# Auth-SL - A System for the Specification and Enforcement of Quality-Based Authentication Policies

Anna C. Squicciarini, Abhilasha Bhargav-Spantzel,
Elisa Bertino, and Alexei B. Czeksis

Department of Computer Science, Purdue University
{asquicci,bhargav,bertino,aczeskis}@purdue.edu

**Abstract.** This paper develops a language and a reference architecture supporting the management and enforcement of authentication policies. Such language directly supports multi-factor authentication and the high level specification of authentication factors, in terms of conditions against the features of the various authentication mechanisms and modules. In addition the language supports a rich set of constraints; by using these constraints, one can specify for example that a subject must be authenticated by two credentials issued by different authorities. The paper presents a logical definition of the language and its corresponding XML encoding. It also reports an implementation of the proposed authentication system in the context of the FreeBSD Unix operating system (OS). Critical issues in the implementation are discussed and performance results are reported. These results show that the implementation is very efficient.

## 1 Introduction

Authentication is the process by which systems verify the identity claims of their users. It determines *who* the user is and if his claim to a particular identity is true; authenticated identities are then the basis for applying other security mechanisms, such as access control. Generally speaking, a user can be authenticated on the basis of something he holds, he is, or he knows.*Something you know* is typically implemented through mechanisms such as password, or challenge-response protocols. The *something you hold* approach is implemented through token-based mechanisms, smartcards, or a PIN that the user possesses and must present in order to be authenticated. Finally, the *who you are* paradigm is based on biometrics and includes techniques such as fingerprint scans, retina scans, voiceprint analysis, and others.

A same system may have resources with different requirements concerning authentication strengths for the users wishing to access them. A straightforward solution to authentication for resources with such heterogeneous requirements is based on a conservative approach that maximizes authentication checks each time a user connects to the system. However, such a solution may result in

S. Qing, H. Imai, and G. Wang (Eds.): ICICS 2007, LNCS 4861, pp. 386–397, 2007.
© Springer-Verlag Berlin Heidelberg 2007

computationally consuming authentication tasks and may also be very expensive and complex to deploy. For example, adopting one-time passwords [12] for all users of an organization, independently from the tasks they have to perform and the resources they have to access, may be very expensive; ideally one would like to require such authentication measures only for users who need to access sensitive resources and use conventional passwords for the other users. Additionally, such an approach does not avoid the risk of session hijacking.

We believe that authentication should be based on a variety of mechanisms targeted to the resource security requirements and be easily configurable. Identity of users should always be known and certain during the whole duration of a user session within the system, especially as the user browses multiple resources. *Continuous authentication* [3] has been proposed to tackle issues related to fake authentication from attackers. Most approaches to continuous authentication are based on biometric techniques, like keyboard typing recognition or face recognition through trusted cameras [7]. However these approaches require costly machinery and tools and in addition are based on the assumption that the one method of authentication is to be accepted for every possible resource the user connects to.

Logic based authentication approaches [1, 15] have been proposed to support a weak form of continuous authentication through the association of multiple principals with each user. However, these approaches have mostly focused on abstract representation of roles, groups, and delegation. Mechanically generated proofs have resulted to be impractical to compute. As we discuss in more detail in the related work section, such approaches are not expressive enough to support fine-grained authentication policies. We thus believe that more articulated solutions are needed based on the use of multiple authentication mechanisms combined through *authentication policies* and on the association of authentication requirements with the protected resources. The goal of our work is to develop such a solution.

We propose an authentication framework based on an expressive *authentication policy language*. By using such language, one can specify how many authentication factors are required and of which type, for accessing specified resources, or impose constraints on the authorities by which credentials used for authentication have to be provided, thus providing a *quality-based* authentication. Flexibility in specifying the various factors for authentication is important as typical two-factor authentication mechanisms may not be sufficient to satisfy the security requirements of a given system [16]. It is important to notice that the SAML (Security Assertion Markup Language) standard [11] supports the encoding of authentication statements for exchange among sites in a distributed system. The goals of our authentication policy language are different from the goals of SAML. SAML is a standard for encoding authentication statements; such a statement typically asserts that a given subject has been authenticated under a certain modality by a given entity at a given time. *SAML thus does not deal with taking authentication decisions*; it only deals with encoding and transmitting such decisions. *The goal of our language is exactly to specify policies driving authentication decisions*; as such policies expressed in our language may also take into account previous authentication decisions, taken for example by other sites in a distributed system, together

with other information in order to reach an authentication decision.In what follows we refer to our framework as *authentication service language*, abbreviated as $Auth-SL$. Our goal is to develop a comprehensive set of functions for specifying, managing, enforcing, and inspecting authentication policies that can be used by parties and applications in a system.

