MÁSTER EN ANÁLISIS DE MALWARE, REVERSING Y BUG HUNTING

Universidad Católica de Murcia
ENIT - Campus Internacional de Ciberseguridad

Estudio práctico de técnicas de ofuscación y contramedidas aplicables

PRESENTA

María del Carmen San José Seco
@drkrysSrng/freyja

PROFESOR

David García

ASIGNATURA

Trabajo de Fin de Máster

29 de agosto de 2023

Índice

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 la base de datos de Virus Total [7].

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 = "UVODFRYSIHLNWPEJXQZAKCBGMT"

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

Figura 1: Ejemplo de reglas Yara para buscar tres strings

2. Diagrama evolutivo del malware

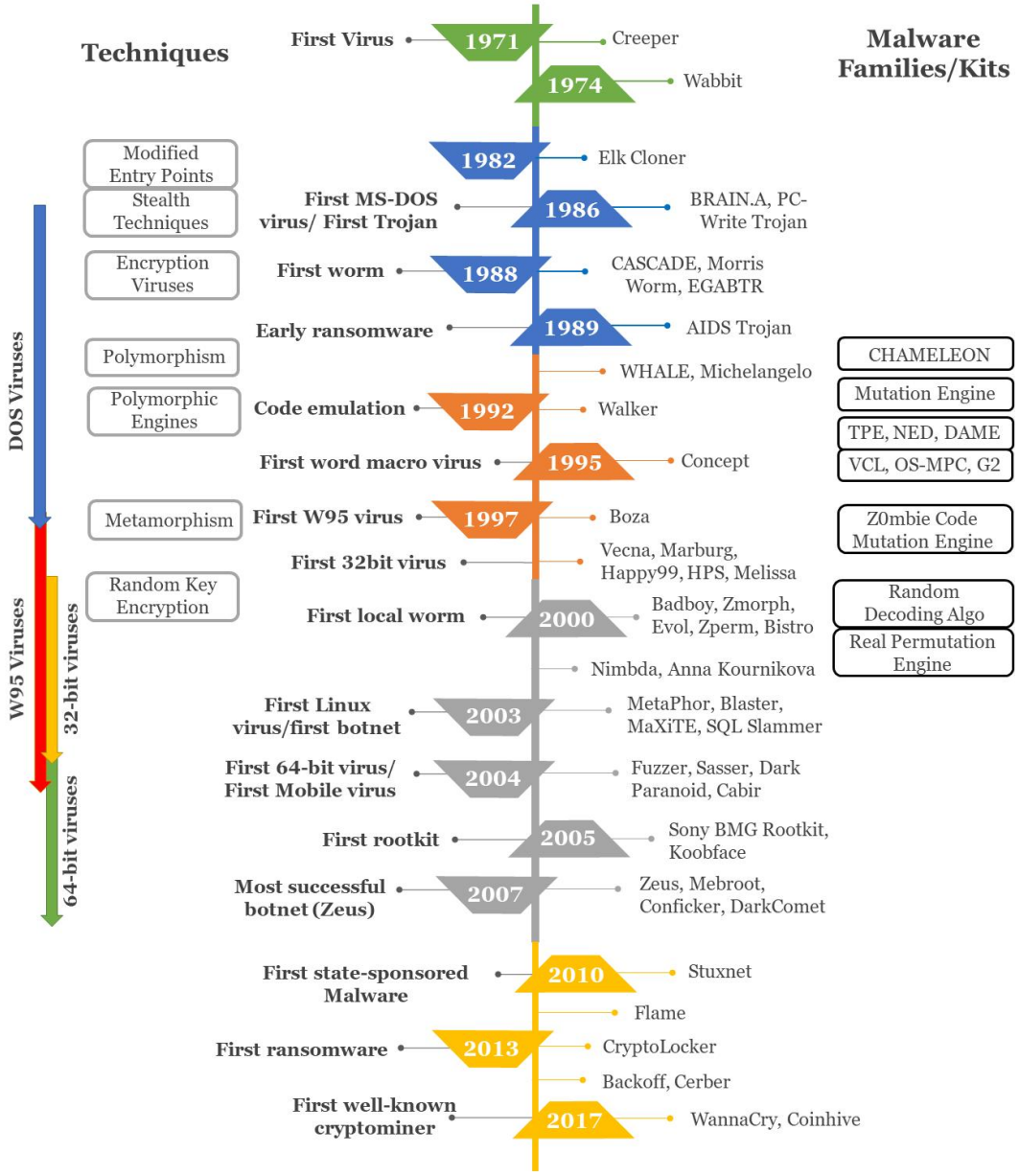

Figura 2: Diagrama histórico

3. Tipos de Ofuscación de malware

3.1. Packing

Se comprime el fichero para que tenga una firma distinta al original. Además de eso, también 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 [8] y Hancitor [9]

Sin embargo, el hecho de que un programa esté empaquetado, ya es sospechoso, por lo que el malware moderno trata de no utilizar este sistema para pasar desapercibido.

