Solución al “hackcontest” de Balearikus Party 2009

Noviembre 2009 © Daniel Kachakil

Introducción

Este documento describe una solución al **hackcontest**, una competición de seguridad informática desarrollada por **Miguel Gesteiro** que ha sufrido varios cambios desde su primera edición (agosto de 2008).

Este concurso ha estado presente en más de una *Party*, con desafíos que han ido variando a lo largo del tiempo, pero se publica de forma online por primera vez para la **Balearikus Party** de 2009 en la siguiente dirección web:

`http://hackcontest.org:666/tour-party/`

El reto consta de 10 desafíos de dificultad variada que podremos ir superando en cualquier orden, tras habernos dado de alta como usuarios registrados.

Desafío 01: "Supera este sencillo formulario" *(4 puntos)*

Nos aparece un formulario con un campo de texto y una pista que nos indica literalmente que "la clave es **muy fácil**".

Su resolución es tan simple como pulsar el botón "acceder" tras introducir en el campo de texto la cadena "muy fácil" (sin comillas, en minúsculas y respetando la tilde)

Desafío 02: "Otro formulario sencillo" *(5 puntos)*

Otro formulario similar al del primer desafío, pero en esta ocasión la pista nos dice que "no siempre hay que hacer caso al webmaster...".

En esta ocasión revisamos el código fuente de la página y comprobamos que el envío del formulario está condicionado al valor que devuelve la función "chk()", cuyo código se encuentra unas líneas más abajo:

```
<form name="acceso" onsubmit="return chk()" method="post">

<script>
    var campoClave = document.getElementById("clave");
    campoClave.focus();

    function chk() {
        if(campoClave.value.length < 1) {
            alert('Clave vacía');
            campoClave.focus();
            return false;
        } else return true;
    }
</script>
```

Lo único que hace esta función es comprobar si se ha introducido algún texto en el campo clave, mostrándonos una alerta y evitando que se envíe el formulario en caso de que se haya dejado la clave en blanco.

Parece que el webmaster ha delegado esta comprobación para que se verifique en el lado cliente mediante JavaScript, así que intentaremos enviar una clave vacía. Para ello podemos proceder de varias formas. Una de ellas consiste en redefinir la función "chk" para que ejecute otro código (o incluso podría estar vacía), introduciendo esta cadena en la URL de nuestro navegador, seguida de un retorno de carro:

```
javascript:function chk() {};
```

Puesto que hemos anulado la función de validación, ahora bastaría con pulsar el botón y tendríamos el desafío superado. Lo mejor de esta solución, además de su sencillez, es que no requiere de ninguna herramienta adicional.

Como solución alternativa, también podríamos haber utilizado un proxy para interceptar una petición válida del navegador (introduciendo cualquier clave) y editar el campo "clave", dejándolo en blanco. O si lo preferimos, incluso podríamos construir y lanzar una petición POST directamente sobre el servidor, lo que nos resulte más fácil.

Desafío 03: "Uno de base de datos SQL" *(10 puntos)*

En esta ocasión tenemos dos campos (usuario y contraseña) sin pistas, así que haremos caso al título de la prueba y asumiremos que detrás del formulario hay una base de datos SQL.

Lo primero que se nos tiene que pasar por la cabeza al hablar de formularios y bases de datos es la inyección SQL, así que empezaremos probando con una comilla simple en el campo del usuario, pero también tendremos que introducir algún valor en el de la contraseña para pasar la primera barrera, que comprueba que ninguno de los campos esté vacío. Así obtendremos el siguiente error descriptivo, que nos revela la instrucción completa (y también que el gestor de bases de datos es un SQLite):

```
ERROR SQL: SELECT * FROM usuarios WHERE usuario='' AND clave='a'::SQL logic
error or missing database
```

Puesto que no se ha filtrado la entrada de datos, podemos superar esta prueba con la clásica inyección de *bypass* que fuerza a las dos condiciones a tomar un valor verdadero, por lo que la consulta devolverá todos los registros de la tabla. Al estar limitado el máximo de caracteres de los campos del formulario, usaremos la versión minimalista de la inyección de *bypass*, tanto en el usuario como en la contraseña:

```
' or ''='
```

También podemos terminar la instrucción en la primera comprobación (con el punto y coma), introduciendo cualquier cadena en la contraseña y esta como usuario:

```
' or 1=1;
```

Como curiosidad, comentar que se podría haber extraído el usuario y la contraseña usando técnicas de inyección a ciegas: "admin" y "al45!34j". En la práctica esto no tiene ninguna utilidad para el reto, pero en un escenario real tal vez esa misma contraseña nos podría dar acceso a otros servicios interesantes...

