

## CÁTEDRA DE CIBERSEGURIDAD CIBERUGR, INCIBE-UGR

Nombre	Reverse cipher
Categoría	REVERSING
Dificultad	MEDIA
Puntos	300

### DESCRIPCIÓN DEL RETO

Nuestro amigo ha empezado a aprender sobre criptografía y dice que ha estado creando un programa de C++ para encriptar frases. Cree que sabe bastante sobre criptografía y que lo que ha creado es seguro, por lo tanto, nos ha propuesto un juego que consiste en que intentemos descifrar la frase que utilizó con este programa, dándonos el resultado del programa al usar esta frase y el programa compilado.

### WRITEUP

#### PRIMEROS PASOS:

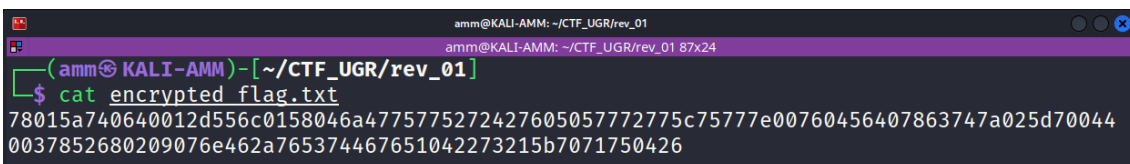
Nos encontramos con únicamente dos archivos, un archivo de texto llamado `encrypted_flag.txt` y otro archivo binario llamado `crypt.bin`

1. Si se lee la flag (archivo txt) se ve una string que se puede presuponer que está en forma hexadecimal ya que todos los caracteres están en el rango de 0 a 9 y entre A y F.

---

*cat encrypted\_flag.txt*

---



```
amm@KALI-AMM: ~/CTF_UGR/rev_01
amm@KALI-AMM: ~/CTF_UGR/rev_01 87x24
(amm@KALI-AMM) - [~/CTF_UGR/rev_01]
$ cat encrypted_flag.txt
78015a740640012d556c0158046a4775775272427605057772775c75777e00760456407863747a025d70044
0037852680209076e462a765374467651042273215b7071750426
```

2. Si se ejecuta el comando `file` con el archivo `crypt.bin` se puede ver que es un archivo binario ejecutable (Los archivos binarios ejecutables en Linux son los llamados ELF).

---

*file crypt.bin*

---

```
(amm@KALI-AMM)~/CTF_UGR/rev_01
└─$ file crypt.bin
crypt.bin: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter
/lib64/ld-linux-x86-64.so.2, BuildID[sha1]=2eb403c8b8a8226a0f215e1c8ccf2fb47c18a501, for GNU/Linux 3
.2.0, not stripped
```

3. Si se ejecuta el binario da un error y explica su funcionamiento, indicando que necesita introducir una cadena en el primer argumento.
4. Si ejecutamos el binario pasando una string, nos encontramos con que devuelve una “string en hexadecimal cifrada”, lo que es parecido a lo que tenemos en el archivo de texto, por lo tanto, suponemos que lo que tenemos en el archivo de texto ha sido generado mediante este binario.

