CSE 127

Discussion 6

Reminder

https://kumarde.com/cse127-wi26/pas/pa5.pdf

- Due date - **3/14 Sat 11:59pm**
- Groups of up to 2
- Three parts
  - Vigenère Cipher
  - MD5 Length Extension
  - MD5 collisions

Part 1: Vigenère Ciphers

The combination of several Caesar Ciphers

```
 Plaintext: ATTACKATDAWN
       Key: BLAISEBLAISE
Ciphertext: BETIUOBEDIOR
```

Key ‘A’ means no shift
Key ‘B’ means shift by 1
Key ‘C’ means shift by 2
…

Idea:

- We know Caesar Cipher is vulnerable to frequency analysis
- However we can’t do frequency analysis on ciphertext of Vigenere Cipher, because each key is different
- Can we reduce Vigenere Cipher into several Caesar Ciphers?
- If we line up the ciphertext in n columns, where n is the length of the keyword. We can do frequency analysis on the n groups.

Kasiski Examination

To know the length of the key we do Kasiski Examination:

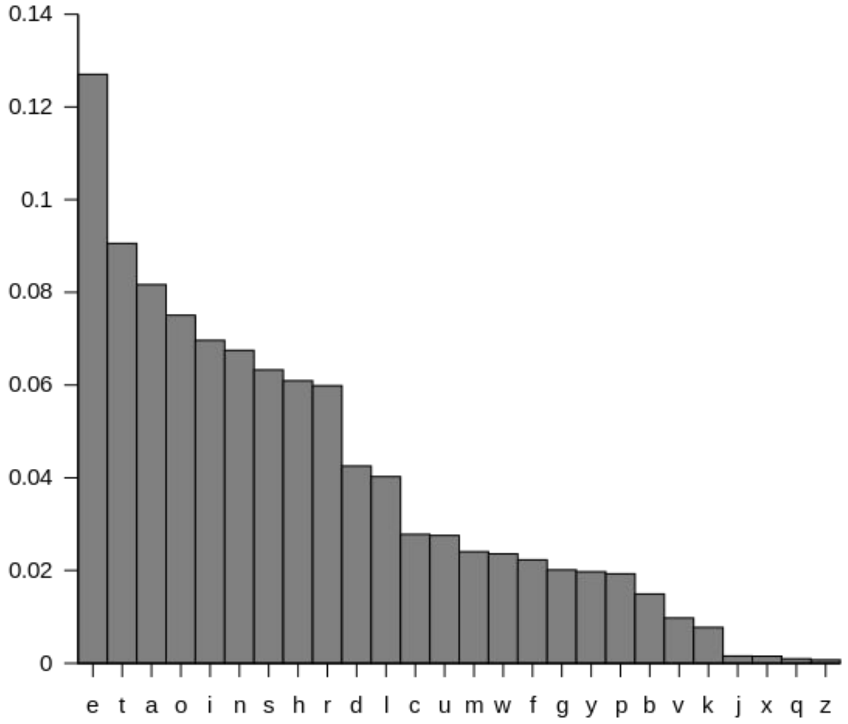
1. Look for strings of characters that are repeated in the ciphertext.
2. Find the distances between those repeated characters.
3. Since those distances are multiples of the key length, we find the greatest common divisor of those distances, which will give us the length of key.
4. The longer the string the better, the more distances you find the better.

```
Plaintext : crypto is short for cryptography. (dist is 20)
Key       : abcdeabcdeabcdeabcdeabcdeabcdeabc (len  is 5)
Ciphertext: csasxo kv siqux gqu csasxohtdthz.
```

https://en.wikipedia.org/wiki/Kasiski_examination

Frequency analysis

After knowing the length, we need to do frequency analysis on each group of ciphertext that was applied the same key.



- The english language have more frequently used letter and less frequently used letters

- In each group in your ciphertext, count the frequency of each letter.
- Since they are shifted by the same amount, this pattern of frequency would emerge.

