Access Control in a World of Software Diversity

Martín Abadi[1], Andrew Birrell[2] and Ted Wobber[2]

[1]*University of California, Santa Cruz*
[2]*Microsoft Research, Silicon Valley*

Abstract

We describe a new design for authentication and access control. In this design, principals embody a flexible notion of authentication. They are compound principals that reflect the identities of the programs that have executed, even those of login programs. These identities are based on a naming tree. Our access control lists are patterns that recognize principals. We show how this design supports a variety of access control scenarios.

1. Introduction

A central concern in securing a computer system is access control: deciding whether to permit a particular form of access to some of the system's resources or data. Classically, the system controls access by using a "reference monitor": a trusted piece of code that is used to make all access decisions [1]. The reference monitor is presented with the identity of a principal making the request, the identity of an object (system resource or data protected by the system), and the specific form of access desired. The reference monitor then makes the access control decision by deciding whether to accept the proffered identity, and by consulting access control information associated with the object. The access control function is a predicate that maps principal, object and operation to a Boolean outcome.

In this paper we consider how to design this access control machinery for a single-host, non-distributed operating system such as Windows or Linux. A direction for our future work will be to extend this design to include distributed systems. For this paper, we ignore issues of compatibility with previous access control machinery.

In the classic design for this purpose, each principal is identified by a small identifier (an SSID in Windows, a user ID in Unix-based systems). The access control data for an operation is an access control list kept with each object, and takes the form of a set whose members are either principals or identifiers for groups. A group, in turn, is a set whose members are either principals or identifiers for further groups. Access is permitted or denied on the basis of the presence of the proffered principal in the closure of the access control list and its constituent groups. (In Windows the group memberships of a principal are actually determined at login time and cached in a token. The semantics are as described above, but the timing is somewhat different: some of the reference monitor's work was done at login time.)

The classic design, unfortunately, has many limitations and drawbacks. These have become increasingly critical in recent years as the diversity of the programs installed in our systems, and of the attacks upon them, have increased. The three drawbacks that we attempt to address in the current design are as follows.

First, the notion that the principal is identified solely with a logged-in user doesn't allow us to express important real-world security situations. The actual user of course isn't really the entity making the access request. The request is being made by a program. The classic design assumes that every program executing in a user's session is acting on the user's behalf and with the user's full trust. That might have been true historically, but it is certainly not true today. For example, the user most likely is happy if Microsoft Word is performing operations on objects that are Microsoft Word documents, but would be unhappy if some ad-ware program was doing so. Similarly, the user might reasonably object if Microsoft Word was spontaneously accessing the user's Quicken database. So we desire that the principal presented to the reference monitor includes some notion of the program that is executing, and also of the program that provoked that execution, and so on back through the execution history.

Second, the classical notion of "logged-in" is inflexible. It is all or nothing, and implies that all mechanisms for authenticating a user are equally trusted. Equivalently, it requires that all authentication mechanisms are part of the trusted computing base. To support a modern execution environment, where principals might arise from a console login, from a remote terminal login, or from the creation of a background service, batch job or daemon, and where authentication might be by password, X.509 certificate, smart card, or by an *ad hoc* decision by an application, we require that these circumstances can be included as part of the identity of the principal presented to the reference monitor. This is a prerequisite to permitting the monitor to base its decisions partly on *how* a principal was authenticated.

Finally, once we have included so much extra information within the idea of principal, it becomes untenable to say that the access control data is just a set of principal identifiers. We must be able to express in the access control data a wide variety of constraints on the acceptable principals, based on the wider variety of information now included in our principal's identities. However, to maintain any real security, the decisions that can be made by this more general mechanism must be expressed in a sufficiently simple language that they make sense to the people who read and write the decisions.

2. Previous Work

Within the confines of a 5-page paper, we can't come close to doing justice to the wide range of proposals that have been made to address some of the problems identified in the introduction.