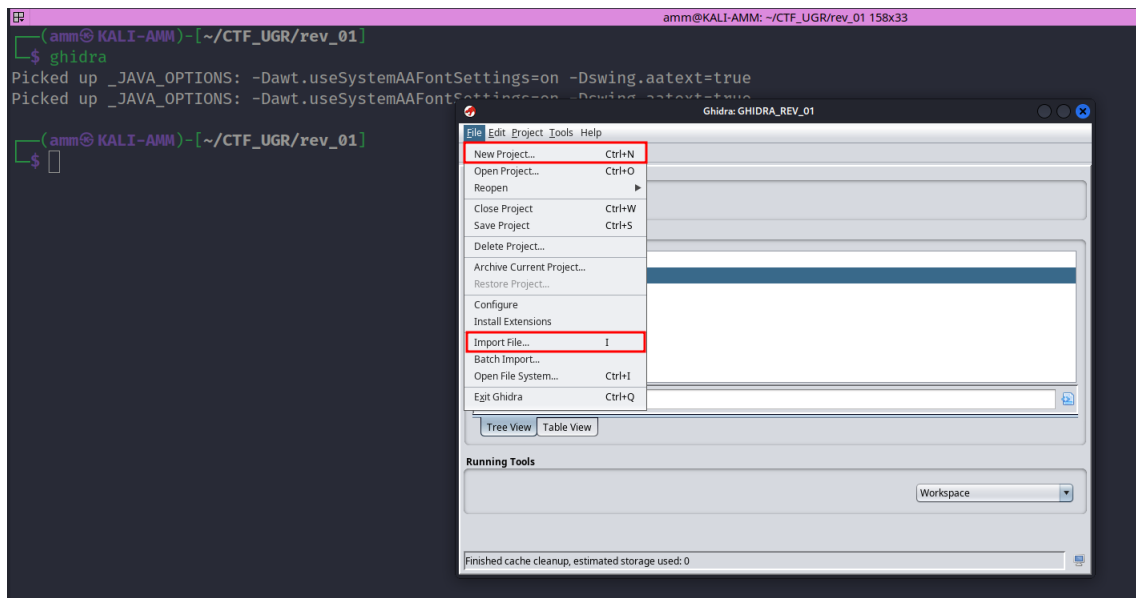
```
amm@KALI-AMM: ~/CTF_UGR/rev_01
└─$ ./crypt.bin
[ ! ERROR ! ] No se ha introducido la cadena
[ USO ] ./crypt.bin <cadena>

(amm@KALI-AMM)~/CTF_UGR/rev_01
└─$ ./crypt.bin "Test"
ENCRYPTED HEX STRING: 780058760441037f

(amm@KALI-AMM)~/CTF_UGR/rev_01
└─$
```

Para saber más sobre cómo funciona el programa deberemos hacer reversing usando GHIDRA:

1. Abrir GHIDRA → Crear un proyecto → Importar el archivo “crypt.bin”



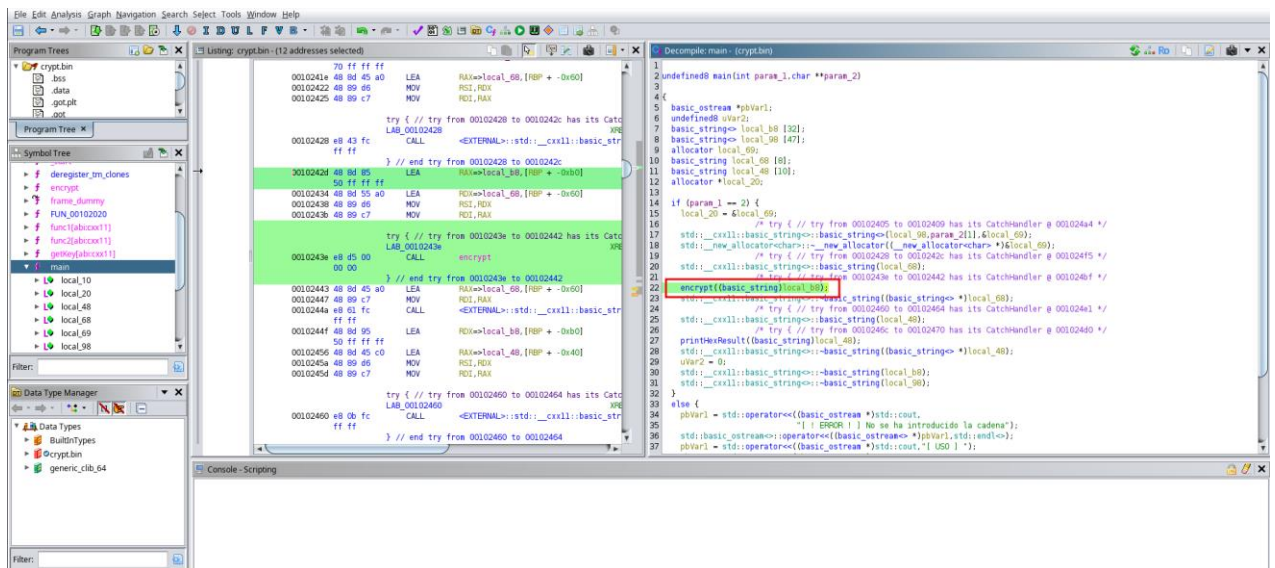
2. Hacer doble click en “crypt.bin” dentro de la ventana de ghidra.
3. Cuando pregunte por analizar, darle a “Yes” → Seleccionar “Aggressive Instruction Finder” (aparte de las opciones ya seleccionadas) y darle a continuar. (Esta opción sirve para que GHIDRA al decompilar intente entender mejor todas

las instrucciones y en el apartado de “código en C” que saca a partir del código en ensamblador ponga el nombre real que tienen las instrucciones que está decompilando, ojo, puede fallar, pero en un caso sencillo como esté puede ser de gran ayuda)

