Exámen

Análisis estático

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

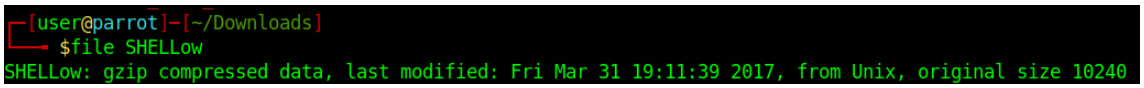

Figura 1. Tipo de archivo *SHELLow.*

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.

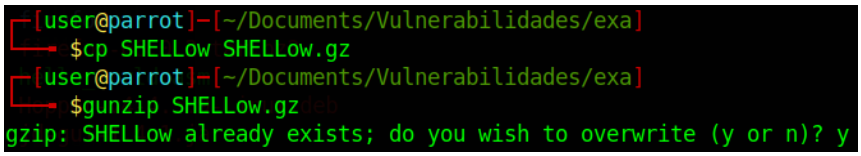

Figura 2. Copiar y descomprimir el archivo *SHELLow.*

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”.

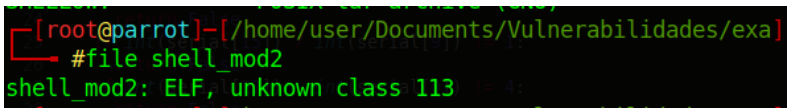

Figura 3. Tipo de archivo *shell_mod2*.

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.

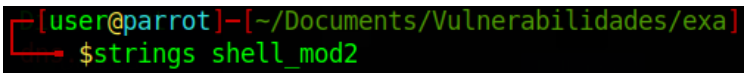

Figura 4. Comando strings.

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.

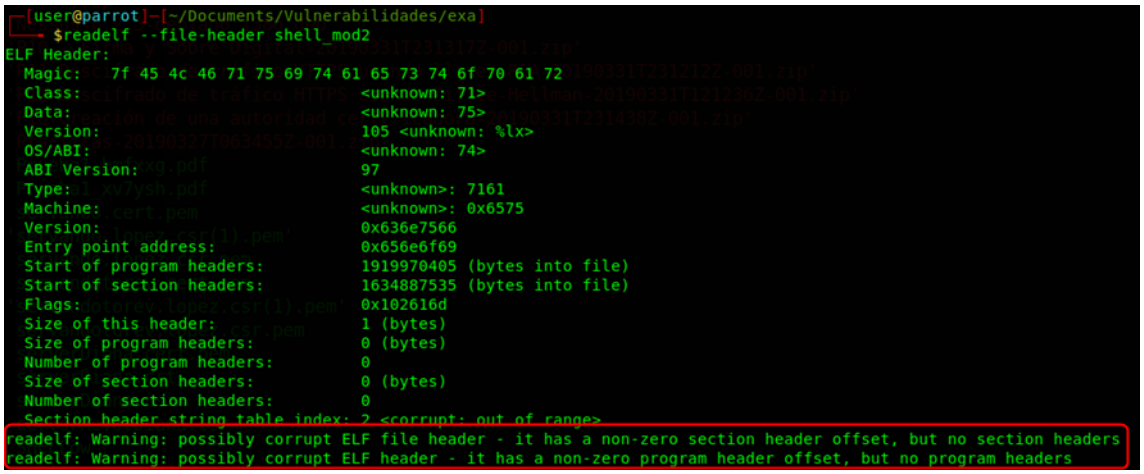

Figura 6. Cabecera del archivo *shell_mod2*.

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*.

