Secure data server

The goal of this project is to implement a security-conscious data server.
(Think: Redis with security.)

Table of Contents

Overview

Clients can connect to the server via TCP and send a textual program, which is a list of commands whose grammar is given below. The server executes the program, sends textual output back to the client, and disconnects. Executing a program may cause data to be stored on the server, which can be accessed later by other programs. The server accepts one connection at a time (so programs never execute concurrently).

Let’s jump right in with an example of the kinds of programs your data server will execute:

```
as principal admin password "admin" do
   create principal alice "alices_password"
   set msg = "Hi Alice. Good luck in Build-it, Break-it, Fix-it!"
   set delegation msg admin read -> alice
   return "success"
```

Running this program successfully will result in the following output being sent back to the client:

```
{"status":"CREATE_PRINCIPAL"}
{"status":"SET"}
{"status":"SET_DELEGATION"}
{"status":"RETURNING","output":"success"}
```

There are a few concepts that this program captures:

Principals: Each program is run as a different user, referred to as a *principal*. Whichever principal runs the program determines what data the program can access. The program in this example is being run by a principal called `admin`, which is the superuser of the system; we will return to `admin`’s abilities later.

Creating principals: The first thing that `admin` does here is create a new principal named `alice` and sets `alice`’s password, using the `create principal`… command.

Creating data: Principals can create and manipulate data. With the `set msg`… command, `admin` creates some data—a string message for `alice`—and stores it as a variable named `msg`.

Delegating access to data: Principals can choose how to share data with others. `admin` shares with `alice` their ability to read the variable `msg` with the `set delegation`… command.

Returning: Programs can `return` computed values back to the client; in the example, the value of the `msg` (`"success"`) is returned.

Output per command, in JSON format: Each successfully executed command results in a *status code* being sent back to the client, expressed as the `status` field of a JSON record. Each command has a different status code. For example, for the `create principal alice "alices_password"` command in the example, the status code was CREATE_PRINCIPAL. For the **`return`** command, there is also an `output` field for a representation of the returned value.

After running the above program, user `alice` could view `admin`’s `msg` variable by running the following:

```
as principal alice password "alices_password" do
   return msg
```

Because `alice` has been delegated read access to `msg`, this program will successfully return the message created by `admin`. Any other user who tries to access that data would be denied access. The rest of the document explains the programming language in more detail.

Running and returning from the server

Here is how the server is executed:

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

This starts up a fresh server, listening for connections on TCP port *PORT* (a positive integer between 1024 and 65535). The administrator password is an optional argument (it is `"admin"` if the argument is missing). Argument formatting requirements, return codes, and other details are given in a later section.

There are four possible outcomes from running a program:

- The program succeeds. In this case, the client will receive outputs from running the program, and the server state is updated appropriately.
- The program fails. This happens when the program is malformed in some way. In this case, the *only* status code sent back to the client is `FAILED`. *Any effects from executing this program's commands so far are rolled back.*
- The program results in a security violation. This happens when the program attempts to do something it is not authorized to do. The *only* status code sent back to the client is `DENIED`. *Any effects from executing this program's commands so far are rolled back.*

We are not concerned about server-side faults. That is, we assume that the data stored at the server is only needed while the server is running. The server should only halt if it is specifically directed to exit by the administrator. The server implementor should avoid coding errors that lead to unexpected termination (e.g., crashes); we assume that the operating system, hardware, etc. are fault-free.

Command language

Overview

The first line of a valid program indicates a principal and her password. Each subsequent line contains a primitive command executed on the principal's behalf. The server outputs a status code to the client's connection for each primitive command executed (assuming the whole program completes successfully). A program concludes by either computing and returning some expression or instructing the server to exit. Primitive commands may define, add, or change entities stored at the server, and these entities are visible to subsequent, properly authorized programs.

Here is a slightly more complicated version of the first example program:

```
as principal admin password "admin" do
   create principal bob "B0BPWxxd"
   set x = "my string"
   set y = { f1 = x, f2 = "field2" }
   set delegation x admin read -> bob
   return y.f1
```

The first line indicates that this program is running on behalf of principal `admin`, whose password is `"admin"`.

