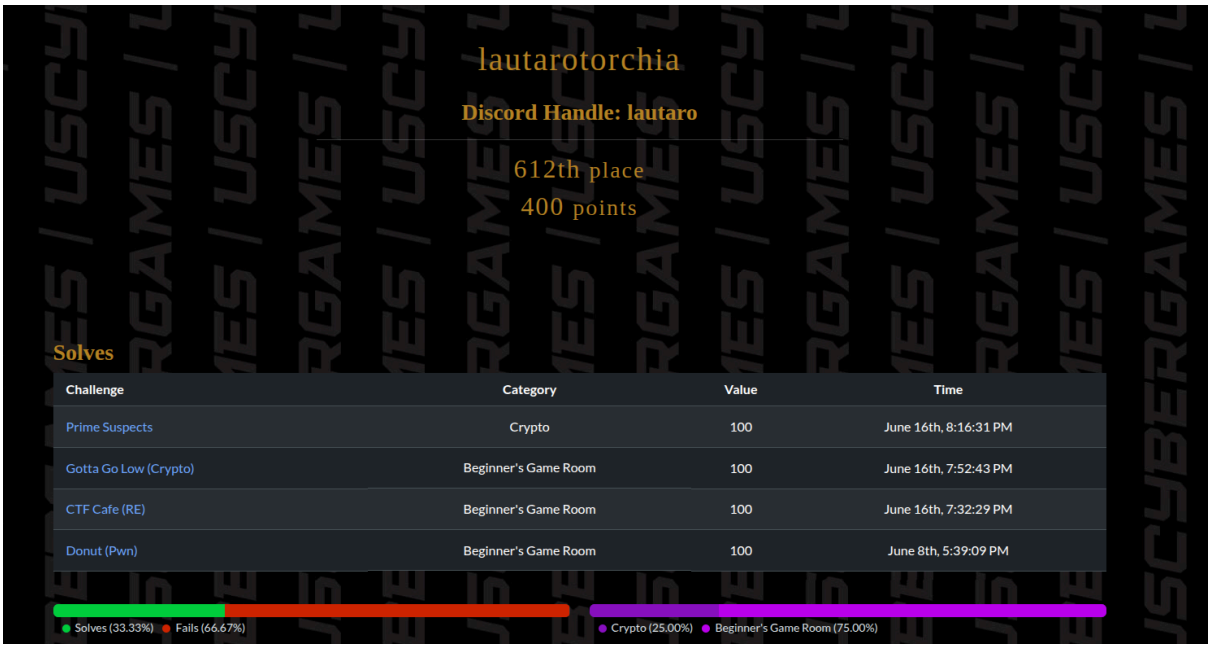
US Cybergames CTF Writeup

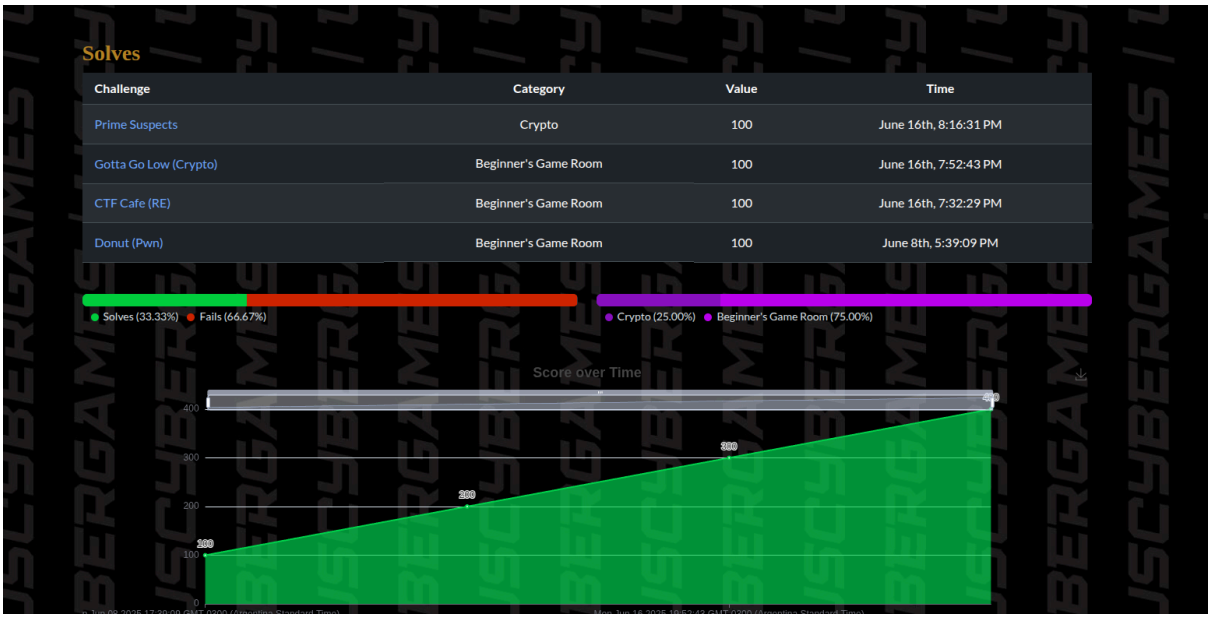
CTF / Plataforma: US Cybergames Beginners Room and Competitive CTF

