RZ 3126 (# 93172) 05/03/99
Computer Science/Mathematics 11 pages

Research Report

PLAS – Policy Language for Authorizations

J. L. Abad-Peiro, H. Debar, T. Schweinberger, and P. Trommler

IBM Research Division
Zurich Research Laboratory
8803 Rüschlikon
Switzerland
{jla,deb,ths}@zurich.ibm.com

LIMITED DISTRIBUTION NOTICE

This report will be distributed outside of IBM up to one year after the IBM publication date.

IBM Research Division
Almaden · Austin · Beijing · Haifa · T.J. Watson · Tokyo · Zurich

PLAS – Policy Language for Authorizations

J. L. Abad-Peiro, H. Debar, T. Schweinberger, and P. Trommler

IBM Research Division, Zurich Research Laboratory, 8803 Rüschlikon, Switzerland
{jla,deb,ths}@zurich.ibm.com

Abstract

A key issue in authorization services and computer security in general is the definition of security policies [1]. To help define security policies we have developed a new policy language for authorization systems (PLAS), and a framework in which to apply it. This paper describes the PLAS framework and shows how can it be used within current fields of research in IT security such as protection against downloadable code and in intrusion-detection systems.

[1] Jean E. Smith and Fred W. Weingarten, editors. *Research Challenges for the Next Generation Internet.* Computing Research Association, May 1997. Report from the Workshop on Research Directions for the Next Generation Internet.

1 Introduction

We have developed a framework to define, evaluate, and enforce access-control policies for intrusion-detection systems, access control, and systems managing downloaded programs. These are three scenarios of ongoing research in the field of authorization service security [SW97, SS96, RTG94]. The definition of policies for our framework is based on a language called PLAS, which stands for policy language for authorization systems. This paper describes the PLAS framework; a description of the PLAS grammar can be found in Appendix A.

The PLAS framework can be used within different authorization scenarios, but it has been designed primarily for systems managing downloaded code, intrusion-detection systems, or to define pure access-control mechanisms. The usage of the framework will determine the integration of the AC_{en} module within a broader-scoped architecture, e.g., at the operating-system level when used for access-control and intrusion-detection purposes, or within the Java Security manager [KLO97] when used for downloaded Java code. Section 3 will describe these scenarios with more detail. In Section 4 we will identify the requirements of these scenarios and establish how PLAS can be used to fulfill those requirements. Section 5 describes the PLAS framework and its components, and Section 6 provides an example of the use of PLAS. We conclude the paper in Section 7 and indicate directions for future research.

The PLAS framework evaluates whether an entity requesting an action is allowed to execute the requested action in a particular context, and authorizes the execution of the action. All elements used for authorizations, i.e., entities, actions, contexts, and facts must be defined following the PLAS syntax before the policies can be evaluated. We describe below these elements within our framework. The idea of using contextual information to grant access permission is not new [BM82]; other projects have based their authorization rules on contextual information, for example [NHMT98].

Our framework provides a single policy-based environment in which system administrators can specify organizational security policies, policies to be respected by downloaded code such as applets, and policies to manage alarm-triggering events for intrusion-detection systems. All systems work with synergy sharing a common set of defined security policies.

The PLAS language is meant to be easy to use yet complete in the scope of policy definition. We have not yet developed, though it is planned, a module to verify inconsistencies of policies and to simulate functioning in a non-operational environment.

2 PLAS Authorization System

In general, we define the terms *authorization* and *access control*, AC, as the legitimate and proper use of resources in a system or environment. The *security policy* of a system defines which security mechanisms are to be provided to users in that system, as well as a suite of conditions to be respected in order to fulfill properly secure operations taken over the system. The definition of security policies is made in a *security model*. In this sense, security models provide the semantic framework in which security requirements can be formally described [Jon78].

The *access-control policy* is the part of a security policy that governs who may access resources and information stored in the system, and how. There are three types of authorization services that implement access-control policies: *definition*, *evaluation*, and *enforcement* services.

There is a wide range of definitions of authorization services. Their complexity ranges from a simple matrix filled in with fixed access rights [HRU76] to a fully developed language for authorization policies [JSSB97, DoCM97, SZ96, BFL96].

