Project 2: Cryptographic Attacks

This project is due on **Thursday, October 17, 2019** at **10 PM**. You will have a budget of five late days (24-hour periods) over the course of the semester that you can use to turn assignments in late without penalty and without needing to ask for an extension. You may use a maximum of two late days per assignment. Late pair projects will be charged to both partners. Once your late days are used up, extensions will only be granted in extraordinary circumstances.

This is a group project; you will work in **teams of two** and submit one project per team. If you have trouble forming a team, try Piazza's partner search forum.

The code and other answers your group submits must be entirely your own work, and you must adhere to the Code of Academic Integrity. 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 Canvas**, following the checklist at the end.

Introduction

In this project, you will investigate vulnerabilities in widely used cryptographic hash functions, including length-extension attacks, collision vulnerabilities, and an implementation vulnerability in a popular digital signature scheme. 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 cryptanalysis 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. In Part 3, you will learn about an attack against certain implementations of RSA padding; then, you will forge a digital signature using your own implementation of this attack.

Objectives:

- Understand how to apply basic cryptographic integrity and authentication 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.
- Understand why padding schemes are integral to cryptographic security.

Part 1. Length Extension

In 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, 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*. Many common hash functions 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.

1.1 Experiment with Length Extension in Python

To experiment with this idea, we will 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://cis.upenn.edu/~cis331/project2/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 will not 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 did not 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()
```

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 did not 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* ‖ *message*). The National Bank of CIS 331, 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://cis331.cis.upenn.edu/project2/api?token=0c6edcc81c7714b37a87cee7bb1f3d89&
user=aturing&command1=ListSquirrels&command2=NoOp
```

where `token` is MD5(*user's 10-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=UnlockAllSafes` 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 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 `UnlockAllSafes` 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 httplib, urlparse, sys
url = sys.argv[1]

# Your code to modify url goes here

parsedUrl = urlparse.urlparse(url)
conn = httplib.HTTPConnection(parsedUrl.hostname,parsedUrl.port)
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 is 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 compiling `fastcoll` from source, you can compile using this makefile:
`https://cis.upenn.edu/~cis331/project2/Makefile`. You can compile it on the VM supplied for Project 1, on Eniac, or on your own machine. In either case, you will need to have installed the Boost libraries. These should already be installed on Eniac. On Ubuntu (including the VM of project 1), you can install Boost 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 will 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 mean no harm."`. The second should execute a pretend malicious payload: `print "You are doomed!"`.

Common issue: If you receive the error "SyntaxError: EOF while scanning triple-quoted string literal", add a blank line at the beginning of the suffix and try again.

What to submit. Two Python 2 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. RSA Signature Forgery

A secure implementation of RSA encryption or digital signatures requires a proper padding scheme. RSA without padding, also known as *textbook RSA*, has several undesirable properties. For example, it is trivial for an attacker with only an RSA public key pair (n,e) to produce a mathematically valid message and signature pair by choosing an s and returning $(s^e \mod n, s)$. Here s^e would be the message, and s would be the signature.

In order to prevent an attacker from being able to forge valid signatures in this way, RSA implementations use a padding scheme to provide structure to the messages that are encrypted or signed. The most commonly used padding scheme in practice is defined by the PKCS #1 v1.5 standard, which can be found at `https://tools.ietf.org/html/rfc2313`. The standard defines, among other things, the format of RSA keys and signatures and the procedures for generating and validating RSA signatures.

