

Análisis del binario SHELLow



Baruch Guerra Rodolfo

UNAM CERT

Análisis de Vulnerabilidades

Índice

Detalles del análisis	3
Generalidades del binario	3
Análisis del funcionamiento del binario	6
Análisis de la función shellcode	12
Validaciones del serial	20
Validación de longitud (salto de línea)	20
Validación de posición de los guiones (-)	21
Validación de la suma de los caracteres del serial	22
Validación bit a bit (TEST)	23

Detalles del análisis

El análisis del binario en cuestión, fue realizado usando un Ubuntu 16.04 LTS y empleando un par de herramientas que ayudan para la revisión de archivos de este tipo. Dichas herramientas, serán mencionadas durante el desarrollo del documento y se explicará cómo es que se pudo obtener un serial válido para el binario a partir del análisis de lo que fue mostrado por las herramientas.

Generalidades del binario

Una vez que tenemos el binario dentro del sistema, lo primero que se hizo para obtener más información, fue ejecutar el siguiente comando:

```
file SHELLow
```

El comando 'file', nos da información sobre el archivo que le mandemos; en este caso, se obtuvo el output:

```
barvch@ubuntu:~/Downloads$ file SHELLow
SHELLow: gzip compressed data, last modified: Fri Mar 31 19:11:39 2017, from Unix
```

Nos muestra la última fecha de modificación, el S.O. y lo más curioso, es que nos indica que el archivo se encuentra comprimido; por lo que el siguiente paso sería descomprimirlo para revisar el contenido del comprimido. Podemos hacer eso con el siguiente comando:

```
tar -xvf SHELLow
```

Y obtenemos como resultado un binario llamado "shell_mod2". Si hacemos ejecutamos el comando ls -l para obtener más información sobre el binario extraído, podemos observar que es un ejecutable:

```
barvch@ubuntu:~/Downloads$ tar -xvf SHELLow
shell_mod2
barvch@ubuntu:~/Downloads$ ls -l
total 28
drwxrwxr-x 7 barvch barvch 4096 Aug 13 18:14 checksec.sh
drwxrwxr-x 2 barvch barvch 4096 Aug 14 16:44 clase
drwxrwxr-x 2 barvch barvch 4096 Aug 13 23:37 dia6
drwxr-xr-x 3 barvch barvch 4096 Nov 25 2008 noip-2.1.9-1
-rwxr-xr-x 1 barvch barvch 7386 Mar 31 2017 shell_mod2
-rw-rw-r-- 1 barvch barvch 2987 Aug 16 18:11 SHELLow
barvch@ubuntu:~/Downloads$
```

Continuando con la tarea de obtener más información inicial, al ejecutar el comando file pero ahora sobre el binario descomprimido, obtenemos el siguiente resultado:

```
barvch@ubuntu:~/Downloads$ file shell_mod2
shell_mod2: ELF, unknown class 113
barvch@ubuntu:~/Downloads$
```

No es de gran ayuda, no nos dice mucha información en sí sobre el binario, por lo que vamos a revisar la existencia de algunas cadenas dentro del binario ejecutando el comando:

```
strings shell_mod2
```

Y como resultado, obtenemos un par de cadenas interesantes:

```
barvch@ubuntu:~/Downloads$ strings shell_mod2
ELFquitaestoparaquefuncioneelprograma
/lib64/ld-linux-x86-64.so.2
libc.so.6
puts
__libc_start_main
__gmon_start__
GLIBC_2.2.5
UH-H
ffffff.
Baia, baH
ia ... sH
i que haH
s llegadH
o lejos
It's timH
e to craH
ckme MisH
s/Mr RevH
erse EngH
inner ;)H
[]A\A]A^A_
;*3$"
87654-32109-87654-321DRO-WSSAP
SHELLow was here :P
```

```
msg1
__FRAME_END__
__JCR_END__
__init_array_end
__DYNAMIC
__init_array_start
__GLOBAL_OFFSET_TABLE__
__libc_csu_fini
ITM_deregisterTMCloneTable
data_start
puts@@GLIBC_2.2.5
edata
fini
__libc_start_main@@GLIBC_2.2.5
data_start
__gmon_start__
dso_handle
IO_stdin_used
msg2
__libc_csu_init
end
start
shellcode
__bss_start
main
__Jv_RegisterClasses
__TMC_END__
ITM_registerTMCloneTable
init
barvch@ubuntu:~/Downloads$
```

Podemos observar que hasta arriba, encontramos una cadena que dice “quitaestoparaquefuncioneelprograma” y que abajo podemos comenzar a ver algunos mensajes como de bienvenida en el binario, además de poder algo parecido a un serial y al final, una firma de SHELLow.

Obteniendo más información del binario, vamos a correr la herramienta llamada “checksec” para poder observar si tiene algún tipo de protección, como por ejemplo canarios o algo semejante:

Y obtenemos el siguiente resultado por parte de checksec:

Bien, ahora vamos intentar correr el binario, si intentamos ejecutarlo recién descomprimido, nos manda el siguiente error:

Recordemos que existe una cadena que dice “quitaestoparaquefuncioneelprograma”, por lo que vamos a abrir el binario con vim y revisar en dónde está esa cadena para quitarla:

Y nos topamos con la cadena en cuestión:

Acto seguido, se editó el binario para eliminar esa cadena y dejándolo de la siguiente manera:

Y ahora, si intentamos ejecutarlo, podemos observar lo siguiente:

```
barvch@ubuntu:~/Downloads$ ./shell_mod2
Baia, baia ... si que has llegado Tejos
It's time to crackme Miss/Mr Reverse Enginner ;)
```

Podemos encontrar las dos cadenas que vimos en la salida de strings y hemos logrado hacer que el binario corra de manera normal, pero podemos ver que el binario parece que se queda esperando una entrada por nuestra parte o algo así, debido a que no nos indica una instrucción o algo así, aún debemos de obtener más información para entender qué está pasando en el binario.

