Raluca Ada Popa
Spring 2018

CS 161 Computer Security

Project 2

Part 1 Due: March 5, 2018, 11:59 PM

Part 2 Due: March 19, 2018, 11:59 PM

Part 3 Due: April 2, 2018, 11:59 PM

Version 1.1: February 22, 2018

Contents

Introduction

Storing files on a server and sharing them with friends and collaborators is very useful. Commercial services like Dropbox or Google Drive are popular examples of a file store service (with convenient filesystem interfaces). But what if you couldn't trust the server you wanted to store your files on? What if you wanted to securely share and collaborate on files, even if the owner of the server is malicious?

In this project, you'll use the cryptographic tools we've taught you to build a file storage client that's secure and efficient despite storing all of your data on a malicious storage server.

Getting Started

Documentation for this project will be hosted on the course website; for how you can get started on this project: `http://inst.eecs.berkeley.edu/~cs161/sp18/projects/2/docs/gettingstarted.html`

For an example of how to work on this project, refer to the Example Workflow Section.

Secure File Store

Your task is to design and implement a secure file store. This file store can be used to store your own files securely, or to share your files with other people you trust.

Your implementation should have two properties:

> **Confidentiality.** Any data placed in the file store should be available only to you and people you share the file with. In particular, the server should not be able to learn any bits of information of any file you store, nor of the name of any file you store.
>
> **Integrity.** You should be able to detect if any of your files have been modified while stored on the server and reject them if they have been. More formally, you should only accept changes to a file if the change was performed by either you or someone with whom you have shared access to the file.

Note on security parameters. It is sufficient that these properties hold with very high probability (i.e., no more risk than arises from brute forcing well-chosen cryptographic keys).

You are given access to two servers:

1. A storage server, which is **untrusted**, where you will store your files. It has three methods:
   - `put(id, value)`, which stores `value` at `id`

- `get(id)`, which returns the value stored at `id`
- `delete(id)`, which deletes the value stored at `id`

2. A public key server, which is **trusted**, that allows you to receive other users' public keys. You have a secure channel to the public key server. It has four methods:
   - `get_encryption_key(username)`, which returns the public encryption key for `username`
   - `put_encryption_key(username, pubkey)`, which sets the public encryption key for your username
   - `get_signature_key(username)`, which returns the public signature key for `username`
   - `put_signature_key(username, pubkey)`, which sets the public signature key for your username

You are not to change the code for either server. If you do, your code will not work with our autograder and you will get no credit.

The storage server is, in practice, just a key-value store. The files you upload to the server are strings of text data. The storage server is untrusted—it can perform arbitrary malicious actions to any data you store there. You should protect the confidentiality and integrity of any data you store on either server.

The storage server has one namespace, so anything written by one user can be read or overwritten by any other user who knows the `id`. Clients interacting with the storage server must take care to ensure that their own files are not overwritten by other clients. Other users or clients might be malicious.

We provide you a framework off of which to build; the documentation for all these files is available online at http://inst.eecs.berkeley.edu/~cs161/sp18/projects/2/docs/.

You must use our provided `crypto.py` API for all of your security-critical operations. Do not implement your own versions of symmetric (or asymmetric) key operations. This API has access to all the raw primitives we have taught you. Do not create a new instance of the Crypto object: use the one passed to you during initialization. It also has a secure random-bytes generator and other accessory methods. You should inspect it to see how it calls into PyCrypto, to understand what security properties you can expect out of this module. We have provided this API to you as a cleaner interface than the hundred possible methods in PyCrypto, and one that operates on strings for easier debugging. But we expect that you will understand the consequences of how this code behaves. **You must NOT call into PyCrypto yourself**.

The skeleton provided in `client.py` calls `BaseClient.__init__()`, which sets up the client attributes, and calls the method `generate_public_key_pairs()`. This method will automatically put the asymmetric keys on the public key server, and save a copy of your private keys to your filesystem. This is the only persistent state that your client can use (you can

assume that for the same username, a client will have the same public/private keys even if restarted). Your client code should call this method exactly once.

Your code must not spawn other processes, read or write to the file system, open any network connections, or otherwise attack the autograder. We will run your code in an isolated sandbox. Any adversarial behavior will be seen as cheating.

The only exception your code may raise is an `IntegrityError`. Your code should handle all other exceptions.

Part 1: A simple, but secure client

You may focus first on Part 1 and start later parts only after you've completed Part 1. Part 1 is designed to get you familiar with the API and crypto operations.

