A Cloud-based RDF Policy Engine for Assured Information Sharing

Tyrone Cadenhead, Vaibhav Khadilkar, Murat Kantarcioglu and Bhavani Thuraisingham
The University of Texas at Dallas
800 W. Campbell Road, Richardson, TX 75080
{thc071000, vvk072000, muratk, bxt043000}@utdallas.edu

ABSTRACT
In this paper, we describe a general-purpose, scalable RDF policy engine. The innovations in our work include seamless support for a diverse set of security policies enforced by a highly available and scalable policy engine designed using a cloud-based platform. Our main goal is to demonstrate how coalition agencies can share information stored in multiple formats, through the enforcement of appropriate policies.

Categories and Subject Descriptors

General Terms
Design, Security

Keywords
Assured Information Sharing, RDF Policies, Cloud Computing

1. INTRODUCTION
In today’s digital age, numerous government agencies generate and store large amounts of data. The data originates from both, private and public collections, and may be used for intelligence gathering and tracking, such as surveillance data (viz. a private source), as well as news feeds (viz. a public source). The agencies collaborate and share their data for common goals. Any improper disclosure of data could result in litigations or losses to the parties involved. To mitigate the security risks of disclosure and to support large datasets, we describe a policy engine framework that uses the Resource Description Framework (RDF) [5] to store data, as well as to define and store policies. RDF is a W3C recommendation for the World Wide Web; it can be used to provide interoperability and semantics for both, the data and the policies, which can then be stored side by side on the same platform. In addition, the advent of cloud computing and the continuing movement toward the software as a service (SaaS) paradigm has posed an increasing need for providing assured information sharing as a service in the cloud. These factors motivated us in developing a policy engine framework that achieves high availability and scalability while maintaining low setup and operation costs to each agency for sharing its resources.

2. ARCHITECTURE
We used a three-tier approach in designing our architecture (see Figure 1). At the front-end, we have a user interface; the middle layer consists of our policy engine logic; and at the backend, we have our data stores. We next define each of the layers in our architecture. Then, we present details of the current policy engines supported by our framework, and finally, we provide a description of the novel features of our implementation.

2.1 Modules in our Architecture
The User Interface Layer is exposed as a series of web browser pages. We use a form-based Authentication pattern, as well as a challenge-response test to distinguish legitimate users from robots (which may pose as normal users). The legitimate users are presented with a querying screen that allows them to compose SPARQL queries once they have been authenticated. Note that SPARQL [6] is a query language for RDF and is used for retrieving data from triple stores. The SPARQL queries are validated and then sent to our policy engine layer, which in turn returns a resultant RDF graph that is then displayed on a web page.

The Policy Engine Layer first evaluates the user queries against the stored data resources (which can be traditional data, or provenance meta data). A data resource is characterized by a uniform resource identifier (URI), which connects to an actual RDF graph in the data storage layer. The policy layer uses a factory object to create the underlying policies. The factory exposes a policy through a consistent interface, thus making it easy to extend our policy engine to support other types of policies in the future. We currently support access control, redaction, and information sharing policies. To support traditional policies, we use SPARQL queries to define views over resources where a view can be...
associated with positive and negative authorizations. Provenance records the history of a piece of data item. However, provenance takes on a directed acyclic graph (DAG) structure, and as such requires its own policies [1]. Therefore, we support the use of regular expression SPARQL queries for access control policies [2], as well as redaction policies [3]. We have also implemented sharing policies over data and provenance that allow cooperating agencies to share information based on mutual agreements.

The Data Layer makes use of a connection factory, which acts as a facade for creating a connection object. A connection object takes as input, a URI, and returns a resultant RDF graph from any of the underlying storage subsystems. The connection factory thus ensures that the policy engine layer is independent of any data storage technology.

Our policy engine framework can be used as a key enabler in augmenting security for RDBMS’s, as well as cloud-based systems. RDBMS’s are developed with atomicity, concurrency and durability in mind, but are normally shipped with limited support for access control. A cloud storage layer allows the agencies to store and scale policies with finer levels of control over RDF resources. The cloud was developed with scalability and availability in mind, but access control was neglected. Our policy engine can be configured to complement policies in a RDBMS system with an entry point for supporting security policies over cloud-based backends.