Many writers (and some writers many times) have proposed authentication schemes that go well beyond the basic notion of logged-in user [2,4,7]. Most commonly, such schemes allow a principal to adopt a "role" with the intention of reducing or enhancing the principal's privileges. Some schemes become quite elaborate, including in the resulting compound principals such details as the principal that signed the certificate proving the identity of an executing program. Such designs provide great power, but with a lot of complexity.

Current Java security mechanisms [5] take some account of program execution history by using stack inspection when making access decisions.

Several systems, including current versions of Windows and most current Unix-based systems, support extensibility in their authentication mechanisms. There are a few specialized systems that have made their access control decisions dependent on how the principal was authenticated [3], but this hasn't made its way into the access control machinery of general-purpose operating systems. We believe that this information can be included and used without adding undue complexity to the design.

Other designs that involve compound principals have also resulted in revisions to the design of access control lists, in order to accommodate the new principals, though in somewhat different ways than the present design. For example, the Taos work included access control lists that expressed some logic about which principals match [6].

3. Our Design

Our design is intended to address the deficiencies described above. Specifically, we want to consider in our access control decisions the identity of the authenticated user, the identity of the agency that performed the authentication, and the identity of the program invocations that have brought the computation to its current point. Then we want to have access control lists that allow us to express succinctly and intelligibly a wide variety of commonly useful access control decisions.

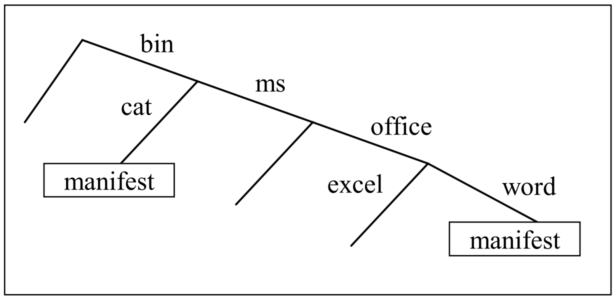
As will be seen below, the key aspects of our design are: a separation of principal names from the policies and mechanisms that led us to trust those names (the "naming tree"); compound principals formed by two operators that represent authentication and program invocation; and an expressive but straightforward access control list mechanism.

3.1. The Naming Tree

The naming tree is a singly rooted tree in which each arc is labeled with a simple string. Some of the nodes in the tree have attached to them a data structure called a "manifest". A manifest specifies a particular executable "program", by providing the file names and cryptographically secure fingerprints of the constituent parts of the program — its executable, shared libraries, data resources, and so forth. Since we want the identity of an invoked program to be part of a principal name, program invocation is a security-related operation, and we require that programs are named by paths through the naming tree.

The naming tree is also used to name users, and to name groups from within access control lists.

Our use of this naming tree lets us separate the mechanisms and policy for constructing the tree from the mechanisms and policy for running a reference monitor. Both are important parts of the overall security decision, but the separation greatly simplifies the authentication and access control mechanisms.



We expect quite familiar mechanisms would be used to construct the tree, though we give no details here. For example, the decision to install a program purporting to be Microsoft Word would likely require a trusted party (such as an authenticated administrator) to inspect certificates (such as X.509) and agree that the proffered bits really deserve to be given such a trusted name. Once that decision has been made, the presence of the resulting manifest at the node named, e.g., “bin/ms/office/word”, makes the administrator’s decision clear, and we can use this in subsequent authentication and access control decisions.

Most likely, the naming tree would have its own access control lists attached to it, to specify which principals can modify which parts of the tree.

3.2. Principal Names

A principal name is a string constructed from arcs in the naming tree and operators “/”, “@”, and “+” according to the following grammar.

• Manifest Name:	MN = Arc | MN “/” Arc
• Relative Role:	RR = Arc | RR “/” Arc
• Role:	R = MN | R “@” RR
• Principal:	P = R | P “+” R

The system provides exactly two operations that affect principals:

- InvokeProcess(MN)
- ForkRole(RR)

“InvokeProcess” runs a program. Its argument “MN” is a manifest name, which is a path from the root of the naming tree to the manifest of the desired program. The system finds the named manifest, loads the appropriate data into a new security context (process, say), and initiates its execution. When the principal that calls InvokeProcess is “P”, then the new security context runs as principal “P+MN”.