The contributions of our work are as follows: (1) The development of a reference architecture for a novel authentication service. (2) The specification of a language to express authentication policies. The proposed language supports the specification of the number of authentication factors required for accessing a resource and the qualification of the authentication factors in terms of a large variety of conditions. (3) An implementation of the proposed authentication service and the policy language in the context of the FreeBSD Unix OS, which allows continuous authentication. That is, the user can fluidly re-authenticate users throughout sessions. Authentication policies can be associated with the protected resources, in addition to being used when the user initially connects to the system; our implementation thus supports the notion of continuous authentication. Auth-SL, as our experiments show, is also very efficient; it improves the functionality of the OS without impacting its performance.

We would like to emphasize that our approach departs from the conventional security "pipeline" according to which, during a user session with a system, authentication is executed only once at the beginning of the session, and then access control is applied multiple times during the session. Our approach proposes a different paradigm under which the activities of authentication and access control can be interleaved in a session, depending on the specific security requirements of the resources accessed during the session. It is important to notice that the conventional pipeline can be supported as a special case of our approach.

The rest of the paper is organized as follows. In Section 2 we present the reference architecture for our authentication service. We then present the formal definition of authentication language and discuss the implementation in FreeBSD Unix. Finally we outline future work.

## 2 Reference Architecture for an Authentication Service

We begin with a reference architecture of our authentication service, to clarify the main logical components. Auth-SL consists of two major subsystems, namely the *authoring subsystem* and the *enforcement subsystem*.

**Authoring subsystem.** This system supports the specification and the management of the authentication policies. One of its key features is that it supports the specification of which mechanism to use through the specification of conditions against the features of the available mechanisms. Such specification relies on two components: a *library of authentication modules*, very much like a set of PAM modules [8]; and a specialized *UDDI Registry* recording all features of the authentication modules that are relevant for the specification of the authentication policies. Each module in the library supports a specific type of authentication.

Such modules can then be dynamically invoked to enforce the specific authentication policies. The information required about the authentication modules that are needed for authoring authentication policies is as follows: (1) *Module's authentication characteristics*. These data describe the settings for the specific mechanism. For example, in a password based authentication, a characteristic is the maximum number of tries allowed, or the minimum length of the password. For token-based authentication, a characteristic is the authentication method (e.g. SSO, Basic-Auth credentials), NTLM credentials (username, password, domain), and X.509 client certificates, and the software used (e.g. IBM Tivoli Client RSA). (2) *Implementation data*. These parameters qualify the specific implementation of a mechanism and can refer to the storage of the secret token, the cryptographic technique used to transmit it, the audit trails and so forth.

The authentication policies that can be expressed thus depend on the authentication modules available, and the characteristics of these modules. Such data are to be considered part of the knowledge needed to specify adequate authentication policies. For example, if a system administrator knows that a given authentication module is weak, due to implementation limits or module vulnerabilities, he can apply stronger authentication policies. Authored authentication policies are stored into a repository referred to as *Authentication Policy Base* providing query capabilities to properly authorized users, such as system administrators and auditors.

**Enforcement subsystem.** Upon an authentication request, such system is in charge of evaluating an authentication policy and make an *authentication decision*. The evaluation is executed by the *Authentication Enforcement Point*, which first retrieves a proper authentication policy. Policy evaluation may also take into account previous *authentication events* concerning the subject being authenticated. To express fine grained constraints over past authentications we collect information on the past authentication in two different logs, serving different purposes: (i) track subjects actions related to authentication and (ii) record the conditions under which a successful authentication is executed. In the first log, referred to as *Authentication event log*, we record authentication events (*event* for short) related to the subjects. An authentication event is basically an authentication executed against a subject. Such log tracks in a chronological order all events related with authentication of the users performed during each session. Once the policy is evaluated, a new event is generated and stored in the log in order to keep track of this authentication step. Each record can refer to either an authentication attempt using a specific factor, the verification and/or the failure of the verification of a given factor. A successful authentication implies successful authentication of multiple factors traced in the event log.

