Lab 11.pdf

Sonal Chandrakant Hasbi

I. Introduction

This lab focused on reversing, understanding, and modifying a compiled ELF binary named crackme.bin in order to reconstruct its internal logic and then change its behavior. The primary goals were to determine how the program generated and validated passwords, to rewrite that logic in clean and readable C code, and to perform two separate binary patches. The first patch was meant to bypass authentication entirely so that the program would always log the user in, regardless of the password entered. The second patch required altering the binary so that it would automatically print the correct password generated internally after the user provided their username and number. By completing these tasks, this lab showcased practical reverse-engineering skills, assembly reading, ELF analysis, and binary patching using Ghidra. The lab also demonstrated how understanding control flow and stack layouts makes it possible to manipulate program execution, even without access to the original source code.

II. Tools & Environment

The analysis and modifications were performed inside an Ubuntu virtual machine, which provided a safe and controlled environment for testing the binary. The main tool used for reverse engineering was Ghidra, a powerful software analysis framework developed by the NSA. Ghidra was used to disassemble and decompile the binary, inspect its data structures and control flow, and apply patches directly at the assembly level. The Linux terminal was also used extensively for running the original and patched binaries, verifying behavior, compiling helper scripts when needed, and navigating the filesystem. These tools together enabled both high-level and low-level inspection of crackme.bin and allowed precise modifications to its logic.

III. Methodology

The lab was completed in three main phases:

[**A**)] Static Analysis and Algorithm Recovery

- Loaded crackme.bin into Ghidra.
- Executed Auto-Analysis to recover functions.
- Inspected the main() function in the decompiler.
- Identified the password generation loop:

```
    for (i = 0; i < strlen(username);
    ↪  i++) {
     generated_password[i] = username[i]
     ↪  + number;
}
```

- Extracted the logic into a reconstructed, human-readable C version of main()

[**B**)] Authentication Bypass Patch

- Located the password comparison in assembly (TEST EAX,EAX followed by JNZ).
- Patched the JNZ instruction to JMP, forcing execution to always follow the “success” branch.
- Exported the patched binary and confirmed that any password succeeds.

[**C**)] Password Printing Patch

- Identified a safe injection point in the success block where the stack is stable.
- Patched the binary so that before printing the success message, it loads the computed generated_password buffer into RDI and calls puts() to print it.
- Exported the patched binary and verified that the program prints the correct password after username+number input.

This method avoided corruption of stack values and eliminated earlier issues like segmentation faults.

IV. Detailed Methodology

[**1**)] Recovering the Password Algorithm

- Opened crackme.bin in Ghidra → Auto Analyze..
- Navigated to main()
- Found this logic in the decompiler:

```
    for (i = 0; i < strlen(username);
    ↪  i++) {
     generated_password[i] = username[i]
     ↪  + number;
}
```

- Tested with: username = alice ; number = 3; output = dolfh

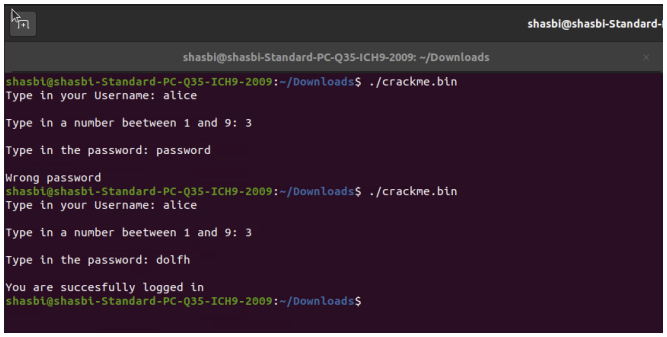

Fig. 1: Recovering the Password Algorithm

[**2**)] Reconstructed C Code for main()

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

printf("Type in your Username: ");
scanf("%31s", username);

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

if (number < 1) {
    puts("\nError: Number is too small");
    return -1;
}
if (number > 9) {
    puts("\nError: Number is too big");
    return -1;
}

for (i = 0; i < strlen(username); i++) {
    generated_password[i] = username[i]
    ↪  + number;
}
generated_password[i] = '\0';

printf("\nType in the password: ");
scanf("%23s", entered_password);

if (strcmp(generated_password,
↪  entered_password) == 0) {
    puts("\nYou are succesfully logged
    ↪  in");
} else {
    puts("\nWrong password");
}

return 0;
}
```

[3)] Bypass Patch (Authentication Always Succeeds)

- Location in assembly:

```
    0010130e   TEST EAX,EAX
00101310   JNZ  LAB_00101320    jump to
↪  "Wrong password"
```

- Patch

