CorCTF{Saturday 21st August 2021 - Monday 23rd August}
Reverse Engineering

babyrev

Minimal Effort

Sunday 22nd August

1 Description

The following description was provided: "well uh... this is what you get when you make your web guy make a rev chall", along with a compressed binary file called babyrev.

2 Process

First I just started off running the babyrev binary on a Kali docker container and seeing what it does. When you run the binary, there's no text output but it is waiting for user input. Once you type something in and press enter I could only get the binary to output the following: "rev is hard i guess...".

2.1 Main() Decompilation

So next step was to open up the binary in Ghidra to view the decompiled source. This is what the decompiled main() function looked like:

```
undefined8 main(void) {
  char cVar1;
  int iVar2;
  size_t sVar3;
  undefined8 uVar4;
  long in_FS_OFFSET;
  int local_100;
  int local_fc;
  undefined8 local_f0;
  char local_e8 [64];
```

```c
char local_a8 [27];
undefined auStack141 [37];
char local_68 [72];
long local_20;

local_20 = *(long *)(in_FS_OFFSET + 0x28);
fgets(local_e8,0x40,stdin);
sVar3 = strcspn(local_e8,"\n");
local_e8[sVar3] = '\0';
sVar3 = strlen(local_e8);
local_f0 = 7;
iVar2 = strncmp("corctf{",local_e8,7);
if (((iVar2 == 0) && (local_e8[sVar3 - 1] == '}')) && (sVar3 == 0x1c)) {
  memcpy(local_a8,local_e8 + local_f0,0x1b - local_f0);
  auStack141[-local_f0] = 0;
  local_100 = 0;
  while( true ) {
    sVar3 = strlen(local_a8);
    if (sVar3 <= (ulong)(long)local_100) break;
    local_fc = local_100 << 2;
    while( true ) {
      uVar4 = is_prime(local_fc);
      if ((char)uVar4 == '\x01') break;
      local_fc = local_fc + 1;
    }
    cVar1 = rot_n(local_a8[local_100],local_fc);
    local_68[local_100] = cVar1;
    local_100 = local_100 + 1;
  }
  sVar3 = strlen(local_68);
  local_68[sVar3 + 1] = '\0';
  memfrob(check,0x14);
  iVar2 = strcmp(local_68,check);
  if (iVar2 == 0) {
    puts("correct!");
    uVar4 = 0;
  }
  else {
    puts("rev is hard i guess...");
    uVar4 = 1;
  }
}
else {
  puts("rev is hard i guess...");
  uVar4 = 1;
}
if (local_20 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
```

```
    __stack_chk_fail();
  }
  return uVar4;
}
```

Before even beginning to read through, I changed all the hex values to decimal values to make it more readable. For future reference of the above code, hex values will be represented in their decimal form unless otherwise specified. I will also skip over lines that don't need explaining or that will be explained in the near future.

I then went through line by line to start beginning to understand what the program was doing.

local_20 = *(long *)(in_FS_OFFSET + 40); Is just storing the stack canary on the stack to ensure it isn't corrupted by the time the main function is ready to return. I'm imagining for an RE challenge that you're not expected to be performing attacks involving buffer overflows so I'm assuming at this point it's irrelevant.

```
fgets(local_e8,64,stdin);
sVar3 = strcspn(userInput,"\n");
userInput[sizeOfInput] = \0;
sizeOfInput = strlen(userInput);
```

This first chunk of code is getting 64 bytes of user input and storing it in variable local_e8, since this is our user input we will rename this variable to userInput. It then calculates the length of the number of characters before the 1st occurrence of a character present in both the strings. In our case since fgets input can be terminated by inputting a newline character, this line will then return the total amount of character in our input. Hence we will rename sVar3 to sizeOfInput.Then it simply inserts null character at the end of our userInput to ensure its terminated correctly.

```
iVar2 = strncmp("corctf{",userInput,7);
if (((cmpResult == 0) && (userInput[sizeOfInput - 1] == '}'))
&& (sizeOfInput == 28)) { //begin if
```

