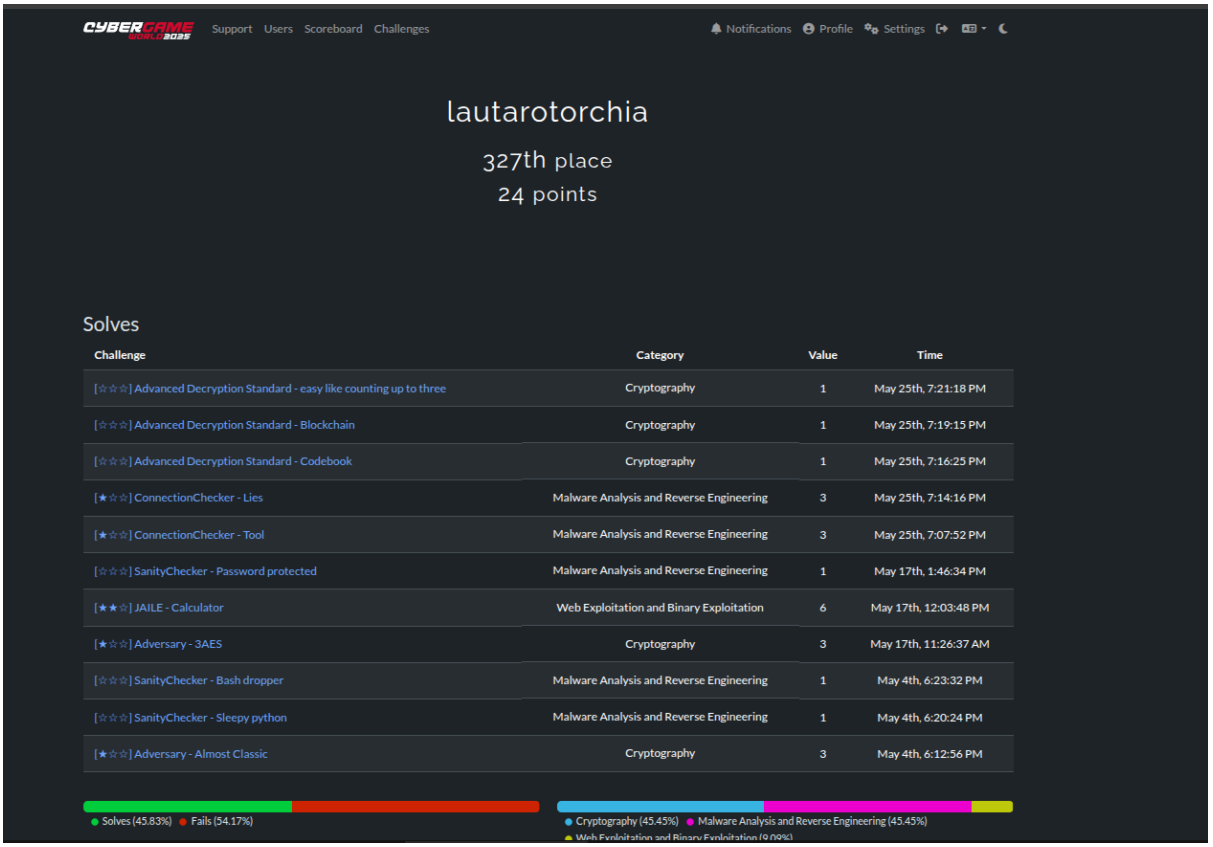
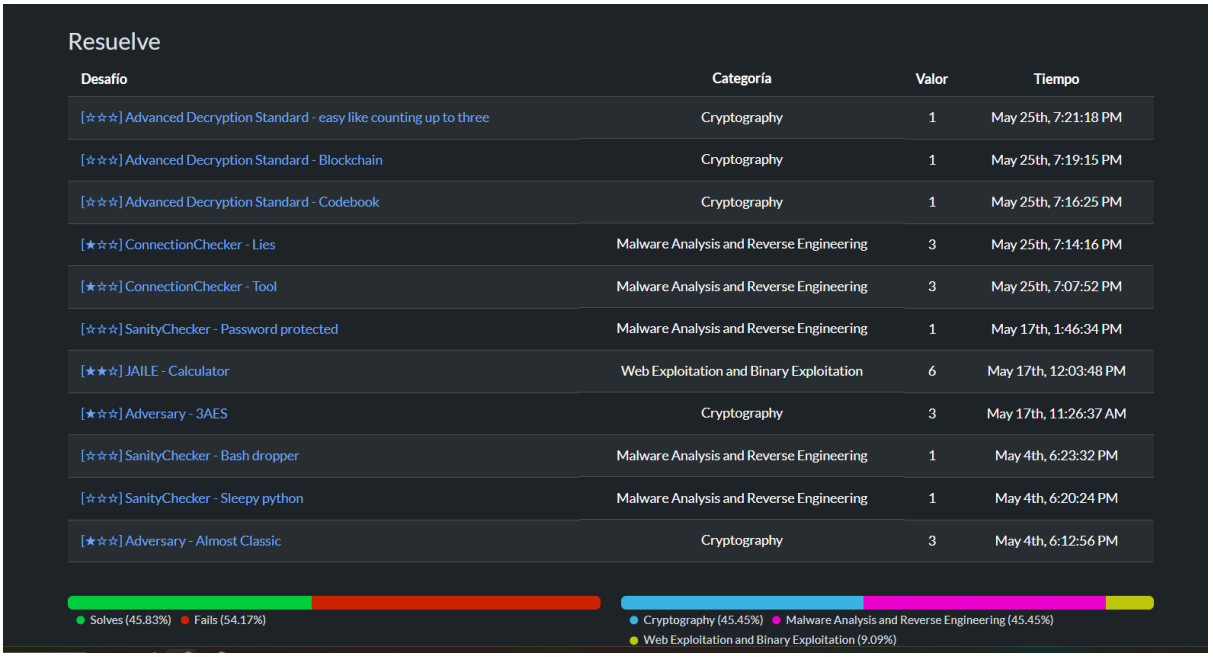
CyberGames CTF Writeup

CTF / Plataforma: CyberGames CTF

Link: https://ctf-world.cybergame.sk/challenges

Mi Nick: lautarotorchia

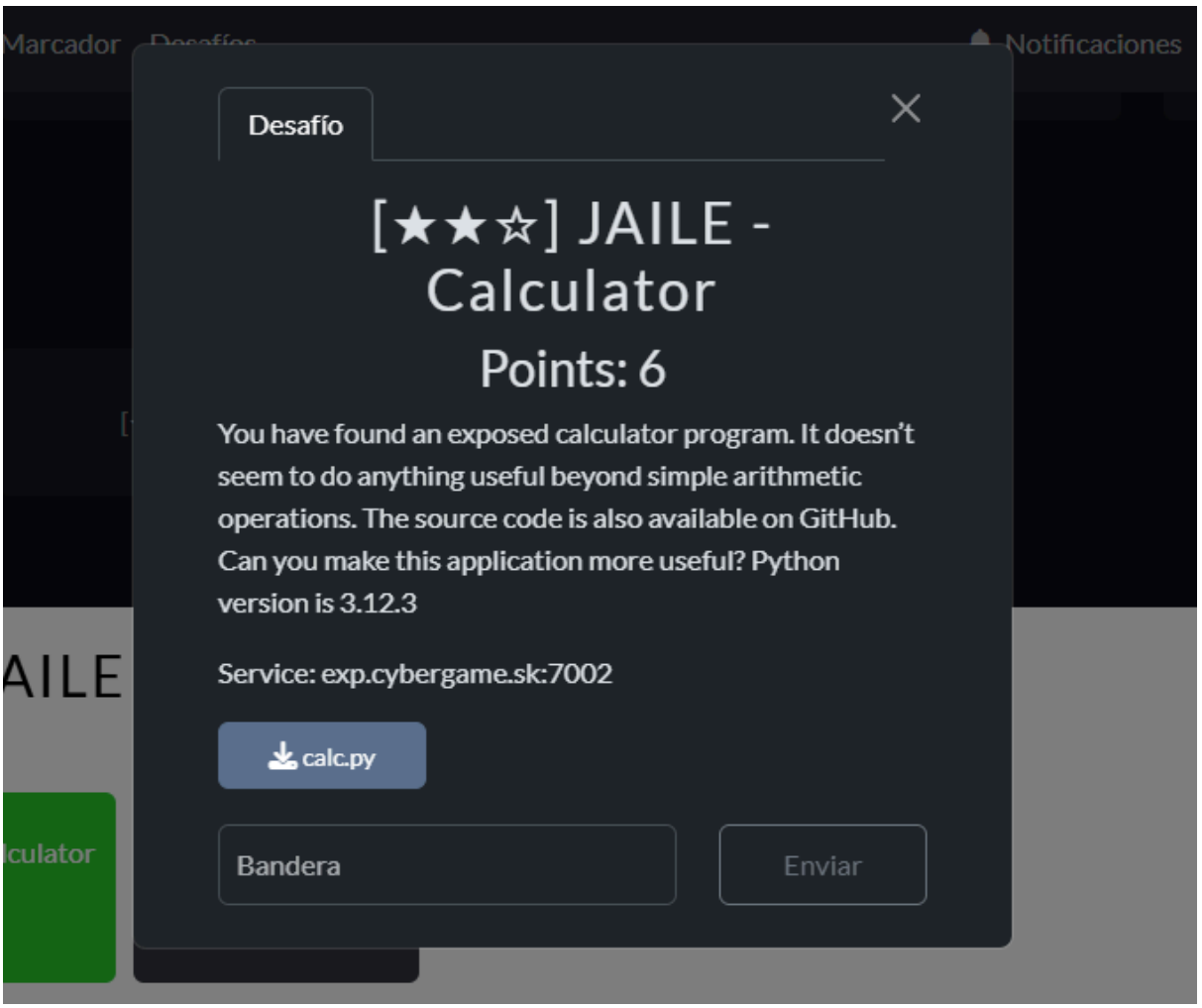




PWN Challenges

Reto: JAILE – Calculator

- **Categoría:** PWN
- Puntos: ★★☆ (6 pts)



Descripción del desafío Se nos proporcionó acceso a un servicio en `exp.cybergame.sk:7002` que se presentaba como una "calculadora". Este servicio recibía una línea de texto y, internamente, ejecutaba `exec('print('+text+')')`. La trampa estaba en los filtros: el servidor comprobaba que la entrada no contuviera las palabras `eval`, `exec`, `import`, `open`, `os`, `read`, `system`, `write`, ni comillas (simples o dobles). El objetivo era conseguir la flag, que asumíamos estaría en un archivo `flag.txt` o similar.

Debugging

Este reto, a pesar de su aparente simplicidad, planteó un interesante desafío debido a las estrictas restricciones impuestas por el servidor. El objetivo era lograr una inyección de código que permitiera leer el contenido de un archivo.

Intento 1: Análisis de la vulnerabilidad y los filtros iniciales. Lo primero que saltaba a la vista era el `exec('print('+text+')')`. Esto es una clásica inyección de código Python. Si no hubiera filtros, un simple `open('flag.txt').read()` sería suficiente. Sin embargo, la lista negra de palabras (`eval`, `exec`, `import`, `open`, `os`, `read`, `system`, `write`) y la prohibición de comillas eran el verdadero obstáculo.

Intento 2: Burlar la prohibición de comillas. Mi primer pensamiento fue cómo construir strings sin comillas. Rápidamente recordé que en Python se pueden construir bytes usando `bytes([...])` y luego decodificarlos a una cadena con `.decode()`. Por ejemplo, `bytes([102,108,97,103,46,116,120,116]).decode()` resultaría en `"flag.txt"`. Esto resolvía la parte de las comillas para nombres de archivos y métodos.

Intento 3: Intentando acceder a funciones prohibidas (`open`, `read`, `system`). Aquí es donde la cosa se ponía interesante. No podía escribir `open`, `read` u `os`.

- **¿`os.system`?** Imposible por el filtro de `os` y `system`.
- **¿`open`?** Si no puedo usar `open`, ¿cómo accedo a funciones de archivos? La clave estaba en que `open` es una función *built-in* de Python. Aquí recordé `__builtins__`. Este objeto global contiene todas las funciones y clases built-in. Si pudiera acceder a él, podría obtener `open` sin escribir la palabra literal.

Intento 4: Accediendo a funciones built-in sin la palabra clave. Python tiene el método `getattr()`, que permite obtener un atributo de un objeto por su nombre como una cadena. Si `__builtins__` no estaba filtrado (y afortunadamente no lo estaba), podía usar `getattr(__builtins__, "open")`. Combinando esto con la técnica de `bytes().decode()`, obtuve la forma de llamar a `open` de manera ofuscada: `getattr(__builtins__, bytes([111,112,101,110]).decode())` (donde `111,112,101,110` son los códigos ASCII para 'o', 'p', 'e', 'n').

