Homework 3: Hash Attacks

This homework is due **Monday, October 18** at **5 p.m.** and counts for 4% of your course grade. Late submissions will be penalized by 10% plus an additional 10% every 12 hours until received. (The professor may grant individual extensions, but only under truly extraordinary circumstances.)

You are free to discuss the problems with other members of the class, but the answers you turn in must be your own work. Email your submission (as text or as a PDF attachment) to `eecs398@umich.edu` with "`HW3 Submission`" as the subject. You should receive a confirmation email within 15 minutes.

Solve both of the following problems. Submit the Python programs you write along with your answers. Don't be intimidated by the length of the assignment; you are only asked to complete a small number of tasks.

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 usually a pseudorandom function, which HMACs seem to approximate and hash functions do not.

   One difference between hash functions and pseudorandom functions is that most hashes are subject to *length extension*. All the hashes 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 finalapplication 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 $b_{n+1}, b_{n+2}, \ldots$ we want to add. This process is called length extension.

   To explore 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 exactly the same way. You can download the `pymd5` module at http://www.eecs.umich.edu/courses/eecs398/homework/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`.

(a) Consider the string `m = "Use HMAC, not hashes"`. We can compute its MD5 hash by running `h = md5(); h.update(m); print h.hexdigest()`, or more compactly, `print md5(m)`. The output should be `3ecc68efa1871751ea9b0b1a5b25004d`.

MD5 processes messages in 512-bit blocks, so, internally, the hash function pads this 160-bit string 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 counter of the number of bits in the unpadded message. (If the 1 and counter 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 a longer message consisting of `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, we can guess the length of *m* and run `bits = len(m + padding(len(m)*8))*8`.

The `pkmd5` module lets you specify these parameters as additional parameters to the `md5` object as follows:

```
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 `x = "Good advice"`. Simply run `h.update(x); print h.hexdigest()` to execute the compression function over `x` and output the resulting hash.

What value do you get? Verify that it equals the MD5 hash of `m + padding(len(m)*8) + x`.

(b) Length extension causes a serious vulnerability when people mistakenly try to create something like an HMAC by using *hash*(*secret* $\|$ *message*). For example, Professor Halderman has constructed 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://jhalderm.com/398app/api?token=c4f7850aa84d5b5f276f433e2724e8fe&user=jhalderm&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]).

Without guessing the password, apply length extension to create a valid URL that appends `&command3=DeleteAllFiles`. You can use the real server to check whether your result is accepted. Hint: You'll need to use the `quote()` function from Python's `urllib` module to encode non-ASCII characters in the URL.

Show your work along with the URL you constructed.

(In 2009, security researchers found that the photo-sharing site Flickr suffered from a length-extension vulnerability almost exactly like the one in this exercise.)

(c) With reference to the construction of HMAC, explain how changing the design in (b) to use `token` = $HMAC_{\textit{user's password}}$(`user=....`) would avoid the length extension vulnerability.

2. **Hash 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.

   (a) The first known collisions were announced August 17, 2004 by Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. Here's one pair of colliding messages they published:

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

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

   Convert each group of hex strings to a binary file.
   (On Linux, run `$ xxd -r -p file.hex > file`.)

   What are the MD5 hashes of the two binary files? Verify that they're the same.
   (`$ openssl dgst -md5 file1 file2`)

   What are their SHA-256 hashes? Verify that they're different.
   (`$ openssl dgst -sha256 file1 file2`)

   (b) In 2004, Wang's method took more than 5 hours to find a collision on a desktop PC. Since then, PCs have gotten faster, and attack techniques have gotten much more efficient. You can compute your own MD5 collisions using a tool written by Marc Stevens that uses a more advanced technique. You can download it here:

   Source: http://www.win.tue.nl/hashclash/fastcoll_v1.0.0.5_source.zip (requires Boost)
   Windows: http://www.win.tue.nl/hashclash/fastcoll_v1.0.0.5.exe.zip

   Generate your own collision with this tool. How long did it take?
   (`$ fastcoll -o file1 file2`)

   What are your files? (To get a hex dump, run `$ xxd -p file`.)
   What are their MD5 hashes? Verify that they're the same.
   What are their SHA-256 hashes? Verify that they're different.

   (c) 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. Run them and verify that they produce 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 (e.g., `print "I come in peace."`); the second should execute a pretend malicious payload (e.g., `print "Prepare to be destroyed!"`). Attach both programs to your submission.

(d) Briefly explain why the technique you explored in (c) 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.)

Extra Credit To reward everyone for their hard work so far this semester, Professor Halderman is holding a contest. The goal is to guess a number that he will announce in class after fall break. If you guess correctly, you'll earn one 24-hour extension that may be used on a future homework of your choice. Here are the rules:

1. In the first lecture after break, the professor will pick an integer in the range $[0, 63]$.
2. Prior to the selection, each student may register one guess. To keep everything fair, your guesses will be kept secret until the selection using the following procedure:
   (a) You'll create a PostScript document that, when printed, produces a single page that contains only the statement: *your_uniqname* `guesses` *n*
   (b) To register your guess, you'll email the MD5 hash of your document to `eecs398@umich.edu` before the professor announces his pick.
3. If your guess is correct, you can claim your extension by emailing your document 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 Professor Halderman if it will teach him not to use a Turing-complete printer control language with a hash function that suffers from collisions.)

□