Each subsequent line is a *primitive command*; each primitive command is executed in order with `admin`'s authority. This program:

- creates a principal bob (and sets bob's password);
- creates a new global variable x and initializes it to `"my string"`;
- creates another global variable y and initializes it to a record with two fields, where the first field, named `f1`, is initialized to x's value (`"my string"`), and the second field, named `f2`, is initialized to the string `"field2"`;
- specifies that bob may read x's contents (by delegating `admin`'s `read` authority on x to bob); and
- finally, returns the value of y's `f1` field.

The output of running this program is sent back to the client. This output is a sequence of *status codes* in JSON format, one per command:

```
{"status":"CREATE_PRINCIPAL"}
{"status":"SET"}
{"status":"SET"}
{"status":"SET_DELEGATION"}
{"status":"RETURNING","output":"my string"}
```

Notice that the RETURNING status code is coupled with an `output` field, which has a JSON representation of the returned value (all other status codes have no additional output).
The created principal (bob) and global variables (x and y) *persist* and so are available to subsequent programs run on the server, assuming those programs' running principal is authorized to access the variables. For example, suppose we were to then run the following program:

```
as principal bob password "B0BPWxxd" do
   return x
```

Because principal bob was granted access to read x, the client should get the following output:

```
{"status":"RETURNING","output":"my string"}
```

The following program would result in a security violation because bob does not have permission to *write* x, only *read* it:

```
as principal bob password "B0BPWxxd" do
   set z = "bob's string"
   set x = "another string"
   return x
```

The output of this program would be:

```
{"status":"DENIED"}
```

(We would have gotten the same output had bob tried to access variable y, since he was not delegated any access to it.)

What about variable z? Programs are *transactional*, which means that either the entire program succeeds or none of it succeeds. Thus, as a result of the security violation, the creation of variable z is *rolled back* so it is as if z was never created and thus subsequent programs will not see it.

Grammar

Below we give a context-free grammar for the command language in Backus-Naur form. This grammar represents the required features of the language. Later on, we discuss optional features, which you can implement for more points, as well as a detailed description of the required output format. The next section gives a description of each command's semantics (i.e., its meaning).

Any program that fails to parse (i.e., is not correct according to the grammar) results in failure. All programs consist of at most 1,000,000 ASCII (8-byte) characters (not a wide character set, like unicode); non-compliant programs result in failure.

In the grammar, elements in **`bold typewriter font`** (i.e., keywords and concrete punctuation) are *terminals*; elements surrounded by < > (like <cmd>) are *non-terminals*; elements in *italics* are *tokens* whose format is as follows:

- *s* indicates a *string constant* having no more than 65,535 characters surrounded by a pair of double quotes. Strings may contain alphanumeric characters, spaces (but no tabs or newlines), and punctuation—specifically commas, semi-colons, periods, question marks, exclamation marks, hyphens, and underscores. Strings match the regular expression *"[A-Za-z0-9_ ,;\.?!-]*"*
- *x*, *p*, *q*, *r*, *y* indicate an *identifier* having no more than 255 characters. Identifiers must be distinct from keywords (collected below), must start with an alphabetic character, and then may contain alphanumeric characters as well as underscore. Identifiers match regular expression *[A-Za-z][A-Za-z0-9_]**

- \n refers to the newline character (character code 10). (Note: \r is *not* supported.)

Here are the rules for the grammar:

```
<prog> ::= as principal p password s do \n <cmd>
<cmd> ::= exit \n | return <expr> \n | <prim_cmd> \n <cmd>
<expr> ::= <value> | [] | { <fieldvals> }
<fieldvals> ::= x = <value> | x = <value> , <fieldvals>
<value> ::= x | x . y | s
<prim_cmd> ::=
            create principal p s
          | change password p s
          | set x = <expr>
          | append to x with <expr>
        | local x = <expr>
          | foreach y in x replacewith <expr>
          | set delegation x q <right> -> p
          | delete delegation x q <right> -> p
          | default delegator = p
<right> ::= read | write | append | delegate
```

Whitespace

Space between terminals, non-terminals, and tokens in the rules above corresponds to whitespace in the parsed program. *Whitespace may include spaces (character code 32) but not tabs or newlines*. For example, here is a legally reformatted version of our example program:

```
as         principal admin password "admin" do
create principal bob "B0BPWxxd"
        set x = "my string"
       set y ={f1=x,f2="field2"}
   set     delegation x admin read-> bob
   return y . f1
```

Notice that there are spaces between words, and at the start and end of lines, and that space need not be around punctuation (such as equals or dot). On the other hand, each command must be on a single line; it is not acceptable to break commands across lines.

