ChronoVM

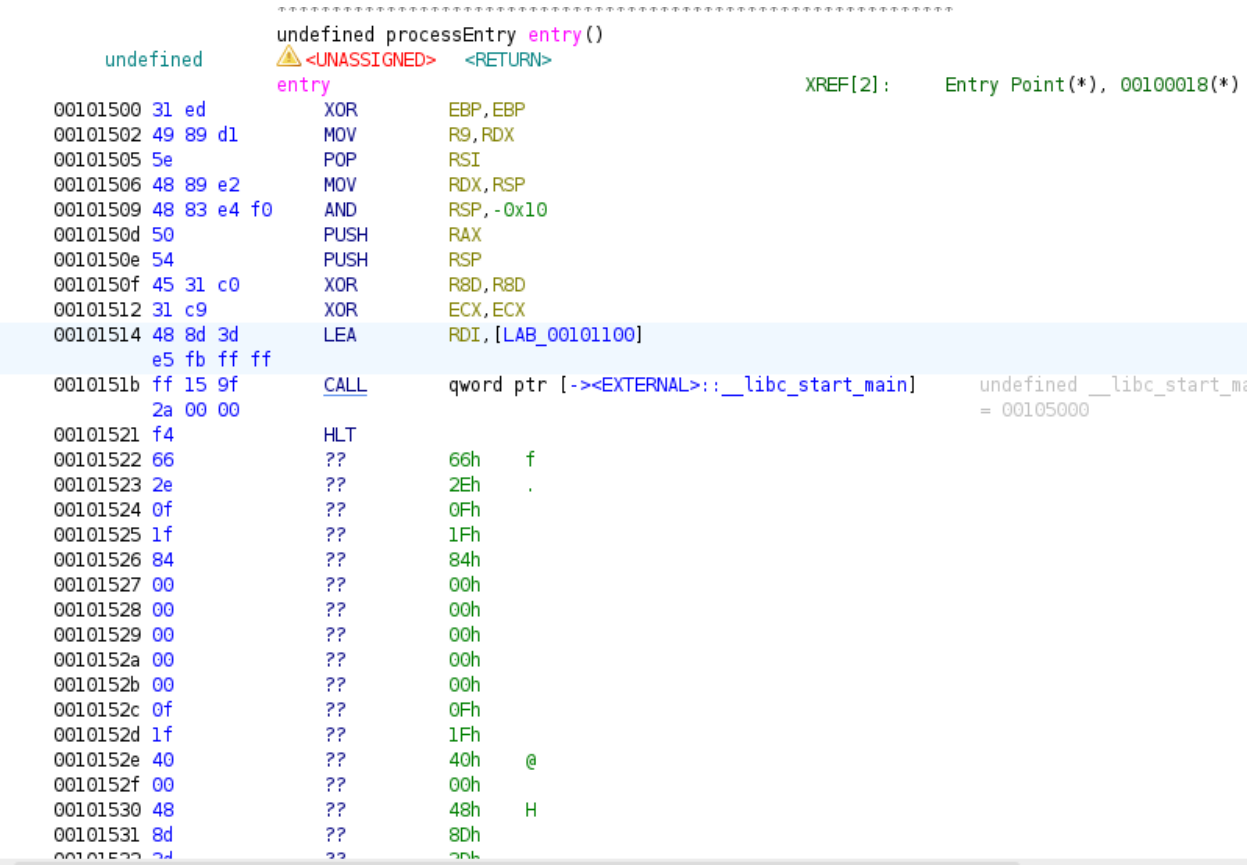
El binario chronovm solicita una clave por entrada estándar y responde únicamente con Correct! o Nope..
No existe ninguna comparación directa contra la flag, ni cadenas visibles que permitan obtenerla trivialmente.

El reto implementa una máquina virtual personalizada (VM) junto con una verificación criptográfica basada en SHA-256, lo que obliga a comprender el flujo interno del binario para poder resolverlo.

```
└─$ file chronovm
chronovm: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=f53391a2da21c16f1230d347e93fd4726fb07bad, for GNU/Linux 3.2.0, stripped
```

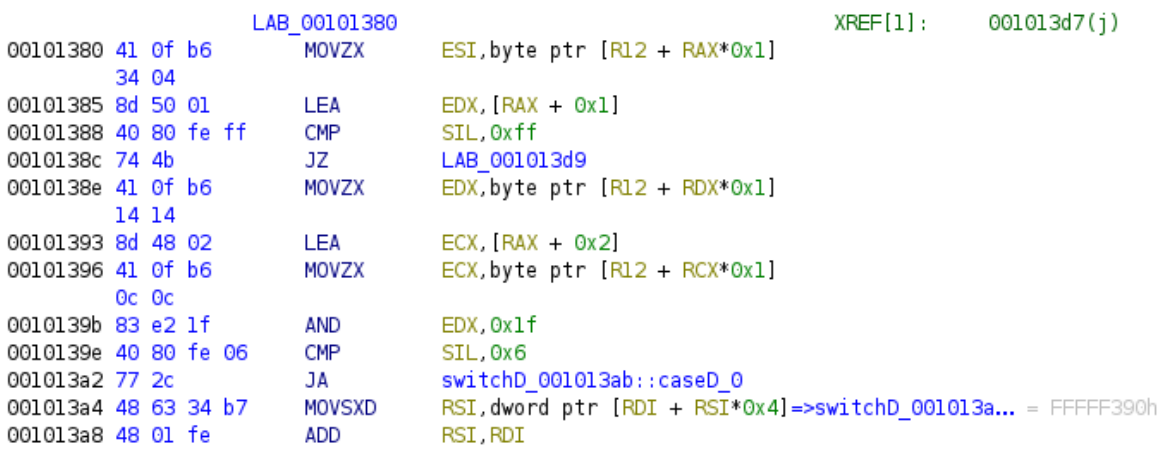
En la función de entrada del binario (entry) se observa el flujo estándar de inicialización de un ejecutable ELF en Linux. El código prepara los registros y realiza una llamada a __libc_start_main, desde donde se transfiere el control a la función main.