Intento 5: Leyendo el archivo y el problema de `read`. Una vez que tenía el objeto file (`f = open('flag.txt')`), necesitaba leer su contenido. Un `f.read()` sería lo normal, pero `read` también estaba filtrado. ¡Doble problema! Sin embargo, la misma lógica de `getattr` se aplicaba aquí. `read` es un método del objeto file. Así que, podía usar `f.__getattribute__(bytes([114,101,97,100]).decode())()`. El `__getattribute__` es similar a `getattr` pero para atributos de instancias.

Intento 6: Ensamblando el payload y probando. Con todas las piezas en su lugar, el payload comenzó a tomar forma: `getattr(__builtins__,bytes([111,112,101,110]).decode())(bytes([102,108,97,103,46,116,120,116]).decode()).__getattribute__(bytes([114,101,97,100]).decode())()` Lo probé en un entorno local de Python con los mismos filtros simulados para asegurarme de que funcionaba antes de enviarlo al servidor remoto. La estructura `print(...)` ya la manejaba el `exec` del servidor, por lo que solo necesitaba enviar lo que iba dentro del `print()`.

Solución del enfoque: La clave residió en el uso inteligente de `__builtins__` y `getattr` (o `__getattribute__`) para sortear los filtros de palabras clave, y el uso de `bytes(...).decode()` para evadir la prohibición de comillas. Este enfoque permitió construir dinámicamente las llamadas a funciones y métodos sin que sus nombres literales aparecieran en el texto enviado al servidor.

Análisis final

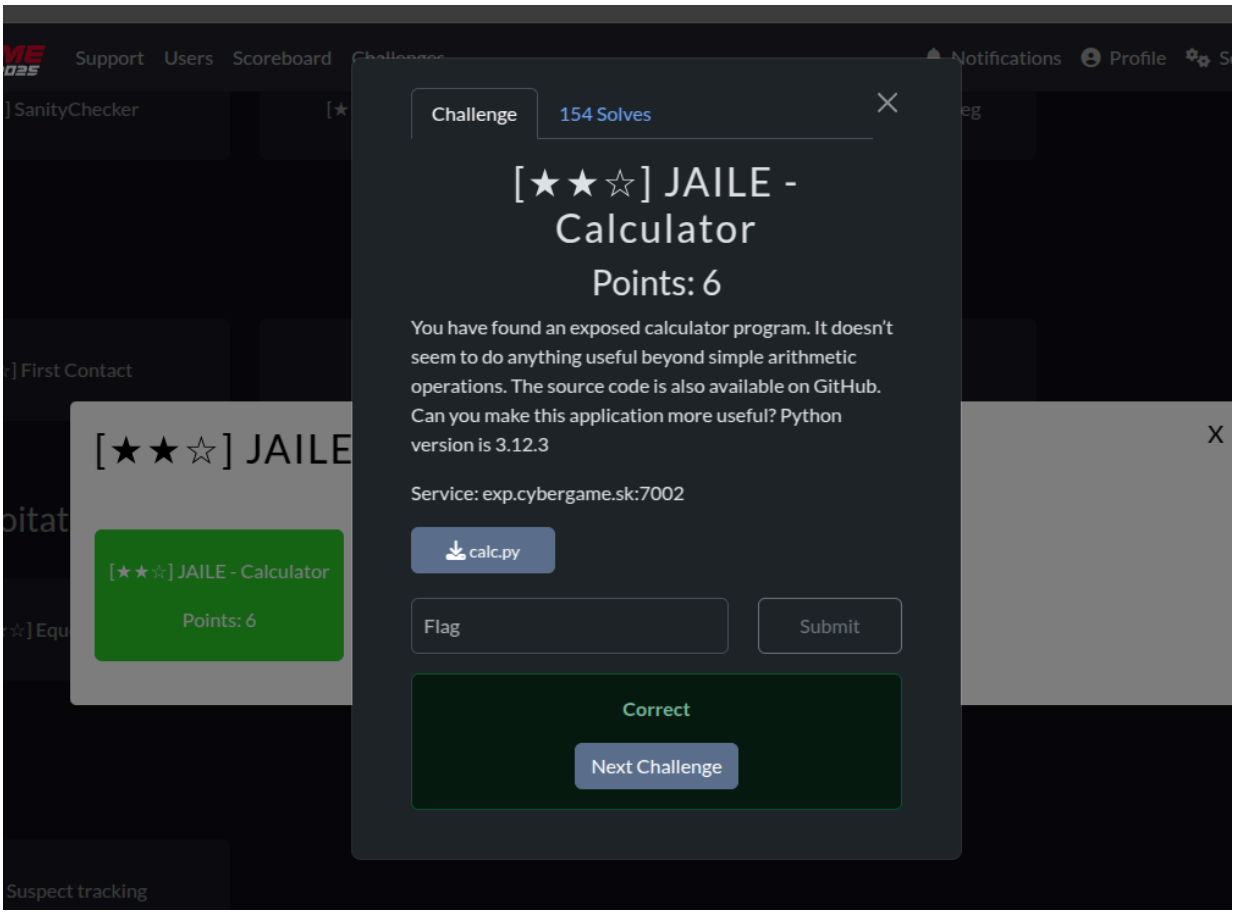
El payload final lo deje en el script que use para resolverlo:
Código: cybergames/pwn/jaile-calculator.py

Ejecución: Este payload se envía directamente al servicio `exp.cybergame.sk:7002`. El servidor lo recibe como `text` y lo ejecuta dentro de un `print()`. Escupe por la consola la flag

Resultado:

`SK-CERT{35c3p1ng_py7h0n_15_345y_745k}`

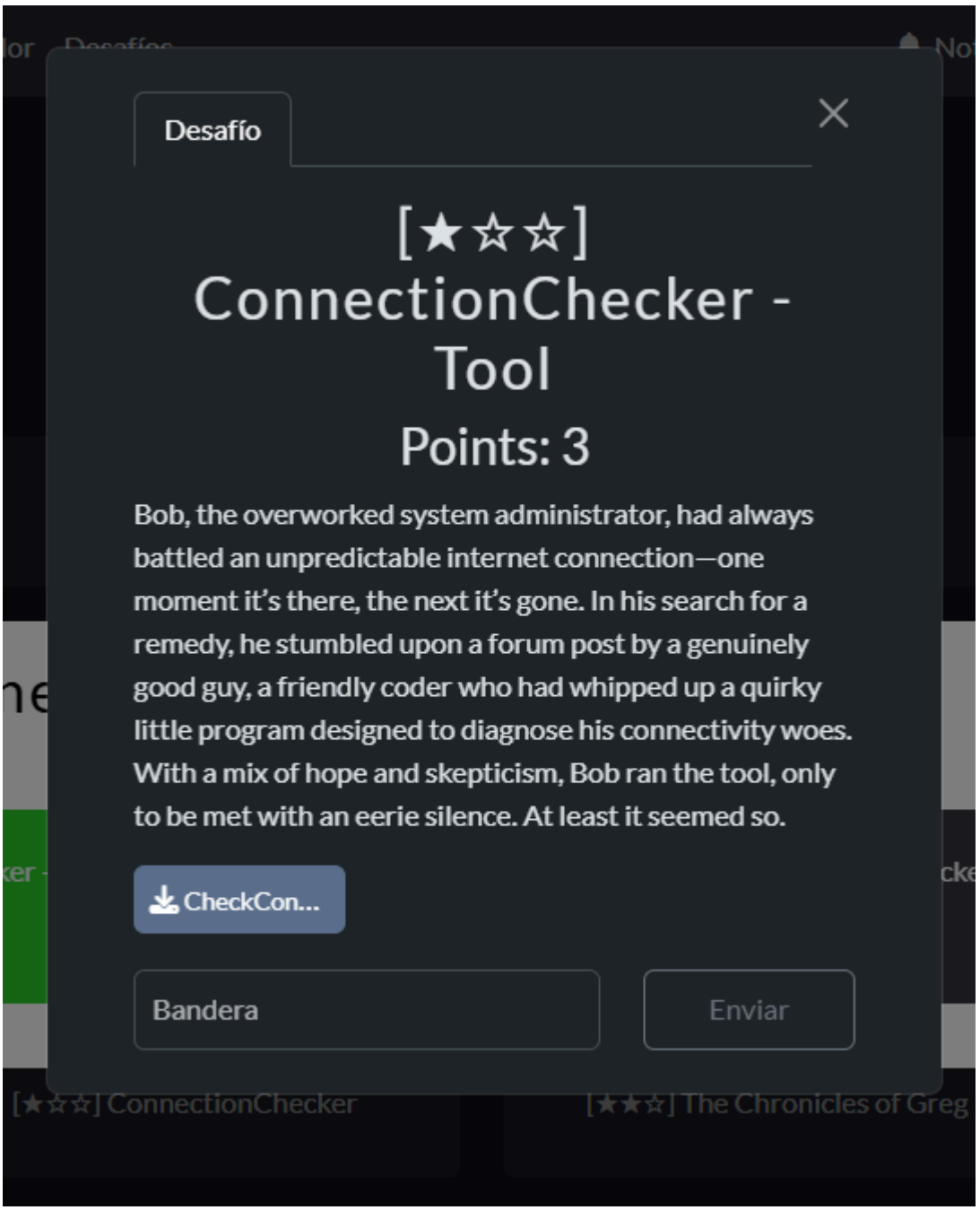
Captura:



Reverse Engineering Challenges

Reto: ConnectionChecker - Tool

- **Categoría:** RE
- Puntos: ★☆☆ (3 pts)



Descripción del desafío Nos encontramos con un archivo `CheckConnection.jar`, presentado como una herramienta de diagnóstico de conexión para Bob, el administrador de sistemas. El problema era que, al ejecutarlo, la aplicación arrancaba pero terminaba inmediatamente sin mostrar ningún tipo de salida o advertencia. La pista clave era que la flag se encontraba "escondida dentro del JAR". Nuestro objetivo era recuperarla.

Debugging

El punto de partida era claro: un archivo `.jar` que no hacía nada visible al ejecutarse. Esto inmediatamente sugirió que la lógica interesante estaba oculta y necesitaba ser extraída o investigada de alguna manera.

Intento 1: Ejecución y observación del comportamiento inicial. Mi primer paso fue, como siempre, intentar ejecutar el programa directamente para observar su comportamiento.

Bash
$ java -jar CheckConnection.jar

Tal como se describía, la aplicación se iniciaba y salía al instante, sin imprimir nada en la consola. Esto confirmaba que no era una aplicación interactiva o que su "salida" era su terminación abrupta.

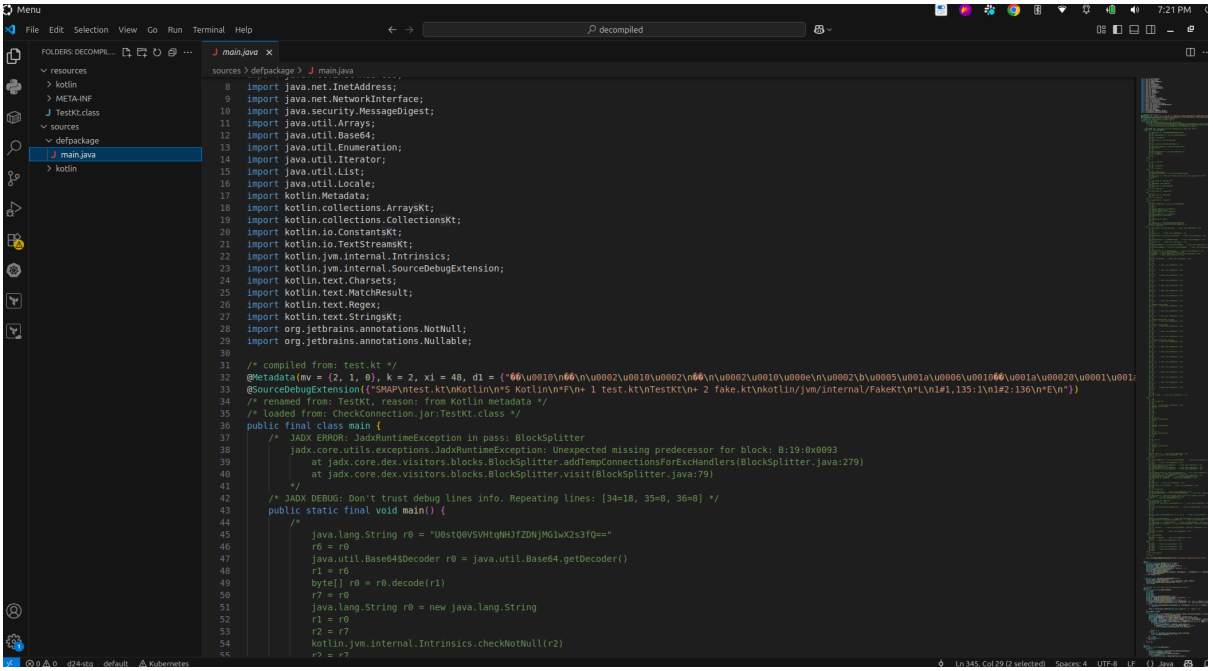
Intento 2: Búsqueda de strings sospechosas dentro del binario. Cuando se trata de archivos binarios o empaquetados, una primera técnica muy útil es buscar cadenas de texto ("strings") que puedan ser reveladoras. En este tipo de retos, las flags suelen tener un formato específico como `SK-CERT{...}`. Así que, decidí buscar patrones que indicaran la presencia de la flag.