3.2. Dead-Code Insertion

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 eta 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 que afecta en nada al 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 Dead-Code Insertion

3.3. XOR

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

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

Código 2: Ejemplo de XOR

3.4. Register reassignment

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 [10]

```
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
```

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

Código 3: Ejemplo de Reasignamiento de registro

3.5. Instruction Substitution

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 [11], y también en malware avanzados como Evol, MetaPHOR [12], Zperm [13] y Avron[14].

```
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 afabeto 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. Code Transposition

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

- La primera técnica, cambia 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 incondiconales o saltos.
- La segunda técnica crea 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. Subroutine reordering

También conocido como Block Reordering, 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 subroutine reordering

```
; Before obfuscation
mov eax, ecx
mov ebx, 10
mul ebx
add eax, 5
mov ecx, eax
; After obfuscation
mov ebx, 10
jmp F1
jnk
F2: push edx; jnk
pop ecx
jmp F3
F1: mul, ebx
add ecx, 1; jnk
add ecx, 5
jmp F2
F3: mul ebx
```

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

3.8. Code integration

Fue por primera vez visto en W95/Zmist [11], 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.

3.9. MBA Expressions

Las expresiones MBA sirven para ofuscar el código utilizando operadores booleanos o polinomios dejando la funcionalidad del código intacta. A continuación vemos dos expresiones equivalentes:

```
def e1(x, y):
  return x + y


def e2(x, y ):
  return (x + y) + 2 * (x & y)


print("El resutado de e1 es: ", e1(3, 4))
print("El resutado de e2 es: ", e2(3, 4))

#Resultado:
El resutado de e1 es:  7
El resutado de e2 es:  7
```

Código 7: Ejemplo de expresiones MBA equivalentes

3.10. Opaque Expressions

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:

```

def o1():
  x = 5

  if x % 2:     # 5 not divyde by 2 so it is always true
    x = 1 << x
    x = 2 * x + 9
    return x

```

```
10
11      def o2():
12        x = 5
13        if ((4 * x * x + 4) % 19) != 0: # For all integers
14          x = 1 << x
15          x = 2 * x + 9
16          return x
17
18
19      def o3():
20        x = 5
21        if ((4 * x * x + 4) % 19) != 0: # for all integers
22          if x % 2:
23            x = 1 << x
24            x = 2 * x + 9
25            return x
26          else:
27            x *= 5
28            return x
29        else:
30          x += 85
31          return x
32
33      # Resultado
34      El resutado de o1 es:  73
35      El resutado de o2 es:  73
36      El resutado 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.

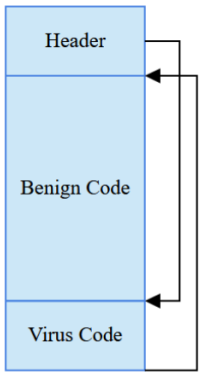

Figura 3: Entry Point Obscuration (EPO)

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 alguas partes del código. Además, el uso de emuladores ver cómo se extrae el malware y que el emuladdor monitoree la memoria y el código que se dumpea. El primer malware Oligomórfico fue Whale DOS [15], identificado por primera vez en 1990. En la siguiente imagen, la rutina de desencriptado se utiliza para desencriptar el body y evitar la detección del malware.

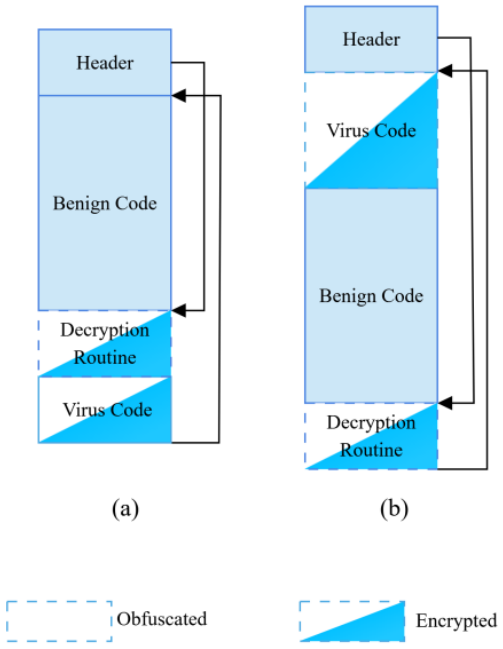

Figura 4: Malware Oligomórfico y Metamórfico

La limitación de esta técnica, es que el bucle de posibles descifradores es finito. Por ejemplo el W95/Memorial [16] 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 [17] y famosos generadores como Phalcon/Skism Mass-Produced Code Generator (PS-MPC) [18] y Virus Creation Lab (VCL)[19] que se siguen utilizando actualmente.

4.2. Polimorfismo

Este malware es capaz de desencriptar como ofuscar y recompilar todo en uno. El cuerpo dsencriptado 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 [20]).
- 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 porlimórficos desarrollado por Bumblebee en España en el año 1999. Apareció en la cuarta versión de la revista 29A. [21]
- Loveletter. Uno de los primeros gusanos en alcanzar un gran número de daños.[22]
- 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. [23]
- CryptoWall. Es un ransomware que cifra los ficheros de la computadora de la víctima exigiendo el pago de un rescate para su descifrado. [26]
- Virlock. Se vio por primer avez 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. [25]
- 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 cookes, datos de navegación... [27]
- 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. [28]
- 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. Explotó 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. [29]

4.3. Metamorphism

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 [10] seguido del W32/Ghost en el 2000 [30]. Este último contenía 10 submódulos, 3.6 millones de posibles variaciones para hacer una Subrutine Reordering. 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 descrifrado 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 [31] creado en 2001. Este tipo de malware tiene un motor me mutación que contiene muchos subprocesos.