3.1 Validating RSA Signatures

You can experiment with validating RSA signatures yourself. Create a file called `key.pub` that contains the following RSA public key:

```
-----BEGIN PUBLIC KEY-----
MFowDQYJKoZIhvcNAQEBBQADSQAwRgJBALB8X0rLPrfgAfXMW73LjKYb5V9QG5LU
DrmsA9CAittsLvh2c082wHwVyCIiWQ8S3AA/jfW839sFN4zAZkW2S3cCAQM=
-----END PUBLIC KEY-----
```

You can view the modulus and public exponent of this key by running:

```
$ openssl rsa -in key.pub -pubin -text -noout
```

Create a file containing only the text `CIS 331 rul3z!`.

```
$ echo -n CIS 331 rul3z! > message
```

The following is a base64-encoded signature of the file `message` using the private key that corresponds to the public key given above.

```
C+XuJ3pAF0p496uGTnqtMaCUTClnKHGsyoK5WjLBfnivIeGQjK1e6KabqdjLKJQ8
WsFrf0Wf/auH3K0Sprg2QQ==
```

Convert this signature into a binary file.

```
$ base64 -d -i sig.b64 > sig
```

Now verify the signature against the file you created.

```
$ openssl dgst -sha1 -verify key.pub -signature sig message
```

We can also use basic math operations in Python to explore this signature further. Remember, RSA ciphertexts, plaintexts, exponents, moduli, and signatures are actually all integers.

Open a Python shell and run the following commands to import the signature as an integer:

```
>>> from Crypto.PublicKey import RSA
>>> from Crypto.Hash import SHA
>>> signature = int(open('sig').read().encode('hex'),16)
```

Next, import the public key file that you created earlier:

```
>>> pubkey = RSA.importKey(open('key.pub').read())
```

The modulus and exponent are then accessible as `pubkey.n` and `pubkey.e`, respectively.
Now reverse the signing operation and examine the resulting value in hex:

```
>>> "%0128x" % pow(signature, pubkey.e, pubkey.n)
```

The ‘‘`%0128x`’’ string above simply specifies that the output should be 128 hex characters, and has nothing to do with RSA. You should see something like:
`‘0001fffff...f8c6ee82f9d0bca80b80f72a5337375c3d99695e’`.
Verify that the last 20 bytes of this value match the SHA-1 hash of your file:

```
>>> SHA.new("CIS 331 rul3z!").hexdigest()
```

This component is intended to introduce you to RSA signatures; you will not submit anything for it.

3.2 PKCS #1 v1.5 Signature Padding

The signed value you examined in the previous section had been padded using the PKCS #1 v1.5 signature scheme. PKCS #1 v1.5 padding for RSA signatures is structured as follows: one `00` byte, one `01` byte, some `FF` bytes, another `00` byte, some special ASN.1 bytes denoting which hash algorithm was used to compute the hash digest, then the bytes of the hash digest itself. The number of `FF` bytes varies such that the size of m is equal to the size of the RSA key.

A k-bit RSA key used to sign a SHA-1 hash digest will generate the following padded value of m:

$$\texttt{00}\quad\texttt{01}\quad\underbrace{\texttt{FF}\cdots\texttt{FF}}_{k/8-38\text{ bytes wide}}\quad\texttt{00}\ \underbrace{\texttt{3021300906052B0E03021A05000414}}_{\text{ASN.1 “magic” bytes}}\quad\underbrace{\texttt{XX}\cdots\texttt{XX}}_{\text{20-byte SHA-1 digest}}$$

When PKCS padding is used, it is important for implementations to verify that every bit of the padded, signed message is exactly as it should be. It is tempting for an implementer to validate the signature by first stripping off the `00 01` bytes, then some number of padding `FF` bytes, then `00`, and then parse the ASN.1 and verify the hash. If the implementation does not check the length of the `FF` bytes and that the hash is in the least significant bits of the message, then it is possible for an attacker to forge values that pass this validation check.

This possibility is particularly troubling for signatures generated with $e = 3$. If the length of the required padding, ASN.1 bytes, and hash value is significantly less than $n^{1/3}$ then an attacker can construct a cube root over the integers whose most significant bits will validate as a correct signature, ignoring the actual key. To construct a “signature” that will validate against such implementations, an attacker simply needs to construct an integer whose most significant bytes have the correct format, including the hashed message, pad the remainder of this value with zeros or other garbage that will be ignored by the vulnerable implementation, and then take a cube root over the integers, rounding as appropriate.