2.2 Policy Engines

A policy is defined by an interface, which allows the implementation of the logic of each policy. A policy engine takes as input, a user’s credentials and an agency’s resource (which is a URI that dereferences an agency’s resource in the data layer). It then evaluates the underlying logic of the policy before returning a new RDF graph (or model) to the user interface layer. An agency requires more than one type of policy to achieve finer levels of control over its resources.

By migrating the policies to the cloud, we remove the restriction on the number of policy definitions previously possible. The following subsections summarize various policy types.

2.2.1 Access Control Policy Engine

Our access control policy engine authorizes users to perform a set of actions on the underlying resources. Unless authorized through one or more access control policies, these users have no access to any of the resources in the data layer.

There are different kinds of access control policies, which can be grouped into three main classes [8]. These policies differ by the constraints they place on the sets of users, actions and objects (access control models often refer to resources as objects). These classes are (1) RBAC, which restricts access based on roles; (2) discretionary access control (DAC), which controls access based on the identity of the user; and (3) mandatory access control (MAC), which controls access based on mandated regulations determined by a central authority. The policy engine layer supports many policy engines; therefore, we can support an implementation of each class of access control policy, as well as any extension of a previously defined access control policy.

2.2.2 Policy Resolution policy Engine

A general form of an access control policy is a tuple of the form (user, resource, authorization), users ∈ Users,
resource ∈ Resource and authorization = \{+ve, −ve\}. A +ve authorization implies that the user can access the resource, while a −ve authorization implies that the user is denied access to the resource.

Figure 2 is a listing of RDF triples, which we will refer to as \(G\). We will use the triples in \(G\) for illustrating different combinations of authorizations next. Let \(P_{+ve}\) be a policy with a −ve authorization effect. After evaluating \(P_{+ve}\) over the input graph, \(G\), assume that the following graph, \(G_1\), is created:

\[
\text{SELECT } \vec{B} \text{ FROM NAMED uri1 FROM NAMED uri2 WHERE P,}
\]

where \(P\) is a graph pattern, \(\vec{B}\) is a tuple of variables appearing in \(P\) and \(\text{uri1}\) and \(\text{uri2}\) are URIs for two resources, \(\text{R1}\) and \(\text{R2}\). Agency 1 owns \(\text{R1}\) and Agency 2 owns \(\text{R2}\). Each of these agencies may define individual policy rules for its respective resources. We define a combined operator \(\odot\), so that a combined policy is now evaluated over \(\text{uri1} \odot \text{uri2}\). The operator \(\odot\) can be implemented as a graph operation over a RDF graph. Note that, \(\odot\), could be one of the following operators: \(\cap, \cup\) or \(\odot\); furthermore, \(\odot\) can be applied as many times to a RDF graph as desired. In order to execute the combined operator, we define a graph recursively as follows.

- \(\epsilon\) is a graph.
- The set of graphs are closed under intersection, union and set difference. Let \(G_1\) and \(G_2\) be two graphs, then \(G_1 \cup G_2\), \(G_1 \cap G_2\) and \(G_1 - G_2\) are graphs, such that if \(t \in G_1 \cup G_2\) then \(t \in G_1\) or \(t \in G_2\); if \(t \in G_1 \cap G_2\) then \(t \in G_1\) and \(t \in G_2\); or if \(t \in G_1 - G_2\) then \(t \in G_1\) and \(t \notin G_2\).

2.2.5 Provenance Policy Engine

The discussion so far ignores the relationships in a RDF graph (i.e., the history of a data item is along the directed paths formed by the triples). There are cases, however, when the relationships among the RDF triples must be taken into consideration while defining security policies. The three policy types discussed so far fail to address the cases where sensitive information is implicit in the various paths within a RDF graph. The provenance policy engines we implemented focus on definitions of policies that are tailored to the execution of access control and redaction policies over a provenance graph. The theory behind the logic of these policy engines is based on [2], which discusses an access control policy language for provenance and [3], which discusses how to perform redaction over provenance.