- Map the shifted letters to the plaintext letters.
- Then you know how much each letter is shifted. (aka. the key)

Part 2: MD5 Length Extension

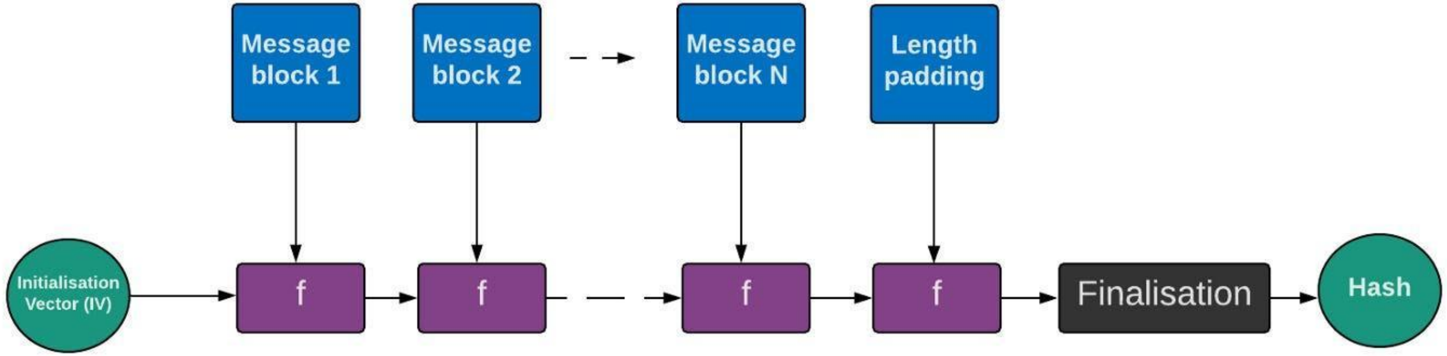
Given H(secret||m)
Create correct hash with x appended, without knowing secret

http://bank.cse127.ucsd.edu/pa5/api?token=7adbfb23d6058b8dcb45402f02198028&user=kumarde&command1=ListSquirrels&command2=NoOp

where token is MD5(*user's 8-character password* || user=...)

Part 2: MD5 Length Extension

- For this part it is pymd5.py which has some functions to get at individual steps of md5 hashing
- Key idea: **padding** is 1 followed by necessary number of zeros at end of message, but you need to be able to have a 1 followed by zeros as part of the message as well
- *Part 2: Experimenting* in the assignment walks you through this and should make the attack understandable



Part 2: MD5 Length Extension

Example: want to generate correct hash with “Good advice” appended to the end, without knowing preimage of that hash.

```
m = "Use HMAC, not hashes"
print(md5(m).hexdigest())
```

```
3ecc68efa1871751ea9b0b1a5b25004d
```

```
bits = (len(m) + len(padding(len(m) * 8))) * 8
```

```
h = md5(state=bytes.fromhex("3ecc68efa1871751ea9b0b1a5b25004d"), count=bits)
```

Restore the state

How many bits we processed so far

```
x = "Good advice"
h.update(x)
print(h.hexdigest())
```

Continue calculating the hash

```
e1ca9db8eae1b8cbfacc63de828af6d0
```

Part 2: MD5 Length Extension

Example: want to generate correct hash with “Good advice” appended to the end, without knowing preimage of that hash.

```
x = "Good advice"
h.update(x)
print(h.hexdigest())
```

```
e1ca9db8eae1b8cbfacc63de828af6d0
```

Same hash

```
result = m.encode("utf-8") + padding(len(m)*8) + x.encode("utf-8")
h_new = md5()
h_new.update(result)
print(h_new.hexdigest())
```

```
e1ca9db8eae1b8cbfacc63de828af6d0
```

So H(m||padding||x)

- Why can’t we just do m+x? What is the padding for?