Link: https://ctf.uscybergames.com/challenges

Mi Nick: lautarotorchia

Resolvi ejercicios tanto en la beginners room como en el Competitive CTF

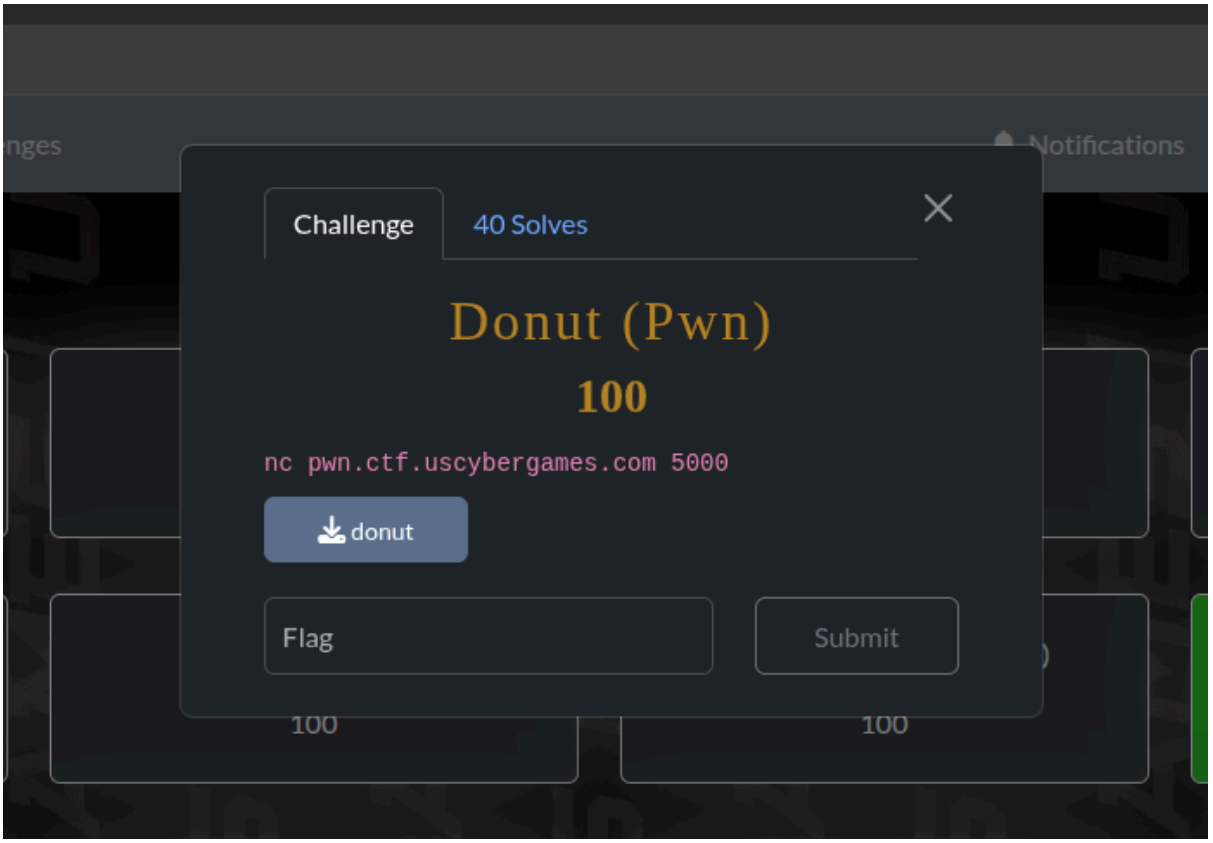




PWN Challenges

Reto: Donut

- **Categoría: PWN**
- **Puntos: 100**



Descripción del desafío El binario `donut` presentaba un menú interactivo con opciones para comprar donas, ganar dinero, y una misteriosa opción de "mantenimiento" (admin). Al inicio del programa, se solicitaba al usuario la "zona horaria" mediante una llamada a la función `gets()`. Esta entrada se almacenaba en un buffer de 32 bytes (`timezone`) que, crucialmente, se encontraba **contiguo y precedía** a una variable global llamada `donuts` en la sección `.bss` del programa.

La opción de mantenimiento tenía una verificación de acceso: solo permitía la entrada si la variable `donuts` tenía el valor mágico `0xCAFEBABE`. Una vez dentro, el programa invocaba un comando utilizando `system("date --date='TZ=\"<timezone>\"'");`, lo que inmediatamente sugería una potencial **inyección de comandos** a través del valor de `timezone`.

Nuestro objetivo era explotar estas vulnerabilidades para acceder a la funcionalidad de administración y ejecutar un comando arbitrario para obtener la flag.

Debugging

Este reto requería una combinación de overflow de buffer para manipular una variable y una inyección de comandos para ejecutar código arbitrario.

Intento 1: Análisis del `gets()` y la estructura de memoria. Lo primero que me llamó la atención fue el uso de `gets()`. Esta es una función notoriamente insegura porque no verifica límites, lo que la convierte en una invitación abierta a los overflows de buffer. Usé `strings donut` para inspeccionar cadenas y encontré la referencia a `system("date --date='TZ=\"%s\"'")` y también pistas sobre la variable `donuts` y el valor `0xCAFEBABE`.

Intento 2: Mapeo de memoria y la relación `timezone` - `donuts`. Para entender cómo `gets()` podía afectar a `donuts`, era esencial conocer la disposición de las variables en memoria. Utilicé un desensamblador/debugger (como `Ghidra` o `objdump` con `gdb`) para examinar la sección `.bss`. Descubrí que la variable `timezone` (el buffer de 32 bytes) estaba directamente antes de la variable global `donuts`. Esto significaba que un overflow en `timezone` (al exceder los 32 bytes) permitiría sobrescribir los bytes de `donuts`.

Intento 3: Superando la verificación `0xCAFEBABE`. La opción de "mantenimiento" dependía de que `donuts == 0xCAFEBABE`. Para acceder a esta funcionalidad, necesitaba sobrescribir `donuts` con ese valor. Dado que el sistema era little-endian (como es común en arquitecturas x86/x64), `0xCAFEBABE` se escribiría como `\xBE\xBA\xFE\xCA`. Esto significaba que mi payload inicial para `timezone` tendría que ser: `[32 bytes de padding] + [\xBE\xBA\xFE\xCA]`. Hice pruebas manuales para confirmar que un padding de 32 bytes exactos era necesario antes de `0xCAFEBABE`.

