Managing Policy Updates in Security-Typed Languages

Nikhil Swamy, Michael Hicks, Stephen Tse, Steve Zdancewic

Computer Security Foundations Workshop

Venice, Italy

June, 2006

Context

- The security behavior of long running programs changes frequently.
  - Principals can enter and leave the system
  - A principal's privilege level can change
- *But, most security-typed languages assume that these kinds of changes never occur.*

Contributions

- RX: a new security-typed language
  - Maintains the confidentiality and integrity of data even in the presence of an evolving security policy.
  - Includes a novel treatment of labels as roles derived from a role-based policy language.
  - Models information flows through the state of the policy by a formal treatment of metapolicy.
  - Gives the programmer control over the effect of policy updates by using a transactional model of memory.

Outline of the Talk

I. Motivation and challenges
II. A model for policy derived from role-based policy languages
III. RX: A programming language integrated with policy updates
    1. Roles as labels and policy queries
    2. Integrating policy updates into a language
    3. Avoiding inconsistent policy updates using a transactional semantics
    4. Preventing information leaks through the policy with metapolicy
IV. Security Properties for RX
V. Related Work
VI. Future Work

Arbitrary Policy Change is Dangerous

- The timing of an update can cause undesirable information flows.
- The context in which an update occurs can can allow an adversary to control which data she is allowed to observe.
- Policy updates can cause the policy to become a channel of secret information.

Timing of an Update is Critical

- Only members of *clinicX* can view patientRec
- Updating policy at L2 allows *Doc* to view patientRec even when not a member of *clinicX*
- Update at L4 invalidates the check in L1, but the flow has already occurred
- Update at L6 might seem to be ok, but can also be problematic

L1: if (*Doc actsFor clinicX*)) {
L2:
L3: show(*Doc*, patientRec)
L4:
L5: }
L6:

resign*(Doc, clinicX)* (policy change)

Transitive Flows

- Update at L4 deletes an *actsFor* edge between A and B and simultaneously adds one between C and A.
- L4 invalidates the check at L1, but it isn't within the scope of L1 --- should such an update be ok?
- The result is that the contents of Brec are copied to Crec, and C *actsFor* B is not stated by π or π'.

L0: initial π : A *actsFor* B, C

L1: if (A *actsFor* B) {
L2: Arec := Brec
L3: }

L4: change π' : B, C actsFor A

L5: if (C *actsFor* A) {
L6: Crec := Arec
L7: }

Policy Integrity

- Principals state their security preferences through the policy.

- Suppose conditionX is controlled by the attacker; then the update in L2 can be triggered by the attacker.

- Who is affected by the update in L2? Policy ownership is important.

L0: initial π : A *actsFor* B, C
L1: if (conditionX)
L2: change π’ : A, B actsFor C

Policy as an Information Channel

- Policy updates can depend on secret data.
- If attacker discovers that DrBob is Pat's doctor, then he can conclude that Pat has HIV.

L1: if (patHasHIV) {
L2: change π': *DrBob actsFor Pat*
L3: }

Design Goals

- One size does not fit all with respect to the timing of policy updates. Must provide some way of controlling when policy updates take effect.
- Principals state their security requirements through policy. Changes to policy must be authorized by the appropriate principals.
- The state of a changing policy can become a channel of information. Must prevent leaks through this channel too.

RX: A Secure Language with Policy Updates

- Types contain a security label constructed from RT roles.
- A *query construct* that examines the runtime policy to establish relations between roles.
- An *update construc*t that allows the policy to be changed from within the program itself.
- A *transactional semantics* that allows the programmer to control how policy updates take effect.
- A formal treatment of information *flows through the state of the policy.*

RX uses a role-based policy language

Why not the DLM?

- Policy in the DLM consists of

  1. A lattice specifying the actsFor relation between principals
  2. Data tagged by labels specifying how the data is permitted to be used.

     - A label is owned by a principal and is literally a set of principals.

- Unclear ownership of the actsFor lattice makes it difficult to constrain *who* can change the lattice

- Labels as literal sets means that policy change requires a relabeling of data

- The actsFor hierarchy is too coarse-grained. A principal delegates all his privileges to another or none.

