura el control de flujo y reconvierte el código ensamblador otra vez al código binario donde se vuelve funcional otra vez.
- **Viral Code:** Contiene las instrucciones principales que se ejecutarán en todas las generaciones de malware. También contiene las instrucciones que coordina el motor de mutación y otros componentes.



Los pasos de **Permutor** y **Expander** se utilizan de forma sofisticada en los W32/Zmist [12] y W32/Etap [33]

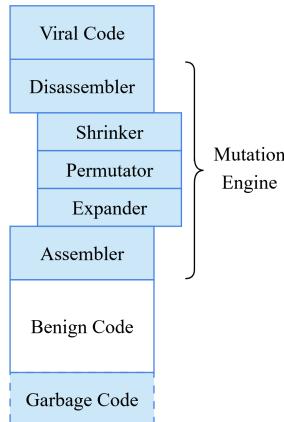


Figura 7: Componentes del Motor Metamórfico

4.4.1. Pasos del Motor Metamórfico para desencriptar

1. La rutina desencripta el cuerpo del Malware y ejecuta una instancia del mismo.
2. La Rutina de Desencriptado desencripta el motor de mutación y lo ejecuta.
3. El Shrinker desofusca el cuerpo del código malicioso.
4. La ofuscación se lleva a cabo introduciendo una nueva y única rutina de desencriptado.
5. El cuerpo del código malicioso se ofusca por el motor de mutación para producir una generación única utilizando varias técnicas . Luego se encripta utilizando un algoritmo único, una clave estática o una clave temporal.
6. Finalmente el motor de mutación es encriptado.

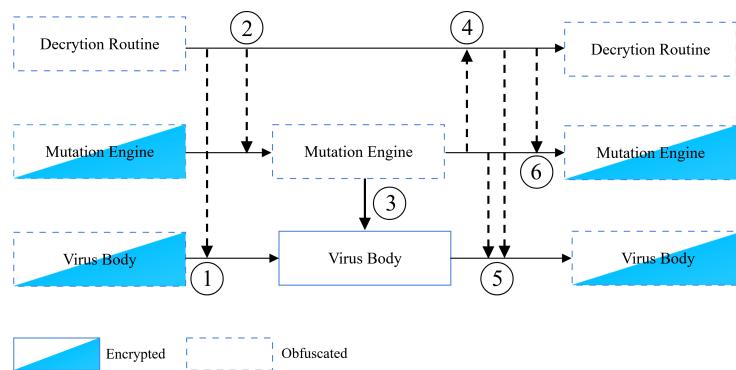


Figura 8: Pasos del Motor Metamórfico para Desencriptar

4.5. Encriptación

La primera forma de encriptación fue por el Malware CASCADE utilizando un simple *XOR* [34].



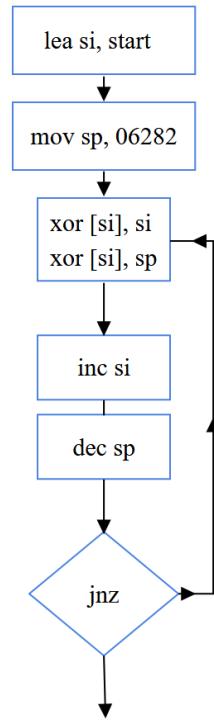


Figura 9: Diagrama de funcionamiento del malware CASCADE

Más adelante, el malware evolucionó de forma que en lugar de tener una sola encriptación, tienen docenas de ellas, como en el caso del DOS/Whale en 1990 [16] o de forma aleatoria como en el caso de W32/MetaPHOR [13]. La encriptación básica, puede desarrollarse con una única clave byte a byte, sin embargo hay formas alternativas en las que la clave se va actualizando a cada paso o incluso utilizando los caracteres encriptados.

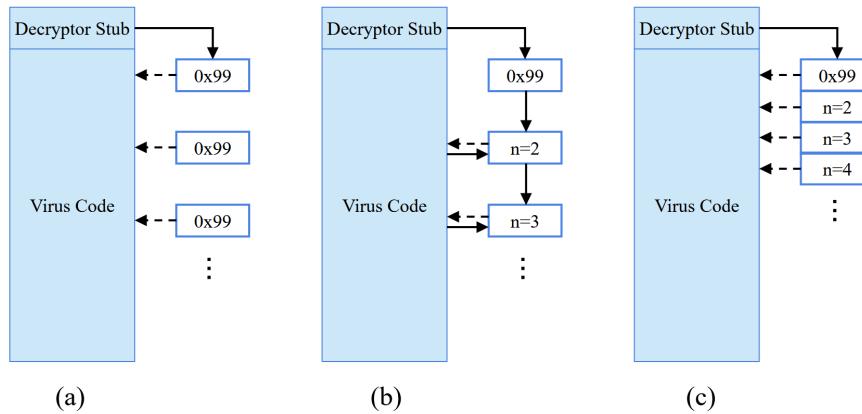


Figura 10: Tipos de encriptación. a. Clave reutilizada. b. La clave va cambiando por cada bloque. c. Se utiliza el código cifrado para encriptar cada bloque

4.6. Compresión

Representa un nivel adicional de ofuscación. Un Packer se define como una utilidad que aplica una forma de compresión al ejecutable, tanto para reducir el tamaño del fichero, evitar el análisis de entropía o introducir una capa



de ofuscación en el encabezado PE. Se estima que el 80 % del malware utiliza algún tipo de empaquetador así como el 90 % de todos los gusanos.

Dos de los Packer más populares son UPX [35] y ASPACK [36]

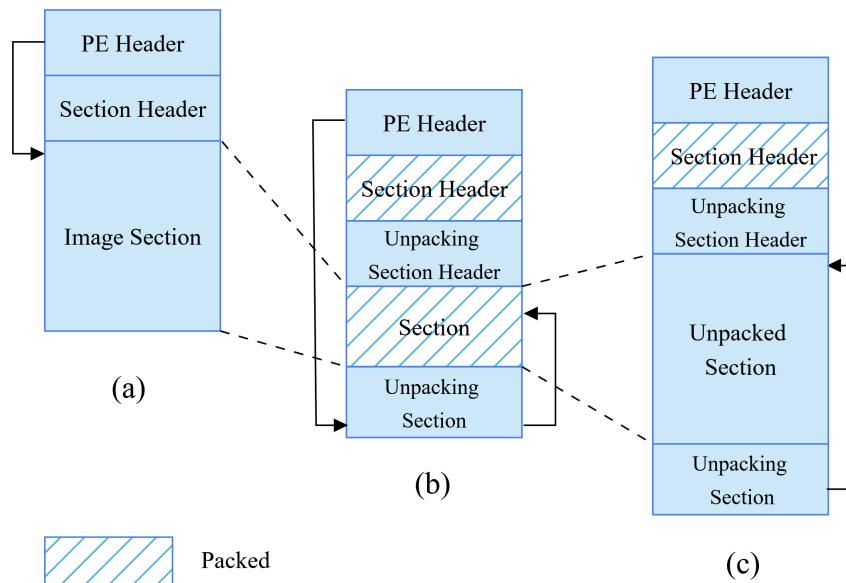


Figura 11: Pasos en un Packer

5. Análisis de Entropía

El término científico Entropía se define generalmente como la medida de aleatoriedad o desorden de un sistema, lo cual es importante para la evasión de Antivirus. El malware suele contener código altamente randomizado, encriptado u decodificado(ofuscado) para hacer el análisis y la detección difíciles. Uno de los métodos que utilizan los productos Anti Malware es un análisis de entropía para identificar ficheros potencialmente maliciosos y Payloads.

Es importante entender este concepto porque cuando se ofusca el código, se suele tener en cuenta que la entropía varía creada por los cambios que se eligen. Una cosa es cambiar la firma o hash pero no se suele tener en cuenta al nivel de entropía, pero los AV/EDR sofisticados sí lo hacen.

Hay un principio a tener en cuenta: Cuanta más entropía, más probable es que los datos estén ofuscados o encriptados y más probable es que el fichero o payload sea malicioso.

Claude E. Shannon introdujo una fórmula en el paper **A Mathematical Theory of Communication** que puede ser utilizado para analizar la entropía de un conjunto de datos. La fórmula es la siguiente:

$$H(X) = - \sum P(x_i) \log P(x_i)$$

Vamos a verlo en un script de Python:

```

1 def entropy(string):
2     "Calculates the Shannon entropy of a UTF-8 encoded string"
3
4     # decode the string as UTF-8
5     unicode_string = string.decode('utf-8')
6
7     # get probability of chars in string
8     prob = [float(unicode_string.count(c)) / len(unicode_string) for c in dict.fromkeys(list(
9         unicode_string))]
10
11    # calculate the entropy
12    entropy = - sum([p * math.log(p) / math.log(2.0) for p in prob])
13
14    return entropy

```

Código 9: Algoritmo de Shannon en Python

En teoría, cuando la entropía supera **3.75** significa que ese texto no está escrito por un humano. Utilizando las adaptaciones del algoritmo de Shannon en [37] y [38] vamos a hacer una herramienta que nos indique el nivel de ofuscación de un fichero en JavaScript.

```

1 #!/bin/python3
2 import math
3
4 def entropy_check(string):
5     "Calculates the Shannon entropy of a UTF-8 encoded string"
6
7     # decode the string as UTF-8
8     unicode_string = string.decode('utf-8')
9
10    # get probability of chars in string
11    prob = [float(unicode_string.count(c)) / len(unicode_string) for c in dict.fromkeys(list(
12        unicode_string))]
13
14    # calculate the entropy
15    entropy = - sum([p * math.log(p) / math.log(2.0) for p in prob])
16
17    return entropy
18
19    filename = input("Write down the path of the file to analyze >")
20
21    option = input("What do you want to do? Analyze file [file] or line per line [line]>").upper()
22    (
23
24    if option == "FILE":
25        with open(filename, 'rb') as f:

```



```
25 content = f.read()
26 print("Test2", entropy_check(content))
27
28
29 elif option == "LINE":
30     with open(filename, 'rb') as f:
31         content = f.readlines()
32
33     for line in content:
34         entropy = entropy_check(line)
35         if entropy > 3.75:
36             print(line[:-1])
37             print("This line has High Entropy", entropy)
```

Código 10: Herramienta de análisis de Entropía

```
Write down the path of the file to analyze >javascript-malware-collection/2017/20170507/20170507_0d258992733e8a397617eae0cbb08acc.js
What do you want to do? Analyze file [file] or line per line [line]>file
Test2 4.460820147920733
```

Figura 12: Análisis del Fichero Completo

```
Write down the path of the file to analyze >javascript-malware-collection/2017/20170507/20170507_0d258992733e8a397617eae0cbb08acc.js
What do you want to do? Analyze file [file] or line per line [line]>line
b'var juEFqegXodwknWtlpOv, OKMwPHjJQymtdVX, ojLqWfitUlxbncuQG, OlwpPLsvRDhBKQEY, hxEobzlsRgNfcDX, lAYcunLBjCUqezpbkw, bLGgrdCMRovlpnx, IGhuToEwXlJBQPS,
This line has High Entropy 5.617101379143724
b'function JIUFOiVePNXYTRaOyB(vPsiWmlGbuvYgxoeO) { \r'
This line has High Entropy 5.118562939644918
b'KgJtXdpCLGQ0uRMmYeV = 'jKAxegJjdOfuKAxegJjdOfEKAxegJjdOfFKAxegJjdOfqKAxegJjdOfeKAxegJjdOfgKAxegJjdOfXodKAxegJjdOfwkKA'+\r"
This line has High Entropy 4.414777089775276
b'"xegJjdOfnWtKAxegJjdOfLKAxex'+\r"
This line has High Entropy 4.0667842134731025
b'"gJjdOfpOKAxeg'+\r"
This line has High Entropy 3.836591668108979
b'"JjdOfvKAxegJjdOf =+\r"
This line has High Entropy 4.001822825622231
b'" KAxegJjdOfnKAxeg'+\r"
This line has High Entropy 3.9139770731827506
b'"gJjdOfw KAxegJjdOfAcKAxegJjdOfftKAxe'+\r"
This line has High Entropy 4.002078406900581
b'"gJjdOfiKAxegJjdOfvKAx'+\r"
```

Figura 13: Análisis del Fichero por Líneas



6. Importancia de las amenazas de Javascript en Windows

Los ataques basados en script se han convertido en una amenaza importante en los últimos años. Algunas estimaciones sitúan estos ataques en el 40 por ciento o más de todos los ciber ataques globales. Un script puede ser cualquier cosa, desde una secuencia de comandos simples utilizados para la configuración del sistema, la automatización de tareas y otros fines generales, hasta un código mucho más avanzado, de múltiples capas y ofuscado. Entre los lenguajes de scripting más utilizados se encuentran PowerShell, VBScript y JavaScript.

