Logic in Access Control

Martín Abadi
University of California at Santa Cruz
abadi@cs.ucsc.edu

Abstract

Access control is central to security in computer systems. Over the years, there have been many efforts to explain and to improve access control, sometimes with logical ideas and tools. This paper is a partial survey and discussion of the role of logic in access control. It considers logical foundations for access control and their applications, in particular in languages for programming security policies.

1 Introduction

Access control consists in deciding whether the agent that issues a request should be trusted on this request. For example, the agent may be a process running on behalf of a user, and the request may be a command to read a particular file. In this example, the access control machinery would be charged with deciding whether the read should be permitted. This authorization decision may, in the simplest case, rely on consulting an access control matrix that would map the user's name and the file name to a set of allowed operations [23]. The matrix may be implemented in terms of access control lists (ACLs), attached to objects, or in terms of capabilities, held by principals. Typically, however, the authorization decision is considerably more complex. It may depend, for example, on the user's membership in a group, and on a digitally signed credential that certifies this membership.

Access control is central to security and it is pervasive in computer systems. It appears (with peculiar features and flaws) in many applications, virtual machines, operating systems, and firewalls. Physical protection for facilities and for hardware components are other forms of access control.

Although access control may sometimes seem conceptually straightforward, it is both complex and error-prone in practice. The many mechanisms for access control are often broken or circumvented.

Over the years, there have been many efforts to explain and to improve access control. Some of those efforts have relied on logical ideas and tools. One may hope that logic would provide a simple, solid, and general foundation for access control, as well as methods for designing, implementing, and validating particular access control mechanisms. In fact, although logic is not a panacea, its applications in access control have been substantial and beneficial.

This short paper is a partial, informal survey and discussion of the role of logic in access control. It considers logical formulations of access control and their applications, emphasizing recent languages for programming security policies in distributed systems. It does not however aim to be a complete overview. It deliberately neglects several relevant topics that have been the subjects of significant bodies of work. These include:

- Decidability results for problems related to access control (e.g., [17, 26, 28]).
- Logical approaches for authorizing code execution, such as those based on proof-carrying code (e.g., [29]).
- Formal verification of security properties (e.g., [32]).

The next section introduces some logical constructs that have been employed in connection with access control. The following section then describes languages for access control, focusing on the Binder language. It is partly based on a recent note [2], which also explores an analogy with languages for data integration such as the Mediator Specification Language of the Tsimmis system [12, 30].

2. Logics

From matrices to logics

An access control matrix may be viewed as a description of a ternary relation, `may-access`. With this interpretation, `may-access(p,o,r)` would hold whenever the matrix gives principal `p` the right `r` on object `o`. Thus, we may obtain a first logic of access control by representing a global access control matrix with the predicate symbol `may-access`, in the setting of a classical predicate calculus. This trivial logic seems of limited direct benefit. It enables us to state facts such as

```
may-access(Alice,Foo.txt,Rd)
```

which says that principal `Alice` can perform the operation read (`Rd`) on object `Foo.txt`, and rules such as

```
may-access(p,o,Wr)
  =>
may-access(p,o,Rd)
```

which makes the `Wr` right stronger than the `Rd` right, but perhaps not much else. However, this trivial logic suggests more elaborate systems with predicate symbols such as `may-access`.

We may reasonably suspect that there is nothing canonical about `may-access`. We may also worry about a proliferation of variants; for instance, we may imagine a predicate symbol `may-jointly-access`, with the idea that `may-jointly-access(p,q,o,r)` holds if `p` and `q` have right `r` on `o` when they request `r` jointly. In addition, we may imagine many useful auxiliary predicates, such as one for expressing that a principal owns an object, and several for grouping principals and objects. We would perhaps be reluctant to develop a logic with the many built-in constructs and axioms that would result.

Nevertheless, rich logics with constructs similar to `may-access` can support a wide range of current models for access control [6, 19, 20]. In addition to primitives for authorization that greatly generalize `may-access`, the logics may include primitives pertaining to groups, roles, object containment, privilege ordering, and perhaps others. In a different direction, such a logic for access control may include a modal operator for reasoning about necessity [9]. There is much room for logical creativity, for better or for worse—richer logics are not automatically more tasteful or more useful.

Logics for distributed systems

Several characteristics of distributed systems complicate access control (e.g., [13, 22]). These include size, heterogeneity, the autonomy of system components, and the possibility of component failures. These complications have resulted in a substantial line of work. They also suggest the possibility of an important role for logical methods.

In what follows we focus on one of the logical constructs that has often been used in this context, `says` [3].

