Lab 3
Proof-Carrying Authorization

15-316: Software Foundations of Security & Privacy
Frank Pfenning

Due Friday, December 6, 2024
150 points

In this lab you will explore proof-carrying authorization from two perspectives.

In Part 1 you have to prove authorization to several servers that we make available on the `linux.andrew.cmu.edu` machines in order to obtain access to a secret. You will then submit the secrets as text files to Gradescope to receive credit. Your file names should be `pca_auth1.txt`, `pca_auth2.txt`, etc, where each file contains a single line with the secret you discovered as discussed below.

In Part 2 you have to write a server that provides access to a resource if a suitable proof is presented and denies access if it is not. You will submit your server code to the autograder, where we will test it against a collection of valid and invalid authorization requests. The file `lab3.zip` should contain your server code. As in Labs 1 and 2, we will call

```
make
```

after unzipping this file. This should create an executable `pca_serve` satisfying the spec given below.

The Authorization Logic

Unlike Labs 1 and 2, the programming language TINY plays no role in this lab. Instead there is the language of formulas of the authorization logic and the language of proofs. Whenever an authorization logic is designed, there is a tradeoff between expressiveness of the logic and the difficulty of proving authorization. The fragment for this lab is inspired by Datalog, extended to

include A **says** P.

Variables	x, y	(capitalized identifiers)	
Constants	c, d	(lowercase identifiers)	
Terms	t, s	$::= \quad x \mid c$	
Predicates	f, g	(lowercase identifiers)	
Atoms	p, q	$::= \quad f(t_1, \ldots, t_n)$	
Antecedents	D	$::= \quad p \mid G \to D \mid \forall x.\, D(x) \mid A$ **says** D	
Goals	G	$::= \quad p \mid A$ **says** p	
Proof Vars.	v	(lowercase identifiers)	
Policies	Γ	$::= \quad v_1 : P_1, \ldots, v_n : P_n$	
Proofs	M, N	$::= \quad v$	
		$\mid \quad M\ N$	$(\to E)$
		$\mid \quad M\ [t]$	$(\forall E)$
		$\mid$ **let** $\{v\}_A = M$ **in** N	$(\mathbf{says}E)$
		$\mid \quad \{M\}_A$	$(\mathbf{says}R)$
		$\mid$ **let** $v = M$ **in** N	(cut)

Principals A, B, etc. are just terms, either constants or variables. The bidirectional proof-checking rules for this fragment of authorization logic are presented below. The left rules in the sequent calculus are turned into so-called *elimination rules*, so $\to L$, $\forall L$, and $\mathbf{says}L$ are renamed to $\to E$, $\forall E$, and $\mathbf{says}E$.

$$\frac{v : P \in \Gamma}{\Gamma \vdash v \Rightarrow P}\ \mathsf{id} \qquad \frac{\Gamma \vdash M \Rightarrow P \to Q \quad \Gamma \vdash N \Leftarrow P}{\Gamma \vdash M\ N \Rightarrow Q} \to E \qquad \frac{\Gamma \vdash M \Rightarrow \forall x.\, P(x)}{\Gamma \vdash M\,[t] \Rightarrow P(t)}\ \forall E$$

$$\frac{\Gamma \vdash M \Rightarrow A\ \mathbf{says}\ P \quad \Gamma, v : P \vdash N \Leftarrow A\ \mathbf{aff}\ Q}{\Gamma \vdash \mathbf{let}\ \{v\}_A = M\ \mathbf{in}\ N \Leftarrow A\ \mathbf{aff}\ Q}\ \mathbf{says}E \qquad \frac{\Gamma \vdash M \Leftarrow A\ \mathbf{aff}\ P}{\Gamma \vdash \{M\}_A \Leftarrow A\ \mathbf{says}\ P}\ \mathbf{says}R$$

$$\frac{\Gamma \vdash M \Leftarrow Q}{\Gamma \vdash M \Leftarrow A\ \mathbf{aff}\ Q}\ \mathbf{aff} \qquad \frac{\Gamma \vdash M \Rightarrow P' \quad P' = P}{\Gamma \vdash M \Leftarrow P} \Rightarrow/\Leftarrow \qquad \frac{\Gamma \vdash M \Rightarrow P \quad \Gamma, v : P \vdash N \Leftarrow \delta}{\Gamma \vdash \mathbf{let}\ v = M\ \mathbf{in}\ N \Leftarrow \delta}\ \mathsf{cut}$$

In the rule of cut, δ is either Q or A **aff** Q. We read (and likely implement) $\Gamma \vdash M \Rightarrow P$ as a function that given Γ and M returns a P or fails, and $\Gamma \vdash M \Leftarrow P$ as a function that given Γ, M, and P either succeeds or fails. It is up to you to decide how to implement the auxiliary judgment $\Gamma \vdash M \Leftarrow A$ **aff** P. Note that you will need an equality function on formulas for the $\Rightarrow/\Leftarrow$ rule, and that you will need substitution for the $\forall E$ rule.