Bash
$ strings CheckConnection.jar | grep -E 'U0stQ0VSVHt|SK-CERT'

¡Bingo! Esta búsqueda arrojó una cadena Base64 que inmediatamente capturó mi atención: `U0stQ0VSVHtqNHJfZDNjMG1wX2s3fQ==`. La combinación de `SK-CERT` (en Base64) con la codificación sugirió fuertemente que habíamos encontrado la flag, o al menos, una parte crucial de ella.

Intento 3: Decompilación para entender la lógica del programa. Aunque ya tenía una posible flag, siempre es una buena práctica entender por qué el programa se comporta de esa manera. Como es un JAR (Java Archive), sabía que contenía bytecode Java (o Kotlin, en este caso, que compila a JVM bytecode). Una herramienta de decompilación como `jadx` era la opción obvia.

Bash
$ jadx -d decompiled/ CheckConnection.jar



Esto generó un directorio `decompiled/` con el código fuente. Rápidamente identifiqué el paquete principal y un archivo de clase que parecía ser el punto de entrada, `TestKt.class` (renombrado a `main.java` por `jadx` para mayor claridad).

Intento 4: Inspección del código decompilado, buscando la cadena y la lógica de salida. Abrí el archivo `main.java` (o `TestKt.java` en algunos casos) y busqué la cadena Base64 que había encontrado. La encontré casi al inicio del método `main()`:

Java
```
String b64 = "U0stQ0VSVHtqNHJfZDNjMG1wX2s3fQ==";
byte[] raw = Base64.getDecoder().decode(b64);
String magic = new String(raw, UTF_8);
// Si no empieza con 'S' sale inmediatamente:
if (magic.length()==0 || magic.charAt(0)!='S')
    System.exit(0);
```

En fragmento de código encontre lo que buscaba: la cadena Base64 era, de hecho, la flag codificada. La aplicación la decodificaba y luego realizaba una verificación trivial (`magic.charAt(0)!='S'`). Si la primera letra de la cadena decodificada no era 'S', el programa simplemente salía (`System.exit(0)`). Dado que `SK-CERT{...}` comienza con 'S', esta verificación siempre pasaría si la decodificación resultaba en la flag esperada. Toda la lógica posterior (MD5, sockets TCP, etc.) nunca se ejecutaba porque el programa se diseñó para "fallar" silenciosamente si esta validación inicial no era satisfecha. Esto explicaba por qué la aplicación no mostraba nada y salía tan rápido.

Solución del enfoque: La flag no estaba "escondida" por un cifrado complejo o una ejecución condicional difícil de alcanzar, sino por una simple codificación Base64 y una verificación inicial que, aunque curiosa, no representaba un obstáculo real. El uso de

`strings` fue suficiente para la solución, y la decompilación sirvió para confirmar el "por qué" del comportamiento.

¿Cómo resolví finalmente? Una vez que tuve la cadena Base64, el resto fue trivial: simplemente la decodifiqué.

Análisis final

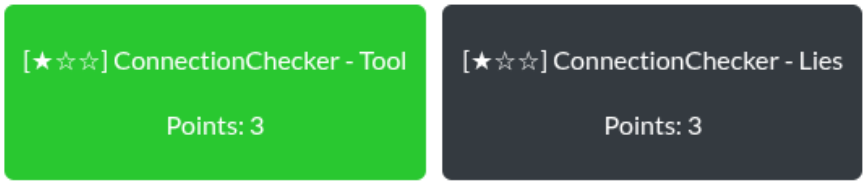
Decodificación de la Base64:

Bash
$ echo U0stQ0VSVHtqNHJfZDNjMG1wX2s3fQ== | base64 -d
SK-CERT{j4r_d3c0mp_k7}

Flag encontrada: `SK-CERT{j4r_d3c0mp_k7}`

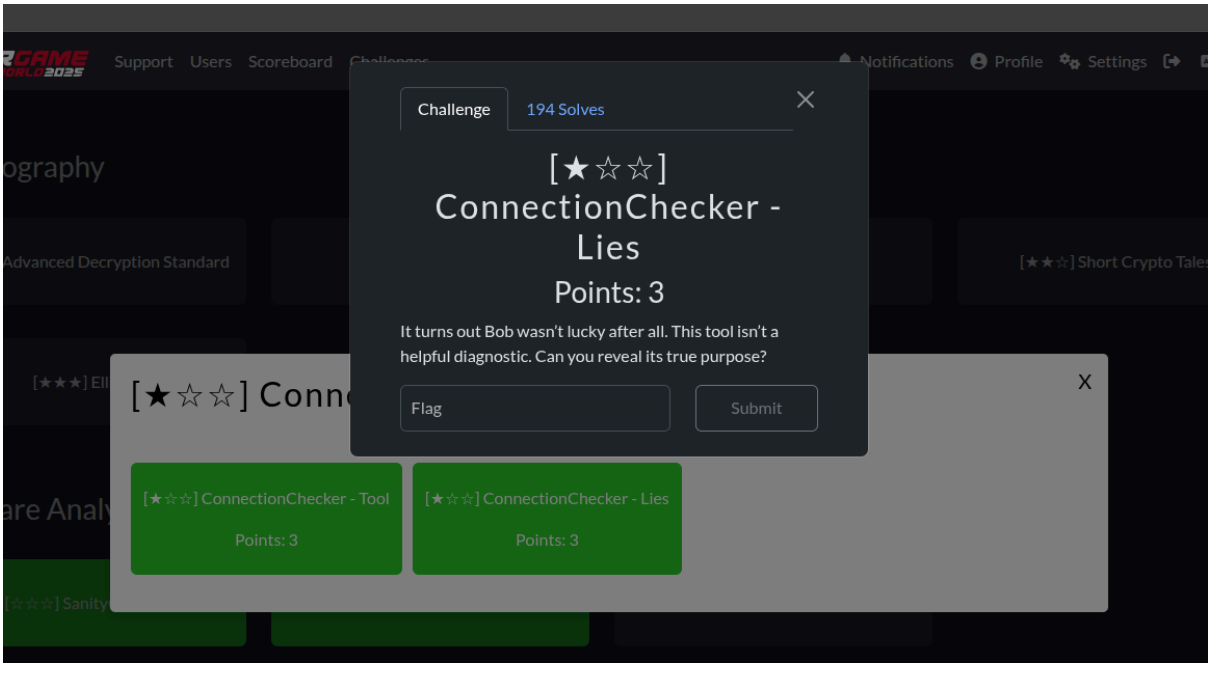
Captura:

[★☆☆] ConnectionChecker



Reto: ConnectionChecker – Lies

- **Categoría:** RE
- Puntos: ★☆☆ (3 pts)



Descripción del desafío La "herramienta" ConnectionChecker, que ya habíamos examinado superficialmente en un reto anterior, resultó ser mucho más que una simple utilidad de diagnóstico de red. Bob, el administrador, sospechó que no solo no ayudaba, sino que estaba ejecutando código oculto y comunicándose con un servidor remoto. Nuestro objetivo en este desafío era "revelar su verdadero propósito" y extraer una segunda flag, que se presumía estaba oculta dentro de su lógica más profunda. El archivo adjunto era, nuevamente, `CheckConnection.jar`.

Debugging

La base de este reto era el archivo `CheckConnection.jar`, el mismo del desafío anterior, pero con la clara indicación de que había más por descubrir que una simple cadena Base64 al inicio. El objetivo era ir más allá de la primera flag y desentrañar la verdadera funcionalidad del programa.

Intento 1: Re-confirmación de la primera capa y preparación para el análisis profundo. Dado que ya habíamos encontrado una flag en el reto anterior (`SK-CERT{j4r_d3c0mp_k7}`) a través de una simple decodificación Base64 en el `main()`, mi primer paso fue confirmar que esa lógica seguía allí.

Esto me recordó que la aplicación simplemente salía si la cadena decodificada no empezaba con 'S', lo cual explicaba su comportamiento "silencioso". Esta primera flag era solo una distracción o una verificación inicial; la verdadera "mentira" y el propósito oculto debían estar más adelante en el código. Para explorar esto, la decompilación con `jadx` era esencial.

Intento 2: Búsqueda de lógica posterior a la verificación inicial. Con el código decompilado (usando `jadx -d decompiled/ CheckConnection.jar`), comencé a examinar el `main()` y cualquier otra función llamada después de la verificación inicial de la primera flag. Mi objetivo era encontrar bloques de código que no se hubieran ejecutado antes.

Intento 3: Descubrimiento de la recolección de información local y la validación MD5. Rápidamente identifiqué secciones de código que intentaban obtener el SSID de la red Wi-Fi (usando comandos del sistema como `iwgetid -r` o `netsh` dependiendo del SO) y la dirección IP local IPv4. Esta información se concatenaba (`SSID|IP`) y luego se le calculaba un hash MD5. ¡Aquí estaba la primera señal de un comportamiento malicioso! Lo más interesante fue una validación:

Si el MD5 calculado coincide con el valor hardcodeado: de2ca7388ab6efb59a977505b9414ca2, la ejecución continúa.

Esto significaba que la parte más "sensible" del malware solo se activaría en un entorno específico. No era necesario forzar este MD5 para resolver el reto, pero era una pista crucial sobre el verdadero propósito del software.

Intento 4: Detección de la comunicación con el servidor remoto. Siguiendo la lógica condicional del MD5, descubrí un bloque de código que abría una conexión de socket a una dirección IP y puerto específicos: `195.168.112.4:7051`. ¡Esto era una clara señal de un backdoor o un comando y control (C2)!

Intento 5: Análisis de la construcción del payload y las operaciones bitwise. Dentro de la lógica de comunicación, encontré un arreglo de 25 bytes inicializado con valores enteros, tanto positivos como negativos. Lo crucial era que estos bytes se sometían a una serie de transformaciones bitwise dentro de un bucle:

- `xor` con el índice `i`
- `resta` de 10
- `negación`
- `suma` del índice `i`
- `rotación derecha` de 2 bits

El resultado de estas operaciones se convertía a UTF-8, se concatenaba con el MD5 (validado previamente) y se codificaba en Base64 antes de ser enviado al servidor. Esto parecía ser el corazón del "filtrado de datos" o de un mensaje de "check-in" con el C2. La "mentira" era que esta herramienta de diagnóstico en realidad estaba construyendo y enviando información.

Intento 6: Extracción de la segunda flag mediante simulación local de la transformación. La descripción del reto pedía "revelar su verdadero propósito" y extraer la "flag oculta en su lógica". No era necesario replicar el entorno exacto para que el MD5 coincidiera, ni establecer una conexión real al servidor. La clave estaba en que el arreglo inicial de 25 bytes y sus transformaciones eran estáticos y deterministas. Si podíamos replicar esas operaciones bitwise, obtendríamos el texto que se enviaría al servidor *antes* de ser codificado en Base64.

Decidí usar una shell de Python para simular esta transformación, ya que Python maneja bien las operaciones bitwise y la conversión de bytes. Transcribí el arreglo `init` y el bucle de operaciones bitwise tal como los vi en el código decompilado.

