Solución al Reto 6 de S21sec

Octubre 2008 © Daniel Kachakil

Introducción

Este documento describe una solución al Reto 6 de S21sec, publicado el 13 de octubre de 2008 en la siguiente dirección web:

```
http://blog.s21sec.com/2008/10/reto-6.html
```

Fase 1

Después de leer con mucha atención el texto que aparece en el post y buscar algún detalle extraño en todos los elementos visibles, nos damos cuenta de que nos falta algo para ponernos manos a la obra… ¿Dónde está el reto?

Tras echarle un vistazo al código fuente de la página, nos damos cuenta de que existe un bloque símbolos encabezado por un texto y comentado en HTML, que se encuentra justo al final del mensaje que leíamos en el post:

```
<!-- Hola, sí, el reto está aquí: Venga, 3,1,2, a ello!!!!:
+>>++++++++++++++++++++++++++++++++[+++<+]>-><.+>++++++++++          ( ... )
++++++++++<+[>++<-]>+.+>++++++++++++++++++++++++[<-+>.]<  -->
```

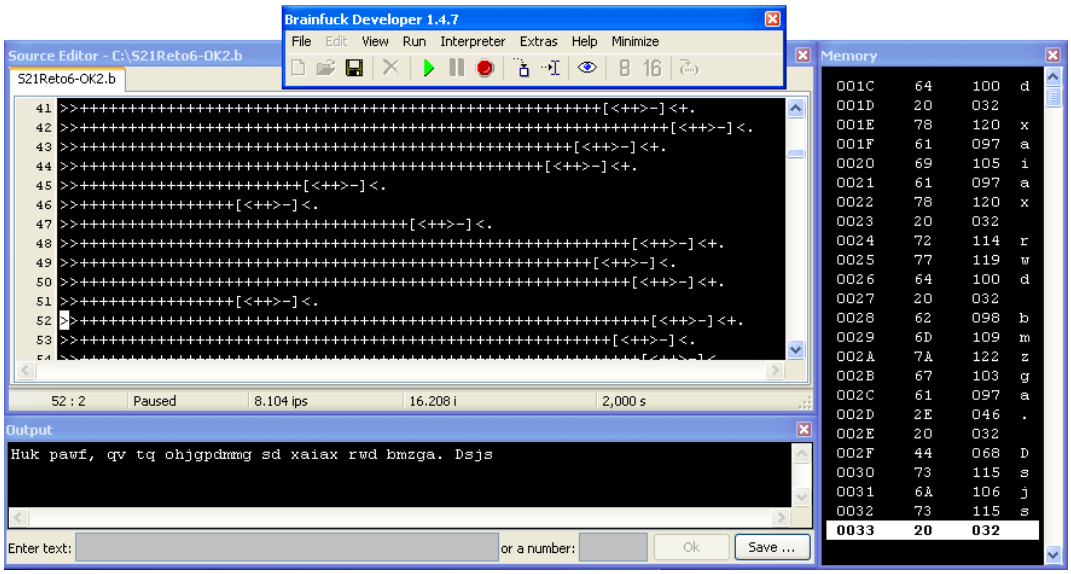
Extraemos el bloque de símbolos al completo, eliminando las marcas HTML que indican el principio y el fin del comentario, así como el texto inicial (que tomaremos como posible pista). Dicho bloque está formado por 18.087 caracteres, combinando únicamente 7 símbolos distintos: + – . < > [ ]

Cuando me enteré de la existencia del reto ya habían sido publicados varios comentarios que indicaban a qué correspondía este curioso código. De todas formas, voy a suponer que partimos de cero, porque a priori podría resultar bastante complicado llegar a esa conclusión buscando información en Internet. Precisamente esos símbolos suelen usarse como operadores o caracteres especiales en los buscadores como Google y tampoco encontraremos resultados útiles si escribimos los nombres de los símbolos como texto (en inglés tampoco). Sin embargo, simplemente intuyendo que se podía tratar de algún tipo de código, podemos usar un buscador específico e introducir cualquier parte de la cadena de símbolos (por ejemplo "[>++<-]"), a ver si hay suerte:

```
http://www.google.com/codesearch?q=%22%5B%3E%2B%2B%3C-%5D%22
```

Sí, como no podía ser de otra forma, la respuesta también estaba en Google. Concretamente hemos lanzado la búsqueda en Google Code Search, un buscador de código fuente público, que en este caso localiza nuestro fragmento aunque lo identifica (erróneamente) como Limbo o como Scheme, tal y como se puede observar en la siguiente captura que muestra los primeros resultados.