H(m||padding)

|| means string concatenation

```
m = "Use HMAC, not hashes"
print(md5(m).hexdigest())
```

```
3ecc68efa1871751ea9b0b1a5b25004d
```

Part 2: MD5 Length Extension

http://bank.cse127.ucsd.edu/pa5/api?token=7adbfb23d6058b8dcb45402f02198028
&user=kumarde&command1=ListSquirrels&command2=NoOp

where token is MD5(*user's 8-character password* || user=...)

Without knowing the password, we want to append *&command3=UnlockAllSafes* to the end of URL, and get the correct hash.

In other words, want to get:
H(pwd ||user=kumarde&command1=ListSquirrels&command2=NoOp…
…*<some padding>&command3=UnlockAllSafes*)

Restore the state using the hash provided,
update(x)
Then you get H(pwd || user=... || padding || x)

Part 2: MD5 Length Extension

HINTS

- python3 len_ext_attack.py "http://.........NoOp"
- Only use *urllib.parse.quote()* for the padding
- Use the Gradescope autograder for testing if your attack works.
- https://deeprnd.medium.com/length-extension-attack-bff5b1ad2f70

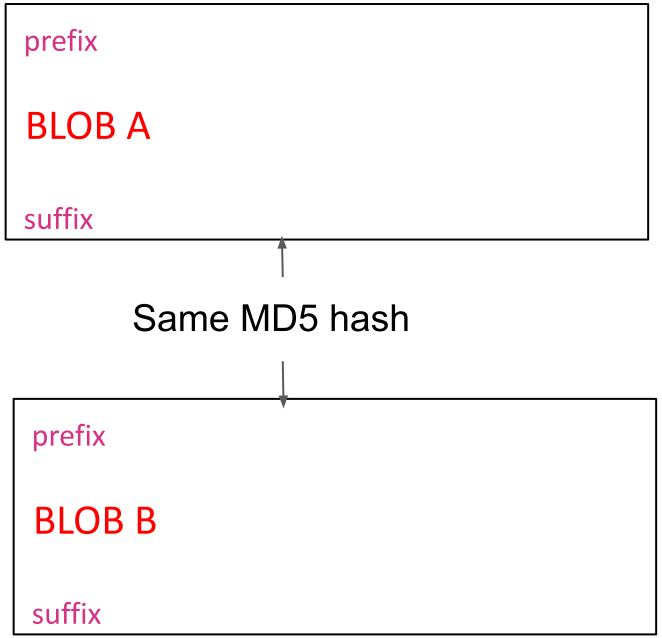
Part 3: MD5 collisions

Two programs with different behavior that hash to the same thing

- We provide *fastcoll* which generates MD5 collisions
- You might need to build this code if its not available on your OS so there is also a makefile to help
- Key idea: once you have a collision, adding identical suffixes to them will also collide because of the length extension property of MD5
  That is, if H(A)=H(B) and len(A)=len(B), then for any S, H(A||S)=H(B||S)
- Explanation of prefix suffix: in this script, the SHA256 of BLOB is stored in `digest`. You can modify the code to make use of this variable!

```
prefix

#!/bin/bash

cat << "EOF" | openssl dgst -sha256 > DIGEST

BLOB

suffix

<BLANK LINE>
EOF

digest=$(cat DIGEST | sed 's/(stdin)= //' )

echo "The sha256 digest is $digest"
```

Part 3: MD5 collisions



- How can you change suffix so that the program behaves like in the write up (print different stuff)?
- Remember the two files have different SHA256 hashes!
- You can reuse and modify the prefix and suffix we gave. (The provided code is just printing out the SHA256 value of the blobs.)

Part 3: MD5 collisions

HINT

- Think about how you can hide junk you are creating, will be useful later as well
- Use `openssl dgst -sha256 file1 file2` and `openssl dgst -md5 file1 file2` to verify
- Remember to submit *good* and *evil*, **not** good.sh or evil.sh, **not** good.py or evil.py

good

```
#!/bin/bash
...
```

submission file example

Any questions, Piazzaaaaaaaaaaaaaaaa!!!

Thank you