4. Espera a que termine de analizar.

### ANÁLISIS CON GHIDRA:

En el apartado de funciones (Symbol Tree) echar un vistazo y empezar por “main”:



The screenshot shows the Ghidra interface with the Symbol Tree on the left, where 'main' is selected. The main window displays the decompiled C code for the main function, which includes an if-statement and a call to an 'encrypt' function. The console at the bottom is empty.

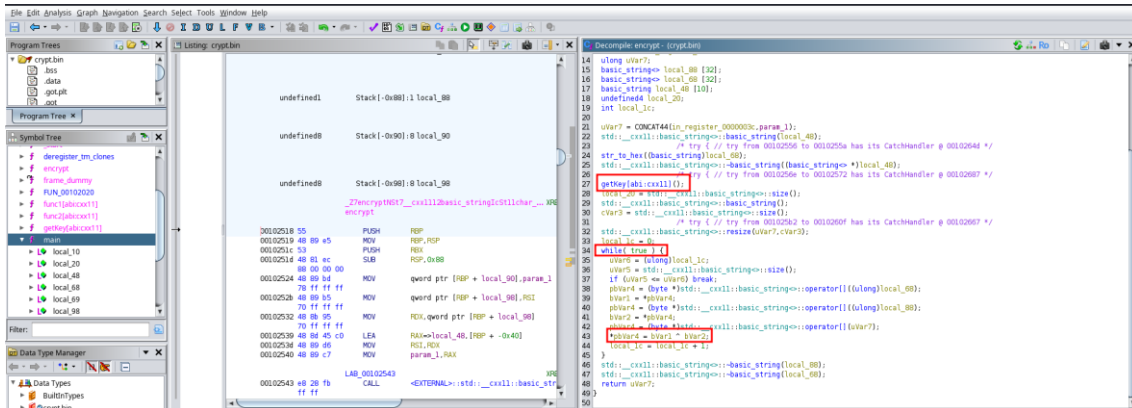
La función main es la función principal que se ejecuta al llamar al programa, tenemos que ir mirando el flujo que sigue el programa para entender que hace.

Podemos ver que hay un if, donde si se cumple la condición, poco después se ejecuta una función llamada “encrypt”, esta función llama la atención pues sabemos que el programa cifra de alguna manera la string que le pasamos, por lo tanto, vamos a ir tirando del hilo comprobando que hace la función.

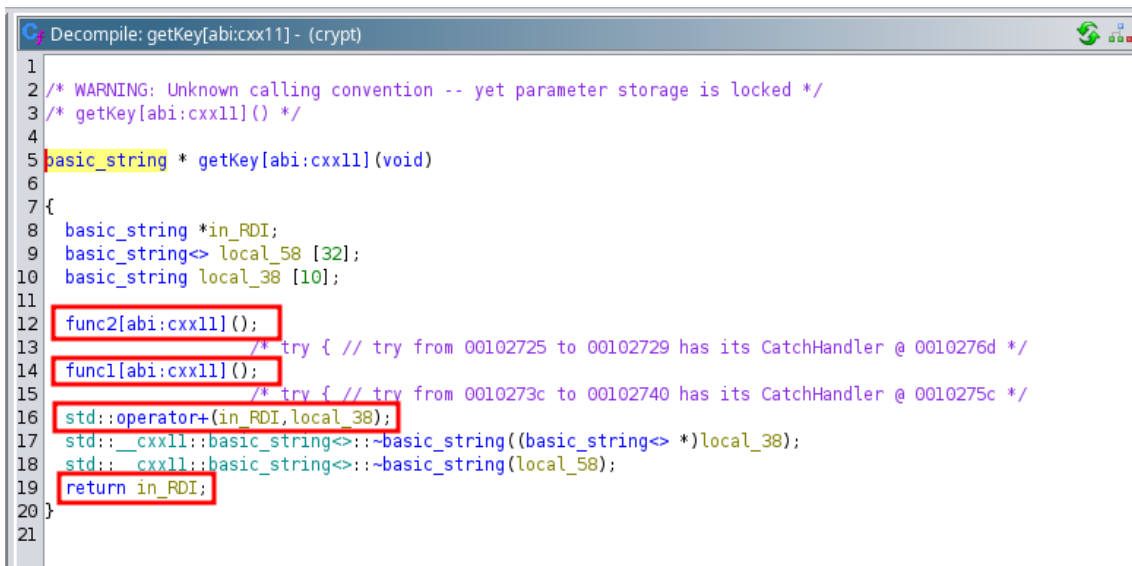
→Hacer doble click en “encrypt”

