

Course ID: ENSE 885AG/ENSE 496AE

Course: Digital Security

Topic: Image Shuffling for Image-in-Image
Steganography

Lecturer: Dr. Yasser Morgan

Name of Students: Daris Lychuk - 200361245

Kegan Lavoy - 200378170

Jamie Plunkett - 200348003

Github link:

<https://github.com/darislychuk/ENSE496AE-Final-Assignment>

Assignment given:

Choose a programming language and development environment.

Requirement 1:

- Write a small program that generates a long list (200 integers) of random numbers seeded by a known value.
- Run the same code on different processors/OS and make sure it always generates the same set of random integers.
- Write another function to compare the output of the files and make sure they are the same.
- Replace the seed value with big-int data type and make sure random function logic still works.
- Replace big-int seed value, also known as “Shared Key” with a large prime number.

Requirement 2:

- Write another small program that generates prime numbers and stores the values in an output file, this will be your “Prime Bank”.
- Store the amount of time required to discover the prime, this will be your “Discovery Times”.
- Force the program to stop when the discovery time of a single prime reaches 1 hour. After class discussion, it was told to keep going past an hour if possible.
- Plot a graph where the vertical axes are the “Discovery Times” and horizontal axes are the prime numbers themselves.

Requirement 3:

- Create Bob and Alice functions that can negotiate a Shared Key while denying Eve any information.
- Use knowledge of Diffie-Hellman from class.
- Alice’s secret integer ‘a’ and Bob’s secret integer ‘b’ are generated from secret passwords.
- Share an arbitrarily large number from the “Prime Bank” and take $g = 2$ or 5 .

Requirement 4:

- Make sure we can use the Shared Key generated from Alice and Bob as the seed for the same list of random integers.
- Use the compare function again to make sure the output files are the same.

Requirement 5:

- Develop code to shuffle a sample of our object (image).
- Make sure the image is unrecognizable after being shuffled.
- The Shared Key generated by Alice and Bob will be used by the shuffling algorithm.
- Bob will shuffle, and Alice will unshuffle the image.
- Repeat on many images to make sure the functionality works on all of them.

How we completed the assignment:

For this assignment, our group chose to work with Java using Eclipse as our IDE. Java, like any other language, has its limitations, but it is a language we are comfortable using. We also decided to keep most “Requirements” of the assignment in separate classes. We ended up having five different Java files. This is done to maintain clean coding practices by having solid code that is easier to read, maintain, edit, and so on.

Requirement 1:

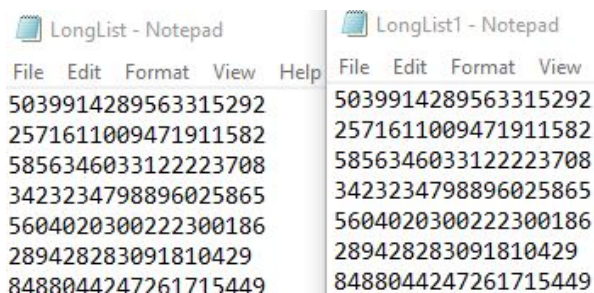
As shown in *Figure 1* below, a seed would be inputted into the “Random” function and the function will print out long int values based on the seed. The random function will reproduce the same random list of generated numbers, as shown in *Figure 2*, as long as the seed value is the same. This was tested on Intel Core i5-5200U and Intel i7-9900k processors, as well as, Windows 10 Pro, Windows 10 Home, and Mint Linux operating systems. If wanting to change the file type or name as to where it prints, change the name and type where you see “LongList1.txt” in *Figure 1* to your own. One downside to Java, is that the language only supports up to long int data type without running into problems. If we wanted to use BigInteger, we would have to write our own BigInteger class, however, we would be unable to use it with our “Random” function anyways. Java also does not offer unsigned long int data type. Therefore, we chose to stick with long int. It is not the biggest data type unfortunately, but does support up to 9 quintillion values. The “Random” function can support larger numbers than shown in *Figure 1*, this is just an example.

```
public static void main(String args[]){
    Random rand = new Random(40236);

    try
    {
        PrintWriter pr = new PrintWriter("LongList1.txt");

        for (int i=0; i< 200 ; i++)
        {
            pr.println(Math.abs(rand.nextLong()));
        }
        pr.close();
    }
}
```

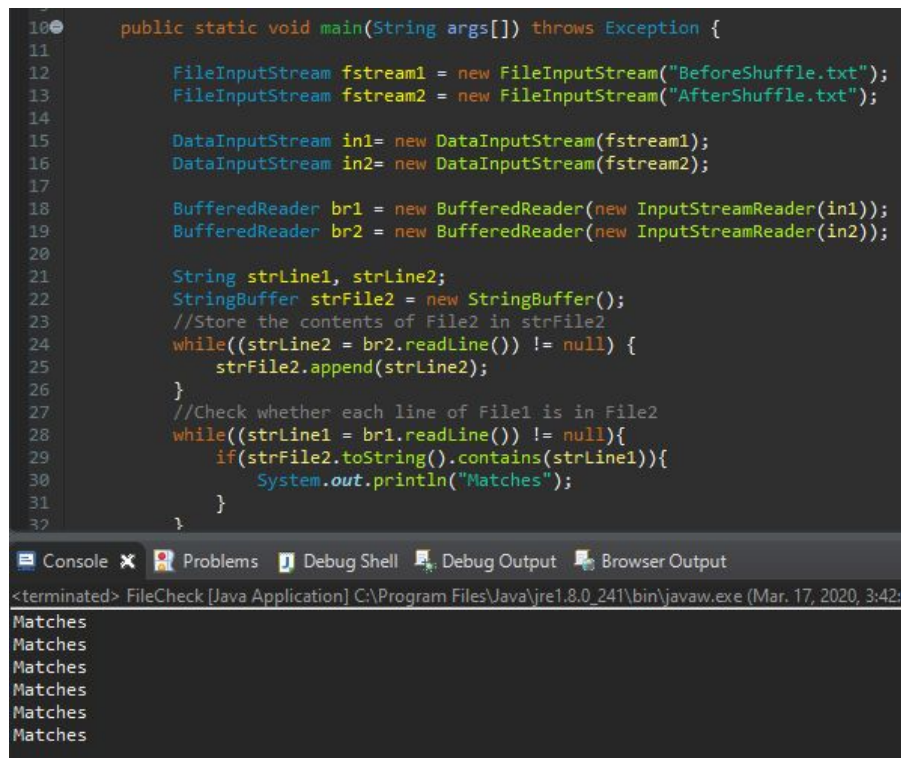
Figure 1: Random Number Generator Function



LongList - Notepad	LongList1 - Notepad
5039914289563315292	5039914289563315292
2571611009471911582	2571611009471911582
585634603312223708	585634603312223708
3423234798896025865	3423234798896025865
5604020300222300186	5604020300222300186
289428283091810429	289428283091810429
8488044247261715449	8488044247261715449

Figure 2: Random Lists Generated

After the lists were generated, we wrote a function that takes in both files and compares them line by line to each other, as shown in *Figure 3*. This function was kept very simple and returns “Matches” in the console as long as they match.



```
10 public static void main(String args[]) throws Exception {
11
12     FileInputStream fstream1 = new FileInputStream("BeforeShuffle.txt");
13     FileInputStream fstream2 = new FileInputStream("AfterShuffle.txt");
14
15     DataInputStream in1= new DataInputStream(fstream1);
16     DataInputStream in2= new DataInputStream(fstream2);
17
18     BufferedReader br1 = new BufferedReader(new InputStreamReader(in1));
19     BufferedReader br2 = new BufferedReader(new InputStreamReader(in2));
20
21     String strLine1, strLine2;
22     StringBuffer strFile2 = new StringBuffer();
23     //Store the contents of File2 in strFile2
24     while((strLine2 = br2.readLine()) != null) {
25         strFile2.append(strLine2);
26     }
27     //Check whether each line of File1 is in File2
28     while((strLine1 = br1.readLine()) != null){
29         if(strFile2.toString().contains(strLine1)){
30             System.out.println("Matches");
31         }
32     }
33 }
```

Console

<terminated> FileCheck [Java Application] C:\Program Files\Java\jre1.8.0_241\bin\javaw.exe (Mar. 17, 2020, 3:42:)

Matches
Matches
Matches
Matches
Matches
Matches

Figure 3: File Check Function

Requirement 2:

After checking the files to make sure they match, a prime number generator function was needed. This was done by using two for loops by starting at 1 and incrementing while testing for primality at each value until ‘n’ was reached, as shown in *Figure 4*. To check if the number was a prime or not, it would test to see what it was modulo by, that resulted in a remainder of 0. When producing a remainder of 0, this means that the value being checked is divisible by that number. The loop starts using modulus on itself by its own number. Meaning if the loop is checking 275911, it will start with $275911 \% 275911$, then $275911 \% 275910$, and so on. While in the loop, if the value being checked ends up having more than 2 remainders of 0 before reaching 0 itself, it knows it is not prime and exits the loop and adds it as a prime value to the “Prime Bank”. All prime numbers will only have a remainder of 0 with itself and 1, and that is how you know they are prime.

```

int i =0;
int num =0;
String primeNumbers = "";
long n = 999999999;
long startTime = System.nanoTime();
for (i = 1; i <= n; i++){
    int counter=0;
    for(num =i; num>=1; num--){
        if(i%num==0){
            counter = counter + 1;
        }
    }
    if (counter ==2){
        long endTime = System.nanoTime();
        long timeElapsed = endTime - startTime;
        primeNumbers = primeNumbers + i + "," +
            timeElapsed / 1000000 + "\n";
        try{
            PrintWriter pr = new PrintWriter("PrimeBank.csv");
            pr.println(primeNumbers);
        }
    }
}

```

Figure 4: Prime Number Generator

We let our prime number generator run for a couple of days. We only stopped it because it started to lag out the computer it was running on, and no other tasks could be completed until we stopped the generator program. Once we got our prime numbers list with the associated discovery times beside as shown in *Figure 5*, with column 'A' being the prime and 'B' being the discovery time, we plotted a graph, as shown in *Figure 6*.

	A	B
1	2	0
2	3	0
3	5	0
4	7	1
5	11	1
6	13	1
7	17	1
8	19	1
9	23	2
10	29	2
11	31	2
12	37	2

Figure 5: Prime Numbers & Associated Discovery Times

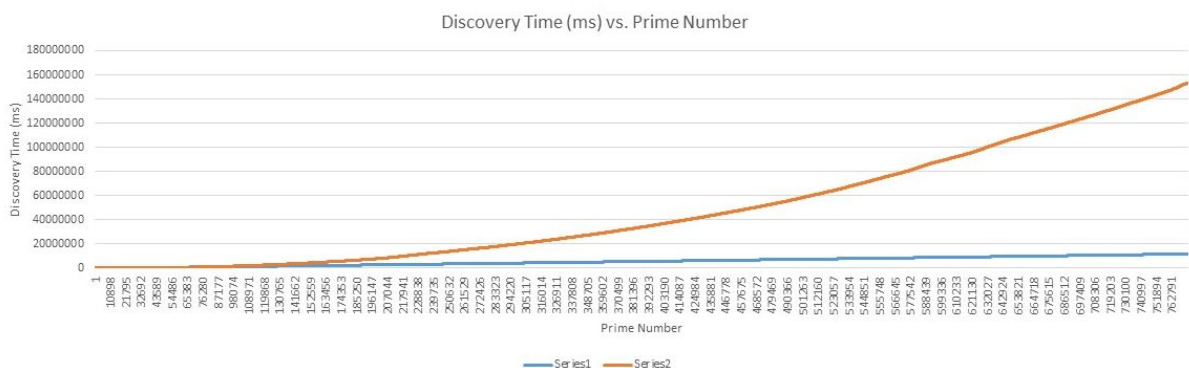


Figure 6: Discovery Time VS. Prime Number

The orange line in *Figure 6* represents the discovery time in milliseconds, while the blue represents the prime number being discovered. The numbers along the bottom are not all prime because the graph just took random values leading up to the biggest prime we found. The graph also does not represent all of the primes found, due to massive amounts of memory required to produce the graph. It was also lagging out pretty bad when trying to make the graph larger.

Requirement 3:

The next requirement was to use Diffie-Hellman to produce a shared key between Bob and Alice. 'P' was a prime value taken from our prime bank, 'G' was either supposed to be 2 or 5 in our case, as instructed, and then 'a' and 'b' were to be generated based on secret text passwords. As shown in *Figure 7* below, we allowed for the secret passwords to be inputted by the user in the console. This functionality could easily be changed to have predetermined passwords in the code, but thought that by entering a new password every time, it would be better.

```
16      P = 275911; //This is a prime taken from excel
17      G = 2;
18
19      System.out.println("Enter password for Alice : ");
20      String APass = AlicePass.nextLine();
21
22      for(int i = 0; i < APass.length(); i++)
23      {
24          a = a + Character.getNumericValue(APass.charAt(i));
25      }
26      a = (long) Math.sqrt(a); // a is the chosen private key
27      x = (long) (Math.pow(G, a)); // gets the generated key
28      x = x%P;
29
30      System.out.println("Enter password for Bob : ");
31      String BPass = BobPass.nextLine();
32
33      for(int i = 0; i < BPass.length(); i++)
34      {
35          b = b + Character.getNumericValue(BPass.charAt(i));
36      }
37      b = (long) Math.sqrt(b); // b is the chosen private key
38      y = (long) (Math.pow(G, b)); // gets the generated key
39      y = y%P;
40
41      keyA = (long) (Math.pow(y, a)%P); //Secret key for Alice
42      keyB = (long) (Math.pow(x, b)%P); //Secret key for Bob
43
44      System.out.println("Secret key for the Alice is : " + keyA);
```

Console Problems Debug Shell Debug Output Browser Output

<terminated> DHKeyAgreement [Java Application] C:\Program Files\Java\jre1.8.0_241\bin\javaw

Enter password for Alice :
mypassword
Enter password for Bob :
hellothere
Secret key for the Alice is : 186874
Secret Key for the Bob is : 186874

Figure 7: Diffie-Hellman Key Exchange Algorithm

In the code above, it takes in the user's password input for Alice and Bob and converts these passwords to a numerical value based on the password. These numerical values are then fed into our Diffie-Hellman algorithm. The algorithm then returns the Shared Key for both Alice and Bob to the console, while keeping their key a secret, because it is their password. The only downside about our Diffie-Hellman algorithm is that because we are using only long int data types, the passwords can not be very long, or our program variables will overflow, causing different Shared Keys to be produced.

Requirement 4:

This requirement had us testing to see if our Shared Key between Bob and Alice could be used as the seed to create the same set of random integer lists as previously done. This was done successfully, and then tested with the file check successfully as well. This section can refer back to *Figure 1*, *Figure 2*, and *Figure 3*, for how this was done.

Requirement 5:

For this requirement, our object to shuffle is an image. We took the Shared Key produced in the last algorithm, and inputted it into shuffle and deshuffle functions. The "Shuffle" and "DeShuffle" function are shown below in *Figure 8*.

```
public static int[] Shuffle(int[] toShuffle, int key)
{
    int size = toShuffle.length;
    int[] exchanges = GetShuffleExchanges(size, key);
    for (int i = size - 1; i > 0; i--)
    {
        int n = exchanges[size - 1 - i];
        int tmp = toShuffle[i];
        toShuffle[i] = toShuffle[n];
        toShuffle[n] = tmp;
    }
    return toShuffle;
}
public static int[] DeShuffle(int[] shuffled, int key)
{
    int size = shuffled.length;
    int[] exchanges = GetShuffleExchanges(size, key);
    for (int i = 1; i < size; i++)
    {
        int n = exchanges[size - i - 1];
        int tmp = shuffled[i];
        shuffled[i] = shuffled[n];
        shuffled[n] = tmp;
    }
    return shuffled;
}
```

Figure 8: Shuffle and DeShuffle Functions

Both "Shuffle" and "DeShuffle" use the Shared Key value previously agreed upon between Bob and Alice, and call another function that also uses the key to shuffle and deshuffle the pixels based on the key given, as shown in *Figure 9*.

```

public static int[] GetShuffleExchanges(int size, int key)
{
    int[] exchanges = new int[size - 1];
    Random rand = new Random(key);
    for (int i = size - 1; i > 0; i--)
    {
        int n = rand.nextInt(i + 1);
        exchanges[size - 1 - i] = n;
    }
    return exchanges;
}

```

Figure 9: Exchanges Pixel Locations Based on Seed

Our program takes in the image, gets both the width and height of the image, and then gets the RGB value of each pixel, otherwise known as pixel value itself, as shown in Figure 10.

```

PrintWriter pr = new PrintWriter("BeforeShuffle.txt");
PrintWriter pr1 = new PrintWriter("AfterShuffle.txt");
BufferedImage originalImage = ImageIO.read(new File(
    "c:\\Users\\Owner\\Documents\\ENSE\\ENSE496AE\\Final\\TERRY.jpg"));

width = originalImage.getWidth();
height = originalImage.getHeight();
int[] pixels = new int[(width*height)];

for(int i=0; i<height; i++) {
    for(int j=0; j<width; j++) {
        int p = originalImage.getRGB(j,i);
        //p = p/167238;
        pixels[count] = p;
        count++;
        // Color c = new Color(originalImage.getRGB(j, i));
        // System.out.println("S.No: " + count + " Red: " + c.getRed() + " G
        System.out.print(pixels[count-1]);
    }
}

```

Figure 10: Getting Image Height and Width and Pixel Value

After getting the appropriate pixel values, the program prints those pixel values into a text file called "BeforeShuffle.txt". This file will be used later to compare the original image to the de-shuffled image.

```

try
{
    pr.println(pixels[count-1]);
}
catch (Exception e)
{
    e.printStackTrace();
    System.out.println("No such file exists.");
}

```

Figure 11: Printing Pixel Values to Text File to Compare

Our program then calls the “Shuffle” function with the given Shared Key, in this case it is ‘167238’, as shown in *Figure 11*. It saves a shuffled image to an output file to allow people to view that the image is truly shuffled.

```
count = 0;
int[] shuffled = Shuffle(pixels,167238);

for(int i=0; i<height; i++) {
    for(int j=0; j<width; j++) {

        System.out.print(shuffled[count]);
        originalImage.setRGB((width - (j+1)),(height - (i+1)),shuffled[count]);
        count++;
    }
}

try
{
    ImageIO.write(originalImage, "jpg", new File(
        "c:\\Users\\Owner\\Documents\\ENSE\\ENSE496AE\\Final\\newShuffledTerry.jpg"));
}
```

Figure 11: Calling ‘Shuffle’ Function

The program then calls the “DeShuffle” function with the same Shared Key value as shown in *Figure 12*. It will then de-shuffle the image and print the pixel values to a text file to compare to the original. Once the pixel values are in a text file, it will print the de-shuffled image to an image file again, as shown in *Figure 13*.

```
System.out.print("\n");
count = 0;
int[] deShuffled = DeShuffle(shuffled,167238)

for(int i=0; i<height; i++) {
    for(int j=0; j<width; j++) {

        System.out.print(deShuffled[count]);

        try
        {
            pr1.println(deShuffled[count]);
        }
        catch (Exception e)
        {
            e.printStackTrace();
            System.out.println("No such file exists.")
        }

        originalImage.setRGB(j,i,deShuffled[count]);
        count++;
    }
}
```

Figure 12: Calling “DeShuffle” Function and Printing Pixel Values to Text

```
try
{
    ImageIO.write(originalImage, "jpg", new File(
        "c:\\Users\\Owner\\Documents\\ENSE\\ENSE496AE\\Final\\newTERRY.jpg"));
}
```

Figure 13: Write the De-shuffled Image to a File

An example of images can be shown in *Figure 14*, *Figure 15*, and *Figure 16* of an original image, a shuffled image, then the de-shuffled image.



Figure 14: Original Museum Image Used

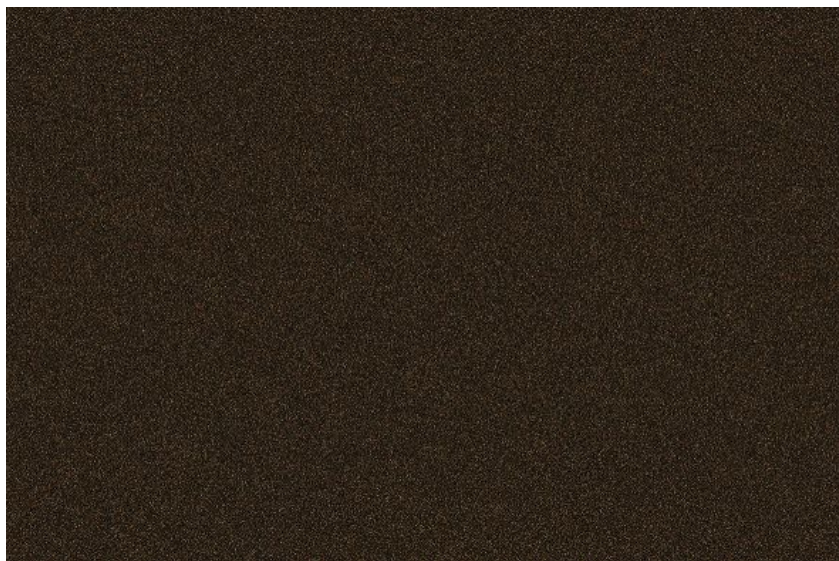


Figure 15: Shuffled Museum Image



Figure 16: De-shuffled Museum Image

We tested our program on many different images, some black and white, some RGB, and different sizes. All tests were successful. We compared pixel values of the original images to those of our de-shuffled images, and they returned identical.

As shown in Figure 17 and Figure 18, by using a hex editor, we were able to see what data inside the images were actually altered. As shown, the values are all different going up to the end of the file. Another strange difference is that the “newShuffledMuseum.png” image has a larger data offset when looking at the values in blue running down the side. It actually doubles in data offset values. Not only does the “Shuffle” function change the data but also the data offset.

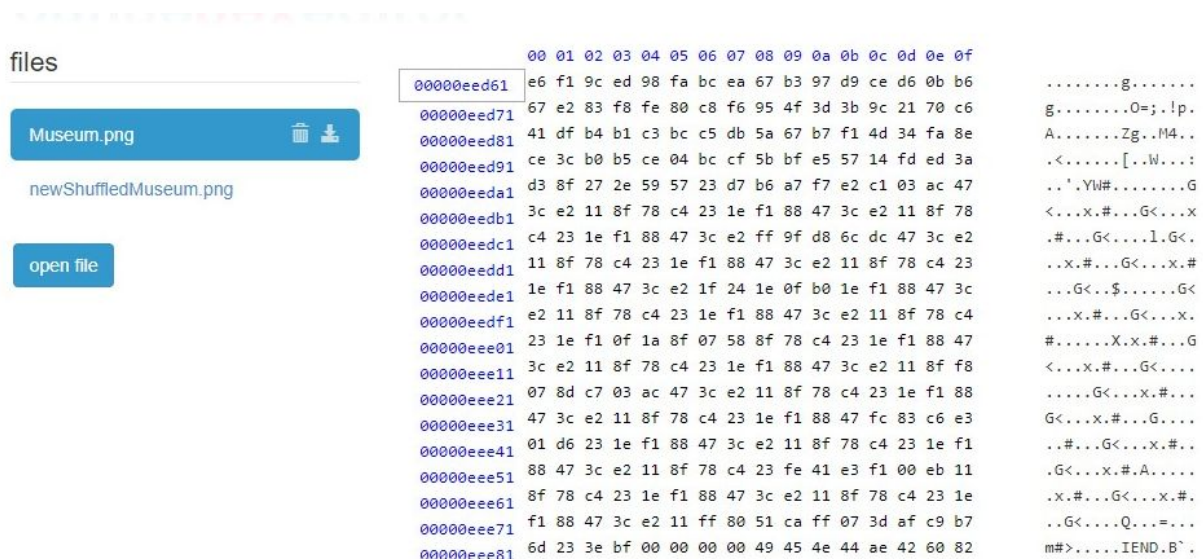


Figure 17: Hex Editor Data of Original Museum Image

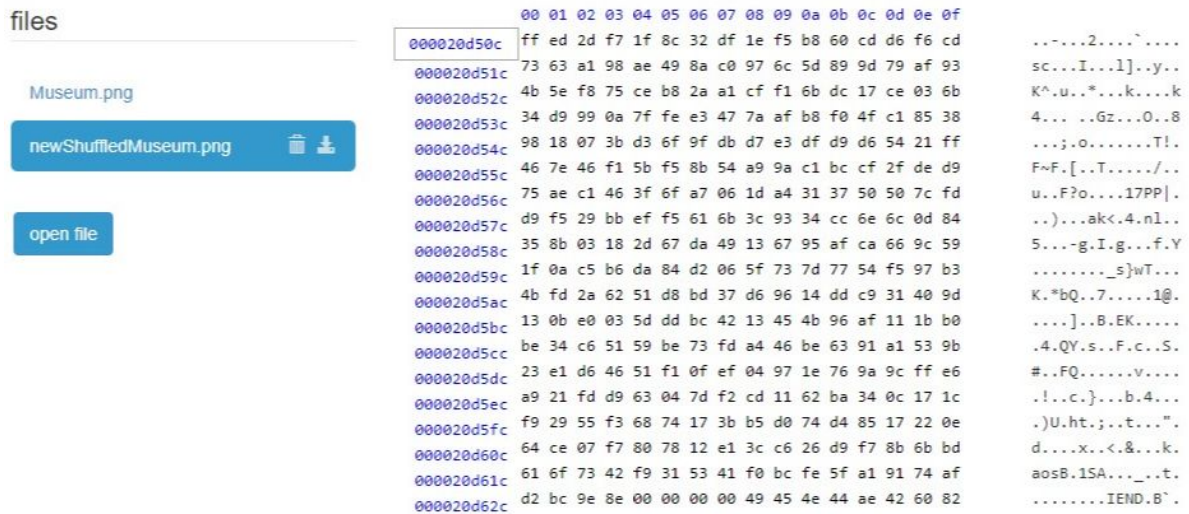


Figure 18: Hex Editor Data of Shuffled Museum Image

In the end, we finished with five different Java functions as shown in *Figure 19*. This was done to maintain clean coding practices as mentioned earlier. If later wanting functionality to all be incorporated together, it would be very easy. We believe our object given would be hidden from any and all intruders and be undetectable.

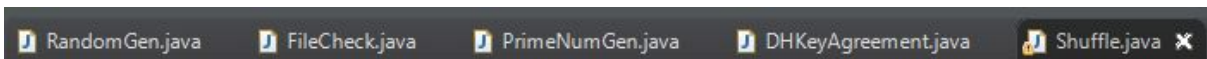


Figure 19: Java Files