

# PRÁCTICA 7

## CTF 2: ANÁLISIS

Realizada por:

**Javier Rojas Horrillo**

### A) ¿Qué bibliotecas usa el programa y qué funciones importa?

Para sacar las **bibliotecas** con las que **está enlazado** el binario, usamos el comando **ldd**:

```
remnux@remnux:~/Desktop/Malware/practica7$ ldd analizame
linux-vdso.so.1 (0x00007ffc9e5f6000)
libcrypto.so.1.1 => /lib/x86_64-linux-gnu/libcrypto.so.1.1 (0x00007fe8df821000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fe8df62f000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007fe8df629000)
libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007fe8df606000)
/lib64/ld-linux-x86-64.so.2 (0x00007fe8dfb18000)
```

En caso de que **no confiáramos en el binario**, **no** deberíamos **usar ldd** ya que en algunos casos puede ejecutar el programa.

Podríamos usar **alternativas** como la opción **-l** de **rabin2** o buscarlas en **cutter**.

Para listar las **funciones** utilizamos la opción **-i** de **rabin2**:

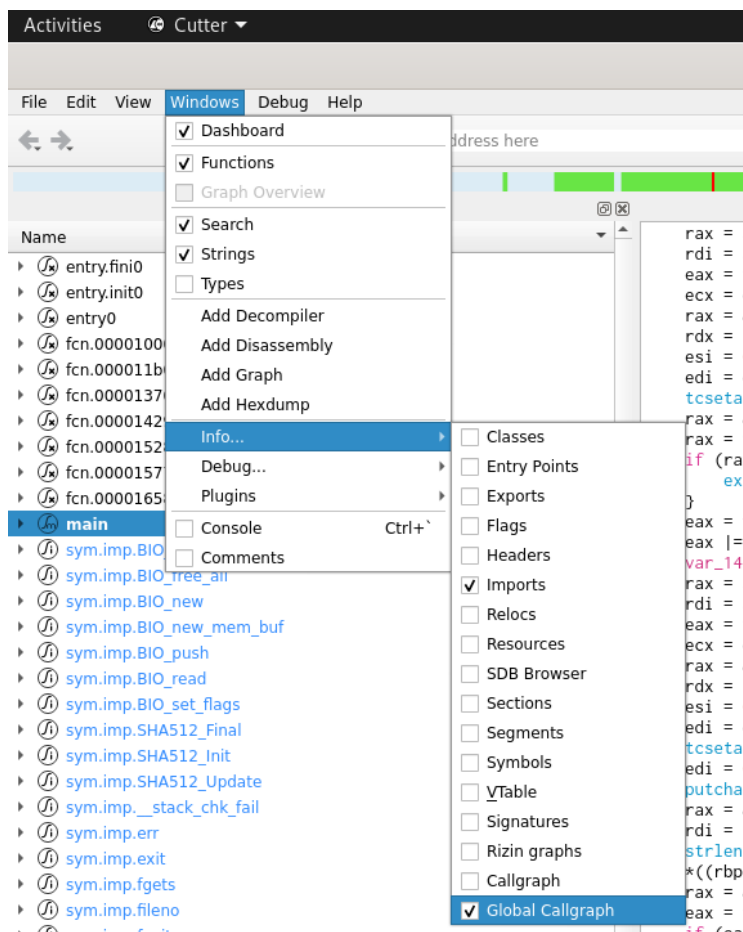
```
remnux@remnux:~/Desktop/Malware/practica7$ rabin2 -i analizame
[Imports]
nth vaddr      bind    type    lib name
-----
1  0x000011c0 GLOBAL FUNC    tcsetattr
2  0x000011d0 GLOBAL FUNC    fileno
3  0x000011e0 GLOBAL FUNC    printf
4  0x000011f0 GLOBAL FUNC    memset
5  0x00000000 WEAK  NOTYPE    __gmon_start__
6  0x00001200 GLOBAL FUNC    puts
7  0x00001210 GLOBAL FUNC    exit
8  0x00001220 GLOBAL FUNC    putchar
9  0x00000000 GLOBAL FUNC    __libc_start_main
10 0x00001230 GLOBAL FUNC    BIO_push
11 0x00001240 GLOBAL FUNC    fgets
12 0x00000000 WEAK  NOTYPE    _ITM_deregisterTMCloneTable
13 0x00001250 GLOBAL FUNC    strlen
14 0x00000000 WEAK  NOTYPE    _ITM_registerTMCloneTable
15 0x00001260 GLOBAL FUNC    BIO_new
16 0x00001270 GLOBAL FUNC    BIO_new_mem_buf
17 0x00001280 GLOBAL FUNC    err
18 0x00001290 GLOBAL FUNC    SHA512_Update
19 0x000012a0 GLOBAL FUNC    tcgetattr
20 0x000012b0 GLOBAL FUNC    BIO_f_base64
21 0x000012c0 GLOBAL FUNC    SHA512_Init
22 0x000012d0 GLOBAL FUNC    __stack_chk_fail
23 0x000012e0 GLOBAL FUNC    BIO_free_all
24 0x000012f0 GLOBAL FUNC    BIO_read
25 0x00001300 GLOBAL FUNC    BIO_set_flags
26 0x00001310 GLOBAL FUNC    SHA512_Final
27 0x00001320 GLOBAL FUNC    memcmp
28 0x00001330 GLOBAL FUNC    fwrite
29 0x00000000 WEAK  FUNC      __cxa_finalize
```