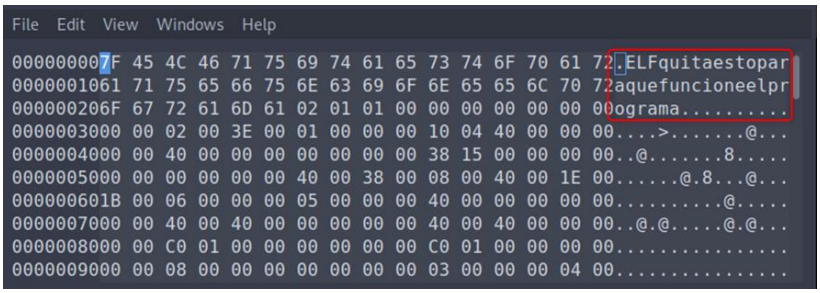

Figura 7. Hexadecimal con la cadena.

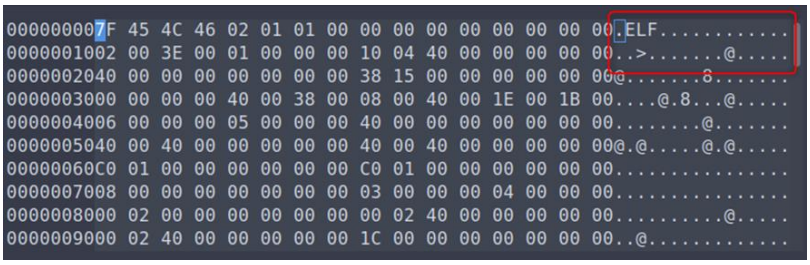

Figura 8. Hexadecimal sin la cadena.

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.

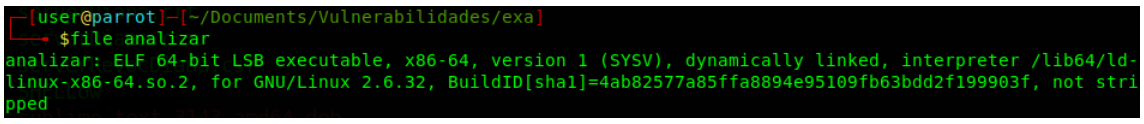

Figura 9. File a *analizar*.

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.

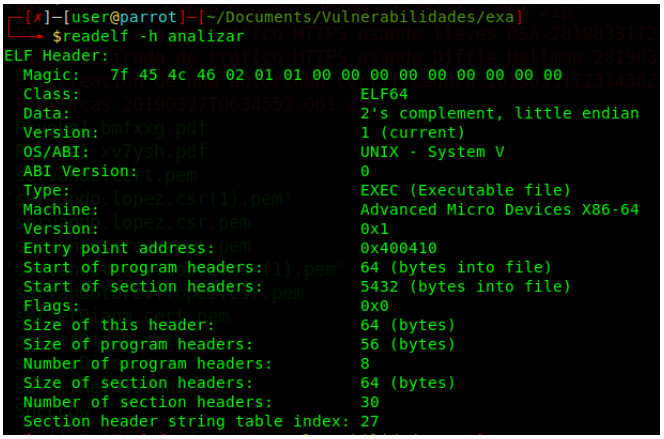

Figura 10. Readelf al archivo *analizar*.

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.

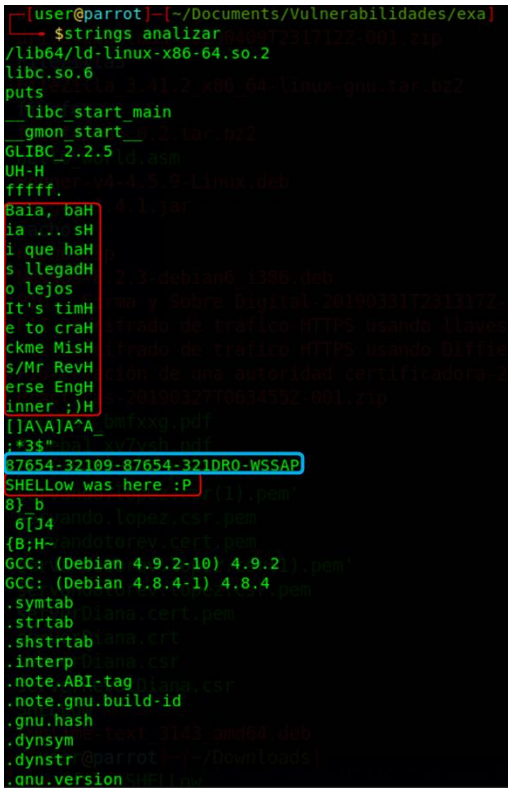

Figura 11. Análisis de salida del comando strings a *analizar*.

