CSAW'24 Archeology Writeup

So I participated in CSAW'24 and managed to solve Archeology, and I wanted to do a writeup about it.

So lets start, we get from the challenge 3 files: chal, message and hieroglyphs. The challenge is to decrypt the message that was encrypted using the chal binary. We first try to run chal and see that it requires a command line argument as the text to encrypt.

```
└─$ ./chal
Usage: ./chal <data-to-encrypt>
```

So we open up a disassembler tool such as ghidra and start reading the code. We notice that the text we input is stored inside pcVar7.

```
pcVar7 = (char *)param_2[1];
```

Then pcVar7 is sent to the "washing_machine" function. So we look inside and see that this function XORs (with the '^' simbol) each character in the string with the previous one (meaning the first character, index 0, is left unchanged) and then reverses the string.

```
void washing_machine(byte *param_1,ulong param_2)

{
  byte bVar1;
  byte local_1b;
  ulong local_18;
  ulong local_10;

  local_1b = *param_1;
  for (local_18 = 1; local_18 < param_2; local_18 = local_18 + 1) {
    local_1b = param_1[local_18] ^ local_1b;
    param_1[local_18] = local_1b;
  }
  for (local_10 = 0; local_10 < param_2 >> 1; local_10 = local_10 + 1) {
    bVar1 = param_1[local_10];
    param_1[local_10] = param_1[(param_2 - local_10) + -1];
    param_1[(param_2 - local_10) + -1] = bVar1;
  }
  return;
}
```

Then we notice the big for loop that puts each character of the text in a specific place and adds a lot of additional characters, all these characters are stored inside local_2728. Then another character is being added to local_2728 ('\a') probably to signal the end of the string

since '\0' is being used in the string so it can't be used as to end the string.

```
for (iStack_12764 = 0; iStack_12764 < iVar5; iStack_12764 = iStack_12764 + 1) {
  local_2728[iStack_12768] = '\0';
  iVar1 = iStack_12768 + 2;
  local_2728[iStack_12768 + 1] = '\x01';
  iStack_12768 = iStack_12768 + 3;
  local_2728[iVar1] = pcVar7[iStack_12764];
  for (iStack_12760 = 0; iStack_12760 < 10; iStack_12760 = iStack_12760 + 1) {
    local_2728[iStack_12768] = '\0';
    local_2728[iStack_12768 + 1] = '\0';
    local_2728[iStack_12768 + 2] =
         *(char *)((long)&local_272d + (long)((iStack_12760 + iStack_12764 * 10) % 5));
    local_2728[iStack_12768 + 3] = '\b';
    local_2728[iStack_12768 + 4] = '\x01';
    local_2728[iStack_12768 + 5] = '\x03';
    local_2728[iStack_12768 + 6] = '\x03';
    local_2728[iStack_12768 + 7] = '\x01';
    local_2728[iStack_12768 + 8] = '\x01';
    local_2728[iStack_12768 + 9] = '\x01';
    local_2728[iStack_12768 + 10] = '\0';
    local_2728[iStack_12768 + 0xb] = '\x02';
    iVar1 = iStack_12768 + 0xd;
    local_2728[iStack_12768 + 0xc] = '\x01';
    iStack_12768 = iStack_12768 + 0xe;
    local_2728[iVar1] = '\x03';
  }
  local_2728[iStack_12768] = '\x04';
  iVar1 = iStack_12768 + 2;
  local_2728[iStack_12768 + 1] = '\x01';
  iStack_12768 = iStack_12768 + 3;
  local_2728[iVar1] = (char)iStack_12764;
}
local_2728[iStack_12768] = '\a';
```

The for loop also uses the local_272d and local_2729 with values 0xaabbccdd and 0xee respectively and local_2729 is being defined right after local_272d, this is important because we can see that a value in the new string (local_2728) is determined by a specific byte in local_272d and it's determined by modding the loop variable (iStack_12760) by 5.
But local_272d is an int which has 4 bytes, so since local_2729 is a char and was defined right after local_272d, when the value of the mod will be 4 it will overflow from local_272d into local_2729 and the value will be 0xee.

```
undefined4 local_272d;
undefined local_2729;

local_272d = 0xddccbbaa;
local_2729 = 0xee;
```

(In ghidra it is represented as undefined but the number after is the number of bytes the parameter will have)

After that, the string (local_2728) is sent to the "runnnn" function which looks complicated but it's not.