The *context data log* instead tracks specific data related to previous authentication. The information stored by such log is used to evaluate whether previously executed authentication can be leveraged for satisfying an authentication policy An instance of the context data log is created when the user begins a session and it is maintained only for the session duration. Each log record stores context data related to the specific authentication performed, and the settings of the

module used. In the current Auth-SL system, each entry in a context data log collects: the type of mechanism used, the time of the authentication execution, the number of failed attempts, the party that originally generated the authentication token used, storage information (remote versus local token storage)and the storage mode (encrypted versus clear text token). Note that Auth-SL does not mandate the specific set of data to be tracked. Additional data may be saved, according to the specific system modules and system security requirements.

The output of the enforcement subsystem is an authentication assertion, which can be returned either to the user or transmitted to some other system or application. Since policies are associated with resources, in most cases the authentication service will interact with the access control system. Typically when subject requires access to a resource, the access control system will will require the authentication service to determine if there are authentication policies associated with the resource and, if this is the case, to evaluate such policies.

## 3 The Policy Language

In this section we discuss the language for the specification of authentication policies. We begin introducing some notation and symbols to be used for the policy specification and then illustrate the syntax of the language.

### 3.1 Constant Symbols

The constant symbols used in our language are described as follows.

*Objects* ($\mathcal{O}$) denotes the set of objects available in the system. Each object has an associated set of operations according to which the object can be accessed. We denote the possible set of operations for object $o$ in $\mathcal{O}$ as $OP_o$[1].

*Authentications Modules* ($\mathcal{AM}$) is a set of authentication modules available in the system library. We assume that modules are described in terms of parameters collected in a set *ModP*. Each module $m \in \mathcal{AM}$ has an associated profile, defined by a subset $\{var_1^m, \ldots, var_k^m\}$ of elements in *ModP*. In particular, each profile always includes a mechanism type name (denoted as *MechType*), specifying the type of mechanism supported by the module. Some mechanisms are also qualified in terms of the algorithm used for authentication, as for instance the cryptographic algorithm or the algorithm used for biometric authentication.

*Policy constraints* ($\mathcal{P}$) is a set of policies used to establish authentication requirements for elements in $\mathcal{O}$. We assume that for each $o \in \mathcal{O}$ there is at most a policy $p \in \mathcal{P}$. Policies are defined as combination of *authentication factors* ($\mathcal{F}$), to qualify the authentication to be executed.

*Time* ($\mathbb{T}$) is the discrete time in the system.

### 3.2 Formal Definitions

Authentication policies are the key elements to drive authentication decisions. The specification of authentication policies relies on the notion of *Authentication*

[1] The set of resources contains at least the object corresponding to the user login.

*Factor.* Authentication factors define the features of a specific authentication, using one specific mechanism in $\mathcal{AM}$, and are described in terms of **descriptors**. Each descriptor has at least one parameter, which is the *alias* -or unique identifier- of the authentication factor.

**Definition 1 (Descriptor).** *A descriptor* d *is a predicate of the form* $p(x, \mathbf{t})$*, in which* $x$ *is a variable, and* $\mathbf{t}$ *is a vector of one or more terms*[2]*.*

Descriptors can be classified into four different categories, according to the specific property they capture.

† *Authentication Verifier descriptors.* These descriptors state properties of the verifier of the authentication token. This could be related to the trusted third party that originated the secret token, or to the module that at the time of verification of the identity token checks its integrity.

† *Module Characteristics.* These properties describe the characteristics of the module used for the authentication and the configuration used to run authentication.

† *Context Information.* These properties refer to external conditions that may arise during a specific authentication.

† *Space and time.* These descriptors attest properties of the authentication factors with respect to space and time constraints.

Properties of a specific authentication could potentially be described in various ways. In Auth-SL, we chose to represent them through a finite set of descriptors to enable specification of fine grained authentication policies. Relevant descriptors necessary to express articulated policy conditions are provided in [17]. Authentication factors are specified through a Boolean conjunction of descriptors.