Dentro de encrypt se llama a una función “str\_to\_hex”. Presuponemos que transforma la string pasada a formato hexadecimal.

Luego se llama a una función “getKey” y en una parte se realiza una **operación XOR** entre dos variables, después de que se use el operador [], el cuál generalmente se usa para acceder a caracteres de una string o elementos en un array/vector, en este caso seguramente los elementos con los que se hace la operación XOR son, los resultantes del operador [] que lo que hará será acceder a los valores hexadecimales, y los elementos de la clave.

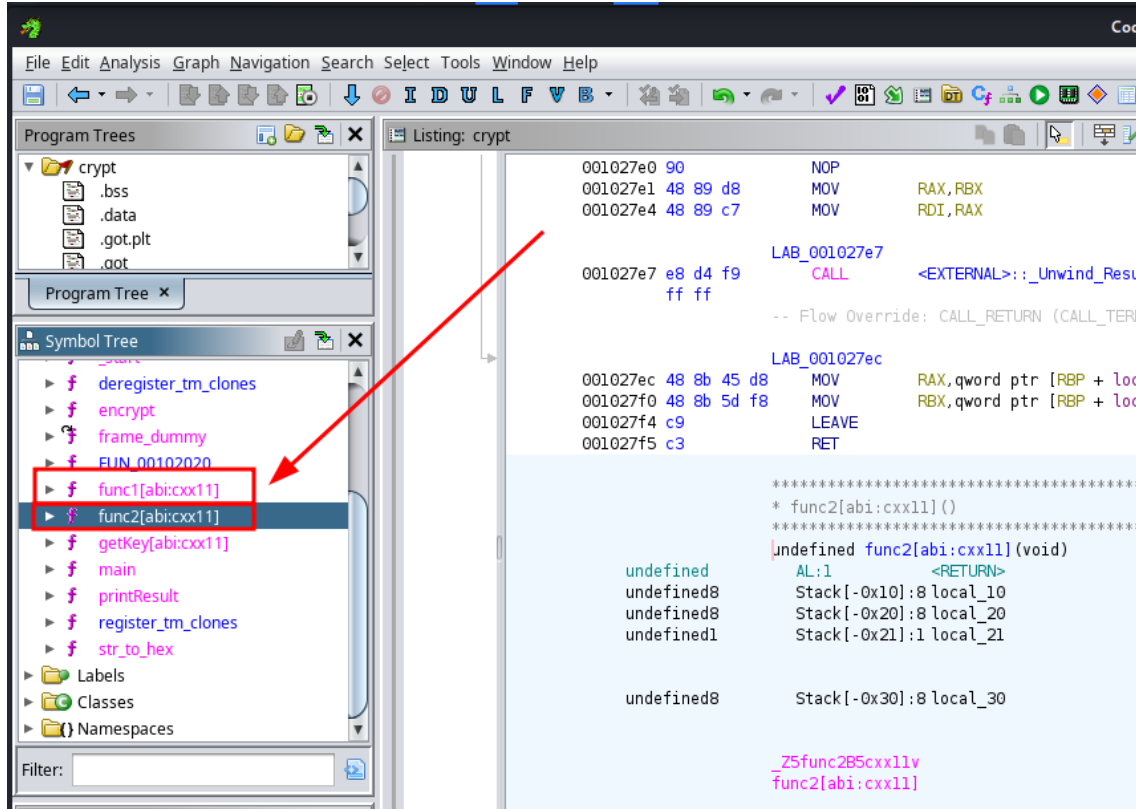


Hacer doble click en la función “getKey”:



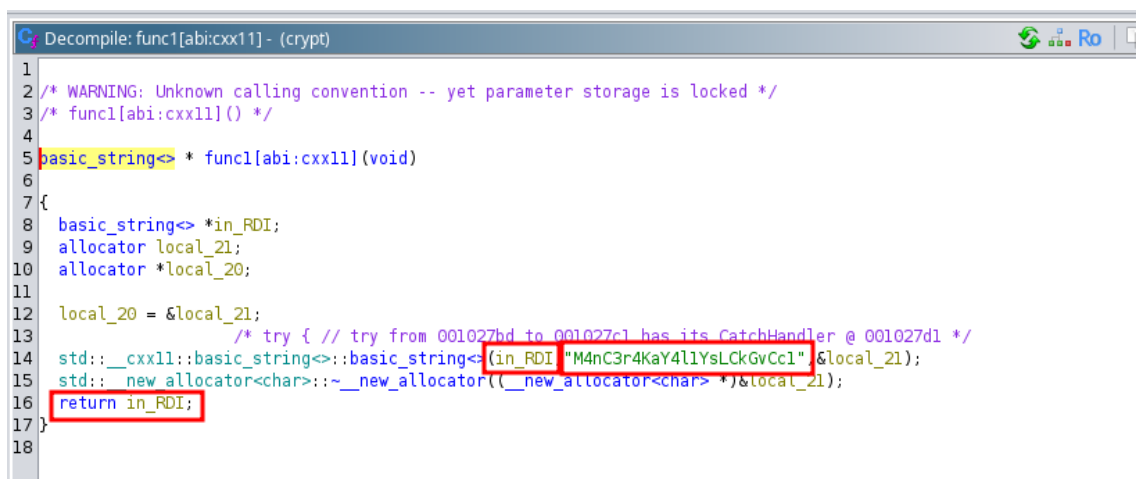
La función llama a dos funciones “func1” y “func2”, realiza una operación de suma y devuelve el resultado.

Comprobar “func1” y “func2” haciendo click sobre los nombres o haciendo click en los nombres de estas funciones mostrados en el árbol de símbolos:



The screenshot shows a debugger interface with a 'Symbol Tree' on the left and an assembly listing on the right. In the 'Symbol Tree', the functions 'func1[abi:cxx11]' and 'func2[abi:cxx11]' are highlighted with a red box. A red arrow points from this box to the assembly listing, which shows the implementation of 'func2[abi:cxx11]' starting at address 001027e7. The assembly listing includes instructions like 'NOP', 'MOV', 'CALL', and 'RET', along with stack frame information.

Func1:



The screenshot shows a decompiler window titled 'Decompile: func1[abi:cxx11] - (crypt)'. The code is as follows:

```

1
2 /* WARNING: Unknown calling convention -- yet parameter storage is locked */
3 /* func1[abi:cxx11]() */
4
5 basic_string<> * func1[abi:cxx11](void)
6
7 {
8     basic_string<> *in_RDI;
9     allocator local_21;
10    allocator *local_20;
11
12    local_20 = &local_21;
13    /* try { // try from 001027bd to 001027c1 has its CatchHandler @ 001027d1 */
14    std::__cxx11::basic_string<>::basic_string<>(in_RDI, "M4nC3r4KaY4l1YsLcKGVcC1", &local_21);
15    std::_new_allocator<char>::~_ new_allocator(( _ new_allocator<char> *)&local_21);
16    return in_RDI;
17 }
18
  
```

The return statement 'return in\_RDI;' is highlighted with a red box.

Devuelve una string llamada in\_RDI, se puede ver directamente la string que se devuelve.

Func2:

```
Decompile: func2[abi:cxx11] - (crypt)
1
2 /* WARNING: Unknown calling convention -- yet parameter storage is locked */
3 /* func2[abi:cxx11]() */
4
5 basic_string<> * func2[abi:cxx11](void)
6
7 {
8     basic_string<> *in_RDI;
9     allocator local_21;
10    allocator *local_20;
11
12    local_20 = &local_21;
13    /* try { // try from 00102822 to 00102826 has its CatchHandler @ 00102836 */
14    std::_cxx11::basic_string<>::basic_string<>(in_RDI, "DGChCDL3B34uJPG", &local_21);
15    std::_new_allocator<char>::~~_new_allocator((_new_allocator<char> *)&local_21);
16    return in_RDI;
17 }
18
```

La función “func2” es prácticamente igual que la función “func1” pero devuelve otro valor.