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.

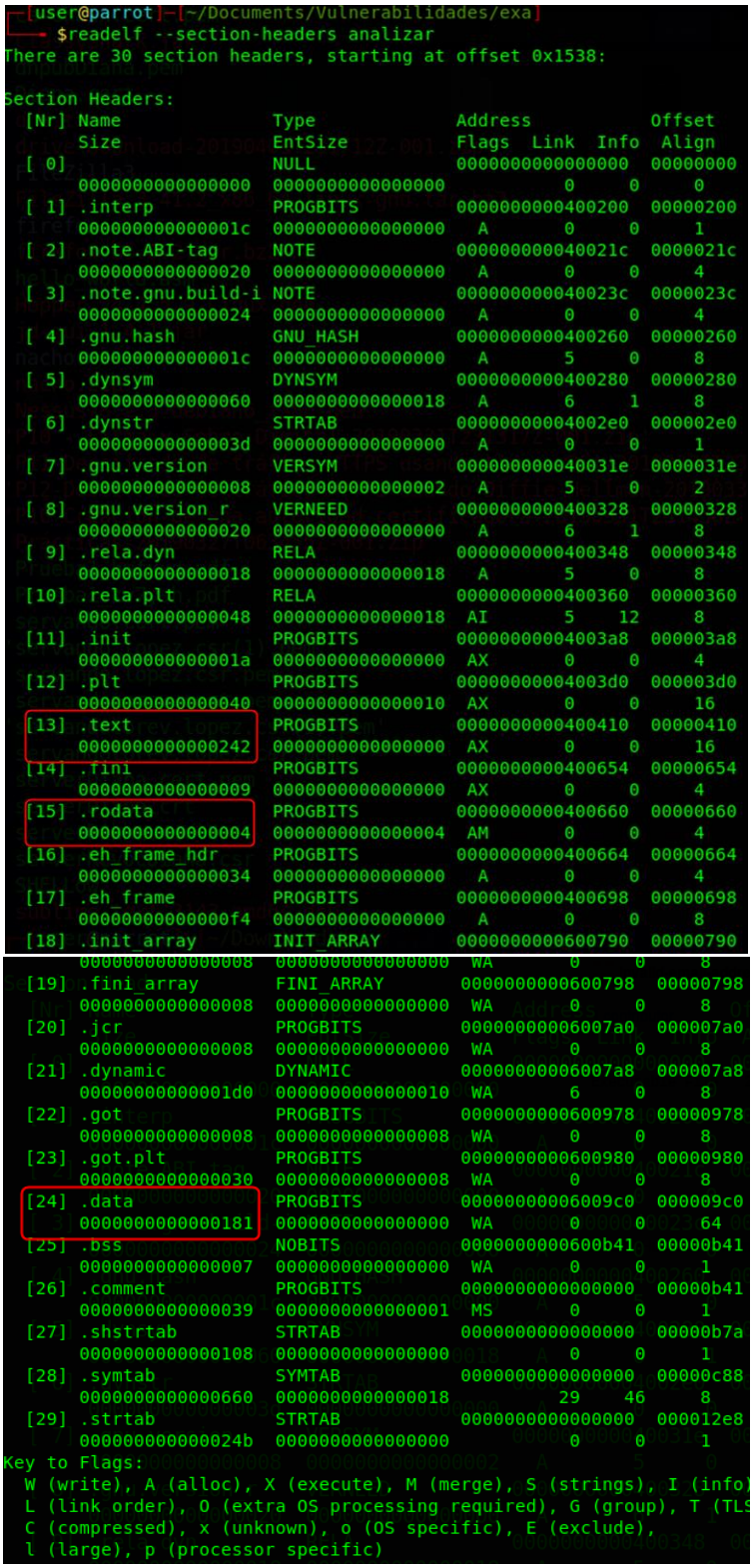

Figura 12. Secciones de la cabecera del archivo *analizar*.

