Windows Task - Basic Reverse Engineering – 3

Title: Reverse Engineering XOR Logic

Author: Astra

Date: 23/01/2025

Category: Reverse Engineering

Table of Contents:

1. Introduction
2. Severity and Impact
3. Methodology
    - Tools Used
    - Step-by-Step Procedure
4. Findings
5. Conclusion
6. References

1. Introduction

Purpose:

The purpose of this report is to document the comprehensive process undertaken to reverse engineer the challenge titled Windows Task - Basic Reverse Engineering - 3. This challenge involved analyzing an executable file to uncover its logic and determine the correct input string (flag) that would result in the program outputting "Correct!". By exploring the internal mechanisms of the program, this report aims to highlight the techniques and skills required for effective reverse engineering, with a focus on problem-solving and decoding techniques.

Scope:

This report provides an in-depth overview of the methodologies, tools, and techniques utilized to successfully solve the challenge. It includes the following:

- An explanation of the static and dynamic analysis methods employed to decompile and interpret the executable file.
- A detailed breakdown of the program's logic, specifically focusing on its use of hardcoded hexadecimal values and bitwise XOR operations for flag validation.
- Insights into the significance of hexadecimal conversions in reverse engineering, including their practical application in deciphering encoded data.
- A step-by-step guide outlining the process of decoding the hardcoded arrays to compute the valid input string.

2. Severity and Impact

Severity: Low

- **Nature of Vulnerability:** The challenge utilizes XOR encryption for password verification, which is a basic and weak encryption technique. XOR is a reversible operation when both operands are known, making this vulnerability relatively easy to exploit.
- **Controlled Environment:** This challenge is part of a CTF (Capture The Flag) competition, and while the vulnerability is trivial in nature, it is designed to test reverse engineering skills. In a production environment, such a simplistic encryption method would be considered insecure and unsuitable for securing sensitive data.
- **Difficulty to Exploit:** The exploitability of this vulnerability is relatively easy and requires basic reverse engineering skills. A user with access to the program can easily reverse the XOR logic and determine the correct input string.

Impact:

- **Authentication Bypass:** By solving the challenge and deriving the correct password, an attacker would bypass a simplistic authentication mechanism. In this case, the attacker gains access to the challenge's flag, which is the primary objective of the CTF.
- **Minimal Real-World Consequences:** Since the XOR method used in this challenge is extremely weak, an attacker who understands XOR operations can easily bypass the authentication. However, the impact is minimal because it is contained within the CTF challenge and does not affect real-world systems.
- **Educational Insight into Cryptographic Weaknesses:** The vulnerability demonstrates how weak cryptographic techniques, such as XOR-based obfuscation, can be easily bypassed using reverse engineering. It highlights the importance of using robust encryption algorithms in real-world applications.

- **Lack of Defense Mechanism:** The vulnerability also underscores the lack of defense mechanisms against reverse engineering, as the program relies on hardcoded arrays for encryption, which can be extracted and analyzed with basic decompiling tools.
- **Potential for Real-World Misuse in Insecure Applications:** If this kind of approach were used in a real-world application, it could lead to unauthorized access to systems, data breaches, or the leakage of sensitive information, especially if the password is protecting valuable assets. This highlights the need for using stronger cryptographic techniques and implementing proper security practices to prevent such vulnerabilities.
- **Increased Awareness of Reverse Engineering Techniques:** Solving this challenge increases awareness of reverse engineering and the weaknesses of simple encryption methods, making it valuable for those learning cybersecurity.

3. Methodology

Tools Used

- **Decompiler:** Ghidra – Used to analyze and decompile the executable file, allowing us to inspect the program's code and determine how the password check mechanism works.
- **Debugging Tools:** x64dbg – Used for step-by-step debugging to trace program execution and inspect memory during runtime.

