Information Security Challenge 2: 20.4.2020

Hints

In the following we provide a sequence of hints that should help the reader to successfully complete the challenge. We assume that the reader has successfully installed the required software and has a running instance of the Docker container (see document Challenge02TechBackground.pdf).

To check if the container is running, simply use your favorite browser and visit the URL http://localhost:5000. In case there is no error and a simple message is shown referring to the location of the crypto API, the server is running and you can start with the challenge.

Hint 1: The crypto API

As it has already been mentioned, the server is running on your local host on TCP port 5000. So, we might simply try to connect to the server with a simple GET request.

```
curl http://localhost:5000
```

The answer of the server points us to the sub-directory /cryptoapi of the server, where we should find the crypto API.

Executing for example:

```
curl http://localhost:5000/cryptoapi
```

returns an error code, telling us that the corresponding method (GET) is not allowed. We thus might try another method, for example try to execute a POST request.

Firing a simple POST request to the server again leads to an HTTP error code. This time the method seems to work, but text reveals that parameters `prefix`, `ciphertext`, and `postfix` are missing and obviously need to have a certain format.

Of course, you can use a trial and error method at this point to guess the correct formatting of the parameters for the request.

Hint 2: The structure of the API

As a next hint you get a description of the API as it might be available for a modern Web API in OpenAPI 3.0 format:

```
openapi: 3.0.0
info:
  title: Simple Encryption Oracle
  description: This is a simple crypto API
  contact:
    email: patrick.schaller@ar.admin.ch
  license:
    name: Apache 2.0
    url: http://www.apache.org/licenses/LICENSE-2.0.html
  version: 1.0.0-oas3
servers:
- url: /
paths:
  /:
    get:
      summary: info about crypto API
      description: Simply shows the path to the crypto API
      operationId: simpleInfo
      responses:
        "200":
          description: Shows path to crypto API
          content:
            text/plain:
              schema:
                type: string
  /cryptoapi:
    post:
      summary: takes a prefix, a ciphertext, and a postfix
      description: The API consumes a prefix in cleartext (HEX
           encoded bytes), a ciphertext (HEX encoded bytes),
           and a postfix in cleartext (HEX encoded bytes). The
           API decrypts the provided ciphertext, concatenates
           the prefix, the cleartext (of the given
           ciphertext), and the postfix and finally returns
           the encrypted concatenation.
           Note that the length of the concatenated string has
           to be a multiple of the cipher's block size (128
           bits).
      requestBody:
        content:
          application/x-www-form-urlencoded:
            schema:
              $ref: '#/components/schemas/body'
        required: true
      responses:
        "200":
          description: ciphertext
        "400":
          description: invalid input
components:
  schemas:
    body:
      required:
      - ciphertext
      - prefix
      - postfix
      type: object
      properties:
        prefix:
          type: string
        ciphertext:
          type: string
        postfix:
          type: string
```

The API description explains, that the crypto API consumes three parameters (`prefix`, `ciphertext`, and `postfix`). Furthermore, the description section of the crypto API (path /cryptoapi) explains how the API works.

According to the description, the API expects the data to be encoded in hexadecimal. Given a POST request, it does the following (assume the POST request contains the following form data: prefix=`prf`, ciphertext=`cpht`, postfix=`pof`):

1. Decrypt the ciphertext `cpht` -> `clrt`
2. Concatenate `prf`, `clrt`, and `pof` -> `prf||clrt||pof`
3. Encrypt the concatenated string encrypt(`prf||clrt||pof`) = `new_cpht`
4. Return the newly created ciphertext `new_cpht`

Now that we know how the API works, we can start playing with the API and observing the behavior.

For example, the following queries might be used:

- `curl http://localhost:5000/cryptoapi -X POST -d "prefix=" -d "ciphertext=41414141414141414141414141414141" -d "postfix="`
- `curl http://localhost:5000/cryptoapi -X POST -d "prefix=41414141414141414141414141414141" -d "ciphertext=41414141414141414141414141414141" -d "postfix=41414141414141414141414141414141"`

Try to understand the behavior of the API and try to identify properties of the underlying encryption primitives.

