Shadow Blind CTF Write-up

Category: Reverse Engineering **Goal:** Uncover the hidden validation mechanism in a service binary and trigger the concealed execution path to recover the flag.

1. Initial Triage and Analysis

The challenge provides two files, which we first extract and examine:

File Name	Type	Description
shadow_core	ELF 64-bit LSB pie executable, stripped	The main service binary.
libshadow.so	ELF 64-bit LSB shared object, stripped	The dynamically loaded library.

An initial execution of the service shows the normal, non-flag path:

```
$ export LD_LIBRARY_PATH=.
```

```
$ ./shadow_core
=== Shadow Bind Service ===
[*] Service mode: INTERACTIVE
[!] Debug mode enabled
[*] Loading internal modules...
[*] Performing integrity validation...
[*] Integrity status: OK
[*] No anomalies detected
```

```
[*] Service ready
```

The challenge description states that the core logic is in the dynamically loaded library, which performs a "quiet inspection of the service itself at runtime."

2. Analyzing the Shared Library (libshadow.so)

We use nm to inspect the exported symbols from libshadow.so to identify the functions involved in the integrity check:

```
$ nm -D libshadow.so
```

```
...
00000000000011f9 T normal_path
0000000000001226 T secret_path
```

```
000000000000131e T system_check
```

The three key functions are system_check, normal_path, and secret_path. The shadow_core binary calls system_check (the integrity validation), which, based on the output, currently executes normal_path. Our goal is to force it to execute secret_path.

The system_check Function

Disassembling system_check reveals the integrity check logic. The critical part of the function is the self-inspection routine:

1. **Open Executable**: The function opens the running executable using the path /proc/self/exe [1].
   - lea 0xd0b(%rip),%rax (loads address of /proc/self/exe)
   - call open
2. **Seek to Offset**: It then uses lseek to move the file pointer to a specific offset.
   - mov $0x3000,%esi (sets the offset to 0x3000, or 12288 decimal)
   - call lseek
3. **Read Data**: It reads 6 bytes from that offset into a local buffer.
   - mov $0x6,%edx (sets the size to 6 bytes)
   - call read
4. **Compare Data**: Finally, it compares the read 6 bytes with a hardcoded value using memcmp.
   - call memcmp

The hardcoded value being compared against is loaded from the .rodata section of libshadow.so. Inspecting the .rodata section confirms the target string:

```
$ objdump -s -j .rodata libshadow.so
```

```
...
 2050 2f70726f 632f7365 6c662f65 78650043  /proc/self/exe.C
```

```
 2060 48524f4d 4100                        HROMA.
```

The comparison is against the 6-byte string **CHROMA**.

The Conditional Jump

The system_check function checks the return value of memcmp. If the comparison is successful (memcmp returns 0), a jump is taken to call secret_path.

Condition	Action
memcmp(...) == 0	Call secret_path
memcmp(...) != 0	Call normal_path

To trigger the hidden path, we must patch the shadow_core binary at offset 0x3000 with the string CHROMA.

3. Decrypting the Flag in secret_path

Before patching, we analyze the secret_path function to understand how the flag is generated. Disassembly shows that the function contains encrypted data and a decryption loop.

Decryption Logic

The function initializes a 20-byte encrypted payload and a rolling XOR key. The decryption loop iterates over the payload, performing a three-way XOR operation:

$$
\text{DecryptedByte} = \text{EncryptedByte} \oplus \text{KeyByte} \oplus
\text{RollingKey}
$$

The RollingKey is then updated to the current EncryptedByte for the next iteration.

Key Values from Disassembly:

Variable	Value	Assembly Source
Initial KeyByte (val_3d)	0x1f	Derived from CHROMA (0x4f524843) through a series of shifts and XORs.
Initial RollingKey (val_3e)	0x56	Hardcoded value.

Variable	Value	Assembly Source
Encrypted Data	20 bytes	Hardcoded movabs instructions.

The encrypted data is: 1f 40 19 46 27 4e 3c 12 6c 46 2b 6e 16 3d 49 37 72 0c 6f 1b 35 42 69 10 55 3d 14 7d 57 21 4d 64 03 00 (Note: The assembly loads 8-byte chunks, which are combined here).

Python Decryption Script

We can replicate the decryption logic in a Python script to recover the flag:

```python
import struct
```

```python
def decrypt():
    # KeyByte (val_3d) derived from 'CHROMA' (0x4f524843)
    val_38 = 0x4f524843
    eax = val_38 >> 0x10
    eax ^= val_38
    val_34 = eax
    eax = val_34 >> 0x8
    eax ^= val_34
    val_3d = (eax ^ 0x0c) & 0xFF # KeyByte: 0x1f

    # Encrypted data from movabs instructions
    data = struct.pack("<QQQQH",
                       0x123c4e274619401f,
                       0x37493d166e2b466c,
                       0x106942351b6f0c72,
                       0x644d21577d143d55,
                       0x0003)

    # Initial RollingKey (val_3e)
    xor_key = 0x56

    flag = ""
    for i in range(len(data)):
        encrypted_byte = data[i]
        if encrypted_byte == 0:
            break

        # Decryption: DecryptedByte = EncryptedByte ^ KeyByte ^
RollingKey
        decrypted_byte = encrypted_byte ^ val_3d ^ xor_key
        flag += chr(decrypted_byte)
        # Update RollingKey: RollingKey = EncryptedByte
        xor_key = encrypted_byte

    print(f"Decrypted Flag: {flag}")

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

Running this script yields the flag: SECE{sh4d0w_b1nd_dyn4m1c_r3s0lv3}.

4. Final Solution: Patching the Binary

Although the flag was recovered through decryption, the intended solution is to trigger the hidden execution path by patching the binary.

We use the dd command to write the string CHROMA at the required offset 0x3000 (12288 decimal) in a copy of the shadow_core executable:

```
# Create a copy of the executable
cp shadow_core shadow_core_patched

# Patch the file at offset 12288 (0x3000) with "CHROMA"
echo -n "CHROMA" | dd of=shadow_core_patched bs=1 seek=12288 conv=notrunc

# Grant execution permissions and run the patched binary
chmod +x shadow_core_patched
export LD_LIBRARY_PATH=.
./shadow_core_patched
```

The output confirms the successful execution of the hidden path:

```
=== Shadow Bind Service ===
[*] Service mode: INTERACTIVE
[!] Debug mode enabled
[*] Loading internal modules...
[*] Performing integrity validation...
[!] Hidden routine unlocked:
SECE{sh4d0w_b1nd_dyn4m1c_r3s0lv3}
```

```
[*] Service ready
```

The flag is **SECE{sh4d0w_b1nd_dyn4m1c_r3s0lv3}**.

References

[1] Linux Man Page. /proc/self/exe. Path to the executable file of the running process.
[2] GNU Binutils.objdump. Utility for displaying information about object files.
[3] GNU Binutils.nm. Utility for listing symbols from object files.
[4] GNU Coreutils.dd. Utility for converting and copying a file.
[5] C Standard Library.memcmp. Function for comparing blocks of memory.
[6] C Standard Library.lseek. Function for repositioning the read/write file offset.