

Exámen

Análisis estático

Se descargo el archivo *SHELLow* y es un tipo de archivo gzip, como se muestra en la figura 1.

```
[user@parrot]~[~/Downloads]
$file SHELLow
SHELLow: gzip compressed data, last modified: Fri Mar 31 19:11:39 2017, from Unix, original size 10240
```

Figura 1. Tipo de archivo *SHELLow*.

Una vez sabiendo el tipo de archivo que es se procedió a crear una copia con el nombre *SHELLow.gz* y con el comando **gunzip** se descomprimió el archivo como se muestra en la figura 2.

```
[user@parrot]~[~/Documents/Vulnerabilidades/exa]
$cp SHELLow SHELLow.gz
[user@parrot]~[~/Documents/Vulnerabilidades/exa]
$gunzip SHELLow.gz
gzip: SHELLow already exists; do you wish to overwrite (y or n)? y
```

Figura 2. Copiar y descomprimir el archivo *SHELLow*.

Una vez descomprimido el archivo, obtenemos el archivo *shell_mod2*, al cual al hacerle un **file**, como se muestra en la figura 3, nos dice que es un archivo de tipo “ELF, unknown class 113”.

```
[root@parrot]~[~/home/user/Documents/Vulnerabilidades/exa]
#file shell_mod2
shell_mod2: ELF, unknown class 113
```

Figura 3. Tipo de archivo *shell_mod2*.

Ahora se procede con el comienzo del análisis estatico, al utilizar el comando **strings** mostrado en la figura 4, se observa que la primera cadena que aparece es la mostrada en la figura 5.

```
[user@parrot]~[~/Documents/Vulnerabilidades/exa]
$strings shell_mod2
```

Figura 4. Comando strings.

```
ELFquitaestoparaquefuncioneelprograma
```

Figura 5. Primera cadena al ejecutar el comando strings.

La cadena mostrada nos da un indicio que debemos quitar “quitaestoparaquefuncioneelprograma”, se procede a ejecutar el comando **readelf** con la bandera para la cabecera del archivo *shell_mod2*, mostrado en la figura 6.

```
[user@parrot]~/Documents/Vulnerabilidades/exa]
$readelf --file-header shell_mod2
ELF Header:
  Magic:   7f 45 4c 46 71 75 69 74 61 65 73 74 6f 70 61 72 19d3172312122-001.zip
  Class:   ELF64
  Data:    little endian
  Version: 0
  OS/ABI:  UNIX: Linux
  ABI Version: 0
  Type:    EXEC (Executable file)
  Machine: x86_64
  Version: 0
  Entry point address: 0x656e6f69
  Start of program headers: 1919970405 (bytes into file)
  Start of section headers: 1634887535 (bytes into file)
  Flags:   0x0
  Size of this header: 52 (bytes)
  Size of program headers: 0 (bytes)
  Number of program headers: 0
  Size of section headers: 0 (bytes)
  Number of section headers: 0
  Section header string table index: 2 <corrupt: out of range>
readelf: Warning: possibly corrupt ELF file header - it has a non-zero section header offset, but no section headers
readelf: Warning: possibly corrupt ELF header - it has a non-zero program header offset, but no program headers
```

Figura 6. Cabecera del archivo *shell_mod2*.

Observamos que nos muestra un mensaje, el cual dice que la cabecera del archivo ELF esta dañado, lo que nos indica que el mensaje al realizar el comando **strings** tiene razón hay que quitar esa cadena para que el programa funcione bien, para quitar esa cadena utilizamos el programa **ghex**, el cual nos permite editar el archivo con los hexadecimales, en la figura 7, se muestra la cadena con el mensaje y en la figura 8 ya se muestra sin el mensaje, este ultimo lo guardaremos con el nombre de *analizar*.

```
File Edit View Windows Help
00000000 7f 45 4c 46 71 75 69 74 61 65 73 74 6f 70 61 72 ELFquitaestopar
00000001 71 75 65 66 75 6e 63 69 6f 6e 65 65 6c 70 72 a quefuncioneelpr
00000002 06 67 72 61 6d 61 02 01 01 00 00 00 00 00 00 00 ograma.....
00000003 00 00 02 00 3e 00 01 00 00 00 10 04 40 00 00 00 ..>.....@...
00000004 00 00 00 00 00 00 00 00 00 00 38 15 00 00 00 00 .@.....8....
00000005 00 00 00 00 00 00 40 00 38 00 08 00 40 00 1e 00 .....@.8...@...
00000006 01 00 06 00 00 00 05 00 00 00 40 00 00 00 00 00 .....@.....
00000007 00 00 40 00 00 00 00 00 00 00 40 00 40 00 00 00 ..@.....@.....
00000008 00 00 01 00 00 00 00 00 00 00 c0 01 00 00 00 00 .....@.....
00000009 00 00 08 00 00 00 00 00 00 00 03 00 00 00 04 00 .....@.....
```

Figura 7. Hexadecimal con la cadena.

```
00000000 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00 ELF.....
00000001 00 00 3e 00 01 00 00 00 10 04 40 00 00 00 00 00 ..>.....@...
00000002 00 00 00 00 00 00 00 00 00 00 38 15 00 00 00 00 .@.....8....
00000003 00 00 00 40 00 38 00 08 00 40 00 1e 00 1b 00 .....@.8...@...
00000004 00 00 05 00 00 00 40 00 00 00 00 00 00 00 00 00 .....@.....
00000005 00 40 00 00 00 00 40 00 40 00 00 00 00 00 00 00 ..@.....@.....
00000006 00 01 00 00 00 00 c0 01 00 00 00 00 00 00 00 00 .....@.....
00000007 00 00 00 00 00 00 03 00 00 00 04 00 00 00 00 00 .....@.....
00000008 02 00 00 00 00 00 00 00 02 40 00 00 00 00 00 00 .....@.....
00000009 02 40 00 00 00 00 1c 00 00 00 00 00 00 00 00 00 ..@.....
```

Figura 8. Hexadecimal sin la cadena.

