Trabajo práctico N°1

Etcheverri Franco, *Padrón Nro. 95.812*
franverri@hotmail.com
Vicari Dario, *Padrón Nro. 86.559*
dariovicari@yahoo.com.ar
Gonzalez Esteban, *Padrón Nro. 54.476*
egonza@fi.uba.ar

1er. Cuatrimestre de 2016
66.20 Organización de Computadoras
Facultad de Ingeniería, Universidad de Buenos Aires

1. Introducción

El trabajo práctico número uno consiste en el análisis de cuatro programas escritos en C que poseen ciertas vulnerabilidades por su forma en que están escritos para lograr .exˆplotarlasˆ lograr que el programa se comporte como nosotros querramos.

Básicamente lo que se busca es que cada programa imprima por consola el mensaje de ˆou win!ˆcuando durante el curso nomarl de ejecución del mismo no se llegaría a dicha impresión en la pantalla.

Si bien cada uno de los programas presenta características particulares que serán detlladas luego, todos ellos presenan problemas de seguridad debido a la funcion "gets"de C que no limita la cantidad de caracteres que van a ser leidos y guardados, es por ellos que si se ingresa una cantidad mayor a la que se debería, se van a sobreescribir datos en direcciones de memoria que, a priori, no deberían.

2. Desarrollo

Para una mejor organización y comprensión del trabajo efectuado para llevar adelante el trabajo, se dvidira esta sección en 4 (una para cada programa de C a resolver).

2.1. stack1.c

El primer programa es el más simple de los cuatro debido a que para lograr que se imprima por consola el mensaje ˆou win!"debemos conseguir que la condición del if se cumpla: **if (cookie == 0x41424344)**

Los caracteres involucrados son los siguientes:

- **0x41:** 'A'
- **0x42:** 'B'
- **0x43:** 'C'
- **0x44:** 'D'

Como podemos observar todos ellos son imprimibles y nos permite ingresarlos por consola cuando el programa lo solicite.

2.1.1. Analisis del stack

AGREGAR LA IMAGEN DEL STACK

2.1.2. Diseño de la entrada a proporcionar al programa

Como ya se explico anteriormente, la variable cookie debe contener los caracteres "ABCD". Adicionalmente habrá que llenar el buffer con 80 caracteres cualesquiera ya que no va a influir en el resultado final del programa.

Por lo tanto el string a ingresar va a ser de la siguiente forma: "xxxxxxx...xxxxDCBA". Donde las 'x' hacen referencia a los 80 caracteres. Un aspecto importante a tener en cuenta es el motivo por el cual se invirtió el orden de las últimas cuatro

letras. Esto se debe a que nos encontramos insertando caracteres directamente en memoria por lo que debemos hacerlo acorde al endianess. Una forma práctica de darnos cuenta de cómo debemos ingresar los caracteres es haciendo la prueba con “ABCD” y viendo que el programa no se comporta como esperabamos.

Si añadimos un breakpoint justo en la linea anterior a la comprobación de la variable “cookie”, ejecutamos el programa hasta llegar a ese punto y luego vemos el contenido de la variable cookie, podemos notar que es: “0x44434241”. Por lo tanto inferimos que la forma correcta de ingresar los datos va a ser invirtiendo el orden de los mismos.

2.1.3. Corridas de prueba

AGREGAR IMAGEN DE LA CORRIDA

En la imagen se pueden observar ambas corridas de las mencionadas anteriormente, una utilizando el string “xxxxxxx...xxxxDCBA” y otra el “xxxxxxx...xxxxABCD” que comprueban que la correcta es la primera.

2.2. stack2.c

La particularidad que se presenta en este programa, a diferencia del primero, es que no todos los caracteres que necesitamos para que se verifique la condición del if son imprimibles: **if (cookie == 0x01020305)**

- **0x01:** ‘SOH’ (Inicio de encabezado)
- **0x02:** ‘STX’ (Inicio de texto)
- **0x03:** ‘ETX’ (Fin de texto)
- **0x05:** ‘ENQ’ (Consulta)

2.2.1. Analisis del stack

AGREGAR LA IMAGEN DEL STACK

2.2.2. Diseño de la entrada a proporcionar al programa

Para lograr el objetivo en este caso no vamos a poder ingresar los caracteres por teclado directamente en la consola al ser no imprimibles. Por lo tanto debemos realizar un programa que prepare el string a ingresar al **stack2**. Para ello debemos ingresar los valores deseados en la función ”getsçomo si fueran con el teclado, pero desde el exploit (Programa o código que explota una vulnerabilidad del sistema o de parte de él). Para hacerlo necesitamos usar pipes (tuberías), que nos servirán para redireccionar la salida (stdout) del proceso padre (exploit) a la entrada (stdin) del proceso hijo (stack2).