Mechanisms to evaluate and enforce access control in operating systems and networks are currently inspired by the ISO/IEC standard [OSI94]. These security services are placed in different layers within the OSI architecture model depending on the type of service and application [OSI84, OSI91]. Under OSI's assumptions, an *initiator* (entity or object requesting an access) must go through an AC-enforcing service before reaching a *target* (entity or object whose service is requested). The AC decision service evaluates the legitimacy of the initiator request for access. The AC-enforcing service allows or denies the connection between initiator and target according to the AC decision function.

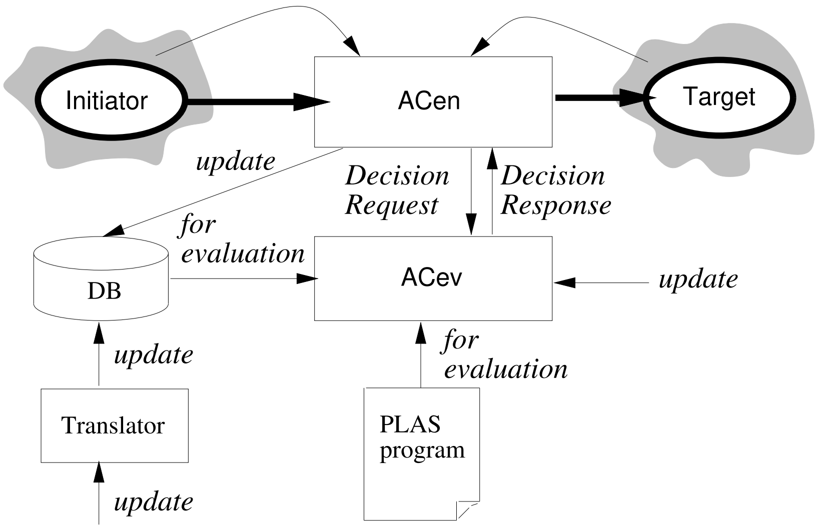

Figure 1: Architecture of an authorization system using PLAS.

The PLAS architecture is based on the OSI model previously described. Hence, we also divide AC services into definition, enforcement, and evaluation. Figure 1 shows the evaluation module, AC_{ev}, and the enforcement module, AC_{en}. The service for the definition of policies is defined by the syntax and semantics of PLAS programs.

3 Scenarios in which PLAS is useful

In this section we will describe scenarios in which the PLAS framework can be deployed. We will address traditional access control like that found in many operating systems, downloaded code such as Java applets, and intrusion-detection systems.

3.1 Access control

There are many access-control protocols, systems, and mechanisms. The differences among them range from *what* kind of access they control, *where* the entities involved are located, and *how* this control is performed (definition, evaluation, and enforcement). Focusing on the *how* part, we can designate several models of access control with which to classify systems and mechanisms, e.g., discretionary access control [CFMS95], mandatory access control [McL94], and role-based access control [SCFY96].

Using PLAS policies we can write evaluation programs that allow the system to perform discretionary, mandatory, and role-based evaluations of access control privileges, e.g., at the operating-system level.

3.2 Downloaded code

One of the main differences between access authorizations in systems that allow one to download remote applications and systems that do not, resides in the trust that users invest in the parties involved. If downloaded applications are not allowed (very rare nowadays), the operating system and the applications installed are normally trusted and must be protected from the (potentially untrusted) rest of the users. If downloaded applications exist, the user trusts its operating system more than the downloaded application itself, and even less so the provider of the application. These differences in the trust relationships have major implications on the way in which authorizations must be defined. With PLAS, the user of the system or the administrator can define an authorization policy to be satisfied by each principal providing downloaded code.

For example, Internet-based applications (beyond animated web pages) might require access to sensitive data. In this case, users want to ensure privacy and integrity of their data against a third party. Hence, users are interested in restricting access to data as much as possible, and in such a way that the set of actions necessary to carry out the tasks remains minimal. The reasoning behind this is that if minimal access to the system is granted, the risk of information misuse can be reduced.

