Lab 11: Binary Analysis

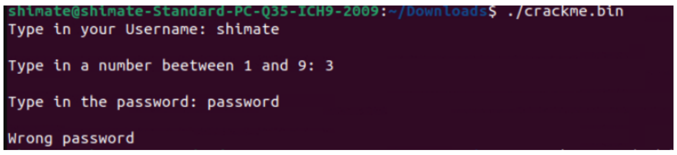
Shivali Mate

I. Introduction

THIS lab introduces the use of Ghidra, to inspect compiled machine code, interpret assembly logic, and modify program behavior. Binary analysis enables an in-depth understanding of how executable programs behave without relying on their source code. The provided binary crackme.bin accepts a username, a numeric key, and a password, and the objective was to analyze, reconstruct, and patch the binary to reveal its internal logic and hidden password validation mechanism. Through this task, the process of static reverse engineering, patching, and debugging was explored to understand how compiled logic corresponds to high-level functionality [1][2].

II. Methodology

Binary analysis involves examining compiled executables to uncover underlying operations such as control flow, data manipulation, and program logic. The initial step required loading crackme.bin [4] into Ghidra, where the disassembled and decompiled code was reviewed to locate the main function and understand how inputs were handled. The program requested three user inputs username, number, and password and used these to perform a comparison check as shown in Fig. 1.



```
shimate@shimate-Standard-PC-Q35-ICH9-2009:~/Downloads$ ./crackme.bin
Type in your Username: shimate

Type in a number beetween 1 and 9: 3

Type in the password: password

Wrong password
```

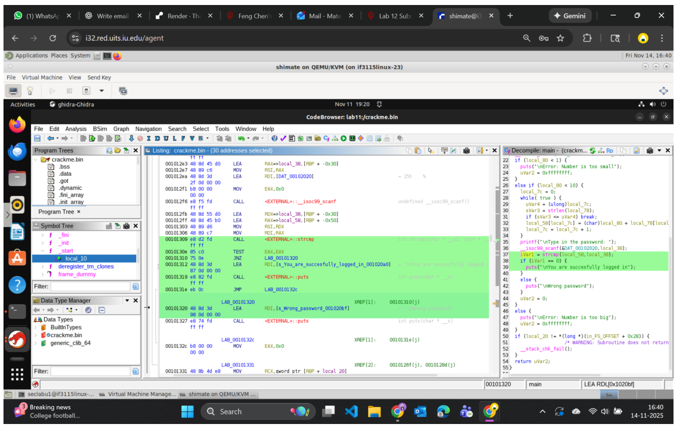
Fig. 1. Execution Output of crackme.bin

A. Using Ghidra for Analysis

Ghidra was installed and initialized to analyze the ELF 64-bit binary. It automatically generated the program's symbol tree, memory layout, and function references. The Decompiler window translated assembly code into high-level C-like syntax, making it easier to identify conditional jumps and library calls. By tracing function references and external calls like scanf, puts, and strcmp, the password verification logic was located in the main function. This environment enabled the identification of the exact instructions responsible for password comparison and validation as shown in Fig. 2 [2].

B. Code Analysis of Password Logic

The reconstructed C code shown below shows how the program generates the expected password by adding the entered number to each character of the username. The resulting string is then compared with the user-supplied password using strcmp(). When the correct transformation is applied, the