4.4. Metamorphic Engine

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

- **Disassembler:** Responsable de convertir el código binario en instrucciones de ensamblador.
- **Shrinker:** Eliminar la mayoraía 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 contine 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 [11] y W32/Etap [32]

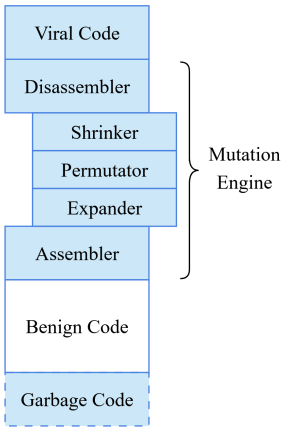

Figura 5: Componentes del Motor Metamórfico

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 únca 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.

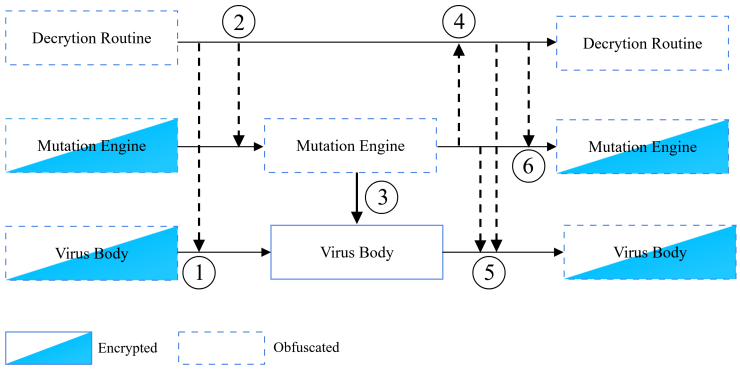

Figura 6: Pasos del Motor Metamórfico para Desencriptar

4.5. Encryption

La primera forma de encriptación fue por el Malware CASCADE utilizando un simple *XOR* [33]

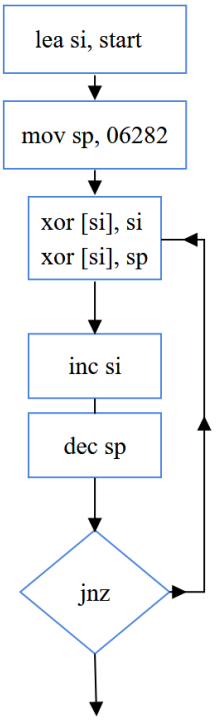

Figura 7: Diagrama de funcionamiento del Malware CASCADE

Más adelante, el malware evolucionó de forma que en lugar de tener una sóla encriptación, tienen docenas de ellas, como en el caso del DOS/Whale en 1990 [15] o de forma aleatoria como en el caso de W32/MetaPHOR [12] La encriptación básica, puede desarollarse con una única clave byte a byte, sin embargo hay formas alternativas en la que la clave se va actualizando a cada paso o incluso utilizando los caracteres encriptados.

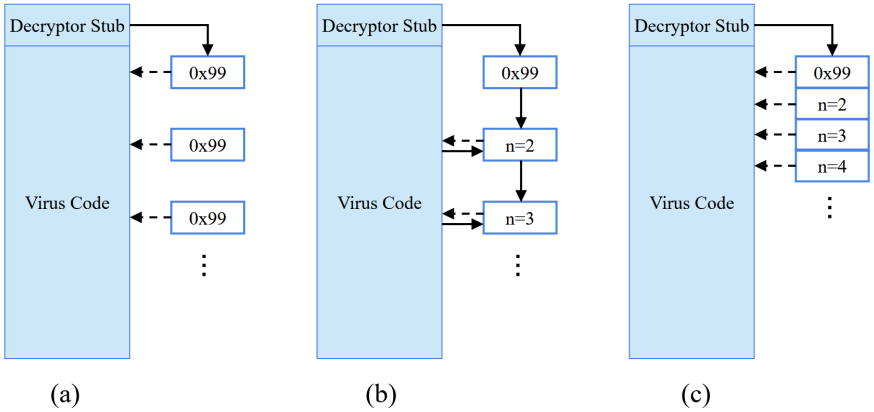

Figura 8: Tipos de encriptación. a. Clave reutilizada. b. La clave va cambiando por cada bloue. c. Se utiliza el código cifrado para encryptar cada bloque

4.6. Compression

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 [34] y ASPACK [35]

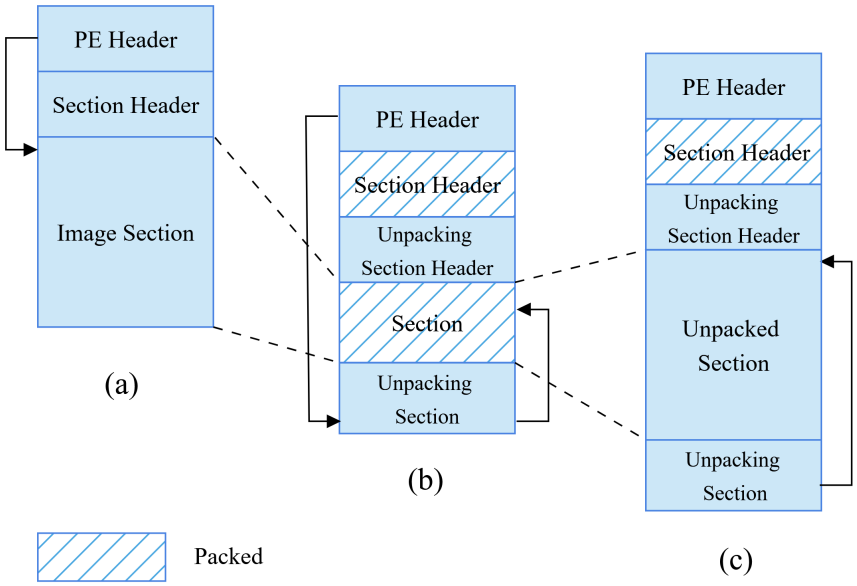

Figura 9: Pasos en un Packer

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:

```
    def entropy(string):
    "Calculates the Shannon entropy of a UTF-8 encoded string"

    # decode the string as UTF-8
    unicode_string = string.decode('utf-8')

    # get probability of chars in string
    prob = [float(unicode_string.count(c)) / len(unicode_string) for c in dict.fromkeys(list(
        unicode_string)) ]

    # calculate the entropy
    entropy = - sum([ p * math.log(p) / math.log(2.0) for p in prob ])

    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 [36] y [37] vamos a hacer una herramienta que nos indique el nivel de ofuscación de un fichero en JavaScript.

