

Primero, se obtuvo el archivo, el cual se observó que estaba comprimido, por lo que intenté antes de extraerlo, mostrar algún contenido del mismo, para ello se empleó el comando **zcat**

```
chicoterry@chicoterry:~/Documents/cert/analysis/examen|
⇒ file SHELLow
SHELLow: gzip compressed data, last modified: Fri Mar 31 19:11:39 2017, from Unix
chicoterry@chicoterry:~/Documents/cert/analysis/examen|
⇒ zcat SHELLow > content
chicoterry@chicoterry:~/Documents/cert/analysis/examen|
⇒
```

Viendo el contenido del archivo, se notó algo extraño, por lo cual lo tuve que remover para asegurarme que todo fuera bien

[illegible]

Se removió manualmente con **vim** y ahora si, se extrajo este archivo con el programa tar:

```
chicoterry@chicoterry:~/Documents/cert/analisis/examen|
⇒ tar -xvf content
```

Lo que resultó en un binario de tipo ELF (64 bits). Se ejecutó y lo primero que se obtuvo fue lo siguiente, un indicio para saber que este era el programa sobre el que se iba a realizar el análisis.

```
, for GNU/Linux 2.6.32, BuildID[sha1]=4ab82577a85ffa8894e95109fb63b
chicoterry@chicoterry:~/Documents/cert/analisis/examen|
⇒ ./shell_mod2
Baia, baia ... si que has llegado lejos
It's time to crackme Miss/Mr Reverse Enginner ;)
```

Con esto, se trató de realizar un análisis (básico) de forma estática, obteniendo primero algunos strings raros que llamaron la atención:

```
It's timH
e to craH
ckme MisH
s/Mr RevH
erse EngH
inner ;)H
[[]A[]A^A_
;*3$"
87654-32109-87654-321DRO-WSSAP
SHELLow was here :P
8}_b
```

```
_end
_start
shellcode
__bss_start
main
_Jv_RegisterClasses
TMC_END
```

Con **objdump -d** se desensambló el código, y se pudo ver solo una llamada importante dentro de la función main, la cual llamaba a lo apuntado por el registro rdx

```

40057a: 48 89 45 a0      mov    %rax,-0x60(%rbp)
40057e: 48 b8 73 2f 4d 20 movabs $0x76655220724d2f73,%rax
400585: 52 65 76
400588: 48 89 45 a8      mov    %rax,-0x58(%rbp)
40058c: 48 b8 65 72 73 65 20 movabs $0x676e452065737265,%rax
400593: 45 6e 67
400596: 48 89 45 b0      mov    %rax,-0x50(%rbp)
40059a: 48 b8 69 6e 6e 65 72 movabs $0x293b2072656e6e69,%rax
4005a1: 20 3b 29
4005a4: 48 89 45 b8      mov    %rax,-0x48(%rbp)
4005a8: c6 45 c0 00      movb   $0x0,-0x40(%rbp)
4005ac: 48 8d 45 d0      lea    -0x30(%rbp),%rax
4005b0: 48 89 c7          mov    %rax,%rdi
4005b3: e8 28 fe ff ff   callq  4003e0 <puts@plt>
4005b8: 48 8d 45 90      lea    -0x70(%rbp),%rax
4005bc: 48 89 c7          mov    %rax,%rdi
4005bf: e8 1c fe ff ff   callq  4003e0 <puts@plt>
4005c4: 48 c7 45 f8 40 0a 60 movq    $0x600a40,-0x8(%rbp)
4005cb: 00
4005cc: 48 8b 55 f8      mov    -0x8(%rbp),%rdx
4005d0: b8 00 00 00 00   mov    $0x0,%eax
4005d5: ff d2           callq  *(%rdx)
4005d7: c9             leaveq
4005d8: c3             retq
4005d9: 0f 1f 80 00 00 00 00 nopl   0x0(%rax)

```

Ya que dentro de main, todo lo demás correspondía (a grandes rasgos), a las cadenas que se imprimían en la salida estándar.