2.2.3. Corridas de prueba

AGREGAR IMAGEN DE LA CORRIDA

En la imagen se puede observar que al ejecutar el exploit, este se encarga de ejecutar el programa stack2 con las entradas necesarias para lograr el mensaje de “you win”.

2.3. stack3.c

Este programa es similar al stack2, ya que la variable "cookie" posee caracteres no imprimibles (**if (cookie == 0x01020005)**). La única diferencia es el caracter **0x01:** 'NUL' (Caracter nulo). Para nuestros fines prácticos esto no va a modificar la forma de resolución del enunciado, ya que la funcion "gets" como podemos ver en la descripción de su funcionamiento, no se ve afectada por el caracter nulo:

"**gets()** reads a line from stdin into the buffer pointed to by s until either a terminating newline or EOF, which it replaces with a null byte. No check for buffer overrun is performed"

2.3.1. Analisis del stack

AGREGAR LA IMAGEN DEL STACK

2.3.2. Diseño de la entrada a proporcionar al programa

La forma en que plantearemos este stack va a ser igual al anterior, por lo que también vamos a generar un programa exploit que se encargara de llamar a stack2 insertandole por entrada estandar los caracteres deseados.

2.3.3. Corridas de prueba

AGREGAR IMAGEN DE LA CORRIDA

En la imagen se puede observar que al ejecutar el exploit, este se encarga de ejecutar el programa stack3 con las entradas necesarias para lograr el mensaje de "you win".

2.4. stack4.c

Por último, en el stack4, se presenta una particularidad que va a necesitar un análisis más detallado para lograr explotar la vulnerabilidad del programa. la variable "cookie" nuevamente posee caracteres no imprimibles (**if (cookie == 0x000d0a00)**):

- **0x0d:** 'CR' (Retorno de carro)
- **0x0a:** 'LF' (Nueva linea)
- **0x00:** 'NUL' (Caracter nulo)

La secuencia CR LF era común en los primeros ordenadores que tenían máquinas de teletipo (como el ASR33) como dispositivo de terminal. Esta secuencia era necesaria para posicionar el cabezal de la impresora al principio de una nueva línea. Como esta operación no se podía hacer en tiempo "1 carácter", había que dividirla en dos caracteres. A veces era necesario enviar CR LF NUL (siendo NUL el carácter de control que le manda "no hacer nada"), para asegurarse de que el cabezal de impresión parara de moverse. Después de que estos sistemas mecánicos desaparecieran, la secuencia CR LF dejó de tener sentido, pero aun así se ha seguido usando.

El caracter de fin de linea va a hacer que la función **gets** deje de leer los caracteres que le preceden, por lo que si intentamos resolver el enunciado del stack4 al igual que lo hicimos con el stack2 y el stack3 no vamos a poder.

2.4.1. Analisis del stack

AGREGAR LA IMAGEN DEL STACK

2.4.2. Diseño de la entrada a proporcionar al programa

Para este caso vamos a tener que enfocarnos en las direcciones de memoria de las instrucciones del programa. Lo que vamos a hacer sera modificar el contenido de la dirección de retorno para que en lugar de retornar al sitio desde donde se llamo el programa, nos lleve el puntero de ejecución al "printf" y de esta manera conseguiremos que nos imprima por consola la cadena "you win!", aun sin cumplir la condición del if.