Análisis del funcionamiento del binario

Revisando con más profundidad el binario que hemos logrado ejecutar, se encontró que cuando el binario se queda 'pasmado', en el trasfondo, se ha abierto un puerto (39321), el cual se encuentra a la escucha y es producto de la ejecución el binario.

Ejecutando:

```
netstat -tulpn
```

Se obtiene lo comentado en el párrafo de arriba:

```
barvch@ubuntu:~/Downloads$ netstat -tulpn
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (only servers)

```

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State	PID/Program name
tcp	0	0	127.0.1.1:53	0.0.0.0:*	LISTEN	-
tcp	0	0	127.0.0.1:631	0.0.0.0:*	LISTEN	-
tcp	0	0	0.0.0.0:39321	0.0.0.0:*	LISTEN	31158/shell_mod2
tcp6	0	0	:::1:631	:::*	LISTEN	-
udp	0	0	0.0.0.0:50600	0.0.0.0:*	-	-
udp	0	0	0.0.0.0:631	0.0.0.0:*	-	-
udp	0	0	127.0.1.1:53	0.0.0.0:*	-	-
udp	0	0	0.0.0.0:68	0.0.0.0:*	-	-
udp	0	0	0.0.0.0:5353	0.0.0.0:*	-	-
udp	0	0	0.0.0.0:39242	0.0.0.0:*	-	-
udp6	0	0	:::51686	:::*	-	-
udp6	0	0	:::5353	:::*	-	-

```
barvch@ubuntu:~/Downloads$
```

Podemos comenzar a especular sobre lo que está pasando y porqué el binario se queda pasmado. Parece ser que cuando ejecutamos el binario, se abre el puerto en cuestión (como si éste fuera un servidor) y el binario espera una respuesta de alguien que se conecte por ese puerto. Podemos empezar a especular también que tal vez hace esto por medio de sockets y

que el binario hace eso; crear un socket a la escucha por ese puerto y que parece que se queda pasmado porque está esperando que alguien le mande algo por ese puerto, por lo que ahora, vamos intentar conectarnos a ese puerto como si fuéramos un cliente utilizando la herramienta netcat. Ejecutamos:

```
netcat 127.0.0.1 39321
```

```
barvch@ubuntu:~/Downloads$ netcat 127.0.0.1 39321
```

Y como resultado, podemos ver que se establece una conexión a ese puerto de manera satisfactoria y que ahora, podemos ingresar una cadena o cualquier cosa para probar qué pasa por parte del servidor; si ingresamos la cadena “hola”, ocurre lo siguiente:

Por parte del cliente:

```
barvch@ubuntu:~/Downloads$ netcat 127.0.0.1 39321
hola
barvch@ubuntu:~/Downloads$
```

En el lado del servidor:

```
barvch@ubuntu:~/Downloads$ ./shell_mod2
Baia, baia ... si que has llegado lejos
It's time to crackme Miss/Mr Reverse Enginner ;)
Segmentation fault (core dumped)
barvch@ubuntu:~/Downloads$
```

Observamos que de manera inmediata, se muere la conexión tanto en el cliente como en el servidor, como si hubiésemos ingresado una cadena errónea o algo parecido. Por parte del servidor, podemos ver que nos da un Segmentation Fault y por parte del cliente, se cierra la conexión al puerto; debido a que el servidor ha dejado de funcionar y no se puede mantener viva la conexión.

Tenemos que investigar más sobre qué es lo que está haciendo el binario, por lo vamos a examinarlo con **gdb**; el **debugger de GNU** para poder revisar con más detalle, qué está pasando y porqué nos está arrojando Segmentation Fault cuando ingresamos el “hola”.

Iniciamos el análisis con gdb ejecutando el comando:

```
gdb -q shell_mod2
```

Y ahora, vamos a proceder a enlistar las funciones que gdb encuentre dentro del binario ejecutando:

```
info functions
```

Y obtenemos como resultado:

```
barvch@ubuntu:~/Downloads$ gdb -q shell_mod2
Reading symbols from shell_mod2...(no debugging symbols found)...done.
(gdb) info functions
All defined functions:

Non-debugging symbols:
0x00000000004003a8  _init
0x00000000004003e0  puts@plt
0x00000000004003f0  __libc_start_main@plt
0x0000000000400400  __gmon_start__@plt
0x0000000000400410  _start
0x0000000000400440  deregister_tm_clones
0x0000000000400480  register_tm_clones
0x00000000004004c0  __do_global_ctors_aux
0x00000000004004e0  frame_dummy
0x0000000000400506  main
0x00000000004005e0  __libc_csu_init
0x0000000000400650  __libc_csu_fini
0x0000000000400654  _fini
(gdb) █
```

Podemos ver, que gdb sólo nos indica que hay una función en el binario; la función main; esto es extraño, debido a que ya vimos que existen más cadenas a la hora de usar strings, entonces, vamos a ver qué está pasando dentro de main para revisar en dónde están esas cadenas que vimos (msg1, msg2 y shellcode).

Con ayuda de otra herramienta que se llama Hopper, podemos consultar estos strings que encontramos. Se ha montado el binario dentro de Hopper y se ha detectado lo siguiente para “msg1”:

Parece ser como un serial. La cadena es: 87654-32109-87654-321DRO-WSSAP, la cuál también fue reportada cuando se analizó el binario con strings.

