USING SPKI/SDSI FOR DISTRIBUTED MAINTENANCE OF ATTRIBUTE RELEASE POLICIES IN SHIBBOLETH•

Sidharth Nazareth
Department of Computer Science
Dartmouth College
Hanover NH 03755 USA
Sidharth.P.Nazareth.Adv03@Alum.Dartmouth.ORG

Sean Smith
Department of Computer Science
Dartmouth College
Hanover NH 03755 USA
sws@cs.dartmouth.edu

ABSTRACT

The *Shibboleth* middleware from Internet2 provides a way for users at higher-education institutions to access remote electronic content in compliance with the inter-institutional license agreements that govern such access. To protect end-user privacy, Shibboleth permits users to construct attribute release policies that control what user credentials a given content provider can obtain. However, Shibboleth leaves unspecified how to construct these policies.

To be effective, a solution needs to accommodate the typical nature of a university: a set of decentralized fiefdoms. This need argues for a *public-key infrastructure (PKI)* approach—since public-key cryptography does not require parties to agree on a secret beforehand, and parties distributed throughout the institution are unlikely to agree on anything. However, this need also argues against the strict hierarchical structure of traditional PKI—policy in different fiefdoms will be decided differently, and originate within the fiefdom, rather than from an overall root.

This paper presents our design and prototype of a system that uses the decentralized public-key framework of *Simple Public Key Infrastructure/Simple Distributed Security Infrastructure (SPKI/SDSI)* to solve this problem.

KEYWORDS

Shibboleth, SPKI/SDSI, privacy.

1. INTRODUCTION

In this wired age, one might think of delivery of licensed Web content as a relation between two entities: the individual requesting the content, and the server providing that content. Whether the server provides that content to that individual depends on what deal they arrange. However, in educational settings, many aspects of this transaction occur at the level of the institution, not of the individual. The *higher education institute (HEI)* negotiates (and often pays for) access to licensed material from a given content provider; whether an individual user can access this material depends on his or her relation to the HEI. Furthermore, it should come as no surprise that most HEIs are neither monolithic nor particularly centralized—the university fragments into schools and departments; and typically each increasingly local unit runs things differently from its peer units.

To address the basic problem of content delivery to users at HEIs, Internet2/MACE (with support from IBM) developed the *Shibboleth* system. Working within Web infrastructure and legacy authentication systems, Shibboleth permits content providers to learn, via the requesting user's HEI, whether the user has the necessary credentials to see that content.

Educational institutions value privacy, and Shibboleth respects that by hiding the user's identity from the content provider, and by letting users have *attribute release policies* that control what credentials get released to what content providers. However, the basic system does not address how HEIs, with their Byzantine

• We are grateful to Denise Anthony, David Chadwick, Carl Ellison, John Erickson, and Ed Feustel for their helpful comments. This research has been supported in part by the Mellon Foundation, NSF (CCR-0209144), AT&T/Internet2 and the Office for Domestic Preparedness, Department of Homeland Security (2000-DT-CX-K001). This paper does not necessarily reflect the views of the sponsors. Preliminary reports on this material appeared as the first author's master's thesis [21] and subsequent tech report [22].

fiefdoms, can create, maintain, and resolve these attribute release policies. For any given user, many parties may need to participate, following that user's local organization structure; but for two different users, these structures may differ.

This paper reports our use of SPKI/SDSI to design and prototype a system—SPADE—that solves this problem with Shibboleth. The existence of multiple distributed parties in Shibboleth suggests the use of PKI; decentralization and the focus on authorization suggest the use of the SPKI/SDSI PKI framework. The egalitarian nature of SPKI/SDSI allows the system to model the natural hierarchy of the HEI. This also provides a data point for two sparsely populated spaces: PKI that does not focus on *identity*, and PKI that does not focus on the standard X.509 format. Section 2 presents the basic Shibboleth system. Section 3 presents the attribute release policy problem. Section 4 presents the SPKI/SDSI framework. Section 5 and Section 6 then present our system. Section 7 reviews related work, and Section 8 suggests some avenues for future work.

2. SHIBBOLETH BACKGROUND

At its essence, education focuses on the sharing of information. In this information age, HEIs naturally have moved towards sharing information via the electronic medium of the Web; where appropriate, institutions would like to share material to students and staff at other institutions.

