CTF Write-up: rev_easy

Challenge Description

The challenge presented an unknown binary, rev_easy, with the description: "An unknown binary guards access with a secret key. The program doesn't lie, but it doesn't speak plainly either." The goal was to find the secret key, which is the flag, in the format SECE{...}.

Phase 1: Initial Analysis

The first step was to analyze the provided binary to understand its basic properties and identify potential areas of interest.

Command	Output Summary	Conclusion
file rev_easy	ELF 64-bit LSB pie executable, x86-64, stripped	A standard Linux executable. The "stripped" nature indicates that symbol names (like function names) are removed, requiring more effort in static analysis.
strings rev_easy	Enter flag: , Nope., Wrong flag., Correct! , WFIJ", r{VR~, etc.	The presence of success/failure messages and a prompt confirms it's a typical reverse engineering challenge. The non-printable strings (WFIJ", etc.) are likely the hardcoded encrypted flag data.

Phase 2: Reverse Engineering the Validation Logic

Since the binary was stripped, we used objdump to disassemble the code and locate the main logic, which is typically where the flag validation occurs. The entry point was identified around the address 0x10c0.

A. Input Handling and Length Check

The disassembly revealed the following sequence of operations:

1. **Prompt**: The program prints the string "Enter flag: " (found at address 0x2004 in .rodata).
2. **Input**: It reads user input into a buffer on the stack.
3. **Newline Removal**: It uses strcspn and a null-byte write to remove the trailing newline character from the input.
4. **Length Check**:

```
111a:    e8 21 ff ff ff           call   1040 <strlen@plt>
```

```
111f:    48 83 f8 12              cmp    $0x12,%rax
```

```
1123:    75 61                    jne    1186 <wrong_flag_message>
```

5 The instruction cmp $0x12,%rax compares the input length (%rax) with the hexadecimal value 0x12, which is **18** in decimal. This immediately tells us the flag must be exactly 18 characters long.

B. The Encryption Loop

The core validation logic is a loop starting at address 0x1140. This loop iterates over the 18 characters of the user's input, applying a series of arithmetic and bitwise operations, and comparing the result to a hardcoded value.

Initial Key Values (Registers):

The loop is initialized with three key values, which are loaded into the 32-bit registers %edi, %esi, and %edx:

Register	Initial Value (Hex)	Initial Value (Decimal)	Role
%edi	0x5a	90	XOR key 2
%esi	0xffffffef	-17 (signed 32-bit)	ADD key
%edx	0x0d	13	XOR key 1

Per-Character Validation Logic (Assembly at 0x1140):

```
1140:    0f b6 01                 movzbl (%rcx),%eax  ; eax = input_char
```

```
1143:    31 d0                    xor    %edx,%eax    ; eax = eax ^ edx
1145:    01 f0                    add    %esi,%eax    ; eax = eax + esi
1147:    31 f8                    xor    %edi,%eax    ; eax = eax ^ edi
1149:    41 38 00                 cmp    %al,(%r8)    ; Compare result
with encrypted_byte
```

```
114c:    75 2a                    jne    1178 <nope_message>
```

The validation can be summarized by the following equation, where C is the input character and E_i is the hardcoded encrypted byte at index i:

$$
((C \oplus \text{EDX}) + \text{ESI}) \oplus \text{EDI} = E_i
$$

Key Update Logic (Assembly after comparison):

The keys are updated at the end of each iteration, making the encryption a rolling cipher:

Register	Update Operation
%edx	add $0x7,%edx (EDX += 7)
%esi	sub $0x3,%esi (ESI -= 3)
%edi	add $0x2,%edi (EDI += 2)

Phase 3: Extracting the Encrypted Data

The hardcoded encrypted data is the array that the result of the character operation is compared against. This array was found in the .rodata section, which is the read-only data segment of the binary.

Using objdump -s -j .rodata rev_easy, we identified the relevant bytes starting at address 0x2040:

```
2040 17611f2d 5746494a 22929972 7b56527e  .a.-WFIJ"..r{VR~
```

The 18 encrypted bytes (E_0 to E_{17}) are: [0x17, 0x61, 0x1f, 0x2d, 0x57, 0x46, 0x49, 0x4a, 0x22, 0x92, 0x99, 0x72, 0x7b, 0x56, 0x52, 0x7e, 0x01, 0x1b]

Phase 4: Decryption Script

To find the original character C, we need to reverse the encryption equation:

$$
C = (((E_i \oplus \text{EDI}) - \text{ESI}) \oplus \text{EDX}) \pmod{256}
$$

However, due to the complexity of handling 32-bit signed arithmetic (especially with the negative ESI value) and the modulo operations, a brute-force simulation of the forward logic is often more reliable in a scripting environment.

The Python script below simulates the loop and tests all 256 possible input characters (inp) for each position until the result matches the hardcoded encrypted byte (target).

```
def solve():

    # Encrypted bytes from .rodata at 0x2040 (18 bytes)
    encrypted = [
        0x17, 0x61, 0x1f, 0x2d, 0x57, 0x46, 0x49, 0x4a,
        0x22, 0x92, 0x99, 0x72, 0x7b, 0x56, 0x52, 0x7e,
        0x01, 0x1b
    ]

    # Initial 32-bit key values
    edi = 0x5a
    esi = 0xffffffef # -17 in 32-bit signed
    edx = 0x0d

    flag = ""
    for i in range(len(encrypted)):
        target = encrypted[i]

        found = False
        for inp in range(256):
            # 1. eax = inp
            eax = inp

            # 2. eax = eax ^ edx (32-bit XOR)
            eax = (eax ^ edx) & 0xFFFFFFFF

            # 3. eax = eax + esi (32-bit ADD)
            eax = (eax + esi) & 0xFFFFFFFF

            # 4. eax = eax ^ edi (32-bit XOR)
            eax = (eax ^ edi) & 0xFFFFFFFF

            # 5. Compare lower 8 bits (AL) with target
            if (eax & 0xFF) == target:
                flag += chr(inp)
                found = True
                break

        if not found:
            flag += "?"

        # Update key values for the next iteration
```

```python
        edx = (edx + 7) & 0xFF
        esi = (esi - 3) & 0xFFFFFFFF
        edi = (edi + 2) & 0xFF

    print(f"Flag: {flag}")

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

Executing the script yields the flag:

```
$ python3 solve.py
Flag: SECE{rev4fun_2025}
```

Conclusion

The final flag, verified by running it against the original binary, is:

SECE{rev4fun_2025}

The challenge was a classic example of a stripped binary requiring static analysis to uncover a custom, rolling-key encryption algorithm, which was then reversed using a Python simulation.

References

[1] Disassembly of rev_easy binary using objdump.
[2] Extracted strings fromrev_easy binary using strings.
[3] Extracted.rodata section from rev_easy binary using objdump -s -j .rodata.