Una vez cambiado el hexadecimal, procedemos a realizar de nuevo con el comando **file** a *analizar* para ver que tipo de archivo es, como se muestra en la figura 9.

```
[user@parrot]~/Documents/Vulnerabilidades/exa]
$file analizar
analizar: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=4ab82577a85ffa8894e95109fb63bdd2f199903f, not stripped
```

Figura 9. File a *analizar*.

Al observar la salida del comando **file** a *analizar* observamos que nos muestra que es un ejecutable de 64 bits, con ligado dinámico, con forma de almacenamiento Little-endian.

Procedemos a realizar de nuevo el comando **readelf** con la bandera para ver la cabecera, como se muestra en la figura 10 y observamos que ahora nos muestra más datos.

```
[*]-[user@parrot]-[~/Documents/Vulnerabilidades/eva]
$readelf -h analizar
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:   ELF64
  Data:      2's complement, little endian
  Version: 1 (current)
  OS/ABI:   UNIX - System V
  ABI Version: 0
  Type:     EXEC (Executable file)
  Machine:  Advanced Micro Devices X86-64
  Version:  0x1
  Entry point address: 0x400410
  Start of program headers: 64 (bytes into file)
  Start of section headers: 5432 (bytes into file)
  Flags:     0x0
  Size of this header: 64 (bytes)
  Size of program headers: 56 (bytes)
  Number of program headers: 8
  Size of section headers: 64 (bytes)
  Number of section headers: 30
  Section header string table index: 27
```

Figura 10. Readelf al archivo *analizar*.

Ahora volvemos a ejecutar el comando **strings** sobre el archivo *analizar* para observar la salida y al analizar la salida, se observan cadenas de texto que pueden dar indicios al funcionamiento del programa, estas cadenas de texto están encerradas en cuadros de color rojo en la figura 11 y encerrado en un cuadro azul parece ser un serial.

```
[*]-[user@parrot]-[~/Documents/Vulnerabilidades/eva]
$strings analizar
/lib64/ld-linux-x86-64.so.2
libc.so.6
puts
__libc_start_main
__gmon_start__
GLIBC_2.2.5
UH-H
ffff.
8ala, baH
ia ... sH
i que haH
s llegadH
o lejos
It's timH
e to craH
ckme Mish
s/Mr RevH
erse EngH
inner ;)H
[]A\A]A^A
;*3$"
87654-32109-87654-32109-WSSAP
SHELLow was here :P
8} b
6[j4
{B;H-
GCC: (Debian 4.9.2-10) 4.9.2
GCC: (Debian 4.8.4-1) 4.8.4
.symtab
.strtab
.shstrtab
.interp
.note.ABI-tag
.note.gnu.build-id
.gnu.hash
.dynsym
.dynstr
.gnu.version
```

Figura 11. Análisis de salida del comando strings a *analizar*.

