Project 1: Cryptographic Attacks

This project is due on **14Sept2020** at **6 p.m. (1800) CT** and late submissions will be handled as discussed in the syllabus. Submissions will recieve a zero (0) if submitted after 17Sept2020 at 6 p.m. Please plan accordingly for your schedule and turn in your project early if appropriate.

The code and other answers you submit must be entirely your own work. You are encouraged to consult with other students about the concepts of the project and the meaning of the questions via Piazza but you may not look at any part of someone else's solution or collaborate on solutions. You may consult published references, provided that you appropriately cite them (e.g. with program comments), as you would in an academic paper.

In this project, you will investigate historical vulnerabilities in a widely used cryptographic hash function (Part 1), exploit incorrect usage of a hash function to bypass an authentication system (Part 2) length, and optionally imitate cryptanalytic efforts used to defeat certain encryption schemes (Part 3). Each section will walk you through the necessary background and context that was not covered in lecture but please ask on Piazza if any aspects are unclear (though do **not** post solutions or solution steps).

Part 1. MD5 Collisions (50 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 creating *hash 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
```

Convert each message's hex string into a binary file[1] and answer the following questions:

[1] On Linux, copy-paste the hex to a file and run `$ xxd -r -p msgX.hex > msgX.bin`

1. What are the MD5 hashes of the two binary files? **Verify that they have the same MD5 hash.**[2]

2. What are their SHA256 hashes? **Verify that they do not have the same SHA256 hash.**

What to submit There is nothing to submit for this portion of the assignment.

1.1 Generating Collisions Yourself

Since Wang's initial collision in 2004, researchers have introduced vastly more efficient collision finding algorithms. For many years, it has been possible to compute your own MD5 collisions using a laptop and a tool written by Marc Stevens which uses a more advanced technique compared to Wang's. You can download the `fastcoll` tool via `http://www.win.tue.nl/hashclash/fastcoll_v1.0.0.5.exe.zip` (Windows executable) or `http://www.win.tue.nl/hashclash/fastcoll_v1.0.0.5-1_source.zip` (source code)

If you are building `fastcoll` from source, you can compile using the `Makefile.fastcoll` file on Canvas[3]. Once you can run the fastcoll binary, perform the following actions and answer the associated questions:

1. Generate your own collision with this tool[4]. How long did it take?

2. What are your files? To get a hex dump, run `$ xxd -p file`.

3. What are their MD5 hashes? Verify that they're the same.

4. What are their SHA256 hashes? Verify that they're different.

What to submit An ascii-only text file named `collisions.txt` containing your answers. Format your file using the template:

```
# Question 1
time_for_fastcoll (e.g. 3.456s)

# Question 2
file1:
file1_hex_dump
file2:
file2_hex_dump

# Question 3
md5_hash
```

[2] `$ openssl dgst -md5 msg1.bin msg2.bin` or `$ md5sum msg[12].bin`

[3] On Ubuntu, you can install the dependencies using `apt-get install libboost-all-dev`. On OS X, you can install the dependencies via the Homebrew package manager using `brew install boost`. On Windows, you're on your own...

[4] `$ time fastcoll -o file1 file2`

```
# Question 4
file1:
file1_sha256_hash
file2:
file2_sha256_hash
```

1.2 A Hash Collision Attack

The collision attack lets us generate two messages with the same MD5 hash and any chosen (identical) prefix. Due to MD5's Merkle-Damgård construction, 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* have the same MD5 hash.

Use `fastcoll` to generate two Python3 scripts with identical prefixes and the same MD5 hash but that print completely different strings to the screen. The `collide_attack_one.py` script should print `print("Hashing is not encryption!")` and the `collide_attack_two.py` script should print `print("Security through obscurity is snake-oil!")`. Both files should also print the SHA256 of the blob used to collide the two files.

NOTE: You should to use fastcoll's `-p` flag and *not* any self-referential functionality such as `__file__`.

What to submit Two Python3 files (will most likely contain non-ascii characters) with the above functionality with the names `collide_attack_one.py` and `collide_attack_two.py`.

Part 2. Length Extension (50 points)

In most applications, you should use HMACs such as HMAC-SHA256 instead of plain cryptographic hash functions (e.g. MD5, SHA-1, or SHA-256) when combining arbitrary content with a secret. Hash functions fail to match our intuitive security expectations in many ways and sometimes can result in serious vulnerabilities in real-world systems.

One difference between hash functions and HMACs is that many hash functions are subject to *length extension* due to their use of the Merkle-Damgård construction. Each of the vulnerable hash functions are built around a *compression function* f which maintains an internal state s (initialized to a fixed constant). Messages are processed in fixed-sized blocks by applying the compression function to the current state and current block to compute an updated internal state, i.e. $s_{i+1} = f(s_i, b_i)$. The result of 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 additional input block $b_{n+1}, b_{n+2}, \ldots$. By doing so, it is possible to create $H(input_1 || input_2)$ by only knowing $H(input_1)$ and $input_2$. This process is called "length extension", and it can be used to attack many application which fail to account for its existence.

2.1 Experiment with Length Extension in Python

To experiment with this idea, we'll use a Python implementation of the MD5 hash function but SHA-1 and SHA-256 are vulnerable to length extension too. You can download the `pymd5` module from Canvas and learn how to use it by browsing the file's comments. To follow along with these examples, run Python in interactive mode (`$ python3 -i`) and run the command `from pymd5 import md5, padding`.

Consider the string "Use HMAC, not hashes". We can compute its MD5 hash by running:

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

or, more compactly, `print(md5(m).hexdigest())`. The output should be:

```
3ecc68efa1871751ea9b0b1a5b25004d
```

MD5 processes messages in 512-bit blocks, such that the hash function internally pads *m* to 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 64-bit 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:

```
from codecs import decode
h = md5(state=decode("3ecc68efa1871751ea9b0b1a5b25004d", "hex"), count=bits)
```

Now you can use length extension to find the hash of a longer string that appends the suffix "Good advice.".

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

This will execute the compression function over `x` and output the resulting hash. Verify that it equals the MD5 hash of `m.encode() + padding(len(m)*8) + x.encode()`. 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.

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

2.2 Conduct a Length Extension Attack

Length extension attacks can cause serious vulnerabilities when people incorrectly try to replicate an HMAC by using $hash(secret \parallel message)$[5]. The National Bank of COMP-5970 made a poor hire when they chose their security engineers and now hosts an API using the above incorrect HMAC construction for authentication. This API allows its client-side applications to perform actions on behalf of a user by loading URLs of the form:

`https://proj1.comp5370.org/api?token=5fca9858ef492b6c435a0bb9607e8ac2`
`&user=admin&action1=lock-safe&customer1=Alice`

where `token` is MD5(*user's 8-character password* || `user=` ... [*the rest of the URL*]).

The API has the actions `no-op`, `lock-safe`, and `unlock-all-safes` and multiple actions can be accomplished via having multiple `actionX` pairs in the URL query[6]. If `action`*n*`=no-op`, then `customer`*n*'s value is ignored completely. Using the techniques that you learned in the previous section and without guessing the password, apply length extension to create a URL with the `unlock-all-safes` action that is treated as valid by the server API.

You have permission to use the API on the proj1.comp5370.org server until the 0% deadline to check whether your command is accepted. You may use the API as much as you like to check whether a URL is correct but **no automated interaction with the server is allowed**. Copy-pasting the URL into a browser is perfectly acceptable and a highly recommended way to check whether it has been correctly length-extended.

Hint: You might want to use Python's `urllib` module to encode non-ASCII characters in the URL.

What to submit A Python 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 `unlock-all-safes` action.
3. Prints the length-extended URL to the screen.

You should make the following assumptions:

- The input URL will have the same form as the sample above, but we may change the server hostname and the values of `token`, `action`*X*, and `customer`*X*. These values may be of substantially different lengths than in the sample.
- The input URL may be for a user with a different password, but the length of the password will be unchanged.

[5] || is the symbol for contatenation, i.e. "hello" || "world" = "helloworld".

[6] Usage examples in the example-urls.txt on Canvas.

Part 3. Classical Cryptanalysis (Up to 15 Bonus Points)

Here is a Python dictionary of the relative frequency of letters in English text:

```
{ "A": .08167, "B": .01492, "C": .02782, "D": .04253, "E": .12702, "F": .02228,
  "G": .02015, "H": .06094, "I": .06996, "J": .00153, "K": .00772, "L": .04025,
  "M": .02406, "N": .06749, "O": .07507, "P": .01929, "Q": .00095, "R": .05987,
  "S": .06327, "T": .09056, "U": .02758, "V": .00978, "W": .02360, "X": .00150,
  "Y": .01974, "Z": .00074 }
```

Here is some plaintext:

```
ethicslawanduniversitypolicieswarningtodefendasystemyouneedtobeabletot
hinklikeanattackerandthatincludesunderstandingtechniquesthatcanbeusedt
ocompromisesecurityhoweverusingthosetechniquesintherealworldmayviolate
thelawortheuniversitysrulesanditmaybeunethicalundersomecircumstancesev
enprobingforweaknessesmayresultinseverepenaltiesuptoandincludingexpuls
ioncivilfinesandjailtimeourpolicyineecsisthatyoumustrespecttheprivacya
ndpropertyrightsofothersatalltimesorelseyouwillfailthecourseactinglawf
ullyandethicallyisyourresponsibilitycarefullyreadthecomputerfraudandab
useactcfaaafederalstatutethatbroadlycriminalizescomputerintrusionthisi
soneofseverallawsthatgovernhackingunderstandwhatthelawprohibitsifindou
btwecanreferyoutoanattorneypleasereviewitsspoliciesonresponsibleuseoft
echnologyresourcesandcaenspolicydocumentsforguidelinesconcerningproper
```

The *population variance* of a finite population X of size N and mean μ is given by

$$\mathrm{Var}(\mathrm{X}) = \frac{1}{\mathrm{N}} \sum_{\mathrm{i}=1}^{\mathrm{N}} (\mathrm{x_i} - \mu)^2.$$

1. What is the population variance of the relative letter frequencies in the given plaintext?

2. For each of the following keys — `yz`, `xyz`, `wxyz`, `vwxyz`, `uvwxyz` — encrypt the plaintext with a Vigenère cipher and the given key, then calculate the population variance of the relative letter frequencies in the resulting ciphertext.

3. Viewing a Vigenère key of length k as a collection of k independent Caesar ciphers, calculate the mean of the frequency variances of the ciphertext for each one. (E.g., for key `yz`, calculate the frequency variance of the even numbered ciphertext characters and the frequency variance of the odd numbered ciphertext characters. Then take their mean.)

4. Consider the ciphertext that was produced with key `uvwxyz`. Let's assume you are trying to figure out the length of the key used to encrypt that ciphertext. Calculate the mean of the frequency variances that arise if you assume that the key had length 2, 3, 4, and 5 (using the same technique you did in part (4)). How can this help you determine the length of the key?

What to submit A Python script named `analysis.py` that takes in text via `stdin` and outputs the answers to the previous questions in the following format:

```
1: variance_of_plaintext

2_xy: variance_yz_ciphertext
2_xyz: variance_xyz_ciphertext
2_wxyz: variance_wxyz_ciphertext
2_vwxyz: variance_vwxyz_ciphertext
2_uvwxyz: variance_uvwxyz_ciphertext

3_yz: mean_variance_yz_caesar_ciphers
3_xyz: mean_variance_xyz_caesar_ciphers
3_wxyz: mean_variance_wxyz_caesar_ciphers
3_vwxyz: mean_variance_vwxyz_caesar_ciphers
3_uvwxyz: mean_variance_uvwxyz_caesar_ciphers

4_len2: mean_variance_keylen_2
4_len3: mean_variance_keylen_3
4_len4: mean_variance_keylen_4
4_len5: mean_variance_keylen_5
```

Errata

27Aug2020 → 02Sept2020

1. Remove illogical ascii-only requirement for Python scripts submitted to fulfill Part 1.2
2. Fix question-vs-submission numbering error in Part 3