Lab 1: Classical Cryptanalysis and Attacking Cryptographic Hashes

This lab is due on **February 7, 2016** at **11:59PM**, following the submission checklist below. Late submissions will be penalized according to course policy. Your writeup MUST include the following information:

1. List of collaborators (on all parts of the project, not just the writeup)
2. List of references used (online material, course nodes, textbooks, wikipedia,...)
3. Number of late days used on this assignment
4. Total number of late days used thus far in the entire semester

If any of this information is missing, at least 20% of the points for the assignment will automatically be deducted from your assignment. See also discussion on plagiarism and the collaboration policy on the course syllabus.

While we provide you with the tools to run this lab at any platform (Windows, Linux and Mac OS), I strongly recommend working at a Linux or Mac OS machine as it will make things considerably easier for you. The instructions given for the rest of the description of the lab are therefore explicitly for this case (unless otherwise noted).

Administration: This lab will be administered by Sophia Yakoubov.

Introduction

In Section 1 of this lab, you will break two classical ciphers: the substitution cipher and the Vigenere cipher.
In the rest of the lab, you will investigate vulnerabilities in widely used cryptographic hash functions, including length-extension attacks and collision vulnerabilities. In Section 2, 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 Section 3, you will use a cutting-edge tool to generate different messages with the same MD5 hash value (collisions). You will then investigate how that capability can be exploited to conceal malicious behavior in software.

Objectives:

- Understand some of the vulnerabilities of classical cryptography.
- 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.

1 Classical Cryptanalysis

1.1 Breaking the Substitution Cipher

The following text was encrypted with a substitution cipher. (Spaces and punctuation marks were preserved from the plaintext.)

```
THVVWCZTQ!
CZ WECQ YIAQQ, LFN JCII IVAHZ WEV DCPPVHVZYV MVWJVVZ VZYHLOWCFZ AZD ANWEVZWCYAWCFZ.
WECQ RVQQATV CQ VZYHLOWVD JCWE A BVHL CZQVYNHV VZYHLOWCFZ QYEVRV. CDVAIIL, AZ
VZYHLOWCFZ QYEVRV QEFNID AIIFJ FZIL ANWEFHCGVD OAHWCVQ, JEF XZFJ WEV XVL, WF HVAD
WEV RVQQATV. EFJVBVH, LFN SNQW HVAD WEV RVQQATV JCWEFNW XZFJCZT WEV XVL. EVZYV,
WEV VZYHLOWCFZ QYEVRV CQ CZQVYNHV.
```

Find the plaintext. Feel free to do this by hand, or with a computer program. You can find tables of English letter frequencies on the web.

1.2 Breaking the Vigenere Cipher

The one-time pad is a long sequence of random letters. These letters are combined with the plaintext message to produce the ciphertext. To decipher the message, a person must have a copy of the one-time pad to reverse the process. A one-time pad should be used only once (hence the name) and then destroyed. This is the first and only encryption algorithm that has been proven to be unbreakable.

To encipher a message, you take the first letter in the plaintext message and “add" it to the letter from the one-time pad. To decipher a message, you take the ciphertext and “subtract” the letter from the one-time pad. By “adding”, we mean the following:

Let A be the 0th letter in the alphabet. ... Let Z be the 25th letter in the alphabet. To “add” letter S (the 18th letter) and C (the 2nd letter), take:

$$18 + 2 \mod 26 = 20$$

which gives us U (the 20th letter). To “subtract” letter U (the 20th letter) and C (the 2nd letter), take:

$$20 - 2 \mod 26 = 18$$

which gives us S (the 18th letter).

Here’s a larger example:

plaintext	SECRETMESSAGE
one-time pad	CIJTHUUHMLFRU
ciphertext	UMLKLNGLEDFXY

Dr. X was not paying attention in her security class, and decided to reuse the same 5-letter “one-time pad” over and over to encipher her plaintext. The result is shown below. (We’ve retained the punctuation in the message to make codebreaking easier for you.)

```
TWT CMGWI ZJ TWT AIOEAP XO QT DICJGP MN IWPMR ETCWOCH, SSUHTD, TAETCW, ACS
PJFTREW, AVPTRSI JYVEPHZRAQAP WEPGNLEH PYH STXKYRTH, DLAAA YST QT GMOAPEID,
PCO RO LPCVACID WHPAW MSHJP, FUI JASN EGZFAQAP GAJHP, WUEEZVTTS MC OPIS SR
PUQMRBPEMOC, PYH PPGEMCJALVLN SPWCGXMMNV ISI PAPNI TD QP WEPGNLES, PYH TWT
AIRHDYW OG ISMNVH ES BT HPMZTS.
```

Submit your python program and the deciphered message. You can use whatever decipherment technique you want; feel free to read online to find ideas! (Just make sure to cite your sources!)