Sin embargo, observando con detalle los resultados y el contexto en el que se encuentran en su origen, podemos llegar a determinar que se trataba de Brainfuck, un curioso lenguaje de solo 8 instrucciones inspirado en el modelo teórico de la máquina de Turing, que a su vez es capaz de modelar cualquier algoritmo. La cuestión es que parece que vamos a necesitar algún intérprete o compilador para Brainfuck y si lo encontramos con algún un depurador, mejor aún, ya que así podremos ver lo que está ocurriendo paso a paso. Por ejemplo, podemos optar por Brainfuck Developer, un programa gratuito cuya funcionalidad nos resultará más que suficiente.



Si cargamos el programa y lo ejecutamos sin más, veremos que no hace nada útil y que su ejecución parece no terminar nunca. Sin entrar en muchos detalles sobre este tema, nos será de utilidad conocer un poco la sintaxis de Brainfuck, ya que este reto lo requiere (así sabremos si vale la pena esperar, o si estamos perdiendo el tiempo):

- El signo ***más***, incrementa en una unidad el valor de la posición actual de memoria, mientras que el signo ***menos*** la decrementa.
- El ***mayor que*** avanza el puntero a la siguiente posición de memoria, mientras que el ***menor que*** lo hace retroceder a la posición anterior.
- Los ***corchetes*** indican el comienzo y el fin de un bucle, de forma que si el valor de la posición actual es cero, no se entra (o se sale) del mismo.
- El ***punto*** imprime el valor de la posición actual, como carácter ASCII.

Analizando el comienzo de nuestro algoritmo, vemos que el primer bucle que aparece ([+++<+]) tiene toda la pinta de ser infinito, puesto que siempre suma y por tanto nunca llegará a una posición cuyo valor sea cero. Sin embargo, esta suposición no es del todo correcta, ya que al superar el valor máximo (255 si trabajamos con 8 bits), el valor actual volvería a empezar desde cero y el bucle terminaría su ejecución, pero en todo caso no se da esta circunstancia en nuestro bucle, ya que no se está sumando una unidad y no pasa por cero al comprobarlo, por lo que el bucle sí que resulta ser infinito.

Podemos pensar que es alguna una especie de trampa, así que intentamos eliminar el bucle infinito, para ver si el programa continúa su ejecución con normalidad, pero tampoco conseguimos nada, ya que se engancha en el siguiente bucle y así sucesivamente. Parece que ha llegado el momento de echar mano de alguna pista y lo único extraño que tenemos es esa "cuenta atrás" (3,1,2) y tal vez en el propio texto del post, la frase en la que aparece "si se tercia" (una de las acepciones de terciar es la de dividir algo en tres partes)

La clave para superar esta fase estaba en reordenar el programa, ya que esos números nos estaban indicando que el fichero está permutado en grupos de tres elementos. El tercer elemento del código original aparecerá en la primera posición del permutado, mientras que el primero aparecerá en la segunda posición y el segundo en la tercera. Veámoslo esquemáticamente:

Original:	1	2	3	4	5	6	7	8	9	10	11	12	…
Permutado:	3	1	2	6	4	5	9	7	8	12	10	11	…

Para obtener el fichero original a partir del permutado, basta con recorrerlo de tres en tres posiciones y permutarlo utilizando este sencillo algoritmo (siendo *Entrada* el vector de caracteres o fichero del que ya disponemos y *Salida* el resultante):

```
Mientras i < Longitud(Entrada)
       Salida[i]   = Entrada[i+1]
       Salida[i+1] = Entrada[i+2]
       Salida[i+2] = Entrada[i]
       i = i + 3
```

Una vez reordenado el fichero, obtendremos un código con una estructura mucho más coherente y a su vez fácil de entender e interpretar, puesto que para conseguir un determinado valor, se almacena su mitad en una posición de memoria y se duplica este valor dentro del bucle. En caso de necesitar un valor impar, se le suma una unidad al final. Finalmente se imprime por pantalla y se avanza al siguiente carácter.

```
>>++++++++++++++++++++++++++++++++++[<++>-]<.
>>++++++++++++++++++++++++++++++++++++++++++++++++++++++++++[<++>-]<+.
>>++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++[<++>-]<+.
>>++++++++++++++++[<++>-]<.
( Continúa... )
```

