Computer Assignment 1 Deadline: Jan 29, 2023, 11:55 PM
Winter 2023 (Upload it to Gradescope.)

Project Description

This assignment includes two parts. We will start by finding the key that was used to encrypt a ciphertext, then move on to leverage the md5 hash functions vulnerability to construct a length extension attack. There are two Python files you need to download:
vegenereCiphers.py, lengthExtensionAttack.py

Part 1: Vigenere ciphers

For this problem, write a Python program so that it can find the key used to encrypt a specific ciphertext.

Vigenere ciphers is a method of encrypting alphabetic text by using a series of interwoven Caesar ciphers, based on the letters of a key. More details can be found on Wikipedia.

You can use this link https://cryptii.com/pipes/vigenere-cipher to generate a ciphertext to test your solution.

Hints:
To figure out the key, the first thing we need to do is determine the length of the key. You can try the following 2 methods:

1. **Kasiski Examination**
2. **Index of coincidence** (We recommend this method.)

Also, the **Letter Frequency** is useful for determining the letter of the key.

How Gradescope autograder tests your submission:
The autograder will import your submission file. Then, it will create an instance of your solution. For each test case, the autograder will call getKey() method with a parameter ciphertext which is encrypted by the key. The test result is the comparison result between your getKey() return value and the actual key.

You can try as many times as you want before the deadline. We will take the latest submission as your final solution.

Part 2: Length Extension

To avoid a Man-In-The-Middle attack (MITM), the server needs to verify the integrity of the request received. A common solution is using a Message authentication code (MAC). To calculate MAC, a server, or client will concatenate a secret (like a password) and a message, then apply a hash function. For example MAC = MD5(password || message).

Some hash algorithms such as MD5, SHA1, and SHA2 are based on Merkle-Damgard construction, but those algorithms have a vulnerability which is as long as we know the value of `hash(secret || message)` and the length of the secret. Then we can calculate the hash value of `hash(secret || message || padding || m')` where m' is arbitrary data and padding is bytes used to fill a block (we will talk in more detail about it later).

The reason is that the Merkle-Damgard construction uses a compression function F and an internal state S to do the hash calculation. S initialized to a fixed constant. Messages are split into fixed-size blocks. Then, the algorithm will apply compression function F with the initial S on the first block, and use the result to update the internal state S. After that, it applies compression function F with updated S on the second block, use the result to update S again, and so on.

2A- Quick Demo

We will take MD5 as an example to demonstrate the basic attack idea. You can download the pymd5 module here.

Consider the string "Is MD5 security enough?". we can compute its MD5 hash by running:

```
from pymd5 import md5, padding
m = "Is MD5 security enough?"
h = md5()
h.update(m)
print(h.hexdigest())
```

The output should be 41a166c915f60d99ee4f498557905e09.

MD5 processes messages in 512-bit blocks, so internally, the hash function pads m to a multiple of that length. The padding consists of bit 1, followed by as many 0 bits as necessary, followed by a 64-bit count of the number of bits in the unpadded message. (If the 1 and the count won't fit in the current block, an additional block is added.) You can use the function `padding(count)` in the pymd5 module to compute the padding that will be added to a count-bit message.

Even if we didn't know m, we could compute the hash of longer messages of the general form `m + padding(len(m)*8) + suffix` by setting the initial internal state of our MD5 function to `MD5(m)`, instead of the default initialization value, and setting the function's message length counter to the size of m plus the padding (a multiple of the block size). To find the padded message length, guess the length of m and run `bits = (length_of_m + len(padding(length_of_m * 8))) * 8`.

The pymd5 module lets you specify these parameters as additional arguments to the md5 object:

```
h = md5(state=bytes.fromhex("41a166c915f60d99ee4f498557905e09"), count= bits)
```

Now you can use length extension to find the hash of a longer string that appends the suffix "It is NOT". Simply run:

```
x = "It is NOT"
  h.update(x)
  print(h.hexdigest())
```

to execute the compression function over x and output the resulting hash. Verify that it equals the MD5 hash of `m.encode("utf-8") + padding(len(m)*8) + x.encode("utf-8")`. In Python 3, we need to convert m and x from strings to bytes so that we can add these to the padding, which is a byte type. Notice that, due to the length-extension property of MD5, we didn't need to know the value of m to compute the hash of the longer string—all we needed to know was m's length and its MD5 hash.

2B- Real Attack

Some UCLA students build a blog website. They allow users to perform actions by URL. For security purposes, they choose to use the MAC as a token to validate the authorization of requests. The form of a URL is as follows:

http://blog.ucla.edu/photo/api?token=73268f28ebb97395bd541685c3a94d31&user=Bob&command1=closePhoto&command2=logOut

where the token is `MD5(user's 8-character password || user=... [the rest of the decoded URL starting from user= and ending with the last command])`.
For example, for the above URL, the user's password is "PaSSwORd", then the token is calculated by `MD5(PaSSwORduser=Bob&command1=closePhoto&command2=logOut)`.

Using the techniques that you learned in the previous section and without guessing the password, apply the length extension to create a URL ending with **&command3=releaseAccess** that would be treated as valid by the server API.

Hint: You might want to use the quote() function from Python's urllib.parse module to encode non-ASCII characters in the URL.

How Gradescope autograder tests your submission:
The autograder will import your submission file. Then, it will create an instance of your solution.

For each test case, the autograder will call getNewURL() method with a parameter URL which is the original request URL. The test result is the comparison result between your getNewURL() return value and the original request URL.

You can try as many times as you want before the deadline. We will take the latest submission as your final solution.

You can watch this video to learn more about Part 2 (link).

Resources:

1. Kasiski Examination: https://en.wikipedia.org/wiki/Kasiski_examination
2. Index of Coincidence: https://www.dcode.fr/index-coincidence
3. Letter Frequency: https://en.wikipedia.org/wiki/Letter_frequency
4. Length extension attack: https://en.wikipedia.org/wiki/Length_extension_attack
5. Man-In-The-Middle attack: https://en.wikipedia.org/wiki/Man-in-the-middle_attack
6. Merkle-Damgard construction: https://en.wikipedia.org/wiki/Merkle%E2%80%93Damg%C3%A5rd_construction

Notes:

1. The plaintext used for the test is a meaningful text in English with at least 10k letters, and the length of the key is 3 to 6 (including 3, 6).
2. Plaintext and key contain only lowercase letters[a-z].
3. A text written in the English language has an index of coincidence of 0.066
4. All test URLs will have the same form as the sample above, but the values of token, hostname, user, command1, and command2 will be changed. Also, each URL may use a different password, but the length will always be 8.

What to Submit:

1. vegenereCiphers.py
2. lengthExtensionAttack.py

Also, please add comments at the end of the python files to explain your solution. Submissions without enough explanation will only get partial credit