NOTE: It turns out that this way of repeating the use of a "one-time pad" is also known as a Vigenère cipher, after Blaise de Vigenere, who published his description of the cipher before the court of Henry III of France, in 1586. It has been used throughout history. For example, the Confederate States of America used the Vigenere cipher during the American Civil War. The Confederacy's messages were far from secret and the Union regularly cracked their messages. Throughout the war, the Confederate leadership primarily relied upon three key phrases as their "key": "Manchester Bluff", "Complete Victory" and, as the war came to a close, "Come Retribution".

2 Length Extension

In most applications, you should use message authentication codes (MACs) such as HMAC-SHA256 instead of plain cryptographic hash functions (e.g. the hash functions MD5, SHA-1, or SHA-256) in the "key concatenated with message construction", that is

$$H(k||m)$$

where H is the cryptographic hash function, k is the secret key, m is the message, and $||$ means concatenation. The above "key concatenated with message" construction is not a good MAC and not a good pseudorandom function. This is because cryptographic hashes sometimes fail to match our intuitive security expectations.

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 $b_{n+1}, b_{n+2}, \ldots$ that we want to add. This process is called length extension, and it can be used to attack many applications of hash functions.

2.1 Experiment with Length Extension in Python

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.
To experiment with the idea of length extension attacks, 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[1] `http://www.cs.bu.edu/~goldbe/teaching/HW55814/static/pymd5.py` and learn how to use it by running `$ pydoc pymd5` [2]. 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 + 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:

```
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()
```

[1] For this and the following url's **do not** copy-paste them in your browser; the tilde character will not paste correctly. Simply click on it to open it at your browser.

[2] For Windows machines make sure you add the path to `pydoc` to the environment variable PATH.

to execute the compression function over `x` and output the resulting hash. Verify that it equals the MD5 hash of `m + padding(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.

2.2 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 $hash(secret \parallel message)$. For example, Professor Goldberg 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://www.cs.bu.edu/~goldbe/teaching/HW55814/lab1/
    api?token=11ed1b5786c5fc4d4fa4294f4d281df1
  &user=sgoldberg&command1=ListFiles&command2=NoOp
```

where `token` is MD5(*user's 8-character password* $\parallel$ `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, the path 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 Python snippet:

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

# Your code to modify url goes here

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

3 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 or Mac OS, run `$ xxd -r -p file.hex > file`.)[3]

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

[3]For Windows machines you can download an executable port for command `xxd` from `http://ge.tt/5jfutZq/v/0`. Then just move it to your lab folder. Likewise for `openssl` command to work, you must first download OpenSSL for Windows -GNU32 version recommended.

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.

3.1 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 it here:

`http://www.cs.bu.edu/~goldbe/teaching/HW55814/static/fastcoll_v1.0.0.5_patched.zip`
(source; Linux or Mac; requires Boost)
or
`http://www.cs.bu.edu/~goldbe/teaching/HW55814/static/fastcoll_v1.0.0.5.win.zip`
(Windows executable)

Note that installing Boost –particularly for Mac OS– can be quite troublesome, therefore I suggest that, only for generating the files file1, file2 (and col1, col2 in the next section), you move to a Windows machine and run the executable. After that, resume work at a Linux or Mac OS machine.

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 on the four questions.

3.2 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`). I suggest that you run this step at a Linux or Mac OS machine, since the Windows equivalent of the `cat` command (`copy`) does not always behave ideally. 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!"`

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.

Fact: Sadly, it turns out that the MD5 "key concatenated with message" construction is sadly, still used all over the Internet in old network protocols that have not been updated since the 1990s. (More on this later in the course!) For instance, it is used to "authenticate" TCP connections, and to "authenticate" connections in the Network Time Protocol (NTP). However, modern network protocols no longer use this construction.

4 Writeup

With reference to the construction of HMAC, explain how changing the design of the API in Section 1.2 to use `token` $= HMAC_{\textit{user's password}}$(`user=....`) would avoid the length-extension vulnerability.

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

Submission Checklist

Upload to websubmit an archive file (`tar/.tar.gz/.zip/.rar`) named `lab1.`*`yourname.extension`*. The archive should contain only the following files:

- **(Part 1.1)** The plaintext corresponding to the ciphertext.
- **(Part 1.2)** The plaintext corresponding to the ciphertext, as well as a Python script used to find it.
- **(Part 2.2)** 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.
- **(Part 3.1)** A text file named `generating_collisions.txt` with your answers to the four short questions.
- **(Part 3.2)** Two Python scripts, named `good.py` and `evil.py`, that share an MD5 hash, have different SHA-256 hashes, and print the specified messages.
- **(Part 4)** A text file containing your answer, named `writeup.txt`.