RZ 3812 (# Z1112-002) 12/08/11
Computer Science 8 pages

Research Report

Simplified Authentication and Authorization for RESTful Services in Trusted Environments

Eric Brachmann*

Dresden University of Technology, Germany
eric.brachmann@mailbox.tu-dresden.de

Gero Dittmann and Klaus-Dieter Schubert

IBM Systems & Technology Group, 71032 Böblingen, Germany
gero@ieee.org, kdschube@de.ibm.com

*Eric Brachmann was with IBM Germany Research & Development at the time of this work.

Simplified Authentication and Authorization for RESTful Services in Trusted Environments

Eric Brachmann[1] *, Gero Dittmann[2], Klaus-Dieter Schubert[2]
[1] Dresden University of Technology, Germany; eric.brachmann@mailbox.tu-dresden.de
[2] IBM Systems & Technology Group; 71032 Boeblingen, Germany; gero@ieee.org, kdschube@de.ibm.com

Abstract

In some trusted environments, such as an organization's intranet, local web services may be assumed to be trustworthy. This property can be exploited to simplify authentication and authorization protocols between resource providers and consumers, lowering the threshold for developing services and clients. Existing security solutions for RESTful services, in contrast, support *untrusted* services, a complexity-increasing capability that is not needed on an intranet with only *trusted* services.

We propose a central security service with a lean API that handles both authentication and authorization for trusted RESTful services. A user trades credentials for a token that facilitates access to services. The services may query the security service for token authenticity and roles granted to a user. The system provides fine-grained access control at the level of resources, following the role-based access control (RBAC) model. Resources are identified by their URLs, making the authorization system generic. The mapping of roles to users resides with the central security service and depends on the resource to be accessed. The mapping of permissions to roles is implemented individually by the services. We rely on secure channels and the trusted intermediaries characteristic for intranets to simplify the protocols involved and to make the security features easy to use, cutting the number of required API calls in half.

1 Introduction

Organizations usually deploy protected intranets with restricted access. This paper presents an approach to simplify security protocols within these trusted environments for a gain in agility.

Consider, for instance, engineers who commonly share large amounts of data, such as test or analysis results, across a development department. To be more productive, they often write scripts and tools for this data. Sharing these tools can increase the productivity of an entire organization. In spite of this positive impact, providing such resources is often not part of the engineers' job descriptions. A framework to facilitate and encourage resource sharing must hence provide easy access and an instant payoff for engineers in order for the system to be used and extended. Furthermore, access has to be possible using a wide variety of programming languages because developers often have strong and diverse preferences and limited flexibility in this regard.

Web services meet these requirements, as they facilitate the sharing of both data and functionality, thus supporting heterogeneity and distributed ownership. RESTful services implemented with HTTP and XML are particularly suitable because they leverage well-known technologies for which libraries are available in many programming languages. The fixed API of HTTP provides a uniform way to access all resources. Frameworks like *Ruby on Rails* render the generation of RESTful services even easier. In our context, these technologies enable developers to rapidly implement their own services and to write scripts that employ services made available by their peers.

However, access to design data must be controlled and restricted, requiring authentication and authorization of users. Although our target are services that reside within the trusted internal network of an organization, certain users may not be allowed to see, use or edit certain resources. Hence, authorization is necessary to manage access rights, and authentication is needed to reliably identify users in the first place. Incorporating such security features should not require much initial training or knowledge of complicated protocols so that it remains easy for developers to implement libraries for their preferred programming languages themselves.

From our aim to build our service system in an intranet environment we derive some assumptions that help us achieve the desired simplicity of the security framework. First, all channels are encrypted and authenticated by using the TLS/SSL protocol. Thereby we largely eliminate the possibility of eavesdropping and man-in-the-middle attacks. Second, we trust all services in the system: We suppose that services perform only legitimate actions on behalf of the user. An intranet for which these assumptions hold we call *trusted*.

However, we want to avoid storing user passwords on clients and sending them to custom-built services. Furthermore, users should only have to provide their credentials once or once every few days. After that, access to services has to be possible without entering credentials again. Therefore, the system has to offer single sign-on.