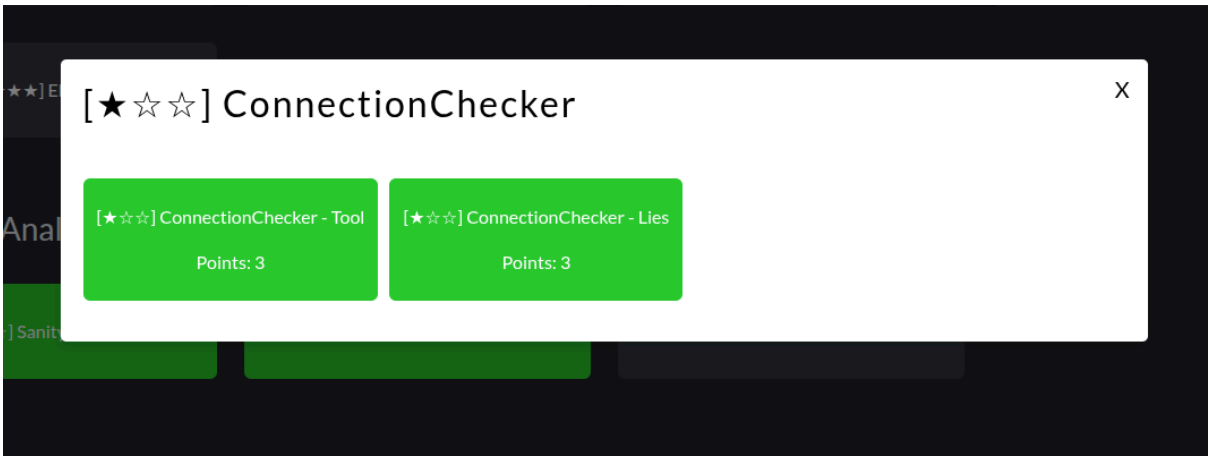
Y funcionó,El `print` final reveló la segunda flag: `SK-CERT{k3y_f0r_c253rv3r}`. Esta flag, que representaba el "valor transformado" enviado al servidor, era la verdadera revelación del propósito oculto de la herramienta: un keylogger o un mecanismo de exfiltración de información.

Solución del enfoque: El "verdadero propósito" no era el diagnóstico de red, sino la recolección de información del sistema y su comunicación con un servidor remoto. La segunda flag estaba incrustada en un payload codificado con operaciones bitwise, que solo se revelaba replicando esa transformación.

Resultado:

`SK-CERT{k3y_f0r_c253rv3r}`

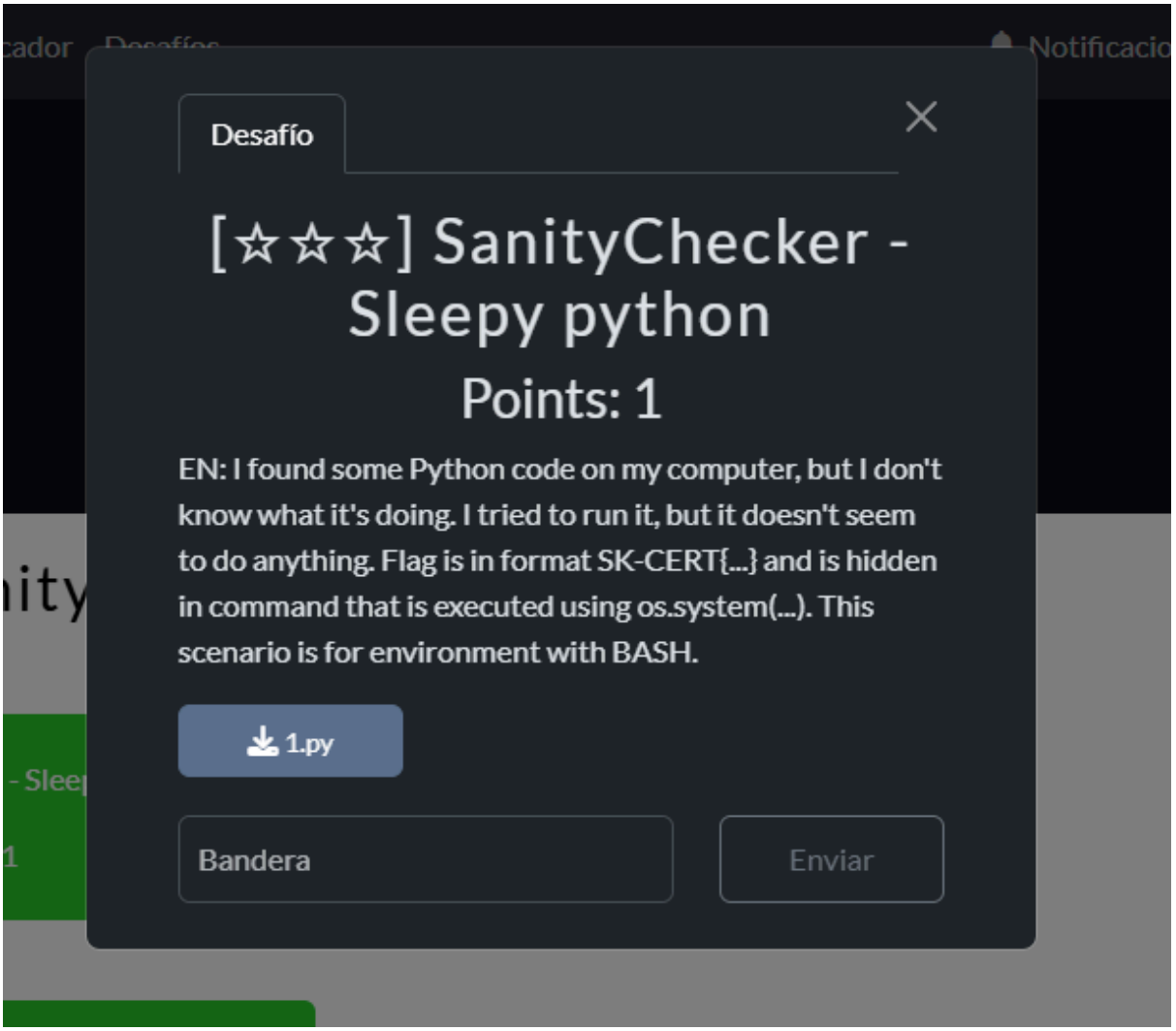
Captura:



A continuacion tambien explico en conjunto una serie de ejercicios de RE un tanto mas faciles:

Reto: SanityChecker - Sleepy Python

- **Categoría:** RE
- Puntos: ★☆☆ (1 pts)



Descripción del desafío Se nos entregó un script Python aparentemente inofensivo y ofuscado, que no mostraba ninguna actividad visible. La consigna, sin embargo, nos indicaba que la flag estaba oculta dentro de un comando que se ejecutaba internamente mediante `os.system(...)`, asumiendo un entorno con shell BASH. El objetivo era desentrañar el script y extraer la flag.

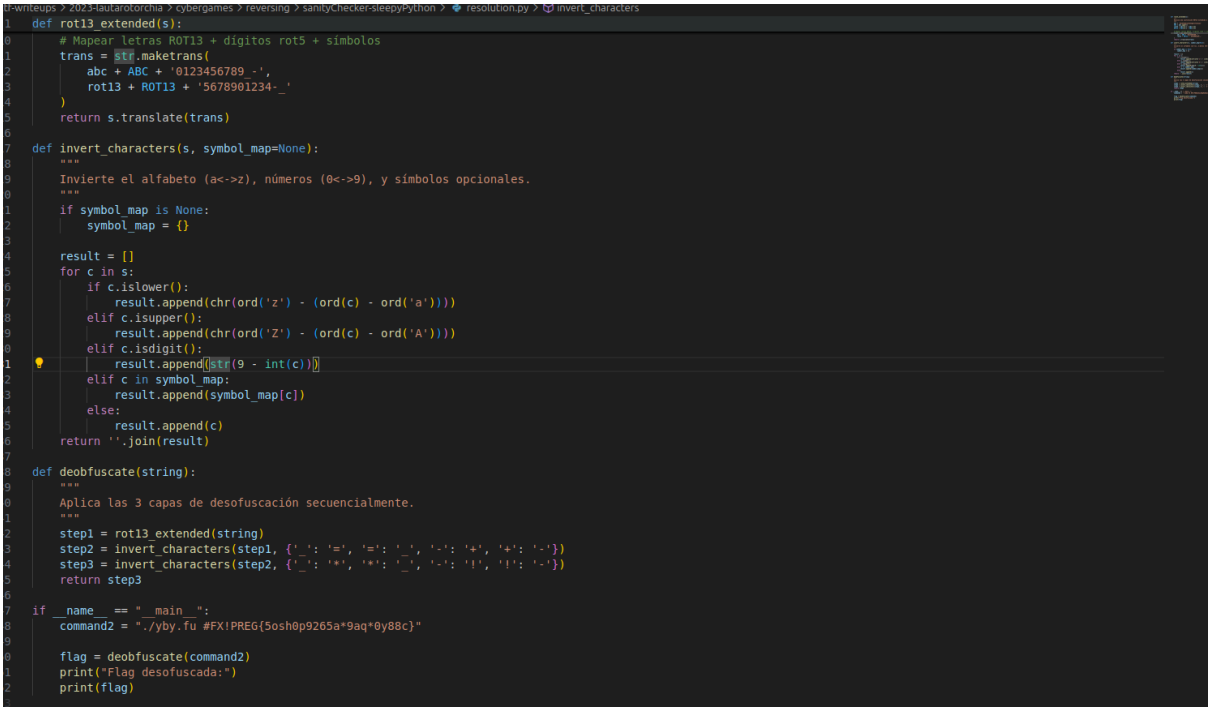
Debugging

Solución del enfoque: Al revisar el script, la presencia de múltiples funciones con nombres como `deobfuscate1`, `deobfuscate2`, `deobfuscate3` inmediatamente sugirió que la clave del reto era la **desofuscación de cadenas**. El hecho de que el script pareciera no hacer nada visible, sumado a la pista de un `os.system(...)`, apuntaba a que los comandos importantes estaban ocultos por estas transformaciones.

¿Cómo resolví finalmente? Mi enfoque fue directo: identifiqué las funciones de desofuscación y las apliqué a las variables que contenían cadenas ofuscadas (`command1`, `command2`, `data`). Al ejecutar estas funciones en el orden en que el script lo hacía, las cadenas se revelaron. Específicamente, al desofuscar la variable `command2`, el resultado fue el comando `./lol.sh` seguido de un comentario de shell: `#SK-CERT{0bfu5c4710n_4nd_5l33p}`. Aquí estaba la flag, incrustada claramente. La presencia de un largo `time.sleep()` en el script original simplemente confirmaba que el análisis estático era el camino, ya que el programa "dormiría" por un tiempo excesivamente largo.

Análisis final

Script: cybergames/reversing/sanityChecker-sleepyPython/resolution.py



Extracción de la flag: Aplicando las funciones de desofuscación al `command2` ofuscado, se reveló la siguiente cadena:

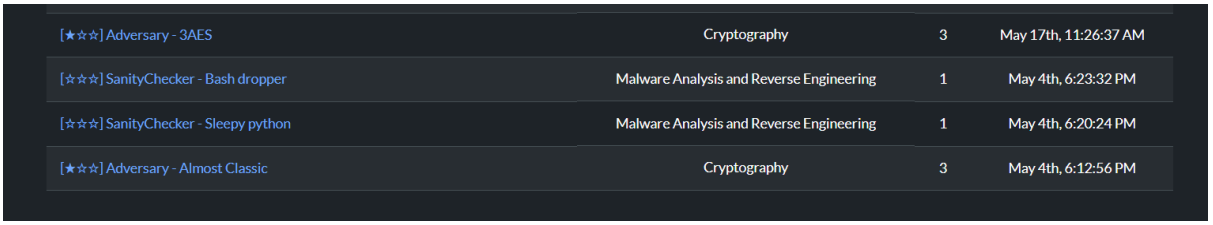
Bash

./lol.sh #SK-CERT{0bfu5c4710n_4nd_5l33p}

De ahí se extrajo la flag clara.

