Instructors: Matt Fredrikson, Jean Yang TA: Samuel Yeom

Due date: 2/16/2017 at 11:59pm

Assignment 2

In the previous assignment, you implemented a simple policy where only `admin` or the owner of a variable could write to it, and anyone else could read from it. In this assignment, you will add support for more nuanced policies that principals can change over time.

First, we provide a specification of the policy language. We provide theoretical exercises to help you understand the policy model, and then we ask you to implement the policy model. We will tell you exactly what you will need to add to your previous implementation, so read carefully!

Policy model. The policy model associates principals and variables with permissions. We represent the permissions as follows:

⟨**right**⟩ ::= `read` | `append` | `write` | `delegate`

If a capability is not explicitly given for a permission in this description, then it is not granted by that permission. We describe the permissions below:

`read`**:** If principal p has `read` permission on x, then p is allowed to refer to x in an expression.

`append`**:** If a principal p has `append` permission on x, and x evaluates to a list, then p is allowed to concatenate items to the value referenced by x.

`write`**:** If principal p has `write` permission on x, then p is allowed to make arbitrary changes to x in a command. Note that `write` permissions grant strictly more capability than `append`: any command that requires `append` permission can be executed if the principal has `write` permission instead on the variable in question.

`delegate`**:** If a principal p has `delegate` permission on x, then p is allowed to grant or revoke permissions on x for other principals in the system.

Permissions are additive and, with the exception of `append` and `write`, distinct in the capabilities they grant. A principal with only `read` access to x is unable to change it in any way or delegate permissions. A principal with only `write` access is able to change or append to x, but unable to read or delegate permissions on it. In order to read and make changes to x, the principal needs *both* `read` and `write` permissions.

The set of principals in the system correspond to identifier strings. (See Homework 1 for restrictions on the set of allowed principal identifiers). There are two default principals that must be present on the server:

`admin`**:** The administrator is granted full permissions on any variable in the system, as well as the ability to add new principals and terminate the server.

`anyone`**:** The `anyone` principal is used to grant permissions to all users on the system. A principal p has permission ⟨**right**⟩ on x if principal `anyone` has that permission on x.

For a full description of which permissions are required to execute each primitive command, consult the handout for Assignment 1, and in particular the portions with the header **Security violations**. Note that two commands that you have already implemented have semantics that are updated in the following section.

Language updates. To prevent collisions when naming principals in commands, you should update your implementation of the `create principal` command as follows:

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

Failure conditions:

- p already exists as a principal; this includes the case where p is `anyone`.

Security violations:

- Current principal is not `admin`

Successful status code: `CREATE_PRINCIPAL`

`set` x $=$ ⟨**expr**⟩**:** Updates global variable x in the store to map to the result of evaluating ⟨**expr**⟩. If x does not already exist, this command creates it. The principal who successfully executes this command is given `read`, `append`, `write`, and `delegate` permissions on x.

Failure conditions:

- Evaluating ⟨**expr**⟩ results in failure.

Security violations:

- x already exists in the server state, and the current principal does not have `write` permissions on x.
- Evaluating ⟨**expr**⟩ results in a security violation.

Successful status code: `SET`

To support user-defined policies, you will add support for two new primitive commands:

`set delegation` x ⟨**right**⟩ `->` p**:** Updates the access control policy so that principal p is given permission ⟨**right**⟩ on variable x.

Failure conditions:

- p does not exist as a principal.
- x does not exist as a variable.

Security violations:

- The current principal p_c invoking the command satisfies either of the following:
  1. p_c is not `admin`.
  2. p_c does not have `delegate` permissions on the variable x.

Successful status code: `SET_DELEGATION`

`delete delegation` x ⟨**right**⟩ `->` p**:** Updates the access control policy so that principal p no longer has permission ⟨**right**⟩ on variable x. If p does not currently have ⟨**right**⟩ permission on x, then the command has no effect.

Failure conditions:

- p does not exist as a principal.
- x does not exist as a variable.

Security violations:

- The current principal p_c invoking the command satisfies either of the following:
  1. p_c is not `admin`.
  2. p_c does not have `Delegate` permissions on the variable x.

Successful status code: `DELETE_DELEGATION`

Formal model. To implement a reference monitor for this policy model, you should use a security automaton as your formal model. The states in the security automaton are defined by the following:

- P, a set of tuples (p, s) where p is a principal and s is a password string.
- c, the currently-acting principal.
- V, a set of variable identifiers that currently exist on the server.
- A, a set of tuples (p, x, r) where p is a principal, x a variable identifier, and r an access permission.

The initial state of the automaton is:

- $P = \{(\texttt{admin}, s), (\texttt{anyone}, \perp)\}$, where s is the administrator's password (given as a command line argument), and $\perp$ is the empty string.
- $c = \texttt{anyone}$, so the server is executing on behalf of `anyone` until a script presents the correct password credentials for a given principal.
- $V = \emptyset$
- $A = \emptyset$

The input symbols, and the conditions for which they are generated, are as follows:

SetCurrentPrin(p, s)**:** Script attempts to login as principal p using password s.

Oper(v, r)**:** Current principal invokes and operation that requires permission r on variable v.

AddRight(p, v, r)**:** Current principal delegates permission r on variable v to principal p.

RemoveRight(p, v, r)**:** Current principal revokes permission r on variable v from principal p.

AddPrincipal(p, s)**:** Current principal creates a new principal named p with password s.

UpdatePrincipal(p, s)**:** Current principal updates principal p's password to s.

AddVariable(v)**:** Current principal creates a variable named v.