Podemos concluir que el contenido de msg1 y msg2, son las cadenas que fueron arrojadas por strings y con ayuda de Hopper, podemos asegurar esta asociación.

Ahora, para continuar con el análisis del contenido de main en gdb, vamos a ejecutar el siguiente comando haciendo que podamos tener un desglose de la función main:

```
layout asm
```

Y ahora, podemos observar de manera más eficaz todas las instrucciones que están pasando en main:

```
b+ 0x400506 <main>      push    rbp
0x400507 <main+1>      mov     rbp, rsp
0x40050a <main+4>      sub     rsp, 0x70
0x40050e <main+8>      movabs  rax, 0x6162202c61696142
0x400518 <main+18>     mov     QWORD PTR [rbp-0x30], rax
0x40051c <main+22>     movabs  rax, 0x73202e2e2e206169
0x400526 <main+32>     mov     QWORD PTR [rbp-0x28], rax
0x40052a <main+36>     movabs  rax, 0x6168206575712069
0x400534 <main+46>     mov     QWORD PTR [rbp-0x20], rax
0x400538 <main+50>     movabs  rax, 0x646167656c6c2073
0x400542 <main+60>     mov     QWORD PTR [rbp-0x18], rax
0x400546 <main+64>     movabs  rax, 0x736f6a656c206f
0x400550 <main+74>     mov     QWORD PTR [rbp-0x10], rax
0x400554 <main+78>     movabs  rax, 0x6d69742073277449
0x40055e <main+88>     mov     QWORD PTR [rbp-0x70], rax
0x400562 <main+92>     movabs  rax, 0x617263206f742065
0x40056c <main+102>    mov     QWORD PTR [rbp-0x68], rax
0x400570 <main+106>    movabs  rax, 0x73694d20656d6b63
0x40057a <main+116>    mov     QWORD PTR [rbp-0x60], rax
0x40057e <main+120>    movabs  rax, 0x76655220724d2f73
0x400588 <main+130>    mov     QWORD PTR [rbp-0x58], rax
0x40058c <main+134>    movabs  rax, 0x676e452065737265
0x400596 <main+144>    mov     QWORD PTR [rbp-0x50], rax
```

Yendo un poco más abajo, podemos darnos cuenta que en <main+173> y en <main+185>, encontramos la instrucción **puts**, la cual sirve para mandar alguna cadena en particular a la salida estándar. En este caso, todas las operaciones que aparecen hasta antes de

<main+173>, son las reservaciones en memoria que se estaban haciendo para imprimir los mensajes que se nos presentan cuando se ejecuta el binario:

- **<main+173>**: “Baia, baia ... si que has llegado lejos”
- **<main+185>**: “It's time to crackme Miss/Mr Reverse Enginner ;)”

Ahora, justo en la instrucción que sigue (<main+190>), podemos observar en gdb lo siguiente:

```
05bf <main+185>    call    0x4003e0 <puts@plt>
05c4 <main+190>    mov     QWORD PTR [rbp-0x8],0x600a40
05cc <main+198>    mov     rdx,QWORD PTR [rbp-0x8]
05d0 <main+202>    mov     eax,0x0
05d5 <main+207>    call    rdx
05d7 <main+209>    leave
```

Se está moviendo el valor 0x600a40 a rbp-0x8, pero lo curioso es si en gdb ejecutamos:

```
x/s 0x600a40
```

Podemos observar el siguiente output:

```
(gdb) x/s 0x600a40
0x600a40 <shellcode>:  "\272|5\356\317\332\334\331t$\364^1m:\203\356\374\061V\017\003Vs\327\033\376}\340\002\376Ge\311
^CS\302e\033\374\266g\375\305\003L\003S\022\330]10\361;\337\276\251\264\032\237\371\370+\032J\327<!\372x\v\030(4\016\332
&]E\217  \020\206\024\377\252\{\277\202QJ\027\235\070}_b\001\226\251\324\001\220K\353\200\352t\363\212\244E:\n\365\
243<7\372\206I5\202\030\267E{\213\301\267\064\235\344\211\307)\251\024\374 6[J4\237\201\277~.\235\265c\i\310C\024T*{\
245\246\337-\355+K\352\341{B;H~\354\212kw\350\346*", <incomplete sequence \371>
(gdb)
```

```
<shellcode>:
"\272|5\356\317\332\334\331t$\364^1m:\203\356\374\061V\017\003Vs\327\033\376}\340\002\376Ge\311
^CS\302e\033\374\266g\375\305\003L\003S\022\330]10\361;\337\276\251\264\032\237\371\370+\032J\327<!\372x\v\030(4\016\332
&]E\217  \020\206\024\377\252\{\277\202QJ\027\235\070}_b\001\226\251\324\001\220K\353\200\352t\363\212\244E:\n\365\
243<7\372\206I5\202\030\267E{\213\301\267\064\235\344\211\307)\251\024\374 6[J4\237\201\277~.\235\265c\i\310C\024T*{\
245\246\337-\355+K\352\341{B;H~\354\212kw\350\346*", <incomplete sequence \371>
```


Podemos ver, que se está mandando a llamar a shellcode, pero que, shellcode no es una función como tal, sino que se está casteando como si fuera una y es por ese motivo, que cuando hicimos **info functions** en gdb, no se veía a la shellcode como una función. Se puede corroborar esto que se menciona con la siguiente captura de pantalla que se obtuvo de Hopper, donde al analizar la función main, se puede ver claramente que se **llama a shellcode**:

```

00000000004005b8      lea      rax, qword [rbp+var_70]
00000000004005bc      mov      rdi, rax
00000000004005bf      call     j_puts
00000000004005c4      mov      qword [rbp+var_8], shellcode
00000000004005cc      mov      rdx, qword [rbp+var_8]
00000000004005d0      mov      eax, 0x0
00000000004005d5      call     rdx
00000000004005d7      leave

```

Entonces, hasta ahora sabemos que main imprime las dos cadenas que vemos cuando se ejecuta el binario y que además, se llama a la 'función oculta' llamada shellcode, la cual aún no analizamos. Falta por descubrir entonces, en dónde es que se está creando el socket para que el binario abra el puerto y quede a la escucha.

