PUE2020 – Resolución de desafíos

ESET Research Lab

5/19/2021 – version 1.1

Premio Universitario ESET 2020 - Resolución de desafíos

Contents

1_Captura.zip

En la captura vamos a encontrar 3 archivos copiados por SMB desde el equipo 102.168.86.24:

1) El archivo handshake.jpg que tiene una pista sobre el tipo de cifrado que usaremos después: Vigenère y un mensaje oculto con StegHide.
2) El archivo Tabla de quien.png que contiene la tabla estándar para descifrar el mensaje y una pista para encontrar la clave.
3) El archivo Mensaje.txt donde está el mensaje cifrado:

NIP UN ZXVNIAY WYKFNDVWW ZV VF ZGVUSHNDM.
GI TFSBM VS: EACWTARZWXBVCHAHTJOP

La clave para descifrar el mensaje se encuentra en los paquetes ICMP enviados desde el equipo 192.168.86.28. De los paquetes ICMP obtenemos la clave en Base64:
RUwgTUVOU0FKRSBERSBURVhUTyBTRSBERUNJRlJBIENPTiBMQSBDTEFWRTogR0lSQUFOVElWSVJVUw=
La pasamos a ASCII y tenemos la clave:

EL MENSAJE DE TEXTO SE DECIFRA CON LA CLAVE: GIRAANTIVIRUS

Utilizando la Tabla de Vigenère, se descifra el mensaje:

HAY UN MENSAJE ESCONDIDO EN EL HANDSHAKE.
LA CLAVE ES: ENJOYSAFERTECHNOLOGY

Utilizando StegHide podemos obtener el mensaje embebido en la imagen handshake.jpg que contiene el flag: **3s3T_PUE2o2o_1nt3rc3pt10n**.

2_TheWall.zip

El desafío consiste en obtener los valores hexadecimales de un archivo a partir de una secuencia de colores. En el archivo "the wall.bmp" encontramos una pared con ladrillos de colores y unos símbolos en la parte superior. El texto de la parte superior está escrito en Braille y traducido dice:

LA CLAVE DEL ZIP ES GOLONDRINAS

Luego, si decodificamos el valor hexadecimal de cada color de la pared obtenemos la siguiente matriz:

```
50 4B 03 04 14 00 01 00 00 00 2E 4E 73 51 9C 76
D5 96 24 00 00 00 18 00 00 00 08 00 00 00 66 6C
61 67 2E 74 78 74 0D E1 1B 59 21 71 87 93 B2 B4
C5 BD C0 7A 89 52 82 FB A9 2B 37 D9 46 9C 08 88
31 00 84 E8 AC E3 23 DE 9E 96 50 4B 01 02 3F 00
14 00 01 00 00 00 2E 4E 73 51 9C 76 D5 96 24 00
00 00 18 00 00 00 08 00 24 00 00 00 00 00 00 00
20 00 00 00 00 00 00 00 66 6C 61 67 2E 74 78 74
0A 00 20 00 00 00 00 00 01 00 18 00 C8 B8 B0 69
72 BE D6 01 C8 B8 B0 69 72 BE D6 01 27 3B 9A 65
72 BE D6 01 50 4B 05 06 00 00 00 00 01 00 01 00
5A 00 00 00 4A 00 00 00 00 00 00 00 00 00 00 00
```

La matriz corresponde al *offset* hexadecimal de un archivo "flag.zip", que sería esta:

```
OFFSET      flag.zip
------------------------------------------------------------------------------
0x00000000  50 4B 03 04 14 00 01 00 00 00 2E 4E 73 51 9C 76 |PK.........NsQ.v|
0x00000010  D5 96 24 00 00 00 18 00 00 00 08 00 00 00 66 6C |..$...........fl|
0x00000020  61 67 2E 74 78 74 0D E1 1B 59 21 71 87 93 B2 B4 |ag.txt...Y!q....|
0x00000030  C5 BD C0 7A 89 52 82 FB A9 2B 37 D9 46 9C 08 88 |...z.R...+7.F...|
0x00000040  31 00 84 E8 AC E3 23 DE 9E 96 50 4B 01 02 3F 00 |1.....#...PK..?.|
0x00000050  14 00 01 00 00 00 2E 4E 73 51 9C 76 D5 96 24 00 |.......NsQ.v..$.|
0x00000060  00 00 18 00 00 00 08 00 24 00 00 00 00 00 00 00 |........$.......|
0x00000070  20 00 00 00 00 00 00 00 66 6C 61 67 2E 74 78 74 |........flag.txt|
0x00000080  0A 00 20 00 00 00 00 00 01 00 18 00 C8 B8 B0 69 |...............i|
0x00000090  72 BE D6 01 C8 B8 B0 69 72 BE D6 01 27 3B 9A 65 |r......ir...';.e|
0x000000A0  72 BE D6 01 50 4B 05 06 00 00 00 00 01 00 01 00 |r...PK..........|
0x000000B0  5A 00 00 00 4A 00 00 00 00 00 XX XX XX XX XX XX |Z...J...........|
```

Para reconstruir el archivo ZIP, primero ponemos la secuencia hexadecimal en un archivo "Hexa.txt". No hace falta respetar el formato de la matriz, podemos poner directamente los valores hexadecimales sin espacios; sí es necesario respetar el orden. Luego, ejecutamos el comando:

```
xxd -r -p Hexa.txt out.bin
```

Al abrir con un editor de texto el archivo "out.bin" vemos que la cabecera del archivo es 'PK', lo cual indica que es un archivo del tipo ZIP.

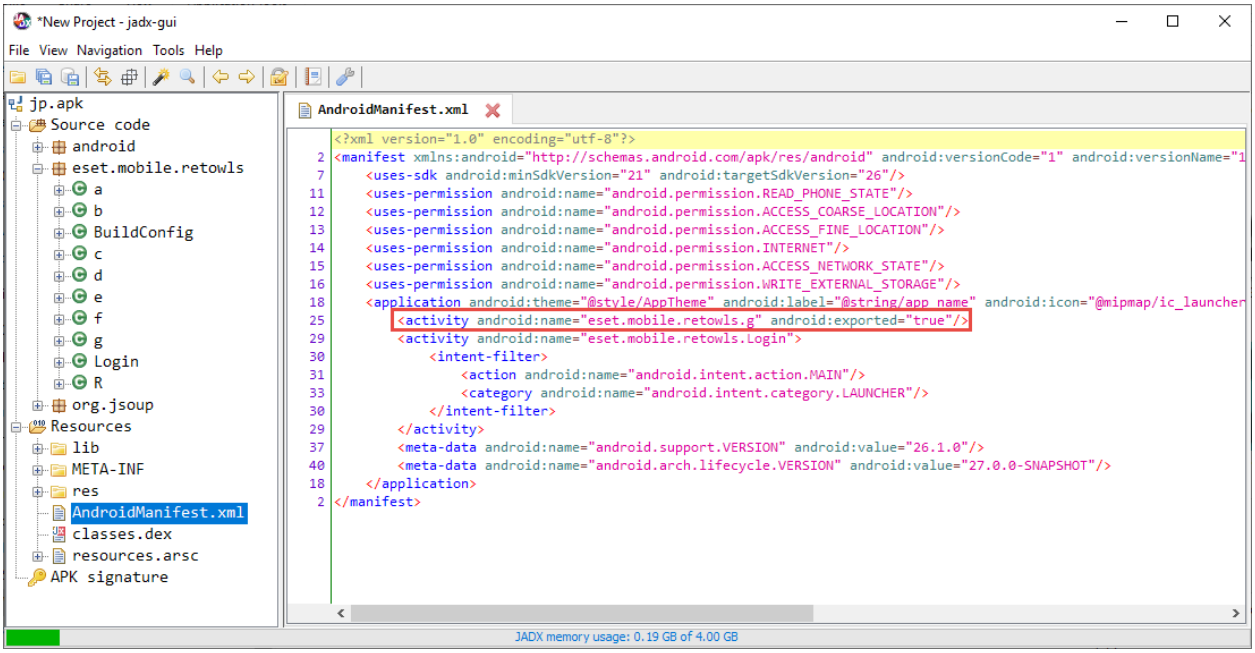
Cambiamos la extensión de "out.bin" a "out.zip" y abrimos el archivo. Dentro encontramos un archivo de texto que se llama "flag.txt".

Lo extraemos con la clave "GOLONDRINAS" y obtenemos el flag dentro del archivo de texto: **3s3T_PUE2o2o_H3x4d3c1m4L**.

3_DinosAreBack.apk

El desafío consiste en comprender la lógica de una aplicación para Android para poder encontrar el flag. Para ello, podemos comenzar haciendo un **análisis estático** de esta aplicación, aparentemente utilizada por los empleados del Parque Jurásico. Una herramienta muy intuitiva que podemos usar con este objetivo es jadx.

Al abrir el APK, una opción sería comenzar analizando los recursos para recolectar pistas que puedan estar allí almacenadas. Por ejemplo, si echamos un vistazo al manifiesto veremos que tenemos dos actividades. A pesar de que, al iniciarse, la app llama a la clase **Login**, tenemos otra actividad exportada **g**. Esto significará que probablemente deberemos llamarla utilizando adb durante el análisis dinámico.



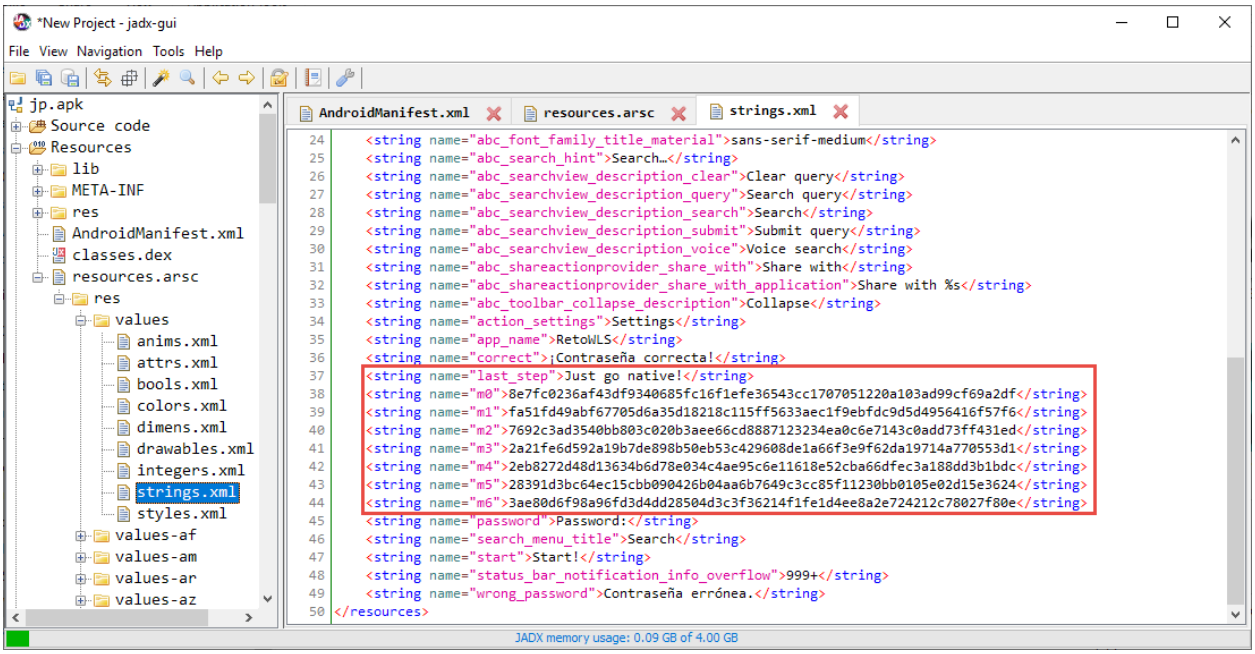
Indagando más en estas clases, vemos que el método **onCreate** de la clase **Login** chequea la existencia de los permisos de geolocalización, carga un layout y luego imprime la URL de un vídeo en los logs del sistema. Si visitamos el vídeo, veremos una representación de la clásica pantalla que aparecía ante un frustrado Ray Arnold en Parque Jurásico, negándole acceso al sistema. Pareciera indicarnos que este no es el camino.

```
public void onCreate(Bundle bundle) {
    super.onCreate(bundle);
    this.context = getApplicationContext();
    if (Build.VERSION.SDK_INT >= 23) {
        checkLocationPermission();
        checkReadPhoneState();
    }
    setContentView((int) R.layout.activity_log_in);
    Log.i("AHAHAH", "https://www.youtube.com/watch?v=K3PrSj9XEu4");
}
```

Sin embargo, si miramos el método **onCreate** de la actividad g, las cosas parecen más interesantes. Vemos que se guarda el texto "Welcome to Jurassic Park!" en el archivo de preferencias compartidas llamado "JP_prefs" bajo la clave "hammond".

```
public void onCreate(Bundle bundle) {
    super.onCreate(bundle);
    setContentView((int) R.layout.activity_login);
    this.context = getApplicationContext();
    this.txtPassword = (EditText) findViewById(R.id.editText);
    this.prefs = getSharedPreferences("JP_prefs", 0);
    SharedPreferences.Editor edit = this.prefs.edit();
    edit.putString("hammond", "Welcome to Jurassic Park!");
    edit.commit();
}
```

Continuando nuestra exploración de los recursos, vemos que en el archivo de strings encontramos una entrada de nombre “last_step” (en español: último paso) y valor “Just go native!”: un posible indicador de que al final de desafío quizás tengamos que lidiar con librerías nativas. Además, vemos algunos hashes.



Si los ingresamos en algún sitio como CrackStation para obtener el texto original, la plataforma nos arrojará las siguientes coincidencias:

Hash	Type	Result
8e7fc0236af43df9340685fc16f1efe36543cc1707051220a103ad99cf69a2df	sha256	that
fa51fd49abf67705d6a35d18218c115ff5633aec1f9ebfdc9d5d4956416f57f6	sha256	is
7692c3ad3540bb803c020b3aee66cd8887123234ea0c6e7143c0add73ff431ed	sha256	one
2a21fe6d592a19b7de898b50eb53c429608de1a66f3e9f62da19714a770553d1	sha256	big
2eb8272d48d13634b6d78e034c4ae95c6e11618e52cba66dfec3a188dd3b1bdc	sha256	pile
28391d3bc64ec15cbb090426b04aa6b7649c3cc85f11230bb0105e02d15e3624	sha256	of
3ae80d6f98a96fd3d4dd28504d3c3f36214f1fe1d4ee8a2e724212c78027f80e	sha256	sh*t

Mientras analizamos las clases del APK, otra cosa que notamos es la clase **d**, la cual contiene algunas strings cifradas.

```
public class d {
    public static String a = "12321322869";
    public static final String b = e.pqz("htDaDYyZOg==");
    public static final String c = e.pqz("tNrV");
    public static final String h = e.pqz("5dnREIScO57b4GAj");
    public static final String i = e.pqz("5fvTCo+RKq7apg==");
    public static final String k = e.pqz("5dnbEYaCN6KK");
    public static final String l = e.pqz("5cvQFI2fKa+K");
    public static final String m = e.pqz("5f/QG5GfN6WI109KNHoHGcNbxcOzZ57tl8xC");
    public static final String n = e.pqz("5fnbEZqdMbXB62Uj");
    public static final String z = e.pqz("hMzfHIi9Ow==");
}
```

El método **pqz** de la clase **e** llama a su vez a los métodos de la clase **a**. Para descifrar estos textos podemos crear un proyecto auxiliar y ejecutar los métodos de descifrado de **a** que se encuentran en el código, o instrumentar el ejecutable luego durante el análisis dinámico con herramientas como Frida. Si hacemos lo primero, obtendremos los textos planos y sabremos que se trata de un proceso de antiemulación que intenta detectar un emulador comparando algunas propiedades del sistema contra las strings ofuscadas de la clase **d**.

```
Android
sdk
google_sdk
Emulator
generic
unknown
Android SDK built for x86
Genymotion
CrackMe
```

Esto nos demuestra que para correr el APK deberemos utilizar un módulo anti-antiemulación o editar el código del APK para bypassear la validación. Pero primero, sigamos analizando el código a ver qué más encontramos.

Un método interesante es **decrypt**, que es llamado desde una vista. Si observamos los *layouts* en los recursos, encontraremos que se ejecuta cuando se presiona el botón "*Start!*" en **activity_login.xml**, asociado a la clase **g**. Este método toma una contraseña ingresada por el usuario, verifica que coincida con un determinado patrón, que su SHA-256 sea "6fe0ae2d9513f56083806f6d362d64995a4bce33f092a348cd4f39f77ca6dbb2", y llama al método chk encargado de la anti-emulación que vimos previamente.

```
public void decrypt(View view) {
    try {
        String trim = this.txtPassword.getText().toString().toLowerCase().trim();
        if (!this.PASS_PATTERN.matcher(trim).matches() || !checkPassword(trim) || f.chk(this.context)) {
            Toast.makeText(getApplicationContext(), R.string.wrong_password, 1).show();
            startActivity(new Intent("android.intent.action.VIEW", Uri.parse("https://www.youtube.com/watch?v=K3PrSj9XEu4")));
            return;
        }
        Toast.makeText(getApplicationContext(), R.string.correct, 1).show();
        new LongOperation().execute(new String[]{trim});
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

El patrón que debe cumplir la contraseña es que debe contener seis caracteres y solo contener letras de la h a la z, y dígitos del 0 al 6.

```
public class g extends AppCompatActivity {
    private final Pattern PASS_PATTERN = Pattern.compile("^([h-z0-6]){6}$");
    private Context context;
```

Si no logramos romper el hash en algún sitio como CrackStation, deberemos utilizar fuerza bruta para encontrar la contraseña correcta. Para ello, puedes crear un script como el siguiente:

```
1  import string
2  import hashlib
3  import time
4  import itertools
5
6  start_time = time.time()
7
8  result = itertools.product("0123456hijklmnopqrstuvwxyz", repeat=6)
9  while True:
10     key = ''.join(result.next())
11     m = hashlib.sha256()
12     m.update(key)
13     if m.hexdigest() == '6fe0ae2d9513f56083806f6d362d64995a4bce33f092a348cd4f39f77ca6dbb2':
14         print key
15         break
16
17 elapsed_time = time.time() - start_time
18 print time.strftime("%H:%M:%S", time.gmtime(elapsed_time))
```

Tras dejarlo ejecutando por algún tiempo encontraremos la clave que necesitaremos: "pu3pu3".

Si todo lo anterior no se cumple, el ejecutable abrirá un navegador con el video que mencionamos al principio, nuevamente indicando que algo ha salido mal. Por el contrario, si todas las validaciones son correctas, se utilizará la clave ingresada para descifrar una string pasada por parámetro al método **decryptSecret**, y luego el texto plano resultante se utilizará en el método **chkwb**.