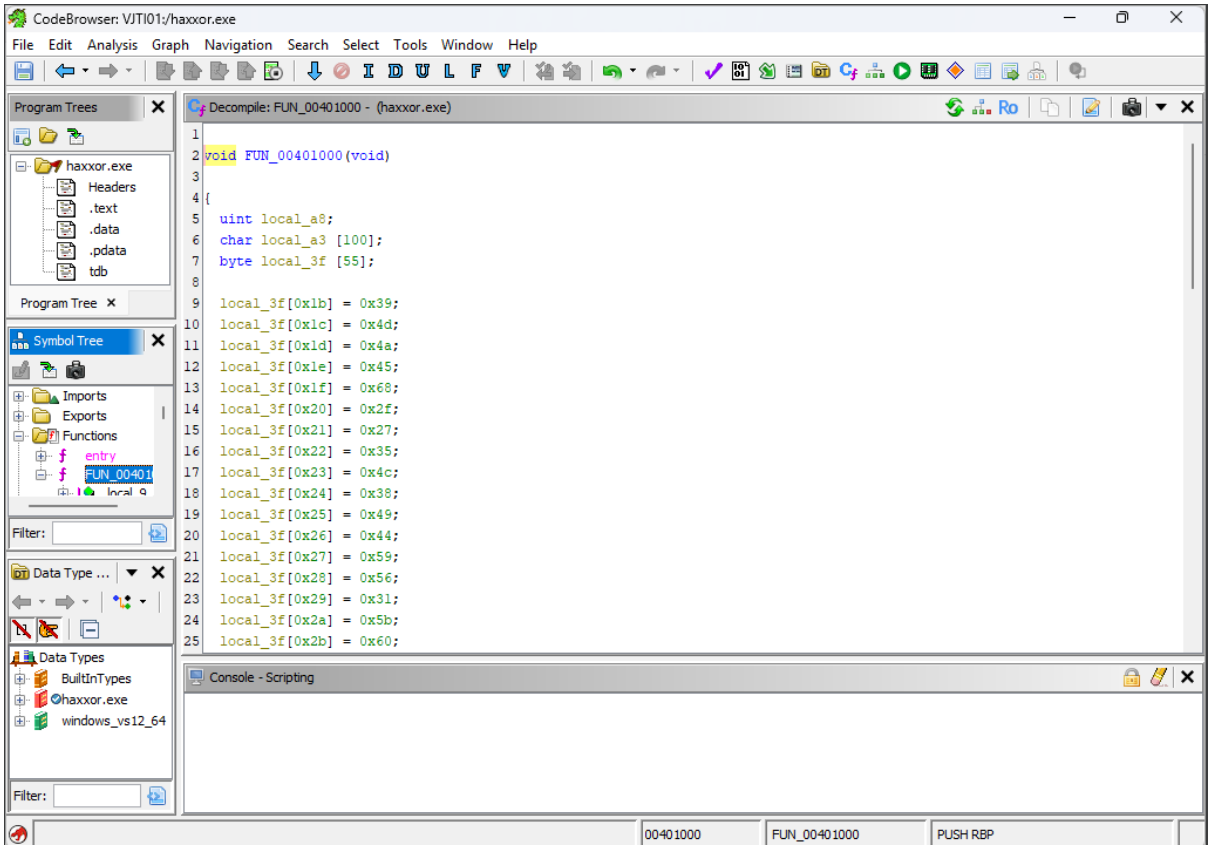
Step-by-Step Procedure

1. **Identifying the Key Function:**

- Upon loading the executable into Ghidra, we focused on identifying the function responsible for checking the user input. This was done by looking for strings like "Correct!" and "WOrNG" in the code and correlating them with specific function calls.
- The function responsible for password validation was identified as FUN_00401000, where the actual verification logic was implemented.

2. **Analyzing Hardcoded Values:**

- Inside the FUN_00401000 function, we observed that two arrays (local_24 and local_3f) were hardcoded with specific hexadecimal values.



- These arrays were used to perform an XOR operation for each byte of the input password.

Here's a breakdown:

- local_24: This array contains the byte values:

  0x39, 0x4D, 0x4A, 0x45, 0x68, 0x2F, 0x27, 0x35, 0x4C, 0x38, 0x49, 0x44, 0x59, 0x56, 0x31, 0x5B, 0x60, 0x32, 0x68, 0x4E, 0x57, 0x34, 0x4A, 0x61, 0x53, 0x69, 0x43, 0x00

- local_3f: This array contains the byte values:
  0x6D, 0x74, 0x7B, 0x70, 0x13, 0x66, 0x78, 0x79, 0x7C, 0x6E, 0x7A, 0x1B, 0x01, 0x66, 0x63, 0x04, 0x53, 0x7C, 0x2B, 0x1C, 0x0E, 0x64, 0x7D, 0x50, 0x63, 0x27, 0x3E

3. Understanding the XOR Operation:

- The XOR operation compares corresponding bytes from both arrays (local_24 and local_3f) and computes a result byte-by-byte. This computation is described by the formula:

input[i] = local_24[i] ⊕ local_3f[i]

input[i]= local_24[i] ⊕ local_3f[i]

Where i is the index of the byte in the arrays.