Análisis de la función shellcode

Ahora, vamos a analizar qué es lo que ocurre en shellcode para poder entender el funcionamiento del binario. Hasta el momento, hemos visto que llegamos a esta función por un call que se hace en <main+190>, pero aún tenemos que revisar qué pasa dentro de esta función.

Cuando llegamos a esta función en gdb, podemos observar las siguientes instrucciones:

```

> 0x600a40 <shellcode>      mov     edx,0xcfee357c
0x600a45 <shellcode+5>      fcmovb st,st(4)
0x600a47 <shellcode+7>      fnstenv [rsp-0xc]
0x600a4b <shellcode+11>     pop     rsi
0x600a4c <shellcode+12>     xor     ecx,ecx
0x600a4e <shellcode+14>     mov     cl,0x3a
0x600a50 <shellcode+16>     sub     esi,0xffffffffc
0x600a53 <shellcode+19>     xor     DWORD PTR [rsi+0xf],edx
0x600a56 <shellcode+22>     add     edx,DWORD PTR [rsi+0x73]
0x600a59 <shellcode+25>     xlat    BYTE PTR ds:[rbx]
0x600a5a <shellcode+26>     sbb     edi,esi
0x600a5c <shellcode+28>     jge     0x600a3e
0x600a5e <shellcode+30>     add     bh,dh
0x600a60 <shellcode+32>     rex.RXB
0x600a61 <shellcode+33>     gs leave
0x600a63 <shellcode+35>     pop     rsi
0x600a64 <shellcode+36>     rex.XB push r11
0x600a66 <shellcode+38>     ret     0x1b65
0x600a69 <shellcode+41>     cld
0x600a6a <shellcode+42>     mov     dh,0x67
0x600a6c <shellcode+44>     std
0x600a6d <shellcode+45>     (bad)
0x600a70 <shellcode+48>     add     edx,DWORD PTR [rbx+0x12]

```

Para más comodidad, podemos hacer un breakpoint dentro de gdb ahora que estamos situados en la función que nos es de interés ejecutando:

```
b *0x600a40
```

Y de esta forma, llegar más rápido al inicio de la función cuando estemos haciendo pruebas futuras.

Para entender qué está pasando con más detalle dentro de esta función, vamos a ejecutar en gdb:

```
layout regs
```

Y se nos abrirá un panel en la parte posterior de gdb para poder ver con más comodidad, qué está pasando con los registros a la hora que vamos corriendo el programa instrucción por instrucción. Podemos continuar con el flujo del binario ejecutando **si** dentro de gdb.

Algo muy curioso ocurre cuando llegamos a este punto en el flujo del programa:

Register group: general			
rax	0x0	0	
rbx	0x0	0	
rcx	0x3a	58	
rdx	0x1e4178b	31725451	
rsi	0x600a49	6294089	
rdi	0x1	1	
rbp	0x7fffffffde00	0x7fffffffde00	
rsp	0x7fffffffdd90	0x7fffffffdd90	
r8	0x293b2072656e6e69	2971004054881070697	
r9	0x73694d20656d6b63	8316262988452293475	
r10	0x76655220724d2f73	8531315368433364851	

0x600a4b	<shellcode+11>	pop	rsi
0x600a4c	<shellcode+12>	xor	ecx,ecx
0x600a4e	<shellcode+14>	mov	cl,0x3a
0x600a50	<shellcode+16>	sub	esi,0xffffffffc
0x600a53	<shellcode+19>	xor	DWORD PTR [rsi+0xf],edx
0x600a56	<shellcode+22>	add	edx,DWORD PTR [rsi+0x73]
> 0x600a59	<shellcode+25>	xlat	BYTE PTR ds:[rbx]
0x600a5a	<shellcode+26>	sbb	edi,esi
0x600a5c	<shellcode+28>	jge	0x600a3e
0x600a5e	<shellcode+30>	add	bh,dh
0x600a60	<shellcode+32>	rex.RXB	