In other words, occurrences of the “+” operator within a principal name represent the history of program invocations that resulted in the currently executing program.

There is one variation of InvokeProcess. A manifest might have been marked as a “service”, in which case the new security context runs as the principal “MN”, independently from its invoker.

“ForkRole” runs the same program as calls it, but in a new security context. Its argument “RR” is a path relative to the program’s manifest name. When the principal that calls ForkRole is “Q”, then the new security context runs as principal “Q@RR”.

In other words, occurrences of the “@” operator within a principal name indicate where a program has decided to adopt a distinguished role, relative to its manifest. This indication says nothing about whether the role is more or less privileged — that has meaning only to the extent that access control lists grant more or less access to the new principal name.

One critical use of ForkRole is to indicate when a program makes an authentication decision. For example, the system might run a console login program by invoking the manifest “bin/login” as a service, thus executing as principal “bin/login”. When the console login program has received a satisfactory user name “andrew" and password from the console, it will use ForkRole to start running itself as “bin/login @ andrew”, then use InvokeProcess to run Andrew’s initial command shell “bin/bash”, which will then be executing as the principal “bin/login @ andrew + bin/bash”.

Similarly, we might run the manifest “bin/sshd” to listen for incoming SSH connections. After satisfactory authentication through the normal SSH public-key mechanisms it might adopt the role “bin/sshd @ andrew” then run the command shell, which would execute as “bin/sshd @ andrew + bin/bash”.

In these two scenarios, if Bash decides to run “cat” (whose manifest is named “bin/cat”) and cat tries to open a file, we would have an access request to the file system from either the principal “bin/login @ andrew + bin/bash + bin/cat” or the principal “bin/sshd @ andrew + bin/bash + bin/cat” respectively. The reference monitor for the file system would then consult the access control list on the requested file to decide whether the given principal should be granted access.

Nowhere in these scenarios has the system trusted any of the programs involved: login, sshd, bash, or cat. All the system did was to certify the program invocations involved, and that bin/login and bin/sshd chose to adopt the role “andrew”. In this design trust occurs only in constructing the naming tree (trusting that the programs really deserve their given names) and as a result of the way in which we write access control lists (which embody our access control decisions).

3.3. Access Control Lists

With complex principal names such as those we propose above, having an access control list ("ACL") be merely a list (or set) of principal names does not give us nearly enough convenience and expressive power. For example, we might want to permit access to a user while executing some of a particular set of programs, or when authenticated by some particular set of programs (e.g., bin/login or bin/sshd, but not bin/ftpd); or we might want to give access to a program regardless of its user. While we could perhaps list all allowed principals, that would be awkward at best. Instead we use patterns that recognize principal names.

The exact pattern recognition language that we use is not critical to this idea, although the choice of language will certainly have an impact on the usability of the design, and therefore on the security of the resulting systems. We present here a recognizer that is a specialized subset of regular expressions. Obviously, more or less complex recognizers are possible, allowing the expression of more or less complex access control decisions.

An ACL is a string constructed from arcs in the naming tree and operators, as follows:

- Atom = Arc | "/" | "@" | "+"
- Item = Atom | "." | "(" ACL ")" | Item "*" | "{" GroupName "}"
- GroupName = Arc | GroupName "/" Arc
- Seq = Item | Seq Item
- ACL = Seq | ACL "|" Seq

The matching rules are similar to those for conventional regular expressions:

- Any Atom matches itself
- "." matches any single Arc (explicitly excluding "/", "@", and "+")
- "(ACL)" matches ACL
- "Item *" matches zero or more sequential occurrences of Item (greedily)
- "{ GroupName }" matches whatever is matched by the contents of the node GroupName in the naming tree
- "Seq Item" matches Seq followed immediately by Item
- "ACL | Seq" matches either ACL or Seq

A principal "P" matches an ACL "A" iff the string P matches the regular expression that is the contents of A. The match must be complete — all of P, not just a substring of it.