Comments

Programs support line-ending comments. In particular, any line in the input program may end with **//** followed by text up until the end of the line. It is also permitted for a comment to be on a line by itself, as long the line begins with **//** *without any preceding whitespace*. Comments must conform to the Ruby regular expression *[\/][\/][A-Za-z0-9_ ,;\.?-!]*$*

Reserved keywords

The following are *keywords* that cannot be used for program-defined variables, record fields, or principals: **`all`**, **`append`**, **`as`**, **`change`**, **`create`**, **`default`**, **`delegate`**, **`delegation`**, **`delegator`**, **`delete`**, **`do`**, **`exit`**, **`foreach`**, **`in`**, **`local`**, **`password`**, **`principal`**, **`read`**, **`replacewith`**, **`return`**, **`set`**, **`to`**, **`write`**. Note that `admin` and `anyone` are not keywords, they are predefined principals. Keywords for optional features (whether or not you implement those features) should also be excluded; these include **`split`**, **`concat`**, **`tolower`**, **`notequal`**, **`equal`**, **`filtereach`**, **`with`**, **`let`**.

Command Language Detailed Description

In what follows, we describe the semantics of various program constructs. In some cases we point out that the current principal must have a certain permission on a variable `x` or else there is a security violation; we discuss how to compute permissions in a later section. We note that in general, if there is a failure and a security violation in the same expression or command, the security violation takes precedence. For example, for the expression *x***.***y* where *x* is a string (when it should be a record) that the current principal is not permitted to read, the program status code will be `DENIED` rather than `FAILED`.

Programs (<prog>)

In words, a program <prog> begins with the line **`as principal`** *p* **`password`** *s* **do** and is followed by <cmd> on the next line. A <cmd> itself may consist of multiple <prim_cmd>s separated by newlines, finally concluding with either **`exit`** or **`return`** <expr>.

The *security state* of the system consists of a set of *delegation assertions* made by programs that have called the **`set delegation`** command. This state is used to determine whether a particular principal has one or more of four possible permissions for a given variable *x*; these permissions are **`read`**, **`write`**, **`append`**, or **`delegate`** (in the grammar above, they referred to as <right>). Such permissions are required for various commands, as described below; how the security state is used to determine permissions is discussed later in this document.

A program **`as principal`** *p* **`password`** *s* **`do`** `\n` <cmd>

- Fails if principal *p* does not exist.
- Security violation if the password *s* is not *p*'s password.
- Otherwise, the server terminates the connection after running <cmd> under the authority of principal *p*.

Commands (<cmd>)

A <cmd> is zero or more primitive commands (described below), each ending with a newline, concluding with either **`exit`** or **`return`** <expr>.

If the command is **`exit`**, then it outputs the status code is `EXITING`, terminates the client connection, and halts with return code 0 (and thus does not accept any more connections). This command is only allowed if the current principal is `admin`; otherwise it is a security violation.

If the command is **`return`** <expr> then it executes the expression and outputs status code `RETURNING` and the JSON representation of the result for the key `"output"`; the output format is given at the end of this document.

Expressions and variables (<expr>, <value>, and <fieldvals>)

This programming language has <value>s that can be variables, records, or strings:

- *x*
    - Returns the current value of variable *x*.
    - Fails if *x* does not exist
    - Security violation if the current principal does not have **`read`** permission on *x*.
- *x* . *y*
    - If *x* is a record with field *y*, returns the value stored in that field.
    - Fails if *x* is not a record or does not have a *y* field.
    - Security violation if the current principal does not have **`read`** permission on *x*.
- *s*
    - This is a string constant, which evaluates to itself

Expressions can be <value>s, as well as two types of containers:

- **`[]`**
    - this is an empty *list*, and evaluates to itself.
- **{** *x1* **=** <value>**,** *x2* **=** <value>**,** ... **,** *xn* **=** <value> **}**
    - this is a *record*, which comprises *field labels* (*x1*, *x2*, ..., *xn*) which are here initialized to their corresponding <value>s. This also evaluates to itself.