**B) Genera y pega en el PDF un grafo de llamadas global del programa (en formato horizontal para que se pueda ver fácilmente).**

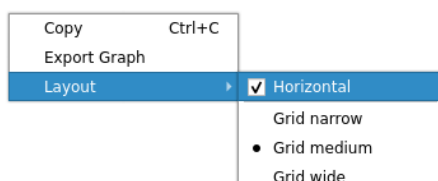
Abrimos el binario **cutter**:

```
remnux@remnux:~/Desktop/Malware/practica7$ cutter analizame
"0.4.0" "0.4.0"
Setting PYTHONHOME = "/tmp/.mount_cutteriE0rTX/usr" for AppImage.
PYTHONHOME = "/tmp/.mount_cutteriE0rTX/usr"
Plugins are loaded from "/home/remnux/.local/share/rizin/cutter/plugins"
Native plugins are loaded from "/home/remnux/.local/share/rizin/cutter/plugins/native"
Python plugins are loaded from "/home/remnux/.local/share/rizin/cutter/plugins/python"
Loaded 0 plugin(s).
Plugins are loaded from "/tmp/.mount_cutteriE0rTX/usr/share/ubuntu/rizin/cutter/plugins"
Plugins are loaded from "/tmp/.mount_cutteriE0rTX/usr/local/share/rizin/cutter/plugins"
Plugins are loaded from "/tmp/.mount_cutteriE0rTX/usr/share/rizin/cutter/plugins"
Native plugins are loaded from "/tmp/.mount_cutteriE0rTX/usr/share/rizin/cutter/plugins/native"
Loaded 2 plugin(s).
Plugins are loaded from "/var/lib/snapd/desktop/rizin/cutter/plugins"
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze function calls
[x] Analyze len bytes of instructions for references
[x] Check for classes
[x] Analyze local variables and arguments
[x] Type matching analysis for all functions
[x] Applied 0 FLIRT signatures via sigdb
[x] Propagate noreturn information
[x] Use -AA or aaaa to perform additional experimental analysis.
```

