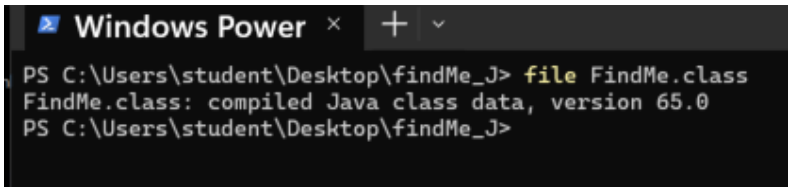
FindMe_J Challenge

In this challenge, we were tasked with reverse engineering a Java .class file (FindMe.class) to capture a hidden flag. The focus was on selecting and using appropriate tools to achieve this goal, without identifying vulnerabilities or applying mitigations.

Step 1:
To verify the file type, we used the file command '**file FindMe.class',** to inspect the .class file format.



This confirmed the file was a valid Java .class file, compiled with bytecode version 65.0 (Java 21).

Step 2: Bytecode Disassembly
After confirming the file format, we used the ***Javap*** tool to disassemble the bytecode and inspect the program's structure and logic. Before disassembling, we verified that the version of Javap was compatible with the .class file by checking its version.

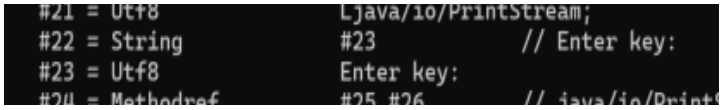
This confirmed that the Javap tool was version 21, matching the bytecode version 65.0 identified in step 1. Therefore, the file is compatible with the tools we used.

Next, We ran the following command: '**javap -verbose FindMe.class**' to disassemble the bytecode.

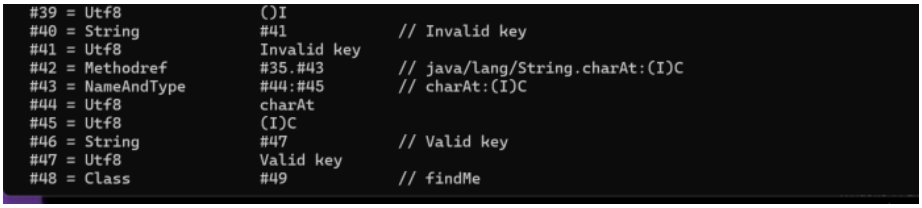
```
PS C:\Users\student\Desktop\findMe_J> javap -verbose FindMe.class
Classfile /C:/Users/student/Desktop/findMe_J/FindMe.class
  Last modified 18 Nov 2024; size 1520 bytes
  SHA-256 checksum b3b4ba2ff89a0e86ef12fd61140f4f70e26a9f04ff3fbb224944052947b7f0fb
  Compiled from "findMe.java"
public class findMe
  minor version: 0
  major version: 65
  flags: (0x0021) ACC_PUBLIC, ACC_SUPER
  this_class: #48                         // findMe
  super_class: #2                         // java/lang/Object
  interfaces: 0, fields: 0, methods: 2, attributes: 1
Constant pool:
   #1 = Methodref          #2.#3          // java/lang/Object."<init>":()V
   #2 = Class              #4             // java/lang/Object
   #3 = NameAndType        #5:#6          // "<init>":()V
   #4 = Utf8               java/lang/Object
   #5 = Utf8               <init>
   #6 = Utf8               ()V
   #7 = Class              #8             // java/util/Scanner
   #8 = Utf8               java/util/Scanner
   #9 = Fieldref           #10.#11        // java/lang/System.in:Ljava/io/InputStream;