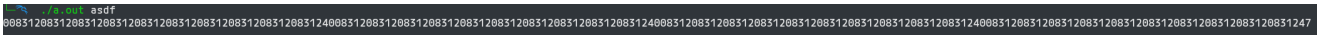
```
void runnnn(long param_1)

{
  int iVar1;
  int iVar2;
  byte bVar3;
  byte bVar4;
  bool bVar5;
  int local_10;

  local_10 = 0;
  bVar5 = true;
  while (bVar5) {
    iVar1 = local_10 + 1;
    switch(*(undefined *)(param_1 + local_10)) {
    case 0:
      iVar2 = local_10 + 2;
      local_10 = local_10 + 3;
      *(undefined *)((long)&regs + (long)(int)(uint)*(byte *)(param_1 + iVar1)) =
           *(undefined *)(iVar2 + param_1);
      break;
    case 1:
      iVar2 = local_10 + 2;
      local_10 = local_10 + 3;
      *(byte *)((long)&regs + (long)(int)(uint)*(byte *)(param_1 + iVar1)) =
           *(byte *)((long)&regs + (long)(int)(uint)*(byte *)(param_1 + iVar1)) ^
           *(byte *)((long)&regs + (long)(int)(uint)*(byte *)(param_1 + iVar2));
      break;
    case 2:
      iVar2 = local_10 + 2;
      bVar3 = *(byte *)(param_1 + iVar1);
      local_10 = local_10 + 3;
      bVar4 = *(byte *)(param_1 + iVar2);
      *(byte *)((long)&regs + (long)(int)(uint)bVar3) =
           *(char *)((long)&regs + (long)(int)(uint)bVar3) << (bVar4 & 0x1f) |
           (byte)((int)(uint)*(byte *)((long)&regs + (long)(int)(uint)bVar3) >> (8 - bVar4 & 0x1f));
      break;
    case 3:
      local_10 = local_10 + 2;
      *(undefined1 *)((long)&regs + (long)(int)(uint)*(byte *)(param_1 + iVar1)) =
           sbox[(int)(uint)*(byte *)((long)&regs + (long)(int)(uint)*(byte *)(param_1 + iVar1))];
      break;
    case 4:
      iVar2 = local_10 + 2;
      local_10 = local_10 + 3;
      memory[(int)(uint)*(byte *)(param_1 + iVar2)] =
           *(undefined *)((long)&regs + (long)(int)(uint)*(byte *)(param_1 + iVar1));
      break;
    case 5:
      iVar2 = local_10 + 2;
      local_10 = local_10 + 3;
      *(undefined1 *)((long)&regs + (long)(int)(uint)*(byte *)(param_1 + iVar1)) =
           memory[(int)(uint)*(byte *)(param_1 + iVar2)];
      break;
    case 6:
      local_10 = local_10 + 2;
      putchar((uint)*(byte *)((long)&regs + (long)(int)(uint)*(byte *)(param_1 + iVar1)));
      break;
    case 7:
      bVar5 = false;
      local_10 = iVar1;
      break;
    case 8:
      bVar3 = *(byte *)(param_1 + iVar1);
      bVar4 = *(byte *)(param_1 + (local_10 + 2));
      *(byte *)((long)&regs + (long)(int)(uint)bVar3) =
           (byte)((int)(uint)*(byte *)((long)&regs + (long)(int)(uint)bVar3) >> (bVar4 & 0x1f)) |
           *(char *)((long)&regs + (long)(int)(uint)bVar3) << (8 - bVar4 & 0x1f);
      local_10 = local_10 + 3;
      break;
    default:
      puts("Invalid instruction");
```

```
      bVar5 = false;
      local_10 = iVar1;
    }
  }
  return;
}
```

