XORROX – NahamCon 2022

James Nelson

05/01/2022

The XORROX challenge gives you an output file that has the contents of two arrays, “xorrox” and “enc”, along with a python file that contains the algorithm used to encrypt the flag. The algorithm first opens a text file which contains the flag and reads the contents of it. It generates a random integer for every character in the flag and stores them in a “key” array which is not provided to us.

```
#!/usr/bin/env python3

import random

# I changed the name of the file to open so I could run the algorithm with a fake flag for testing.
with open("fakeflag.txt", "rb") as filp:
    # strip() removes leanding and trailing spaces in a string.
    flag = filp.read().strip()

# returns a random int for every character in the flag
key = [random.randint(1, 256) for _ in range(len(flag))]

# empty arrays to be filled
xorrox = []
enc = []
```

The algorithm then enumerates over the key array with two variables, “i” and “v”, the first of which will start at 0 and correspond to the current index of the key array, the second will hold the value of the current array element. Inside the loop, a “k” variable is set to one at the beginning of each iteration. There is an inner loop that follows, which runs for a decrementing range from the current value of “i” until 0. The first iteration of this inner loop is always skipped because the value of “i” will be equal to 0. For the further iterations of the outer loop, what will happen inside the inner loop is that the “k” value will be set equal to the current value of “k”

XOR'd with the current key value, and then XOR'd again with the previous key up until but not including the first key value, which will not be used in this loop.

```
17  # i will contain the index of the current element,
18  # v will contain the value
19  for i, v in enumerate(key):
20
21      # k is set to 1 to begin every iteration
22      k = 1
23
24      # Looping from the current value of i, decrementing until 0
25      # This part doesn't run on the first iteration due to i being 0
26      # The loop stops when i = 0, so the first key is not used in this loop
27      for j in range(i, 0, -1):
28          # x ^= 3 is the same as saying x = x ^ 3
29          # ^ is the XOR operator, it sets each bit to 1 if only one of two bits is 1.
30          k ^= key[j]
31
```

The final two lines of code in the outer loop append the current "k" value to the "xorrox" array, and append the "enc" array with the result of an XOR operation of the current flag value with the current key value. After the outer loop, the script opens an output file and writes both the "xorrox" and "enc" arrays to the file.

```
32      # appends the k value to the first empty array
33      # the first element of this array will always be set to 1 because the
34      # internal loop doesn't run the first round.
35      xorrox.append(k)
36
37      # appends the result of the current flag character
38      # XOR'd with the current value of the key array.
39      enc.append(flag[i] ^ v)
40
41  # I changed the filename so I could differentiate between the
42  # test one and the original output file.
43  with open("jcn_output.txt", "w") as filp:
44      # using f-strings to write output, aka "Literal String Interpolation"
45      filp.write(f"{xorrox=}\n")
46      filp.write(f"{enc=}\n")
47
```

If we look further into XOR operations, we can see an interesting property about them. The XOR (Exclusive OR) operation, is a bitwise operation, meaning it works on the individual

bits of the operands. Let's say we have 4 bits for X and 4 bits for Y, and we use them both as the operands for an XOR operation (X ^ Y).

XOR

X	Y	=	X^Y
0	0	=	0
0	1	=	1
1	0	=	1
1	1	=	0

X^Y	X	=	Y
0	0	=	0
1	0	=	1
1	1	=	0
0	1	=	1

X^Y	Y	=	X
0	0	=	0
1	1	=	0
1	0	=	1
0	1	=	1

If we perform the initial operation, we get a result "X^Y", and if we use that result and perform an XOR operation with "X^Y" and either of X or Y, we will get the other operand Y or X as our result.

