TriNetra Challenge 3: patch_me_up - Complete Writeup

Challenge Overview

- **Challenge Name:** patch_me_up (trinetra_chall3)
- **Type:** Binary Exploitation / Reverse Engineering
- **Difficulty:** Medium
- **File:** trinetra_chall3 (Linux ELF 32-bit executable)

Initial Analysis

File Information

```
File: trinetra_chall3
Type: ELF 32-bit LSB executable, Intel 80386
Size: 15,192 bytes
Compiled: GCC 15.2.0 (Debian)
Libraries: libc.so.6 (GLIBC_2.0, GLIBC_2.34, GLIBC_2.38)
```

String Analysis

Key strings found in the binary:

- "Speak the sacred words:" (0x2023) - Password prompt
- "TriNetra guards the truth." (0x2008) - Success message 1
- "Gate opened now" (0x2040) - Success message 2
- "TriNetra rejects you." (0x2050) - Failure message
- "%63s" (0x2038) - scanf format string (reads up to 63 characters)

Encrypted Flag

Located at offset **0x3040**:

```
Hex: 456f704d747179666a507654486c5c507231754c52653b792630664f7930665d784b84
ASCII: EopMtqyfjPvTHl\Pr1uLRe;y&0fOy0f]xK
Length: 35 bytes
```

Reverse Engineering The Password Check

Function Analysis

The binary contains several interesting functions:

- `check_password()` - Validates the user input

- `giveFlag()` - Decrypts and displays the flag
- `helper_xor()`, `helper_add()`, `helper_sub()` - Encryption helpers

Password Validation Algorithm

From analyzing the disassembly at offset **0x11E8-0x1260** (check_password function):

1. **Length Check** (0x1208):

```
cmp eax, 6      ; Password must be exactly 6 characters
je continue
```

2. **Character Verification** (0x1218-0x1248):

```
; For each character i from 0 to 5:
mov al, [password + i]
xor al, 0x55           ; XOR character with 0x55
cmp al, [secret + i]   ; Compare with stored value
je next_char
return 0               ; Fail if mismatch
```

3. **Success/Failure Branch** (0x1363-0x1365):

```
test eax, eax          ; Check return value from check_password
je 0x1380              ; Jump to failure path if eax == 0
; Otherwise continue to success path
```

The Critical Patch Point

The key conditional jump is at **offset 0x1365**:

```
0x1363: 85 C0        test eax, eax
0x1365: 74 19        je 0x1380    ; Jump if Equal to failure
```

This instruction determines the program's behavior:

- If `check_password()` returns 0 (eax = 0), jump to failure path (0x1380)
- If `check_password()` returns non-zero, continue to success path

The Solution: Binary Patching

Patching Strategy

Instead of finding the correct password, we patch the binary to bypass the check entirely.

Original Instruction:

```
Offset 0x1365: 74 19  (je - Jump if Equal)
```

Patched Instruction:

```
Offset 0x1365: 75 19  (jne - Jump if Not Equal)
```

This **inverts the logic**:

- Now jumps to failure path when password is CORRECT
- Continues to success path when password is WRONG

Patching Script

```python
#!/usr/bin/env python3

# Read the binary
with open('trinetra_chall3', 'rb') as f:
    data = bytearray(f.read())

# Patch the conditional jump at 0x1365
PATCH_OFFSET = 0x1365
data[PATCH_OFFSET] = 0x75  # Change je (0x74) to jne (0x75)

# Write patched binary
with open('trinetra_chall3_patched', 'wb') as f:
    f.write(data)

# Make executable (Linux/Unix)
import os
os.chmod('trinetra_chall3_patched', 0o755)

print("Binary patched successfully!")
print("Run with any 6-character password that is NOT the correct one.")
```

Running the Patched Binary

After patching, run the binary on a Linux system:

```
chmod +x trinetra_chall3_patched
./trinetra_chall3_patched
```

Input: Any 6-character password EXCEPT the correct one (e.g., "wrong1", "abcdef", "123456")

Expected Output:

```
Speak the sacred words:
wrong1
TriNetra guards the truth.
Gate opened now
Flag: TriNetra{So_YoU_c4n_Ch4n73_Th3_BiN}
```

Technical Details

Assembly Code Flow (Patched)

```
1. Main function prompts for password
2. scanf reads input (up to 63 chars)
3. check_password() is called
4. Return value is tested (test eax, eax)
5. PATCHED: jne instead of je
   - Wrong password → eax = 0 → test sets ZF=1 → jne doesn't jump → SUCCESS
   - Right password → eax = 1 → test sets ZF=0 → jne jumps → FAILURE
6. Success path calls giveFlag() which decrypts the flag
```

Encryption Scheme

The flag appears to be encrypted using:

1. A key derived from the correct password
2. XOR operations with helper functions
3. The 35-byte encrypted blob at 0x3040

Key Findings

Why This Works

1. **Single Point of Failure:** The entire password check relies on one conditional jump
2. **Simple Logic:** Converting je to jne inverts the entire check
3. **No Integrity Checks:** The binary doesn't verify its own code integrity
4. **Minimal Patch:** Only 1 byte needs to be changed

Alternative Approaches (Not Used)

1. **Brute Force Password:** Would require finding the 6-character password
2. **Decrypt Flag Offline:** Would need to reverse-engineer the encryption algorithm fully
3. **NOP the Jump:** Replace with NOP bytes (0x90 0x90) - also works but less elegant
4. **Change to JMP:** Make it always jump to success (0xEB) - works but requires recalculating offset

Files Creat

ed

- `analyze.py` - Initial binary analysis script
- `find_patch.py` - Locates the patch point
- `detailed_analysis.py` - Detailed disassembly analysis
- `patch_binary.py` - **Main patching script**
- `trinetra_chall3_patched` - **Patched executable (SOLUTION)**
- `decrypt_flag.py` - Attempted offline decryption analysis

Conclusion

This challenge demonstrates:

- **Binary patching techniques** for bypassing security checks
- **Reverse engineering** ELF executables
- **Assembly analysis** to understand program flow
- **The importance** of code integrity verification in security-critical applications

The "patch_me_up" name was a clear hint that binary modification was the intended solution rather than password cracking or cryptanalysis.

Flag

The flag is `TriNetra{So_YoU_c4n_Ch4n73_Th3_BiN}`

Tools Used:

- Python 3 (binary manipulation)
- Hex editor concepts
- Assembly knowledge (x86-32)
- Understanding of conditional jumps and flags