Intento 4: La inyección de comandos en `system("date...")`. Una vez que pude acceder al modo de mantenimiento, la vulnerabilidad de inyección de comandos era la siguiente en explotar. La cadena `system("date --date='TZ=\"<timezone>\"'");` era perfecta. El valor de `<timezone>` se inserta directamente dentro de comillas dobles, que a su vez están dentro de comillas simples. Un `;` permite encadenar comandos en la shell. Así, `"; cat /flag.txt; #"` sería un payload ideal:

- `;`: Termina el comando `date`.
- `cat /flag.txt`: Ejecuta el comando para leer la flag.
- `;`: Un segundo `;` para asegurar que el `cat` se ejecute antes de cualquier cosa que el programa pudiera intentar después.
- `#`: Comenta el resto de la línea, incluyendo la comilla final del `date` y cualquier otra cosa que el programa agregara, evitando errores de sintaxis en la shell.

Intento 5: Ensamblando el payload final y el trigger. Con todas las piezas identificadas, el payload completo para la entrada de `timezone` sería: `[32 bytes de padding] + [\xBE\xBA\xFE\xCA] + [; cat /flag.txt; #]`

El proceso para el exploit sería:

1. Conectarse al binario.
2. Enviar el payload completo en el prompt de "zona horaria".
3. Seleccionar la opción `3` (Maintenance) del menú para que se active la verificación de `donuts` y luego la llamada a `system()`.

Solución del enfoque: La explotación exitosa requirió la combinación de un **buffer overflow** preciso para sobrescribir una variable de control (`donuts`) y así bypassar una autenticación, seguido de una **command injection** en la llamada a `system()` para ejecutar código arbitrario y obtener la flag. La clave fue la ubicación contigua de las variables y la naturaleza de `gets()` y `system()`.

Análisis final

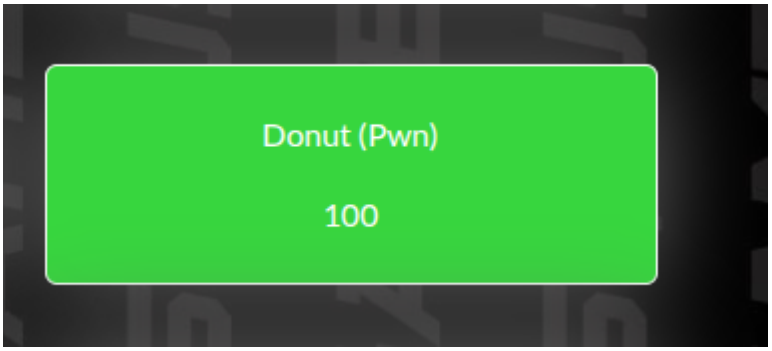
Payload completo: El payload enviado al `gets()` del prompt de la zona horaria fue: `b"A"*32 + b"\xbe\xba\xfe\xca" + b"; cat /flag.txt; #"`

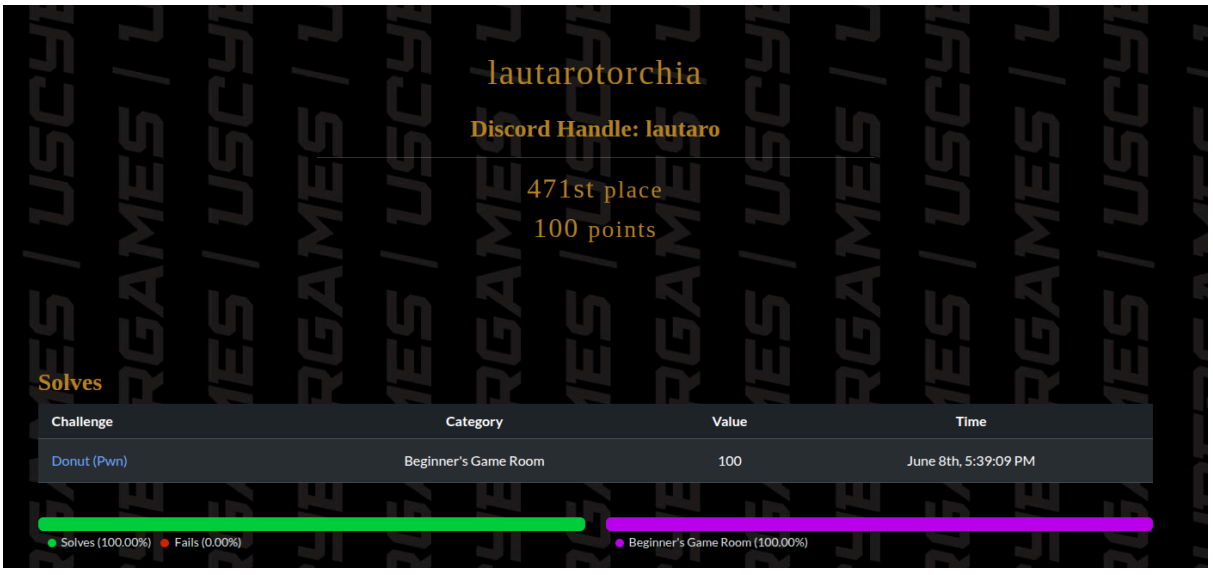
Ejecución: Se utilizó un script en Python `resolution.py`) para interactuar con el binario:

python3 uscyber/pwn/resolution.py

Flag obtenida: `SVBGR{my_fav0rIte_fl4vor_1s_Or3o_54ac91c0}`

Capturas de pantalla:

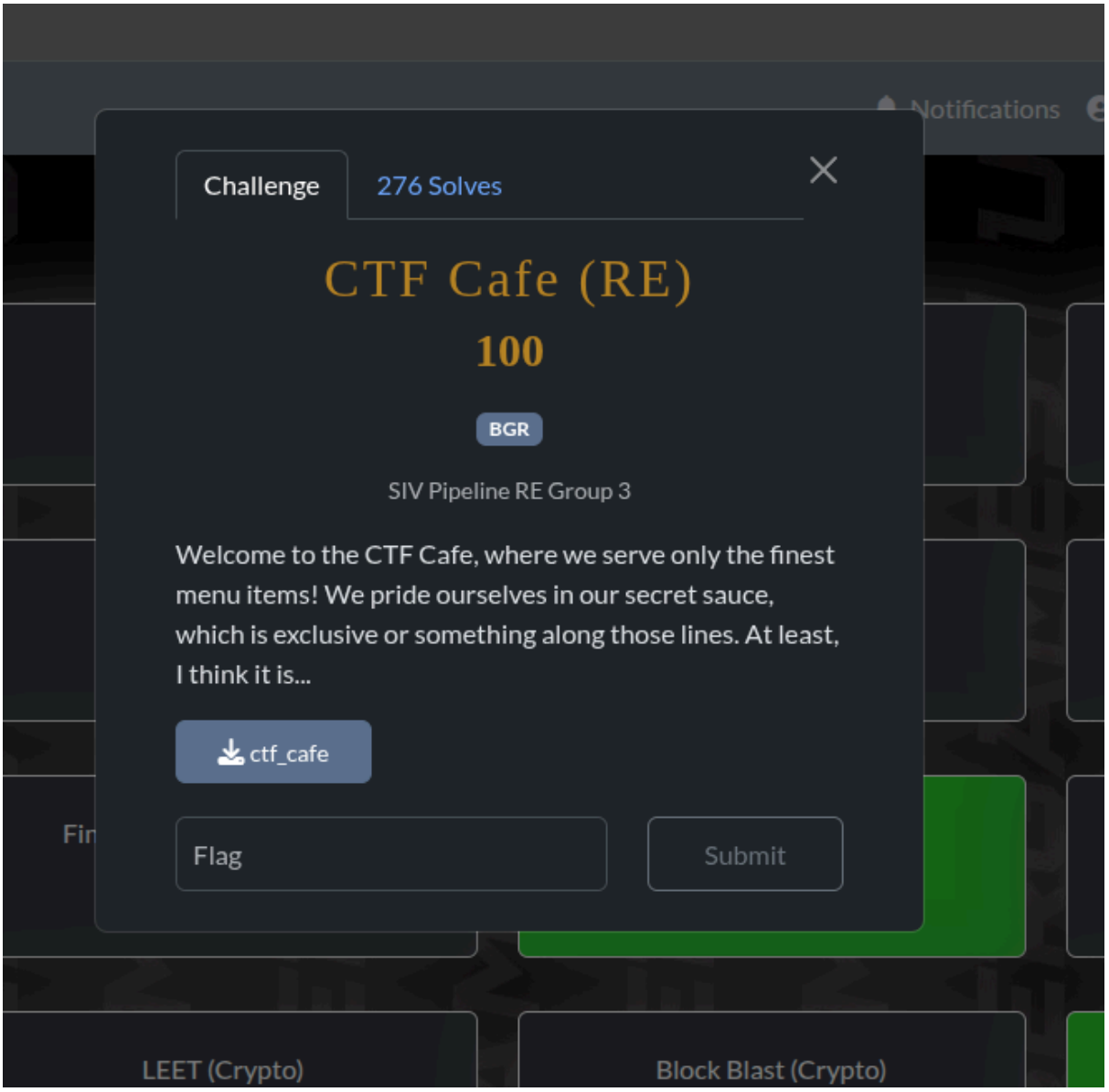




Reverse Engineering Challenges

Reto: CTF Cafe

- **Categoría: RE**
- **Puntos: 100**



Descripción del desafío Se nos proporcionó un binario llamado `ctf_cafe`, un ejecutable ELF de 64 bits para GNU/Linux, dinámicamente enlazado y, crucialmente, **no *stripped***. Al ejecutarlo, presentaba un menú interactivo con opciones como "View Menu", "Place Order", "View Total" y "Exit". Al seleccionar la opción "Exit", el programa solicitaba una "Enter 8-byte hex key…" antes de revelar una supuesta "Secret Sauce".

Una inspección inicial con `strings` no mostró la flag en claro, pero sí reveló referencias al *prompt* de la clave y al símbolo `secret_sauce`. El objetivo del reto era descubrir cómo el

programa construía y revelaba esta "Secret Sauce" ofuscada en su propia sección de datos, y así extraer la flag.