Estamos parados en la instrucción 0x600a59 y el valor de rcx es de 58, pero si continuamos el recorrido paso por paso, nos daremos cuenta que entramos a una especie de ciclo que no está marcado dentro de las instrucciones de shellcode y que de manera paralela, el valor de rcx se va decrementando cada vez que se entra de nuevo al ciclo invisible:

```

Register group: general
rax      0x0      0
rbx      0x0      0
rcx      0x37     55
rdx      0x60cd7a47 1624078919
rsi      0x600a51 6294097
rdi      0x1      1
rbp      0x7fffffffde00 0x7fffffffde00
rsp      0x7fffffffdd90 0x7fffffffdd90
r8       0x293b2072656e69 2971004054881070697
r9       0x73694d20656d6b63 8316262988452293475
r10      0x76655220724d2f73 8531315368433364851

0x600a4b <shellcode+11> pop    rsi
0x600a4c <shellcode+12> xor    ecx,ecx
0x600a4e <shellcode+14> mov    cl,0x3a
> 0x600a50 <shellcode+16> sub    esi,0xffffffffc
0x600a53 <shellcode+19> xor    DWORD PTR [rsi+0xf],edx
0x600a56 <shellcode+22> add    edx,DWORD PTR [rsi+0x73]
0x600a59 <shellcode+25> xlat    BYTE PTR ds:[rbx]
0x600a5a <shellcode+26> sbb    edi,esi
0x600a5c <shellcode+28> jge    0x600a3e
0x600a5e <shellcode+30> add    bh,dh
0x600a60 <shellcode+32> rex.RXB

native process 36880 In: shellcode
(gdb) si
0x0000000000600a56 in shellcode ()
(gdb) si
0x0000000000600a59 in shellcode ()
(gdb) si
0x0000000000600a50 in shellcode ()
(gdb) si
0x0000000000600a53 in shellcode ()
0x0000000000600a56 in shellcode ()
0x0000000000600a59 in shellcode ()
0x0000000000600a50 in shellcode ()
(gdb)

```

Cada vez que entramos de nuevo al ciclo, se decrementa el valor de rcx y siempre se realiza un xor a rsi+0xf con el valor de edx. Recordemos que el resultado del XOR, queda almacenado dentro del primer operando, por lo que el resultado siempre es almacenado en **rsi+0xf**.

Si consultamos el estado de `rsi+0xf` después de un par de iteraciones dentro del loop (25), podemos ver que tiene el siguiente valor:

```
x/s $rsi+0xf
```

```
Register group: general
rax      0x0      0
rbx      0x0      0
rcx      0x19     25
rdx      0x461f364a 1176450634
rsi      0x600acd 6294221
rdi      0x1      1
rbp      0x7fffffffde00 0x7fffffffde00
rsp      0x7fffffffdd90 0x7fffffffdd90
r8       0x293b2072656e6e69 2971004054881070697
r9       0x73694d20656d6b63 8316262988452293475
r10      0x76655220724d2f73 8531315368433364851

0x600a4b <shellcode+11> pop     rsi
0x600a4c <shellcode+12> xor     ecx,ecx
0x600a4e <shellcode+14> mov     cl,0x3a
0x600a50 <shellcode+16> sub     esi,0xffffffffc
0x600a53 <shellcode+19> xor     DWORD PTR [rsi+0xf],edx
> 0x600a56 <shellcode+22> add     edx,DWORD PTR [rsi+0x73]
0x600a59 <shellcode+25> xlat     BYTE PTR ds:[rbx]
0x600a5a <shellcode+26> sbb     edi,esi
0x600a5c <shellcode+28> jge     0x600a3e
0x600a5e <shellcode+30> add     bh,dh
0x600a60 <shellcode+32> rex.RXB

native process 36880 In: shellcode L?? PC: 0x600a56
0x0000000000600a59 in shellcode ()
0x0000000000600a50 in shellcode ()
0x0000000000600a53 in shellcode ()
0x0000000000600a56 in shellcode ()
0x0000000000600a59 in shellcode ()
0x0000000000600a50 in shellcode ()
0x0000000000600a53 in shellcode ()
0x0000000000600a56 in shellcode ()
(gdb) x/s $rsi+0xf
0x600adc <shellcode+156>: "\365H1\c\\i\310C\024T*{\245\246\337-\355+K\352\341{B;H~\354\212kw\350\346*", <incomplete sequence \371>
(gdb)
```

Prestemos atención a que es muy parecido a lo que vimos al momento de hacer la llamada a shellcode desde main, pero se ha reducido el tamaño. El loop va a acabar hasta que el valor de `rcx` sea 0, parece que por cada iteración, se va quitando un pedazo de la cadena original y se decrementa `rcx`. Si continuamos el proceso hasta que el valor de `rcx` = 4, y de nuevo consultamos el valor de `$rsi+0xf` después de hacer el XOR, obtenemos el siguiente resultado:

```

rcx      0x4      4
rdx      0xc80ba17a      3356205434
rsi      0x600b21 6294305
rdi      0x1      1
rbp      0x7fffffffde00  0x7fffffffde00
rsp      0x7fffffffdd90  0x7fffffffdd90
r8       0x293b2072656e69  2971004054881070697
r9       0x73694d20656d6b63  8316262988452293475
r10      0x76655220724d2f73  8531315368433364851

```

```

0x600a4e <shellcode+14> mov     cl,0x3a
0x600a50 <shellcode+16> sub     esi,0xffffffffc
0x600a53 <shellcode+19> xor     DWORD PTR [rsi+0xf],edx
> 0x600a56 <shellcode+22> add     edx,DWORD PTR [rsi+0xf]
0x600a59 <shellcode+25> loop   0x600a50 <shellcode+16>
0x600a5b <shellcode+27> xor     esi,esi
0x600a5d <shellcode+29> mul     esi
0x600a5f <shellcode+31> inc     esi
0x600a61 <shellcode+33> push    0x2
0x600a63 <shellcode+35> pop     rdi
0x600a64 <shellcode+36> add     al,0x29

```

```

native process 36880 In: shellcode
0x600b2c <shellcode+236>:      "ZVH\273U\303b\246\206,\006^\213f\267.\020\210BE"
gdb) si
0x0000000000600a50 in shellcode ()
gdb) x/s $rsi+0xf
0x600b2c <shellcode+236>:      "ZVH\273U\303b\246\206,\006^\213f\267.\020\210BE"
gdb) si
0x0000000000600a53 in shellcode ()
gdb) si
0x0000000000600a56 in shellcode ()
gdb) x/s $rsi+0xf
0x600b30 <shellcode+240>:      "/bin\206,\006^\213f\267.\020\210BE"
gdb)