Part 1 is due March 5, 2018, 11:59 PM.

Simple Upload/Download

Implement a file store with a secure (but possibly inefficient) upload/download interface.This will require you to implement the methods `upload` and `download`.

The methods must ensure the following properties hold.

Property 1 (Secure Download) *When not under attack by the storage server or another user,* `download(name)` *MUST* [1] *return the* ***last*** *value stored at* `name` *by the current user, or* `None` *if no such file exists. It MUST NOT raise* `IntegrityError` *or any other error when not under attack. However,* `download(name)` *MUST NOT ever return an* ***incorrect*** *value. A* `value` *(excluding* `None`*) is "incorrect" if it is* ***not*** *one of the values currently or previously stored at* `name` *by the current user.*

`download(name)` *MAY raise* `IntegrityError` ***or*** *return* `None` *if under* ***any*** *attack by the server or other users.* `download(name)` *MUST NOT raise any other errors.*

It SHOULD raise `IntegrityError` *if the file has been tampered with. It SHOULD return* `None` *if it appears that no value for* `name` *exists for the user.*

Property 2 (Secure Upload) `upload(name, value)` *MUST place the value* `value` *at* `name` *so that future* `download`*s for* `name` *return* `value`*.*

This function SHOULD return True. It MAY return False if the upload fails due to a malicious server.

Any person other than the owner of `name` *MUST NOT be able to learn even partial information about* `value` *or* `name` *with probability better than random guesses.*

You may assume filenames are alphanumeric (they match the regex `[A-Za-z0-9]+`). Filenames will not be empty. This will be the case for all parts of this project. The contents of the file can be arbitrary: you must not make any assumptions there, but they are provided as Python Unicode strings (for easier debugging). You may assume usernames consist solely of lowercase letters (`[a-z]+`).

The autograder does not look at the return value of the `upload` method.

You do not need to implement any capabilities for sharing files between users (this is Part 2). We have provided an implementation of the storage server—do not change it.

[1] See RFC 2119 for the definitions of MUST, MUST NOT, SHOULD, and MAY.

Note that we require you to protect the confidentiality and integrity of both the contents of the file you store and the name it is stored under. A malicious storage server must not be able to learn either, or change them. The length of the file doen't need to be kept confidential.

When used in a non-adversarial manner, different users should be allowed to have files with the same name: they should not overwrite each other's files. An adversary may be able to overwrite a user's valid data, but any changes should be reported as an `IntegrityError`.

Confidentiality for this project follows from the IND-CPA game, but with minor alterations. The adversary choses two files with values of the same size $F_0 = (\text{name}_0, \text{value}_0)$, $F_1 = (\text{name}_1, \text{value}_1)$ to be stored, where the client will choose one randomly. Then the adversary can ask for a polynomial amount of arbitrary files $F_i = (\text{name}_i, \text{value}_i)$ to be stored where name_i cannot be equal to name_0 or name_1. If the adversary can figure which of F_0 or F_1 was stored on the server, then we have lost confidentiality.

One specific attack **you are not required to handle** is that of a **rollback attack**: if Alice uploads the file F to the server and then updates it later to F' with a second upload, Alice does not need to detect if the server "rolls back" its state and returns F when Alice requests the file back. This is why Property 1 is written as it is.

Why do we not require this? Without additional state, it would be impossible. The server could always rollback to the "empty" state where it contains no data at all, and return `None` for every `get`, and the client would not be able to detect this.

Your client must not assume it can keep any state other than its asymmetric keys that it was created with. You must assume that your client can be killed and restarted, and everything should still work. (For example: you cannot place a dictionary in your client, make `upload` insert into the dictionary, make `download` get from the dictionary, and claim to be secure because you send nothing to the `StorageServer`.)

Note that this means if you require temporary symmetric keys, you will need to be able to save and restore them using only the two persistent asymmetric keys each client is given.

You do not need to handle the case where two users interact with the server *concurrently*: you can assume only one user will interact with the server at any point in time. (That is, you do not need to worry about implementing locking—if one user has issued an API call, then no other user does so until that API call completes.)

Testing your submission: We have a set of tests which we will run on your code, for both functionality and security. In the provided framework, a file called `run_part1_tests.py` contains all the functionality tests we will run on your code, but only 1 security test. (We have many more security tests!) To run these tests, run `python3 run_part1_tests.py`. This will use your `Client` implementation from `client.py`. It will output Pass/Fail for each test we have and a one-sentence explanation of what the test does if it fails.[2]

[2]We will make minor modifications to the functionality tests before we run them (e.g., by changing the names of keys and values) to ensure that solutions are not hard-coded to pass our tests.