Desafío 04: "Un formulario un tanto enrevesado" *(20 puntos)*

Nos encontramos de nuevo con un formulario con un solo campo y sin pistas, por lo que comenzaremos analizando el código fuente de la página. Lo primero que llama la atención es un comentario HTML con el texto "zona encriptada no disponible", seguido de 200 saltos de línea en blanco. Se trata de un intento para despistarnos, ya que el código útil está justo debajo de esa zona:

```
<script src="desafios/04/protect.js" type="text/javascript"></script>
<script type="text/javascript">
    <!-- inpdata="CF575862E6E195C3D094F0BB ... ";document.writeln(dp()); //-->
</script>
```

Por una parte se está incluyendo un fichero JavaScript que contiene algunas funciones para la ofuscación de código y, por otra parte, se inicializa una variable con el código ofuscado y se escribe dinámicamente en la página. Puesto que no queremos entretenernos con el "cómo", vayamos directamente a por el "qué" y para ello no

tenemos más que mostrar lo que devuelve de la función "dp" tecleando lo siguiente en la barra del navegador (al igual que hicimos en el desafío 2):

```
javascript:alert(dp())
```

Esta acción nos mostrará una ventana de alerta con el fragmento de código HTML que nos falta para completar la página. Se trata precisamente del código que compone el formulario, destacando las siguientes partes:

```
<form name="acceso" onsubmit="return autenticar(clave)" method="POST">
...
<script type="text/javascript" src="none">
...
    function autenticar(clave){
        if(clave.value=="xachoveu!") return true;
        msjAviso('te equivocaste...');
        return false;
    }
</script>
```

Tras leer este código, parece evidente que la clave que buscamos es "xachoveu!", pero curiosamente no resulta ser así (y todo tiene su explicación). Hemos pasado por alto un pequeño detalle y es el atributo "src" de la etiqueta "script", que sirve para indicar que la ubicación de un fichero externo. El nombre del fichero elegido ("none") no hace más que confundirnos, puesto que a primera vista cualquiera pensaría que no se ha especificado ningún fichero, pero estaría equivocado, porque este fichero existe y está siendo utilizado por nuestro formulario. Podemos descargarlo desde aquí:

```
http://hackcontest.org:666/tour-party/none
```

En cualquier caso, a igualdad de nombres, debería prevalecer siempre la última definición de la función, tal y como establece esta sección del W3C *("All SCRIPT elements are evaluated in order as the document is loaded")*, así que alguno podría pensar que no importa lo que se haya definido con anterioridad. Ese planteamiento sería totalmente correcto, pero tampoco podemos olvidar que en esta otra sección del mismo documento *("If the src has a URI value, user agents must ignore the element's contents and retrieve the script via the URI")*, se establece que si la etiqueta "script" tiene un valor en "scr", se debe ignorar el contenido del elemento y obtener el script del fichero indicado, que es exactamente lo que está pasando en nuestro caso, así que tendremos que centrarnos en el contenido del fichero.

Observamos que el fichero "none" contiene una línea de código bastante larga, que evalúa y ejecuta una función creada para ofuscar el código ("*function(p,a,c,k,e,r)*"). Podemos ver lo que hace esa función si sustituimos el "eval" inicial por un "alert" y lo ejecutamos en la barra de navegación, destacando el siguiente fragmento de código:

```
function autenticar(x){
    if(x.value=="\u0061c\u0068\u0074\165ng\u0021") return true
... }
```

Si decodificamos estos códigos (lo más rápido es pasarle la cadena a la función "alert" y ejecutarlo en la barra de direcciones del navegador), vemos que la clave que esconde es "achtung!", pero eso tampoco funcionará...