To understand it better we can copy this function into a c file edit it a bit so it will work and run it in a debugger. But this function needs the for loop to set up the string to work. Then we run it and start stepping in the function see what it does. Something I noticed when doing that is that there is a pattern in the loop so I added printf function to each case printing the number of the case and got this:



So there is a pattern, it goes to case 0 and then it iterates through cases 0,8,3,1,2 by this order 10 times and then it goes to case 4. It does this iteration for all the characters and then goes to case 7 which exits the loop. So to understand what "runnnn" does we need to understand these cases, so lets going through them:

- case 0: Sets a certain value of the string in the reg variable.
- case 1: XORs a character of the string with the next one.
- case 2: Rotates regs to the left by a value in the string.
- case 3: Substitutes the value of regs with a value in the sbox with regs value being the index.
- case 4: Inserts the value of regs into the memory variable.
- case 8: Rotates regs to the right by a value in the string.

So since we found a pattern in the loop maybe there are other patterns. So using gdb we can set a break point in each of the relevant cases, them being 1,2,3,8 as these are the only ones where there are calculations, and look for other patterns.

Lets start with case 1. We can print out the value of the next variable and see what it is.

```
Breakpoint 1, runnnn (param_1=140737488336320) at runnnn.c:61
61              *(char *)((long)&regs +
(gdb) print *(unsigned char *)((long)&regs + (long)*(char *)(param_1 + iVar2))
$12 = 170 '\252'
(gdb) continue
Continuing.

Breakpoint 1, runnnn (param_1=140737488336320) at runnnn.c:61
61              *(char *)((long)&regs +
(gdb) print *(unsigned char *)((long)&regs + (long)*(char *)(param_1 + iVar2))
$13 = 187 '\273'
(gdb) continue
Continuing.

Breakpoint 1, runnnn (param_1=140737488336320) at runnnn.c:61
61              *(char *)((long)&regs +
(gdb) print *(unsigned char *)((long)&regs + (long)*(char *)(param_1 + iVar2))
$14 = 204 '\314'
(gdb) continue
Continuing.

Breakpoint 1, runnnn (param_1=140737488336320) at runnnn.c:61
61              *(char *)((long)&regs +
(gdb) print *(unsigned char *)((long)&regs + (long)*(char *)(param_1 + iVar2))
$15 = 221 '\335'
(gdb) continue
Continuing.
```

We can see that this values are the values of local_272d. So case 1 XORs by 0xaa then 0xbb and so on until 0xee.

Lets view case 2 now. We can see that the value of bVar4 (the variable that determines how much bits to shift) is always 3.

```
(gdb) b 74
Breakpoint 1 at 0x1374: file runnnn.c, line 74.
(gdb) r
Starting program: /home/harel/CSAW/Archeology/a.out asdf
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, runnnn (param_1=140737488336320) at runnnn.c:74
74            *(char *)((long)&regs + (long)(int)(unsigned int)bVar3) =
(gdb) print bVar4
$1 = 3 '\003'
(gdb) continue
Continuing.

Breakpoint 1, runnnn (param_1=140737488336320) at runnnn.c:74
74            *(char *)((long)&regs + (long)(int)(unsigned int)bVar3) =
(gdb) print bVar4
$2 = 3 '\003'
(gdb) continue
Continuing.

Breakpoint 1, runnnn (param_1=140737488336320) at runnnn.c:74
74            *(char *)((long)&regs + (long)(int)(unsigned int)bVar3) =
(gdb) print bVar4
$3 = 3 '\003'
(gdb) continue
Continuing.

Breakpoint 1, runnnn (param_1=140737488336320) at runnnn.c:74
74            *(char *)((long)&regs + (long)(int)(unsigned int)bVar3) =
(gdb) print bVar4
$4 = 3 '\003'
(gdb)
```