Normally, the goal of a user is to execute an untrusted application safely, rather than to decide what level of access to grant the principal that signed the code. Thus, it seems natural to focus on the application requirements when defining what the code will be authorized to do. A security model that takes these considerations into account is the *application profile model* (APM) described in [Tro98]. The security policy of the APM stipulates that each principal provide a set of applications in which the user is interested. Each of these applications must access a certain set of resources called the *application profile*. However, the focus of the application profile model on application requirements can lead to a violation of a global security policy. If this policy is also defined in terms of PLAS it can be enforced by the same security mechanism that guarantees some sort of baseline security.

PLAS is useful for defining an application profile where the application has access to sensitive information but has to access the network during certain stages of the execution. PLAS allows us to specify these requirements in the form of contexts.

3.3 Intrusion-Detection Systems

PLAS policies and *facts*[1] are used as input for a "translator", which produces a set of intrusion-detection signatures corresponding to different attacks [Kum95]. These signatures are evidence of the existence of valid events occurring on the system, e.g., actions that have been authorized by the policy, or forbidden events, i.e., when the system is misused or attacked. The signatures express these constraints in a way that is directly matched to the audit trail, so that for each audit a decision is taken either to accept the action, for example generating an alarm, or to reject it.

[1]PLAS facts express the knowledge that PLAS programs manipulate with policies. Their definition is part of the language syntax.

The translator takes quadruples into account (subject, object, action, context). The *subject* of the event is either a user, translated into a set of uids, or a process, translated into a full-path program name. The *action* is translated into a group of system calls that must be performed. The *object* is a file in the broad sense of the term (e.g., ordinary files, directories, devices, etc.) and is translated into a full-path name and inode-related information.

The context notion is more complex to translate. In our framework the simplest type of contextual information to translate is time, which is directly available to PLAS from a directly attached audit system. More complex types of contextual information are created from dependencies between past and future events. In this case, each individual event in the chain is a signature, but the result of this signature is to activate another signature, not to generate directly an accept or deny alarm. Therefore, we maintain chains of active signatures, which are in turn monitored to detect whether the context information is being breached.

4 Requirements for PLAS programs

The case scenarios from the previous section have different requirements to be specified with a policy language. We will describe how these requirements are mapped onto the features provided by the PLAS programming language.

4.1 Downloaded code

Application profiles as well as principals that sign applications are considered PLAS *entities*. Depending on the underlying execution platform, *actions* will either be method invocations, including parameters passed, or commands of a scripting language together with the arguments.

In some systems (e.g., Java-based platforms) the security policy is not enforced at the level of the system API but rather reduced to a small set of low-level implementation methods. The advantage of this approach is that the security functionality can be very small and hence easier to be verified for correctness. On the other hand, it would be interesting to know which high-level method has invoked a low-level method in order to provide an access-control evaluation for API methods, e.g., having a method to append a message to a log file. The message must have a time stamp and be formatted in a certain way. The system offers an *append* method to attach a message to the log file that uses a *write* method to actually write the data to the file. To enforce a policy such that the application can append only a properly formed message to the file, we must not allow the application to call *write* directly but only through the *append* method. This behavior can be achieved in the PLAS framework by deploying the concept of a *context*. The context describes the calling hierarchy (stack trace) that is matched with the actual calling hierarchy. We call this a *local context* as opposed to the concept of *global context* that refers to the access history of the application.

4.2 Intrusion-Detection Systems

The use of PLAS in an intrusion-detection context is to derive information about the expected behavior of the information system, and exert *"a posteriori"* control over whether the actions that have been taken are correct with regard to the security policy defined by the administrator of the system. The security policy in itself is a difficult component to introduce inside an intrusion-detection system because it is often formulated with broad principles which cannot easily be translated into intrusion-detection signatures. Therefore, we limit ourselves to translating an authorization policy, which is more limited in scope and more formally defined. In order to translate the access-control policy into intrusion-detection signatures, PLAS must express its policy using elements found in an information system audit trail. This information includes but is not limited to user identities, programs, processes, files, actions, and dates.

PLAS programs provide the required level of definition of authorization policies for our framework.

5 PLAS programs

A PLAS program is composed of a list of definitions of entities, actions, and contexts as characterized in Section 2, plus a list of facts that represent events that have taken place in the system, and a list of authorization policies based on all previous elements.