Historical fact: This attack was discovered by Daniel Bleichenbacher, who presented it in a lightning talk at the rump session at the Crypto 2006 conference. His talk and more details can be found in this posting: `https://www.ietf.org/mail-archive/web/openpgp/current/msg00999.html`. At the time, many important implementations of RSA signatures were discovered to be vulnerable to this attack, including OpenSSL. In 2014, the Mozilla library NSS was found to be vulnerable to this type of attack: `https://www.mozilla.org/security/advisories/mfsa2014-73/`.

This component is intended to introduce you to PKCS padding; you will not submit anything for it.

3.3 Constructing Forged Signatures

The National Bank of CIS 331 has a website at `http://cis331.cis.upenn.edu/project2/bank` that its employees use to initiate wire transfers between bank accounts. To authenticate each transfer request, the control panel requires a signature from a particular 2048-bit RSA key that is listed on the website's home page. Unfortunately, this control panel is running old software that has not been patched to fix the signature forgery vulnerability.

Using the signature forgery technique introduced by Bleichenbacher, produce an RSA signature that validates against the National Bank of CIS 331 site.

What to submit. A Python 2 script called `bleichenbacher.py` that:

1. Accepts a double-quoted string as a command-line argument (e.g., "cis331+jdoe+1.23").
2. Prints a base64-encoded forged signature of the input string.

You have our permission to use the control panel at `http://cis331.cis.upenn.edu/project2/bank` to test your signatures. We have provided a Python library, `roots.py`, that provides several useful functions that you may wish to use when implementing your solution. You can download `roots.py` at `https://cis.upenn.edu/~cis331/project2/roots.py`. Your program can assume that the PyCrypto library and `roots.py` are available, and may use standard Python libraries, but should otherwise be self-contained.

In order to use these functions, you will have to import `roots.py`. You may wish to use the following template:

```
from roots import *
from Crypto.Hash import SHA
import sys

message = sys.argv[1]

# Your code to forge a signature goes here.

root, is_exact = integer_nthroot(27, 3)
print integer_to_base64(root)
```

Part 4. Writeup

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

3. Since 2010, NIST has specified that RSA public exponents must be at least $2^{16}+1$. Briefly explain why Bleichenbacher's attack would not work for these keys.

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

Submission Checklist

Upload to Canvas a gzipped tarball (`.tar.gz`) named `project2.`*`pennkey1`*`.`*`pennkey2`*`.tar.gz`. The tarball should contain only the following files. **These will be autograded, so make sure you submit with the proper names and behaviors.** Do not make your files dependent on local files or esoteric libraries.

Section 1.2

`len_ext_attack.py`: A Python script which accepts a URL as input, performs the specified attack on the web application, and outputs the server's response.

Section 2.1

`generating_collisions.txt`: A text file with your answers to the four short questions.

Section 2.2

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

Section 3.3

`bleichenbacher.py`: A Python 2 script that takes a string argument and outputs a signature forged using Bleichenbacher's attack.

Writeup

`writeup.txt`: A text file containing your answers to the three wrap-up questions.

Bonus Challenge [Extra Credit]

Generate a digital signature for the sentence "My name is <your name>." that verifies correctly using OpenSSL with the following 1024-bit RSA public key. (Hint: The modulus might not have been generated like a normal RSA modulus.):

```
-----BEGIN PUBLIC KEY-----
MIGdMA0GCSqGSIb3DQEBAQUAA4GLADCBhwKBgQCC98+XVTxNUDsTutF6jNKjc4jd
kLng6gAKlDd1xg5lhwl2NPqb6VYUIXDsV3Q5hZs6N1xyLkuHAtq4mMETuJG8CC8y
Tqw2j9Yt5xPdbX9WMYqr6mIHiC1GN9Jk5S6sIW+8pCUYhCQWG6yg1sB9cj5aY5UP
2iW6LawXSLUHj2m9JQIBAw==
-----END PUBLIC KEY-----
```