CodeBrowser: lab11:/crackme.bin
Listing: crackme.bin - (30 addresses selected)
Decompile: main - (crackme.bin)

```
001012e3 48 8d 45 d0    LEA     RAX=>local_38,[RBP + -0x30]
001012e7 48 89 c6       MOV     RSI,RAX
001012ea 48 8d 3d       LEA     RDI,[DAT_00102020]                          = 25h    %
         2f 0d 00 00
001012f1 b8 00 00       MOV     EAX,0x0
         00 00
001012f6 e8 f5 fd       CALL    <EXTERNAL>::__isoc99_scanf                  undefined __isoc99_scanf()
         ff ff
001012fb 48 8d 55 d0    LEA     RDX=>local_38,[RBP + -0x30]
001012ff 48 8d 45 b0    LEA     RAX=>local_58,[RBP + -0x50]
00101303 48 89 d6       MOV     RSI,RDX
00101306 48 89 c7       MOV     RDI,RAX
00101309 e8 d2 fd       CALL    <EXTERNAL>::strcmp                          int strcmp(char * __s1, char * ...
         ff ff
0010130e 85 c0          TEST    EAX,EAX
00101310 75 0e          JNZ     LAB_00101320
00101312 48 8d 3d       LEA     RDI,[s_You_are_succesfully_logged_in_001020a0]  = "\nYou are succesfully logged ...
         87 0d 00 00
00101319 e8 82 fd       CALL    <EXTERNAL>::puts                            int puts(char * __s)
         ff ff
0010131e eb 0c          JMP     LAB_0010132c

                    LAB_00101320                    XREF[1]:     00101310(j)
00101320 48 8d 3d       LEA     RDI,[s_Wrong_password_001020bf]             = "\nWrong password"
         98 0d 00 00
00101327 e8 74 fd       CALL    <EXTERNAL>::puts                            int puts(char * __s)
         ff ff

                    LAB_0010132c                    XREF[1]:     0010131e(j)
0010132c b8 00 00       MOV     EAX,0x0
         00 00

                    LAB_00101331                    XREF[2]:     0010126f(j), 0010128d(j)
00101331 48 8b 4d e8    MOV     RCX,qword ptr [RBP + local_20]
```

```
  if (local_80 < 1) {
    puts("\nError: Number is too small");
    uVar2 = 0xffffffff;
  }
  else if (local_80 < 10) {
    local_7c = 0;
    while( true ) {
      uVar4 = (ulong)local_7c;
      sVar3 = strlen(local_78);
      if (sVar3 <= uVar4) break;
      local_58[local_7c] = (char)local_80 + local_78[local
      local_7c = local_7c + 1;
    }
    printf("\nType in the password: ");
    __isoc99_scanf(&DAT_00102020,local_38);
    iVar1 = strcmp(local_58,local_38);
    if (iVar1 == 0) {
      puts("\nYou are succesfully logged in");
    }
    else {
      puts("\nWrong password");
    }
    uVar2 = 0;
  }
  else {
    puts("\nError: Number is too big");
    uVar2 = 0xffffffff;
  }
  if (local_20 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return
    __stack_chk_fail();
  }
  return uVar2;
}
```

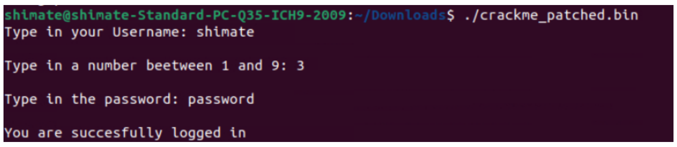
Fig. 2. Ghidra Decompiler and Listing View of crackme.bin

program prints "You are successfully logged in." This logic confirms that the password is a shifted variant of the username. The compiled C representation of this functionality is provided below:

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    int number;
    int i;
    char username[32];
    char generated_password[32];
    char input_password[24];

    printf("Type in your Username: ");
    scanf("%s", username);

    printf("\nType in a number between
        1 and 9: ");
    scanf("%d", &number);

    if (number < 1) {
        puts("\nError: Number is too small");
        return -1;
    }
    else if (number < 10) {
        i = 0;
        while (1) {
            generated_password[i] =
                username[i] + (char)number;
            i++;
        }

        printf("\nType in the password: ");
        scanf("%s", input_password);

        strcmp(generated_password,
            input_password);
        puts(generated_password);
        return 0;
    }
    else {
        puts("\nError: Number is too big");
        return -1;
    }

    return 0;
}
```

C. Patching for Bypassing Password Validation

To bypass authentication, the conditional jump following the strcmp instruction (JNZ LAB_00101320) was modified so that the program would always execute the success message, regardless of the password entered. This patch redirected program control to the success output unconditionally. After exporting the modified binary, running it demonstrated successful login output even for incorrect passwords, as shown in Fig. 3.