Primitive commands (<prim_cmd>)

Other than **`return`** and **`exit`**, a <cmd> is an ordered list of *primitive commands* separated by newlines; we detail each primitive command below. Note that commands may include expressions; these are executed as discussed above. If an expression fails or issues a security violation, then the command that invokes it does.

`create principal` *p s*
Creates a principal *p* having password *s*.

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. There is also a preconfigured principal anyone whose initial password is unspecified, and which has no inherent authority. (See also the description of **`default delegator`**, below, for more about this command, and see the permissions discussion for more on how principal anyone is used.)

Failure conditions:

- Fails if *p* already exists as a principal.
- Security violation if the current principal is not `admin`.

Successful status code: `CREATE_PRINCIPAL`

`change password` *`p s`*

Changes the principal *p*'s password to *s*.

Failure conditions:

- Security violation if the current principal is neither `admin` nor *p* itself.

Successful status code: `CHANGE_PASSWORD`

`set` *`x`* **`=`** <expr>

Sets *x*'s value to the result of evaluating <expr>, where *x* is a global variable. If *x* does not exist this command creates it. If *x* is created by this command, and the current principal is not `admin`, then the current principal is delegated **`read`**, **`write`**, **`append`**, and **`delegate`** rights from the `admin` on *x* (equivalent to executing **`set delegation`** *`x`* `admin` **`read ->`** *p* and **`set delegation`** *`x`* `admin` **`write ->`** *p*, etc. where *p* is the current principal).

Setting a variable results in a "deep copy." For example, consider the following program:

```
set x = "hello"
set y = x
set x = "there"
```

At this point, y is still `"hello"` even though we have re-set x.

Failure conditions:

- Security violation if the current principal does not have **`write`** permission on *x*.

Successful status code: `SET`

`append to` *`x`* **`with`** <expr>

Adds the <expr>'s result to the end of *x*. If <expr> evaluates to a record or a string, it is added to the end of x; if <expr> evaluates to a list, then it is concatenated to (the end of) *x*.

Failure conditions:

- Fails if *x* is not defined or is not a list.
- Security violation if the current principal does not have either **`write`** or **`append`** permission on *x*.

Successful status code: `APPEND`

`local` *`x`* **`=`** <expr>

Creates a local variable *x* and initializes it to the value of executing <expr>. Subsequent updates to *x* can be made as you would to a global variable, e.g., using **`set`** *`x`*, **`append`**...**`to`** *x*, **`foreach`**, etc. as described elsewhere in this section. Different from a global variable, *local variables are destroyed when the program ends—they do not persist across connections.*

Failure conditions:

- Fails if *x* is already defined as a local or global variable.

Successful status code: LOCAL

foreach *y* **in** *x* **replacewith** <expr>

For each element *y* in list *x*, replace the contents of *y* with the result of executing <expr>. This expression is called for each element in *x*, in order, and *y* is bound to the current element.

As an example, consider the following program:

```
as principal admin password "admin" do
   set records = []
   append to records with { name = "mike", date = "1-1-90" }
   append to records with { name = "dave", date = "1-1-85" }
   local names = records
   foreach rec in names replacewith rec.name
   return names
```

This program creates a list of two records, each with fields `name` and `date`. Then it makes a copy of this list in `names`, and updates the contents of `names` using **foreach**. This **foreach** iterates over the list and replaces each record with the first element of that record. So the final, returned variable `names` is `["mike","dave"]`. (The original list `records` is not changed by the **foreach** here.)

Failure conditions:

- Fails if *x* is not a list or if *y* is already defined as a local or global variable.
- Security violation if the current principal does not have **read** and **write** permission on *x*.
- If any execution of <expr> fails or has a security violation, then entire **foreach** does.