We propose a central security service with an API comprised of three methods. The first method is called by users and trades the user credentials for a security token that facilitates single sign-on to the complete service system. The token is stored on the client and sent along with every service request. The second method is called by services to validate a token and with it the authenticity of the requesting user. The third method is called by services to

*Brachmann was with IBM Germany R&D at the time of this work.

check whether the user is allowed to act in a certain role on a certain resource. This final call determines whether a user has sufficient rights for a request. We use roles to bundle permissions of groups of users and associate them with individual resources for fine-grained access control. The security service provides the necessary mapping of users to roles and resources, and offers functionality for their administration.

The remainder of this paper is organized as follows: Section 2 surveys existing web service security solutions. Section 3.1 provides an overview of the architecture of our framework with a brief description of all components. Sections 3.2 and 3.3 describe how we handle centralized authentication and authorization, respectively. In Section 4 we elaborate on how we implemented the central security service, and Section 5 concludes the paper.

2 Related Work

Security extensions for web services have been defined in the WS-Security, WS-Trust, and WS-Federation standards. The first two specify how to secure SOAP messages and how identification tokens can be attached and exchanged. WS-Federation deals with the issue of propagating identity proof between disparate security realms. These protocols add up to complex frameworks for authentication and authorization for SOAP-based web services. Similar developments for REST, in contrast, have not yet gained momentum.

Typical RESTful services exploit the HTTP protocol. The basic means of authentication in this context are HTTP Basic and HTTP Digest. They specify how to send user credentials to a service. However, to resend them with every access requires them to be stored at the client, posing a potential security risk. In fact, we want to avoid sending such sensitive information to application services altogether.

Web authentication protocols inspired by Kerberos[1], such as WebAuth[2] or the Central Authentication Service (CAS)[3], achieve this by trading credentials for tokens and transmitting them instead, providing centralized authentication and single sign-on. Construction and exchange of their tokens follow sophisticated patterns to support untrusted services. As our target services reside in a protected intranet environment, the complexity of those protocols is an unnecessary burden for service developers. Existing client libraries are designed for protecting web sites, for example, by forwarding users to login forms that cannot be used by headless clients such as automated data-processing scripts.

Web-based systems on a trusted intranet may use simpler cookie-based SSO solutions, such as the one presented in [4]. These propositions rely on a web browser for storing cookies and offer no centralized authorization management.

The Security Assertion Markup Language (SAML) [5] and the eXtensible Access Control Markup Language (XACML) [6] are often used in SOAP-based systems for authentication and authorization requests and to exchange policy information. There is, however, no established binding of these protocols for HTTP-based RESTful services. Moreover, we only need a fraction of the SAML features and, therefore, have opted for a less complex and natively RESTful protocol. Nevertheless, our architecture would also work with SAML. Our authorization approach follows the SAML philosophy with separate identity provider (IdP) and service provider (SP). Our authorization system follows the XACML architecture with a central policy decision point (PDP) and distributed policy enforcement points (PEPs). Exchanging policy information is outside the scope of this paper.

A RESTful interface for XACML Policy Decision Points to handle authorization is proposed in [7]. RESTful messages are translated to SOAP and forwarded to the central authorization component. This approach is useful for an intranet that is already equipped with a running XACML infrastructure or for exploiting the rich expressiveness of XACML for access control. Otherwise, there is little gain to justify the translation overhead introduced.

In [8], a cookie scheme is combined with role-based authorization. Session cookies stored by the client contain information about the user. These cookies are cryptographically protected by a message authentication code so that the user cannot change the information they contain. Authorization is implemented by introducing user and role object classes in LDAP. Organizations already deploying LDAP user directories need to change their existing LDAP schema accordingly, which might not be feasible.

Hecate [9] is a framework that provides centralized authorization for RESTful resources. The authors propose an XML dialect to define access rules for all available resources of the system, including the possibility of resource-aware filtering for fine-grained access control. However, service developers need to learn a new XML dialect for protecting their resources. Moreover, Hecate does not deal with authentication.