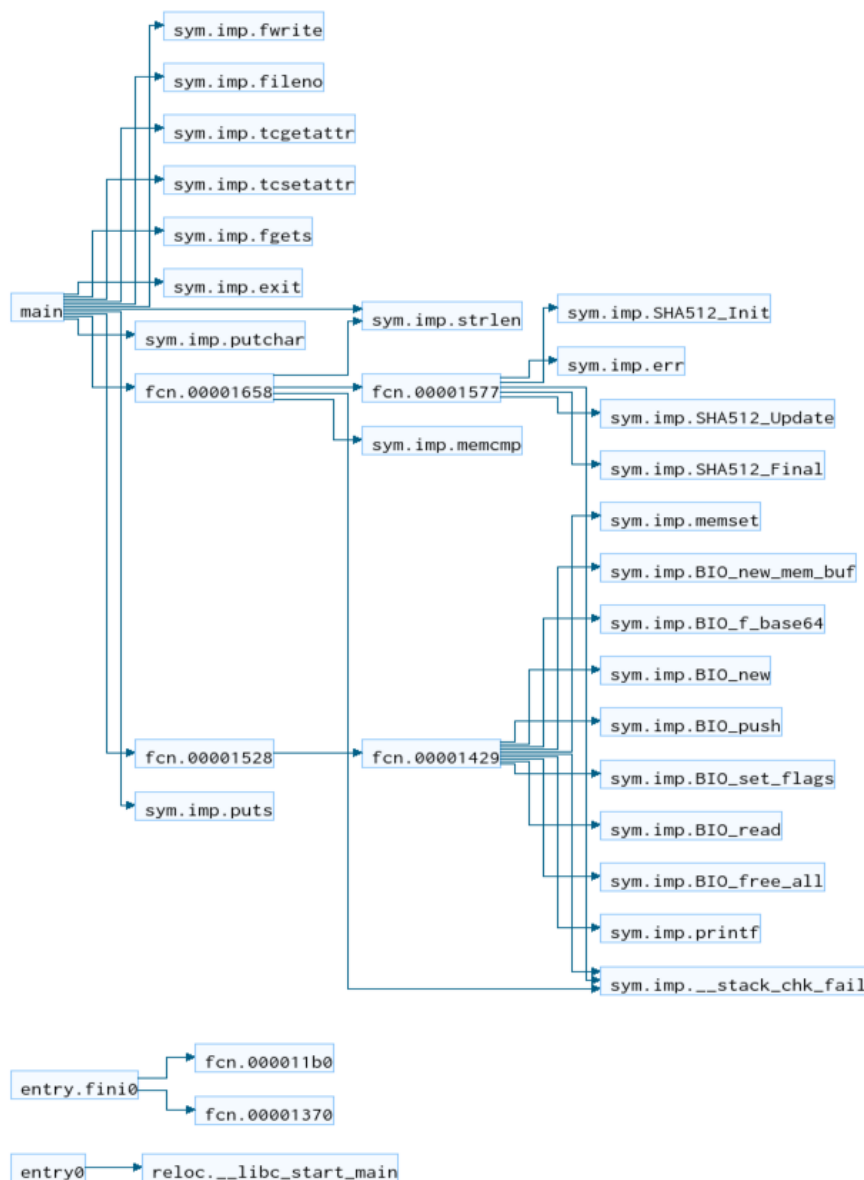
Desde cutter hacemos lo siguiente para mostrar el **grafo de llamadas global del programa**:



Para que el grafo se **extienda en horizontal** para una mejor visualización, debemos marcar lo siguiente:



El **grafo de llamadas global del programa** es el siguiente:



**C) Crea una copia del binario y parchea esa copia para que se muestre el mensaje secreto sin tener que saber la contraseña. ¿Cómo has dado con la condición? ¿Cuál es el mensaje secreto? Explica cómo lo has hecho y pega en el PDF una captura del terminal después de ejecutar el binario parcheado.**

Decompilando el **main** podemos ver que hay una instrucción que **evalúa** si se **devuelve** el **mensaje secreto** o no dependiendo de la contraseña que se le ponga.

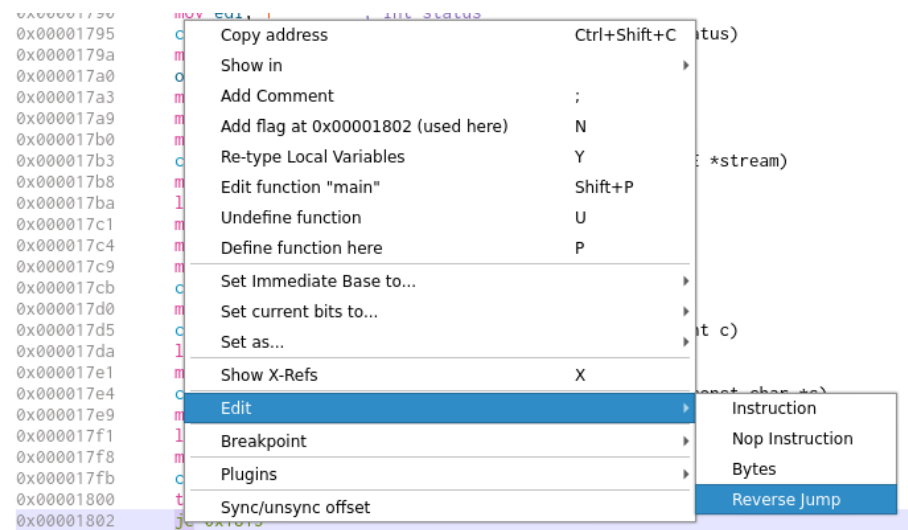
```
eax = fcn_00001658 (rax);  
if (eax != 0) {  
    fcn_00001528 ();  
    exit (0);  
}  
puts ("wrong password");  
return exit (1);
```

La **dirección** de esta instrucción es **0x00001802**, podemos cargar el binario para escritura con la flag **-w** e **invertir** esa **instrucción** para que **siempre devuelva** el **mensaje** oculto (a no ser que sea la contraseña correcta).

