Attack on Titan CTF Challenge - Technical Analysis

Binary Overview

Filename: `titan_challenge`
Architecture: x86_64
Compiler: GCC with anti-debugging features
Language: C with obfuscation techniques

Anti-Analysis Measures

Anti-Debugging Implementation

The binary implements several anti-debugging techniques in `init_anti_debug()`:

```
// Ptrace self-attach detection
if (ptrace(PTRACE_TRACEME, 0, 1, 0) == -1) {
    printf("🛡️ Titan hardening detected!\n");
    exit(1);
}
```

Bypass: Patch the ptrace call or use LD_PRELOAD to intercept it.

Timing Analysis

```
// Single-stepping detection via timing
gettimeofday(&start, NULL);
// ... dummy operations ...
gettimeofday(&end, NULL);
if (microseconds > 50000) {
    printf("⏱️ Time anomaly in the Paths detected!\n");
    exit(1);
}
```

Bypass: Patch the timing check or run without debugger stepping.

Process Environment Detection

```
// Environment variable analysis
const char* suspicious_env[] = {"LINES", "COLUMNS", "_", "SHLVL", NULL};
if (suspicious_count >= 4) {
```

```
    printf("🎯 Abnormal environment detected!\n");
    exit(1);
}
```

Bypass: Unset environment variables before execution.

TracerPid Detection (Linux)

```
// Check /proc/self/status for debugging
FILE* status = fopen("/proc/self/status", "r");
if (strncmp(line, "TracerPid:", 10) == 0) {
    int pid = atoi(line + 10);
    if (pid != 0) exit(1);
}
```

Bypass: Hook file operations or modify /proc filesystem.

Memory Layout Analysis

Global Variables

```
static volatile int paths_coordinate = 0;
static volatile int coordinate_system = 0x13579BDF;
static volatile int eldian_memory = 0xCAFEBABE;
```

These serve as integrity check values and anti-tampering mechanisms.

Encrypted Password Arrays

Wall Maria (Method 1 Encryption)

```
unsigned char wall1_answer[] = {0xEC, 0xF9, 0xE7, 0xFB, 0xF8, 0xF9, 0xE8,
0xE2, 0x00};
```

Decryption Algorithm:

```
for (int i = 0; i < len; i++) {
    buffer[i] = ((data[i] ^ i) ^ KEY_FOUNDING) - 3;
}
```

Analysis:

- XOR with array index (position-dependent)
- XOR with KEY_FOUNDING (0xAA)
- Subtract 3 from result
- Produces "COLOSSAL"

Wall Rose (Method 2 Encryption)

```
unsigned char wall2_answer[] = {0x52, 0x50, 0x4E, 0x58, 0x5F, 0x4D, 0x51,
0x00};
```

Decryption Algorithm:

```
for (int i = 0; i < len; i++) {
    int rev_idx = len - 1 - i;
    buffer[i] = ((data[rev_idx] ^ KEY_FEMALE) - i) ^ KEY_CART;
}
```

Analysis:

- Process array in reverse order
- XOR with KEY_FEMALE (0x55)
- Subtract current index
- XOR with KEY_CART (0x45)
- Produces "ARMORED"

Wall Sheena (Method 3 Encryption)

```
unsigned char wall3_answer[] = {0xEB, 0x29, 0xBE, 0x17, 0xEE, 0x00};
```

Decryption Algorithm:

```
if (i == 0) buffer[i] = data[i] ^ KEY_FOUNDING ^ 0x03;
else if (i == 1) buffer[i] = data[i] ^ KEY_CART ^ 0x29;
else if (i == 2) buffer[i] = data[i] ^ KEY_FEMALE ^ KEY_FOUNDING;
else if (i == 3) buffer[i] = data[i] ^ KEY_WARHAMMER ^ KEY_JAW;
else if (i == 4) buffer[i] = data[i] ^ KEY_FOUNDING ^ KEY_CART ^ KEY_FEMALE;
```

Analysis:

- Position-specific XOR operations
- Different key combinations per character

- Produces "BEAST"

Function Analysis

Main Execution Flow

```
int main() {
    init_anti_debug();           // Anti-debugging setup
    check_integrity();           // Memory integrity verification
    fake_check_routine();        // Decoy function
    display_banner();            // UI presentation

    // Execute three challenges sequentially
    for (int i = 0; i < 3; i++) {
        challenges[i]();         // wall_maria/rose/sheena_challenge()
        // Integrity checks between challenges
    }

    display_victory();           // Flag revelation
}
```

Validation Function

```
int validate_answer(const char* input, unsigned char* encrypted, int method) {
    int len = 0;
    while (encrypted[len] != 0 && len < 255) len++;  // Calculate length

    char* decrypted = decrypt_complex(encrypted, len, method);
    return (strcmp(input, decrypted) == 0);
}
```

Vulnerabilities:

- No input length validation
- Buffer overflow potential in decrypt_complex()
- Timing attack possible via strcmp()

Integrity Check System

```
int check_integrity() {
    int check1 = (coordinate_system ^ 0xDEADBEEF) ^ 0xDEADBEEF;
    int check2 = rot_left(eldian_memory, 13) ^ 0x5A5A5A5A;
```

```
    return (check1 == 0x13579BDF) &&
           (check2 == rot_left(0xCAFEBABE, 13) ^ 0x5A5A5A5A);
}
```

