Secure IoT Controller

Overview

The IoT Controller manages a smart home. It receives periodic updates from a set of sensors (e.g., current temperature, power consumption, etc.) and is able to control a set of output devices (e.g., turning the lights on or off).

User Programs

Users of the smart home are able to write programs in a domain-specific language and send them to the IoT Controller. It will execute these programs one at a time (not concurrently), and in the process will update its output devices, or store information at the controller for future use, and return information about the current state of the smart home. In this language, devices and sensors are represented as variables. To update the state of an output device, the program sets the corresponding value. To see the current value of a sensor, the program reads that value. Which sensor/device values are accessible to which users (aka *principals*) is controlled by an access control policy which permits users to control which other users can read, write, or otherwise access sensors or system state.

Here is an example program:

```
as principal admin password "admin" do
  create principal alice "alices_password"
  set door = 0
  set delegation door admin read -> alice
  set delegation door admin write -> alice
  return 1
***
```

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":"SET_DELEGATION"}
{"status":"RETURNING","output":"1"}
```

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 sensors and devices 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.

Manipulating output device state: Principals can access and modify the state of output devices. With the **set door**... command, admin updates the state of the door device. This device controls whether the door is open (1) or closed (0). By setting door to 0, admin ensures the door is closed.

Delegating access to data: Principals can choose how to share data with others. admin shares with alice their ability to read and write the state of the door device with the **set delegation**... command.

Returning: Programs can **return** computed values back to the client; in the example, the value of the integer "1" is returned to indicate the program executed correctly.

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 the door device's state by running the following:

```
as principal alice password "alices_password" do
  return door
***
```

Because alice has been delegated read access to door, this program will successfully return the value 0 (indicating the door is closed). Any other user who tries to access that device would be denied access.

In general, there are three possible outcomes from running a program:

- **The program succeeds**. In this case, the client will receive outputs from running the program, and the controller's 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 either DENIED_WRITE or DENIED_READ depending on the action prevented due to insufficient rights. *Any effects from executing this program's commands so far are rolled back.*

Sensor and Device Configuration

When the controller starts up it learns about the sensors and devices in the smart home from a configuration file. This file is formatted as JSON and distinguishes sensors from output devices, and indicates their initial values. These values are always integers, but are stored as strings in the file.

```
{
  "sensors": {
        “temperature”:”80”,
        “home_energy”: “50”,
        “smoke”: “0”,
        “owner_location”: “0”
  },
  "output_devices": {
        “lights”:”0”,
        “alarm_status”: “1”,
        “air_conditioning”: “0”
  }
}
```

This indicates that there is a sensor for the current temperature, which currently is 80 degrees (*temperature*), the house is currently using 50 kW (*home_energy*), no smoke is currently detected (*smoke*), and the owner (or at least their smartphone, which is being tracked) has left the house (*owner_location*). For the output devices, the lights are off (*lights*), the alarm is activated (*alarm_status*), and the air conditioning system is off (*air_conditioning*). Your controller should work for varying configurations (i.e., arbitrary sets of sensors/devices and sensors/device values).

Sensor Updates

Periodically, the Controller needs to be notified that sensors have taken on new values. To keep things simple, these sensor updates are communicated as programs written in the same domain-specific language as users’ programs. These programs are executed under the authority of a special principal, hub. The idea is that a separate hub device polls the sensors and periodically informs the controller of changes to their readings. (The hub can also be thought of as snooping the results of user program executions in order to see when output devices have been set, so directives can be relayed to them.) The hub principal has the authority to update all sensors’ values defined in the configuration, but cannot read them or any other system variables.

Here is an example of a sensor update being sent by the hub:

```
as principal hub password "hub_password" do
  set temperature = 180
  set smoke = 1
```

```
  return 0
***
```

This message indicates that the current reading of the temperature sensor is 180 degrees, and that smoke has been detected by the smoke alarm. (Oh no!)

IFTTT Rules for Asynchronous Updates

In order to immediately respond to changes in the smart home status, users are able to install *If This Then That (IFTTT) rules* on the controller. IFTTT rules define a triggering condition (e.g., temperature greater than 80) and a set of commands to execute when the condition is met (e.g., turn on cooling). Consider the following example (call it program P), which generalizes the first program we saw:

```
as principal admin password "admin" do
  create principal bob "B0BPWxxd"
  set rule too_hot if temperature >= 80 then set air_conditioning = 2
  activate rule too_hot
  set delegation air_conditioning admin read -> bob
  return temperature.0
***
```

Per the first line, this program executes with admin's authority. Each line is a *primitive command*; this program does the following:

- creates a principal bob (and sets bob's password);
- creates a new IFTTT rule named *too_hot* that states that if the temperature sensor's input exceeds 80 degrees, then the controller should set the air conditioning system to cool the room (option 2).
- activates the rule *too_hot* so that the controller will begin monitoring the temperature for a triggering state.
- specifies that bob may read the air_conditioning device's status (by delegating admin's read authority on air_conditioning to bob); and
- returns the value of temperature at time 0.

IFTTT rules are considered each time a program completes successfully; that program could be a user program or a sensor update program provided by the hub. Rules whose conditions are met should be executed under the authority of the principal who set the rule. For example, if we ran program P and then ran the above sensor update program (under the authority of principal hub) then on conclusion of the update program's execution, program P's too_hot rule would be triggered, and run with admin's authority (since that was the principal that created the rule, when running P). The status codes returned from executing the sensor update program would be as follows:

```
{"status":"SET"}
{"status":"SET"}
{"status":"RETURNING","output":"0"}
```

```
{"rule":"too_hot","status":"SET"}
```

The first three lines are the status codes from the sensor update itself. The last line shows that the too_hot rule has been triggered, with the result that air_conditioning has been set to 2. Rules that are not triggered are not listed in the status. If a rule performs an operation for which it is not authorized (e.g., reading or writing a variable), or otherwise fails (e.g., by trying to read a non-existent variable) then the rule fails, any of the rule's effects are rolled back, and the rule is automatically deactivated. Rule execution is presented in detail in a later section.

Running and returning from the controller

Here is how the controller (in an executable named *server*) is executed:

```
./server PORT CONFIG_FILE [ADMIN_PASSWORD [HUB_PASSWORD]]
```

This starts up a fresh controller, listening for connections on TCP port *PORT* (a positive integer between 1024 and 65535). The controller reads the sensor/output device configuration from the given *CONFIG_FILE*, formatted in JSON. Finally, both the admin password and hub password are optional arguments (the former is "admin" if the argument is missing, and "hub" for the latter). Argument formatting requirements, return codes, and other details are given in a later section.

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

Controller Command Language

This section describes the domain specific language used by users to write programs run at the controller. The language is also used to communicate sensor updates to the controller (as provided by the principal hub).

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 controller 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 controller to exit. Primitive commands may access sensor data and change the state of output devices (e.g., turn lights on or off) the requesting principal is authorized to access. They can also set IFTTT rules to be executed during device operation.

Recall the program we discussed earlier:

```
as principal admin password "admin" do
  create principal bob "B0BPWxxd"
  set rule too_hot if temperature >= 80 then set air_conditioning = 2
  activate rule too_hot
  set delegation air_conditioning admin read -> bob
  return temperature.0
***
```

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_RULE"}
{"status":"ACTIVATE_RULE"}
{"status":"SET_DELEGATION"}
{"status":"RETURNING","output":75}
```

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) *persists* and so subsequent programs are able to run as the new principal. For example, suppose we were to then run the following program:

```
as principal bob password "B0BPWxxd" do
  return air_conditioning
***
```

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

```
{"status":"RETURNING","output":”0”}
```

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

```
as principal bob password "B0BPWxxd" do
  set air_conditioning = 1
  return air_conditioning
***
```

The output of this program would be:

```
{"status":"DENIED_WRITE"}
```

What about reading air_conditioning? 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 controller will not return the status of cooling, even though bob does have access to 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 a detailed description of the required output format. The next section gives a description of each command's semantics (i.e., its meaning).

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. Any program that fails to parse (i.e., is not correct according to the grammar) results in failure. *Parse errors take precedence over security violations in the program itself*; see the next section for more detail.
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_]**
- *i* indicates a 32-bit integer constant. Integers match the regular expression *-?[0-9]+.* The largest expressible integer is 2147483647; the smallest is -2147483648.

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>
<cond> ::= <value> <cond_op> <value>
<func> ::= **mean** | **min** | **max** | **count**
<expr> ::= <value> | <func> *x* | <func> *x i i* | <value> <math_op> <value>
<cond_op> ::= **=** | **>** | **<** | **>=** | **<=**
<math_op> ::= **+** | **-** | **/** | *****
<value> ::= *x* | *i* | *x.i*

```
<prim_cmd> ::=
        create principal p s
      | change password p s
      | set x = <expr>
      | local x = <expr>
      | if <cond> then <prim_cmd>
      | set delegation <tgt> q <right> -> p
      | delete delegation <tgt> q <right> -> p
      | default delegator p
      | print <expr>
      | set rule x =  if <cond> then <prim_cmd>
      | activate rule x
      | deactivate rule x
<tgt> ::= all | x
<right> ::= read | write | delegate | toggle
```

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 rule too_hot if temperature >= 80 then set air_conditioning = 1
  activate rule too_hot
  set   delegation      air_conditioning admin read -> bob
  return temperature.0
***
```

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 principals or variables: **activate**, **all**, **as**, **begin**, **change**, **count**, **create**, **deactivate**, **default**, **delegation**, **delegator**, **do**, **end**, **exit**, **for**, **if**, **max, mean, min, password**, **principal**, **print**, **read**, **reset**, **return**, **rule**, **set**, **then**, **to**, **toggle**, **write**, *******. Note that admin, hub, and anyone are not keywords, they are predefined principals. (Some of these keywords are due to optional features; even if you don't implement those features, you should not allow the keywords to be used as variables.)

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 during evaluation (according to the failure conditions given below) and a security violation in the same expression or command, the security violation takes precedence**. For example, for the expression *x.i* where *x* is a string (when it should be a sensor, output device, or user defined variable) that the current principal is not permitted to read, the program status code will be DENIED_*X* (READ or WRITE) rather than FAILED. Conversely, a parse error (i.e., a failure according to the grammar above and prior to evaluation) *anywhere in the program* takes precedence over any security violation. In particular, if you implement the controller by reading and processing one command at a time, then if you hit a security violation but a subsequent command has a parse error, you must issue FAILED rather than DENIED_*X*. Additionally, in this case, all variable state changes directed by the program should not occur.

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>. A program completes with sequence *******.

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 the possible permissions for a given variable *x*; these permissions are **read**, **write**, **delegate**, and **toggle** (in the grammar above, they are 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.

Executing a program begins by confirming that *s* is *p*'s password. Then the server runs <cmd> under the authority of principal *p*. The semantics for executing a <cmd> is given next. If <cmd> executes successfully, then the server considers all active rules. It executes the commands of the rules whose conditions are met under the authority of the principal *q* that created the rule. We discuss rule

execution later in the document. Once all rules have been considered and executed the server terminates the connection.

Commands (<cmd>)

A <cmd> is zero or more primitive commands (described below), each ending with a newline, concluding with either **exit** or **return** <expr>. Each primitive command's status code is returned as it executes, assuming the whole program completes successfully.

If the command is **exit**, then it outputs the status code is EXITING, terminates the client connection, and halts the controller 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 DENIED_WRITE.

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. A **return** will fail if evaluating <expr> fails, or will exhibit a security violation if evaluating <expr> does.

Expressions, conditions, and variables (<expr>, <value>, <cond_op> and <math_op>)

This programming language has <value>s that can be variables or integers:

x

Returns the current value of variable *x*. Variable *x* represents a list of integers, representing the variable's update history.
Fails if *x* does not exist or has been set to a rule
Security violation if the current principal does not have **read** permission on *x*. (DENIED_READ)

x.i

Evaluates to the ith most-recent integer taken by *x*. Recall that every time *x* is updated, a new integer is assigned. Index 0 (i.e., the expression *x*.0) is the current integer; the next-most recent is index 1 (i.e., *x*.1), etc.
Fails if *x* does not exist, has been set to a rule, or *i* is greater than the number of updates which have occurred to *x*.
Security violation if the current principal does not have **read** permission on *x*. (DENIED_READ)

i

This is a signed integer constant, which evaluates to itself. These should be treated as 32 bit values in two's complement format (i.e., like a Java int).

Expressions <expr> can be <value>s, predefined functions over a variable *x*'s current and former integer values, or an arithmetic operation <math_op> on a pair of <value>s. All arithmetic operations use signed

integer arithmetic with overflow, i.e., mod 2^{32} . If a sub-expression (the <value> part) fails or issues a security violation, then the <expr> itself does.

f x [, i, j]

This evaluates to the function *f* over the list of current and past values of *x*. If i and j are not given, f is computed over all past values. If i and j are given, then f is calculated over values of x from index i to index j. If i is greater than j, the function evaluates to 0. The function f can be one of the following:

- **mean** -- computes the arithmetic mean of the list of numbers. Works by computing their sum and then dividing by the count, using integer arithmetic (i.e., dropping any fractional component, post division). It's possible that the sum could overflow.
- **max** -- computes the maximum of the list of numbers
- **min** -- computes the minimum of the list of numbers
- **count** -- computes the length of the list of numbers

Fails if *x* does not exist.

Security violation DENIED_READ if the current principal does not have **read** permission on *x*.

<value> <math_op> <value>

This expression evaluates <math_op> on its two (integer) arguments. Recall that operations can both underflow and overflow (wrapping around), and that integer division rounds down, i.e., drops the fractional part. Dividing by zero is a failure.

The conditional guard <cond> has form <value> <cond_op> <value>. It appears in normal conditionals or in rules. When a <cond> is evaluated, it is either *true* or *false*; if *true* then the corresponding primitive command (in either the conditional or the rule) will be executed. See more on rules and conditionals below. When evaluating <value> <cond_op> <value>, the <value> components are evaluated to integers (resulting in a failure if one or both are not integers) and then compared according to the relational <cond_op> operator as expected. These are signed integer comparisons.

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 principals admin and hub whose passwords are given by the second and third command-line arguments; or "admin" and "hub" if the password are 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 DENIED_WRITE 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:

Fails if *p* does not exist

Security violation DENIED_WRITE 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 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**, 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).

Failure conditions:

Fails or exhibits security violation if evaluating <expr> does

Fails if *x* is already set to a rule

Security violation x exists and the current principal does not have **write** permission on *x*. (DENIED_WRITE)

Successful status code: SET

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*, 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 or exhibits security violation if evaluating <expr> does

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

Successful status code: LOCAL

if <*cond*> **then** <*prim_cmd*>

Evaluates the <cond>, and if the result is *true*, the given <prim_cmd> is executed. Otherwise, the <prim_cmd> is skipped. Note, for simplicity, you may assume that the <prim_cmd> cannot be an if/then

or set rule command to avoid the more complicated parsing necessary to handle this case. Instead, this nested structure will be an optional feature as described below.

Successful status code: If <cond> is *true* and the <prim_cmd> is executed, the <prim_cmd>'s status code is used. Otherwise, the status code COND_NOT_TAKEN is used.

set rule *x* = **if** <*cond*> **then** <*prim_cmd*>

Sets *x*'s value to the given rule object, which triggers whenever the rule is activated and its <cond> is *true*. When triggered (see discussion of rule triggering below), the rule executes <prim_cmd>. 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**, **toggle**, 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). Note, for simplicity, you may assume that the <prim_cmd> cannot be an if/then or set rule command to avoid the more complicated parsing necessary to handle this case. Instead, this nested structure will be an optional feature as described below.

Notes:

- No security checks are made on the variables in <cond> or <prim_cmd> when the rule is first set. These checks take place only when the rule is actually triggered.
- Rules are not first class. You cannot read a rule *x* as if it were a normal variable; any <expr> containing variable *x* where *x* is set to a rule will fail (see above). As such, the following program will fail:

```
set rule x = if temperature >= 80 then set air_conditioning = 1
set y = x
```

Failure conditions:

Fails if x already contains a normal value

Security violation if *x* exists and the current principal does not have **write** permission on *x.* (DENIED_WRITE)

Successful status code: SET_RULE

activate rule *x*

Toggles the rule *x* into the active state. This means that the controller will monitor the status of *x*'s condition and execute it when the condition is non-zero.

Failure conditions:

Fails if *x* does not exist.

Fails if *x* is not a rule.

Security violation if the current principal does not have **toggle** permission on *x*. (DENIED_WRITE)

Successful status code: ACTIVATE_RULE

deactivate rule *x*

Toggles the rule *x* into the inactive state. This means that the controller will not monitor the status of *x*'s condition.

Failure conditions:

Fails if *x* does not exist.
Fails if *x* is not a rule.
Security violation if the current principal does not have **toggle** permission on *x*. (DENIED_WRITE)

Successful status code: DEACTIVATE_RULE

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

When <tgt> is a sensor or output device *x*, indicates that *q* delegates <right> to *p* on *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). When <tgt> 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*, *q,* or *x* do not exist.
Security violation (DENIED_WRITE) if the running principal is not admin or *q;* if principal is *q* and <tgt> is variable *x*, then *q* must have **delegate** permission on *x*

Successful status code: SET_DELEGATION

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

When <tgt> is a variable *x*, indicates that *q* revokes a delegation assertion of <right> to *p* on *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 <tgt> 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*, *q,* or *x* do not exist.
Security violation (DENIED_WRITE) unless the current principal is admin, *p,* or *q*; if principal is *q* and <tgt> is variable *x*, then *q* needs **delegate** permission on *x*.

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** rights of *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 (DENIED_WRITE) if the current principal is not admin.

Successful status code: DEFAULT_DELEGATOR

print <expr>

Returns the result of the evaluated <expr> to the client (or fails/denies if <expr> does).

Successful status code: PRINT, <expr result>

Rule Execution

Once a program's main <cmd> has been executed successfully, the controller should consider all currently activated rules. These rules are considered in the order they were set (i.e., according to the most recent time **set rule** *x* = … was executed for each active rule x). For each rule x = **if** <cond> **then** <prim_cmd>, we take the following steps.

- The rule's condition <cond> is evaluated with the authority of the principal *p* who set the rule. If evaluating the condition results in a **failure or security violation**, then evaluation of that rule stops, the **rule is immediately deactivated** (so it won't be considered again until a program explicitly re-activates it), and the appropriate status code (FAILED or DENIED_X) is issued.
- If the condition is *true*, then the associated <prim_cmd> is executed with p's authority, as described above; if execution fails or results in a security violation, then all effects are rolled back, the rule is deactivated, and the appropriate status code is issued.
- If the condition is *false*, then nothing happens; i.e., no status code is issued.

For example, suppose principal bob's program ran the following:

```
set rule too_hot if temperature >= 80 then set air_conditioning = 2
activate rule too_hot
```

This sets a rule that checks the temperature sensor and sets the air_conditioning device if the temperature is getting too hot. This rule will be considered each time a program completes (including bob's program which first set the rule). If the condition is satisfied and bob has the appropriate privileges, then the **set** command will be run, updating air_conditioning to 2, and issuing a successful SET status code. On the other hand, if bob does not have read privilege on temperature, or does not have write privilege on air_conditioning, then the rule will fail with a security violation; in the former case it will issue a DENIED_READ status code, and in the latter a DENIED_WRITE. If bob does have read privilege on temperature, but the condition temperature >= 80 is not satisfied, then no status code is issued.

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**, **delegate**, **toggle**) on variable *x* is determined by the current *security state* (call it S_d), 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_d 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_d is empty. This statement adds delegation assertion x admin **read ->** bob to S_d. 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_d 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_d.

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

---

set delegation x bob **read** -> alice
set delegation x admin **write** -> anyone

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

- alice has **read** permission on x. This is because admin has **read** permission on x (rule 1) and admin delegates that permission to bob (rule 3, and first assertion in S_d), and bob delegates that permission to alice (rule 3, and second assertion in S_d).
- alice has **write** permission on x. This is because admin has **write** permission on x (rule 1), admin delegates this permission to anyone (rule 3, and third assertion in S_d), and since anyone has this permission then alice does (rule 2). (bob has **write** 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*. If the latter, *p* must have **delegate** permission on *x*. (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*. If the later, *p* must have **delegate** permission on *x*. (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.

Successfully executing **set delegation** *x p* <right> **->** *q* adds the delegation assertion *x p* <right> **->** *q* to S_d, while executing **delete delegation** *x p* <right> **->** *q* revokes assertion *x p* <right> **->** *q if it is explicitly present in* S_d; otherwise the command does nothing. Consider our second example above, after which S_d 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_d, nothing is removed. On the other hand, executing

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

serves to remove the assertion from S_d 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 arguments (admin and hub), 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

When starting up the server, if 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.

Input

Your server should read (and potentially process) an input program until the ******* terminator is read. At this point, *all further input should be ignored* (i.e., it does not constitute a parsing error). If the server receives an incomplete program (e.g., without the ******* terminator) your program can act in any of the following ways:

> It could return FAILED because some input that was read is malformed, or a read-in command tries to execute an illegal action.
> It could hang while waiting for additional input, most notably the ******* terminator, to complete the program (even if the input read to that point is malformed)
> It could issue the TIMEOUT status if input is not received within 30 seconds (which is what the oracle does; see below).
> *The program will never output non-error status for incomplete programs*

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_*X*"; 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 "output" field, whose value is a JSON representation of the <expr> as a string.
> For all status codes sent as part of rule execution, the extra JSON field "rule" is added, set to the name of the rule whose command was executed.

Here is an example output where an expression is returning the integer 27:

```
{"status":"RETURNING","output":"27"}
```

As an example of a rule's execution, consider the following example given previously:

```
set rule too_hot if temperature >= 80 then set air_conditioning = 2
```

This rule will be considered each time a program completes. If the condition is satisfied and bob has the appropriate privileges, then the following status will be issued after the rule executes

```
{"rule":"too_hot","status":"SET"}
```

On the other hand, if bob does not have read privilege on temperature, then the rule will fail with a security violation:

{"rule":"too_hot","status":"DENIED_READ"}

If bob does have read privilege on temperature, but the condition temperature >= 80 is not satisfied, then no status code is issued.

Optional features

Optional features can only be implemented in the order given below -- that is, Variable Indexes must be the first optional feature you implement (if any), followed by command blocks, etc.

Variable indexes

This feature permits a program to index a past value of a variable using another variable, not just an integer constant. We update the syntax of <value> expressions as follows:

<value> ::= *x* | *i* | *x.i* | *x.y*

The first three variants are as before. The last, x.y, says that we can allow variables x to be time-indexed not just by a constant i, but also by the (most recent) value of a variable y. For example

```
as principal admin password “admin” do
  set x = 10
  set x = 11
  set x = 12
  set y = 2
  return x.y
***
```

Running this program will return the value 10 (the value of x at index 2, since y’s value is 2).

Failure conditions:

Fails if *x* or *y* do not exist or are not integers

Fails if *x* does not have a value at index *y*

Security violation DENIED_READ issues if the current principal is not permitted to read either *x* or *y*

Nested if/then and rules; command blocks

This optional feature allows <prim_cmd>s to be nested within a <prim_cmd> in two ways forms.

First, in the basic specification, we indicated that the <prim_cmd> within the if/then and set rule commands could be assumed not to be another if/then or set rule command. In this optional feature, we allow these <prim_cmd>s to be nested. Therefore, the following command is now allowed:

```
if x > 0 then if y > 0 then set z = 0
```

Second, the command blocks feature permits grouping together primitive commands in a block. This feature is most useful when employed in conjunction with conditionals, rules, or loops (another optional feature, below). We extend the syntax of primitive commands as follows:

```
<prim_cmd> ::= ...
        begin <prim_cmd> <block>
<block> ::= \n <prim_cmd> <block>
      | end
```

This indicates that a block begins with the keyword **begin**, is followed (on the same line) by a primitive command, and then is followed either by the keyword **end**, or a series of primitive commands, one per line, the last of which concludes (on the same line) with the keyword **end**. For example, the following are legal command blocks:

```
 begin set x = 1 end
 begin set x = 1
  set y = 2
  if x > y then set z = 0 end
 if x < y then begin set x = 1
  set z = 2 end
```

On the other hand, the following are *not* legal

```
begin
 set x = 1 end
begin set x = 1
 set y = 1
end
```

Both of the illegal examples are wrong for similar reasons: Neither a **begin** nor **end** can appear on a line, on their own.

Execution of a block proceeds by executing commands one at a time, just like in a full program. If any command fails or issues a security violation, the entire block of commands does. If all commands in the block execute successfully, then one status code is issued per command.

For command blocks within a rule, only the first successful command in the block's output includes the additional 'rule' tag indicating the executed rule's name.

Resetting variable history

Assigning to a variable with **set** updates a variable's current value, remembering its old values. The **reset** command assigns to a variable but removes all of its history at the same time. The syntax is as follows:

<prim_cmd> ::= ...
reset *x* = <expr>

Thus, the following program would fail because after the reset, the variable x has no history:

set x = 1
reset x = 2
return x.1
Failure conditions: Same as for the **set** command
Successful status code: RESET (with no additional arguments/fields)

For loops

For loops work by setting a fresh iterator variable to an initial value and then executing the body of the loop until the iterator variable reaches the value of the bound; each iteration increases the iterator variable by 1, while the loop bound is determined at the outset of executing the loop. Here is the syntax

<prim_cmd> ::= ...
for *x* = <value> **to** <value> **do** <prim_cmd>

Here, x is the iterator variable. To execute this command, we evaluate the first <value> expression to some integer i, and the second to some integer j. Then we initialize x to i, treating it as we would a local variable. Each execution of the loop increases x by 1, until x exceeds j, at which point the loop stops, and x is removed from the store.

Notes:

- Updating the iterator variable x at the conclusion of each iteration should *overwrite* any prior history, as with the reset syntax above. Thus, at the start of the loop body's execution x.i should fail when i is not 0.
- Updates to the variables appearing in the bound will not affect loop termination. This is because the second <value> (the bound expression) is evaluated before the loop starts executing. For example, the following loop would iterate 10 times (not 5), returning 0.

```
as principal admin password “admin” do
 set x = 10
 for i = 1 to x do set x = x - 1
 return x
***
```

Failure conditions:

Fails if *x* already exists or if evaluating one of the values <value> or the <prim_cmd> does

Security violation issued if evaluating one of the values <value> or the <prim_cmd> does

Successful status code: FORLOOP followed by the status codes of each successful execution of the loop's body <prim_cmd>

- For the example above, the status code FORLOOP would be followed by ten SET status codes, one for each iteration of the loop body

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":["%PORT%","configuration","password"],
 "programs":[
  "as principal admin password \"password\" do\nset lights = 0\nreturn lights\n***",
  "as principal admin password \"password\" do\nreturn temperature\n***"
 ],
 "configuration": {
  "sensors": {
   "temperature":"80",
   "home_energy": "50",
   "smoke": "0",
   "owner_location": "0"
  },
  "output_devices": {
   "lights":"0",
```

```
    "alarm_status": "1",
    "air_conditioning": "0"
   }
  }
}
```

The oracle will provide outputs from the controller for the submitted programs. Here is output for the above query:

```
{
  "return_code": 0,
  "output": [
    [
      {
        "status": "SET"
      },
      {
        "status": "RETURNING",
        "output": "0"
      }
    ],
    [
      {
        "status": "RETURNING",
        "output": "80"
      }
    ]
  ]
}
```

The infrastructure will automatically assign a port for the oracle server to run on. You can specify this assigned port by using the %PORT% variable, which will be replaced with the assigned port in the command line arguments.