Ahora, mediante el comando strace podremos listar llamadas al sistema que se realizan, estas nos dan mayor información:

```

[0] <> strace -fxi ./shell_mod2
[00007f71f8c27677] execve("./shell_mod2", ["/shell_mod2"], [/* 73 vars */]) = 0
[00007f5f2bdc7879] brk(NULL) = 0x1983000
[00007f5f2bdc85a7] access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
[00007f5f2bdc869a] mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f5f2bfd5000
[00007f5f2bdc85a7] access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
[00007f5f2bdc8547] open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
[00007f5f2bdc84d2] fstat(3, {st_mode=S_IFREG|0644, st_size=144608, ...}) = 0
[00007f5f2bdc869a] mmap(NULL, 144608, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f5f2bfa7000
[00007f5f2bdc8647] close(3) = 0
[00007f5f2bdc85a7] access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
[00007f5f2bdc8547] open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
[00007f5f2bdc8567] read(3, "\xf7\xf5\xf4\x4c\x46\x02\x01\x01\x03\x00\x00\x00\x00\x00\x00\x03\x00\x3e\x00\x01\x00\x00\x00", 832) = 832
[00007f5f2bdc84d2] fstat(3, {st_mode=S_IFREG|0755, st_size=1689360, ...}) = 0
[00007f5f2bdc869a] mmap(NULL, 3795360, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f5f2baf0000
[00007f5f2bdc8737] mprotect(0x7f5f2bba4000, 2097152, PROT_NONE) = 0
[00007f5f2bdc869a] mmap(0x7f5f2bda4000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x195000) = 0x7f5f2bda4000
[00007f5f2bdc869a] mmap(0x7f5f2bdaa000, 14752, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f5f2bdaa000
[00007f5f2bdc8647] close(3) = 0
[00007f5f2bdafeb8] arch_prctl(ARCH_SET_FS, 0x7f5f2bfd7500) = 0
[00007f5f2bdc8737] mprotect(0x7f5f2bda4000, 16384, PROT_READ) = 0
[00007f5f2bdc8737] mprotect(0x7f5f2bfd2000, 4096, PROT_READ) = 0
[00007f5f2bdc8717] munmap(0x7f5f2bfa7000, 144608) = 0
[00007f5f2baea0d2] fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 3), ...}) = 0
[00007f5f2baefcd9] brk(NULL) = 0x1983000
[00007f5f2baefcd9] brk(0x19a4000) = 0x19a4000
[00007f5f2baea760] write(1, "Baia, baia ... si que has llegad...", 40Baia, baia ... si que has llegado lejos) = 40
[00007f5f2baea760] write(1, "It's time to crackme Miss/Mr Rev...", 49It's time to crackme Miss/Mr Reverse Enginner ;) = 49
[0000000000600a68] socket(PF_INET, SOCK_STREAM, IPPROTO_IP) = 3
[0000000000600a7e] bind(3, {sa_family=AF_INET, sin_port=htons(39321), sin_addr=inet_addr("0.0.0.0")}, 16) = 0
[0000000000600a83] listen(3, 0) = 0

```


Llama la atención una llamada a la función socket, que parece bindea al puerto 39321, esto por desgracia no lo noté antes, por lo que corroboramos si efectivamente logra abrir un puerto (lo cual es muy probable ya que no corre sobre un puerto privilegiado).

Para ello se usa el comando **netstat** mientras el programa se ejecuta:

```
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 127.0.0.1:5939          0.0.0.0:*               LISTEN      1599/teamviewerd
tcp        0      0 192.168.122.1:53        0.0.0.0:*               LISTEN      1935/dnsmasq
tcp        0      0 127.0.1.1:53            0.0.0.0:*               LISTEN      1294/dnsmasq
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      1156/sshd
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN      17029/cupsd
tcp        0      0 0.0.0.0:39321           0.0.0.0:*               LISTEN      1066/shell_mod2
tcp        0      0 0.0.0.0:902             0.0.0.0:*               LISTEN      1797/vmware-authdla
tcp6       0      0 :::27017                 :::*                    LISTEN      30351/docker-proxy
tcp6       0      0 :::6379                  :::*                    LISTEN      30677/docker-proxy
tcp6       0      0 :::2223                  :::*                    LISTEN      31536/docker-proxy
tcp6       0      0 :::80                    :::*                    LISTEN      32730/docker-proxy
tcp6       0      0 :::1234                  :::*                    LISTEN      30043/docker-proxy
```

Por lo que se corrobora a través del program name, que la ejecución abre un puerto. Dicho esto, nos intentamos conectar:

```
udp6       0      0 :::56023                 :::*
chicoterry@chicoterry:~/Documents/cert/analysis/examen|
⇒ nc 127.0.0.1 39321
fsfdssfds
chicoterry@chicoterry:~/Documents/cert/analysis/examen|
⇒
```

La conexión se realiza de forma exitosa, sin embargo al dar un salto de línea, se cierra la ejecución y termina el programa con **segmentation fault**.