La instrucción en ensamblador es la siguiente:

```
0x00001802  je 0x1813
0x00001804  call fcn.00001528 ; fcn.00001528(void)
0x00001809  mov edi, 0 ; int status
0x0000180e  call exit ; sym.imp.exit ; void exit(int status)
0x00001813  lea rdi, str.wrong_password ; 0x2039 ; const char *s
```

Con cutter podemos **invertir la condición** de la siguiente forma:

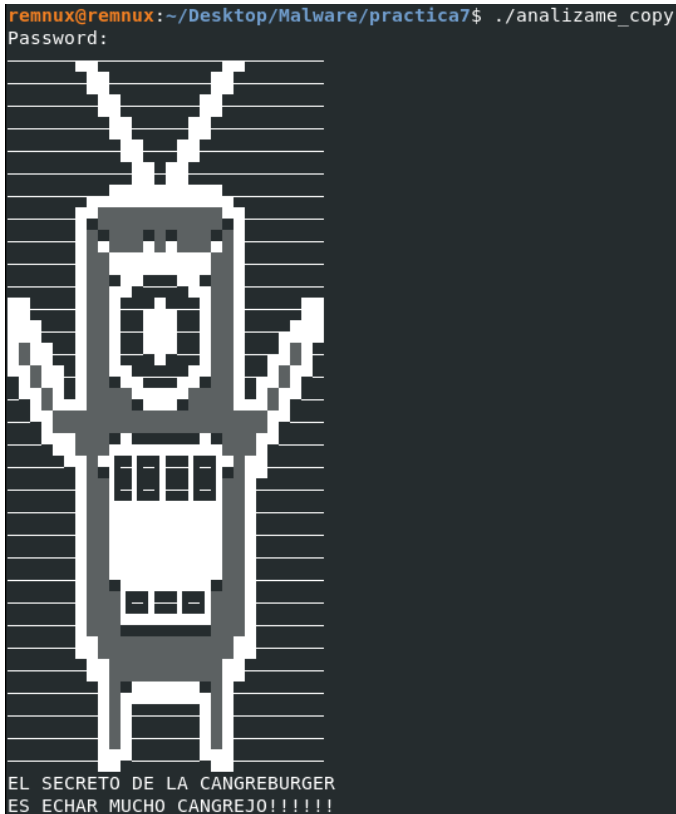


Quedando el código de la siguiente forma:

```
0x00001802  jne 0x1813
0x00001804  call fcn.00001528
0x00001809  mov edi, 0 ; int status
0x0000180e  call exit ; sym.imp.exit ; void exit(int status)
0x00001813  lea rdi, str.wrong_password ; 0x2039 ; const char *s

eax = fcn_00001658 (rax);
if (eax == 0) {
    fcn_00001528 ();
    exit (0);
}
puts ("wrong password");
```

Ahora cuando introduzcamos **cualquier cosa que no sea la contraseña** correcta, nos devolverá lo siguiente:



**D) ¿Dónde y cómo está almacenada la información secreta que escribe el binario por su salida cuando la contraseña es correcta? ¿Qué función o funciones se encargan de escribir el mensaje secreto por la salida? Especifica la sección y el offset en el que se encuentran los datos en el binario y si están codificados/ofuscados.**

En el apartado de *Strings* podemos ver que hay cierta **información** que parece **codificada** en la sección **.data** con **offset 0x00004080**. Eso puede significar que es la **información secreta codificada**:

[illegible]

Quick Filter

96 Items

Dashboard Strings Imports Search Global Callgraph Disassembly Hexdump

En cuanto a la **función muestra el mensaje secreto**, como podemos ver en el código del apartado anterior, la función que se encuentra **dentro del if** y que se podría encargar de mostrar la información secreta es **fcn 00001528**:

```
eax = fcn_00001658 (rax);
if (eax == 0) {
    fcn_00001528 ();
    exit (0);
}
puts ("wrong password");
```

Esta función, a su vez **llama a otra función**, fcn\_00001429:

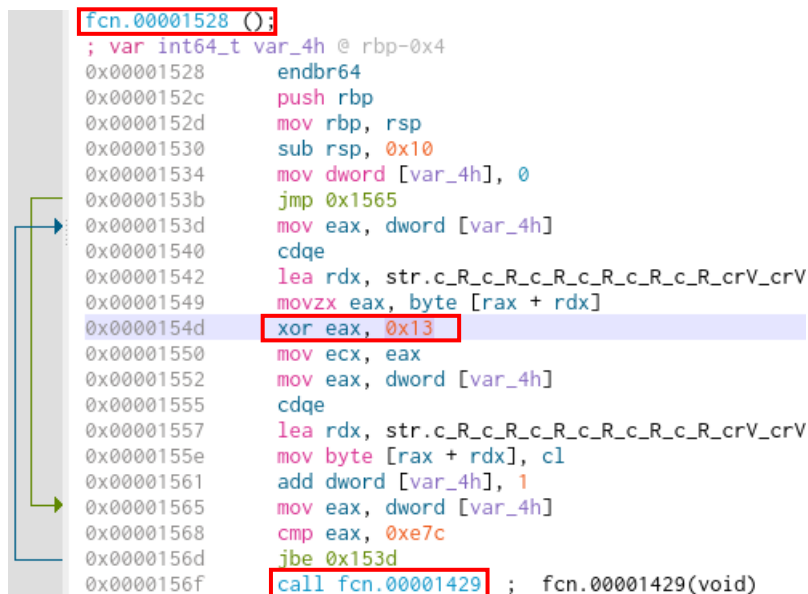
```
int64_t fcn_00001528 (void) {  
    int64_t var_4h;  
    var_4h = 0;  
    while (eax <= 0xe7c) {  
        eax = var_4h;  
        rax = (int64_t) eax;  
        rdx = 'c@R'c@R'c@R'c@R'c@R'c@R'crV'crV'c@R'c@R'c@R'c@R'c@R'c@R'c@R'c@R'c@R'c@R'c@R'c@R'crV'crV'c@R'c@R'c@R'  
        eax = *((rax + rdx));  
        eax ^= 0x13;  
        ecx = eax;  
        eax = var_4h;  
        rax = (int64_t) eax;  
        rdx = 'c@R'c@R'c@R'c@R'c@R'c@R'crV'crV'c@R'c@R'c@R'c@R'c@R'c@R'c@R'c@R'c@R'c@R'c@R'c@R'crV'crV'c@R'c@R'c@R'  
        *((rax + rdx)) = cl;  
        var_4h++;  
        eax = var_4h;  
    }  
    fcn_00001429 ();  
    return rax;  
}  
  
int64_t fcn_00001429 (void) {  
    int64_t var_1020h;  
    int64_t var_1018h;  
    void * s;  
    int64_t canary;  
    rax = *(fs:0x28);  
    canary = *(fs:0x28);  
    eax = 0;  
    rax = &s;  
    memset (rax, 0, fcn.00001000);  
    esi = 0xffffffff;  
    rdi = 'c@R'c@R'c@R'c@R'c@R'c@R'crV'crV'c@R'c@R'c@R'c@R'c@R'c@R'c@R'c@R'c@R'c@R'c@R'c@R'crV'crV'c@R'c@R'c@R'c@R'  
    rax = BIO_new_mem_buf ();  
    var_1020h = rax;  
    rax = BIO_f_base64 ();  
    rdi = rax;  
    rax = BIO_new ();  
    var_1018h = rax;  
    rdx = var_1020h;  
    rax = var_1018h;  
    rsi = rdx;  
    rdi = rax;  
    rax = BIO_push ();  
    var_1020h = rax;  
    esi = 0x100;  
    rdi = rax;  
    BIO_set_flags ();  
    rcx = &s;  
    rax = var_1020h;  
    edx = fcn.00001000;  
    rsi = rcx;  
    rdi = rax;  
    BIO_read ();  
    rax = var_1020h;  
    rdi = var_1020h;  
    BIO_free_all ();  
    rax = &s;  
    rsi = rax;  
    eax = 0;  
    printf (0x00002004);  
    rax = canary;  
    rax ^= *(fs:0x28);  
    if (? != ?) {  
        stack_chk_fail ();  
    }  
    return rax;  
}
```

Esas **2 funciones** son las **encargadas** de **mostrar** la información secreta.

**E) ¿Sabrías extraer la información secreta (decodificarla/desofuscarla) sin tener que parchear el binario y/o ejecutarlo? Si es así, indica cómo lo harías paso a paso (herramientas, etc.) y muestra el resultado.**

Como se ha comentado en el apartado anterior, en cutter podemos ver que hay 2 funciones que se encargan de mostrar el secreto.

