Test Plan

Overview

On a high level, we automatically tested most extensively the code furthest away from the user. This included data structures like tries and multisets, utility functions like substring, and complex procedures like calculating edit distance or memoization. Since the functionality of these modules is fairly separated from their usage in the application, we were not convinced that manual testing through the UI would adequately cover all facets of their implementation. Consequently, we also ensured that these modules in particular had very high (near or at 100%) test coverage. While we did test other modules significantly, we were more lax about test coverage, instead preferring manual integration testing to ensure the feature worked well. Persistence, for instance, would have been difficult to properly unit test, because the only strings we ever persist are the outputs of our serialization functions. To properly test the real usage of Persistence, we need to test it in the context of the system by actually storing and retrieving logins and passwords. This is even more true for any functions in bin/, of course, as all of those rely directly on user input.

More details by module:

- GUI/CLI
    - These were fully tested manually, since we had no ability to test them in an automated manner. We ran the program, pressing all buttons and running all commands to test that the program doesn't crash or show obviously buggy behavior. In this sense, it appears to be correct for the user.
- Types
    - These functions were only used for debugging or display to the user, so we only tested them manually.

- Autocomplete testing
    - Tests compare_words function of autocomplete used within autocomplete and elsewhere
        - Ensures correctness by randomly generating 10 words, shifting each character in each word up by 10, and ensuring that the previous and final words don't match (as they would have 0 characters in common)
        - Multiple trials and randomness ensure reproducible, generalizable results
    - Checks if a randomly generated word and a 3 letter substring of it pass compare_words

        - Finally, checks if autocomplete works properly (finding 3 letter shared word from “pass” among randomly generated passwords and login, returning list of valid ones)

- Persistence testing

    - Note: At the top of the file, we use a functor to replicate all the functions of persistence.mli, because we change the read/write paths to the test data directory and the files to test_unencryptables and test_encryptables.

    - In terms of testing, we test reading and writing an encryptable password, then another encryptable login (ensuring both the password and login are readable), then deleting one of them and ensuring only the other is there.

        - Finally, the test file tests writing and reading an unencryptable aka Masterpassword

    - This basically tests every possible order of using persistence functionality, ensuring data is stored properly and file permissions via chmod are valid regardless of order the user does things in.

- GenPassword testing

    - The testing is split into two test lists, one called “tests”, and one called “qcheck_tests”

    - One helper function was made, called gen_20_passwords, which effectively generates 20 passwords both with or without special characters (0-50 character length)

        - This is used in tests, to ascertain none of the 20 passwords are the same as each other

    - Two other tests in “tests” ensure that passwords with special characters do contain some special characters, and passwords without them don’t (to ensure the respective functions are working correctly).

        - The final two tests ensure the passwords have the right length

    - This testing ensures correctness as first of all, it ensures randomness (if they work correctly, it is statistically very unlikely for the tests to fail, given the possible number of passwords)

        - Moreover, it ensures the respective functions use special characters properly with correct total number of characters

    - In improving the implementation of GenPassword, we created a bug in which the generated password would only consist of one character (which we fixed). Thus,

as part of regression testing, we included QCheck tests that would make sure this doesn't happen again (with very high probability).

- Util testing
    - These functions were tested in a mostly ad-hoc manner, but with an eye for property testing. All of the functions tested had the most important parts of their equational specification tested on randomized inputs, giving us faith that they work over their full domain.
    - We didn't explicitly aim for code coverage at first, but confirmed with bisect that these functions were fully covered and added a couple of tests as necessary when that was not true.
- Edit distance
    - Because edit distance is a relatively complex procedure, we wrote many tests for it.
    - This included a few deterministic tests as a sanity check. They also served to document exactly the function's intended functionality—when we were writing the module, these deterministic tests were the ones we ran in utop to make sure the function worked correctly (before moving on to a full test suite).
    - The test file also included several randomized tests, checking various properties that one might reasonably expect from an edit distance measure. Special care was taken to consider edge cases, such as empty strings.
- Memoization
    - Because memoization was only used in our project to support EditDistance, we were confident that edit distance tests covered the module fairly well. Nonetheless, we wrote some tests (using the case study of fibonacci) to confirm the only effect of memoization is a performance boost.
    - We did explicitly test performance (since that is the entire purpose of the module), checking that we can calculate the 1000th fibonacci number in a reasonable amount of time, which would take far longer than the lifetime of the universe to compute without memoization.
- Data structures (Trie/OrderedMultiset)
    - We relied heavily on an equational specification for our data structures. Since, like edit distance, these are nontrivial pieces of code, we tested them extensively.
    - The purpose of including a Trie in our code was to speed up our password strength checker. We did not include an automated performance test for this (since it is less black-and-white than fibonacci and we didn't have the tools to

adequately control our environment). Nonetheless, we manually confirmed that the password-strength checker returns imperceptibly quickly using a Trie whereas it takes several seconds with a list.