Bueno, el truco está en que cuando decía que el fichero "none" contiene una línea de código, es lo que el autor quería que viéramos, puesto que si nos fijamos bien, unas líneas más abajo nos encontramos con otra función con el mismo nombre, de código muy parecido pero no por ello dejan de ser totalmente diferentes (y recordemos que prevalecerá ésta última). Si repetimos el procedimiento anterior (sustituir la función "eval" por "alert"), esta vez sí que obtendremos la contraseña correcta: "askojscrip"

Por cierto, a pesar de haberos explicado la solución enrevesada, una forma muchísimo más rápida y eficiente de resolver la prueba consiste en ayudarse de algún plugin que permita ver el código fuente generado (como Web Developer para FireFox, Developer Toolbar para IE6-7 o bien simplemente pulsando F12 en IE8), de forma que podamos seleccionar el formulario y saber que la función a la que se invoca se llama "autenticar". Teniendo esto claro, podemos mostrar directamente su código fuente sin más que invocar una ventana de alerta con el nombre de la función (pero siempre prescindiendo de los paréntesis al final del nombre, ya que en ese caso se mostraría el valor que devuelve tras su ejecución y no es lo que queremos):

```
javascript:alert(autenticar)
```

Si usamos Internet Explorer solo nos quedará decodificar la contraseña ("\u0061\u0073k\u006Fj\u0073crip"), pero curiosamente en FireFox se muestra la contraseña directamente como texto en claro. ¿A que visto así no era tan enrevesado? ¿Creéis que se merece esos 20 puntos? ;-)

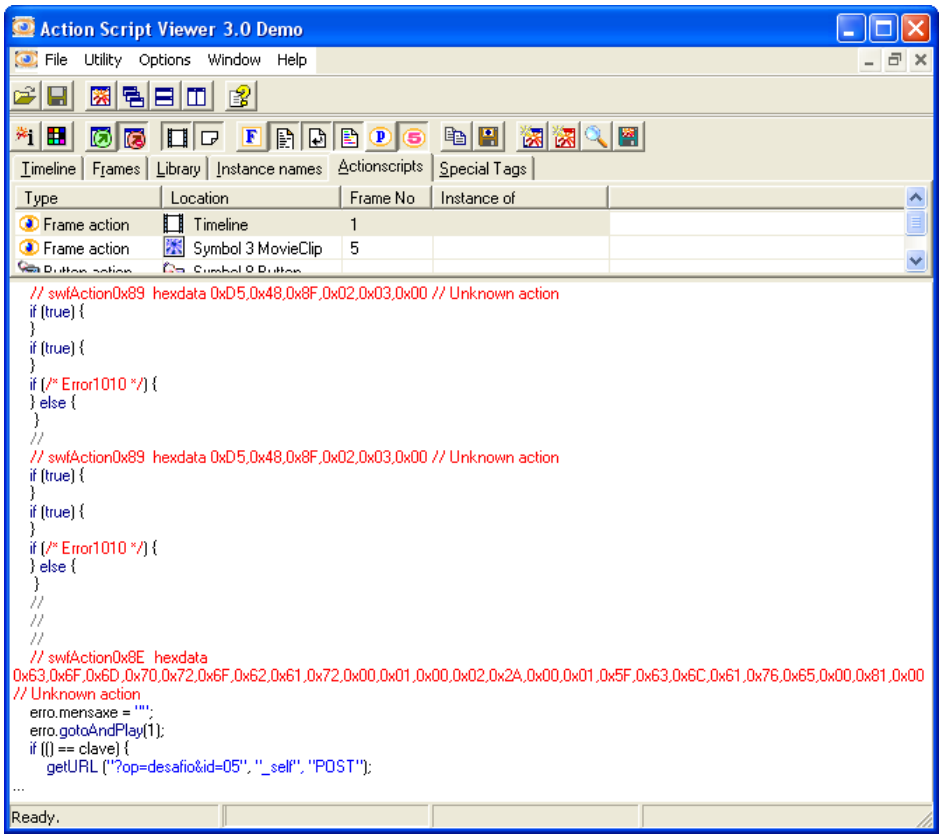
Desafío 05: "Un formulario muy especial..." *(15 puntos)*

En esta ocasión nos encontramos ante un formulario muy parecido a los anteriores, pero con la particularidad de que está desarrollado en Flash. Capturando paquetes o intercalando un proxy, comprobamos rápidamente que la película de Flash no interacciona con el servidor, por lo que asumiremos que la validación está embebida en el propio fichero SWF y nos lo descargaremos para analizarlo localmente.