Si ahora ejecutamos el programa resultante, su ejecución termina de forma casi instantánea y obtenemos como resultado el siguiente texto ininteligible:

```
Huk pawf, qv tq ohjgpdmmg sd xaiax rwd bmzga. Dsjs wntm hwjuwma rokw vwwee
rwkusmgmflw wd nisiawflz fuqzwjg, yex emw qs yiedgfwk ga ocfljsnezws vw
vzsowxjsvj.
hmqzwlwoebshmfljpgblgtsmrmpsjjsrwidmflgnvqwflamioesuhmfooocetsjmapskusjbaepsjjsme
fckwakbuucftsbjggwgftseorokwljzsbiflgrdp Niwfs kpedhw
```

Fase 2

Asumiendo que hemos completado con éxito la primera fase, suponemos que hemos obtenido un texto cifrado y que, por tanto, tendremos que descifrarlo. Podemos empezar a probar con el cifrado César, ya que es tan simple que resulta bastante trivial probar las 25 posibles soluciones, pero seguimos sin obtener nada en claro.

El siguiente algoritmo a probar podría ser el cifrado de Vigenère, basado en aplicar el cifrado César pero utilizando diferentes sustituciones para cada posición, en lugar de utilizar siempre la misma. En este tipo de cifrado se elige una clave que sirve para indicar el desplazamiento de cada una de las posiciones. A pesar de ser más robusto, este cifrado sigue siendo vulnerable al criptoanálisis basado en el método de Kasiski o en el análisis del índice de coincidencia para determinar la longitud de la clave de cifrado, de forma que posteriormente se puede aplicar el correspondiente análisis de frecuencias a cada uno de los bloques de cifrado, como en el César.

Por no reinventar la rueda, buscaremos un programa que implemente estas técnicas (y si es online, seguramente la tarea nos resultará aún más cómoda):

```
http://smurfoncrack.com/pygenere/
```

Tras introducir el texto cifrado y seleccionar el idioma adecuado (que en este caso era el español) obtenemos un texto descifrado que ya se aproxima mucho al original, aunque aún no llega a ser exacto. Este algoritmo calcula que la contraseña podría ser **"VAMOSJSE"**, así que ahora que podemos deducir prácticamente todas las palabras, nos resultará muy fácil ajustar las letras erróneas y obtener la contraseña correcta, que en este caso era **"VAMOSSSS"**, descifrando el texto completo correctamente (por ejemplo, usando este script disponible también online)

```
Muy bien, ya te aproximas al final del juego. Para esta tercera fase debes
descargarte el siguiente fichero, del que ya dispones la contrasenia de
descifrado.
hachetetepepuntopuntobarrabarrawwwpuntosveintiunosecpuntocombarradescargasbarrare
toseisguionbajoguionbajofasetrespuntozip Buena suerte
```

De ahí obtenemos la URL desde la que podremos descargarnos un fichero para la tercera y última fase del reto:

```
http://www.s21sec.com/descargas/reto6__fase3.zip
```

Fase 3

Para comenzar con esta fase descomprimiremos el fichero ZIP descargado en el último paso de la fase anterior, teniendo en cuenta que la contraseña que lo protegía era la misma que obtuvimos para el cifrado de Vigenère, toda ella en mayúsculas (VAMOSSSS). No vemos ningún comentario en el fichero ZIP y el nombre ("fichero"), así como la extensión inexistente del único fichero que encontramos tampoco nos ayuda en nada.

Si analizamos el contenido del fichero, nos encontramos con una cabecera conocida (GIF89a), según la cual deberíamos estar ante una imagen en formato GIF. Sin embargo, por más que intentamos abrir el fichero con algún visor o editor gráfico, no conseguimos visualizar nada, ya que no se reconoce el formato en ningún caso.