Flag obtenida: `SK-CERT{0bfu5c4710n_4nd_5l33p}`

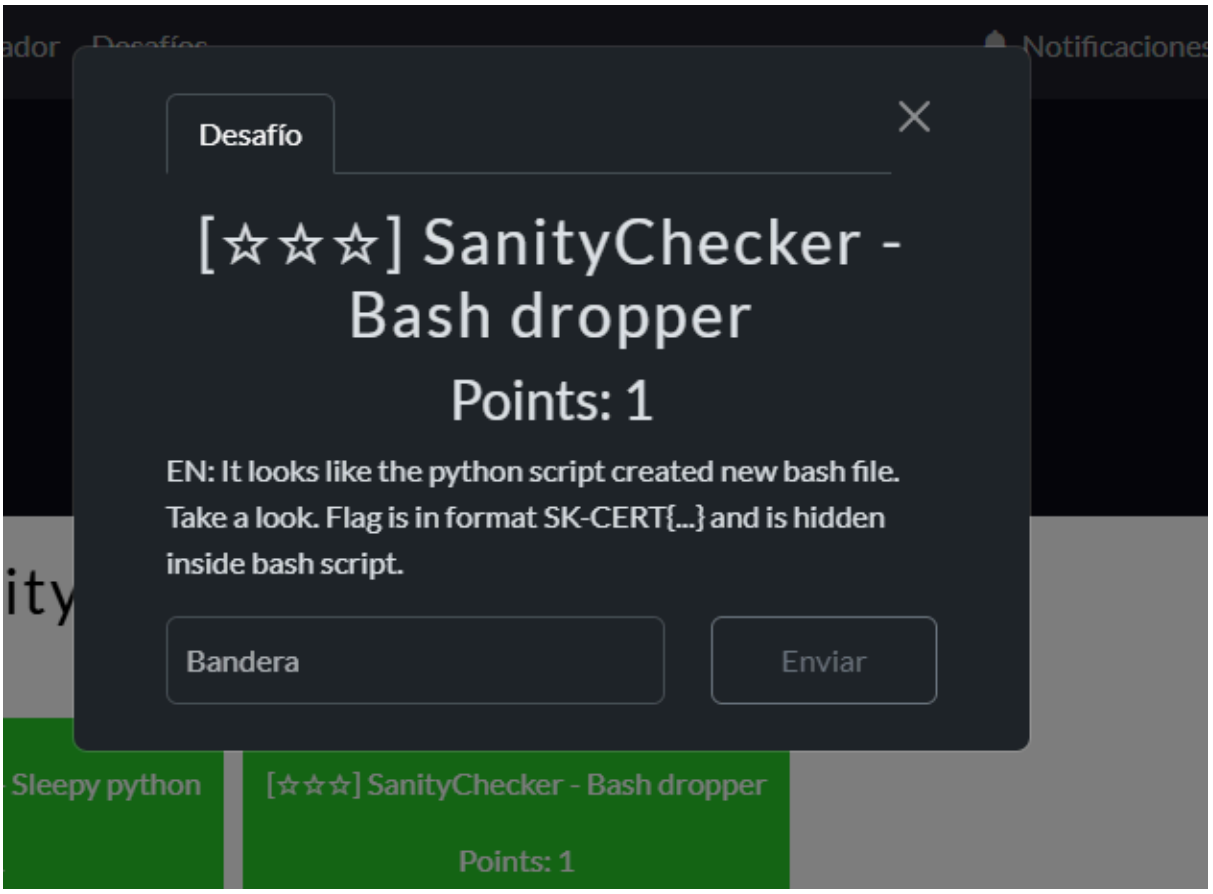
Captura:



[★☆☆] Adversary - 3AES	Cryptography	3	May 17th, 11:26:37 AM
[☆☆☆] SanityChecker - Bash dropper	Malware Analysis and Reverse Engineering	1	May 4th, 6:23:32 PM
[☆☆☆] SanityChecker - Sleepy python	Malware Analysis and Reverse Engineering	1	May 4th, 6:20:24 PM
[★☆☆] Adversary - Almost Classic	Cryptography	3	May 4th, 6:12:56 PM

Reto: SanityChecker - Bash Dropper

- **Categoría:** RE
- Puntos: ★☆☆ (1 pts)



Descripción del desafío Se nos entregó un script en Python que, al ejecutarse, generaba un archivo `lol.sh` con contenido ofuscado. La consigna indicaba claramente que la flag, con el formato `SK-CERT{...}`, estaba oculta dentro de este script bash generado. El objetivo era encontrar y extraer esa flag sin necesidad de ejecutar el script malicioso.

Debugging

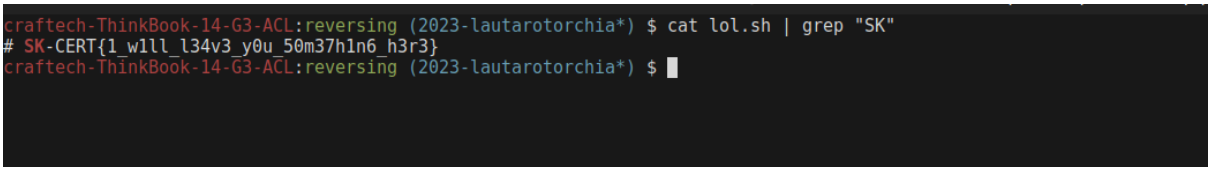
Solución del enfoque: La descripción del reto ya ofrecía una pista crucial: la flag estaba "oculta dentro de ese script bash". Dado que era un script, la primera acción lógica fue simplemente abrirlo y examinar su contenido.

¿Cómo resolví finalmente? Al abrir el archivo `lol.sh` en un editor de texto, noté que, a pesar de su contenido ofuscado y un bucle infinito que imprimía "You are hacked!", la flag estaba sorprendentemente ubicada en un **comentario al inicio del archivo** con el formato `# SK-CERT{...}`. No fue necesario ejecutar el script ni desofuscar nada; bastó con una simple inspección visual o el uso de `grep` para extraerla.

Análisis final

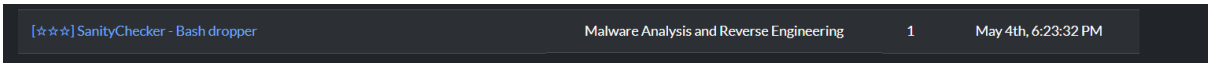
Resultado: Ejecutando un simple `grep` o abriendo el archivo `lol.sh`, la flag se encontraba directamente:

`SK-CERT{1_w1ll_l34v3_y0u_50m37h1n6_h3r3}`



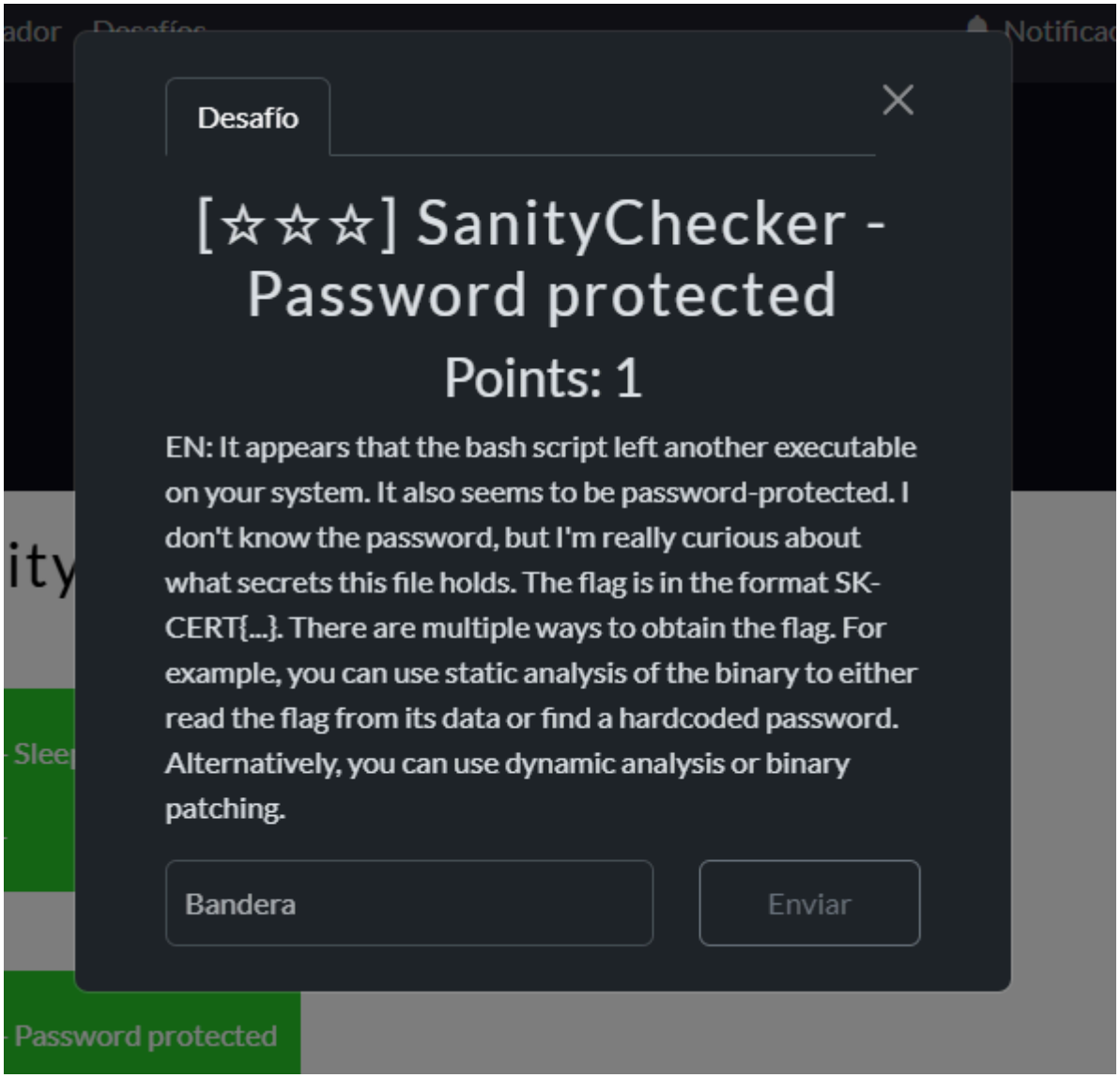
Conclusión: Este reto es un ejemplo clásico de un "dropper" sencillo: un script que genera otro archivo potencialmente malicioso. La clave para la solución fue entender que la flag no requería una desofuscación compleja, sino una inspección básica del archivo generado, destacando la importancia de revisar incluso los comentarios en los artefactos proporcionados.

Captura:



Reto: SanityChecker - Password Protected

- **Categoría:** RE
- Puntos: ★☆☆ (1 pts)



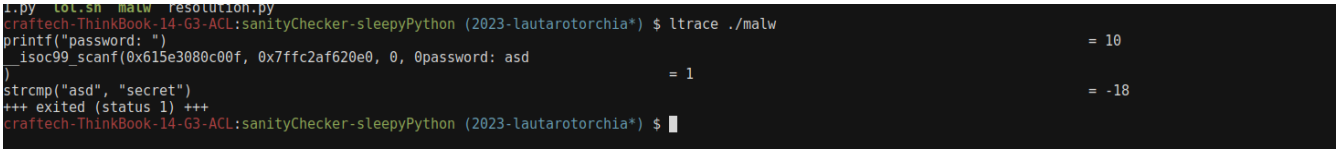
Descripción del desafío En este desafío, se nos entregó un binario ejecutable llamado `malw`, el cual, según la consigna, era generado por un script bash (`lol.sh`) – aunque para este reto, solo el binario era relevante. Al intentar ejecutar `malw`, este solicitaba una contraseña para continuar. Nuestro objetivo era encontrar la contraseña, ejecutar el binario exitosamente y obtener la flag oculta.

Debugging

Solución del enfoque: Ante un binario que pide una contraseña, la primera pregunta es "¿dónde está la contraseña?". Una herramienta fundamental en reverse engineering para binarios es `ltrace`, que permite rastrear las llamadas a librerías dinámicas. Esto es ideal para ver si se están comparando strings o pidiendo input.

¿Cómo resolví finalmente? Ejecuté el binario usando `ltrace`:

ltrace ./malw



La salida de `ltrace` fue reveladora. Pude observar que el binario imprimía "password: " y, lo crucial, mostraba una llamada a una función de comparación de cadenas (e.g., `strcmp`). En la salida de `ltrace`, la contraseña esperada, `"secret"`, aparecía explícitamente como uno de los argumentos de esa llamada a función.

Una vez que obtuve la contraseña (`secret`), simplemente ejecuté el binario e ingresé la clave cuando me la solicitó. Tras ingresar la contraseña correcta, el programa simuló una operación ("removed 20%", "removed 40%", etc.) y finalmente imprimió la flag.