**Definition 2 (Authentication factor).** *An authentication factor is a Boolean conjunction of descriptors* $d_1, \ldots, d_k$*, each of the form* $d = p(x, \mathbf{t})$*, such that: (1) The same factor variable* $x$ *appears in every descriptor* $d_m = p(x, \mathbf{t})$ $\forall m \in [1, k]$ *(2)* $\exists d_j, j \in [1, k]$ *such that* $p_j(x, a) = Mechanism(x, a), a \in MechType$*.*

We describe a factor in terms of the descriptors $\{d_1, \ldots, d_k\}$ composing it, when the exact arguments of the descriptors are not needed. As from Definition 2, authentication factors can be defined using any possible combination of descriptors. The only mandatory descriptor is the one specifying the mechanism to use.

*Example 1.* Examples of authentication factors are the following:

1) $Mechanism(z_2, Biometric) \wedge Algorithm(z_2, VeriFinger) \wedge TimeBefore$ $(z_2, t'')$,
2) $Mechanism(z_1, Kerberos) \wedge TimeBefore(z_1, t)$

The authentication factors, as defined, are stand alone in that the specification of one single factor is not related to any other factor. However, this is not adequate for the specification of complex and multi-factor authentication policies.

[2] Recall that a term is either a variable like *cid* or it is a compound term $f(t_1, \ldots, t_k)$ where $f$ is a function symbol of arity $k$ and $t_1, \ldots, t_k$ are smaller terms.

To correlate different factors and their characteristics specific constraints can be specified. Factor constraints are specified as logic formulae in which the occurring variables are the factor identifiers. We assume the existential and universal formula be specified always over attributes having a finite domain. The domain of constraints supported belongs to the class of *order and inequality constraint domain* [10]. This domain include binary predicates as defined in our comparison assertions set presented in Section 3.1.

**Definition 3 (Factor Constraints).** *Let $d_1, \ldots, d_k$ be authentication descriptors specified according to Definition 1. A factor constraint $\phi$ for descriptors $d_1, \ldots, d_k$ is a first order logic formula defined expressing conditions against variables appearing in $\{d_1, \ldots, d_k\}$.*

*Example 2.* Let $d_1, d_2$ be two different authentication descriptors. An example of constraints are:

$$\phi_1 = \exists(TrustedParty(x_1, value_1) \wedge TrustedParty(x_2, value_2)) \wedge value_1 \neq value_2$$
$$\phi_2 = \exists(TimeBefore(z_1, t'')) \wedge TimeBefore(z_2, t')) \wedge t' > t''$$

The first constraint requires that the two factors be issued by different trusted parties. This is useful to impose authentication to be proved trough credentials issued by different authorities. The second constraint implies an ordering in the execution of the factors and requires factor $d_1$ to be executed after $d_2$.

We are now in the position to formalize the notion of authentication policy.

**Definition 4 (Authentication Policy).** *An authentication policy* p *is a tuple of the form $\langle obj, op, [d_1, \ldots, d_k], Ts, \Phi \rangle, k \geq 1$, where:*

- *$obj \in \mathcal{O}$ is the object target of the policy;*
- *$op \in OP_{obj}$ denotes a non-empty set $\{op_1, \ldots, op_k\}$ of operations according to which obj is to be accessed.*
- *$[d_1, \ldots, d_k]$ is a list of authentication factors, such that $d_j \neq d_m$ if $j \neq m$;*
- *Ts denotes the number of mandatory authentication factors to be verified, thus $1 \leq Ts \leq k$;*
- *$\Phi$ is a set of factor constraints $\{\phi_1, \ldots, \phi_k\}$; each $\phi_i$, $i \in [1, k]$ , is specified in terms of descriptors appearing in $d_1, ..., d_k$.*

An authentication policy is by definition specified by a combination of factors to be evaluated. The execution of all the factors may or may not all be mandatory, as specified by threshold value, denoted by *Ts*. The specification of *Ts* enhances the flexibility of authentication by establishing the sufficient demands needed to authenticate the user. The listed factors are to be evaluated accordingly.

*Example 3.* The following is an example of authentication policy:

$p = \langle file1, \{open\}, [f_1, f_2], 2, \phi_2 \rangle$ states that to be authenticated for opening $file1$ the user identity should be checked by executing both factors $f_1$ and $f_2$. Here, $f_1$ and $f_2$ correspond to the factors in Example 1 and $\phi_2$ denotes the constraint of Example 2.

To avoid specification of policies which cannot be processed by the policy enforcement point, authentication policies should be *well-formed*.

