Práctica Protección y Seguridad

Tomás Felipe Melli

Noviembre 2024

Índice

1 Ejercicio 1

Si el usuario tiene control sobre la entrada estándar, el siguiente código tiene problemas de seguridad.

```
void saludo(void) {
    char nombre[80];
    printf("Ingrese su nombre: ");
    gets(nombre);
    printf("Hola %s!\n", nombre);
}
```

Se considera como un problema de seguridad que un usuario atacante pueda realizar operaciones que no le eran permitidas, ya sea tanto acceder a datos privados, ejecutar código propio en otro programa o inhabilitar un servicio. Determinar:

a) ¿Dónde se introduce el problema de seguridad?

b) ¿Qué datos del programa (registros, variables, direcciones de memoria) pueden ser controladas por el usuario?

c) ¿Es posible sobreescribir la dirección de retorno a la que vuelve la llamada de alguna de las funciones `printf` o `gets`?

d) ¿Se soluciona el problema de seguridad si se elimina el segundo `printf`?

Solución :

a) La vulnerabilidad explotable se encuentra en el hecho de que el usuario tiene control sobre la variable **nombre** y que la función **gets** es una función

```
    char* gets(char* str);
```

que lo que hace es **leer desde STDIN** y luego almacena la información en un buffer, en este caso *nombre*. La vulnerabilidad ocurre, ya que esta función, **sólo finaliza la escritura en el buffer apuntado por el parámetro str cuando por STDIN encuentra el caracter nulo \0 o el salto de línea \n**. Esto le da la posibilidad a un atacante de realizar un ataque de tipo **Buffer Overflow**.

b) El usuario en principio controla la variable nombre que ingresa por el STDIN. Luego, si explota la vulnerabilidad, podría capturar información de todo el stackframe de la función.

c) En lo que respecta al IP que se apila en los stackframes de la funciones *printf* o *gets* no debería por qué suceder que falle el retorno.

d) El segundo *printf* no representa el problema de seguridad. Lo que habría que modificar es la función **gets** por

```
    fgets(nombre, sizeof(nombre), stdin);
```

Para poder tener control sobre el tamaño del valor que se va a definir como nombre (que no supere los 79 bytes + el caracter nulo).

2 Ejercicio 2

El siguiente es un sistema de login que valida los datos contra una base de datos.

```
struct credential {
    char name[32];
    char pass[32];
};

bool login(void) {
    char realpass[32];
    struct credential user;
    // Pregunta el usuario
    printf("User: ");
    fgets(user.name, sizeof(user), stdin);
    // Obtiene la contrase a real desde la base de datos y la guarda en realpass
    db_get_pass_for_user(user.name, realpass, sizeof(realpass));
    // Pregunta la contrase a
    printf("Pass: ");
    fgets(user.pass, sizeof(user), stdin);
    return strncmp(user.pass, realpass, sizeof(realpass)-1) == 0;
    // True si user.pass == realpass
}
```

Suponiendo que la función `db_get_pass_for_user()` es totalmente correcta y no escribe fuera de `realpass()`:

a) Hacer un diagrama de cómo quedan ubicados los datos en la pila, indicando claramente en qué sentido crece la pila y las direcciones de memoria. Explicar, sobre este diagrama, de qué datos (posiciones de memoria, buffers, etc.) tiene control el usuario a través de la función `fgets()`.

b) Indicar un valor de la entrada, que pueda colocar el usuario, para loguearse como `admin` sin conocer la contraseña de este.

Solución :

a) Cuando se llama a una función en C, este es el stackframe de la función. Si recibe algún parámetro estará en la base de la pila, la dire de retorno apilada sobre este y luego, hacia posiciones más bajas de memoria, se irán apilando las variables locales como en este caso la **realpass** que es de 32 bytes y el **struct** de 64. *La pila está bien alineada a 8 bytes.