Análisis final

Ejecución:

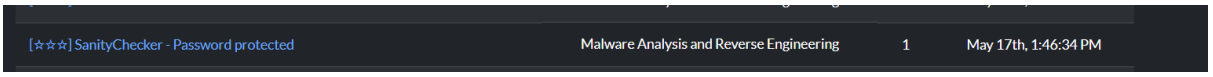
./malw

password: secret

Flag obtenida:

`SK-CERT{h4rdc0d3d_pl41n73x7}`

Captura:

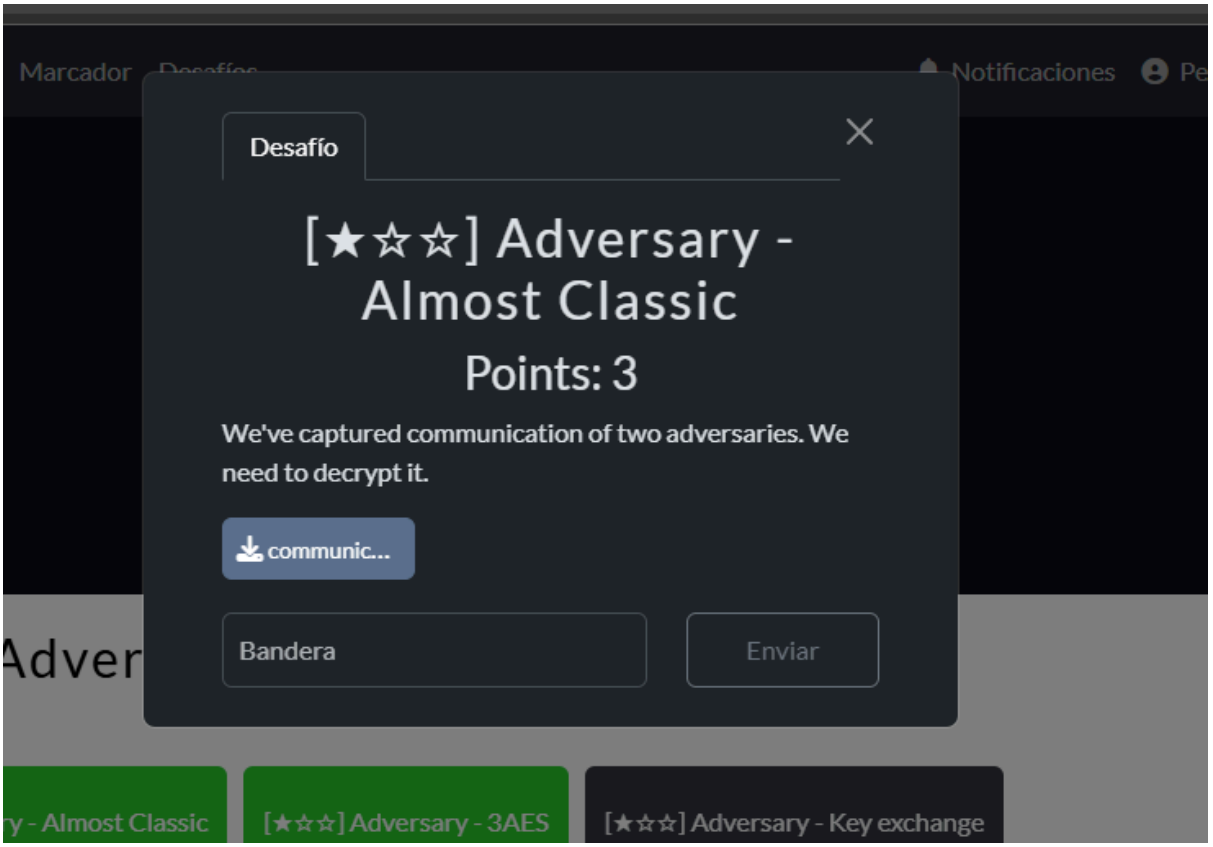


Crypto Challenges

Reto: Almost Classic

- **Categoría:** Cryptography
- **Puntos:** ★☆☆ (3 pts)

Descripción del desafío



Se entrega un archivo `communication.txt` cifrado. Cada línea está prefijada por `X:` o `Y:` y, al descifrar, obtenemos textos en claro, siendo la última línea la flag en formato `SK-CERT{...}`.

Debugging

- **Primera idea:** Usar CyberChef para probar sustituciones ROT13 y otras rotaciones comunes; no obtuve resultados coherentes.
- **Segunda idea:** Utilizar `str.translate()` en Python, pero olvidé mapear minúsculas y algunas letras no se traducían.

- **Solución del enfoque:** Me di cuenta de que Atbash era el cifrado correcto; añadí lógica para dejar intactos los caracteres no alfabéticos y la función tradujo todas las líneas sin errores.

¿Cómo resolví finalmente?

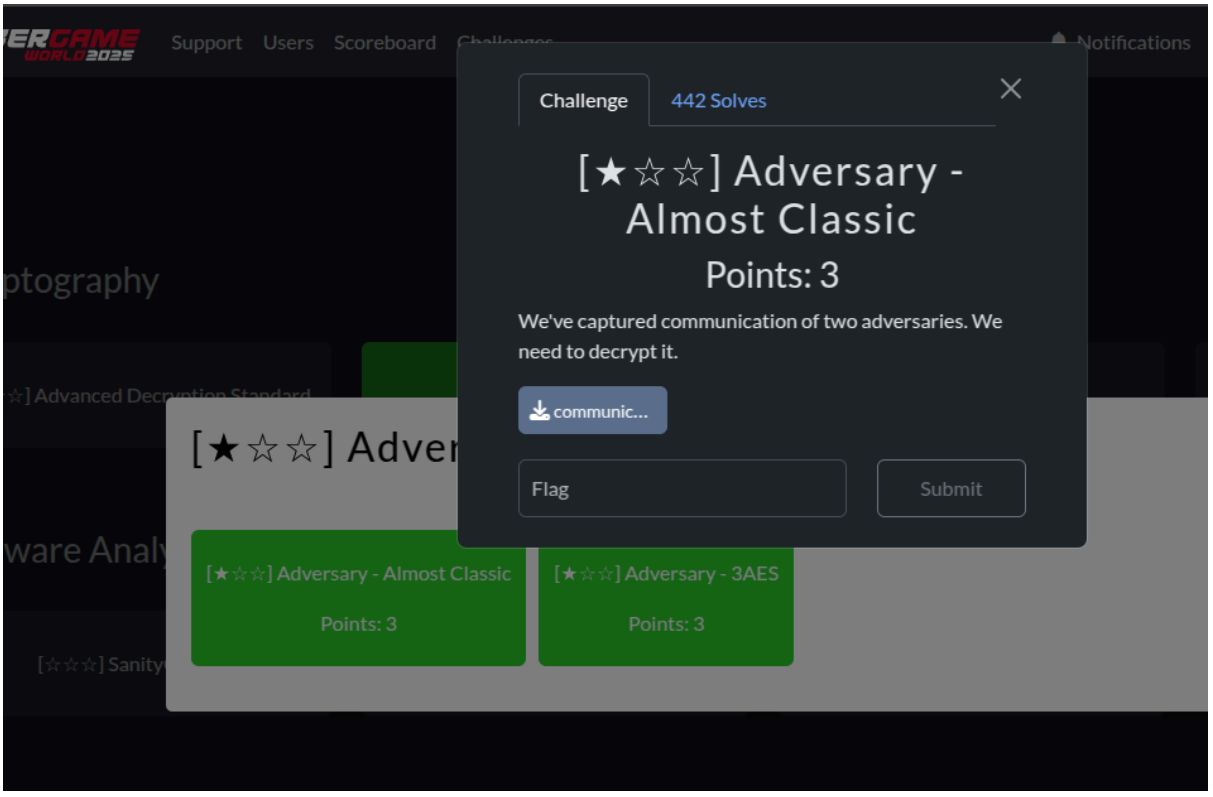
Mientras comprobaba sustituciones comunes, noté que algunas letras parecían mapearse a su opuesto exacto en el alfabeto (por ejemplo A→Z, B→Y). Esa simetría me recordó el cifrado **Atbash**, así que ajusté mi función para intercambiar cada letra por su opuesta y dejar intactos los caracteres no alfabéticos. Al ejecutar el script completo, todas las líneas se volvieron legibles.

Análisis final

- **Código:** cybergames/crypto/adversary/almostClassic/resolution.py
- **Resultado:**
  `SK-CERT{have_you_ever_heard_about_a_block_cipher???}`

```
craftech-ThinkBook-14-G3-ACL:almostClassic (2023-lautarotorchia*) $ python3 resolution.py
X: Lets coordinate the drop point.
Y: Agreed but we have to be careful. Our cryptographers warned us about this method. They say it wont hold for long.
X: We dont have time to set up anything better. Come to the usual place in the Pentagon, Stavbarska 42. The bartender will give you the package.
Y: Fine but if we get compromised because of the cipher you have to double your stakes. SK-CERT{have_you_ever_heard_about_a_block_cipher???}
craftech-ThinkBook-14-G3-ACL:almostClassic (2023-lautarotorchia*) $
```

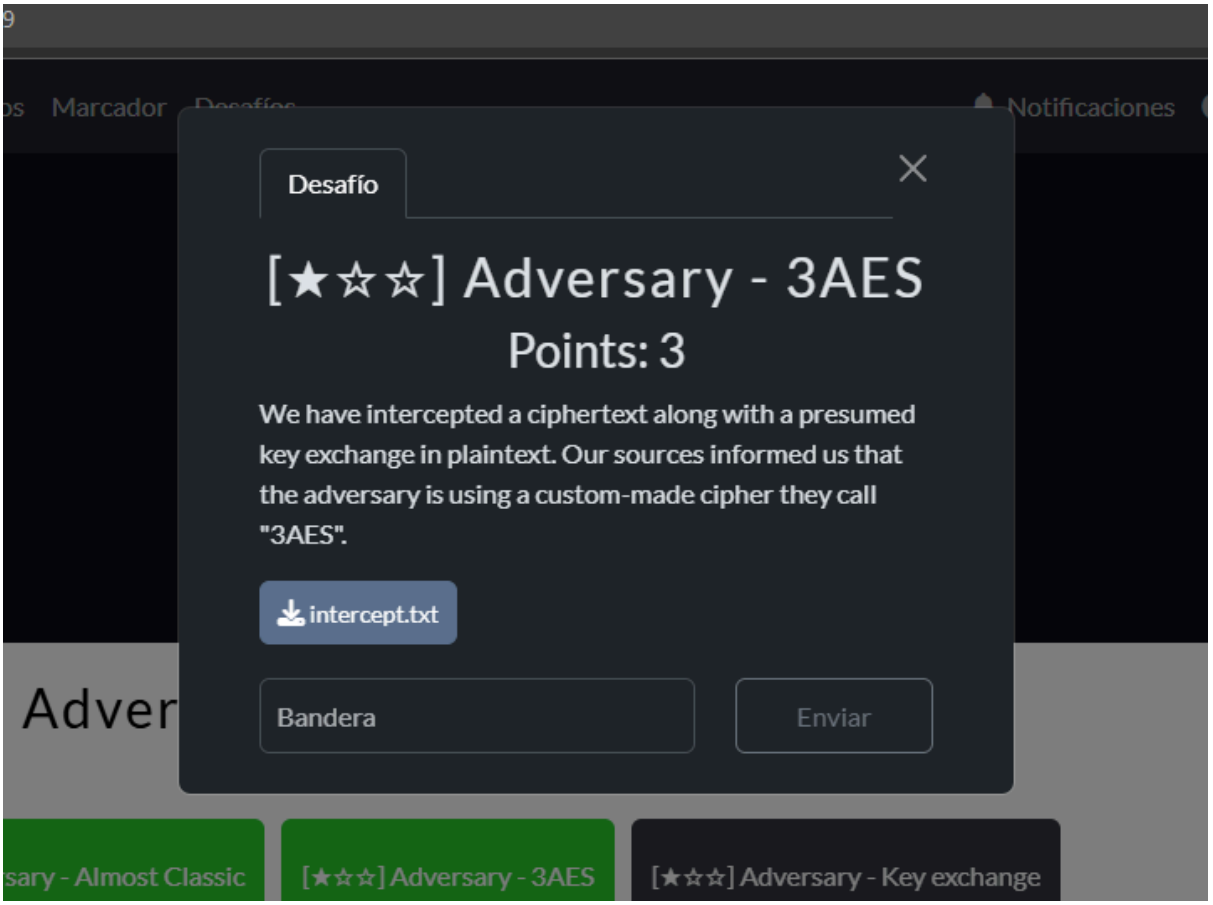
Captura:



Reto: Adversary – 3AES

- **Categoría:** Cryptography
- Puntos: ★☆☆ (3 pts)

Descripción del desafío



Se proporcionó un archivo `intercept.txt` que contiene tres claves AES-256 (`K1`, `K2`, `K3`) y varios blobs cifrados en Base64 etiquetados como `X` y `Y`. El esquema es un cifrado propietario “3AES” de triple capa.