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

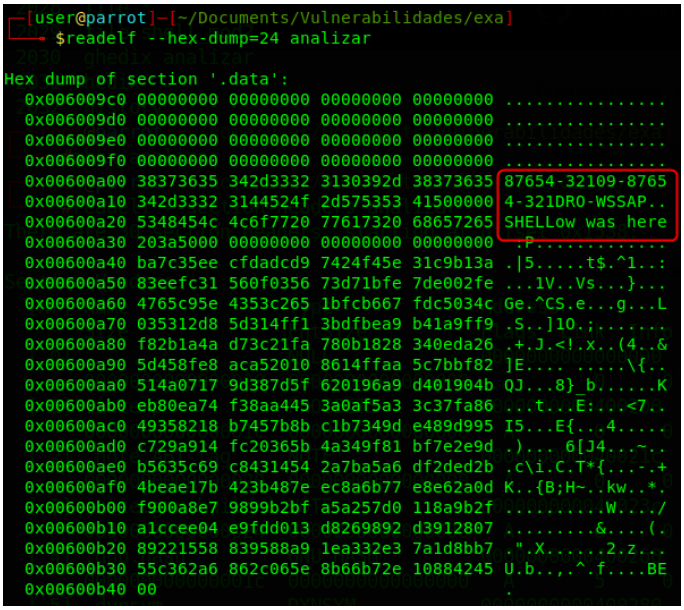

Figura 13. Volcado en hexadecimal de .data.

Figura 14. Volcado en hexadecimal de .text.

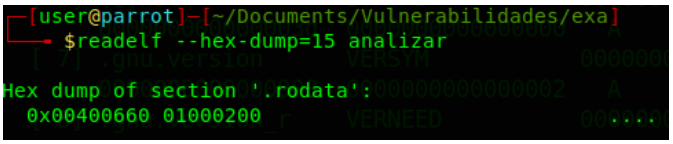

Figura 15. Volcado en hexadecimal de .rodata.

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.

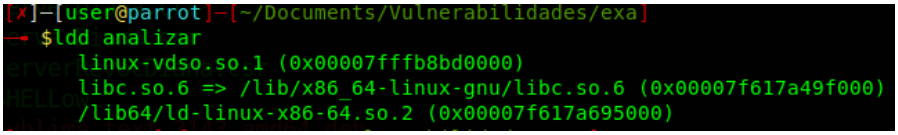

Figura 16. Bibliotecas necesarias para la ejecución del programa.

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 esta inicializado en la sección de datos, la **T** son funciones, lo que me llamo la atención es que *msg1* y *msg2* tiene una D lo que significa que esta inicializado en la sección de datos y *shellcode* también esta 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 .