**Definition 5 (Well-formed policy).** *Let* obj *be a object,* op *be the associated operation and let* $p = \langle obj, op, [d_1, \ldots, d_k], j, \Phi \rangle$, $k \geq 1$*, be an authentication policy.* p *is a well-formed policy for* obj *if the following condition holds:* $Ts = j, j \leq k$ *and a set of* $j$ *factors* $d_{m1}, \ldots, d_{mj}$ *exists in* $[d_1, \ldots, d_k]$ *such that each* $\phi \in \Phi$ *that involves factor variables in* $d_{k1}, \ldots, d_{kj}$ *is satisfiable.*

By definition, satisfiability of the constraints needs to be guaranteed. Also constraints expressed in terms of factor variables referring to factors that are not part of the subset need to be satisfiable. That is, if they refer to factors that are not part of the list, the policy is not well formed. We clarify this concept with a simple example.

*Example 4.* Consider a policy that specifies $[f_1, f_2, f_3]$ and requires at least 2 out of 3 factors to be verified. If among the constraints in $\Phi$ there is a constraint $\phi_1$ that compares qualities of the factor $f_1$ with qualities of factor $f_2$ and there is a second constraint $\phi_2$ that compares qualities of $f_2$ with qualities of $f_3$, then the policy is not well-formed. The constraints can only be evaluated if all the 3 factors are verified, and this contradicts the threshold value.

Verifying whether a policy is well-formed or not is a decidable and deterministic problem, as a consequence of the fact that the set of factors and constraints is always finite and of the adopted constraint language.

## 4 Implementation of the Authentication Service in FreeBSD Unix

As part of our work, we have developed a prototype of the authentication service in the context of the FreeBSD Unix OS[9]. The main components identified in the framework reported in Section 2 have been translated into modified modules/operations for implementation in FreeBSD. A sketch of the resulting prototype architecture is presented in Figure 1. The core of the system, which is represented by the authentication enforcement point, has been implemented trough a set of APIs, for policy access and context access. We elaborate on those components as well as on the above issues in what follows and we also report some performance results.

**Policy Encoding.** Each object in the OS is associated with one authentication policy, composed by one or more authentication factors. In order to support an efficient processing of policies, we provide an internal representation of policies expressed according to the C language. Auth-SL policies are encoded using XML and then parsed into C functions by an authoring tool. Each policy function is associated with a unique ID. Policy functions are parameterized with actual constraint values that appear in the policy factors. The C functions, which evaluate the logic of a particular policy, take as input parameters the context data log and the parameter values that qualify the arguments of the descriptors for factor.

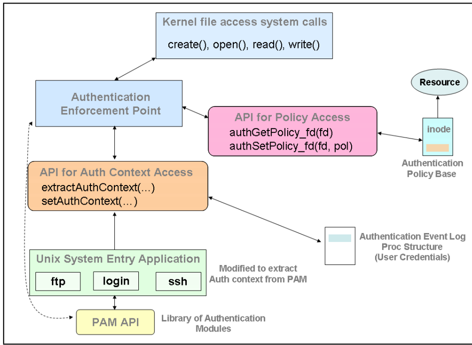


**Fig. 1.** Prototype architecture

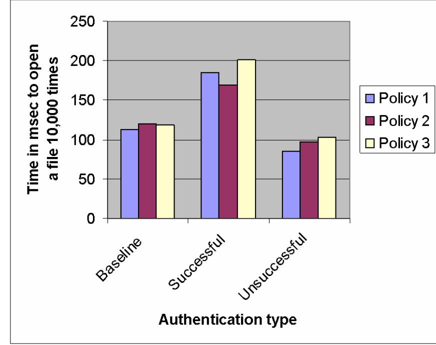


**Fig. 2.** Auth-SL authentication services

**Policy Storage and Binding.** The storage strategy adopted for the policies is a key element for ensuring good performance and effective management of the policies. We exploit the extended attributes stored in the `Extended file Attributes` (EA) of the `inode` for policy storage. As shown in Figure 1, the `inode` is connected with the resource and the set of API used for policy access. EAs are included as part of the UNIX File System Version 2 (UFS2) for FreeBSD. The extended file attributes provide a mechanism for supporting the association of various metadata with files and directories; such metadata are not directly used by the file system (unlike other attributes such as the *owner, permissions, size,* and *creation/modification times*) [14]; rather they are meant to be used by programs for associating attributes with files. However, due to the limited amount of space available in EA, the whole policy structure cannot be stored. Moreover, policy functions cannot be stored along with the objects, as no executable code can be stored at the inodes. Thus, we store the policies in a central repository and refer them from the EA through a unique id, referred to as `policyID`. We also use the EA to store the constraints for evaluating the policy identified by `policyID`.