Ahora con el comando **readelf** con la bandera para ver las secciones de la cabecera del archivo *analizar*, mostrado en la figura 12, observamos las secciones de la cabecera, de estas las que nos interesan son *.data* ya que es ahí donde están las variables inicializadas del programa, *.rodata* que son los datos de solo lectura (cadenas) y *.text* que es la parte ejecutable del programa.

```

[user@parrot]~[~/Documents/Vulnerabilidades/eva]
$ readelf --section-headers analizar
There are 30 section headers, starting at offset 0x1538:

Section Headers:
[Nr] Name                Type              Address            Offset
     Size              EntSize          Flags    Link    Info   Align
[ 0] .null                 NULL              0000000000000000   00000000
     0000000000000000  0000000000000000   0      0      0
[ 1] .interp                PROGBITS          0000000000400200   00000200
     000000000000001c  0000000000000000   A      0      0      1
[ 2] .note.ABI-tag          NOTE              000000000040021c   0000021c
     0000000000000020  0000000000000000   A      0      0      4
[ 3] .note.gnu.build-id     NOTE              000000000040023c   0000023c
     0000000000000024  0000000000000000   A      0      0      4
[ 4] .gnu.hash              GNU_HASH          0000000000400260   00000260
     000000000000001c  0000000000000000   A      5      0      8
[ 5] .dynsym                DYNSYM            0000000000400280   00000280
     0000000000000060  0000000000000018   A      6      1      8
[ 6] .dynstr                STRTAB            00000000004002e0   000002e0
     000000000000003d  0000000000000000   A      0      0      1
[ 7] .gnu.version            VERSYM            000000000040031e   0000031e
     0000000000000008  0000000000000002   A      5      0      2
[ 8] .gnu.version_r          VERNEED           0000000000400328   00000328
     0000000000000020  0000000000000000   A      6      1      8
[ 9] .rela.dyn              RELA               0000000000400348   00000348
     0000000000000018  0000000000000018   A      5      0      8
[10] .rela.plt              RELA               0000000000400360   00000360
     0000000000000048  0000000000000018   AI     5     12     8
[11] .init                  PROGBITS          00000000004003a8   000003a8
     000000000000001a  0000000000000000   AX     0      0      4
[12] .plt                   PROGBITS          00000000004003d0   000003d0
     0000000000000040  0000000000000010   AX     0      0     16
[13] .text                  PROGBITS          0000000000400410   00000410
     0000000000000242  0000000000000000   AX     0      0     16
[14] .fini                  PROGBITS          0000000000400654   00000654
     0000000000000009  0000000000000000   AX     0      0      4
[15] .rodata                PROGBITS          0000000000400660   00000660
     0000000000000004  0000000000000004   AM     0      0      4
[16] .eh_frame_hdr          PROGBITS          0000000000400664   00000664
     0000000000000034  0000000000000000   A      0      0      4
[17] .eh_frame              PROGBITS          0000000000400698   00000698
     00000000000000f4  0000000000000000   A      0      0      8
[18] .init_array            INIT_ARRAY         0000000000600790   00000790
     0000000000000008  0000000000000000   WA     0      0      8
[19] .fini_array            FINI_ARRAY         0000000000600798   00000798
     0000000000000008  0000000000000000   WA     0      0      8
[20] .jcr                   PROGBITS          00000000006007a0   000007a0
     0000000000000008  0000000000000000   WA     0      0      8
[21] .dynamic               DYNAMIC           00000000006007a8   000007a8
     00000000000001d0  0000000000000010   WA     6      0      8
[22] .got                   PROGBITS          0000000000600978   00000978
     0000000000000008  0000000000000000   WA     0      0      8
[23] .got.plt               PROGBITS          0000000000600980   00000980
     0000000000000030  0000000000000008   WA     0      0      8
[24] .data                  PROGBITS          00000000006009c0   000009c0
     0000000000000181  0000000000000000   WA     0      0     64
[25] .bss                   NOBITS            0000000000600b41   00000b41
     0000000000000007  0000000000000000   WA     0      0      1
[26] .comment                PROGBITS          0000000000000000   00000b41
     0000000000000039  0000000000000001   MS     0      0      1
[27] .shstrtab              STRTAB            0000000000000000   00000b7a
     0000000000000108  0000000000000000   A      0      0      1
[28] .symtab                SYMTAB            0000000000000000   00000c88
     0000000000000660  0000000000000018   29     46     8
[29] .strtab                STRTAB            0000000000000000   000012e8
     000000000000024b  0000000000000000   0      0      1
Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
L (link order), O (extra OS processing required), G (group), T (TLS),
C (compressed), x (unknown), o (OS specific), E (exclude),
l (large), p (processor specific)

```

Figura 12. Secciones de la cabecera del archivo *analizar*.

Ahora con el comando `readelf` y con la bandera para mostrar el volcado en hexadecimal mostraremos las secciones `.data`, `.text` y `.rodata` como se muestran en las figuras 13, 14 y 15.

```
[user@parrot]--[~/Documents/Vulnerabilidades/exa]
$readelf --hex-dump=24 analizar

Hex dump of section '.data':
0x006009c0 00000000 00000000 00000000 00000000 .....
0x006009d0 00000000 00000000 00000000 00000000 .....
0x006009e0 00000000 00000000 00000000 00000000 .....
0x006009f0 00000000 00000000 00000000 00000000 .....
0x00600a00 38373635 342d3332 3130392d 38373635 87654-32109-8765
0x00600a10 342d3332 3144524f 2d575353 41500000 4-321DRO-WSSAP..
0x00600a20 5348454c 4c6f7720 77617320 68657265 SHELLow was here
0x00600a30 203a5000 00000000 00000000 00000000 .....
0x00600a40 ba7c35ee cfdadc9 7424f45e 31c9b13a .]5.....t$.^1..:
0x00600a50 83eefc31 560f0356 73d71bfe 7de002fe ...1V..Vs...}...
0x00600a60 4765c95e 4353c265 1bfc667 fdc5034c Ge.^CS.e...g...L
0x00600a70 035312d8 5d314ff1 3bdfbea9 b41a9ff9 .S..]10;.....
0x00600a80 f82b1a4a d73c21fa 780b1828 340eda26 .+.J.<!.x... (4..&
0x00600a90 5d458fe8 aca52010 8614ffaa 5c7bbf82 ]E....\....{..
0x00600aa0 514a0717 9d387d5f 620196a9 d401904b QJ...8} _b.....K
0x00600ab0 eb80ea74 f38aa445 3a0af5a3 3c37fa86 ...t...E:...<7..
0x00600ac0 49358218 b7457b8b c1b7349d e489d995 I5...E{...4....
0x00600ad0 c729a914 fc20365b 4a349f81 bf7e2e9d .).... 6[J4...~..
0x00600ae0 b5635c69 c8431454 2a7ba5a6 df2ded2b .c\i.C.T*{....+.
0x00600af0 4beae17b 423b487e ec8a6b77 e8e62a0d K...{B;H-.kw...*.
0x00600b00 f900a8e7 9899b2bf a5a257d0 118a9b2f .....W.../...
0x00600b10 a1ccee04 e9fdd013 d8269892 d3912807 .....&....(..
0x00600b20 89221558 839588a9 1ea332e3 7a1d8bb7 ...X.....2.Z...
0x00600b30 55c362a6 862c065e 8b66b72e 10884245 U.b...^..f....BE
0x00600b40 00
```

