Information Security Challenge 2: 20.4.2020

Technical Background

Introduction

In this challenge will look at what is sometimes called a “Crypto Oracle”. By such a “Crypto Oracle” we mean an API that takes some input applies some cryptographic function and responds in one or the other way. Typically, a “Crypto Oracle” uses a secret key that is only known to the “Crypto Oracle” and that should not be leaked. Even if the API uses cryptographically strong components and at first glance looks “secure” it sometimes allows a sophisticated attacker to abuse the API in an unexpected way and to break a security property of the system that provides the API.

A quite famous example of such an attack, the so called “Padding-Oracle-Attack”, is described in [1]. “Padding” here refers to the fact, that cryptographic functions typically take input of a predefined length. In case the input does not meet the length requirements, the input is “padded” in predefined way. For example, consider encryption with a blockcipher, where the blockcipher requires input of a certain length (blocklength). The input is then padded in a clever way, such that it meets the length requirements of the cipher, and such that the padding can be identified and removed after decryption. In the case of the “Padding-Oracle-Attack” an API given a ciphertext, responds with different error codes depending on errors occurring in the decryption process of the input. If the decrypted plaintext is padded incorrectly the API responds with a “Padding-Error” code, if the decrypted plaintext is padded correctly the API responded with a different error code, or simply accepted the input. By modifications of a given ciphertext and by interpreting the response of the API in a clever way, this simple difference in error codes enables an attacker to decrypt a given ciphertext without getting hold of the key or breaking the underlying cryptographic algorithm.

Although the setting of such a “Crypto Oracle” seems pretty artificial, we use a couple of them every day. Consider for example a web application that uses HTTPS, obviously the webserver makes sure that in- and output to the TLS-tunnel has the correct form and is encrypted with the right keys, changing the content of the IP-packet would probably lead to some kind of an error. Similarly, your hard-disk encryption mechanism (API) will encrypt your data in memory before storing it on disk and decrypt the data read from disk.

In our challenge today, we are given a ciphertext and a Web API (REST API). Our goal is to recover the plaintext contained in the ciphertext.

Prerequisites

The Web API runs on Docker, so you need a machine with Docker installed.

Besides Docker, you need tools to:

- compose and send request to the Web API

- a tool to manipulate strings (encode/decode them from/to bytes, work with HEX-representations)
- a tool to automate generation of byte-strings, send them to the Web API and work with the response of the Web API

In my case, I have used Linux for the development of the challenge and have tested it under Linux and OSX. The tools I have used are:

- `curl`: a command line tool to compose web requests
- Python 3 with the module `requests` to automate the sending and reception of requests to the Web API

In order to run the Web-API locally, execute the following command in a Shell or Terminal (it is assumed that the Docker daemon is running):

```
docker run -p 5000:5000 dockidoc/challenge02:1.4
```

The command will pull the image `dockidoc/challenge02:1.4` from Docker Hub (you need to be connected to the Internet of course), create and run the corresponding container locally. The option "`-p 5000:5000`" will implement port forwarding on TCP port 5000 between your host the running container. Thus, the Web-API should now be accessible on your host on http://localhost:5000.

The Challenge

The ciphertext to be decrypted with the use of the Web-API is provided is the following string:

```
21cb831dee353d8fbcbaf36cd82aab980257156247ec0310893dc4b0b2df0a
928ecf5382f65c40bd12e5ab12563981650ef327305967d4747a920ca0a867
e47e78a7d73e402b1cc99687bbd024a1f55438c5399d4a2a608ad27c197ee3
ab5b22d16c0f806ad53f74cdfc3d303079a2434eec374356cd936f0893f37d
587601cebfe63a6304700d8de9a57fdd0bf35362
```

The document and the challenge are supposed to be self-contained, in the sense that, given your machine fulfills the prerequisites, you should be able to complete the challenge on your own by working through this document and by the help of the document containing hints (distributed on Monday). Of course, you can and should complete the challenge as much as possible without this manual, however, since the reader is not expected to be an expert on topics needed to complete the challenge, this manual might be helpful to complete the challenge in a reasonable amount of time.

The challenge is supposed to be solved using the REST API provided by the Docker container only, i.e. the container itself has not been set up with a focus on security.