El usuario tiene control sobre los datos que ingrese para modificar los valores del struct. Esto es :

```
1     struct credential user;
2     // Pregunta el usuario
3     printf("User: ");
4     fgets(user.name, sizeof(user), stdin);
```

Si prestamos atención, fgets tiene condición de corte cuando el tamaño del dato ingresado sea **sizeof(user)**, y este valor es igual a 32 bytes de *name* y 32 bytes de *pass*. Por tanto, podría escribir con el *user.name* todo el struct (es decir desde 0xF50 hasta 0x0FD0). Cuando retorna de la función que lee la verdadera contraseña, pide **user.pass** para constatar de que tiene las credenciales :

```
1     // Pregunta la contrase a
2     printf("Pass: ");
3     fgets(user.pass, sizeof(user), stdin);
```

El tamaño máximo que puede tomar la contraseña de nuevo se excede, y como este dato lo controla el usuario, podría escribir desde 0xFB0 (user.pass) hasta 0xFF0 (el final de realpass).

b) Para explotar esta vulnerabilidad el usuario podría **ingresar una password de 32 bytes** repetida 2 veces para hacer que la función de comparación devuelva true al comparar user.pass contra realpass. Ejemplo: cualquier letra (char), siempre la misma 64 veces. El user no importa, mandale cualquier nombre.

3 Ejercicio 3

La siguiente función se utiliza para verificar la contraseña del usuario actual `user` en el programa que desbloquea la sesión luego de activarse el protector de pantalla del sistema. El usuario ingresa la misma por teclado.

```
1  bool check_pass(const char* user) {
2      char pass[128], realpass[128], salt[2];
3      // Carga la contrase a (encriptada) desde /etc/shadow
4      load_pass_from(realpass, sizeof(realpass), salt, user, "/etc/shadow");
5
6      // Pregunta al usuario por la contrase a.
7      printf("Password: ");
8      gets(pass);
9
10     // Demora de un segundo para evitar abuso de fuerza bruta
11     sleep(1);
12
13     // Encripta la contrase a y compara con la almacenada
14     return strcmp(crypt(pass, salt), realpass) == 0;
15 }
```

- `void load_pass_from(buf, buf_size, salt, user, file)` lee del archivo `file` la contraseña encriptada del usuario `user` y la almacena en el buffer `buf`, escribiendo a lo sumo `buf_size-1` caracteres y un "\0" al final de estos; guarda además en `salt` el valor de los dos caracteres que se usaron para encriptar la contraseña guardada en `file`. Si el usuario no se encuentra, se almacena en `buf` un valor de contraseña inválido, no generable por la función `crypt`.
- `char* crypt(pass, salt)` devuelve la contraseña `pass` encriptada usando el valor `salt`.
- `/etc/shadow` almacena información sensible del sistema y un usuario común no tiene acceso a este archivo, tiene permisos `r-- --- --- root root /etc/shadow`.

a) La línea 4 del código hace un llamado a una función que debe leer el archivo `/etc/shadow`, protegido para los usuarios sin privilegio. Explicar qué mecanismo permite lanzar este programa a un usuario normal y hace que éste tenga acceso a `/etc/shadow` sin alterar los permisos del archivo.

b) Explicar por qué esta función tiene problemas de seguridad. ¿Qué datos sensibles del programa controla el usuario?

c) Si solo los usuarios que ya ingresaron al sistema, que de por sí pueden ejecutar cualquier programa desde la terminal, son los que pueden utilizar el protector de pantalla y por tanto esta función, ¿este problema de seguridad en el protector de pantalla del sistema compromete la integridad del sistema completo? Justificar.

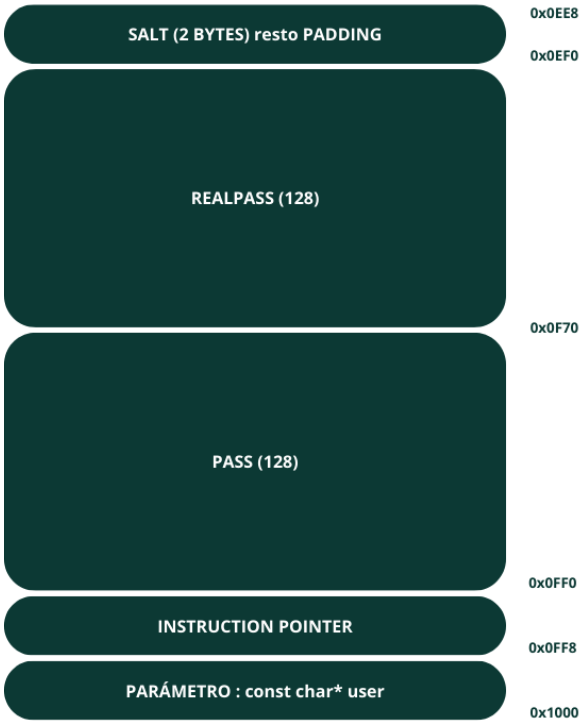
Solución:

a) El mecanismo para permitirle a un programa de usuario correr con privilegios más elevados es **setear el bit SETUID**. Con esto veremos en la terminal con ls -l que esta función que llama el programa check_pass.c)tendrá en la parte del **owner** que en este caso es root: -rw**s**r-xr-x 1 root root 1234567 load_pass_from. Esto sólo afecta los permisos del programa que tenga este bit activado. Por ello, sólo corre con privilegios de root : load_pass_from.