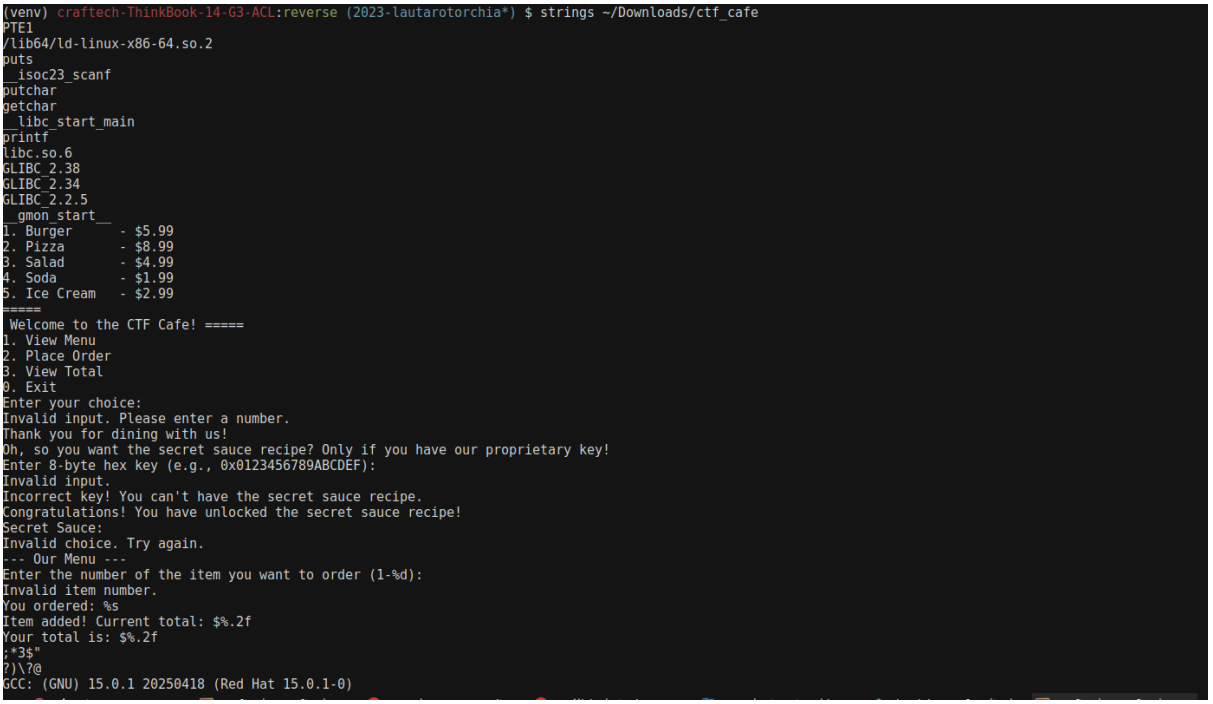
Debugging

Este reto se centró en el análisis estático y dinámico de un binario ELF para desentrañar un algoritmo de ofuscación de datos, donde la clave fue la combinación de herramientas de línea de comandos y un descompilador como Ghidra.

Intento 1: Verificación inicial y búsqueda de cadenas. Mi primer paso fue obtener información básica del binario y buscar cadenas reveladoras.

```
file ctf_cafe
```

```
strings ctf_cafe
```



`file` confirmó que era un "ELF 64-bit... not stripped", lo cual era genial, ya que los nombres de símbolos estarían intactos. `strings` me mostró las opciones del menú, el *prompt* "Enter 8-byte hex key…", y lo más importante: la cadena "secret_sauce". La ausencia de la flag en claro me indicó que estaba ofuscada y se revelaría en tiempo de ejecución.

Intento 2: Inmersión en Ghidra para el análisis estático. Con la pista de `secret_sauce`, cargué el binario `ctf_cafe` en **Ghidra**. Esta herramienta es invaluable para este tipo de retos porque integra desensamblado, *decompilación* y análisis de datos en una interfaz visual.

- **Localizando `secret_sauce` en Ghidra:** Utilicé la ventana "Symbol Tree" (o "Search" por el nombre) para encontrar `secret_sauce`. Ghidra me mostró

directamente que era una variable global en la sección `.data` y su dirección virtual (e.g., `0x4030c0`). Esto reemplazó la necesidad de `nm` y `readelf` para esta parte, ya que Ghidra mapea automáticamente las secciones y *offsets*.

- **Análisis del código con referencias cruzadas (Xrefs):** Una vez en la dirección de `secret_sauce`, busqué las referencias cruzadas (*Xrefs*) a esa ubicación. Esto me llevó directamente al código que manipulaba esta variable, que se encontraba principalmente en la función `main`.
- **Comprensión de la lógica de ofuscación (Pseudo-código):** La ventana de *Decompiler* de Ghidra fue extremadamente útil. Pude ver el pseudo-código de la función `main` y rápidamente identifiqué un bucle. Este bucle iteraba sobre los bytes de `secret_sauce` y realizaba una operación XOR con bytes de otra pequeña *array* (`0x4030e8`, que también identifiqué rápidamente en Ghidra como otro *array* de datos de 8 bytes). El resultado de cada XOR era enviado a `putchar()`.

El pseudo-código era esencialmente así:

```
// int i_var;
// char *secret_sauce_data = (char *)0x4030c0; // Array ofuscado
// char *xor_key_data = (char *)0x4030e8;     // Array clave XOR de 8 bytes

for (i_var = 0; i_var <= 0x20; i_var = i_var + 1) { // 33 iteraciones
  byte1 = secret_sauce_data[i_var];
  byte2 = xor_key_data[i_var % 8]; // Clave cíclica
  putchar(byte1 ^ byte2);
}
// putchar('\n');
```

Esto me reveló que: * La "Secret Sauce" ofuscada era un bloque de 33 bytes (de 0 a 0x20). * Se utilizaba un *array* de 8 bytes (`xor_key_data`) como clave para una operación XOR cíclica. * El resultado se imprimía carácter por carácter.

Intento 3: Extracción de los datos y descifrado en GDB. Con el algoritmo y las direcciones claras gracias a Ghidra, el siguiente paso fue extraer los datos reales de la memoria y aplicar el XOR. Aunque podría haber escrito un script Python para leer el binario y hacerlo, usar GDB me permitió verificar el proceso en tiempo real.

```
gdb ./ctf_cafe
```

```
(gdb) break main
```

```
(gdb) run
```

Una vez en el `main` (después de la interacción inicial del menú), procedí a inspeccionar las regiones de memoria y aplicar el XOR.

- Para obtener el *array* ofuscado: `(gdb) x/33xb 0x4030c0`

- Para obtener el *array* de la clave XOR: `(gdb) x/8xb 0x4030e8`

Finalmente, utilicé un bucle directamente en GDB para realizar la operación XOR y ver el resultado:

```
(gdb) set $i = 0
(gdb) while $i <= 0x20
  > printf "%c", *((unsigned char*)0x4030c0 + $i) ^ *((unsigned char*)0x4030e8 + ($
  > set $i = $i + 1
  > end
(gdb) printf "\n"
```

El resultado impreso en la consola de GDB fue la flag en claro. La "clave" que solicitaba el programa al salir era irrelevante para este proceso de descifrado, ya que la rutina de `Secret Sauce` se ejecutaba siempre.

