Project 3: Cryptography

This project is split into two parts, with the first checkpoint due on **March 4th, 2019** at **6:00pm** and the second checkpoint due on **March 13th, 2019** at **6:00pm**. The first checkpoint is worth 20 points, and the second checkpoint is worth 100 points. We strongly recommend that you get started early.

This is a group project; we strongly encourage you work in **teams of two**. You MUST submit ONLY one project per team. If you have trouble forming a team, post to Piazza's partner search forum.

The code and other answers your group submits MUST be entirely your own work, and you are bound by the Student Code. You MAY consult with other students about the conceptualization of the project and the meaning of the questions, but you MUST 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 in only one of the group member's svn directory, following the submission checklist given at the end of each checkpoint. Details on the filename and submission guideline is listed at the end of the document.

"Anyone, from the most clueless amateur to the best cryptographer, can create an algorithm that he himself can't break."

– Bruce Schneier

Introduction

In this project, you will be using cryptographic libraries to decrypt multiple types of ciphers, break them, and launch attacks on widely used cryptographic hash functions. In 3.1.2, you will be decrypting ciphers with given ciphertexts and key values. Then, you will use the same technique to break a weak cipher with a limited key space. In 3.1.3, you will start out with a small exercise that uses a hash function to observe the avalanche effect, and then build a weak hash algorithm and find a collision on a given string. In 3.2.1, we will guide you through attacking the authentication functionality of an imaginary server API. The attack will exploit the length-extension vulnerability of hash functions in the MD5 and SHA family. In 3.2.2, 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 3.2.3, you will be performing a padding oracle attack using AES-encrypted ciphertext. In 3.2.4, you will implement another attack on RSA. Last, in 3.2.5, you will use a collision attack to undermine the integrity of a set of certificate documents.

Objectives:

- Become familiar with existing cryptographic libraries and how to utilize them
- Understand pitfalls in cryptography and appreciate why you *should not write your own cryptographic libraries*
- Execute a classic cryptographic attack on MD5 and other broken cryptographic algorithms
- Appreciate why you should use HMAC-SHA256 as a substitute for common hash functions

Guidelines

- You SHOULD work in a group of 2.
- You MUST use Python 2.7.
- Your answers may or may not be the same as your classmates'.
- All the necessary files to start the project will given under the folder called "Crypto" in your git repository. We've also generated some empty files in which you will submit your answers. You MUST submit your answers in the provided files; we will only grade what's there!

3.1 Checkpoint 1 (20 points)

3.1.1 Python tutorial

In this section, you will be writing several python scripts to do string encoding and manipulations needed to correctly read our input files and submit your answers.

3.1.1.1 Reading .hex files

In the later parts of this MP, you will be reading in .hex files, which are plaintext files containing an ASCII string representation of a single hexadecimal number. This is the content of an example .hex file:

```
3dab821d92b5ca7f48beee066996b8abc82f7e5646a0561710ea5bc11c80d
```

The following Python code snippet will read the contents of the file as a string and store it in `file_content`:

```
# strip() remove any leading or trailing whitespace characters
with open('file_name') as f:
    file_content = f.read().strip()
```

From here, there's a number of things that you could do. Depending on the cryptographic library that you are using, you may need to use different data types, but here we list the most common conversions that you may need:

```
# parse the string into a binary array representing the hexadecimal number
binary_content = file_content.decode('hex')

# parse the string into integer
integer_parsed = int(file_content,16)

# parse an integer to a hex string and remove the leading '0x'
str = hex(integer_parsed)[2:]

# parse an integer to a binary string and remove the leading '0b'
str = bin(integer_parsed)[2:]
```

3.1.1.2 Exercise (2 points) (*Difficulty: Easy*)

Files

1. `3.1.1.2_value.hex`: an ASCII string representing a hexadecimal value

Based on what you learned in the previous section, we want to to convert the given value into different representations and submit them in the specified files.