  #10 = Class              #12            // java/lang/System
  #11 = NameAndType        #13:#14        // in:Ljava/io/InputStream;
  #12 = Utf8               java/lang/System
  #13 = Utf8               in
  #14 = Utf8               Ljava/io/InputStream;
  #15 = Methodref          #7.#16         // java/util/Scanner."<init>":(Ljava/io/InputStream;)V
  #16 = NameAndType        #5:#17         // "<init>":(Ljava/io/InputStream;)V
  #17 = Utf8               (Ljava/io/InputStream;)V
  #18 = Fieldref           #10.#19        // java/lang/System.out:Ljava/io/PrintStream;
  #19 = NameAndType        #20:#21        // out:Ljava/io/PrintStream;
  #20 = Utf8               out
  #21 = Utf8               Ljava/io/PrintStream;
  #22 = String             #23            // Enter key:
  #23 = Utf8               Enter key:
  #24 = Methodref          #25.#26        // java/io/PrintStream.println:(Ljava/lang/String;)V
  #25 = Class              #27            // java/io/PrintStream
  #26 = NameAndType        #28:#29        // println:(Ljava/lang/String;)V
  #27 = Utf8               java/io/PrintStream
  #28 = Utf8               println
  #29 = Utf8               (Ljava/lang/String;)V
  #30 = Methodref          #7.#31         // java/util/Scanner.nextLine:()Ljava/lang/String;
  #31 = NameAndType        #32:#33        // nextLine:()Ljava/lang/String;
  #32 = Utf8               nextLine
  #33 = Utf8               ()Ljava/lang/String;
  #34 = Methodref          #35.#36        // java/lang/String.length:()I
  #35 = Class              #37            // java/lang/String
  #36 = NameAndType        #38:#39        // length:()I
  #37 = Utf8               java/lang/String
  #38 = Utf8               length
  #39 = Utf8               ()I
  #40 = String             #41            // Invalid key
  #41 = Utf8               Invalid key
  #42 = Methodref          #35.#43        // java/lang/String.charAt:(I)C
  #43 = NameAndType        #44:#45        // charAt:(I)C
  #44 = Utf8               charAt
  #45 = Utf8               (I)C
  #46 = String             #47            // Valid key
  #47 = Utf8               Valid key
  #48 = Class              #49            // findMe
```