La clave será la suma (concatenación) de las dos strings que devuelven las funciones “func1” y “func” respectivamente, podemos suponer que la string devuelta por “func1” será la primera, tanto por el nombre de la función como porque en ambas funciones hemos visto que lo último que se ejecuta (y se devuelve) es lo que GHIDRA nombra como in\_RDI, y “func1” es la última función en llamarse.

Por lo tanto, la clave sería: “M4nC3r4KaY4I1YsLcKgvCc1DGChCDL3B34uJPG”

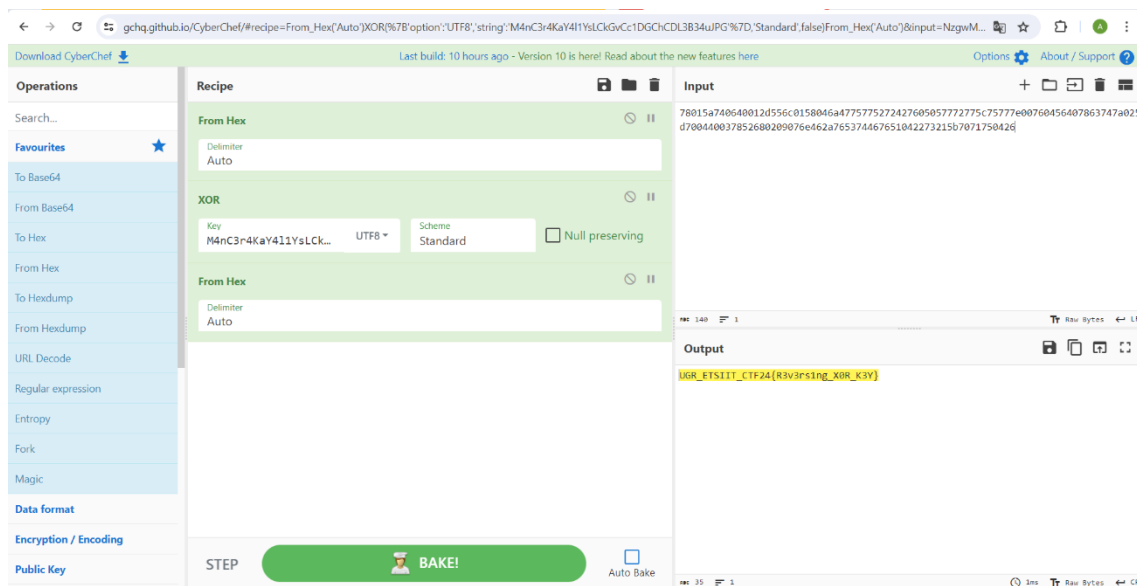
## SOLUCIÓN:

Sabiendo la clave y que probablemente el **cifrado** es solamente la **operación XOR** de la flag pasada a hexadecimal, **tendremos que realizar estas operaciones en orden inverso utilizando la clave que hemos extraído**, para realizar estos pasos he optado por usar la herramienta Cyberchef pero se podría realizar de muchas más formas (usando un script de Python o un programa en C++):

URL de CyberChef: <https://gchq.github.io/CyberChef/>

Los pasos en Cyberchef serán los siguientes:

- 1) Pasar de hexadecimal a string, para que Cyberchef tome la flag cifrada como una string. **(FROM HEX)**
- 2) Realizar la operación XOR usando la clave extraída. **(XOR)**  
**Nota:** Importante que la codificación de la clave sea UTF8 o Latin1, por defecto viene con formato HEX.
- 3) Pasar de hexadecimal a caracteres ASCII (string) **(FROM HEX)**



The screenshot shows the CyberChef web interface. The 'Recipe' panel on the left contains three steps: 'From Hex', 'XOR', and 'From Hex'. The 'Input' field contains a long hexadecimal string. The 'Output' field shows the decoded result: 'UGR\_ETS11T\_CTF24{R3v3rs1ng\_X0R\_K3Y}'.