What to submit

1. Convert the value in `3.1.1.2_value.hex` to decimal and submit the decimal number as a string in `sol_3.1.1.2_decimal.txt`
2. Convert the value in `3.1.1.2_value.hex` to binary and submit the binary number as a string in `sol_3.1.1.2_binary.txt`

3.1.2 Symmetric Encryption, Public Key Encryption, and Cryptographic Hashes

In this section, you will be writing your own cryptographic library to decrypt a substitution cipher, and using existing cryptographic libraries to experiment with a symmetric encryption called AES and a public key encryption called RSA.

3.1.2.1 Substitution Cipher (3 points) (*Difficulty: Easy*)

Files

1. `3.1.2.1_sub_key.txt`: key
2. `3.1.2.1_sub_ciphertext.txt`: ciphertext

`sub_key.txt` contains a permutation of the 26 uppercase letters that represents the key for a substitution cipher. Using this key, the *i*th letter in the alphabet in the plaintext has been replaced by the *i*th letter in `3.1.2.1_sub_key.txt` to produce ciphertext in `3.1.2.1_sub_ciphertext.txt`. For example, if the first three letters in your `3.1.2.1_sub_key.txt` are `ZDF`..., then all `A`s in the plaintext have become `Z`s in the ciphertext, all `B`s have become `D`s, and all `C`s have become `F`s. The plaintext we encrypted is a clue from the game show Jeopardy and has only uppercase letters, numbers and spaces. Numbers and spaces in the plaintext were not encrypted; they appear exactly as they did in the plaintext. Your task is to write a Python script in `sol_3.1.2.1.py` that decrypts a substitution ciphertext with a given key and writes the plaintext to a specified file. Your script must take three arguments from the command line: the ciphertext file, the key file, and the output file. We will run your script as follows:

```
$ python your_script.py ciphertext_file key_file output_file
```

Additionally, you have to submit the plaintext, which is obtained by using the key `3.1.2.1_sub_key.txt` to decrypt `3.1.2.1_sub_ciphertext.txt`, in the file `sol_3.1.2.1.txt`.

What to submit Your Python script in `sol_3.1.2.1.py` and your plaintext in `sol_3.1.2.1.txt`

3.1.2.2 AES: Decrypting AES (3 points) (*Difficulty: Easy*)

Files

1. `3.1.2.2_aes_key.hex`: key
2. `3.1.2.2_aes_iv.hex`: initialization vector
3. `3.1.2.2_aes_ciphertext.hex`: ciphertext

`3.1.2.2_aes_key.hex` contains a 256-bit AES key represented as an ASCII string of hexadecimal values. `3.1.2.2_aes_iv.hex` contains a 128-bit initialization vector in a similar representation. We encrypted a Jeopardy clue using AES in CBC mode using this key and initialization vector and wrote the resulting ciphertext (also stored in hexadecimal) to `3.1.2.2_aes_ciphertext.hex`. Create a Python script named `sol_3.1.2.2.py` that decrypts the ciphertext using the provided information and outputs the plaintext to a specified file. Your script must take four arguments from the command line: the ciphertext file, the key file, the initialization vector file, and the output file. We will run your script as follows:

```
$ python your_script.py ciphertext_file key_file iv_file output_file
```

Cryptographic Library

For this checkpoint, we recommend PyCrypto, an open-source crypto library for python. PyCrypto can be installed using pip with `sudo pip install pycrypto` or by going to their website at `https://www.dlitz.net/software/pycrypto/`.

What to submit Your Python script in `sol_3.1.2.2.py` and the decrypted message in `sol_3.1.2.2.txt`.

3.1.2.3 AES: Breaking A Weak AES Key (3 points) (*Difficulty: Easy*)

Files

1. `3.1.2.3_aes_weak_ciphertext.hex`: ciphertext

As with the last task, we encrypted a Jeopardy clue using 256-bit AES in CBC mode and stored the result in hexadecimal in the file `3.1.2.3_aes_weak_ciphertext.hex`. But for this task, we haven't supplied the key. All we'll tell you about the key is that it is 256 bits long and its 251 most significant (leftmost) bits are all 0s. The initialization vector was set to all 0s. First, find all plaintexts in the given key space. Then, you will review the plaintexts to find the correct plaintext that is the Jeopardy clue and the corresponding key.