Curiosamente, no conseguimos obtener el código fuente por más que lo intentamos con varios descompiladores de Flash, seguramente porque estará ofuscado o protegido con alguna técnica o herramienta que lo automatice (en este caso era SWFprotect 1.6). Ya que el SWF está comprimido (tal y como indica su cabecera: CWS), empezaremos descomprimiéndolo con alguna herramienta como la de DCOMSoft para obtener el fichero equivalente descomprimido (cuya cabecera empieza por FWS). Si abrimos este fichero con el bloc de notas, veremos que se leen algunas cadenas como texto en claro. Ya os adelanto que una de ellas es la clave correcta, así que la prueba se podría haber sacado con el método de prueba y error, pero aquí expondré la versión más técnica, larga y rebuscada, que es la que usé yo para resolverla.

Por las pruebas realizadas con varias herramientas, parece ser que los mejores resultados los obtendremos con el Action Script Viewer en su versión 3.0 demo (que a pesar de su antigüedad, es más que suficiente para nuestro propósito). Esta versión solamente muestra las primeras 25 líneas de los primeros 5 cuadros o fotogramas y tampoco está preparado para abrir las versiones más recientes de Flash (posteriores a la versión 6).

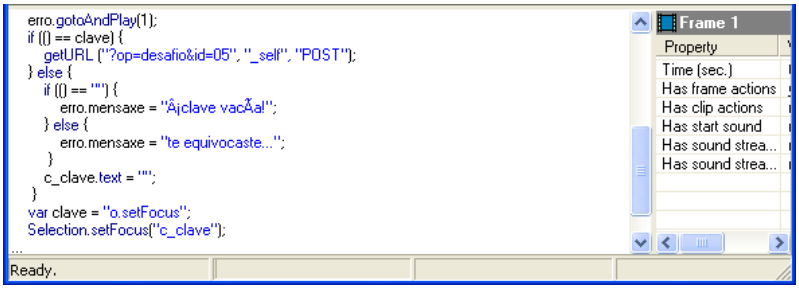
A pesar de los mensajes de aviso que aparecen al abrir el fichero, el programa logra visualizar un fragmento del código, aunque lo que muestra no es muy coherente.



Lo único útil que podemos apreciar es la URL de destino que se invoca cuando acertamos la clave, pero nos falta el parámetro que se envía por POST, así que no nos sirve de mucho. Al haber varias acciones (o instrucciones) desconocidas para el programa, se me ocurrió que se podría investigar cuál es exactamente la función de cada una de ellas, para lo cual me apoyé en dos referencias de la propia web de Adobe: ActionDecoder.java y Dissasembler.java. En el primer fichero nos encontramos con una enumeración de todas las instrucciones disponibles (actionNames) y en el segundo tenemos varios bloques de código que nos indican cómo se interpretan algunas de dichas instrucciones o acciones.

El primer byte (*opcode*) indica el tipo de instrucción a decodificar (por ejemplo: *if*=0x9D, jump=0x99, *nop*=0x77, *function*=0x9B, *jump*=0x99, etc). Una vez reconocida la instrucción, se leen los parámetros de la misma, cuyo número y longitud son variables en el caso general. Nuestro objetivo es el de eliminar selectivamente tantas instrucciones como podamos, pero siempre manteniendo la coherencia (sin destrozar el fichero), con el fin de poder descompilar el fragmento (de 25 líneas como máximo) que contenga el código que valida la clave introducida.

Empezamos localizando el primer bloque que aparece en el desensamblado (es decir, 89 06 00 D5 48 8F…) y comprobamos que está a partir del offset 40. Una vez localizado este bloque inicial, iremos eliminando bloques de instrucciones y volvemos a intentar la descompilación tras cada bloque eliminado. Por ejemplo, si eliminamos exactamente 95 bytes (es decir, hasta justo antes de un bloque 96 02 00 05 00 9D…), obtenemos el siguiente código:



Aunque el código obtenido sigue sin ser del todo válido, si nos fijamos bien veremos que la variable "clave" está inicializada con la cadena de texto "o.setFocus", que resulta ser la contraseña válida. Precisamente su similitud con el propio lenguaje de ActionScript es la que hizo que dicha cadena nos pasara totalmente desapercibida al inspeccionar el fichero en busca de cadenas sospechosas.