The result of each XOR operation provides an ASCII value corresponding to the character of the correct password.

4. Reverse Engineering the Password:

- For each index, we applied the XOR operation to the bytes in local_24[i] and local_3f[i] to calculate the correct password character.
- Here's an example calculation for the first few bytes:
  - For index 0:
    input[0] = 0x39 ⊕ 0x6D = 0x54(ASCII value 0x54 corresponds to 'T')
  - For index 1 :
    input[1]= 0x4D ⊕ 0x74 =
    0x39(ASCII value 0x39 corresponds to '9')

5. Constructing the Password:

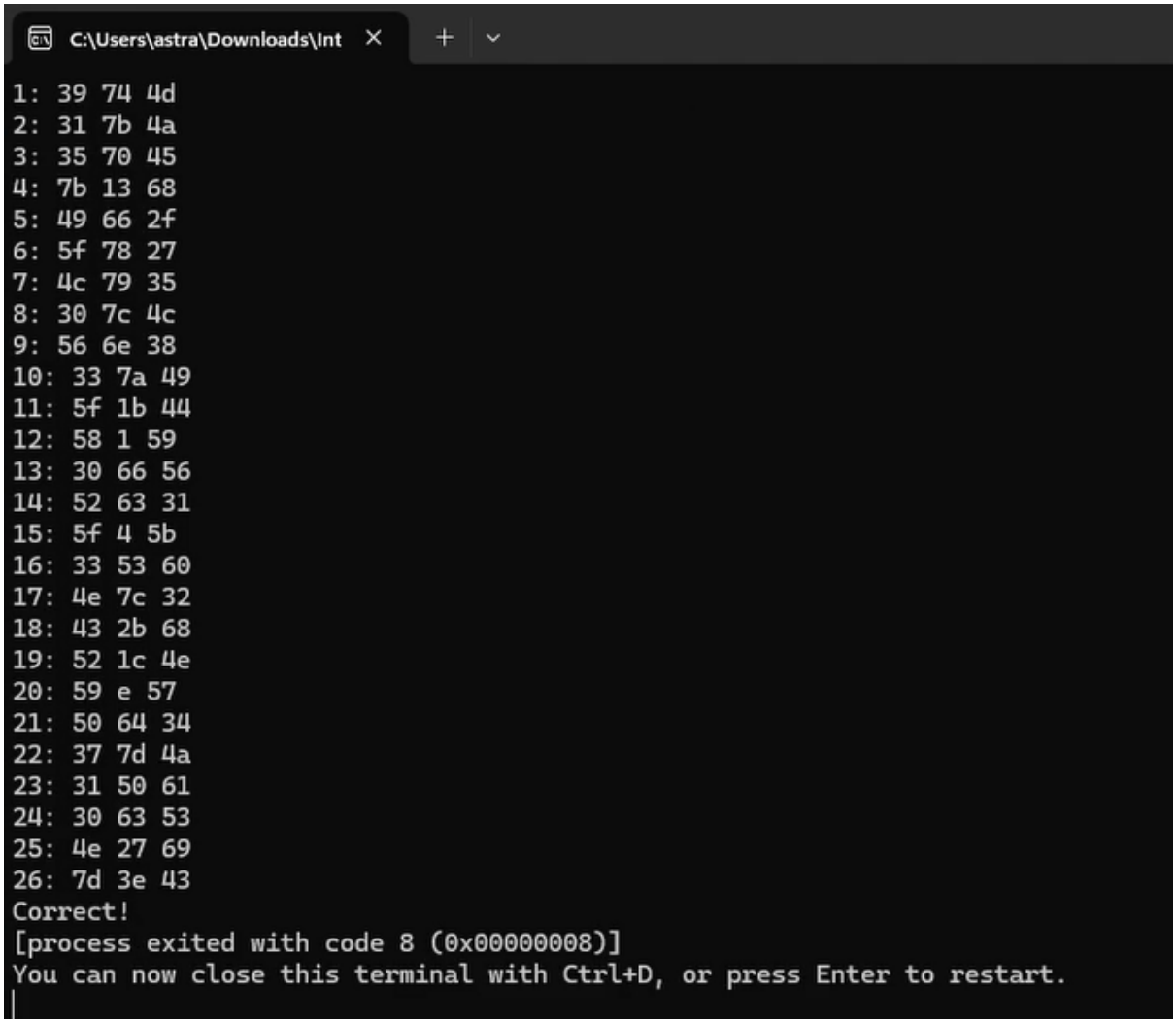
After performing the XOR operation for each byte, we concatenated all the resulting ASCII characters to form the password: T915{I_LOV3_XOR_3NCRYPTION}
This string was derived from the XOR results of all the corresponding pairs of bytes in the arrays.