No se detecta ninguna lógica de validación o comparación de la flag en esta etapa, confirmando que el binario sigue un flujo convencional y que el análisis debe centrarse en funciones posteriores.



En esta sección del código se identifica el bucle principal de la máquina virtual implementada por el binario. Cada iteración lee una instrucción del bytecode y ejecuta una operación específica mediante un mecanismo de dispatch tipo switch, basado en el valor del opcode.

Se observan comparaciones contra valores constantes (CMP SIL, 0x6) y saltos a diferentes casos, lo que corresponde a operaciones como XOR, sumas, rotaciones y sustituciones mediante S-box. Este diseño introduce una capa adicional de abstracción que dificulta significativamente la inversión directa del algoritmo.



Antes de ejecutar la máquina virtual, el binario aplica una transformación inicial a la entrada del usuario.

Esta lógica puede observarse claramente en Ghidra en un bucle que itera sobre cada carácter ingresado.

Tras ejecutar la máquina virtual, el binario obtiene un estado final de 32 bytes.

Este estado no se imprime, ni se compara directamente con la flag.

En su lugar, el binario realiza una comparación byte a byte contra un estado objetivo almacenado en vm.bin.

Para evitar la exposición directa del oracle durante la ejecución normal, el binario solo muestra el valor Matches cuando se habilita un modo de depuración mediante una variable de entorno.

Debido a que el binario no compara la flag directamente, sino que utiliza una métrica de similitud global (Matches), no es posible validar la entrada carácter por carácter ni aplicar fuerza bruta. El valor Matches actúa como una función de fitness ruidosa, ya que puede aumentar incluso con caracteres incorrectos y no es monotónico.

Para resolver el reto, se aborda el problema como una búsqueda heurística en un espacio grande de candidatos. Se utiliza una estrategia de beam search, en la que se mantienen varios candidatos simultáneamente y solo se conservan aquellos que presentan una mayor similitud con el estado objetivo. De esta manera, se reduce el impacto de los máximos locales y se permite una convergencia progresiva hacia la flag correcta sin invertir directamente la máquina virtual.

Solucion:

```python
#!/usr/bin/env python3
import subprocess
import re
import string
from collections import defaultdict

BIN = "./chronovm"

# charset típico de flags CTF
CHARSET = string.ascii_letters + string.digits + "_{}"

# Parámetros (puedes subirlos si quieres más potencia)
MAX_LEN = 40 # límite de longitud
BEAM_WIDTH = 200 # cuántos candidatos mantener por nivel
REPEATS = 1 # repetir ejecución para promedio (sube a 3 si quieres más estabilidad)



def get_matches(candidate: str) -> int:
"""
Ejecuta el binario y extrae X de 'Matches: X/32'.
Devuelve -1 si no encuentra la línea.
"""
total = 0
for _ in range(REPEATS):
p = subprocess.run(
[BIN],
input=(candidate + "\n").encode(),
stdout=subprocess.PIPE,
stderr=subprocess.DEVNULL,
check=False,
)
out = p.stdout.decode(errors="ignore")
m = re.search(r"Matches:\s*(\d+)\s*/\s*32", out)
if not m:
return -1
total += int(m.group(1))
return total // REPEATS



def solve():
# empezamos con algo razonable
start = "FIS"
s0 = get_matches(start)
```

```
if s0 < 0:
print("[-] No pude leer 'Matches: X/32'. ¿Estás usando el chronovm con oracle Matches?")
return

beam = [(start, s0)]
seen = set([start])

print(f"[*] Start beam: {beam}")

for step in range(MAX_LEN):
scored = defaultdict(int)

# expandimos el beam
for cand, sc in beam:
for ch in CHARSET:
nxt = cand + ch
if nxt in seen:
continue
seen.add(nxt)

m = get_matches(nxt)
if m < 0:
print("[-] Salida inesperada del binario (no encontré Matches).")
return

scored[nxt] = m

# si ya está perfecto, listo
if m == 32:
print("\n[✓] FLAG ENCONTRADA:", nxt)
return

if not scored:
print("[-] No hay expansiones nuevas (beam vacío).")
return

# seleccionamos top candidates por score y preferimos más cortos si empatan
beam = sorted(scored.items(), key=lambda x: (-x[1], len(x[0])))[:BEAM_WIDTH]

best_cand, best_score = beam[0]
print(f"[+] Step {step:02d} | best: ({best_cand!r}, {best_score}/32) | beam_size={len(beam)}")
```

```
# heurística: si ya llegamos a '}', prioriza terminación (opcional)
for cand, sc in beam:
if cand.endswith("}") and sc == 32:
print("\n[✓] FLAG ENCONTRADA:", cand)
return

print("[-] No encontrada dentro del límite. Sube BEAM_WIDTH/MAX_LEN o amplía CHARSET.")



if __name__ == "__main__":
solve()
```