Instructors: Matt Fredrikson, Jean Yang TA: Samuel Yeom

Due date: April 28 at 11:59pm

Assignment 5

1 Audit (32 points)

In class, we discussed audit, and presented a logic that resembled the authorization logic from earlier in the semester. **In this assignment, you will extend your code from Assignment 2 so that the server can provide a proof that each executed command was properly authorized.** In particular, you will design and implement an audit mechanism that is based on the authorization logic and consists of the following parts:

Logger. The logger will record a sequence of commands that were executed by the server. Importantly, the logger does *not* need to record all *attempted* commands. If a user submits a program that fails due to a security violation, or fails to commit for any other reason, then the information recorded by the logger does not need to reflect the event.

Prover. Given an entry from the log file, the prover constructs a proof in the authorization logic that the recorded event was allowed according to the security policy from Assignment 2.

The set of principals will be the set of usernames in the server's memory, including `admin`. Principals can be added when `admin` requests the appropriate action. In the authorization logic presented in class, we had $p ::= \textbf{key}(s) \mid \textsf{identifier} \mid p.s$. In this assignment, no one will sign any statements, nor will there be any ambiguity about which principal an identifier refers to. Hence, we will remove the parts of the authorization logic that have to do with $\textbf{key}(s)$ and $p.s$. The resulting logic is reproduced below for your convenience.

Statements, where s is a string and p is a principal (**delegates** has been slightly modified):

$$\begin{array}{rcl} \phi & ::= & \textbf{action}(s) \\ & \mid & p \textbf{ says } \phi \\ & \mid & p \textbf{ speaksfor } p \\ & \mid & \textbf{delegates}(p, p, \textbf{action}(s)) \\ & \mid & \phi \rightarrow \phi \end{array}$$

Inference rules (Delegate-E has been slightly modified):

$$\textsc{Says-I2}\ \frac{\phi}{p \textbf{ says } \phi} \qquad \textsc{Says-Impl}\ \frac{p \textbf{ says } (\phi_1 \rightarrow \phi_2) \qquad p \textbf{ says } \phi_1}{p \textbf{ says } \phi_2}$$

$$\textsc{Speaks-E1}\ \frac{p_1 \textbf{ says } (p_2 \textbf{ speaksfor } p_1) \qquad p_2 \textbf{ says } \phi}{p_1 \textbf{ says } \phi}$$

$$\textsc{Delegate-E}\ \frac{p_1 \textbf{ says delegates}(p_1, p_2, \textbf{action}(s)) \qquad p_2 \textbf{ says action}(s)}{p_1 \textbf{ says action}(s)}$$

You will use the code in the file `proof.ml` to represent statements and proofs in the authorization logic. You will need to add `proof.ml` to `Makefile` in an appropriate place so that all dependencies between files are satisfied. In particular, `proof.ml` requires `ast.ml`, and some of your other files will depend on `proof.ml`.

In the given code, the type `stmt` represents statement ϕ, and the type `proof` represents an application of an inference rule. For example, in `SaysI2 of proof * stmt`, the `proof` should be a proof of ϕ, and the `stmt` should be p **says** ϕ. In rules with two premises, the first `proof` should be a proof of the premise on the left, and the second should be a proof of the premise on the right. `Given of int * stmt` represents statements that are assumed to be true, either because it is one of the policies or because it is in the log. In the former case, the `int` should be zero, and otherwise, it should be the line number of the statement in the log.

For this assignment, you can assume that the input programs will contain neither `delete delegation` commands nor the principal `anyone`. Operationally, this means that you do not need to think about these programs when you complete this problem, and none of the test cases used for grading will invoke such programs.

Part 1. (6 points) First, you will need to make the authorization logic a bit more specific to the operations carried out by your server. The authorization logic was general, and encapsulated all types of requests in the **action**(s) statement, which corresponds to a request for the action denoted by string s. In this setting, there is a limited set of actions that can follow from requests. In the space provided below, list a minimal set of statements necessary to achieve complete auditing of the server's history. When you have identified a complete set, there should no longer be a need for the **action**(s) statement in the logic's grammar. Then, write the code for the OCaml type `action` in the skeleton code provided in `proof.ml`.

Example. One type of action corresponds to reading a variable x, so we might extend the syntax with a **read**(x) action.

Solution 1.

Part 2. (6 points) Now that you have specialized the authorization logic to the server, you need a set of policies that make it possible to prove authorizations using the rules we discussed in class. Recall that a policy in this context is simply an assumption, in the form of a statement from the logic, that the prover can use towards a goal. In this context, the prover's goals will be statements of the form:

$$\texttt{admin}\ \textbf{says}\ \textbf{action}(s)$$

where **action**(s) stands for one of the actions that you wrote in Part 1. In the space provided below, write the policies and briefly justify them.