We will not have any significantly new tests for functionality. If we add any new tests to the framework, we will announce this and release an updated set.

These tests are provided to make sure that anyone who attempts the project will get full points on functionality. This project is to test your ability to write secure code, not to implement a key-value store.

There will be no performance tests for Part 1 (although your code should definitely terminate!). Each test must complete in under one minute, but we expect tests to run in a few seconds.

Submission and Grading

Using `glookup` with `submit proj2-part1`, you should submit your final version of `client.py` file (and only that file) by March 5, 2018, 11:59 PM.

You are responsible for verifying that your code passes all functionality tests.

We plan to grade Part 1 shortly after the deadline. We will grade your last submission, and provide:

- the raw score based on functionality tests
- the raw score based on a large set of security tests
- a one sentence summary of any security tests failed (so you have the chance to fix these issues for Part 2 and Part 3)

Your final score on this part of the project will be the minimum of the functionality score and security score. Each failed security test will lower the security score, weighted by the impact of the vulnerability.

We will **not** accept regrade requests on the autograder results, except in cases where there was a bug in the autograder. If you feel this has occurred, please post a private question on Piazza to instructors and we will look at your code.

Submission Summary

In summary, you must submit the following directory tree for Part 1:

```
client.py
feedback.txt        (optional)
```

Part 2: Sharing and revocation

Make sure you read instructions for Part 2 fully before starting. It is likely that the design of your system for Part 2 will be significantly different from your previous solution. A similar thing might happen for Part 3 as well, so you may choose to read Part 3 before starting Part 2, and try to design your solution for both parts together.

Part 2 is due March 19, 2018, 11:59 PM.

Sharing

A file store becomes much more interesting when you can use it to share files with your collaborators. Implement the sharing functionality by implementing the methods `share()` and `receive_share()`.

When Alice wants to share a file with Bob, she will call `msg = alice.share("bob", filename)` to obtain a sharing message. Alice will then pass Bob `msg` through an out-of-band channel (e.g., via email). You must not assume that this channel is secure. A man-in-the-middle might receive or modify the sharing message after Alice sends it but before Bob receives it.

After Alice passes `msg` to Bob, if Bob wishes to accept the file, he will call `bob.receive_share("alice", newfilename, msg)`.[3] Bob should now be able to access Alice's file under the name `newfilename`. In other words, Alice accesses the file under the name `filename`; Bob accesses it using the name `newfilename`.

`msg` must be a Python string. During grading, we will pass `msg` from one client to another on your behalf. Sharing a document must not require any other communication between the clients.

Property 3 *After* `m = a.share("b", n1); b.receive_share("a", n2, m)`, *user* `b` *MUST now have access to file* `n1` *under the name* `n2`. *Every user with whom this file has been shared (including the owner) MUST see any updates made to this file immediately. To user* `b`, *it MUST be as if this file was created by them: they MUST be able to read, modify, or re-share this file.*

This also changes Property 1 and Property 2 from above. A `download()` operation MUST return the last value written by anyone with access to the file (the owner, or anyone with whom the file was shared). Only those with access to the file should be able to read or modify it.

Sharing is tricky. Note that both filenames refer to the same underlying file, and any updates performed by anyone who has access to the file should be immediately visible to all other users with access to the file. By "update", we are referring to the case where a user invokes

[3] Bob populates the first two arguments himself after he receives what he believes is a valid `msg` from Alice. Consequently, these two arguments cannot be tampered with by the man-in-the-middle attacker.

`upload(f, v2)` on a file `f` that was previously uploaded and whose previous contents were `v1`.

Sharing should be transitive. If Alice shares a file with Bob who shares it with Carol, any changes to this file by any of the three should be visible to all three immediately. Sharing a file with someone who has already received it results in unspecified behavior (you may do whatever you choose). It is okay if the storage server learns which other users you have shared a file with.

We require a minimal amount of efficiency: assuming a file of size m is shared with n users, and Alice shares the file with a new user, you may perform a linear (in $O(n+m)$) number of either public or private key encryption operations. This is simple to achieve: any reasonable scheme should be at least this efficient. It is possible to do significantly better—and you are free to do so if you choose—but we will not evaluate you on this.[4]

Your client may only keep state for performance reasons. Your implementation must work if your client is restarted in between every operation. Any state maintained on your client must be able to be reconstructed from data that exists on the server. Your clients may not directly communicate with each other.