Hint 3: The Crypto Primitives

Given that we have understood how to talk to the crypto API, we should now try to understand the cryptographic primitives used by the API.

From the API description we know that apparently a block cipher is used for encryption. Furthermore, the documentation says something about the block-size of 128-bits. Experimenting with input to the API we note:

- The byte-length of the parameter `ciphertext` needs to be a multiple of 128 bits, in order for the decryption to be applicable (note that by the hex-encoding two hex-digits correspond one byte).
- In total the sum of the lengths of parameters `prefix` and `postfix` have to sum up to a multiple of 128 bits (16 bytes).

Experiment with different inputs for the parameters `prefix` and `postfix`.

Is it possible to find out something about the underlying block cipher? Or perhaps about the encryption mode used by the API?

Hint 4: Encryption Mode

Playing around with the API and different inputs, we find out that the cleartext blocks of 128 bits get mapped to the same ciphertext. For example, using the command:

```
curl http://localhost:5000/cryptoapi -X POST -d
"prefix=41414141414141414141414141414141" -d
"ciphertext=41414141414141414141414141414141" -d
"postfix=41414141414141414141414141414141"
```

we get the response:

```
e44085fa2bda33d86aa340b4c16c05ad41414141414141414141414141414141
41e44085fa2bda33d86aa340b4c16c05ad
```

Given our knowledge about how the API works the response corresponds to the following:

```
encrypt("AAAAAAAAAAAAAAAA"||decrypt("AAAAAAAAAAAAAAAA")||"AAAA
AAAAAAAAAAAA")
```

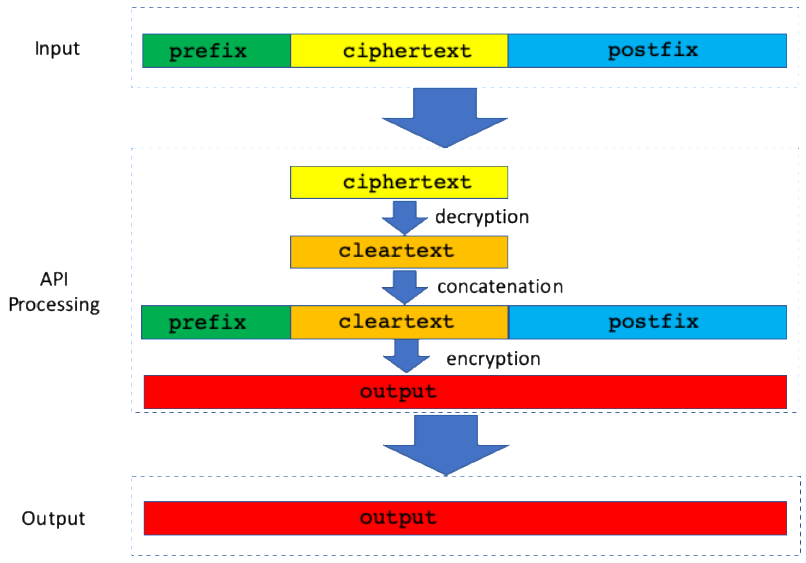
I.e., the content of the parameter `ciphertext` is first decrypted (with a key we do not know), then the content of the parameter `prefix` is concatenated with the cleartext from decrypting the content of parameter `ciphertext` and finally concatenated with the content of the `postfix` parameter, before the concatenated string is encrypted again (with the unknown key).
Looking at the middle part of the response, we recognize the original input of the parameter ciphertext. Looking at the prefix and the postfix of the response, we note that they are the same. We thus learn, that the same cleartext blocks get mapped to the same output blocks.

- What does this tell us about the encryption mode used by the API?
- How could we use the knowledge we have about the encryption mode to complete the challenge?
- How you could choose `prefix` and `postfix` in order to learn something from the response?
- Especially, consider the possibility to choose `prefix` and `postfix` of different lengths. How could this help you to learn something about the content of the ciphertext?