    - The purpose of the OrderedMultiset was memory efficiency; this again we were not able to automatically test, but we did note that in testing it (along with changes to Trie to use the multiset) reduced our memory footprint by ~4-6x when loading 10M weak passwords to check against.

- StrengthCheck

    - StrengthCheck did not get as many tests as it was relatively simple and easy to manually test. However, we did confirm the main functionality in automated tests as prevention for regressions.

    - Since loading the list of weak passwords at startup can take a while, we do it concurrently—as part of this, StrengthCheck includes some functionality to confirm if it has been fully initialized yet. Because we did not want our tests to be lengthy (slowly loading a large weak password file), we did not test this; however, it was clear through manual testing that this (relatively simple) feature worked well.

    - Because, as mentioned, the full password list was very large and takes a while to load, we ran these tests with a much smaller subset (only 100 weak passwords) which we believed to check the same fundamental functionality.

- Encrypt testing

    - The test suite was constructed with black box randomized tests in mind.

    - Obviously, the most important thing to test with encryption is that it is reversible; specifically, that there exists a decrypt function that is a left inverse of the encryption function. Since the password manager only uses symmetric encryption, we want to ensure that encrypting and then decrypting with the same key returns the original plaintext. Thus, we used QCheck to generate random tests that would check if for an arbitrary plaintext (string), encrypting the plaintext and then decrypting the result returns the original plaintext.

    - Another thing that was important to test was if the key is changed after encryption but before decryption, the decrypted ciphertext should not be the same as the original plaintext. This is important for security reasons, because it would be bad if an adversary could decrypt your passwords without knowing the real key. Once again, we used QCheck to generate random tests to check this property for arbitrary plaintexts.

    - We also wanted to check that the encryption function is reasonably secure enough; since the plaintext consists of names, usernames, passwords, and urls,

we simply defined this to be that the ciphertext cannot contain any of these fields as a substring. Once again, this property was tested randomly using QCheck.

    - These three kinds of tests ensured what we believe to be the most important properties of successful encryption: left invertibility, the importance of knowing the actual key, and basic ciphertext security. Thus, we believe that these tests show reasonable correctness.

    - At least for now, we purposefully did not emphasize glassbox testing. Since the encryption functions are mainly calls to Cryptokit ciphers, we believed that glassbox testing would be essentially testing the coverage of the cipher implementation, which is not our responsibility.

    - Furthermore, we did not check that the actual ciphertext matches the expected ciphertext given our encryption function. This is mainly for two reasons. One, due to salting and hashing the key, as well as padding the plaintext, it is very difficult to know what the key is nor even the plaintext, let alone the expected ciphertext. Two, although we used the AES-256 cipher, we did not want to restrict our tests under the assumption that the encryption function implemented this particular cipher. What if we wanted to change the cipher implementation? We may actually plan to continue updating this program in our own time, where we could add more ciphers, so we didn't want to rely on one particular cipher for our test-writing.

- Serialization testing

    - The most obvious randomized blackbox test that came to mind was to ensure that our serialization of passwords and logins was reversible. In theory, we would need to check that serialization is bijective, but testing one of the directions would involve creating specific jsons, which we deemed too difficult for us at the moment. Thus, we used QCheck to test that converting an encryptable password or login into a string and back returns the original data on random passwords and logins.

    - Although those are by far the most important tests for serialization, we also wrote some deterministic tests to be more confident that our serialization worked correctly. In the early stages of development, we took a naive approach to serialization that would break if the encryptables included commas, so after improving serialization, we decided the best course of action was to test serialization on a password and login that had commas.

    - Since json uses brackets, we also wrote a similar test for brackets to further justify the correctness of serialization. Just for good measure, we also included a test for quotes and a test for escape characters.

    - It was too difficult for us to check if a serialized password or login matches its expected result, since that would involve having to create a json ourselves. For

this same reason, we also did not check the functions that involved converting encrypted data to and from jsons.

- MasterPassword testing
    - We once again used randomized blackbox testing to check the function that salts and hashes strings. Since the actual implementation of the function only consists of a call to the Bcrypt module, it wouldn't make much sense to perform glass-box testing. We used blackbox testing to ensure that the salting and hashing function fulfills certain properties to ascertain security.
    - A reasonably salted and hashed string should not contain the original string (for reasonably long enough strings) as a substring, so we used QCheck to check this property on arbitrary strings.
    - Trivially, salting and hashing two different strings should result in different outputs, so we tested this as well.
    - More interestingly, salting and hashing the same string twice should actually produce two different outputs, since this is the main purpose of salting. Thus, we generated random tests for this property as well.
    - We found it too difficult to test the check_master_pwd function, as the implementation involves reading the salted and hashed master password in the data file and comparing it to the argument string. Not only is it difficult to test anything that involves reading from or writing to a file, but also if the master password were ever changed, such tests would break.