What to submit Find the **key** of the appropriate plaintext and submit it as a hex string in `sol_3.1.2.3.hex`. Remember that this AES key is 256 bits long.

3.1.2.4 Decrypting a ciphertext with RSA (3 points) (*Difficulty: Easy*)

Files

1. `3.1.2.4_RSA_private_key.hex`: RSA private key (d) as hexadecimal string
2. `3.1.2.4_RSA_modulo.hex`: RSA modulo (N) as hexadecimal string
3. `3.1.2.4_RSA_ciphertext.hex`: an encrypted prime number that is encrypted with 1024-bit RSA as a hexadecimal string

In this part, we used 1024-bit textbook RSA to encrypt a prime number using your public key and stored it in `3.1.2.4_RSA_ciphertext.hex` as a hex string. Create a Python script named `sol_3.1.2.4.py` that takes as arguments the ciphertext, the private key, and the RSA modulo to compute the plaintext prime number and write it as a hex string to a specified file. We will run your script as follows:

```
$ python your_script.py ciphertext_file key_file modulo_file output_file
```

Hint You SHOULD complete this part using Python's `math` library.

What to submit Your Python script in `sol_3.1.2.4.py` and the prime number as a hex string in `sol_3.1.2.4.hex`.

3.1.3 Hash Functions

This section will give you a chance to explore cryptographic hashing using existing cryptographic libraries and illustrate the potential pitfalls of writing your own cryptographic functions.

3.1.3.1 Avalanche Effect (3 points) (*Difficulty: Easy*)

Files

1. `3.1.3.1_input_string.txt`: original string
2. `3.1.3.1_perturbed_string.txt`: perturbed string

`3.1.3.1_input_string.txt` contains another Jeopardy clue in ASCII.
`3.1.3.1_perturbed_string.txt` is an exact copy of this string with one bit flipped. We're going to use these two strings to demonstrate the avalanche effect by generating the SHA-256 hash of both strings and counting how many bits are different in the two results (a.k.a. the Hamming distance.)
What are their SHA-256 hashes? Verify that they're different:

```
$ openssl dgst -sha256 3.1.3.1_input_string.txt 3.1.3.1_perturbed_string.txt
```

Create a Python script named `sol_3.1.3.1.py` that takes as an argument two text files and an output file, and outputs the Hamming distance of the SHA-256 hash of the string in the two files as a hex string to a specified output file. We will run your script as follows:

```
$ python your_script.py file_1.txt file_2.txt output_file
```

What to submit Your Python script in `sol_3.1.3.1.py` and the Hamming distance as a hex string in `sol_3.1.3.1.hex`.

3.1.3.2 Weak Hashing Algorithm (3 points) (*Difficulty: Medium*)

Files

1. `3.1.3.2_input_string.txt`: input string

Below you'll find the pseudocode for a weak hashing algorithm we're calling `WHA`. It operates on bytes (block size of 8 bits) and outputs a 32-bit hash.

```
WHA:
Input{inStr: a binary string of bytes}
Output{outHash: 32-bit hashcode for the inStr as a series of hex values}
Mask: 0x3FFFFFFF
outHash: 0
for byte in input
  intermediate_value = ((byte XOR 0xCC) Left Shift 24) OR
                       ((byte XOR 0x33) Left Shift 16) OR
                       ((byte XOR 0xAA) Left Shift 8) OR
                       (byte XOR 0x55)
```

```
    outHash = (outHash AND Mask) + (intermediate_value AND Mask)
  return outHash
```

First, you'll need to implement `WHA` in Python. Here are some sample inputs you can use to test your implementation: `WHA("Hello world!") = 0x50b027cf` and `WHA("I am Groot.") = 0x57293cbb`

In the file `3.1.3.2_input_string.txt`, you'll find another Jeopardy clue (surprise!). Your goal is to find another string that produces the same `WHA` output as this Jeopardy clue. In other words, demonstrate that this hash is not second preimage resistant.