Desafío 06: "Autentificación bancaria" *(20 puntos)*

Parece que esta vez dejamos atrás los típicos formularios en los que debíamos acertar la clave. Se nos solicita un valor de una tarjeta de coordenadas que varía cada vez que invocamos el formulario. Como pista, sabemos que el valor que tendremos que introducir consta de dos letras mayúsculas, lo cual ya nos puede ir sugiriendo que no es nada recomendable intentar resolverlo de forma manual (si el alfabeto es de 26 letras, solo tendríamos una entre 676 posibilidades de acertar).

Parece que la opción más sencilla consistiría en fijar una respuesta cualquiera (ej: "AA") y replicar la misma petición POST tantas veces como sea necesario hasta conseguir una respuesta diferente, pero observamos que en dicha petición no solamente se envía la respuesta, sino también las propias coordenadas a las que corresponde. Por tanto, sería más lógico que fijáramos las coordenadas y fuéramos variando la respuesta hasta dar con el resultado correcto (ej: "k3" con "AA", "AB", "AC", etc).

Con esta técnica averiguamos, por ejemplo, que las coordenadas "k3" tienen como respuesta "ER", pero a pesar de haber acertado la respuesta, el sistema detecta que hemos hecho trampa (para ello bastaría con guardarse en el lado servidor la última coordenada enviada y cotejarla con la recibida por parte del cliente, siempre relacionando ambas con la cookie de sesión del usuario). Tal vez una forma de evitar esta comprobación sería la de crear otra cuenta para el reto y lanzar el ataque de fuerza bruta con esta otra cuenta mientras tenemos la nuestra a la espera de saber el resultado, pero lo lógico en un escenario real sería que las tarjetas de coordenadas fueran diferentes para cada usuario y por esa razón ni siquiera lo intenté. La otra opción es solicitar el formulario tantas veces como hiciera falta hasta que nos vuelva a pedir las coordenadas de las que habíamos obtenido su respuesta correcta.

Hecho esto, pasaremos a la segunda fase de autenticación, en la que se nos solicitarán 6 coordenadas que tendremos que acertar de forma simultánea. Para ello no hay más que ir sacándolas una a una tal y como hicimos en la primera fase. Afortunadamente, en esta ocasión la aplicación no detecta que hemos hecho trampa y podemos superar el desafío. De haber estado controlado este intento de fraude, la solución pasaría por obtener el mayor número posible de coordenadas (o la tarjeta completa para asegurarnos al 100% de superar la segunda fase al primer intento).

Desafío 07: "Criptografía: ¿cuál es la frase secreta?" *(15 puntos)*

En este desafío simplemente se nos pide que descifremos el siguiente mensaje: "ZxBPxO - ExzHzLKQBPQ YxIBxOFHRP mxOQV 2009".

La estructura del mensaje tiene toda la pinta de representar una frase que ha sido cifrada con un algoritmo clásico. Nos podemos apoyar en la herramienta Cryptool para analizar el mensaje, empezando por el algoritmo más simple: el cifrado César. Si aplicamos un criptoanálisis básico, la herramienta determina que la clave de cifrado es la "W", descifrando el texto tal y como se muestra a continuación:

```
ZxBPxO - ExzHzLKQBPQ YxIBxOFHRP mxOQV 2009
CaESaR - HacKcONTEST BaLEaRIKUS paRTY 2009
```

A pesar de que el texto resultante es claramente comprensible y lógico, no resulta ser una solución válida, por más que intentemos algunas variaciones (ponerlo entre comillas, todo en mayúsculas o en minúsculas, con o sin la palabra "Caesar" y el guión, etc). Si el algoritmo de cifrado es el correcto (que lo es, evidentemente), solamente nos queda el pequeño detalle de jugar con el alfabeto. Cryptool usa por defecto el alfabeto A-Z (en mayúsculas), respetando las mayúsculas y minúsculas, en la posición en la que aparecen en el texto cifrado. Sin embargo, podemos variar el alfabeto (Opciones / Texto) e incluir otros conjuntos, como los dígitos, las minúsculas, o las puntuaciones, así como los caracteres que queramos.

