

Examen práctico análisis de vulnerabilidades

Teniendo el archivo descargado del repositorio, con el comando *file* me di cuenta que no era un binario, si no un archivo comprimido. Lo descomprimí con *tar -xzf*.

```
—[oscare@parrot]—[~/Documents/vulne/examen]  
— $file SHELLow  
SHELLow: gzip compressed data, last modified:  
—[oscare@parrot]—[~/Documents/vulne/examen]  
— $file shell_mod2  
shell_mod2: ELF, unknown class 113  
—[oscare@parrot]—[~/Documents/vulne/examen]  
— $
```

Cuando intentaba ejecutar el binario, me salía error de *Exec format error*, con *strings* me apareció la cadena que había que borrarle al binario, apoyándome de cómo es que se forma la cabecera ELF de los binarios de Linux.¹ Entonces borré el string que aparecía en el header. Primero intenté ejecutarlo sin gdb esperando que resulte algún error de Overflow en la entrada del programa. Como no pude obtener nada de esta forma, decidí entrar en gdb, mostrando primero las funciones definidas con *info functions*. Posteriormente un *disas main*, no había nada claro en un principio así que empecé a recorrer el programa con *si*.

En cuanto llegué a *call rdx* y el programa brincó a *shellcode*, aún cuando este no era una función definida. Mucho tiempo después, cuando no salía nada se me ocurrió utilizar Hopper para analizar el binario, y aquí me di cuenta que *shellcode* era una variable definida en *.data*.

Siguiendo con el análisis, me di cuenta que el programa realizaba varias *syscalls*, busqué cada una con el valor de *rax*, pero eran llamadas medio raras (*sys_geteuid*, con *rax = 0x31*, por ejemplo).

Seguía con el programa hasta que gdb se detenía como esperando una entrada, donde volví a intentar un Overflow sin éxito. Se me ocurrió revisar si es que se abría un puerto con

¹ https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

netstat -tanop. En efecto se abría un socket en el puerto 39321. Tercamente también buscaba Overflow en la conexión con el socket, claro sin éxito siempre.

Decidí, tiempo después, por fin seguir analizando el binario sin buscar Overflows. Entonces intenté probar con el serial que aparecía con *strings* en la entrada del socket, ahí fue donde comenzó a cobrar poco más sentido todo. Vi que se lee la entrada dada hasta que se encuentra con un `\n (0xA)`. Después compara el contador de los caracteres *rcx* con `0x1d`, que en caso de no coincidir directamente mandaba a la finalización del binario. Posteriormente compara que existan guiones medios cada 5 caracteres. Finalmente suma el código ASCII del serial dado, y si es distinto a `0x8e0` termina la ejecución del binario.

Durante todo este análisis, encontré precisamente cómo había que manejar el binario:

En esta *syscall* es donde se abre el socket para la conexión, incluso en la parte de abajo se observa cómo es que se espera a que se establezca una conexión:

```

0x600a7f <shellcode+63> mov     al,0x32
0x600a81 <shellcode+65> syscall
0x600a83 <shellcode+67> mov     al,0x2b
0x600a85 <shellcode+69> syscall
0x600a87 <shellcode+71> push    rax
0x600a88 <shellcode+72> pop     rdi
0x600a89 <shellcode+73> xor     rdx,rdx

```

active process 4361 In: shellcode

```

x000000000000600a7b in shellcode (0)
x000000000000600a7c in shellcode (1)
x000000000000600a7e in shellcode (2)
x000000000000600a7f in shellcode (3)
x000000000000600a81 in shellcode (4)

```

Ascii Table - ASCII character codes and html, octal, Mozilla Firefox (sandboxed or root)

Menu Parrot Terminal Ascii Table - ASCII

Sigue la lectura mandada desde el socket, que se obtiene con esta *syscall*, en la que igual la ejecución en gdb continuaba una vez que ya le había mandado el serial por la conexión al socket:

```

0x600a94 <shellcode+84> xor     eax,eax
0x600a96 <shellcode+86> syscall (syscall)
> 0x600a98 <shellcode+88> xor     rax,rax
0x600a9b <shellcode+91> mov     al,0x4a
0x600a9d <shellcode+93> sub     al,0x40
0x600a9f <shellcode+95> xor     rcx,rcx
0x600aa2 <shellcode+98> cmp     BYTE PTR [rsp+rcx*1],al

native process 4361 In: shellcode
0x00000000000000a8e in shellcode: (je return)
0x00000000000000a91 in shellcode: (jbe return)
0x00000000000000a92 in shellcode: (jbe return)
0x00000000000000a93 in shellcode: (jbe return)
0x00000000000000a94 in shellcode: (jbe return)
0x00000000000000a96 in shellcode: (je acknowledge)
0x00000000000000a98 in shellcode: (jbe trans_block)
(gdb)
Menu  Parrot Terminal  Ascii Table - ASCII char...

