

A Proof-Carrying File System

Deepak Garg

Carnegie Mellon University

CyLab Student Seminar
February 12, 2008

- The Problem:

- Proof-carrying Authorization well understood [AF99,Bau03,...]
- Always based in higher-order logic
- Many authorization systems at first-order
 - (ABLP93, RT, XACML, Binder, Keynote, ...)
- Seemingly easier meta-theory, easier policy analysis, easier theorem proving
- Why not do PCA at first-order?

- The Steps:

- Design a suitable logic
- Prove meta-theorems
- Deploy the logic in PCA!

Why a File System for PCA?

- Unexplored by PCA
 - Great for a thesis!
- Little infrastructure needed
 - One laptop suffices
- Real reason: File systems really need PCA!

Why PCA for a File System?

- Traditional file system security:
 - Unix permissions: rwxrwxrwx
 - AFS and VxFS: access control lists (ACLs)
 - Windows: ACLs

- Bits/ACLs do not often suffice
 - Impossible to express: Environmental constraints
 - Time, Configuration files, ...
 - Difficult to express: Attribute based authorization
 - Principals (Group membership, student, professor, ...)
 - Files (Public, confidential, private, executable, ...)
 - Difficult to manage: rapidly changing policies
 - Source of many errors
 - Next to impossible: automatically check policy snapshot for errors

- PCA handles all these!

Logic: Basic Ingredients

- First-Order, Constructive
- Explicit “says” construct, with time
 $A, B ::= P \mid A \supset B \mid \forall x.A \mid \langle K, I \rangle A$
- Axioms/proof rules from modal logic
- Proof-theory: cut-elimination, consistency
- Explicit proof-terms
- Simple proof verification
- No explicit delegation (encodable)
- Linearity (?)

Proof-carrying Authorization Example

- Policies:
 - Administrator says, “If owner of a file F says K may read F , then K may read F ”
 - Administrator says, “Alice owns foo.pdf”
 - Alice says, “Bob may read foo.pdf during 2008”
- Policies (formalized):

$$\begin{aligned} p1 : & \langle \text{admin}, [R] \rangle \forall K, K', F, I. \\ & (\text{owns}(K', F) \wedge (\langle K', I \rangle \text{read}(K, F))) \\ & \supset \langle \text{admin}, I \rangle \text{read}(K, F) \\ p2 : & \langle \text{admin}, [R] \rangle \text{owns}(\text{Alice}, \text{foo.pdf}) \\ p3 : & \langle \text{Alice}, [2008] \rangle \text{read}(\text{Bob}, \text{foo.pdf}) \end{aligned}$$

- Scenario: Bob wants to read foo.pdf on 2/12/2008
 - Bob must construct a proof M such that
 $M : \langle \text{admin}, [2/12/2008] \rangle \text{read}(\text{Bob}, \text{foo.pdf})$

Proof-carrying Authorization Example

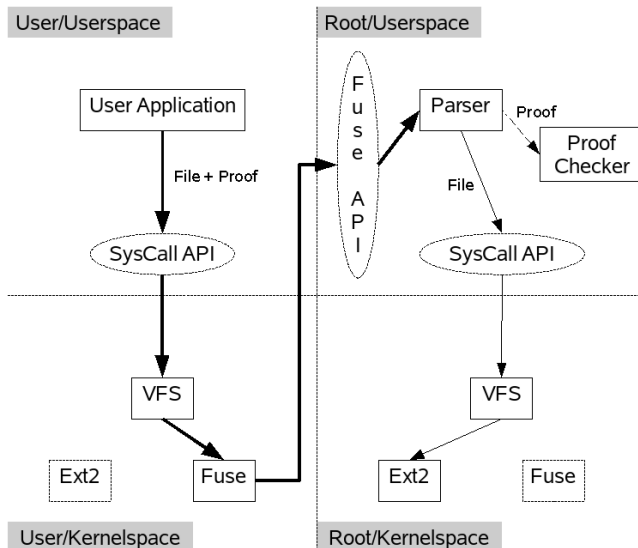
$$\begin{aligned} p1 : & \langle \text{admin}, [R] \rangle \forall K, K', F, l. \\ & (\text{owns}(K', F) \wedge (\langle K', l \rangle \text{read}(K, F))) \\ & \supset \langle \text{admin}, l \rangle \text{read}(K, F) \\ p2 : & \langle \text{admin}, [R] \rangle \text{owns}(\text{Alice}, \text{foo.pdf}) \\ p3 : & \langle \text{Alice}, [2008] \rangle \text{read}(\text{Bob}, \text{foo.pdf}) \end{aligned}$$

