Writeup Laberinto

Introducción

Este reto nos presentaba un archivo inputs.csv con series de 4 bits (A, B, C, D) y una imagen que mostraba un circuito electrónico con transistores NPN. El objetivo, como siempre, era extraer una flag, probablemente oculta en la salida del circuito al procesar las entradas dadas.

Descripción

Hace unos días comencé mis prácticas formativas de la mano del gran maestro Juan Santos. Desde mi llegada a la empresa, no ha dejado de evaluar constantemente mis conocimientos de electrónica con pequeñas pruebas, pero la última de todas se me está resistiendo...

Análisis Inicial (o la falta de él)

Aquí es donde golpeó la realidad: hacía años que no tocábamos un diagrama de circuito, las tablas de verdad nos sonaban lejanas y recordar la diferencia entre NAND y NOR bajo presión no estaba en nuestros planes inmediatos.

Honestamente, no teníamos ni el tiempo ni las ganas de desempolvar los apuntes de electrónica digital o empezar a dibujar diagramas. Sabíamos que teníamos 4 entradas binarias y, presumiblemente, 1 salida binaria por cada fila de entrada.

La Epifanía del Perezoso: ¡Fuerza Bruta!

Si no sabemos cuál es la función lógica correcta, pero sabemos que tiene 4 entradas y 1 salida... ¿cuántas funciones posibles hay?

- Hay 2^4 = 16 combinaciones posibles de entrada (de 0000 a 1111)
- Para cada una de esas 16 combinaciones, la salida puede ser 0 o 1
- Por lo tanto, el número total de funciones lógicas distintas de 4 entradas y 1 salida es 2^16 = 65,536

65,536 puede sonar grande, pero para un ordenador moderno y un script de Python, probarlas todas es totalmente factible. ¡Mucho más rápido que intentar recordar cómo funcionaban los mapas de Karnaugh!

La estrategia se volvió obvia:

1. **Representar cada función**: Usar un número entero entre 0 y 65,535. Cada número representa una tabla de verdad única de 16 filas.
2. **Iterar**: Crear un bucle que vaya desde function_id = 0 hasta 65535.
3. **Procesar Entradas**: Para cada function_id, aplicar la función correspondiente a las entradas del CSV.
4. **Decodificar y Comprobar**: Convertir los bits resultantes a ASCII y buscar el formato de flag.

Implementación

Modificamos el script original para implementar esta lógica de fuerza bruta. La función clave fue apply_brute_force_logic que usaba shifts de bits y máscaras para obtener la salida correcta para cualquier función dada su ID:

```
def apply_brute_force_logic(a_str, b_str, c_str, d_str, function_id):
  input_index = int(a_str + b_str + c_str + d_str, 2)
  output_bit = (function_id >> input_index) & 1
  return output_bit

# Bucle principal
for func_id in range(65536):
    output_bits = ""
    for a, b, c, d in input_data:
        output_bit = apply_brute_force_logic(a, b, c, d, func_id)
        output_bits += str(output_bit)

    decoded_string = decode_bits_to_ascii(output_bits)
    if "hfctf" in decoded_string:
        print(f"Found flag with function ID: {func_id}")
```

```
    print(f"Decoded: {decoded_string}")
    break
```

Ejecución y Resultado

Después de ejecutar nuestro script durante unos segundos, encontramos lo que estábamos buscando:

```
--- ¡Coincidencia Encontrada! ---
Función Lógica ID: 42538 (Decimal) / 0xA62A (Hex)
Bits Generados:
01101000011001100110001101110100011001100111101100110100011011000110011101110101011011100110
00010010110101110110001100110111101001011111010010000001100001011100110101111100110011011100
11011000110101010101100011011010000011010001100100010111110111001100110000011000100111001000
11001101011111010110000100111001100101010100100011111101111101
Resultado Decodificado: hfctf{4lgun4_v3z_H4s_3scUch4d0_s0br3_XN0R??}
----------------------------------------
```

Flag: hfctf{4lgun4_v3z_H4s_3scUch4d0_s0br3_XN0R??}