En primer lugar vamos a ver en que posición de memoria se encuentra la instrucción del "printf". Para ello vamos a utilizar la herramienta GDB y el comando "disas stack4" para obtener dicha información:

```
(gdb) disas stack4
Dump of assembler code for function stack4:
0x400ba0 <stack4>:      lui     gp,0xfc0
0x400ba4 <stack4+4>:    addiu   gp,gp,29840
0x400ba8 <stack4+8>:    addu    gp,gp,t9
0x400bac <stack4+12>:   addiu   sp,sp,-128
0x400bb0 <stack4+16>:   sw      gp,16(sp)
0x400bb4 <stack4+20>:   sw      ra,120(sp)
0x400bb8 <stack4+24>:   sw      s8,116(sp)
0x400bbc <stack4+28>:   sw      gp,112(sp)
0x400bc0 <stack4+32>:   move    s8,sp
0x400bc4 <stack4+36>:   addiu   v0,s8,104
0x400bc8 <stack4+40>:   lw      a0,-32744(gp)
0x400bcc <stack4+44>:   nop
0x400bd0 <stack4+48>:   addiu   a0,a0,3680
0x400bd4 <stack4+52>:   addiu   a1,s8,24
0x400bd8 <stack4+56>:   move    a2,v0
0x400bdc <stack4+60>:   lw      t9,-32664(gp)
0x400be0 <stack4+64>:   nop
0x400be4 <stack4+68>:   jalr    t9
0x400be8 <stack4+72>:   nop
0x400bec <stack4+76>:   lw      gp,16(s8)
0x400bf0 <stack4+80>:   addiu   a0,s8,24
0x400bf4 <stack4+84>:   lw      t9,-32672(gp)
---Type <return> to continue, or q <return> to quit---return
0x400bf8 <stack4+88>:   nop
0x400bfc <stack4+92>:   jalr    t9
0x400c00 <stack4+96>:   nop
0x400c04 <stack4+100>:  lw      gp,16(s8)
0x400c08 <stack4+104>:  nop
```

```
0x400c0c <stack4+108>:  lw      a0,-32744(gp)
0x400c10 <stack4+112>:  nop
0x400c14 <stack4+116>:  addiu   a0,a0,3704
0x400c18 <stack4+120>:  lw      a1,104(s8)
0x400c1c <stack4+124>:  lw      t9,-32664(gp)
0x400c20 <stack4+128>:  nop
0x400c24 <stack4+132>:  jalr    t9
0x400c28 <stack4+136>:  nop
0x400c2c <stack4+140>:  lw      gp,16(s8)
0x400c30 <stack4+144>:  lw      v1,104(s8)
0x400c34 <stack4+148>:  lui     v0,0xd
0x400c38 <stack4+152>:  ori     v0,v0,0xa00
0x400c3c <stack4+156>:  bne     v1,v0,0x400c64 <stack4+196>
0x400c40 <stack4+160>:  nop
0x400c44 <stack4+164>:  lw      a0,-32744(gp)
0x400c48 <stack4+168>:  nop
0x400c4c <stack4+172>:  addiu   a0,a0,3720
0x400c50 <stack4+176>:  lw      t9,-32664(gp)
```

Podemos observar que en la dirección **0x400c3c** se realiza la comparación del if, por lo que nuestra dirección de retorno va a tener que ser la siguiente: **0x400c40** que particularmente no hace nada pero las que le preceden se encargaran de imprimir el mensaje deseado.

Una forma de conseguir que la dirección de retorno sea sobreescrita es llenando el buffer con la dirección deseada para producir el desborde y que llegue hasta el RA. Para ello se va generar un ciclo con un tamaño grande para que sobreescriba más allá de la dirección de la variable **cookie**.

2.4.3. Corridas de prueba

AGREGAR IMAGEN DE LA CORRIDA

En la imagen se puede observar que si bien el valor de la variable cookie no coincide con el esperado por el if, igualmente se ejecuta la instrucción del printf logrando el mensaje de "you win!".

2.5. Codificación

Para pasar de cualquier archivo a uno en base 64, es decir, codificar la información, utilizamos una funcion llamada b64 la cual recibe los siguientes parámetros:

"static int b64(char opt, char *infilename, char *outfilename, int linesize)"

- **opt:** indica si debe codificarse ('e') o decodificarse ('d') el archivo.
- **infilename:** contiene el nombre del archivo origen que va a ser procesado.
- **outfilename:** contiene el nombre del archivo destino donde se van a guardar los datos luego de realizar el codificado o decodificado.