Figura 13. Volcado en hexadecimal de `.data`.

```
[user@parrot]--[~/Documents/Vulnerabilidades/exa]
$readelf --hex-dump=13 analizar

Hex dump of section '.text':
0x00400410 31ed4989 d15e4889 e24883e4 f0505449 1.I..^H..H...PTI
0x00400420 c7c05006 400048c7 c1e00540 0048c7c7 ..P.@.H...@.H..
0x00400430 06054000 e8b7ffff fff4660f 1f440000 ..@.....f..D...
0x00400440 b84f0b60 0055482d 480b6000 4883f80e .O.`UH-H.`H...
0x00400450 4889e576 1bb80000 00004885 c074115d H..v.....H..t.]
0x00400460 bf480b60 00ffe066 0f1f8400 00000000 .H.....f.....
0x00400470 5dc36666 6666662e 0f1f8400 00000000 ].ffff.....f...
0x00400480 be480b60 00554881 ee480b60 0048c1fe .H.`UH..H.`H...
0x00400490 034889e5 4889f048 c1e83f48 01c648d1 .H..H..H..?H..H.
0x004004a0 fe7415b8 00000000 4885c074 0b5dbf48 .t.....H..t..]H
0x004004b0 0b6000ff e00f1f00 5dc3660f 1f440000 .`.....].f..D...
0x004004c0 803d7a06 20000075 11554889 e5e86eff .=z...u.UH...n.
0x004004d0 fff5dc6 05670620 0001f3c3 0f1f4000 ..].g. ....@.
0x004004e0 bfa00760 0048833f 007505eb 930f1f00 .....H.?..u....
0x004004f0 b8000000 004885c0 74f15548 89e5ffdc .....H..t.UH...
0x00400500 5de97aff ffff5548 89e54883 ec7048b8 ].z...UH..H..pH.
0x00400510 42616961 2c206261 488945d0 48b86961 Baia, baH.E.H.ia
0x00400520 202e2e2e 20734889 45d848b8 69207175 ...sH.E.H.i qu
0x00400530 65206861 488945e0 48b87320 6c6c6567 e haH.E.H.s lleg
0x00400540 61644889 45e848b8 6f206c65 6a6f7300 adH.E.H.o lejos.
0x00400550 488945f0 48b84974 27732074 696d4889 H.E.H.It's timH.
0x00400560 459048b8 6520746f 20637261 48894598 E.H.e to craH.E.
0x00400570 48b863b6 6d65204d 69734889 45a048b8 H.ckme MisH.E.H.
0x00400580 732f4d72 20526576 488945a8 48b86572 s/Mr RevH.E.H.er
0x00400590 73652045 6e674889 45b048b8 696e6e65 se EngH.E.H.inne
0x004005a0 72203b29 488945b8 c645c000 488d45d0 r ;)H.E..E..H.E.
0x004005b0 4889c7e8 28feffff 488d4590 4889c7e8 H...((..H.E.H...
0x004005c0 1cfeffff 48c745f8 400a6000 488b55f8 ....H.E.@.`H.U.
0x004005d0 b8000000 00ffd2c9 c30f1f80 00000000 .....
0x004005e0 41574189 ff415649 89f64155 4989d541 AWA..AVI..AUI..A
0x004005f0 544cd25 98012000 55488d2d 98012000 TL.%...UH...
0x00400600 534c29e5 31db48c1 fd034883 ec08e895 SL).1.H...H....
0x00400610 fdffff48 85ed741e 0f1f8400 00000000 ...H..t.....
0x00400620 4c89ea4c 89f64489 ff41ff14 dc4883c3 L..L..D..A...H..
0x00400630 014839eb 75ea4883 c4085b5d 415c415d .H9.u.H...[JAY]
0x00400640 415e415f c366662e 0f1f8400 00000000 A^A_.ff.....
0x00400650 f3c3
```

Figura 14. Volcado en hexadecimal de `.text`.

```
[user@parrot]~[~/Documents/Vulnerabilidades/eva]
$readelf --hex-dump=15 analizar
Hex dump of section '.rodata':
0x00400660 01000200 : VERNEED 00....
```

Figura 15. Volcado en hexadecimal de .rodata.

Observamos que en .data aparece el que parece ser un serial y en .text aparece una cadena de texto.

Con el comando **ldd**, nos muestra las bibliotecas que necesita el programa para su ejecución, mostrado en la figura 16.

```
[X]~[user@parrot]~[~/Documents/Vulnerabilidades/eva]
$ldd analizar
linux-vdso.so.1 (0x00007fffb8bd0000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f617a49f000)
/lib64/ld-linux-x86-64.so.2 (0x00007f617a695000)
```

Figura 16. Bibliotecas necesarias para la ejecución del programa.

Linux-vdso.so.1: No existe en el sistema de archivos, es una implementación que se usa para invocar llamadas al sistema de manera eficiente.

Libc.so.6: Contiene las bibliotecas estándar que utiliza casi cualquier programa en el sistema. Este paquete incluye las bibliotecas compartidas de la biblioteca estándar de C y de la biblioteca estándar de matemáticas, así como muchas otras bibliotecas.

/lib64/ld-linux-x86-64.so.2: Es llamada cuando se ejecuta el programa y su función es inicializar la carga de las bibliotecas dinámicas.

Procedemos a ejecutar el comando **nm** para listar símbolos (variables y funciones) del binario *analizar*, mostrado en la figura 17.