OpenID [10] is a protocol offering decentralized authentication, which has limited benefit in intranet solutions where services typically belong to the domain of a single identity provider.

Thanks to its widespread deployment, oAuth [11] has received much attention in the area of authentication and authorization. However, it covers the specific use case of granting third parties access to private user resources and hence does not fit our scenario.

Our framework offers centralized authentication with single sign-on capabilities and centralized authorization, both accessed through a RESTful services API. It exploits the properties of a trusted intranet to provide a lean protocol API and easy access to its security features.

3 Framework

3.1 Overview

Fig. 1 shows the architecture of our framework. The central components are the services themselves. They offer resources that users access for their everyday work. Access should be simple, and the creation of new services should be feasible for development engineers in addition

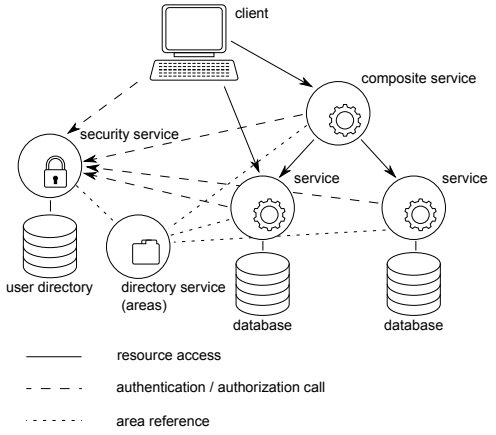


Figure 1: Architecture of the security framework. Line styles represent the different types of interaction.

to their core responsibilities. Therefore, our services follow a RESTful design. Simple services may just provide access to data records from a database and support handy, data-specific queries. Composite services combine or extend the functionality of one or more other services. Users access either type of service via a client. A client may be a web application or some other kind of graphical user interface. A client may also be a command-line tool or a custom-built script that interacts with services on behalf of the user to carry out a specific task. Authentication and authorization are managed in a centralized manner by the security service. Users interact with the security service to prove their identity. The security service is connected to the organization's user directory to check credentials. Services interact with the security service to validate the identity of a requesting user and possibly to check the user's access rights for an access-restricted resource.

In our framework authorization rules are associated with special resources called *areas*. An area may be a department, a project, or a data-specific unit. Areas serve as reference points for groups of resources in the service system. An area directory service holds these area resources and thereby provides common references for services and the security service to map role-area pairs to groups of users. The notion of areas is borrowed from the Jazz platform [12].

3.2 Authentication

Authentication is the process of proving the user's identity to the service she attempts to interact with. It ensures that only legitimate members of the organization use the service and is the foundation for authorization. For a reliable identification, a user and a service share a secret that only they know, i.e., the user's credentials. In this way, when receiving the credentials a service can be sure that the associated user is at the sending end. To relieve the services from managing user credentials and authentication logic themselves, we introduce a central authority that handles authentication for all services.

3.2.1 Tokens

In our proposed solution, this central authority is implemented by the security service (see Fig. 1). A user sends his credentials only to the security service, and only once. The security service verifies the credentials against the user directory of the organization, which stores the credentials of all employees, for example as part of LDAP user profiles. If the credentials are valid, the security service creates a token for the user. A token identifies a user for a certain period of time, e.g., several days. During this time, a user does not need to present the credentials again but presents this token instead. A security token in our framework is an opaque string that does not carry any information itself. Instead, the security service keeps a table that maps tokens to users and that also contains the expiration dates of tokens. A token contains 20 random letters and digits to prevent a brute-force attack from guessing it. Our system would also work with cryptographic tokens, e.g., to detect manipulation attempts.

In contrast to other systems like CAS, our tokens are not bound to individual services. The CAS protocol [3] describes several types of tokens (called *tickets* there). Most important are ticket-granting tickets and service tickets. Ticket-granting tickets are stored in a cookie on the client. The client sends them to the CAS server to obtain service tickets. The client then attaches the service ticket to a service request. The service checks the user's authenticity by validating the service ticket with the CAS server. Service tickets can only be validated once and are bound to one service. Tickets that were eavesdropped by attackers are therefore useless and pose no security risk.