b) El usuario utiliza el teclado para escribir la password cuando el sistema se la demanda en :

```
1          // Pregunta al usuario por la contrase a.
2          printf("Password: ");
3          gets(pass);
```

Y justamente, el usuario controla esa variable que está almacenada en el stackframe de la función chech_pass. Como la función **gets**, como ya vimos, frena de escribir cuando recibe \0 o \n. No tiene limitación de tamaño, y por tanto podría modificar los datos del stack que están por encima de pass (ya que la pila crece hacia posiciones más chicas, pero se escribe de posiciones más bajas a más altas).



Dicho esto, el usuario podría comprometer la **dirección de retorno** y **parámetro de entrada** que en realidad esta última poco importa ya que fue utilizada previamente al exploitable bug, en la función load_pass_from. Sin embargo, sí se podría romper el IP.

c) El tema es si el usuario, por más que no sea un atacante, escriba mal la contraseña (muy mal porque escribir 136 bytes es bastante mucho) rompería la dirección de retorno, lo que afectaría la **disponibilidad** del sistema, en caso de que éste genere una excepción como una #GP o Page Fault.

4 Ejercicio 4

Considerando que el usuario controla el valor del parámetro f, analizar el siguiente código de la función `signo`.

```
#define NEGATIVO 1
#define CERO 2
#define POSITIVO 3

int signo(float f) {
    if (f < 0.0) return NEGATIVO;
    if (f == 0.0) return CERO;
    if (f > 0.0) return POSITIVO;
    assert(false && "Por aca no paso nunca");
    return 0; // Si no pongo esto el compilador se queja =(
}
```

a) ¿El usuario tiene alguna forma de que se ejecute el `assert()`? Pista: Piense en el formato IEEE-754.

b) En las versiones "release" los `asserts` suelen ignorarse (opción de compilación). ¿Sería seguro utilizar la función `signo` sobre un dato del usuario y esperar como resultado alguno de los valores 1, 2 o 3?

Solución:

a) En el formato IEEE-754, los componentes principales de la representación son:

- **Signo (S)**: El bit más significativo que indica si el número es positivo o negativo. 0 es positivo, 1 es negativo.
- **Exponente (E)**: el valor al cuál se debe elevar la base (b) para obtener el valor real.
- **Mantisa (M)**: La parte significativa del número, que es una representación binaria del número.

El Standard también normaliza : ceros signados, números finitos, los de-normalizados (que son esos valores que sufrieron undreflow(que no se pueden representar de lo pequeños que son con la precisión disponible pero no son 0), infinitos signados, y **los NaNs(not a number) que son para representar resultados indefinidos**). Dicho esto, lo que se podría hacer es definir a f como (0.0 / 0.0).

b) Por lo que vimos, es una vulnerabilidad potencialmente explotable. Existe el escenario en que la función no responde al comportamiento esperado.

5 Ejercicio 5

Un esquema de protección implementado por algunos sistemas operativos consiste en colocar el stack del proceso en una posición de memoria al azar al iniciar (stack randomization). Indique cuáles de las siguientes estrategias de ataque a una función con problemas de seguridad se ven fuertemente afectadas por esta protección, y explique por qué:

a) Escribir el valor de retorno de una función utilizando un buffer overflow sobre un buffer en stack dentro de dicha función.

b) Utilizar el control del valor de retorno de una función para saltar a código externo introducido en un buffer en stack controlado por el usuario.

c) Utilizar el control del valor de retorno de una función para ejecutar una syscall particular (por ejemplo `read`) que fue usada en otra parte del programa.