En nuestro caso bastará con incluir también las minúsculas para obtener una frase mucho más coherente con la escritura normal, que resulta ser la solución válida:

```
ZxBPxO - ExzHzLKQBPQ YxIBxOFHRP mxOQV 2009
Caesar - hackcontest Balearikus Party 2009
```

Desafío 08: "Necesitarás un Linux" *(10 puntos)*

Lo primero que debemos hacer en este caso es descargarnos un fichero ejecutable en formato ELF. Para ver lo que hace, necesitaremos un Linux (de ahí el nombre de la prueba), o bien podemos analizarlo de forma estática desde Windows con herramientas como el IDA Pro (desde el cual también podríamos depurarlo de forma remota en caso de disponer de otra máquina con Linux).

Empezaremos copiando el fichero en un sistema operativo Linux y otorgándole permiso de ejecución (*chmod +x 08*). Si lo ejecutamos tal cual, deberíamos ver un mensaje como este:

```
open: Permission denied
eres root?
```

Sin embargo, si lo ejecutamos como root (*sudo ./08*), aparecerá una frase, tal vez a modo de adivinanza o acertijo que tendremos que resolver (o tal vez no):

```
Tres eran tres, primero las cabezas, luego los cuerpos y por fin los pies...

(Presiona CTRL-C para terminar)
```

El programa se queda a la espera del CTRL-C, aparentemente sin hacer nada más (en realidad sí que hacía algo, pero eso lo explicaré más adelante). Por tanto, optamos por el análisis del ejecutable, ya que nos resulta sospechoso el hecho de que un programa necesite ser ejecutado como *root* simplemente para mostrar un mensaje por la consola (aunque también podría haber sido una prueba facilona de manejo básico de Linux, quién sabe).

Podemos ver las llamadas que se hacen al sistema operativo con el comando strace (*sudo strace ./08*) y comprobaremos que el programa sí que está haciendo algo tras mostrar ese curioso mensaje (acciones que parecen repetirse en un bucle):

```
...
ioctl(3, KDSETLED, 0)                   = 0
ioctl(3, KDSETLED, 0x1)                 = 0
nanosleep({0, 150000000}, NULL)         = 0
ioctl(3, KDSETLED, 0)                   = 0
nanosleep({0, 150000000}, NULL)         = 0
ioctl(3, KDSETLED, 0x1)                 = 0
nanosleep({0, 150000000}, NULL)         = 0
ioctl(3, KDSETLED, 0)                   = 0
nanosleep({0, 150000000}, NULL)         = 0
ioctl(3, KDSETLED, 0x1)                 = 0
nanosleep({0, 150000000}, NULL)         = 0
ioctl(3, KDSETLED, 0)                   = 0
nanosleep({0, 150000000}, NULL)         = 0
nanosleep({0, 500000000}, NULL)         = 0
ioctl(3, KDSETLED, 0x1)                 = 0
nanosleep({0, 500000000}, NULL)         = 0
ioctl(3, KDSETLED, 0)                   = 0
nanosleep({0, 150000000}, NULL)         = 0
ioctl(3, KDSETLED, 0x1)                 = 0
nanosleep({0, 500000000}, NULL)         = 0
ioctl(3, KDSETLED, 0)                   = 0
...
```

Si analizamos lo que hace cada una de estas llamadas, comprobamos que la constante *KDSETLED* se utiliza para establecer el estado de los LEDs del teclado (el valor 1 corresponde a la constante *NUM_SCR*, es decir, el LED de *Scroll Lock*) y la función *nanosleep* se utiliza para hacer una pausa de los nanosegundos especificados en su parámetro. A partir de ahí, es fácil deducir que el LED se enciende y se apaga siguiendo una secuencia bastante característica y conocida: un bucle infinito de mensajes SOS en código Morse (tres parpadeos cortos, tres largos y otros tres cortos, repetidos de forma indefinida).

Tras hacer algunas pruebas sin éxito, logramos superar el desafío al introducir la secuencia tal y como nos la genera el programa, es decir, en Morse y usando como caracteres el guión medio y el punto normal (…---…). Matizo esto porque también se podrían haber usado otros como el punto medio (·), el guión bajo (_) o el largo (–).

En general, creo que esta prueba debería haber sido más sencilla de lo que lo fue para mí, ya que usé el Virtual PC como máquina virtual y su implementación parece ser que no envía las señales correspondientes al teclado físico, por lo que no se encendía ningún LED en mi teclado. De ahí que necesariamente tuviera que optar por la elección de analizar el comportamiento del ejecutable en vez de observar el teclado, que seguramente habría sido la forma más fácil de resolver el desafío.