En la función **fcn\_00001528** hay un bucle que va haciendo **XOR** de los bytes del **buffer** con **0x13** y a continuación llama a la otra función comentada:



```
fcn.00001528 ();  
; var int64_t var_4h @ rbp-0x4  
0x00001528     endbr64  
0x0000152c     push rbp  
0x0000152d     mov rbp, rsp  
0x00001530     sub rsp, 0x10  
0x00001534     mov dword [var_4h], 0  
0x0000153b     jmp 0x1565  
0x0000153d     mov eax, dword [var_4h]  
0x00001540     cdqe  
0x00001542     lea rdx, str.c_R_c_R_c_R_c_R_c_R_c_R_c_rV_crV  
0x00001549     movzx eax, byte [rax + rdx]  
0x0000154d     xor eax, 0x13  
0x00001550     mov ecx, eax  
0x00001552     mov eax, dword [var_4h]  
0x00001555     cdqe  
0x00001557     lea rdx, str.c_R_c_R_c_R_c_R_c_R_c_R_c_rV_crV  
0x0000155e     mov byte [rax + rdx], cl  
0x00001561     add dword [var_4h], 1  
0x00001565     mov eax, dword [var_4h]  
0x00001568     cmp eax, 0xe7c  
0x0000156d     jbe 0x153d  
0x0000156f     call fcn.00001429 ; fcn.00001429(void)
```

La función **fcn\_00001429** se encarga de decodificar **base64**:



```
fcn.00001429 ();  
; var int64_t var_1020h @ rbp-0x1020  
; var int64_t var_1018h @ rbp-0x1018  
; var void *s @ rbp-0x1010  
; var int64_t canary @ rbp-0x8  
0x00001429     endbr64  
0x0000142d     push rbp  
0x0000142e     mov rbp, rsp  
0x00001431     sub rsp, fcn.00001000 ; 0x1000  
0x00001438     or qword [rsp], 0  
0x0000143d     sub rsp, 0x20  
0x00001441     mov rax, qword fs:[0x28]  
0x0000144a     mov qword [canary], rax  
0x0000144e     xor eax, eax  
0x00001450     lea rax, [s]  
0x00001457     mov edx, fcn.00001000 ; 0x1000 ; size_t n  
0x0000145c     mov esi, 0 ; int c  
0x00001461     mov rdi, rax ; void *s  
0x00001464     call memset ; sym.imp.memset ; void *memset(void *s,  
0x00001469     mov esi, 0xffffffff ; -1  
0x0000146e     lea rdi, str.c_R_c_R_c_R_c_R_c_R_c_R_c_rV_crV_c_R_c_R_c_R_c  
0x00001475     call BIO_new_mem_buf ; sym.imp.BIO_new_mem_buf  
0x0000147a     mov qword [var_1020h], rax  
0x00001481     call BIO_f_base64 ; sym.imp.BIO_f_base64  
0x00001486     mov rdi, rax  
0x00001489     call BIO_new ; sym.imp.BIO_new  
0x0000148e     mov qword [var_1018h], rax  
0x00001495     mov rdx, qword [var_1020h]  
0x0000149c     mov rax, qword [var_1018h]  
0x000014a3     mov rsi, rdx  
0x000014a6     mov rdi, rax  
0x000014b0     call BIO_write ; sym.imp.BIO_write
```



Javier Rojas Horrillo  
Malware y Amenazas Dirigidas

Por lo tanto, para obtener el mensaje secreto **cogemos** los **bytes** de la dirección donde hemos dicho antes que se **encontraba el secreto (0x00004080)** y nos ayudamos de herramientas como **CyberChef** para **hacer la conversión**:

[illegible][illegible]

**F) ¿Y realizando un análisis dinámico? Si es así, indica cómo lo harías paso a paso (herramientas, etc.) y muestra el resultado.**

Para realizar el análisis dinámico vamos a utilizar la herramienta **radare2**.

Necesitamos **ejecutar el código**, por lo tanto, le asignamos **permisos de ejecución** al binario y lo abrimos con **r2** en **modo depuración (-d)**.

```
remnux@remnux:~/Desktop/Malware/practica7$ chmod +x analizame
remnux@remnux:~/Desktop/Malware/practica7$ r2 -d analizame
-- Mess with the best, Die like the rest
[0x7f6269fd3100]>
```