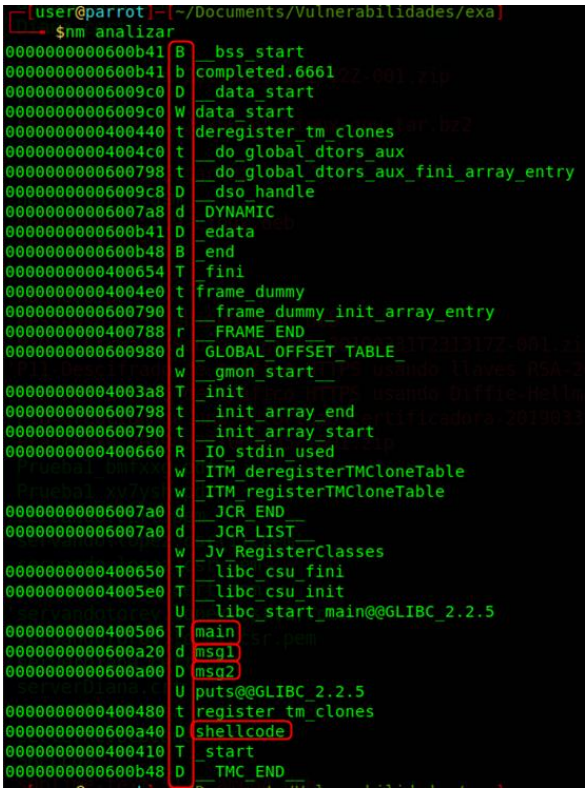

Figura 17. Listar símbolos del binario.

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.

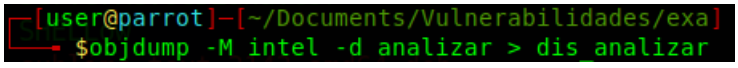

Figura 18. Objdump a analizar.

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.

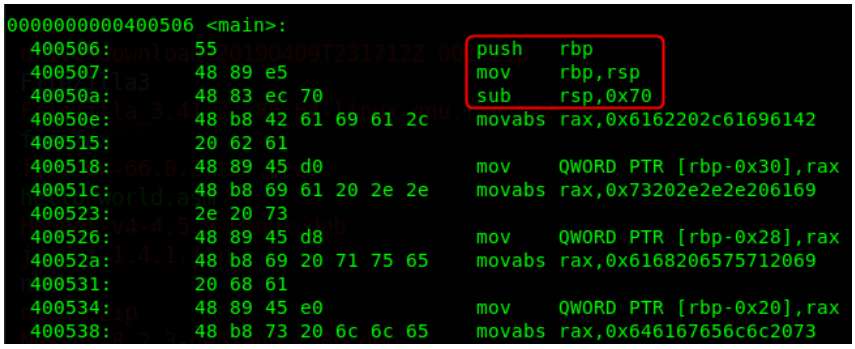

Figura 19. Creación del prólogo en el main.

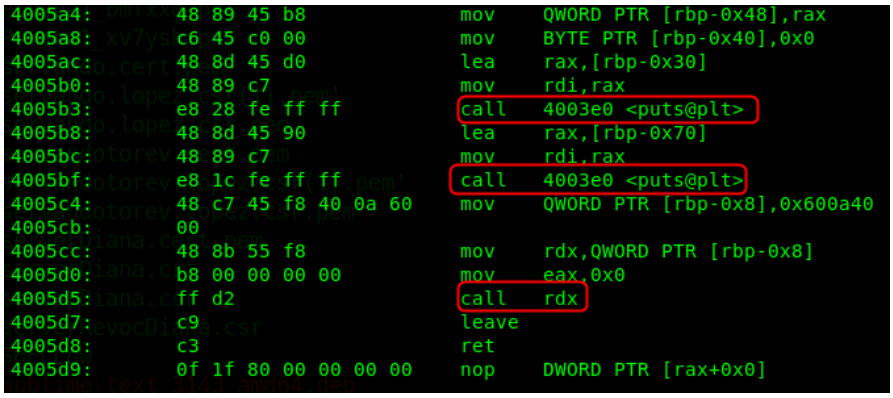

Figura 20. Llamadas a funciones dentro del main.

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 llamo la atención es que en la última parte hay un mensaje que dice *ingeniería inversa*.

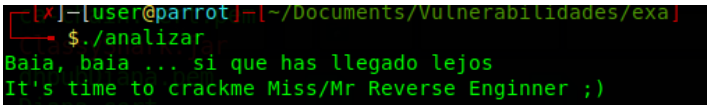

Figura 21. Ejecución del programa *analizar*.