Si bien los ataques de PowerShell son los más utilizados, los actores de amenazas maliciosas también utilizan JavaScript de Windows para muchos de los mismos fines. Fuera de un navegador, que ejecuta JavaScript de forma encapsulada, lo que limita en gran medida la interacción de ese código en el sistema operativo, Windows proporciona funciones para la ejecución de JavaScript con Windows Script Host (WSH), que ejecuta JavaScript (y otros lenguajes de secuencias de comandos compatibles con Windows) bajo los procesos de Windows *wscript.exe* y *cscript.exe*, lo que proporciona una superficie de ataque que los adversarios pueden aprovechar.

El malware JavaScript puede variar desde un simple dropper destinado a entregar malware adicional hasta partes de malware multipropósito con todas las funciones.

A continuación se enumerarán ejemplos de malware prominentes en el panorama JavaScript "puro" que a menudo desafían las firmas de detección estáticas mediante una gran ofuscación de código y sin emplear archivos binarios compilados. [39]

6.1. WJworm

Vengeance Justice Worm es un malware en JavaScript que combina características de Gusano, Robo de Información, Troyano de Acceso Remoto (RAT), Malware de Denegación de Servicio (DOS) y spam-bot. Fue descubierto en 2016.

Se propaga por adjuntos de email infectando dispositivos de almacenamiento externo.

Una vez que se ejecuta por la víctima, el VJWorm ofuscado, enumera los dispositivos instalados, y si un dispositivo se encuentra, lo infecta.

Obtiene información del sistema, del usuario, el anti-virus instalado, las cookies del navegador, la presencia de *vbc.exe* en el sistema (Compilador de Microsoft .NET de Visual Basic) para comprobar si .NET está instalado en el sistema y puede afectar para instalar malware adicional.

VJWorm enviará la información al Servidor de Command-and-Control y esperará las siguientes instrucciones como descargar y ejecutar malware adicional.

6.2. WSHRat

También conocido como Houdini, H-worm, Dunihi y otros alias, es otro malware *commodity malware*². Descubierto en 2013, fue desarrollado originalmente en VBS. Su variante en 2019, emergió en una versión de JavaScript de la versión inicial.

Como todos los Troyanos de acceso remoto (RATs), el propósito principal de WSHRAT es mantener el acceso a la máquina, ejecutando comandos remotos y descargar malware adicional.

Se propaga por adjuntos de email y también es capaz de infectar dispositivos externos.

Una vez ejecutado por la víctima, el altamente ofuscado WSHRat seguirá un proceso similar al descrito anteriormente con VJWorm, para obtener los datos del usuario y del sistema y reportarlos al Command-and-Control. A continuación infectar los dispositivos de almacenamiento externo y esperar a las siguientes instrucciones.

Las variantes en VBS han sido reportadas recientemente, involucradas en campañas de espionaje en la industria aeronáutica.

6.3. STRRAT

Es un RAT basado en Java, con un dropper/wrapper en JavaScript que fue descubierto en 2020. Su payload core (fichero .jar) contiene varias capas de ofuscación y codificación dentro del JavaScript wrapper/dropper.

²RAT, Troyano de Acceso Remoto.



STRRAT se propaga por adjuntos en emails. Su capacidad incluye funcionalidades de RAT (acceso remoto, ejecución remota de comandos), browser, email client harvesting³ y una funcionalidad ransomware, cifrando los ficheros añadiendo una extensión .crimson al dispositivo, que pueden ser recuperados fácilmente si su contenido no se modifica.

A diferencia de otros malware hechos en Java, no requiere tener Java instalado para infectar al sistema y funcionar. Cuando el dropper/wrapper de JavaScript se ejecute, se instala una versión de Java para que pueda ejecutarse el malware.

6.4. BlackByte Ransomware

Es un Ransomware que ha sido recientemente descubierto con un core payload que es una .DLL desarrollada en .NET con un wrapper en JavaScript. Emplea una ofuscación muy alta tanto el wrapper como la .DLL.

Una vez que el wrapper es ejecutado, el malware desofusca el payload lo ejecuta en memoria. La .DLL se descarga y BlackByte hará un checkeo del Sistema Operativo instalado y finaliza si el lenguaje del sistema es de Europa del Este.

A continuación chequeará la presencia de antivirus y sandbox relacionados con .DLL, intentando bypassar la AMSI⁴, borrar las shadow copies⁵ del sistema para evitar la recuperación del sistema y modificar algunos servicios del sistema como el Firewall. Cuando el sistema esté preparado para encriptar los ficheros, descargará una clave simétrica que utilizará para encriptar los ficheros del sistema, si este fichero no lo encuentra, finaliza su ejecución.

A diferencia de la mayoría del Ransomware actual, BlackByte utilizar una clave de cifrado simétrica ni genera una clave única de cifrado para cada víctima, lo que quiere decir que la misma clave puede ser utilizada para desencriptar todos los ficheros que el malware ha encriptado.

Esto facilita el manejo de las claves para los actores de BlackByte, a costa de un esquema de encriptación más débil y una recuperación del sistema más fácil.

Como la mayoría de los Ransomware, tiene capacidad de gusano para infectar más puntos de la misma red.

6.5. Carbanak/FIN7 JavaScript Backdoor

Descubierto en 2014, son uno de los actores de amenazas con motivación financiera más prolíficos y exitosos en acción de la actualidad, responsables de pérdidas estimadas en mil millones de dólares para innumerables instituciones financieras en todo el mundo.

El principal medio para difundir su malware consiste en correos electrónicos de Phishing dirigidos y eficaces.

Sin embargo, se ha descubierto recientemente una backdoor en JavaScript asociada con el actor, parece indicar que su malware que estaba basado principalmente en PowerShell, ha sido migrado a JavaScript en un intento de volverse menos detectable para los proveedores de seguridad.

Una vez ejecutada, la backdoor iniciará un delay de dos minutos en un esfuerzo por evitar la detección automatizada de la Sandbox y luego recabará la información de la máquina infectada: su IP, su MAC, el hostname de su DNS y lo reportará al servidor de Command-and-Control y ejecutará cualquier código que reciba como respuesta.

Emplea Cobalt Strike[?] como malware de seguimiento posterior a una infracción.⁶

³Conseguir cuentas de correo

⁴Interfaz de examen antimalware

⁵Copias de Seguridad

⁶Herramienta de Red Team para probar los sistemas y sus mecanismos de protección.



7. Pasos para Desofuscar un Código Malware en JavaScript

7.1. JavaScript Beautifuly

En ocasiones nos encontramos con código en JavaScript que está escrito en una sola línea, como ocurre con la muestra `2b0c9059fece8475c71fbde6cf4963132c274cf7ddebafbf2b0a59523c532e.js` del malware WJWorm. Para ello, tenemos que utilizar la herramienta JavSCript Beautify de CyberChef

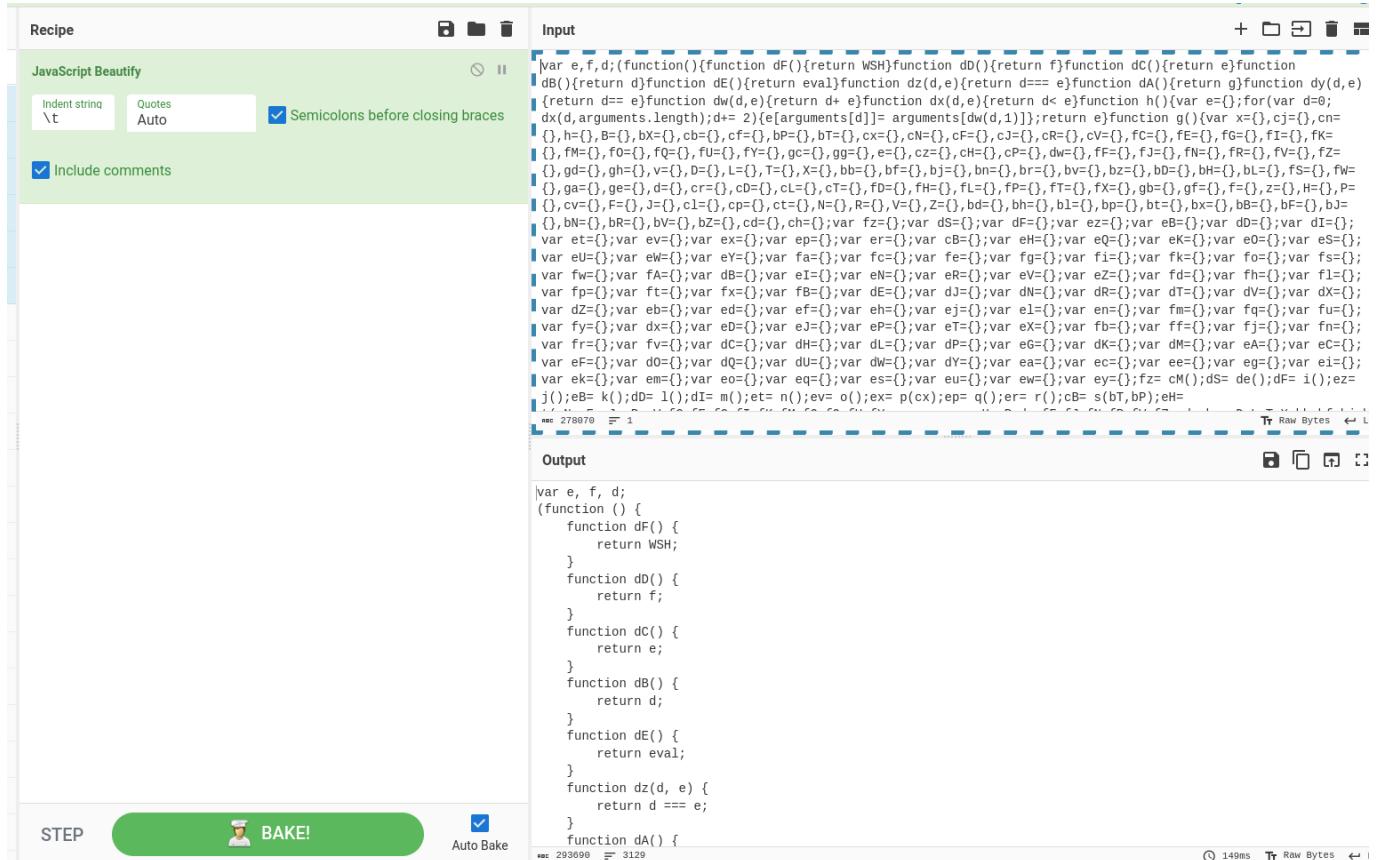


Figura 14: CyberChef JavaScript Beautifuly

7.2. IIFE: Immediately Infoked Function Expression

Muestra <https://github.com/bl4de>

Es una forma de ejecutar una función en JavaScript sin llamarla directamente.