Timeouts

It is possible that a client may not send a complete program (which includes the *** terminator) to your server within a reasonable time. To handle this situation, the oracle will output a TIMEOUT status if it does not receive the *** terminator within 30 seconds. This behavior is **optional** in your implementation; see the input handling section for possible actions. If you do implement it, there is a test for it that will net you extra points.

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 dvotipka@cs.umd.edu 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 controller is minimizing **elapsed time**. That is, we want the controller 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 **confidentiality, integrity**, and **availability**. We say that confidentiality is violated if an attacker is able to read stored data (global variable, sensor values, or output device state) for which it is not granted access. We say that integrity is violated if an attacker is able to modify output device state or 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 because of an attacker's actions. We are interested in the security of sensor data and output devices 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 controller, but cannot view (or corrupt) others' communications with the

server. As such, break-it teams will provide evidence of a defect by providing a sequence of programs—a test— that when executed on the target controller will behave incorrectly or insecurely. See the Break-It Round submission requirements for detailed descriptions of these inputs. Together, the attacker can use these inputs to demonstrate incorrect or insecure behaviors of the target system. In particular:

We consider it a **security bug** in a targeted submission when running some program

- the oracle returns DENIED_READ but the target doesn't (confidentiality)
- the oracle returns DENIED_WRITE but the target doesn't (integrity)
- the oracle returns correctly (EXITING or RETURNING status), but the target says DENIED_READ or DENIED_WRITE (availability)
    - An availability security bug needs to demonstrate an attack by performing the action without causing a crash or availability problem, performing the attack, and then performing the initial action again and it leads to a crash or availability problem
- the oracle returns any status within 30 seconds but the target fails to return within a much longer window (availability).

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.). This does not include differences in sensor values.
Any test for which the oracle returns TIMEOUT cannot be used to claim a bug in a submission.

Build-it Round Submission

Each build-it team will have a gitlab repository created for them that all team members and the instructor staff will have access to. 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.

Do not make your repository public! This would be disadvantageous for you in the competition as other contestants might be able to see it. But more importantly, this would also constitute an academic integrity violation since this would mean sharing code with others not on your team.

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.

Break/Fix

In the second part of the competition, teams will be given access to each others' IoT controller implementations and will have the opportunity to submit "breaks" in the form of test cases. These breaks demonstrate correctness and security issues with the target team's implementation by comparing the results of the target's execution to that of the oracle. We describe the types of breaks, rules for breaking, and the submission method for break in more detail below.

When a valid break is submitted, the target team will be given access to the breaks. The team will then have the opportunity to fix their code, resolving the underlying bug triggered by the break, and gain points back on their build score. In the Fix-It round section below, we discuss rules for fixes and how fixes are to be submitted.

Break-It

Summary

During this round, breakers will attack build-it implementations of the IoT controller. The three types of attacks that breakers can demonstrate are **correctness**, **crash**, and **security** violations. The primary method for break submission, is through an input test file which demonstrates incorrect behavior by the target program (as compared to the oracle). In this case, teams must also submit a textual justification for the break in the break commit message. I.e., you must explain why the behavior constitutes a security violation according to the specification. The majority of this section focuses on this type of break submission.

Teams may also submit textual descriptions for breaks that are computationally infeasible to generate a test case for, but possible within the project specification (i.e., if the break requires a brute force attack that would take days to complete). Again, teams must include in their textual description a justification for the break. This type of break submission will be manually judged by the course instructors. We provide additional details at the end of this section regarding this type of break submission.

Note, for all break submissions, the commit message format given below should be used on submission.

You may submit up to 5 attacks against each build-it submission.

Scoring

Valid breaks affect both the submitting teams break score and the target team's build score. Once a break is validated, the submitting team's break score increases M points and the target team's build score decreases M points (M=100 for security, 50 for crash, and 25 for correctness, as described in the following sections). After the first 24 hour period after the break is validated until the vulnerability is fixed, the breaking team starts receiving additional points and the target team starts losing points (at a rate of M points per day). Consider the example where team A identifies a security violation in team B's code and team B does not fix it for 36 hours. Team A's break score would increase 150 points and team B's build score would decrease 150 points.

Once the fix is submitted, the break score is adjusted to account for the fact that multiple teams may have identified the same vulnerability. This is the case when a team submits a single fix, which resolves multiple breaks. In this case, breaking teams share the points evenly for the periods of time in which their breaks overlap. Consider a modification of our original example where team C submits a break for the same vulnerability identified by team A against team B, but does it 30 hours later. Team A is initially given 150 points (because their break lasted 36 hours) and team C is given 100 points (because their break only lasted 6 hours). However, after the fix is submitted, teams A and C split the 25 points from the last 6 hour period, updating team A's gained points to 137.5 and team C's to 12.5.

General Input Variables

Command line arguments passed to the server may depend on variables that are dynamically generated. Therefore, the grader will replace variables present in arguments with their true value. These are the input variables that will be replaced:

- %PORT% - Port of the server

Security Violations

Security violations are worth the most points: 100 points (split between all teams that find the security violation).

To demonstrate a security violation, breakers will submit an input test file that contains a list of input programs to run on the server and the JSON for the initial sensor and output device configuration. Note, input programs can include programs from the admin, hub, and other users created by the admin, giving the attacker full control of the operational environment. For reference, we include an example list of possible sensors and output devices below.

The grader will then run each program on both the oracle and the targeted build-it submission. We consider it a security bug in a test's targeted submission when

- the oracle returns DENIED_READ but the target doesn't (confidentiality)

- the oracle returns DENIED_WRITE but the target doesn't (integrity)
- the oracle returns correctly (EXITING or RETURNING status), but the target says DENIED_READ or DENIED_WRITE (availability)
    - An availability security bug needs to demonstrate an attack by performing the action without causing a crash or availability problem, performing the attack, and then performing the initial action again and it leads to a crash or availability problem
- the oracle returns any status within 30 seconds but the target fails to return within a much longer window (availability).

Any test for which the oracle's status is `TIMEOUT` cannot be used to claim a bug in a submission. See the "invalid break submissions" section below for more details.

