CTF Writeup -- Houseplant CTF

Preston Kemp -- NYU Offensive Security Spring 2020

Introduction

For my CTFtime ranked CTF challenge I chose the Houseplant CTF, this challenge was held Apr 25 and ended Apr 26. I focused on the reverse engineering challenges, since they were most like the kind of work that we had done in class.

Other challenges first:

In order to get to a challenge that was *actually* difficult, I had to complete several easy challenges first, most were basic python reverse engineering problems, I'll touch on those briefly:

1. EZ:This was the first challege that had to be solved in order to move on. The flag for this challenge was `rtcp{tH1s_i5_4_r3aL_fL4g_s0_Do_sUbm1T_1t!}` and it was simply in a comment inside the provided python file.

2. PZ: This was the second challenge that had to be solved. This challenge was also very straight forward, the program is a series of functions with if statements, most of the functions aren't really used and their intent appears to be to confuse the reader. The function worth analyzing in this example is `checkpass()` which, does as it sounds it does and checks the password (flag) that the user enters. If the user enters `rtcp{iT5_s1mPlY_1n_tH3_C0d3}` the user is granted access. The password in this case is also the flag.

3. Lemon: This was the third challenge that was necessary to continue. This challenge was also straight forward and that it involved using what essentially amounted as multiple substring to reconstruct the flag that was present in the code itself. The flag ended up being: `rtcp{y34H_tHiS_a1nT_sEcuR3}`

4. Squeezy: This was the fourth and final challenge before getting access to the more difficult challenges. This challenge involved an encoded version of the flag, meaning it needed to be transformed to recover the original flag. When analyzing the code, I noticed a couple of calls to Base64 encode and decode, so obviously the flag was encoded.

    1. I started by decoding the encoded flag `HxEMBxUAURg6I0QILT4UVRolMQFRHzokRBcmAygNXhkqWBw=` from Base64 and recovered the unencoded flag
    2. We are also given a variable called "key" which is `meownyameownyameownyameownyameownya`
    3. The flag was more than just encoded though, there was a function called "woah" that did the following:

```
def woah(s1,s2):
    return ''.join(chr(ord(a) ^ ord(b)) for a,b in zip(s1,s2))
```

the ^ symbol in Python is used to bitwise XOR two operands. This is pretty clearly just XORing the input of s1 and s2 (the key and the user's input) charcter by character. Since XOR is commutative, ($a \oplus b = b \oplus a$), we can simply run our decoded Base64 flag back through this function as the input (s2) and pass the same key (s1) through the function and retrieve the actual flag.

Example solution:

```
import base64

def woah(s1,s2): #part of pass 3
    return ''.join(chr(ord(a) ^ ord(b)) for a,b in zip(s1,s2))

def solvePass3():
    flag = b'HxEMBxUAURg6I0QILT4UVRolMQFRHzokRBcmAygNXhkqWBw='
    key = "meownyameownyameownyameownyameownya"
    a = base64.b64decode(flag, altchars=None)
    b = a.decode()
    print(woah(key, b))

def main():
    solvePass3()

main()
```

Bingo! We got the flag `rtcp{y0u_L3fT_y0uR_x0r_K3y_bEh1nD!}`

The Problem

Up till the reversing challenges in this CTF have been pretty easy/straightforward, this next challenge is significantly more difficult, as it involves multiple steps in order to completely reverse it.