Findings:

- The program prompts the user to input a key with the message: *"Enter Key:"*.
- Checks the input length (must be exactly 23 characters).



The bytecode also contains the strings: *"Invalid key"* and *"Valid key"*, suggesting the program outputs these messages based on validation success or failure.



While the bytecode provided insight into the validation process, it did not directly reveal the flag. This indicated further investigation was necessary.

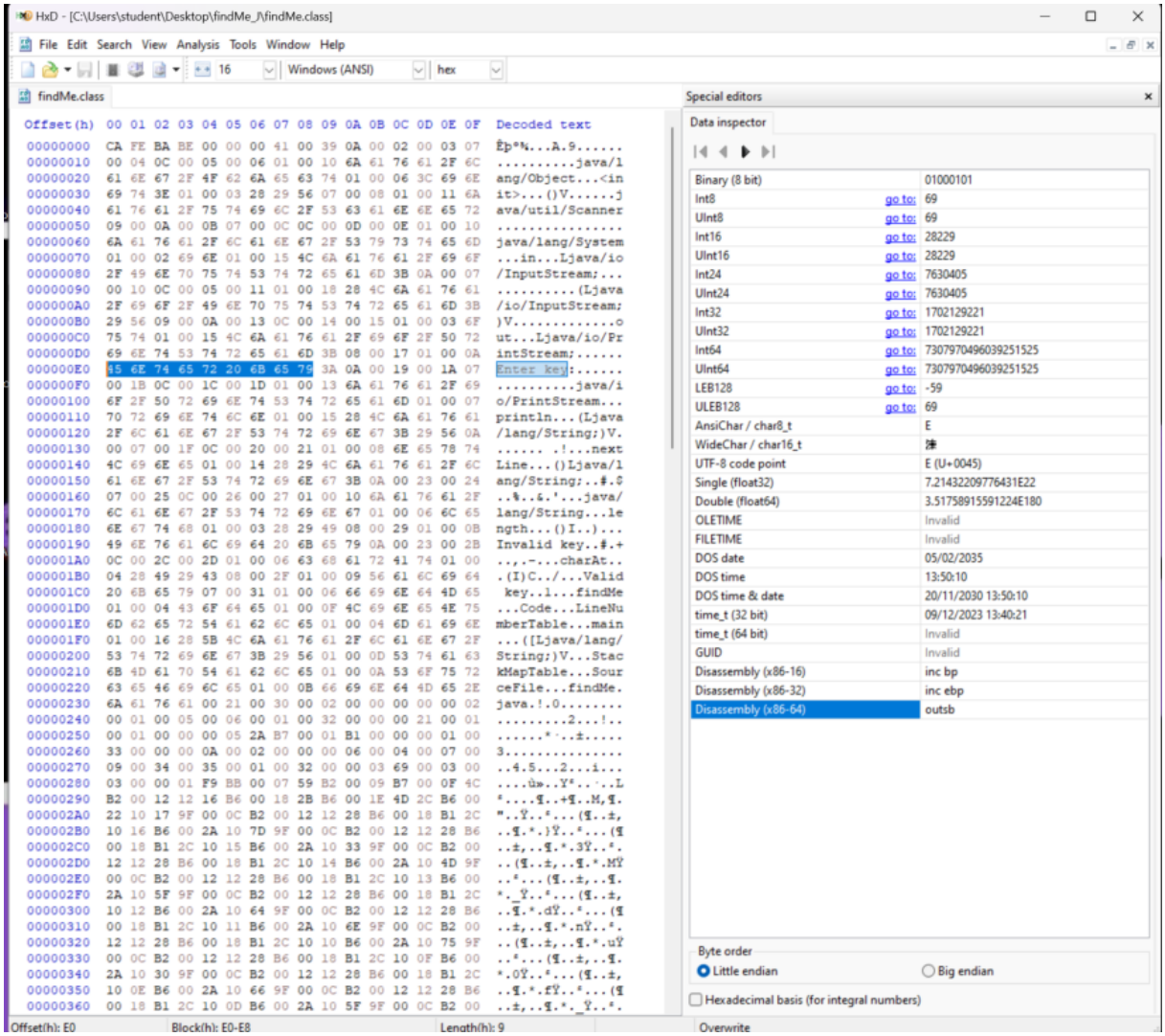
Step 3: Hex Editor Inspection

To explore further, we opened the FindMe.class file in the HxD Hex Editor to inspect its raw data and search for readable strings.

Findings:

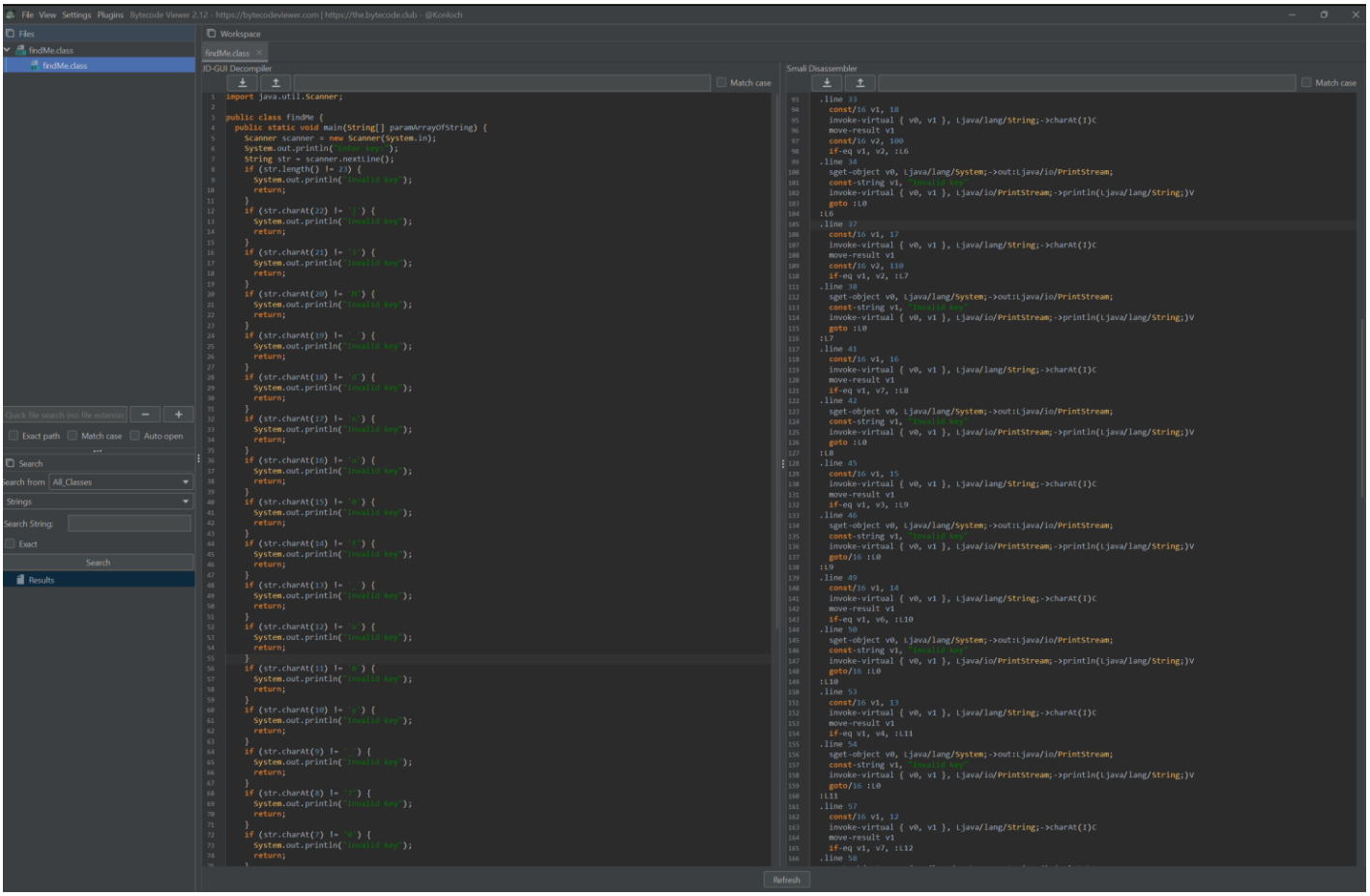
- Readable strings such as *"Enter key:"*, *"Invalid key"*, and *"Valid key"* were identified.
- These strings aligned with the messages seen in the disassembled bytecode but did not provide additional clues to uncover the flag.

We couldn't find any additional clues to uncover the flag and no hidden patterns or additional information were found beyond the readable strings.



Step 4: Decompiled Code Analysis

After inspecting the file with the hex editor and finding no additional clues, we used the Bytecode Viewer with the JD-GUI panel to decompile the FindMe.class file into readable Java code.



Findings:

- The decompiled code reveals that the program prompts the user to enter a key.
- The input must be exactly 23 characters.
- Each character in the input string is compared to a predefined sequence using the charAt method. The charAt() method allows for accessing specific characters in a string by index.
- The program performs validation sequentially, checking characters from position 0 to 22.
- If all conditions are satisfied, the program outputs **Valid key**. Otherwise, it outputs **Invalid key**.

By analysing the charAt checks, we reconstructed the correct key by identifying the characters at each position:

charAt(22): '}'
charAt(21): '3'
charAt(20): 'M'
charAt(19): '_'
charAt(18): 'd'
charAt(17): 'n'
charAt(16): 'u'
charAt(15): '0'
charAt(14): 'f'
charAt(13): '_'
charAt(12): 'u'
charAt(11): '0'
charAt(10): 'y'
charAt(9): '_'
charAt(8): '7'
charAt(7): '0'
charAt(6): '7'
charAt(5): '{'
charAt(4): 'f'
charAt(3): 't'
charAt(2): 'c'
charAt(1): 'R'
charAt(0): 'V'

The sequence derived from the bytecode is:
}3M_dnu0f_u0y_707{ftcRV.