```
@SuppressLint({"StaticFieldLeak"})
private class LongOperation extends AsyncTask<String, Void, String> {
    private LongOperation() {
    }

    /* access modifiers changed from: protected */
    public String doInBackground(String... strArr) {
        try {
            c.chkwb(g.this, b.decryptSecret("RU0ROfKoQJk=@~@~@GdZjB0PyFOL4X7ANOS0evA==@~@~@P1soRudy7MU4znlntDnO/Q==", strArr[0]));
            return "Executed";
        } catch (Exception e) {
            e.printStackTrace();
            return "Executed";
        }
    }
}
```

En primer lugar, el método **chkwb** obtiene el valor almacenado bajo la clave “hammond” en el archivo de preferencias “JP_prefs”. El método **foo** por su parte toma una string en base64 y la decodifica 10 veces, devolviendo finalmente “-”. Este carácter es utilizado por **chkwb** para separar el texto obtenido del archivo de preferencias, y luego compara el hash de cada segmento con los hashes que vimos previamente almacenados en el archivo **strings.xml**.

Esto significa que es necesario que JP_prefs incluya la icónica cita “that-is-one-big-pile-of-sh*t” en vez de “Welcome to Jurassic Park!” para la clave “hammond” al momento de ejecutarse **chkwb**.

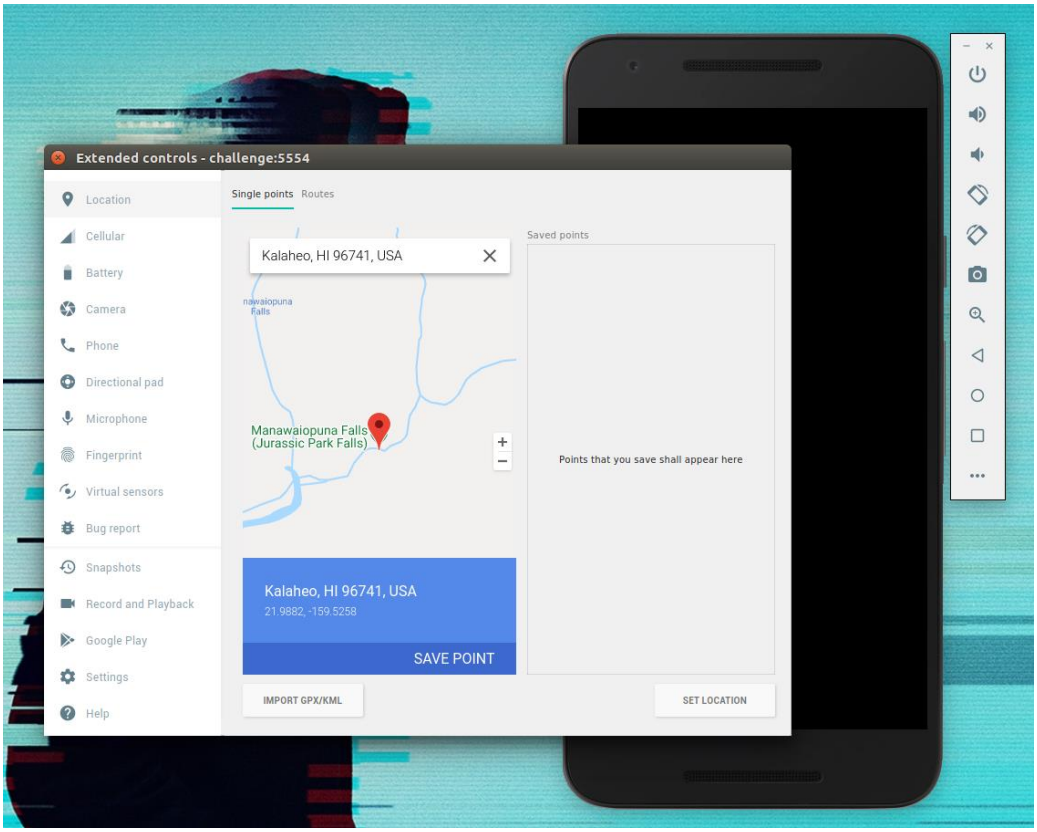
```
public static String chkwb(Context context, String str) {
    try {
        String string = context.getSharedPreferences("JP_prefs", 0).getString("hammond", "");
        String[] split = string.split(foo());
        if (!Login.areWeLanding()
            || !ie(dh("SHA-256", split[0]), context.getString(R.string.m0))
            || !ie(dh("SHA-256", split[1]), context.getString(R.string.m1))
            || !ie(dh("SHA-256", split[2]), context.getString(R.string.m2))
            || !ie(dh("SHA-256", split[3]), context.getString(R.string.m3))
            || !ie(dh("SHA-256", split[4]), context.getString(R.string.m4))
            || !ie(dh("SHA-256", split[5]), context.getString(R.string.m5))
            || !ie(dh("SHA-256", split[6]), context.getString(R.string.m6))) {
            return "";
        }
        blue(str + string);
        return "";
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
```

```
private static String foo() {
    String str = "Vm0wd2QyVkZNVWRYV0docFVtMVNWVmx0ZEhkVlZscDBUVlpPVmsxWGVIbFdiVFZyVm0xS1IyTkliRmRXTTFKTVZsVmFWMVpWTVVWaGVqQTk=";
    int i = 0;
    while (i < 10) {
        i++;
        str = new String(Base64.decode(str, 0));
    }
    return str;
}
```

Además, el método **Login.areWeLanding** valida que la ubicación del emulador sea las cataratas Manawaiopuna, que sirvieron como lugar de aterrizaje al Parque Jurásico. Otro chequeo que deberemos evadir ya sea modificando el ejecutable…

```
public static boolean areWeLanding() {
    return location != null && location.getLatitude() == 21.9882298d && location.getLongitude() == -159.5258259d;
}
```

… o cambiando la ubicación GPS en el emulador.



Si todos los hashes coinciden y la ubicación es la correcta, "that-is-one-big-pile-of-sh*t" se concatena con el texto descifrado pasado por parámetro con la clave ingresada por el usuario, y luego se envía al método **blue** de la librería **native-lib**.

```
public class c {
    public static native boolean blue(String str);

    static {
        System.loadLibrary("native-lib");
    }
```

Llegado este punto y habiendo comprendido la lógica de la app, podemos pasar a un análisis dinámico. Como mencionamos, deberemos modificar el APK para evadir algunas protecciones. Para hacerlo, editaremos el código smali con apktool y jarsigner. Primero, descompilamos el APK:

```
java -jar apktool.jar d -f -o <ruta-carpeta-output> <ruta-apk-original>
```

Ahora, realizamos los siguientes cambios:

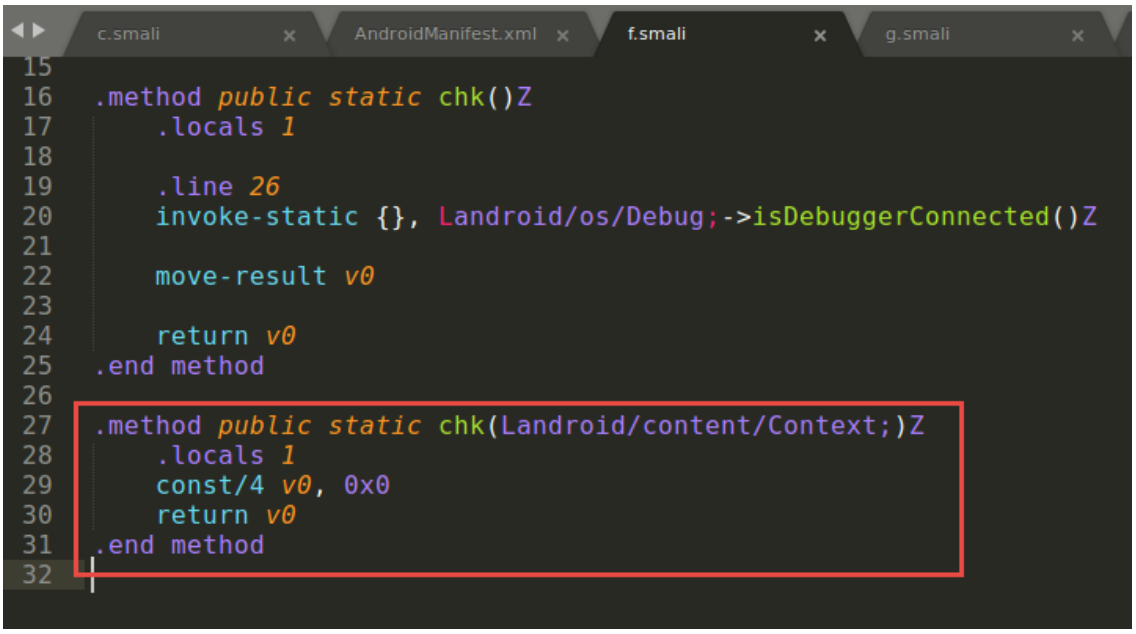
- Modificar la actividad que se llama al iniciar la aplicación. Simplemente intercambiamos los nombres de las clases **g** y **Login** en el AndroidManifest.xml. El código final debiese lucir así:

```
c.smali | AndroidManifest.xml | f.smali | g.smali
6     <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
7     <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
8     <application android:allowBackup="true" android:icon="@mipmap/ic_launcher" android:label="@string/
      app_name" android:roundIcon="@mipmap/ic_launcher_round" android:supportsRtl="true" android:theme="
      @style/AppTheme">
9         <activity android:exported="true" android:name="eset.mobile.retowls.Login"/>
10        <activity android:name="eset.mobile.retowls.g">
11            <intent-filter>
12                <action android:name="android.intent.action.MAIN"/>
13                <category android:name="android.intent.category.LAUNCHER"/>
14            </intent-filter>
15        </activity>
16        <meta-data android:name="android.support.VERSION" android:value="26.1.0"/>
17        <meta-data android:name="android.arch.lifecycle.VERSION" android:value="27.0.0-SNAPSHOT"/>
18    </application>
19</manifest>
```

La alternativa a esto es enviar un intent vía adb para llamar la actividad durante la ejecución.

```
adb shell am -n eset.mobile.retowls/.g
```

- Quitar el chequeo anti-emulación del método **chk**. Para ello haremos que el método siempre devuelva 0, es decir, false.



- Modificar en la clase **g** el texto por defecto que se guarda en las preferencias compartidas para que coincida con lo requerido por el método **chkwb**.

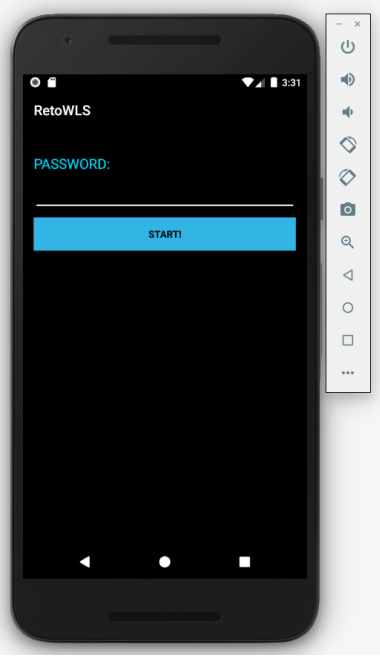
- Modificar la clase c para que el chequeo de la ubicación GPS siempre sea verdadero.

```
c.smali   AndroidManifest.xml   f.smali   g.smali
52    invoke-static {}, Leset/mobile/retowls/c;->foo()Ljava/lang/String;
53
54    move-result-object v2
55
56    invoke-virtual {v0, v2}, Ljava/lang/String;->split(Ljava/lang/String;)[Ljava/lang/String;
57
58    move-result-object v2
59
60    .line 25
61    invoke-static {}, Leset/mobile/retowls/Login;->areWeLanding()Z
62
63    const/4 v3, 0x1
64
65    if-eqz v3, :cond_0
66
67    const-string v3, "SHA-256"
68
```

Finalmente, recompilamos la aplicación, la firmamos, e instalamos en el emulador.

```
java -jar apktool.jar b -o <ruta-apk-recompilado> <ruta-carpeta-output>
jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore <ruta-keystore>
<ruta-apk-recompilado> <alias-name>
adb install <ruta-apk-recompilado>
```

Ahora corremos la aplicación en algún emulador como Genymotion o el Android emulator. Veremos una pantalla que nos pide una contraseña. Ingresamos la que encontramos previamente con fuerza bruta: “pu3pu3”.



Una buena práctica al momento de analizar aplicaciones es chequear los logs que generan. Si hacemos esto filtrando por el tag “AHAHAH” que aparecía en el código del APK, veremos que tras ingresar la clave correcta nos aparece un mensaje:

```
          ~/Downloads/reto/3_APK$ adb logcat -s "AHAHAH"
--------- beginning of system
--------- beginning of main
01-27 03:45:59.881  7575  7575 I AHAHAH  : https://www.youtube.com/watch?v=K3PrSj9XEu4
--------- beginning of crash
01-27 03:51:40.343  8158  8158 I AHAHAH  : https://www.youtube.com/watch?v=K3PrSj9XEu4
01-27 04:17:43.548  9012  9039 I AHAHAH  : Good! The flag is waiting for you!
```

Sabiendo que hay una librería native-lib de la cual se llama el método blue, tenemos dos opciones: la primera es dumpear la memoria del emulador con alguna herramienta como fridump para ver si obtenemos algo interesante (*spoiler alert*, fridump nos traerá la bandera que ha quedado guardada en memoria y la que podemos buscar rápidamente porque sabemos que todos los flags comienzan con "3s3T_PUE2o2o_"). La segunda opción es descargar las librerías que pueden encontrarse en la carpeta lib de los recursos del APK y analizarlas con alguna herramienta como radare2.

Al hacer lo segundo, vemos que la función blue toma la cadena pasada por parámetro "Malcolm:that-is-one-big-pile-of-sh*t" y la utiliza como clave para descifrar mediante operaciones XOR un texto almacenado estáticamente en forma de array de *integers* dentro de esta función.

Así, encontramos la bandera: **3s3T_PUE2o2o_4ndr01dRulz!**.

4_ElTesoroEscondido.zip

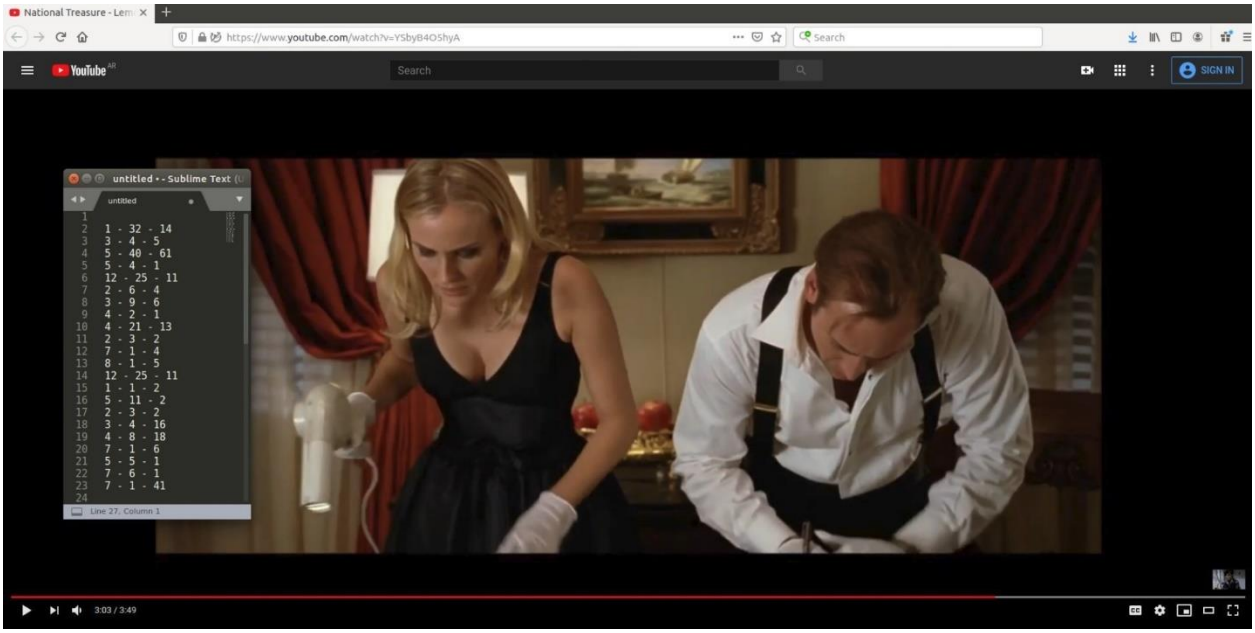
El desafío comienza con un ZIP de nombre 4_ElTesoroEscondido.zip. Al descomprimirlo, nos encontramos con tres imágenes: clues.jpg, givemebooks.jpg y hiddeninplainsight.png. Comenzaremos analizando el primer archivo ya que, por su nombre, podría indicar que contiene pistas importantes para la resolución del desafío.

clues.jpg

givemebooks.jpg

hiddeninplainsight.png

Al abrir la imagen, vemos una captura de pantalla de un navegador que se encuentra reproduciendo un vídeo de YouTube, en el cual se observa una escena de la película La leyenda del tesoro perdido. Además, vemos un conjunto de códigos escritos en un editor de texto. Si buscamos este video y nos dirigimos al minuto de reproducción que figura en la captura, descubriremos que el texto en el editor corresponde a un cifrado Ottendorf, donde en cada línea se aprecia el número de página, línea y letra de un texto determinado. El desafío consistirá ahora en identificar qué texto.



Cuando estamos resolviendo un desafío de esteganografía, una buena estrategia quizás sea chequear los metadatos del archivo, y evaluar si hay algún archivo embebido. Para conseguir lo primero, utilizaremos la herramienta exiftool sobre el archivo. Veremos que este archivo nos indica en sus comentarios que esconde otra pista además del código Ottendorf que acabamos de encontrar a simple vista.

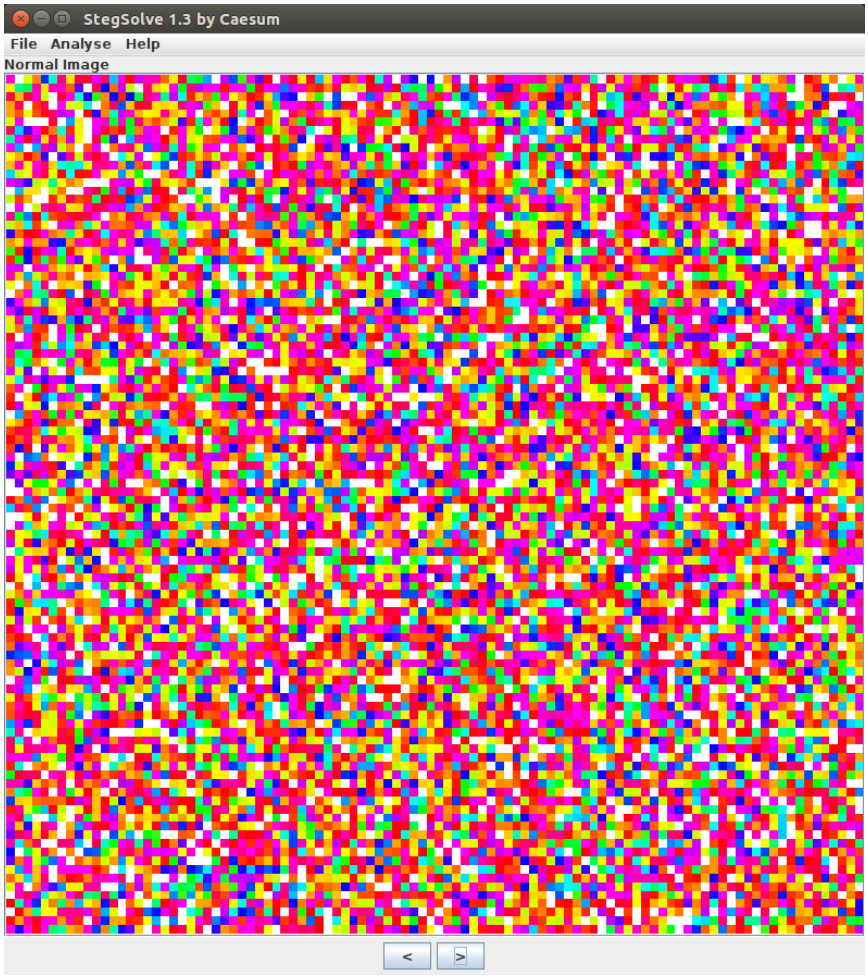
```
[redacted]/reto/4_ElTesoroEscondido$ exiftool clues.jpg
ExifTool Version Number         : 10.10
File Name                       : clues.jpg
Directory                       : .
File Size                       : 215 kB
File Modification Date/Time     : 2020:11:21 13:38:33-03:00
File Access Date/Time           : 2021:01:22 19:01:23-03:00
File Inode Change Date/Time     : 2020:11:21 13:38:33-03:00
File Permissions                : rw-rw-r--
File Type                       : JPEG
File Type Extension             : jpg
MIME Type                       : image/jpeg
JFIF Version                    : 1.01
Exif Byte Order                 : Big-endian (Motorola, MM)
Orientation                     : Horizontal (normal)
X Resolution                    : 1
Y Resolution                    : 1
Resolution Unit                 : None
Y Cb Cr Positioning             : Centered
Compression                     : JPEG (old-style)
Thumbnail Offset                : 214
Thumbnail Length                : 33595
Comment                         : Escondo otra pista en mis metadatos
Image Width                     : 1843
Image Height                    : 909
Encoding Process                : Baseline DCT, Huffman coding
Bits Per Sample                 : 8
Color Components                : 3
Y Cb Cr Sub Sampling            : YCbCr4:2:0 (2 2)
Image Size                      : 1843x909
Megapixels                      : 1.7
Thumbnail Image                 : (Binary data 33595 bytes, use -b option to extract)
```

Ahora utilizaremos binwalk para intentar identificar otros archivos embebidos en clues.jpg. En este caso, veremos que hay una imagen escondida dentro del archivo inicial. Utilizaremos la misma herramienta para extraerla.

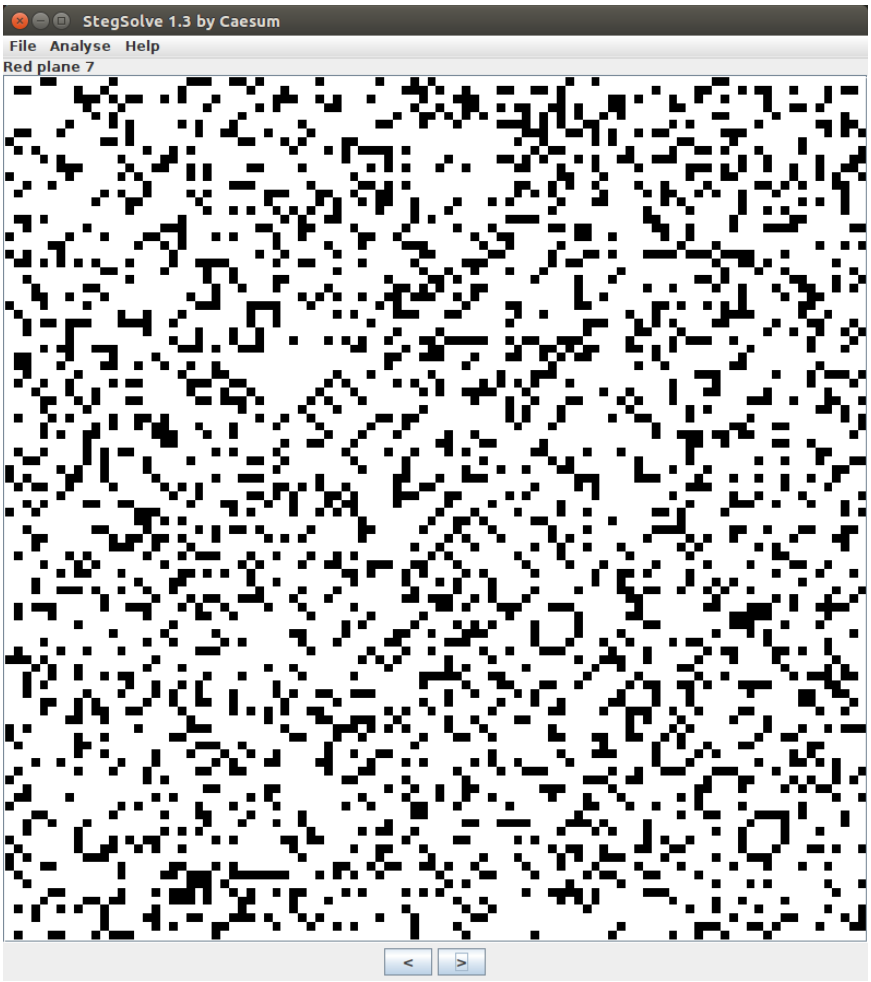
```
[redacted]/reto/4_ElTesoroEscondido$ binwalk clues.jpg

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             JPEG image data, JFIF standard 1.01
30            0x1E            TIFF image data, big-endian, offset of first image directory: 8
214           0xD6            JPEG image data, JFIF standard 1.00
244           0xF4            TIFF image data, big-endian, offset of first image directory: 8
4749          0x128D          Unix path: /www.w3.org/1999/02/22-rdf-syntax-ns#"><rdf:Description rdf:about="uuid:fa
f5bdd5-ba3d-11da-ad31-d33d75182f1b" xmlns:xmp="http://

[redacted]/reto/4_ElTesoroEscondido$ binwalk -dd='.*' clues.jpg
[redacted]/reto/4_ElTesoroEscondido$
```

Así encontraremos la siguiente imagen, la cual nos indica que al contar las líneas del texto debemos incluir el título, y que al contar los caracteres debemos incluir espacios.

Ahora pasaremos a analizar el archivo hiddeninplainsight.png. Si tienes algunos anteojos 3D al alcance, resolver este desafío puede ser tan fácil como colocártelos y leer el texto en la imagen. Para quienes no disponemos de tan conveniente objeto, deberemos utilizar algún programa de edición de imágenes para editar sus canales RGBA (en español: rojo verde azul alfa) hasta lograr que el texto escondido sea visible. También podemos utilizar StegSolve con este propósito: apenas carguemos la imagen en la herramienta podremos usar las flechas en la parte inferior para visualizar diferentes paneles RGBA.


StegSolve 1.3 by Caesum
File Analyse Help
Normal Image
<
>

Así daremos con un panel donde las letras son visibles y revelan el texto “PASS ELVIEJOTRUCO”. Aunque aún no requerimos una contraseña, lo tendremos en cuenta al momento de analizar el próximo archivo.



¿Recuerdas que necesitábamos encontrar el texto sobre el cual aplicar el cifrado Ottendorf? Resulta curioso que el último archivo lleve el nombre givemebooks.jpg. Analizando sus metadatos encontramos una pista: nos indica que el propósito del análisis es encontrar el nombre de un PDF, y luego deberemos buscar este nombre en algún motor de búsqueda.

```
                    /reto/4_ElTesoroEscondido$ exiftool givemebooks.jpg
ExifTool Version Number         : 10.10
File Name                       : givemebooks.jpg
Directory                       : .
File Size                       : 5.1 MB
File Modification Date/Time     : 2020:11:21 12:09:36-03:00
File Access Date/Time           : 2021:01:22 19:06:32-03:00
File Inode Change Date/Time     : 2020:11:21 12:45:08-03:00
File Permissions                : rw-rw-r--
File Type                       : JPEG
File Type Extension             : jpg
MIME Type                       : image/jpeg
JFIF Version                    : 1.01
Resolution Unit                 : inches
X Resolution                    : 72
Y Resolution                    : 72
Comment                         : Encuentra el título del PDF y pregunta al sabio Google
Image Width                     : 5461
Image Height                    : 8192
Encoding Process                : Baseline DCT, Huffman coding
Bits Per Sample                 : 8
Color Components                : 3
Y Cb Cr Sub Sampling            : YCbCr4:2:0 (2 2)
Image Size                      : 5461x8192
Megapixels                      : 44.7
```

Ahora nos fijaremos si existe algún archivo embebido con steghide y nos daremos con que nos requiere una contraseña.

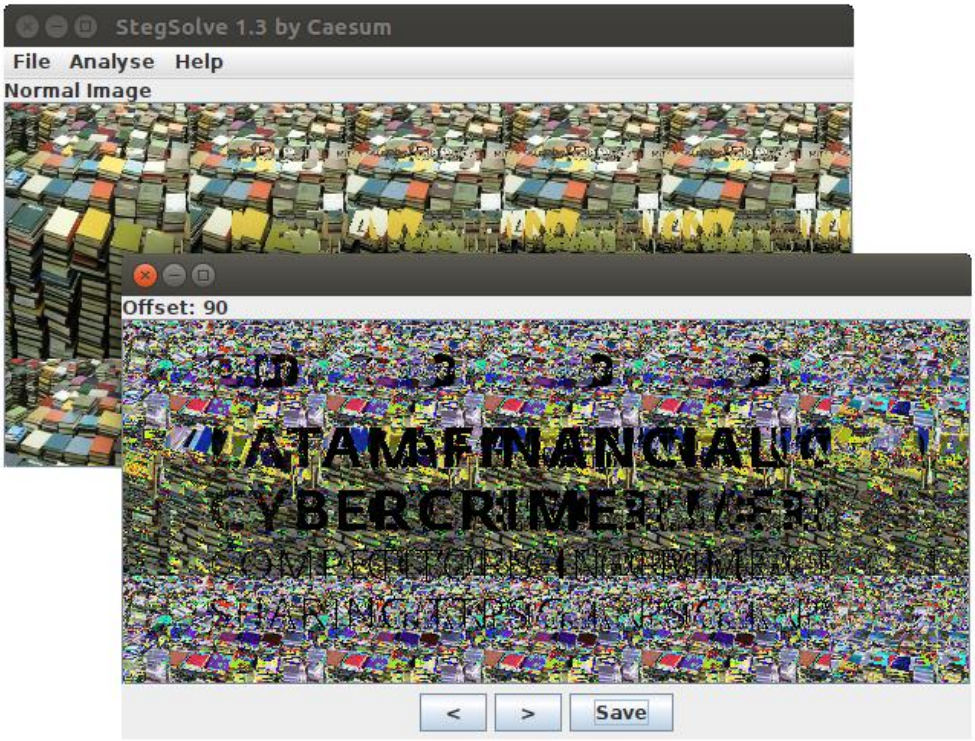
```
                    /reto/4_ElTesoroEscondido$ steghide info givemebooks.jpg
"givemebooks.jpg":
  format: jpeg
  capacity: 302,2 KB
Try to get information about embedded data ? (y/n) y
Enter passphrase:
steghide: could not extract any data with that passphrase!
```

¡Qué bueno que aún no hemos utilizado la contraseña que encontramos en la imagen anterior! Utilizaremos ELVIEJOTRUCO para intentar extraer los archivos embebidos… *et voilà.* Obtendremos el archivo hidden.png — o como sea que hayas decidido llamarlo —.

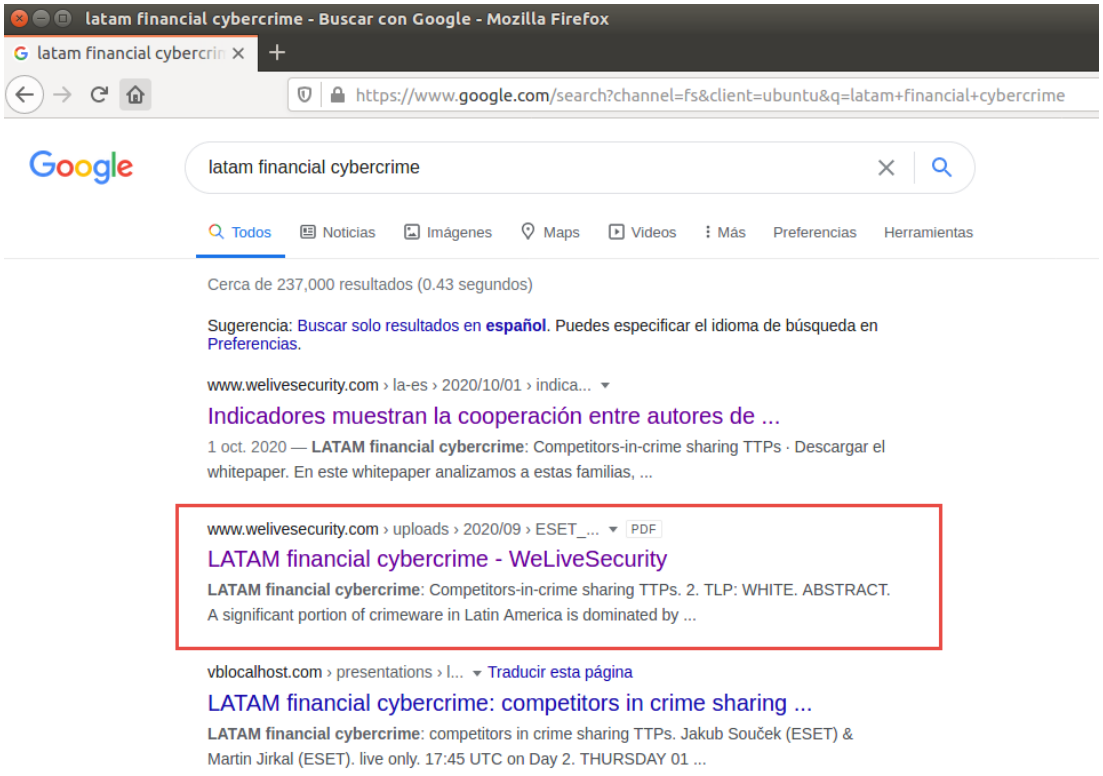
```
                    /reto/4_ElTesoroEscondido$ steghide extract -sf givemebooks.jpg -p ELVIEJOTRUCO
wrote extracted data to "hidden.png".
```

Abrimos esta imagen nuevamente con StegSolve y veremos que presenta algunas distorsiones. También vemos que tenemos un patrón de libros que se ha utilizado para cubrir alguna otra imagen más grande. Si has escuchado hablar de los estereogramas, quizás esto te haya hecho recordarlos. Se trata de una técnica que puede ser utilizada en la estenografía para esconder una imagen dentro de otra.

Muchas personas pueden resolver estereogramas simplemente usando su visión, pero quienes no tenemos ojos tan ávidos debemos recurrir a herramientas informáticas. En este caso, podemos nuevamente usar StegSolve. Simplemente abrimos la imagen y nos dirigimos a Analyze > Stereogram Solver. Allí podremos jugar con el offset hasta dar con el valor correcto. A continuación, vemos que aplicando con un offset igual a 90 se leen claramente las palabras “LATAM FINANCIAL CYBERCRIME”. También es posible identificar el logo de ESET.



Lo anterior nos indica que necesitamos encontrar un documento de ESET que lleve ese título. Si recordamos la pista anterior, podemos buscarlo en Google y rápidamente encontraremos la respuesta.



Ahora solo falta buscar cada página, línea y carácter en el PDF.

3	1 - 32 - 14
s	3 - 4 - 5
3	5 - 40 - 61

T	5 - 4 - 1
_	12 - 25 - 11
P	2 - 6 - 4
U	3 - 9 - 6
E	4 - 2 - 1
2	4 - 21 - 13
o	2 - 3 - 2
2	7 - 1 - 4
o	8 - 1 - 5
_	12 - 25 - 11
B	1 - 1 - 2
0	5 - 11 - 2
o	2 - 3 - 2
k	3 - 4 - 16
W	4 - 8 - 18
0	7 - 1 - 6
r	5 - 5 - 1
M	7 - 6 - 1
z	7 - 1 - 41

Finalmente, damos con la bandera: **3s3T_PUE2o2o_B0okW0rMz**.

5_RSA.zip

La explicación resumida del desafío puede concentrarse en los siguientes conceptos:

- Puede parecer bastante complicado a primera vista, pero hay desafíos parecidos que los deberían orientar bien. Aunque no es igual y seguramente van a tener que pensar un poco.
- Las claves que se dan son vulnerables porque los módulos públicos `n` no son coprimos. En base a eso, se puede calcular el Máximo Común Divisor de ambas claves y averiguar el valor de los dos factores primos individuales.
- Con esos dos valores es simple calcular el exponente privado `d`, reconstruir la clave privada y descifrar el mensaje.

Para resolver el desafío se brindan dos claves públicas y un mensaje cifrado con una de ellas. Para que sea posible resolverlo, es necesario extraer una clave privada solo con esta información. Esto nos da una idea de que las claves públicas tienen algún tipo de vulnerabilidad que nos va a permitir obtenerla.
En este caso, los pares de claves fueron creados con un generador de baja entropía que permite que se repitan valores de los números primos que conforman al módulo público **n**.

$$n = p * q$$

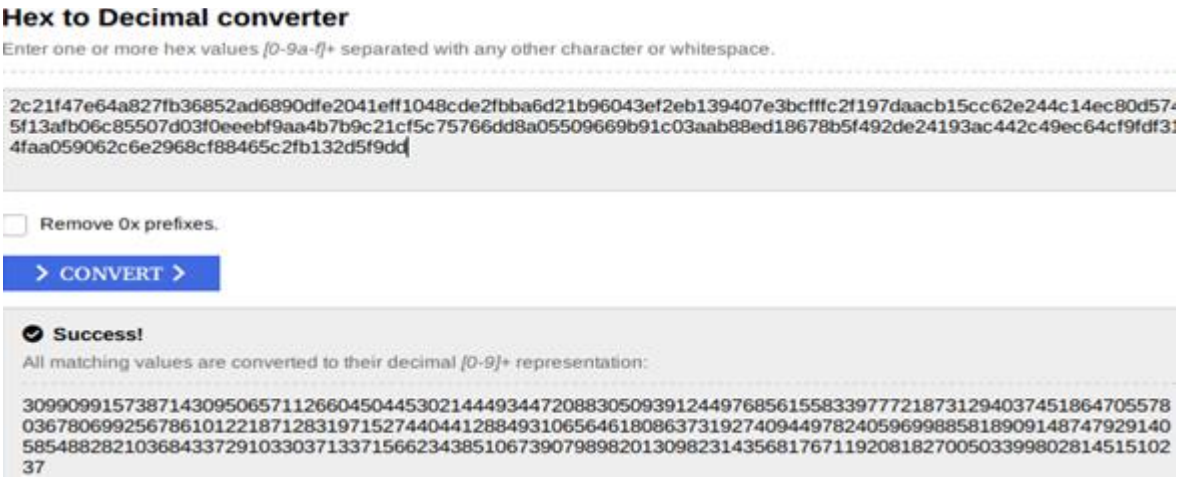
Típicamente si los valores de `p` y `q` son realmente aleatorios y lo suficientemente grandes, es prácticamente imposible determinar sus valores solo conociendo `n`. Pero si se tienen al menos dos valores de `n` y se determina que poseen un Máximo Común Denominador (MCD), entonces resulta fácil calcular los factores primos de cada clave.

$$\mathrm{MCD}(a,b) = \frac{a \cdot b}{\mathrm{mcm}(a,b)}$$

Sabiendo que los factores son primos, cualquier MCD que calculemos que sea > 1 es un factor de `n`, y calcular el segundo es trivial ya que solo se necesita realizar una división.
El primer paso para resolver este problema es calcular los valores de **n** y **e** de cada clave pública. OpenSSL nos permite extraer esta información con un único comando:

```
┌─[   @parrot]─[~/Desktop/
└──╼ $var=$(openssl rsa -pubin -in ./1.pem -text -noout)
┌─[   @parrot]─[~/Desktop/
└──╼ $echo "${var//:}"
RSA Public-Key (1022 bit)
Modulus
    2c21f47e64a827fb36852ad6890dfe
    2041eff1048cde2fbba6d21b96043e
    f2eb139407e3bcfffc2f197daacb15
    cc62e244c14ec80d574b5f13afb06c
    85507d03f0eeebf9aa4b7b9c21cf5c
    75766dd8a05509669b91c03aab88ed
    18678b5f492de24193ac442c49ec64
    cf9fdf3184faa059062c6e2968cf88
    465c2fb132d5f9dd
Exponent 65537 (0x10001)
```

Ahora, debemos convertir el módulo `n` a su representación decimal:



Una vez que tenemos el valor decimal de `n` para cada clave, podemos comprobar si existe un MCM. Utilizamos el algoritmo de Euclides (estándar) para realizar este cálculo, es posible encontrar en internet el código para implementar este algoritmo en cualquier lenguaje de programación.
Efectivamente, obtenemos un resultado mayor a 1.

```
$python3 ./gcd.py
3561108085236297067026521587296526080361258627565403384371877202312009259338898543467535604822587645465346416939524827012380101592534748338198582227910059
```

El número que obtuvimos lo podemos usar como un factor, supongamos p, en cualquiera de las dos claves públicas. El segundo factor es diferente en cada caso y se calcula simplemente como:

q = n / p

Si bien se está explicando cómo hacer este proceso de forma manual, existen herramientas que permiten encontrar el MCM y derivar los valores de `p` y `q` de forma automatizada.
Una vez obtenidos los valores individuales de `p` y `q`, podemos calcular el exponente privado `d` de cada clave. Esto es posible ya que el valor de `e` y el valor de `φ(n)=(p-1)*(q-1)` son coprimos. Podemos utilizar nuevamente el algoritmo de Euclides y calcular `d` como el inverso multiplicativo de `e mod φ(n)`.

```python
# Las siguientes dos funciones se utilizan para calcular el inverso multiplicativo
def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        g, y, x = egcd (b % a, a)
        return (g, x - (b // a) * y, y)

def modinv(a, m):
    g, x, y = egcd(a, m)
    if g != 1:
        raise Exception('el inverso multiplicativo no existe')
    else:
        return x % m

# componentes del certificado publico extraido
n = 135358467027268608298936489867618642128244107420143096038413512213826721562358088331216273626746436436147343351022879143394846467678109538153729727714231993451509634
p = 3561108085236297067026521587296526080361258627565403384371877202312009259338898543467535604822587645465346416939524827012380101592534748338198582227910059
q = 3801021024563676142978712270262143790076542495879408603230054582946471282308420572309325541042480733314219879402357000835483657286134810787413815280950031
e = 65537

# Calculo de el exponente privado d
d = modinv(e, n - (p + q - 1))
```

Una vez obtenido el valor de `d`, podemos crear la clave privada. En este caso utilizando la librería pycrypto de Python. Finalmente, utilizamos la clave privada para descifrar el mensaje y resolver el desafío:

```
# Utilizamos los valores computados para crear una clave privada
def createPrivateKey(n, e, d, p, q):
    privateKey = RSA.construct((n, e, d, p, q))
    print ("\n\nClave privada exportada: ", privateKey.exportKey())
    return privateKey

def democtf():
    msg = b'AeY+0M21RWCF3o2d+dRvoYmpf55H/JEgAOjZcAxUIKQTmIw5Oxk+EogoonoM5yDT8Ww+bvGrIDU+3SO+dAGBtWLud
    privateKey = PKCS1_OAEP.new(createPrivateKey(n, e, d, p, q))
    decrypt = privateKey.decrypt(b64decode(msg))
    print ("\n\nMensaje descifrado: ",decrypt)

democtf()
```

```
→ $python3 ./rsa_challenge.py


Clave privada exportada:  b'-----BEGIN RSA PRIVATE KEY-----\nMIICWAIBAAKBgBNGk1PL2D7edM5pEoro58piwdnME76iz6r31if1f/0hYBqMkRpR\ntn1Fgn2uj+mPGWIyXLPvVI7FWXZt
GRC4Eea4g8l1qB+0L6Bl8DY6DgZ5P/+epcoe\nttQMUDiBsWGRPxUbrHyuZes1zf/lxVlHHW369las7ipeP57QlEB8lWYFAgMBAAEC\ngYAO+YtHVFvhK62PFe4D4MECaAq1i2fjeFThO70yOfV+z8j7lyy
7R7QASloADfdb\nuYI82WdPmqUUbZj8+fC5Hmc5a/blERE+KDqz2T3bhYKXArP909Omh6sTC+HAYgSq\newvG/R0Iq/M1BzhnMSGC9G53hmuOienlMzq8hieHnokFGQJAQ/5Vb5yG0xV1ivcO\nKggkwRM2
ukN9r5jApHerTPKC4OOUQziq1eLiBw2Z+6pbQMml641Z7co+2y5AyvHn\n5yw9qwJASJMA9gjJDRBAaoQ9Vaa/aUyhRY81KE3MfqjLIhN4TvZMPLjv1p81Vk4q\n/2ky3sTp5Q7hEI2LXotMs2eRnzZbDwJ
AE/uipmvNYCo1YoMbgZTlIG2CLxn4zWvK\nakvVZ4b6mheokl0MMFI+9a2OrLcznIXnIqRnYrCG6qncVwjjjBh/gwJAJmVn/8M8\nI2YyvmgCTztuT5aUSEnw9MbcZzKEbh3g693RDdtGQMVwoZ2vj/unR0
bk/CP4HRrV\n3opSZSRpXdOTLwJABAzOsZU084TVHBh7BTdbyCoEQN9r0agwpqAZvfr/meXBf+R9\nW1aMMS2Cg4NyQn+DZ4u4Qq4/ZM7dNrPyzsXINA==\n-----END RSA PRIVATE KEY-----'


Mensaje descifrado:  b'3s3T_PUE2o2o_R$4ch4l13nG3'
```

La flag es **3s3T_PUE2o2o_R$4ch4l13nG3**.

Imports utilizados y recomendados para la resolución:

- Crypto.PKCS1_OAEP
- Crypto.RSA

6_CrackMe.zip

Para resolver este ejercicio primero vamos a realizar un pequeño análisis estático del archivo identificando su arquitectura y cadenas de caracteres (llamadas *strings*) que puedan llegar a ser de interés.

IMAGEN 1 – DETECCIÓN DEL TIPO DE ARCHIVO Y STRINGS

Como podemos ver es un archivo ELF y de las strings podemos algunas que podrían llegar a interesantes (remarcadas en rojo). Dadas estas strings, el archivo aparenta pedirnos una contraseña, por ende vamos a correrlo dinámicamente para ver si realmente es así o hace otra cosa.

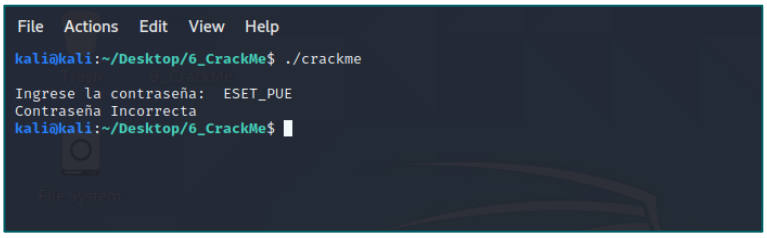


IMAGEN 2 – CORRIDA DINÁMICA

Luego de intentar con varias contraseñas, obtenemos el mismo mensaje de “Contraseña Incorrecta”, con lo cual vamos a abrir el archivo con el IDA Pro para analizarlo.

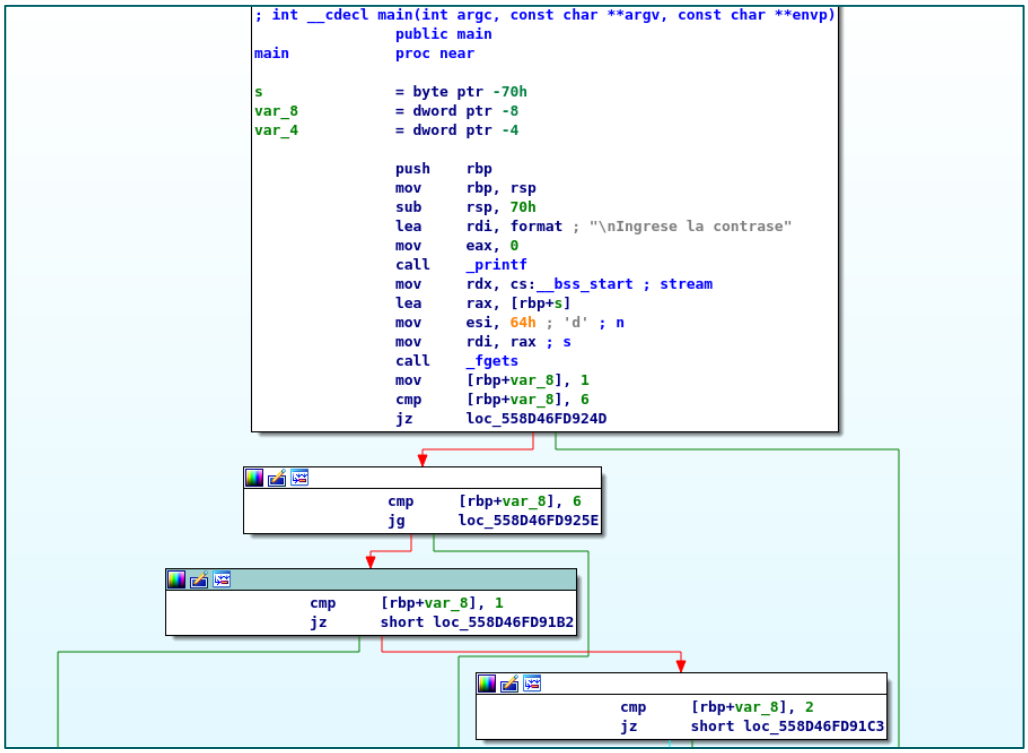


IMAGEN 3 – ARCHIVO LEVANTADO EN IDA PRO

Como se puede ver en la imagen anterior estamos parados sobre la lógica principal del programa donde tenemos algunas llamadas a funciones como **printf**, la cual se encarga de mostrar por pantalla el mensaje de "Ingrese la contraseña:", pero hay algo interesante que se puede destacar de esta lógica, en la dirección de memoria **rbp+var_8** le asigna el valor 1 y luego va comparando ese valor varias veces de la siguiente manera:

- Se fija si es igual a 6
- Se fija si es más grande que 6
- Se fija si es igual 1
- Se fija si es igual a 2

De todas estas comparaciones que hace, podemos ver que el programa termina en distintos lugares, los cuales pueden mostrarnos mensajes como "Contraseña Incorrecta", "test" o "error". Pero hay una lógica que tiene adentro un string que dice "Tu flag es: …" la cual parece ser interesante y se puede ver a continuación en la siguiente captura de pantalla.

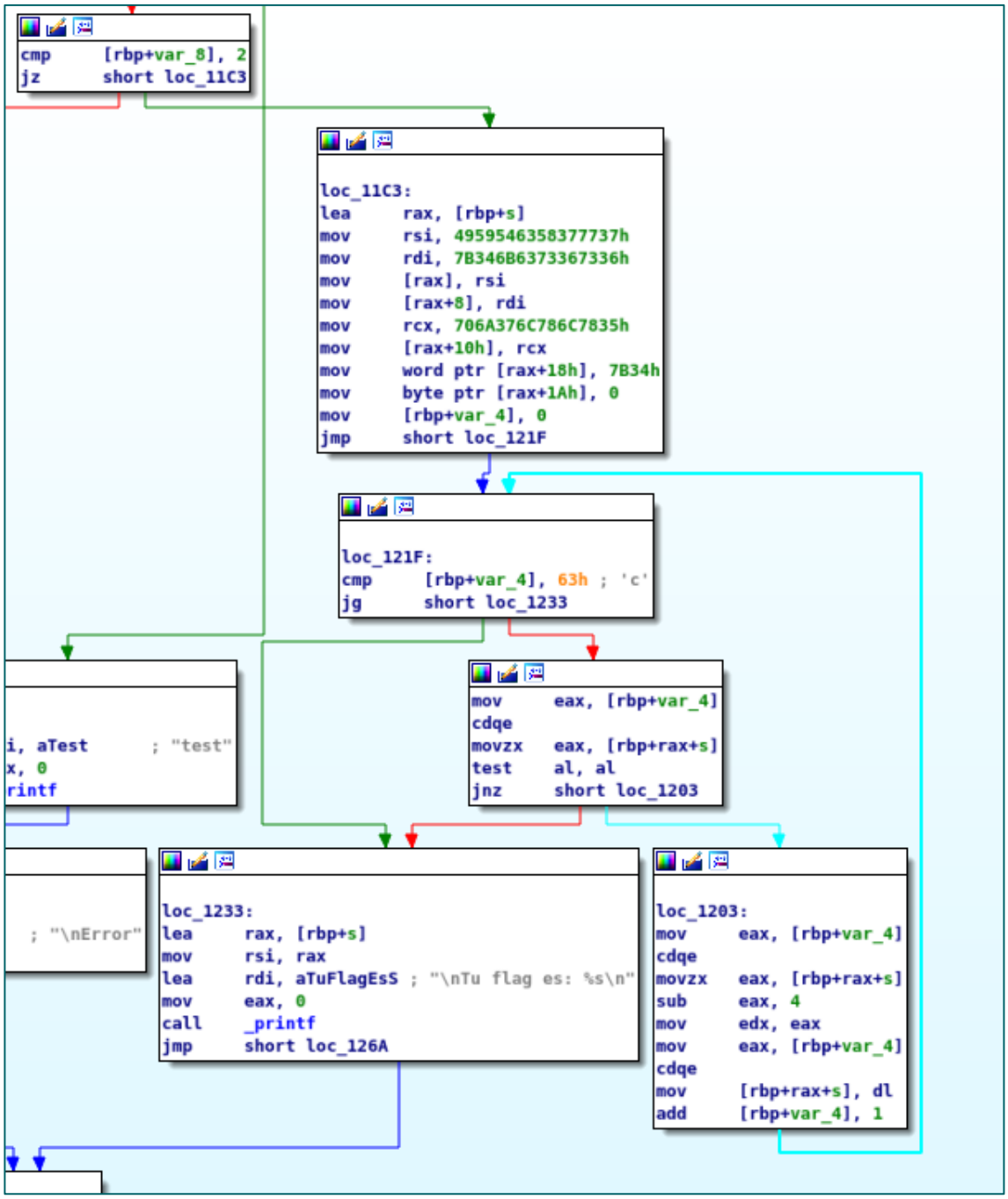


IMAGEN 4 – LÓGICA DE LA FLAG

Parece ser que esta sección de código carga algunos valores hexadecimales en memoria y luego, por medio de los registros, son modificados y vueltos a poner en memoria mediante una iteración. También vemos que tiene un contador en la dirección **rbp+var_4** y cuando llega a cierto valor, corta la iteración y muestra el mensaje "Tu flag es: %s\n" donde agarra el valor generado de la iteración anterior y lo muestra por pantalla. Entonces, sabemos que las comparaciones mencionadas en la imagen 3, no nos están permitiendo llegar a esta sección. ¿Cómo podemos resolver esto? Alterando la lógica del programa (crack).

Volvamos a revisar las comparaciones de la imagen 3.