The relevant section of the challenge is here:

```
    public static int[] realflag =
{9,4,23,8,17,1,18,0,13,7,2,20,16,10,22,12,19,6,15,21,3,14,5,11};
    public static int[] therealflag =
{20,16,12,9,6,15,21,3,18,0,13,7,1,4,23,8,17,2,10,22,19,11,14,5};
    public static HashMap<Integer, Character> theflags = new HashMap<>();
    public static HashMap<Integer, Character> theflags0 = new HashMap<>();
    public static HashMap<Integer, Character> theflags1 = new HashMap<>();
    public static HashMap<Integer, Character> theflags2 = new HashMap<>();
    public static boolean m = true;
    public static boolean g = false;
    public static boolean check(String input){
        boolean h = false;
        String flag = "ow0_wh4t_4_h4ckr_y0u_4r3";
        createMap(theflags, input, m);
        createMap(theflags0, flag, g);
        createMap(theflags1, input, g);
        createMap(theflags2, flag, m);
```

```
        String theflag = "";
        String thefinalflag = "";
        int i = 0;
        if(input.length() != flag.length()){
            return h;
        }
        if(input.charAt(input.length()-2) != 'o'){
            return false;
        }
        if(!input.substring(2,4).equals("r3") || input.charAt(5) != '_' ||
input.charAt(7) != '_'){
            return false;
        }
        //rtcp{h3r3s_a_fr33_fl4g!}
        for(; i < input.length()-3; i++){
            theflag += theflags.get(i);
        }
        for(; i < input.length();i++){
            theflag += theflags1.get(i);
        }
        for(int p = 0; p < theflag.length(); p++){
            thefinalflag += (char)((int)(theflags0.get(p)) + (int)
(theflag.charAt(p)));
        }
        for(int p = 0; p < theflag.length(); p++){
            if((int)(thefinalflag.charAt(p)) > 146 && (int)
(thefinalflag.charAt(p)) < 157){
                thefinalflag = thefinalflag.substring(0,p) + (char)((int)
(thefinalflag.charAt(p)+10)) + thefinalflag.substring(p+1);
            }
        }
        return thefinalflag.equals("□□ì¨ ¢«¢¥Ç©© ÂëÏãÒ□ËãhÔÊ");
    }
```

At first glance, this method takes a string, transforms it and checks to see if it matches `thefinalflag` variable above.

This probably sounds obvious, I've learned that when solving CTFs it's important to break down the problem into chunks. Trying to solve the entire problem at once isn't easy and can lead to frustration. Since we're wanting to reverse the transformation of `thefinalflag` we can do so by simply performing the transformation steps in reverse order.

First Reversing:

```
        for(int p = 0; p < theflag.length(); p++){
            if((int)(thefinalflag.charAt(p)) > 146 && (int)
(thefinalflag.charAt(p)) < 157){
                thefinalflag = thefinalflag.substring(0,p) + (char)((int)
(thefinalflag.charAt(p)+10)) + thefinalflag.substring(p+1);
            }
        }
```

In the above loop we can see that the loop (if the condtional succeeds):

1. Takes the existing final flag and concatenates it to the final flag variable
2. Grabs the character at the current iteration of the loop in `thefinalflag` and adds 10 to its value, before casting it back to a `char`
3. Concatenates the rest of `thefinalflag` onto the end of `thefinalflag`

It's worth noting that the above loop is pretty gross, and is likely written to be confusing to the reader. Here's how we can reverse it:

```
        for (int i = 0; i < flagToReverse.length; i++) {
            if (flagToReverse[i] > 156 && flagToReverse[i] < 167) {
                thefinalflag += (char)(flagToReverse[i] - 10);
            } else {
                thefinalflag += (char)(flagToReverse[i]);
            }
        }
```

`thefinalflag` in this case is the nonsense string that was being compared to at the end of the `check()` method.

Second Reversing:

```
        for(int p = 0; p < theflag.length(); p++){
            thefinalflag += (char)((int)(theflags0.get(p)) + (int)
(theflag.charAt(p)));
        }
```

To reverse:

```
        for (int i = thefinalflag.length() - 1; i >= 0; i--) {
            theflag = (char)(((int)thefinalflag.charAt(i))-
((int)theflags0.get(i))) + theflag;
        }
```

It's worth noting that above the loop needs to be written this way so that the iterator, `i`, reflects the actual reverse order that the loop operates in the other direction, otherwise we'll get bogus results. This is because `theflags0` is a map.

