Ataque de Buffer Overflow

Christofer Fabián Chávez Carazas
Universidad Nacional de San Agustín
Seguridad Computacional

8 de abril de 2017

1. Descripción del Experimento

El experimento consiste en provocar un *buffer overflow* paso a paso, ver el manejo de la memoria y los problemas que genera, y cómo podemos utilizar el ataque a nuestro favor. El programa que se utilizó para el experimento se muestra a continuación. Para ver el paso a paso se utilizó el programa *gdb*. Todo el experimento se realizó en una máquina virtual con ubuntu 32-bits con 1 Gb de memoria RAM.

```c
#include <stdio.h>

int read_req(void){
    char buf[128];
    int i;
    gets(buf);
    i = atoi(buf);
    return i;
}

int main(int ac, char ** av){
    int x = read_req();
    printf("x= %d\n",x);
}
```

2. Programa

El programa a ejecutar consiste en una simple función. Comienza creando un buffer de 128 bytes de tamaño. Luego se le pide al usuario que ingrese un dato y se almacena en el buffer. Después se convierte el buffer en un entero para luego retornarlo e imprimirlo en pantalla. En la Tabla 1 se muestra una lista de entradas que se le dio al programa y lo que imprimió el programa para cada una. Cuando nosotros ponemos un número muy grande bota un resultado muy diferente, esto se debe a que el número es truncado para que pueda entrar dentro de la variable *i*. La función *atoi()* itera la cadena de caracteres buscando un dígito, si encuentra algo que no lo es deja de iterar. Por esta razón cuando ingresamos "12345AAA", la función itera hasta encontrar la primera 'A' y devuelve 12345 como respuesta. En el caso de la entrada "AAAAAAAAA", la función termina en la primera iteración retornando 0

Input	Output
12345	12345
123456789123456789	2147483647
12345AAA	12345
AAAAAAAAAAA	0
130 veces 'A'	Error (stack smashing detected)

Tabla 1: Resultados que arroja el programa

como respuesta. La última entrada de la tabla hace que el programa termine por un desbordamiento de buffer, esto ocurre porque la entrada es más grande que el buffer, en el caso del experimento, una cadena de 130 ases.

3. Paso a Paso

Abrimos el programa con *gdb* y colocamos un *breakpoint* al inicio de la función *read_req()*. Corremos el programa hasta antes de que se ejecute la función *gets()*. En la Figura 1 se muestran los registros, generados por el comando *info registers*, y podemos ver la dirección de nuestra pila 0xbfffeee0. En la Figura 2 podemos ver la dirección de memoria del inicio del buffer 0xbfffeeec y la dirección de la variable *i* 0xbfffeee8. Analizando estas direcciones, nuestra pila comienza con la dirección de retorno en bfffeee0 hasta bfffeee8 donde comienza la variable *i*. Esta variable termina en bfffeeec donde comienza el buffer. Continuamos con la ejecución del programa y ponemos 181 veces la letra 'A'. En la figura 3 se puede ver que la dirección del buffer está lleno de letras 'A'. Lo que pasa en esta situación es que se desborda el buffer de nuestro programa y llena la dirección de retorno, y al intentar leer esta dirección ocurre un error.

Al volver a correr el programa hasta antes de que el programa bote el error, cambiamos la dirección de retorno a la del main. Lo que pasa en este caso es que si imprime el valor sólo que ya no tiene otro valor de retorno y vuelve a botar error. Este ataque es muy explotable ya que, al desbordar el buffer, se puede poner un nuevo puntero en la dirección de retorno que direccione el programa a una función maliciosa.

4. Conclusiones

- El buffer overflow es uno de los ataques más explotables y manipulables.
- Siempre hay que verificar el tamaño que se guarda en un buffer.

```
(gdb) info reg
eax            0x0      0
ecx            0xbfffefb0       -1073746000
edx            0xbfffefd4       -1073745964
ebx            0xb7fc1000       -1208217600
esp            0xbfffeee0       0xbfffeee0
ebp            0xbfffef78       0xbfffef78
esi            0x0      0
edi            0x0      0
eip            0x80484df        0x80484df <read_req+20>
eflags         0x246    [ PF ZF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb)
```

Figura 1: Registros antes del *gets()*

```
(gdb) print &buf[0]
$2 = 0xbfffeeec "\034\202\004
(gdb) print &i
$3 = (int *) 0xbfffeee8
```

Figura 2: Direcciones del inicio del buffer y la variable i

```
(gdb) print &buf[0]
$5 = 0xbfffeeec 'A' <repeats 181 times>
(gdb) x $ebp
0xbfffef78:     0x41414141
(gdb) x $ebp+4
0xbfffef7c:     0x41414141
(gdb)
```

Figura 3: Registros de la pila