Desafío 09: "Una imagen vale más que..." *(20 puntos)*

Para superar este desafío tendremos que descubrir la clave que se oculta en una imagen (en formato PNG y escala de grises). Todo indica que estamos ante un nivel de esteganografía y tendremos que averiguar cómo se ha escondido el mensaje para así poder saber lo que se esconde en la imagen o en el fichero.

Tras comprobar que no se esconde ninguna información en la propia imagen, en sus metadatos ni al final del fichero, en principio solamente nos queda abordar el problema desde la perspectiva binaria (técnicas de ocultación en el bit de menor peso, tal vez después de su conversión a mapa de bits, etc). Un análisis visual suele ayudar bastante en esta tarea (separar la imagen en colores RGB, en planos de bits, etc).

Pero hay un pequeño detalle que debería llamarnos la atención en uno de estos pasos de análisis, precisamente visualizando el histograma de colores de la imagen:

Evidentemente, no es nada habitual encontrarse con un histograma así de peculiar en una imagen normal. Ese histograma tiene toda la pinta de representar algo que tenemos muy presente en el día a día: un código de barras EAN-13

Para decodificarlo podemos usar alguna herramienta online (como ZXing) que nos permita subir la imagen capturada (previo recorte de la zona que contiene el código de barras) y así resolveremos esta fase. La clave era "8456234683663".

En este enlace podemos ver cómo se puede aplicar esta misma técnica para esconder otra imagen e incluso alguna que otra palabra (aunque la técnica tiene sus limitaciones por ser un histograma, como es lógico).

Desafío 10: "Nivel de criptografía" *(10 puntos)*

En este nivel se nos pide que resolvamos un enigma: "XTTQUPGNZAIWIA" y como pista tenemos tres letras: "LOL".

Si tomamos como referencia los algoritmos de criptografía clásica, la mayoría de los que utilizan una clave de este tipo, lo hacen reordenando el alfabeto y ubicando la clave al principio. Por ejemplo, usando la palabra "CLAVE" de la siguiente manera:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
CLAVEBDFGHIJKMNOPQRSTUWXYZ
```

Por tanto, a priori parece que no tendría mucho sentido utilizar una clave con tan solo dos letras efectivas, descartando así la mayoría de cifrados clásicos. Sin embargo, consideraremos que la palabra "enigma" no aparece por casualidad, ya que existe un método de cifrado muy conocido con ese nombre (la máquina Enigma), cuya clave también es de tres letras (dependiendo de la implementación concreta).

Si tenemos el algoritmo y también tenemos la clave, ya solamente nos queda el paso de descifrar el mensaje. Para ello podemos usar algún software que implemente la máquina Enigma, como podría ser esta aplicación online desarrollada en Flash (la cual también está incluida en el paquete Cryptool).

El primer paso es el de establecer el valor de los discos o rotores, haciéndolos coincidir con nuestra clave (I = "L", II = "O", III = "L"). Hecho esto, pegamos el mensaje cifrado y obtenemos su correspondiente descifrado: "QGROOLZTHKZFAP"

Tal vez aquí lo más difícil (por lo menos ese fue mi caso) era darse cuenta de que ese texto es directamente la solución, ya que el resultado no es que sea muy inteligible a primera vista. Pero si lo analizamos más a fondo, podemos entrever algunas palabras escritas una forma un tanto peculiar (y no tan raras de ver en según qué jerga), como son: "ROOLZ = rules" y "THKZ = thanks"

Agradecimientos y comentarios

Aunque mi participación en este reto fue bastante posterior a su publicación en la Balearikus Party y a pesar de no optar a los premios (porque eso es lo de menos en este tipo de retos), la experiencia me ha resultado bastante divertida a la vez que didáctica en algunas pruebas, destacando el desafío 5 (por ser más complicado de lo que parecía) y algunos detalles curiosos de las otras por su originalidad y por los diversos intentos de despiste, en los que reconozco haber caído en más de una ocasión. ;-)

Termino dando las gracias a **Miguel Gesteiro** por el desarrollo, mantenimiento y organización del reto y a **RoMaNSoFt** por haberme avisado de la existencia del mismo (de forma bastante convincente, picándome para que resolviera el desafío 5) y también por haberme resuelto alguna que otra duda con la que me encontré en alguna fase.

Saludos,

Daniel Kachakil
dani@kachakil.com