Considerando el siguiente ejemplo, no se va a ejecutar nada:

```

1 function hello(message) {
2   console.log(message)
3 }

```

Código 11: Ejemplo de IFFE

Sin embargo, si ejecutamos este otro, sí se nos ejecutará la función:

```

1 function hello(message) {
2   console.log(message)
3
4
5   hello('This is test')

```

Código 12: Ejemplo de IFFE

Ahora vamos a hacer IFFE de este código. También se ejecutará la función.

```

1   (function hello(message) {
2     console.log(message)
3   })('This is test')
```

Código 13: Ejemplo de IFFE

Esto es posible por dos motivos:

- Los paréntesis alrededor de una expresión de una función hace que la expresión sea válida y se ejecute.
- En el paréntesis ('This is test') ejecuta la función pasando ese texto como argumento. Técnicamente (fn(x) {}) (x) es equivalente a (fn(x)) pero con su ejecución incluida.

Entonces, si un malware tiene el siguiente código:

```

1   (function(quhuvu6) {
2
3     // ...
4
5   }("41553a304f0b442551284206" + "672651014d1e1a60127" + ...)
```

Código 14: Ejemplo de IFFE

Significa que se ejecuta la función automáticamente con ese largo ASCII como argumento.

7.3. Expresiones, operador coma, parseInt() y toString()

Muestra <https://github.com/bl4de>

Vamos a ver la siguiente función cicuza ():

```

1   function cicuza(syhri) {
2     var fahomyfo = [];
3     for (var segovmiw4 = parseInt((0).toString(36)) /*CN1b367Z19XZqi8XgI67*/ ; segovmiw4 <
4       syhri["1"] + ("F", "T", "H", "e") + "n" + ("G", "n", "O", "g") + (29).toString(36) + (
5         "u", "X", "U", "p", "h"); segovmiw4 += parseInt((2).toString(36))) {
6       fahomyfo[("E", "w", "f", "F", "p") + ("G", "i", "L", "u") + "s" + "h"] = parseInt(syhri["s"
7         " + "u" + "b" + "s" + ("M", "h", "M", "U", "f", "t") + ("M", "q", "r")](segovmiw4,
8         85, 19, 84, 9, 2)), parseInt((42).toString(0x24)) /*uShFAoMcgqPvcds6w2xD*/ );
9     }
10    return fahomyfo;
11  }
```

Código 15: Ejemplo de operadores

Vamos a renombrar fahomyfo a un nombre con sentido para verlo mejor. En la función, se declara esta variable como array y luego tras un for y cierta lógica, se devuelve un resultado, así que llamaremos a la variable result:

```

1   function cicuza(syhri) {
2     var result = [];
3     for (var segovmiw4 = parseInt((0).toString(36)) /*CN1b367Z19XZqi8XgI67*/ ; segovmiw4 <
4       syhri["1"] + ("F", "T", "H", "e") + "n" + ("G", "n", "O", "g") + (29).toString(36) + (
5         "u", "X", "U", "p", "h"); segovmiw4 += parseInt((2).toString(36))) {
6       result[("E", "w", "f", "F", "p") + ("G", "i", "L", "u") + "s" + "h"] = parseInt(syhri["s"
7         " + "u" + "b" + "s" + ("M", "h", "M", "U", "f", "t") + ("M", "q", "r")](segovmiw4,
8         85, 19, 84, 9, 2)), parseInt((42).toString(0x24)) /*uShFAoMcgqPvcds6w2xD*/ );
9     }
10    return result;
11  }
```

Código 16: Ejemplo de operadores

Ahora vamos a ver el bucle for. Primero definimos el valor inicial de la variable con la que empieza:

```

1   for (var segovmiw4 = parseInt((0).toString(36)) /*CN1b367Z19XZqi8XgI67*/ ; ....
```

Código 17: Ejemplo de operadores



Con lo cual `parseInt((0).toString(36))` tiene dos operaciones concatenadas:

- `(0)` vale 0
- `.toString(36)` es un método que devuelve una representación de un objeto JavaScript. Cada objeto de JavaScript tiene este método, en este caso el número 0 será el 0 en representación de String.
- Cuando se llama a la función `.toString` con un número como parámetro, que puede ser un número entero entre 2 y 36 especificando la base en la que se representará el valor numérico. Así que en este caso será el número 0 en base 36. Resultando el número 0

En la segunda llamada `parseInt(0)` devuelve 0 también. Este método parsea cualquier String a su valor Entero y si la conversión no es posible, devuelve un `Nan`, Not a Number. Entonces el resultado será **0**.

Vamos a renombrar la variable `segovmiw4` como `i` y poner el 0 como inicial:

```

1  function cicuza(syhri) {
2      var result = [];
3      for (var i = 0; i < syhri["l"] + ("F", "T", "H", "e") + "n" + ("G", "n", "O", "g") + (29).
4          toString(36) + ("u", "X", "U", "p", "h")); i += parseInt((2).toString(36))) {
5          result[("E", "w", "f", "F", "p") + ("G", "i", "L", "u") + "s" + "h"](parseInt(syhri["s"
6              + "u" + "b" + "s" + ("M", "h", "M", "U", "t") + ("M", "q", "r"))(i, (85, 19,
7              84, 9, 2)), parseInt((42).toString(0x24)) /*uShFAoMcgqPvcds6w2xD*/));
7      }
8      return result;
9  };

```

Código 18: Ejemplo de operadores

Ahora nos vamos a enfocar en el siguiente código:

```

1  i < syhri["l"] + ("F", "T", "H", "e") + "n" + ("G", "n", "O", "g") + (29).toString(36) + ("u"
2      , "X", "U", "p", "h"));

```

Código 19: Ejemplo de operadores

Como sabemos, se puede leer un String como si fuese un array. Pero lo curioso es que en este caso se están utilizando paréntesis: `("F", "T", "H", "e")`. Si probamos a ejecutar `("F")` y `("F", "T", "H", "e")` vemos que las comas, lo que hace es separar los valores y sólo se utiliza el último valor:

```

>> a = ("f")
< "f"
>> b = ("F", "T", "H", "e")
< "e"
>> |

```

Figura 15: Conjunto de Caracteres String

Podemos leer los primeros caracteres dentro de `i < syhri[...]` que son `l e n g`, el quinto es una construcción como la anterior pero en este caso `(29).toString(36)` que es el la 29 cifra Hexadecimal que es la t. La última expresión devuelve la h y finalmente con la concatenación del operador + podemos ver que el resultado es `syhri["length"]`.

Si renombramos esa variable como `val` y desofuscamos la última parte, podemos ver que `parseInt((2).toString(36))` es el número 2 en formato String.

Entonces, hasta ahora tenemos lo siguiente:

```

1  function cicuza(val) {
2      var result = [];
3      for (var i = 0; i < val["length"]; i += 2) {

```



```

4     result[("E", "w", "f", "F", "p") + ("G", "i", "L", "u") + "s" + "h"] parseInt(val["s" +
5         "u" + "b" + "s" + ("M", "h", "M", "U", "f", "t") + ("M", "q", "r"))(i, (85, 19, 84,
6             9, 2)), parseInt((42).toString(0x24)) /*uShFAoMcgqPvcds6w2xD*/ );
7 }

return result;
};

```

Código 20: Ejemplo de operadores

Utilizando los métodos anteriores, podemos observar que el array quedará como `result ["push"]`.

JavaScript puede llamar cada Objeto utilizando el `..`. Por ejemplo si tenemos un array llamado `arr`, podemos llamar a su método `push` con el punto:

```

1 let arr = [] // declare Array object named arr
2 arr.push(10) // adds 10 as a first element of Array arr
3 arr.push(20) // adds 20 as a second element
4 console.log(arr) // prints [10, 20]

```

Código 21: Ejemplo de operadores

Si observamos la siguiente línea:

```

1 parseInt(val["s" + "u" + "b" + "s" + ("M", "h", "M", "U", "f", "t") + ("M", "q", "r"))(i,
2     (85, 19, 84, 9, 2)), parseInt((42).toString(0x24)) /*uShFAoMcgqPvcds6w2xD*/ )

```

Código 22: Ejemplo de operadores

Devolverá la función `substr` que tiene dos argumentos, el primero es el índice del primer char de la parte a devolver y el segundo (opcional) es el tamaño, sino, se parte hasta el final del String.

Así que tendremos lo siguiente hasta ahora:

```

1 parseInt(val["substr"])(i, (85, 19, 84, 9, 2)), parseInt((42).toString(0x24)) /*
2     uShFAoMcgqPvcds6w2xD*/

```

Código 23: Ejemplo de operadores

```

1 val["substr"](i, 2)

```

Código 24: Ejemplo de operadore

La última parte, `parseInt ((42).toString(0x24))` que como sabemos parsea 42 a un entero utilizando la base 0x24, que al ser un valor Hexadecimal, su valor decimal será 36. Convirtiendo a String este valor, tendremos el valor 16.

```

1 function cicuza(val) {
2     var result = [];
3     for (var i = 0; i < val.length; i += 2) {
4         result.push(parseInt(val.substr(i, 2), 16));
5     }
6     return result;
7 }

```

Código 25: Ejemplo de operadores desofuscado

Lo que hace esta función es que el argumento `val` tiene como valor inicial un String extremadamente largo, la función itera sobre el mismo, sustrae cada 2 caracteres, los convierte de Hexadecimal a Decimal y finalmente lo devuelve como un array.

Si cogemos la primera parte del String "41553a304f0b442551284206", 41 pasa a ser 65, 55, 85, 3a se convierte en 58, 30 en 48, 4f en 79 y así en adelante. Al final acabará siendo un array que contiene valores decimales, que será la variable `defiq` del malware

```

1 (...)

2 var defiq = cicuza(quhuvu6); // here we are so far :)
3 var permy = "H@D"7a84O";
4 var paghimqycgi = {
5     getpy: "myqniroqa3"
6 };

```



```

7
8   (...)
```

Código 26: Ejemplo de operadores final

7.4. Consiguiendo el constructor de la función

Un método muy utilizado en la ofuscación de malware en JavaScript , es ocultar las definiciones de funciones y sus llamadas. Por ejemplo:

```

1   function doubleX(x) {
2     return x * 2
3   }
4
5   doubleX(10) // returns 20
```

Código 27: Ejemplo ofuscación con constructor

Vamos a ofuscar este código:

```

1 let xcf = new Function("x","return x * 2")
2
3 xcf(10) // returns 20 as well
```

Código 28: Ejemplo ofuscación con constructor

`Function` es un método en JavaScript que construye una nueva función, como argumentos acepta una lista que es lo que tiene que devolver y el último argumento, el cuerpo de la función.

Sin embargo, el método `a.forEach` tiene su propia propiedad llamada `constructor` que es `Function()` ella misma.

Es decir, sustituyendo el código anterior:

```

1 let xcf = a.forEach.constructor("x","return x * 2")
2
3 xcf(10) // yep, it works! 20
```

Código 29: Ejemplo ofuscación con constructor

Hay un método muy similar en el código de nuestro malware:

```

1 let sdfgfdg = "".substr
2 sdfgfdg.call("malware",1,2) // "al"
```

Código 30: Ejemplo ofuscación con constructor

Este código equivale a:

```

1 "malware".substr(1,2) // "al"
```

Código 31: Ejemplo ofuscación con constructor

Esto pasa con los métodos `call()`, `apply()` y `bind()` también.