Third and Fourth Reversing:

I'm doing these together since they are essentially the same problem, but focused on different ranges of `theflag` variable and using different coresponding maps.

```
int i = 0;
for(; i < input.length()-3; i++){
    theflag += theflags.get(i);
}
for(; i < input.length();i++){
    theflag += theflags1.get(i);
}
```

The loops above take the maps that were created earlier and essentially scrambles the input into different positions (remember that the maps are the input from the user). We need to reverse the order of the values in the map in order to put the characters back into their proper positions.

In order to reverse:

```
for (int i = theflag.length() - 1; i >= theflag.length() - 3; i--) {
      theflags1.put(i, theflag.charAt(i));
  }
  for (int i = theflag.length() - 4; i >= 0; i--) {
      theflags.put(i, theflag.charAt(i));
  }
```

Fifth and Six Reversing:

This step isn't really so much a step in the transform process as it is just placing the actual characters into an array where we can extract the actual flag. This is performed by the `createMap` function in the original CTF.

```
for (int i = 0; i < theflag.length(); i++) {
    if (realflag[i] < 21) {
        actualflag[i] = theflags.get(realflag[i]);
    }
}
for (int i = 0; i < theflag.length(); i++) {
    if (therealflag[i] > 20) {
        actualflag[i] = theflags1.get(therealflag[i]);
    }
}
```

The Final Step:

In the original CTF transform, there are a few parts of the flag that are given to us, so we set those to be the values provided:

```
actualflag[2] = 'r';
actualflag[3] = '3';
actualflag[5] = '_';
actualflag[7] = '_';
```

```
        actualflag[actualflag.length-2] = 'o';

        return (new String(actualflag));
```

And then we return with our `actualflag`!

I also wrote a few helper function to help format the flag with the correct `rtcp{flag}` format.

Once we call our reversing method we get `rtcp{h3r3s_4_c0stly_fl4g_4you}` out! And hey it looks like we got it!

Conclusion

Just a few closing notes about the above challenge, as I stated previously the difficulty in challenges like this aren't that they're *hard* per se, but they do require some endurance and a willingness to break down the problem into pieces which can be solved individually.

Full reversing code below:

```
public static String solveTough(String flag) {
        char[] flagToReverse = "□□ì¨ ¢«¢¥Ç©© ÂëÏãÒ□ËãhÔÊ".toCharArray();
        String thefinalflag = "";
        createMap(theflags0, flag, g);
        createMap(theflags2, flag, m);
        char[] actualflag = new char[24];
        String theflag = "";

        for (int i = 0; i < flagToReverse.length; i++) {
            if (flagToReverse[i] > 156 && flagToReverse[i] < 167) {
                thefinalflag += (char)(flagToReverse[i]-10);
            } else {
                thefinalflag += (char)(flagToReverse[i]);
            }
        }
        for (int i = thefinalflag.length()-1; i >= 0; i--) {
            theflag = (char)(((int)thefinalflag.charAt(i))-
((int)theflags0.get(i))) + theflag;
        }
        for (int i = theflag.length()-1; i >= theflag.length()-3; i--) {
            theflags1.put(i, theflag.charAt(i));
        }
        for (int i = theflag.length()-4; i >= 0; i--) {
            theflags.put(i, theflag.charAt(i));
        }
        for (int i = 0; i < theflag.length(); i++) {
            if (realflag[i] < 21) {
                actualflag[i] = theflags.get(realflag[i]);
            }
        }
        for (int i = 0; i < theflag.length(); i++) {
            if (therealflag[i] > 20) {
```

```
                actualflag[i] = theflags1.get(therealflag[i]);
            }
        }

        actualflag[2] = 'r';
        actualflag[3] = '3';
        actualflag[5] = '_';
        actualflag[7] = '_';
        actualflag[actualflag.length-2] = 'o';

        return (new String(actualflag));
    }
    public static String formatFlag(String flag) {
        return "rtcp{".concat(flag).concat("}");
    }
```