```

Comenzamos a ver la cadena “/bin”, por lo que si continuamos con la ejecución del programa y repetimos los pasos de arriba:

```
Register group: general
rax      0x0      0
rbx      0x0      0
rcx      0x3      3
rdx      0x367503a9  913638313
rsi      0x600b25  6294309
rdi      0x1      1
rbp      0x7fffffffde00  0x7fffffffde00
rsp      0x7fffffffdd90  0x7fffffffdd90
r8       0x293b2072656e69  2971004054881070697
r9       0x73694d20656d6b63  8316262988452293475
r10      0x76655220724d2f73  8531315368433364851

0x600a4e <shellcode+14> mov     cl,0x3a
0x600a50 <shellcode+16> sub     esi,0xffffffffc
0x600a53 <shellcode+19> xor     DWORD PTR [rsi+0xf],edx
> 0x600a56 <shellcode+22> add     edx,DWORD PTR [rsi+0xf]
0x600a59 <shellcode+25> loop   0x600a50 <shellcode+16>
0x600a5b <shellcode+27> xor     esi,esi
0x600a5d <shellcode+29> mul     esi
0x600a5f <shellcode+31> inc     esi
0x600a61 <shellcode+33> push   0x2
0x600a63 <shellcode+35> pop     rdi
0x600a64 <shellcode+36> add     al,0x29

native process 36880 In: shellcode
0x600b30 <shellcode+240>:      "/bin\206,\006^\213f\267.\020\210BE"
(gdb) si
0x000000000000600a59 in shellcode ()
(gdb) si
0x000000000000600a50 in shellcode ()
(gdb) si
0x000000000000600a53 in shellcode ()
(gdb) si
0x000000000000600a56 in shellcode ()
(gdb) x/s $rsi+0xf
0x600b34 <shellcode+244>:      "//sh\213f\267.\020\210BE"
(gdb)
```

Y ahora nos encontramos con “/sh”, si continuamos con este ciclo hasta llegar al final, podemos ver que el final es:


```
Register group: general
rax      0x0      0
rbx      0x0      0
rcx      0x1      1
rdx      0x4f47872b      1330087723
rsi      0x600b2d 6294317
rdi      0x1      1
rbp      0x7fffffffde00  0x7fffffffde00
rsp      0x7fffffffdd90  0x7fffffffdd90
r8       0x293b2072656e6e69      2971004054881070697
r9       0x73694d20656d6b63      8316262988452293475
r10      0x76655220724d2f73      8531315368433364851

0x600a4e <shellcode+14> mov     cl,0x3a
0x600a50 <shellcode+16> sub     esi,0xffffffffc
0x600a53 <shellcode+19> xor     DWORD PTR [rsi+0xf],edx
> 0x600a56 <shellcode+22> add     edx,DWORD PTR [rsi+0xf]
0x600a59 <shellcode+25> loop   0x600a50 <shellcode+16>
0x600a5b <shellcode+27> xor     esi,esi
0x600a5d <shellcode+29> mul     esi
0x600a5f <shellcode+31> inc     esi
0x600a61 <shellcode+33> push    0x2
0x600a63 <shellcode+35> pop     rdi
0x600a64 <shellcode+36> add     al,0x29

native process 36880 In: shellcode
0x600b38 <shellcode+248>:      "ST_\260\020\210BE"
(gdb) si
0x0000000000600a59 in shellcode ()
(gdb) si
0x0000000000600a50 in shellcode ()
(gdb) si
0x0000000000600a53 in shellcode ()
(gdb) si
0x0000000000600a56 in shellcode ()
(gdb) x/s $rsi+0xf
0x600b3c <shellcode+252>:      ";\017\005\n"
(gdb) █
```

Cuando salimos del loop, nos damos cuenta que todo el contenido de la función shellcode, “vuelve a la normalidad” y ahora podemos ver con claridad el loop en el que estábamos metidos en las capturas de arriba:

```
Register group: general
rax      0x0      0
rbx      0x0      0
rcx      0x0      0
rdx      0x594c9666      1498191462
rsi      0x0      0
rdi      0x1      1
rbp      0x7fffffffde00      0x7fffffffde00
rsp      0x7fffffffdd90      0x7fffffffdd90
r8       0x293b2072656e6e69      2971004054881070697
r9       0x73694d20656d6b63      8316262988452293475
r10      0x76655220724d2f73      8531315368433364851

0x600a4e <shellcode+14> mov     cl,0x3a
0x600a50 <shellcode+16> sub     esi,0xffffffffc
0x600a53 <shellcode+19> xor     DWORD PTR [rsi+0xf],edx
0x600a56 <shellcode+22> add     edx,DWORD PTR [rsi+0xf]
0x600a59 <shellcode+25> loop    0x600a50 <shellcode+16>
0x600a5b <shellcode+27> xor     esi,esi
> 0x600a5d <shellcode+29> mul     esi
```

Podemos concluir hasta este punto, que se ha creado una shell usando /bin/sh y que las instrucciones ya se encuentran “normales” y ya no se encuentran fuscadas.