Vamos a aplicar esto al código de nuestro malware:

```

1 var xewubdiwhit = "kydka"[(12).toString(36) + (24).toString(36) + ("r", "w", "h", "Z", "n")
2   + ("n", "X", "L", "s", "w", "s") + "t" + (27).toString(36) + "u" + "c" + ("d", "m", "b",
3   "t") + ("z", "E", "z", "n", "o") + ("N", "J", "r")];
```

Código 32: Ejemplo ofuscación con constructor

Entonces esto acabará siendo:

```

1 var xewubdiwhit = "kydka"[ "constructor" ]
```

Código 33: Ejemplo ofuscación con constructor

Que es lo mismo que:



```
1 var fnConstructor = String.constructor
```

Código 34: Ejemplo ofuscación con constructor

Así que el código de nuestro malware quedará de la siguiente manera:

```
1 (function(quhuvu6) {
2     var defiq = cicuza(quhuvu6);
3     var permy = "H@D~7a84O";
4     var paghimqycgi = {
5         getpy: "myqniroqa3"
6     };
7     var fnConstructor = String.constructor;
8     var tyttaluli = "mokzine";
9
10    var dikol = [];
11    var mirjokbynet = 1; // (27, 50, 52, 21, 1) equals 1
12
13    (...)
```

Código 35: Ejemplo ofuscación con constructor

7.5. Operadores Lógicos

Si analizamos el siguiente fragmento:

```
1 while (mirjokbynet <= permy[("h", "g", "B", "W", "I") + "e" + (23).toString(0x24) + "g" + ("u",
2     "Z", "W", "u", "t") + (17).toString(36)]) {
3     dikol = (permy[("M", "H", "s") + ("C", "N", "D", "u") + (11).toString(0x24) + (28).toString
4         (0x24) + ("T", "k", "t") + "r"])(permy[(21).toString(0x24) + "e" + "n" + (16).toString
5         (36) + ("L", "S", "x", "t") + ("I", "D", "h") /*Q5E278CBpoixvOtUNpix*/] - mirjokbynet))
6     [("d", "R", "p", "s") + ("H", "K", "A", "s", "D", "p") + "I" + "i" + ("Y", "b", "h", "t")
7     ) /*O7MOfVrRkP9RlXlfKLxi*/](','');
8     for (var juqno = +!!false; juqno < defiq[("R", "t", "E", "I") + ("y", "e", "f", "R", "e") +
9         (23).toString(36) + (16).toString(36) + "t" + "h" /*H3RMPYZEeu55OVeGgb1v*/]; juqno++) {
10        defiq[juqno] = defiq[juqno] ^ dikol[juqno % dikol["I" + (14).toString(36) + "n" + (16).
11            toString(0x24) + (29).toString(0x24) + ("X", "O", "c", "m", "h")]]["c" + ("G", "w", "R
12            ", "e", "h") + (10).toString(36) + ("A", "W", "V", "i", "r") + "C" + ("Z", "O", "W", "o")
13            + ("R", "N", "A", "y", "d") + "e" + "A" + "t" /*YZz3OuivKuwgqjkFVKu0*/]((88, 53,
14            3, 90, 0));
15    }
16    mirjokbynet++;
17}
```

Código 36: Ejemplo de Operadores Lógicos

Además de lo que hemos visto anteriormente, hay unos usos extraños a la hora de utilizar operadores lógicos.

Primero, vamos a simplificar el código y desofuscarlo con las técnicas que ya conocemos, además de renombrar `mirjokbynet` a `k` y `juqno` a `j` también:

```
1 while (k <= permy.length) {
2     dikol = (permy.substr(permy.length - k)).split(',');
3     for (var j = +!!false; j < defiq.length; j++) {
4         defiq[j] = defiq[j] ^ dikol[j % dikol.length].charCodeAt(0);
5     }
6     k++;
7 }
```

Código 37: Ejemplo de Operadores Lógicos

Si observamos la variable de la condición del `while` llamada `permy` se utiliza y su definición asigna "`H@D~7a84O`" al principio del script `var_permy_= "H@D~7a84O";`

Ahora analizaremos el fragmento:



```
| while (k <= 9) { // 9 is value of permy.length
```

Código 38: Ejemplo de Operadores Lógicos

La siguiente línea, define el valor de `dikol`:

```
| dikol = (permy.substr(permy.length - k)).split('');
```

Código 39: Ejemplo de Operadores Lógicos

Como en este caso `k` es 0, `dikol` va a ser el último elemento del string `permy` que se parte en un array con la orden `split('')`, así que su valor en la primera iteración del while es `[0]`.

Ahora vamos a la parte del for:

```
| for (var j = +!!false; j < defiq.length; j++) {
  defiq[j] = defiq[j] ^ dikol[j % dikol.length].charCodeAt(0);
}
```

Código 40: Ejemplo de Operadores Lógicos

¿Qué evalua `+!!false`?

- inicialmente es `false`
- luego, el primer `!` que es un NOT lógico, lo cambia a `true`
- más tarde, el siguiente `!` lo vuelve a cambiar a `false`
- finalmente, el `+` hace un casting de Booleano a Entero, de `false` al valor 0

```
① Referrer Policy: Ignorar
» false
← false
» !false
← true
» !!false
← false
» +!!false
← 0
» |
```

Figura 16: Operadores lógicos

También tenemos otros operadores lógicos dentro del bucle for. Si nos fijamos `defiq` es un array con valores decimales (basándonos en las técnicas descritas anteriormente). Su primer elemento es el 65. El resultado de la expresión `efiq[j] ^ dikol[j % dikol.length].charCodeAt(0)` se calcula de la siguiente manera:

- `j` es el valor de control del bucle. Comienza con el valor 0 así que el resultado `j % dikol.length` es 0 ($0 \% 1 = 0$). Recordemos que en esta iteración, `dikol.length` vale 1.
- ahora, tenemos la expresión `efiq[j] ^ dikol[0].charCodeAt(0)` que equivale a `65 ^ 79`. El primer elemento del array es 65 como resultado de `dikol[0].charCodeAt(0)` `dikol[0]` que es la O mayúscula. 'O'. `charCodeAt(0)` es el número 79.



- el operador \wedge es el operador XOR. El resultado de $65 \wedge 79$ es 14.

A continuación el bucle va a ir incrementando el valor de la variable k, comenzando la operación otra vez y el array con diferentes valores.

7.6. Ofuscación de caracteres y Strings

7.6.1. Conjunto de enteros evaluados con eval

Nos podemos encontrar como en esta muestra, sacada de un pdf con PDFStreamDumper, un código JavaScript dentro de un documento PDF. Si nos fijamos, la función var_n_="" ; _eval(eval('Stri'+n_+'ng.fr'+_+n_+'omCharC con un listado de números. Si nos vamos a CyberChef, reemplazamos las comas por espacios y ejecutamos *From Char-code* con la base en 10, vemos que se transforma en un código en Javascript. Este ejemplo está sacado de la Sample2.zip, la password es *infected*:

Figura 17: Eval + conjunto de enteros

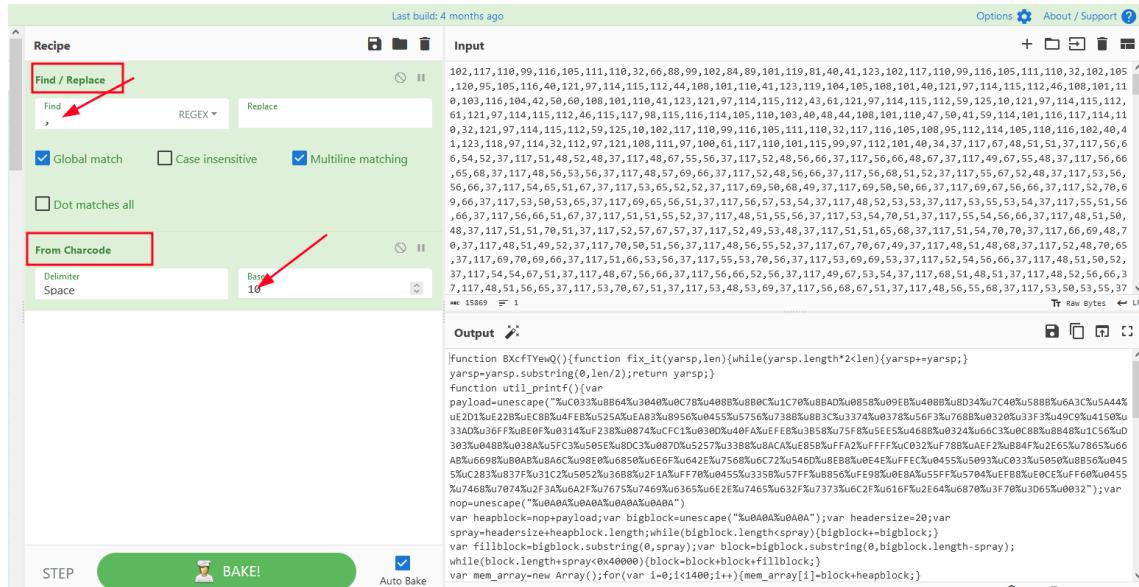


Figura 18: Eval + conjunto de enteros

7.6.2. Caracteres en Unicode

Podemos encontrarnos los caracteres en formato unicode, pero con la herramienta *JavaScript Beautify* ya lo traduce automáticamente a la vez que formatea el código JavaScript:

Figura 19: Caracteres unicode

7.6.3. Caracteres en Hexadecimal

En el siguiente snippet del malware WSHRat [39], podemos observar que tiene muchas partes que están ofuscadas en valores hexadecimales:

Figura 20: Caracteres hexadecimales

7.6.4. Caracteres de URL

El siguiente snippet de código de la versión de VB de STRRAT, como podemos observar, es una larga cadena de caracteres codificados en formato URL:

TFM: Estudio práctico de técnicas de ofuscación y contramedidas aplicables

Figura 21: Caracteres hexadecimales

7.7. Base64

El siguiente snippet de código forma parte del payload de STRRAT[39], como podemos observar, es un largo Base64, que a continuación decodificará.

```
var lmao$$$_=WSH.CreateObject("microsoft.xmldom").createElement("mko")
lmao$$$_.dataType="bin.base64"
lmao$$$_.text="dmFyIGxvbmdUZXh0MTsNCnZhcBsb25nVGV4dCA9ICJVRXNEQkJRJi1DJi1nSSYtTVFSR0Z
lUxUVUVVpkamoxUhD6Ji1VUlhkTC9n0XZMSU50WEzWUmVXc2p0a0lraRUdK0XRs0FZceWNpdG1VWY0L0R3b2Y
GpUQkt3ZWfjMCYtYUN0cVklSm13MwZNTlpoMzRTYmNDY05WaE1qemlhS0ZkWExWFuckgwR29JM3QzbkpoVWj
GZmdzllTguaeGQ4K2hEJi1FS0J6NSYtQ3ZKWTUxdmNVUUZqRDFyajQ0empqNyYtbEJMqdqVkp0L0kxdyYtJi0
i1CMCYtJi0mLUJqWhKTvlXMWliefTl5Wlh0dmRYSmpaWE12WTI5dVptbG5MbLI0ZCYtWeIzUnBDTSYtJi0mLTA
1IwdHNha0taSSs5WEpSb0UyVGdsRzkvQk9yc2ZmRnZHJi1hK3J3bXB0Ji0xJi0vZVVVTZNSctlbGJuBdRSEp
no2VmpKZEpHaDFxYlw2U2RreFuNy94Ji11aFJaUHBUSXp4ZG9iY2RUN3gztzUyZjFCTEJ3aXVqRmZIdCYtJi0
```

Figura 22: Caracteres en base64

8. Freyja Deobfuscation Tool

@drkrysSrng/freyja

Se ha desarrollado una herramienta, basándonos en los análisis que hemos mencionado anteriormente para facilitar la desofuscación de malware en JavaScript y su análisis. Tiene las siguientes funcionalidades:

- Limpia el código y lo tabula de manera similar, no sólo tabulando las instrucciones, sino que parsea los caracteres en hexadecimal y unicode.
- Chequeo de la entropía con el algoritmo de Shannon, línea por línea o por texto completo.
- Cuando los caracteres Unicode o en Hexadecimal no están parseados ya que no están en formato String sino que es una variable ofuscada, se parsean a mayores también.
- Desofusca funciones como:
 - `toString`, parsea los números a la base indicada y luego los pasa a string
 - conjuntos de números dentro de `eval` a string
 - evalúa y sustituye la función `unescape`
 - evalúa la función `parseInt`
- Concatena cadenas de caracteres separadas con el símbolo +
- Búsqueda de strings en base64 y su decodificación.

