Input validation Report

# Build/Run Instructions

## Important config variables:

* Log4net <file value=”log.txt” /> log.txt is at the executable’s location
* Postman collections variable “BaseURL” change port# to match docker connection.

## Container Setup

There is a provided phonebookimg.tar file. This is an export of a release configuration of the development environment. Execute the Run.bat batch file included to run the API. This batch file loads, runs, and executes the API. docker load can take a long time sometimes.

1. docker load -i “%cd%/phonebookimg.tar”
2. docker run -dt -v "%cd%:/root/.microsoft/usersecrets:ro" -v "%cd%:/root/.aspnet/https:ro" -e "ASPNETCORE\_ENVIRONMENT=Development" -e "ASPNETCORE\_URLS=https://+:443;http://+:80" -P --name PhoneBook --entrypoint tail phonebook -f /dev/null
3. docker exec -d PhoneBook dotnet PhoneBook.dll

## Postman Unit Testing:

The postman\_collection, Test\_Accept.csv, and Test\_Fail.csv files are included to execute the unit tests for the API. Open the postman desktop/web application and import the collection. Start the API and check what port# it is connected to. Change the collection variable “BaseURL” accordingly. Now open the runner in postman and select one of the csv files. The recommended order is deleteAll, list, add, and then either deleteByName or deleteByNumber. If both are selected, then one will fail due to the previous deleting the entry already. If deleteAll is not first, then previous entries might cause add to fail due to duplicates. Use this configuration for Test\_Accept. Test\_Fail is very similar but exclude the deleteByName or deleteByNumber functions because the entry failed the input validation. This means that there is no entry to delete.

The log file is PhoneBook.log, check to see that logs are being made properly.

# Code Description

The PhoneBook REST API implements input verification for all inputs, log4net for logging changes and requests, and SQLite for persisting the entries to disk.

The code validates the input by checking the name and phone number against a list of regular expressions before any actions are done with the input. This is done with the Validation Service class which provides two functions: ValidName(string name) and ValidNumber(string number). Both functions check the given string against the list of regular expressions in \_namePatterns and \_numberPatterns respectively. This implementation lets other classes use the two functions to easily validate data. It also keeps the regular expression values hidden from other classes and from possible attackers.

Log4net is the logging framework that was chosen to write to a log file. This generates a detailed history of the log entries’: date, level, method, and the input. Any successful entry to the API is logged as INFO and any input validation failure is logged as WARNING. While the info logs are required, I added the warning logs so there is a history of possible input validation attacks.

SQLite is implemented to keep the values of the entries between runs. This is done by implementing the EntityFramworkCore.Sqlite framework. All of the functions used are predefined such as add, delete, find, set. These are parameterized queries and prevent the data from being interpreted as code. Otherwise, the rest of the implementation is standard from what tutorial I saw.

# Regular Expression Design

I implemented the name regular expressions to reflect the three versions given in the document. There is a base regular expression for a string that can be a first, middle, or last name. This base regular expression is then inserted into the regular expression that are used to validate the data.

I implemented the phone number regular expressions to be as specific to the requirements as possible. Instead of having multiple different formats for the different formats provided, I made one format that has optional fields when applicable. Additionally, I used repetition specifiers to only allow the specific number of digits defined in the assignment description.

For the name and phone number regular expressions, I only allowed letters for names and digits for phone numbers. This further constrains the format and makes attacks very difficult. Throughout the design process, I tried to allow what is required and no less.

## Miscellaneous notes

A deleteAll function is provided to simplify clearing the database. Instead of manually getting the list and deleting all entries, deleteAll does it in one action. This is useful for unit testing.

# Assumptions

I am assuming that we are not allowed to use an external phone number validator because the formats might be slightly different for this assignment. There is a google phone number regex library port for c# that is available, but I am actively avoiding it.

Some of the phone numbers provided in the assignment description should be valid but are said to be invalid. vice versa as well. As I was researching phone number formats, there are disagreements between real world phone number format rules and the examples the professor provided. For example, 011 701 111 1234 is supposedly a valid phone number but 011 means the next number is an international prefix and 701 does not exist. 701 is an area code for north Dakota. I am going to ask the professor about this but I’m not nit picking the edge cases as much as I would. 1(703)123-1234 also doesn’t have a plus sign or 011 or 00 so its invalid strictly speaking. I would prefer a set of rules to implement instead of having to guess what the professor wants. I made my regular expressions to line up with the given examples.

Because checking whether the provided phone number has valid data is difficult, I’m assuming the user will be inputting their phone number which is legitimate and not just random numbers in the correct format. To clarify, I’m still checking the format and whether it matched one of the provided phone number formats, but some international prefix codes do not exist. Some area\city codes don’t exist. I believe this level of validation is above the scope of this project and should be implemented by the organizations that make these standards.

I’m assuming that – is the same as a space for names. Additionally, I am assuming ‘ is allowed anywhere in the name, just not two or more in a row.

I’m assuming country code can’t start with a 0 because of an example given in the document.

# Pros/Cons

Because I used a SQLite framework to save the entries, the functions provided by this framework doesn’t require forming your own queries. This means that the framework used has simple ‘parameterized’ queries that have already been precompiled for you. Using these functions to interact with the database forces the data to be treated as data and eliminates the possibility of an SQL injection attack.

The implementation of the logging system used logs both the successful and unsuccessful API requests. This allows the admin of the API to see the changes to the backend database, along with possible attempts to attack the API. With the information from these possible attempts, the admin can make sure that the API is secure against the attacks. This benefits the visibility of attacks but also causes more information and clutter to fill the logs. The information is logged in the controller, but more information could be logged in the DictionaryPhoneBookService if the logger was used there. This information could include what regex the input was matched against.

The input validation checks the phone numbers against multiple possible regular expressions. This is very useful for checking the format of the provided input, but not whether the data in the input is valid. For example, there are country codes that don’t exist. My implementation prevents all legitimate attacks, but some invalid phone numbers can be inserted. The regular expressions that I created are made to match many types of formats so they can be hard to read at times. This reduces the number of regular expressions to check and reduce redundancy but are harder to match from a human perspective.