PWNED!

t2'15 challenge solutions

Juha Kivekäs

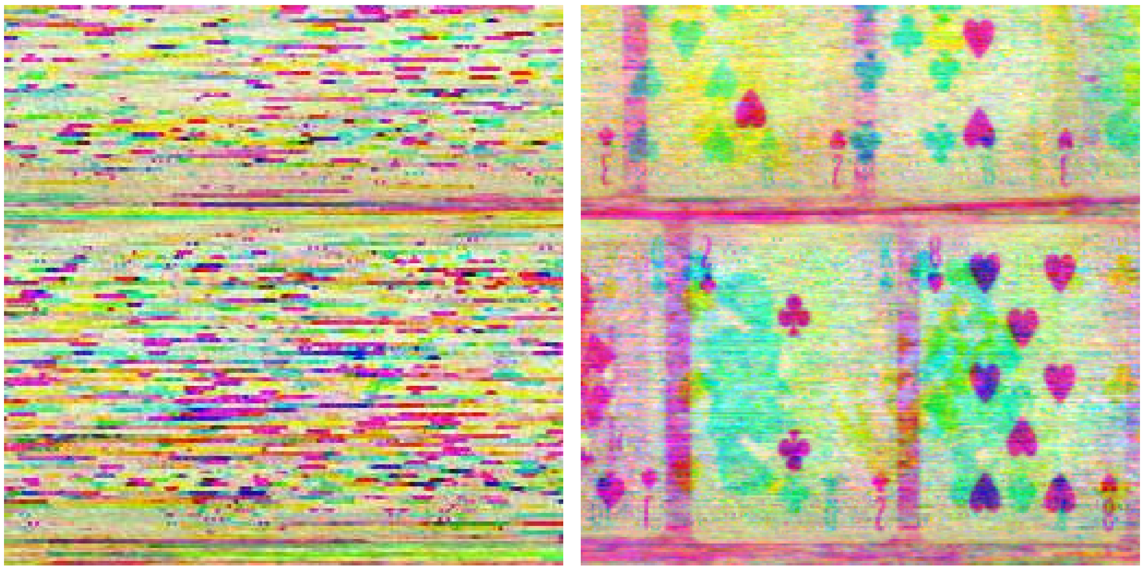
Figure 1: A cropping of the scrambled image and the same area unscrambled.

Acquire Encryption Key to Director Communications

The first stage in the challenge was to break an image scrambling cipher. The JavaScript scrambling function takes an integer key and generates three sub-keys for each of the primary color channels. These channel keys would be seeds for PRNGs that modify the key for every line of the image. The scrambling rotates each channel on a row an amount of pixels determined by the PRNGs. Scrambling operates on 2x2 pixel squares, but this detail is not relevant for this solution.

Since all row data is kept intact except for the horizontal synchronization of the colors, we can re-sync the shifts done by the scrambler. The way I chose to do this was to sync each color on every row with the row above it. This works because the change in data between single pixel rows is actually quite small in most images.

The correct shift value will be chosen such that we get the least square sum of the pixel differences. This will sync the individual channels quite nicely, though not perfectly. Syncing the color channels also to each other could be done for getting the colors globally synced, not just per-channel. This was going to be the next step, but it wasn't needed for obtaining the key. One thing to note is that we can use the unscrambler also to scramble images for use in testing our cracker.

Figure shows a crop of the original image and the resulting unscrambled image. With some editing we can choose a channel where some text on bottom is most visible and enhance to get the text readable. Just take a sha1 of the string and we get the hash.

```
8264dcfe05f46e42937b65d78f95e2daece006cd
```

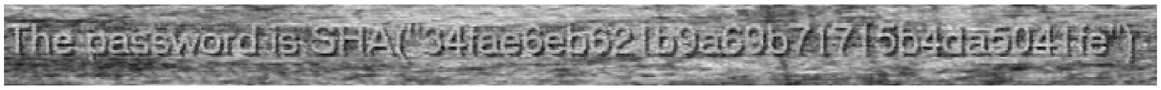
The password is SHA("34fae6eb621b9a69b7f7f5b4da504ffe").

Figure 2: The solution to the challenge visible.