Because the first key value is never used in the XOR operations, we can't infer the first key value from the "xorrox" array; we only have the result of the XOR operation between the key value and the flag value, which is the first value of the "enc" array. Since we don't actually need the first key value to decrypt any other part of the string, we could always skip this for now and then see what might be missing, but since I know we are most likely going to be looking for a string similar to "flag{…}" I went ahead and inferred the first character of the flag will be "f". The lower case "f" has a Unicode integer value of 102 so I store that in a "flag" array in a new python script. Because the inner loop doesn't trigger on the first iteration of the outer loop, the first "enc" value is simply the result of an XOR operation between the first flag value and the first key value. I ran an XOR operation between the first flag value of 102 and the first "enc" value of 26 (which you can simply enter 102^26 in an interactive python session) which will return the first value of the key array, 124. I add this value to a key array of my own in the new python script, I copy and paste the contents of the output file containing the "enc" and "xorrox" arrays, and I create a "flagstring" variable and set it equal to "f".

```
flag = [102]
key = [124]
enc=[26, 188, 220, 228, 144, 1, 36, 185, 214, 11, 25, 178, 145, 47, 237, 70, 244, 149,
    98, 20, 46, 187, 207, 136, 154, 231, 131, 193, 84, 148, 212, 126, 126, 226, 211, 10, 20, 119]
xorrox=[1, 209, 108, 239, 4, 55, 34, 174, 79, 117, 8, 222, 123, 99, 184, 202, 95, 255,
    175, 138, 150, 28, 183, 6, 168, 43, 205, 105, 92, 250, 28, 80, 31, 201, 46, 20, 50, 56]
flagstring = 'f'

```

The second key value is the simplest to derive, because the inner loop runs, but only one XOR operation is done inside, which is the where the k value is set to the current value of k (which is 1) XOR'd with the current key value (key[1]). The result is then appended as the second element of the "xorrox" array, so to get the second key value we only need to get the result of the second value of "xorrox" XOR'd with 1. I then set a variable "n" equal to 2, so we can derive the rest of the key values using a loop.

```
'''
The second xorrox value is only the key value XOR'd with 1
so the second key value is equal to xorrox[1] XOR 1
I did this outside of the loop so that I could use less code
inside the loop, since the first key value will not be used
as an XOR operand.
'''
key.append(xorrox[1]^1)

n = 2
```

Inside the loop, I first set an “m” variable to the result of the current xorrox value (n) XOR’d with the previous key value (n – 1). There is an inner loop which runs in descending order from a range of n – 1 up until it hits 1, this means it will not use this inner loop on the first iteration, but on subsequent iterations all this loop does is set “m” equal to the result of an XOR operation between the current “m” value and each key in descending order but not including the first key value.

```
while n < len(xorrox):
    # here is why I did the first key append outside of this loop,
    # it would have referenced the first key value if n was equal to 1
    m = (xorrox[n]^key[n-1])

    # I stopped the range at 1 because we already solved
    # for that element.
    for i in range((n-1), 1, -1):
        # XOR m with every prior key value in descending order
        m = (m^key[i-1])
```

At the end of every iteration of the outer loop, I set the “m” value equal to the current “m” value XOR 1 (this is the reverse order that happens in the encryption algorithm, where “k” always started as 1 when performing the XORs), then append the “m” value to the key array and increment “n” by 1. This loop will derive all of the keys except the second one (which we derive before the loop) and the first one (which we can either infer or just skip for now).

```
31      # After the for loop,
32      # XOR m with 1, this is the exact reverse order of the encryption.
33      m = m^1
34
35      # append the key value and increment the n value
36      key.append(m)
37      n += 1
38
```

I print the keys and the “enc” array to the screen for testing purposes and to check the validity of the keys when running on test data which was generated with a fake flag file and the original python script that encrypted the actual flag. I then run a for loop to derive the flag values which are the result of an XOR operation between corresponding “enc” and key values. Inside this loop I also append the Unicode character of the flag value to the “flagstring” variable. I then print both the flag array (for testing purposes) and the flag string to the screen.

```
39  # I print the key and enc values to the screen for testing
40  print(f"{key=}\n")
41  print(f"{enc=}\n")
42
43  '''
44   I take each key value and XOR it against the corresponding enc value.
45   This returns an integer value, which I append to a flag array.
46   I then use the chr() method to append the unicode character value
47   to a string.
48  '''
49  for m in range(1, len(key)):
50      flag.append(key[m]^enc[m])
51      flagstring += chr(flag[m])
52
53  # Printing the flag array values and the flag string.
54  print(f"{flag=}\n")
55  print(f"{flagstring=}\n")
56
```

This script, when run with the “enc” and “xorrox” values from the given output file, will derive all of the keys (except the first one which we infer), and print the flag to the screen.

Files related to this challenge and this write-up can be found at:

https://github.com/j4655/XORROX_Write-Up