```
    #!/bin/python3
    import math

    def entropy_check(string):
    "Calculates the Shannon entropy of a UTF-8 encoded string"

    # decode the string as UTF-8
    unicode_string = string.decode('utf-8')

    # get probability of chars in string
    prob = [float(unicode_string.count(c)) / len(unicode_string) for c in dict.fromkeys(list(
        unicode_string)) ]

    # calculate the entropy
    entropy = - sum([ p * math.log(p) / math.log(2.0) for p in prob ])

    return entropy

    filename = input("Write down the path of the file to analyze >")

    option = input("What do you want to do? Analyze file [file] or line per line [line]>").upper
        ()


    if option == "FILE":
    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 10: 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, OlwpPLsvRDhBKQEV, hxEobzlsRgNfcdX, lAYcunLBjCUqezpbkw, bLGgrdCMRovlpnx, IGhuToEwXlJBQPS,
This line has High Entropy 5.617101379143724
b'function JIUFoiVePNXYTRaOyB(vPsiWmlGbuYgxoe0) { \r'
This line has High Entropy 5.118562939644918
b"KgJtXdpCLGQOuRMmYeV = 'jKAxegJjdOfuKAxegJjdOfEKAxegJjdOfFKAxegJjdOfqKAxegJjdOfeKAxegJjdOfgKAxegJjdOfXodKAxegJjdOfwkKA'+\r"
This line has High Entropy 4.414777089775276
b"'xegJjdOfnWtKAxegJjdOflKAxe'+\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 KAxegJjdOfAcKAxegJjdOftKAxe'+\r"
This line has High Entropy 4.002078406900581
b"'gJjdOfiKAxegJjdOfvKAx'+\r"
```

Figura 11: 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 pude 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 utlizados se encuentran PowerShell, VBScript y JavaScript.

Si bien los ataques de PowerShell son los más utilizados, los actores de amenazas maliciosos 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 malwrae 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. [38]

6.1. WJworm

Vengeance Justice Worm es un malware en JavaScript que combina caracrerísticas 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 externos.

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 adicionar.

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 iriginalmente 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.

[2] RAT, 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 harvesting [3] 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 bypasear la AMSI [4], borrar las shadow copies [5] 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 sitema, si este fichero no lo encuentra, finaliza su ejecución.

A diferencia de la mayoría del Ransonmware 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]

[3] Conseguir cuentas de correo

[4] Interfaz de examen antimalware

[5] Copias de Seguridad

[6] Herramienta de Red Team para probar los sitemas y sus mecanismos de protección.

7. Pasos para Desofuscar un Código Malware en JavaScript

7.1. JavaScript Beautifully

En ocasiones nos encontramos con código en JavaScript que está escrito en una sola línea, como ocurre con la muestra *2b0c9059feece8475c71fbbde6cf4963132c274cf7ddebafbf2b0a59523c532e.js* del malware WJWorm. Para ello, tenemos que utilizar la herramienta JavSCript Beautify de CyberChef

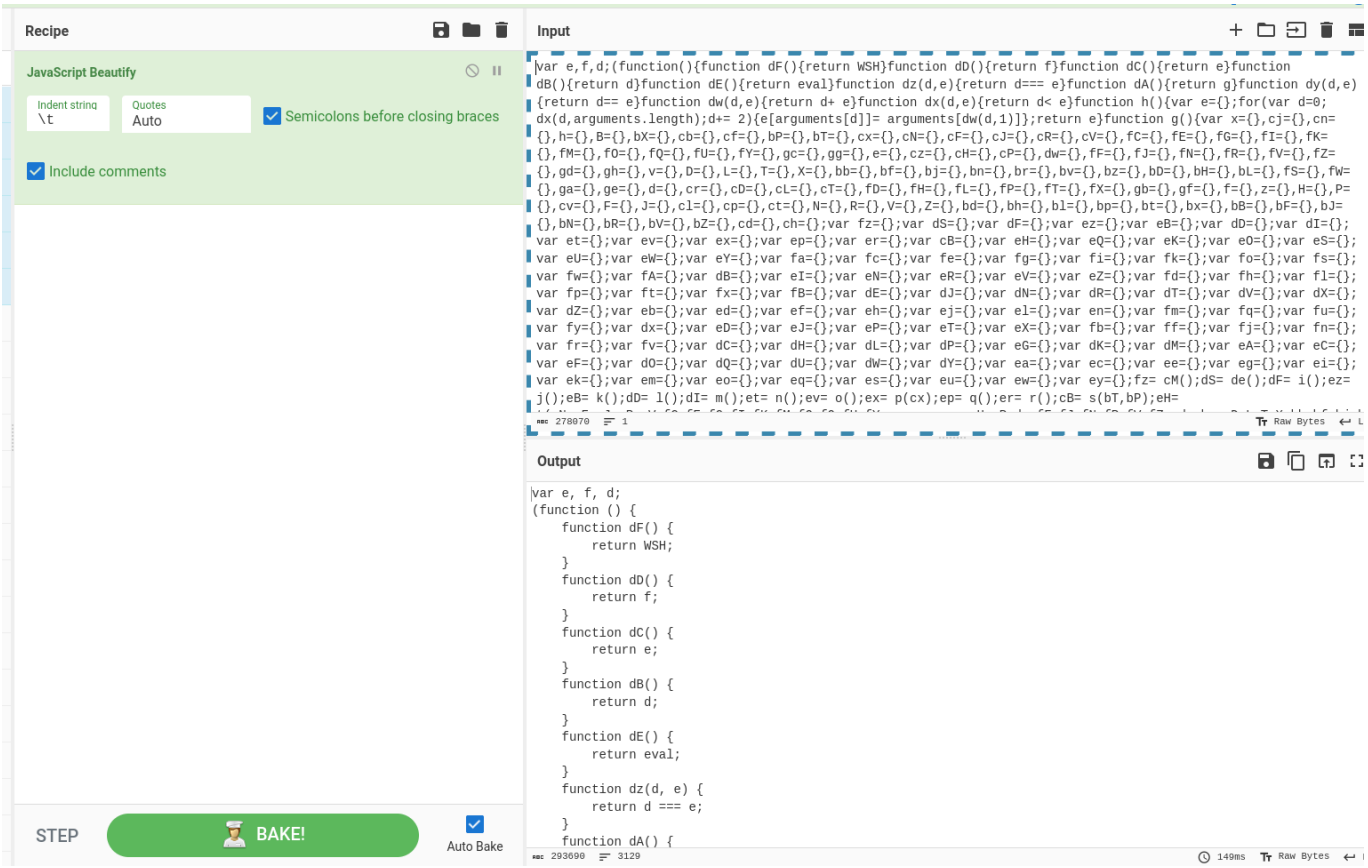

Figura 12: CyberChef JavaScript Beautifully

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:

```
function hello(message) {
  console.log(message)
}
```

Código 11: Ejemplo de IIFFE 1

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

```
function hello(message) {
  console.log(message)
}

hello('This is test')
```

Código 12: Ejemplo de IIFFE 2

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

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

Código 13: Ejemplo de IIFFE 3

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:

```
(function(quhuvu6) {

  // ...

}("41553a304f0b442551284206" + "672651014d1e1a60127" + ...
```

Código 14: Ejemplo de IIFFE 4

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()`:

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

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,a sí que llamaremos a la variable `result`:

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

Código 16: Ejemplo de operadores 2

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

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

Código 17: Ejemplo de operadores 3

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:

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

Código 18: Ejemplo de operadores 4

Ahora nos vamos a enfocar en el siguiente código:

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

Código 19: Ejemplo de operadores 5

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:

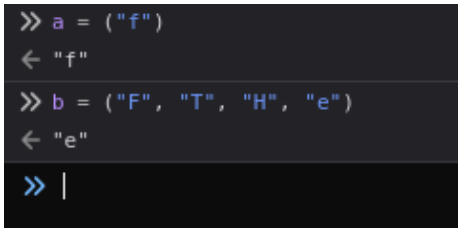

Figura 13: Conjunto de Caracteres String

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:

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

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

Código 20: Ejemplo de operadores 6

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:

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

Código 21: Ejemplo de operadores 7

Si observamos la siguiente línea:

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

Código 22: Ejemplo de operadores 8

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:

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

Código 23: Ejemplo de operadores 9

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

Código 24: Ejemplo de operadores 10

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.

```
    function cicuza(val) {
      var result = [];
      for (var i = 0; i < val.length; i += 2) {
        result.push(parseInt(val.substr(i, 2), 16));
      }
      return result;
    };
```

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

```
    (...)
    var defiq = cicuza(quhuvu6);    // here we are so far :)
    var permy = "H@D~7a84O";
    var paghimqycgi = {
      getpy: "myqniroqa3"
    };
```

```
(...)
```

Código 26: Ejemplo de operadores post

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:

```
function doubleX(x) {
  return x * 2
}

doubleX(10) // returns 20
```

Código 27: Ejemplo de obtener el constructor 1

Vamos a ofuscar este código:

```
let xcf = new Function("x","return x * 2")

xcf(10)   // returns 20 as well
```

Código 28: Ejemplo de obtener el constructor 2

`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 que es `Function()` ella misma.

Es decir, sustituyendo el código anterior:

```
let xcf = a.forEach.constructor("x","return x * 2")

xcf(10)  // yep, it works! 20
```

Código 29: Ejemplo de obtener el constructor 3

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

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

Código 30: Ejemplo de obtener el constructor 4

Este código equivale a:

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

Código 31: Ejemplo de obtener el constructor 5

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

Vamos a aplicar esto al código de nuestro malware:

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

Código 32: Ejemplo de obtener el constructor 6

Entonces esto acabará siendo:

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

Código 33: Ejemplo de obtener el constructor 7

Que es lo mismo que:

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

Código 34: Ejemplo de obtener el constructor 8

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

```
(function(quhuvu6) {
  var defiq = cicuza(quhuvu6);
  var permy = "H@D~7a84O";
  var paghimqycgi = {
    getpy: "myqniroqa3"
  };
  var fnConstructor = String.constructor;
  var tyttaluli = "mokzine";