8.1. Funcionamiento

Cuando ejecutamos al inicio la herramienta tenemos las siguientes opciones:

```
venv> drakary$ python -m freyja_deobfuscator.freyja -h
Freyja Desobfuscating Tool

usage: [-h] -f FILEIN [-o FILEOUT] [-l LEVEL] [-b] [-e {LINE,FILE,line,file}]

options:
-h, --help            show this help message and exit
-f FILEIN, --filein FILEIN
                      Input file name
-o FILEOUT, --fileout FILEOUT
                      Output file name
-l LEVEL, --level LEVEL
                      Level 0: All options.Level 1: Just Beautify the File.Level 2: Parse Hex numbers to String.Level 3: Parse Unicode characters.Level 4: Deobfuscate toString with numbers.Level 5: Deobfuscate toString with Hex numbers.Level 6: Deobfuscate Eval with a list of numbers.Level 7: Deobfuscate unescape function inside chars.Level 8: Deobfuscate char sets.Level 9: Deobfuscate parseInt function.Level 10: Append Chars.
-b                  Extract Base64 strings
-e {LINE,FILE,line,file}
                      Shannon Entropy. Specify either "line" or "file"
```

Figura 23: Opciones de Freyja

8.2. Entropía

Si queremos analizar la entropía de Shannon de un fichero, tenemos dos opciones: línea por línea o del fichero completo. La diferencia es que si lo hacemos línea por línea, solamente nos mostrará las líneas sospechosas ya que en teoría si la entropía nos da mayor de 3.75, se supone que ese texto no está escrito por un humano.

8.2.1. Línea por Línea

El comando sería:

```
python freyja.py -f filein.js -e line
```

Código 41: Análisis de Entropía por líneas



```

    '/@ @@ /      *@@(   (@      @#      @@%      '#@@%
*@. #@|      *@ (@&   (@#@#  %@#@#      @@      /@ .@@.
*@@@|      *@ (@@   (@  @@@  @#      ,@@  @@  /@(@@.
/@          /@@*#@      @&  @@,  &@@  /@ .
(@          (@ ,@@%  @@      @@  @@*      #@
%@          %@      ,  @@      .@@      @@.      &@
```

x♥x↙x☒Ψ Freyja Desobfuscating Tool x♥x↙x☒Ψ

Input file: ../samples/javascript-malware-collection/2016/20160311/20160311_01284d18e603522cc8bdabed57583bb3.js
Output file: out.js
We all checking entropy line
Entropy 3.9161269465882835, code: timeStamp = "ToFil";
Entropy 3.970573095811684, code: var width = "sd/23r";
Entropy 3.7849418274376423, code: runescape = "Crea";
Entropy 3.9321380397593764, code: parseHTML = "pon";
Entropy 4.438721875540868, code: orig = (function Object.prototype.chainable() {
Entropy 3.932138039759377, code: rmouseEvent = 166;
Entropy 4.578147767168037, code: boxSizingReliableVal = "%/", marginLeft = "ate", click = "HTTP";
Entropy 3.77357262275185, code: matchers = 40;
Entropy 4.252953173936921, code: rtrim = "pos", disconnectedMatch = "n", pageX = "ite", setup = "t", rsingleTag = 7;
Entropy 4.451601986212192, code: elemData = "/nro", clearTimeout = "DB.", cssNumber = "am", fadeIn = 3, overwritten = "close";
Entropy 3.892407118592878, code: var focusin = 101;
Entropy 3.8841837197791884, code: tokenCache = "ep";
Entropy 3.7841837197791883, code: timeout = "Creat";
Entropy 3.921029621737614, code: var rteopenamespace = "G".

Figura 24: Entropía por líneas

8.2.2. Fichero completo

El comando sería:

```
python freyja.py -f filein.js -e file
```

Código 42: Análisis de Entropía del fichero completo

```

    '/@ @@ /      *@@(   (@      @#      @@%      '#@@%
*@. #@|      *@ (@&   (@#@#  %@#@#      @@      /@ .@@.
*@@@|      *@ (@@   (@  @@@  @#      ,@@  @@  /@(@@.
/@          /@@*#@      @&  @@,  &@@  /@ .
(@          (@ ,@@%  @@      @@  @@*      #@
%@          %@      ,  @@      .@@      @@.      &@
```

x♥x↙x☒Ψ Freyja Desobfuscating Tool x♥x↙x☒Ψ

Input file: ../samples/javascript-malware-collection/2016/20160311/20160311_01284d18e603522cc8bdabed57583bb3.js
Output file: out.js
We all checking entropy file
File entropy is 5.349805318669943, if higher than 3.75 is not human written

Figura 25: Entropía del fichero completo

8.2.3. Código fuente

Similar al que se ha explicado en la sección 5, pero adaptado a nuestra tool:

```

1      """
2          Calculates the Shannon entropy of a UTF-8 encoded string
3      """
4
5      def entropy_check(string):
6          # decode the string as UTF-8
7          unicode_string = string.decode('utf-8')
8
9          # get probability of chars in string
```



```

10     prob = [float(unicode_string.count(c)) / len(unicode_string) for c in dict.fromkeys(list
11         (unicode_string))]
12
13     # calculate the entropy
14     entropy = - sum([p * math.log(p) / math.log(2.0) for p in prob])
15
16     return entropy
17
18 """
19     Gets the string we are going to analyse, line per line or the full string from the file
20 """
21 def calculate_entropy(filename, option):
22     option = option.upper()
23
24     if option == "FILE":
25         with open(filename, 'rb') as f:
26             content = f.read()
27             print(f"File entropy is {entropy_check(content)}, if higher than 3.75 is not
28                 human written")
29
30     elif option == "LINE":
31         with open(filename, 'rb') as f:
32             content = f.readlines()
33
34         for line in content:
35             entropy = entropy_check(line)
36             if entropy > 3.75:
37                 line_str = line[:-1]
38                 line_str = line_str.decode('utf-8')
39
40                 print(f"Entropy {entropy}, code: {line_str}")

```

Código 43: Análisis de Entropía del fichero completo

8.3. Desofuscación

Se han desarrollado varias de las técnicas del apartado 7 en esta herramienta. Para que nos analice la herramienta el comando es el siguiente:

```
python freyja.py -f ./samples/simple_js_malware_code/do_not_run.js -o output.js -l 2
```

Código 44: Desofuscación de ficheros JavaScript

- -f Fichero a desofuscar.
- -o Fichero de salida desofuscado.
- -l Nivel de ofuscación. Le indicamos la técnica que queremos que utilice.
 - Nivel 0: Todas las opciones
 - Nivel 1: Técnica *beautify*, tabular el fichero y darle formato al JavaScript.
 - Nivel 2: Parsear los caracteres Hexadecimal a Strings.
 - Nivel 3: Parsear los caracteres Unicode a Strings.
 - Nivel 4: Desofuscar la función `toString`.
 - Nivel 5: Desofuscar la función `toString` con número hexadecimal.
 - Nivel 6: Desofuscar la función `eval` cuando tiene una lista de números.
 - Nivel 7: Desofuscar la función `unescape` cuando contiene caracteres.
 - Nivel 8: Desofuscar conjuntos de caracteres.
 - Nivel 9: Desofuscar la función `parseInt`.
 - Nivel 10: Concatena caracteres aunque estén en varias líneas.



8.3.1. Ejemplo de Nivel 1: Beautify

Para ello, se ha utilizado la librería `jsbeautifier` de Python:

```
1     """
2     Cleans the javascript file beautifying it
3     """
4     def beautify_file(filename):
5         res = jsbeautifier.beautify_file(filename, options)
6
7     return res
```

Código 45: Desofuscación de ficheros JavaScript Nivel 1: Beautify

```
1 python freyja.py -f ./samples/Sample1/ejercicio6.js -l 1
```

Código 46: Desofuscación de ficheros JavaScript Nivel 1: Beautify



```
1 |pre_interactions=null;pre_including8=null;pre_with=null;pre_Similar=null;pre_customer=null;pre_with2=null;pre_training2=null;pre_provide5=r
2
```

Figura 26: Nivel 1: Beautify Ofuscado



```

out.js
1 pre_interactions = null;
2 pre_including8 = null;
3 pre_with = null;
4 pre_Similar = null;
5 pre_customer = null;
6 pre_with2 = null;
7 pre_training2 = null;
8 pre_provide5 = null;
9 pre_with = null;
10 pre_malicious1 = null;
11 pre_purposes = null;
12 pre_software5 = null;
13 varpre_computer = this[{
14     ou2: "A"
15 }.ou2 + {
16     rol: "c"
17 }.rol + {
18     e0: "t"
19 }.e0 + {
20     w2: "i"
21 }.w2 + {
22     uc2: "v"
23 }.uc2 + {
24     ls1: "e"
25 }.ls1 + {
26     nfo3: "X"
27 }.nfo3 + {
28     il: "0"
29 }.il + {
30     oft0: "b"
31 }.oft0 + {
32     c2: "j"
33 }.c2 + {
34     o1: "e"
35 }.o1 + {
36     en1: "c"
37 }.en1 + {
38     cce2: "t"
39 }.cce2];
40 varWSSS12 = this[{
41     o3: "W"
42 }.o3 + {
43     RL2: "S"
44 }.RL2 + {
45     ar1: "c"
46 }.ar1 + {

```

Figura 27: Nivel 1: Beautify Desofuscado

8.3.2. Ejemplo de Nivel 2: Parsear Hexadecimal

Para ello, se ha utilizado la librería `re` de Python, muy famosa para utilizar expresiones regulares. Se buscan los caracteres Hexadecimales que suelen ser 2 números que empiezan por `\x` y luego se parsean a String:

```

1 """
2     Parse hex numbers to char
3 """
4
5 def parse_hex(data):
6     def replace_hex(match):
7         try:
8             decoded = bytes.fromhex(match.group(1)).decode('utf-8')
9         except UnicodeDecodeError:
10             decoded = bytes.fromhex(match.group(1)).decode("iso-8859-1")
11         return decoded
12
13     pattern = r'\\x([0-9a-fA-F]{2})'
14
15     data = re.sub(pattern, replace_hex, data)
16
17     return data

```

Código 47: Desofuscación de ficheros JavaScript Nivel 2: Hexadecimal



```
1 python freyja.py -f ./samples/b122473d00566758d09c695d191b368e0c815c65e8acc0f00da7a88e45cc8a9e.js -1 2
```

Código 48: Desofuscación de ficheros JavaScript Nivel 2: Hexadecimal

```
eval(
  '\x76\x61\x72\x20\x6D\x4E\x79\x54\x20\x3D\x20\x57\x53\x7B\x30\x7D\x2E\x43\x72\x65\x61\x74\x65\x4F\x62\x6A\x65\x63\x7B\x31\x7D\x28\x22\x6
  'H',
  't',
  'mldo',
  'ateEl'
);

```

Figura 28: Nivel 2: Hexadecimal Ofuscado

```
eval(
  'var mNyT = WS{0}.CreateObject("microsoft.x{2}m").CreateObject("bsc");Array.prototype.sh1nEKon = eval'.h5aLFnU(
    'H',
    't',
    'mldo',
    'ateEl'
  );
  [function () {
    '.toString()'
  }
}
```

