Project 1: Cryptographic Attacks

This project is due on **14Feb2020** at **6 p.m. (1800)** and late submissions will be handled as discussed in the syllabus. Submissions will recieve a zero (0) if submitted after 17Feb2020 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 conceptualization of the project and the meaning of the questions via Canvas discussions, 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.

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.

Part 1. Length Extension

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) because hashes (a.k.a. digests) fail to match our intuitive security expectations. One difference between hash functions and HMACs is that many hashes are subject to *length extension*. As discussed in class, this is due to the their use of 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 $b_{n+1}, b_{n+2}, \ldots$ 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 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." 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.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.

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

Length extension attacks can cause serious vulnerabilities when people mistakenly try to construct something like an HMAC by using $hash(secret \parallel message)$[1]. The National Bank of COMP-5970, 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://aaspring.space/api?token=5bc7cbc10b1871604a31a2de0092bcd1&user=admin
&action1=lock-safe&customer1=Alice
```

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

The API has the actions `no-op`, `lock-safe`, and `unlock-all-safes` (usage examples in the example-urls.txt on Canvas). 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 aaspring.space server until the 0% deadline to check whether your command is accepted.

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. Successfully performs the action 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`, `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.
- The server's output might not exactly match what you see during testing.

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.

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

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` (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 provided on Canvas. 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. Format your file using this template. **Make sure to follow the format precisely as this part is autograded**:

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

# Question 2
file1:
file1_hex_dump
file2:
file2_hex_dump

# Question 3
md5_hash

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

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. $\textit{prefix} \parallel \textit{blob}_A \parallel \textit{suffix}$ and $\textit{prefix} \parallel \textit{blob}_B \parallel \textit{suffix}$.

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 `one.py` script should print `print("Hashing is not encryption!")` and the `two.py` script should print `print("Security through obscurity!")`. 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__`.