Project 1: Cryptography

This homework is due **Wednesday, February 8, 2023** at **11:59 p.m.** and counts for 10% of your course grade. Late submissions will be penalized by 10% plus an additional 10% every 5 hours until received. Late work will not be accepted after 24 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 may work in **teams of at most two** and submit **one** project per team. Please find a partner as soon as possible! If you have difficulties forming a team, post to Piazza's *Search for Teammates* forum. The final exam will cover project material, so you and your partner should collaborate on each part. **Failure to follow assignment instructions (e.g., submitting wrong or non-working code, improper formatting, etc.) will be ineligible for regrade requests**.

The code and other answers your group submits must be entirely your own work, and you are bound by the University's Student 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., in your code comments).

Introduction

In this project, you will investigate vulnerabilities in widely used cryptographic hash functions, including length-extension attacks and 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 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. 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 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 $b_{n+1}, b_{n+2}, \ldots$ 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 at `http://cs4440.eng.utah.edu/files/project1/pymd5.py` and learn how to use it by running `$ python3 -m 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 + 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", 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 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 Central Bank of CS 4440, 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://cs4440.eng.utah.edu/project1/api?token=402a574d265dc212ee64970f159575d0
&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=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.

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

What to submit: A Python 3.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 (i.e., `argv[1]`).
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` (although they will remain in the same order). 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 http.client as httplib
from urllib.parse import urlparse, quote
import sys, re
from pymd5 import md5, padding
url = sys.argv[1]

# Your code to modify the URL goes here!

parsedUrl = 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'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://cs4440.eng.utah.edu/files/project1/fastcoll.exe.tar.gz`
  (pre-built executable, for Windows)

- `http://cs4440.eng.utah.edu/files/project1/fastcoll.src.tar.gz`
  (source code, for Linux and macOS)

If you are building `fastcoll` from source, we have provided a script `install_boost.sh` to set up the necessary dependencies. On macOS, you will first need to install `wget` via the Homebrew package manager using `brew install wget`.

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. As this part is autograded, **make sure to format your answers using the following template:**

```
# 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. *prefix* || $blob_A$ || *suffix* and *prefix* || $blob_B$ || *suffix*.

We can leverage this to create two programs that have identical MD5 hashes but exhibit wildly different behaviors. We will use Python, but almost any language would do. Start by putting the following **four** lines (note the three spaces required on the third line) into a file called `prefix`:

```
#!/usr/bin/env python3
# coding: iso-8859-1
< Line #3 should be just three spaces. >
MSG = bytes(r"""
```

Then, put the following **four** lines (note the empty line at the beginning) into a file called `suffix`:

```
< Line #1 should be completely empty. >
""", "iso-8859-1")
from hashlib import sha256
print(sha256(MSG).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. You may have to re-run fastcoll a few times as it occasionally generates data that conflicts with Python's string termination.

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 3.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. 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. One property is that it is trivial for an attacker with only an RSA public key pair (n, e) to produce a mathematically valid $(\text{message}, \text{signature})$ pair by choosing an s and returning (s^e, s).

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 values 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 text file called `key.pub` that contains the following RSA public key:

```
-----BEGIN PUBLIC KEY-----
MFowDQYJKoZIhvcNAQEBBQADSQAwRgJBALvMZ+YhjwKpteNYzzbPHvPqdvMrs2Rf
HeIhK+ui9v0YHNyFW6aBwwG/6sfbvxx40lePBWjRhpoxDC5A/J+rVXkCAQM=
-----END PUBLIC KEY-----
```

Confirm that the key has a 512-bit modulus with an exponent of 3. 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 `CS 4440 rul3z!` (`$ echo -n 'CS 4440 rul3z!' > myfile`). The following is a base64-encoded signature of the file using the private key corresponding to the public key above.

```
H91NNINrWyDFL8hEJFYV0G237A90luqr2EqFKZUicV0LzVtmvSeJC
jVmY5MxuJ33zCQ1iShZm2Yd7DRrKBLDSw==
```

Copy the base64-encoded signature to a file, and then convert it from base64 to raw binary. (`$ base64 --decode -i sig.b64 > sig`). Verify the signature against the file you created:

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

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.