Find a string with the same `WHA` output as `3.1.3.2_input_string.txt` and submit it in `sol_3.1.3.2.txt`. Also, submit the code for your implementation of the WHA algorithm in `sol_3.1.3.2.py`. Your Python script should take as arguments a text file and an output file, and outputs the WHA hash of the content of the file as a hex string in the specified file. We will run your script as follows:

```
$ python your_script.py file.txt output_file
```

What to submit Your Python script in `sol_3.1.3.2.py` and the collision string in `sol_3.1.3.2.txt`

Checkpoint 1: Submission Checklist

The following blank files for checkpoint 1 have been created in your git repository under the directory Crypto. Put your solutions inside the corresponding files, then commit it to git. All .hex and .txt files MUST be submitted as ASCII plaintext, and any lines in .txt and .hex submissions that begin with a '#' will be ignored.

- `partners.txt` [One NetID on each line]
- `sol_3.1.1.2_decimal.txt`
- `sol_3.1.1.2_binary.txt`
- `sol_3.1.2.1.py`
- `sol_3.1.2.1.txt`
- `sol_3.1.2.2.py`
- `sol_3.1.2.2.txt`
- `sol_3.1.2.3.hex`
- `sol_3.1.2.4.py`
- `sol_3.1.2.4.hex`
- `sol_3.1.3.1.py`
- `sol_3.1.3.1.hex`
- `sol_3.1.3.2.py`
- `sol_3.1.3.2.txt`

Example content of a .txt solution file

```
# this line is ignored
SPN WMKTQIW QR SPBW HQGRSEMW HQVS QY VEKW
```

Example content of a .hex solution file

```
# this line is also ignored
3dab821d92b5ca7f48beee066996b8abc82f7e5646a0561710ea5bc11c80d
```

3.2 Checkpoint 2 (100 points)

3.2.1 Length Extension (15 points) (*Difficulty: Medium*)

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-size blocks by applying the compression function to the current state and the 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.

3.2.1.1 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 in the same way. You should have a pymd5.py module in your SVN directory. Documentation for pymd5 is available 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 the 512-bit length. The padding consists of a 1 bit, 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()
```

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.

3.2.1.2 Conduct a Length Extension Attack

Files

1. `3.2.1.2_query.txt`: query
2. `3.2.1.2_command3.txt`: command3

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* ∥ *message*), where ∥ indicates concatenation. For example, Professor Vuln E. Rabble 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://ece422.org/project3/api?token=b301afea7dd96db3066e631741446ca1&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*]). The domain name is given as an example, we did not set up a web server for this assignment.

Text files with the query of the URL `3.2.1.2_query.txt` and the command line to append `3.2.1.2_command3.txt` are provided. Using the techniques that you learned in the previous section and without guessing the password, apply length extension to create a new query in the URL ending with the command specified in the file, `&command3=DeleteAllFiles`, that is treated as valid by the server API. We will run your script as follows:

```
$ python your_script.py query_file command3_file output_file
```

Create a Python script named `sol_3.2.1.2.py` that takes as a command line argument a filename containing a valid query in the URL and modifies it such that it will execute a `DeleteAllFiles` command as the user, then output the new query to a specified file. You may assume that the query will always begin with the token.

Hint: You might want to use the `quote()` function from Python's `urllib` module to encode non-ASCII characters in the padding.

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 Your Python script in `sol_3.2.1.2.py` and the modified query in `sol_3.2.1.2.txt`.

3.2.2 MD5 Collisions (15 points) (*Difficulty: Medium*)

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.

3.2.2.1 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://github-dev.cs.illinois.edu/cs461-sp19/_public/blob/master/crypto/Makefile`. You will also need the Boost libraries. On Ubuntu, you can install these using `apt-get install libboost-all-dev`. On macOS, 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.

This component is intended to introduce you to MD5 collisions; you will not submit anything for it.

3.2.2.2 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'll use Python, but almost any language would do. Copy and paste the following three lines into a file called `prefix`: (Note: typing the below lines yourself may lead to an encoding mismatch and an error may occur when running the resulting Python code)

```
#!/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 come in peace."` The second should execute a pretend malicious payload: `print "Prepare to be destroyed!"`. Note that we may rename these programs before grading them.