Ponemos un **breakpoint** en **main** (db main) y vemos las **siguientes 100 instrucciones** (pd 100).

```
[0x7fdbc9982100]> db main
[0x7fdbc9982100]> dc
hit breakpoint at: 0x55fd260d96cd
[0x55fd260d96cd]> pd 100
;-- main:
;-- rax:
;-- rip:
0x55fd260d96cd b f30f1efa endbr64
0x55fd260d96d1 55 pushq %rbp
0x55fd260d96d2 4889e5 movq %rsp, %rbp
0x55fd260d96d5 4881ec600100 subq $0x160, %rsp
0x55fd260d96dc 89bdacfeffff movl %edi, -0x154(%rbp)
0x55fd260d96e2 4889b5a0feff movq %rsi, -0x160(%rbp)
0x55fd260d96e9 64488b042528 movq %fs:0x28, %rax
0x55fd260d96f2 488945f8 movq %rax, -8(%rbp)
0x55fd260d96f6 31c0 xorl %eax, %eax
0x55fd260d96f8 488b05213800 movq reloc.stderr, %rax ;
68a5c0
```

Encontramos la **instrucción** donde se **evalúa** si la **contraseña es correcta** o no:

```
0x55fd260d97f1 488d85f0feff leaq -0x110(%rbp), %rax
0x55fd260d97f8 4889c7 movq %rax, %rdi
0x55fd260d97fb e858feffff callq 0x55fd260d9658
0x55fd260d9800 85c0 testl %eax, %eax
0x55fd260d9802 740f jle 0x55fd260d9813
0x55fd260d9804 e81ffdffff callq 0x55fd260d9528
0x55fd260d9809 bf00000000 movl $0, %edi
0x55fd260d980e e8fd99ffff callq sym.imp.exit
0x55fd260d9813 488d3d1f0800 leaq str.wrong_password, %rdi
0x55fd260d981a e8e1f9ffff callq sym.imp.puts
0x55fd260d981f bf01000000 movl $1, %edi
0x55fd260d9824 e8e7f9ffff callq sym.imp.exit
0x55fd260d9829 0f1f80000000 nopl (%rax)
```

Una vez sabemos la dirección de la instrucción, ponemos un **breakpoint** ahí y **ejecutamos** hasta este:

```
[0x55fd260d96cd]> db 0x55fd260d9800
[0x55fd260d96cd]> dc
Password:
hit breakpoint at: 0x55fd260d9800
[0x55fd260d9800]>
```

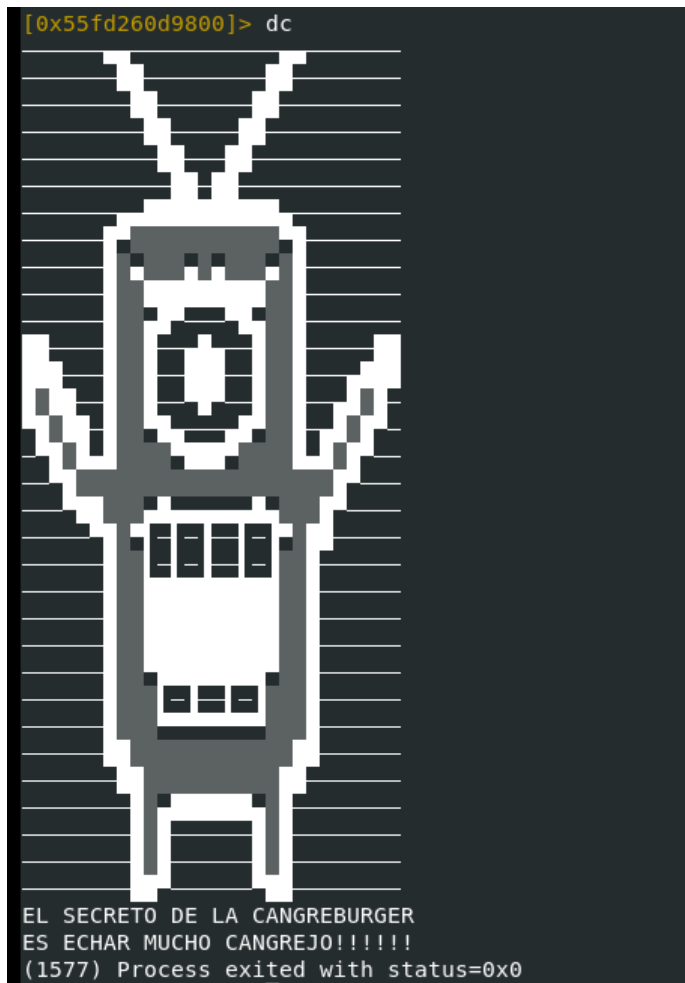
Ahora debemos **modificar** (dr) el valor de **eax** para que el **resultado** de **test** sea **favorable** y se interprete como que la **contraseña** introducida es **correcta**.

```
if (eax != 0) {  
    fcn_00001528 ();  
    exit (0);  
}
```

eax distinto de 0

```
[0x55fd260d9800]> dr eax  
0x00000000  
[0x55fd260d9800]> dr eax=0x01  
0x00000000 ->0x00000001  
[0x55fd260d9800]> dr eax  
0x00000001
```

Continuamos la ejecución del programa y nos muestra el mensaje oculto.



