Assignment 1

E21 Network- and System Security
Aarhus University

October 4, 2021

CBC Padding Oracle and RSA Signatures

Group 8

Student no.:	*Name:*	*Initials:*
201608930	Andreas Harfeld Jakobsen	AH
201706031	Morten Lyng Rosenquist	MR
201707689	Valdemar Trøjgård Tang	VT

Source code

The source code can be found in the zip folder or can be downloaded from github using the following commands:

```
1 git clone https://github.com/mlRosenquist/au-syssec-e21-grp8-assignments
    .git
2 cd au-syssec-e21-grp8-assignments/assignment-1
```

CBC Padding Oracle

This exercise concerns exploiting vulnerabilities in the CBC mode of AES encryption. The source code of the server is inspected in order to figure out how to perform the attack. In short the attack must recover the secret in the string *f'You never figure out that "{secret}". :)'*, which can then be used to obtain a quote by modifying the text such that it equals *{secret} + ' plain CBC is not secure!'*.

Recovering plain text

In order to recover plain text the properties of how CBC is decrypted are exploited. How the decryption works can be seen on figure 0.1. Upon decryption it can be seen that the the output of the block cipher decryption function is XORed with the ciphertext from the previous block, or in the case of the first block, the IV is used instead. Thus we are able to alter the output of the plaintext by altering the ciphertext in the previous block.

The CBC mode uses the PKCS#7[5] padding scheme which is used to pad the message such that it can be divided into blocks of 16 bytes. The message is padded with bytes equal to the value of the number of bytes needed to pad the message to a multiple of 16 bytes. The corresponding plaintext is calculated as follows: $P = C_0 \oplus D(C_1)$. As we can control C_0, we can then try all possible values for a given byte which we XOR with the corresponding byte of C_0. Eventually, we will hit the correct byte which will give us a zero in the plaintext as XORs for identical values cancel out. The oracle will however still return an error upon decryption as the message is not correctly padded. However by XORing C_0 with the correct padding corresponding to the byte position that we are trying to recover, the oracle will no longer return an error message upon decryption, as the ciphertext decrypts to valid plain text. Thus the byte that is XORed onto the ciphertext when an error message is not returned, is the value of the plaintext in the given position. This can then be done until all the bytes in the block are recovered starting from the last byte.

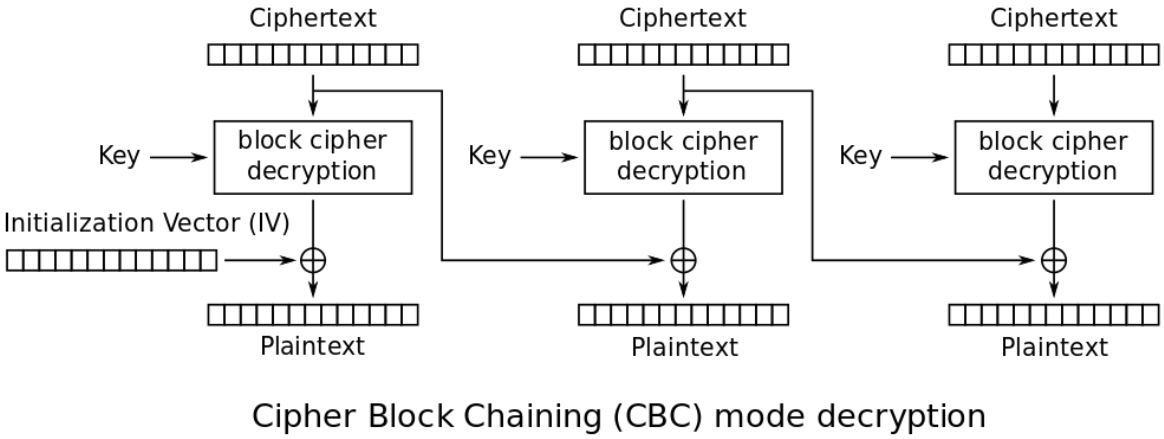

Figure 0.1: *CBC decryption operation. Taken from: https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#/media/File:CBC_decryption.svg*

Inserting desired text

Inserting desired text is done using the same principals for recovering plain text, however with a few minor changes. For each block the zeroing vector is obtained using the same method as the for when recovering plain text. As we need to change the text, this results in altering the ciphertext for the previous block to change the plaintext output in the current block. Thus we cannot use the zeroing vector obtained from recovering the plaintext, as we alter the ciphertext. Thus we are forced to start from the last block, and move towards the first block as we have to recover the zeroing vector for the altered ciphertext. To substitute the text, the block of previous ciphertext is XORed with found zeroing vector and then the desired plaintext. After finding the zeroing vector for a given block, the desired plaintext must be XORed together with the zeroing vector onto the ciphertext, in order to obtain the altered ciphertext, which is to be used when recovering the next block.

Implementation and results

The attack is implemented in python, and can be found in the file cbc_padding_attack.py. The file contains function definitions used in the attack, and the attack code in the bottom of the file. As the attack takes a while, intermediate results from recovering the secret at pasted into the code.

The function ***paddingAttack(token, knownText, desiredText)*** performs the plain text substitution attack, and the function ***recoverSecret()*** recovers the plain text from the token. The function ***attack_block(prevBlock, block*** recovers the zeroing vector for a given block. The secret is recovered from the plaintext and is found to be:

> *I should have used authenticated encryption because ...*

The program can be run and the quotes printed using the following commands in the folder: *assignment-1/cbc-padding-adversary*.

```
python3 -m venv venv
. ./venv/bin/activate
pip install -r requirements.txt
python3 ./cbc_padding_attack.py "arg"
cat ./resultFile.txt
```

For running the different parts of the attack, a command line argument must be specified. The following are available and produce the corresponding documents:

- ”recoverPlaintext” - produces *plaintext.txt*
- ”subAttack” - produces *validToken.txt* and *quote.txt*
- ”getQuotes” - produces *quotes.txt*

The result file(s) may end up in different places depending on the specific environment, so you may have to look a bit for them. The first two parts involving the actual attack, take around 15-20 mins to complete each.

References

The following implementations and articles have been used as inspiration for the implementation, [2],[6],[7].

RSA Signatures

The RSA Signatures task is seperated in two. The first is about exploiting the weakness of textbook RSA. The weakness will be utilized to forge a signature of a desired message that the server did not sign. The second task is about implementing RSA-PSS in the server to avoid the possibility of such attacks.

Exploiting Textbook RSA Signatures

The first step was to understand the API we were to exploit. The source code was investigated, and Postman was utilized to test the various endpoint of a locally hosted version of the API. The next step was then to implement methods to hit these endpoints.

Having the necessary functionality to hit the endpoint, we were ready to perform the attack. As we had no prior knowledge of the exploit we found inspiration in a stackexchange

post[8]. From the source code of the API, it was derived that the desired message is:

$$m = \textit{"You got a 12 because you are an excellent student! :)"}$$

And what we want to achieve is obtaining the signature, s, of that message where:

$$s = m^d \mod n$$

Firstly we take the hex and numeric value of m, and see that the numeric value of m is a multiple of 5. This is used in the next step. Then we utilize that the API foolishly allows us to sign a random message. Because m is a multiple of 5 we pick that as our random message. We call this message m_1 and its signature s_1:

$$s_1 = m_1^d \mod n$$

Another message, m_2 , is then constructed:

$$m_2 = m * m_1^{-1} \mod n$$

Again we retrieve the signature from the API:

$$s_2 = m_2^d \mod n$$

The signature for the desired message, m, can now be retrieved as:

$$s = s_1 * s_2 \mod n$$

The signature is calculated. The quote endpoint is called with m and s in the grade cookie. And finally we have forged a signature of a desired message that the server never signed.

To execute the attack. Execute the following commands from the assignment-1/rsa-signatures-adversary folder.

```
python3 -m venv venv
. ./venv/bin/activate
pip install -r requirements.txt
python3 ./rsa-attack.py
cat ./quotes.txt
```

Implementing RSA-PSS

The next task was to improve the API to avoid adversaries fabricating signatures as seen in the attack described above. This is done with the PSS padding scheme. For the implementation, the standard defined in [4] is followed. The implementation is furthermore

inspired by a python implementation of the standard [1] . The primitives and helper functions used in the algorithm is taken directly from the inspiration[1]. The implementation can be found in *main.py* containing the flask endpoints.

Choice of Random Number generator

As a part of the RSA-PSS algorithm it is necessary to generate salt, for which a random number generator is used. The original random number generator in python is aimed at usages for simulations and modelling and is not secure for number-generation used in cryptography. However, in python 3.6 the secret module was added allowing for secure number-generation aimed at cryptographic applications [3].

Testing RSA PSS

To ensure that our implementation works, an arbitrary message is encrypted and a corresponding signature is generated. Hereafter the message and signature are validated to ensure the integrity of the message. The message is then altered and validated again with the signature generated from the previous message, where the validation returns false indicating that the message has been altered. This can be seen by executing the file *rsa-signature-task/main.py* using the first part of the commands. The file also runs the webserver given in the assignment, but with RSA-PSS implemented instead of plain textboox RSA. The RSA attack from the previous part can then be attempted again using the last part of the following commands. The attack should not be able to obtain any quotes but instead get a *"Are you trying to cheat"* message.

```
#Go to assignment-1 folder
python3 -m venv venv
. ./venv/bin/activate
pip install -r rsa-signatures-adversary/requirements.txt
pip install -r rsa-signatures-task/requirements.txt
export FLASK_APP=rsa-signatures-task/main.py
flask run

#In new terminal in assignment-1 folder
. ./venv/bin/activate
python3 rsa-signatures-adversary/rsa-attack.py "local"
cat ./quotes.txt
```

References

[1] bdauvergne. *python-pkcs1*. unknown. URL: https://github.com/bdauvergne/python-pkcs1.

[2] Sam Bowne. *Proj 14: Padding Oracle Attack*. 2017. URL: https://samsclass.info/141/proj/p14pad.htm?fbclid=IwAR3XRWznua8JSVi9-JyZ2ytdd74RO1DJU1e-KybxeMoZpDUM9KzFjHVokWs.

[3] Python Documentation. *secrets — Generate secure random numbers for managing secrets*. 2021. URL: https://docs.python.org/3/library/secrets.html.

[4] IETF. *RSA Cryptograpy Specification V2.2*. 2016. URL: https://datatracker.ietf.org/doc/html/rfc8017#section-8.1.

[5] B. Kaliski. *PKCS #7: Cryptographic Message Syntax*. 1998. URL: https://datatracker.ietf.org/doc/html/rfc2315.

[6] Panos Sakkos. *RECOVERING PLAINTEXTS WITH PADDING ORACLE ATTACKS*. 2021. URL: https://le4ker.me/tech/2017/05/29/padding-oracle-attack.html?fbclid=IwAR2Sa5xrV_0BgL4NgGQ1fr3JoTszNSGlBVlTfhhP8FyrjA16P6Mh1kOe7oM.

[7] ScullSecurity. *Going the other way with padding oracles: Encrypting arbitrary data!* 2016. URL: https://blog.skullsecurity.org/2016/going-the-other-way-with-padding-oracles-encrypting-arbitrary-data?fbclid=IwAR2qTY28aiTWC950or9mc2XbQ4FbFAcVvnkZXm9no.

[8] Stackexchange. *Chosen-Message-Attack RSA-Signature*. 2016. URL: https://crypto.stackexchange.com/questions/35644/chosen-message-attack-rsa-signature?fbclid=IwAR3SVNhhOAbP7rsnJW59twNkgB7sYt4zfPelsRsOJVhV7X60ysXk92OvtY.