Assignment 1 Feedback

Please find below the assignment TAs' feedback for Assignment 1.

General Feedback

Generally well done as expected, but students should be mindful of a few common mistakes that we spotted.

Many students failed to identify and explain the vulnerability in the challenges before going into the technical explanation of their solution or steps taken, which resulted in losing 1-2 marks.

- e.g. State "rand() is not cryptographically secure as its state can be replicated if we know the seed" before going into the solution
- Another example is H.2 - you should state "MD5 is cryptographically weak as its collisions can be trivially computed" before going into the solution
- No matter how trivial it may be, you should always state and explain the vulnerability before talking about your solution

For the future assignment, please submit your PDF writeup **separate** from your zip file.

We don't want you to just detail the steps that you took. Rather, try to link what you did to solve the challenges, to the concepts covered in the lecture, and in general, detail what you learned as you solved the challenge.

Please title your writeup sections properly - many people named the challenge "PDF collisions?!" as "H.1" when it is "H.2"...

Challenge-specific feedback

M.3 Red Herring

More of a small tip according to the trend observed in solution scripts submitted - you can iterate both forwards and backwards through an array of bytes.

The easiest way to build the dictionary would be to split at `[REDACTED]`, then build a dictionary while iterating forward from the front of the output file and plaintext file, and completing the dictionary by iterating backwards from the back of the output file and plaintext file. Most programming languages should allow you to specify the start, the end, and the iterative step in a for loop. If you set the start to the end of an array and the step to -1 you will be able to iterate backwards. For example in python you could do:

```
plaintext_split = plaintext.split('[REDACTED]')
mapping = {}

for letter, enc in zip(plaintext_split[0], ciphertext):
```

```
    mapping[enc] = letter
for letter, enc in zip(plaintext_split[1][::-1], ciphertext[::-1]):
    mapping[enc] = letter
```

This should produce a dictionary `mapping` that is sufficient to solve the challenge. Even if we don't know the length of the redacted flag, we can still know for sure that all other bytes are the same. Iterating from the back takes advantage of the fact that everything beyond the redacted flag is the same.

H.1 Lazy Programmer

A few people struggled with getting their solution to work in Python, and ended up resorting to using C. This is because Python and C use different PRNG generators (Python uses Mersenne Twister while Python C uses LCG). As a workaround, you can use Python to call C libraries:

```
from ctypes import CDLL
from ctypes.util import find_library

int seed = 999 # Input seed here
libc = CDLL(find_library("c"))
libc.srand(seed)
rand_num = libc.rand()
```

From the consultation and the questions posted, it also seems that many people weren't aware that they might have a mismatching timezone locally. The server runs on UTC+0, and C's time functions are timezone-aware by default, which results in local executions of the challenge working but then failing on remote (even executing the binary provided also uses your local timezone!).

H.2 PDF Collisions?!

There is a common misconception: just because 2 blocks have the same hash, does not mean that prepending the same bytes to both of these blocks result in the two binaries having the same hash again.

For example:

```
from hashlib import md5

msgA =
bytes.fromhex("cbe3762e9db2ecf29898db70ac026f69ace228cde438b3fd495a3a7df
3a943b698baefe5ec213a6f8b02ce399517f313c91800b6ae23a632b2d79935b5f60b2c8
15a1900efc3e75b5758e4943ef5d582c7f3991b505b32222bc340c8a65fa8b8cfbd42e89
f447136793b8a7b04681a867b69d00d26c23a4c6bafc5daf0b0ce62")
msgB =
bytes.fromhex("cbe3762e9db2ecf29898db70ac026f69ace2284de438b3fd495a3a7df
3a943b698baefe5ec213a6f8b02ce399597f313c91800b6ae23a632b2d799b5b5f60b2c8
15a1900efc3e75b5758e4943ef5d582c7f3999b505b32222bc340c8a65fa8b8cfbd42e89
```

```
f447136793b8a7b04e819867b69d00d26c23a4c6bafc55af0b0ce62")

# Observe that msgA and msgB are different, but have the same MD5 hash
assert msgA != msgB and md5(msgA).digest() == md5(msgB).digest()

# Now, prepend the letter "a" to both blocks
toPrepend = b"a" * 256
newMsgA = toPrepend + msgA
newMsgB = toPrepend + msgB

# Observe that now both hashes are different
assert md5(newMsgA).digest() != md5(newMsgB).digest()
```

How tools like `fastcoll` or `hashclash` work is when you specify a prefix (in this case, Assignment 1.pdf), the tool will find a set of blocks that **cancels out** the previous hash difference (using some math that is out of scope of this module). That is why if you hash only the blocks that was appended to the assignment PDF, they do not have the same hash.

For appending of data, this only works **if the prefix length is already a multiple of the hash block size**. In the context of this challenge, you can also solve it by padding "Assignment 1.pdf" to a block size of 32 with arbitrary data, then generating 2 completely new different binaries with the same MD5 hash, and finally prepend "Assignment 1.pdf" to each of the new binaries. This would also result in the same MD5 hash.

An oversight on my part was that PDF, at some point, allowed an arbitrary number of bytes before the PDF magic header. As a result, some people solved this challenge by generating 2 hash collision blocks, then appending the PDF at the end of each block. This was also accepted. For reference: https://stackoverflow.com/a/32179564/19723226

Something else that people failed to include in their writeup was that appending bytes at the end of the PDF does not result in any changes in the PDF visually when you open the document. This is specific to the PDF format, where any trailing bytes after the `%%EOF` will be ignored. Note that this is not always the case in all formats.