The University of Melbourne

SWEN90010: High Integrity Systems Engineering

Assignment 3

Due Date: 11:59pm, Friday June 5th 2020 (Melbourne time)

1 Introduction

The assignment is worth 20% of your total mark and is done in pairs (the same pairs as assignments 1 and 2).

The aim of this assignment is to use the SPARK Ada toolset to implement and verify the correctness and security of a small command-line Password Manager utility (similar to the one that was the focus of Assignment 2 in *SWEN90006 2019*).

The Password Manager's behaviour has changed quite a bit from SWEN90006. tl;dr: do read

That is, your pair will implement the functionality of the password manager program (as specified below) in SPARK Ada, and use the SPARK Prover to prove that it is free of runtime errors and, additionally, that it is also secure.

As usual, get started early and use your pair to maximum advantage.

Download, install and check you can run the GNAT tools (see Section 3.1) ASAP!

2 The Password Manager

The program you have to implement is a command-line utility for managing website passwords. It takes input from the terminal (i.e. standard input, aka stdin).

2.1 Commands

Each line of input is a *command*. Commands conform to the following grammar:

<COMMAND> ::= "get" <NAME>
| "rem" <NAME>
| "put" <NAME> <NAME>
| "unlock" <NUMBER>
| "lock" <NUMBER>

Tokens in the grammar above are separated by one or more *whitespace* characters, namely space, tab, and line-feed. Each <NAME> is a string of *non-whitespace* characters. <NUMBER> is a 4-digit string of non-whitespace characters that represents a non-negative number (i.e. a natural number) in the range 0000 . . . 9999.

- The password manager can be in one of two states, either *locked* or *unlocked*.
- When the user starts the password manager, they supply (via a command-line argument) a 4-digit string *masterpin* that represents a 4-digit PIN (i.e. a number in the range

0000 . . . 9999), which is the *master PIN* needed to unlock the password manager. If no master PIN is supplied, the password manager should exit immediately.

- The password manager begins in the locked state.
- For a string *pin* that represents a 4-digit PIN, the command "unlock *pin*" does nothing when the password manager is in the unlocked state. Otherwise, it checks whether *pin* is equal to the master PIN and, if so, changes the state of the password manager to unlocked. If *pin* is not equal to the master PIN, then the state of the password manager is not changed.
- For a string *newpin* that represents a 4-digit PIN, the command "lock *newpin*" does nothing when the password manager is in the locked state. Otherwise, it updates the master PIN to become *newpin* and changes the state of the password manager to locked.
- For a URL string *url*, the command "get *url*" does nothing when the password manager is in the locked state. Otherwise it looks up *url* in the password database and then prints out the password stored for this URL, if any (and otherwise prints nothing).
- For a password string *password*, the command "put *url password*" does nothing if the password manager is in the locked state. Otherwise, it updates the password database to add a mapping for the URL *url* to map to the password *password*.
- The command "rem *url*" does nothing if the password manager is in the locked state. Otherwise it removes from the password database any mapping associated with the URL string *url*.

2.2 Changes from SWEN90006

- The "save" and "list" commands have been removed.
- The password manager only takes input from the terminal (i.e. from stdin).
- The master password has been replaced by a 4-digit PIN called the *master PIN*.
- When the password manager is started, the user supplies on the command line the master PIN, as a command line argument.
- PINs are 4-digit strings in the range 0000 . . . 9999.
- The password manager can be in one of two states: either *locked* or *unlocked*.
- The "unlock" and "lock" commands have been added, which change the state of the password manager between locked and unlocked. The "lock" command allows updating the master PIN (see above).
- Passwords longer than 100 characters are invalid.
- URLs longer than 1024 characters are invalid.
- PINs that are not 4-digit strings in the range 0000 . . . 9999 are invalid.
- Input lines (i.e. commands) longer than 2048 characters are invalid.
- When receiving invalid input, the password manager should exit immediately (possibly after printing an appropriate error message).