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

  (...)
```

Código 35: Ejemplo de obtener el constructor 9

7.5. Operadores Lógicos

Si analizamos el siguiente fragmento:

```
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[(21).toString(0x24) + "e" + "n" + (16).toString
      (36) + ("L", "S", "x", "t") + ("I", "D", "h") /*Q5E278CBpoixvOtUNpix*/ ] - mirjokbynet))
      [("d", "R", "p", "s") + ("H", "K", "A", "s", "D", "p") + "l" + "i" + ("Y", "b", "h", "t"
      ) /*O7MOfVrRkP9RlXlfKLxi*/ ]('');
  for (var juqno = +!!false; juqno < defiq[("R", "t", "E", "l") + ("y", "e", "f", "R", "e") +
      (23).toString(36) + (16).toString(36) + "t" + "h" /*H3RMPYzEeu55OVeGgb1v*/ ]; juqno++) {
    defiq[juqno] = defiq[juqno] ^ dikol[juqno % dikol["l" + (14).toString(36) + "n" + (16).
        toString(0x24) + (29).toString(0x24) + ("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" /*YZz3OuivKuwgqjkFVKu0*/ ]((88, 53,
        3, 90, 0));
  }
  mirjokbynet++;
};
```

Código 36: Ejemplo de Operadores Lógicos 1

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` yu `juqno` a `j` también:

```

while (k <= permy.length) {
  dikol = (permy.substr(permy.length - k)).split('');
  for (var j = +!!false; j < defiq.length; j++) {
    defiq[j] = defiq[j] ^ dikol[j % dikol.length].charCodeAt(0);
  }
  k++;
};
```

Código 37: Ejemplo de Operadores Lógicos 2

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 3

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

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

Código 39: Ejemplo de Operadores Lógicos 4

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 5

¿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`

Figura 14: 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.lenght` 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 ^ es el operador XOR. El resultado de `65␣^␣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. Código en un conjunto de enteros

Nos podemos encontrar como en esta muestra, sacada de un pdf, la función `var␣n␣=␣"";␣eval(eval('Stri'+␣n␣+␣'ng.f` con un listado de números. Si nos vamos a CyberChef, reemplazamos las comas por espacios y ejecutamos *From Charcode* con la base en 10, vemos que se transforma en un código en Javascript:

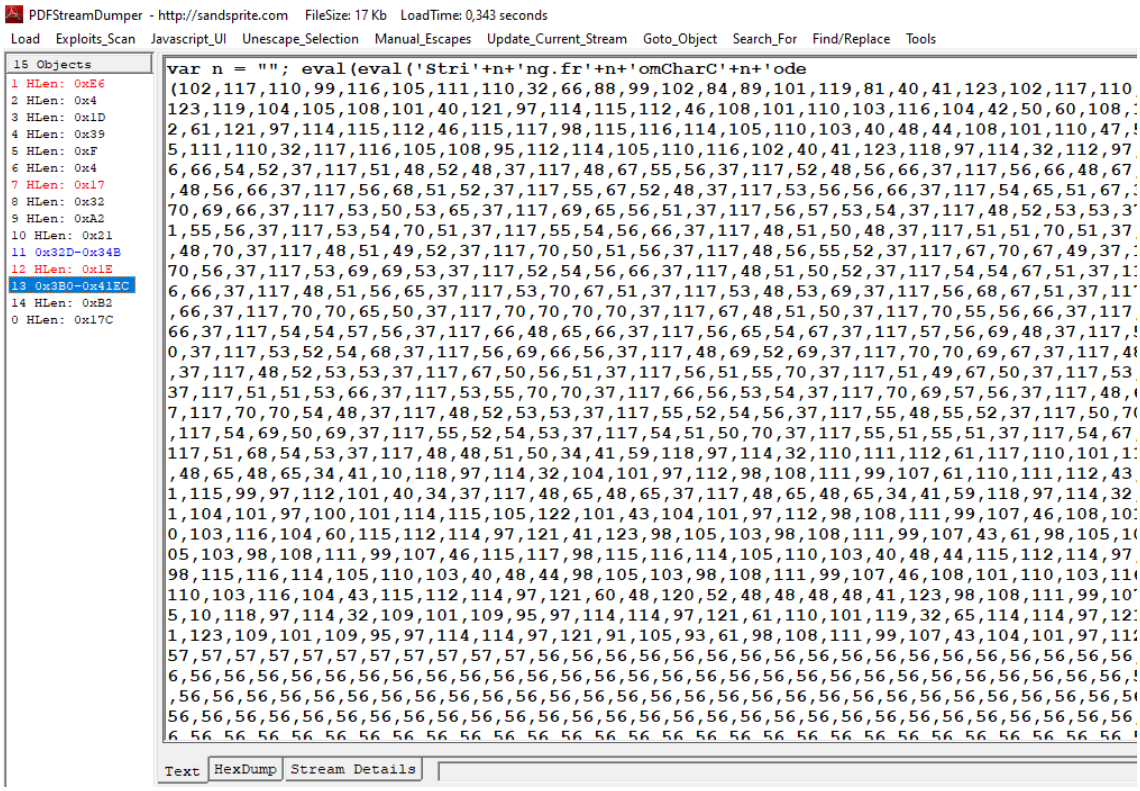

Figura 15: Conjunto de enteros

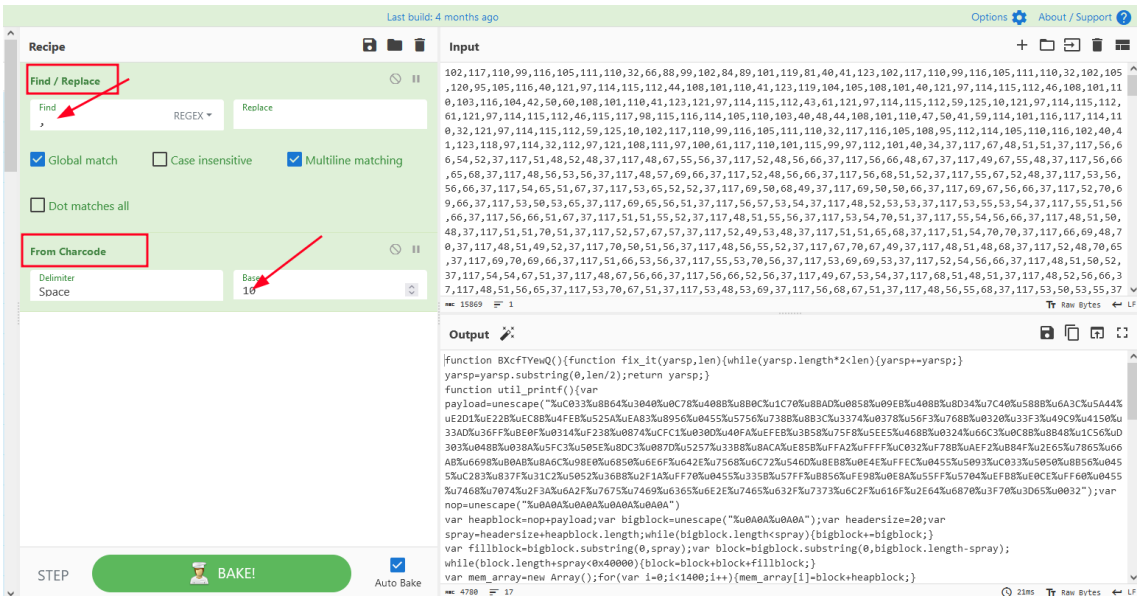

Figura 16: Conjunto de enteros

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:

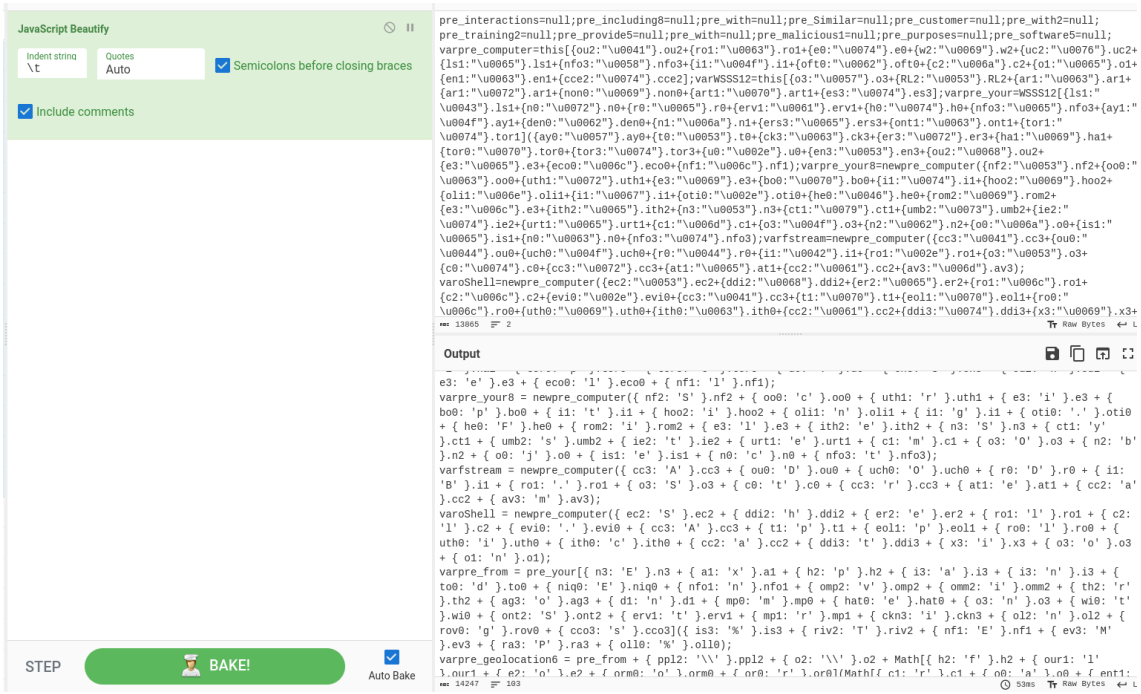

Figura 17: Caracteres unicode

7.6.3. Caracteres en Hexadecimal

En el siguiente snippet del malware WSHRat [38], podemos observar que tiene muchas partes que están ofuscadas en valores hexadecimales:

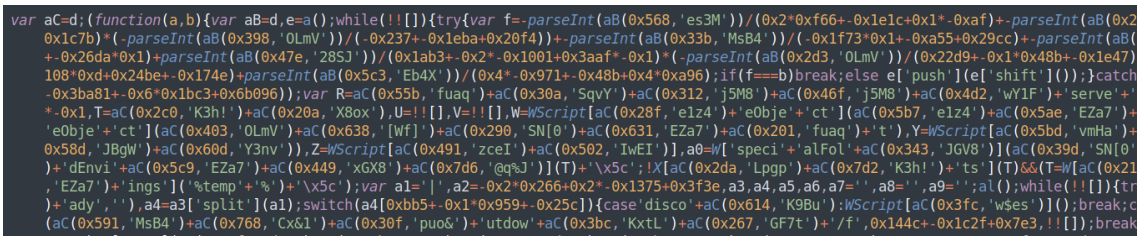

Figura 18: Caracteres hexadecimales

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:

Figura 19: Caracteres hexadecimales

7.7. Base64

El siguiente snippet de código forma parte del payload de STRRAT[38], como podemos observar, es un largo Base64, que a continuación decodificará.

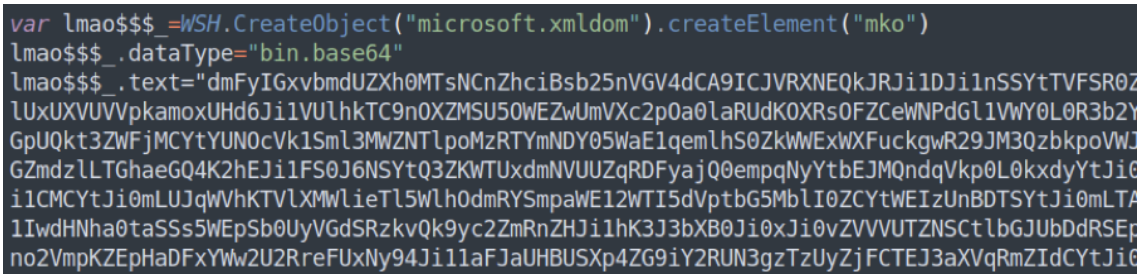

Figura 20: Caracteres hexadecimales

8. Freyja Deobfuscation Tool

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

Esta herramienta tendrá las siguientes funcionalisades:

- Limpia el código y lo tabula de manera similar, no sólo tabulando las instrucciones, sino que parsea los caracteres en hexadecimal y unicode.
- Checkeo de entropía, 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.
- Búsqueda de strings en base64 y su decodificación.

Índice de figuras

Código Fuente

9. 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 Virus Total
https://virustotal.github.io/yara/

[4] Malware Polimórfico
https://ayudaleyprotecciondatos.es/2021/04/29/malware-polimorfico/