Successful status code: FOREACH

set delegation *x q* <right> **->** *p*

Indicates that *q* delegates <right> to *p* on variable *x*, so that *p* is given <right> whenever *q* is. If *p* is `anyone`, then effectively all principals are given <right> on *x* (for more detail, see here). If *x* is the keyword **all** then *q* delegates <right> to *p* for *all* variables on which *q* (currently) has **delegate** permission.

Failure conditions:

- Fails if either *p* or *q* does not exist.
- Security violation if the running principal is not `admin` or *q* or if *q* does not have **delegate** permission on *x*.

Successful status code: SET_DELEGATION

delete delegation *x q* <right> **->** *p*

Indicates that *q* revokes a delegation assertion of <right> to *p* on variable *x*. In effect, this command revokes a previous command **set delegation** *x q* <right> -> *p*; see below for the precise semantics of what this means. If *x* is the keyword **all** then *q* revokes delegation of <right> to *p* for *all* variables on which *q* has **delegate** permission.

Failure conditions:

- Fails if either *p* or *q* does not exist.
- Security violation unless the current principal is `admin`, *p*, or *q*; if the principal is *q*, then it must have **`delegate`** permission on *x* (no special permission is needed if the current principal is *p*: any non-`admin` principal can always deny himself rights).

Successful status code: `DELETE_DELEGATION`

`default delegator` `=` *p*

Sets the "default delegator" to *p*. This means that when a principal *q* is created, the system automatically delegates **`all`** from *p* to *q*. Changing the default delegator does not affect the permissions of existing principals. The initial default delegator is anyone.

Failure conditions:

- Fails if *p* does not exist
- Security violation if the current principal is not `admin`.

Successful status code: `DEFAULT_DELEGATOR`

Enforcing Command Permissions

In the command descriptions, we indicate that security violations can occur if a particular principal does not have a particular permission on a variable *x*. Whether or not a principal *p* has a permission <right> (**`read`**, **`write`**, **`append`**, or **`delegate`**) on variable *x* is determined by the current *security state* (call it ***S***), which is the set of active *delegation assertions* made by **`set delegation`** *q x* <right> **`->`** *p* statements, and the following three rules:

1. `admin` has <right> on *x* (for all rights <right> and variables *x*)
2. A principal *p* has <right> on *x* if principal anyone has <right> on *x*.
3. A principal *p* has <right> on *x* if there exists some *q* that has <right> on *x and* ***S*** includes a delegation assertion *q x* <right> **`->`** *p*

The third rule essentially combines uses of all three rules to establish rights transitively.

Example. Consider our very our first example, which included the statement

```
    set delegation x admin read -> bob
```

Prior to running this statement, ***S*** is empty. This statement adds delegation assertion `x admin read -> bob` to ***S***. We can establish that principal bob has **`read`** permission on variable x by applying rules 3 and 1 together. From rule 3, we establish that bob (playing the role of principal *p*) has **`read`** permission (a <right>) on x as long as there exists some *q* such that *q* has **`read`** permission on x and ***S*** contains delegation assertion *q* `x read -> bob`. In this case, that *q* is `admin`: by rule 1, `admin` has right **`read`** on x, and by the **`set delegation`** statement above, `admin x read -> bob` is in ***S***.

Continuing the example, suppose we subsequently executed the following statements successfully:

```
        set delegation x bob read -> alice
        set delegation x admin append -> anyone
```

Doing so adds two more delegation assertions to ***S***, which then consists of three assertions: `admin x read -> bob`, `x bob read -> alice`, and `x admin append -> anyone`. With these, we could now establish:

- `alice` has **`read`** permission on `x`. This is because `admin` has **`read`** permission on `x` (rule 1), `admin` delegates that permission to bob (rule 3, and first assertion in ***S***), and bob delegates that permission to `alice` (rule 3, and second assertion in ***S***).
- `alice` has **`append`** permission on `x`. This is because `admin` has **`append`** permission on `x` (rule 1), `admin` delegates this permission to `anyone` (rule 3, and third assertion in ***S***), and since `anyone` has this permission then `alice` does (rule 2). (bob has **`append`** permission by the same reasoning.)