Any of these elements composing a PLAS program can be read from a database file, e.g., containing the facts to let programs be cleaner for their design. However, the construction of security policies using PLAS relies heavily on the consistent definition of these elements across the system in which they exist.

The importance of a global PLAS element definition is extremely noticeable in the IDS case scenario. In this case information may come from very different sources, e.g., the audit system. Hence synchronization of terminology is critical for the correct evaluation of authorization rights. In the downloaded-code scenario the element definition is self-contained. But in either case we need a module translator mapping internal PLAS definitions to external ones which depend on the scenario itself.

Every PLAS program is reduced to a *main policy* whose evaluation determines the output of the AC evaluation module. Policies defined inside the policy body of a PLAS program are bound together as subpolicies of the *main policy* by logical operators such as *and*, *or*, *not*, etc., depending on the PLAS program itself.

The best way to understand how a PLAS program is defined is with an example. In the next section we illustrate a simple one.

6 Example of PLAS programs

To demonstrate the use of PLAS to define a security policy we use the following informally stated requirement:

> "Nobody can open a network connection after having read a non-public file."

```
modul DownloadedCode_IO;

// Predefined system conditions with customized evaluation realizations.
// For example, the authenticated condition evaluates to TRUE if the
// referred entity can be authenticated whereas the actual mechanism
// might well depend on the entity type,
// e.g. certificate validation or user interaction.
import condition  R/- system  "syscond"  "system.db";

action open_connection {
  Name    "socket";         // socket() system call or, e.g. in Java
                            // creation of java.net.Socket instances
  Group   "applet, aglet";  // or system group IDs
}

action open_file {
  Name    "open, fopen";
  Info    "open file system calls, etc...";
}

context pub_dir {
  Name    "/pub";
}

context remote_ctx {
  Name    "www.semper.org";
  Type    "host";
}

export policy main {
  everybody can do open_file    in pub_dir    if system.authenticated  or
  everybody can do open_file    in remote_ctx if system.authenticated  or
  nobody can do open_connection               if file_opened           or
  nobody can do open_file                     if net_connected
}

condition net_connected {
  the entity has done open_connection
}

condition file_opened {
  the entity has done open_file
}
```

Other examples of PLAS programs and scenarios are available from the authors upon request. Among them we have a "sudo"-like PLAS program that, when properly used with a UNIX operating system, provides an authorization service equivalent to the one provided by the "sudo" program [Mil96] with additional logging functionalities to be used together with an intrusion-detection system. Another example of a PLAS program is one in which an applet coming from a financial institution has to be authorized to write and read only from a particular set of data files, and not from any other one.

7 Summary

In this paper we have described the PLAS framework for the definition of security policies. We have discussed the application of the PLAS framework to define security policies within several authorization scenarios, i.e., for traditional access-control systems, protection against downloaded code, and intrusion-detection systems. The requirements of PLAS programs to be used practically have also been described and an example of a PLAS program is presented. We include in Appendix A the grammar of the PLAS language.

References

[AHU74] A. Aho, J. Hopcroft, and J. Ullman. *Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[BFL96] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Research in Security and Privacy, Oakland, CA, May 1996. IEEE Computer Society, Technical Committee on Security and Privacy, IEEE Computer Society Press.

[BM82] U. Bussolati and G. Martella. Data security management in distributed databases. *Information Systems*, 1982.

[CFMS95] S. Castano, M.G. Fugini, G. Martella, and P. Samarati. *Database Security.* Addison-Wesley – ACM Press, 1995.

[DoCM97] Imperial College of Science Technology Department of Computing and England Medicine, University of London. Distributed software engineering. Published in Internet, 1997.

[HRU76] M. Harrison, W. Ruzzo, and J. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, August 1976.

[Jon78] A. K. Jones. Protection mechanism models: their usefulness. *Foundations of Secure Computation*, 1978.

[JSSB97] Sushil Jajodia, Pierangela Samarati, V.S. Subrahmanian, and Elisa Bertino. A unified framework for enforcing multiple access control. In *Proceedings ACM SIGMOD International Conference on Management of Data*, May 1997.