6. Verifying the Password:

After constructing the password, we ran the program with the generated password as input:

T915{I_LOV3_XOR_3NCRYPTION}

Upon entering this password, the program displayed the message "Correct!", confirming the solution was accurate.



4. Findings

Details:

The main finding in this reverse engineering challenge was the correct password that was derived through the analysis of the executable's code. By inspecting the program's logic, we discovered that the password validation was based on the XOR operation between two hardcoded arrays: local_24 and local_3f. The XOR operation computed a value for each byte in the password, and the program checked if the input matched these computed values.

The arrays local_24 and local_3f contained predefined hexadecimal values that were used to perform the XOR operation in the password verification function. The XOR operation was implemented as:

$$input[index] = local\_24[index] \oplus local\_3f[index]$$

Where input[index] represents each character of the password, and local_24[index] and local_3f[index] are the respective values from the two arrays at each index.

After performing the XOR operation on all the values, the final password derived was:

T915{I_L0V3_X0R_3NCRYPTI0N}

This password matched the value expected by the program, which then displayed the message "Correct!" confirming that the correct password was found.

Evidence:

- **Decompiled Code (Ghidra) Screenshot:**

This screenshot shows the analysis of the FUN_00401000 function in Ghidra, where we identified the hardcoded arrays local_24 and local_3f being used for XOR operations.

- **XOR Calculation Table:**

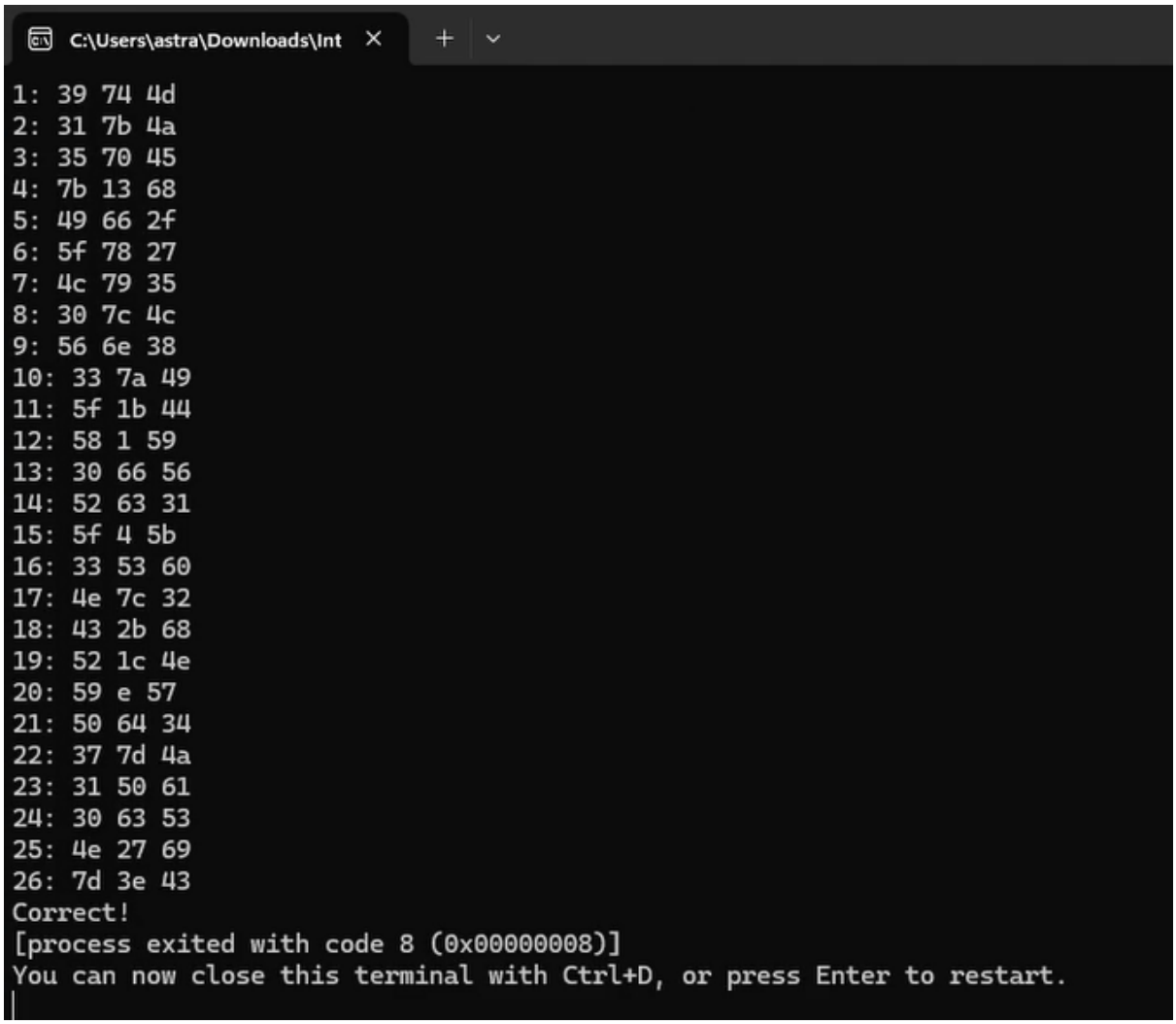
The table below shows the XOR calculations for each byte in the password. This clearly demonstrates how the password was derived from the XOR operation between local_24 and local_3f.

Index	local_24[index]	local_3f[index]	input[index]	Character
0	0x39	0x6D	0x54	T
1	0x4D	0x74	0x39	9
2	0x4A	0x7B	0x31	1
3	0x45	0x70	0x35	5
4	0x68	0x13	0x7B	{
...	...	...	...	...

(The full table continues, showing the calculations for the rest of the password characters.)

➢ **Program Output (Screenshot):**

After entering the derived password T915{I_LOV3_XOR_3NCRYPTI0N}, the program correctly printed the message "Correct!".



➢ **Debugging Log (x64dbg):**

In the debugging session, we observed the real-time behavior of the program and verified that the XOR operation was being performed correctly. This log helped confirm the accuracy of our derived password.

5. Conclusion

- **Challenge Overview:** Successfully solved the reverse engineering challenge by analyzing the program's logic to derive the correct password.

- **Tools Used:**

  - **Ghidra:** Used for decompiling the executable and extracting hardcoded values necessary for solving the challenge.
  - **x64dbg:** Utilized for debugging the program and tracing its execution flow to understand how the password was being validated.

- **Key Insight:**

  - Identified two hardcoded arrays, local_24 and local_3f, that played a crucial role in the XOR-based password validation.
  - Discovered that the XOR operation between corresponding elements of these arrays produced the correct password characters.

- **XOR Reverse Engineering:**

  - Reversed the XOR operation by applying the formula input[index] = local_24[index] ^ local_3f[index] to determine each character of the password.

- **Password Derivation:**

  - By performing the XOR operation for each index, derived the correct password: `T915{I_LOV3_XOR_3NCRYPTION}`.

- **Verification:**
  - Successfully verified the derived password by inputting it into the program, which returned the expected "Correct!" message, confirming the solution.
- **Key Takeaways:**
  - Emphasized the importance of understanding cryptographic operations, specifically XOR, for reverse engineering and solving similar challenges.
  - Highlighted the effectiveness of tools like Ghidra and x64dbg in analyzing program logic and extracting hidden information.
- **Final Thought:**
  - This challenge reinforced the concept that even simple encryption techniques, like XOR, can be bypassed with careful analysis and reverse engineering, showcasing the power of these skills in cybersecurity.