La salida nos indica que la letra **D** es para lo que está inicializado en la sección de datos, la **T** son funciones, lo que me llamó la atención es que *msg1* y *msg2* tienen una D lo que significa que están inicializados en la sección de datos y *shellcode* también está inicializado en la sección de datos, este nombre es un poco inusual, ya que por lo regular una *shellcode* son hexadecimales que se insertan en un programa para que realice algo dentro del programa como darnos una shell.

```

[user@parrot]-(~/Documents/Vulnerabilidades/exa)
$nm analizar
0000000000600b41 B _bss_start
0000000000600b41 b completed.666132-001-rip
00000000006009c0 D _data_start
00000000006009c0 W data_start
0000000000400440 t deregister_tm_clones
00000000004004c0 t do_global_ctors_aux
0000000000600798 t do_global_ctors_aux_fini_array_entry
00000000006009c8 D dso_handle
00000000006007a8 d _DYNAMIC
0000000000600b41 D _edata
0000000000600b48 B _end
0000000000400654 T _fini
00000000004004e0 t frame_dummy
0000000000600790 t frame_dummy_init_array_entry
0000000000400788 r _FRAME_END
0000000000600980 d GLOBAL_OFFSET_TABLE
00000000004003a8 T _init
0000000000600798 t _init_array_end
0000000000600790 t _init_array_start
0000000000400660 R _IO_stdin_used
0000000000400650 w ITM_deregisterTMCloneTable
00000000006007a0 w ITM_registerTMCloneTable
00000000006007a0 d _JCR_END
00000000006007a0 d _JCR_LIST
0000000000400650 w Jv_RegisterClasses
0000000000400650 T libc_csu_fini
00000000004005e0 T libc_csu_init
0000000000400506 U libc_start_main@@GLIBC_2.2.5
0000000000400506 T main
0000000000600a20 d msg1
0000000000600a00 d msg2
0000000000400480 U puts@@GLIBC_2.2.5
0000000000600a40 t register_tm_clones
0000000000400410 T _start
0000000000600b48 D _TMC_END

```

Figura 17. Listar símbolos del binario.

Ahora con el comando **objdump** y dos banderas vamos a desensamblar el archivo *analizar*, en formato Intel y guardaremos la salida en un archivo llamado *dis_analizar* como se muestra en la figura 18.

```

[user@parrot]-(~/Documents/Vulnerabilidades/exa)
$objdump -M intel -d analizar > dis_analizar

```

Figura 18. Objdump a analizar.

Al analizar el archivo *dis_analizar* observamos en el main la creación del prólogo como se muestra en la figura 19, después observamos las funciones que llama dentro del main en la figura 20.

Al analizar las funciones que se llaman dentro del main observamos que se llama dos veces a la función *puts* y una vez a la función *rdx*, esta última función no tiene mucho sentido, lo que la hace una función anormal dentro del programa.

```

0000000000400506 <main>:
400506: 55 41 50 97 23 17 12 00  push    rbp
400507: 48 89 e5                mov     rbp, rsp
40050a: 48 83 ec 70             sub     rsp, 0x70
40050e: 48 b8 42 61 69 61 2c 00  movabs  rax, 0x6162202c61696142
400515: 20 62 61                mov     QWORD PTR [rbp-0x30], rax
400518: 48 89 45 d0             movabs  rax, 0x73202e2e2e206169
40051c: 48 b8 69 61 20 2e 2e     mov     QWORD PTR [rbp-0x28], rax
400523: 2e 20 73                movabs  rax, 0x6168206575712069
400526: 48 89 45 d8             mov     QWORD PTR [rbp-0x20], rax
40052a: 48 b8 69 20 71 75 65     movabs  rax, 0x646167656c6c2073
400531: 20 68 61                mov     QWORD PTR [rbp-0x10], rax
400534: 48 89 45 e0             mov     QWORD PTR [rbp-0x8], rax
400538: 48 b8 73 20 6c 6c 65     movabs  rax, 0x646167656c6c2073

```

Figura 19. Creación del prólogo en el main.

```

4005a4: 48 89 45 b8      mov     QWORD PTR [rbp-0x48],rax
4005a8: 8b 45 c0 00      mov     BYTE PTR [rbp-0x40],0x0
4005ac: 48 8d 45 d0      lea     rax,[rbp-0x30]
4005b0: 48 89 c7         mov     rdi,rax
4005b3: e8 28 fe ff ff  call    4003e0 <puts@plt>
4005b8: 48 8d 45 90      lea     rax,[rbp-0x70]
4005bc: 48 89 c7         mov     rdi,rax
4005bf: e8 1c fe ff ff  call    4003e0 <puts@plt>
4005c4: 48 c7 45 f8 40 0a 60 mov     QWORD PTR [rbp-0x8],0x600a40
4005cb: 00
4005cc: 48 8b 55 f8      mov     rdx,QWORD PTR [rbp-0x8]
4005d0: b8 00 00 00 00  mov     eax,0x0
4005d5: c7 ff d2        call    rdx
4005d7: c9              leave
4005d8: c3              ret
4005d9: 0f 1f 80 00 00 00 00 nop     DWORD PTR [rax+0x0]

```

Figura 20. Llamadas a funciones dentro del main.

Una vez realizado todo lo anterior se procede a realizar el análisis dinámico del binario.

Análisis dinámico

Como primer paso se procede a ejecutar el programa para observar su comportamiento, para esto lo ejecutamos como se muestra en la figura 21 y observamos una salida de texto, lo que me llamó la atención es que en la última parte hay un mensaje que dice *ingeniería inversa*.

```

[user@parrot]~/Documents/Vulnerabilidades/exa
$ ./analizar
Baia, baia ... si que has llegado lejos
It's time to crackme Miss/Mr Reverse Enginner ;)

```

Figura 21. Ejecución del programa *analizar*.

Se procede a utilizar **gdb** para ver el comportamiento del programa como se muestra en la figura 22.

```

[root@parrot]~/home/user/Documents/Vulnerabilidades/exa
#gdb -q analizar

```

Figura 22. Ejecución de gdb.