[KLO97] G. Karjoth, D.B. Lange, and M. Oshima. A security model for aglets. *IEEE Internet Computing*, 1(4):68–77, July/August 1997.

[Kum95] Sandeep Kumar. *Classification and Detection of Computer Intrusions.* PhD Thesis, Purdue University, Purdue, IN, August 1995.

[McL94] J. McLean. Security models. In J. Marciniak, editor, *Encyclopedia of Software Engineering*. Wiley & Sons, 1994.

[Mil96] Todd C. Miller. Sudo - a utility to allow restricted root access. Published in the Internet at http://www.courtesan.com/courtesan/products/sudo/, November 1996.

[NHMT98] U. Nitsche, R. Holbein, O. Morger, and S. Teufel. Realization of a context-dependent access control mechanism on a commercial platform. In [PP98].

[OSI84] OSI. Information technology – open systems interconnection – basic reference model. DIS 7498-1, ISO/IEC, 1984. ITU-T Rec. X.200.

[OSI91] OSI. OSI reference model – part 2: Security architecture. Technical report, CCITT, 1991. Same as ISO/IEC JTC/SC21 IS 7498-2.

[OSI94] OSI. Information technology – open systems interconnection – security frameworks in open systems – part 3: Access control. DIS 10181-3, ISO/IEC JTC1, 1994. ITU-T Rec. X.812.

[PP98] Reinhard Posch and György Papp, editors. *Global IT Security. Proceedings of the 14th IFIP International Information Security Conference (SEC'98)*, Vienna, Austria, and Budapest, Hungary, August 31–September 4 1998, (OCG/IFIP, Vienna, Austria, 1998).

[RTG94] P. Rolin, L. Toutain, and S. Gombault. Network security probe. In *CCS'94, Proceedings of the 2nd ACM Conference on Computer and Communication Security*, pages 229–240, November 1994.

[SCFY96] R.S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, February 1996.

[SS96] R. Sandhu and P. Samarati. Authentication, access control, and audit. *ACM Computing Surveys*, 28(1):241–243, March 1996.

[SW97] Jean E. Smith and Fred W. Weingarten, editors. *Research Challenges for the Next Generation Internet.* Computing Research Association, May 1997. Report from the Workshop on Research Directions for the Next Generation Internet.

[SZ96] R. Simon and M. E. Zurko. User-centered security. *ACM SIGSAC, New Security Paradigms Workshop*, September 1996.

[Tro98] Peter Trommler. The application profile security model for downloaded excecutable content. In [PP98], pages 261–270.

Appendix A: PLAS language definition

The PLAS language is designed with a context-free grammar (BNF) as defined in [AHU74] for specifying policies. The elements of a context-free grammar are

- tokens or terminal symbols,
- nonterminal symbols,
- productions rules (nonterminal, arrow, sequence of nonterminals), and
- designation of one nonterminal as the start symbol.

Terminal and Nonterminal Symbols

Nonterminal symbols are written in capital letters, whereas terminal symbols appear in lowercase. We consider the set of terminal symbols as the union of the following sets: the set of all ASCII strings, the set of all numbers, and the set {"begin_evaluation", "end_evaluation", ";" , "=", ".", "begin_rules", "end_rules", "no", "or", "and", "all", "can", "the", "in", "not", "has", "from", "with", "if", "otherwise", "allowed", "forbidden"}. Nonterminal symbols used in the grammar are the following:

Production Rules

```
CompilationModul        →   ModulDeclaration
                            ( IncludeDeclaration )*
                            ( ImportDeclaration )*
                            ( TypeDeclaration )*

ModulDeclaration        →   <modul> Name <;>
IncludeDeclaration      →   <include> String <;>
ImportDeclaration       →   <import>
                            ( <entity>|<action>|<context>
                            |<fact>|<policy>|<condition> )
                            ( < -/W >|< R/- >|< R/W >)?
                            Name String String <;>

TypeDeclaration         →   [ <export> ]
                            ( EntityDeclaration
                            | ActionDeclaration
                            | ContextDeclaration
                            | FactDeclaration
                            | PolicyDeclaration
                            | ConditionDeclaration
                            )

EntityDeclaration       →   <entity> ( EntityBody | Name < . > Name <;> )
ActionDeclaration       →   <action> ( ActionBody | Name < . > Name <;> )
ContextDeclaration      →   <context> ( ContextBody | Name < . > Name <;> )
FactDeclaration         →   <fact> FactBody
PolicyDeclaration       →   <policy> ( PolicyBody | Name < . > Name <;> )
ConditionDeclaration    →   <condition> ( ConditionBody | Name < . > Name <;> )

EntityRef               →   ( <entity> EntityBody ) | EntityID
ActionRef               →   ( <action> ActionBody ) | ActionID
ContextRef              →   ( <context> ContextBody ) | ContextID

EntityBody              →   < {>
                            ( <Name: > XName <;>
                            <Type> EType <;>
                            |<Group> EGroup <;>
                            |<Info> XInfo <;>
```

```
                                 )*
                                 <} >

ActionBody              →        < {>
                                 ( <Command> ACommand <; > | <Info> XInfo <; > )*
                                 <} >

ContextBody             →        < {>
                                 ( <Name: > XName <; >
                                 <Owner> COwner <; >
                                 | <Type> CType <; >
                                 | <Time> CTime <; >
                                 | <Info> XInfo <; >
                                 )*
                                 <} >

FactBody                →        < {>
                                 ( <Entity> EntityID <; >
                                 | <Action> ActionID <; >
                                 | <Context> ContextID <; >
                                 )*
                                 <} >

PolicyBody              →        < {> PolicyExpression <} >
PolicyExpression        →        UnaryPolicyTerm [ Op2 PolicyExpression ]
UnaryPolicyTerm         →        [ <not> ]
                                 PolicyTerm
                                 [ <otherwise allowed> | <otherwise denied> ]
PolicyTerm              →        <(> PolicyExpression <)>
                                 | PolicyStatement
                                 | PolicyID

ConditionBody           →        < {> ConditionExpression <} >
ConditionExpression     →        UnaryConditionTerm [ Op2 ConditionExpression ]
UnaryConditionTerm      →        [ <not> ] ConditionTerm
ConditionTerm           →        <(> ConditionExpression <)>
                                 | PrimaryCondition
                                 | ConditionID

PrimaryCondition        →        FactStatement
PolicyStatement         →        QStmtEntity
                                 ( <can do>|<can not do> )
                                 QStmtAction
                                 [ QStmtContext ]
                                 [ <if> ConditionTerm ]

FactStatement           →        QCondEntity
                                 <has done>
                                 QCondAction
                                 QCondContext

QStmtEntity             →        <everybody>|<nobody>| EntityRef
QStmtAction             →        <everything>|<nothing>| ActionRef
QStmtContext            →        <everywhere>|<nowhere>|<in> ContextRef
QCondEntity             →        <everybody>|<nobody>|<somebody>|
                                 <the entity>|<the requester>| EntityRef
QCondAction             →        <everything>|<nothing>|<something>
                                 |<the action>|<the requested action>| ActionRef
QCondContext            →        <everywhere>|<nowhere>|<somewhere> |
                                 <in the context>|<in the requested context>|<in> ContextRef

Op2                     →        ( <and>|<or>|<xor> )
EntityID                →        [ Name < . > ] Name
ActionID                →        [ Name < . > ] Name
ContextID               →        [ Name < . > ] Name
PolicyID                →        [ Name < . > ] Name
ConditionID             →        [ Name < . > ] Name
XName                   →        String
XInfo                   →        String
EGroup                  →        String
EType                   →        String
ACommand                →        String
COwner                  →        String
```

```
CType     →   String
CTime     →   String
String    →   ”””
              ( ~ [ ”””,”\”,”\n”,”\r” ]
              | ( ”\” ( [ ”n”,”t”,”b”,”r”,”f”,”\”,”’”,””” ]
              | [”0”-”7”] ( [”0”-”7”] )?
              | [”0”-”3”] [”0”-”7”] [”0”-”7”]
              )
              )
              )*
              ”””

Name      →   Letter ( Letter | Digit )*
Letter    →   [”A”-”Z”, ”_”, ”a”-”z”]
Digit     →   [”0”-”9”]
```