Hint 5: Using the Knowledge about the Encryption Mode

As you most probably have found out, the API seems to use Electronic CodeBook (ECB) encryption mode. As a consequence, the cleartext gets split into blocks of predefined size and each block gets encrypted independently before being re-concatenated as ciphertext-blocks. We do not really care about the blockcipher in use, except that we know the block-size (128 bits).

We now have a closer look at what the API does:

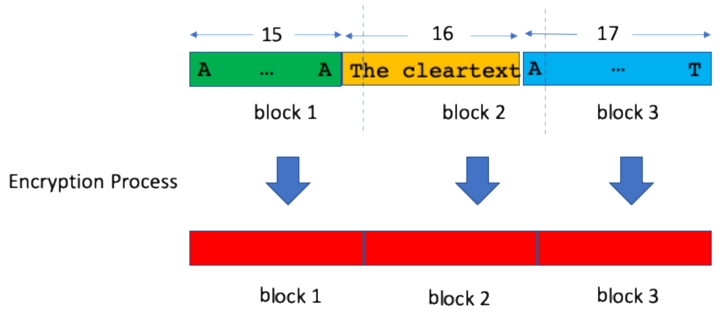


In the diagram we see the processes inside the API. Obviously, the orange box contains what we are looking for, namely the cleartext associated with the ciphertext of our challenge.
Note the different sizes of `prefix` and `postfix`.

- What happens if `pre`- and `postfix` are not multiples of the blocksize?
- Is it possible to learn something about the cleartext, if choose `pre`- and `postfix` in a clever way?
- Remember the property of ECB-mode to map the same input cleartext to the same output ciphertext. Could we perhaps choose `pre`- and `postfixes` in a clever way (possible varying the input), such that the output allows us to deduce something about the `cleartext`?
- Try to understand on a byte-level (letter) what happens in the concatenation and encryption process if the lengths of prefix and postfix are not multiples of the blocksize.

Hint 6: Solution (Theory)

Following the remark in the last hint, we now look at what happens in the API on a byte level.



In the figure we assume that green prefix has been chosen to consist of 15 ‘A’s (15 bytes), i.e., one byte less than the blocksize of the cipher. Furthermore, we assume for simplicity that the ciphertext has length also of 16 bytes. Thus, in the concatenation process the first letter (byte) of the cleartext is added to the 15 ‘A’s of the prefix in order to build the first block for encryption. Similarly, the first letter of block 3 is concatenated to the cleartext to build the second block for encryption. Finally, note the last block (part of the postfix). In case postfix contained 16 ‘A’s and a ‘T’ in its last position, block 1 and block 3 contain the same input for the encryption process and thus the corresponding ciphertexts of block 1 and block 3 conincide.
Thus, the idea is to vary the last letter of the postfix up to the point, where the first and the last block of the ciphertext coincide, if they do we have learned the first letter of the cleartext.
Having found the first letter of the cleartext, we shift everything to the left, i.e., `prefix` consists of 14 ‘A’s and the ‘T’, postfix contains 16 ‘A’s, a ‘T’, and varies its last letter up to the point, where again block 1 and block 3 of the ciphertext coincide. Repeating this step over and over, we can recover the whole cleartext…

Hint 7: Cheating ;-)

There is a rather simple way to cheat by opening running the container with a BASH-terminal. By running the following docker command, you will have a bash-shell on the corresponding container:

```
docker run -it dockidoc/challenge02:1.4 /bin/bash
```

You then find yourself in the directory `/app` of the container where the code of the API can be found in file `app.py`. In the file you find the password and thus should be able to decrypt the challenge, by getting further information about used encryption algorithm from the file.

Decrypting the challenge with the password found in the configuration file can be done either with Python and its Crypto-module or for example with a tool like OpenSSL, is left as an exercise to the reader…