Programmable Agents for Generic Distributed Authentication

Nick Antonopoulos\textsuperscript{1}
James Heather\textsuperscript{1}

\textsuperscript{1} Software Systems Research Centre,
Department of Computing,
School of Electronics and Physical Sciences
University of Surrey,
Guildford, Surrey
GU2 7XH,
United Kingdom

\texttt{N.Antonopoulos@surrey.ac.uk}
\texttt{J.Heather@surrey.ac.uk}

Tel: (+44) 1483 686052
Fax: (+44) 1483 686051

1. Introduction

Public key cryptography, once esoteric, is now ubiquitous. The rise of the Internet, the cheapness of encryption, and an increased focus on security, have made it desirable if not essential for every machine and every serious user to have at least one public and private key pair.

The biggest challenge resulting from this explosion in popularity is that of key management. How can one obtain the public key of a given user? How can one determine who is authorised to access resources while allowing authorisation to be delegated?

There are many systems available that aim to deal effectively with this problem, but the three most common are SPKI [1], PGP’s web of trust [2], and X.509 [3].

Here we discuss these three systems and their associated problems, and then propose a new setup that retains the advantages of each while circumventing the weaknesses.

1.1 SPKI

In SPKI, authorisation is granted to the holder of a particular private key rather than to a named individual. Every public key is also a certification authority; there are no globally trusted entities.
To gain access to a resource, a user must supply a request signed with his private key, and a certificate chain proving that this key is authorised to access the resource. This certificate chain should eventually terminate in a certificate issued by the owner of the resource.

1.2 PGP’s web of trust

PGP allows a user to specify the other users whom he trusts. Thereafter, the trust rules are applied transitively: if a user trusts his friend, he implicitly trusts his friend’s friends. Acceptance of a certificate then depends on finding a trust path reaching from one user to the other.

Additionally, a user may specify marginal trust for a certificate. The rules require two separate marginally trusted users to trust a third party before the third party is accepted as trusted.

1.3 X.509

X.509 certificates underpin SSL, the basis of security on the Web. An X.509 certificate associates a public key with a URL, and specifies the uses to which the corresponding private key may be put. Each certificate is signed with the private key associated with another certificate, forming a certificate chain; the chain eventually terminates in a certificate that is signed with its own private key.

In order for a certificate to be validated, this self-signed certificate must be one known to, and trusted by, the user performing the validation. What this requires is globally trusted entities with globally known public keys. Typically, these public keys are built into the browser or operating system.

2. Problems

There are three key features of a PKI system that are desirable:

1. recording of attribute dependencies;

2. a high degree of expressiveness;

3. decentralisation.

However, each system so far discussed leaves room for improvement in at least one of these areas.

2.1 Attribute dependencies

PGP and X.509 both allow for a certificate to associate several identities or attributes with a public key. A PGP certificate might identify the holder as ‘Employee of ACME’,
and also as ‘Manager of ACME’; an X.509 certificate might give the holder the right to perform employee-related functions, and also managerial functions.

However, it is clear that the latter depends on the former. If the holder ceases to be an employee, he should also cease to be a manager. If the employee attribute or identity is revoked, the manager attribute or identity should also be revoked.

Neither PGP nor X.509 can record this dependency. In fact, neither system allows revocation of a single attribute: only entire certificates may be revoked. It would clearly be advantageous to have a system in which such attribute revocation could be performed; and if this is to be performed consistently, it requires recording of dependencies.

2.2 Expressiveness
SPKI provides no mechanism for specifying that a user must meet two or more conditions in order to gain access; that is, the ‘and’ operator is lacking. It is also impossible to specify distrust; that is, that a particular agent is not authorised regardless of any certificate chain he may produce. In addition, SPKI does not allow authorisation to depend on the sequence of the certificates in the chain. Finally, there is no provision for revocation: SPKI insists that certificates should always be issued with a short lifespan, so that revocation can be approximated by simply not renewing a certificate.

PGP does not allow users to distinguish direct trust from transitive trust: if Andrew trusts Belinda who trusts Charles, the rules will force Andrew to trust Charles exactly as if he had expressly determined to trust him. The rules of PGP are, in fact, rather stringent in general, and cannot be tailored to the demands of the user.

2.3 Decentralisation
Today’s computing ethos is everywhere that of decentralisation. Fault tolerance and efficiency are the main advantages, but there is also a security risk involved in placing one’s entire access control system on a small number of machines.

SPKI requires that all of its certificates be stored centrally. The design of the system makes this unavoidable; this is possibly SPKI’s weakest point.

X.509 does not require full centralisation, but it does (for a global network like the Internet) require globally trusted entities with globally known public keys. The problem is that worldwide security depends on the security of a few keys stored on a few machines. One successful hacking attempt on one certification authority can result in fraudulent certificates being accepted worldwide.