This compares the first 7 bytes of our userInput with the string "corctf{" and store the result in iVar2. strncmp returns 0 if the strings are identical, a number greater than 0 if the ASCII value of first unmatched character of "corctf{" is greater than userInput, or a value less than 0 if the ASCII value of first unmatched character of "corctf{" is less than userInput. Hence we will call iVar2 cmpResult. It then checks cmpResult is the == 0 meaning that the first seven characters of our input are "corctf{", it also checks that the last character is "}" and that there are 28 characters total. If any conditions fail, execution will flow into the else conditions which simply outputs: "rev is hard i guess...", sets the return value to 1 and returns.

```
memcpy(local_a8,userInput + 7,27 - 7);
local_100 = 0;
while( true ) {//begin while
```

This will copy 20 bytes starting from the 8th character (inclusive) of userInput (i.e. the flags actual value) into the address of local_a8. Hence we'll call this variable flagValue. Its pretty safe to assume that the variable local_100 is a counter for the while loop so we will just name it counter.

```
sizeOfInput = strlen(flagValue);
if (sizeOfInput <= (ulong)(long)counter) break;
local_fc = counter << 2;
```

Inside the while loop now we find the length of the flagValue string which should be 20 and updates the sizeOfInput variable. We then check to see if the amount of characters in the flagValue (i.e. 20) is less than the counter, if so then break out of the while loop, if not then continue. It then takes the counter value and left shits it by 2 and then stores that value in a variable local_fc, hence we shall call it bitShiftValue.

```
while( true ) {
uVar4 = is_prime(bitShiftValue);
```

Takes our bitShiftValue and passes it to a function called is_prime and stores the result in a variable. We'll call that variable isPrimeValue and now we will jump into the is_prime() decompilation to see what it does.

2.2 is_prime() Decompilation

If we look at Ghidra's decompilation of the is_prime() function this is what we get:

```
undefined8 is_prime(int param_1)

{
  undefined8 uVar1;
  double dVar2;
  int local_c;

  if (param_1 < 2) {
    uVar1 = 0;
  }
  else {
    for (local_c = 2; dVar2 = sqrt((double)param_1), local_c <= (int)dVar2;
    local_c = local_c+1) {
      if (param_1 % local_c == 0) {
        return 0;
      }
    }
    uVar1 = 1;
  }
  return uVar1;
}
```

It checks to see if the parameter passed into the function param_1 is less than 2, if so then it stores the value of 0 in uVar1 which we can see is out return variable. Since the numbers 0 and 1 are not prime and they cause the function to return 0, we can start beginning construct the possible functionality of the code, returning 0 for not prime numbers and 1 for prime. If we continue on through the code we can verify this assumption. From the statement above if the parameter to the function is not less then 2 then it will follow this else branch of execution. It begins a for loop starting with a counter at 2 and going until the counter is less than or equal to the square root of the parameter of the function in increments of 1.This is because there is a proof that states if there is no number that divides x that is less than the $\sqrt{x}$ then x is prime. For each iteration of the for loop it will check if the parameter to the function, modulo with the index == 0. The modulus operation will return the the remainder of a division between two numbers and is denoted by the '%' character. i.e. 4 % 3 == 1, 4 % 2 == 0, and if the numerator of the divide is smaller than the denominator then the modulo function will just return the numerator. If any number can divide the parameter to the function cleanly before the for loop ends then the parameter is not prime and the if condition inside the for loop will be taken and the function will return 0. Our theory is starting to be verified that 0 is returned for non-prime numbers. If the loop finishes and the function hasn't returned it means that nothing divided the parameter and hence the parameter is prime. is_prime() returns 0 for non-prime numbers and 1 for prime numbers. Back to main now

```
if (isPrimeValue == 1) break;
bitShiftValue = bitShiftValue + 1;
cVar1 = rot_n((int)flagValue[counter],bitShiftValue);
```