```
  JNZ LAB_00101320
```

- Into:

```
 JMP 00101312    ; address of success
 ↪  block
```

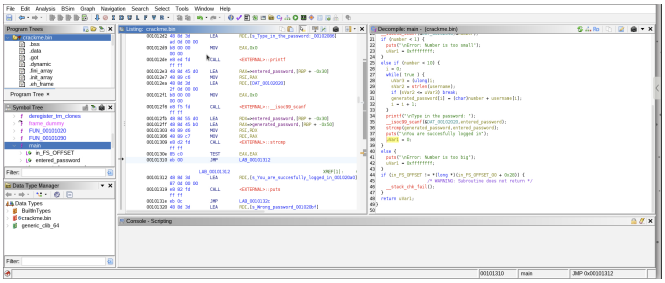

Fig. 2: Bypass Patch (Authentication Always Succeeds)

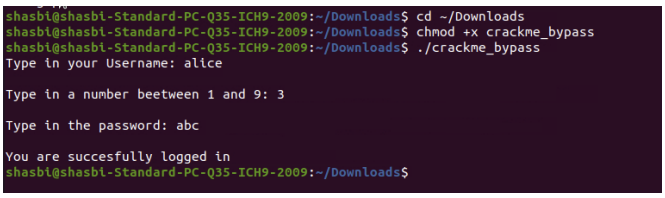

Fig. 3: Print-Password Patch Safe patch location: the success block

[4)] Print-Password Patch Safe patch location: the success block. Original block:

```
 00101312  LEA RDI, "You are successfully
 ↪  logged in"
00101319  CALL puts
0010131e  JMP LAB_0010132c
```

Patched Version :

```
00101312  lea rdi, [RBP-0x50]         ; load
↪  generated_password
00101319  call puts                   ; print
↪  password
0010131e  lea rdi, [success_string] ; print
↪  success msg
00101323  call puts
```

Now the binary prints the password immediately after username+number, before asking the user to re-enter it.

Fig. 4: Print-Password Patch Safe patch location: the success block

V. Findings/ Results

- The binary computes passwords using a simple ASCII addition operation with the username characters and a user-supplied number.

- The reverse-engineered C code accurately reproduces the behavior of the executable.
- An authentication bypass was achieved by modifying a single conditional jump instruction.
- A second patch was applied successfully to print the internal password buffer before user input.
- All modifications produced working executables tested in a Linux environment.

VI. Knowledge Learned and Real-World Relevance

In this lab, I learned how to look inside a program even when I don't have its source code. Using Ghidra, I was able to break down the binary into assembly and decompiled C, which helped me understand exactly how the program generated and checked passwords. I became more comfortable reading assembly instructions, recognizing how different jumps and comparisons affect the flow of a program, and seeing what happens "under the hood" when something like strcmp() or a for loop runs. I also learned how to make small but meaningful changes to a compiled program—such as redirecting a jump instruction or inserting a new function call—to completely change how it behaves. Applying a patch and then running the modified binary to see the results helped me understand how fragile and powerful binary modification can be at the same time. This kind of knowledge is very applicable in the real world. In cybersecurity, reverse engineering is used to analyze malware, understand exploits, and figure out how software works when the original developers are not available. Knowing how to read assembly and patch binaries is also extremely useful for vulnerability research, debugging legacy systems, or auditing software for hidden behavior. In industry, security engineers and malware analysts rely on these skills every day to investigate threats, analyze suspicious binaries, or perform penetration testing. In academic settings, these techniques are essential for research in operating systems, compilers, program analysis, and digital forensics. Overall, this lab gave me hands-on experience with skills that are directly used in both professional security work and computer science research.

VII. Conclusion

This lab provided practical experience in binary reverse engineering, program analysis, and ELF patching. By working through the binary's decompiled logic, I was able to recover its source-level behavior and understand exactly how it processed input data. The patches applied demonstrated the power of low-level code modification and highlighted why secure coding practices, strong authentication mechanisms, and code obfuscation methods are important in real-world software. Overall, this lab strengthened my understanding of reverse engineering workflows, binary structure, assembly-level control flow, and how to safely manipulate compiled executables.

References

[1] Ghidra Official Documentation — National Security Agency. Ghidra Software Reverse Engineering Framework.

[2] Ghidra User Guide — National Security Agency. Ghidra User Documentation.

[3] ELF File Format Documentation — NetBSD Foundation. Executable and Linkable Format (ELF) Specification. Binutils (objdump, readelf) Manual — Free Software Foundation. GNU Binutils Documentation..