RT_0: A Role-based Policy Language

Roles are interpreted as sets of principals

$[\![\rho]\!]_\Pi$ includes all principals X

where $\rho \leftarrow X \in \Pi$

as well as $[\![\rho']\!]_\Pi$

where $\rho \leftarrow \rho' \in \Pi$

principal	P		
principal sets	X	$::=$	$\{P_1, \ldots, P_n\}$
role	ρ	$::=$	$P.r$
policy stmt	s	$::=$	$\rho \leftarrow X \mid \rho_1 \leftarrow \rho_2$
policy	Π	$::=$	$\{s_1, \ldots, s_n\}$

A sample policy

Pat.doctors	$\leftarrow$	$\{DrSue\}$
Pat.doctors	$\leftarrow$	$Clinic$.staff
$Clinic$.staff	$\leftarrow$	$\{DrAlice, DrBob\}$

Benefits of a Role-based Policy

- Owned Roles: The role A.r is owned by principal A
  - Only A can add or remove statements defining A.r
- Membership is distinct from delegation
  - A.r $\leftarrow$ B states that A considers B to be in the A.r role
  - A.r $\leftarrow$ B.r states that A considers all members of B.r to also be in A.r. B can introduce new members into A.r by altering B.r
- Ownership and Delegation together define who can change which parts of a policy

Roles as Labels

atomic labels	L	$::=$	ρ
compound labels	ℓ	$::=$	$(L_C, L_I) \mid \ell \sqcup \ell$
types	t	$::=$	bool
security types	τ	$::=$	t_ℓ

- Atomic labels are roles; roles are interpreted as sets
  - Adds a level of indirection: by changing the definition of a role the security level of a type can change, but the label does not.
- Labels contain a confidentiality and an integrity component --- compound labels are interpreted as a pair of sets
- Labels are arranged on a lattice according their interpretation

A Program Updates Its Own Security Policy

- Can add or delete RT_0 statements from the policy
  - ∂_1 ::= add *Pat*.docs $\leftarrow$ *Clinic*.staff
  - ∂_2 ::= del *Clinic*.staff $\leftarrow$ *DrBob*
- Individual ∂'s are grouped together to take effect atomically.

Timing of Updates

```
S1:  if(Pat.healthRecords ⊑ Clinic.staff)
       clinicRec := patSymptoms;
S2:  if(leaveClinic)
       update(del(Pat.doctors ← Clinic.staff));
```

- Assume clinicRec is confidential to members of *Clinic*.staff and patSymptoms to *Pat*.healthRecords.
- Assignment in S1 is justified by the policy query
- The policy update in S2 may alter the result of the query in S1
  - Should such an update be allowed?
  - What if S2 was nested within S1?

Transactions Control Update Timing

$$\text{statements} \qquad S \quad ::= \quad \ldots \mid \texttt{trans}_Q\, S$$

- RX provides a declarative construct for specifying a scope within which policy updates must respect past and future flows.
- All memory effects that occur within S are logged as in a transaction.
- Q represents a set of policy assumptions which if violated by an update in S cause the transaction to be rolled back.
- Potential leaks that can occur due to rollback are eliminated by the type system.

Policy as an Information Channel

- Runtime configuration of a program includes a memory *and* a policy
    - The attacker has a view of both memory and policy
- As policy evolves, the attacker can gain information by observing the policy too.
- If attacker discovers that DrBob is Pat's doctor, then he can conclude that Pat has HIV.

L1: if (patHasHIV) {
L2: update(*Pat*.docs ← DrBob)
L3: }

Metapolicy : Policy is data too

- For each role ρ, $C(\rho)$ is the set of principals that can interpret ρ as a set.
  - $C(\rho)$ is the confidentiality metapolicy.
- Similarly, $I(\rho)$ is the set of principals that *trust* the definition of ρ.
  - $I(\rho)$ is the integrity metapolicy.

Preventing Leaks through Policy

- Typechecker accepts this only if it can show (similar to memory updates)
    - Confidentiality of patHasHIV is not greater than C(*Pat*.docs)
    - Integrity of patHashHIV is not less than I(*Pat*.docs)
- Prevents the attacker from learning patHasHIV, and from effecting an unauthorized change to *Pat*'s policy.

L1: if (patHasHIV) {
L2: update(*Pat*.docs ← DrBob)
L3: }

Requirements of a Metapolicy

- Delegation introduces dependences between roles
  - A.r $\longleftarrow$ B.r in the policy means that information flows from B.r to A.r
    - Any change to B.r is reflected in the interpretation of A.r
- Metapolicy for B.r cannot be stricter (more confidential, less trustworthy) than A.r
- Also require I(A.r) to include at least A
  - The definition of a role is trusted by the owner

Noninterference

- Configurations of a program include policy and memory
  - Observability of policy is determined by metapolicy $C(\cdot)$
  - Memory observability is standard
- RX programs preserve the low-equivalence of a pair of configurations until a policy change declassifies policy or data to the attacker
- Obtain an end-to-end guarantee by piecing together non-declassifying subtraces
- Timing and termination insensitive

Related Work

- FCS 2005, Hicks et al
- Broberg & Sands, Flow Locks
- Almeida-Matos & Boudol, CSFW 2005 (Nondisclosure)
- ... (to do)

Future Directions

- Multi-threaded and distributed setting
  - Expect transactions to be useful here
- A hierarchy of policies and metapolicies to provide better control over policy evolution
- Policies communicated between processes
- Applied to
  - Medical information systems
  - Cross-domain security in a mostly trusted environment --- e.g. military intelligence

Summary

- RX supports inlined policy updates, both additions and revocations
- Provides the programmer with control to maintain a consistent policy
- A framework for metapolicy to control information leaks through policy
- Uses a role-based language to provide a natural administrative model for policy

http://www.cs.umd.edu/projects/PL/RX

EXTRA SLIDES ....