Solución:

a) Si desbordamos el buffer que controlamos como usuarios y escribimos la dire de retorno, no importa mucho dónde se encuentre el stackframe de la función que tiene la vulnerabilidad, ya que la escritura la hacemos como vimos dentro de ese espacio de memoria. Después a dónde vamos ... es el punto b.

b) Si queremos desbordar el buffer y pisar el valor de IP para saltar a código que está en algún lado de la memoria, ahí estamos jodidos, ya que no sabemos precisamente dónde estará nuestro código maligno.

c) ** Aclaración sobre syscalls : Las *syscalls* generalmente no se randomizan porque estas funciones se encuentran en la sección de código del kernel (o, en sistemas operativos de espacio de usuario, en las librerías estándar como la `C standard library`) que están mapeadas en direcciones fijas en la memoria. Dicho esto, no habría problema con ejecutar una syscall pisando en IP.

6 Ejercicio 6

Suponiendo que el usuario controla el parámetro `dir`, el siguiente código, que corre con mayor nivel de privilegio, intenta mostrar en pantalla el directorio `dir` usando el comando `ls`. Sin embargo, tiene problemas de seguridad.

```
1 #define BUF_SIZE 1024
2 void wrapper_ls(const char * dir) {
3     char cmd[BUF_SIZE];
4     snprintf(cmd, BUF_SIZE-1, "ls %s", dir);
5     system(cmd);
6 }
```

a) Muestre un valor de la cadena `dir` que además de listar el directorio actual muestre el contenido del archivo `/etc/passwd`.

b) Posteriormente se reemplazó esta por la función `secure_wrapper_ls` donde el tercer parámetro de `snprintf` en vez de ser `"ls %s"` se reemplaza por `"ls "%s" "`. Muestre que la modificación no soluciona el problema de seguridad.

c) Posteriormente se agregó una verificación de la cadena `dir`: no puede contener ningún carácter “;”. Muestre que esta modificación tampoco soluciona el problema.

d) Proponga una versión de esta función que no tenga problemas de seguridad.

Solución:

a) Este tipo de vulnerabilidades en las que el **formato del string** que ingresa el usuario no está sanitizado se llama **Format String** y en casos como estos que encima ejecuta con niveles elevados de privilegio, es un problema. Ya que como el usuario controla dir podría poner **; cat etc/passwd** pudiendo ejecutar tanto ls en el actual como la lectura del archivo /etc/passwd como cualquier otra cosa, ya que **;** permite la **concatenación de comandos**.

b) **Aclaraciones : lo que sucede acá es que se le agregan las comillas dobles a `"%s"` para que todo lo escrito en dir se considere como un string. Donde si ponemos lo mismo que antes como dir será : **ls ”; cat etc/passwd”**, es decir, se considerará que existe un directorio con ese nombre. Cómo podemos explotar esto ? Le cerramos las comillas y listo para que ejecute **ls ”mi-directorio”; cat etx/passw; ls ”mi-directorio”** y esto lo hacemos con **dir = mi-directorio”; cat etx/passw; ls ”mi-directorio.**

c) **Aclaraciones : tenemos otro **caracter peligroso &**. El símbolo & en la línea de comandos en sistemas Unix/Linux se usa para ejecutar un comando en segundo plano (es decir si se ejecuta : comando1 & comando2 se ejecutan sin importar si alguno falla y no es necesario que comando2 espere a que termine comando1 (como sí lo hace ; que es secuencial (aunque no importa si falla el anterior))). También es útil para ejecutar múltiples comandos en la misma línea, si se usa doble && ahí si importa el resultado del anterior. En fin, si usamos este símbolo, todavía podemos explotar esta vulnerabilidad. Usamos **dir = mi-directorio" & cat etx/passw& ls "mi-directorio.**

d) Para eliminar problemas de seguridad en una función como esta, se podría implementar una **validación de formato** también llamada **AllowList** donde sólo le dejamos moverse por un conjunto de directorios autorizados y sino una **BlockList** que recorremos la entrada y si encontramos una entrada inválida, retornamos :

```
    // con ALLOW LIST
    int is_valid_char(char c) {
        // Permitimos letras, n meros, guiones, puntos, y barras
        return (isalnum(c) || c == '-' || c == '.' || c == '/');
    }

    // Funci n para verificar si el nombre del directorio es v lido
    int is_valid(const char* dir) {
        while (*dir) {
            if (!is_valid_char(*dir)) {
                return 0; // Entrada inv lida si encontramos un car cter no permitido
            }
            dir++;
        }
        return 1; // Entrada v lida si todos los caracteres son permitidos
    }

    #define BUF_SIZE 1024
    void wrapper_ls(const char * dir) {
        char cmd[BUF_SIZE];
        snprintf(cmd, BUF_SIZE-1, "ls %s", dir);
        if (!is_valid(dir)) return;
        system(cmd);
    }
```