Hint. Think about what happens when users create variables. What assumptions do we need to give the prover so that statements of the form above can eventually be proven? Because the authorization logic does not have any quantifiers, you will need to add more assumptions when variables or principals are added to the server. Policies are often of the form `admin` **says** $(\phi_1 \to \phi_2)$, which can be used in proofs with Says-Impl.

Solution 2.

Part 3. (10 points) Construct a mapping from *commands* executed by the server to *log entries*, and update your implementation with this mapping to create the logger. You should make sure that your log enables the following functionality.

1. The audit log should account for **every command** executed and successfully committed on the server, in the order in which they were executed.
2. Given your mapping from commands to log entries, it should be straightforward for an auditor to locate the log entry corresponding to a given command.
3. An auditor should be able to use the log, in combination with your answer for Part 2, to construct an authorization proof for any successful command.

In particular, log entries should be `stmt`s the form:

$$p \textbf{ says } \phi$$

where p is the principal who executed the command that maps to ϕ. When the changes made by the commands are committed, you should use the provided function `string_of_stmt` to convert the log entries to strings and append them to `log.txt`, with each entry on its own line.

Part 4. (10 points) Implement the prover. After each successful execution of a command, the prover should produce a `proof` of the statement `admin` **says** ϕ, where ϕ is the statement appearing in the corresponding log entry p **says** ϕ. Use the provided function `string_of_proof` to convert the proof to a string and append it to `proof.txt`. The grader will use a modified version of the function `string_of_proof` to automatically check the proofs, so it is essential that you call this function.

2 Declassification (18 points)

Recall the **match** declassification operator

$$\frac{\Gamma \vdash e_1 : \ell_1 \quad \Gamma \vdash e_2 : \ell_2}{\Gamma \vdash \mathbf{match}(e_1, e_2) : \ell_1 \sqcap \ell_2},$$

where $\mathbf{match}(e_1, e_2)$ evaluates to `true` if and only if e_1 and e_2 evaluate to the same value.

In class, we used an example with the typing judgments $\Gamma \vdash \texttt{guess} : \mathsf{L}$ and $\Gamma \vdash \texttt{pin} : \mathsf{H}$. We assumed that the attacker has the ability to repeatedly call $\mathbf{match}(\texttt{guess}, \texttt{pin})$ with its choice of the value of `guess` and observe the output. We showed that, even with this ability, a deterministic attacker cannot always figure out the value of an n-bit string `pin` in poly(n) time, which means that **match** is in some sense a "safe" declassification operator.

For the declassification operators defined below, determine whether an attacker can always use it to figure out the value of an n-bit string `pin` in poly(n) time. If so, describe how. If not, prove why not. All declassification operators below have the same typing rule as **match**.

(a) (3 points) Greater than: $\mathbf{gt}(\texttt{guess}, \texttt{pin})$ evaluates `true` if and only if `guess`, interpreted as an integer, is greater than the n-bit string `pin`, interpreted as an integer.

(b) (3 points) Error-correcting match: $\mathbf{ecm}(\texttt{guess}, \texttt{pin})$ evaluates to `true` if and only if `guess` is an n-bit string that differs from the n-bit string `pin` by at most 1 bit.

(c) (3 points) Prefix: $\mathbf{prefix}(\texttt{guess}, \texttt{pin})$ evaluates to `true` if and only if `guess` is a prefix of the n-bit string `pin`.

In the following declassification operators, `guess` is replaced by `indices`, which is a subset of $\{1, \ldots, n\}$ that indicates which bits of the n-bit string `pin` are relevant. One way to think about this is that each bit of `pin` is the answer to a yes/no question about a person, and the declassification operator is used to ask questions about a subset of the n people.

Because it would be trivial to recover `pin` if `indices` could contain only one element, we add the restriction that `indices` must have size at least 10. In your answer, consider only the case where $n \gg 10$.

(d) (3 points) Sum: $\mathbf{sum}(\texttt{indices}, \texttt{pin})$ evaluates to the sum of the relevant bits. (Relevance is indicated by `indices`.)

(e) (3 points) Parity: $\mathbf{parity}(\texttt{indices}, \texttt{pin})$ evaluates to `true` if and only if the sum of the relevant bits is even.

(f) (3 points) Majority: $\mathbf{maj}(\texttt{indices}, \texttt{pin})$ evaluates to `true` if and only if more than half of the relevant bits are 1.