```
shimate@shimate-Standard-PC-Q35-ICH9-2009:~/Downloads$ ./crackme_patched.bin
Type in your Username: shimate

Type in a number beetween 1 and 9: 3

Type in the password: password

You are succesfully logged in
```

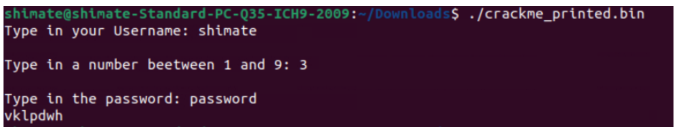
Fig. 3. Execution Output of crackme_patched.bin showing successful login

D. Patching for Password Display

A second patch was performed to display the generated password instead of only validating it. The instruction sequence following the strcmp call was replaced with

```
        LEA RDI,[RBP + -0x50];
        CALL <puts>
```

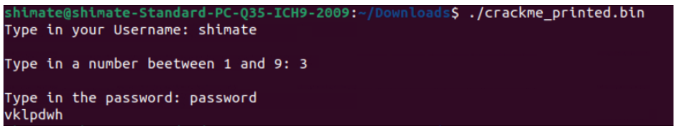
The puts operation is called, which prints the buffer storing the internally computed password. Since Ghidra's assembler only supports positive offsets, the offset was written asto like `[RBP + -0x50]` to reference the same memory location. The value `0x50` corresponds to the stack offset where the local variable `local_58` the generated password buffer is stored relative to the base pointer `RBP`. The negative sign indicates that the variable is located below the base pointer within the function's stack frame, which is typical for local variables.



```
shimate@shimate-Standard-PC-Q35-ICH9-2009:~/Downloads$ ./crackme_printed.bin
Type in your Username: shimate

Type in a number beetween 1 and 9: 3

Type in the password: password
vklpdwh
```

Fig. 4. Execution output of crackme_printed.bin displaying the password

The modified binary displayed the correct password corresponding to the given username and number Fig. 4, which was later verified by running the original program with that password Fig. 5.



```
shimate@shimate-Standard-PC-Q35-ICH9-2009:~/Downloads$ ./crackme_printed.bin
Type in your Username: shimate

Type in a number beetween 1 and 9: 3

Type in the password: password
vklpdwh
```

Fig. 5. Execution output of original crackme executed with the password

III. Tools and Learning Experience

The lab used Ghidra 12.x as the primary reverse-engineering framework, alongside the Linux terminal for executing patched binaries. Through this exercise, the process of exploring assembly logic, mapping registers like RBP, RDI, and EAX, and converting low-level operations into human-readable logic was practiced. The use of Ghidra's Decompiler and Listing views simplified pattern recognition between assembly and C syntax. Understanding of stack frames, memory offsets, and conditional branches was enhanced while performing live binary patching and testing [2][3].

IV. Findings and Discussion

The analysis revealed that the binary employed a simple Caesar-style shift algorithm to generate passwords. The patching tasks demonstrated how program logic can be altered by editing conditional instructions and leveraging stack memory to print hidden data. These techniques mirror common software-cracking methodologies used in security analysis and malware reverse engineering. In practice, similar approaches are applied in vulnerability research and software debugging. However, the process also highlighted potential tool limitations, Ghidra occasionally rejects certain assembly syntaxes, requiring manual workarounds, and the necessity for careful handling to prevent corrupted exports or segmentation faults [3].

V. Conclusion

This lab provided hands-on experience in static binary analysis and code patching using Ghidra. It demonstrated how compiled programs can be disassembled, logically reconstructed, and modified to alter runtime behavior. The exercise enhanced understanding of low-level execution, memory addressing, and instruction manipulation. Through reverse-engineering the crackme.bin program, the fundamental principles of password-generation algorithms, stack referencing, and conditional jump control were observed, forming a foundational skill set for advanced reverse-engineering and binary exploitation studies.

References

[1] Evans, D. *Guide to x86 Assembly Language.* University of Virginia. Available at: http://www.cs.virginia.edu/~evans/cs216/guides/x86.html

[2] National Security Agency. *Ghidra: Software Reverse Engineering Framework.* GitHub Repository. Available at: https://github.com/NationalSecurityAgency/ghidra

[3] Wikipedia. *TEST (x86 instruction).* Available at: https://en.wikipedia.org/wiki/TEST_(x86_instruction)

[4] Indiana University Canvas, Lab 11 Assets: crackme.bin, available under course files for Cyber Defense Laboratory.