Figura 29: Nivel 2: Hexadecimal Desofuscado

8.3.3. Ejemplo de Nivel 3: Parsear Unicodes

Para ello, se ha utilizado la librería `re` de Python, muy famosa para utilizar expresiones regulares. Se buscan los caracteres unicode que suelen ser 4 caracteres comenzando por `\u` y luego se parsean a String:

```
1 """
2 Parse char unicode to char
3 """
4 def parse_unicode(data):
5 def replace_unicode(match):
6     return chr(int(match.group(1), 16))
7
8 pattern = r'\\u([0-9a-fA-F]{4})'
9
10 data = re.sub(pattern, replace_unicode, data)
11
12 return data
```

Código 49: Desofuscación de ficheros JavaScript Nivel 3: Unicode

```
1 python freyja.py -f ./samples/b122473d00566758d09c695d191b368e0c815c65e8acc0f00da7a88e45cc8a9e.js -1 3
```

Código 50: Desofuscación de ficheros JavaScript Nivel 3: Unicode

```
        kd3$_ = kd3$_ + hDrF04m["substr"](gr0UnD4[i], 1);
      }
      i = i+1;
    }while(i<gr0UnD4.length);
    tmmp[j] = '\u006B\u0064\u0033\u0024\u005F';
    kd3$_ = "";
}
Array.prototype.\u0052\u0045\u0053\u0055\u004c\u0054 = tmmp;
[function(){
  load([]);
}][0]();
};
```

Figura 30: Nivel 3: Unicode Ofuscado

```

        IT (kd3$_ == "") {
            kd3$_ = hDrF04m["substr"](gr0UnD4[i], 1);
        } else {
            kd3$_ = kd3$_ + hDrF04m["substr"](gr0UnD4[i],
        }
        i = i + 1;
    } while(i < gr0UnD4.length);
    tmmp[j] = kd3$_;
    kd3$_ = "";
}
Array.prototype.RESULT = tmmp;
[function () {
    load([]);
}][0]();
String.prototype.x7u = [61, 8, 54, 11, 6, 45, 42, 53, 42, 11,

```

Figura 31: Nivel 3: Unicode Desofuscado

8.3.4. Ejemplo de Nivel 4 y 5: Reemplazar la función `toString`

Para ello, se ha utilizado la librería `re` de Python, muy famosa para utilizar expresiones regulares. Se busca texto tipo (23).`toString(0x24)` o (14).`toString(36)`

```

1 """
2 """
3     Replaces toString function parsing its context to string integer equivalent
4     after parsing the integer to the new base
5 """
6 def replace_tostring(data):
7     # Regular expression pattern to match (number).toString( number )
8     pattern = r'\\(\s*(\d+)\s*)\\s*\\.toString\\(\s*(\d+)\\s*\\)'
9
10    # Find all matches of the pattern
11    matches = re.finditer(pattern, data, re.MULTILINE)
12
13    length = 0
14
15    for match in matches:
16        # Extract the two numbers from the matched groups
17        first_number = int(match.group(1))
18        second_number = int(match.group(2))
19
20        to_string = numpy.base.repr(first_number, second_number)
21
22        to_string = "\\\" + to_string + "\\"
23
24        data = data[:match.start() - length] + to_string + data[match.end() - length:]
25
26        length = length + len(match.group(0)) - 3
27
28    return data
29
30 """
31     Replaces toString function parsing its context to string integer equivalent
32     after parsing the hex to integer and then to the new base
33 """
34 def replace_hex_tostring(data):
35     # Regular expression pattern to match (number).toString( number )
36     pattern = r'\\(\s*(\d+)\s*)\\s*\\.toString\\(\s*(0x[\da-fA-F]+)\\s*\\)'
37
38     # Find all matches of the pattern
39     matches = re.finditer(pattern, data, re.MULTILINE)
40
41     length = 0
42
43     for match in matches:

```



```

45      # Extract the two numbers from the matched groups
46      first_number = int(match.group(1))
47      second_number = match.group(2)
48
49      second_number = int(second_number, 16)
50
51      to_string = numpy.base_repr(first_number, second_number)
52
53      to_string = "\\" + to_string + "\\"
54
55      data = data[:match.start() - length] + to_string + data[match.end() - length:]
56
57      length = length + len(match.group(0)) - 3
58
59  return data

```

Código 51: Desofuscación de ficheros JavaScript Nivel 4 y 5: toString

```
python freyja.py -f ./samples/simple_js_malware_code/do_not_run.js -1 4
```

Código 52: Desofuscación de ficheros JavaScript Nivel 4 y 5: toString

```

while (mirjokbynet <= permy(["h", "g", "B", "W", "l") + "e" + (23).toString(0x24) + "g" + ("u", "Z", "W", "u", "t") + (17).toString(36)
dikol = (permy(["M", "H", "s") + ("C", "N", "D", "u") + (11).toString(0x24) + (28).toString(0x24) + ("T", "k", "t") + "r"])(permy([
for (var juqno = +!false; juqno < defiq(["R", "t", "E", "l") + ("y", "e", "f", "R", "e") + (23).toString(36) + (16).toString(36)
defiq[juqno] = defiq[juqno] ^ dikol[juqno % dikol["l" + (14).toString(36) + "n" + (16).toString(0x24) + (29).toString(0x24) +
}:
mirjokbynet++;
};
```

Figura 32: Nivel 4 y 5: toString Ofuscado

```

var mirjokbynet = (27, 50, 52, 21, 1);
while (mirjokbynet <= permy(["h", "g", "B", "W", "l") + "e" + (23)
.toString(0x24) + "g" + ("u", "Z", "W", "u", "t") + "r"])(permy([
dikol = (permy(["M", "H", "s") + ("C", "N", "D", "u") + (11)
.toString(0x24) + (28)
.toString(0x24) + ("T", "k", "t") + "r"])(permy([
.toString(0x24) + "e" + "n" + "G" + ("L", "S", "x", "t") + ("I", "D", "h") /*QSE278CBpoixv0tUNpix*/ ] - mirjokbynet))[
("d", "R",
for (var juqno = +!false; juqno < defiq(["R", "t", "E", "l") + ("y", "e", "f", "R", "e") + "N" + "G" + "t" + "h" /*H3RMPYzEu550
defiq[juqno] = defiq[juqno] ^ dikol[juqno % dikol["l" + "E" + "n" + (16)
.toString(0x24) + (29)
.toString(0x24) + ("x", "O", "c", "m", "h")])["c" + ("G", "w", "R", "e", "h") + "A" + ("A", "W", "V", "i", "r") + "C" + ("Z",
}:
mirjokbynet++;
};
```

Figura 33: Nivel 4: toString Desofuscado

```

var mirjokbynet = (27, 50, 52, 21, 1);
while (mirjokbynet <= permy(["h", "g", "B", "W", "l") + "e" + "N" + "g" + ("u", "Z", "W", "u", "t") + (17)
.toString(36)) {
dikol = (permy(["M", "H", "s") + ("C", "N", "D", "u") + "B" + "S" + ("T", "k", "t") + "r"])(permy([
.toString(36) + ("L", "S", "x", "t") + ("I", "D", "h") /*QSE278CBpoixv0tUNpix*/ ] - mirjokbynet))[
("d", "R", "p", "s") + ("H", "t",
for (var juqno = +!false; juqno < defiq(["R", "t", "E", "l") + ("y", "e", "f", "R", "e") + (23)
.toString(36) + (16)
.toString(36) + "h" /*H3RMPYzEu550VeGb1v*/ ]; juqno++)
defiq[juqno] = defiq[juqno] ^ dikol["l" + (14)
.toString(36) + "n" + "G" + "T" + ("X", "O", "c", "m", "h")])["c" + ("G", "w", "R", "e", "h") + (10)
.toString(36) + ("A", "W", "V", "i", "r") + "C" + ("Z", "O", "W", "o") + ("R", "N", "A", "y", "d") + "e" + "A" + "t" /*YZz30
}:
mirjokbynet++;
};
```

Figura 34: Nivel 4: toString Hex Desofuscado

8.3.5. Ejemplo de Nivel 6: Parsear eval si tiene una lista de números

Para ello, se ha utilizado la librería `re` de Python, muy famosa para utilizar expresiones regulares. Se buscan listas de números como en la muestra extraída de un pdf `sample_2.js` que contiene una lista de números dentro de una expresión `eval`:



```
1      """
2          Replaces eval with a list of numbers
3      """
4      def parse_eval_list_numbers(data):
5          def replace_with_chars(match):
6              numbers = re.findall(r'\d+', match.group(1))
7              chars = ''.join(chr(int(num)) for num in numbers)
8              return "\\" + chars + "\\"
9
10     pattern = r'[^']*?\((?:\d+(?:\s*,\s*)?)+\)'
11     data = re.sub(pattern, replace_with_chars, data)
12
13     return data
```

Código 53: Desofuscación de ficheros JavaScript Nivel 6: Eval

```
python freyja.py -f ./samples/sample_2_extracted/sample_2.js -l 6
```

Código 54: Desofuscación de ficheros JavaScript Nivel 6: Eval

```
; eval(eval('String.fromCharCode'+ode(102,117,110,99,116,105,111,110,32,66,88,99,102,84,89,101,119,81,40,41,123,102
```

Figura 35: Nivel 6: Eval Ofuscado

Figura 36: Nivel 6: Eval Desofuscado

8.3.6. Ejemplo de Nivel 7: Parsear la función unescape

Para ello, se ha utilizado la librería `re` de Python, muy famosa para utilizar expresiones regulares. Se desofusca el texto que está en Unicode o en formato URL parseándolo a String:

```
1
2     """
3         Replaces unescape function parsing the content to string
4     """
5     def parse_unescape(data):
6         pattern = r'unescape\\([\\''']([^\\"']+)\\''')\\'
7
8         def replacement(match):
9             unicode_escaped = urllib.parse.unquote(match.group(1))
10            replaced_unicode_escaped = unicode_escaped.replace('%', '\\')
11            decoded_string = bytes(replaced_unicode_escaped, 'utf-8').decode('unicode_escape')
12            return "\'" + decoded_string + "\'"
13
14     data = re.sub(pattern, replacement, data)
15     return data
```

Código 55: Desofuscación de ficheros JavaScript Nivel 7: Eval

```
python freyja.py -f ./samples/javascript-malware-collection/2016/20160311/20160311_01284  
d18e603522cc8bdabed57583bb3.js -l 7
```

Código 56: Desofuscación de ficheros JavaScript Nivel 7: unescape

```

completed[ R .chainable() + backgroundClip + n ](speed.chainable(((matchers + 66) / (Math.pow(adbck, 2) - initiaInonit), (1 - needScor
function content() {
    eval('' + tuples["WScr" + delegateType]["Sle" + tokenCache](((Math.pow(453543, head)-205700727849)/(tween+0)));
}
function focus() {
    eval('' + curCSSTop[rtrim + "itio" + disconnectedMatch] = (7*scripts,(43-merge)));
}
function cur() {
    eval('' + curCSSTop[id]);
}
function accepts() {
    eval('' + options["wr" + pageX](host["Res" + parseHTML + "seBody"].chainable()));
}
function curCSS() {
    eval('' + options["wr" + pageX](host["Res" + parseHTML + "seBody"].chainable()));
}

```

Figura 37: Nivel 7: unescape Ofuscado

```

completed[ R .chainable() + backgroundClip + n ](speed.chainable(((matchers + 66) / (Math.pow(adbck, 2) - initiaInonit), (1 - needScor
function content () {
    eval('' + tuples["WScr" + delegateType]["Sle" + tokenCache](((Math.pow(453543, head)-205700727849)/(tween+0)));
}
function focus () {
    eval('' + curCSSTop[rtrim + "itio" + disconnectedMatch] = (7*scripts,(43-merge)));
}
function cur () {
    eval('' + curCSSTop[id]);
}
function accepts () {
    eval('' + options["wr" + pageX](host["Res" + parseHTML + "seBody"].chainable()));
}
function curCSS () {
    eval('' + options["wr" + pageX](host["Res" + parseHTML + "seBody"].chainable()));
}