Here is an example input test file for a security violation, which is very similar to what is provided to the oracle during build-it, but also mentions the type of break (security violations can have type "confidentiality", "integrity", or "availability") and the ID of the targeted team:

```
{
    "type": "integrity",
    "target_team": 9,
    "arguments": {
        "argv": ["%PORT%",”%CONFIG%”,"password"],
        "base64": false
    },
    "programs": [
        {"program": "as principal admin password \"password\" do\nset x =
\"x\"\nreturn x\n***", "base64": false},
        {"program": "as principal admin password \"wrongpassword\" do\nreturn
x\n***", "base64": false}
    ],
    “configuration”: {
        "sensors": {
            “temperature”:”80”
         },
        "output_devices": {
            “lights”:”0”
        }
    }
}
```

The previous example would demonstrate an integrity violation where the server ignores the password and allows a user who does not properly authenticate to login. The second program would return the value of `x`, when the program should return DENIED_WRITE.

Binary base64 encoded inputs are indicated by providing the optional boolean field `"base64"`. `"argv"` inputs should not contain any nul (0) bytes.

Correctness Violations

Correctness violations are worth the least points: 25 points (split between all teams that find the correctness violation).

To demonstrate a correctness violation, breakers will once again submit an input test file that contains a list of inputs to run on the server (including admin, hub, and general user programs). The grader will then run each input on both the oracle and the targeted build-it submission. We consider it a correctness bug in a test's targeted submission when
both the oracle and target provide output, but the output differs (different status code, returned value, etc.).

As with security violations, any test for which the oracle returns **TIMEOUT** cannot be used to claim a bug in a submission. See the "invalid break submissions" section below for more details.

```
{
        "type": "correctness",
        "target_team": 9,
        "arguments": {
                "argv": ["%PORT%","%CONFIG%","password"],
                "base64": false
        },
        "programs": [
                {"program": "as principal admin password \"password\" do\nset x =
                        \"x\"\nreturn x\n***", "base64": false},
                {"program": "as principal admin password \"password\" do\nreturn
                        x\n***","base64": false}
        ],
        “config”: {
                "sensors": {
                        “temperature”:”80”
                 },
                "output_devices": {
                        “lights”:”0”
                }
        }
}
```

Binary base64 encoded inputs are indicated by providing the optional boolean field `"base64"`. `"argv"` inputs should not contain any nul (0) bytes.

Crash Violations

Crash violations are potentially security relevant, so they are worth more points than correctness violations, but fewer than security violations: 50 points (split between all teams that find the violation).

A crash occurs when a program unexpectedly terminates due to a violation of memory safety. To demonstrate a crash, submit a test with the **type** attribute set to **crash**. The given arguments and input programs should cause the target implementation to crash with a memory safety violation (like overflowing a buffer, causing a use after free, etc). Here is an example input test file for a crash violation:

```
{
        "type": "crash",
        "target_team": 9,
        "arguments": {
                "argv": ["%PORT%","%CONFIG%","password"],
                "base64": false
        },
        "programs": [
                {"program": "as principal admin password \"password\" do\nset x =
\"x\"\nreturn x\n***", "base64": false},
                {"program": "as principal admin password \"password\" do\nreturn x\n***"}
        ],
        “config”: {
                "sensors": {
                        “temperature”:”80”
                 },
                "output_devices": {
                        “lights”:”0”
                }
        }
}
```

Binary base64 encoded inputs are indicated by providing the optional boolean field `"base64"`. `"argv"` inputs should not contain any null bytes.

The grader will automatically be able to detect or reject crash violations in some cases. For the other cases, breaks will be marked for manual judgement, and course staff will perform a manual review.

Invalid Break Submissions

Correctness violations against ambiguities or differences between the specification and oracle are invalid and ***will be rejected***.

Any break submission that exhausts resources is not valid (for example, causing the target to run out of memory is not a valid break). To remove any ambiguity about this, we have instrumented the oracle to count the size of the normal state and the security state on the server. If this size exceeds 10 million bytes, the oracle will signal a timeout failure, and the break will be considered invalid. Roughly speaking, we compute the size as follows:

* each variable stored on the server counts as 1 byte plus the length of the variable name plus the size of the value assigned to that variable
* the size of values is computed as follows
  - a string's size is its length, in characters
  - a record is 1 byte for each field, plus the length of the field's name, plus the size of the field's value
  - a list is 1 byte for each element, plus the size of the element
* each delegation assertion of the form `x` _q_ `<right> ->` _p_ stored on the server counts as the sum of the size of the variable name `x` (its length), 1 byte for the choice of `<right>`, and the size of principals _q_ and _p_ (which are 1 byte if either `anyone` or `admin` or their length otherwise).

In addition, we assume that all updates to the store during a program's execution add to the size of the store at the start of the program, since the old values need to be retained, for possible rollback. Once the
program completes successfully, old values for variables and the delegations are discarded.

But, to be honest, none of the above really matters. This is just an automated way of avoiding resource exhaustion attacks. And in any case, the actual consumed size will differ based on the implementation language and strategy, but the above definition is a ballpark figure that is precisely enforced by the oracle.

Sensors and Output Devices

Here are some examples of sensors and output devices connected to the hub and their associated possible value ranges. Note, this is not a complete list of all possible sensors and output devices. The input JSON file may contain more sensors and output devices (or less or none) than the ones given on this list. Therefore your implementation should handle an arbitrary set of sensors and output devices.

Sensors

1. **temperature** - the current temperature of the home in degrees fahrenheit
2. **fridge_temp** - the current temperature of the smart fridge in degrees fahrenheit
3. **home_energy** - the current energy usage of the home in kW
4. **fridge_energy** - the current energy usage of the smart fridge in kW
5. **owner_location** - the home owner's physical location;
6. **smoke** - indicates whether smoke is detected

Output Devices

7. **air_conditioning** - indicates whether the air conditioning unit is currently off, set to heat or set to cool
8. **lights** - indicates whether the home's lights are off or on
9. **window_shades** - indicates whether the home's window shades are currently down or up
10. **Alarm_sys** - indicates whether the home's alarm system is currently deactivated or active
11. **door** - indicates whether the home's door is currently closed or open

Disputes

Build-it teams will also have the opportunity to dispute invalid break submissions during the fix-it round. A dispute must argue that the break test does not illustrate a violation of the specification according to the above rules (e.g., because the oracle was incorrect).

Builder's code

You can find the builder's source code by visiting your participation page, and clicking on **Builder's Code** in the sidebar.

Break submissions - Test cases

Break submissions are made via git. Create a **break** folder in the top-level folder of your team's repository (which is the same repository used to submit code during the build-it phase). Within that folder, you will specify test cases as directories. Each test directory will contain a JSON file, **test.json**, following the above formatting instructions --- one file per test case. Any changes to a test case directory will replace the previous break submission.

A textual description is required that justifies your break, and describes the bug in the target submission. This description must *explain why the test case is demonstrating a bug*; it is not sufficient to just textually describe what the test case is doing. This file is important for the fix-it round, and may take some effort. For example, if a team fails to check for **write** permission on the **set** command, a valid

break would be a program which writes to a variable (e.g., *door*) the calling principal (e.g., *bob)* does not have the **write** permission on. The description of why this test case is demonstrating a bug would be as follows:

"Bob does not have write access for the door output device they are attempting to set in the second program. However, the target IoT system does not return DENIED_WRITE and instead allows the door to be written, causing an integrity violation."