Part 1: The Attack [4 * 15 = 60 points]

The servers `pca_serve<n>` are invoked as follows on the `linux.andrew` machines:

```
% ~fp/bin/pca_serve<n> <filename>.pca <filename>.pcx
```

where $n \in \{1, 2, 3, 4, 5\}$. The server uses a policy you can find in `~fp/bin/pca_serve<n>.pca`. The command line arguments have the following meaning:

- *filename*`.pca` may contain additional affirmations, all of which must be made by the user running the command, identified by their Andrew id. It is possible for this file to be empty.

- *filename*.pcx contains a typing $M : P$ that is supposed to grant authorization. The proof M is your responsibility, the authorization goal P is specified for each server (see below).

Important: The secret will be a hash of some information including the Andrew id of the user running the command above. If you work in a team, this user should submit the discovered secrets to Gradescope. Each should be in a separate file with the name `pca_auth<n>.txt`. At this point we believe that only 4 of the 5 authorizations can be obtained.

When the server terminates, it will print one of the following:

- `success <secret>` (with exit code 0). You proved authorization.
- `error` (with exit code 1). The file does not exist, or the file or input does not have the correct format.
- `failure` (with exit code 2). The file is syntactically correct, but the proof was not a valid proof of authorization.

Here are the goal formulas for authorization for each server:

1. `admin says mayGrade(<user>, <user>, hw1)` where `<user>` is your user id.
2. `journal says publish(article1)`
3. `hipaa says mayRead(abo, fpfmr)`
4. `admin says mayRead(bob, file1)`
5. `admin says mayOpen(runming, ghc6017)`

Part 2: The Server [90 points]

Your server, `pca_serve`, is invoked with

```
% ./pca_serve <filename>.pca <filename>.pcx
```

and should parse *filename*.pca as a policy Γ and *filename*.pcx as a typing $M : P$ and perform the following steps:

1. Verify that *filename*.pca is a well-formed policy. This means:
   (a) Each $v : P$ in Γ should have a unique v.
   (b) In each declaration, all term variables should be quantified and not shadow one another. For example, `v : A says hello(fp);` should be rejected since A is not quantified, and `v : !A. is_friend(A) -> !A. A says hello(fp);` should be rejected since the inner `!A` shadows the outer one.
2. Verify that $\Gamma \vdash M \Leftarrow A$. You do not need to worry about the quality of error messages.

You can validate your test files with our server:

```
% ~fp/bin/pca_serve <filename>.pca <filename>.pcx
```

Your server should return the same exit codes as our server from Part 1, except that you have no secret to reveal.

Language Reference

Syntax for Formulas

```
<idchar> ::= [A-Za-z_0-9]
<lower> ::= [a-z] <idchar>*
<upper> ::= [A-Z] <idchar>*

<term>  ::= <lower> | <upper>
<terms> ::= <term>
          | <term> ',' <terms>

<form>  ::= <lower> '(' <terms>? ')'
          | <form> '->' <form>
          | '!' <upper> '.' <form>
          | <term> 'says' <form>
          | '(' <form> ')'
```

Ambiguities are resolved as follows:

- The affirmation `A says` has highest precedence
- The logical operator `->` is right associative
- The quantifier `!x.` is a prefix with lower precedence than `->`

Syntax for Policies and Proofs

```
<pvar>  ::= <lower>

<policy> ::=
           | <pvar> ':' <form> ';' <policy>

<proof> ::= <pvar>
          | <proof> <proof>
          | <proof> '[' <term> ']'
          | '{' <proof> '}' '_' <term>
          | 'let' '{' <pvar> '}' '_' <term> '=' <proof> 'in' <proof>
          | 'let' <pvar> '=' <proof> 'in' <proof>
          | '(' <proof> ')'

<typing> ::= <proof> ':' <form>
```

Ambiguities are resolved as follows:

- Juxtaposition is left associative, so `x y z` is `(x y) z`
- `let {v}_A = M in` abd `let v = M in` are prefixes with lower precedence than juxtaposition

Example Files

First, the policy in `example.pca`.

```
c1 : admin says (!X. p(X) -> q(X));
c2 : admin says p(nineteen);
```

Then, the proof in `example.pcx`.

```
{
  let {x1}_admin = c1 in
  let {x2}_admin = c2 in
  x1 [nineteen] x2
}_admin
:
admin says q(nineteen)
```

We verify this with

```
% ~fp/bin/pca_serve example.pca example.pcx
(...some output...)
success <proof>
```