Mark Seliternikov
picoCTF - Vault Door 7 [400 points]

Challenge description:

vault-door-7 | 400 points

Tags: Category: Reverse Engineering

AUTHOR: MARK E. HAASE

Description

This vault uses bit shifts to convert a password string into an array of integers. Hurry, agent, we are running out of time to stop Dr. Evil's nefarious plans! The source code for this vault is here: VaultDoor7.java

Hints

1 2

Once you open the source code you can see that each char (byte) is converted to binary, then 4 bytes are joined end to end to form 1 unit of 4 bytes. Then the binary data is converted to an int and is stored in the source code (very bad idea if you want to protect your password).

Here you can see the user's input is expected to be in the flag format. Then the contents of the flag (inside the flag) are being passed to variable.

```
    System.out.print("Enter vault password: ");
    String userInput = scanner.next();
String input = userInput.substring("picoCTF{".length(),userInput.length()-1);
if (vaultDoor.checkPassword(input)) {
```

Later a function the check the input variable (flag contents) is being called in order to check if it matches.

```
if (vaultDoor.checkPassword(input)) {
    System.out.println("Access granted.");
} else {
    System.out.println("Access denied!");
    }
}
```

Here you can see a function is called to turn the password to an array of ints that is formed from turning char to byte and then combining those bytes

```
byte[] hexBytes = hex.getBytes();
for (int i=0; i<8; i++) {
    x[i] = hexBytes[i*4]   << 24
         | hexBytes[i*4+1] << 16
         | hexBytes[i*4+2] << 8
         | hexBytes[i*4+3];
}
```

Here's a comment in the source code exposing the whole mechanism:

```
// Each character can be represented as a byte value using its
// ASCII encoding. Each byte contains 8 bits, and an int contains
// 32 bits, so we can "pack" 4 bytes into a single int. Here's an
// example: if the hex string is "01ab", then those can be
// represented as the bytes {0x30, 0x31, 0x61, 0x62}. When those
// bytes are represented as binary, they are:
//
// 0x30: 00110000
// 0x31: 00110001
// 0x61: 01100001
// 0x62: 01100010
//
// If we put those 4 binary numbers end to end, we end up with 32
// bits that can be interpreted as an int.
//
// 00110000001100010110000101100010 -> 808542562
//
// Since 4 chars can be represented as 1 int, the 32 character password can
// be represented as an array of 8 ints.
```

These are the int values that are compared to the password:

```
int[] x = passwordToIntArray(password);
return x[0] == 1096770097
    && x[1] == 1952395366
    && x[2] == 1600270708
    && x[3] == 1601398833
    && x[4] == 1716808014
    && x[5] == 1734291511
    && x[6] == 960049251
    && x[7] == 1681089078;
```

In order to solve this I had to do the reverse. I had to start from the numerical value back to the ASCII value. I wrote this short python script in order to do that.
In the final loop that prints the actual flag I added the flag format inorder to make the final result in the whole flag format.

```
numList = [
        1096770097,
        1952395366,
        1600270708,
        1601398833,
        1716808014,
        1734291511,
        960049251,
        1681089078,
        ]

chrList = list()

for num in numList:

    num = (bin(num))[2:].zfill(32)

    for i in range(0, 32, 8):

        chrList.append(chr(int(num[i: i + 8], 2)))

print("picoCTF{", end='')
for char in chrList:

    print(char, end='')

print("}")
```

My script takes an int from the list and then converts it to binary and then fills 0 accordingly

```
num = (bin(num))[2:].zfill(32)
```

What it does then is convert the binary data into 4 separate binary values (using a loop), decode each 8 bit value back to an int value and then get the ASCII value of that int.
After that it adds it to the chrList (list containing separate ASCII values representing the flag contents)

```
chrList.append(chr(int(num[i: i + 8], 2)))
```

After running the python script this is the result:

```
mark@bubuntu:~/Desktop$ python3 solver.py
picoCTF{A_b1t_0f_b1t_sh1fTiNg_07990cd3b6}
```

And that is indeed the correct result :)