2.3 Policy Sequence

A protected resource could have multiple associated policies defined over the different policy types. Each policy type produces a new subgraph of its input RDF graph. Therefore, the original input graph will go through a series of transformations until a final RDF graph is returned to the user. It is important to note that the effect of a policy is directly dependent on the RDF graph it receives as input, and furthermore, the effect may be different from the original effect the policy was intended to achieve. In other words, the success of a policy rule (which is implemented as a SPARQL query) returning a particular set of RDF triples is dependent on the transformation step at which the rule was applied. Let us revisit the structure of a query again, but this time using the CONSTRUCT variation of a SPARQL query:

\[
\text{CONSTRUCT } G \text{ WHERE P,}
\]

The newly constructed graph \(G\) contains a set of triples that satisfy condition \(P\) in the input graph. For the resource in Figure 2, a policy protecting the triples, when \(P\) is
will fail if either the name or project home page triples were earlier removed or altered by a previous access control or redaction policy. These considerations motivated us to design a policy precedence feature in the framework. In the user interface layer, an agency determines the ordering of its policies, as well as the ordering of the corresponding policy rules. The policy sequence is then stored in a RDF sequence file (using the “rdfseq” feature of the RDF specification). When a query is evaluated, the policy framework will in turn invoke each policy (and corresponding policy rule) in the intended order.

2.4 Features of our Policy Engine Framework

In the subsections below, we present some novel features of our policy engine framework.

2.4.1 Policy Reciprocity

Agency 1 wishes to share its resources if Agency 2 also shares its resources with it. Current access control and redaction policies do not provide for this reciprocity. Our framework provides information sharing policies, which allow agents to define policies based on reciprocity and mutual interest amongst cooperating agencies.

We present two sample information sharing policies below:

1. $\forall r_1 \in Agency_1, \forall r_2 \in Agency_2$, use $r_1 \cup r_2$.
   This policy states that Agency 1 shares all its resources with any resource of Agency 2 as a union of the resources (i.e., $\lor \in \{\lor, \land\}$).

2. let $r_1, r_2, \ldots, r_n \in Agency_1$, use $r_1 \lor r_2, r_1 \land r_2, r_2 \in Agency_2$.
   This policy offers a finer level of control and defines the combined operator, $\lor \in \{\land, \lor\}$.

Policy Symmetry. A consequence of policy reciprocity is to have symmetry in the sharing of policies. For example, Agency 1 shares its resources with Agency 2 with a combined operator, $\lor$, if Agency 2 also shares its resources with Agency 1 using the same combined operator, $\lor$. We present a sample information sharing policy below:

1. $\forall r_1 \in Agency_1, \forall r_2 \in Agency_2$, Agency 1 uses $r_1 \lor r_2$ if Agency 2 also uses $r_2 \lor r_1$.

Conditional Policies. Another consequence of policy reciprocity is allowing the use of conditional sharing policies. For example, Agency 1 shares its resources with Agency 2 if Agency 2 does not share Agency 1’s resources with Agency 3. We present a sample information sharing policy below:

1. $\forall r_1 \in Agency_1, \forall r_2 \in Agency_2$, Agency 1 defines $r_1 \lor r_2$.
   If $\forall r_3 \in Agency_3$, then
   - Agency 2 does not define any sharing policy of the form $r_1 \lor r_3$, or
   - Agency 2 does not define any sharing policy of the form $r_1 \subseteq r_2 \lor r_3$, where $\lor \in \{\lor, \land\}$.

2.4.2 Develop and Scale Policies

Agency 1 wishes to extend its existing policies with support for constructing policies at a finer granularity. Our policy engine provides a policy interface that should be implemented by all policies; therefore, we can add newer types of policies as needed. In addition, our policy framework provides three configurations: (i) a standalone version for development and testing; (ii) a version backed by a relational database; and (iii) a cloud-based version that achieves high availability and scalability while maintaining low setup and operation costs.

Sequencing effects: Agency 1 wishes to vary the result set to a user’s query based on the user’s credentials. The policy sequence feature can be used to configure different outcomes by permuting the policies and their respective rules.

2.4.3 Justification of Resources

Agency 1 asks Agency 2 for a justification of resource R2. The current commercial access control policies are mainly designed to protect single data items while current redaction policies are designed for redacting text and images. Our policy engine allows agents to define policies over provenance; therefore, Agency 2 can provide the provenance to Agency 1, but protect it by using access control or redaction policies.

3. CONCLUSIONS

We implemented a cloud-centric RDF policy engine that provides a flexible security checkpoint for data resources. The policy engine can be used to extend traditional policies with new kinds of policies, e.g., sharing policies that allow cooperating organizations to securely share information.

4. REFERENCES