Ahora, si revisamos un poco más abajo dentro de shellcode, vemos que se empiezan a realizar algunas llamadas al sistema:

- **<shellcode+38>**: Una línea antes, al toma el valor de 0x29 -> **Creación del socket**
- **<shellcode+60>**: Una línea antes, al toma el valor de 0x31 -> **Binding del socket**
- **<shellcode+65>**: Una línea antes, al toma el valor de 0x32 -> **Listen**
- **<shellcode+69>**: Una línea antes, al toma el valor de 0x2b -> **Accept**

Es importante revisar el valor que está asignado antes de hacer la syscall para poder entender qué syscall se está haciendo. Se puede obtener más información aquí: https://chromium.googlesource.com/chromiumos/docs/+/_master/constants/syscalls.md

Entonces, hasta este punto, ya sabemos el cómo se está creando el socket dentro del sistema. Cuando lleguemos a esa instrucción en gdb, se quedará ‘pasmado’ como ocurría cuando ejecutamos el programa así a secas, pero ahora comprendemos que estamos llegando hasta la instrucción de accept del socket, por lo que está a la espera de que alguien se conecte al socket y le envíe algo para poder proseguir con el flujo.

Validaciones del serial

Ya que llegamos a este punto, comprendemos cómo es que se crea el socket dentro del binario y que está a la espera de que alguien se conecte; las siguientes instrucciones del binario, son simples validaciones que se le hacen a lo que sea mandado por el socket para verificar que sea un serial correcto, por lo que en esta sección, se van a revisar esas validaciones que se hacen a lo que se mandado por el socket para validar que sea un serial correcto.

Validación de longitud (salto de línea)

Esta primea validación, va desde <shellcode+95> hasta <shellcode+112> y es importante tener el cuenta que antes de llegar al <shellcode+95>, el valor de rax es de 10, ésto debido a que se es usado dentro de la validación.

Register group: general			
rax	0xa	10	
rbx	0x0	0	
rcx	0x600a98	6294168	
rdx	0x20	32	
rsi	0x7fffffffdd80	140737488346496	
rdi	0x4	4	
rbp	0x7fffffffde00	0x7fffffffde00	
rsp	0x7fffffffdd80	0x7fffffffdd80	
r8	0x293b2072656e6e69	2971004054881070697	
r9	0x73694d20656d6b63	8316262988452293475	
r10	0x76655220724d2f73	8531315368433364851	

0x600a9d	<shellcode+93>	sub	al,0x40
> 0x600a9f	<shellcode+95>	xor	rcx,rcx
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>

El valor de rax siempre será de 10 durante todo el loop, debido a que ese valor en ASCII es “\n”, es decir, un salto de línea. En <shellcode+95>, se hace que rcx = 0. Lo que se compara en <shellcode+98>, es básicamente preguntar si el carácter con el que se está iterando, es un salto de línea, en caso de que sí, va a <shellcode+108> para comparar el último valor que tiene rcx contra 0x1d(29 en decimal) y en caso de que no sean iguales, va a <shellcode+255> que es donde sucede el segmentation fault que ocurre cuando esta o cualquiera de las otras validaciones falla. En caso de que no se encuentre en el carácter que se está iterando el salto

de línea, incrementa en 1 el valor de rcx y comienza de nuevo el ciclo. Básicamente, el ciclo busca la posición en donde se encuentra un salto de línea, que indica a su vez, el fin de la cadena y su longitud, por lo que podemos decir que esta validación, verifica el número de caracteres presente dentro de lo que sea mandado por el socket. **La longitud del serial debe de ser 29.**

Validación de posición de los guiones (-)

Esta primea validación, va desde <shellcode+118> hasta <shellcode+136> y es importante tener el cuenta que al llegar a <shellcode+118>, se hace un reset a rcx, por lo que ahora vale 0, y acto seguido, se le suman 5.

Register group: general			
rax	0xa	10	
rbx	0x0	0	
rcx	0xb	11	
rdx	0x20	32	
rsi	0x7fffffffdd80	140737488346496	
rdi	0x4	4	
rbp	0x7fffffffde00	0x7fffffffde00	
rsp	0x7fffffffdd80	0x7fffffffdd80	
r8	0x293b2072656e6e69	2971004054881070697	
r9	0x73694d20656d6b63	8316262988452293475	
r10	0x76655220724d2f73	8531315368433364851	

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
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>

En esta validación, se comienza revisando que el quinto caracter, sea un guión (0x2d), en caso de que no, va a <shellcode+255> y da el segmentation fault. Pero en caso de que sí, le suma 6 a cl, por lo que ahora vale 11 y en <shellcode+136>, se compara si el valor de cl es es menor o igual a 11, en caso de que sí, regresa a <shellcode+124> para validar el siguiente guión y hace lo mismo para el tercer guión, cuando llega a la condición en <shellcode+133>, ya no se cumple y por ende, ya no regresa al ciclo y continúa con la siguiente validación. **En el serial, deben de existir 3 guiones en las posiciones 5,11 y 17**

Validación de la suma de los caracteres del serial

Esta primera validación, va desde <shellcode+138> hasta <shellcode+172>.

Register group: general			
rax	0xa	10	
rbx	0x0	0	
rcx	0xb	11	
rdx	0x20	32	
rsi	0x7fffffffdd80	140737488346496	
rdi	0x4	4	
rbp	0x7fffffffde00	0x7fffffffde00	
rsp	0x7fffffffdd80	0x7fffffffdd80	
r8	0x293b2072656e6e69	2971004054881070697	
r9	0x73694d20656d6b63	8316262988452293475	
r10	0x76655220724d2f73	8531315368433364851	

>	0x600ac5	<shellcode+133>	cmp	cl,0x11	
	0x600ac8	<shellcode+136>	jbe	0x600abc	<shellcode+124>
	0x600aca	<shellcode+138>	xor	rcx,rcx	
	0x600acd	<shellcode+141>	mov	cl,0x1c	
	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]	

Antes de entrar a la validación, se hace que nuestro contador (rcx) sea igual a la longitud de la cadena menos uno, y se hace un reset a rax y rbx; con la finalidad de ir de derecha a izquierda, caracter por caracter e irle sumando el valor de cada caracter en ASCII a rax. Va a repetir ese proceso con todos los caracteres y al final repite el ciclo una última vez con el último caracter.