If the bit shift value is prime then it breaks out of the while loop, if not then it increments the bit shift value by 1 and tries again. This means that for whatever the count value is, left shift that value by 2 and then keep incrementing it until you find a prime number. Now we call a function called rot_n() which will take a single character which is our flagValue[counter] as well a the bitShiftValue which ended up being the first prime number after you left shifted the counter by 2. We now look at the decompiled rot_n() to see what it does:

2.3 rot_n() Decompilation

```
char rot_n(char flagvaluecharacter ,int bitShiftValue)

{
  char *pcVar1;

  pcVar1 = strchr(ASCII_UPPER,(int)flagvaluecharacter);
  if (pcVar1 == (char *)0x0) {
    pcVar1 = strchr(ASCII_LOWER,(int)flagvaluecharacter);
    if (pcVar1 != (char *)0x0) {
      flagvaluecharacter = ASCII_LOWER[(flagvaluecharacter +
      -97 + bitShiftValue) % 26];
    }
  }
  else {
    flagvaluecharacter = ASCII_UPPER[(flagvaluecharacter +
    -65 + bitShiftValue) % 26];
  }
  return flagvaluecharacter;
}
```

You will notice some values such as ASCII_UPPER and ASCII_LOWER they are just representations of the alphabet in upper case and lower case form respectively, stored in memory. It calls strchr() which takes two parameters, param1 is a string to be searched and param2 is a character to search for int the string. It searches for the first occurrence of the character, in this case the string is the upper case alphabet and the character is the flagvaluecharacter and it returns a pointer to the first occurrence of the character in the string or NULL if it doesn't exist in the string. This will then be stored in the pcVar1 variable. If the search for the character in the upper case alphabet came back as NULL then search the lowercase alphabet and store it back in pcVar1, if its not null i.e. it found the flagvaluecharacter in the string of the uppercase alphabet then take the ASCII value of the flagvaluecharacter -65 from it, (because uppercase A in ASCII is 65), giving you the position of the letter in the alphabet i.e. A is 0, B is 1 (because arrays start at 0). Then plus the bitShiftValue to the position giving you a new letter in the alphabet. You are essentially shifting the letter in the alphabet by the amount of the bitshiftvalue. You must modulo the resulting number by 26 so ensure that you don't exceed the amount of letters in the alphabet but instead wrap around to the beginning. Imagine you had the letter z, which is in the 26th character in the alphabet meaning its in the 25th position in the array. You shift that by 2, it would now be in position 27 which is larger than the size of the array that stored the alphabet and so you would get an indexOutOfBounds exception, hence you must modulo the number by 26 which is the same as looping the value around to the beginning of the alphabet. e.g. if we shifted z by 1, you would expect to get the letter a if the alphabet loops around, z is in the 25 position, shift it by 1, 25+1 = 26, 26 % 26 = 0. alphabet[0] = a. After you calculate the new character store it in flagvaluecharacter and then return that variable.

If it wasn't able to find the character in the uppercase alphabet then do the search the lower case alphabet. If the lowercase search didn't came back as NULL, i.e. it found the flagvaluecharacter in the string of the lowercase alphabet then take the ASCII value of the flagvaluecharacter subtract 97 from it giving you the characters position in the alphabet just like before, (because lowercase a in ASCII is 97). Then the rest behaves the same as the above, shift the value by the bitShiftValue, and modulo the result by 26. Return this value into the variable flagvaluecharacter and then return. Notice, that if the flagvaluecharacter fails both checks, the check for being uppercase and the check for being lowercase i.e. its not an alphabetic character, then you don't perform any calculations or manipulations on that character and you simply return it. Back to main.