Ahora ponemos que nos muestra las instrucciones en formato Intel con el comando que se muestra en la figura 23.

```

(gdb) set disassembly-flavor intel

```

Figura 23. Instrucciones en formato Intel.

Colocamos un break point en el main mostrado en la figura 24, posteriormente con los comandos mostrados en las figuras 25 y 26, podremos ver cómo se va ejecutando y como van cambiando los registros lo que nos permitirá colocar break points para ir analizando el comportamiento del programa mostrado en la figura 27.

```

(gdb) b main
Breakpoint 1 at 0x40050a

```

Figura 24. Break point en el main.

`(gdb) layout asm`

Figura 25. Layout asm.

`(gdb) layout regs`

Figura 26. Layout regs.

Register group: general									
rax	0x400506	4195590	rbx	0x0	0	rcx	0x7ffff7f95718	140737353701144	
rdx	0x7fffffe0f8	140737488347384	rsi	0x7fffffe0e8	140737488347368	r10	0x1	1	
rbp	0x7fffffe000	0x7fffffe000	rsp	0x7fffffe000	0x7fffffe000	r8	0x7ffff7f96d80	140737353706880	
r9	0x7ffff7f96d80	140737353706880	r10	0xffffffffffff324	-3292	r11	0x7ffff7dfdfb0	140737352032176	
r12	0x400410	4195344	r13	0x7fffffe0e0	140737488347360	r14	0x0	0	
r15	0x0	0	rip	0x40050a	0x40050a <main+4>	eflags	0x246	[PF ZF IF]	
cs	0x33	51	ss	0x2b	43	ds	0x0	0	
es	0x0	0	fs	0x0	0	gs	0x0	0	

0x400506 <main>	push	rbp
0x400507 <main+1>	mov	rbp, rsp
0x40050a <main+4>	sub	rsp, 0x70
0x40050e <main+8>	movabs	rax, 0x6162202c61696142
0x400510 <main+10>	mov	QWORD PTR [rbp-0x30], rax
0x40051c <main+22>	movabs	rax, 0x73202e2e2e206169
0x400520 <main+28>	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+58>	mov	QWORD PTR [rbp-0x18], rax

Figura 27. Visualización de los registros e instrucciones.

Bajamos en el main y buscamos cuando llama a la función `rdx` mostrado en la figura 28.

```

0x4005bc <main+182>    mov     rdi, rax
0x4005bf <main+185>    call   0x4003e0 <puts@plt>
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
0x4005e2 <_libc_csu_init+2> mov     r15d, edi

```

Figura 28. Llamada a la función `rdx`.

Colocamos un break point en `main+207` que es donde se hace la llamada a la función `rdx` como se muestra en la figura 29.

```

(gdb) b *main+207
Breakpoint 2 at 0x4005d5

```

Figura 29. Break point en `main+207`.

Una vez colocado con el comando `c`, continuara la ejecución del programa hasta donde se llama a la función `rdx` y con el comando `si` entramos a la función como se muestra en la figura 30.

```

> 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

```

Figura 30. Instrucciones dentro de la función `rdx`.

Al analizar todas las instrucciones que se realizan dentro de esta función observamos que hay varias comparaciones y saltos a instrucciones como se muestra en la figura 31.

```

0x600a73 <shellcode+51> fcomp DWORD PTR [rbp+0x31]
0x600a76 <shellcode+54> rex.WRXB icebp
0x600a78 <shellcode+56> cmp     ebx,edi
0x600a7a <shellcode+58> mov     esi,0x9f1ab4a9
0x600a7f <shellcode+63> stc
0x600a80 <shellcode+64> clc
0x600a81 <shellcode+65> sub     ebx,DWORD PTR [rdx]
0x600a83 <shellcode+67> rex.WX xlat BYTE PTR ds:[rbx]
0x600a85 <shellcode+69> cmp     al,0x21
0x600a87 <shellcode+71> cli
0x600a88 <shellcode+72> js      0x600a95 <shellcode+85>

```

Figura 31. Comparaciones y salto dentro de la función rdx.

Continuamos con la ejecución del programa con el comando **si** y observamos que se entra en un ciclo con **rcx** toma el valor de 58 y va disminuyendo hasta llegar a 0, mostrado en la figura 32 al salir del ciclo continua con las siguientes instrucciones, el ciclo corresponde a shellcode+16 hasta shellcode+25, al observar las siguientes instrucciones se encontró con varias llamadas al sistema como se muestra en la figura 33.

```

> 0x600a50 <shellcode+16> sub     esi,0xffffffff
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     DWORD PTR [rbp-0x20],edi

```

Figura 32. Ciclo donde disminuye rcx de 58 a 0.

```

0x600a74 <shellcode+52> pop     rsi
0x600a75 <shellcode+53> push    rdx
0x600a76 <shellcode+54> push    0x10
0x600a78 <shellcode+56> pop     rdx
0x600a79 <shellcode+57> push    0x31
0x600a7b <shellcode+59> pop     rax
0x600a7c <shellcode+60> syscall
0x600a7e <shellcode+62> pop     rsi
0x600a7f <shellcode+63> mov     al,0x32
0x600a81 <shellcode+65> syscall
0x600a83 <shellcode+67> mov     al,0x2b

```

Figura 33. Llamadas al sistema.