Maintaining the security state. As mentioned in the description of the commands, above, executing **`set delegation`** and **`delete delegation`** requires certain permissions. To recap:

- **`set delegation`** *x p* <right> **`->`** *q* requires that the current principal be either `admin` or *p*, and that *p* has **`delegate`** permission on *x*, if *x* is a normal variable. (If *x* is the keyword **`all`**, *p* needs no special permissions.)
- **`delete delegation`** *x p* <right> **`->`** *q* requires that the current principal be either `admin` or *p*, and that *p* has **`delegate`** permission on *x,* if *x* is a normal variable. (If *x* is the keyword **`all`**, *p* needs no special permissions.) Alternatively, as long as x is not `all`, the current principal may be *q*, in which case no special permission is required.

For normal variables *x*, successfully executing **`set delegation`** *x p* <right> **`->`** *q* adds the delegation assertion *x p* <right> **`->`** *q* to ***S***, while executing **`delete delegation`** *x p* <right> **`->`** *q* revokes assertion *x p* <right> **`->`** *q if it is explicitly present in **S***; otherwise the command does nothing. Consider our second example above, after which ***S*** has three assertions. If we were to execute

```
        delete delegation x admin read -> alice
```

Then this statement has no effect: Because this assertion is not directly present in ***S***, nothing is removed. On the other hand, executing

```
        delete delegation x bob read -> alice
```

serves to remove the assertion from ***S*** that `x bob read -> alice`.

Executing **`set delegation all`** *p* <right> **`->`** *q* adds (zero or more) assertions of the form *x p* <right> **`->`** *q* for all variables *x* on which *p* has **delegate** permission. Conversely, **`delete delegation all`** *p* <right> **`->`** *q revokes* (zero or more) assertions of the form *x p* <right> **`->`** *q* for those variables *x* on which *p* has **delegate** permission.

The security state, consisting of the set of active delegations due to **`set delegation`** and **`delete delegation`** commands, is also *transactional*—any changes to the security state made by a program are rolled back if that program fails or issues a security violation

Details on Formats and Error Conditions

Command line arguments

Any command-line input that is not valid according to the rules below should cause the program to exit with a return code of 255. When the server cleanly terminates, it should exit with return code 0.

- Command line arguments cannot exceed 4096 characters each
- The port argument must be a number between 1,024 and 65,535 (inclusive). It should be provided in decimal without any leading 0's. Thus 1042 is a valid input number but the octal 052 or hexadecimal 0x2a are not.
- The password argument, if present, must be a legal string ***s***, per the rules for strings given above, but without the surrounding quotation marks.

Server startup and shutdown

If when starting up the server the port specified by the first argument is taken, the server should exit with code 63. The server should exit with return code 0 when it receives the SIGTERM signal.

Output

All output from the server to the client will be in JSON format, defined as follows.

- Each command's JSON output is printed on a single line and is followed by a newline ('\n', the ASCII character with code decimal 10).
- Each command's JSON output includes the key `"status"` which indicates the status code of each <prim_cmd>, **`return`**, or **`exit`**. The value of `"status"` is a string constant dependent on the result of executing the command. For example, if there is a security violation, the `"status"` is `"DENIED"`; if there is a failure, the status is `"FAILED"`**.** The legal `"status"` codes are given with the description of each command.
- The output of a return <expr> command results in a `"status"` value `"RETURNING"`. The output JSON record also contains an `"status"` field, whose

value is a JSON representation of the <expr>. In particular,

  - If the <expr> is a string, then the value is a JSON string
  - If the <expr> is a record, the the value is a JSON record
  - If the <expr> is a list, the the value is a JSON array (whose contents are formatted as JSON strings or records, as needed)

Here is an example output where an expression is returning the string `"example"`:

```
{"status":"RETURNING","output":"example"}
```

Here is an example output where an expression is returning a list of strings and records:

```
{"status":"RETURNING","output":["example","test",
{"f":"val","g":"val2"}]}
```

Here is an example output where an expression is returning a record:

```
{"status":"RETURNING","output":{"key1":"elem1","key2":"string"}}
```

Optional features

Here we detail three features you may optionally implement for extra points: string functions, filtering lists, and recursive expressions.

