

## Project 1: Attacking Cryptographic Hashes

This project is due on **Wednesday, January 28 at 6 p.m.** and counts for 8% of your course grade. Late submissions will be penalized by 10% plus an additional 10% every 5 hours until received. You may submit one assignment late this term without this penalty. In either case, late work will not be accepted after 20 hours past the deadline. If you have a conflict due to travel, interviews, etc., please plan accordingly and turn in your project early.

This is a group project; you will work in **teams of two** and submit one project per team. Please find a partner as soon as possible. If have trouble forming a team, post to Piazza's partner search forum.

The code and other answers your group submits must be entirely your own work, and you are bound by the Honor Code. You may consult with other students about 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 outside your group. You may consult published references, provided that you appropriately cite them (e.g., with program comments), as you would in an academic paper.

Solutions must be submitted electronically via CTools, following the submission checklist below.

---

### Introduction

In this project, you will investigate vulnerabilities in widely used cryptographic hash functions, including length-extension attacks and collision vulnerabilities. In Part 1, we will guide you through attacking the authentication capability of an imaginary server API. The attack will exploit the length-extension vulnerability of hash functions in the MD5 and SHA family. In Part 2, you will use a cutting-edge tool to generate different messages with the same MD5 hash value (collisions). You'll then investigate how that capability can be exploited to conceal malicious behavior in software.

### Objectives:

- Understand how to apply basic cryptographic integrity primitives.
- Investigate how cryptographic failures can compromise the security of applications.
- Appreciate why you should use HMAC-SHA256 as a substitute for common hash functions.

## Part 1. Length Extension

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, also known as digests, fail to match our intuitive security expectations. What we really want is something that behaves like a pseudorandom function, which HMACs seem to approximate and hash functions do not.

One difference between hash functions and pseudorandom functions is that many hashes are subject to *length extension*. All the hash functions we've discussed use a design called the Merkle-Damgård 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-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 block  $b_{n+1}, b_{n+2}, \dots$  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

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 at <https://www.eecs.umich.edu/courses/eecs388/static/pymd5.py> and learn how to use it by running `$ pydoc pymd5`. To follow along with these examples, run Python in interactive mode (`$ python -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, 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 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  $m + \text{padding}(\text{len}(m)*8) + \text{suffix}$  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 `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:

```
h = md5(state="3ecc68efa1871751ea9b0b1a5b25004d".decode("hex"), 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  $m + \text{padding}(\text{len}(m)*8) + x$ . 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

One example of when length extension causes a serious vulnerability is when people mistakenly try to construct something like an HMAC by using `hash(secret || message)`, where `||` indicates concatenation. For example, Professor Vuln E. Rabble has created a web application with an API that allows client-side programs to perform an action on behalf of a user by loading URLs of the form:

```
http://eecs388.org/project1/api?token=b301afea7dd96db3066e631741446ca1
&user=admin&command1=ListFiles&command2=NoOp
```

where `token` is `MD5(user's 8-character password || user=... [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 might want to use the `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 2.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 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.
- The server's output might not exactly match what you see during testing.

You can base your code on the following example:

```
import urllib, urlparse, sys
url = sys.argv[1]

# Your code to modify url goes here

parsedUrl = urlparse.urlparse(url)
conn = urllib.HTTPConnection(parsedUrl.hostname)
conn.request("GET", parsedUrl.path + "?" + parsedUrl.query)
print conn.getresponse().read()
```

## Part 2. MD5 Collisions

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
```

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`)

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

## 2.1 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. 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:

[http://www.win.tue.nl/hashclash/fastcoll\\_v1.0.0.5.exe.zip](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](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 this makefile: <https://www.eecs.umich.edu/courses/eecs388/static/project1/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?  
(`$ 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.

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

## 2.2 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 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.  $prefix \parallel blob_A \parallel suffix$  and  $prefix \parallel blob_B \parallel suffix$ .

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 three lines into a file called `prefix`:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
blob = ""
```

and put these three lines into a file called `suffix`:

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

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.py; cat col2 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 program before grading them.

**What to submit** Two Python 2.x scripts named `good.py` and `evil.py` that have the same MD5 hash, have different SHA-256 hashes, and print the specified messages.

## Part 3. Writeup

1. With reference to the construction of HMAC, explain how changing the design of the API in Section 1.2 to use `token = HMACuser's password(user=....)` would avoid the length-extension vulnerability.
2. Briefly explain why the technique you explored in Section 2.2 poses a danger to systems that rely on digital signatures to verify the integrity of programs before they are installed or executed. Examples include Microsoft Authenticode and most Linux package managers. (You may assume that these systems sign MD5 hashes of the programs.)

**What to submit** A text file named `writeup.txt` containing your answers.

## Submission Checklist

Upload to CTools a gzipped tarball (`.tar.gz`) named `project1.uniqname1.uniqname2.tar.gz`. The tarball should contain only the following files:

### Section 1.2

A Python script named `len_ext_attack.py` that inputs a URL, performs the specified attack on the web application, and outputs the server's response.

### Section 2.2

A text file named `generating_collisions.txt` with your answers to the four short questions, using the following template:

#### Submission Template

1. *time\_for\_fastcoll* (e.g. 3.456s)

----

2.

file1:

*file1\_hex\_dump*

file2:

*file2\_hex\_dump*

----

3. *md5\_hash*

----

4.

file1: *file1\_sha256\_hash*

file2: *file2\_sha256\_hash*

### Section 2.3

Two Python scripts, named `good.py` and `evil.py`, that share an MD5 hash, have different SHA-256 hashes, and print the specified messages.

## Writeup

A text file containing your answers to the two questions, named `writeup.txt`.

## Bonus Challenge (Optional)

To reward everyone for their hard work on this project, we are holding a contest. The goal is to guess a number that we will announce in class on February 4. If you follow the instructions and you guess correctly, we'll drop your lowest homework grade at the end of the term.

Here are the rules:

1. At the start of class on February 4, the professor will announce the winning number, which will be an integer in the range  $[0, 63]$ .
2. Prior to the announcement, each student may register one guess. To keep everything fair, your guesses will be kept secret using the following procedure:
  - (a) You'll create a PostScript file that, when printed, produces a single page that contains only the statement: "*your\_username* guesses *n*".
  - (b) You'll register your guess by sending an email with the MD5 hash of your file to `eecs388-staff@umich.edu`. We must receive your hash value before the professor announces the pick.
  - (c) You and your project partner may collaborate and produce a file that includes both of your usernames, or you may choose to work independently.
3. If your guess is correct, you can claim the prize by emailing your PostScript file to the same address. The professor will verify its MD5 hash, print it to a PostScript printer, and check that the statement is correct.

*Hint:* You're allowed to trick us if it will teach us not to use a Turing-complete printer control language with a hash function that has weak collision resistance.