Lo que realiza esta función básicamente es ir leyendo caracteres del archivo origen y guardandolos en un buffer múltiplo de 3 (el que usaremos va a ser exactamente de 3 chars es decir 24 bites) y cuando el buffer se encuentra lleno codificarlo a b64 dividiendolo en 4 elementos, cada uno de ellos de 6 bites.

Esto hace que cualquier combinacion de unos y ceros vaya a parar a un elemento dentro de la tabla de caracteres permitidos por la codificación en base 64. Por último, estos nuevos caracteres codificados van a ser escritos en el nuevo archivo destino.

Cabe destacar que, en el caso de que se llegue al fin de archivo antes de llenar el buffer nuevamente, se procedera a completarlo con caracteres '=' para lograr que sea múltiplo de 3 y poder codificarlo.

Un aspecto interesante de la implementación es el codificado del buffer de 24 bits. Para ellos se van a utilizar variables del tipo char y operadores logicos para acomodar la posición de los bits significativos. Van a ocurrir tres posibilidades diferentes en este buffer:

- **Primer elemento:** al obtener el primer elemento del buffer de caracteres obtendríamos 8 bits, de los cuales sólo nos interesan los 6 primeros. Por lo tanto al aplicar un desplazamiento lógico de dos lugares hacia la derecha de esa variable donde alojamos el caracter nos preveería el efecto deseado, ya que los dos bits mas significativos quedarían en cero, que es equivalente a darle importancia únicamente a los 6 bits restantes.

- **Segundo elemento:** para poder conformar el segundo elemento del archivo codificado en base 64 necesitaríamos los 2 bits que no se utilizaron del primer char para el elemento anterior y 4 bits más correspondientes al segundo char del buffer. Para ello se va a realizar operaciones logicas de and y de desplazamiento. La forma en que opera el programa se puede observar en la Figura 1 para no tener que explicar en palabras como se procesa cada elemento lo cual hace más dificil la comprensión.

2.6. Deodificación

Para el decodificado, es decir, a partir de un archivo en base 64 transformalo a un archivo que acepte todos los caracteres de la tabla ASCII, se utiliza la misma función pero pasandole como parámetro el caracter 'd' en la opcion .ºpt".

El procedimiento es similar al de la codificación pero ahora se utiliza un buffer con 4 elementos para decodificar, ya que para poder obtener 3 caracteres formados por 8 bits necesito 4 de 6 bits. Por lo que las operaciones para cada paso son las mostradas en la Figura 2

3. Conclusiones

Dada la finalidad del trabajo práctico, que consistía en familiarizarse con las herramientas de software que usaremos en los siguientes trabajos, implementando un programa (y su correspondiente documentación) que resuelva el problema, podemos concluir que se logró dicho objetivo al aprender a utilizar herramientas que hasta el momento no manejabamos.

	Char								Operación	Char							
1	a7	a6	a5	a4	a3	a2	a1	a0	Desplazo 2 a derecha	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]
	X	X	X	X	X	X	-	-		0	0	a7	a6	a5	a4	a3	a2
2	a7	a6	a5	a4	a3	a2	a1	a0	and con 00000011 y	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]
	-	-	-	-	-	-	X	X	desplazo 4 lugares a la	0	0	b1	b0	0	0	0	0
	b7	b6	b5	b4	b3	b2	b1	b0	and con 11110000 y desplazo 4 lugares a la derecha	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]
	X	X	X	X	-	-	-	-		0	0	0	0	c7	c6	c5	c4
	or entre ambos resultados y me queda:									0	0	b1	b0	c7	c6	c5	c4
3	b7	b6	b5	b4	b3	b2	b1	b0	and con 00001111 y	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]
	-	-	-	-	X	X	X	X	desplazo 2 lugares a la	0	0	c3	c2	c1	c0	0	0
	c7	c6	c5	c4	c3	c2	c1	c0	and con 110000 y desplazo 6	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]
	X	X	-	-	-	-	-	-	lugares a la derecha	0	0	0	0	0	0	d7	d6
	or entre ambos resultados y me queda:									0	0	c3	c2	c1	c0	d7	d6
4	c7	c6	c5	c4	c3	c2	c1	c0	and con 00111111	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]
	-	-	X	X	X	X	X	X		0	0	c5	c4	c3	c2	c1	c0