A Sample Policy in RT_0

$$\begin{array}{lcl}
Pat.\text{doctors} & \leftarrow & \{DrSue\} \\
Pat.\text{doctors} & \leftarrow & Clinic.\text{staff} \\
Pat.\text{insurers} & \leftarrow & \{BCBS\} \\
Pat.\text{healthRecords} & \leftarrow & Pat.\text{doctors} \\
Clinic.\text{staff} & \leftarrow & \{DrAlice, DrBob\} \\
Clinic.\text{insuranceCos} & \leftarrow & \{BCBS, Aetna\} \\
DrPhil.\text{self} & \leftarrow & \{DrPhil\}
\end{array}$$

All of *Pat*'s doctors can view her health records

All staff at *Clinic* can considered *Pat*'s doctors

RX Term Syntax (Partial)

$$
\begin{array}{llll}
\text{queries} & q & ::= & L_1 \sqsubseteq L_2 \\
\text{expressions} & E & ::= & \texttt{true} \mid \texttt{false} \mid x \mid E_1 \oplus E_2 \\
\text{statements} & S & ::= & \texttt{skip} \mid x := E \mid S_1; S_2 \\
 & & & \mid \texttt{while}\ (E)\ S \mid \texttt{if}\ (E)\ S_1\ S_2 \\
 & & & \mid \texttt{if}\ (q)\ S_1\ S_2
\end{array}
$$

- Queries q examine the runtime policy to establish the lattice ordering relation between *atomic* labels

- In the statement $\mid \texttt{if}\ (q)\ S_1\ S_2$ the static semantics permits S1 to assume the label ordering q

A Program Updates Its Own Policy

$$
\begin{array}{llll}
\text{policy stmt} & s & ::= & \rho \leftarrow X \mid \rho_1 \leftarrow \rho_2 \\
\\
\text{update} & \delta & ::= & \texttt{add}\ s \mid \texttt{del}\ s \\
\text{updates} & \Delta & ::= & \delta \mid \delta, \Delta \\
\text{statements} & S & ::= & \ldots \mid \texttt{update}\ \Delta
\end{array}
$$

- Can add or delete statements from the policy
- Individual ∂'s are grouped together into a Δ to take effect atomically
- Paper treats policy statements *s* as expressions allowing updates Δ to be constructed at runtime
  - More restrictive syntax presented here assumes that all updates are known statically

Some Typing Judgments

$$\begin{array}{lcll} \text{policy context} & Q & ::= & \{q_1, \ldots, q_n\} \\ \text{typing context} & \Omega & ::= & (\Gamma, \text{pc}, Q) \end{array}$$

$$\frac{\Omega[Q = \Omega.Q \cup \{q\}] \vdash S_1 \qquad \Omega \vdash S_2}{\Omega \vdash \texttt{if}\ (q)\ S_1\ S_2} \qquad \frac{\Omega.\Gamma(x) = t_\ell \qquad \Omega \vdash E : t_\ell \qquad \Omega.Q \vdash \Omega.\text{pc} \sqsubseteq \ell}{\Omega \vdash x := E}$$

$$\frac{\Omega \vdash E : \text{bool}_\ell \qquad \Omega[\text{pc} = \Omega.\text{pc} \sqcup \ell] \vdash S_i \qquad i \in \{1, 2\}}{\Omega \vdash \texttt{if}\ (E)\ S_1\ S_2}$$

- Ω consists of an environment, a pc label, and a *policy context Q*
- Top-left rule: *Q* accumulates the the results of policy queries
- Standard rules for assignments and if-stmt:
  - *Q* is used to establish label ordering

The who, what, when and how of policy change

- Which principals are allowed to change the policy?
- What parts of the policy are they allowed to change?
- When during execution can the change take place?
- How is a change reflected in the program's behavior?

Choosing a Security Property

How much attention to pay to "*Past Flows*"?

- Suppose A:=B is consistent with Π, but not consistent with Π'
- *Should we rule out Program* P *as insecure?*
- What if the assignment A:=B was not already executed?
- Similar issue with "*Future Flows*"

Program P

<policy = Π>

…

A := B;

…

<update policy to Π'>

…

C := D

The least we require is for all flows exhibited by a program to be consistent with the *current* policy

Static Reasoning about Dynamic Policy

- Static enforcement permits a strong security guarantee
- But, we still want the actual runtime policy to be indeterminate
- Need to combine a static and a dynamic approach
  - The program must *interact* with the state of the policy before causing a flow to occur. (Similar to access control)