Project 1: Cryptographic Attacks

This project is due on **Tuesday, February 15** at **11:59pm**. This is a group project: you can work in teams of two and submit one writeup per team. If you are having trouble forming a teammate, please post to the #partner-search channel on Slack.

Your answers here must be entirely your own work. You may discuss the problems with other students, but you may not directly copy answers, or parts of answers. Start by downloading **proj1.zip** from Canvas: it will contain all the resources you need.

Part 1. Length Extension (10 points)

In most applications, you should use MACs such as HMAC-SHA256 instead of plain cryptographic hash functions (e.g. MD5, SHA-1, or SHA-256), because hashes often fail to match our intuitive security expectations, which can lead to unforeseen vulnerabilities. One of these unexpected features is that many hashes are subject to length extension. Hash functions such as MD5 or SHA-1 use a design called the Merkle-Damgård construction. Messages are processed in fixed-sized blocks by applying a compression function to each block in order, and using the result to update an internal state. This state starts as a default initialization value, and updates each block. The state after the final application of the compression function becomes the output of the hash function. A consequence of this design is that if we know the hash of an n-block message, we can find the hash of longer messages by applying the compression function for each block that we want to add. This process is called length extension, and it can be used to attack many applications of hash functions.

1.1 Experiment with Length Extension in Python (0 points)

To experiment with this idea, we'll use a Python implementation of the MD5 hash function, though SHA-1 and SHA-256 are vulnerable to length extension too. The project resources contain the **pymd5.py** file, which contains library functions we will use for this. Put it in the same directory as your solution.

NOTE: The **pymd5** file on Canvas has been updated since the recitation! Please re-download it, or you may have trouble with some of the code snippets below.

Use a Jupyer notebook on coding.csel.io, or your own local Python environment, to perform the following experiments:

Start by importing the library with the command:

```
from pymd5 import md5, padding
```

Consider the string “Use HMAC, not hashes”. We can compute its MD5 hash by running the following command:

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

The output should be: 3ecc68efa1871751ea9b0b1a5b25004d

*Note: the **b** in front of some of the message- this is used in Python to indicate the string should be treated as an array of bytes, not unicode characters. We will use this whenever possible, as we will deal with some byte arrays which cannot be converted into regular Python strings.*

MD5 processes messages in 512-bit blocks, so the hash function always pads a message to be a multiple of that length. The padding consists of the 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 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 a given message, which we will call **m**, we could compute MD5(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, 512). This will reset it to the state it was in when the original hash was being calculated! To find the padded message length, guess the length of m and run

```
bits = int(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("3ecc68efa1871751ea9b0b1a5b25004d"),
      count=bits)
```

Now you can use length extension to find the hash of a longer string that appends the suffix “Good advice.” Simply run:

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

To execute the compression function over x and output the resulting hash. Verify that it equals the MD5 hash of

```
m + padding(len(m)*8) + suffix
```

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. This component is intended to introduce length extension and familiarize you with the Python MD5 module we will be using; you will not need to submit anything for it.

1.2 Conduct a Length Extension Attack (10 points)

Length extension attacks can cause serious vulnerabilities when people mistakenly try to construct something like an HMAC by using **hash(secret + message)**. The National Bank of CSCI 3403, which is not up-to-date on its security practices, hosts an API that allows its client-side applications to perform actions on behalf of a user by loading URLs of the form:

https://csci3403.com/proj1/api?token=1e755d78dcb4d783b2573b8d04fcc48a&user=admin&command1=ListFiles&command2=NoOp

where **token** is MD5([user's 8-character password] + [the rest of the URL starting from user= and ending with the last command]).

Using the techniques that you learned in the previous section and without guessing the password, apply length extension to create a URL ending with **&command3=DeleteAllFiles** that is treated as valid by the server API. You have permission to use our server to check whether your command is accepted.

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

Historical fact: In 2009, security researchers found that the API used by the photo-sharing site Flickr suffered from a length-extension vulnerability almost exactly like the one in this exercise.

What to submit: A Python 3.x script named len_ext_attack.py that:

1. Accepts a valid URL in the same form as the one above as a command line argument.
2. Modifies the URL so that it will execute the DeleteAllFiles command as the user.
3. Successfully performs the command on the server and prints the server's response.

You should make the following assumptions:

- The input URL may be for a user with a different password, but the length of the password will always be 8 characters.
- The server's output might not exactly match what you see during testing.

You can base your code on the following example (it might make sense to hard-code the command line argument if testing in a Jupyter notebook):

```
import http.client
import urllib.parse
import sys

if len(sys.argv) != 2:
     print('Requires the URL to extend as a command line argument.')
     exit(1)

original_url = sys.argv[1]

# Your code to modify url goes here
new_url = ...

# The following code requests the URL and returns the response from
the server
parsed_url = urllib.parse.urlparse(new_url)
conn = http.client.HTTPSConnection(parsed_url.hostname,
parsed_url.port)
conn.request("GET", parsed_url.path + "?" + parsed_url.query)
print(conn.getresponse().read())
```

Part 2. MD5 Collisions (15 points)

MD5 was once the most widely used cryptographic hash function, but today it is considered dangerously insecure. This is because cryptanalysts have discovered efficient algorithms for finding collisions- pairs of messages with the same MD5 hash value. The first known collisions were announced on August 17, 2004, by Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. Here's one pair of colliding messages they published:

Message 1:

```
d131dd02c5e6eec4693d9a0698aff95c 2fcab58712467eab4004583eb8fb7f89
55ad340609f4b30283e488832571415a 085125e8f7cdc99fd91dbdf280373c5b
d8823e3156348f5bae6dacd436c919c6 dd53e2b487da03fd02396306d248cda0
e99f33420f577ee8ce54b67080a80d1e c69821bcb6a8839396f9652b6ff72a70
```

Message 2:

```
d131dd02c5e6eec4693d9a0698aff95c 2fcab50712467eab4004583eb8fb7f89
55ad340609f4b30283e4888325f1415a 085125e8f7cdc99fd91dbd7280373c5b
d8823e3156348f5bae6dacd436c919c6 dd53e23487da03fd02396306d248cda0
e99f33420f577ee8ce54b67080280d1e c69821bcb6a8839396f965ab6ff72a70
```

Copy the two messages into plain text files (preserving all spaces and newlines exactly as they appear above) and convert each into a binary file:

```
$ xxd -r -p file.hex > file
```

1. What are the MD5 hashes of the two binary files? Verify that they're the same.
   `$ md5sum file1 file2`
2. What are their SHA-256 hashes? Verify that they're different.
   `$ sha256sum -sha256 file1 file2`

This component is intended to introduce you to MD5 collisions; you will not submit anything for it.

2.1 Generating Collisions Yourself (5 points)

In 2004, Wang's method took more than 5 hours to find a collision on a desktop PC. Since then, researchers have introduced vastly more efficient collision finding algorithms. You can compute your own MD5 collisions using a tool written by Marc Stevens that uses a more advanced technique.

You can generate collisions using the fastcoll tool, which is in the zipped resources file on canvas. There is a Windows binary in **windows/fastcoll_v1.0.0.5.exe**, as well as a Linux executable that should work on coding.csel.io, in **linux/fastcoll**. Sadly, each binary is a different version, and will have slightly different commands. The source code can also be found in **fastcoll_v1.0.0.5_source.zip**, if you would prefer to build it yourself. If you are building fastcoll from source, you can compile it using the provided Makefile. You will also need the Boost libraries. On Ubuntu, you can install these using apt-get install libboost-all-dev. On OS X, you can install Boost via the Homebrew package manager using `brew install boost`.

1. Generate your own collision with this tool. How long did it take?
   Linux: `$ time fastcoll -`
   Windows: `$ time fastcoll_v1.0.0.5.exe -o md5_data1 md5_data2`
2. What are your files? To get a hex dump,
   `$ xxd -p file.`
3. What are their MD5 hashes? Verify that they're the same.
4. What are their SHA-256 hashes? Verify that they're different.

What to submit: A text file named generating_collisions.txt containing your answers.

2.2 A Hash Collision Attack (10 points)

The collision attack lets us generate two messages with the same MD5 hash and any chosen (identical) prefix. Due to MD5's length-extension behavior, we can append any suffix to both

messages and know that the longer messages will also collide. This lets us construct files that differ only in a binary “blob” in the middle and have the same MD5 hash, i.e. *prefix* + $blob_A$ + *suffix* and *prefix* + $blob_B$ + *suffix*.

We can leverage this to create two programs that have identical MD5 hashes but wildly different behaviors. We’ll use Python, but almost any language would do. Put the following lines into a file called prefix:

```
#!/usr/bin/python
# -*- coding: ISO-8859-1 -*-
blob = """
```

and put these lines into a file called suffix:

```
"""

from hashlib import sha256
print(sha256(blob.encode()).hexdigest())
```

Now use fastcoll to generate two files with the same MD5 hash that both begin with prefix:

Linux: `$ ./fastcoll prefix`
Windows: `$ fastcoll_v1.0.0.5.exe -p prefix -o md5_data1 md5_data2`

Then append the suffix to both:

```
$ cat md5_data1 suffix > file1.py; cat md5_data2 suffix > file2.py
```

Verify that file1.py and file2.py have the same MD5 hash but generate different output.

Extend this technique to produce another pair of programs, good and evil, that also share the same MD5 hash. One program should execute a benign payload: print "I come in peace." The second should execute a pretend malicious payload: print "Prepare to be destroyed!". Note that we may rename these programs before grading them.

What to submit Two Python 3.x scripts named good.py and evil.py that have the same MD5 hash, have different SHA-256 hashes, and print the specified messages.

Submission Checklist

Part 1.2: **len_ext_attack.py**
Part 2.1: **generating_collisions.txt**
Part 2.2: **good.py** and **evil.py**