Moving on to case 3. Here we just need to find the sbox and all its 256 values. We can view it in the disassembler.

```
                    sbox


00102020 48 5c bc        undefine...
         97 81 91
         60 ad 94 ...
    00102020 48              undefined148h                     [0]
    00102021 5c              undefined15Ch                     [1]
    00102022 bc              undefined1BCh                     [2]
    00102023 97              undefined197h                     [3]
    00102024 81              undefined181h                     [4]
    00102025 91              undefined191h                     [5]
    00102026 60              undefined160h                     [6]
    00102027 ad              undefined1ADh                     [7]
    00102028 94              undefined194h                     [8]
    00102029 cb              undefined1CBh                     [9]
    0010202a 92              undefined192h                     [10]
    0010202b 39              undefined139h                     [11]
    0010202c 1a              undefined11Ah                     [12]
    0010202d 0f              undefined10Fh                     [13]
    0010202e 30              undefined130h                     [14]
    0010202f 2d              undefined12Dh                     [15]
    00102030 45              undefined145h                     [16]
    00102031 de              undefined1DEh                     [17]
    00102032 14              undefined114h                     [18]
    00102033 a2              undefined1A2h                     [19]
    00102034 08              undefined108h                     [20]
    00102035 57              undefined157h                     [21]
    00102036 b6              undefined1B6h                     [22]
    00102037 ae              undefined1AEh                     [23]
    00102038 76              undefined176h                     [24]
    00102039 8e              undefined18Eh                     [25]
    0010203a 87              undefined187h                     [26]
```

Now to case 8. It's the same as case 2 just shifting the other direction, bVar4 is 3.

```
(gdb) b 122
Breakpoint 1 at 0x158f: file runnnn.c, line 122.
(gdb) r
Starting program: /home/harel/CSAW/Archeology/a.out asdf
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, runnnn (param_1=140737488336320) at runnnn.c:122
122             *(char *)((long)&regs + (long)(int)(unsigned int)bVar3) =
(gdb) print bVar4
$1 = 3 '\003'
(gdb) continue
Continuing.

Breakpoint 1, runnnn (param_1=140737488336320) at runnnn.c:122
122             *(char *)((long)&regs + (long)(int)(unsigned int)bVar3) =
(gdb) print bVar4
$2 = 3 '\003'
(gdb) continue
Continuing.

Breakpoint 1, runnnn (param_1=140737488336320) at runnnn.c:122
122             *(char *)((long)&regs + (long)(int)(unsigned int)bVar3) =
(gdb) print bVar4
$3 = 3 '\003'
(gdb) continue
Continuing.

Breakpoint 1, runnnn (param_1=140737488336320) at runnnn.c:122
122             *(char *)((long)&regs + (long)(int)(unsigned int)bVar3) =
(gdb) print bVar4
$4 = 3 '\003'
(gdb)
```

Now that we know what "runnnn" does, we can continue. After "runnnn", the encrypted string is in the memory variable and is sent to the "washing_machine" function again. After that, we can see a while loop where the glyphs from the hieroglyphs.txt file are being read and copied into the acStack_12738 variable.

```
__stream = fopen("hieroglyphs.txt","r");
if (__stream != (FILE *)0x0) {
  iStack_1275c = 0;
  while( true ) {
    *(undefined8 *)(puVar8 + -0x1788) = 0x101b8a;
    pcVar7 = fgets(acStack_12738 + (long)iStack_1275c * 0x100,0x100,__stream);
    if ((pcVar7 == (char *)0x0) || (0xff < iStack_1275c)) break;
    *(undefined8 *)(puVar8 + -0x1788) = 0x101b37;
    sVar6 = strcspn(acStack_12738 + (long)iStack_1275c * 0x100,"\n");
    acStack_12738[sVar6 + (long)iStack_1275c * 0x100] = '\0';
    iStack_1275c = iStack_1275c + 1;
  }
```