Tools used

- C
- convert (ImageMagick)
- GIMP

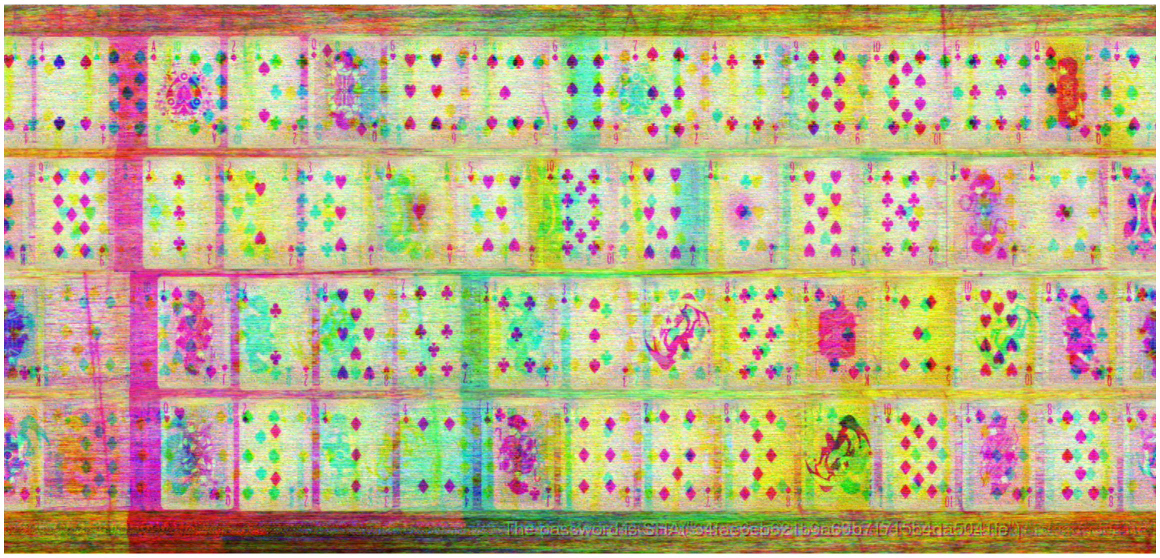

Figure 3: The reconstruction of the deck image.

Letter by the Director

The letter to the director is a follow-up to the acquisition of the key for director communications. The cipher used is the pontifex cipher as described in the Cryptonomicon [1]. An explanation of the cipher is available on the creators blog [2].

In pontifex the key used is a deck of cards. Sending an image to the recipient with the ordering of the cards visible is one of the suggested ways of transmitting a key so we only need to reconstruct the deck now. The ordering of the cards can be seen in the unscrambled image and then put in a format that some implementation of the cipher supports. The deck is laid out with the top card in the upper left corner and the cards running from left to right. After decryption we can take the sha1 of the cleartext file and we get the hash. There are some nasty whitespace that has to match the original file, so this took a couple of tries before it went right.

```
24a0c082cf931a281e917eacf06aba10de6ed8cf
```

Tools Used

- python
- perl implementation of pontifex [2]

Missile System Decryptor

The decryption program is actually a properly implemented Linear Congruential Generator stream cipher. The values used in the LCG are well chosen [3] and the hash function can output any 64-bit key if a well chosen password is used. The generator only has a period of 2^{64} so it's in the realm of doing an exhaustive search on the key space. LCGs are broken and show a lot of linearity so there is prbably a more sophisticated way to attack this.

We can do an exhaustive search on this by iterating over all 8-byte blocks that could be the start of the file. By xoring the supposed cleartext with the ciphertext we get the key to which we can apply the LCG to decrypt the following blocks. If the few following blocks seem like the expected cleartext there is a chance we used the right key for decryption. After manual verification of the most likely cleartexts we can obtain the correct ciphertext and key.

One of the hints [4] tells us indirectly that the cleartext is a gzip file since the gzip format was initially released on 31.10.1992 [5], the date in the commented section of the source code. This gives us a lot of information on the data contained in the cleartext. When the gzip command-line tool is used with default settings the files have a structure like this:

```
1f 8b 08 08 xx xx xx xx
00 yy zz zz zz zz zz zz
zz zz zz zz zz zz zz zz
```

Here `x` is an epoch time value, `y` an operating system flag, and `z` a null terminated filename in ISO 8859-1 encoding. The first eight bytes in gzip files contain some static bytes and a timestamp which makes the key space radically smaller. Namely the key space is 2^{32}, which and can be searched within a minute. Parsing and storing the decrypts in an easily browsable format will help ruling out the false positives. I used CSV files and browsed them with SQLite and grep.