Register group: general				
rax	0x0	0		
rbx	0x52	82		
rcx	0x3	3		
rdx	0x7fffffffdd8c	140737488346508		
rsi	0x7fffffffdd80	140737488346496		
rdi	0x4	4		
rbp	0x7fffffffde00	0x7fffffffde00		
rsp	0x7fffffffdd80	0x7fffffffdd80		
r8	0x293b2072656e6e69	2971004054881070697		
r9	0x73694d20656d6b63	8316262988452293475		
r10	0x76655220724d2f73	8531315368433364851		

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]
0x600ae3	<shellcode+163>	add	rax,rbx
0x600ae6	<shellcode+166>	cmp	rax,0x8e0
0x600aec	<shellcode+172>	jne	0x600b3f <shellcode+255>

Una vez que termina con el primer carácter, se hace una comparación con el valor 0x8e0, que es 2272 en decimal. Básicamente, **la suma de todos los caracteres del serial debe de ser igual a 2272**, sino es igual, el binario irá a la zona donde da el segmentation fault.

Validación bit a bit (TEST)

Esta primera validación, va desde <shellcode+174> hasta <shellcode+210>


r10	0x76655220724d2f73	8531315368433364851
-----	--------------------	---------------------

0x600aee	<shellcode+174>	lea	rdx,[rsp+0xc]
0x600af3	<shellcode+179>	xor	rcx,rcx
0x600af6	<shellcode+182>	mov	cl,0x5
0x600af8	<shellcode+184>	xor	rax,rax
0x600afb	<shellcode+187>	test	BYTE PTR [rdx+rcx*1],0x41
0x600aff	<shellcode+191>	jne	0x600b0a <shellcode+202>
0x600b01	<shellcode+193>	test	BYTE PTR [rdx+rcx*1],0x61
0x600b05	<shellcode+197>	jne	0x600b0a <shellcode+202>
0x600b07	<shellcode+199>	inc	rax
> 0x600b0a	<shellcode+202>	loop	0x600afb <shellcode+187>
0x600b0c	<shellcode+204>	test	rax,0x3

Se guarda en rdx la cadena desde la posición 15, con el fin de analizarla desde el caracter 5 hasta el 1, donde se realiza una operación TEST, la cual hace una operación bit a bit del caracter que esté iterando contra 0x41 (A en ASCII), en caso de pasar el test, vuelve a hacer

otro TEST pero ahora comparándolo contra 0x61(a en ASCII), en caso de pasarlo, se incrementa en uno el valor de rax y se repite el ciclo con los 5 caracteres que serán analizados. Al final del ciclo, se hace un test bit a bit del valor de rax contra 3 y en caso de que el la zero flag no sea 1, irá al apartado donde da segmentation fault.

0x600b0a <shellcode+202>	loop	0x600afb <shellcode+187>
0x600b0c <shellcode+204>	test	rax,0x3
0x600b12 <shellcode+210>	jne	0x600b3f <shellcode+255>
0x600b14 <shellcode+212>	xor	rax,rax
0x600b17 <shellcode+215>	xor	rbx,rbx
0x600b1a <shellcode+218>	xor	rcx,rcx
0x600b1d <shellcode+221>	xor	rdx,rdx
0x600b20 <shellcode+224>	push	0x3
0x600b22 <shellcode+226>	pop	rsi
0x600b23 <shellcode+227>	dec	esi
0x600b25 <shellcode+229>	mov	al,0x21



Una vez que pasemos el último test, hemos completado de manera satisfactoria todas las validaciones que se hacen en la función de shellcode.

Dado que el examen consistía en generar un serial válido con nuestro nombre y ahora que se han identificado todas las validaciones que se hacen dentro de la función, se ha generado el siguiente serial: **RODOL-FOBAR-UC@?>-HGUERRU~~~~**

Para comprobar que nuestro serial es válido, podemos ejecutar el binario y conectarnos con el cliente e ingresar el serial que ha sido generado para mi caso.

Se ejecuta el binario para que esté listo el lado del cliente:

```
barvch@ubuntu:~/Downloads$ ./shell_mod2
Baia, baia ... si que has llegado lejos
It's time to crackme Miss/Mr Reverse Enginner ;)
```

Y ahora, nos conectamos como cliente y le mandamos el serial generado para el examen, el cual ya cumple con todas las validaciones que son realizadas por shellcode para que pueda funcionar y por último, lanzamos un "ls" y "whoami" para verificar tengamos una shell:

```
barvch@ubuntu:~/Downloads$ netcat 127.0.0.1 39321
RODOL-FOBAR-UC@?>-HGUERRU~~~~
ls
InsecureBankv2.apk
SHELLow
android
checksec.sh
clase
dia6
frida-demo.apk
material.tar.gz
noip-2.1.9-1
primer.py
shell_mod2
whoami
barvch
```

Y podemos ver que cuando ejecutamos el comando ls, nos regresa el listado de los archivos que tengo en mi directorio de Downloads y que el resultado del whoami es mi nombre de usuario dentro de ubuntu, por lo que la shell está funcionando de manera correcta y hemos generado un serial válido.