Reversing this sequence from 0 -22, revealed the flag:
VRctf{707_y0u_f0und_M3}.

```java
import java.util.Scanner;

public class findMe {
   public static void main(String[] var0) {
      Scanner var1 = new Scanner(System.in);
      System.out.println("Enter key:");
      String var2 = var1.nextLine();
      if (var2.length() != 23) {
         System.out.println("Invalid key");
      } else if (var2.charAt(22) != '}') {
         System.out.println("Invalid key");
      } else if (var2.charAt(21) != '3') {
         System.out.println("Invalid key");
      } else if (var2.charAt(20) != 'M') {
         System.out.println("Invalid key");
      } else if (var2.charAt(19) != '_') {
         System.out.println("Invalid key");
      } else if (var2.charAt(18) != 'd') {
         System.out.println("Invalid key");
      } else if (var2.charAt(17) != 'n') {
         System.out.println("Invalid key");
      } else if (var2.charAt(16) != 'u') {
         System.out.println("Invalid key");
      } else if (var2.charAt(15) != '0') {
         System.out.println("Invalid key");
      } else if (var2.charAt(14) != 'f') {
         System.out.println("Invalid key");
      } else if (var2.charAt(13) != '_') {
         System.out.println("Invalid key");
      } else if (var2.charAt(12) != 'u') {
         System.out.println("Invalid key");
      } else if (var2.charAt(11) != '0') {
         System.out.println("Invalid key");
      } else if (var2.charAt(10) != 'y') {
         System.out.println("Invalid key");
      } else if (var2.charAt(9) != '_') {
         System.out.println("Invalid key");
      } else if (var2.charAt(8) != '7') {
         System.out.println("Invalid key");
      } else if (var2.charAt(7) != '0') {
         System.out.println("Invalid key");
      } else if (var2.charAt(6) != '7') {
         System.out.println("Invalid key");
      } else if (var2.charAt(5) != '{') {
         System.out.println("Invalid key");
      } else if (var2.charAt(4) != 'f') {
         System.out.println("Invalid key");
      } else if (var2.charAt(3) != 't') {
         System.out.println("Invalid key");
      } else if (var2.charAt(2) != 'c') {
         System.out.println("Invalid key");
      } else if (var2.charAt(1) != 'R') {
         System.out.println("Invalid key");
      } else if (var2.charAt(0) != 'V') {
         System.out.println("Invalid key");
      } else {
         System.out.println("Valid key");
      }
   }
}
```

Validation:
To verify if the flag is correct, we ran the program and entered the flag, which produced the output ‘Valid key’, confirming the flag was correct.

Entering a random key produced the output ‘Invalid key’.

```
cfg-vr@codefirstgirls:~/Desktop/findMe_J./findMe_J$ java findMe
Enter key:
VRctf{707_y0u_f0und_M3}
Valid key
cfg-vr@codefirstgirls:~/Desktop/findMe_J./findMe_J$ java findMe
Enter key:
abcdef123@
Invalid key
```

Conclusion:
We successfully captured the flag by using a combination of tools such as javap, HxD Hex Editor and Bytecode Viewer with JD-GUI. The key to finding the flag was carefully analysing the decompiled code and identifying the sequence of character checks for input validation. The flag, **VRctf{707_y0u_f0und_M3},** was hidden within the program's validation logic and was validated by running the program.

Tools:
Javap
HxD Hex Editor
Bytecode Viewer with JD-GUI

Reference:

https://javaalmanac.io/bytecode/versions/
https://www.freecodecamp.org/news/how-to-execute-and-run-java-code/
https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javap.html
https://shadowintel.medium.com/jvm-reverse-engineering-7607c471bdc4
https://www.geeksforgeeks.org/java-string-charat-method-example/
CFG session 11 slide – Overview of Java Bytecode
CFG session 12 slide – Decompilation Techniques and Tools