- 1 Bob goes to machine, authenticates himself (important!)
- 2 Bob sends request (read, foo.pdf, M)
- 3 Machine checks that M is a proof of $\langle \text{admin}, [2/12/2008] \rangle \text{read}(\text{Bob}, \text{foo.pdf})$
- 4 Machine allows access or denies it

Basic Architecture

- Implemented as a **shadow file system** over ext2 with Fuse
- Mounted file system has two modules:
 - **Kernel module** traps *all* file system calls; reroutes to user space module (fixed API)
 - **User space module** handles file system calls:
 - 1 Checks access (proof checking)
 - 2 Passes call to underlying ext2 file system
- User space module runs as **root** to bypass ext2 permission checks
- Programmer API remains unchanged; all POSIX calls work unchanged (ideally)

Basic Architecture



Passing Proofs to FS calls

- No possibility of dialog; we want to retain the system call API
- Bob wants to read `foo.pdf`
- `proof_file` contains proofs of access to `foo.pdf`
- Encode proof file in the file name
- Syntax: `@:foo.pdf@proof_file:@foo.pdf`
- Multiple components:
`/usr0/ @:dg@pf1:@dg/ @:foo.pdf@pf2:@foo.pdf`
- Work with all calls, including system calls (open, read, write, ...) and shell commands
 - `@` and `:` are legitimate characters in file names, but do not usually occur in them

- `/usr0/ @:dg@pf1:@ dg/ @:foo.pdf@pf2:@foo.pdf`

- Sample file pf1:

```
%exec "/usr0/dg" [List of certificates] <Proof>
```

- Sample file pf2:

```
%read "/usr0/dg/foo.pdf" [List of certificates] <Proof>
```

```
%write "/usr0/dg/foo.pdf" [List of certificates] <Proof>
```

What do I prove?!

- Proposition to be proved is uniquely determined by four things:
 - Caller's process id
 - File/directory accessed
 - Permission requested (read, write or execute)
 - Current time
- Example: to *read* file `foo.pdf` at time t , proc id 145 must show that:
`<admin, [t, t]> may_read(int2pid(145), str2file(foo.pdf))`
- No nonces!

Demo

Difficulties: Public Key Infrastructure (PKI)

- Why a PKI?
 - Certificates have to be signed and verified!
- Which PKI should we use?
 - PCA is independent of PKI
 - We use GnuPG (open source, no CA)
- How do we relate users to public keys?
 - Currently hardcoded
 - Practically from a database/file
- What uniquely identifies a principal – key or user id?
 - User id in our implementation (to make policies intuitive)

Difficulties: Backwards Compatibility

- Proofs should be optional
 - Existing code should work (albeit slower)
 - Less burden on users
- Proof-checker handles this:
 - Generates missing proof based on `rwxrwxrwx` ext2 permissions
 - Checks the proof
- FS with PCA is *at least* as permissive as underlying ext2

Difficulties: Efficiency

- Proof verification takes time
 - Certificate checking
 - Parsing (seems to be a bottleneck)
- Current performance:
 - Approx 70 creates/deletes per sec. (bonnie++)
 - Visible delay when certificates are checked (approx. 3 per call)
- End goal:
 - No visible delay in common cases
 - FS not designed for massive disk operations (compiling, for instance)
- Possibilities (too many):
 - Caching, port to C, ...

Difficulties: User Defined Predicates

- Bob wants to create a predicate $\text{friendBob}(K)$, meaning that K is a friend of Bob
- How does Bob specify this?
- How does Bob tell the file system how to verify the predicate?
- Can Alice make the statement $\text{friendBob}(\text{Charlie})$?

Difficulties: Managing Certificates and Proofs

- Certificates/proofs contain logical syntax
 - Users can't write logical syntax!
- Need for automated tools to handle common cases for policies and generate proofs
- Policies can be large and complicated
 - Need to analyze policies automatically

Issues: Trusted Computing Base (TCB)

- How do we know that the TCB is bug free?
- The TCB is huge:
 - Proof-checker (700 lines of SML code + perl script)
 - GnuPG
 - Fuse module
 - Ext2 file system

- Conclusions:
 - PCA is useful in a file system
 - PCA at first-order is efficient, expressive, useful
- Time in PCA's logic is useful:
 - Makes system expressive
 - Easy to enforce
- Future work
 - Performance tuning
 - Smoother integration with PKI
 - User tools for proof and policy generation
 - Policy analysis