Analysis:

- Verifies global variable integrity
- Uses bit rotation for obfuscation
- Can be bypassed by maintaining expected values

Flag Construction Anti-Analysis

Runtime String Construction

The flag is built character-by-character to avoid static detection:

```
char hidden_flag[64];
int idx = 0;

hidden_flag[idx++] = 'C';
hidden_flag[idx++] = 'T';
hidden_flag[idx++] = 'F';
// ... continues for full flag
hidden_flag[idx] = '\0';
```

Anti-Static Analysis Techniques:

- No string literals containing the flag
- Runtime construction only
- Memory-only existence
- Immune to `strings` command

Memory Analysis

Flag Location: Stack-allocated in `display_victory()`
Lifetime: Temporary, destroyed on function exit
Detection: Requires dynamic analysis or memory dumps during execution

Reverse Engineering Approach

Static Analysis Strategy

1. **Identify Key Functions**:

```
objdump -t titan_challenge | grep -E "(wall|challenge|validate)"
```

2. **Extract Encrypted Arrays**:

```
objdump -s -j .data titan_challenge
```

3. **Analyze Decryption Logic**:

```
objdump -d titan_challenge | grep -A 20 "decrypt_complex"
```

Dynamic Analysis Strategy

1. **Bypass Anti-Debugging**:

```
# Patch ptrace call
echo -e '\x90\x90\x90\x90\x90' | dd of=titan_challenge bs=1
seek=$PTRACE_OFFSET conv=notrunc
```

2. **Hook Validation Function**:

```
// LD_PRELOAD hook for strcmp
int strcmp(const char *s1, const char *s2) {
    printf("Comparing: '%s' vs '%s'\n", s1, s2);
    return original_strcmp(s1, s2);
}
```

3. **Memory Breakpoints**:

```
gdb ./titan_challenge
(gdb) break validate_answer
(gdb) break display_victory
```

Automated Analysis

Decryption Script

```
def decrypt_method1(data, key=0xAA):
    result = ""
    for i, byte in enumerate(data[:-1]):  # Exclude null terminator
        decrypted = ((byte ^ i) ^ key) - 3
        result += chr(decrypted)
    return result

def decrypt_method2(data, key_female=0x55, key_cart=0x45):
    result = ""
    length = len(data) - 1
    for i in range(length):
```

```
        rev_idx = length - 1 - i
        decrypted = ((data[rev_idx] ^ key_female) - i) ^ key_cart
        result += chr(decrypted)
    return result

def decrypt_method3(data, keys):
    result = ""
    key_founding, key_cart, key_female, key_warhammer, key_jaw = keys

    for i, byte in enumerate(data[:-1]):
        if i == 0:
            decrypted = byte ^ key_founding ^ 0x03
        elif i == 1:
            decrypted = byte ^ key_cart ^ 0x29
        elif i == 2:
            decrypted = byte ^ key_female ^ key_founding
        elif i == 3:
            decrypted = byte ^ key_warhammer ^ key_jaw
        elif i == 4:
            decrypted = byte ^ key_founding ^ key_cart ^ key_female
        else:
            decrypted = byte
        result += chr(decrypted)
    return result

# Usage
wall1_data = [0xEC, 0xF9, 0xE7, 0xFB, 0xF8, 0xF9, 0xE8, 0xE2, 0x00]
wall2_data = [0x52, 0x50, 0x4E, 0x58, 0x5F, 0x4D, 0x51, 0x00]
wall3_data = [0xEB, 0x29, 0xBE, 0x17, 0xEE, 0x00]

print("Wall 1:", decrypt_method1(wall1_data))      # COLOSSAL
print("Wall 2:", decrypt_method2(wall2_data))      # ARMORED
print("Wall 3:", decrypt_method3(wall3_data, [0xAA, 0x45, 0x55, 0x77, 0x33]))
# BEAST
```

Security Assessment

Vulnerabilities

1. **Buffer Overflow**: `fgets()` input not properly bounded
2. **Format String**: Potential in printf statements
3. **Race Conditions**: Anti-debugging checks can be raced
4. **Logic Bypass**: Integrity checks can be patched

Defensive Measures

1. **Code Obfuscation**: Function pointer indirection
2. **Control Flow Integrity**: Checksum verification
3. **Anti-Tampering**: Multiple integrity verification points
4. **Environmental Detection**: Debugger and analysis tool detection

Bypass Techniques

1. **Static Patching**: Modify anti-debugging checks
2. **Dynamic Hooking**: Intercept system calls
3. **Emulation**: Run in controlled environment
4. **Memory Manipulation**: Direct memory access to bypass checks

Conclusion

This CTF challenge demonstrates several important reverse engineering concepts:

- **Multi-layer encryption** with different algorithms per challenge
- **Effective anti-static analysis** through runtime string construction
- **Comprehensive anti-debugging** using multiple detection vectors
- **Thematic integration** balancing entertainment with technical depth

The challenge can be solved through either domain knowledge (Attack on Titan lore) or technical analysis (reverse engineering the encryption), making it accessible to different skill levels while maintaining educational value.