To get full points for an optional feature you must implement all the features that precede it. In particular, to get points for filtering lists, you must also implement string functions; to get points for recursive expressions, you must implement string functions and filtering lists, too.

Note that implementing an optional feature exposes you to more possible attacks. If you don't implement string functions, break-it teams cannot find bugs in string functions.

String functions

We extend expressions to include calls to functions on strings.

```
<expr> ::=  … | <func> ( <args> )
<func> ::= split | concat | tolower
<args> ::= <value> | <value> , <args>
```

In short, we extend allowable expressions to include calls to functions `split`, `concat`, and `tolower`, which take <value> arguments expected to evaluate to strings ***s***. Here's what they do:

- `split(`*s1*`,`*s2*`)`
  - returns a record `{ fst = `*s11*`, snd = `*s12*` }` where *s11* and *s12* are the result of splitting string *s1*. String *s11* is the first *N* characters of *s1* where *N* is the length of *s2*, and string *s12* is the remainder of *s1*. If *N* is greater than the length of *s1* then `fst = `*s1* and `snd = ""`.
  - Fails if *s1* and/or *s2* are not strings.
- `concat(`*s1*`,`*s2*`)`
  - returns a new string that is the concatenation of *s1* and *s2*. The concatenated string is truncated to 65535 characters (if it would exceed that length).
  - Fails if *s1* or *s2* is not a string.
- `tolower(`*s*`)`
  - returns a new string that converts all uppercase characters in *s* to lowercase.
  - Fails if *s* is not a string.

Example

```
as principal admin password "admin" do
   set x = "hello"
   set y = split(x,"--")
   set z = concat(x,y.fst)
   return { x=x, y=y.snd, z=z }
```

Produces output

```
{"status":"SET"}
{"status":"SET"}
{"status":"SET"}
{"status":"RETURNING","output":
{"z":"hellohe","x":"hello","y":"llo"}}
```

Filtering lists

We add a new command and two new functions in support of it.

<prim_cmd> ::= … | **`filtereach`** *y* **`in`** *x* **`with`** <expr>
<fun> ::= … | `equal` | `notequal`

In short, we extend commands with one additional command, and functions with two new functions:

- **`filtereach`** *y* **`in`** *x* **`with`** <expr>
  - for every element in *x*, execute <expr> where occurrences of *y* are bound to the record's value. If the <expr> evaluates to the empty string `""`, then that element is retained; otherwise it is removed.
  - Fails if *x* is not a list or *y* is already defined.

- Security violation if the current principal does not have **read** and **write** access on *x*.

- `equal(<value>,<value>)`
    - takes two arguments and returns `""` if they are equal, and `"0"` if they are not.
    - Arguments are permitted to be strings or records; fails otherwise.
- `notequal(<value>,<value>)`
    - takes two arguments and returns `""` if they are equal, and `"0"` if they are not.
    - Arguments are permitted to be strings or records; fails otherwise.

Example

```
as principal admin password "admin" do
   set records = []
   append to records with { name = "mike", date = "1-1-90" }
   append to records with { name = "sandy", date = "1-1-90" }
   append to records with { name = "dave", date = "1-1-85" }
   filtereach rec in records with equal(rec.date,"1-1-90")
   return records
```

Produces output

```
{"status":"SET"}
{"status":"APPEND"}
{"status":"APPEND"}
{"status":"APPEND"}
{"status":"FILTEREACH"}
{"status":"RETURNING","output":[{"date":"1-1-90","name":"mike"},
{"date":"1-1-90","name":"sandy"}]}
```

Recursive expressions

We extend the grammar for expressions to allow locally-defined variables within an expression:

<expr> ::= … | **let** *x* **=** <expr> **in** <expr>

When this expression form is executed we create a local variable *x* and assign the result of execution the first <expr> part to it and then execute the <expr> part. When <expr> completes, we delete the local variable *x*. This expression fails if *x* is already defined (as a local or global variable).