A standard framework would make it easier to share information—furthering the goals of education—by freeing each pair of institutions from having to develop their own scheme. Considering the scenario of a user Alice at institution I_A requesting some content from institution I_B leads to some design requirements: Institution I_B needs to know whether Alice is authorized to see the information, according to the agreement between I_B and I_A. Institution I_A probably already has its own system of authenticating whether a user is Alice; the framework should use that. Institution I_B does not need to know Alice's identity, but just whether she's authorized; institution I_A would probably rather not reveal Alice's identity to I_B. We can't expect Alice to use anything more than the desktop tools already common at I_A —e.g., a Web browser, and possibly an I_A-specific authentication tool, such as Sidecar [24]. Similarly, we would like the added infrastructure for I_A and I_B to be only a small delta from their current infrastructure, and to conform to standards.

Shibboleth [12] provides such a framework. Shibboleth is a federated administrated system that supports inter-institutional authentication and authorization for sharing of resources, available to users from those institutions. As a "middleware architecture," Shibboleth seeks to accommodate the different security systems existing in organizations and college campuses today. Shibboleth promotes interoperability by using standards, such as SAML and documented methods of information exchange; it assumes that users employ standard Web browsers to access these resources. Shibboleth places a great importance on user privacy: the content provider only knows an opaque, session-specific *handle* for Alice, not her name.

Components and Terminology: To continue with the above usage scenario, suppose user Alice at institution I_A (the *origin* site) requests resource R from institution I_B (the *target* site). Institution I_A has some legacy way to determine who Alice is. Institution I_B needs to decide whether to send resource R back to Alice.

Shibboleth addresses this problem by adding additional authorization infrastructure at both sides that exchange some additional messages. Figure 1 shows this situation. Alice's request for R lands at the *Shibboleth Indexical Reference Establisher (SHIRE)* at I_B (message M_1 in Figure 1). The SHIRE is usually a Web server, and redirects the user's request to the *Where Are You From (WAYF)* module (message M_2). The WAYF module—usually a part of the target site—interacts with the user and asks her where she is from. (The user usually has to select from a drop-down menu. In this way, the WAYF can also control which institutions have access to the target site.) The WAYF stores the mapping between the user's origin site and the URL of the user's *Handle Service (HS)*, an origin-side component that ensures the user has authenticated within I_A and that creates pseudonymous *handles* for users. Once the WAYF determines Alice's HS, it asks the HS for her handle (message M_3). The HS responds by returning the handle to SHIRE at the target site (message M_4).

Shibboleth then permits the target site to ask the origin site if the user with this handle has the necessary credentials for this resource. In Shibboleth, these credentials are termed *attributes*; Shibboleth provides for one or more *Attribute Authorities (AA)* at the origin site. (This overlap with X.509 "attribute" terminology is

unfortunate.) As part of the message M_4, the HS also tells the SHIRE the URL of the AA to use for this user. At the target site, SHIRE passes the handle, request, and AA URL to the *Shibboleth Attribute Requester (SHAR)* (message M_5 in Figure 1). SHAR contacts the user's AA and asks for this user's attributes, in an *Attribute Query Message (AQM)* (message M_6 in Figure 1). The AA retrieves the relevant attributes and returns them to the SHAR in an *Attribute Response Message (ARM)* (message M_7). After the SHAR receives the user's attributes from the AA, it sends them to the *Resource Manager (RM)* for the resource R (message M_8 in Figure 1). The RM may have its own *Attribute Acceptance Policy (AAP)*, which decides whether the user will be granted or denied access to the resource based on the attributes presented.

Shibboleth uses the *Security Assertion Markup Language (SAML)* for the AQM and ARM.