```

Figura 38: Nivel 7: unescape Desofuscado

8.3.7. Ejemplo de Nivel 8: Parsear conjuntos de caracteres

Para ello, se ha utilizado la librería `re` de Python, muy famosa para utilizar expresiones regulares. Se buscan conjuntos de caracteres separados por +, ya que JavaScript sólo deja el último de la lista:

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
"""
If there is a char set, it replaces it with the last value ("A", "B", "C") is "C"
"""
def parse_char_set(data):
    pattern = r'\+\s*\((\s*[a-zA-Z]\s*,\s*)+\s*\)([a-zA-Z])\s*\)\s*\+'
    # Find all matches of the pattern
    matches = re.finditer(pattern, data, re.MULTILINE)
    length = 0
    for match in matches:
        to_string = "+ \'" + match.group(2) + "\' +"
        data = data[:match.start() - length] + to_string + data[match.end() - length:]
        length = length + len(match.group(0)) - 7
    pattern = r'\+\s*\((\s*[a-zA-Z]\s*,\s*)+\s*\)([a-zA-Z])\s*\)\s*'

```

```

22 # Find all matches of the pattern
23 matches = re.finditer(pattern, data, re.MULTILINE)
24
25 length = 0
26
27 for match in matches:
28     to_string = "+" + match.group(2) + "\"
29
30     data = data[:match.start() - length] + to_string + data[match.end() - length:]
31
32     length = length + len(match.group(0)) - 6
33
34 pattern = r'\s*\((\s*[a-zA-Z]\s*\s*)+\s*([a-zA-Z])\s*\)\s*\)+'
35
36 # Find all matches of the pattern
37 matches = re.finditer(pattern, data, re.MULTILINE)
38
39 length = 0
40
41 for match in matches:
42     to_string = "\\" + match.group(2) + "\"
43
44     data = data[:match.start() - length] + to_string + data[match.end() - length:]
45
46     length = length + len(match.group(0)) - 6
47
48 return data

```

Código 57: Desofuscación de ficheros JavaScript Nivel 8: Conjuntos de caracteres

```
python freyja.py -f freyja -f ./samples/simple_js_malware_code/do_not_run.js -l 8
```

Código 58: Desofuscación de ficheros JavaScript Nivel 8: Conjuntos de caracteres

```

var mirjokbynet = (27, 50, 52, 21, 1);
while (mirjokbynet <= permy["h", "g", "B", "W", "l"] + "e" + (23).toString(0x24) + "g" + ("u", "Z", "W", "U", "t") + [1]
dikol = (permy["H", "S"] + ("C", "N", "D", "O") + (11).toString(0x24) + (28).toString(0x24) + ("T", "K", "F")
for (var juqno = +!false; juqno < defiq[("R", "t", "E", "l") + ("y", "e", "f", "R", "e") + (23).toString(36) + (16)
defiq[juqno] = defiq[juqno] ^ dikol[juqno % dikol["l" + (14).toString(36) + "n" + (16).toString(0x24) + (29).to
mirjokbynet++;
}
for (var vaxofibcid = +!false; vaxofibcid < defiq[(21).toString(0x24) + (14).toString(0x24) + ("o", "w", "n") + "g" + (
defiq[vaxofibcid] = "ms"[c" + "o" + ("T", "L", "W", "T", "n") + ("p", "z", "K", "o", "P", "s") + (29).toString(0x24)

```

Figura 39: Nivel 8: Conjuntos de caracteres Ofuscados

```

(function (quhuuu6) {
    var defiq = cicuza(quhuuu6);
    var permy = "H@D~7a840";
    var paghimqycgi = {
        getpy: "myqniroqa3"
    };
    var xewubdiwhit = "kydka" [(12)
        .toString(36) + (24)
        .toString(36) + "n" + "s" + "t" + (27)
        .toString(36) + "u" + "c" + "t" + "o" + "r"];
    var tyttaluli = "mokzine";

    var dikol = [];
    var mirjokbynet = (27, 50, 52, 21, 1);

    while (mirjokbynet <= permy["l" + "e" + (23)
        .toString(0x24) + "g" + "t" + (17)
        .toString(36)]) {

```

Figura 40: Nivel 8: Conjuntos de caracteres Desofuscados



8.3.8. Ejemplo de Nivel 9: Parsear la función parseInt

Para ello, se ha utilizado la librería `re` de Python, muy famosa para utilizar expresiones regulares. Se desofusca pasando a entero el parámetro de la función. En este caso reutilizamos la salida del nivel 4 `out.js`:

```

1
2      """
3          Replaces parseInt function parsing the content char to a in integer value
4      """
5      def parse_int(data):
6          pattern = r'parseInt\("(\\d+)"\)'
7
8          # Find all matches of the pattern
9          matches = re.finditer(pattern, data, re.MULTILINE)
10
11         length = 0
12
13         for match in matches:
14             data = data[:match.start() - length] + match.group(1) + data[match.end() - length:]
15
16             length = length + len(match.group(0)) - 1
17
18     return data

```

Código 59: Desofuscación de ficheros JavaScript Nivel 9: parseInt

```

1 python freyja.py -f out.js -l 9

```

Código 60: Desofuscación de ficheros JavaScript Nivel 9: parseInt

```

function cicuza (syhri) {
    var fahomyfo = [];
    for (var segovmiw4 = parseInt("0" /*CN1b367Z19XZqi8XqI67*/; segovmiw4 < syhri["l" + ("F", "T", "H", "e") + "n" + ("G", "n", "0", "g") + "T" + ("u", "X", "U", "p", "h")]; segovmiw4 += parseInt("2")); {
        fahomyfo[("E", "w", "f", "F", "p") + ("G", "i", "L", "u") + "s" + "h"](parseInt(syhri["s" + "u" + "b" + "s" + ("M", "h", "U", "f", "t") + ("M", "q", "r")))(segovmiw4, (85, 19, 84, 9, 2)), parseInt((42) .toString(0x24)) /*uShFAoMcqqPvcds6w2xD*/ );
    }
    return fahomyfo;
}

```

Figura 41: Nivel 9: parseInt Ofuscado

```

function cicuza (syhri) {
    var fahomyfo = [];
    for (var segovmiw4 = 0 /*CN1b367Z19XZqi8XqI67*/; segovmiw4 < syhri["l" + ("F", "T", "H", "e") + "n" + ("G", "n", "0", "g") + "T" + ("u", "X", "U", "p", "h")]; segovmiw4 += 2) {
        fahomyfo[("E", "w", "f", "F", "p") + ("G", "i", "L", "u") + "s" + "h"](parseInt(syhri["s" + "u" + "b" + "s" + ("M", "h", "U", "f", "t") + ("M", "q", "r")))(segovmiw4, (85, 19, 84, 9, 2)), parseInt((42) .toString(0x24)) /*uShFAoMcqqPvcds6w2xD*/ );
    }
    return fahomyfo;
}

```

Figura 42: Nivel 9: parseInt Desofuscado

8.3.9. Ejemplo de Nivel 10: Parsear las concatenaciones de caracteres

Para ello, se ha utilizado la librería `re` de Python, muy famosa para utilizar expresiones regulares. Se concatenan todos los caracteres separados por `+`. Para el ejemplo reutilizaremos el `out.js`:

```

1
2      """
3          Appending characrers between + like "A" + "B"
4      """
5      def append_strings_multi(data):
6          # Regular expression pattern to match (number).toString( number )
7          pattern = r'[\\"\\]\s+\+\s+[\\"\\]'
8
9          # Find all matches of the pattern
10         matches = re.finditer(pattern, data, re.MULTILINE)
11
12     return data

```



```

13     length = 0
14
15     for match in matches:
16
17         to_string = ""
18
19         data = data[:match.start() - length] + to_string + data[match.end() - length:]
20
21         length = length + len(match.group(0))
22
23     return data

```

Código 61: Desofuscación de ficheros JavaScript Nivel 10: Concatenación

```
1 python freyja.py -f out.js -l 10
```

Código 62: Desofuscación de ficheros JavaScript Nivel 10: Concatenación

```

while (mirjokbynet <= permy["l" + "e" + (23)
    .toString(0x24) + "g" + "t" + (17)
    .toString(36)]) {
    dikol = (permy["s" + "u" + (11)
        .toString(0x24) + (28)
        .toString(0x24) + "t" + "r"](permy[(21)
            .toString(0x24) + "e" + "n" + (16)
            .toString(36) + "t" + "h" /*Q5E278CBpoixv0tUNpix*/ ] - mirjokby
    for (var juqno = +!false; juqno < defiq["l" + "e" + (23)
        .toString(36) + (16)
        .toString(36) + "t" + "h" /*H3RMPYZEeu550VeGgb1v*/ ]; juqno++
        defiq[juqno] = defiq[juqno] ^ dikol[juqno % dikol["l" + (14
            .toString(36) + "n" + (16)
            .toString(0x24) + (29)
            .toString(0x24) + "h"]]["c" + "h" + (10)
            .toString(36) + "r" + "C" + "o" + "d" + "e" + "A" + "t" /*Y
    }
    mirjokbynet++;

```

Figura 43: Nivel 10: Concatenación Ofuscada

```

var mirjokbynet = (27, 30, 32, 21, 17,
    while (mirjokbynet <= permy["le" + (23)
        .toString(0x24) + "gt" + (17)
        .toString(36))) {
        dikol = (permy["su" + (11)]
            .toString(0x24) + (28)
            .toString(0x24) + "tr"](permy[(21)
                .toString(0x24) + "en" + (16)
                .toString(36) + "th" /*Q5E278CBpoixv0tUNpix*/ ] - mirjokbynet))["split" /*07M0fVrRkP9RlxLxfKLxi*/ ]('
    for (var juqno = +!false; juqno < defiq["le" + (23)
        .toString(36) + (16)
        .toString(36) + "th" /*H3RMPYZEeu550VeGgb1v*/ ]; juqno++) {
        defiq[juqno] = defiq[juqno] ^ dikol[juqno % dikol["l" + (14
            .toString(36) + "n" + (16)
            .toString(0x24) + (29)
            .toString(0x24) + "h"]]["ch" + (10)
            .toString(36) + "rCodeAt" /*YZz3OuvKuwgqjkFVKu0*/ ]((88, 53, 3, 90, 0));
    }
    mirjokbynet++;
};

```

Figura 44: Nivel 10: Concatenación Desofuscada

8.4. Búsquedas de Base64

Esta herramienta también puede buscar y decodificar strings de Base64 en un fichero de texto. Para ello utiliza la librería `re` de Python para buscar con la expresión regular referente a strings codificados en Base64:

```
1
2
```



```

3     """
4         Finds base64 between '' and "" and then deletes first and last character
5     """
6     def find_base64(data):
7         base64_pattern = r'["\'](?:[A-Za-z0-9+/]{4})*(?:[A-Za-z0-9+/]{2}==|[A-Za-z0-9+/]{3}==|[A-Za-
8             -z0-9+/]{4})["\']'
9         base64_strings = re.findall(base64_pattern, data)
10
11    base64_strings_cleaned = []
12    for match in base64_strings:
13        match = match[1:-1]
14        if len(match) >= 10:
15            base64_strings_cleaned.append(match)
16    return base64_strings_cleaned
17
18    """
19        Decodes base64 and saves it to json output file
20    """
21    def decode_base64(encoded_data):
22        decoded_strings = []
23        for base64_string in encoded_data:
24            decoded_bytes = base64.b64decode(base64_string)
25            try:
26                decoded_string = decoded_bytes.decode("utf-8")
27            except UnicodeDecodeError:
28                decoded_string = decoded_bytes.decode("iso-8859-1")
29
30        decoded_dict = {'original': base64_string, 'decoded': decoded_string}
31        decoded_strings.append(decoded_dict)
32    return decoded_strings

```

Código 63: Búsqueda de Base64