We, however, prevent eavesdropping by using secure channels. In our case, the restrictions that CAS places on service tickets are an unnecessary burden. If security tokens expire after the first validation, users have to request new tokens for every service access. Our users should have to request their security token only once. If tokens are bound to services, composite services cannot forward them to other services without additional communication with the security service. As we trust our services to be uncorrupted, we want to allow them to reuse tokens on behalf of the user for simplicity of the protocol. In this way, a composite service can access other services on behalf of the user without requesting new tokens. The receiving service will process the request as if it had come directly from the user.

3.2.2 Accessing a Service

All services in our framework require that requests be augmented with a security token identifying the requesting user. Access without a token is not permitted and returns an error (see Fig. 2). Therefore, a user without a valid token who wishes to use a service first performs an authentication call to the security service, providing her credentials. She receives a token in return that may be reused for several days until it expires. Now, to access a service, she passes this token along with the HTTPS call, for example as an additional GET parameter or as an HTTPS header.

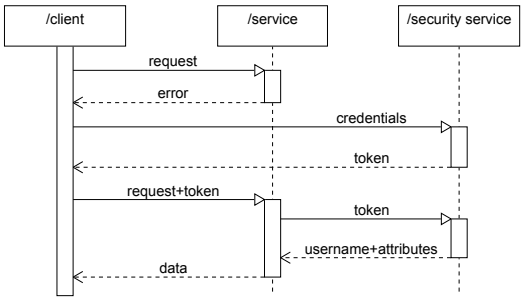


Figure 2: Authentication protocol. A service denies an unauthenticated request. The user performs authentication by trading his credentials for a security token. The authenticated request is granted.

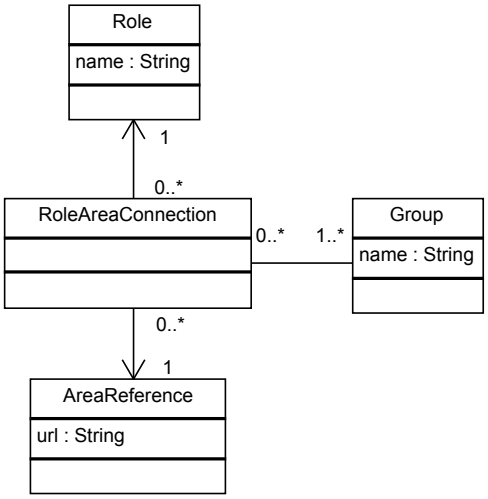


Figure 3: Authorization classes. Group objects hold authorized users for unique role-area pairs.

Upon receiving a request, a service checks whether a token is provided. It extracts the token and sends it to the security service for validation. A token is valid if there is a corresponding entry in the token table and its expiration date has not passed. The security service responds with the user name associated with the token, possibly together with additional user attributes that might be of use in the domain of the service system. The security service responds with an error code if the token cannot be found in the token table or if it has expired. Upon receiving a positive answer, the original service processes the user request after checking the user's authorization if necessary (see Section 3.3). Once the service has verified the identity of the user, it may establish a session with this user to prevent unnecessary authentication calls during further communication.

3.3 Authorization

Although the identity of a user has already been verified in the preceding authentication steps, a service might still have to decline a request if the resource to be accessed is sensitive and the user lacks sufficient clearance. We manage the required authorization at the resource level and integrate it with our central security service to relieve services of the burden of managing user groups and associating them with authorization rules. This central authorization corresponds to the concept of a policy decision point as defined by the XACML standard [6]. Based on the decision of the security service, the services controlling a resource grant or deny access, acting as policy enforcement points.

3.3.1 User Groups, Roles and Areas

The security service manages user groups and roles. According to the role-based access control (RBAC) model [13], a role represents a responsibility in the context of an organization. User groups, on the other hand, often reflect the structure of an organization. We exploit existing user groups by assigning roles to entire groups rather than individual users, leaving user management with the group owners rather than duplicating it for roles. Our role definitions are generic and can hence be reused by multiple services for multiple resources. We make the mapping of roles to user groups dependent on the resources to ensure that the same generic role can be associated with different users for different resources. For instance, the role "admin" might require membership in the group "management" for resource "A", but membership in the group "engineers" for resource "B".