Your break description should be put in a file called **description.txt** within the test's directory. This file should be no more than 300 words long. It may refer to relevant portions of this spec (link the appropriate part reachable from the spec's table of contents) and/or line numbers in the submitted code, if you like. For example, if you create a security test with a JSON file at **break/break1/test.json**, there should be a textual description in **break/break1/description.txt**.

Break submissions - Textual description

If it is impossible to produce a functioning break test case for computational reasons (e.g., significant brute forcing is required), you may submit a textual description of the break. This description should clearly indicate why the issue is a *bug,* identify where the issue occurs in the target team's implementation. Additionally, this description should include instructions for how to exploit the bug in a general setting. That is, it should indicate the limitation of the BIBIFI infrastructure or setting that prevents an exploit and lays out the steps to take "in the real world" to exploit it instead. Note, this type of submission is only valid for security violations.

This description will be manually evaluated by the instructor team and should provide sufficient detail for them to clearly understand the issue and how it would be exploited. If the instructors deem that the described break could be demonstrated with a test case input, the textual description submission will be deemed *invalid* and the team will be instructed to submit a test case.

Additionally, these submissions should not target issues beyond the given threat model. For example, if the submission describes a confidentiality violation by snooping traffic between the hub and controller, this would be *invalid* because we consider the network secure in our threat model.

Textual break submissions are also made via git. Create a **breakdesc** folder in the top-level folder of your team's repository (which is the same repository used to submit code during the build-it phase). Your break description should be put in a file called **description.txt** within a new directory for this vulnerability. This file should be no more than 500 words long. For example, the textual description could be placed in **breakdesc/break1/description.txt**. It may refer to relevant portions of this spec (link the appropriate part reachable from the spec's table of contents) and/or line numbers in the submitted code, if you like. When committed, this break's commit message should be of the form given below for break commits.

The description.txt file should have the following format:

Type: [confidentiality|integrity|availability]

Target Team:

Description of bug:

Why the bug is a valid break according to the specification:

Where the issue occurs in the target team's implementation:

Specific steps to exploit the issue:

Why it is infeasible to produce a test case to break this bug within BIBIFI:

Scoring

Breaking teams that submit valid break descriptions will receive 250 pts toward their break score. If another team identifies the same vulnerability later, this score will be divided evenly between teams. Note, there is no increase in this value over time and no points are subtracted from the target team's build score.

To disincentivize spurious break description submissions, teams submitting *invalid* break descriptions will incur a 25 pt deduction from their break score. This penalty will only be assessed against teams who have previously submitted **two** invalid break descriptions.

Fix-It Round

Summary

The fix-it round is an opportunity to gain build-it points back.

You will be given access to valid breaks against your code once they have been submitted (this will occur simultaneously with the Break-It round). It may be that many of these breaks, though superficially different, target the same underlying flaw. Therefore, when you fix a flaw, several of the breaks may no longer work. These will be unified: You will only lose points for one of them, not all of them. Therefore, some of the points you have already lost (for all of the duplicate breaks) will be restored.

Even if you are not in the running to win the build-it competition, it is **in your interests to fix your code**.

First of all, you may be in a better position that you realize. In past contests, we have found that one fix addresses six breaks, on average; that's a pretty good bump to your score.

Finally, taking the time to fix each vulnerability will give you a deeper understanding of the vulnerabilities exploited and how to mitigate them in the future.

Scoring

Valid fixes can have a significant effect on a team's build score (M=100 for security, 50 for crash, and 25 for correctness). First, because teams lose M points every 24 hours a break is not fixed, fixing the break prevents further point loss. Additionally, if multiple breaks target the same vulnerability (i.e., they can be resolved with a single fix), then the fixing team only loses points for a single break. Consider the case where team A and team C identify the same security vulnerability in team B's code at the same time and team B does not fix this vulnerability for 28 hours. Initially team B would lose 400 points because these are considered to be two separate breaks. However, if team B produces a single fix to resolve both breaks, they would instead only lose 200 points.

Fix setup

Teams will make edits directly to the code in their build directories. Commits made during this round to the build directory will be considered Fix commits (please use the commit message format given below for Fix Commits).

Fix submissions - Code update

Your fixes should address individual break submissions.

A fix should address the root cause of a particular test case failing. The fix should be the smallest change you can make to get this test to pass, while addressing the general underlying issue in your code. There is an inherent tension here as a fix of the underlying issue could require a significant code change if the system design needs to be changed. What we are trying to avoid is single commits that address multiple, unrelated issues that do not share common underlying factors. If you have a question about whether a particular fix would or would not be allowed, please ask the instructors prior to making the change.

Now it may be that the change you make to fix one test case causes other test cases (e.g., those submitted other different teams with the same underlying problem) to pass as well. If so, then you will get points back for those tests.

It is important that each fix submission **only address a single underlying bug or vulnerability in the code**. We will then judge whether we agree that your changes are really covering the issue uncovered by this test. If we disagree, the fix will be disallowed and no points will be given back. For example, if you hardcode a check for the specific break's test program that reports correct output without actually evaluating the program, this would not actually resolve the issue.

To submit a fix, fix your implementation in **build/**. Commit this fix (with the commit message structure given below) and note the git commit hash. This is your *fix submission*.

Note that **you should make no commits to your **build/** directory that are not part of fix submissions**. If you do, we will **disqualify all subsequent fixes.**

Once pushed, fixes will be run on the required correctness tests from build-it, as well as the break-it tests that are claimed to be fixed. If the fix passes all these tests, the fix will be marked for judgement; otherwise it will be rejected. We will judge fixes throughout the round.

Fix submissions - Textual description

Again, if it is impossible to produce a working fix for due to course constraints (e.g., complete system rebuild on the last day), you may submit a textual description of the fix. This description should clearly indicate the specific issue triggered by the break, lay out a specific set of changes necessary to remedy the break if given sufficient resources, and give a detailed description of how the stated fix would resolve the underlying issue triggered by the break. This description will be manually evaluated by the instructor team and should provide sufficient detail for them to clearly understand your solution. If the instructors deem that the described fix could be implemented in a reasonable time, the textual description submission will be deemed *invalid* and the team will be instructed to submit a fix.

Textual fix submissions are also made via git. Create a **fixdesc** folder in the top-level folder of your team's repository (which is the same repository used to submit code during the build-it phase). Your fix description should be put in a file called **description.txt** within a new directory for this fix. This file should be no more than 500 words long. For example, the textual description could be placed in **fixdesc/mynewfix/description.txt**. It may refer to relevant portions of this spec (link the appropriate part reachable from the spec's table of contents) and/or line numbers in the submitted code, if you like. When committed, this fix's commit message should be of the form given below for fix commits.

