Instructors: Matt Fredrikson, Jean Yang TA: Samuel Yeom

Due date: 2/2/2017 at 11:59pm

# Assignment 1

## 1 Build a basic data server (30 points)

Your task is to implement a data server with a basic security policy. Clients should be able to connect to your server via TCP, send a program written in the server's scripting language, and view the results as a JSON string. Programs can store data on the server, which can later be retrieved by other programs that are executed. The server does not persist data across executions, so when its process is terminated, all existing data becomes inaccessible when the server is later restarted.

Below is a detailed specification of what your server is expected to achieve. *Read this carefully, as you will be graded against a test suite developed by the course staff to determine compliance with the spec!* In particular, be sure to hand in a solution containing a `Makefile` that, when run, produces binary executables `server` and `interp`. Note also that your implementation must be done in OCaml. You are encouraged to use the scaffolding code provided by the course staff (see details in the **Code Handout** section).

Please note that while this specification may look daunting in its length, much of the code needed to meet the spec, particularly with regards to parsing and output formatting, has been provided. The intent of this document is to provide you with a sufficient level of detail to successfully implement a secure initial server, and to design a robust test suite.

*Note: The specification is based on that given at `https://builditbreakit.org/static/doc/fall2016/index.html`. You might find it helpful to consult that page for further examples of the language, but be aware that the language described in the current document is not identical to that used in the Build It, Break It, Fix It competition, nor are the requirements for your server. If you feel that there is ambiguity in the spec, post a question to Piazza.*

### Server Specification

**Running the server.** The server should run by executing the command:

```
./server PORT [PASSWORD]
```

This command initates a fresh server that listens for incoming TCP connections on port `PORT`, and (optionally) initializes the administrator password to `PASSWORD`.

The server should only halt when directed by the administrator via the `exit` command (details in a later section). We assume that the hardware and operating system on which the server is running are fault-free.

**Command-line interpreter.** To facilitate testing and debugging, the core functionality of the server should be accessible through a non-network interface. This interface is contained in the binary `interp`, which can be invoked in either of two ways:

```
./interp
```

When `interp` is given no arguments, it accepts a program from `stdin`, executes it, and terminates.

```
./interp PROGRAM1 [PROGRAM2] ...
```

When given command line arguments for the names of files containing programs, `interp` executes them in the order given, then terminates.

**Arguments and return codes.** The server should impose the following restrictions on command line arguments:

- Command line arguments cannot exceed 4096 characters each.
- The port number should be between 1024 and 65535, in decimal without leading 0's.
- The password may contain alphanumeric characters, spaces, commas, semi-colons, periods, question marks, exclamation points, hyphens, and underscores, but *not* tabs or newlines.

The server should implement the following conventions for return codes:

- When invalid command-line arguments are received, the server should terminate with status code 255.
- When the server cleanly terminates, it should return status code 0.
- When the port number given on the command line is taken, the server should exit with code 63.

## Scripting Language: Syntax

A script always begins with a line identifying the principal and their password. For example, the following line identifiers the administrator, attempting to login with the password "admin". Each subsequent line contains a *primitive command* that is executed on behalf of the principal. Scripts always end with three asterisk characters on their own line.

The context-free grammar below is a complete description of the scripting language. Each rule consists of a non-terminal identifier, given in bold-face font and surrounded by angular brackets (e.g., ⟨**cmd**⟩), followed by one or more production rules. Elements in typewriter font (e.g., `exit`) are terminals, and correspond to keywords and punctuation required by the language. Elements in italics are tokens, and we use the following convention:

- $s$ indicates a string constant surrounded by a pair of double quotes. String constants may contain alphanumeric characters, spaces, commas, semi-colons, periods, question marks, exclamation points, hyphens, and underscores, but *not* tabs or newlines.
- $x, p, q, r, y$ indicates an identifier. Identifiers must be unique from any of the keywords (typewriter-face non-terminals in the grammar below) in the language, must start with an alphabetic character, and may contain alphanumeric characters in addition to underscores.

- `\n` refers to the newline character (note: `\r` is not supported).

The language has the following limits on the length of various entities. Programs that fail to match the grammar, or violate any of the limits, are non-compliant and must result in failure. These failures take precedence over security violations in the program. The server should indicate such a failure by returning a {"`status`" : "`FAILED`"} response.

- Programs must consist of no more than 1,000,000 ASCII (8-byte) characters.
- String constants must consist of no more than 65,535 ASCII (8-byte) characters.
- Identifiers must consist of no more than 255 ASCII (8-byte) characters.

| | | |
|---|---|---|
| ⟨**prog**⟩ | ::= | `as principal` $p$ `password` $s$ `do \n` ⟨**cmd**⟩ $***$ |
| ⟨**cmd**⟩ | ::= | `exit \n` \| `return` ⟨**expr**⟩ $\backslash n$ \| ⟨**prim_cmd**⟩ `\n` ⟨**cmd**⟩ |
| ⟨**prim_cmd**⟩ | ::= | `create principal` $p$ $s$ |
| | \| | `change password` $p$ $s$ |
| | \| | `set` $x$ $=$ ⟨**expr**⟩ |
| | \| | `append to` $x$ `with` ⟨**expr**⟩ |
| | \| | `filtereach` $y$ `in` $x$ `with` ⟨**expr**⟩ |
| ⟨**expr**⟩ | ::= | ⟨**value**⟩ \| `[]` \| {⟨**fieldvals**⟩} |
| | \| | `if` ⟨**bool_expr**⟩ `then` ⟨**expr**⟩ `else` ⟨**expr**⟩ |
| ⟨**bool_expr**⟩ | ::= | `true` \| `false` \| `equal`(⟨**value**⟩, ⟨**value**⟩) \| `notequal`(⟨**value**⟩, ⟨**value**⟩) |
| ⟨**fieldvals**⟩ | ::= | $x =$ ⟨**value**⟩ \| $x =$ ⟨**value**⟩, ⟨**fieldvals**⟩ |
| ⟨**value**⟩ | ::= | $x$ \| $x.y$ \| $s$ |

## Semantics

The state of the system is divided into the *store* and *security state*. The store is a mapping from variable identifiers to literal values. The security state is a mapping from variable identifiers to access rights, as well as a mapping from principal identifiers to password strings. The system is preconfigured with principal admin whose password is given by the second command-line argument; or "admin" if that password is not given.

**Transactional execution.** Until all commands in the body are executed, the server must maintain a transaction containing all updates to state made by the program. Whenever the server returns either `FAILED` or `DENIED` status, all changes to state made by the current program are aborted. If the program finishes without returning either of these statuses, then the server should commit the program's state updates so that programs executed subsequently can see the updates.

**Programs.** A program `as principal` $p$ `password` $s$ `do \n` ⟨**cmd**⟩ $***$ has the following semantics:

- Fails with {"`status`" : "`FAILED`"} if $p$ is not a principal in the system.
- Returns {"`status`" : "`DENIED`"} if $s$ is the wrong password for $p$.
- Otherwise, executes ⟨**cmd**⟩ under the authority of $p$, terminating the connection afterwards.

**Commands.** A command consists of zero or more primitive commands, ending with either `return` or `exit`. Each successfully executed command (including primitive commands described below) results in a JSON-formatted status message being returned to the client. We describe the semantics of primitive commands below, and the two terminating commands here:

- To execute `exit`, the server returns {"`status`" : "`EXITING`"}, terminates the client connection, and halts the server with status code 0. This command is only allowed to execute on behalf of `admin`. If a principal other than `admin` attempts to execute it, the server returns {"`status`" : "`DENIED`"}.
- To execute `return`⟨**expr**⟩, the server evaluates the expression, returns status "`RETURNING`", and the JSON representation of the result keyed under "`output`".

**Primitive commands.** Primitive commands typically comprise most of the body of a program. Commands may include expressions, whose semantics are given below. If an expression fails or results in a security violation, then the containing expression does as well.

- `create principal` $p$ $s$. Updates the security state to include a new principal $p$ with password corresponding to the string constant $s$.

  **Failure conditions:**
  – $p$ already exists as a principal

  **Security violations:**
  – Current principal is not `admin`

  **Successful status code:** `CREATE_PRINCIPAL`

- `change password` $p$ $s$. Updates the security state to set principal $p$'s password to $s$.

  **Failure conditions:**
  – $p$ does not exist as a principal

  **Security violations:**
  – Current principal is not `admin` or $p$

  **Successful status code:** `CHANGE_PASSWORD`

- `set` $x$ = ⟨**expr**⟩. Updates global variable $x$ in the store to map to the result of evaluating ⟨**expr**⟩. If $x$ does not already exist, this command creates it.

  **Failure conditions:**
  – Evaluating ⟨**expr**⟩ results in failure.

  **Security violations:**
  – Current principal does not have `write` permissions on $x$.

  **Successful status code:** `SET`

- `append to` $x$ `with` ⟨**expr**⟩. If $x$ evaluates to a list, then the result of evaluating ⟨**expr**⟩ is concatenated to it.

  **Failure conditions:**
  – $x$ is not defined.

– Evaluating $x$ results in a value other than a list.

**Security violations:**

– Current principal does not have either `write` or `append` permissions on $x$.

**Successful status code:** `APPEND`

- `filtereach` $y$ `in` $x$ `with` ⟨**bool_expr**⟩. For each element $y$ in list $x$, evaluate ⟨**bool_expr**⟩ with $y$ bound to the current element of $x$. If the expression evaluates to *false*, then remove it from $x$. Otherwise, it remains in $x$.

  **Failure conditions:**

  – $x$ does not exist.

  – Evaluating $x$ results in a value other than a list.

  – $y$ is already defined.

  – If any evaluation of ⟨**expr**⟩ results in a failure, then this command does.

  **Security violations:**

  – Current principal does not have `read` and `write` permissions on $x$.

  – If any evaluation of ⟨**expr**⟩ results in a security violation, then this command does.

  **Successful status code:** `FILTEREACH`

**Expressions.** Expressions evaluate to strings, lists, or records.

- A string is a sequence of ASCII characters obeying the length restrictions described in the previous section.
- A list is a sequence of one or more string or record values.
- A record is an unordered collection of field, value pairs (described by ⟨{**fieldvals**}⟩ in the grammar).

The evaluation rules for expression constructs are as follows:

- The simplest expression type corresponds to the empty list `[]`, which evaluates to itself.

  **Failure conditions:**

  – None

  **Security violations:**

  – None

- A string constant $s$ also evaluates to itself.

  **Failure conditions:**

  – None

  **Security violations:**

  – None

- A variable expression $x$ returns the current value of $x$ in the store.

  **Failure conditions:**

– $x$ does not exist in the store

**Security violations:**

– Current principal does not have `read` permissions on $x$.

- A field lookup expression $x.y$ returns the value stored in field $y$ of record $x$.

  **Failure conditions:**

  – $x$ is not a record.

  – $y$ is not a field of the record $x$.

  **Security violations:**

  – Current principal does not have `read` permissions on $x$.

- A record constructor $\{x_1 = \langle\textbf{value}\rangle, \ldots, x_n = \langle\textbf{value}\rangle\}$ evaluates to a record with fields $x_1, \ldots, x_n$ that are initialized to the evaluation of their corresponding $\langle\textbf{value}\rangle$. Field values may contain only strings when evaluated, not arrays or nested records.

  **Failure conditions:**

  – One or more $\langle\textbf{value}\rangle$ does not evaluate to a string.

  **Security violations:**

  – None

- An if-then-else expression `if` $\langle\textbf{bool\_expr}\rangle$ `then` $\langle\textbf{expr}\rangle$ `else` $\langle\textbf{expr}\rangle$ first evaluates $\langle\textbf{bool\_expr}\rangle$ according to the rules for Boolean expression evaluation given below. If the result is *true*, then the subexpression corresponding to the `then` branch is evaluated and returned. Otherwise, the subexpression corresponding to the `else` branch is evaluated and returned.

  **Failure conditions:**

  – If any of the subexpressions fails, then evaluation of this expression fails.

  **Security violations:**

  – If any of the subexpressions results in a security violation, then this expression does as well.

**Boolean expressions.** Boolean expressions evaluate to either *true* or *false*.

- The constants `true` and `false` evaluate to *true* and *false*, respectively.

  **Failure conditions:**

  – None

  **Security violations:**

  – None

- The expression `equal`$(\langle\textbf{value}\rangle, \langle\textbf{value}\rangle)$ evaluates to *true* if the arguments evaluate to the same value, and *false* otherwise.

  **Failure conditions:**

  – Evaluating either argument results in failure.

  **Security violations:**

  - Evaluating either argument results in a security violation.

- Similarly, the expression `notequal`(⟨**value**⟩, ⟨**value**⟩) evaluates to *false* if the arguments evaluate to the same value, and *true* otherwise.

  **Failure conditions:**
  - Evaluating either argument results in failure.

  **Security violations:**
  - Evaluating either argument results in a security violation.

### Security Policy

The server enforces a data access policy on which principals can `read`, `write`, and `append` to each global variable in the store. The data access policy is uniform across all variables and principals:

- The principal who initially creates a variable is given `read`, `write`, and `append` permissions.
- All other principals who are not `admin` are given `read` permissions. Note that this applies to principals who are created after the variable was first initialized, so that new users should be able to read variables already in the server's state.
- `admin` is given `read`, `write`, and `append` permissions on all variables.

This policy is fixed, and should not change throughout the execution of the server.

Additionally, the server enforces an administrative policy that dictates changes to the security state.

- Only `admin` can add new principals.
- Only `admin` and the principal $p$ can make changes to $p$'s password.
- Only `admin` can halt the server using `exit`.

### Input and output.

The server should read an input program until receiving three consecutive asterisks, as described in the grammar. All input after the terminating asterisks should be ignored. On receiving an incomplete program, the server can do any of the following:

- It can process the commands that it has received so far, and return {"`status`" : "`FAILED`"} or {"`status`" : "`DENIED`"} if any command results in a failure or security violation.
- It can hang while waiting for the terminating sequence of asterisks.
- It can timeout after 30 seconds of not receiving the terminating sequence of asterisks. In this case, it should output {"`status`" : "`TIMEOUT`"} and begin waiting for a new connection.

All output produced by the server should be in JSON format as follows:

- Each command's JSON output should be printed on a single line, ending with `\n`.
- Each JSON output should contain a `'status'` key containing the status code of the command (e.g., `CREATE_PRINCIPAL`, `SET`, etc.).
- The output of a `return` ⟨**expr**⟩ command contains `status RETURNING`, as well as an additional key `output` that contains the result of evaluating ⟨**expr**⟩. If ⟨**expr**⟩ is a string, then this value is a JSON string. If ⟨**expr**⟩ is a record, then this is a JSON record. If ⟨**expr**⟩ is a list, then this is a JSON array.

For more information on the JSON format, consult `http://json.org`. The code handout also contains facilities for producing JSON output in `Interp.ResponseToJSON`.

## Code Handout

Note that before attempting to compile the skeleton, you should ensure that the `yojson` package is installed and accessible by your Ocaml installation. You can check this by running the command `opam list`, and looking for the relevant package. If you do not have it installed, then run `opam install yojson`.

The `code` directory contains the following files:

```
code/
├── tests/
│   ├── basic_append
│   ├── basic_append.expected
│   └── ...
├── Makefile
├── aeson.ml
├── ast.ml
├── auth.ml
├── cmdline.ml
├── interp.ml
├── lexer.mll
├── parser.mly
├── rlist.ml
├── server.ml
└── state.ml
```

The `tests` directory contains, unsurprisingly, a number of initial test cases containing examples and their expected output. The remaining files contain the following:

`Makefile` Standard makefile configured to produce the necessary binaries (`interp`, `server`). You should not need to modify this unless you decide to add additional source files to your implementation.

`aeson.ml` Functions for formatting JSON output.

`ast.ml` Type definitions and helper funtions for working with abstract syntax trees of parsed programs.

`auth.ml` Currently empty. This is the recommended location for defining the types and functions needed to implement the server's security policy.

`cmdline.ml` Entry point for the debugging shell `interp`. Processes command line arguments as described in the previous section.

`interp.ml` Intended to contain the language interpreter. This is the core server functionality. At the moment, this provides basic types for response codes, JSON formatting facilities, and skeleton code for the main interpreter.

`lexer.mll` `ocamllex` lexer definition for the language grammar. You should not need to modify this.

`parser.mly` `ocamlyacc` parser definition for the language grammar. You should not need to modify this.

`rlist.ml` A type definition and basic `map`, `append`, and `filter` functions for a list that can be configured to store elements either right-to-left or left-to-right. This is used by `ast.ml`.

`server.ml` Entry point for the main server binary `server`. Establishes a listening TCP socket on the port specified via the command line, and passes incoming data to the interpreter.

`state.ml` Intended to house the types and helper functions needed to manage the server's state.

## 2 Develop a test suite (20 points)

For the second part of the assignment, you will develop a comprehensive test suite to demonstrate that your implementation matches the specification. In particular, you want to make sure that your server outputs JSON responses, returns with the appropriate values, enforces the security policy and input restrictions, as given in the specification.

To complete this part of the assignment, create a series of tests in the `tests` directory named `test`$-n$, where $n$ is an integer starting at 1. You should also provide a corresponding `test`$-n$`.expected` containing the expected correct output for each test case. Update the source for this document by uncommenting the section at the end, in the section "Test case documentation", and for each test explain:

- What aspect of the specification, or correctness in general, the test is meant to address.
- The way in which it accomplishes that goal.
- Whether the test uncovered a bug in your implementation at any point.

You are not required to provide a set number of test cases, but keep in mind that your implementation for Part 1 will be graded on correctness with respect to the specification. In the following, a "bug" is a deviation from some requirement in the spec. Your grade for this part will be factored in the following way:

- For each bug the graders find in your implementation that does *not* have a corresponding test case meant to address it, you will be docked two points in Part 1 and two points in Part 2.
- For each bug the graders find corresponding to a spec requirement for which you have one or more test cases, you will be docked one point in Part 1.
- *If you turn in fewer than ten test cases*, then for any bug the graders find in your implementation, you will be docked two points in Part 1 and three points in Part 2.

Note that you should not turn in test cases that are simple transformations of those that have already been provided (e.g., cases that test exactly the same set of features, or simply rename variables and constants).