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

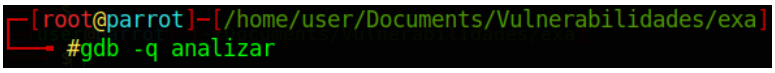

Figura 22. Ejecución de gdb.

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

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.

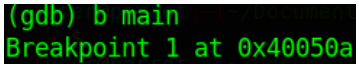

Figura 24. Break point en el main.

```
(gdb) layout asm
```

Figura 25. Layout asm.

```
(gdb) layout regs
```

Figura 26. Layout regs.

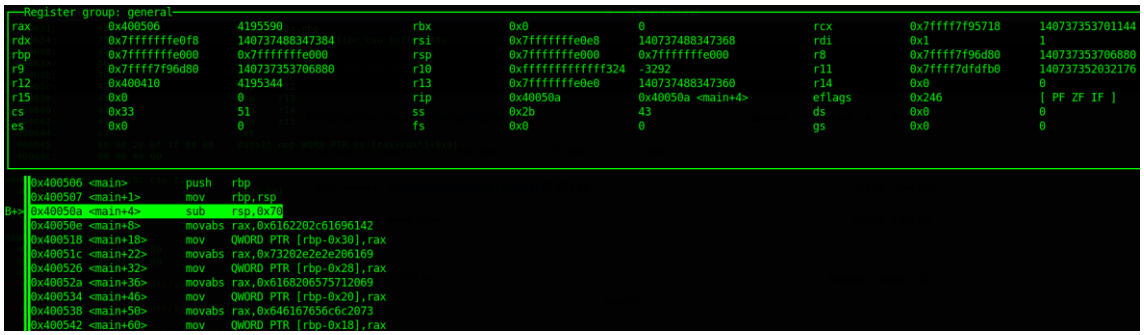

Figura 27. Visualización de los registros e instrucciones.

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>   fcmovu 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,0xfffffffc
  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.

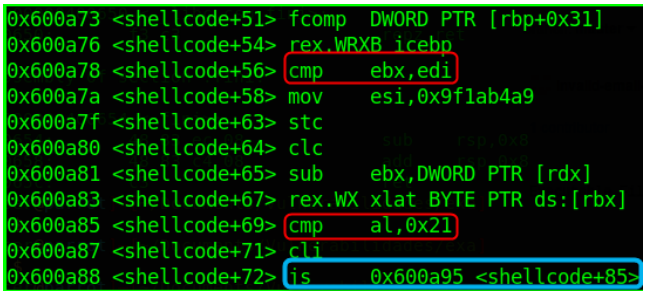

Figura 31. Comparaciones y salto dentro de la función rdx.

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.

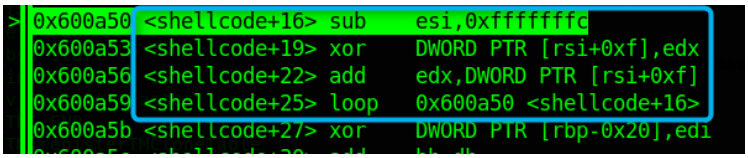

Figura 32. Ciclo donde disminuye rcx de 58 a 0.

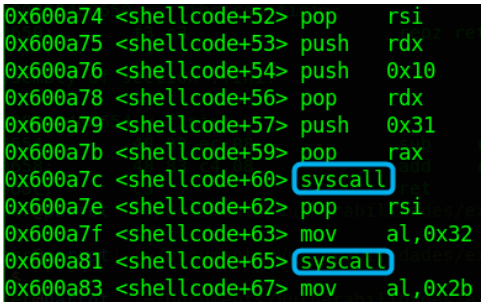

Figura 33. Llamadas al sistema.

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.