2.3 A Demo Session with the Password Manager

The user supplies the initial master PIN when starting the application, which is called `main`. For example, to start the application setting the master PIN to "1234" the user would run:

```
$ ./main 1234
```

The password manager accepts commands from the user and its prompt indicates whether it is in the locked or the unlocked state. Initially, it is always locked. Here is an example session showing its expected output for valid commands.

```
$ ./main 1234
locked>   unlock 1234
unlocked> put http://google.com test_password
unlocked> put http://apple.com  another_password
unlocked> get http://google.com
test_password
unlocked> lock 2345
locked>   unlock 1234
locked>   unlock 2345
unlocked> get http://apple.com
another_password
unlocked> rem http://apple.com
unlocked> get http://google.com
test_password
unlocked> get http://apple.com
unlocked> lock 1234
```

3 Your tasks

Get started early. This assignment is worth 20 marks in total.

3.1 Task -1: Download and Install GNAT Community Edition

Download and install GNAT Community Edition from: `https://www.adacore.com/download`.

Ensure that the `bin/` directory is in your `PATH` so that you can run the Ada tools directly. If your setup is correct, you should be able to run commands like `gnatmake` and `gnatprove` and see output like the following (noting that in this case the commands were run on MacOS):

```
$ gnatmake --version
GNATMAKE Community 2019 (20190517-83)
Copyright (C) 1995-2019, Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

$ gnatprove --version
2019 (20190517)
```

```
Why3 for gnatprove version 1.2.0+git
/Users/toby/opt/GNAT/2019/libexec/spark/bin/alt-ergo: Alt-Ergo version 2.3.0
/Users/toby/opt/GNAT/2019/libexec/spark/bin/cvc4: This is CVC4 version 1.7.1-prerelease
/Users/toby/opt/GNAT/2019/libexec/spark/bin/z3: Z3 version 4.8.0 - 64 bit
```

3.2 Task 0: Downloading and Building the Helper Code

You need to implement the password manager in SPARK Ada. However you are provided with some helper code to get you started.

Download the ZIP file containing the helper code from the LMS. It contains a small number of Ada packages:

- `main.adb`: a top level main program that shows basic usage of the other packages. You will replace this with the top level of your password manager implementation.
- `MyCommandLine`: Provides a simple SPARK API for accessing command line arguments.
- `MyString`: Provides a simple SPARK abstract data type for strings. Used for representing lines of input as well as URLs and passwords.
- `MyStringTokeniser`: Provides a simple SPARK interface for tokenising strings. By "tokenising" we mean to break a string up into its various whitespace-separated tokens (where each token contains no whitespace).
- `PasswordDatabase`: Provides a simple SPARK API for a database that maps URLs to passwords.
- `PIN`: Provides a simple SPARK abstract data type to represent 4-digit PINs in the range 0000 . . . 9999.

Besides `main.adb` don't modify the other supplied code, excpet for adding comments.

After unpacking the ZIP file, it will create the directory `assignment3` in which the Ada code is placed. You can build the code by running '`gnatmake` main' in that directory.

```
$ gnatmake main
gcc -c main.adb
gcc -c mycommandline.adb
gcc -c mystring.adb
gcc -c mystringtokeniser.adb
gcc -c passworddatabase.adb
gcc -c pin.adb
gnatbind -x main.ali
gnatlink main.ali
```

Note: if you are building the code on MacOS, you might get the warning message:

```
ld: warning: URGENT: building for OSX, but linking against dylib
(/usr/lib/libS ystem.dylib) built for (unknown). Note: This will be
an error in the future
```

This warning can be safely ignored.

Building the code should produce the placeholder `main` that you can then run. As mentioned, the supplied main code simply shows some examples of how to use the other supplied packages.

3.3 Task 1: Understanding `MyStringTokeniser` (3 marks)

Your first task is to understand the `MyStringTokeniser` package. This package has no comments but it does have SPARK annotations which describe aspects of its central procedure `Tokenise`.

You should read and modify the provided `main.adb` program: in particular the parts of it that use the `MyStringTokeniser` package. This will help you to get an idea of what this package is doing.