Figura 1: Codificación de elementos

	Char								Operación	Char							
1	a7	a6	a5	a4	a3	a2	a1	a0	Desplazo 2 a izquierda	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]
	0	0	X	X	X	X	X	X		a5	a4	a3	a2	a1	a0	0	0
	b7	b6	b5	b4	b3	b2	b1	b0	Desplazo 4 a derecha	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]
	0	0	X	X	-	-	-	-		0	0	0	0	0	0	b5	b4
	or entre ambos resultados y me queda:									a5	a4	a3	a2	a1	a0	b5	b4
2	b7	b6	b5	b4	b3	b2	b1	b0	Desplazo 4 a izquierda	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]
	0	0	-	-	X	X	X	X		b3	b2	b1	b0	0	0	0	0
	c7	c6	c5	c4	c3	c2	c1	c0	Desplazo 2 a derecha	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]
	0	0	X	X	X	X	-	-		0	0	0	0	c5	c4	c3	c2
	or entre ambos resultados y me queda:									b3	b2	b1	b0	c5	c4	c3	c2
2	c7	c6	c5	c4	c3	c2	c1	c0	Desplazo 6 a izquierda	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]
	0	0	-	-	-	-	X	X		c1	c0	0	0	0	0	0	0
	d7	d6	d5	d4	d3	d2	d1	d0		[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]
	0	0	X	X	X	X	X	X		0	0	d5	d4	d3	d2	d1	d0
	or entre ambos resultados y me queda:									c1	c0	d5	d4	d3	d2	d1	d0

Figura 2: Decodificación de elementos

Por otra parte, por el lado del programa escrito en C propiamente hablando, la dificultad fue acorde al propósito mencionado anteriormente, ya que si bien nos llevó su tiempo pensarlo y realizarlo, no fue excesivo y nos permitió enfocarnos en le correcta utilización de las herramientas complementarías y la elaboración de un informe adaptándonos a las normas pedidas por el curso.

4. Código en C++

4.1. Main.c

```
#include <stdio.h>
#include "base64.h"
#include "constantes.h"
#include "funciones.c"


int main(int argc, char **argv){

   int iError = 0;
   int iIndexARG = 0;
   char cModo = ValidarArgumentos(argc,argv);
   switch(cModo){
         case MODO_DECODIFICADOR:
            iIndexARG = 1;
    case MODO_CODIFICADOR:
            iError = b64(cModo,argv[iIndexARG+2],argv[iIndexARG+4],B64_DEF_LINE_SIZE);
    if(iError==0){
               puts(MSG_OK);
            }else{
               puts(b64_msgs[iError]);
            }
    break;
    case MODO_AYUDA:
    puts(MSG_HELP);
    break;
    case MODO_VERSION:
    puts(MSG_VERSION);
    break;
    default:
    puts("error");
    break;

   }
   return 0;
}
```

4.2. Funciones.c

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

```
int ExisteArchivo(char* NombreArchivo){
int iReturn = -1;
if( access( NombreArchivo, F_OK ) != -1 ){
     iReturn = 0;
}
return iReturn;
}



/*
Esta funcion tiene como objetivo validar los parametros con las cuales se invoca el progra
del programa a la entrada.
*/


char ValidarArgumentos(int argc, char **argv){

char cOutput = 'E';
DEBUG == 1 ? printf("argc es %d \n",argc) : printf("\n") ;

if(argc <  CONST_ARGC_MIN || argc > CONST_ARGC_MAX )
{
    puts(MSG_ERR_ARGS);
    puts(MSG_HELP);
}else{

if(strcmp(argv[1],CONST_ARG_HELP) == 0 ){
cOutput = MODO_AYUDA;
}else if(strcmp(argv[1],CONST_ARG_VERSION) == 0){
cOutput = MODO_VERSION;
}else{

if(strcmp(argv[1],CONST_ARG_DECODE) == 0) //Modo Decodificador
{
if(strcmp(argv[2],CONST_ARG_INPUT_NAME) == 0 && strcmp(argv[4],CONST_ARG_OUTPUT_NAME) == 0
if(ExisteArchivo(argv[3]) == 0)
{
cOutput = MODO_DECODIFICADOR;
}else{
puts(MSG_NO_FILE);
}
}else{
    puts(MSG_NO_ARGS);
    puts(MSG_HELP);
}

}else{ //Modo Codificador
if(strcmp(argv[1],CONST_ARG_INPUT_NAME) == 0 && strcmp(argv[3],CONST_ARG_OUTPUT_NAME) == 0
if(ExisteArchivo(argv[2]) == 0)
```

```
{
cOutput = MODO_CODIFICADOR;
}else{
puts(MSG_NO_FILE);
}
}else{
    puts(MSG_NO_ARGS);
    puts(MSG_HELP);
}

}

}
}
return cOutput;
}

int ValidarArchivo(char* NombreArchivo){
int iReturn = -1;
if( access( NombreArchivo, F_OK ) != -1 ){
     iReturn = 0;
}
return iReturn;
}
```