```

Después de un ciclo que no sé para que funcionaba, viene el ciclo en el que se cuentan los caracteres del serial, que la suma se almacena en *rcx*. Compara el carácter con el valor del registro *rax*, si son iguales entonces sale del ciclo.

```

Register group: general
rax      0xa 10
rcx      0x6 6
rsi      0x7fffffff010 140737488347152
rbp      0x7fffffff090 0x7fffffff090
r8       0x7ffff7f9f500 140737353741568
r10      0x601010 6295568

> 0x600aa2 <shellcode+98> cmp     BYTE PTR [rsp+rcx*1],al
0x600aa5 <shellcode+101> je      0x600aac <shellcode+108>
0x600aa7 <shellcode+103> inc     rcx
0x600aaa <shellcode+106> jmp     0x600aa2 <shellcode+98>
0x600aac <shellcode+108> cmp     rcx,0x1d
0x600ab0 <shellcode+112> jne     0x600b3f <shellcode+255>
0x600ab6 <shellcode+118> xor     rcx,rcx

```

Es en este momento cuando compara la longitud del serial: compara que el contador sea igual a *0x1d*, en caso de no coincidir, termina la ejecución del programa.

```

Register group: general
rax      0xa 10
rcx      0x1c 28
rsi      0x7fffffff010 140737488347152
rbp      0x7fffffff090 0x7fffffff090
r8       0x7ffff7f9f500 140737353741568
r10      0x601010 6295568

0x600aa2 <shellcode+98> cmp     BYTE PTR [rsp+rcx*1],al
0x600aa5 <shellcode+101> je      0x600aac <shellcode+108>
0x600aa7 <shellcode+103> inc     rcx
0x600aaa <shellcode+106> jmp     0x600aa2 <shellcode+98>
> 0x600aac <shellcode+108> cmp     rcx,0x1d
0x600ab0 <shellcode+112> jne     0x600b3f <shellcode+255>
0x600ab6 <shellcode+118> xor     rcx,rcx

```

Oscar Espinosa Curiel

Ahora compara que exista el guion medio cada 5 caracteres. Para esto se utiliza *rcx* como contador, es decir, este almacena las posiciones relativas del guion, la cual suma al *rsp* para obtener el carácter en esa posición. Todas las comparaciones se hacen con el código ASCII.

```
Register group: general
rax      0xa      10
rcx      0x5      5
rsi      0x7fffffff010 140737488347152
rbp      0x7fffffff090 0x7fffffff090
r8       0x7ffff7f9f500 140737353741568
r10      0x601010 6295568

0x600ab6 <shellcode+118> xor rcx,rcx
0x600ab9 <shellcode+121> add cl,0x5
> 0x600abc <shellcode+124> cmp BYTE PTR [rsp+rcx*1],0x2d
0x600ac0 <shellcode+128> jne 0x600b3f <shellcode+255>
0x600ac2 <shellcode+130> add cl,0x6
0x600ac5 <shellcode+133> cmp cl,0x11
0x600ac8 <shellcode+136> jbe 0x600abc <shellcode+124>
```

Si se cumple con esta condición, entra en otro ciclo que suma cada uno de los valores ASCII de la cadena. Primero se copia el carácter a *bl*, y este valor se suma con *rax*. Se suma hasta que *rcx* llega a cero.

```
Register group: general
rax      0x0      0
rcx      0x1c     28
rsi      0x7fffffff010 140737488347152
rbp      0x7fffffff090 0x7fffffff090
r8       0x7ffff7f9f500 140737353741568
r10      0x601010 6295568
rbx      0x41     65
rdx      0x20     32
rdi      0x4      4
rsp      0x7fffffff010 0x7fffffff010
r9       0x77     119
r11      0x346    838

0x600acf <shellcode+143> xor rax,rax
0x600ad2 <shellcode+146> xor rbx,rbx
0x600ad5 <shellcode+149> mov bl,BYTE PTR [rsp+rcx*1]
> 0x600ad8 <shellcode+152> add rax,rbx
0x600adb <shellcode+155> loop 0x600ad2 <shellcode+146>
0x600add <shellcode+157> xor rbx,rbx
0x600ae0 <shellcode+160> mov bl,BYTE PTR [rsp+rcx*1]
```