The description.txt file should have the following format:

Break IDs: list of breaks exploiting the bug that this fix resolves

Description of bug the break triggered:

Code changes necessary to fix the bug:

Why the given fix would resolve the bug:

Why the given fix is infeasible to implement:

Scoring

Scoring for textual descriptions of fixes is very similar to actual code fixes. Once a description has been judged, the associated breaks will no longer trigger build score decreases every 24 hours and duplicate breaks will no longer be scored separately. However, because the code is not actually fixed, the breaking teams will continue to receive M points (where M is determined by the violation type) every 24 hours on their break score. Additionally, other teams may submit the same break for the same issue and receive points, though this will not cause a decrease in the target team's build score.

Disputes

If you discover a break against your system that you believe is incorrect, write a justification for why you think it is invalid. The justification (no more than 100 words per test case) should indicate why your system's behavior is acceptable according to the spec, even if the oracle behaves differently. Team leaders can submit break disputes on the break's submission page. We will consider them during the judging phase.

Commit and Design Spec Requirements

Summary

In addition to the code developed during the course, we also require each team provide textual descriptions of their systems' design and reasoning for changes to the code. This includes both a system design document, which will be iteratively updated as development progresses, and a commit descriptions for each change to the system. Below we give a details for each of these required documents.

System Design Document

In the system design documents, teams will answer three main questions: how system is organized to meet required and optional functions, how an attacker can affect the system, and mitigations in place against potential attacks.

Regarding the question of system organization, teams must present all components of their system. This includes describing each independent component and its function and how components interact with each other. This document provides the basis for instructors to understand and evaluate your code, so make sure the document is sufficiently detailed.

In addition to laying out the system, teams also need to show the attack surface of the system. This includes any points where an attacker can manipulate input to the system and the system's actions. For each point where an attacker can manipulate the system, teams should indicate the range of possible actions an attacker could take.

Finally, the design document should also indicate any mitigations put in place to protect the system from the identified potential attacks. For each mitigation, indicate the security requirement (i.e., confidentiality, integrity, and availability) the mitigation intended to protect. Note, it is possible that a single mitigation addresses multiple security requirements.

Teams are free to structure the design document according to their preference (e.g., visual and/or textual descriptions). However, we recommend Lucidchart as potential option since all UMD students have access to the online graphing tool through Terpware and it allows collaborative editing.

When to submit:

The design document should match the current state of the system, so any commit that includes changes to the system's design should also include an updated design document. This includes changes made during both the Build-It round and Break-It/Fix-It round.

Commit Requirements

Any time a commit is made to a teams' codebase, it should be accompanied by a description of the change. Below we discuss the requirement of commit messages for each round of the competition.

Build Commits

During the Build-It round, as teams are working to create their secure IoT system, commit messages should include the following information:

- Description of the change - A high-level discussion of what changes were made and how this affects program functionality.
- Reason for the change - That is, why wasn't the previous code sufficient?
- Requirement it is associated with - What functionality, performance, and/or security requirement is the commit meant to address (e.g., message confidentiality). Note, this can include multiple requirements.
- How did you come up with this change? - Was there any specific inspiration for the change such as lecture topics, prior experience, or StackOverflow?
- Was an update to your design document required?

Note, to make a longer commit, use the command "git commit" (without giving the "-m" option). This will open vim and allow you to add your commit message. For a more detailed walkthrough of this process, see this Medium post.

When to submit:

Commits should be atomic as much as possible. That is, they should be focused on adding a single feature or fixing a single bug. This can include changes to multiple files, but should be associated with the same overarching feature or fix. For example, changing the encryption algorithm used between the hub and controller requires updating the algorithm used by both hub and controller programs. This would only require one commit.

If it is impossible to split the commit by function or fix, indicate all associated requirements. Err on the side of more commits or ask the instructors for clarification.

Break Commit

During the Break-It round, as teams are looking for and exploiting vulnerabilities in other teams' systems, commit messages should include the following information:

- Description of issue in target's program - A high-level discussion of what vulnerability is being exploited.
- How was the break found? - Were you trying to exploit a vulnerability you mitigated in your own system, a vulnerability another team found in your system, based on an in-depth code review, or something else?

- Requirement broken - What functionality, performance, and/or security requirement does the break target (e.g., message confidentiality). Note, this can include multiple requirements.
- Is it a fixed version of a failed break? - That is, did you attempt to submit the break previously, but it failed? What did you change?
- How hard was the break to find? - Rate the difficulty of identifying the vulnerability on a scale of 1-5, with 1 being not difficult at all to 5 being extremely difficult.
- How hard was the break to exploit? - Rate the difficulty of creating a successful break once you identified the vulnerability on a scale of 1-5, with 1 being not difficult at all to 5 being extremely difficult.

When to submit:

Break commit messages should be submitted each time a break is submitted.

Fix Commits (Break Reflections)

When you can fix your code
During the Fix-It round, as teams are remediating vulnerabilities identified in their systems, commit messages should include the following information:

- What was wrong about your implementation? - That is, what vulnerability in your code did the breaking team take advantage of?
- What caused the problem? - What do you think lead the vulnerability to be introduced in the first place? For example, was it a misunderstanding of the requirements or a miscommunication between teammates?
- How did you fix the vulnerability?
- How confident are you that your code is now secure? - Rate on a scale from 1-5, with 1 being not confident at all and 5 being extremely confident.
- How hard was it to identify the problem? - Based on the submitted break, rate how hard it was to determine the underlying vulnerability on a scale from 1-5, with 1 being not difficult at all to 5 being extremely difficult.
- Was an update to the design required?

When you are unable to fix your code
If you are able to determine the vulnerability exploited by the breaking team, but determine that it is not practical to remedy (it would require weeks of work), then you can make a commit stating your decision not to fix. Note, your justification for not fixing will be assessed by the instructors who will determine whether it is sufficiently impractical to deserve points toward your criterion score. For this type of commit, clearly label the line (or lines) of vulnerable code and include a commit message of the following format:

- What was wrong about your implementation? - That is, what vulnerability in your code did the breaking team take advantage of?
- What caused the problem? - What do you think led the vulnerability to be introduced in the first place? For example, was it a misunderstanding of the requirements or a miscommunication between teammates?
- **Why is it impractical to fix the vulnerability?**
- **How would you fix the vulnerability if you had sufficient time or resources?**
- How confident are you that the fix you described above would be secure? - Rate on a scale from 1-5, with 1 being not confident at all and 5 being extremely confident.
- How hard was it to identify the problem? - Based on the submitted break, rate how hard it was to determine the underlying vulnerability on a scale from 1-5, with 1 being not difficult at all to 5 being extremely difficult.
- Was an update to the design required?

When to submit:

A fix commit should be submitted for each vulnerability (this can cover multiple breaks if they all exploit the same vulnerability). If you are able to fix the code, the fix commit message can be submitted with an update to the code to remedy the identified vulnerability. Otherwise, you should clearly mark the vulnerable code and submit a non-fix justification as described above.