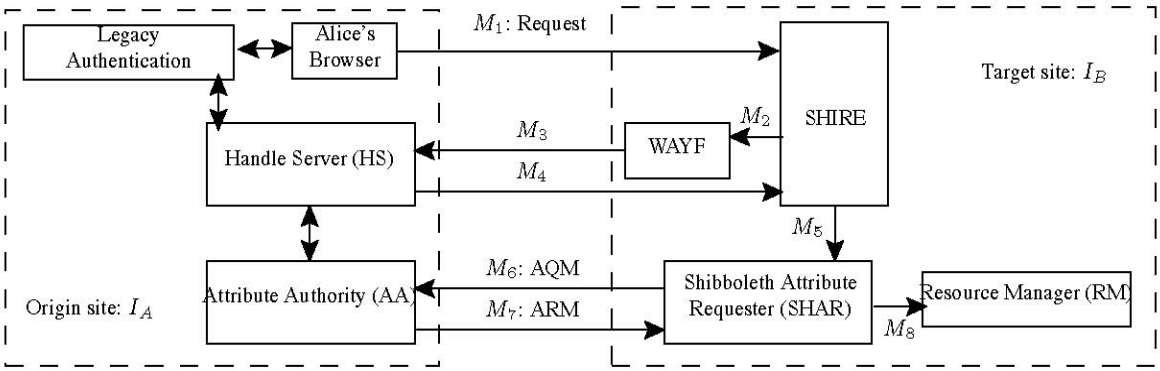

Figure 1: The basic usage scenario, with Shibboleth

3. THE ATTRIBUTE PROBLEM

User attributes are the basis for how the target site makes its authorization decisions. In Shibboleth, attributes are usually name/value pairs relevant to the user. Shibboleth also provides for *hidden attributes* relevant to the origin and target site, but not necessarily to the user. For example, Dartmouth may have a contractual agreement with the Smithsonian Institute that allows Dartmouth users to access it; thus Dartmouth must provide additional information, such as a contract number.

Shibboleth places a large emphasis on user privacy. However, Shibboleth target sites are greedy and will try to obtain as many of the user's attributes as possible. To balance access and privacy, Shibboleth allows its users a choice in what information gets released about them and to which site. To achieve this balance, Shibboleth lets each user have an *Attribute Release Policy (ARP)*. When the SHAR at the target site asks the AA at the origin site for attributes for the user with a specific handle, the AA retrieves that user's ARP and only releases attributes consistent with that policy. Shibboleth ARPs consist of a list of entries, each with three main fields: a destination SHAR name, a resource URL, and a list of attributes that can be released to this SHAR and URL (if the user has these attributes and the SHAR wants them). The SHAR is usually the Website where the URL resides and which hosts the resource. As noted earlier, the origin site might have hidden attributes, such as a contract number, and an institution-specific ARP to specify when they should be released. Thus, the AA may serve to add additional attribute values into the Attribute Response Message (ARM) sent to the SHAR. Since is not possible for a user and her institution to be able to provide ARPs for every possible target resource on the Internet, and every SHAR and URL pairing, Shibboleth permits *default* and *wildcarded ARPs*.

The Shibboleth standard requires that the AA must provide users at the origin side a means by which they can specify their Attribute Release Policies, This is usually done using a GUI such as a web browser and enables the user to control his own privacy. The downside, of course, is that a user's choice of ARP may not be proper to be able to grant him access to a target resource. Due to this problem, it is often preferable for the user to be aware of each site's attribute requirements, possibly shown on the interface.

The Problem. Shibboleth defines the basic structure and use of an ARP. However, how the AA retrieves a user's ARP is not part of the Shibboleth standard and is left open to interpretation. The user may have a single ARP or multiple ARPs. They may be dispersed throughout the organization or they may be collected in one place. How the user's ARP is retrieved, validated and enforced is left to the implementers. The

Shibboleth draft [12] states: "AA implementers are free to support many different kinds of ARPs with varying semantics as long as the AA can efficiently process requests and determine the effective policy to apply...Shibboleth doesn't specify or constrain how an AA can answer these kinds of questions."

In a typical HEI, the lines of administrative control are dispersed—e.g., the user is one place; his or her department office somewhere else; the college office yet a third place. Furthermore, the structure of this distribution will vary from user to user. In a typical HEI, it also likely that the decision procedure for attribute release will follow such administrative lines. This leaves us with questions. How do we build an ARP system that permits, for each user, the ARP to be easily resolved from the components each of these various parties introduces? How do we build an ARP system that enables each such party to easily build their local components?

4. SPKI/SDSI: LIGHTWEIGHT AUTHORIZATION PKI

To solve our distributed trust problem, the AA at the origin site needs to collect policy components created by many other parties throughout the institution. This task leads to some challenges. How does the AA verify the authenticity of these components produces by parties distributed throughout the institution? How does the AA merge these components? How does the AA even find these components?

The technology of public-key cryptography is well-suited to the first challenge—since parties distributed throughout the institution are unlikely to be able to agree on a secret beforehand (nor to agree on anything else, for that matter). To apply this technology to this problem, we looked at the tools available for *public-key infrastructure (PKI)*: "mechanisms for creating, distributing and using public keys" [19]. Typically, the main purpose of PKI is to specify how names and attributes should be bound to public keys. Usually PKI does this by using a *public key certificate*: a signed statement binding something to a public key.

Of the existing PKI tools in use today, *X.509* has emerged as the common standard. However, X.509 has a number of drawbacks. To address these, Ron Rivest and Butler Lampson at MIT designed the *Simple Distributed Security Infrastructure (SDSI)* [23] with the main goal of facilitating "the building of secure, scalable, distributed computing systems" [7]. Around the same time, Carl Ellison proposed the *Simple Public Key infrastructure (SPKI)* [11] to simplify authorization. In 1998, these two proposals merged to form *SPKI/SDSI*, to provide flexible and lightweight authorization. (Our preliminary reports [21,22] contain more discussion on the SPKI/SDSI vs X.509 tradeoffs.)

Choosing a Tool. SPKI/SDSI focuses on authorization, not authentication. In the Shibboleth model, the user is assumed to be already authenticated when we retrieve the user's relevant ARPs. However, the focus now is on the user's ARP and his authorization with respect to the target site resource. Shibboleth does not specify how the user construct his ARP or how the origin site resolves it. A SPKI/SDSI-based PKI can make use of its egalitarian design and authorization-centric approach to allow users and their HEI to specify roles in which those users can participate in, and then to follow these authorization flows to resolve the ARPs. For all the reasons, SPKI/SDSI seemed a like good fit in this area—although no one had yet examined its use in Shibboleth.

5. DESIGN

To address the attribute release problem in HEIs, we designed and prototyped *SPADE (SPKI/SDSI for Attribute Release Policies in a Distributed Environment)*. SPADE logically divides an institution or organization into *domains* that correspond to physical or structural divisions, such as project groups or academic departments. Each such domain contains a group of users, as well as an administrator who distributes and manages trust for that domain.

SPADE uses SPKI/SDSI to establish and manage trust. The main concept behind SPADE is that each party in an organization uses signed SPKI/SDSI authorization certificates to specify an ARP component (in the tag field). SPADE then acts as an extension of the Shibboleth Attribute Authority (AA). When standard requests for user attributes come into the AA, they are redirected to SPADE which contacts the relevant user domain and obtains the user's ARP component. This ARP is combined with other ARP component in the organization by obtaining a chain of certificates from the relevant domains of the organization, starting with

the individual domain user and his administrator. The ARP components (the tag fields in these certificates) are intersected to derive the final ARP. By extending the AA function in this way, this approach does not require modifying the Shibboleth standard.

Users and Roles. In a higher education institution, users typically act in roles—e.g., faculty, assistant, or student—within some organizational unit. In SPADE, we identify subjects by roles bound to their public keys, via a SPKI/SDSI name certificate created by the administrator of that user's domain.

In typical settings, what a user does within a role depends not just on that user's will, but also on local culture and policy. SPADE needs to reflect this flow by letting a user create their own ARP component, to control what attributes are released when the user acts in that role, but also by letting organizations establish ARP components that further constrain what users can do in a given role.

This organization-level restriction can apply before a user ARP component is created. For example, consider the above three roles, at a hypothetical college. The college may decide that, of these roles, only faculty and assistant should be allowed to release their credit card number to Shibboleth resources. This organization-level decision should constrain what a user is actually allowed to do when she creates her ARP in the first place (e.g. a user in the student role should not be able to create an ARP that allows release of credit card number).

Organization-level restrictions may need to apply dynamically, as well. For example, suppose Alice is faculty, and chose to allow her credit card number to be released to any of the college's business partners. Two weeks later, her college's security officer learns of credit card fraud taking place at vendor.com—but suspects that the college users—including Alice—are not aware of this fraud. The security officer needs to be able to modify its ARP to prevent any of its users' credit card numbers from going to vendor.com. Thus, even though Alice has been allowed to specify that she would like to release her credit card number to vendor.com, and she has indeed specified that, the security officer's ARP prevents the attribute from going through to the site. A week later, the fraud has been detected and taken care of and vendor.com has assured its users that it is safe to send personal information to the site again. The college now modifies its ARP again to allow all credit card numbers going to vendor.com to be released once again. In the meantime, whether Alice chooses to do so or not remains her personal choice. (This example gives only one level of organization; scenarios exist for multiple levels as well.)

Deriving Policy. SPADE derives attribute release policies by permitting each relevant party to create an authorization certificate expressing its ARP component as an S-expression in the tag field, and then combining the component policies from a chain of certificates.

We start by considering the input that an administrator (e.g., the "security officer" in the above scenario) needs to have in creating ARPs. The admin needs to express different policy for different subordinate roles. For each such role, the admin may wish to confine the attributes such a user may even choose to release to a given site, when creating his or her policy. (E.g., above, the admin originally allowed faculty to choose to release credit card numbers to vendor.com.) For each such role, the admin may wish to prevent certain attributes from being released to a given site—even if that role's policy permits that. (E.g., above, the admin wanted to temporarily prevent faculty from releasing credit card numbers to vendor.com. Right now, we must do that by removing an item from a positive list.) For each such role, the admin may wish to require that attributes specific to some agreement between that college and some given site be passed on, without bothering the user.

To express this information, the admin creates a SPKI/SDSI certificate whose issuer is the admin's public key, and whose subject is this subordinate role. Within the tag field, this certificate may contain three different lists of ARP entries (e.g., tuples of a SHAR, URL, and attribute list) for that role. One list specifies the maximum possible set of *releasable attributes (rATTRIBUTES)* that the user may choose from when constructing his or her own attribute release policy. Another list *(ATTRIBUTES)* dynamically constrains which of the rATTRIBUTES can be released at any point in time. The final list specifies *hidden attributes (hATTRIBUTES)*, which should be passed on without bothering the user. Since this certificate expresses rights and permissions for a following certificate, its propagate bit is set.

The admin can split this specification across multiple certificates: each for the same subject, but whose ARP entries (for each type of attribute) discuss different SHAR-URL pairs. If the admin partitions the specification into a certificate for rATTRIBUTES and one for the other two types, we call the former a *user template* and the latter a *filter certificate*.

An end-user wants to express what attributes she wishes to release to a given site, when she is acting in a specific role. To do this, she creates a self-signed SPKI/SDSI certificate (since she is speaking about herself),

that gives a list of ARP entries (tuples of a SHAR, URL, and rATTRIBUTE list). The rATTRIBUTE field refers to the attributes that the user chooses to release. Users typically create one ARP cert to handle all their roles. But, as with admin certs, the user can split this specification across multiple certificates whose ARP entries discuss different SHAR-URL pairs. A user's role is bound to her public key by using the user's own ARP cert, and the admin's name certificate which binds the role name to the user's public key.

SPADE provides a tool for users to create user ARP certificates, and for the Shibboleth AA to derive an ARP for a given user in a given role. Both processes involve standard SPKI/SDSI tag intersection over an authorization flow of certificates. When a user creates an ARP certificate for a role, SPADE lets the user select attributes from a list—but this list only consists of the intersection of the rATTRIBUTE lists in the relevant admin certs. When dynamically resolving an ARP for a user and role, SPADE first intersects the ATTRIBUTE lists in the relevant admin certs to derive the list of attributes the admins currently feel permissible, and then intersects that result with the rATTRIBUTE list in the user cert. SPADE also unions the hATTRIBUTES lists in the relevant admin certs to derive the list of hidden attributes that admins currently wish to always be released, and unions that with the result of the above intersection.

Attribute Values. Full Shibboleth attributes consist of a pair of *name* and *value*. Our policies discuss the attribute names only, not the values; we store the full attribute in a database at the user's domain. Thus, we preserve confidentiality even if outside users inspect the policies or certificates.

Domains. SPADE now has a way to derive policy along SPKI/SDSI authorization flows, but we want these flows to follow the decentralized organizational lines that arise in real educational institutions. To do this, SPADE divides an organization into a number of logically separable *domains*, representing sets of users within some natural organizational unit—e.g., the "History Department at Dartmouth." In our initial prototype, we assume these sets are disjoint, but the system could also work as along as each user-role pair resides in at most one domain. (Allowing joint membership is an area of future work.) SPADE also gives each domain a *domain controller* that retrieves the ARP certificates forming the authorization flow for a particular user in this domain. The SPADE *head controller* maintains contact with all the domain controllers in the HEI's SPADE infrastructure.

SPADE requires that the domains within an HEI be organized as a set of trees. Each domain has at most one predecessor and any number of successors. Within each tree, a unique *source domain* has no predecessor. A *leaf domain* has no successor. An *intermediate domain* has at least one of each. (We could extend from trees to more general directed acyclic graphs.)

Domain organization follows the hierarchical structure of the organization: the predecessor domain is the logical next higher domain in the organizational hierarchy; the successor domain is the logical next lower domain. For example, for the "Arts and Science" domain at Dartmouth, the "Dartmouth" domain is its predecessor, and the "History Department" and "Computer Science Department" are among its successors. Our ARP authorization flows follow this structure: starting with the source domain; each domain then establishes a cert that propagates ARP constraints to its successor.

Each domain contains an *administrator* and is uniquely identified by its administrator's public key. The administrator obtains the public key of the user (preferably offline) and then binds it to the user's name using a SPKI/SDSI Name certificate. (This is solely for the purpose of identifying the owner of the public key when using our SPADE GUI; is not used in authorization decisions.) The administrator creates domain roles and assigns users to them using the SPKI/SDSI mechanism for creating groups. (Thus, authorization takes place based on the user's role, in the spirit of *Role Based Access Control—RBAC*.) The administrator creates SPKI/SDSI Authorization Certificates to constrain the ARPs that SPADE derives for these roles and any successor domains. The administrator also manages the trust relationships between his domain and other domains at the HEI. (This structure is discussed further below.) A domain's administrator is responsible for correctly identifying that domain's predecessor and successors. (In fact, our prototype uses SPKI/SDSI certificates to express these links.) Each user within a domain is responsible for creating her Attribute Release Policies for different target resources in Shibboleth. This is done via a user ARP certificate, as discussed above.

Putting it All Together. When SPADE-enhanced Shibboleth receives an AQM, it starts with the user's ARP certificate and traces back the flow for that user (see below). After SPADE verifies the signatures and derives the resultant ARP, it retrieves the requested attributes—if the resultant ARP permits that—and returns them to the Attribute Authority in the usual Shibboleth way, by embedding in a SAML ARM.

Figure 2 shows the entire process.

1. The Shibboleth Attribute Requester (SHAR) at the target site contacts the Attribute Authority (AA) at the origin site for the relevant user's attributes. The SHAR passes the AA the user handle, and the SHAR and URL pair of the target resource that the user wishes to access.
2. The AA resolves the handle into the user name and public key (by using an HEI-issued X.509 Authentication Certificate from the user's browser) and passes this information on to the SPADE head controller together with the SHAR and URL values for which the attribute values were requested.
3. The head controller resolves the domain of the user and contacts the user's domain controller (DC). The head controller, in effect, hands off the processing to the user's DC and waits for the attribute values. In this example, the user is from the History Department.

Since the user specifies which of his many roles he is acting in when he logs into the system, the DC knows what role in which the user is acting and retrieves the group cert that matches that role to that user's public key The user's DC obtains the user's ARP certificate from the domain database. It also obtains the domain administrator's filter for the user's role.

4. The user's DC, after checking the administrator's database for the URL of the predecessor domain, contacts the predecessor's DC and requests its administrator's ARP certificate for this SHAR and URL. In this case, the "History Department" predecessor domain is the "Arts and Science" Domain.
5. The user's DC then obtains the "Arts and Science" predecessor domain from its database. (In this case, it's the "Dartmouth" domain.) The user's DC retrieves Dartmouth's Admin ARP for the "History Department" (and this SHAR and URL). The user's DC also stops retrieving certificates here because it knows that "Dartmouth" is this HEI's source domain. The user's DC then intersects the user's ARP, the "History Department" ARP for that user's role, the "Arts and Science" ARP for the "History Department," and the "Dartmouth" ARP for "Arts and Sciences" to form the resultant ARP. The values of the attribute names specified in the resultant ARP are pulled from the user's database.
6. The user's DC returns the user's attribute values to the SPADE head controller.
7. The head controller in turn returns the attributes to the AA.
8. The AA, now using the Shibboleth specification, bundles the attributes into SAML and sends them off to the SHAR which decides whether or not the user gets access to the target resource.

We note that purpose of SPADE is to protect user attributes from malicious target sites. The problem of protecting target sites from malicious users is orthogonal to this work.

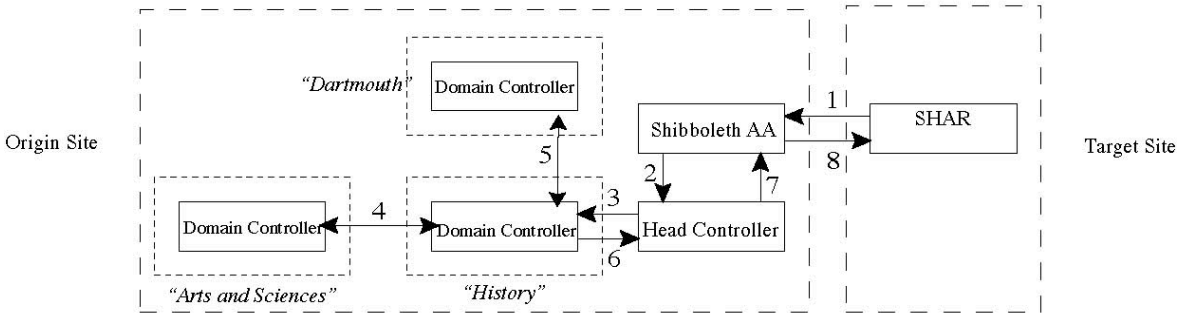

Figure 2: Example of SPADE attribute release

6. PROTOTYPE

We have prototyped SPADE, and connected it to a Shibboleth test deployment at Dartmouth. As discussed above, our SPADE prototype acts as an extension of the Shibboleth Attribute Authority (AA). We have also implemented a web-based GUI that allows the users and domain administrators to log on and manage their policies. An administrator uses the GUI to authorize new users in the system, to create roles to which users may be assigned, assign those roles, and to create admin ARP certs that constrain policies for users and subordinate domains. A user may use the GUI to create and modify her attribute release policy.

Our prototype uses Java, HTML, and Java Servlets. The code is divided into three main parts: We started with the SDSI Java code developed at MIT [8]. This code is a GUI package designed to familiarize users with SDSI operations such as creating certificates and deriving authorization decisions from certificate

chains. Starting with this base, we made a number of modifications (adding approximately 1250 lines of code). As part of this work, we stripped away the GUI code and adding helper functions in relevant classes. We also implemented a library to intersect SPKI/SDSI certificate tags, based on the SPKI/SDSI standard [11]. This library processes normal as well as special S-expressions such as the wildcarded (*), (* range), (* prefix) and (* set) formats. We also wrote helper files to process SPADE-relevant operations quickly, such as obtaining the names of all roles in a domain by resolving its authorization certificates and name certificates.
The bulk of SPADE is our approximately 4000 lines of Java servlet code. This component plugs into the SPKI/SDSI code to create objects such as public keys, name certificates, and authorization certificates and to verify certificate chains. It handles the web-based GUI: building dynamic Web pages, processing user input and interacting with the prototype databases (which are actually file system directory hierarchies). SPADE also contains a small amount of HTML code to call and process servlets. In our experimental setup, the entire code package[1] and directory hierarchy (approximately 5500 lines of code) sits on a server, where it is plugged into a Shibboleth test club implementation (Shibboleth v 0.8). Our preliminary tech report [22] shows a full series of screenshots from the prototype.

7. RELATED WORK

Shibboleth uses attribute release policy to protect user privacy; SPADE uses SPKI/SDSI to provide a decentralized way to manage them. In this section, we quickly review some other principal work in this general area.

The *Open Profiling Standard (OPS)* [15, 16] employs a "personal profile" to let Web users specify what attributes they wish to release. The *Platform for Privacy Preferences (P3P*) project [8] provides an XML-based way for content providers (and other Web entities) to describe their data collection practices, so that users can make informed privacy judgments before interacting. The area of *trust negotiation* (e.g., [25]) looks at interactive ways for client and server to negotiate release of private credentials. The *eXtensible Access Control Markup Language (XACML)* [3] is an XML-based language for expressing a user's security policies and involves authorization for resources. Lorch et al [20] have explored the XACML approach as part of a large exploration of XACML-based access control. Like XACML, the *Security Assertion Markup Language (SAML)* [13] is another XML-based OASIS initiative. SAML provides a way to exchange information about user authentication, authorization and attributes between online sites. IBM's *Enterprise Privacy Authorization Language (EPAL)* [5] is an XML-based model for "enterprise-internal privacy policies." However, unlike Shibboleth and SPADE, EPAL does not permit end subjects to manipulate policy.

The work of Blaze et al [1,2] defined *trust management* as "a unified approach to specifying and interpreting security policies, credentials, and relationships that allows direct authorization of security-critical actions." The *Distributed Trust Management System* at the University of Maryland, Baltimore County [18] uses rights and delegations with certificates to create a trust management system. This infrastructure uses X.509 certificates and Prolog policies to enforce security. IBM's *Trust Establishment Framework (TEF*) [17] maps the subject of a certificate to a role, based on the certificate and policy. Chadwick's *Privilege Management Infrastructure (PMI)* [6] looks at authorization in the same hierarchical spirit as X.509 identity: "a PMI is to authorization what a PKI is to authentication." *Ponder* [10] and the *Simple Policy Control Protocol (SPOCP)* [14] are policy models without a trust management system.

(Space forces us to be concise here. Our preliminary reports [21,22] contain fuller suveys.)

8. CONCLUSION

The principal goal of SPADE is to show the viability and effectiveness of using SPKI/SDSI as a basis for a distributed trust system for specifying and conveying policies for digital libraries—particularly in institutes of higher education where separate fiefdoms make public key technology preferable to shared secrets or central servers, but where disparate local hierarchy makes traditional PKI inappropriate. SPADE uses

[1] The current code is available for download at **http://www.cs.dartmouth.edu/~sws/sidharth_thesis/**

SPKI/SDSI certificates to delegate authority and to manage policies in an organization. SPADE allows users to define and create their own ARPs, specific to their organizational assigned role in that domain. SPADE allows other domains in the organization to exert their influence on the individual user's ARP and to thus base the final ARP on the intersection of these ARPs. As a sociologist colleague observed [4]: "SPADE is able to capture the necessary features of most of modern organization (hierarchical structure) and the benefits of hierarchy, without also capturing the costs/negative aspects of hierarchy. This is very cool."

Future work includes trying the system in pilot populations. Do users and admins understand it? Is the policy language sufficiently expressive to match real-world scenarios? What about performance? Future work also includes extending this approach to decentralized trust management to other arenas in Shibboleth (such as attribute acceptance policies, at the content provider) and elsewhere.

REFERENCES

[1] Blaze M. 1999. The Role of Trust Management in Distributed Security. *Secure Internet Programming: Issues in Distributed and Mobile Object Systems*. Springer-Verlag LNCS 1603.

[2] Blaze M., Feigenbaum J., Lacy J. 1996. Decentralized Trust Management. *IEEE Symp. on Security and Privacy*.

[3] Anderson A. et al. 2003. *OASIS eXtensible Access Control Markup Language (XACML)*. http://www.oasis-open.org/committees/download.php/1642/

[4] Anthony D. 2003. Personal communication.

[5] Ashley, P. et al, 2003. *Enterprise Privacy Authorization Language.* IBM Research. http://www.zurich.ibm.com/security/enterprise-privacy/epal/Specification/.

[6] Chadwick D. 2002. The PERMIS X.509 Role Based Privilege Management Infrastructure. *7th ACM Symposium on Access Control Models and Technologies (SACMAT 2002*), pp 135- 140.

[7] Clarke D. et al. 2001. Certificate Chain Discovery in SPKI/SDSI. *Journal of Computer Security*, 9(4): pp.285–322.

[8] Cranor L. et al. 2002. *The Platform for Privacy Preferences 1.0 (P3P1.0) Specification*. W3C Recommendation. http://www.w3.org/TR/2002/REC-P3P-20020416/ .

[9] *Cryptography and Information Security Group Research Project: SDSI*. http://theory.lcs.mit.edu/~cis/sdsi.html.

[10] Dulay N. et al. 2001. A Policy Deployment Model for the Ponder Language. *7th IFIP/IEEE International Symposium on Integrated Network Management*.

[11] Ellison C. et al. 1999. *RFC 2693: SPKI Certificate Theory*. http://www.faqs.org/rfcs/rfc2693.html

[12] Erdos M., Cantor S. 2002. *Shibboleth Architecture Draft v05*. http://shibboleth.internet2.edu/docs/draft-internet2-shibboleth-arch-v05.doc

[13] Farrell S. et al. 2002. *SAML v1.0, Assertions and Protocol*. http://www.oasis-open.org/committees/download.php/1371/

[14] Hedberg R., Wiberg T. *The SPOCP Protocol: SPOCP Project document*. ftp://ftp.su.se/pub/spocp

[15] Hensley P. et al. 1997. *Implementation of OPS Over HTTP, Version 1.0*. W3C Recommendation. http://www.w3.org/TR/NOTE-OPS-OverHTTP.html .

[16] Hensley P. et al. 1997. *Proposal for an Open Profiling Standard, Version 1.0*. W3C Recommendation. http://www.w3.org/TR/NOTE-OPS-FrameWork.html .

[17] Herzberg A. et al. 2000. Access Control Meets Public Key Infrastructure, Or: Assigning Roles to Strangers. *IEEE Symposium on Security and Privacy*. p. 2-14.

[18] Kagal L., Finin T., and Peng Y. 2001. A Framework for Distributed Trust Management. *IJCAI-01 Workshop on Autonomy, Delegation and Control.*

[19] Kaufman C., Perlman R., and Speciner M. 2002. *Network Security: Private Communication in a Public World (2nd Edition)*. Prentice Hall.

[20] Lorch, M. et al. 2003. First Experiences Using XACML for Access Control in Distributed System. *ACM Workshop on XML Security*. p. 25-27.

[21] Nazareth S. *SPADE: SPKI/SDSI for Attribute Release Policies in a Distributed Environment*. Master's thesis, Department of Computer Science, Dartmouth College, May 2003. http://www.cs.dartmouth.edu/~sws/theses/sidharth.pdf.

[22] Nazareth S, Smith S. 2004. *Using SPKI/SDSI for Distributed Maintenance of Attribute Release Policies in Shibboleth.* Computer Science Technical Report TR2004-485, Dartmouth College.

[23] Rivest, R., Lampson, B. 1996. *SDSI - A Simple Distributed Security Infrastructure.* MIT. http://theory.lcs.mit.edu/~rivest/sdsi10.html

[24] SideCar. http://www.cit.cornell.edu/kerberos/sidecar.html

[25] Winslett M. et al. 2002. Negotiating Trust on the Web. *IEEE Internet Computing* 6(6): pp 30-37