"GroupName" provides a mechanism for sharing parts of the recognition machinery amongst multiple ACL's. We place groups within the same naming tree as manifests and role names, with the same assumption that their presence there reflects a trust decision made by a suitable administrator. Recursively defined groups are not permitted.

A reference monitor will grant P its requested access to an object iff P matches the relevant ACL. In doing so, the reference monitor is just performing string manipulation on the principal name and the ACL contents — it doesn't need to use the naming tree itself, except to read referenced groups. (We do not consider here details of controlling access modes, such as "read" and "write"; the reference monitor will of course only grant P the appropriate mode.)

4. Usage Examples

To keep the examples legible, assume that the naming tree contains the following groups, whose names, for simplicity, are single arcs in the tree:

- path = . (/ .) *
- role = (@ { path }) *
- app = { path } { role }
- trusted = bin / (login | sshd)
- authors = Andrew | Martìn | Ted
- office = bin / ms / office / . /
- good = bin / cmd.exe | { office }

The following ACL is similar to the baseline semantics of existing systems, that is, give access to an explicitly named user, if authenticated by a trusted program:

- {trusted} @ Ted (+ {app}) *

More precisely, the above ACL permits access from any program invoked (directly or indirectly) from one of our trusted authentication programs, provided that the authentication program has adopted the role "Ted". In contrast with existing systems, however, the choice of which authentication programs should be trusted is made in the ACL. We could trust different sets of authentication programs for different objects or for different users.

Our next example is similarly simple, but not at all like traditional access control: it gives access from any of a specific set of programs, regardless of the user who invoked them:

- ({app} +) * {office} {role}

One might use such an ACL, for example, to allow Microsoft Office applications to access some auxiliary files, regardless of who is running the applications, while preventing users doing anything else with the auxiliary files.

Our final example gives access for users when authenticated by a trusted program as a member of "authors", but only when the entire invocation chain of programs is acceptable:

- {trusted} @ {authors} (+ {good}) *

5. Conclusion

Our design has several important aspects that work well together. First, the naming tree lets us separate the policy and mechanisms for certifying programs and groups from the day-to-day authentication and access control mechanisms. Second, we provide just two operators for composing principals, providing sufficient expressiveness while retaining simplicity. Third, we use these principals to avoid requiring that the system trust particular authentication programs. Finally, we generalize ACL's to be pattern recognizers, thereby allowing compact expression of sophisticated access control decisions that make full use of the expressiveness of our principals.

We believe that this design allows for authentication and access control in a modern operating system, suitable for the more stringent requirements of a modern security posture in a world with diverse software.

6. Acknowledgements

This work was done in the context of the Singularity research project in Microsoft Research, run by Galen Hunt. In particular, Ulfar Erlingsson and Dan Simon made significant contributions to our many discussions about this design.

References

1. Anderson, "Computer Security Technology Planning Study Volume II", ESD-TR-73-51, , Air Force Systems Command,Oct. 1972.
2. Badger et al. "A Domain and Type Enforcement UNIX Prototype". USENIX Comp. Sys., 9(1):47--83, Winter 1996
3. Fried & Lowry "BigDog: Hierarchical Authentication, Session Control, and Authorization for the Web". USENIX Second Workshop on Electronic Commerce, Nov. 1996.
4. Gasser et al, "The Digital Distributed System Security Architecture", Proc. 1989 National Computer Security Conf., (1989), pp. 305-319
5. Gong et al, "Inside Java 2 Platform Security, Second Edition". Addison-Wesley (May 2003).
6. Lampson et al. "Authentication in Distributed Systems: Theory and Practice". Trans. Comp. Sys., 10(4):265-310, Nov. 1992
7. Wobber et al. Authentication in the Taos Operating System. Trans. Comp. Sys., 12(1):3-32, Feb. 1994.