Solución del enfoque: La flag no estaba directamente en el binario ni se revelaba por una clave de entrada, sino que era el resultado de una operación XOR simple entre un bloque de datos estático y un pequeño *array* de clave cíclico. La clave para resolverlo fue la combinación eficiente del análisis estático con Ghidra para comprender el algoritmo y las ubicaciones de datos, y la depuración dinámica con GDB para extraer y descifrar la

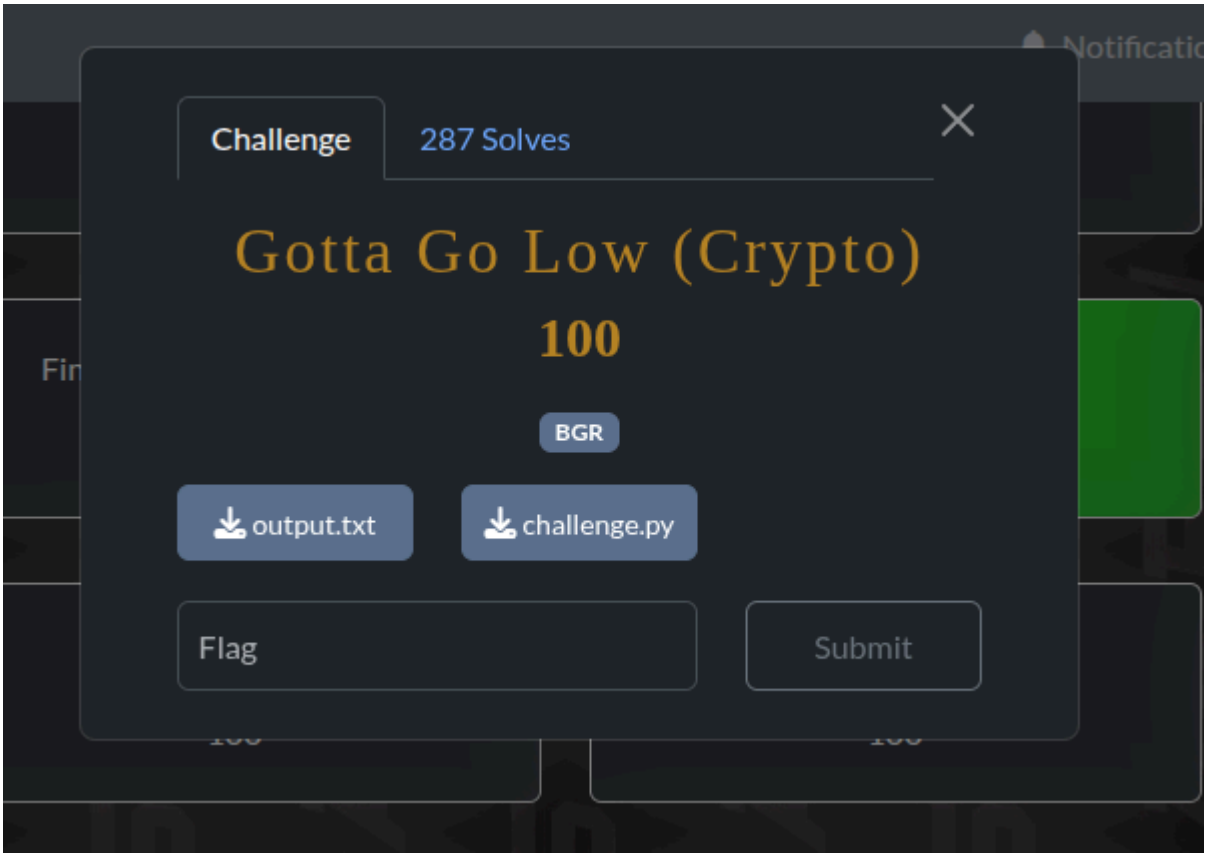
Flag obtenida: `SVBGR{d3c0mp1l3rs_m4k3_l1f3_34sy}`

Captura:

Crypto Challenges

Reto: Gotta go low

- **Categoría: Crypto**
- **Puntos: 100**



Descripción del desafío Se nos proporcionó un script `challenge.py` que generaba un par de claves RSA con un exponente público `e = 3`. Este script luego utilizaba estas claves para encriptar un mensaje corto, el cual tenía el formato `SVBGR{...}`. Adicionalmente, se nos entregó un archivo `output.txt` que contenía los valores de `e`, el módulo `n`, y el `ciphertext` resultante de la encriptación. El objetivo del reto era recuperar la flag sin necesidad de factorizar `n`, basándose en una vulnerabilidad asociada al uso de un exponente bajo.

Debugging

Este reto se centró en una vulnerabilidad bien conocida de RSA cuando se utiliza un exponente público (`e`) muy pequeño y el mensaje original no está debidamente *padded* o es suficientemente corto.

Intento 1: Revisar los archivos entregados y entender el proceso de encriptación. Lo primero que hice fue abrir `challenge.py` para entender la mecánica de la encriptación. Confirmé que el script definía explícitamente `e = 3`. Luego, generaba `p` y `q` (números

primos grandes de 512 bits cada uno) y calculaba `n = p * q`. El mensaje (la flag) se convertía a un entero (`plaintextInt`), y el `ciphertext` se calculaba como `pow(plaintextInt, e, n)`. Después, examiné `output.txt`. Este archivo contenía los valores concretos de `e` (confirmando que era `3`), un valor `n` muy grande y el `ciphertext` también como un entero grande.

Intento 2: Pensar en la vulnerabilidad de `e=3` y el tamaño del mensaje. Al ver `e = 3`, inmediatamente se encendió una alarma. Sé que en RSA, si el exponente público `e` es muy pequeño (como 3) y el mensaje (`plaintextInt`) es también relativamente pequeño, es posible que `plaintextInt^e` (es decir, `plaintextInt^3`) **no sea mayor que `n`**. Si `plaintextInt^3` es menor que `n`, entonces la operación modular `plaintextInt^3 mod n` simplemente resulta en `plaintextInt^3`, sin ninguna reducción. En este escenario, `ciphertext = plaintextInt^3`. Para verificar esto, hice una estimación: un mensaje con el formato `SVBGR{...}` de, digamos, 20-30 caracteres, se convertiría en un entero de aproximadamente 160-240 bits (cada carácter son 8 bits, entonces 20 * 8 = 160, 30 * 8 = 240). Si elevamos un número de 240 bits a la potencia de 3, el resultado sería un número de alrededor de 240×3=720 bits. Dado que `n` era de aproximadamente 1024 bits, era muy probable que `plaintextInt^3` fuera efectivamente menor que `n`. Esto confirmó mi intuición: la vulnerabilidad de "exponente pequeño sin *padding*" era la clave.