7 Ejercicio 7

Suponiendo que el usuario controla la entrada estándar, el siguiente código tiene problemas de seguridad:

```
#define BUF_SIZE 1024

int suma_indirecta(void)
{
    int buf[BUF_SIZE];
    int i, v;
    memset(buf, 0, sizeof(buf));

    while (cin >> i >> v)
    {  // Leo el  ndice  y el valor
        if (i == -1) break;  // Un  ndice  -1 significa que tengo que terminar.
        if (i < BUF_SIZE) buf[i] = v;  // Guardo el valor en el buffer
    }
    // Calculo la suma de los valores
    v = 0;
    for (i = 0; i < BUF_SIZE; i++)
        v += buf[i];
    return v;
}
```

a) El código verifica que el valor de i no se pase del tamaño del buffer (`BUF_SIZE`). ¿Es suficiente esta verificación para garantizar que no se escribe fuera de `buf`?

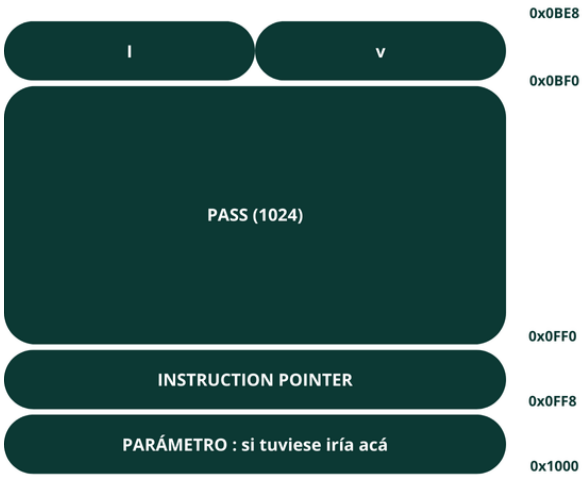
b) Considerando que la dirección de retorno de esta función (`suma_indirecta`) se encuentra en una posición de memoria más alta (mayor) que `buf`, ¿existe algún valor de i que permita sobreescribirla al ejecutar el cuerpo del `while`? Justifique. Pista: Pensar en la aritmética de punteros que se realiza dentro del cuerpo del `while`.

c) Si el compilador protegiera el stack colocando un canario de valor desconocido (incluso aleatorio), en una posición de memoria antes (una posición menor) del `return address` de cada función, ¿aún es posible modificar el retorno de la función `suma_indirecta` y retornar de la misma satisfactoriamente?

Solución:

a) El problema del índice es que no se verifica cuando es menor a -1. Podría escribir i = -2 y entraría en la guarda $if(i < BUF\_SIZE)\ buf[i] = v;$.

b) **Aclaración : en C como no hay mecanismos para prevenir este tipo de accesos podría comportarse de formas inesperadas como : **sobrescribiendo datos adyacentes** cerca del array en tanto y cuanto el proceso tenga el acceso adecuado a ese segmento, sino **segmentation fault**. Dicho esto, el uso de valores negativos de i escribiría fuera del espacio reservado para el buf :



Con esto en mente, recordemos que cuando escribimos, dentro del buffer no se hace desde 0x0FF0 hacia arriba (hacia direcciones más pequeñas, se hace al revés, desde 0x0BF0 hacia las direcciones más grandes) por tanto, un índice negativo, entendiendo que hacemos : *(buf + i) escribiría en direcciones posteriores a 0x0BE8. Cómo logramos romper esto ? Resulta que en esta notación *(buf + i) i es considerado un **unsigned int** por tanto, podríamos ingresar un número que interpretado como int sea negativo pero que al ser llevado a *(buf + i) se vuelva tan positivo que supere el BUF_SIZE de forma de pisar el IP. Con una sóla vez alcanza ? No, porque el buf[i] escribe de a 4bytes por tanto esto lo tenemos que hacer dos veces para cubrir los 8bytes del IP.

c) El compilador podría implementar el **canary**, se pisaría todo igual, el canary no impide que se sobreescriban los datos, impide la ejecución de código malicioso sobreescrito en el stack frame. No, luego de que el compilador chequea si el canary es el mismo y no lo es, Aborta la ejecución y lanza SIGABRT y algunos sistemas implementan un *core dump* en donde se captura el estado del programa cuando se detectó el fallo para poder debbugear y ver cuál es el potencial exploitable bug.