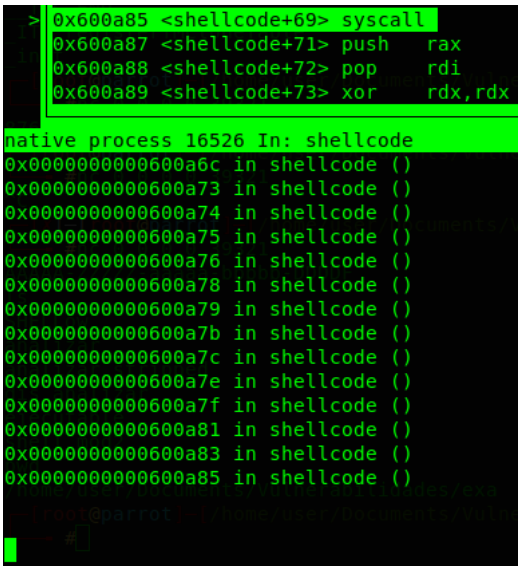

Figura 34. Llamada al sistema en shellcode+69.

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.

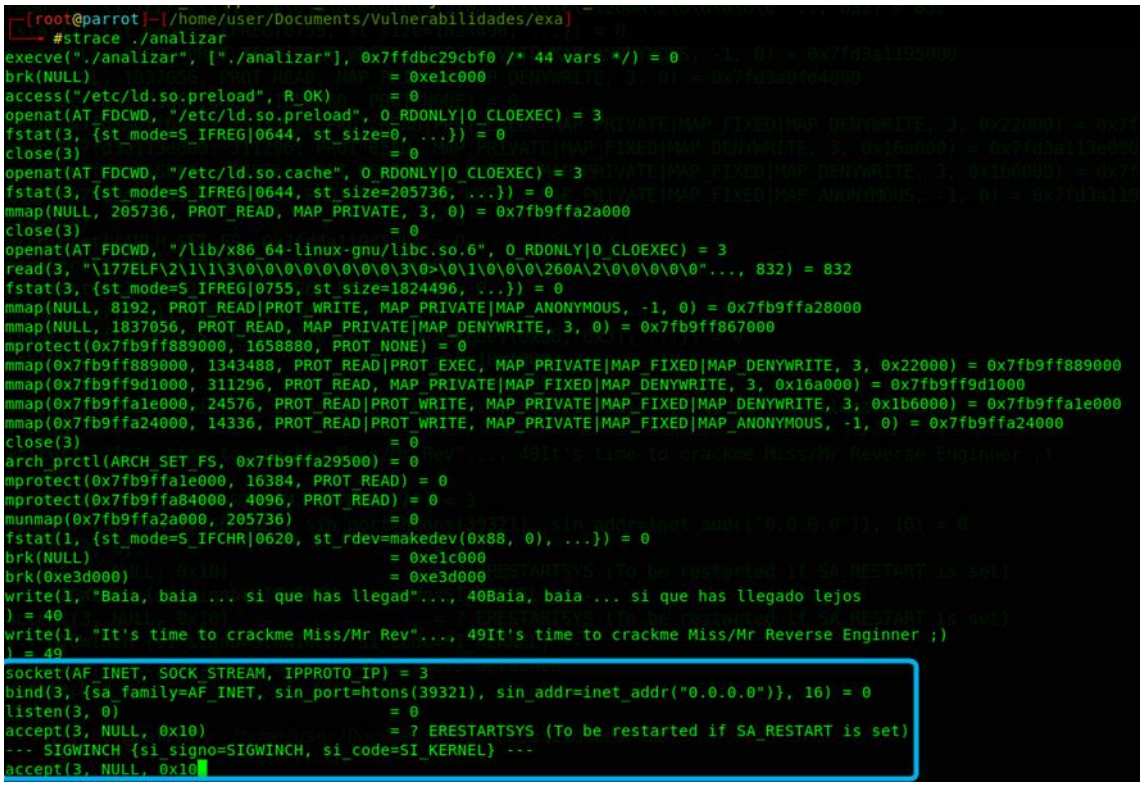

Figura 35. Ejecución del comando strace.

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.

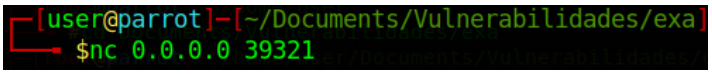

Figura 36. Conexión con netcat.