Background Cryptography

Encryption Algorithms

Probably the most well-known cryptographic operation is *encryption*. An encryption algorithm *enc* takes a key *k* and a so-called plaintext message *m* as input and outputs the so-called ciphertext *c*:

$$c = enc(k,m)$$

The corresponding *decryption algorithm* takes as input a key *k'* and ciphertext *c'* and outputs a plaintext message *m'*:

$$m' = dec(k',c')$$

In case of so-called *symmetric encryption algorithms,* we require the correctness property:

$$dec(k,enc(k,m)) = m$$

I.e., if we first encrypt a cleartext message *m* with a key *k* and decrypt the resulting ciphertext *c* again with the key *k*, we get back the original message *m*.
The use case of this type of algorithm is to hide the content (plaintext message) of a ciphertext from anybody who does not hold the corresponding key. For example, if Alice and Bob share a secret key *k*, using this kind of algorithm they can exchange messages secretly over an insecure network, such as the Internet, by encrypting the messages and by sending the ciphertext over the network. Obviously, this kind of use case requires that it is impossible for anybody who does not hold the key to learn anything about content of a ciphertext. For more details on cryptographic algorithms, see for example [2].

Modern algorithms used to implement this kind of cryptographic tool are *block ciphers*. Block ciphers take a block of data of predefined size and a key (also of predefined size) and return the corresponding ciphertext. Today's most prominent block cipher is AES [3]. AES works on data blocks of 128 bits and offers the possibility to use keys of different length (128, 196, and 256 bits).

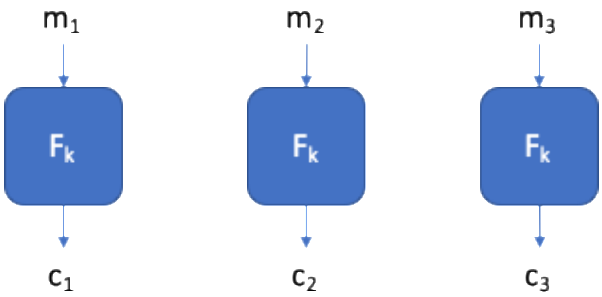
AES has been evaluated in a public selection process and has widely tested in 2001. Up to today AES is considered a secure blockcipher and there are no attacks known that are "substantially" better than brute-force (simply trying all possible keys).

Encryption Modes

Given a blockcipher (e.g., AES) we can encrypt blocks of data of a predefined size. An immediate follow-up question is: "How can we encrypt messages of arbitrary size?"

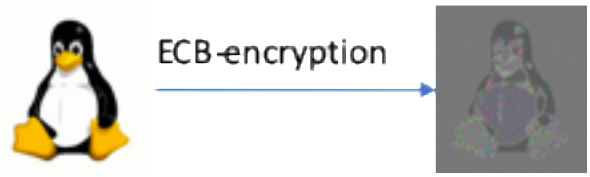
Electronic Codebook Mode (ECB)

The most obvious way to use a blockcipher for messages of arbitrary length is to somehow extend the cleartext message to a multiple of the block size and to apply the blockcipher to every sub-block of the message. I.e, if the message m consists of the sub-blocks m_1, m_2, and m_3 where each of the sub-blocks has the length of the blockcipher, we can apply the blockcipher (using a key k) to every sub-block independently to get the ciphertext that consists of the corresponding sub-blocks c_1, c_2, and c_3.



In the figure, encryption of a block with key k is denoted by the box F_k.

The main disadvantage of this encryption mode is that the same cleartext results in the same cyphertext. A typical example for this drawback is illustrated in the following figure:



In this example the picture of Tux has been encrypted using a strong blockcipher in ECB-mode. As a consequence of the property of ECB-mode, parts of the picture that contain the same information (color) are mapped to the same ciphertext. Thus, the color information of a single pixel might be perfectly hidden, however, the information of the picture in its entirety is still there…

Besides ECB-mode there exist a number of other chaining modes, such as Cipher-Back-Chaining (CBC) mode or Output FeedBack (OFB) mode, two other often used encryption modes. See [2] for more information on encryption modes.

Encoding, Representation of Information etc.

One of the "technical challenges" that come up when working on this kind of problem is the encoding and representation of data. The goal of this paragraph is to provide you the necessary tools in Python to work on the challenge. For any further information on the topic see for example [4]

Information is typically represented in a given language and as such consists of a sequence of letters that forms words. On a computer information is represented as sequences of bits and the process of mapping a letter (of a given alphabet) into a sequence of bits is called *encoding*.
In the following example, we encode the letter `A` into the number, the so-called *code point*, before we convert the code point into a representation in bits (or as a byte):

```
'A' --(encoding)--> 65 --(byte conversion)--> 01000001
```

In the example we have first represented the code point in decimal (`65`), then we have converted the decimal representation into its representation into its binary representation (`01000001`), to be even more precise we would have to further specify the bit ordering

(here so called “Big-endian”). Another often used representation is in hexadecimal, in the case of the letter ‘A’ at code point 65 this would correspond the hexadecimal representation to 0x41 or simply 41.

In Python textual data is handled with strings, i.e. objects of type str. To find a code point of a single character the function ord(.) returns the Unicode code point the given character. The other direction, i.e. finding a character representation for a given code point is done with the function chr(.).