What to submit Two Python 2.x scripts named `sol_3.2.2_good.py` and `sol_3.2.2_evil.py` that have the same MD5 hash, have different SHA-256 hashes, and print the specified messages.

3.2.3 Exploiting a Padding Oracle (25 points) (*Difficulty: Hard*)

In the `3.2.3_ciphertext.hex` file, you will find an AES-encrypted ciphertext, encrypted using a random initialization vector and a fixed secret key. Your goal, of course, is to find a way to decrypt this ciphertext.

Before encrypting, the plaintext was padded to a multiple of 16 bytes using a custom padding scheme as follows: The first byte of padding is `0x10`, the next padding byte is `0x0f`, the next is `0x0e`, and so on, until a multiple of 16 bytes is reached. If the plaintext was already a multiple of 16 bytes, then the entire 16-byte sequence `{0x10,0x0f,0x0e,...,0x01}` is appended to the plaintext. Thus the following examples:

- `"a"` becomes `"a\x10\x0f...\x02"`
- `"abcde"` becomes `"abcde\x10\x0f...\x06"`
- `"abcdefghijklmnop"` becomes `"abcdefghijklmnop\x10\x0f...\x01"`

The following Python code implements this padding scheme:

```
def pad(msg):
    n = len(msg) % 16
    return msg + ''.join(chr(i) for i in range(16, n, -1))
```

The web application located at `http://cs461-mp3.sprai.org:8081/mp3/` can be used to check the integrity of your ciphertext. It reads the ciphertext in hex from the URL query string, decrypts it using the secret key, removes the padding, and confirms whether or not the resulting plaintext corresponds to the Jeopardy clue you must provide as the solution to this task. For example, running the following shell command:

```
$ curl http://cs461-mp3.sprai.org:8081/mp3/${netid}/?$(cat 3.2.3_ciphertext.hex)
```

should return a response with HTTP status code `200 OK` and containing the string:

```
Correct ciphertext!
```

If the ciphertext is incorrect, or if the web application encounters any error while decrypting, then you will receive an error code.

We have also provided a local server for you to run, using the command:

```
$ python ece422-mp3-paddingoracle-server-standalone.py
```

You might think that this integrity checker isn't much help to you. But actually, the padding scheme and the error reporting of the web application interact in a devastating way. Your task is to use the technique described in Vaudenay's 2002 paper[1] to recover the plaintext.

For your reference, the following Python code will load a URL and print the HTTP status code:

```
import urllib2

def get_status(u):
    req = urllib2.Request(u)
    try:
        f = urllib2.urlopen(req)
        print f.code
    except urllib2.HTTPError, e:
        print e.code
```

What to submit Your Python script in `sol_3.2.3.py` and the decrypted message in `sol_3.2.3.txt`.

3.2.4 Mining your Ps and Qs (25 points) (*Difficulty: Hard*)

The "Pretty Bad Privacy" encryption tool, `pbp.py`, can be used to insecurely encrypt files to a 1024-bit RSA public key. [2]

Each line of the `https://github-dev.cs.illinois.edu/cs461-sp19/_public/blob/master/crypto/moduli.hex` file contains a 1024-bit RSA modulus, 10,000 of these in total.

In `3.2.4_ciphertext.enc.asc` you have been provided the ciphertext of a Jeopardy clue, which has been encrypted using PBP with one of the RSA moduli in the file, and public exponent $e = 65537$. Factoring *any* of the 1024-bit moduli before the assignment is due is infeasible; furthermore you don't even know which one to start on!

[1] `https://www.iacr.org/archive/eurocrypt2002/23320530/cbc02_e02d.pdf`

[2] PBP is a "hybrid encryption" mode. It uses 1024-bit RSA (with OAEP padding, rather than textbook RSA), to encrypt a random 256-bit key, and then uses this as an AES key to encrypt the (padded) message.

Sometimes, badly malfunctioning implementations of RSA fail to generate unique prime numbers.The RSA moduli in the provided list were generated without sufficient entropy, and some of them share common factors. If two RSA moduli share a common factor, it is trivial to compute their GCD and factor both moduli. Unfortunately, looping over all pairs of moduli does not scale well, so you'll have some difficulty finishing the project unless you use a more efficient algorithm.