Intento 3: Confirmar `ciphertext < n`. Para estar completamente seguro, usé un intérprete de Python para cargar los valores de `n` y `ciphertext` de `output.txt` y realizar la comparación directa, la cuenta dentro del interprete dio True. Esto validó mi hipótesis de que `plaintextInt^3` era igual a `ciphertext`, sin reducción modular. Esto significaba que la operación inversa sería una simple raíz cúbica.

Intento 4: Implementar la raíz cúbica entera y la conversión a texto. Ahora que sabía que `plaintextInt` era la raíz cúbica exacta de `ciphertext`, necesitaba una forma de calcular la raíz cúbica entera de un número potencialmente muy grande. Python es ideal para esto, ya que maneja enteros de tamaño arbitrario. Busqué o implementé una función que pudiera calcular la raíz cúbica entera de forma precisa.

Una vez obtenida la raíz (`root`), que representaba `plaintextInt`, el paso final fue convertir este entero de nuevo a su representación en bytes y luego a una cadena UTF-8. Para ello, calculé la longitud necesaria en bytes (`byte_len = (root.bit_length() + 7) // 8`), convertí el entero a bytes (`plaintext_bytes = root.to_bytes(byte_len, 'big')`), y finalmente decodifiqué a `UTF-8`.

Solución del enfoque: La clave para resolver el desafío fue reconocer la vulnerabilidad de RSA asociada a un exponente público `e` bajo (`e=3`) y un mensaje corto **sin *padding* adecuado**. En este escenario, `ciphertext = plaintextInt^3` sin reducción modular, lo que permite recuperar el `plaintextInt` simplemente calculando la raíz cúbica entera exacta del `ciphertext`. El resto fue la conversión del entero resultante de nuevo a texto ASCII.

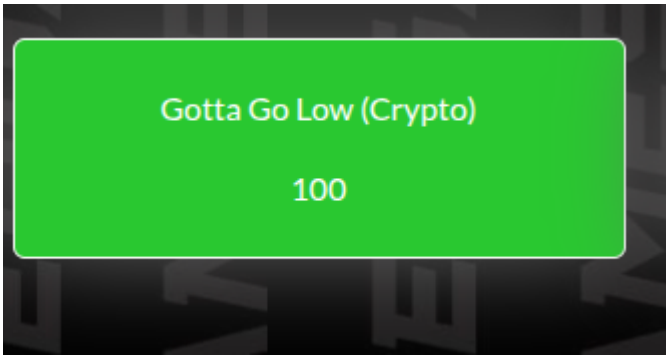
Análisis final

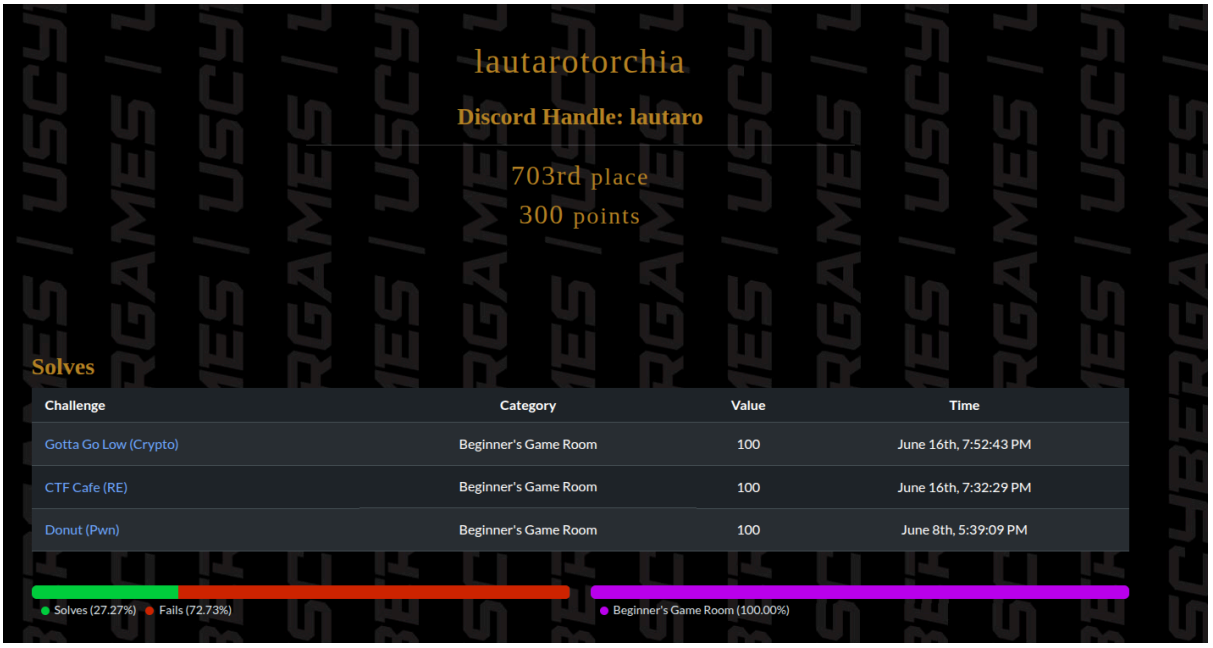
Flag obtenida: :`SVBGR{l0w_3xp0n3nt5_@r3_n0t_s@fe}`

Script de Solución Completo: luego de entender el ejercicio, realice un script que realiza el proceso completo y te indica la flag: **`uscyber/crypto/gottaGoLow/resolution.py`**

```
(venv) craftech-ThinkBook-14-G3-ACL:crypto (2023-lautarotorchia*) $ python gotta_go_low_solve.py output.txt
Flag recuperada:
SVBGR{l0w_3xp0n3nt5_@r3_n0t_s@fe}
(venv) craftech-ThinkBook-14-G3-ACL:crypto (2023-lautarotorchia*) $
```

Captura:





Reto: Prime suspects

- **Categoría: Crypto**
- **Puntos: 100**

Descripción del desafío Se nos proporcionó un conjunto de parámetros RSA aparentemente completo, pero el enunciado sugería que era "sospechosamente débil". Los parámetros dados fueron:

```
n = 10206436730517562300500336780396373599221071772171956321876059887889771063019
e = 65537
c = 665385836500877526533641120993228820260832602079581881911479000198511853145222
```

Nuestro objetivo era descubrir el mensaje original (la flag) cifrada en `c`, aprovechando la debilidad del sistema.