So the rot_n() function took a character in the flagValue array starting at 0 since count == 0, and shifted that character, if it was an alphabetic character, by the amount of the first prime number after you left shift the count value by 2.More specifically it turns out like this:

for count = 0, left bit shift by 2 $->$ 0, first prime $>$ 0 = 2
for count = 1, left bit shift by 2 $->$ 4, first prime $>$ 4 = 5
for count = 2, left bit shift by 2 $->$ 8, first prime $>$ 8 = 11
for count = 3, left bit shift by 2 $->$ 12, first prime $>$ 12 = 13
for count = 4, left bit shift by 2 $->$ 16, first prime $>$ 16 = 17
for count = 5, left bit shift by 2 $->$ 20, first prime $>$ 20 = 23
for count = 6, left bit shift by 2 $->$ 24, first prime $>$ 24 = 29
for count = 7, left bit shift by 2 $->$ 28, first prime $>$ 28 = 29
for count = 8, left bit shift by 2 $->$ 32, first prime $>$ 32 = 37

for count = 9, left bit shift by 2 − > 36, first prime > 36 = 37
for count = 10, left bit shift by 2 − > 40, first prime > 40 = 41
for count = 11, left bit shift by 2 − > 44, first prime > 44 = 47
for count = 12, left bit shift by 2 − > 48, first prime > 48 = 53
for count = 13, left bit shift by 2 − > 52, first prime > 52 = 53
for count = 14, left bit shift by 2 − > 56, first prime > 56 = 59
for count = 15, left bit shift by 2 − > 60, first prime > 60 = 61
for count = 16, left bit shift by 2 − > 64, first prime > 64 = 67
for count = 17, left bit shift by 2 − > 68, first prime > 68 = 71
for count = 18, left bit shift by 2 − > 72, first prime > 72 = 73
for count = 19, left bit shift by 2 − > 76, first prime > 76 = 79

The prime number on the right hand side will be what is inputted into the rot_n() function for its respective count value. e.g. rot_(flagValue[5],23). So we know the shift number for every place in the flagValue so if we find some encoded flag value that is encoded using this, we can decode it.

```
local_68[counter] = cVar1;
counter = counter + 1;
//end of while here
}
sizeOfInput = strlen(rottedFlagValue);
rottedFlagValue[sizeOfInput + 1] = '\0';
```

We can see that our result from the rot_n() function for a specific counter value will be stored in a new array we'll call rottedFlagValue, at position counter i.e. rottedFlagValue[counter] = rot_n(flagValue[counter],bit shift and prime operation for counter). Then it increments counter by 1, and goes back to the beginning of the while loop. Once the counter variable reaches the size of the flagValue string then the while loop will be broken out of. It then calculates the size of the new string which should still be 20 and inserts a null character at the end to terminate it.

```
memfrob(check,20);
```

"Check" is a string of characters in the binary. Specifically these bytes:

5f 40 5a 15 75 45 62 53 75 46 52 43 5f 75 50 52 75 5f 5c 4f

These hex bytes represented in UTF-8 are:

_@Z.uEbSuFRC_uPRu_

memfrob() performs an XOR to each character in check with the number 42. This is simply done to obscure data and is not mean to be a secure encryption mechanism as stated in the man pages for the function. It performs this XOR to the first 20 characters in the check string. Once its done the hex bytes in the string have been encoded, it is not stored elsewhere, the string inputted is edited directly. If you want to decode the the string you simply call memfrob() again on the string as XORing it again with the same number will return the value back to normal. What I am assuming is, check has already been entered encoded, and this memfrob() is actually decode it.

```
    strncmpResult = strcmp(rottedFlagValue,check);
    if (strncmpResult == 0) {
      puts("correct!");
      uVar2 = 0;
    }
    else {
      puts("rev is hard i guess...");
      uVar2 = 1;
    }
  }
  else {
    puts("rev is hard i guess...");
    uVar2 = 1;
  }
```

This last bit of the main function compares the rotted flag value with the newly decoded string check. If they are the same then the program outputs "correct!" if they aren't the same then it outputs "rev is hard i guess...". So we know that the check string in memory is the memfrob() encrypted resulting string of the rot function. If we memfrob() decrypt that string and work backwards to undo the rot shifts that have been applied to each character we will be able to get the original string we need to input.