Ahora, se utiliza gdb para tratar de ver el punto en el que el programa “muere”.

rax	0x0	0	rbx	0x0	0
rcx	0x7ffffb15760	140737348982624	rdx	0x600a40	6294080
rsi	0x601010	6295568	rdi	0x0	0
rbp	0x7fffffff970	0x7fffffff970	rsp	0x7fffffff900	0x7fffffff900
r8	0x7ffff7fce700	140737353934592	r9	0x0	0
r10	0x7ffff7dd3b58	140737351859032	r11	0x246	582
r12	0x400410	4195344	r13	0x7fffffffda50	140737488345680
r14	0x0	0	r15	0x0	0


```

0x4005c4 <main+190>    mov     QWORD PTR [rbp-0x8],0x600a40
0x4005cc <main+198>    mov     rdx,QWORD PTR [rbp-0x8]
0x4005d0 <main+202>    mov     eax,0x0
> 0x4005d5 <main+207>    call    rdx
0x4005d7 <main+209>    leave
0x4005d8 <main+210>    ret
0x4005d9              nop     DWORD PTR [rax+0x0]
0x4005e0 <__libc_csu_init> push    r15

```

native process 1736 In: main L?? P

0x0000000004005b8 in main ()

0x0000000004005bc in main ()

0x0000000004005f0 in main ()

Se puede ver como al parecer se salta a una función en tiempo de ejecución (o al menos eso parece), al llamar a un registro de propósito general. Dando un paso más se puede ver que salta a otra sección del programa.

Analizando la parte de **shellcode** se pueden notar reiteradas comparaciones, las cuales en caso de no cumplirse saltan a la finalización del programa de manera abrupta, por lo que son de interés dichas comparaciones, para poder seguir ejecutando el programa y evitar que “trueene”.

rcx	0x600a87	6294151	rdx	0x10	16
rsi	0x0	0	rdi	0x3	3
rbp	0x7fffffff970	0x7fffffff970	rsp	0x7fffffff8f8	0x7fffffff8f8
r8	0x7ffff7fce700	140737353934592	r9	0x0	0
r10	0x7ffff7dd3b58	140737351859032	r11	0x202	514
r12	0x400410	4195344	r13	0x7fffffffda50	140737488345680
r14	0x0	0	r15	0x0	0


```

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
0x600aa5 <shellcode+101>   je      0x600aac <shellcode+108>
0x600aa7 <shellcode+103>   inc     rcx
0x600aaa <shellcode+106>   jmp     0x600aa2 <shellcode+98>

```

native process 1965 In: shellcode

Se observa primero la operación 0x4a – 0x40, lo que resulta en valor 10 decimal, que equivale a un line feed (nueva línea), por lo que si no se lee tal, el registro rcx ira incrementando en 1 para leer lo que apuntaba el tope de la pila y hacia abajo de la misma.

Luego se compara que en la posición 5, 11 y cada 6 caracteres, se encuentre un guión (0x2d), de lo contrario, termina el programa

r12	0x400410	4195344	r13	0x7fffffffda50
r14	0x0	0	r15	0x0
0x600ab6	<shellcode+118>	xor	rcx, rcx	
0x600ab9	<shellcode+121>	add	c1, 0x5	
0x600abc	<shellcode+124>	cmp	BYTE PTR [rsp+rcx*1], 0x2d	
0x600ac0	<shellcode+128>	jne	0x600b3f <shellcode+255>	
0x600ac2	<shellcode+130>	add	c1, 0x6	
0x600ac5	<shellcode+133>	cmp	c1, 0x11	
0x600ac8	<shellcode+136>	jbe	0x600abc <shellcode+124>	
0x600aca	<shellcode+138>	xor	rcx, rcx	

Ahora se van sumando todos los valores de los caracteres, de la cadena ingresada, y una vez obtenidos se espera que la suma de ellos resulte en 0x8e0 (2272 en decimal).

```

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]
0x600ae3 <shellcode+163>    add    rax, rbx
0x600ae6 <shellcode+166>    cmp    rax, 0x8e0

```

Por tal motivo, con ayuda de python y siguiendo el formato que apareció con el comando strings, se buscará un serial valido, el cual resultó en:

!j?or-g}e]f-e"rr}-u#s\$%-a&'(

```

chicoterry@chicoterry:~|⇒ nc 127.0.0.1 39321
!j?or-g}e]f-e"rr}-u#s$%-a&'(
pwd
/home/chicoterry/Documents/cert/analysis/examen
id
uid=1000(chicoterry) gid=1000(chicoterry) groups=1000(chicoterry),4(adm),24(cdrom),27(s
are),129(vboxusers),131(ubridge),132(libvirtd),998(docker)
echo foo bar
foo bar
ls
SHELLow
content
pay.py
res
serial.py
shell_mod2

```

Bibliografía

Burford, S (2002). Reverse Engineering Linux ELF Binaries on the x86 Platform. The University of Adelaide.

Truerandom.bid