Debugging

- **Primera idea**: Extraer y decodificar las claves y blobs en Base64, probar desencriptar solo con AES-ECB; no obtuve texto legible.
- **Segunda idea**: Probar modos AES-CBC y combinaciones de EEE (Encrypt–Encrypt–Encrypt); seguía sin coincidir.
- **Tercera idea**: Investigar esquemas de Triple DES y recordar EDE (Decrypt–Encrypt–Decrypt). Probé aplicar EDE con AES-ECB y funcionó para los primeros blobs.

¿Cómo resolví finalmente?

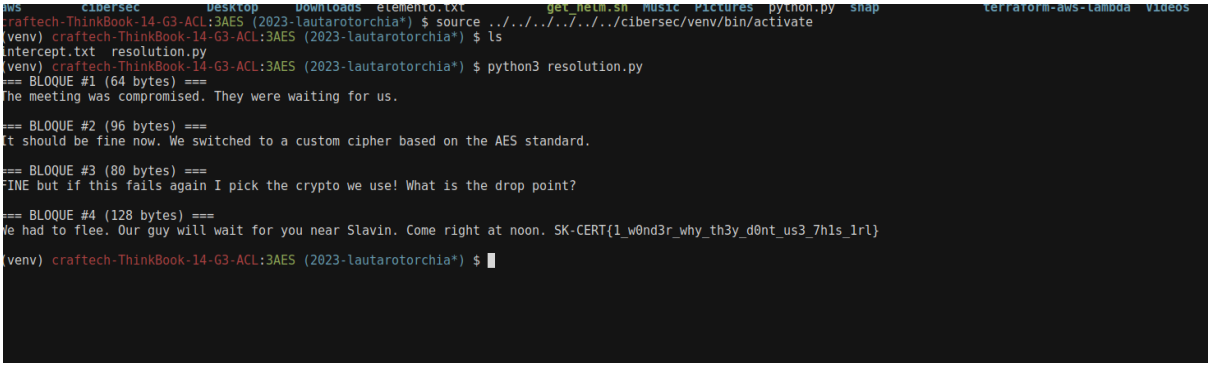
Tras identificar que no había IV y que el único esquema consistente era EDE en modo AES-ECB, definí el proceso en tres pasos – desencriptar con K3, cifrar con K2, y desencriptar con K1 – retirando padding PKCS#7 al final. Al concatenar estos pasos para cada blob, el cuarto bloque reveló la flag.

Análisis final

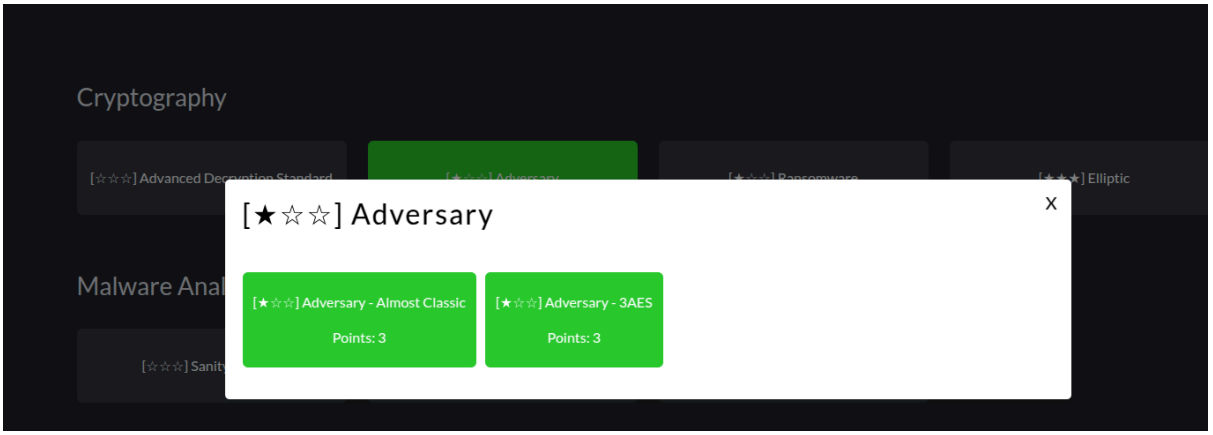
- **Código:** cybergames/crypto/adversary/3AES/resolution.py

Ejecución:

`python3 resolution.py`

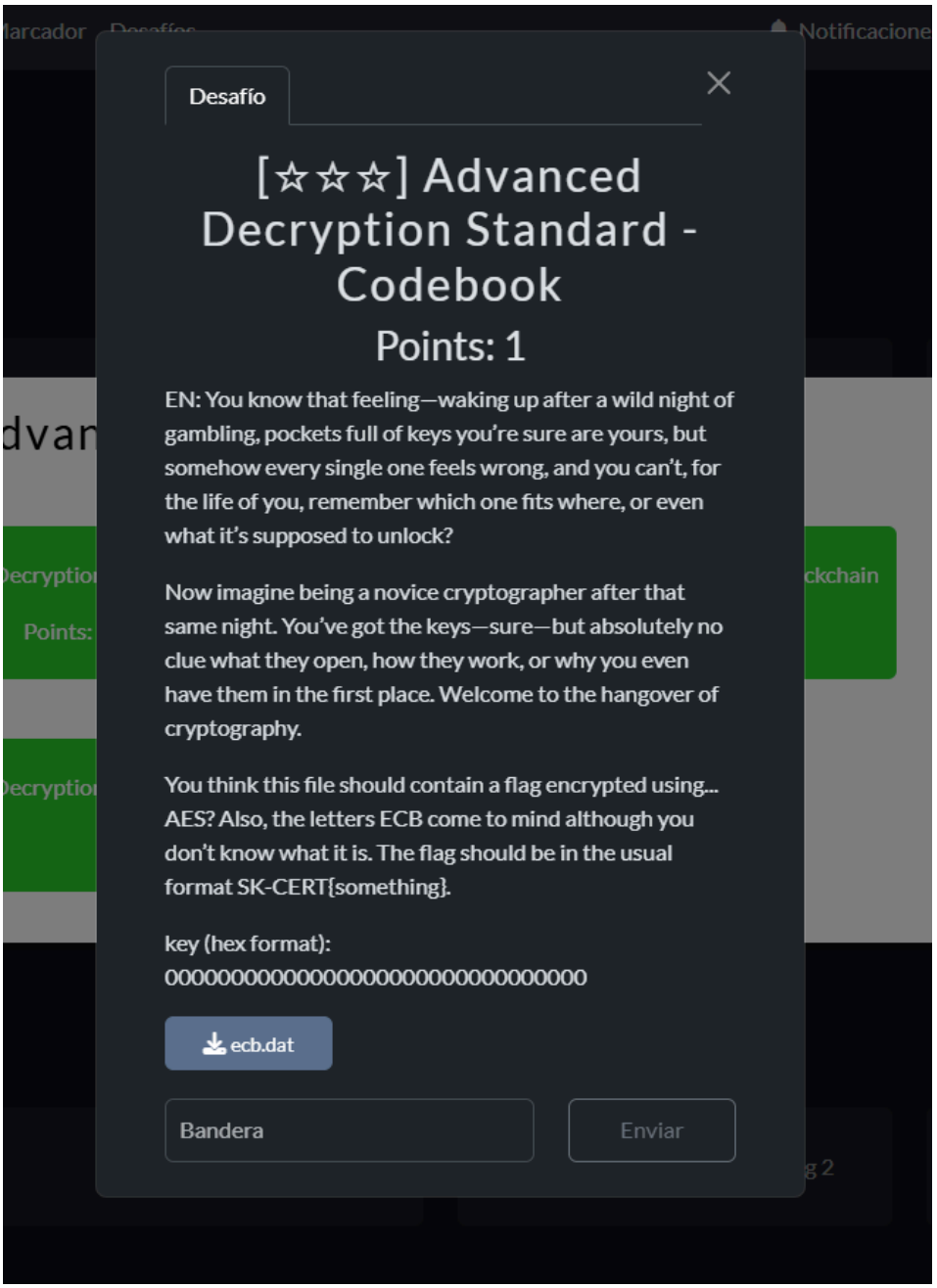


- **Resultado:**
  `SK-CERT{1_w0nd3r_why_th3y_d0nt_us3_7h1s_1rl}`



Reto: Advanced Decryption Standard - Codebook

- **Categoría:** Cryptography
- Puntos: ★☆☆ (1 pts)



Descripción del desafío

Se nos proporcionó un archivo, `ecb.dat`, que contenía un texto cifrado. La descripción indicaba claramente que el cifrado era **AES-128 en modo ECB** y, crucialmente, que la clave era de 16 bytes con valor cero (`00000000000000000000000000000000`). Nuestro objetivo era descifrar el contenido para revelar la flag.

Debugging

Solución del enfoque: La información proporcionada en la descripción del reto fue clave desde el principio. Al saber que se trataba de **AES-128 en modo ECB** con una **clave de ceros**, el camino estaba bastante claro. El único detalle a tener en cuenta para el descifrado fue el manejo del **padding PKCS#7**, un estándar común en cifrados por bloques.

¿Cómo resolví finalmente? La solución fue directa al usar la librería **PyCryptodome**. Cargué el archivo `ecb.dat`, configuré el cifrador AES con la clave de ceros en modo ECB, y realicé el descifrado. El único paso adicional necesario fue implementar la lógica para eliminar el padding PKCS#7, leyendo el último byte del texto descifrado para determinar la longitud del relleno y truncando los bytes correspondientes. Esto me permitió obtener el texto plano legible y, por ende, la flag.

Análisis final

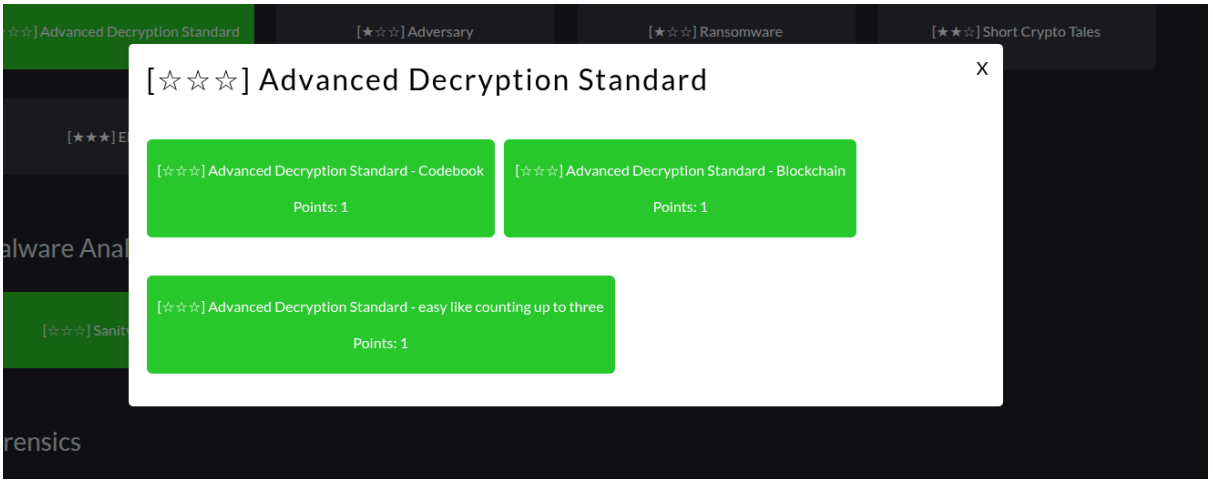
- **Código:** cybergames/crypto/standardDecryption/codebook/resolution.py

```
(venv) craftech-ThinkBook-14-G3-ACL:codebook (2023-lautarotorchia*) $ python3 resolution.py
SK-CERT{f1r57_15_3cb}
(venv) craftech-ThinkBook-14-G3-ACL:codebook (2023-lautarotorchia*) $
```

Resultado:

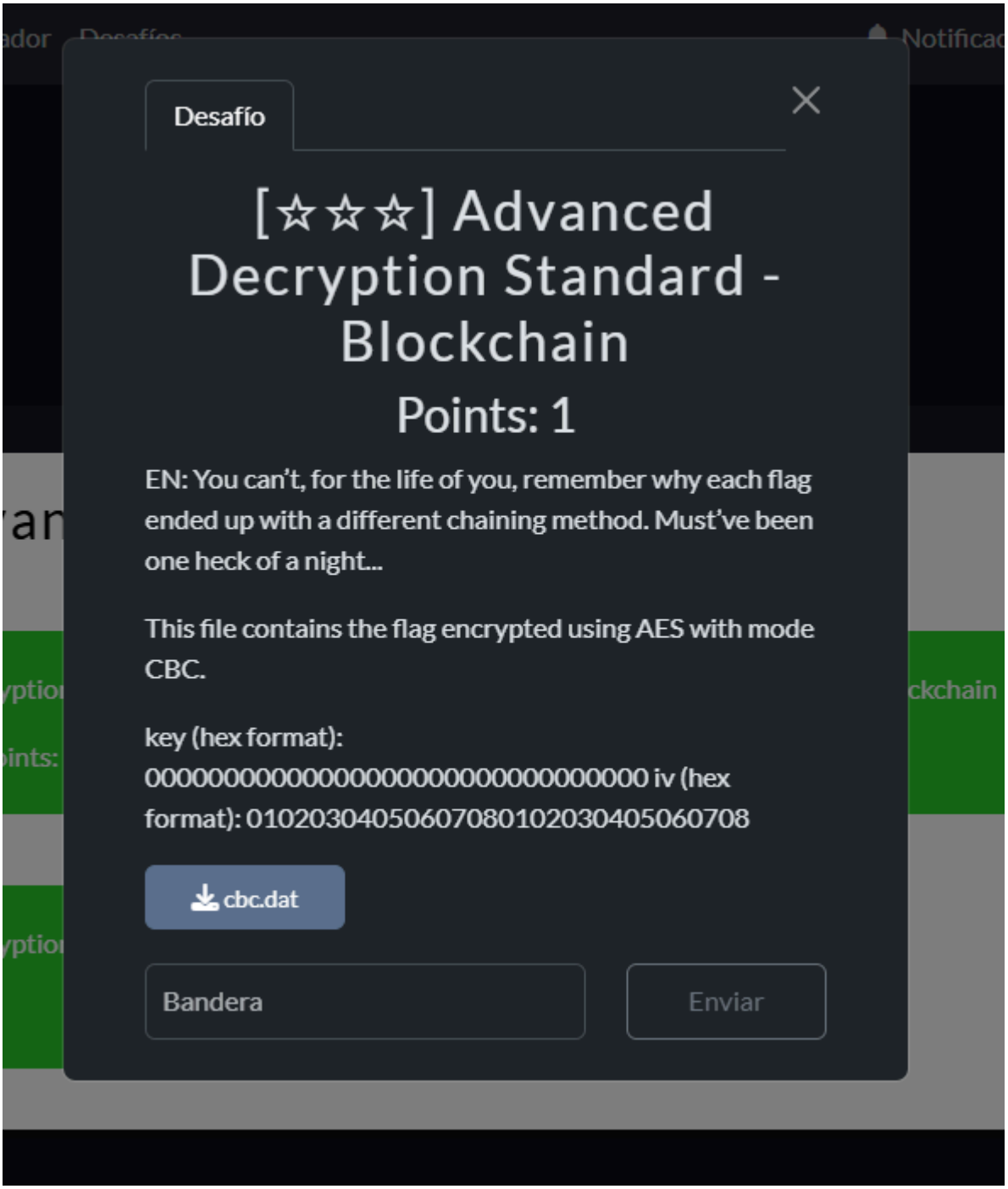
`SK-CERT{f1r57_15_3cb}`

Captura:



Reto: Advanced Decryption Standard – Blockchain

- **Categoría:** Cryptography
- Puntos: ★☆☆ (1 pts)



Descripción del desafío Se nos entrega un archivo `cbc.dat` que contiene la flag cifrada con AES en modo CBC. La clave y el vector de inicialización (IV) se proporcionan en hexadecimal:

- Clave (hex): `00000000000000000000000000000000`

- IV (hex): `01020304050607080102030405060708`

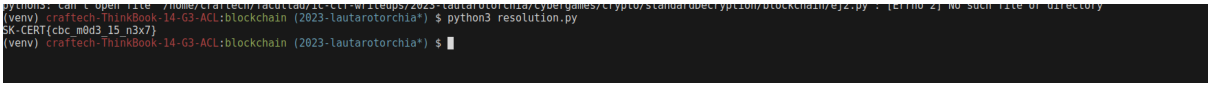
Nuestro objetivo es recuperar el contenido claro (la flag en formato `SK-CERT{...}`) a partir de esos parámetros.

Debugging La resolución de este desafío se centra en la correcta implementación del descifrado AES en modo CBC, teniendo en cuenta el vector de inicialización (IV) y el padding PKCS#7.

¿Cómo resolví finalmente? El script utiliza la librería `PyCryptodome` para realizar el descifrado. Primero, se convierten la clave y el IV de hexadecimal a bytes. Luego, se lee el contenido del archivo `cbc.dat`. Se instancia el cifrador AES en modo CBC con la clave y el IV proporcionados. Tras el descifrado, se elimina el padding PKCS#7, verificando que los últimos bytes coincidan con la longitud del relleno. Finalmente, se decodifica el resultado para obtener la flag.

Análisis final

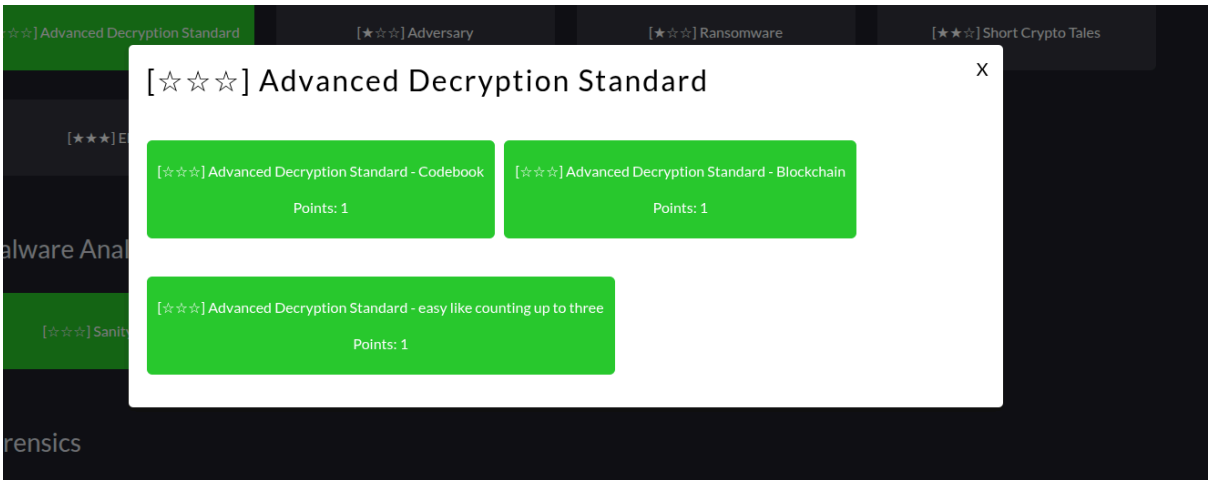
- **Código:** cybergames/crypto/standardDecryption/blockchain/resolution.py



Resultado:

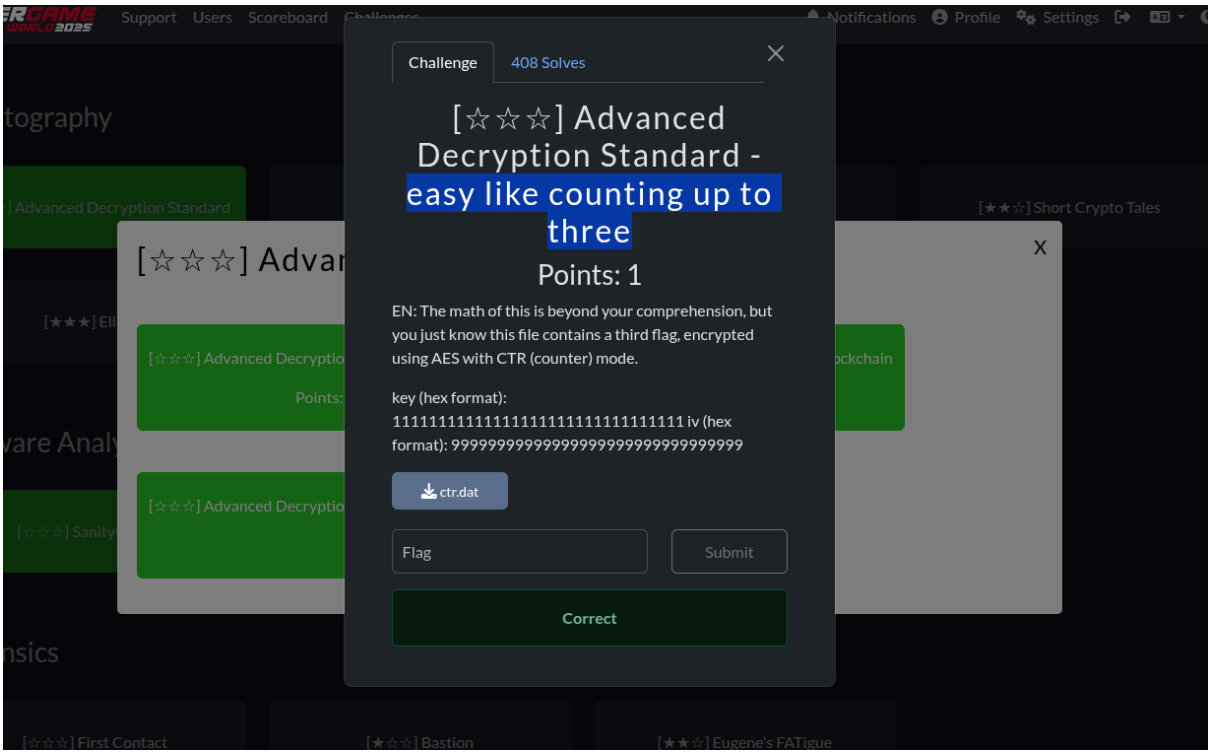
`SK-CERT{cbc_m0d3_15_n3x7}`

Captura:



Reto: Advanced Decryption Standard – easy like counting up to three

- **Categoría:** Cryptography
- Puntos: ★☆☆ (1 pts)



Descripción del desafío Se nos proporcionó un archivo, `ctr.dat`, que contenía una flag cifrada. La descripción detallaba el método de cifrado: **AES en modo CTR (Counter)**. Además, se nos dieron los parámetros clave para el descifrado:

- Clave (hex): `11111111111111111111111111111111`
- IV (hex): `99999999999999999999999999999999`

Nuestro objetivo era descifrar el contenido del archivo usando la clave y el IV provistos para obtener la flag en formato `SK-CERT{...}`.

Debugging

Intento 1: Reconocimiento del modo de cifrado y sus peculiaridades. La información clave aquí era **AES en modo CTR**. A diferencia de ECB y CBC, el modo CTR opera como un "stream cipher" generando un keystream (flujo de clave) que se XORiza con el texto plano. Esto significa que no utiliza padding, lo que simplifica un poco el proceso final de descifrado, ya que no hay que preocuparse por eliminar el relleno. El "IV" en CTR se comporta como un **contador inicial**.

Solución del enfoque: La comprensión de que el modo CTR no requiere manejo de padding y la correcta implementación del objeto contador a partir del IV fueron los puntos clave. Una vez que se configuró el contador adecuadamente, el resto del proceso de descifrado fue directo.

¿Cómo resolví finalmente? La solución implicó los siguientes pasos utilizando `PyCryptodome`:

1. **Convertir la clave y el IV** de sus cadenas hexadecimales a objetos `bytes`.
2. **Construir el objeto contador**: Para ello, convertí el `iv` (bytes) a un entero usando `int.from_bytes(iv, byteorder='big')` y luego creé una instancia de `Counter.new(128, initial_value=iv_int)`. El `128` se refiere al tamaño del bloque de AES en bits.
3. **Inicializar el cifrador AES**: Se creó el objeto cifrador `AES.new` especificando la clave, el modo `AES.MODE_CTR` y, crucialmente, pasando el objeto `ctr` creado en el paso anterior al parámetro `counter`.
4. **Leer el archivo cifrado y descifrar**: Se leyó el contenido binario de `ctr.dat` y se pasó al método `cipher.decrypt()`.
5. **Decodificar y mostrar la flag**: Dado que el modo CTR no añade padding, el resultado del descifrado (`plaintext`) pudo ser directamente decodificado a UTF-8 y la flag se imprimió en la consola.

Análisis final

Código: `solve_ctr.py`

```
(venv) craftech-ThinkBook-14-G3-ACL:countingToThree (2023-lautarotorchia*) $ python3 resolution.py
SK-CERT{4nd_7h3_l457_15_c7r}
(venv) craftech-ThinkBook-14-G3-ACL:countingToThree (2023-lautarotorchia*) $
```

Análisis final

- **Código:** cybergames/crypto/standardDecryption/countingToThree/resolution.py

Captura: