600.445/600.646 Practical Cryptographic Systems February 10, 2025

Assignment 2

Instructor: Matthew Green Due: 11:59 pm, 23rd February 2025

The assignment must be completed individually. You are permitted to use the Internet and any printed references, though your answers and code must be your own!

Supplementary material. As part of this assignment, you are provided with two Python source files: • `problem.py` implements the challenges and the test harness. • `solution.py` provides a starter template for your solution. Please refer to these files for more details.

Submission instructions. This assignment includes both written and programming components. Submit your answers to the written portion as a single PDF file under "Assignment 2 - Written" on Gradescope. For the programming part, submit your completed `solution.py` file under "Assignment 2 - Programming" on Gradescope.

1 Programming

Problem 1: Padding Oracle Attack

A web server authenticates users using encrypted tokens. Each token is generated by encrypting user identification data with AES in CBC mode using a secret key stored on the server. Since AES-CBC operates on fixed-size blocks (16 bytes each), the server applies the PKCS7 padding scheme to handle messages that aren't exact multiples of the block size. Users cannot directly decrypt their tokens, but they must include them in requests to the server for authentication. Upon receiving a token, the server decrypts it to verify the user's identity. However, if the decrypted token has *incorrect padding*, the server responds with an error.

You have obtained an encrypted token. In this problem, you will apply a technique described in Vaudenay's 2002 paper[1] to exploit the server's differing responses based on whether the padding is valid or invalid. Using this, you will recover the plaintext of the encrypted token without knowing the secret key.

`problem.py` implements a simplified version of the scenario described above. Instead of handling web tokens, you will be given the encryption of an arbitrary message. The "server" is represented by a Python function that decrypts a given ciphertext and checks its padding. Refer to `problem.py` for more details. Your task is to complete the function `solve_padding_oracle` in `solution.py`.

Your solution must be able to handle messages of any (non-zero) length. For messages shorter than 500 bytes, your solution must recover the plaintext within a minute—though in this scenario, a successful attack should typically complete in just a few seconds.

[30 points]

[1] You do *not* need to implement the exact attack described in the paper—simpler variants are possible. However, the paper may help you understand how the attack works.

Problem 2: Attacking Stateful CBC

A web server authenticates clients using session identifier cookies, which are stored on client devices. When making a request, the client includes this cookie in the HTTP headers along with other metadata, such as the resource path being accessed. To ensure the privacy of the headers, the client encrypts the entire HTTP header using AES in CBC mode under a secret key stored on the device. However, instead of generating a new random IV for each encryption—which would need to be transmitted with each ciphertext—the encryption scheme makes a seemingly minor optimization: it uses the last ciphertext block from the previous encryption as the IV for the next request. This *stateful* variant of CBC eliminates the need to send an IV with each ciphertext and intuitively behaves as if encrypting a single long message across multiple requests.

Alice's device, which stores such encrypted session cookies, has been partially compromised by you. You cannot extract they key directly, but you have the ability to: 1. Make HTTP requests from Alice's device and observe the resulting ciphertexts. 2. Modify parts of the HTTP request header, such as the resource path, allowing you to manipulate part of the encrypted content. In this problem, you will exploit the fact that the stateful variant of the CBC mode is insecure to recover Alice's session identifier cookie.

`problem.py` implements a simplified version of the scenario described above. It provides a function that emulates the compromised client device by taking the resource path as input and returning an encryption of ⟨path⟩;cookie=⟨cookie⟩.

Your solutions must be able to handle cookies of any (non-zero) length. For cookies shorter than 500 bytes, your solution must compute the output within a minute—though the best possible attack should typically complete in just a few seconds.

1. Implement the `find_cookie_length` function in `solution.py`. This function determines the length of the cookie encrypted by the device.

   [15 points]

2. Implement the `find_cookie` function in `solution.py`. This function recovers the complete cookie encrypted by the device.

   Hint: Can you set the path in a way that allows recovering the cookie byte-by-byte?

   [35 points]

2 Written

Problem 3

Briefly describe your approach to solving Problem 2. Analyze the runtime of your solution to `find_cookie` in terms of number of calls to the function and the length of the cookie.

[5 points]

Problem 4

The CBC-MAC[2] of a message m, under a secret key k, is computed as the *last block* of the ciphertext when m is encrypted in CBC mode using k and an all-zero IV.

[2] See Section 6.1 of the Boneh/Shoup Book to recall the definition of Message Authentication Code (MAC).

The CBC-MAC can be shown to be secure for fixed-length messages i.e., fix a message length L beforehand and only verify the MACs for messages of length L (messages of other lengths automatically fail verification). However, it is insecure when used for variable length messages.

Demonstrate an attack on CBC-MAC when used for variable length messages. Your attack can obtain a MAC on two messages of your choice. It must then forge a MAC on a new message that is not equal to your queries.

[15 points]