```
:0000558D46FD917E                mov     rdi, rax        ; s
:0000558D46FD9181                call    _fgets
:0000558D46FD9186                mov     [rbp+var_8], 1
:0000558D46FD918D                cmp     [rbp+var_8], 6
:0000558D46FD9191                jz      loc_558D46FD924D
:0000558D46FD9197                cmp     [rbp+var_8], 6
:0000558D46FD919B                jg      loc_558D46FD925E
:0000558D46FD91A1                cmp     [rbp+var_8], 1
:0000558D46FD91A5                jz      short loc_558D46FD91B2
:0000558D46FD91A7                cmp     [rbp+var_8], 2
:0000558D46FD91AB                jz      short loc_558D46FD91C3
:0000558D46FD91AD                jmp     loc_558D46FD925E
:0000558D46FD91B2 ; ---------------------------------------------------------------------------
:0000558D46FD91B2
:0000558D46FD91B2 loc_558D46FD91B2:                      ; CODE XREF: main+50↑j
:0000558D46FD91B2                lea     rdi, s          ; "Contrase"
:0000558D46FD91B9                call    _puts
:0000558D46FD91BE                jmp     loc_558D46FD926A
:0000558D46FD91C3 ; ---------------------------------------------------------------------------
:0000558D46FD91C3
:0000558D46FD91C3 loc_558D46FD91C3:                      ; CODE XREF: main+56↑j
:0000558D46FD91C3                lea     rax, [rbp+s]
:0000558D46FD91C7                mov     rsi, 4959546358377737h
:0000558D46FD91D1                mov     rdi, 7B346B6373367336h
```

IMAGEN 5 – INSTRUCCIONES A MODIFICAR

Si préstamos atención a las comparaciones vemos que si las mismas dieran False seguiría a la siguiente instrucción, porque si diera True en cualquiera de las instrucciones donde vemos un **JZ** o **JG** le está indicando a que parte del código ir.

Entonces para crackear este programa vamos a modificar desde la comparación **cmp [rbp+var_8], 1** hasta **cmp [rbp+var_8], 2**, con **NOPs** que básicamente son instrucciones que no hacen nada, y también vamos a modificar la instrucción **JZ** que aparece luego de la última comparación por la instrucción **JMP**. ¿Por qué tocamos estas instrucciones y no otras? Porque sabemos que **rbp+var_8** vale 1, entonces cuando lo compara contra 1, ese resultado va a dar True y se va a ir para otra parte del código. A su vez, por el otro camino, va a comprar ese valor contra 2 y como no son iguales, sucede lo mismo se va hacia otra sección que no nos interesa llegar.

Para realizar cambios en un programa con IDA, primero tenemos que posicionarnos sobre la instrucción que queremos cambiar, luego ir a la vista Hexadecimal (Hex View-1) y los valores hexadecimales que están sombreados son los que representan la instrucción seleccionada, apretamos F2 para poder editar y reemplazamos estos valores por el valor 90 (instrucción NOP), luego de haberlos reemplazado, hay que aplicar estos cambios, para hacerlo tenemos que ir a:

- Edit -> Patch Program -> Apply patches to input file

En la siguiente imagen, se puede ver un ejemplo de cómo quedaría uno de los parches:

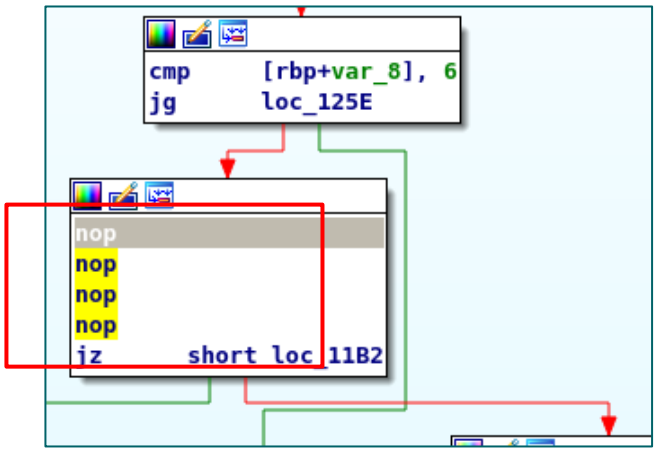


IMAGEN 6 – EJEMPLO DE UN PRIMER PARCHE

Ahora para la última instrucción que habíamos dicho, **JZ**, no la vamos a cambiar por un NOP sino que la queremos cambiar por un **JMP**. El proceso de modificación es el mismo al mencionado anteriormente, pero esta vez en vez de poner un 90 (**NOP**) en todos los valores vamos a poner el valor EB que representa la instrucción **JMP** solo en el primer valor hexadecimal de toda la instrucción seleccionada, como se puede ver en las siguientes imágenes:

IMAGEN 7 – INSTRUCCIÓN JZ EN HEXADECIMAL ANTES DE MODIFICAR

IMAGEN 8 – INSTRUCCIÓN JZ MODIFICADA POR JMP

Luego de aplicar todos los parches mencionados el programa debería habernos quedado de la siguiente manera:

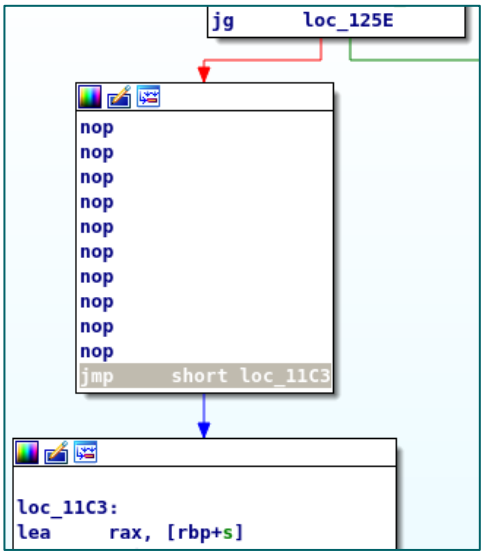


IMAGEN 9 – PARCHE FINAL

Habiendo guardado el programa con las modificaciones, lo ejecutamos y vamos a poder ver la flag:

- **3s3T_PUE2o2o_g0w1thth3fl0w**

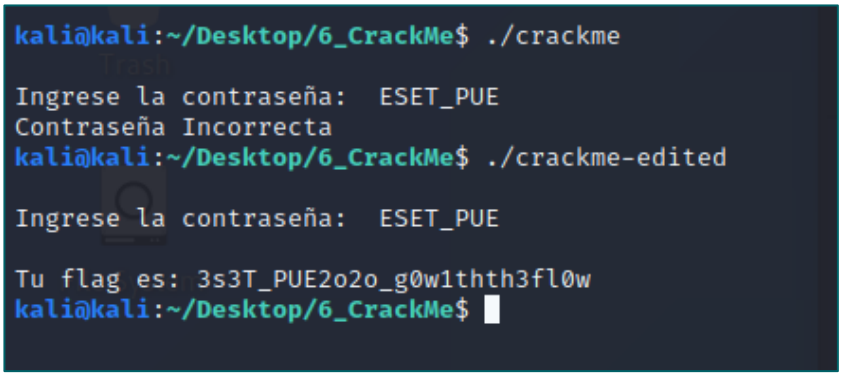


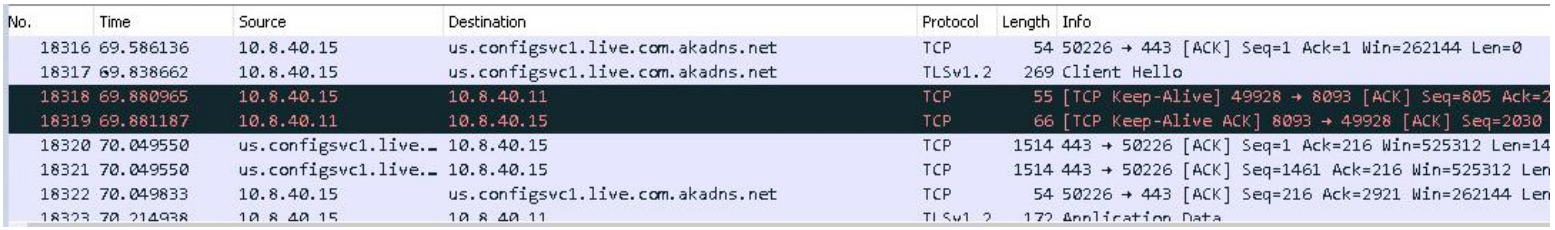
IMAGEN 10 – OBTENCIÓN DE LA FLAG

8_TráficoDeRed.zip

La consigna de este desafío giraba en torno al hipotético caso de que un usuario corporativo había descargado un documento confidencial desde un servidor web interno de la compañía. Para resolver el desafío, debías encontrar el secreto escondido en ese archivo confidencial.

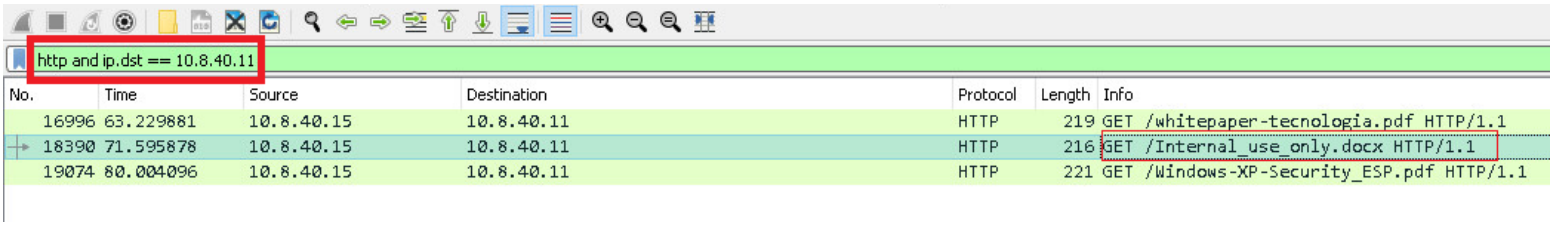
Los pasos a seguir para encontrar el flag se detallan a continuación:

1) Al abrir el archivo PCAP mediante Wireshark y sin aplicar ningún filtro vemos que prácticamente con las únicas direcciones IP **internas** que aparecen en la captura son **10.8.40.11** (servidor web) y 10.8.40.15 (el equipo del usuario)



No.	Time	Source	Destination	Protocol	Length	Info
18316	69.586136	10.8.40.15	us.configsvc1.live.com.akadns.net	TCP	54	50226 → 443 [ACK] Seq=1 Ack=1 Win=262144 Len=0
18317	69.838662	10.8.40.15	us.configsvc1.live.com.akadns.net	TLSv1.2	269	Client Hello
18318	69.880965	10.8.40.15	10.8.40.11	TCP	55	[TCP Keep-Alive] 49928 → 8093 [ACK] Seq=805 Ack=2
18319	69.881187	10.8.40.11	10.8.40.15	TCP	66	[TCP Keep-Alive ACK] 8093 → 49928 [ACK] Seq=2030
18320	70.049550	us.configsvc1.live.…	10.8.40.15	TCP	1514	443 → 50226 [ACK] Seq=1 Ack=216 Win=525312 Len=14
18321	70.049550	us.configsvc1.live.…	10.8.40.15	TCP	1514	443 → 50226 [ACK] Seq=1461 Ack=216 Win=525312 Len
18322	70.049833	10.8.40.15	us.configsvc1.live.com.akadns.net	TCP	54	50226 → 443 [ACK] Seq=216 Ack=2921 Win=262144 Len
18323	70.214938	10.8.40.15	10.8.40.11	TLSv1.2	172	Application Data

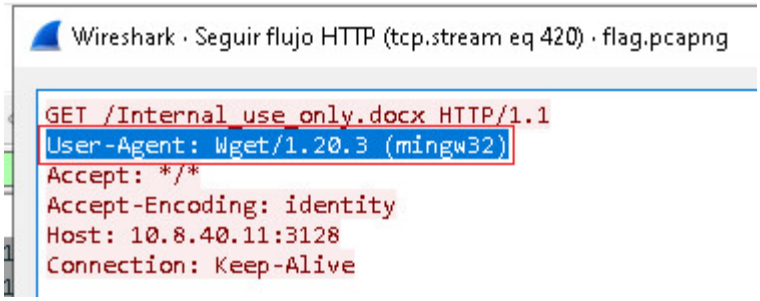
2) Si filtramos la captura de Wireshark buscado solo el tráfico HTTP con destino **10.8.40.11** (servidor web), podemos ver lo siguiente:



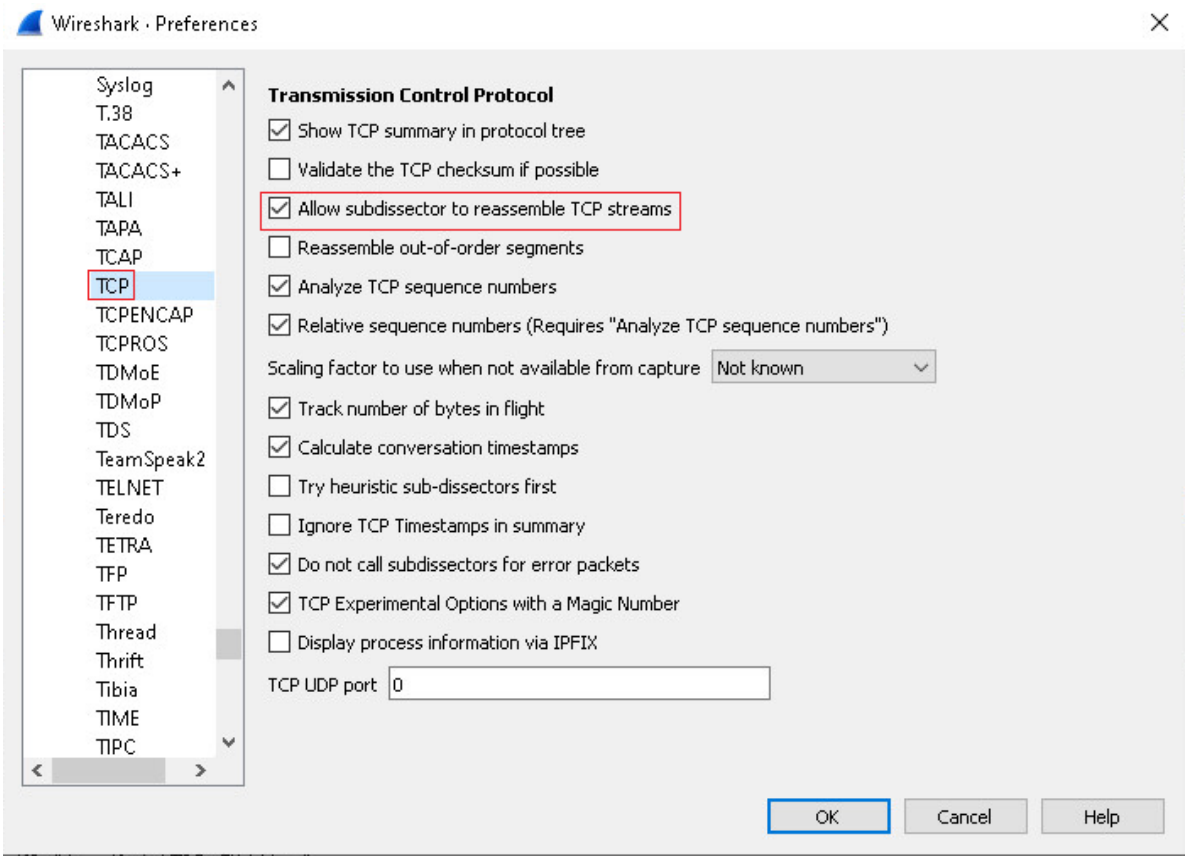
http and ip.dst == 10.8.40.11

No.	Time	Source	Destination	Protocol	Length	Info
16996	63.229881	10.8.40.15	10.8.40.11	HTTP	219	GET /whitepaper-tecnologia.pdf HTTP/1.1
18390	71.595878	10.8.40.15	10.8.40.11	HTTP	216	GET /Internal_use_only.docx HTTP/1.1
19074	80.004096	10.8.40.15	10.8.40.11	HTTP	221	GET /Windows-XP-Security_ESP.pdf HTTP/1.1

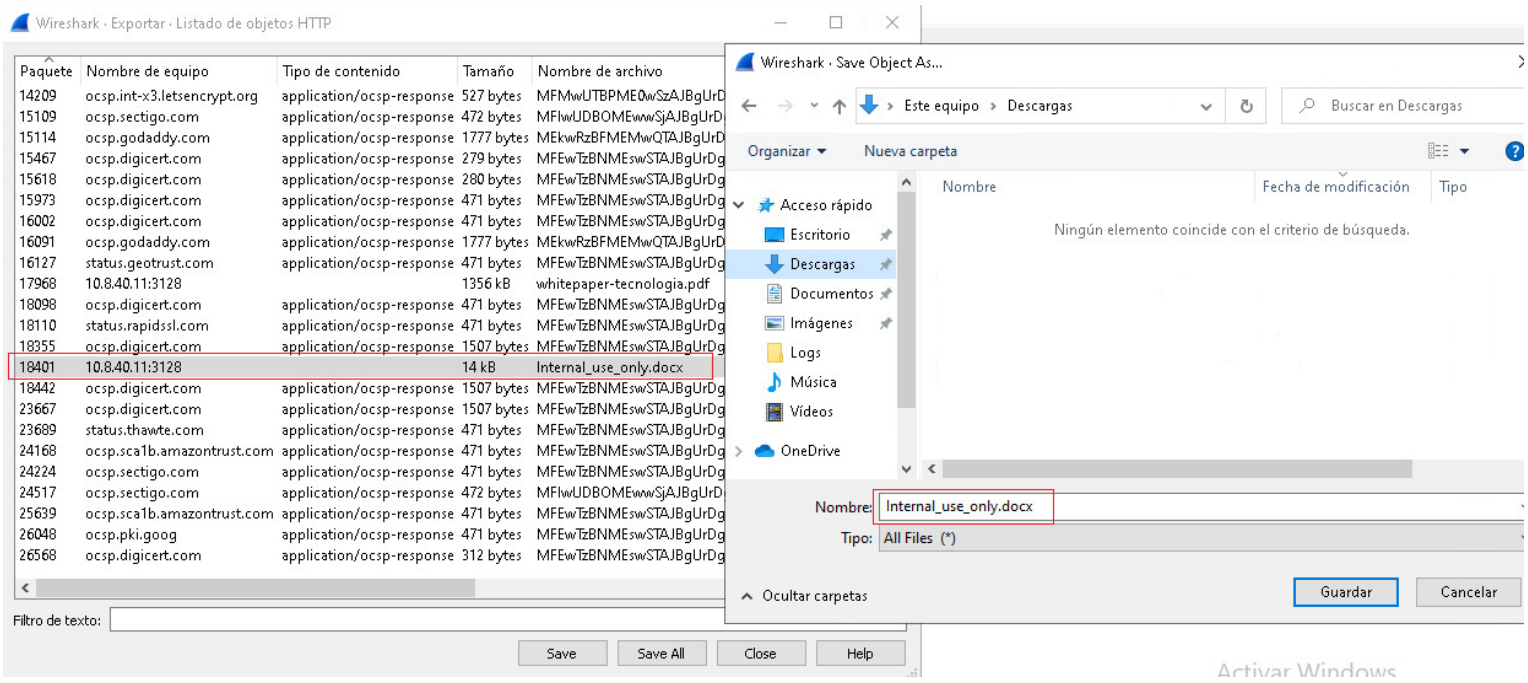
3) Al seleccionar el paquete de más arriba y hacer clic derecho → seguir → flujo HTTP, allí podemos encontrar la información del "user-agent" utilizado para realizar la descarga del documento.



4) Ahora, para poder descargar el documento en cuestión, debemos asegurarnos de que en la configuración de Wireshark esté activada la opción "Edición" → "Preferencias" → "Protocolos" → "TCP" → "Allow subdissector to reassemble TCP streams" (en español: "Permitir que el *subdissector* vuelva a ensamblar secuencias de TCP"). Luego, presionamos "OK".



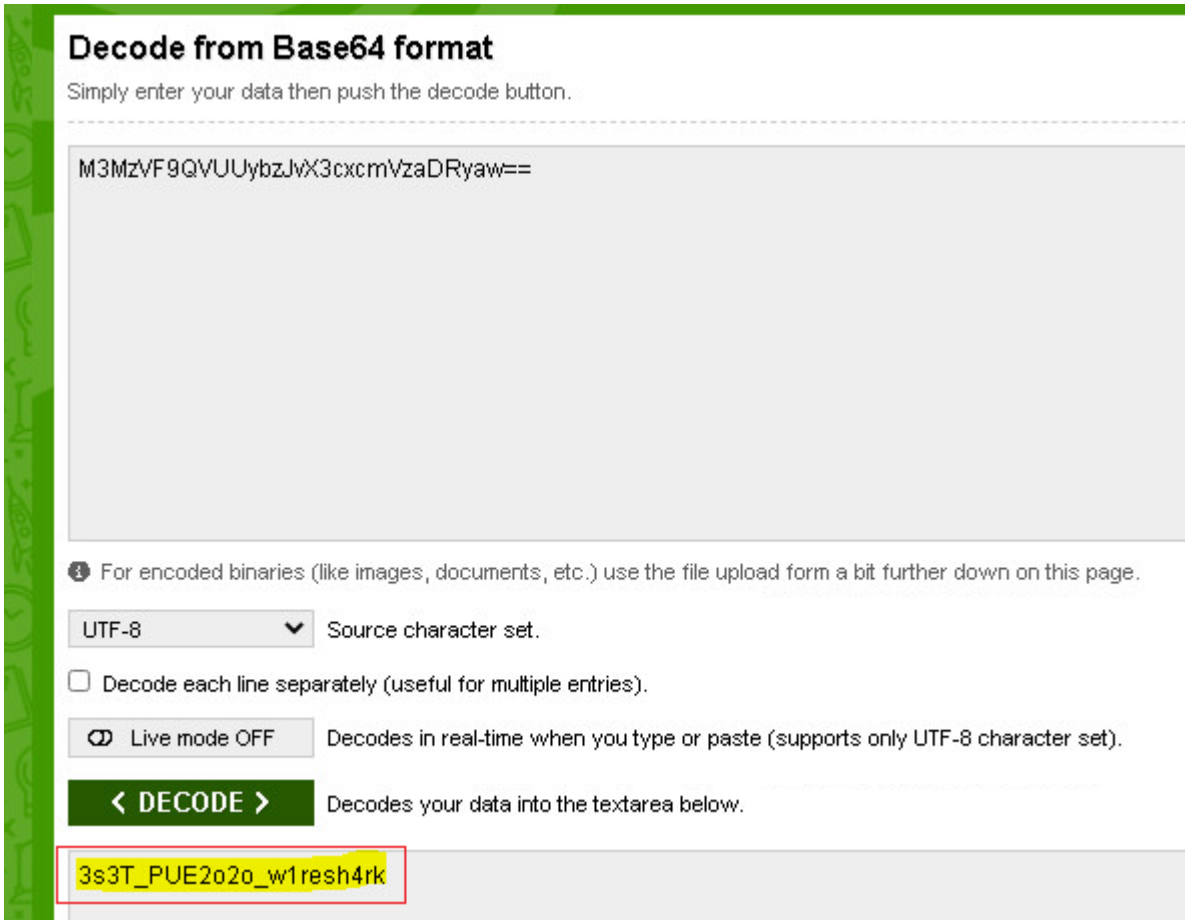
5) Luego, seleccionamos "Archivo" → "Exportar objetos" → "HTTP", buscamos el documento "**Internal_use_only.docx**" de la lista y lo descargamos.



6) Al abrir el documento, a simple vista está vacío.

7) Sin embargo, buscando al seleccionar el contenido del cuerpo del documento vemos que se puede visualizar lo siguiente:

8) Seleccionamos dicho texto, e identificamos que se trata de una cadena en codificada en base64. A partir de conocer esta información, podemos utilizar algún servicio online para decodificar o "traducir" esos caracteres:



De este modo se arriba a la bandera: **3s3T_PUE2o2o_w1resh4rk**.

Recursos y herramientas utilizadas:

- Wireshark (también se puede usar NetworkMiner o herramientas similares)
- https://www.base64encode.org/

9_CaosEnLaVPN.zip

La consigna de este desafío era la siguiente:

¡¡Uno de nuestros profesionales se ha olvidado su contraseña para acceder a la VPN, descubre sus credenciales y obtendrás la flag!!

Para comenzar a resolverlo, utilizando la herramienta PeStudio vemos que fue compilado con .NET y es de 32 bits. Esto lo podemos corroborar viendo las secciones "strings" y la pestaña principal. Ahora, si usamos la herramienta dnSpy-x86, nos encontraremos con la siguiente validación dentro de la clase Form1:

```
!String.IsNullOrEmpty(this.userText.Text) && this.something(this.passText.Text)
```

```
19      private void button1_Click(object sender, EventArgs e)
20      {
21          if (!string.IsNullOrEmpty(this.userText.Text) && this.something(this.passText.Text))
22          {
23              if (MessageBox.Show(this.show_your_self(), this.nice) == DialogResult.OK)
24              {
25                  Application.Exit();
26                  return;
27              }
28          }
29          else
30          {
31              MessageBox.Show(this.magicM, this.wrong);
32          }
33      }
```

Para *crackear* el ejecutable, podemos editar el "if" de la línea 21 quitando la última condición, guardar los cambios en un nuevo archivo e ingresar un nombre de usuario cualquiera.

```
20      {
21          if (!string.IsNullOrEmpty(this.userText.Text))
22          {
23              if (MessageBox.Show(this.show_your_self(), this.nice) == DialogResult.OK)
24              {
25                  Application.Exit();
26                  return;
27              }
28          }
```

Ahora bien, si nos detenemos a entender la lógica del binario, ¿qué hace este?

- Recibe la contraseña ingresada por el usuario y la manipula 4 veces. Antes se fija que el largo de esta sea de 9 caracteres.
- Después tiene 4 rutinas de XOR donde toma cada carácter de la cadena de texto ingresada y le aplica la operación de XOR con una clave que es distinta en cada rutina. Una de las rutinas puede observarse en la siguiente imagen.

```
47      private string manipulate1(string s)
48      {
49          char[] array = s.ToCharArray();
50          for (int i = 0; i < s.Length; i++)
51          {
52              array[i] ^= 'E';
53          }
54          return new string(array);
55      }
```

- Las claves que utiliza son E S E T.
- Después de realizar estas operaciones el resultado final es un nuevo string, del cual obtiene sus respectivos valores hexadecimales y compara que cada carácter sea igual a un carácter puntal.

```
private bool moreSomething(string a)
{
    char[] array = BitConverter.ToString(Encoding.Default.GetBytes(a)).Replace("-", "").ToCharArray();
    return array[0].Equals('6') && array[7].Equals('3') && array[2].Equals('7') && array[3].Equals('2') &
      [10].Equals('2') && array[6].Equals('6') && array[1].Equals('3') && array[8].Equals('4') && array[1
      ('B') && array[13].Equals('4') && array[14].Equals('2') && array[16].Equals('2');
}
```

- Si unimos todos los caracteres, nos da un valor hexadecimal que convertido a texto es "**cr4cK#D!!**".
- Entonces, si tomamos el texto "**cr4cK#D!!**" y hacemos el proceso inverso, vamos a tener la clave correcta que es "**du3dL$C&&**".
- Por último, la lógica que se utiliza para imprimir la flag es similar a la de la password ingresada: una operación de XOR a nivel de bytes.

Finalmente, la bandera es: **3s3T_PUE2o2o_x0r_r3p3aT**.

10_Steganography.zip

Para resolver este desafío, siguen estos pasos:

1. **Utilizar un conversor de binario a texto.**

Puedes usar alguna herramienta online para este propósito, como este sitio.

```
01000101 01101110 00100000 01110101 01101110 00100000 01101101 01100001 01110010
00100000 01100100 01100101 00100000 01100011 01100001 01100100 01100101 01101110
01100001 01110011 00100000 01110000 01110101 01100101 01100100 01100101 01110011
00100000 01000110 01000110 01000110 01000110 01101001 01101100 01110100 01110010
01100001 01110010 00100000 01101100 01100001 00100000 01110011 01100001 01101100
01101001 01100100 01100001
```

Traduciendo lo anterior a ASCII, obtendremos la siguiente pista:

```
"En un mar de cadenas puedes FFFFiltrar la salida"
```

2. **La pista nos sugiere que será necesario utilizar una herramienta para buscar cadenas en hexadecimal.**

La palabra FFFFiltrar es clave. Con base en ella, podemos usar hexdump para encontrar todas las coincidencias de "FF FF".

```
$ hexdump -C Steganography.jpg | grep -i "FF FF"
c0 33 73 33 54 5f ba 98  87 ad ec d9 af 93 ff ff  |.3s3T_..........|
03 f5 0c db c1 12 50 55  45 1c d2 b0 ff ff 06 9a  |......PUE.......|
ff ff 9d e3 f7 b9 bf da  15 32 6f 32 6f 5f a7 af  |.........2o2o_..|
fc 53 74 33 00 8b 1b 80  e4 c8 ff ff ff fa 0f 0b  |.St3............|
02 02 02 02 67 34 4e 30  ff ff 02 02 02 02 02 02  |....g4N0........|
9f ff ff b1 7f b9 af fd  47 72 34 da c3 0a 05 e2  |........Gr4.....|
1a ff 00 f9 7f ff ff c9  d5 f7 af 70 48 79 c8 9c  |...........pHy..|
```

En la salida de consola aparecerá la bandera: **3s3T_PUE2o2o_St3g4N0Gr4pHy**.

11_BruteForce.zip

Para resolver este desafío podemos seguir los siguientes pasos:

1. **Utilizar un conversor de hexadecimal a texto:**

```
4c 61 20 62 61 6e 64 65 72 61 20 64 65 20 65 73 74 65 20 72 65 74 6f 20 6c 61 20
6f 62 74 65 6e 64 72 e1 73 20 61 6c 20 64 65 73 63 69 66 72 61 72 20 6c 61 20 63
6f 6e 74 72 61 73 65 f1 61 20 61 6c 6d 61 63 65 6e 61 64 61 20 65 6e 20 65 73 74
61 20 6c ed 6e 65 61 20 64 65 6c 20 61 72 63 68 69 76 6f 20 2f 65 74 63 2f 73 68
61 64 6f 77 20 0a 24 36 24 55 49 4e 69 43 6c 50 43 24 31 4b 35 73 59 67 41 71 41
71 58 48 38 75 51 51 70 78 48 35 62 54 6e 47 42 49 30 68 52 69 53 6c 51 39 52 56
4f 6f 51 35 69 78 69 4d 33 74 73 56 76 6d 51 6a 41 79 62 2e 2f 37 38 59 30 2e 36
44 6b 78 6f 69 6a 53 50 6b 6c 4f 66 2f 2e 75 35 49 42 37 63 4c 62 31 0a 59 61 20
63 6f 6e 6f 63 65 73 20 6c 61 20 65 73 74 72 75 63 74 75 72 61 20 64 65 20 6c 61
20 62 61 6e 64 65 72 61 2c 20 73 69 20 6e 6f 20 65 73 20 61 73 ed 2c 20 61 71 75
ed 20 74 69 65 6e 65 73 20 75 6e 20 72 65 63 6f 72 64 61 74 6f 72 69 6f 3a 20 33
73 33 54 5f 50 55 45 32 6f 32 6f 5f 0a 42 75 73 63 61 20 6c 6f 73 20 73 65 69 73
20 63 61 72 61 63 74 65 72 65 73 20 72 65 73 74 61 6e 74 65 73 2e 20 43 6f 6e 73
69 64 65 72 61 20 fa 6e 69 63 61 6d 65 6e 74 65 20 6d 61 79 fa 73 63 75 6c 61 73
2c 20 6d 69 6e fa 73 63 75 6c 61 73 20 79 20 6e fa 6d 65 72 6f 73 2e
```

2. **Descifrar la cadena del archivo /etc/shadow para obtener la contraseña (bandera):**

La bandera de este reto la obtendrás al descifrar la contraseña almacenada en esta línea del archivo /etc/shadow

```
$6$UINiClPC$1K5sYgAqAqXH8uQQpxH5bTnGBI0hRiSlQ9RVOoQ5ixiM3tsVvmQjAyb./78Y0.6Dkxo
ijSPklOf/.u5IB7cLb1
```

Ya conoces la estructura de la bandera; si no es así, aquí tienes un recordatorio: `3s3T_PUE2o2o_`
Busca los seis caracteres restantes. Considera únicamente mayúsculas, minúsculas y números.

3. **Utilizar una herramienta como hashcat o escribir un script para probar las combinaciones de caracteres que, agregados cadena conocida, generen el hash de la contraseña (flag), mediante un ejercicio de fuerza bruta:**

Para resolver este paso, aquí tienes una propuesta de solución:

```
#!/usr/bin/perl

$shadow="\$6\$UINiClPC\$\/gYe4jatlKgLLc3qlXm\.Q306A9JHN36RAIXuRK7mW5hokBCllCAM9R9g
\/bErrUj43A3ZdhQbwNdKUQV8v4RdQ1";

@caracteres = ('0'..'9', 'a'..'z', 'A'..'Z');

$flag="3s3T_PUE2o2o_"

$salt="\$6\$UINiClPC\$";

foreach $i (@caracteres)
{
      foreach $j (@caracteres)
      {
            foreach $k (@caracteres)
            {
                  foreach $l (@caracteres)
                  {
                        foreach $m (@caracteres)
                        {
```

```
                                foreach $n (@caracteres)
                                {
                                        $bandera = "$flag$i$j$k$l$m$n";
                                        $cifrado=crypt($bandera,$salt);
                                        print "\t$bandera\t$cifrado\n";
                                        if($cifrado eq $shadow)
                                        {
                                                print "Bandera: $bandera\n";
                                                last;
                                        }
                                }
                        }
                }
        }
}
```

Así, finalmente encontramos la bandera: **3s3T_PUE2o2o_Sh4d0W**.

12_Miscellaneous.zip

Para resolver este desafío, deberemos hacer lo siguiente:

1. **Utilizar un conversor de base64 a texto:**