In contrast to this centralized portion of the RBAC model, the mapping of roles to permissions stays with each individual service. Permissions denote rights to perform particular actions[1] that often depend on the service and the particular resource. It is hence insufficient to grant or block access to a service or a function as a whole. A user might be entitled to use a service to manipulate resource "A" but not to do the same with resource "B". However, when services provide access to large amounts of data it is inconvenient to manage authorization rules for every single resource.

We therefore have a directory service that provides a representation of the logical structure of our organization in the form of areas (see Fig. 1). Areas are resources that serve as authorization reference anchors. Our services associate each resource they provide with an area, and authorization rules also refer to areas. In this way, the area directory serves as a common dictionary for services and the security service to map resources to authorization rules. It does not have to implement any further functionality, but may store meta-information with the areas.

In fact, any REST service and even multiple services might be used as area directory as long as reference resources, i.e., areas, are provided. The difference between arbitrary resources and areas is of a semantic nature. Any resource that is being referenced by authorization rules becomes an area, and any service that holds such resources becomes an area directory. The security service reflects this flexibility by representing areas with URLs that can point to any REST resource. Thereby, regarding authorization, we support heterogeneous organizational structures.

[1]Note that although we deal with RESTful services, permissions are not necessarily limited to CRUD operations.

Each service in our system knows to which areas its resources belong. The security service, in turn, maintains area-specific mappings of roles to user groups. To perform a given action, a user must be assigned particular roles. For example, instructing a service to delete data records might require the role "admin", whereas merely reading those same records requires the role "consumer". The security service implements the authorization mapping by managing lists of roles, user groups and areas (see Fig. 3). Areas are represented by their URL, pointing to the area directory that holds the area in question. Roles are represented only by name; they do not carry any further information. The security service is only responsible for answering the question whether a user has a certain role in an area; it does not specify what a user in a certain role is allowed to do as these permissions are service-specific. The services requiring authorization are themselves responsible to determine the level of access a role grants. This arrangement has the advantage that service owners don't require the assistance of a security-service administrator to define or change the mapping of role-area pairs to permissions.

For each level of access we want to distinguish for an area, we define an appropriate role and establish a relationship to the area reference via a role-area connection (see Fig. 3). This connection contains the information which groups of users may act on the area in that role. By holding the user group information in the role-area connection objects, we facilitate the definition of generic, reusable roles.

Without loss of generality we restrict our discussion to *core RBAC* while our approach may be extended to also support hierarchical roles and separation-of-duty constraints.

3.3.2 Accessing a Resource

When receiving a request, a service determines the roles required to perform the requested action. It also determines the area the accessed resource belongs to. The service then sends the security token of the user, the URL of the area, and the role that matches the requested access to the security service. The security service checks whether the user is a member of all the groups that have been specified for that particular area-role combination. If the user is lacking a group membership or the role or the area cannot be found in the first place, the security service will respond with an error code. In this case, the original service will deny access. If the user is a member of all the necessary groups, the security service will signal success and access will be granted.

Fig. 4 gives an example of the complete authorization process. Fig. 4a shows the three services involved and some of the data they hold. The area directory reflects the structure of an organization with two departments. "Department X" runs two projects named "project foo" and "project bar". A service offers a "unit test report" resource that belongs to "project foo" and an "integration test report" resource that belongs to "project bar". For each of the four standard REST operations, the service contains a mapping to the role that is necessary to perform that operation on the report. The security service knows for every area, e.g., "project foo", which groups a user has to be member of to act in a certain role on that area.

Fig. 4b shows how a user request is processed. The user wants to read the "unit test report". The client sends this request and the user's token to the service. The service determines that the "unit test report" belongs to the area "project foo" and that the role "consumer" is necessary to read that report. It sends a query to the security service whether the user is a "consumer" of resources associated with "project foo". The security service knows that a user has to be a member of the groups "designer" and "team alpha" to act in the role "consumer" on "project foo". If the user is a member of both groups, the security service will approve the query, and the service will send the "unit test report" to the client.

