Assignment 1: Padding Oracle Attack

CS348 Introduction to Information Security, KAIST
(2024 Spring)

Due: 11:59 PM (KST), March 13, 2024

INSTRUCTIONS TO STUDENTS:

- Any collaboration or assistance of any kind is strictly prohibited; all work must be your own.
- Your submission *will surely* be compared with the submissions for your peers for plagiarism detection (e.g., MOSS). Any academic dishonesty will be directly reported to the university.

1 Padding Oracle Attack (70 points)

For this assignment, you will implement a padding attack mentioned in class. This attack was first discussed in the paper "Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS..." by S. Vaudenay. You should read the paper (`https://www.iacr.org/archive/eurocrypt2002/23320530/cbc02_e02d.pdf`) and implement the described attack.

You will have the opportunity to create your own padding attack mentioned in class to extract a message from a ciphertext. You can find a set of valid ciphertexts on the KLMS assignment #1 handout section. The ciphertexts will be given as a 128-bit hexadecimal number (starting with `0x`...) computed as follows:

$$C = C_0 \| C_1 = \texttt{CBC}_K^{\texttt{Encrypt}}(\texttt{PAD}(M)), \tag{1}$$

where the word M is shorter than or equal to 8 ASCII characters (i.e., 64 bits). The block size of the cipher and the length of the secret key K are 64 bits. The first 64-bit block of C is the initialization vector (`IV`), which we denote C_0, and the last 64-bit block is the first cipher block, C_1. Note that we assume the randomized `CBC` encryption; that is, the `IV` (C_0) is randomly selected for every message.

You are given a *padding oracle*, which can be accessed by function calls; see Section 1.2 for the details. You can access the padding oracle as many times as you want. You will develop your padding attack program using Python in this assignment. The padding oracle will accept both C_0 and C_1 (each 64 bits long), and return a 1 or a 0, indicating correct or incorrect padding respectively. The padding oracle works as follows:

$$\{0,1\} = \texttt{Check\_PAD}(\texttt{CBC}_K^{\texttt{Decrypt}}(C)), \tag{2}$$

where you provide the C.

1.1 Where to obtain the ciphertext?

KLMS assignment #1. p1_ciphertexts/ciphertext_#.txt

1.2 How to access the oracle?

You will be given the following file:

- oracle.pyc: Binary code containing padding oracle and decryption oracle.

Python 3.8 and *pycryptodome* package are required for this assignment. The *pycryptodome* package can be installed using pip like the following.

```
pip3 install pycryptodome
```

If you have a different version of Python installed, please refer to these instructions on how to manage different versions of Python using Anaconda. (`https://conda.io/projects/conda/en/latest/user-guide/tasks/manage-python.html#installing-a-different-version-of-python`)

Here is an example code on how to access the oracles in your solution.

```
from oracle import Oracle
# Init Oracle class
oracle = Oracle()
# query padding oracle
ret_pad = oracle.pad_oracle('0x1234567890abcdef', '0x1234567890abcdef');
```

1.3 Submission and grading

Submission. Your code has name `p1_#.py` where # denotes your student id. Your code should accept two command-line arguments C_0 and C_1 in the following format:

- python p1_#.py C_0 C_1

The ciphertext blocks (C_0 and C_1) have hex format starting with `0x`. The output must be the plaintext in ASCII format.

Grading. Your code should successfully obtain the plaintext from a ciphertext. You will get 40 pts if your code works for the ciphertexts that are available on handouts. If it works with all the other ciphertexts (not given to students), an additional 30 pts will be given.

Test your implementation before submission. To test your implementation, you can use ciphertext/plaintext pairs in handouts

- p1_ciphertexts/ciphertext_#.txt: this contains the ciphertext you will use as an input
- p1_plaintexts/ciphertext_#.txt: this contains the plaintext that is corresponding to the ciphertext_#.txt

2 Turning Decryption Oracle into Encryption Oracle (30 points)

What you create in Problem 1 is a single-block decryption oracle that takes a two-block ciphertext (one of which is an IV) and returns a single-block plaintext. Now, in Problem 2, you are asked to create an *encryption oracle* using the provided single-block decryption oracle (i.e., Oracle.dec_oracle(IV, ciphertext))[1]. Your code needs to take an arbitrary message M, pad it if needed, and encrypt it without knowing the secret key. Note that the message M can be longer than one plaintext block.

We provide the single-block decryption oracle so that you can solve Problem 2 even when you fail to solve Problem 1. The single-block decryption oracle is accessed via calling the oracle function. Note that the secret key used in Problem 2 is different from the one used in Problem 1.[2]

2.1 How to access the decryption oracle?

Here is an example code on how to access the oracles in your solution.

```
from oracle import Oracle
# Init Oracle class
oracle = Oracle()
```

[1]You should *not* use your own decryption oracle from Problem 1 even though you have solved Problem 1 because the provided decryption oracle uses a different key for Problem 2

[2]Since the two keys are different, you cannot use the decryption oracle provided in Problem 2 to solve Problem 1.

```
# Set IV yourself. The IV below is an example.
IV = "0x1234567890abcdef"
# This is the example ciphertext.
Ciphertext = "0x1234567890abcdef"
hex_plaintext = oracle.dec_oracle(IV, Ciphertext);
```

2.2 Submission and grading

Submission: Your code has name `p2_#.py`, where `#` denotes the student id. Your code should accept any arbitrary message M in the command-line argument:

- `python p2_#.py "This is the message that needs to be encrypted."`

The output must be the one or more 64-bit hex formats; e.g., `0x12...ef 0x98..ab 0xb7..2d`.

Test your implementation before submission. To test your implementation you can use the decryption oracle again. You can generate as many test cases as you wish to test your implementation.

Upload to Gradescope Tar all your codes (`A1_#.tar`) and upload it to Gradescope assignment #1 via KLMS. Only *.tar* file format is allowed. Even if you use Windows, you can easily make .tar file with your terminal(cmd)[3]. Please find the links in the footnote.
Grading: Your code should encrypt all the multiple test plaintext messages of the instructor's choice and generate valid ciphertext messages.
Disclaimer: Even if you manage to somehow extract the secret key from the binary, solutions that use the secret key directly will not be awarded any score for both problems 1 and 2.

A Late Policy

80% credit will be awarded for 1-day late submission, 50% credit for 2-days late submission. NO credits will be awarded for submissions made later than 2 days after the given deadline.

B Academic Dishonesty

We will run Moss[4] (Measure Of Software Similarity) for detecting any partial or full plagiarism of other students' submissions (or any code you find on online services), in addition to our manual checks. We will also compare your submission with the corpus of student submission in the past semesters. Students with submissions with high similarity scores will be asked to explain their code directly to the TAs and the instructor. Our school takes any form of academic dishonesty seriously[5] and we will strictly follow the policies of the school and the university, should we have any unfortunate event in this assignment.

C Environment Setup

In case you are not familiar with setting up a local computing environment for Python programming, we provide some links to the setup guides for Python[6]. Please find the links in the footnote.

[3] `https://www.addictivetips.com/windows-tips/use-tar-on-windows-10/`
[4] `https://theory.stanford.edu/~aiken/moss/`
[5] School of Computing Student Honor Code: `https://forms.gle/pqTWkfwpgnwqcU2W7` (you do not need to submit this form though)
[6] `https://www.tutorialsteacher.com/python/install-python`