[5] Estudio de las diferentes técnicas de Ofuscación de Malware
https://www.researchgate.net/

[6] h-c0n2020 Arnau Gámez Code obfuscation through Mixed Boolean-Arithmetic expressions
https://github.com/arnaugamez/

[7] Página de búsqueda de malware por su hash Virus Total
https://www.virustotal.com/

[8] Redline Stealer
https://minerva-labs.com/

[9] Malware Hancitor
https://minerva-labs.com/

[10] Malware W95/Regswap
https://www.microsoft.com/

[11] Malware W95/Zmist
https://www.microsoft.com/

[12] Malware MetaPHOR
http://virus.wikidot.com/

[13] Malware Win32/Zperm
https://www.microsoft.com/

[14] Gusano Lirva, alias Avron
https://unaaldia.hispasec.com/

[15] Malware Whale DOS
https://en.wikipedia.org/

[16] Malware Win95/Memorial
https://threats.kaspersky.com/

[17] Malware 1260 o V2PX
https://en.wikipedia.org/

[18] Herramienta PS-MPC de ofuscación de malware polimórfico
https://www.f-secure.com/v-descs/ps-mpc.shtml

[19] Herramienta PS-MPC de ofuscación de malware polimórfico
https://threats.kaspersky.com/

[20] Malware Vienna
https://www.f-secure.com/v-descs/vienna.shtml

[21] Luna, el primer malware polimórfico de la historia.
http://virus.wikidot.com/

[22] Gusano LoveLetter o ILOVEYOU https://threats.kaspersky.com/

[23] Gusano Storm Worm
https://www.hellotech.com/blog/storm-worm-malware

[24] Ransomware CryptoWall
https://www.pcrisk.es/guias-de-desinfeccion/7401-cryptowall-virus

[25] Ransomware Win32/VirLock
https://www.welivesecurity.com/

[26] Ransomware CryptoWall
https://www.pcrisk.es/guias-de-desinfeccion/7401-cryptowall-virus

[27] Ransomware CryptXXX
https://www.pcrisk.es/guias-de-desinfeccion/8250-cryptxxx-ransomware

[28] Ransomware CryptoLocker
https://www.avast.com/es-es/c-cryptolocker

[29] Ransomware WannaCry
https://www.kaspersky.es/

[30] Ransomware Ghost
https://www.malwarerid.com/malwares/el-ransomware-ghost

[31] Troyano Win32/NGVCK
https://www.microsoft.com/

[32] Malware W32/Etap
https://threats.kaspersky.com/

[33] Malware Cascade
https://en.wikipedia.org/
github Source code

[34] UPX Packer
https://upx.github.io/

[35] ASPACK Packer
http://www.aspack.com/

[36] Algoritmo de Shannon
https://gist.github.com/nstarke/bc662d2858756f4812d74f7fb3eab28a

[37] PowerShell Obfuscation Bible
https://github.com/t3l3machus/

[38] Understanding the Windows JavaScript Threat Landscape:
https://www.deepinstinct.com/blog/understanding-the-windows-javascript-threat-landscape

[39] Cobalt Strike
https://www.cobaltstrike.com/

[40] Understanding JavaScript Malware Obfuscation
https://github.com/bl4de/research/tree/master/javascript-malware-obfuscation

[41] Analizando la ofuscación de WSHRAT (VB) https://www.binarydefense.com/