3. System Architecture
The authentication of users in any system can be modelled as a process that takes as input the various identities (typically represented as public keys) of the user, attributes of the
user, and system-wide policies, and produces a Boolean output determining whether the user was successfully identified or not. The meaning of the user-related attributes can vary significantly: some may identify the user’s membership in specific groups, some may specify the level of trust others have in this user, some may describe the roles the user has in a specific domain.

The state-of-the-art systems described in the previous sections provide mechanisms for creating, assigning and managing user attributes that fall into one of these categories; for example, PGP manages trust-related attributes, whereas SPKI focuses more on user-group attributes. We claim that this separation is rather artificial: an authentication system should be flexible enough to handle attributes of various types. In this way, the users and administrators of the system can define their own customisations that tailor the authentication and identity management processes for specific needs. For instance, a company could use role attributes for internal user authentication, group membership attributes for customers, and trust attributes for employees from other companies.

To design such a system, we have to treat user attributes in a uniform manner. Intuitively, we can consider all such attributes as labels that are assigned to user keys whenever they satisfy certain criteria. *The meaning of the label depends on the type of the entity that created it:* PGP keyrings generate trust labels, SPKI name certificates generate group membership or role attributes, etc. Labels assigned to specific keys can be linked to each other with an *existential relationship*, which implies that the assignment of a label to a key is conditional upon the continuous association of other labels with that key. The ability to represent such relationships makes the system more accurate in reflecting user and organisation-specific policies in terms of both assigning and revoking labels. Effectively, the relationship between labels becomes an attribute of the user.

Figure 1 provides an overall view of the architecture of our proposed system by modelling the main entities and their relationships. Each user has a set of pairs of private and public keys that are maintained in his proprietary keystore. The resources to be protected are organised into resource domains, each of which is represented by a domain manager; the domain manager maintains an authentication expression that users must satisfy in order to be allowed to access the resources the manager protects. The authentication expression is a Boolean expression containing the labels (independent or linked) that one or more of the user’s keys must be assigned in order for access to be granted. The system recognises that typical users not only request access to specific resources but also define authentication criteria for their own resources. For this purpose, each user can organise his resources into proprietary domains and define a specific authentication expression for each domain by means of creating and managing his own domain managers.

The main processing entities in our system are the *Simple Label Agents* (SLAs). An SLA is a typed entity that has its own pair of private and public keys and behaves as an autonomous, reactive agent. The type of each SLA is defined as the set of labels it can
assign to user keys. It operates as a mapping function; its input is a set of user keys with arbitrary, independent or linked labels, and its output is the same keys appended with one or more labels from the set of labels that define its type. To achieve this, the SLA contains a mapping table, defined by its owner, which contains pairs of string patterns and labels. Whenever it receives a labelled key, it matches it with the string patterns of the mapping table. If there are one or more matches, then the union of associated labels is appended to the incoming labelled key. If no matches are found then an error is reported.

Simple Label Agents also act as key producers for the system users. This is useful for modelling processes that lie outside the system boundary. For example, a new employee in a company is given a unique personnel number after he has followed a certain procedure (application, presentation, interview, etc.) that cannot be represented within the authentication system. The actual number can be considered as a key given to the employee by an SLA with the name “personnel”, and could be used in conjunction with other attributes to provide the user with access to certain services or domains.

Figure 1. System architecture
SLAs are owned and created by users. Some can be personal, and others can represent department or even enterprise-wide authentication rules. Users create their proprietary SLAs by using the *factory* entity, which is responsible for creating authentic copies of software that is certified to behave as a simple label agent. Each SLA is provided with its own pair of secret and public keys and a certificate verifying the mapping of the SLA’s name to the public key supplied by the factory. In this way the factory acts as a common trust base for all the SLAs it creates. The factory can also initialise the mapping tables of the SLAs with constant, mandatory entries that represent policies that must be followed by all SLAs. In this context, the factory acts as authentication domain manager.

Users can also create *Composite Label Agents (CLAs)* by combining two or more SLAs of the same type in an expression using the *union, intersection* and *complement* operators. Effectively, the SLAs are treated as reusable components to create more complicated policies based on existing ones. As before, the factory mediates the process while ensuring that the SLAs to be combined are of the same type. The CLAs are software agents whose mapping and labelling procedures are similar to those of the Simple Label Agents. We describe their structure and functionality in detail in section 5.