Your task is to use the method described in the "Mining your Ps and Qs" paper,[3] Section 3.3, to compute the pairwise GCDs of the RSA keys provided. Once you have discovered some RSA private keys, you can then attempt to use them to recover the RSA-encrypted AES session key and decrypt the rest of homework file and submit the plaintext in `sol_3.2.4.txt`.

What to submit Your python script in `sol_3.2.4.py` and the decrypted message in `sol_3.2.4.txt`.

3.2.5 Creating Colliding Certificates (20 points) (*Difficulty: Hard*)

The ECE 422 Certificate Authority issues TLS certificates, but charges for each signature. Your challenge is to rip off the CA by creating a pair of distinct (but valid) certificates, which both share the same signature from the CA.

Since the CA signature can be based on the MD5 hash of the body of the ceritficate, you can use `fastcoll` to help you to create a collision. However, you also need to make sure that you know the private key corresponding to the RSA public key in each certificate. A method to achieve this is described in detail in Lenstra's paper[4].

To help you out, we have provided you with `mp3-certbuilder.py`, which is a script that uses the `cryptography` python library and outputs a certificate with a random public key, with the fields to specific values, and signed by the ECE 422 CA (the CA's private key is included in the script). You run the script using the following command:

```
$ python mp3-certbuilder.py {netid} {output filename}.cer
```

This gives you an output certificate which you can view using the following command:

```
$ openssl x509 -in {output filename}.cer -inform der -text -noout
```

The certificate structure is shown below:

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            4e:41:f2:89:eb:7b:92:ea:72:8c:d5:b5:41:82:a7:f4:3d:40:52:43
    Signature Algorithm: md5WithRSAEncryption
```

[3]https://factorable.net/weakkeys12.extended.pdf

[4]`https://www.win.tue.nl/~bdeweger/CollidingCertificates/CollidingCertificates.pdf`

```
    Issuer: CN=ece422
    Validity
        Not Before: March  1 00:00:00 2017 GMT
        Not After : March 27 00:00:00 2017 GMT
    Subject: CN=amingni2, pseudonym=unused, C=US, ST=Illinois
    Subject Public Key Info:
        Public Key Algorithm: rsaEncryption
            Public-Key: (2048 bit)
            Modulus:
                00:93:35:d1:0c:a1:11:a2:7c:53:38:7b:05:ab:3f:
                39:69:de:58:e1:cd:43:af:dc:5e:93:00:9c:4e:18:
                34:8e:86:17:0d:4e:be:06:63:69:34:ae:08:a1:a5:
                0f:b6:fa:d8:d8:3f:c1:cc:a9:c8:2c:ea:01:4c:81:
                55:7b:c7:a5:3f:57:3e:0b:a4:f9:ee:ba:4f:d3:bd:
                46:e0:f8:ee:24:a0:d3:63:4d:9c:d8:65:aa:ad:98:
                2d:ed:18:85:16:d7:64:53:58:e9:2b:20:2a:87:c2:
                15:3b:b2:2e:06:57:23:b4:bd:91:3b:d0:8c:97:fb:
                4e:ec:18:88:41:24:b2:45:ce:0c:1b:11:0b:54:10:
                48:b3:3e:ca:fb:a0:94:dd:7e:20:a5:a6:92:72:1e:
                b6:3d:8a:81:eb:3b:41:94:c5:04:f0:49:e4:77:9f:
                fc:1f:6b:b6:f8:1d:3f:c0:3c:12:a5:cb:a1:68:76:
                29:76:f8:0c:74:07:58:bf:4f:ba:a6:9f:a4:4b:50:
                e2:6a:27:5f:4c:c0:94:47:7a:24:53:e5:eb:73:4c:
                a7:53:7a:a3:0b:b1:60:7f:2a:b9:9a:ed:44:63:20:
                f0:39:32:cb:36:93:6e:92:c0:05:db:c9:10:ae:32:
                8a:2b:df:39:84:28:69:7e:1c:2f:38:b0:a8:c3:e4:
                87:af
            Exponent: 65537 (0x10001)
Signature Algorithm: md5WithRSAEncryption
     88:61:19:2b:74:f0:63:4a:d0:a4:6d:ff:48:5e:b5:01:aa:be:
     c6:0e:82:c6:53:11:86:b4:78:53:39:d4:0d:58:1d:be:11:47:
     a2:69:2a:73:aa:06:1f:4e:65:75:46:f3:59:8f:69:73:75:79:
     6f:cd:0e:a8:7a:56:b8:5c:02:ff:b6:78:8b:dc:ca:96:3f:2f:
     70:21:24:4a:83:ad:d9:bc:b4:88:60:e1:28:ea:9c:7f:0a:c8:
     b6:d2:08:82:aa:cf:31:01:bd:65:41:95:b6:cb:30:3f:0c:e8:
     b1:7c:e0:94:d9:4b:69:87:79:d2:c4:e7:3e:51:3e:6a:2e:df:
     a3:83:84:27:f7:ee:80:fd:c3:28:21:04:c4:46:1b:8b:ff:43:
     73:e7:fe:bd:89:3f:0a:1b:2d:6a:57:62:94:2d:46:56:66:1a:
     80:a1:07:7a:fe:f6:ff:ce:80:7f:8a:bd:3e:e4:06:41:16:4a:
     b4:66:bc:07:87:40:5f:26:d1:48:ab:df:ee:6d:f4:6a:b1:07:
     83:45:44:c6:6a:26:c6:23:d5:58:c5:9e:1e:f0:32:98:35:07:
     b1:08:45:ee:77:d5:b9:27:f6:41:ad:08:f6:63:be:3e:63:9e:
```

