**Challenge Name: Code Breaker Challenge**

**Challenge made by: Yash Dinesh Singh**

**Challenge Description:**

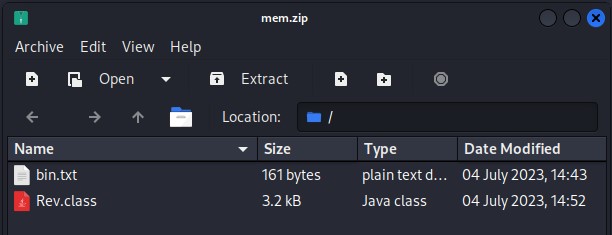
Welcome to the "Code Breaker Challenge"! In this reverse engineering challenge, your mission is to uncover a hidden flag by reverse engineering a Java class file. Prepare to put your reverse engineering skills to the test and decrypt the hidden message!

**Challenge Files:**

**"mem.zip"** - A zip file which contains Rev.class and bin.txt

**"Rev.class"** - An obfuscated Java class file containing the hidden flag and encrypted cipher.

**"bin.txt"** - A supplementary file that contains some binary.

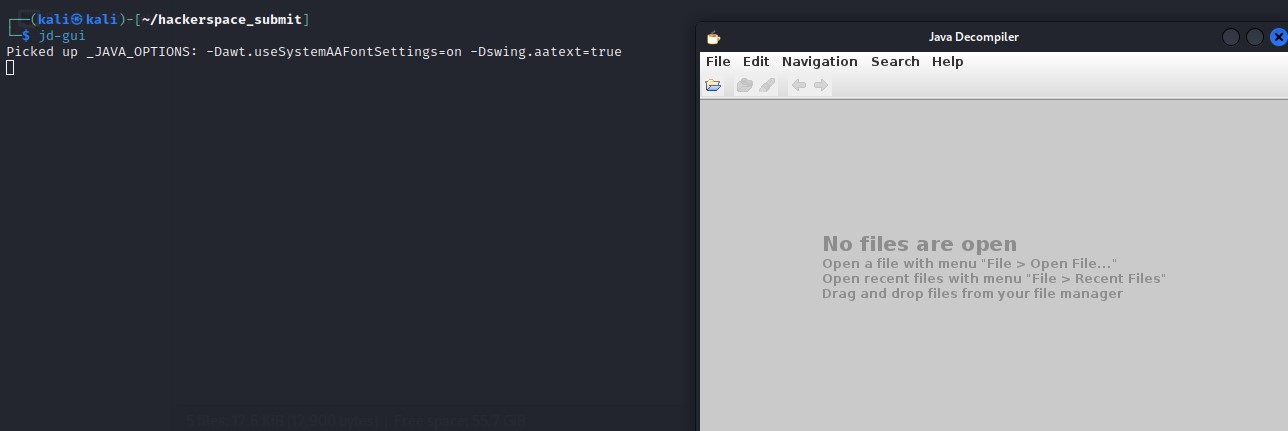


**Walkthrough Guide:**

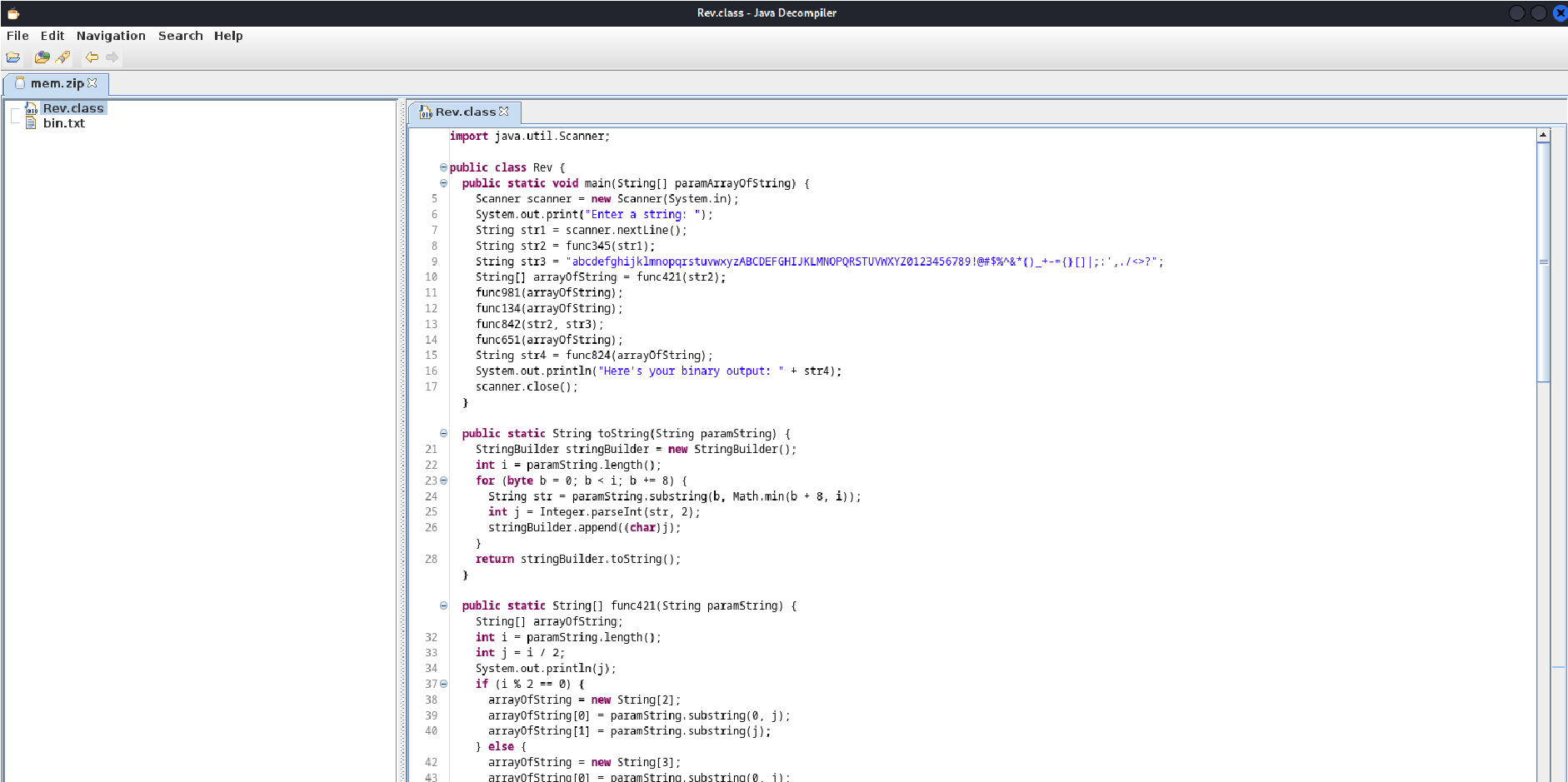
# Step 1: Initial Analysis

After extracting **mem.zip**, 2 files will get extracted, **Rev.class** and **bin.txt**. Begin by examining the **Rev.class** file using a Java bytecode decompiler, such as **JD-GUI** or **FernFlower**. This will allow you to view the decompiled Java source code.