Again, you do not need to worry about rollback attacks with sharing. For example, the server could rollback state and remove a client from receiving updates. You do not have to mitigate this. But remember, if you do notice any discrepancy during operation, you should throw an `IntegrityError`.

Grading: The security tests for this sub-part differ significantly from the previous tests. You must ensure you respect all sharing requirements, and that only valid users are able to read or edit a file. We will also test all functionality aspects, including all the tests from Part 1.

If your implementation relies on more out-of-band messages than a single return value from `share()`, you will get no credit for this sub-part (or revocation).

Revocation

Remote collaboration is a difficult thing, and, unfortunately, one of your collaborators has betrayed you, and you can no longer trust them. You realize that you need to revoke their access to your files.

Implement the `revoke()` method, which allows a user to revoke someone else's access to a given file. You can't stop them from remembering whatever they've already learned or keeping a copy of anything they've previously downloaded, but you can stop them from learning any new information about updates to this file. Only the user who initially created the file may call `revoke()`.

[4] Note that while you do not need to worry about the performance of sharing here, in Part 3 you will be required to augment your design to make updates efficient (in terms of the number of bytes transferred across the network).

Property 4 *If the original creator of the underlying file calls* `revoke(otheruser, name)`*, then afterwards* `otheruser` *MUST NOT be able to observe new updates to* `name`*, and anyone with whom* `otheruser` *shared this file MUST also be revoked. Except for knowing the previous contents of* `name`*, to* `otheruser`*, it MUST be as if they never had received the file.*

This single property has several hidden implications which may not be clear right away. Suppose that in the past, Alice granted Bob access to file F, and now Alice revokes Bob's access. Then we want all the following to be true subsequently:

1. Bob should not be able to update F,
2. Bob should not be able to read the updated contents of F (for any updates that happen after Bob's access was revoked), and
3. If Bob shared the file with Carol, Carol should also not be able to read or update F.
4. Bob should not be able to regain access to F by calling `receive_share()` with Alice's previous `msg`.

Revocation must not require any communication between clients.

You only need to implement functionality to revoke access from direct children. If Alice shares a file with Bob, and Bob shares the file with Carol, you are not required to provide a way for Alice to directly revoke Carol's access. It must work for Alice to revoke Bob's access, and revoking Bob's access should recursively revoke Carol's access.

If Alice shares a file with Bob, and Bob shares the file with Carol, you don't need to provide a way for Bob to revoke Carol's access. We will not test this situation: you only need to ensure that the original creator of the file can revoke others.

If Alice shares a file with Bob, and then revokes Bob's access, it may still be possible (depending on the design of your system) for Bob to mount a denial of service (DoS) attack on Alice's file (for example, by overwriting it with all 0s, or deleting `id`s), but Alice should never accept any changes Bob makes as valid. She should always either raise an `IntegrityError`, or return `None` (if Bob deleted her files).

Similar to sharing we will not grade you on efficiency. You may make a linear number of operations proportional to the size of the file, and the number of users who have received this file.

All the requirements from the previous parts are carried over to this part. Recall that the only state which you can keep in the client is your public and private key. Any other state stored must be only an optimization: it must be recoverable from state stored on the server.

As before, you do not need to worry about rollback attacks with revocation. For example, the server may rollback state and remove a client from receiving updates, or re-share with an old client. You do not have to mitigate this. But if you do notice any discrepancy during operation, you should throw an `IntegrityError`.

Grading: It will be very difficult for you to receive any credit on this part if your implementation does not pass the functionality and security tests from sharing. Since, in this case,

functionality and security are tightly bound (revocation is a security behavior), we will not be providing you with difficult functionality tests. We have provided you with trivial tests, but you should definitely implement your own (although we will not ask for your tests). After submission, our autograder will apply some more difficult tests.

Design Document for Part 2

Write a clear, concise design document to go along with your code. Your design document should be split into two sections. The first contains the design of your system, and the choices you made; the second contains an analysis of its security. Your design document should explain your complete solution, for Part 1 up to revocation.

In the first section, summarize the design of your system. Explain the major design choices you made, including how data is stored on the server. The design should be written in a manner such that an average 161 student could take it, re-implement your client, and achieve a grade similar to yours. A well-written design receiving full points need not be longer than two pages. You will lose points if your design is excessively verbose.[5]

The second part of your design document is a security analysis. Present at least three concrete attacks that you have come up with (which were not released with the Part 1 autograder) and how your design protects against each attack. You should not need more than one paragraph to explain how your implementation defends against each attack you present.

