Project 1: Cryptographic Attacks

This project is due on **Thursday, September 26** at **6 p.m.** and counts for 8% of your course grade. Late submissions will be penalized by 10% of the maximum attainable score, plus an additional 10% every 4 hours until received. Late work will not be accepted after the start of the next lab (of any section) following the day of the deadline. If you have a conflict due to travel, interviews, etc., please plan accordingly and turn in your project early.

You will work in **teams of two** and submit one project per team. Please find a partner as soon as possible. If you have trouble forming a team, post to Piazza's partner search forum. The final exam will cover project material, so you and your partner should collaborate on each part.

The code and other answers your group submits must be entirely your own work, and you are bound by the Honor 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., 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, an implementation vulnerability in a popular digital signature scheme, and a side channel attack. 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. In Part 4, you will demonstrate how CBC mode decryption can be exploited to decrypt a message without the attacker knowing the key. In Part 5, you will be asked to answer several questions related to the concepts in the rest of the project.

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 `https://eecs388.org/static/project1/pymd5.py` and learn how to use it by running `$ pydoc pymd5`. 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, 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:

```
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* ‖ *message*)[1]. The National Bank of EECS 388, 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://eecs388.org/project1/api?token=9e67bf39503090c7f3aff418ff01543f`
`&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 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` (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 requests, urllib, sys, codecs

url = sys.argv[1]

# Your code to modify url goes here

response = requests.get(url)
print(response.content)
```

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://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 this makefile: `https://eecs388.org/static/project1/Makefile`. You will also need the Boost libraries. 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. *prefix* ‖ *blob$_A$* ‖ *suffix* and *prefix* ‖ *blob$_B$* ‖ *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: latin-1
blob = """
```

and put these three lines into a file called `suffix`:

```
"""
from hashlib import sha256
print(sha256(blob.encode("latin-1")).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("Hashing is not encryption!")` The second should execute a pretend malicious payload: `print("Security through obscurity!")`. Note that we may rename these program before grading them.

What to submit   Two Python 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 (message, 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/6sfbvxx4OlePBWjRhpoxDC5A/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 EECS 388 rul3z! (`$ echo -n 'EECS 388 rul3z!'  > myfile`). The following is a base64-encoded signature of the file using the private key corresponding to the public key above.

```
cEnO8WBePTdtCV6DXudzecMMl60N2cZdJjoIZCK/ir/zsuF1GPFp1yoLrA/NO2LZ
FZOC4IpP5tjyDeYizYqoRg==
```

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 -sha256 -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 = 0x00bbcc67e6218f02a9...0c2e40fc9fab5579

# 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 = open('sig.b64').read()
# decode and convert to hexadecimal integer
signature = int(codecs.encode(base64.b64decode(signature), "hex"), 16)
```

Now reverse the signing operation by computing $\texttt{signature}^e$ 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...9baf336a45e7a58f5fddbdc8ba2fa`. Verify that the last 32 bytes of this value match the SHA-256 hash of your file:

```
import hashlib
m = hashlib.sha256()
m.update("EECS 388 rul3z!".encode("latin-1"))
print(m.hexdigest())
```

The hash has been padded using the PKCS #1 v1.5 signature scheme, which specifies that, for a SHA-256 hash with a k-bit RSA key, the signed value will contain the following bytes:

$$\underbrace{\texttt{00 01}}\ \underbrace{\texttt{FF FF FF}\ \cdots\ \texttt{FF}}_{k/8-54\ \text{bytes}}\ \underbrace{\texttt{00}}\ \underbrace{\texttt{30 31 30 0d 06 09 60 86 48 01 65 03 04 02 01 05 00 04 20}}_{\text{ASN.1 "magic" bytes denoting type of hash algorithm}}\ \underbrace{\texttt{XX XX XX XX}\ \cdots\ \texttt{XX}}_{\text{SHA-256 digest (32 bytes)}}$$

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 32 bytes are the correct SHA-256 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`:

$$00\ 01\ \underbrace{\mathtt{FF}}\ 00\ \underbrace{30\ 31\ 30\ 0d\ 06\ 09\ 60\ 86\ 48\ 01\ 65\ 03\ 04\ 02\ 01\ 05\ 00\ 04\ 20}_{\text{ASN.1 "magic" bytes denoting type of hash algorithm}}\ \underbrace{\mathtt{XX\ XX\ XX}\ \cdots\ \mathtt{XX}}_{\text{SHA-256 digest (32 bytes)}}\ \underbrace{\mathtt{YY\ YY\ YY\ YY}\ \cdots\ \mathtt{YY}}_{k/8-55 \text{ arbitrary bytes}}$$

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-256 hash should have $k/8 - 54 = 2048/8 - 54 = 202$ bytes of `FF`s. But what if there were only one `FF`, as in the example shown above? This would leave space for 201 arbitrary bytes at the end of the value. 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. 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 201 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 National Bank of EECS 388 has a website at `https://eecs388.org/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 National Bank of EECS 388 site accepts as valid.

What to submit   A Python 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 `https://eecs388.org/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, `roots.py`, that provides several useful functions that you may wish to use when implementing your solution. You can download `roots.py` at `https://eecs388.org/static/project1/roots.py`. To examine its documentation, run `$ 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))
```

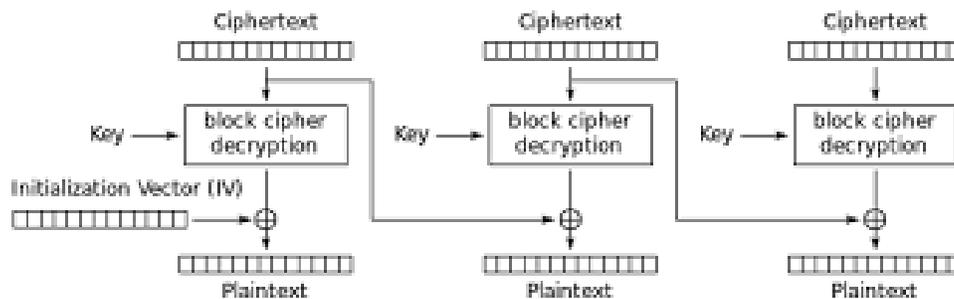# Part 4. Padding Oracle Attack

4.1 Padding Oracle

Figure 1: CBC Mode Decryption

Cipher-Block Chaining, or CBC, is a block cipher mode of encrypting variable length inputs. When encrypting with CBC mode, each plaintext block is XORed with the previous ciphertext block before being encrypted. This also means that when decrypting, each block of plaintext is generated by being XORed with the previous block of ciphertext, as seen in Figure 1.

In order for the message to be a multiple of the block length, CBC mode uses a padding scheme. *PKCS7* defines a standard for padding the message to the necessary length, in which the final block is filled with *B* bytes with the value *B*. For example, if the block size is 16 bytes and your message only fills 12 bytes of the last block, the final 4 bytes will be padded with (04, 04, 04, 04).

A Padding Oracle attack is possible when a system indicates whether the padding of a message is valid before checking the validity of the message. If an attacker knows the padding scheme, they can manipulate an intercepted ciphertext until a padding error does not occur, allowing them to determine, byte-by-byte, what the plaintext contains *without knowing the key!*

4.2 Conducting a Padding Oracle Attack

In our relentless pursuit of ensuring global cyber justice we've discovered a website containing dead drop messages used by spies.

Our preliminary analysis has indicated that there are two endpoints. `https://eecs388.org/ paddingoracle/` contains a list of messages with their corresponding sender and receiver. We believe these messages are generated using AES128 with CBC mode with pre-shared keys between the sender and receiver. `https://eecs388.org/paddingoracle/verify` appears to be a way for distrustful spies to verify that the message is from the sender and to the receiver. Your task is to correctly decipher all three messages.