Terminate**:** Current principal attempts to terminate the server.

1 Transition function

The security automaton (partially) defined above reads input symbols that correspond to policy-relevant actions. For this problem, you need to define the transition function that encodes the policy described earlier in the **policy model** section. That is, for each input symbol, describe which states the symbol is allowed to execute in, and what state the automaton should be in afterwards.

To get you started, we've filled in the transition for the first symbol, $\mathsf{SetCurrentPrin}(p, s)$. Complete the problem by filling in the transition function for the remaining symbols. Note that you can use arbitrary formulas when specifying states in your description; see the *Enforceable Security Policies* paper by Schneider for more examples of textual automaton descriptions. You should modify the source file for this document directly, replacing the placeholder comments with your solution.

Solution

(a) $\mathsf{SetCurrentPrin}(p, s)$

Start state: $(p, s) \in P$ (a transition only exists if p is a principal in the current state and s is the correct password for that principal)

End state:

- $P := P$ (no change)
- $c := p$ (set current principal to p)
- $V := V$ (no change)
- $A := A$ (no change)

(b) $\mathsf{Oper}(v, r)$

(c) $\mathsf{AddRight}(p, v, r)$

(d) $\mathsf{RemoveRight}(p, v, r)$

(e) $\mathsf{AddPrincipal}(p, s)$

(f) $\mathsf{UpdatePrincipal}(p, s)$

(g) $\mathsf{AddVariable}(v)$

(h) $\mathsf{Terminate}$

2 Mediation points

Now you are almost ready to implement the security automaton as a reference monitor for the server. But we first need to specify how the security automaton interacts with the server's functionality. For each construct in the scripting language syntax, identify which input symbols must be generated by a correct implementation to ensure complete mediation.

To get you started, we've filled in the answer for one of the commands. Note that you do not need to list the symbols generated by nested subparts of the syntactic elements. For example, `return` has a nested ⟨**expr**⟩ subpart. When providing your answer for `return`, you do not need to list the symbols that could be generated when ⟨**expr**⟩ is evaluated.

Solution

(a) `as principal` p `password` s `do \n` ⟨**cmd**⟩ $***$

- SetCurrentPrin(p, s)

(b) `exit`

(c) `return` ⟨**expr**⟩

(d) `create principal` p s

(e) `change password` p s

(f) `set` x $=$ ⟨**expr**⟩

(g) `append to` x `with` ⟨**expr**⟩

(h) `filtereach` y `in` x `with` ⟨**bool_expr**⟩

(i) Constant expressions: `[]` (empty list), s (string constant), `true`, `false`

(j) Variable expressions: x, $x.y$

(k) Record constructor: $\{x_1 = \langle\textbf{value}\rangle, \ldots, x_n = \langle\textbf{value}\rangle\}$

(l) `if` ⟨**bool_expr**⟩ `then` ⟨**expr**⟩ `else` ⟨**expr**⟩

(m) `equal`(⟨**value**⟩, ⟨**value**⟩), `notequal`(⟨**value**⟩, ⟨**value**⟩)

3 Implement the reference monitor

Now that you've formalized a reference monitor and identified the necessary mediation points, update your implementation by adding a reference monitor corresponding to your formalism. Whenever executing a command would result in a policy violation, the server should abort all changes made by the current program, update the state of the reference monitor to reflect the aborted changes, and return `DENIED` status.

To help get you started, we have updated the following parts of the code to provide some of the necessary functionality. Feel free to rewrite these portions, or build on them in your implementation.

`ast.ml`: Added cases for `set delegation` and `delete delegation` to the `cmd` type, and added a `Delegate` constructor to the `right` type.

`auth.ml`: We have added several definitions to `Auth` that you can choose to build on in your implementation, or rewrite if you would like to do things differently.

- Containers for the automaton state definition given earlier in the handout, and a `refmon_state` type that uses the container to define the current reference monitor state.
- Two helper functions `add_principal` and `check_password` that demonstrate working with the containers and updating the automaton state.
- An `input_symbol` type matching the options for input symbols listed in the security automaton description.

`interp.ml`: Added `ResponseSetDelegation` and `ResponseDeleteDelegation` constructors to the `response` type. Note that you should update your evaluation functions to account for the new `set delegation` and `delete delegation` commands, as we have not done this for you.

`lexer.mll`, `parser.mly`: Updated the syntax description to account for the new `set delegation` and `delete delegation` commands.

To implement the reference monitor, we recommend that you proceed as follows:

1. Copy the updates discussed above from the handout code to your current implementation.
2. Add a function to initialize the automaton state, given the administrator's initial password.
3. Add functions that take an input symbol and update the automaton state. Use your answer for problem 1 as a guide when implementing these functions.
4. Using your solution to problem 2 as a guide, locate the corresponding mediation points in your code, and insert calls to functions in `auth.ml` responsible for processing input symbols and updating automaton state.
5. At all points, write tests for your implementation!

4 Bounded availability

Recall from lecture the example policy informally described as "no `send` after `read`". Suppose we want to add a restriction that the buffer must eventually become available to other programs. In other words, our program must eventually stop reading.

(a) Is our new policy prefix-closed? Why or why not?

(b) Is our new policy finitely refutable? Why or why not?

Because an execution monitor can only enforce policies that are both prefix-closed and finitely refutable, it cannot enforce availability.

(c) Modify the availability restriction and prove that the resulting policy is both prefix-closed and finitely refutable. Keep in mind that this new restriction is called *bounded* availability.

Solution

(a)

(b)

(c)