4.3. Base64.h

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

static const char cb64[]="ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/

static const char cd64[]="|$$$}rstuvwxyz{$$$$$$$>?@ABCDEFGHIJKLMNOPQRSTUVW$$$$$$XYZ[\\]^_`



#define B64_DEF_LINE_SIZE   72
#define B64_MIN_LINE_SIZE    4
#define B64_SYNTAX_ERROR        1
#define B64_FILE_ERROR          2
#define B64_FILE_IO_ERROR       3
#define B64_ERROR_OUT_CLOSE     4
#define B64_LINE_SIZE_TO_MIN    5
#define B64_SYNTAX_TOOMANYARGS  6

#define B64_MAX_MESSAGES 7
static char *b64_msgs[ B64_MAX_MESSAGES ] = {
            "b64:000:Invalid Message Code.",
```

```
            "b64:001:Syntax Error -- check help (-h) for usage.",
            "b64:002:File Error Opening/Creating Files.",
            "b64:003:File I/O Error -- Note: output file not removed.",
            "b64:004:Error on output file close.",
            "b64:005:linesize set to minimum.",
            "b64:006:Syntax: Too many arguments."
};

#define b64_message( ec ) ((ec > 0 && ec < B64_MAX_MESSAGES ) ? b64_msgs[ ec ] : b64_msgs[

static void encodeblock( unsigned char *in, unsigned char *out, int len )
{
    out[0] = (unsigned char) cb64[ (int)(in[0] >> 2) ];
    out[1] = (unsigned char) cb64[ (int)(((in[0] & 0x03) << 4) | ((in[1] & 0xf0) >> 4)) ];
    out[2] = (unsigned char) (len > 1 ? cb64[ (int)(((in[1] & 0x0f) << 2) | ((in[2] & 0xc0
    out[3] = (unsigned char) (len > 2 ? cb64[ (int)(in[2] & 0x3f) ] : '=');
}

static int encode( FILE *infile, FILE *outfile, int linesize )
{
    unsigned char in[3];
unsigned char out[4];
    int i, len, blocksout = 0;
    int retcode = 0;

*in = (unsigned char) 0;
*out = (unsigned char) 0;
    while( feof( infile ) == 0 ) {
        len = 0;
        for( i = 0; i < 3; i++ ) {
            in[i] = (unsigned char) getc( infile );

            if( feof( infile ) == 0 ) {
                len++;
            }
            else {
                in[i] = (unsigned char) 0;
            }
        }
        if( len > 0 ) {
            encodeblock( in, out, len );
            for( i = 0; i < 4; i++ ) {
                if( putc( (int)(out[i]), outfile ) == EOF ){
                if( ferror( outfile ) != 0 )      {
                    perror( b64_message( B64_FILE_IO_ERROR ) );
                    retcode = B64_FILE_IO_ERROR;
                }
break;
}
            }
```

```
            blocksout++;
        }
        if( blocksout >= (linesize/4) || feof( infile ) != 0 ) {
            if( blocksout > 0 ) {
                fprintf( outfile, "\n" );
            }
            blocksout = 0;
        }
    }
    return( retcode );
}

static void decodeblock( unsigned char *in, unsigned char *out )
{
    out[ 0 ] = (unsigned char ) (in[0] << 2 | in[1] >> 4);
    out[ 1 ] = (unsigned char ) (in[1] << 4 | in[2] >> 2);
    out[ 2 ] = (unsigned char ) (((in[2] << 6) & 0xc0) | in[3]);
}

static int decode( FILE *infile, FILE *outfile )
{
int retcode = 0;
    unsigned char in[4];
    unsigned char out[3];
    int v;
    int i, len;