To work with jd-gui, simply type jd-gui on the terminal and it should open up. Install it if necessary.



Press **Ctrl+O** and open **Rev.class**. (Tip: If it doesn’t open, just open **mem.zip** and from there you can access **Rev.class** and **bin.txt**.)



In **bin.txt** you will find a binary:

**10111110011100101000110011011100010011101010011011111110111001**

**10001011100001110000110110111111101100110011001110010011101110**

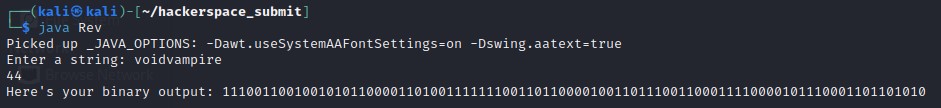
**011011010010100001101100011000000110**

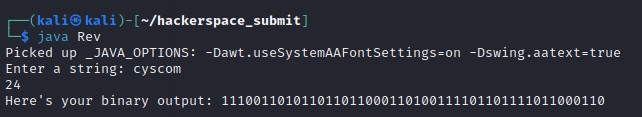
# Step 2: Reverse Engineering

Analyze the decompiled code, paying close attention to the "main" method and any other relevant methods.

For now I will copy the decompiled code into **VSCode** and start analyzing.

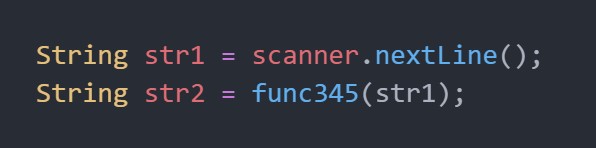
Initially, run the file to check it’s working. The code prompts the user to enter a string, performs various operations and returns an integer and a binary. For example, I passed some test strings.