The formula `p says s` represents that principal `p` makes statement `s`. This statement may simply be a request for an operation. In more interesting cases, the statement may express that `p` delegates some of its authority to another principal `q`, or it may express part of a security policy. For instance, we may write

```
p says may-access(q,o,r)
```

and the rule

```
p says may-access(q,o,r)
  =>
(may-access(p,o,r)
   =>
 may-access(q,o,r))
```

which states that `p` may hand off a right `r` to `q`.

Attractively, `says` abstracts from the details of authentication. When `p says s`, `p` may transmit `s` in a variety of ways:

- on a local channel via a trusted operating system within a computer,
- on a physically secure channel between two machines,
- on a channel secured with shared-key cryptography, or
- in a certificate with a public-key digital signature.

We may assert that `p says s` even when `p` does not directly produce `s`. For example, when `p` is a user and one of its programs sends `s` in a message, we may find it convenient and reasonably accurate to state that `p says s` although `p` itself may never have even seen `s`. In this case, `p says s` means that `p` has caused `s` to be said, or that `s` has been said on `p`'s behalf, or that `p` supports `s`.

If `p says s` and `p` speaks for another principal `q`, then `q says s`. The relation "speaks for" serves to form chains of responsibility in many important situations. A program may speak for a user, much like a key may speak for its owner, much like a channel may speak for its remote end-point. Therefore, some logics include "speaks for" as a primitive.

Existing logics differ in sometimes subtle but important ways in their interpretation of `says` and related constructs. They also vary in their axioms. Sometimes `says` requires no special axioms and is treated quite syntactically, like the `cert` construct of Halpern and van der Meyden [15]. Sometimes `says` has axioms familiar from modal logics, such as:

```
p says (s => s')
  =>
(p says s) => (p says s')
```

and the usual necessitation rule according to which the validity of `s` implies the validity of `p says s`. Sometimes `says` has additional properties. For instance, early on, Lampson suggested the axiom:

```
s => (p says s)
```

Appel and Felten essentially adopt this axiom (as rule name_i) in their work on proof-carrying authentication [4].

It is stronger than the usual necessitation rule, and should be used with caution (if at all). In a classical-logic context, it can yield unexpected consequences such as:

```
(p says s) => (s \/ (p says s'))
```

for every `s'`.

One may imagine that semantics would shed some light on the proper choice of axioms. While semantics have indeed been helpful, so far they provide only limited new insight into notions such as authority and responsibility.

The logics sketched above have had several applications. In particular, they have been used in an operating system [34], in an account of access control in Java Virtual Machines [33], and in an access control system for the Web [5]. They have also influenced some of the languages that are the subject of the next section.

3. Languages

From logics to languages

Languages for access control aim to support the expression and the enforcement of policies (e.g., [7, 8, 10, 11, 21, 25, 27, 31, 35, 36]). The languages are general and flexible enough for programming a wide range of policies—for example, in file systems and for digital rights management.

Many of these languages are targeted at distributed systems in which cryptography figures prominently. They serve for expressing the assertions contained in cryptographic credentials, such as the association of a principal with a public key, the membership of a principal in a group, or the right of a principal to perform a certain operation at a specified time. They also serve for combining credentials from many sources with policies, and thus for making authorization decisions. More broadly, the languages sometimes aim to support trust management.

Several of the most recent language designs rely on concepts and techniques from logic, specifically from logic programming: Li et al.'s D1LP and RT [25–27], Jim's SD3 [21], and DeTreville's Binder [10]. These are explicitly research projects.

Other languages intended for direct use, such as SDSI, SPKI, and XrML 2.0 [11, 31, 36], include related ideas, though typically with less generality. Some of these have been influenced by logical work, but they have not been designed or presented as logical systems. We may however view them as logics, at least in a rudimentary sense. They define systems of notations for describing principals, their statements, authorizations, and sometimes more. The notations come with rules for combining facts and deriving their consequences—for instance, rules for chaining certificates in public-key infrastructures.

One might question whether the deployment of these sophisticated languages would reduce the number of ways in which access control can be broken or circumvented. Policies in these languages might be difficult to write and to understand—but perhaps no worse than policies embodied in Perl scripts and configuration files. There seems to be no hard data on this topic.

A look at Binder

Binder is a good representative for this line of work. It shares many of the goals of other languages and several of their features. It has a clean design, based directly on that of logic-programming languages.

Basically, a Binder program is a set of Prolog-style logical rules. Unlike Prolog, Binder does not allow function symbols; in this respect, Binder is close to the Prolog fragment Datalog. Also unlike Prolog, Binder has a notion of context and a distinguished operator `says`. For instance, in Binder we can write:

```
may-access(p,o,Rd) :- good(p)
may-access(p,o,Rd) :-
     Bob says may-access(p,o,Rd)
```

These clauses can be read as expressing that any principal `p` may access any object `o` in read mode (`Rd`) if `p` is good or if `Bob` says that `p` may do so.