I originally decrypted the check string using CyberShef however wit the resulting string I used a rot decoder online where I manually entered in the shift value for each character in string to decode, however this was not giving me the flag. I suspected that it was an issue with the tools treating the input different from how memfrob() would be treating the input so I wrote the program below to perform the memfrob and the rot decipher for me.

```
#define _GNU_SOURCE
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int main(){

        char checkStringUTF[] = "_@Z.uEbSuFRC_uPRu_\\O";
        int rotValues[] = {2,5,11,13,17,23,29,29,37,37,41,47,53,
        53,59,61,67,71,73,79,83};
        char * memaddr;
        char final[20];
        for (int i = 0; i<20; i++){
                final[i] = 0;
        }

        printf("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");
        printf("This is the UTF output of the\nstored hex
        values: %s\n",checkStringUTF);
        memaddr = memfrob(checkStringUTF,20);
```

```
        for(int i = 0; i < 20; i++) {
                if(memaddr[i] >= 65 && memaddr[i] <= 90){
                        //Capital Letter
                        int calc = memaddr[i]-rotValues[i];
                        if(calc < 65){
                                while(calc < 65) calc = calc + 26;
                        } else if (calc > 90) {
                                while(calc > 90) calc = calc - 26;
                        }
                        final[i] = calc;

                } else if (memaddr[i] >= 97 && memaddr[i] <= 122) {
                        //Lowercase
                        int calc = memaddr[i]-rotValues[i];
                        if(calc < 97){
                                while(calc < 97) calc = calc + 26;
                        } else if (calc > 122) {
                                while(calc > 122) calc = calc - 26;
                        }
                        final[i] = calc;
                }
        }

        printf("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");
        printf("This is what is converts to when\nyou run memfrob():
        %s\n",memaddr);
        printf("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");
        printf("This is my final output:");
        for(int i = 0; i<20; i++){
                if(i == 3 || i == 8 || i == 14 || i == 17){
                        printf("_");
                }
                printf("%c",final[i]);

        }



        return 1;
}
```

checkStringUTF is the array that holds the UTF-8 encoding of the hex bytes that we found in the memory of the binary for the check string with one caveat. There was one hex byte that didn't convert to UTF-8 nicely which was 0x15. The rotValues array are the shift values we calculated earlier with the bit shift and primality check functions. This is the output of the memfrob() function on the checkStringUTF: ujp_oHy_lxiu_zx_uve, this is very different to the output of XORing each of the checkStringUTF characters with 42 using CyberChef which why my initial decryption and decipher of the check string was wrong. The program then reverses the rot shift for each character in the resulting memfrob() string and it decodes to "see_rEv_aint_so_bad". This is looking pretty good up until you realise that there is one character missing. Remember back at the beginning of the main function one of the conditions was the your input has to be 28 characters long, the first 7 must be "corctf{" and the last character must be "}" hence there is 20 characters left for the flag value but the above output only have 19 characters. I believe we have lost that character represented by hex value 0x15. To fix this I wrote a python script that brute forces each 128 ASCII characters in every positions of the flag value. The code is below:

```python
import string
from pwn import *
import time

flagv = "see_rEv_aint_so_bad"

for _ in string.printable:
    for i in range(0,len(flagv)+1):
        flag = "corctf{" + flagv[:i] + _ + flagv[i:] + "}"
        p = process("./babyrev")
        p.sendline(flag)
        print("Trying: "+flag)
        answer = p.recvline()
        if "correct" in answer.decode("utf-8"):
            print("CONGRATS")
        else:
            p.close()
```

It uses pwn tools to start up processes running the babyrev binary, sends it one of the brute force inputs and then if it is correct it outputs "CONGRATS" if its wrong it just moves onto the next try. This resulted in finding the flag which was: corctf{see?_rEv_aint_so_bad}