*in = (unsigned char) 0;
*out = (unsigned char) 0;
    while( feof( infile ) == 0 ) {
        for( len = 0, i = 0; i < 4 && feof( infile ) == 0; i++ ) {
            v = 0;
            while( feof( infile ) == 0 && v == 0 ) {
                v = getc( infile );
                if( feof( infile ) == 0 ) {
                v = ((v < 43 || v > 122) ? 0 : (int) cd64[ v - 43 ]);
if( v != 0 ) {
v = ((v == (int)'$') ? 0 : v - 61);
}
                }
            }
            if( feof( infile ) == 0 ) {
                len++;
                if( v != 0 ) {
                    in[ i ] = (unsigned char) (v - 1);
                }
            }
            else {
                in[i] = (unsigned char) 0;
            }
```

```
        }
        if( len > 0 ) {
            decodeblock( in, out );
            for( i = 0; i < len - 1; i++ ) {
                if( putc( (int) out[i], outfile ) == EOF ){
                if( ferror( outfile ) != 0 )      {
                    perror( b64_message( B64_FILE_IO_ERROR ) );
                    retcode = B64_FILE_IO_ERROR;
                }
break;
}
            }
        }
    }
    return( retcode );
}

static int b64( char opt, char *infilename, char *outfilename, int linesize )
{
    FILE *infile;
    int retcode = B64_FILE_ERROR;

    if( !infilename ) {
        infile = stdin;
    }
    else {
        infile = fopen( infilename, "rb" );
    }
    if( !infile ) {
        perror( infilename );
    }
    else {
        FILE *outfile;
        if( !outfilename ) {
            outfile = stdout;
        }
        else {
            outfile = fopen( outfilename, "wb" );
        }
        if( !outfile ) {
            perror( outfilename );
        }
        else {
            if( opt == 'e' ) {
                retcode = encode( infile, outfile, linesize );
            }
            else {
                retcode = decode( infile, outfile );
            }
if( retcode == 0 ) {
```

```
            if (ferror( infile ) != 0 || ferror( outfile ) != 0) {
                    perror( b64_message( B64_FILE_IO_ERROR ) );
                retcode = B64_FILE_IO_ERROR;
            }
            }
            if( outfile != stdout ) {
                if( fclose( outfile ) != 0 ) {
                    perror( b64_message( B64_ERROR_OUT_CLOSE ) );
                    retcode = B64_FILE_IO_ERROR;
                }
            }
        }
        if( infile != stdin ) {
if( fclose( infile ) != 0 ) {
perror( b64_message( B64_ERROR_OUT_CLOSE ) );
retcode = B64_FILE_IO_ERROR;
}
        }
    }

    return( retcode );
}
```

4.4. Constantes.h

```
//Flags
#define DEBUG 0


//Constantes
#define CONST_ARGC_MIN 2
#define CONST_ARGC_MAX 6
#define CONST_ERR_MSG 600

#define CONST_ARG_HELP "-h"
#define CONST_ARG_VERSION "-V"
#define CONST_ARG_DECODE "-d"
#define CONST_ARG_OUTPUT_NAME "-o"
#define CONST_ARG_INPUT_NAME "-i"


//Mensajes
#define MSG_VERSION "Version 1.0 \n"
#define MSG_NO_ARGS "El programa ha sido invocado sin argumentos.\n"
#define MSG_NO_FILE "El nombre del archivo a convertir es inexistente.\n"
#define MSG_ERR_ARGS "Los argumentos no han sido llamados de forma correcta.\n"
#define MSG_HELP  "La invocacion debe ser de la forma: [-d] -i NOMBRE_ARCHIVO_ORIGEN  -o N
#define MSG_OK "Operacion Completada con Exito\n"
//Modos
```

```
#define MODO_CODIFICADOR 'e'
#define MODO_DECODIFICADOR 'd'
#define MODO_AYUDA 'h'
#define MODO_VERSION 'V'
```

5. Bibliografía y Recursos utilizados

1. https://en.wikibooks.org/wiki/LaTeX/
2. http://stackoverflow.com
3. https://es.wikipedia.org/wiki/Nueva_linea/
4. http://www.bluesock.org/ willg/dev/ascii.html