El valor resultante lo compara con *0x8e0*, si son iguales brinca a la dirección que te devuelve una Shell en el puerto, en caso contrario termina el programa. Como en la primera de las siguientes imágenes la suma no coincide, brinca a la dirección *shellcode+255*, donde termina el programa. En la segunda de estas, que la suma sí coincide, sigue con la ejecución normal que devuelve una Shell en el puerto.

```
Register group: general
rax      0x70d      1805      rbx
rcx      0x0        0          rdx
rsi      0x7fffffff010 140737488347152 rdi
rbp      0x7fffffff090 0x7fffffff090    rsp
r8       0x7ffff7f9f500 140737353741568 r9
r10      0x601010    6295568    r11

0x600add <shellcode+157> xor     rbx,rbx
0x600ae0 <shellcode+160> (null)  mov     bl,BYTE PTR [rsp+rcx*1]
0x600ae3 <shellcode+163> (start of block) add     rax,rbx
> 0x600ae6 <shellcode+166> (end of block) cmp     rax,0x8e0
0x600aec <shellcode+172> (enquiry) jne     0x600b3f <shellcode+255>
0x600aee <shellcode+174> (acknowledged) lea     rdx,[rsp+0xc]
0x600af3 <shellcode+179> (bell) xor     rcx,rcx
0x600af6 <shellcode+182> (backspace) mov     al,0x5
```

```
Register group: general
rax      0x8e0      2272      rbx
rcx      0x0        0          rdx
rsi      0x7fffffff010 140737488347152 rdi
rbp      0x7fffffff090 0x7fffffff090    rsp
r8       0x7ffff7f9f500 140737353741568 r9
r10      0x601010    6295568    r11

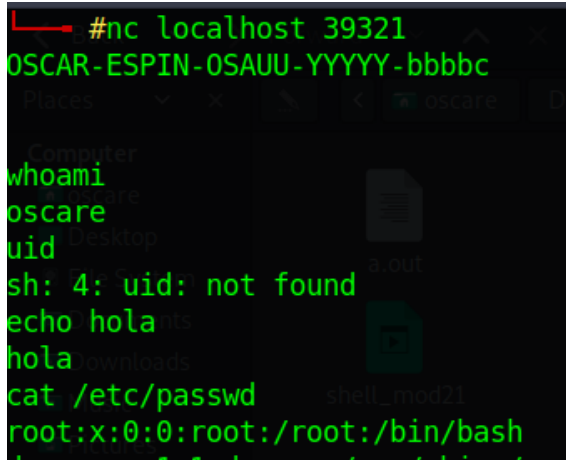
0x600ae0 <shellcode+160> mov     bl,BYTE PTR [rsp+rcx*1]
0x600ae3 <shellcode+163> add     rax,rbx
0x600ae6 <shellcode+166> From here cmp     rax,0x8e0
0x600aec <shellcode+172> jne     0x600b3f <shellcode+255>
> 0x600aee <shellcode+174> lea     rdx,[rsp+0xc] the char. And in case
0x600af3 <shellcode+179> with the xor     rcx,rcx does the trick.
0x600af6 <shellcode+182> mov     al,0x5
```

Entonces, una vez teniendo un serial válido se ejecuta de manera normal el binario. Lo intenté dentro de gdb, pero resultaba en error y se detenía la ejecución.

```
0x0000000000600b29 in shellcode ()
(gdb) c
Continuing.
process 4989 is executing new program: /bin/dash
warning: Cannot access memory at address 0x600b29

[2]+  Stopped                  gdb shell_mod21
[~]-[oscare@parrot]-[~/Documents/vulne/examen]
$gdb shell
```

Entonces, ejecutándolo fuera de gdb se podía sin ningún problema, y tenía entonces una Shell en el puerto del binario, tal como se muestra en la siguiente imagen.



```
#nc localhost 39321
OSCAR-ESPIN-OSAUU-YYYYY-bbbbc
whoami
oscare
uid
sh: 4: uid: not found
echo hola
hola
cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
```

Para calcular mi serial programé un script en Python para calcular los caracteres necesarios para que el serial cumpla con las validaciones que hace el binario, tales como: longitud de 29 caracteres, los guiones medios, la suma de sus valores ASCII. Al final, el serial válido con mi nombre en él es: **OSCAR-ESPIN-OSAUU-YYYYY-bbbbc**

El script que programé no hace más que completar una cadena inicial con mi nombre. Calcula la cantidad de caracteres que faltan para completar el serial, la cantidad que hace falta para completar la suma y el promedio de valor por carácter. Entonces solo concatena el carácter correspondiente a ese valor ASCII (el del promedio calculado). Cada 5 caracteres agrega el guion medio.

```
mi_serial="OSCAR-ESPIN-OSA"
condicion_suma = 2272
condicion_long = 29

suma = 0

for x in mi_serial:
    suma += ord(x)
tmps = condicion_suma - suma
tmp1 = condicion_long - len(mi_serial)
tmpp = tmps/tmp1

while tmp1 > 0:
    if (len(mi_serial) + 1) % 6 == 0:
        mi_serial += '-'
        suma += ord('-')
    else:
        mi_serial += chr(tmpp)
        suma += tmpp
        tmp1 -= 1
```

Oscar Espinosa Curiel

```
    tmpr = condicion_suma - suma
    tmpl = condicion_long - len(mi_serial)
    if tmpl > 0:
        tmpp = tmpr/tmpl

print tmpp
print len(mi_serial)
print suma
print mi_serial
```