You should then carefully read the SPARK annotations on the `Tokenise` procedure.

You need to:

1. Add comments to the file `mystringtokeniser.ads` describing each part of the postcondition of the `Tokenise` procedure. For each part of its postcondition, you need to describe what that part is saying and why it is necessary to have it as part of the postcondition.

   Hint: think about code that uses this package. Try removing parts of the postcondition and run the SPARK prover over code that uses this package to see what happens.

2. Add comments to `mystringtokeniser.adb` that explain the loop invariant for the `Tokenise` procedure and, in particular, why the following part of the loop invariant is necessary:

   ```
   (OutIndex = Tokens'First + Count);
   ```

3.4 Task 2: Implementing the Password Manager (6 marks)

Using the provided code, implement the password manager as specified in this document.

Your implementation should follow good software engineering practices regarding modularity, information hiding / abstraction, loose coupling, and so on.

Therefore, you are strongly encouraged to decompose the core operations of the password manager into a separate Ada package that your `main.adb` can make use of.

3.5 Task 3: Proving it Free of Faults (5 marks)

Your next task is to use the SPARK Prover to prove that your password manager is free of runtime errors, and that all pre-conditions on the provided code are always satisfied.

To do this, you will likely need to write loop invariants, and add defensive checks to your code.

Your goal is to have a working password manager implemented that, when somebody selects *SPARK* $\rightarrow$ *Prove All* in GPS, does not produce any warning or error messages from the SPARK Prover.

3.6 Task 4: Proving it Secure (6 marks)

Your final task is to use the SPARK Prover to prove that your implementation is secure. By "secure" we mean it should *at least* satisfy the following security properties (however this list is intentionally not complete—to get full marks here you will also need to think of and prove additional security properties):

- The Get, Put, Remove and Lock operations can only ever be performed when the password manager is in the unlocked state.
- The Unlock operation can only ever be performed when the password manager is in the locked state.
- The Lock operation, when it is performed, should update the master PIN with the new PIN that is supplied.

By "performed" we mean that the operation has executed. This is different to an operation *attempting* to be executed. For instance, in the following example session the user attempts to perform the Lock operation; however the operation is *not* executed, because the password manager was in the locked state.

```
$ ./main 1234
locked>   lock 2345
locked>
```

For this task you need to **write a short description (no more than one page)** describing:

- The security properties that you proved of your implementation and, for each,
- How you specified the security property using SPARK annotations and how the annotations encode the security property.

If you prove additional properties of your implementation, besides those mentioned above, you should document those in your report.

Your report should be written as Ada comments at the top of your `main.adb` file.

4 Submission

You should submit a ZIP file containing your SPARK code. Your code should live inside the `assignment3/` directory in your ZIP file. We expect to be able to build and run your code by doing the following from the command line, assuming your submission is called `submission.zip`:

```
$ unzip submission.zip
$ cd assignment3/
$ gnatmake main
$ ./main 1234
```

Your `main.adb` should inclue comments that clearly identify *both* authors in your pair.

Your code should build and run against the original packages you were supplied with. However your submission should contain updated versions with (only) additional comments added, as required (see above).

Late submissions Late submissions will attract a penalty of 2 marks for every day that they are late. If you have a reason that you require an extension, email Toby *well before the due date* to discuss this.

Please note that having assignments due around the same date for other subjects is not sufficient grounds to grant an extension. It is the responsibility of individual students to ensure that, if they have a cluster of assignments due at the same time, they start some of them early to avoid a bottleneck around the due date. The content required for this assignment was presented before the assignment was released, so an early start is possible (and encouraged).

5 Academic Misconduct

The University misconduct policy applies to this assignment. Students are encouraged to discuss the assignment topic, but all submitted work must represent the pair's understanding of the topic.

The subject staff take plagiarism very seriously. In the past, we have successfully prosecuted several students that have breached the university policy. Often this results in receiving 0 marks for the assessment, and in some cases, has resulted in failure of the subject.