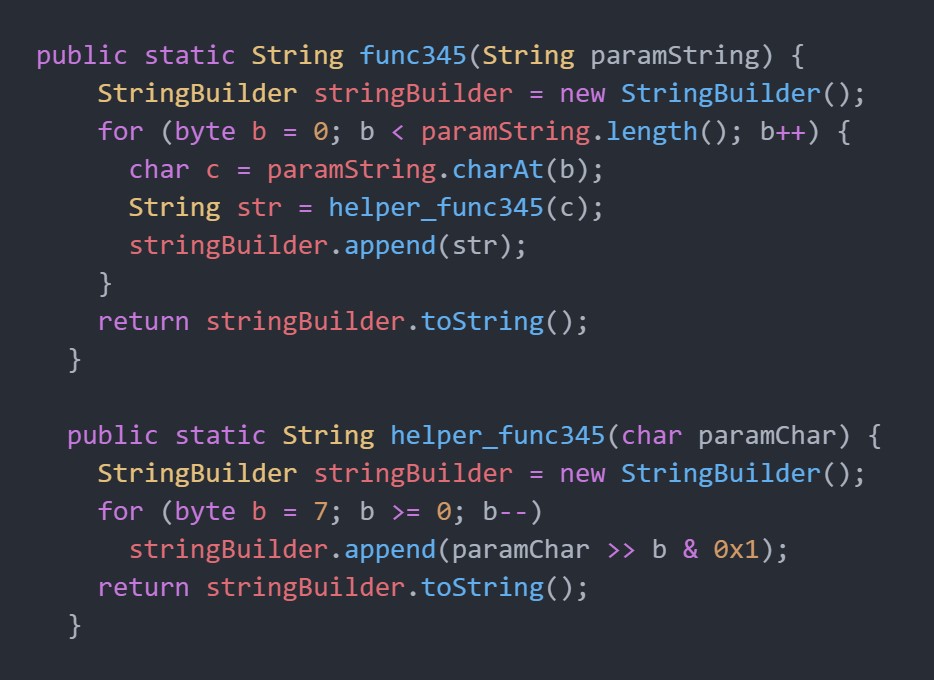


Since a binary was provided in bin.txt, it can be inferred that maybe the flag was passed as a string to get the binary in the **bin.txt** file. So we need to reverse engineer the Java decompiled code to see the operations performed.

Analyzing the code, initially it prompts the user to enter a string and reads it in **str1** using the Scanner class. Then it generates **str2** after passing **str1** in the **func345** method.



The **func345** method takes a string as input and converts each character to its binary representation using the **helper\_func345** method. The converted binary strings are stored in a StringBuilder and returned as a single combined string.

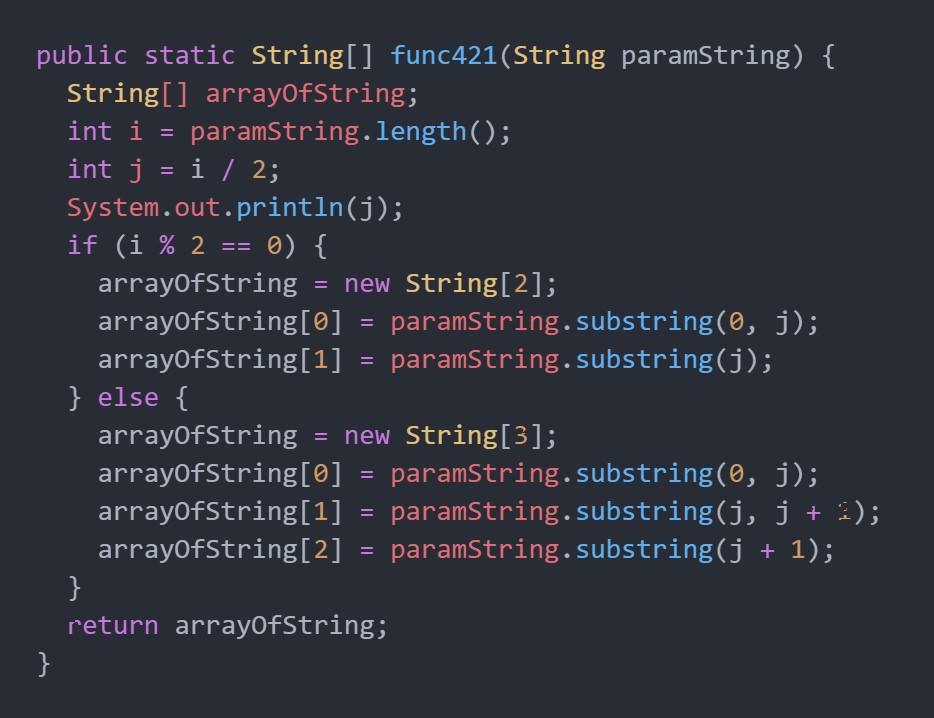


Next we have **str3** which is a string containing all alphabets in uppercase and lowercase, all numbers from 0-9 and some special symbols.