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.

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.

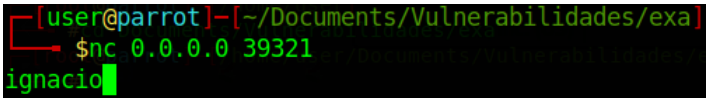

Figura 38. Envió de una cadena.

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.

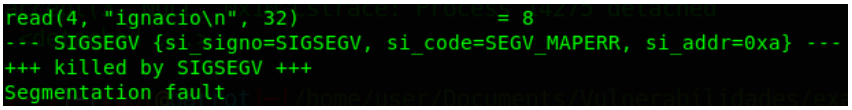

Figura 39. Segmentation fault al recibir la cadena ignacio.

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.

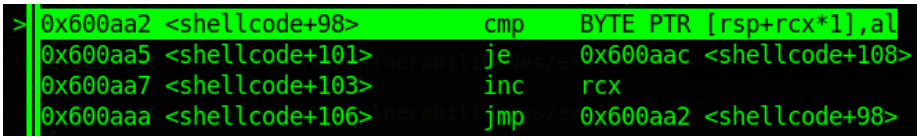

Figura 40. Ciclo donde rcx empieza en 0 hasta llegar a 30.

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..

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.

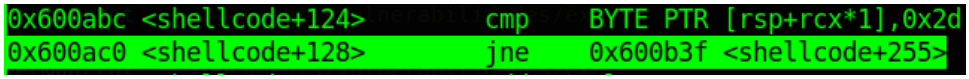

Figura 42. Comparación.

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.

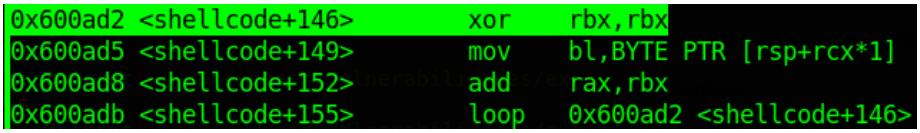

Figura 43. Ciclo.

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.

Figura 44. Comparación.

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.

Figura 46. Llamada al sistema.

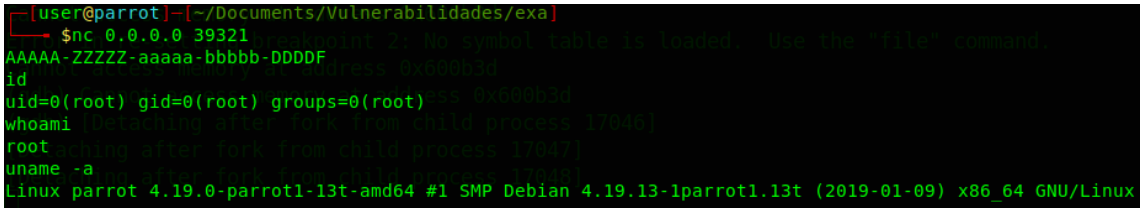

Figura 47. Obtención de una shell.

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-lMCCC-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.

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.

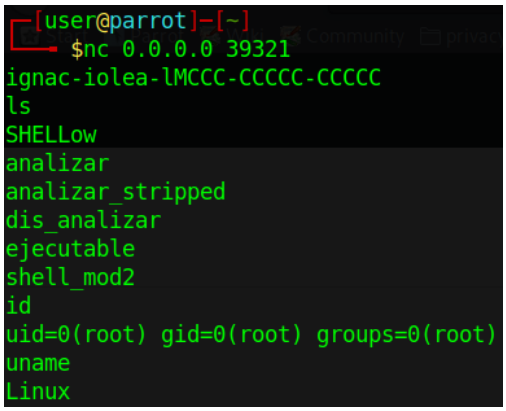

Figura 49. Ingreso de una cadena valida para obtener una shell.