**G) ¿Sabes cómo se comprueba si la contraseña introducida es correcta? ¿Qué función o funciones se encargan de esto? Indica en qué sección y offset está la contraseña almacenada, su formato (si está cifrada, etc.) y qué podrías intentar hacer para conseguir esa contraseña en claro.**

La contraseña es correcta cuando la función **fcn\_00001658** retorna algo **distinto de 0**, ya que así se **cumplirá** la **condición** del **if**:

```
rax = &s;  
eax = fcn_00001658 (rax);  
if (eax != 0) {  
    fcn_00001528 ();  
    exit (0);  
}
```

En esta función, con diferentes **operaciones** y **asignaciones de relleno**, lo realmente importante es que se **compara** (con `memcmp()`) la **cadena introducida** (`arg1`) con la **contraseña** la cual se encuentra **almacenada** en **.data** con offset **0x00004020**.

```
int64_t fcn_00001658 (const char * arg1) {  
    const char * s;  
    const void * s1;  
    int64_t canary;  
    rdi = arg1;  
    s = rdi;  
    rax = *(fs:0x28);  
    canary = *(fs:0x28);  
    eax = 0;  
    rax = s;  
    rdi = s;  
    eax = strlen ();  
    ecx = eax;  
    rax = s;  
    fcn_00001577 (rax, ecx, s1);  
    rax = &s1;  
    edx = 0x40;  
    rsi = 0x00004020;  
    rdi = rax;  
    eax = memcmp ();  
    al = (eax == 0) ? 1 : 0;  
    eax = (int32_t) al;  
    rcx = canary;  
    rcx ^= *(fs:0x28);  
    if (eax != 0) {  
        stack_chk_fail ();  
    }  
    return rax;  
}
```

Podemos ver que se llama a la función **fcn\_00001577**, esta lo que hace es **transformar** la contraseña que se introduce con **SHA-512**, ya que la **contraseña real** **está almacenada de esta forma** y así poder **compararlas**:

```
int64_t fcn_00001577 (int64_t arg3, int64_t arg2, const char * arg1) {  
    int64_t var_f8h;  
    int64_t var_ech;  
    const char * var_e8h;  
    int64_t var_e0h;  
    int64_t canary;  
    rdx = arg3;  
    rsi = arg2;  
    rdi = arg1;  
    var_e8h = rdi;  
    var_ech = esi;  
    var_f8h = rdx;  
    rax = *(fs:0x28);  
    canary = *(fs:0x28);  
    eax = 0;  
    rax = &var_e0h;  
    rdi = rax;  
    eax = SHA512_Init ();  
    if (eax == 0) {  
        rsi = "SHA512_init";  
        edi = 1;  
        eax = 0;  
        err ();  
    }  
    eax = var_ech;  
    rdx = (int64_t) eax;  
    rcx = var_e8h;  
    rax = &var_e0h;  
    rsi = rcx;  
    rdi = rax;  
    eax = SHA512_Update ();  
    if (eax == 0) {  
        rsi = "SHA512_Update";  
        edi = 1;  
        eax = 0;  
        err ();  
    }  
    rdx = &var_e0h;  
    rax = var_f8h;  
    rsi = rdx;  
    rdi = rax;  
    eax = SHA512_Final ();  
    if (eax == 0) {  
        rsi = "SHA512_Final";  
        edi = 1;  
        eax = 0;  
    }  
}
```

Para conseguir la contraseña, como está almacenada con una función **hash SHA-512** es **difícil sacarla** en claro.

Por lo tanto, lo que se podría hacer es un **ataque de fuerza bruta** hasta dar con ella y que devuelva el secreto, utilizando **herramientas** como ***JohnTheRipper***.

En este caso **no hay un máximo número de intentos**, pero en caso de haberlo se podría dejar el **binario modificado** para que se acceda al secreto con **cualquier** cosa **menos** la **contraseña**, como hemos hecho en la práctica, y **si no se accede** sabremos que es la contraseña **correcta**.