Agent Kerckhoffs has conducted some preliminary analysis that may be useful. He has written some pseudocode representing what we suspect the server does when the `/verify` endpoint is accessed:

```
def MAC(m, from, to):
    tag = ":" + from + ":" + to
    return tag

def Enc(m, from, to, k):
    m_and_mac = m || MAC(m, from, to)
    padding = PKCS_7(m_and_mac)
    x = m_and_mac || padding
    # They are using block size of 16 bytes
    # AES_128_CBC prepends the IV to the front of the ciphertext
    c = encryptor_AES_128_CBC(k, x)
    return IV || c

def Dec(c, k):
    try:
        # if this encounters bad padding, it throws an exception
        m = decryptor_AES_128_CBC(k, c)
        return m
    except BadPaddingException as bad_pad:
```

```
        throw bad_pad

def Verify_mac(c, from, to, k):
    try:
        m_and_mac = Dec(c, k)
        Mac = Get_Mac_From_Full(m_and_mac)
        if (Mac != ":" + from + ":" to):
            return "Invalid Validation"
        else:
            return "Valid Validation"
    except BadPaddingException:
            return "Error 400: Invalid Padding"
    except: # if anything else is wrong
            return "Error 400: Corrupt Message"

@app.route('/verify', methods=['GET'])
def dec_oracle_route():
    ciphertext = request.args.get("message")
    from_name = request.args.get("from")
    to_name = request.args.get("to")
    return Verify_mac(ciphertext, from_name, to_name, SHARED_KEY)
```

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

1. Accepts four double-quoted strings as arguments: `padding_oracle_url`, `from`, `to`, and `base16_ciphertext`

2. Prints **only** the decoded message in human-readable text (not base 16). Printing anything else will result in losing formatting points.

and a text file called `plaintext.txt` that contains the three deciphered messages, one per line. You have our permission to use `https://eecs388.org/paddingoracle/verify` to decipher the messages. Your Python script may make network connections to this endpoint.

Start Early!   This part of the project may result in the padding oracle server being congested. Near the deadline, the server may slow down and this will not be a reason for a deadline extension.

Part 5. Writeup

1. With reference to the construction of HMAC, explain how changing the design of the API in Part 1.2 to use $\text{token} = \text{HMAC}_{user's\ password}(\text{user=}\dots)$ would avoid the length extension vulnerability.

2. Briefly explain why the technique you explored in Part 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 (e) should be at least $2^{16} + 1$. Briefly explain why Bleichenbacher's attack would not work for these keys.

4. As you demonstrated in Part 4, a padding oracle attack is an attack that exploits the padding of encrypted data in order to decrypt the message without having the key.

    (a) What is the time complexity of a padding oracle attack on an AES-CBC encryption scheme in big-O notation? Is this concerning?

    (b) Say the people who designed the system you are trying to attack against also took 388 and know about the padding oracle attack. They decide to only send a generic error message whenever something goes wrong in a request, whether that be an incorrect key, incorrect padding, or any other error. What side channel could you exploit that will allow you to continue to use their system as a padding oracle?

    (c) Let's say you are a cunning cryptographer and decide that you are going to build a system with a padding scheme that is secret. Will this stop the padding oracle attack or not? What principle supports or disproves this claim?

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

Submission Checklist

At the deadline, you should have the following files pushed to your GitHub project repository. Some of the files will be generated with some starter code when the repository is created. Carefully follow the instructions on Piazza for creating project groups and for managing late submissions.

Part 1.2

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

Part 2.1

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

Part 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.

Part 3.3

`bleichenbacher.py`: A Python 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.

Part 4

`padding_oracle.py`: A Python script that accepts the sender, receiver, and message and prints the decrypted message.
`plaintext.txt`: A text file containing the three decrypted messages, one per line.

Part 5

`writeup.txt`: A text file containing your answers to the four writeup questions.