Example

```
as principal admin password "admin" do
   set x = { f1 = "hello", f2 = "there" }
   set y = let z = concat(x.f1, " ") in concat(z, x.f2)
   return y
```

Produces output

```
{"status":"SET"}
{"status":"SET"}
{"status":"RETURNING","output":"hello there"}
```

Oracle

We have written an *oracle*, or reference implementation, to adjudicate tests submitted during the Break-it round. The behavior of a targeted build-it submission on a test will be compared against the behavior of the oracle, where the oracle is configured to implement the same set of features the target does (in the case the target does not implement all optional features). Differences in behavior indicate a bug. When a test constitutes evidence of a *correctness* or *security* bug is discussed below.

Contestants may query the oracle during the build-it round to clarify the expected behavior of the server program. Queries are specified as command line arguments and a sequence of input programs. They may be submitted via the "Oracle submissions" link on the team participation page of the contest site. Here is an example query:

```
{
  "arguments":["6300","password"],
  "programs":[
    "as principal admin password \"password\" do\nset x =
\"x\"\nreturn x\n",
    "as principal admin password \"password\" do\nreturn x\n"
  ]
}
```

The oracle will provide outputs from the server and the return code. Here is output for the above query:

```
{
  "return_code":0,
  "output":[
    [{"status":"SET"},{"status":"RETURNING","output":"x"}],
    [{"status":"RETURNING","output":"x"}]
  ]
}
```

Reporting bugs in the oracle

While we have striven to make the oracle and this specification consistent, we may have made some mistakes. Breaker teams will receive 25 points for identifying bugs in the oracle implementation (or problems with the specification, if the oracle is right but the spec is wrong). Points will only be awarded to the first breaker team that reports a particular bug. To report a bug in the oracle, email info@builditbreakit.org with a description of the bug, links to relevant oracle submissions on the contest website, your team name, and team ID.

Implementation Goals

Performance goal

The main performance goal of the server is minimizing **elapsed time**. That is, we want the server to execute each program it receives as quickly as possible. We will measure performance for

- single, short programs
- single, large programs (that involve many operations in the same program); and
- sequences of programs (large and short),

We are not concerned about the memory the program uses (but keep in mind that the reference platform may not be able to execute a program that hogs space too much).

Security goals and threat model

Our security goals are **privacy, integrity**, and **availability**. We say that privacy is violated if an attacker is able to read stored data for which it is not granted access. We say that integrity is violated if an attacker is able to modify stored data for which it should not have access. We say that availability is violated if a principal is denied the ability to access data despite it having the authority to do so. We are interested in the security of normal data (lists, strings, and records) and the security state—e.g., it is a security violation if *p* is not permitted to delegate access to data when the rules say that it should.

While attempting to violate the above security properties, the attacker is assumed to be able to communicate directly with the server, but cannot view (or corrupt) others' communications with the server. As such, as evidence of a defect, break-it teams will provide a sequence of programs—a test—that when executed on the target server will behave incorrectly or insecurely. In particular:

We consider it a **security bug** in a targeted submission when running some program
- the oracle says `DENIED` but the target doesn't (integrity or privacy)

- the oracle returns correctly (`EXITING` or `RETURNING` status), but the target says DENIED (availability)
- the oracle returns any status within 30 seconds but the target fails to return within a much longer window (availability). If the oracle takes longer than 30 seconds to run, it will issue status `TIMEOUT`.

On the other hand, we consider it a **correctness bug** in a targeted submission when
- the oracle returns `FAILED`, but the target returns some other status
- both the oracle and target return correctly, but the output differs (different status, returned value, etc.)

Build-it Round Submission

Each build-it team should initialize a git repository on either github or bitbucket or gitlab and share it
with the `bibifi` user on those services. Create a directory named `build` in the top-level directory of this repository and commit your code into that folder. Your submission will be scored after every push to the repository. (Beware making your repository public, or other contestants might be able to see it!)

To score a submission, an automated system will first invoke `make` in the `build` directory of your submission. The only requirement on `make` is that it must function without internet connectivity, and that it must return within ten minutes. Moreover, it must be the case that your software is actually built, through initiation of `make`, from source (not including libraries you might use). Submitting binaries (only) is not acceptable.

Once make finishes, `server` should be an executable file within the `build` directory. An automated system will invoke them with a variety of options and measure their responses. The executables must be able to be run from any working directory. If your executables are bash scripts, you may find the following resource helpful.