8 Ejercicio 8

Suponiendo que el usuario controla la entrada estándar, el siguiente código tiene problemas de seguridad.

```
#define MAX_BUF 4096
void saludo(void) {
    char nombre[MAX_BUF];
    printf("Ingrese su nombre: ");
    fgets(nombre, MAX_BUF, stdin);
    printf(nombre);
}
```

a) ¿Dónde se introduce el problema de seguridad?

b) ¿Qué datos del programa (registros, variables, direcciones de memoria) pueden ser controlados por el usuario? *(Ayuda: lea con mucho detalle man printf y permita sorprenderse).*

c) ¿Es posible sobreescribir la dirección de retorno a la que vuelve alguna de las llamadas a la función `printf`?

d) ¿Se soluciona el problema de seguridad si, luego del segundo `printf`, se coloca una llamada a `exit(0)`?

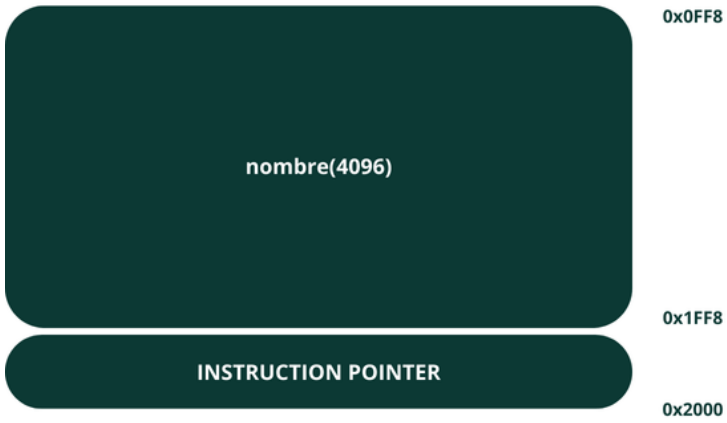
Solución:

a) El problema de seguridad se introduce en la línea 6. Ya que la función `printf` imprime una variable controlada por el usuario y no tiene ningún tipo de saneamiento.
**Aclaración sobre `printf`, existen operadores de formato como :

- **`%x`** : Cada vez que `printf` encuentra un especificador como `%x`, busca en la pila el siguiente dato disponible y lo interpreta como un número hexadecimal. Al hacerlo sin control, `printf` puede leer datos ubicados en posiciones cercanas en la pila, incluyendo direcciones de variables, valores de otras funciones e incluso la dirección de retorno.
- **`%n`** El especificador `%n` en `printf` permite escribir en una posición de memoria indicada. En lugar de imprimir un valor, `%n` almacena la cantidad de caracteres impresos hasta ese punto en la dirección de memoria que se le pasa como argumento.

Entre otros que también pueden ser utilizados para explotar esta vulnerabilidad.

b) Leimos el manual para las aclaraciones de `printf` así que sólo vamos a responder que el usuario tendrá acceso a todo el stackframe :



c) Sí, lo que habría que hacer para explotar la vulnerabilidad es pasar como parámetro "Tomi %x %x " muchas veces hasta ver que empiezan a aparecer valores, o podríamos ir interpretando como enteros hasta que veamos algo que tenga pinta de puntero (IP) y luego capturar el momento en el que aparece, para en otra oportunidad pasar cierta data para escribirle y poder explotar ese bug. Es difícil eso. La fácil es generar un segmentation fault.

d) No soluciona nada ese exit(0). Si queremos resolver la vulnearibilidad, hay que hacer printf("%s", nombre).

9 Ejercicio 9

Teniendo en cuenta el ejercicio anterior, considerar el siguiente código:

```
1 void saludo(void) {
2     char nombre[80];
3     printf("Ingrese su nombre: ");
4     gets(nombre);
5     printf(nombre);
6     exit(0);
7 }
```

Dado que la función saludo nunca retorna y, por ende, no interesa el valor de su `return address`, ¿es segura la segunda llamada a `printf`?

Solución :
La función nunca retorna significa que como tiene el **exit(0)**, se termina la ejecución luego del print, me pregunto igualmente si no hay nada en la pila donde debería estar el IP, ya que alguien llamó a *saludo*. Yo creo que se podría evitar que haga el exit(0) y podríamos romper algo. Porque el **gets** nos deja escribir infinitamente... así que por ahí podemos generar un **segfault**.

