CSE127: Computer Security **Winter 2026**

PA5 — Cryptography

Project Release: February 28, 2026 PT
Deadline: March 14, 2026 by 11:59:59pm PT

Introduction

In this project, we'll start by investigating Vigenère ciphers, then move on to investigating vulnerabilities in widely used cryptographic hash functions, including length-extension attacks and collision vulnerabilities, and an implementation vulnerability in a popular digital signature scheme.

Forming a Group

This is a group project; you can work in a team of size at most two and submit one project per team. You are not required to work with the same partner on every project. You and your partner should collaborate closely on each part.

The code and other answers you submit must be entirely your team's own work. You may discuss the conceptualization of the project and the meaning of the questions, but you may not look at any part of someone else's solution or collaborate with anyone other than your partner. You may consult published references, provided that you appropriately cite them (e.g., with program comments).

Solutions must be submitted to Gradescope.

Part 1: Vigenère ciphers (6 points)

For this problem, solve by hand or write a program (perhaps in Python).

You can read about how the Vigenère cipher works on Wikipedia. Vigenère ciphers can be generally deciphered using Kasiski Examination, which is discussed on the Wikipedia page.

You can find some ciphertext produced with the Vigenere cipher under a certain key on Gradescope as the assignment "PA5: Ciphertext."

We also provided sample code for decrypting ciphertext sample decryption code here.

Encrypting a plaintext letter with a key letter A results in no change, encrypting with a key letter B results in an increment by one place in the alphabet, encrypting with key letter C results in an increment by two places, etc. Also assume that the original plaintext contains only uppercase letters (A–Z) and no spaces or punctuation.

For example, encrypting the plaintext: ATTACKATDAWN with the key: BLAISE results in the following ciphertext:

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

The goal for this part of the assignment is to figure out what key was used to encrypt your ciphertext.

What to submit A text file named `vigenere.key` containing your key.

Historical note: In November 2019, it was discovered that the security company Fortinet was using "XOR encryption with a static key" in some products, which is similar to a Vigenère cipher and has similar (lack of) security properties. https://seclists.org/bugtraq/2019/Nov/38.

Part 2: Length extension (7 points)

For many 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, also known as digests, do not provide security against an attacker who can control or modify the hash value. What we really want is something that behaves like a pseudorandom function, which HMAC and other MAC functions are constructed to behave like and hash functions alone do not.

One difference between hash functions and pseudorandom functions is that a collision-resistant hash function may still be subject to length extension. Many common hash functions use a design called the Merkle-Damgard construction. Each is built around a compression function f and maintains an internal state s, which is initialized to a fixed constant. Messages are processed in fixed-size 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 block $b_{n+1}, b_{n+2}, \ldots$ that we want to add. This process is called length extension, and it can be used to attack many applications of hash functions.

2a. Experimenting

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 in the same way. You can download the `pymd5` module here and learn how to use it by running `pydoc pymd5`. To follow along with these examples, run Python in interactive mode and run the command `from pymd5 import md5, padding`.

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

```
from pymd5 import md5, padding
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, so internally, the hash function 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 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

$$m + \text{padding}(\text{len}(m) \cdot 8) + \text{suffix}$$

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:

$$\text{bits} = (\text{length\_of\_m} + \text{len}(\text{padding}(\text{length\_of\_m} \cdot 8))) \times 8.$$

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

```
h = md5(state=bytes.fromhex("3ecc68efa1871751ea9b0b1a5b25004d"), count=512)
```

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

```
x = "Good advice"
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:

$$m.\text{encode}(\text{"utf-8"}) + \text{padding}(\text{len}(m) \cdot 8) + x.\text{encode}(\text{"utf-8"})$$

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 `bytes` 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.

This part of the assignment 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.

2b. Conduct a length extension attack

Length extension attacks can cause serious vulnerabilities when people try to construct something like a MAC by using `hash(secret || message)` using a hash function like MD5, SHA1, or SHA2, that is vulnerable to length extension. (SHA3, unlike SHA-256, SHA1, and MD5, uses a different construction and is not vulnerable to length extension attacks, so `SHA3(secret || message)` is a secure MAC. This is why cryptography is hard.)

The National Bank of CSE 127, 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:

```
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=...`) (the rest of the decoded 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=UnlockAllSafes` that would be treated as valid by the server API.

Note: Because of its bad security practices, the National Bank of CSE 127 has taken down its website. So you'll have to use Gradescope to test if your attack URL works. Gradescope will tell you if your attack works on the example URL provided when you upload your attack.

Hint: You might want to use the `quote()` function from Python's `urllib.parse` 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:

- Accepts a valid URL in the same form as the one above as a command line argument.
- Modifies the URL so that it will execute the `UnlockAllSafes` command as the user.
- Prints the new URL to the command line.

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`, `user`, `command1`, and `command2`. 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.

You can base your code on the following example:

```
import sys, urllib.parse
from pymd5 import md5, padding
url = sys.argv[1]

# Your code to modify url goes here

print(new_url)
```

Part 3: MD5 collisions (7 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 above hex strings into file1.hex and file2.hex. Convert each group of hex strings into a binary file. (On Linux, run `xxd -r -p file.hex > file`.)

1. What are the MD5 hashes of the two binary files? Verify that they're the same. (`openssl dgst -md5 file1 file2`)
2. What are their SHA-256 hashes? Verify that they're different. (`openssl dgst -sha256 file1 file2`)

You don't need to submit anything for the above questions.

Generating collisions yourself

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, and found a collision for SHA1 as well, though that attack is not yet efficient enough to give as an undergraduate assignment. You can compute your own MD5 collisions using a tool written by Marc Stevens that uses a more advanced technique.

You can download the `fastcoll` tool here:

`https://www.win.tue.nl/hashclash/fastcoll_v1.0.0.5.exe.zip` (Windows executable)

or

`https://cseweb.ucsd.edu/classes/wi26/cse127-a/fastcoll` (M* Series Mac OS executable)

or

`https://www.win.tue.nl/hashclash/fastcoll_v1.0.0.5-1_source.zip` (source code)

If you are compiling `fastcoll` from source, you can compile using this `Makefile`. You will also need to have installed the Boost libraries. On Ubuntu, you can install using `apt-get install libboost-all-dev`. On OS X, you can install Boost via the Homebrew package manager using `brew install boost`.

Note: If you are compiling from source (e.g., on MacOS), you will need to make sure you are pointing `g++` to the right boost includes and libraries, which may be at a slightly different path than what is listed in the Makefile (e.g., the -I and -L flags). This may mean slight modifications to the Makefile for your system.

1. Generate your own collision with this tool. How long did it take? (`time ./fastcoll -o file1 file2`)
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 SHA-256 hashes? Verify that they're different.

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 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 || blobA || suffix` and `prefix || blobB || suffix`.

We can leverage this to create two programs (Bash scripts) that have identical MD5 hashes but wildly different behaviors. We're using shell scripts, but this could be done using a program in almost any language. Put the following two lines into a file called `prefix`:

```
#!/bin/bash
cat << "EOF" | openssl dgst -sha256 > DIGEST
```

and put these four lines (starting with a blank line) into a file called `suffix`:

```

EOF
digest=$(cat DIGEST | sed 's/(stdin)= //')
echo "The sha256 digest is $digest"
```

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

(`fastcoll -p prefix -o col1 col2`)

Then append the suffix to both (`cat col1 suffix > file1.sh; cat col2 suffix > file2.sh`). Verify that `file1.sh` and `file2.sh` 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: `echo` or print "I mean no harm." The second should execute a pretend malicious payload: `echo` or print "You are doomed!"

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

Submission Checklist

Submit the following to Gradescope:

- `vigenere.key` (for Part 1)
- `len_ext_attack.py` (for Part 2)
- `good` (for Part 3)
- `evil` (for Part 3)