The authentication process in our system is as follows: Assume that a user wishes to gain access to a certain information domain—say, a web site. A program representing the user (*a user proxy*) contacts the appropriate domain manager (1) that returns the authentication expression the user must satisfy in order to gain access to the domain (2). The expression refers by name to an arbitrary set or chain of label agents (simple and/or composite), and the labels that at least one of the user keys must be assigned by these agents in order to be authenticated by the domain manager. The user proxy sends the contents of the user’s keystore as well as the authentication expression to the first referenced label agent (3). The agent uses its mapping table to deduce the labels to be assigned to one or more of the user keys. If there is at least one match, then the appropriate labels are appended to the user keys. The updated user keystore is then forwarded to the next label agent identified in the authentication expression. The last label agent in the chain, assuming that its matching process was successful, sends a copy of the labelled user keys back to the proxy to be saved in the user’s keystore for future use (4.a) and another copy to the domain manager (4.b). The domain manager verifies the signatures in the received labelled keys and sends a *challenge* (randomly generated string) to the user proxy (5). The user proxy returns the challenge, signed by the private counterparts of the public user keys that were labelled during the aforementioned process (6). After the successful verification of these signatures, the domain manager can generate and send the user proxy a secret key to be used for future access requests, thus bypassing the need to use any intermediate entities (label agents).

**4. Simple Label Agents**

The simple label agents are the main building blocks of our architecture. They are independent pieces of software that apply specific production rules to annotate user keys with certain labels. Essentially, the production rules of an SLA specify the labels (sequence or independent) a user key must already have from other label agents in order
to gain one or more labels from this agent. For this reason, the association of new labels with a user key remains valid only if the association of the prerequisite labels remains valid as well. To represent this relationship between labels produced by different agents, each simple label agent uses the method of encapsulation: it appends its own labels to the already existing ones and signs the whole string with its private key.

Specifically if the incoming, possibly labelled, user keys match at least one production rule (string pattern), then the SLA appends an ordered tuple whose components are:

- **Name**: the name of the labelling agent (URL),
- **Type**: the type of the label agent,
- **Domain**: the name of the factory that created this agent,
- **Label(s)**: the tags this agent attached to the incoming user key,
- **Exp_date**: the date after which this association becomes invalid, and
- **Signature**: the signature of the label agent that encapsulates the user key together with all the associated tuples.

Suppose that there is a web site containing the personal details and marks of all the MSc students in the Department of Computing of the University of Surrey (UniS). Only the department’s academic staff can access the site. The authentication expression for the web site is

```
*/[computing, department, UniS, */[role, jobtype, UniS, academic]
```

where the square brackets denote the boundaries of each tuple and the symbol ‘/’ indicates the existential relationships between the tuples.

Also assume that the label agent *role* has the following rules:

```
*/[*[, department, UniS, lecturer] => academic
*/[*[, department, UniS, teaching_assistant] => academic
```

which indicate that any key labelled as *lecturer* or *teaching_assistant* from any department in the UniS domain is also labelled as *academic*. If this agent receives the following key:

```
user_public_key1/[computing, department, UniS, lecturer, signature_computing]
```
then it will match the first entry of the agent’s mapping table and the output will be:

\[
\text{user\_public\_key1/\{computing, department, UniS, lecturer, signature\_computing\}/}
\]

\[
[\text{role, jobtype, UniS, academic, signature\_role}]
\]

which will then be sent to and accepted by the domain manager guarding the aforementioned web site. The role agent will also send a copy of the labelled key back to the user’s keystore. Observe that the label \textit{academic} can exist as part of the user’s key1 only if the label \textit{lecturer} is present as well.

The fact that SLAs are autonomous software agents that can be created by any user and referred to by name in the authentication expression of any domain manager implies that they can be elegantly represented as specialised Web services.

Figure 2 shows the model and interface of a typical SLA expressed in XML schemas and Web Service Description Language (WSDL) respectively. Note that each SLA maintains, among other information, a list of domain managers (names and corresponding public keys) that refer to the specific agent in their authentication expressions. This is useful for label revocations. If the table of an SLA changes, then the agent sends a message to all the listed domain managers in order to notify them of the change. The domain managers can use this information to mark any user keys that have been labelled by this agent in the past as invalid. If a user attempts to use a key previously labelled by that agent then the domain manager will require the key to be revalidated by going through that label agent again.