As we find the correct cleartext we get the key to decrypt the whole file. Now we can gunzip the decrypted file and get the `missile_system.bf` file which gets us our next hash.

```
832a1583c4ce5f5cd0b797577e9a11cb567cc138
```

Tools used

- C and pthread
- sed, grep, sqlite3
- radare2 (for hexdumps)

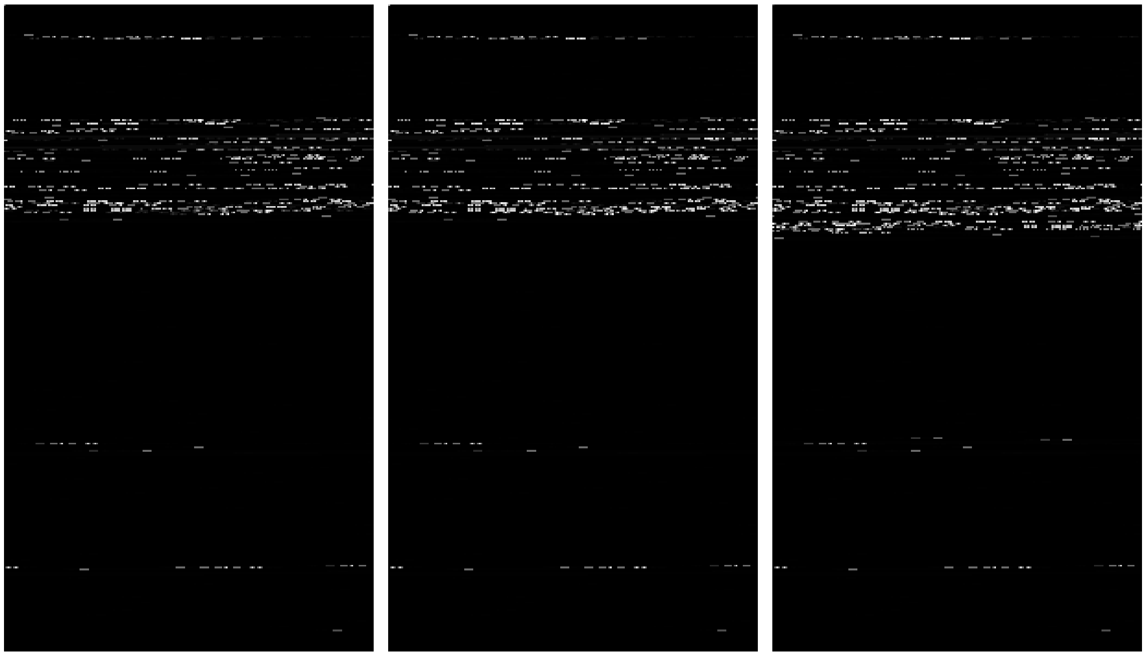

Figure 4: The coverages for the empty password and passwords "t" and "u" visualized.

Missile System Verifier

This challenge was based on the decryption of the `missile_system.enc` file. A large brainfuck [6] program was given that would prompt for a password and tell "Sorry!" and enter an infinite busy loop when an incorrect password was given. An interpreter [7] written in C was used for running the program.

Assuming the password check is something like strcmp that will return early if the password is incorrect we can argue that a timing attack would be very efficient. Since the program enters a busy loop, any simple timing attack is hard since we don't know when to stop execution and measure the execution time.

By noting that there is only one input ',' and one output '.' command in the program we can apply some instrumentation. These are interesting points since we know that all password processing happens between the first execution of ',' and the following execution of '.'. In practice this means that processing happens after we give input, but before "Sorry!" in printed.

The interpreter was modified to store information on code coverage during execution. A patch of the modifications is available in appendix A. This way the amount of times each instruction is executed can be extracted. By collecting coverage data between the first input instruction and the following output instruction we can see how the execution path differs when the program is given different passwords.