```
KDgsNikKKDEwLDgpCig2LDgpCig5LDcpCig0LDE2KQooMyw3KQooNSwxMykKKDEsMTEpCigxMywxNCk
KKDIsMTApCigxMSw4KQ==
(8,6)
(10,8)
(6,8)
(9,7)
(4,16)
(3,7)
(5,13)
(1,11)
(13,14)
(2,10)
(11,8)
```

2. **Resolver el crucigrama.**

Horizontal

2. ESET is the first internet security provider to add a dedicated layer into its solution. ESET _____ checks and enforces the security of the pre-boot environment that is compliant with a specific BIOS replacement specification. Itis designed to monitor the integrity of the firmware and if modification is detected, it notifies the user.

uefiscanner

9. This type of detection monitors typically exploitable applications (browsers, document readers, email clients, etc...). Instead of just aiming for specific CVE identifiers, it focuses on exploitation techniques used during execution to detect and block threats immediately.

exploitblocker

10. ESET solutions emulate different components of computer hardware and software to execute a suspicious sample in an isolated virtualized environment. This helps catch heavily obfuscated malware which is attempting to evade detection. As this is local to a computer, this protection is not as powerful as ESET's cloud solution, ESET Dynamic Threat Defense.

in-productsandbox

11. This detection method allows ESET to detect malicious network communication used by botnets, and at the same time identify the application causing the network traffic.

botnetprotection

12. When inspecting an object such as a file or URL, before any scanning takes place, our products check the local cache for known malicious or whitelisted benign objects to improve scanning performance. Afterwards, ESET LiveGrid's reputation system is queried for the object's reputation to see if an object has been seen elsewhere as malicious. This improves scanning efficiency and enables faster sharing of malware intelligence with our customers.

reputation&cache

13. Malware can be obfuscated in such a way that not all execution paths can be analyzed; it can contain conditional or time triggers for the code; and, very frequently, it can download new components during its lifetime. To tackle these issues, _____ monitors the behavior of a malicious process and scans it once it decloaks in memory.

advancedmemoryscanner

Vertical

1. Allows detection of known vulnerabilities on the network level. This allows ESET to implement detection and blocking of network exploits in widely used protocols like SMB, RPC, RDP, etc... And adds protection against network spreading malware or network attacks by an attacker.

networkattackprotection

3. This detection method has 2 pieces. One piece can scan javascript in web browsers to find and block malware. The other piece uses Microsoft's Antimalware Scan Interface (AMSI) to scan inside of powershell to detect malicious scripts or commands.

script-basedattacksprotection

4. This protection is sometimes called Cloud Malware Protection System. Unknown, potentially malicious samples collected from endpoints are submitted to the cloud and subjected to automatic sandboxing and behavioral analysis, which results in the creation of automated detections if malicious characteristics are confirmed.

livegridprotection

5. This ESET feature allows an administrator to allow or block different kinds of removable devices (example USB drives, CD/DVD, portable devices, etc...).

devicecontrol

6. ESET has added its own _____ engine to endpoint products. This engine is called Augur and uses the combined power of neural networks (like deep learning and long short-term memory). This allows it to label objects as clean, potentially unwanted, or malicious.

dnadetections

7.While the malicious code can be easily modified or obfuscated by attackers, the behavior of objects cannot be changed so easily and ESET _____ are designed to take advantage of this principle. We perform deep analysis of code, extracting the "genes" that are responsible for its behavior. Such behavioral genes contain much more information than the indicators of compromise (10Cs) that some so called "next-gen" solutions claim to be "the better alternative" to signature detection.

ransomwareshield

8. This additional layer of security helps protect users from encryption malware. This is done by monitoring and evaluating executed applications based on their reputation and behavior.

machinelearning

3. Identificar las coordenadas en el crucigrama.

(8,6)	m
(10,8)	u
(6,8)	l
(9,7)	t
(4,16)	i
(3,7)	-
(5,13)	l
(1,11)	a
(13,14)	y
(2,10)	e
(11,8)	r

Bandera: 3s3T_PUE2o2o_multi-layer

13_TheRicksMustBeCrazy.zip

Este desafío consiste en buscar información dentro de los metadatos de las imágenes listadas en la carpeta. Ya que son 460 archivos ponerse a revisar imagen por imagen puede ser un trabajo dispendioso y poco práctico, así que podemos hacer un pequeño script que nos permita recuperar toda la información que pueda resultar interesante para resolver el desafío. Podrás ver un ejemplo de script al final de esta resolución.

Una revisión de los metadatos de las imágenes nos arroja que hay ciertos archivos que tienen más información en comparación con el resto de los archivos:

```
3s3T_PUE2o2o_161.jpg
ExifOffset : 26
DateTimeOriginal : 0001:01:01 00:00:02
3s3T_PUE2o2o_162.jpg
ExifOffset : 26
DateTimeOriginal : 0001:01:01 00:00:02
3s3T_PUE2o2o_163.jpg
ExifOffset : 26
DateTimeOriginal : 0001:01:01 00:00:02
3s3T_PUE2o2o_164.jpg
ExifOffset : 26
DateTimeOriginal : 0001:01:01 00:00:02
3s3T_PUE2o2o_166.jpg
ExifOffset : 26
DateTimeOriginal : 0001:01:01 00:00:02
3s3T_PUE2o2o_167.jpg
ExifOffset : 26
DateTimeOriginal : 0001:01:01 00:00:02
3s3T_PUE2o2o_168.jpg
ExifOffset : 26
DateTimeOriginal : 0001:01:01 00:00:02
3s3T_PUE2o2o_169.jpg
ExifOffset : 2134
XPTitle : b'V\x00\x00\x00'
ImageDescription : V
XPSubject : b'4\x00d\x00 \x003\x000\x00 \x007\x002\x00 \x007\x004\x00 \x007\x009\x00\x00\x00'
DateTimeOriginal : 0001:01:01 00:00:02
3s3T_PUE2o2o_17.jpg
ExifOffset : 26
DateTimeOriginal : 0001:01:01 00:00:02
3s3T_PUE2o2o_170.jpg
ExifOffset : 26
DateTimeOriginal : 0001:01:01 00:00:02
3s3T_PUE2o2o_171.jpg
ExifOffset : 26
DateTimeOriginal : 0001:01:01 00:00:02
```

Imagen 1. Archivo '169' con metadatos adicionales

```
3s3T_PUE2o2o_239.jpg
ExifOffset : 26
DateTimeOriginal : 0001:01:01 00:00:02
3s3T_PUE2o2o_24.jpg
ExifOffset : 26
DateTimeOriginal : 0001:01:01 00:00:02
3s3T_PUE2o2o_240.jpg
ExifOffset : 26
DateTimeOriginal : 0001:01:01 00:00:02
3s3T_PUE2o2o_241.jpg
ExifOffset : 26
DateTimeOriginal : 0001:01:01 00:00:02
3s3T_PUE2o2o_242.jpg
ExifOffset : 26
DateTimeOriginal : 0001:01:01 00:00:02
3s3T_PUE2o2o_243.jpg
ExifOffset : 2134
XPTitle : b'I\x00I\x00\x00\x00'
ImageDescription : II
XPSubject : b'5\x000\x00 \x005\x005\x00 \x004\x005\x00 \x003\x002\x00 \x006\x00f\x00 \x003\x002\x00 \x006\x00f\x00
\x005\x00f\x00\x00\x00'
DateTimeOriginal : 0001:01:01 00:00:02
3s3T_PUE2o2o_244.jpg
ExifOffset : 26
DateTimeOriginal : 0001:01:01 00:00:02
3s3T_PUE2o2o_246.jpg
ExifOffset : 26
DateTimeOriginal : 0001:01:01 00:00:02
3s3T_PUE2o2o_247.jpg
ExifOffset : 26
DateTimeOriginal : 0001:01:01 00:00:02
3s3T_PUE2o2o_248.jpg
ExifOffset : 26
DateTimeOriginal : 0001:01:01 00:00:02
```

Imagen 2. Archivo '243' con metadatos adicionales

Para no buscar en todo el listado, podemos listar solamente aquellos archivos que tengan estos metadatos adicionales para entender de que se tratan:

```
3s3T_PUE2o2o_169.jpg
ImageDescription : V
3s3T_PUE2o2o_169.jpg
XPSubject : b'4\x00d\x00 \x003\x000\x00 \x007\x002\x00 \x007\x004\x00 \x007\x009\x00\x00\x00'
3s3T_PUE2o2o_243.jpg
ImageDescription : II
3s3T_PUE2o2o_243.jpg
XPSubject : b'5\x000\x00 \x005\x005\x00 \x004\x005\x00 \x003\x002\x00 \x006\x00f\x00 \x003\x002\x00 \x006\x00f\x00 \x005\x00f\x
00\x00\x00'
3s3T_PUE2o2o_463.jpg
ImageDescription : IV
3s3T_PUE2o2o_463.jpg
XPSubject : b'3\x004\x00 \x006\x00e\x00 \x006\x004\x00 \x005\x00f\x00\x00\x00'
3s3T_PUE2o2o_49.jpg
ImageDescription : I
3s3T_PUE2o2o_49.jpg
XPSubject : b'3\x003\x00 \x007\x003\x00 \x003\x003\x00 \x005\x004\x00 \x005\x00f\x00\x00\x00'
3s3T_PUE2o2o_75.jpg
ImageDescription : III
3s3T_PUE2o2o_75.jpg
XPSubject : b'5\x002\x00 \x003\x001\x00 \x006\x003\x00 \x006\x00b\x00 \x005\x00f\x00\x00\x00'
```

Imagen 3. Los archivos con metadatos adicionales

Con el listado de los archivos que tienen los metadatos adicionales, es más fácil darse cuenta de que se sugiere un orden de acuerdo con el valor *ImageDescription*: 49, 243, 75, 463, 169.

Lo que resta es ordenar las cadenas de caracteres de estas imágenes que se encuentran en el campo *XPSubject*, que en este caso están en formato hexadecimal:

33 73 33 54 5f 50 55 45 32 6f 32 6f 5f 52 31 63 6b 5f 34 6e 64 5f 4d 30 72 74 79

Convirtiendo la cadena anterior a formato ASCII vamos a encontrar el flag:

3s3T_PUE2o2o_R1ck_4nd_M0rty

A continuación, anexamos un ejemplo de script .IPYNB que se podría utilizar junto a la aplicación Jupyter Notebook para obtener los metadatos:

```
{
 "cells": [
  {
   "cell_type": "code",
   "execution_count": null,
   "metadata": {
    "collapsed": true
   },
   "outputs": [],
   "source": [
    "#Librerias para el manejo de metadatos\n",
    "from PIL import Image\n",
    "from PIL.ExifTags import TAGS\n",
    "#Libreria para el manejo de archivos\n",
    "import os"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "metadata": {
    "collapsed": true
   },
   "outputs": [],
   "source": [
    "# Listar nombres de los archivos jpg en la carpeta\n",
    "ruta_archivos = '[ruta_de_los_archivos]'\n",
    "contenido = os.listdir(ruta_archivos)\n",
    "imagenes = []\n",
    "for archivo in contenido:\n",
    "    if os.path.isfile(os.path.join(ruta_archivos, archivo)) and fichero.endswith('.jpg'):\n",
    "        imagenes.append(archivo)"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "metadata": {
    "collapsed": true
   },
   "outputs": [],
   "source": [
```

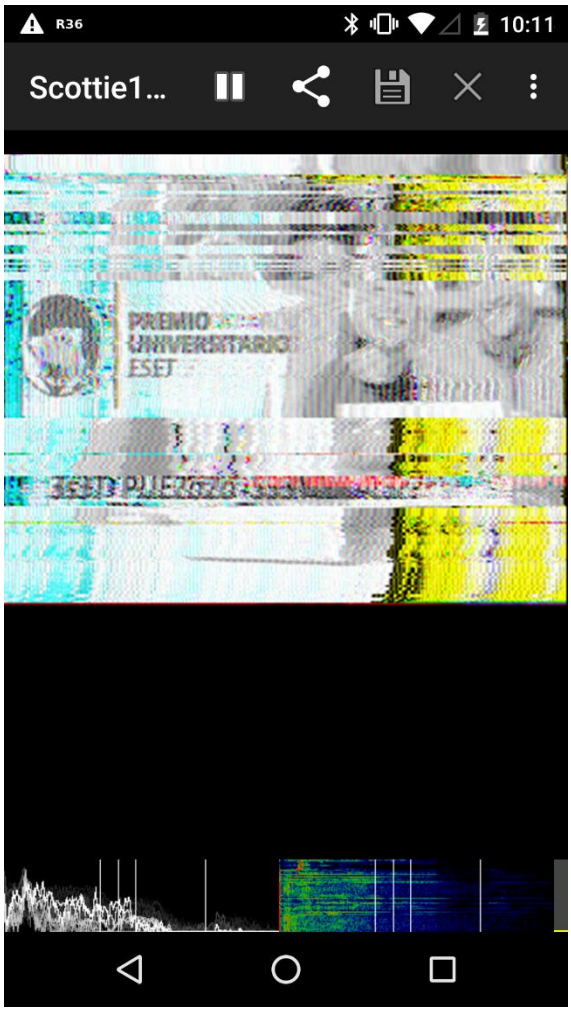
```
    "# Diccionario para cargar los datos\n",
    "exif_dic = {}\n",
    "\n",
    "for nombre in imagenes:\n",
    "        \n",
    "    # Obtener metadatos\n",
    "    imagen = Image.open(nombre)\n",
    "    datos_exif = imagen._getexif()\n",
    "  \n",
    "    # Ordenar los metadatos\n",
    "    for tag, value in datos_exif.items():\n",
    "        \n",
    "        #Extraer los nombres de los tags y su contenido\n",
    "        etiqueta = TAGS.get(tag, tag)\n",
    "        valor = datos_exif.get(tag)\n",
    "        exif_dic[etiqueta] = valor\n",
    "        \n",
    "        # Imprimir los datos\n",
    "        if str(etiqueta) == 'ImageDescription'or str(etiqueta) == 'XPSubject':\n",
    "            print(nombre)\n",
    "            print(str(etiqueta) + \" : \" +str(valor)) "
   ]
  }
 ],
 "metadata": {
  "kernelspec": {
   "display_name": "Python 3",
   "language": "python",
   "name": "python3"
  },
  "language_info": {
   "codemirror_mode": {
    "name": "ipython",
    "version": 3
   },
   "file_extension": ".py",
   "mimetype": "text/x-python",
   "name": "python",
   "nbconvert_exporter": "python",
   "pygments_lexer": "ipython3",
   "version": "3.6.3"
  }
 },
 "nbformat": 4,
 "nbformat_minor": 2
}
```

14_EscuchaLoQueVes.zip

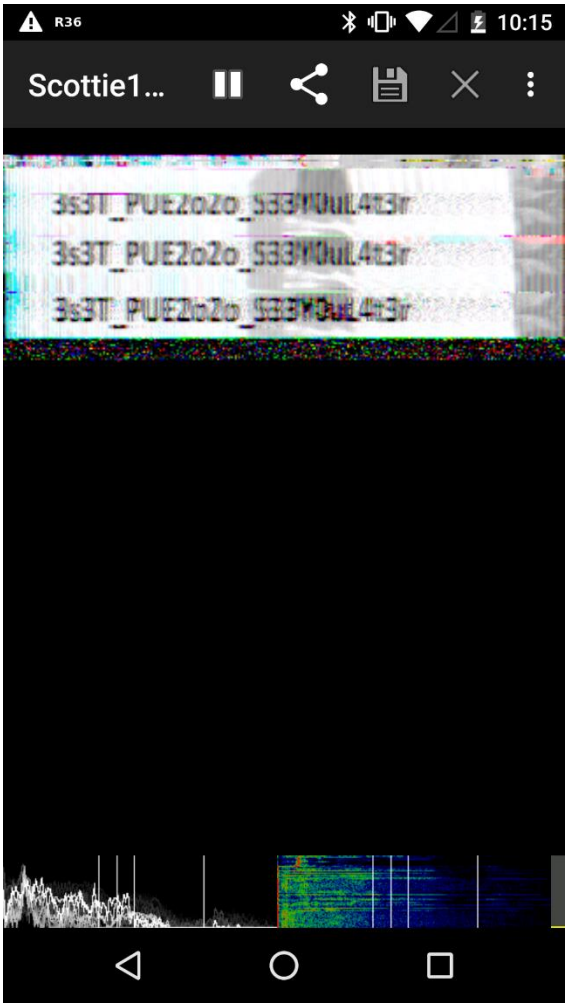
Como el nombre del archivo lo anticipa, este desafío implica la posibilidad de escuchar (con audios) lo que vemos (imágenes). En efecto, el archivo que se nos es brindado como parte de la consigna es un audio que parece esconder datos. Si consultamos motores de búsqueda como Google para saber qué tipos de codificaciones permiten transmitir imágenes mediante sonidos, rápidamente nos encontraremos con el protocolo SSTV (el mismo que fue utilizado para transmitir imágenes desde la Luna).

Para procesar el archivo de audio podemos, por ejemplo, descargar una de las tantas apps decodificadoras de SSTV en nuestro teléfono. Para la resolución de este desafío utilizaremos Robot36. Nos pedirá acceso al micrófono, y le permitiremos escuchar el audio que reproduciremos con algún otro dispositivo cercano.

A medida que avanza el audio, podremos ver cómo aparece una imagen en la aplicación.



Se trata de la imagen del Premio Universitario ESET, pero parece contener la bandera en su parte inferior. De esta flag podemos distinguir claramente el comienzo: 3s3T_PUE2o2o_. Para poder identificar el resto, podemos volver a reproducir el audio desde el minuto 1:20 las veces que sea necesario para obtener el mensaje secreto de manera clara.

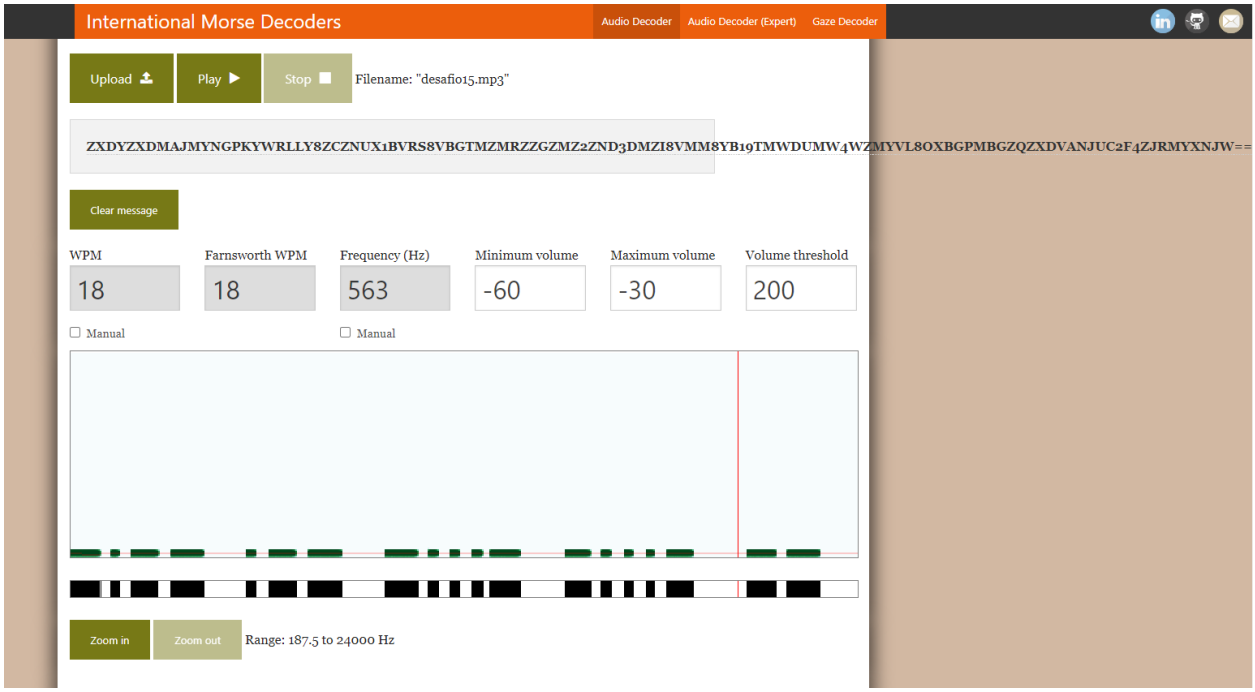


Tras varios intentos, obtenemos la bandera: **3s3T_PUE2o2o_S33Y0uL4t3r**.

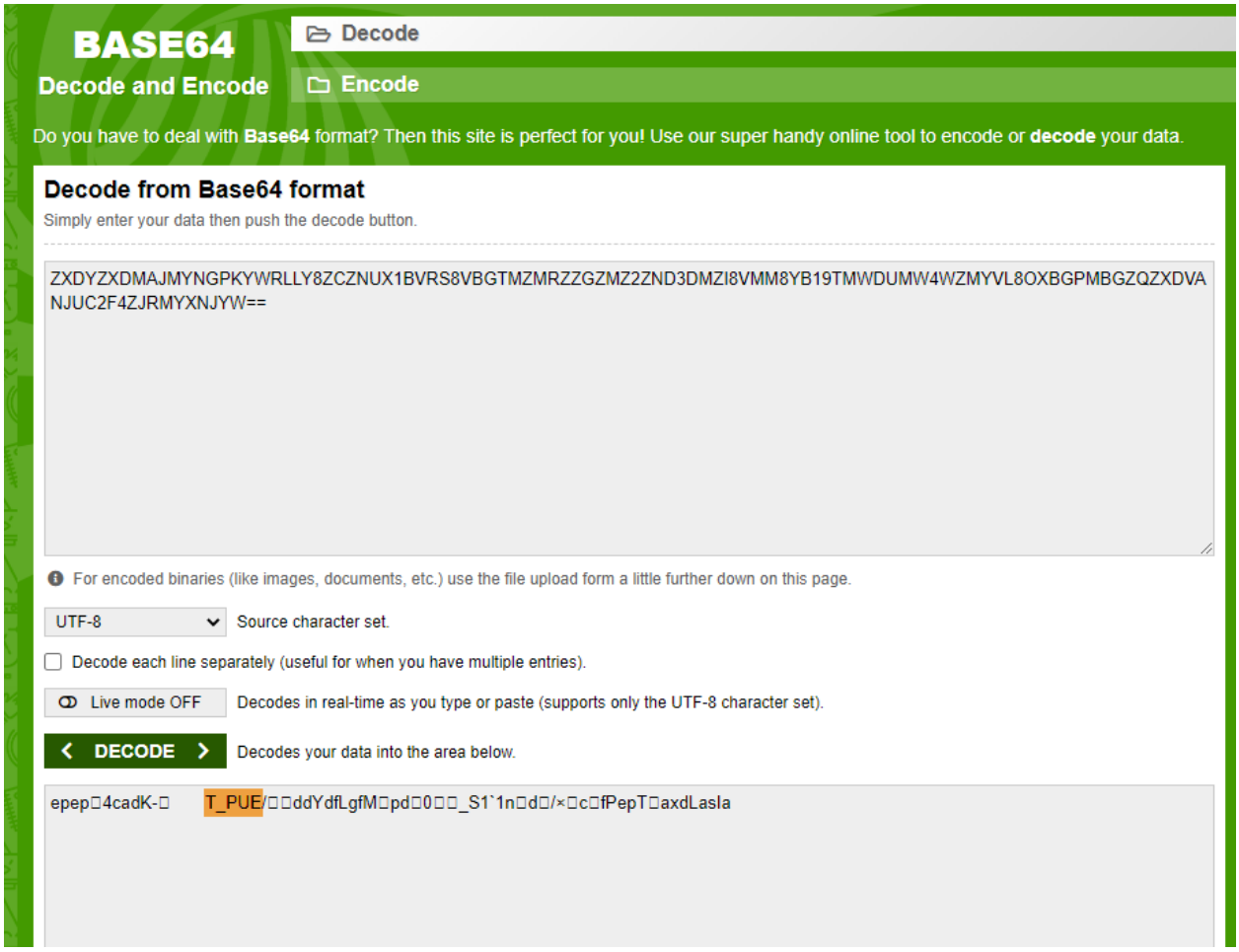
15_Radio.zip

Al abrir el archivo de audio nos damos cuenta de que se trata de código Morse. Para poder leerlo, recurrimos a alguna aplicación decodificadora; pueden encontrarse muchas para teléfonos móviles como Morse Code Reader, y también online como Morse Code Adaptive Audio Decoder. Para la resolución de este ejercicio, utilizaremos esta última.

Una vez cargado el audio, presionamos el botón Play y el sitio comenzará a decodificar el mensaje oculto. Al terminar, obtendremos la cadena "ZXDYZXDMAJMYNGPKYWRLLY8ZCZNUX1BVRS8VBGTMZMRZZGZMZ2ZND3DMZI8VMM8YB19TMWDUMW4WZMYVL8OXBGPMBGZQZXDVANJUC2F4ZJRMYXNJYW==".

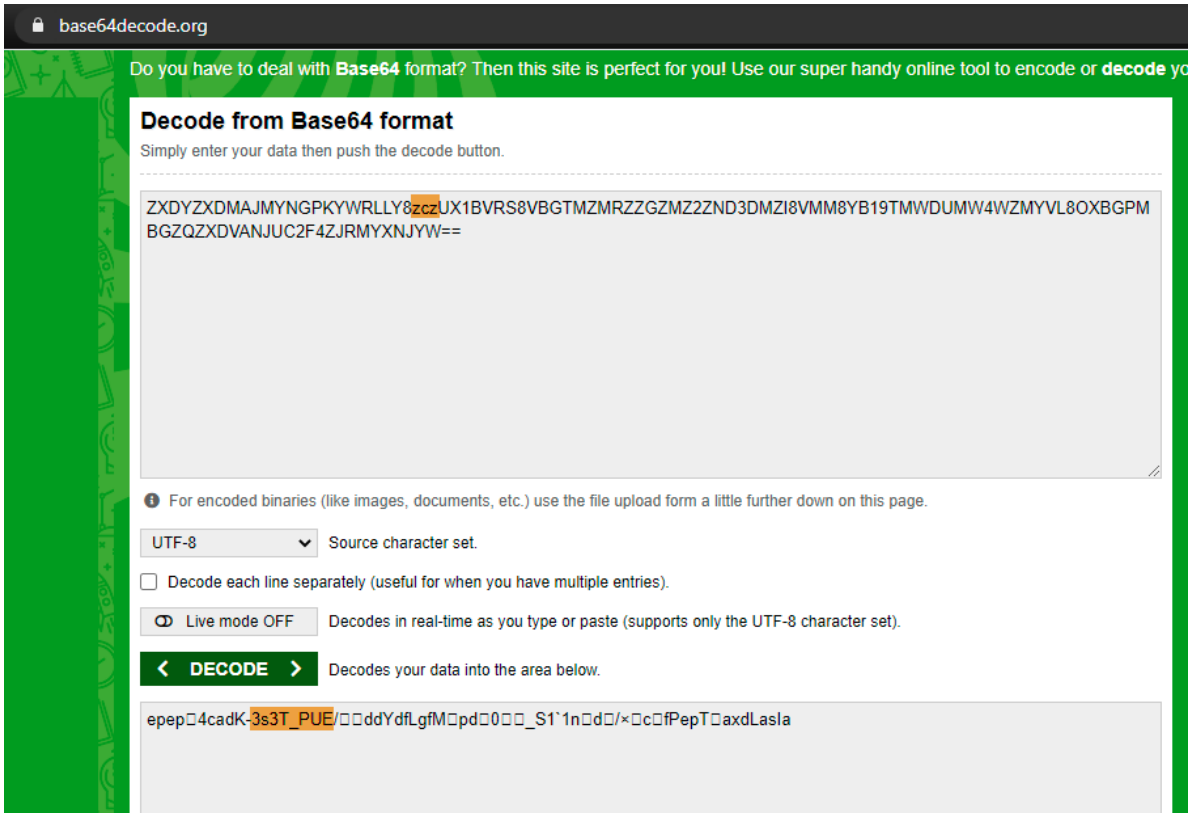


Al ver el formato del resultado, nos damos cuenta de que se trata de una cadena de texto codificada en base 64. Ahora podemos utilizar un decodificador online o un script en Python para decodificar el texto.

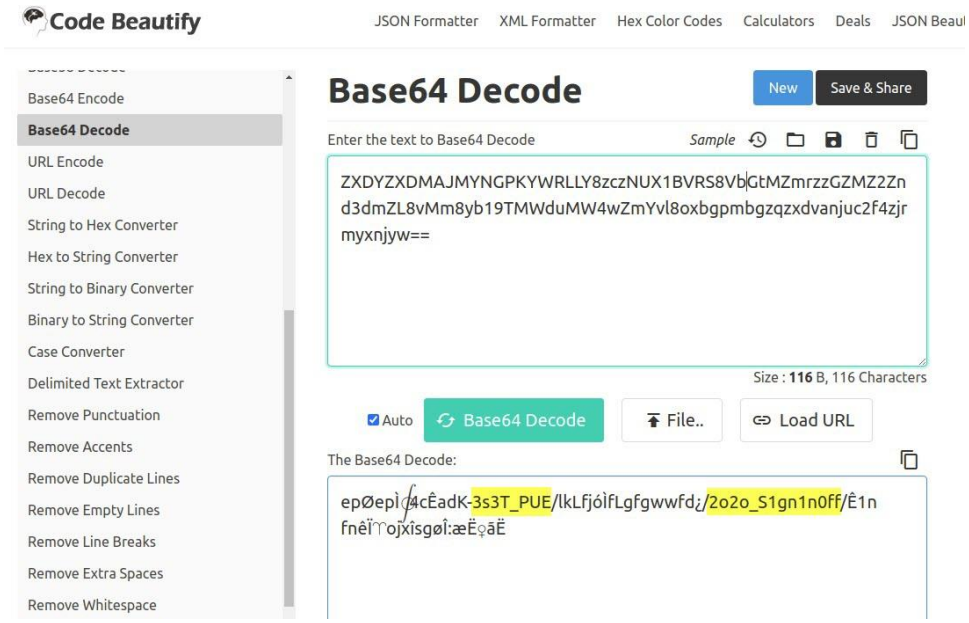


Como vemos, pareciera que la cadena está malformada y que solo estamos obteniendo basura. Sin embargo, puede observarse "T_PUE" en medio, lo cual debiese llamarnos la atención ya que el formato de la bandera es muy similar según lo estipulado en las bases y condiciones, y también acorde a las banderas de otros desafíos en esta misma competencia. ¿Qué ocurre? Pues que al pasar la bandera en base 64 a código Morse se ha perdido información sobre qué caracteres debiesen estar en mayúscula y cuáles en minúscula para que la cadena de texto pueda ser decodificada correctamente.

De hecho, podemos ver que, cambiando algunos caracteres de la cadena original a minúscula, se revelan más caracteres de la bandera y ahora se observa que comienza con 3s3T_PUE.



¿Qué podemos hacer entonces? La opción más sencilla es tomar en mayúscula la porción de texto codificado que nos permite obtener la primera parte de la clave, y convertir el sobrante a minúscula. Luego, podemos ir transformando de a uno y en manera secuencial los caracteres, jugando con mayúsculas y minúsculas hasta identificar cambios que eliminen los caracteres basura del texto decodificado. En unos minutos obtenemos la flag completa como se ve en la siguiente imagen.



Finalmente, luego de correr el script un buen tiempo, llegamos a obtener la bandera: **3s3T_PUE2o2o_S1gn1n0ff**.

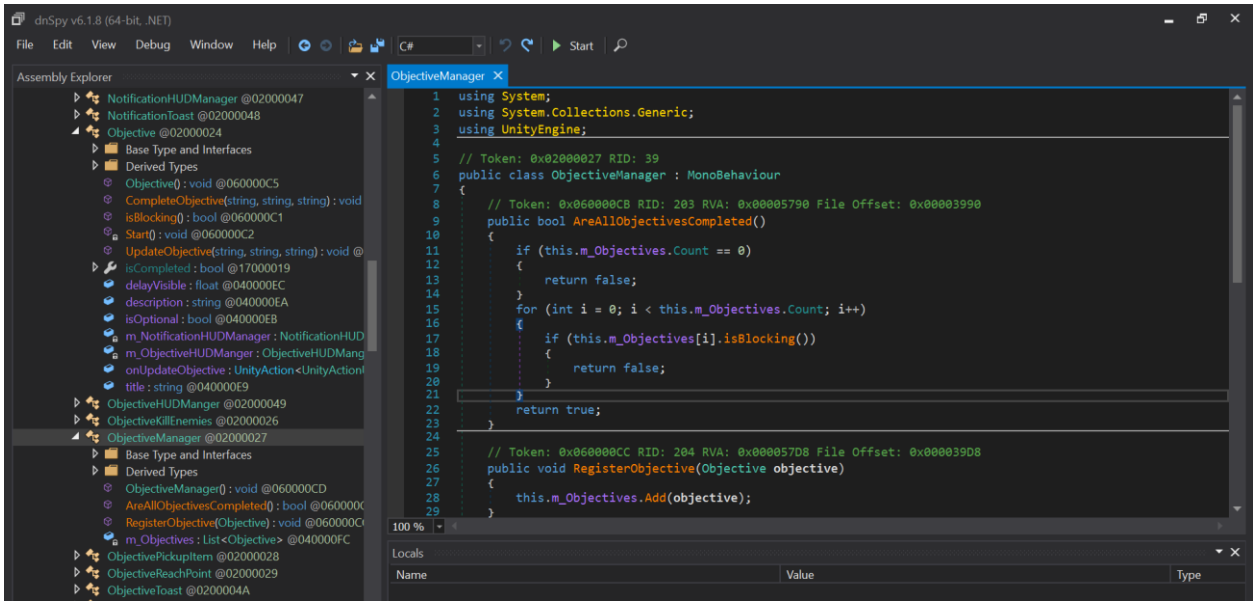
16_Win2GetFlag.7z

En este desafío, encontramos un pequeño juego echo con Unity llamado "Win to get flag". La idea es ganarles a todos los enemigos y al final conseguir la bandera. El desafío tiene diversos puntos de resolución, desde alterar la memoria en tiempo real para tener el 100% de vida, mover al jugador a un punto ciego y ganarles a los enemigos, tratar de obtener los archivos del juego y así obtener información útil (imágenes del juego, strings, etc.) y finalmente modificar el objetivo del juego para ganar al inicio (la vía menos divertida, claro).

Claro que ganar utilizando las habilidades de gamer era una tarea complicada, porque los enemigos pueden reducir la vida del personaje a 0 en tan solo 3 disparos. Por tal motivo, en este write-up explicaremos la forma más fácil de ganar: modificar el objetivo "*Eliminate all the enemies*".

Primero debemos investigar un poco para saber la estructura básica de un juego sencillo echo en Unity y, sabiendo que el archivo "Assembly-CSharp" contiene todos los scripts que no están en el folder del editor, vamos a inspeccionar si nos sirve de algo.

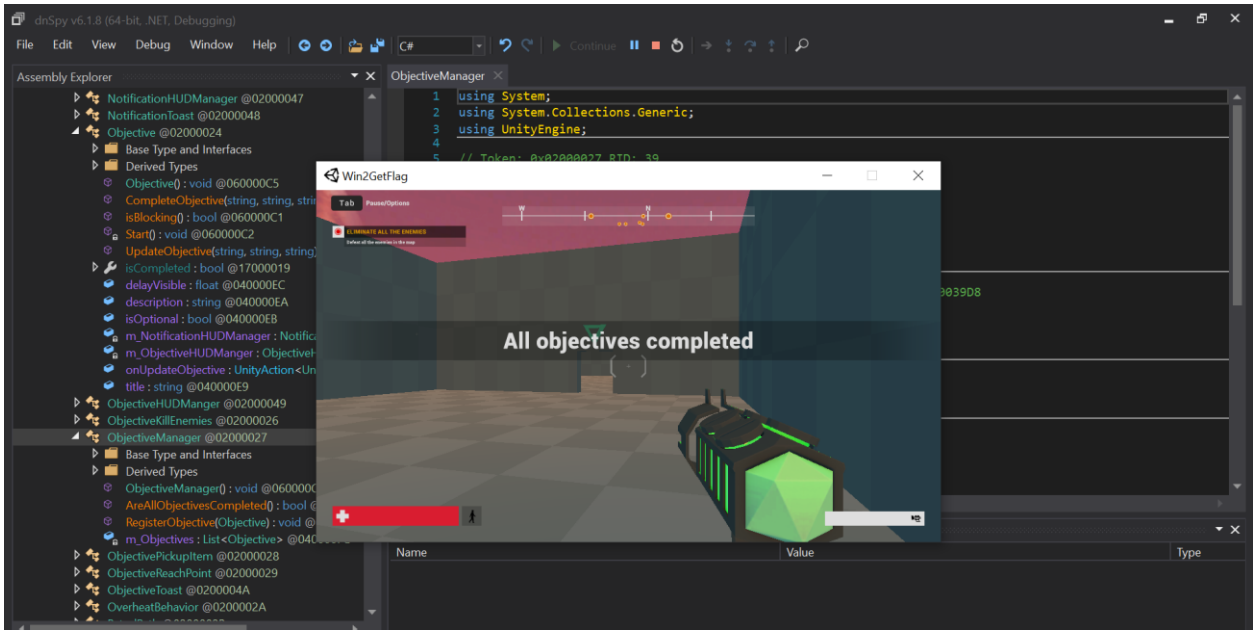
Utilizando una herramienta muy popular llamada dnspy para editar ensamblados .NET y Unity, abrimos el archivo y observamos en primera instancia archivos interesantes, dentro de la clase ObjectiveManager podemos observar el método AreAllObjetivesCompleted(), el cual resulta muy atractivo. ¿Qué sucede cuando este objetivo es cumplido? Editando el método hacemos que regrese siempre verdadero.



Así el método quedaría de la siguiente manera:

```
public bool AreAllObjectivesCompleted()
    {
        return true;
    }
```

Compilando el archivo, guardamos los cambios sobre el mismo DLL, probamos el ejecutable y vemos que los cambios realizados surtieron efecto. Podemos observar que cumplimos con el objetivo del juego que era necesario para ganar.



Una vez desplegada la pantalla de victoria, se mostrará una imagen con el siguiente texto "3h3I_EJT2d2d_W4rz1cv_V4b3". Un poco extraño, ¿no creen? ¡Tal vez no! Con un poco de experiencia y búsqueda por Google, vemos que es un texto cifrado con la técnica de desplazamiento de *Caesar,* con una

herramienta online podemos obtener el texto original obteniendo llegando a la bandera "**3s3T_PUE2o2o_H4ck1ng_G4m3**".

Un reto bastante fácil y divertido. Si quieres poner a prueba tus habilidades de *gamer*, intenta resolver el reto alterando la memoria del juego y ¡derrota a todos los enemigos! - *Happy Hacking*.