**PAM module extension.** PAM presents a common solution for organizing multiple authentication mechanisms into a single, high-level API for authentication programs. These programs, which are usually system entry applications, like login or sshd, can use the PAM API to authenticate a user while hiding the details of the underlying authentication mechanism used. The PAM library consists of several modules, each implementing a particular authentication scheme. A system administrator uses a set of configuration files in /etc/pam.d/ to associate each system entry application with one or more PAM [9]. Although well designed, PAM modules cannot be used as they are in the Auth-SL system. This follows from the fact that our system relies on controlling not only which authentication mechanism is to be used, but also its parameters. We thus created an `authentication_context` object storing: *type of mechanism, time of authentication, number of authentication tries, threshold, TTP, storage location (local or remote), and storage mode (encrypted, plain text, etc.).* We extended

`pam_unix` module by adding code to store the `authentication_context` object. Thus we are able to retrieve, control, and record module-specific data during every authentication attempt. The problem of continuous authentication is addressed by creating a set of configuration files, each of which specifies the PAM module that provides a particular type of authentication service. The name of each configuration file reflects the underlying authentication mechanism.

**Policy enforcement and continuous authentication.** Enforcement of an authentication policy is a multi-step activity, illustrated in Figure 1. As shown, the *authentication enforcement point* is invoked by *kernel file access calls*, which have been connected with the *Unix system entry applications* through a library of APIs for context access. Specifically, policy enforcement is as follows. The authentication activity is initiated when a user initially logs in as a subject (actually a process) and then attempts to perform an operation, such as open, read, write, on an object $o$, such as a file, device, process, or socket. The operation as part of its execution requires the Authentication Enforcement Point (AEP) to perform an authentication enforcement operation. The AEP gathers the authentication context $c$ (from the context log stored in the `ucred` struct) of the subject and the policy identifier along with the parameter values stored within the extended file attributes associated with the object $o$ being requested. This is achieved by calling the function *authGetPolicy_fd()*, which returns the policy identifier, by function *authGetPolicy_Const()* which returns the constraints to be passed for the policy evaluation and by the *extractAuthCotext()* function. Once these data are gathered, the function *ContextSatisfies()*, which is the core of the enforcement activity, attempts to match the authentication context logged with the authentication factors required by the authentication policy. The policy identifier is passed as input to the function to select the policy to be enforced.

**Performance evaluation.** We have conducted several experiments to evaluate the performance of our solution. The tests were carried out on a Intel(R) Xeon(TM) 2.80GHz CPU with 1 GB of RAM. The performance of the prototype has been measured in terms of CPU time (in milliseconds). We present the results of the evaluation of the policies. Due to lack of space we report only some of the experimental results. Our testing consisted of timing the execution of policy functions to determine whether the factors have been verified or not, by looking into the context data log. For the experiments, we considered three simple policies: the first with one a single factor; the second with two factors and zero constraints; and the third with two factors and one constraint binding the two factors. Each policy is composed of two factor assertions, and refers to a password authentication mechanism. The results show that our implementation does not introduce significant latency (as by Figure 2). When policies are not satisfied, the time needed for the open command to complete is significantly reduced. This follows from the fact that the authentication check is performed prior to the application of any access control. If the required authentication factors are not satisfied, the open process terminates quickly. Hence, it is clear that the evaluation of our authentication policies do not significantly burden the system.

## 5 Related Work

Quality of authentication has been explored as authentication confidence by Ganger et al. [6]. In this approach the system remembers its confidence in each authenticated principals identity. Authorization decisions explicitly consider both the "authenticated" identity and the system confidence in that authentication. The categorization of the authentication type is based on either the possession of secrets or tokens, e.g. passwords or smartcards, or on user specific characteristics like biometrics. Such an approach however does not support a fine granular quality of authentication. We instead provide an expressive policy language supporting quality of authentication. We also provide a reference architecture for authentication services and have implemented a version of it. Our approach is thus more comprehensive and provides a fine granularity control over authentication.

Authentication policies have also been implemented in WebSphere [13] as a part of a flexible set of authentication protocols. These authentication protocols are required to determine the level of security and the type of authentication, which occur between any given client and server for each request. Compared to Websphere policies, our authentication policies are more expressive and have more efficient evaluation as they are enforced at the kernel level.