Debugging

Este reto apuntaba a una debilidad común en RSA: el tamaño inadecuado del módulo `n`, que lo hace vulnerable a la factorización directa.

Intento 1: Leer el enunciado y anotar los parámetros. Lo primero fue registrar los valores proporcionados para `n`, `e` y `c`. El valor de `e = 65537` es un exponente público estándar y seguro. Sin embargo, la frase "sospechosamente débil RSA" en el enunciado me hizo dudar del tamaño de `n`. Mi primera hipótesis fue que `n` podría ser demasiado pequeño para una seguridad moderna, lo que permitiría su factorización.

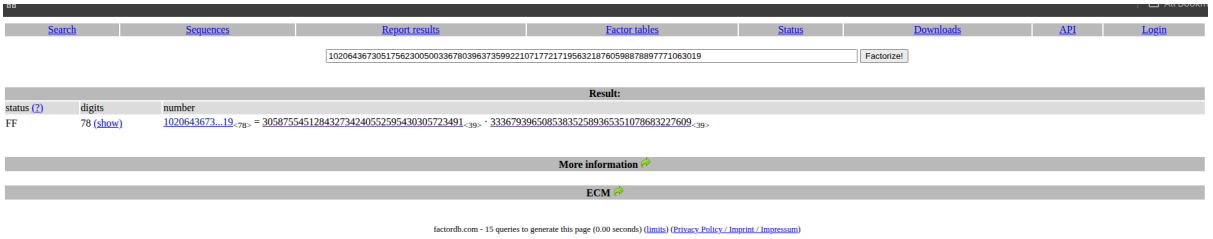
Intento 2: Verificar el tamaño de `n`. Para confirmar mi hipótesis, abrí un intérprete de Python y verifiqué la longitud en bits de `n`.

La salida fue "Bits de n: 256".Un módulo RSA de solo 256 bits es **extremadamente pequeño** para estándares actuales y es factorizable con herramientas computacionales comunes. Esto confirmó que la vulnerabilidad principal era la facilidad de factorizar `n`.

Intento 3: Pensar en la vulnerabilidad concreta y el plan de ataque. Con `e=65537` (un exponente público grande) y un `n` pequeño, la vulnerabilidad no era un ataque de "exponente bajo sin *padding*" (como en "Gotta Go Low"). En este caso, la debilidad radicaba en la simplicidad de la **factorización de `n`**. El plan de ataque era claro:

1. Factorizar `n` para obtener los números primos `p` y `q`.
2. Calcular la función totiente de Euler, $\varphi(n)=(p-1)(q-1)$.
3. Calcular la clave privada de desencriptación `d`, que es el inverso modular de `e` modulo $\varphi(n)$ ($d \equiv e^{-1} \pmod{\varphi(n)}$).
4. Descifrar el `ciphertext c` usando `d` y `n` ($m \equiv c^d \pmod{n}$).
5. Convertir el mensaje `m` (un entero) a texto legible.

Intento 4: Factorizar `n` usando Factordb.com. La forma más sencilla y rápida de factorizar números pequeños es a través de servicios en línea como Factordb.com. Abrí mi navegador y pegué el valor completo de `n` en el campo de búsqueda de https://factordb.com. El sitio procesó `n` rápidamente y me mostró sus factores. Al hacer clic en "Show" para ver los valores completos, obtuve dos números primos de aproximadamente 39 dígitos cada uno:



p = 305875545128432734240552595430305723491

q = 333679396508538352589365351078683227609

Copié ambos valores completos para usarlos en los cálculos posteriores.

Intento 5: Verificar factores y calcular `φ(n)` y `d` en Python. Volví al REPL de Python para realizar los cálculos. Primero, verifiqué que los factores `p` y `q` que obtuve de Factordb.com fueran correctos multiplicándolos y comparando el resultado con `n`. Luego, calculé φ(n)=(p−1)(q−1). Finalmente, calculé la clave privada `d` usando la función `pow(e, -1, phi)` de Python, que calcula el inverso modular.

Confirmé que `gcd(e, phi)` era 1, lo cual es necesario para que `d` exista, y que `d` se calculó correctamente.

Intento 6: Descifrar el `ciphertext c` y convertirlo a texto. Con la clave privada `d` en mano, el último paso fue descifrar el `ciphertext c` usando la fórmula m≡cd(modn). El resultado `m` es un entero que representa el mensaje original. Finalmente, convertí este entero `m` de nuevo a su representación en bytes y luego a una cadena de texto (UTF-8, o `latin-1` si hubiera problemas con caracteres especiales).

Solución del enfoque: La clave para resolver el desafío fue identificar la debilidad principal del RSA: un módulo `n` de tamaño insuficiente (256 bits), lo que permitió su factorización trivial usando herramientas en línea como Factordb.com. Una vez factorizado `n` en sus primos `p` y `q`, el resto del proceso (calcular φ(n), la clave privada `d`, y descifrar el mensaje) siguió el procedimiento estándar de RSA.

Análisis final

Flag obtenida: Realizando los cálculos descritos (factorización de `n`, cálculo de `φ(n)` y `d`, y descifrado de `c`), la flag recuperada es: `SVUSCG{sm4ll_pr1m3s}`

Script de Solución Completo: luego de entender el ejercicio, realice un script que realiza el proceso completo y te indica la flag:
`uscyber/crypto/primeSuspect/resolution.py`

```
(venv) craftech-ThinkBook-14-G3-ACL:crypto (2023-lautarotorchia*) $ python solve.py
Flag: SVUSCG{sm4ll_pr1m3s}
(venv) craftech-ThinkBook-14-G3-ACL:crypto (2023-lautarotorchia*) $
```

Captura: