CrackMe Challenge Write-up

1. Introduction

Challenge Name: CrackMe

Category: Reverse Engineering

Objective: Reverse engineer the binary to find the correct password.

Reverse engineering challenges require us to understand how a program works internally without access to its source code. We analyze the binary file to figure out how it processes inputs. In this case, we need to find the password that the program checks to allow us to "log in."

2. Initial Binary Analysis

The provided file is a 64-bit ELF executable for Linux. It's not stripped, meaning the function names are still present, which makes reverse engineering easier. We can inspect the file type using the command:

```
file crackme
```

This reveals that the binary is a 64-bit ELF executable, providing clues about the architecture and runtime environment.

3. Using Ghidra for Reverse Engineering

Ghidra, an open-source reverse engineering tool, helps decompile binary code back into a human-readable form. Here's how to use it for this challenge:

1. Open Ghidra and import the crackme binary.
2. Allow Ghidra to analyze the binary automatically.

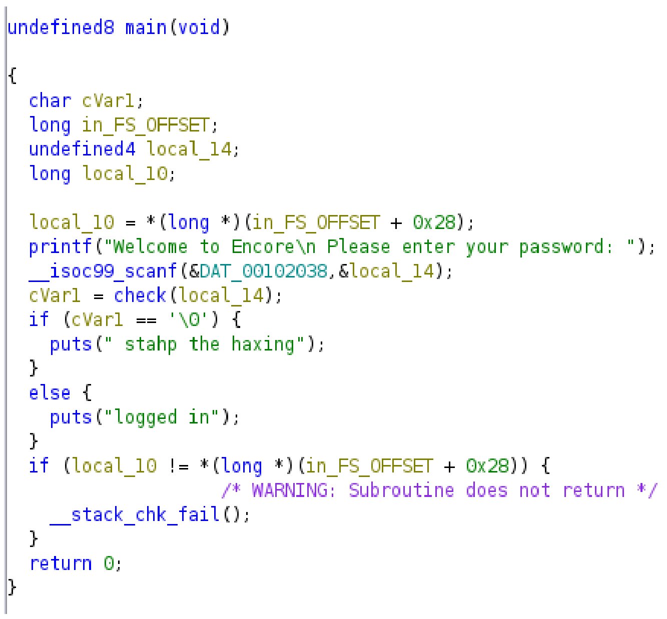
3. After analysis, open the Symbol Tree and locate the `main` function, which is the entry point of the program.

4. Analyzing the Main Function

Upon decompiling the `main` function, we observe:

- The program prints a welcome message.
- It prompts the user to input a password using `scanf`.
- The entered password is passed to a function named `check`.
- The `check` function compares the input password to a hardcoded value.
- If the password is correct, the program prints "logged in." Otherwise, it prints "stahp the haxing."

This reveals that the `check` function is the key to solving the challenge.



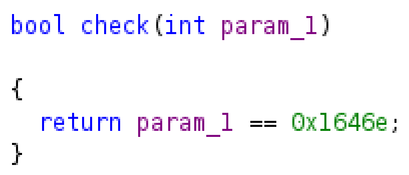
```
undefined8 main(void)

{
  char cVar1;
  long in_FS_OFFSET;
  undefined4 local_14;
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  printf("Welcome to Encore\n Please enter your password: ");
  __isoc99_scanf(&DAT_00102038,&local_14);
  cVar1 = check(local_14);
  if (cVar1 == '\0') {
    puts(" stahp the haxing");
  }
  else {
    puts("logged in");
  }
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return 0;
}
```

5. Exploring the Check Function

The `check` function is responsible for validating the password. Decompiling this function, we see:

```
bool check(int param_1) {
    return param_1 == 0x1646e;
}
```

The function compares the user's input with the hardcoded value `0x1646e`. If the password matches this hexadecimal value, the program returns success.



```
bool check(int param_1)

{
  return param_1 == 0x1646e;
}
```

6. Converting Hexadecimal to Decimal

In programming, numbers are sometimes represented in hexadecimal format. The value `0x1646e` is the hexadecimal representation of the password.

We convert it to decimal:

- `0x1646e (hex)` = `91246 (decimal)`

Thus, the correct password for this challenge is 91246.

7. Testing the Password

Now that we've identified the password, we can test it by running the program:

```
./crackme
```

When prompted for the password, enter 91246. If the reverse engineering was correct, the program will print `logged in`, confirming the success.

8. Understanding the Logic Behind the Program

Here's a breakdown of the logic:

- The program asks for a password.
- It passes the input to the `check` function.
- The `check` function compares the password with a hardcoded value (`0x1646e`).
- If the comparison is successful, the program prints "logged in"; otherwise, it rejects the input.
- By reverse engineering the binary, we identified the hardcoded password and used it to solve the challenge.

9. Lessons

- **Reverse Engineering Tools**: Tools like Ghidra are invaluable for decompiling and analyzing binaries.
- **Hexadecimal to Decimal Conversion**: Often, important values (like passwords) are stored in hexadecimal format, requiring conversion for easier understanding.
- **Function Analysis**: By focusing on specific functions (like `check`), we can trace the logic of the program and reverse engineer its behavior.

10. Conclusion

In this challenge, we used Ghidra to decompile a binary and identify the hardcoded password. This challenge introduced basic reverse engineering techniques, the process of analyzing binary logic, and the importance of understanding different number systems in security challenges.

Tools Used:

- **Ghidra**: For decompiling and analyzing the binary.
- **Hexadecimal Converter**: For converting the password from hex to decimal.