The first observation we make is that inputs over ten characters long exit very early, so the password length can be guessed to be 10. We can collect the "coverage of a password" by simply running the program, give it a password, and saving the coverage file. By first collecting the coverage of an empty password and the trying out all single letter passwords and comparing their coverages by absolute error to the coverage of the empty password we find that the coverage for the password "u" differs most from the empty password coverage. This indicates that the password starts with "u", see figure .

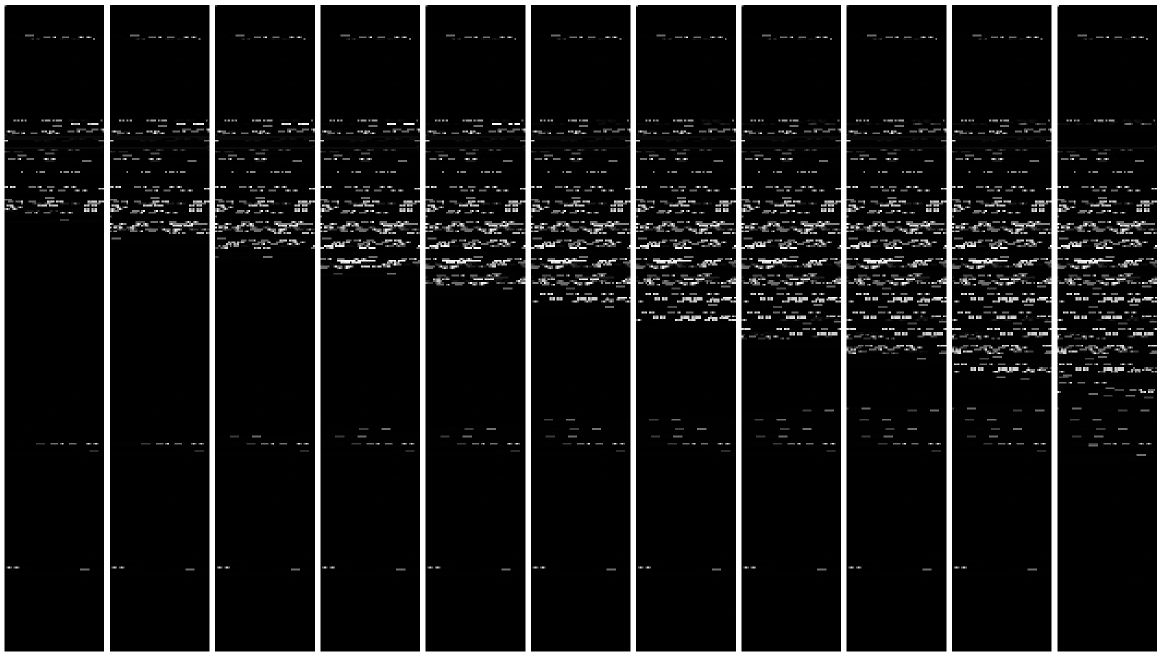

Figure 5: The coverage increases for every correct character in the pasword.

By always trying all the characters and comparing the coverage to the coverage of the previously known longest password we can find which character is the correct one to append to our known password. In figure you can see how the code coverage increases when the amount of correct characters in the password increases from zero to ten.

The coverage oracle used in this attack is very powerful and gives us the password within a minute and by taking the sha1 of it we get the hash.

```
c27257f5163a442e7d26593aeabc8e29e5edf05c
```

Tools used

- convert and compare (ImageMagick)
- gnuplot
- vimdiff
- patched brainfuck interpreter [7]

Historical meeting instructions

The cipher in this challenge is a combination of old-school ciphers. First there is an autokey vigenere cipher and after that part of the text is still encrypted with a rotation cipher.

Using the index of coincidence we can deduce that the key length is 16 characters. As the original email starts with names, question marks, and exclamation marks we can get a general feel of the aggressiveness in message. Using this and the word sizes in the response we can guess that the response starts with "I am very sorry". An automatic vigenere solver like the one given in the hint [4] can also be used to get started on this cipher and strengthen these observations.