Example:

```
>>> ord('A')
65
>>> chr(65)
'A'
```

Binary data is represented in Python as bytes or bytearrays. Bitwise-operations for example are done on objects of this data type. So most crypto-operations work on bytes- or bytearray-objets, that represent encoded strings.

Examples:

Encode a string into bytes:

```
>>> str.encode('Hello')
b'Hello'
```

Note the trailing ‘b’ denotes the type (bytes), the fact that the letters look like letters results from the fact, that ASCII code points are presented with the corresponding character (and the fact that ASCII code points map directly to UTF-8 code points).
The following example contains the character ‘ö’, that has no code point in ASCII:

```
>>> str.encode('Hallöchen')
b'Hall\xc3\xb6chen'
```

As a consequence, ‘ö’ is represented with its UTF-8 encoding.

Similarly, bytes-object can be decode into strings using the function bytes.decode().

Examples:

```
>>> bytes.decode(b'Hello')
'Hello'
```

and our previous example with ‘ö’:

```
>>> bytes.decode(b'Hall\xc3\xb6chen')
'Hallöchen'
```

Last but not least, byte-object as often represented with the hexadecimal representation of their bytes. This is best explained with examples:

```
>>> bytes.hex(b'A')
'41'
```

Here we have simple translated the byte-representation of the letter 'A' (remember that this corresponds to 65 in decimal) to its hexadecimal equivalent ('41').

Or for longer text string ('ABCD'):

```
>>> bytes.hex(str.encode('ABCD'))
'41424344'
```

Note that we first encoded the string, then have translated the resulting bytes to its hexadecimal representation.

For the other way of the translation:

```
>>> bytes.fromhex('41424344')
b'ABCD'
```

Note here that the resulting type is still `bytes`, if we want to get back a string, we would have to decode it accordingly:

```
>>> bytes.decode(bytes.fromhex('41424344'))
'ABCD'
```

Using a Web API

Obviously, you can use a browser to connect to a Web API, however, you are somewhat limited in terms of the request you can compose. There are a number of browser extensions, that enable you to compose requests to Web API.

A very common graphical tool to compose request to Web APIs is postman [5]. Here we will use the Unix command line tool `curl` [6] and the Python module `requests` [7] for automation.

The following examples show you how to use `curl` and `requests` as you might perhaps need the tools in order to complete the challenge. Of course, you are free to use any other tool of your choice.

Examples curl

- Simple GET-request to a webserver: `curl www.armasuisse.ch`
  (unfortunately, we a redirect here, so use the option `-L` to let curl follow the redirects and to get the html sources of www.armasuisse.ch)
- Simple GET-request to a given given port-number: `curl http://localhost:5000`
  (of course the local server has to listen on port 5000 to get a response)
- Simple POST-request: `curl http://localhost:5000 -X POST`
- Simple POST-Request with parameter `param1` equal 'abc' and parameter `param2` set to 'efg':
  `curl http://localhost:5000 -X POST -d "param1=abc" -d "param2=efg"`

Examples requests

`requests` is a Python package you need to install for example with pip (Python package manager), e.g., with the command `pip install requests`.

Everything we have done with `curl` in the last section can also be done with `requests` in a Python script, more importantly we have can use requests to automate request generation and the handling of the answers we get.

If we store the following lines of code in a file `test1.py`:

```
import requests

def simpleGET(urlstring):
    x = requests.get(urlstring)
    return x.text

print(simpleGET('http://localhost:5000/'))
```

If we execute the file with Python, the output is the response of the corresponding server.

Similarly, the following script would compose a POST request to a server setting the corresponding POST-parameters:

```
import requests

def simple_POST(url,p1,p2):
    post_param = {
        'param1': p1,
        'param2': p2
    }
    x = requests.post(url, post_param)
    return x.text

print(simple_POST('http://localhost:5000/<post_api>','parameter1','parameter2)
```

If the above code would be stored in a file and executed as a Python script, the url (here http://localhost:5000/<post_api>) would be called with a post request with the corresponding parameters sent as post-parameters in the request body.

References

[1]: "Security Flaws Induced by CBC Padding Applications to SSL, IPSEC, WTLS,…", Serge Vaudenay, EUROCRYPT 2002
[2]: "A Graduate Course in Applied Cryptography", Dan Boneh and Victor Shoup, 2017, https://crypto.stanford.edu/~dabo/cryptobook/BonehShoup_0_4.pdf
[3]: "Announcing the Advanced Encryption Standard (AES)", Federal Information Processing Standards Publication 197, NIST, 2001, https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf
[4]: https://docs.python.org/3/library/stdtypes.html
[5]: https://www.postman.com/tools
[6]: https://curl.haxx.se/docs/httpscripting.html
[7]: https://pypi.org/project/requests/