You may use as reference the Part 1 design document we provided to you for our reference solution. This is a design document which would receive full credit if we were grading it on Part 1 alone.

Grading: The design document is worth 15 points, split roughly equally between the two sections.

The first section is graded on your ability to explain your design to the reader effectively. Be sure to include the following in the document:

- What state is stored on the server to allow for sharing, as well as the contents of the sharing message.
- What state is changed to revoke a file, and how you meet all the revocation requirements.

The second section is graded on the attacks and defenses you present. You should have at least three attacks and corresponding defenses to get full points. If you give more attacks, we will grade your best three (we will grade them in order, so place your strongest attacks

[5] If after writing your design document, you realize you have a 10-page document with 100 lines of code and think to yourself "My 162 GSI would be proud of this," you will be disappointed in your grade. That is not a design document. That is an implementation with comments.

first to make it easier for the readers, so they can stop after finding three that suffice for full credit). Do not give more than 4.

Submission and Grading

Using `glookup` with `submit proj2-part2`, for Part 2 you should submit your final version of `client.py` with all of the functionality for Part 1 and sharing/revocation and optionally `feedback.txt` by March 19, 2018, 11:59 PM. You will also submit your design document as `design.pdf` on gradescope.

As in Part 1, we provide a set of functionality tests for Part 2, in `run_part2_tests.py`. You should also make sure that you still pass all the autograder tests we gave you for Part 1.

As in Part 1, your score on sharing and revocation will be based on all functionality and security tests from Parts 1 and 2. Your final score for these sub-parts will be the minimum of your functionality and security scores.

We will **not** accept re-grade requests on the autograder results, except in cases where there was a bug in the autograder. If you feel this has occurred, please post a private question on Piazza to instructors and we will look at your code.

Submission Summary

In summary, you must submit the following for Part 2 on glookup:

```
client.py
feedback.txt        (optional)
```

and the following on Gradescope:

```
design.pdf
```

Part 3: Efficient Updates

Make sure you read instructions for Part 3 fully before starting. It is possible that the design of your system for Part 3 will be significantly different from your solution for the previous parts. However, your system must continue to support all of the functionality from Parts 1 and 2.

Part 3 is due April 2, 2018, 11:59 PM.

Efficient Updates

Design and implement a solution for efficiently updating files that are already stored on the server. For this sub-part, you must efficiently handle very large files—potentially multiple gigabytes long. This makes maintaining confidentiality and integrity more difficult. Be aware that when the server is malicious, it can perform arbitrary actions at arbitrary points in time during your execution. For example, you cannot assume that two consecutive calls to `server.get(f)` will return the same value.

The requirements are exactly the same as Part 1 and Part 2, except that now we want your solution to be efficient when making a small update to a large file.

By efficiency, we are referring to the amount of data that must be transferred over the network connection to the storage server. By "update", we are referring (as before) to the case where the user invokes `upload(f, v2)` on a file `f` that was previously uploaded and whose previous contents were `v1`. Your solution only needs to be efficient for updates that replace the file with another file of the same length. You should efficiently handle changes anywhere in the file, from a few bytes to the whole file. (Other kinds of updates, e.g., those that insert data somewhere or delete data somewhere, do not have to be especially efficient.)

Your client may store state (including, for example, the previous version of a file). But, this must only be an optimization: your client must still work correctly if it loses all of its state except for its asymmetric keys for encryption and signatures. If your client loses all of its state, we do not require that it still performs efficient updates; but it must still behave correctly.

If you store state, be careful to validate that your state is **current**. For example, suppose Alice uploads a file with some value, and saves a copy of the file locally. She then shares it with Bob, who updates the file's value. If Alice subsequently wants to update the value on the server, her client must make sure to override Bob's changes (if required) so that the final value on the server is the same as what Alice intends it to be, instead of simply uploading the parts that differ from her local copy. You should be able to still perform efficient updates correctly if other users make updates to different parts of the file.

Keep state in memory—you do not need to worry about serialization to disk. Let S be the total number of bytes of data a client has saved on the server. As long as you keep only $O(S)$

bytes of data in the client, we guarantee our autograder tests will not cause your program to run out of memory.[6]

Grading: In `run_part3_tests.py` we have provided you with a server that counts the total number of bytes you send and receive across the network. We have provided four performance test cases alongside functionality tests for Part 3.