In order for the first 16 character block to decrypt to "iamverysorryxxxx" the key should be "tloyhwmlszayxxxx". When we apply autokey deciphering with this key we get the first sentence to be:

```
I am very sorry jbf ool klshf olyy veykwommando Franodrbnwljrly.
```

Since we know the sender is apologizing to "Franozenspecker" we can fix the key to be "tloyhwmlszaybkls". We are still getting a lot of garbled output, namely every even numbered block of 16 characters.

```
I am very sorry for tol klshf olyy vilyrommando Franozenswljrly.
```

By guessing some words like "the", "here is the recipe", and "oberkommando" from the first paragraph we quickly notice that the remaining cipher is a monoalphabetic substitution cipher. Breaking the substitution cipher is just a matter of generating the substitution table as we go. Finding readable words in the text to support our guessing is not a problem as there is a lot of semi-clear text available. The substitution turns out to be just a rotation by seven.

After having the whole reply email deciphered we get to know that "heinz loves apfelstrudel" which gives us the sha1 for this level. He actually loves it so much he had it as a part of hes password "mehrapfelstrudel" which is "tloyhwmlszaybkls" rotated backwards by seven. Using "mehrapfelstrudel" as the key means the rotation will be applied to the odd blocks rather than the even ones.

```
dc575fc48ef547e39ffe4ab2dd3391796e1c27f0
```

Tools used

- python with matplotlib
- gnuplot
- online vigenere solver [8]
- C with ANSI escape codes for visual output

Authentication cookie

If we change the last byte of the 'auth' cookie on the website we will get a padding error as response from the server. This gives us a padding oracle that can be used to attack the encryption of the cookie. The cookie uses the CBC mode of operation and by using the avalanche effect and the oracle we can decrypt the message one character at a time. The general description of the padding oracle attack is out of scope for this solution and is described in-depth elsewhere [9].

```
def oracle(auth):
    auth_b64 = str(base64.b64encode(bytes(auth)), 'utf-8')
    r = requests.get('http://127.0.0.1:12345/issue37.txt', cookies={'auth':auth_b64})
    return r.text[0:16] != 'IncorrectPadding'
```

This oracle can be plugged in to an implementation of the attack and soon we'll have most of the encrypted data. As long as we don't know the IV used in the CBC encryption we will not be able to decrypt the very first block of the cookie. However we can reconstruct the rest of the message to be:

```
h33&password=NistOmgc1992&expiry=never
```

From here we get a password `NistOmgc1992`, and by taking the sha1 of it we get the hash for this level.

```
4367e5919f5d622a0cd4a8bea00983c236bc2672
```

Tools used

- python
- IceWeasle

References

[1] Cryptonomicon by Neal Stephenson

[2] Bruce Schneiers blog, `https://www.schneier.com/solitaire.html`

[3] Linear congurental generators, `http://en.wikipedia.org/wiki/Linear_congruential_generator`

[4] The hints published during the challenge, `http://t2.fi/tag/hint/`

[5] `http://en.wikipedia.org/wiki/Gzip`

[6] Brainfuck on esolang, `http://esolangs.org/wiki/brainfuck`

[7] Brainfuck interpreter in C, `http://github.com/nicokoch/brainfuck-interpreter`

[8] Online vigenere solver, `http://www.guballa.de/vigenere-solver`

[9] Security flaws induced by CBC padding applications to SSL, IPSEC, WTLS..., Serge Vaudenay, `http://www.iacr.org/cryptodb/archive/2002/EUROCRYPT/2850/2850.pdf`

Appendix A: Coverage patch

Original interpterer is availabale on github [7]

Apply this patch to commit c995136a0432341f27264e613eff6a96c2153170 to get the version of the interpreter used in the solution. The patch is made specifically for the challenge but is good for general purpose coverage reporting with small adjustments. For a license, see the repository.

```
108a109
>     long location;
117a119
>     comm->location = 0;
141c143
<
---
>             comm->location = i;
271a274,281
> char coverage[70000];
> void dump_coverage(){
>   FILE* fp = fopen("coverage.dat", "wb");
>     fwrite(coverage, sizeof(char), 70000, fp);
>     fclose(fp);
>     exit(0);
> }
>
273a284,285
>     int i, state = 0;
>     memset(coverage, 0x00, sizeof(char)*70000);
279a292,296
>         if(state == 1){
>             for(i=0; i<command_struct->magnitude; i++){
>                 coverage[command_struct->location+i]++;
>             }
>         }
297a315,318
>             if(state == 1){
>                 dump_coverage();
>                 state = 2;
>             }
300a322,324
>             if(*ptr == '\n'){
>                 state = 1;
>             }
```