Usually, you would use a cryptography library to import a public key. However, for the purposes of this part of the assignment, you can just manually assign the modulus and exponent as integers in Python based on the earlier output from OpenSSL. You may find the following command useful:

```
$ openssl rsa -in key.pub -text -noout -pubin | egrep '^ ' | tr -d ' :\n'
```

Launch Python in interactive mode and assign the modulus and the exponent to integer variables:

```
# n is the modulus from the key.
# You can just assign it as a hexadecimal literal--remember to start with 0x
# It will look something like:
n = 0x00d56d87ba372303f8...104dd059e49e435f

# e is the exponent from the key
e = 3
```

We can also load the signature into Python. Like the modulus and the exponent, we'll convert the signature to an integer:

```
signature = int.from_bytes(open('sig', mode='rb').read(), byteorder='big')
```

Now reverse the signing operation by computing $\mathtt{signature}^{\mathtt{e}}$ mod `n`:

```
x = pow(signature, e, n)
```

You can print the resulting value as a 64-byte (512-bit) integer in hex:

```
print("%0128x" % x)
```

You should see something like `0001fffff...ccc66fd5f58739932f0585d3e8362`. Verify that the last 20 bytes of this value match the SHA-1 hash of your file:

```
import hashlib
m = hashlib.sha1()
m.update("CS 4440 rul3z!".encode())
print(m.hexdigest())
```

The hash has been padded using the PKCS #1 v1.5 signature scheme, which specifies that, for a SHA-1 hash with a *k*-bit RSA key, the signed value will contain the following bytes:

$$\mathtt{00\ 01}\ \underbrace{\mathtt{FF\ FF\ FF\ \cdots\ FF}}_{k/8-38\text{ bytes}}\ \mathtt{00}\ \underbrace{\mathtt{30\ 21\ 30\ 09\ 06\ 05\ 2B\ 0E\ 03\ 02\ 1A\ 05\ 00\ 04\ 14}}_{\text{ASN.1 “magic” bytes denoting type of hash algorithm}}\ \underbrace{\mathtt{XX\ XX\ XX\ XX\ \cdots\ XX}}_{\text{SHA-1 digest (20 bytes)}}$$

The number of `FF` bytes varies such that the size of the result is equal to the size of the RSA key. Confirm that the value you computed above matches this format. It is crucial for implementations to verify that *every bit* is exactly as it should be, but sometimes developers can be lazy…

3.2 Bleichenbacher's Attack

It's tempting for an implementer to validate the signature padding as follows: (1) confirm that the total length equals the key size; (2) strip off the bytes 00 01, followed by *any number* of FF bytes, then 00; (3) parse the ASN.1 bytes; (4) verify that the next 20 bytes are the correct SHA-1 digest.

This procedure does not check the length of the FF bytes, nor does it verify that the hash is in the least significant (rightmost) bytes of the string. As a result, it will accept malformed values that have "garbage" bytes following the digest, like this example, which has only one FF:

$$\texttt{00 01 } \mathbf{\texttt{FF}} \texttt{ 00 } \underbrace{\texttt{30 21 30 09 06 05 2B 0E 03 02 1A 05 00 04 14}}_{\text{ASN.1 ``magic'' bytes denoting type of hash algorithm}} \; \underbrace{\texttt{XX XX XX} \cdots \texttt{XX}}_{\text{SHA-1 digest (20 bytes)}} \; \underbrace{\texttt{YY YY YY YY} \cdots \texttt{YY}}_{k/8-39\text{ arbitrary bytes}}$$

Convince yourself that this value would be accepted by the incorrect implementation described above, and that the bytes at the end labeled YY would be ignored. When an implementation uses this lenient, incorrect parsing, an attacker can easily create forged signatures that it will accept.

