Project 1: Crypto means Cryptography

This project is due on **Thursday, February 7** at **6 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 will work in **teams of two or three** and submit one project per team. Please find a partner as soon as possible. If 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 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.

Solutions must be submitted electronically via **Dropbox**, following the submission checklist below.

Introduction

In this project, you will investigate vulnerabilities in widely used cryptographic hash functions, including length-extension attacks and collision vulnerabilities. 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.

Objectives:

- Understand how to use basic cryptographic primitives.
- Implement a simple ECDSA verification, and understand how to use it.
- Investigate how cryptographic failures can compromise the security of applications.

Part 1. Hash functions

The SHA256 hash function produces 256-bit outputs for any given input. In this part, you will find a *partial* pre-image of the SHA256 hash function.
Your task is to find a string that starts with all the identikeys of your group, followed by a number, that when hashed with SHA256 produces a (256-bit) number that starts with at least **24-bits of 0s**. As an example, the string `ewust-2556157` produces the SHA256 hash `00000036e567244755226583b96f8bc0bd0bf0759947ab1780410ec936d3e79c`. You can verify this using the Python hashlib library:

```
import hashlib

print hashlib.sha256('ewust-2556157').hexdigest()
# Prints '00000036e567244755226583b96f8bc0bd0bf0759947ab1780410ec936d3e79c'
```

This number starts with 26 bits of 0s (each hex nibble is 4 bits, and 3 is written as 0b0011, giving us 6*4+2 bits of 0s)
For multiple group members, separate identikeys using a hypen. For example: `dawi4532-erwu6167-` followed by a number.

What to submit

1. A plaintext file, named `preimage.txt` that contains your input string (e.g. `ewust-2556157` followed by a space, and its SHA256 hex digest.
2. A Python script, named `preimage.py` that you used to generate preimage.txt.

An example `preimage.txt` file for my identikey would be:

```
ewust-2556157 00000036e567244755226583b96f8bc0bd0bf0759947ab1780410ec936d3e79c
```

Bonus challenge: (optional)

Try to find the smallest SHA256 output for your group's prefix. Hint: you may want to write a program in something faster than Python!
Rules: the string must start with your group's identikeys (as in Part 1.2). You may not use any CPU resources that do not belong to you (i.e. no hijacking university computers to carry out your computation)!

Submit your group's string (and hash) by email to `ewust@colorado.edu`, with the subject 'Hash challenge' by class on February 7th. We'll verify and announce the winner in class on February 12th.

Part 2. Signatures

In this part, you'll investigate using ECDSA signatures. You'll implement a verification function in Python, and then use it to pair messages with their signatures.
For this part, you'll need two files:

- Our ECDSA library: `https://ecen5033.org/static/ecdsa.py`
- The template script: `https://ecen5033.org/static/verify.py`

Save these to the same directory. In this part, you'll use the provided `ecdsa.py` library to implement an `ecdsa_verify` function in the `verify.py` template.

We've implemented an `ecdsa_sign` function in `ecdsa.py`, and you can see it used lower in the same file (after "__main__"). You can run this by running the `ecdsa.py` file:

```
python ecdsa.py
```

This should print out a random private key, its corresponding public key, a signature, and then it should exit with a 'Not implemented' Exception. This is because the script calls the `ecdsa_verify` function in `verify.py`, which you will have to fill in. Your code will have to hash the message, and compute the necessary point multiplications/additions and return if the signature is valid or not. The function should return True for valid signatures, and False otherwise.

Part 2.1 Write `ecdsa_verify`

Your first task is to implement the `ecdsa_verify` function in `verify.py`. Once complete, running `python ecdsa.py` should run without producing an exception, and end by printing out:

```
Signature is Valid!
Correctly rejected invalid signature
```

We've provided all the math functions that you'll need in `ecdsa.py`, such as `ECPoint.mult` for point mulitiplication, and `ECPoint.add` for point addition. There are also encode/decode functions that may be useful. You should review `ecdsa_sign` in `ecdsa.py` for examples of how to use the provided library to do point multiplication and hashing, and refer to `https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm` for the algorithm needed to verify signatures.

You won't need to change anything in `ecdsa.py`, all your changes should be to `verify.py`. We recommend you test your implementation with different messages, signatures, and keys to ensure that it works properly; `ecdsa.py` tests a correct and incorrect signature.

Part 2.2 Pairing signatures

In this part, you'll use the `ecdsa_verify` function you wrote previously to pair a set of messages with their corresponding signatures.

In `verify.py`'s `__main__` section, there are 6 messages in the `msgs` array, and 6 signatures in `sigs`. Each message has a signature that it pairs with, though note they are shuffled (i.e. `msgs[0]` doesn't correspond with `sigs[0]`).

Your task is to pair the messages with their signatures. When run (`python verify.py`), for each message, on its own line, print the message, followed by a space, and then its corresponding signature (in hex/encoded form). Order of the messages is not important, just ensure they are paired with their proper signature.

Part 2.3 Extract the private key

Note: **Required for graduate students** If there is a graduate student in your group, you must complete Part 2.3. Otherwise, this is optional (but encouraged!).

The signatures provided in Part 2.2 allow you to recover the private key used to sign the messages. In this part, you will recover the nonce, compute the private key, and use it to sign a new message (using `ecdsa_sign` in `ecdsa.py`). Your message should be a single line (contain no line-breaks), and should contain the identikey's of your group, but is otherwise up to you. You should print it, followed by a space, and its signature as if it were one of the messages in Part 2.2. Be sure to double check that it validates with your `ecdsa_verify` function!

What to submit

1. Your (modified) `verify.py` file, that completes Parts 2.1, 2.2 and for grad students, Part 2.3.

Submission Checklist

Upload to **Dropbox** (`https://www.dropbox.com/request/MG3nwX1h120YMlUacala`) a gzipped tarball (`.tar.gz`) named `project1.`*`identikey1`*`.`*`identikey2`*`.tar.gz`. You can make a tarball with

`$ tar czf project1.`*`identikeys`*`.tar.gz ./preimage.py ./preimage.txt ./verify.py`

The tarball should contain only the following files:

Part 1

A Python script **`preimage.py`** computing your partial pre-image for your group, and the **`preimage.txt`** plaintext file containing the string and its hash that begins with 24-bits of 0s.

Part 2

Your modified **`verify.py`** Python script, that completes Part 2.1 and 2.2, and if your group contains a graduate student, Part 2.3.