Al llegar a la instrucción shellcode+69 observamos que es una llamada al sistema y al darle enter no nos muestra las siguientes instrucciones, como se muestra en la figura 34, como si esperara una entrada el programa.

```

> 0x600a85 <shellcode+69> syscall
0x600a87 <shellcode+71> push    rax
0x600a88 <shellcode+72> pop     rdi
0x600a89 <shellcode+73> xor     rdx,rdx

```

```

native process 16526 In: shellcode
0x000000000000600a6c in shellcode ()
0x000000000000600a73 in shellcode ()
0x000000000000600a74 in shellcode ()
0x000000000000600a75 in shellcode ()
0x000000000000600a76 in shellcode ()
0x000000000000600a78 in shellcode ()
0x000000000000600a79 in shellcode ()
0x000000000000600a7b in shellcode ()
0x000000000000600a7c in shellcode ()
0x000000000000600a7e in shellcode ()
0x000000000000600a7f in shellcode ()
0x000000000000600a81 in shellcode ()
0x000000000000600a83 in shellcode ()
0x000000000000600a85 in shellcode ()

```

Figura 34. Llamada al sistema en shellcode+69.

Detenemos la ejecución del programa y nos salimos de **gdb**, ahora con el comando **strace** y como entrada a este comando ejecutamos el programa analizar, para observar las llamadas al sistema que realiza el programa analizar, el comando y la salida del comando se muestra en la figura 35.

```
[root@parrot:~]# user/user/Documents/Vulnerabilidades/eva/
#strace ./analizar
execve("./analizar", ["/home/user/Documents/Vulnerabilidades/eva/"], 0x7ffdfbc29cbf0 /* 44 vars */) = 0
brk(NULL) = 0x1c000
access("/etc/ld.so.preload", R_OK) = 0
openat(AT_FDCWD, "/etc/ld.so.preload", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=0, ...}) = 0
close(3) = 0
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=205736, ...}) = 0
mmap(NULL, 205736, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fb9ffa2a000
close(3) = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0\0\0\1\0\0\0\260A\2\0\0\0\0"... , 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1824496, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fb9ffa28000
mmap(NULL, 1837056, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7fb9ff867000
mprotect(0x7fb9ff889000, 1658880, PROT_NONE) = 0
mmap(0x7fb9ff889000, 1343488, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x22000) = 0x7fb9ff889000
mmap(0x7fb9ff9d1000, 311296, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x16a000) = 0x7fb9ff9d1000
mmap(0x7fb9ff9fa000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1b6000) = 0x7fb9ff9fa000
mmap(0x7fb9ffa24000, 14336, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7fb9ffa24000
close(3) = 0
arch_prctl(ARCH_SET_FS, 0x7fb9ffa29500) = 0
mprotect(0x7fb9ffa0000, 16384, PROT_READ) = 0
mprotect(0x7fb9ffa84000, 4096, PROT_READ) = 0
munmap(0x7fb9ffa2a000, 205736) = 0
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}) = 0
brk(NULL) = 0x1c000
brk(0xe3d000) = 0xe3d000
write(1, "Baia, baia ... si que has llegad"... , 40Baia, baia ... si que has llegado lejos
) = 40
write(1, "It's time to crackme Miss/Mr Rev"... , 49It's time to crackme Miss/Mr Reverse Engineer ;)
) = 49
socket(AF_INET, SOCK_STREAM, IPPROTO_IP) = 3
bind(3, {sa_family=AF_INET, sin_port=htons(39321), sin_addr=inet_addr("0.0.0.0")}, 16) = 0
listen(3, 0) = 0
accept(3, NULL, 0x10) = ? ERESTARTSYS (To be restarted if SA_RESTART is set)
--- SIGWINCH {si signo=SIGWINCH, si_code=SI_KERNEL} ---
accept(3, NULL, 0x10) = ? ERESTARTSYS (To be restarted if SA_RESTART is set)
```

Figura 35. Ejecución del comando strace.

Al analizar la salida observamos que se crea un socket en el puerto 39321 y la dirección IP 0.0.0.0 y esta a la espera de una conexión, con lo que se procede desde otra terminal a crear la conexión a ese puerto y dirección IP con el comando **netcat**, como se muestra en la figura 36.

```
[user@parrot]-[~/Documents/Vulnerabilidades/extra]
$nc 0.0.0.0 39321
```

Figura 36. Conexión con netcat.

Volvemos a regresar a la terminal donde ejecutamos **strace** y observamos que se aceptó una conexión y esta a la espera de recibir algo, como se muestra en la figura 37.

```
accept(3, NULL, 0x10) = 7 ERESTARTSYS (To be restarted if SA_RESTART is set)
--- SIGWINCH {si_signo=SIGWINCH, si_code=SI_KERNEL} ---
accept(3, NULL, 0x10) = 4
read(4,
```

Figura 37. Conexión establecida con un cliente.

Ahora con la conexión establecida se procede a mandar una cadena de texto desde donde creamos la conexión con **netcat** y escribimos Ignacio y lo mandamos, como se muestra en la figura 38.

```
[user@parrot]-[~/Documents/Vulnerabilidades/extra]
$nc 0.0.0.0 39321
ignacio
```

Figura 38. Envío de una cadena.

Al regresar a la terminal donde se ejecutó **strace** observamos que hubo un segmentation fault al recibir la cadena ignacio, como se muestra en la figura 39.

```
read(4, "ignacio\n", 32) = 8
--- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_MAPERR, si_addr=0xa} ---
+++ killed by SIGSEGV +++
Segmentation fault
```

Figura 39. Segmentation fault al recibir la cadena ignacio.