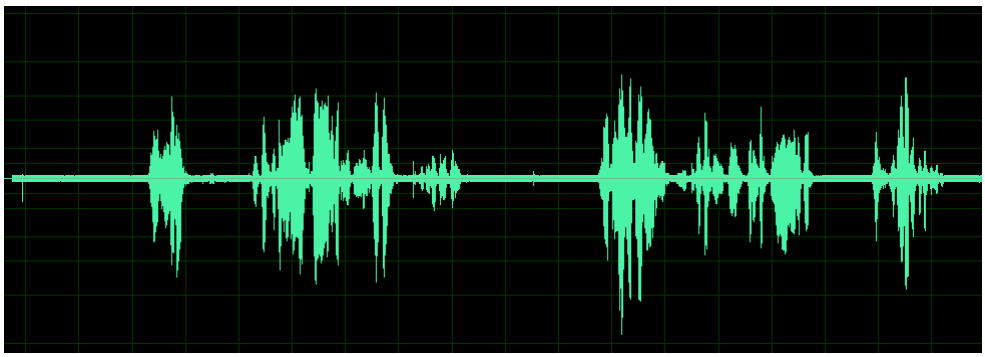
Según las especificaciones del formato GIF, los dos siguientes bytes (02 0E) indican la anchura de la imagen, mientras que los dos siguientes (0B 00) indican la altura. Por tanto, en nuestro caso se supone que tenemos una imagen cuyas dimensiones tendrían que ser de 3586 x 11 píxeles, pero un fichero de esas características no debería superar los 20 kilobytes, aunque nuestro fichero casi llega a los 300. Por otro lado, no termina en punto y coma, ni podemos reconocer los bloques que define el formato, etc.

También llama la atención el hecho de que aparezca la palabra "hola" en la posición 50 y que aparezca "adios" justo al final del fichero, así que intentamos extraer el contenido de ese bloque, ignorando la cabecera GIF, pero este nuevo bloque tampoco parece tener ningún encabezado reconocible.

Si buscamos en Google algún fragmento del fichero, tal vez tengamos suerte y encontremos otros ficheros similares que también contengan el mismo fragmento y así podremos tener alguna pista acerca del formato real. Al no tener muchos caracteres imprimibles, buscaremos por ejemplo una cadena que se repite bastante al inicio del fichero: "ÿÿÿÿÿÿ" (que coincide con los valores FF FF FF FF FF FF). Probamos con varios tipos de ficheros (PDF, PIC, EMZ, etc) hasta que llegamos a un resultado que aparece en la segunda página de resultados y que nos puede llamar la atención: un fichero RAW (audio en formato PCM, sin comprimir)

Si seguimos analizando el contenido del fichero, nos daremos cuenta de que casi todas las posiciones de offsets impares tienen un valor de 00 o de FF (o valores muy cercanos a estos), lo que nos puede llevar a pensar que se trata de valores de 16 bits, probablemente con signo. Por otro lado, entre un valor (16 bits) y el siguiente no suele haber una diferencia muy grande. Si además tenemos en cuenta que había una pista que nos hablaba de Bach, Vivaldi y Mozart, entonces no sería tan extraño encontrarnos con un fichero de audio.

Con la extensión "WAV" no tenemos suerte, así que intentaremos con la extensión "RAW", que no tiene ninguna cabecera, pero nos obligará a especificar el formato con el que queremos interpretar esos datos. Podemos usar cualquier programa como Adobe Audition (comercial, pero tiene versión de prueba) y especificar el formato adecuado: 8 kHz, 16 bit, mono, formato Motorola (MSB, LSB). Al no tener cabecera, la única forma de ajustar estos valores es con el método de prueba y error.



Simplemente con echarle un vistazo a la onda resultante podemos ver claramente que se trata de audio, concretamente de una pequeña grabación de voz en la que se nos agradece la participación en el reto y se nos indica la palabra clave para completarlo.

Agradecimientos y comentarios

En el fondo los pasos a seguir eran sencillos, aunque su resolución dependía en gran medida de la suerte que tuviéramos con la interpretación de la primera pista y en no dejarnos llevar por la cabecera GIF de la última fase, que supongo que estaba ahí para despistar más que para ayudar.

Por último, me gustaría terminar este documento dando las gracias a Mikel Gastesi por este reto (y por supuesto a todo aquél que haya colaborado en sus ideas o desarrollo). Ha sido el primero de S21sec en el que participo y os aseguro que por mi parte no será el último… ¡Hasta el próximo! ;-)

Saludos,

Daniel Kachakil
dani@kachakil.com