This might seem like an important aspect of the program but if you search for ‘**str3**’ in **VSCode** using Ctrl+F, you can see **str3** has only been used twice, once while defining and second in the **func842** function call. Going to **func842**, we can see it returns a integer value but it’s not been stored thus both **str3** and **func842** are of completely no importance and unnecessary for reversing the code.

Moving on, we have a String array arrayOfString defined as a return value of **func421** method. **The func421** method splits **str2** into two or three parts, depending on whether its length is even or odd. If even, it is split into two parts, and if odd, it is split into three parts. The split parts are returned as an array of strings, arrayOfString. This function also prints the half size of binary, which currently is of no importance.



Now it’s important to note that the binary value returned as **str2** will never be odd, so the else condition in **func421** is also unnecessary, the method will basically divide the string binary in 2 parts, as simple as that.

Now comes a series of function calls: **func981**, **func134**, **func842** and **func651**, all of which pass arrayOfString as their parameter.

The **func981** method swaps the first and last elements of arrayOfString. The **func134** method reverses each string (or each part of the binary) in the arrayOfString array using the **helper\_func134** method. The **func651** method swaps characters at odd positions between the two strings in arrayOfString.

After this, **str4** is initialized to return the value of the **func842** method. The method concatenates all strings in arrayOfString to form a single string, str4 which is printed out as final binary.

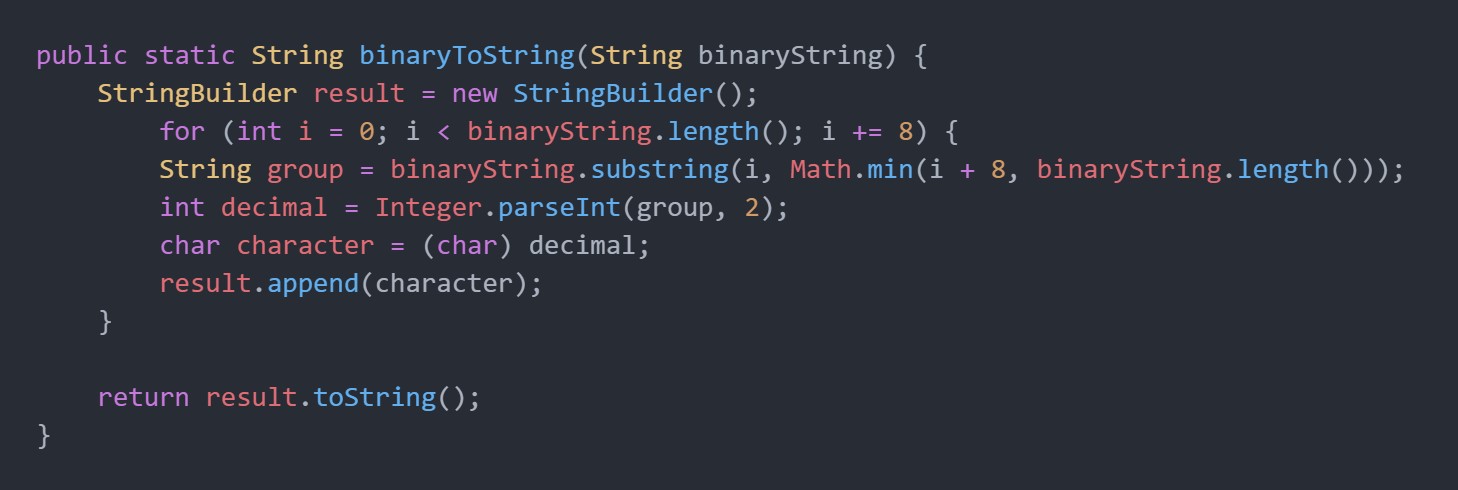
# Step 3: Flag Extraction

Since a binary was provided in **bin.txt**, it can be inferred that the flag was passed as a string to get the binary in the bin.txt file. Now that we have understood what the Java program does, we need to reverse the process to get back to the flag from binary.

This process will be: splitting the binary, exchanging the odd elements of each part, reversing each part, exchanging both the parts, combining the parts and finally converting binary to string.

First, we will create a new Java file to preserve the original code. In that we will first input the binary string, then perform split operation using **func421**, exchanging elements in each part using **func651**, reversing the split parts in array using **func134**, then exchanging the 2 parts using **func981**, combining both parts using **func824** and finally converting the binary to string using function binaryToString which will use Integer.parseInt(string,2) but for every 8 binary characters together since directly using the function will return in some errors due to binary being too long. You can also use any online tool for this and will get the same result.

This is the binaryToString function that I made:



As you can see, the only function we needed to make on our own was the binaryToString, otherwise this challenge was easily solvable by using the same functions provided in the file.

Now, we will run this **Answer.java** file and pass the binary from bin.txt and we will get the flag: **hacker{1\_l0ve\_gr33n}**