Here only `:-` and `says` have built-in meanings. The other constructs (even `may-access`) have to be defined or axiomatized. As in Prolog, `:-` stands for reverse implication ("if"). For instance,

```
may-access(Alice,Foo.txt,Rd)
```

would follow from these clauses and from

```
Bob says may-access(Alice,Foo.txt,Rd)
```

As in previous logical treatments of access control, `says` serves for representing the statements of principals and their consequences. Thus,

```
Bob says may-access(Alice,Foo.txt,Rd)
```

holds if there is a statement from `Bob` that contains a representation of the formula

```
may-access(Alice,Foo.txt,Rd)
```

or it can be derived if there is a statement from `Bob` that contains a representation of the formula

```
may-access(Alice,Foo.txt,Wr)
```

and another one that contains a representation of the clause

```
may-access(p,o,Rd) :-
     may-access(p,o,Wr)
```

Each formula is relative to a context (a source of statements). In our example, `Bob` is a context. Another context is implicit: the local context in which the formulas apply. For example,

```
may-access(p,o,Rd) :-
    Bob says may-access(p,o,Rd)
```

is to be interpreted in the implicit local context, and `Bob` is the name for another context from which the local context may import statements.

In addition to logic-programming rules, Binder includes a special proof rule for importing clauses `a :- a1, ..., an` from one context into another. The rule applies only to clauses where the atom `a` in the head is not of the form `q says s`. When importing a clause from context `p`, the rule replaces `a` with `p says a`, and replaces `ai` with `p says ai` if `ai` is not of the form `q says s`, for `i` = `1`..`n`. For example, when `Charlie` exports the clauses:

```
may-access(p,o,Rd) :- good(p)
may-access(p,o,Rd) :-
    Bob says may-access(p,o,Rd)
```

the local context obtains:

```
Charlie says may-access(p,o,Rd) :-
    Charlie says good(p)
Charlie says may-access(p,o,Rd) :-
    Bob says may-access(p,o,Rd)
```

This proof rule is complicated enough to call for some logical analysis. It can be partly justified by standard modal logic, in particular via the theorem

```
p says s /\ p says s'
  =>
p says (s /\ s')
```

and the axiom

```
p says (s => s')
  =>
(p says s) => (p says s')
```

Here, `p` represents the context that exports a clause (that is, `Charlie`, in our simple example). However, more is needed, even for our example. The proof rule can be derived once we add the strong axiom

```
s => (p says s)
```

Fortunately, a restricted form of this axiom suffices:

```
(q says s) => (p says q says s)
```

Binder does not assume or require that predicate symbols mean the same in every context. For example, `Bob` might not even know about `may-access`, and might assert `peut-lire(Alice,Foo.txt)` instead of `may-access(Alice,Foo.txt,Rd)`. In that situation, one may translate explicitly with the clause:

```
may-access(p,o,Rd) :-
    Bob says peut-lire(p,o)
```

On the other hand, Binder does not provide much built-in support for local name spaces. A closer look reveals that the names of contexts have global meanings. In particular, when `Charlie` exports:

```
may-access(p,o,Rd) :-
    Bob says peut-lire(p,o)
```

the local context obtains:

```
Charlie says may-access(p,o,Rd) :-
    Bob says peut-lire(p,o)
```

without any provision for the possibility that `Bob` might not be the same locally and for `Charlie`.

Other systems, such as SDSI and SPKI, support more elaborate naming mechanisms, with corresponding logical explanations and problems (e.g., [1, 14–16, 18, 24]).

Acknowledgments

I am grateful to Andrew Appel, Mike Burrows, William Cook, John DeTreville, Butler Lampson, and Roger Needham for discussions that contributed to this paper.

This work was partly supported by the National Science Foundation under Grants CCR-0204162 and CCR-0208800.

References

[1] Martín Abadi. On SDSI's linked local name spaces. *Journal of Computer Security*, 6(1-2):3–21, 1998.

[2] Martín Abadi. On access control, data integration, and their languages. In Andrew Herbert and Karen Sprck Jones, editors, *Computer systems: Papers for Roger Needham*. 2003. The volume, with minor updates, may be published by Springer-Verlag.

[3] Martín Abadi, Michael Burrows, Butler Lampson, and Gordon Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, October 1993.

[4] Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In *Proceedings of the 5th ACM Conference on Computer and Communications Security*, pages 52–62, November 1999.

[5] Lujo Bauer, Michael A. Schneider, and Edward W. Felten. A general and flexible access-control system for the Web. In *Proceedings of the 11th USENIX Security Symposium 2002*, pages 93–108, 2002.

[6] Elisa Bertino, Barbara Catania, Elena Ferrari, and Paolo Perlasca. A logical framework for reasoning about access control models. *ACM Transactions on Information and System Security*, 6(1):71–127, February 2003.