```
1 python freyja.py -b example.txt -o example.json
```

Código 64: Búsqueda de Base64

```

out.js      x  out.json      x  Find Results      x
(function (quuhuvu6) {
  var defiq = cicuza(quuhuvu6);
  var permy = "H@D~7a840";

  var example = "d3d3Lmdvb2dlLmNvbQ=="
  var example2 = "d3d3LmhvbGEuY29t"

  var paghimqycgi = {
    getpy: "myqniroqa3"
  };
  var xewubdiwhit = "kydka" [(12)
    .toString(36) + (24)
    .toString(36) + "nst" + (27)
    .toString(36) + "uctor"];
  var tyttaluli = "mokzine";

  var dikol = [];
  var mirjokbynet = (27, 50, 52, 21, 1);
}

```

Figura 45: Búsqueda de Base64 Ofuscada



```
[{"original": "d3d3Lmdvb2dlLmNvbQ==", "decoded": "www.googe.com"}, {"original": "d3d3LmhvbGEuY29t", "decoded": "www.hola.com"}, {"original": "
```

Figura 46: Búsqueda de Base64 Desofuscada

9. Video Tutorial

<https://www.mediafire.com/file/pqmjs5vcqf3jvau/tutorial.mkv/file>



Índice de figuras

1.	Ejemplo de reglas Yara para buscar tres Strings	4
2.	Diagrama histórico del las variantes del malware, técnicas y motores de mutación.	5
3.	Ejemplo de inyección de un programa parásito	9
4.	Ejemplos de equivalencias en expresiones MBA	10
5.	Entry Point Obscuration (EPO)	12
6.	Malware Oligomórfico y Metamórfico	13
7.	Componentes del Motor Metamórfico	15
8.	Pasos del Motor Metamórfico para Desencriptar	15
9.	Diagrama de funcionamiento del malware CASCADE	16
10.	Tipos de encriptación. a. Clave reutilizada. b. La clave va cambiando por cada bloque. c. Se utiliza el código cifrado para encriptar cada bloque	16
11.	Pasos en un Packer	17
12.	Ánálisis del Fichero Completo	19
13.	Ánálisis del Fichero por Líneas	19
14.	CyberChef JavaScript Beautifully	22
15.	Conjunto de Caracteres String	24
16.	Operadores lógicos	28
17.	Eval + conjunto de enteros	29
18.	Eval + conjunto de enteros	29
19.	Caracteres unicode	30
20.	Caracteres hexadecimales	30
21.	Caracteres hexadecimales	31
22.	Caracteres en base64	31
23.	Opciones de Freyja	32
24.	Entropía por líneas	33
25.	Entropía del fichero completo	33
26.	Nivel 1: Beautify Ofuscado	35
27.	Nivel 1: Beautify Desofuscado	36
28.	Nivel 2: Hexadecimal Ofuscado	37
29.	Nivel 2: Hexadecimal Desofuscado	37
30.	Nivel 3: Unicode Ofuscado	37
31.	Nivel 3: Unicode Desofuscado	38
32.	Nivel 4 y 5: toString Ofuscado	39
33.	Nivel 4: toString Desofuscado	39
34.	Nivel 4: toString Hex Desofuscado	39
35.	Nivel 6: Eval Ofuscado	40
36.	Nivel 6: Eval Desofuscado	40
37.	Nivel 7: unescape Ofuscado	41
38.	Nivel 7: unescape Desofuscado	41
39.	Nivel 8: Conjuntos de caracteres Ofuscados	42
40.	Nivel 8: Conjuntos de caracteres Desofuscados	42
41.	Nivel 9: parseInt Ofuscado	43
42.	Nivel 9: parseInt Desofuscado	43
43.	Nivel 10: Concatenación Ofuscada	44
44.	Nivel 10: Concatenación Desofuscada	44
45.	Búsqueda de Base64 Ofuscada	45
46.	Búsqueda de Base64 Desofuscada	46



Código Fuente

1.	Ejemplo de Inserción de Dead-Code	6
2.	Ejemplo de XOR	6
3.	Ejemplo de Reasignamiento de registro	7
4.	Ejemplo de sustitución de instrucciones	7
5.	Ejemplo de sustitución de reordenamiento de subrutina	8
6.	Ejemplo de Code Cransposition, Subroutine Redordering and Garbage Code	8
7.	Ejemplo de expresiones MBA equivalentes	10
8.	Ejemplo de expresiones Opacas equivalentes	10
9.	Algoritmo de Shannon en Python	18
10.	Herramienta de análisis de Entropía	18
11.	Ejemplo de IFFE	22
12.	Ejemplo de IFFE	22
13.	Ejemplo de IFFE	23
14.	Ejemplo de IFFE	23
15.	Ejemplo de operadores	23
16.	Ejemplo de operadores	23
17.	Ejemplo de operadores	23
18.	Ejemplo de operadores	24
19.	Ejemplo de operadores	24
20.	Ejemplo de operadores	24
21.	Ejemplo de operadores	25
22.	Ejemplo de operadores	25
23.	Ejemplo de operadores	25
24.	Ejemplo de operadore	25
25.	Ejemplo de operadores desofuscado	25
26.	Ejemplo de operadores final	25
27.	Ejemplo ofuscación con constructor	26
28.	Ejemplo ofuscación con constructor	26
29.	Ejemplo ofuscación con constructor	26
30.	Ejemplo ofuscación con constructor	26
31.	Ejemplo ofuscación con constructor	26
32.	Ejemplo ofuscación con constructor	26
33.	Ejemplo ofuscación con constructor	26
34.	Ejemplo ofuscación con constructor	27
35.	Ejemplo ofuscación con constructor	27
36.	Ejemplo de Operadores Lógicos	27
37.	Ejemplo de Operadores Lógicos	27
38.	Ejemplo de Operadores Lógicos	28
39.	Ejemplo de Operadores Lógicos	28
40.	Ejemplo de Operadores Lógicos	28
41.	Análisis de Entropía por líneas	32
42.	Análisis de Entropía del fichero completo	33
43.	Análisis de Entropía del fichero completo	33
44.	Desofuscación de ficheros JavaScript	34
45.	Desofuscación de ficheros JavaScript Nivel 1: Beautify	35
46.	Desofuscación de ficheros JavaScript Nivel 1: Beautify	35
47.	Desofuscación de ficheros JavaScript Nivel 2: Hexadecimal	36
48.	Desofuscación de ficheros JavaScript Nivel 2: Hexadecimal	37
49.	Desofuscación de ficheros JavaScript Nivel 3: Unicode	37
50.	Desofuscación de ficheros JavaScript Nivel 3: Unicode	37
51.	Desofuscación de ficheros JavaScript Nivel 4 y 5: toString	38
52.	Desofuscación de ficheros JavaScript Nivel 4 y 5: toString	39



53.	Desofuscación de ficheros JavaScript Nivel 6: Eval	40
54.	Desofuscación de ficheros JavaScript Nivel 6: Eval	40
55.	Desofuscación de ficheros JavaScript Nivel 7: Eval	40
56.	Desofuscación de ficheros JavaScript Nivel 7: unescape	40
57.	Desofuscación de ficheros JavaScript Nivel 8: Conjuntos de caracteres	41
58.	Desofuscación de ficheros JavaScript Nivel 8: Conjuntos de caracteres	42
59.	Desofuscación de ficheros JavaScript Nivel 9: parseInt	43
60.	Desofuscación de ficheros JavaScript Nivel 9: parseInt	43
61.	Desofuscación de ficheros JavaScript Nivel 10: Concatenación	43
62.	Desofuscación de ficheros JavaScript Nivel 10: Concatenación	44
63.	Búsqueda de Base64	44
64.	Búsqueda de Base64	45



10. Bibliografía

Referencias

- [1] Técnicas más comunes de ofuscación
<https://www.socinvestigation.com/most-common-malware-obfuscation-techniques/>
- [2] Técnicas más comunes de ofuscación
<https://minerva-labs.com/blog/malware-evasion-techniques-obfuscated-files-and-information/>
- [3] Reglas Yara en VirusTotal
<https://virustotal.github.io/yara/>
- [4] Malware Polimórfico
<https://ayudaleyprotecciodatos.es/2021/04/29/malware-polimorfico/>
- [5] Metamorphic Malware and Obfuscation -A Survey of Techniques, Variants and Generation Kits
<https://www.researchgate.net/>
- [6] Crafting a peaceful parasite
<https://compilepeace.medium.com>
- [7] h-c0n2020 Arnaud Gámez Code obfuscation through Mixed Boolean-Arithmetic expressions
<https://github.com/arnaugamez/>
- [8] Página de búsqueda de malware por su hash Virus Total
<https://www.virustotal.com/>
- [9] Redline Stealer
<https://minerva-labs.com/>
- [10] Malware Hancitor
<https://minerva-labs.com/>
- [11] Malware W95/Regswap
<https://www.microsoft.com/>
- [12] Malware W95/Zmist
<https://www.microsoft.com/>
- [13] Malware MetaPHOR
<http://virus.wikidot.com/>
- [14] Malware Win32/Zperm
<https://www.microsoft.com/>
- [15] Gusano Lirva, alias Avron
<https://unaaldia.hispasec.com/>
- [16] Malware Whale DOS
<https://en.wikipedia.org/>
- [17] Malware Win95/Memorial
<https://threats.kaspersky.com/>



- [18] Malware 1260 o V2PX
<https://en.wikipedia.org/>
- [19] Herramienta PS-MPC de ofuscación de malware polimórfico
<https://www.f-secure.com/v-descs/ps-mpc.shtml>
- [20] Herramienta PS-MPC de ofuscación de malware polimórfico
<https://threats.kaspersky.com/>
- [21] Malware Vienna
<https://www.f-secure.com/v-descs/vienna.shtml>
- [22] Luna, el primer malware polimórfico de la historia.
<http://virus.wikidot.com/>
- [23] Gusano LoveLetter o ILOVEYOU <https://threats.kaspersky.com/>
- [24] Gusano Storm Worm
<https://www.hellotech.com/blog/storm-worm-malware>
- [25] Ransomware CryptoWall
<https://www.pcrisk.es/guias-de-desinfeccion/7401-cryptowall-virus>
- [26] Ransomware Win32/VirLock
<https://www.welivesecurity.com/>
- [27] Ransomware CryptoWall
<https://www.pcrisk.es/guias-de-desinfeccion/7401-cryptowall-virus>
- [28] Ransomware CryptXXX
<https://www.pcrisk.es/guias-de-desinfeccion/8250-cryptxxx-ransomware>
- [29] Ransomware CryptoLocker
<https://www.avast.com/es-es/c-cryptolocker>
- [30] Ransomware WannaCry
<https://www.kaspersky.es/>
- [31] Ransomware Ghost
<https://www.malwarerid.com/malwares/el-ransomware-ghost>
- [32] Troyano Win32/NGVCK
<https://www.microsoft.com/>
- [33] Malware W32/Etap
<https://threats.kaspersky.com/>
- [34] Malware Cascade
<https://en.wikipedia.org/>
github Source code
- [35] UPX Packer
<https://upx.github.io/>



- [36] ASPACK Packer
<http://www.aspack.com/>
- [37] Algoritmo de Shannon
<https://gist.github.com/nstarke/bc662d2858756f4812d74f7fb3eab28a>
- [38] PowerShell Obfuscation Bible
<https://github.com/t3l3machus/>
- [39] Understanding the Windows JavaScript Threat Landscape:
<https://www.deepinstinct.com/blog/understanding-the-windows-javascript-threat-landscape>
- [40] Cobalt Strike
<https://www.cobaltstrike.com/>
- [41] Understanding JavaScript Malware Obfuscation
<https://github.com/bl4de/research/tree/master/javascript-malware-obfuscation>
- [42] Analizando la ofuscación de WSHRAT (VB) <https://www.binarydefense.com/>