The interface an SLA supports includes operations enabling its user-owner to manage the contents of its mapping table. The most important operations are \textit{forward} and \textit{prove}. The \textit{forward} operation implements the mapping functionality we described earlier in this section; the \textit{prove} operation emulates the name resolution procedure in SPKI. It is used when the authentication expression of a domain manager specifies the labels the user keys must possess, but doesn’t indicate any particular sequence in which these labels are to be acquired. In this case, the contents of the user’s keystore are supplied to the agent that produces the required labels. The agent finds which production rules must be satisfied in order for a key to take the required labels and activates the \textit{prove} method of other label agents accordingly. If there several rules that produce the required labels, the agent can use depth-first or breadth-first searching strategies to find a path from one of these rules that terminates with at least one of the user’s public keys.
5. Composite Label Agents

Composite Label Agents provide an elegant mechanism for composing complex label production rules by reusing existing ones. A CLA implements an expression between two or more SLAs of the same type. The mapping table of the CLA is created by concatenating the mapping tables of the constituent SLAs. The XML schema describing the mapping table of a CLA introduces an attribute that contains the name of the originator SLA for each entry. The CLA functions much the same way as an SLA. Three simple rules are used to determine the output of a CLA whenever it is activated by an incoming labelled user key:
Rule 1 (union of SLAs): The incoming user key is matched against all entries of the CLA’s mapping table. The output is the union of all the labels produced by each matching rule. The union operation is useful for finding out, for example, whether a user key is trusted by one or more arbitrary users.

Rule 2 (intersection of SLAs): The incoming user key is matched against all entries of the CLA’s mapping table. The output is the set of labels produced by taking the intersection of the label sets generated by all matching rules for each constituent SLA. The intersection operation is useful for finding out, for example, whether a user is a member in a number of specified groups.

Rule 3 (complement of SLA): The incoming user key is matched against all entries of the CLA’s mapping table. If there is at least one match then the CLA generates an error message. The complement operation is useful for authenticating users that do not play a specific role in a business environment.

By combining the above rules we can define the behaviour of CLAs implementing more complex set expressions. Figure 3 shows the process for creating a CLA. It is worth mentioning here that each SLA maintains a list of all the CLAs it participates in. Whenever the SLA’s table is updated, the SLA forwards the new version of its table to all the dependent CLAs. Users are not allowed to modify directly the mapping table of CLAs for consistency purposes.

![Diagram](image_url)

**Figure 3. The process for creating composite label agents**
6. Implementation

In a full implementation of this architecture, the simple and composite label agents would be represented as SOAP processors that communicate directly with each other via SOAP/HTTP. The SOAP processing model can easily embody the concept of multiple intermediate label agents annotating user keys with chains of labels.

To demonstrate the applicability and feasibility of our system, we have built a demonstrator and installed it internally within the Department of Computing to protect a pool of shared files. The simple and composite label agents are implemented as server-side scripts written in Perl (because of its advanced pattern matching capabilities). The mapping tables are stored as DBM files with the key being the production rule and the value being the set of labels to be appended if the rule is matched. We implemented the forward method of the label agent’s interface by using Perl’s redirect function. Figure 4 shows the graphical interface via which users can create and update label agents, create resource domains, and send access requests to domains. The second snapshot shows that a user was successfully authenticated to access the domain Computing; for future accesses he can use a secret password produced by that domain manager.

![Figure 4. Interface of prototype implementation](image-url)
7. Evaluation

The efficient management and enforcement of authentication information and policies is a critical process that can determine the success or otherwise of distributed information systems. Current approaches manage and enforce authentication policies based on independent user attributes with specific semantics: trust OR roles OR group membership, etc. We are proposing an authentication management system that represents attributes of any kind as labels. The system defines a set of peer instances of a specialised Web service called label agent. It operates as a synergetic network of autonomous agents that collectively annotate user public keys with chains or sets of labels. A label chain indicates the dependencies between labels by using the principle of multiple encapsulated digital signatures, thus making the system more accurate in representing authentication policies and more efficient in dealing with revocation. Simple Label Agents are treated as reusable components that can be combined, forming more complicated label production rules based on existing ones. The system does not introduce any centralisation as every user can create proprietary label agents that can then be referred to by name (URL) by any resource domain manager for authentication purposes.

In contrast to our architecture, SPKI does not record label dependencies in its name certificates, with the result being that it has to rely purely on expiry dates for revocation. Its expressiveness is restricted by the fact that the ‘and’ operator between labels cannot be represented. SPKI is centralised in nature: it requires all the name certificates to be stored in a central server. PGP can only model trust labels that are produced based on predefined, constant rules, which cannot be modified to meet individual needs. X.509 suffers from the same problems as SPKI with the addition that it relies on globally trusted authorities and globally unique names. None of the above systems attempts to address the issue of reusability in the management of authentication information.

Organisations can use our approach and customise it to suit their proprietary needs. The system can effectively model roles, trusts, user groups, and so on, since everything depends on the vocabulary of labels used and the semantics associated with these labels. It’s distributed nature makes the system more fault-tolerant and applicable in emerging, distributed computational paradigms such as GRIDs, peer-to-peer networks, Web services and pervasive computing.

References