Then, there is another loop that prints the encrypted text. Inside the loop we see that each value in the memory variable is stored inside bVar2 and it's be used as an index to print the

glyphs out from acStack_12738.

```
for (iStack_12758 = 0; iStack_12758 < iVar5; iStack_12758 = iStack_12758 + 1) {
  bVar2 = memory[iStack_12758];
  *(undefined8 *)(puVar8 + -0x1788) = 0x101c01;
  printf("%s",acStack_12738 + (long)(int)(uint)bVar2 * 0x100);
}
```

So now we need to figure out how to decrypt the message, so lets write the decryption steps:

- Find the indexes of the glyphs in the encrypted text and put these indexes as chars in an array
- Perform the inverse function to the "washing_machine" function.
- Perform the inverse function to the "runnnn" function.
- Perform the inverse function to the "washing_machine" function again.
- Print the decrypted text.

So now we have a plan but we still need to figure out the inverse functions. Lets start with the "washing_machine", this function XORs each char with the next one and then reverses the string. To inverse this we need to reverse the string, and then, since XOR is the inverse of itself, XOR each char with the previous one starting for the end of the string. This is how it should look like:

```
def inv_wash(cipher_text):
    cipher_text = cipher_text[::-1]

    for i in range(len(cipher_text) - 1, 0, -1):
        cipher_text[i] ^= cipher_text[i - 1]

    return cipher_text
```

Now lets inverse "runnnn". Lets do it case be case for simplicity's sake, and we can ignore cases 0 and 4 since these cases just set values in regs or memory.

- case 1: We know the values being XORed with so it inverse we need to XOR by the same values. So since it start with 0xaa and ends with 0xee we will set an array like this: `xor = [0xaa, 0xbb, 0xcc, 0xdd, 0xee]`
- case 2: Since case 2 rotates left we need to rotate right.
- case 3: To inverse this we first need to find the inverse of the sbox. I wrote a script to do that:

```c
int main() {

  unsigned char inv_sbox[256];

  printf(format: "inv_sbox = { ");

  for (int i = 0; i < 256; i++) {
    inv_sbox[sbox[i]] = i;
  }

  for (int i = 0; i < 256; i++) {
    printf(format: "%d, ", (int)inv_sbox[i]);
  }

  printf(format: "}\n");

  return 0;
}
```

```
└─  ./a.out
inv_sbox = { 74, 252, 195, 85, 38, 40, 63, 216, 20, 158,
93, 151, 146, 202, 110, 11, 56, 164, 236, 79, 57, 157, 2
2, 206, 34, 181, 233, 159, 224, 132, 127, 145, 42, 24, 9
 113, 54, 99, 36, 222, 213, 106, 78, 144, 7, 23, 65, 67,
16, 153, 17, 100, 230, 140, 172, 82, 234, 84, 114, 29, 1
```

- case 8: Since this case rotates right we need to rotate left.

Now we can build the inverse function of "runnnn":

```python
def inv_runnnn(cipher_text):
    for i in range(0, len(cipher_text)):
        for j in range(9, -1, -1):
            cipher_text[i] = (((cipher_text[i] >> 3) | (cipher_text[i] << 5)) & 0xff)
            cipher_text[i] ^= xor[j % 5]
            cipher_text[i] = inv_sbox[cipher_text[i]]
            cipher_text[i] = (((cipher_text[i] << 3) | (cipher_text[i] >> 5)) & 0xff)

    return cipher_text
```

Now we just need to build the main function, run it and get the flag.

```
def main():
    cipher_text = []

    with open("message.txt", 'r') as m:
        message = m.read()

    with open("hieroglyphs.txt", 'r') as h:
        glyphs = h.read().splitlines()

    for i in range(0, len(message)):
        cipher_text.append(glyph_to_num(message[i], glyphs))

    cipher_text = inv_wash(cipher_text)
    cipher_text = inv_runnnn(cipher_text)
    cipher_text = inv_wash(cipher_text)

    print(bytes(cipher_text))
```