Al recordar el análisis estático se procede a introducir la cadena que parece un serial mostrado en la figura 11, se vuelve a ejecutar el comando **strace** y se abre de nuevo una conexión con **netcat** y se envía ahora el serial, pero se observa que hay un segmentation fault, se procede a ver el comportamiento del programa ahora con **gdb**, volvemos a abrir **gdb** con los pasos anteriores y se entra a la función **rdx** y continuamos con la ejecución hasta la llamada de sistema que está en el shellcode+69, la cual abre la conexión para conectarnos por medio de **netcat** y mandamos el serial, observamos que al mandar el serial en **gdb** podemos seguir ejecutando las siguientes instrucciones, observamos que se crea otro ciclo desde la instrucción shellcode+98 hasta shellcode+106, mostrado en la figura 40, donde **rcx** comienza desde 0 hasta 30 cuando se termina el ciclo.

```
> 0x600aa2 <shellcode+98> cmp     BYTE PTR [rsp+rcx*1],a1
0x600aa5 <shellcode+101> je      0x600aac <shellcode+108>
0x600aa7 <shellcode+103> inc     rcx
0x600aaa <shellcode+106> jmp     0x600aa2 <shellcode+98>
```

Figura 40. Ciclo donde **rcx** empieza en 0 hasta llegar a 30.

Cuando termina el ciclo y se hace una comparación con **rcx** y **0x1d**, mostrado en la figura 41 observamos que **0x1d** vale 29 en decimal y **rcx** vale 30 por lo que

nos son iguales y salta hasta la instrucción shellcode+255 que es cuando termina el programa y nos muestra un segmentation fault..

```
0x600aac <shellcode+108>    cmp    rcx,0x1d
0x600ab0 <shellcode+112>    jne    0x600b3f <shellcode+255>
```

Figura 41. Comparación entre rcx y 0xd y salto cuando no son iguales.

Se procede ahora a mandar 29 0's y observamos que sale del ciclo pero en la comparación de shellcode+124 y salta nuevamente a shellcode+255 de la instrucción shell-code+128 ya que no son iguales mostrado en la figura 42.

```
0x600abc <shellcode+124>    cmp    BYTE PTR [rsp+rcx*1],0x2d
0x600ac0 <shellcode+128>    jne    0x600b3f <shellcode+255>
```

Figura 42. Comparación.

Se procede a ver el formato del serial que viene al ejecutar el comando strings y se observa que tiene 4 guiones, por lo que ahora se pondrán 5 0's y seguido estará un guion hasta formar una cadena de 29 caracteres, volvemos a ejecutar los comando y se le manda esa cadena de 0's y guiones, por lo que en el gdb ya no salta a shellcode+255 de la instrucción shellcode+128 y continua con la ejecución del programa, hasta llegar a la instrucción shellcode+146 donde inicia un ciclo hasta la instrucción shellcode+155, donde rcx disminuye su valor hasta 0, mostrado en la figura 43.

```
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>
```

Figura 43. Ciclo.

Siguiendo con la ejecución en la instrucción shellcode+166 se hace una comparación de rax con 0x8e0, mostrado en la figura 44. El valor de rax en este caso vale 1380 mostrado en la figura 45 y en decimal equivale a 2272 por lo que no valen igual y saltan a la instrucción shellcode+255 de la instrucción shellcode+166.

```
0x600ae6 <shellcode+166>    cmp    rax,0x8e0
0x600aec <shellcode+172>    jne    0x600b3f <shellcode+255>
```

Figura 44. Comparación.

```
(gdb) i r rax
rax      0x564      1380
```

Figura 45. Valor de rax.

Por lo que al seguir ejecutando mostrará el mensaje de segmetantion fault y terminará el programa por lo que se decide a ingresar una cadena de 29 caracteres que contenga 4 guiones y sumen en decimal 2272, se encontró la siguiente cadena ""AAAAA-ZZZZZ-aaaaa-bbbbb-DDDDF", se vuelve a ejecutar el programa hasta la instrucción shellcode+253 mostrada en la figura 46, hace

una llamada al sistema y nos devuelve una shell en la terminal donde utilizamos el comando netcat mostrada en la figura 47.

```
0x600b3d <shellcode+253> syscall
```

Figura 46. Llamada al sistema.

```
[user@parrot]~/Documents/Vulnerabilidades/exa
$nc 0.0.0.0 39321
AAAAA-ZZZZZ-aaaaa-bbbbb-DDDDF
id
uid=0(root) gid=0(root) groups=0(root)
whoami
root
uname -a
Linux parrot 4.19.0-parrot1-13t-amd64 #1 SMP Debian 4.19.13-1parrot1.13t (2019-01-09) x86_64 GNU/Linux
```

Figura 47. Obtención de una shell.

Conociendo todo esto ahora procedemos a crear una cadena valida con nuestro nombre, que cumpla con los requisitos previos, la cadena que cumple con esos requisitos es “ignac-iolea-IMCCC-CCCCC-CCCCC”, al seguir con las instrucciones se verifico el valor de rax en la comparación el cual es de 2272 cumpliéndose la igualdad se muestra el valor de rax en la figura 48.

```
(gdb) i r rax
rax                0x8e0        2272
```

Figura 48. Valor de rax.

Al seguir con la ejecución del programa se obtiene una shell en el lado donde se utilizo el comando netcat, mostrado en la figura 49, donde se muestra que se ingreso una cadena valida con el nombre y apellido para así obtener una shell.

```
[user@parrot]~
$nc 0.0.0.0 39321
ignac-iolea-IMCCC-CCCCC-CCCCC
ls
SHELLow
analizar
analizar_stripped
dis_analizar
ejecutable
shell_mod2
id
uid=0(root) gid=0(root) groups=0(root)
uname
Linux
```

Figura 49. Ingreso de una cadena valida para obtener una shell.