10 Ejercicio 10

Versiones de la popular biblioteca de encriptación OpenSSL anteriores a la 0.9.8d tenían un problema de seguridad en la función `SSL_get_shared_ciphers`. A continuación se encuentra una versión simplificada de la función en cuestión. La idea de la función es concatenar todos los nombres que vienen en `ciphers[]` colocando un carácter ":" entre ellos.

```
char *SSL_get_shared_ciphers( char* ciphers[], int cant_chipers, char *buf,int len)
{
    char *p;
    const char *cp;
    int i;

    if (len < 2)
        return(NULL);

    p=buf;

    for (i=0; i<cant_ciphers; i++)
    {
        /* Decrement for either the ":" or a "\0" */
        len--;
        for (cp=ciphers[i]; *cp; )
        {
            if (len-- == 0)
            {
                *p="\0";
                return(buf);
            }
            else
                *(p++)= *(cp++);
        }
        *(p++)=":";
    }
    p[-1]="\0";
    return(buf);
}
```

Dado que el usuario controla el contenido de `ciphers` y `cant_ciphers`, siendo el primero un array válido de `cant_ciphers` posiciones definidas donde cada una apunta a una cadena válida de a lo sumo 100 caracteres y al menos un carácter, y que `buf` es un buffer de tamaño `len` que no es controlado por el usuario:

a) ¿Es posible escribir información del usuario más allá del límite de tamaño de `buf`, a pesar de todos los chequeos de longitud? Explicar cómo.

b) En la versión 0.9.8d corrigieron este problema cambiando la línea "`if (len-- == 0)`" por "`if (len-- <= 0)`". Sin embargo, dos versiones después encontraron que esta corrección aún permitía sobreescribir un valor 0 un byte después de `buf`. Si bien poder escribir un byte en 0 luego de un buffer no parece permitir tomar control del programa, también es considerado un problema de seguridad. ¿Qué riesgos podría correr un servidor que utiliza esta biblioteca para conexiones encriptadas por Internet?

Solución:

a) Para responder a esta pregunta miremos bien esta sección del código :

```
        for (i=0; i<cant_ciphers; i++)
        {
            /* Decrement for either the ":" or a "\0" */
            len--;
            for (cp=ciphers[i]; *cp; )
            {
                if (len-- == 0)
                {
                    *p="\0";
                    return(buf);
                }
                else
                    *(p++)= *(cp++);
            }
            *(p++)=":";
        }
        p[-1]="\0";
        return(buf);
```

La función lo que hace es, captura un areglo de palabras y las concatena de forma de obtener algo como : hola:mundo:soy:tomi. El usuario tiene control sobre ciphers[] y sobre la cantidad. Para lograrlo, itera sobre la cantidad de ciphers y dentro de cada uno, copia char a char. Este es el bucle que nos interesará por la línea 7 del extracto de arriba (18 en la original). En particular, la forma en que el if maneja la comparación, en este caso, captura len, compara y luego disminuye el valor. Por tanto, va a existir el caso en que len = 1 y la comparación te deje hacer la guarda del *else*, pero ahora len = 0. Pero supongamos que después del else se te acabó la palabrita, (*cp = "\0"). Porque recordemos que **len** es la del **buffer**. Entonces, sale del bucle interno, agrega ":", tenemos otra palabra, pero len = -1 entonces, sigue escribiendo. Dónde ?

Como vemos en el diagrama, justo tiene el IP.

b) La modificación de la línea 18 en el original a `if (len-- <= 0)` resuelve el tema de que no se escriba mucho más, pero sigue teniendo el temita de que el 0 se escribe cuando la longitud es 0, pero como es post-decremental la operación, el espacio que ocupará \0 ya es fuera del espacio del buffer. Para el caso en que un servido haga uso de esta biblioteca para las conexiones encriptadas podría suceder que se comprometa la seguridad de la conexión.

11 Ejercicio 11

En algunas combinaciones de sistema operativo y compilador, el siguiente código permite al usuario tomar control de la ejecución del programa:

```
1 void leo_y_salgo(void) {
2     char leo[80];
3     gets(leo);
4     exit(1);
5 }
```

Dado que al regresar de la función `gets` el programa termina la ejecución ignorando el valor de retorno de la función `leo_y_salgo`, para tomar control del programa se debe evitar volver de esta función. Sabiendo que en estos sistemas al inicio del stack se almacena la dirección de los distintos handlers de excepciones del proceso (división por cero, error de punto flotante, etc.), explique cómo puede tomar control de la ejecución sin regresar de la función `gets`.