[7] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D. Keromytis. RFC 2704: The KeyNote trust-management system, version 2. On the Web at `http://www.ietf.cnri.reston.va.us/rfc/rfc2704.txt`, September 1999.

[8] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 164–173, May 1996.

[9] Jason Crampton, George Loizou, and Greg O'Shea. A logic of access control. *The Computer Journal*, 44(2):137–149, 2001.

[10] John DeTreville. Binder, a logic-based security language. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 105–113, May 2002.

[11] Carl Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian Thomas, and Tatu Ylönen. RFC 2693: SPKI certificate theory. On the Web at `http://www.ietf.cnri.reston.va.us/rfc/rfc2693.txt`, September 1999.

[12] Hector Garcia-Molina, Yannis Papakonstantinou, Dallan Quass, Anand Rajaraman, Yehoshua Sagiv, Jeffrey D. Ullman, Vasilis Vassalos, and Jennifer Widom. The TSIMMIS approach to mediation: Data models and languages. *Journal of Intelligent Information Systems*, 8(2):117–132, 1997.

[13] Morrie Gasser, Andy Goldstein, Charlie Kaufman, and Butler Lampson. The Digital Distributed System Security Architecture. In *Proceedings of the 1989 National Computer Security Conference*, pages 305–319, October 1989.

[14] Adam J. Grove and Joseph Y. Halpern. Naming and identity in epistemic logics, I: The propositional case. *Journal of Logic and Computation*, 3(4):345–378, 1993.

[15] Joseph Y. Halpern and Ron van der Meyden. A logic for SDSI's linked local name spaces. *Journal of Computer Security*, 9(1-2):47–74, 2001.

[16] Joseph Y. Halpern and Ron van der Meyden. A logical reconstruction of SPKI. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, pages 59–72, June 2001.

[17] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, August 1976.

[18] Jon Howell and David Kotz. A formal semantics for SPKI. In *Proceedings of the Sixth European Symposium on Research in Computer Security (ESORICS 2000)*, volume 1895 of *Lecture Notes in Computer Science*, pages 140–158. Springer-Verlag, 2000.

[19] Sushil Jajodia, Pierangela Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 31–42, May 1997.

[20] Sushil Jajodia, Pierangela Samarati, V. S. Subrahmanian, and Eliza Bertino. A unified framework for enforcing multiple access control policies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, volume 26(2) of *SIGMOD Record*, pages 474–485, May 1997.

[21] Trevor Jim. SD3: A trust management system with certified evaluation. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 106–115, May 2001.

[22] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, November 1992.

[23] Butler W. Lampson. Protection. In *Proceedings of the 5th Princeton Conference on Information Sciences and Systems*, pages 437–443, 1971.

[24] Ninghui Li. Local names in SPKI/SDSI. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, pages 2–15, July 2000.

[25] Ninghui Li, Benjamin N. Grosof, and Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. *ACM Transactions on Information and System Security*, 6(1):128–171, February 2003.

[26] Ninghui Li and John C. Mitchell. Datalog with constraints: A foundation for trust-management languages. In *Proceedings of the Fifth International*

Symposium on Practical Aspects of Declarative Languages (PADL 2003), volume 2562 of *Lecture Notes in Computer Science*, pages 58–73. Springer-Verlag, January 2003.

[27] Ninghui Li, John C. Mitchell, and William H. Winsborough. Design of a role-based trust-management framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 114–130, May 2002.

[28] Ninghui Li, William H. Winsborough, and John C. Mitchell. Beyond proof-of-compliance: Safety and availability analysis in trust management. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, May 2003. To appear.

[29] George C. Necula. Proof-carrying code. In *Conference record of POPL '97, the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, 1997.

[30] Ioannis G. Papakonstantinou. *Query processing in heterogeneous information systems*. PhD thesis, Stanford University, 1997. Available at `http://www.db.ucsd.edu/people/yannis.htm`.

[31] Ronald L. Rivest and Butler Lampson. SDSI — A Simple Distributed Security Infrastructure. On the Web at `http://theory.lcs.mit.edu/˜cis/sdsi.html`, 1996.

[32] John Rushby. Design and verification of secure systems. *Proceedings of the 8th ACM Symposium on Operating System Principles, ACM Operating Systems Review*, 15(5):12–21, December 1981.

[33] Dan S. Wallach, Andrew W. Appel, and Edward W. Felten. SAFKASI: a security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology*, 9(4):341–378, October 2000.

[34] Edward Wobber, Martín Abadi, Michael Burrows, and Butler Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems*, 12(1):3–32, February 1994.

[35] eXtensible Access Control Markup Language (XACML) version 1.0. OASIS Standard, at `http://www.oasis-open.org/committees/xacml/repository/`, 2003.

[36] eXtensible Rights Markup Language (XrML) version 2.0. At `http://www.xrml.org/`.