Our work has some relationship with existing work on authentication logics [1, 2, 15]. For lack of space we limit our discussion to the seminal paper by Abadi et, al. [15], which has goals close to ours. The authors propose a logic based authentication language which has been implemented in the Taos OS. A key notion is such approach is the notion of identity that includes simple principals, credentials and secure channels. The authentication system allows a weak form of continuous authentication through the "speaks-for" notion, that in practice represents subsumption among principals, and the use of authentication cache. By contrast Auth-SL supports the specification of fine-grained authentication requirements that are independent from principals. Besides simple subsumption of principals, Auth-SL supports true multi-factor authentication, enforced through a combination of authentication factors. In addition Auth-SL supports the specification of freshness requirements. Expressing our authentication mechanism in terms of authentication logics could yield to a limited characterization of Auth-SL, which would exclude interesting features such as fine grained conditions against factors and support of temporal constraints. We will further investigate possible extensions of Auth-SL with ideas from the work on authentication logics.

Operating systems define various policies for access control. In particular Security Enhanced Linux [5] (also known as SELinux) provides an expressive policy language which can be used for defining authentication policies. Differently from SELinux, provide a simple syntax which is expressive to describe the various types of authentications and the requirements. Our policies are translated to C functions which are executed at the time of the authentication check. Thus, as compared to SELinux policies, our policies are much simpler to define. Moreover, since our policies are finally encoded as C functions which are pointed to by file objects, we do not require a centralized policy enforcement as in SELinux.

## 6 Future Work

We plan to extend this work in various directions. The first direction concerns the specification of when the authentication has to be executed; such as when specific events occur, or at periodic time intervals. A second direction concerns the possibility of specifying different authentication policies for different users of the system; this extension would also require an additional component for the policy language and mechanisms for associating policies with users. Finally we plan to implement an authentication service for use by applications and federated digital identity management systems.

## References


1. Abadi, M., Burrows, M., Lampson, B.W., Plotkin, G.D.: A calculus for access control in distributed systems. ACM Trans. Program. Lang. Syst. 15(4), 706–734 (1993)
2. Abadi, M., Thau Loo, B.: Towards a declarative language and system for secure networking. In: NetDB 2007. Proceedings of the Third International Workshop on Networking Meets Databases, Cambridge, MA, USA (2007)
3. de Alfaro, L., Manna, Z.: Continuous verification by discrete reasoning. Technical Report CS-TR-94-1524 (1994)
4. v. 1.0 Extensible Markup Language (XML). W3c recommendation, 2006, `http://www.w3.org/XML/`
5. SELinux for Distributions, `http://selinux.sourceforge.net/`
6. Ganger, G.R.: Authentication confidences, pp. 169–169 (2001)
7. Klosterman, A., Ganger, G.: Secure continuous biometric-enhanced authentication (2000)
8. Pluggable Authentication Modules, `www.sun.com/software/solaris/pam/`
9. FreeBSD Project. Freebsd home page, `http://www.freebsd.org`
10. Revesz, P.Z.: Constraint databases: A survey. In: Semantics in Databases, pp. 209–246 (1995)
11. SAML. v. 1.0 specification set (2002), `http://www.oasis-open.org/committees/security/#documents`
12. RSA SecureId, `http://www.rsasecurity.com/node.asp?id=1156`
13. IBM WebSphere Software, `www-306.ibm.com/software/websphere/`
14. Watson, R.N.M.: Trustedbsd adding trusted operating system features to freebsd. In: USENIX Annual Technical Conference (2001), `http://www.usenix.org`
15. Wobber, E., Abadi, M., Burrows, M., Lampson, B.: Authentication in the taos operating system. ACM Trans. Comput. Syst. 12(1), 3–32 (1994)
16. Yang, G., Wong, D.S., Wang, H., Deng, X.: Formal analysis and systematic construction of two-factor authentication scheme (short paper). In: Ning, P., Qing, S., Li, N. (eds.) ICICS 2006. LNCS, vol. 4307, Springer, Heidelberg (2006)
17. Bertino, E., Bhargav-Spantzel, A., Squicciarini, A.C.: Policy languages for digital identity management in federation systems. In: POLICY 2006. Proceedings of Workshop on Policies for Distributed Systems and Networks, pp. 54–66 (2006)