Solución:

Como dice la consigna, en estos sistemas tenemos al inicio del stack la dirección de los handlers de excepciones. Acceder al stack y pisarlo, lo podemos hacer ya que a **gets** la hacemos pelota. El asunto es escribirse el **shellcode** de forma de llenar el buffer de ese código maligno que sea un shellcode pensado para handlear cualquiera de las interrupciones que podemos manejar porque las direcciones de sus handlers están ahí. Para responder este ejercicio se puede consultar la subsección siguiente la 11.1 que hablamos sobre **shellcode**.

11.1 Notas adicionales sobre SHELLCODE

El proceso de almacenamiento del shellcode en la pila generalmente involucra los siguientes pasos:

1. **Escribir el shellcode en formato binario:** El shellcode se escribe en lenguaje ensamblador, pero debe ser ensamblado y enlazado para producir una secuencia de bytes en código máquina que se pueda almacenar en memoria. El proceso de ensamblado convierte el código en instrucciones que la CPU puede ejecutar.

2. **Inyectar el shellcode en un buffer:** Cuando se explota una vulnerabilidad (como en este caso el desbordamiento de un buffer), el atacante inyecta los bytes del shellcode directamente en la pila del programa. Este shellcode se coloca en una sección de la memoria contigua a las variables y la dirección de retorno de la función, que es una parte crítica de la explotación.

3. **Formato binario en la pila:** En la pila, el shellcode es simplemente una secuencia de bytes. Cada byte corresponde a una instrucción del código máquina. Este código máquina no tiene ningún formato especial, aparte de ser ejecutable. Se almacenan en la pila como una secuencia de bytes que el procesador puede leer y ejecutar directamente cuando se salta a la dirección de memoria correspondiente.

4. **Dirección de retorno sobrescrita:** En una explotación de desbordamiento de búfer, el atacante suele sobrescribir la dirección de retorno de la función con la dirección de inicio del shellcode en la pila. Esto hace que el programa, al regresar de la función, ejecute el shellcode en lugar de la siguiente instrucción que originalmente debería haber sido ejecutada.

Escribimos el **handler** :

```
handler:
    ; Aqu  est  el c digo del manejador que se ejecutar  al recibir un segfault
    ; Cargamos la direcci n de /bin/sh en los registros adecuados para execve

    xor rdi, rdi              ; Limpiamos rdi (argumento 1: nombre del programa, NULL)
    lea rsi, [rel shell]      ; Cargamos la direcci n de la cadena "/bin/sh" en rsi
                                (argumento 2: argumento de la l nea de comandos)
    xor rdx, rdx              ; Limpiamos rdx (argumento 3: variables de entorno, NULL)

    ; Llamada al sistema execve
    mov rax, 0x3b             ; N mero de syscall para execve (en Linux, 0x3b es execve)
    syscall                   ; Ejecuta execve("/bin/sh", NULL, NULL)

shell:
    db '/bin/sh', 0           ; Cadena "/bin/sh" terminada en NULL
```

Lo tenemos que **ensamblar** a binario :

```
; Compilar el c digo NASM en formato ELF64
nasm -f elf64 handler.asm -o handler.o

; Enlazar el archivo objeto a un ejecutable
ld handler.o -o handler
```

Después lo tenemos que convertir en **Shellcode** (formato hexadecimal) :

```
objdump -d handler | grep '[0-9a-f]:' | \
    sed 's/^[[:space:]]*[0-9a-f]*:[[:space:]]*//;s/[[:space:]]\+/\n/g' | \
    tr -d '\n' | sed 's/\(..\)/\\x\1/g'
```

Una vez que está en este formato, en hexa, lo podemos poner en el buffer y que el **handler**, que le vamos a modificar la dirección nosotros(porque modificamos la pila), comience en la dirección donde está esto :

```
\x48\x31\xd2\x48\x31\xc0\x48\x31\xff\x48\x89\xf2\x48\x83\xc4\x08
\x48\x83\xec\x28\x48\x8d\x3d\x00\x00\x00\x00\x48\x8d\x7d\x00
\x0f\x05
```