The Weather App (PWN)

Connect to the App

First, I connected to the remote server using:

```
nc 176.101.48.153 20271
```

After connecting, I looked around and saw some files. I noticed a file named `weather`, which looked interesting.

Check the File

I ran `ls -ahl` and `file weather`, and saw that it's an ELF binary:

```
weather.bin: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=f0e2b8e9334da4dc53f7398988ba2e021693cbd2, for GNU/Linux
3.2.0, not stripped
```

So, I downloaded it to my system using:

- **On my system**:

  ```
  nc -lvp 1234 > weather.bin
  ```

- **On the remote server**:

  ```
  nc <my_ip> 1234 < weather
  ```

Ghidra Analysis

I opened the binary in Ghidra. In the `main` function, I saw the following code:

```
undefined8 main(void)

{
  int iVar1;
  int iVar2;
  size_t __n;
  long in_FS_OFFSET;
```

```
  addrinfo *local_58;
  char *local_50;
  addrinfo local_48;
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  memset(&local_48,0,0x30);
  local_48.ai_family = 2;
  local_48.ai_socktype = 1;
  iVar1 = getaddrinfo(HOST,PORT,&local_48,&local_58);
  if (iVar1 != 0) {
    perror("getaddrinfo");
                    /* WARNING: Subroutine does not return */
    exit(1);
  }
  iVar1 = socket(local_58->ai_family,local_58->ai_socktype,local_58-
>ai_protocol);
  if (iVar1 < 0) {
    perror("socket");
                    /* WARNING: Subroutine does not return */
    exit(1);
  }
  iVar2 = connect(iVar1,local_58->ai_addr,local_58->ai_addrlen);
  if (iVar2 < 0) {
    perror("connect");
                    /* WARNING: Subroutine does not return */
    exit(1);
  }
  local_50 = (char *)malloc(0x3d);
  get_flag(local_50);
  __n = strlen(local_50);
  send(iVar1,local_50,__n,0);
  puts("Weather INFO was sent");
  close(iVar1);
  freeaddrinfo(local_58);
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return 0;
}
```

The logic creates a socket connection, calls a function named `get_flag`, then sends the result. So I checked the `get_flag` function:

```
void get_flag(long param_1)

{
  long in_FS_OFFSET;
  int local_5c;
  undefined8 local_58;
  undefined8 local_50;
  undefined8 local_48;
  undefined8 local_40;
  undefined8 local_38;
  undefined5 local_30;
  undefined3 uStack_2b;
  undefined5 uStack_28;
  undefined8 local_23;
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  local_58 = 0x211d1b161c0e0f13;
  local_50 = 0x52f6a03052d350d;
  local_48 = 0x6a051e1f0c161509;
  local_40 = 0xa0509341e05280f;
  local_38 = 0x51d341334130935;
  local_30 = 0x36361b3219;
  uStack_2b = 0x1d1469;
  uStack_28 = 0x16690d0569;
  local_23 = 0x502769146a1e0516;
  for (local_5c = 0; local_5c < 0x3d; local_5c = local_5c + 1) {
    *(byte *)(param_1 + local_5c) = *(byte *)((long)&local_58 +
(long)local_5c) ^ 0x5a;
  }
  *(undefined *)(param_1 + 0x3d) = 0;
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}
```

The variables like `local_58`, `local_50`, etc. hold some strange-looking hexadecimal values. These are likely encrypted data, especially given the XOR (`^ 0x5A`) operation inside the loop. That was a clear sign of simple XOR-based obfuscation.

--

Step-by-Step Explanation

1. **Encrypted Flag**

   The flag is stored across multiple 8-byte variables like `local_58`, `local_50`, etc.

2. **Decryption Loop**

   ```
   for (local_5c = 0; local_5c < 0x3d; local_5c++) {
       *(byte *)(param_1 + local_5c) = *(byte *)((long)&local_58 +
   (long)local_5c) ^ 0x5a;
   }
   ```

This XORs 61 bytes with 0x5A and stores the result in the buffer.

3. **Null Terminator**

   ```
   *(undefined *)(param_1 + 0x3d) = 0;
   ```

   Adds a null byte at the end to terminate the string.

4. **Stack Canary**

   The function checks for stack smashing with a canary guard, a standard anti-overflow measure.

Extract the Flag

Now that we understand the logic, we can extract the flag using the following Python script:

```python
from struct import pack

# Combine all into one byte string
data = (
    pack("<Q", 0x211d1b161c0e0f13) +
    pack("<Q", 0x052f6a03052d350d) +
    pack("<Q", 0x6a051e1f0c161509) +
    pack("<Q", 0x0a0509341e05280f) +
    pack("<Q", 0x051d341334130935) +
    pack("<Q", 0x00000036361b3219)[:5] +
    pack("<I", 0x1d1469)[:3] +
    pack("<Q", 0x0000016690d0569)[:5] +
    pack("<Q", 0x502769146a1e0516)
)

# XOR with 0x5A
flag = bytes([b ^ 0x5A for b in data])
print(flag.decode())
```

This script reconstructs the encrypted flag, XORs it with 0x5A, and prints the result.

```
IUTFLAG{Wow_Y0u_SOLVED_0Ur_DnS_PoSInInG_ChAll3NG3_W3LL_D0N3}
```