This possibility is particularly troubling when RSA is used with $e = 3$. Consider the case with RSA encryption: If we encrypt an unpadded message m that is much shorter than k-bits, then $m^3 < n$. Thus, the "encrypted" message $c = m^e = m^3 \bmod n$ does not "wrap around" the modulus n. In this case, RSA doesn't provide good security, since the attacker can simply take the normal cube root of the ciphertext to find the plaintext, $m = c^{1/3}$. It's easy to reverse normal exponentiation, as opposed to modular exponentiation!

Now recall that RSA signature validation is analogous to RSA encryption. If the signature uses $e = 3$, the validator calculates $s^e = s^3 \bmod n$ and checks that the result is the correct PKCS-padded digest of the signed message.

Here comes the attack: For a 2048-bit key, a correctly padded value for an RSA signature using a SHA-1 hash should have $k/8 - 38 = 2048/8 - 38 = 218$ bytes of FFs. But what if there were only one FF, as in the example shown above? This would leave space for 217 arbitrary bytes at the end of the value. A correct implementation of signature validation would notice the extra bytes and reject the signature, but the weak implementation described above would ignore these bytes.

To construct a signature that would validate against such an implementation, the attacker needs to find a number x such that $x^3 < n$ and where x^3 matches the format of the malformed example shown above. Since the implementation ignores the last 217 bytes, the attacker has enough flexibility to find a perfect cube of this form. To do this, you can construct an integer whose most significant bytes have the correct format, including the digest of the target message, and set the last 217 bytes to 00. Then take the cube root, 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 is described in this mailing list post: https://mailarchive.ietf.org/arch/msg/openpgp/5rnE9ZRN1AokBVj3VqblGlP63QE. At the time, many important implementations of RSA signatures were found to be vulnerable to this attack, including OpenSSL. In 2014, the Firefox TLS implementation was also found to be vulnerable to this type of attack: https://www.mozilla.org/security/advisories/mfsa2014-73/.

3.3 Constructing Forged Signatures

The Central Bank of CS 4440 has a website at http://cs4440.eng.utah.edu/project1 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, unpatched software that is vulnerable to signature forgery.

Using the signature forgery technique described above, produce an RSA signature that the Central Bank of CS 4440 site accepts as valid.

What to submit: A Python 3.x script called `bleichenbacher.py` that:

1. Accepts a double-quoted string as a command-line argument.
2. Prints a base64-encoded forged signature of the input string.

You have our permission to use http://cs4440.eng.utah.edu/project1 to test your signatures, but when we grade your program it will not have access to the network.

We have provided a Python library that includes several useful functions you may wish to use when implementing your solution: http://cs4440.eng.utah.edu/files/project1/roots.py.
To examine its documentation, run `$ python3 -m pydoc roots`.

To use these functions, you will have to import `roots.py`. You can start with the following template:

```
from roots import *
import hashlib
import sys
message = sys.argv[1]

# Your code to forge a signature goes here!

print(integer_to_base64(forged_signature))
```

Submission Instructions

Upload to **Canvas** a gzipped tarball (`.tar.gz`) named `project1.`*`uid1`*`.`*`uid2`*`.tar.gz` that contains only the files listed below. **These will be autograded, so make sure that your solutions conform to the expected filenames, formatting, and behaviors.** You can generate the tarball at the shell using this command:

```
tar -zcf project1.uid1.uid2.tar.gz len_ext_attack.py \
   generating_collisions.txt good.py evil.py bleichenbacher.py
```

To aide in formatting, we provide the following reference template for you to fill in:
`http://cs4440.eng.utah.edu/files/project1/project1.uid1.uid2.tar.gz`

Part 1.2

`len_ext_attack.py`: A Python 3.x script that accepts a URL as a command-line argument, performs the specified attack, and outputs the server's response.

Part 2.2

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

Part 2.3

`good.py` and `evil.py`: Two Python 3.x scripts that share an MD5 hash, have different SHA-256 hashes, and print the specified messages.

Part 3.3

`bleichenbacher.py`: A Python 3.x script that accepts a string as a command-line argument and outputs a forged signature for that string that is considered valid by the bank website.