```
        62:26:de:6e:8e:1f:e9:9e:29:4f:6f:67:d7:62:cc:f2:ec:e6:
        b7:e0:0f:66
```

The two certificates you create MUST have the following fields set correctly:

- Issuer Common Name: ece422
- Subject Common Name: your NetID
- Not valid before: March 1, 2017
- Not valid after: March 27, 2017
- Country code: US
- State or Province: Illinois
- Signature Algorithm: md5WithRSAEncryption

However, other fields, such as Subject Pseudonym, are optional and can be set to whatever you desire. You must submit these certificates in the files `sol_3.2.5_certA.cer` and `sol_3.2.5_certB.cer`. You must also include the factors for the RSA modulus of the public keys in `sol_3.2.5_factorsA.hex` and `sol_3.2.5_factorsB.hex` (one factor per line, as a hexadecimal number). Each individual factor must be a prime numer larger than 256 bits, with the total size of the RSA modulus being at least 2000 bits. The signature from the ECE 422 CA must be identical for both certificates.

What to submit Your colliding certificates in `sol_3.2.5_certA.cer` and `sol_3.2.5_certB.cer` and the RSA factors in `sol_3.2.5_factorsA.hex` and `sol_3.2.5_factorsB.hex`.

Checkpoint 2: Submission Checklist

The following empty files for Checkpoint 2 have been created in your git repository under the directory Crypto. Put your solutions inside the corresponding files then commit it to git. All .hex and .txt files MUST be submitted as ASCII plaintext, and any lines in .txt and .hex submissions that begin with '#' will be ignored.

- `partners.txt` [One NetID on each line]
- `sol_3.2.1.py`
- `sol_3.2.1.txt`
- `sol_3.2.2_good.py`
- `sol_3.2.2_evil.py`
- `sol_3.2.3.txt`
- `sol_3.2.3.py`
- `sol_3.2.4.txt`
- `sol_3.2.4.py`
- `sol_3.2.5_certA.cer`
- `sol_3.2.5_certB.cer`
- `sol_3.2.5_factorsA.hex`
- `sol_3.2.5_factorsB.hex`

Example content of a .txt solution file

```
# this line is ignored
SPN WMKTQIW QR SPBW HQGRSEMW HQVS QY VEKW
```

Example content of a .hex solution file

```
# this line is also ignored
3dab821d92b5ca7f48beee066996b8abc82f7e5646a0561710ea5bc11c80d
```