One of performance tests tests algorithmic performance for single-byte updates when not changing the size of the file. You should strive to update in logarithmic size, i.e., the number of bytes transferred on a single-byte update should be logarithmic in the size of the file. However, we will not use this test for grading.

The other three performance tests report the number of bytes transferred and a score based on example thresholds. We will use these three tests for grading the performance of your system. As a guideline, for an efficient implementation, a reasonable amount of data transferred for test `z01_SimplePerformanceTest` is ∼10 KB, and for `z03_SharingPerformanceTest` is ∼100 KB.

You should not worry about $O(1)$ constants in updates until after your code is algorithmically faster (for example, do not worry about keys being hex encoded, or using JSON). These constants award a few points, but much less than the algorithmic portion.

Our autograder will also run additional security tests on your system, as in the previous parts.

Design Document for Part 3

Write a clear, concise design document to go along with your code. The design document should have only one section, explaining your final and complete solution for Part 1 through Part 3.

Write the design document in the same manner as you wrote the first section of the design document for Part 2, such that an average 161 student could take it, re-implement your client, and achieve a grade similar to yours for Parts 1-3.

If your design has not changed from Part 2, you may reuse the first section of your design document from Part 2, but you must submit a new design document per these instructions.

Grading: The design document is worth 10 points. The design document is graded on your ability to explain your design to the reader effectively. Be sure to include (a) how you perform efficient updates, and (b) a short performance analysis.

[6] We introduce this requirement solely for your benefit, so you don't have to worry about memory management. In practice, it would be bad to assume that all of the bytes fit in memory, since very likely it would be possible to store many more bytes on the server than you have room for on your client.

Submission and Grading

For Part 3, you should submit your final version of `client.py` with all of the functionality for Part 1 and Part 2 and the optional `feedback.txt` by April 2, 2018, 11:59 PM, 11:59PM using `glookup` with `submit proj2-part3`. You will also submit your design document as `design.pdf`, plus optional feedback.

You should also make sure that you still pass all the autograder tests we gave you for Part 1 and Part 2.

You can submit your project multiple times. As before, we will grade your latest submission.

As in previous parts, your score on Efficient Updates will be based on the performance of your implementation as well as all functionality and security tests from Parts 1, 2, and 3. Your final score for this sub-part will be the minimum of your performance, functionality, and security scores.

Submission Summary

In summary, you must submit the following for Part 3 on glookup:

```
client.py
feedback.txt        (optional)
```

and the following on gradescope:

```
design.pdf
```

Errata

Update 1 (February 27):

1. Clarification to the IND-CPA game where both files must have contents of equal size.

Appendix

Reference Implementation

We have written an inefficient, insecure implementation of a client. We have provided this to you in `insecure_client.py`. This code is also an example of how to extend the `BaseClient` class. This implementation provides all the functionality requirements of this project, but has no security properties at all.[7]

This client gives each user their own "namespace" within the master server by concatenating the username, a slash, and then the filename and using that as the `id` for the storage server.

The client works by maintaining two types of objects on the server storage: pointers and data. A data object has the contents of a file. A pointer acts as a reference to the file. (If you've taken operating systems, you can think of pointers as symlinks.) When a user updates a file that is a pointer, she follows the pointers until a data file is reached, and then updates the corresponding data file. Sharing works by providing the other user with a pointer to the file, and revocation removes the pointer. This satisfies the revocation properties that sub-children are also revoked.

Example Workflow

An example workflow for developing on your personal machine is as follows:

- Copy the framework code into a folder named `project2`.
- Sync changes to your class account using `scp`:

  ```
  scp -r project2/ cs161-xxx@hiveXX.cs.berkeley.edu:~/project2
  ```

  This will copy the `project2` folder and its contents to your home directory.
- SSH into a Hive machine with your class account.
- Do all Python console work using `python3` or `ipython3` in your SSH session.

[7] While `insecure_client.py` will pass all functionality tests, it does not satisfy any of the security requirements, so submitting it will earn you a score of 0 on the project. As detailed in each part's "Submission and Grading" section, your score is the minimum of your security, functionality, and (in part 3) performance scores, so a security/performance score of 0 will earn you 0.

- Run all Python code using python3, e.g. `python3 client.py`.

You can also do all of this while seated at a machine in the instructional labs, or inside an SSH session using vim or emacs.

While not officially supported by course staff, it is also possible to set up Python3 and install PyCrypto on your own machine. You should double check your code by re-running the functionality tests against your code on Hive. **Remember to follow all the steps in the submission instructions for each part of this project!**