4 Prototype

4.1 Implementation

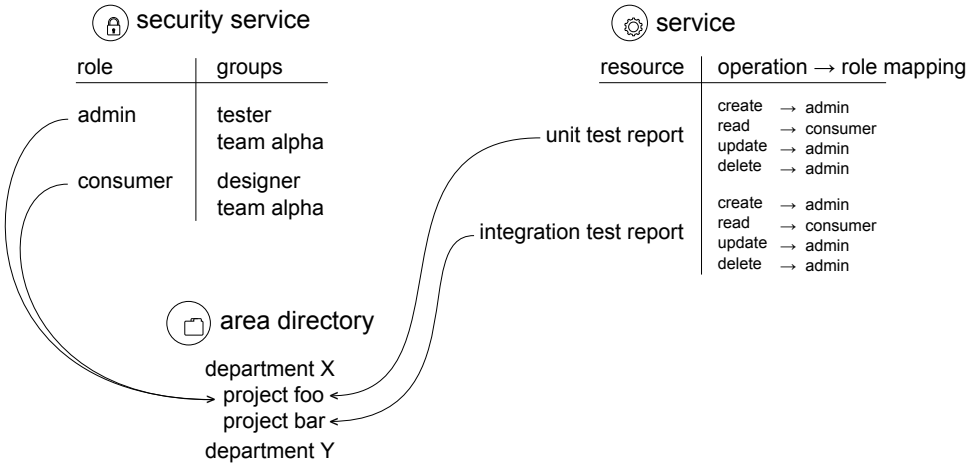
We have implemented a prototype of the proposed framework in Ruby on Rails 3.0.3. For service deployment, we use a Phusion Passenger module with enabled TLS/SSL protection on an Apache web server.

Some of the available single sign-on (SSO) solutions come with ready-to-use implementations. None of these solutions meet all of our requirements, but we deploy one of them internally for the generation and the management of security tokens as well as the connection to our user directory. We picked the Central Authentication Service (CAS) because it uses opaque strings as tokens without any cryptographic overhead, which we do not need. Furthermore, CAS implementations are available for many programming languages including Ruby, namely rubyCAS [14]. RubyCAS directly supports LDAP user directories. However, any CAS implementation should work.

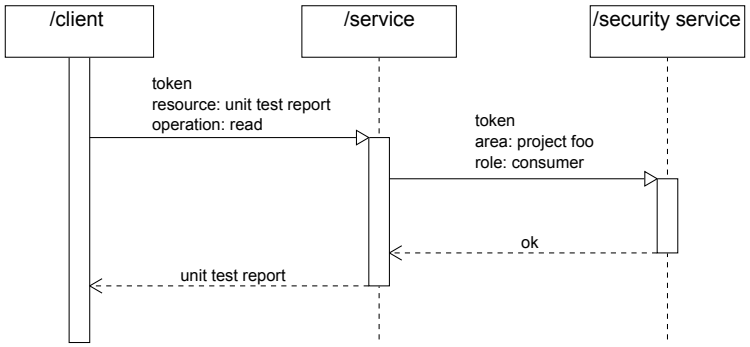
Note that no service except the security service communicates directly with the CAS server. The security service with its lean API wraps the CAS server completely, making the latter an internal component of the former.

We use CAS ticket-granting tickets as security tokens as they do not expire after validations and are not bound to services. The CAS protocol does not include the direct validation of ticket-granting tickets and, consequently, this feature is missing in CAS implementations. Our validation process uses the ticket-granting ticket to request an auxiliary service ticket and validates that in a second internal call. This mechanism is encapsulated within the security service, whereas CAS service tickets would require users to issue two calls.

As discussed in Section 3.3, the security service contains a mapping of user groups to roles and areas for authorization. Infrastructure for managing user groups is typically already available in an organizations' intranet, usually as part of a user directory. It would therefore be redundant to implement groups as lists of users in the security service. It is more convenient to reuse the group infrastructure or even preexisting groups of the user directory. Our RoleAreaConnection object (see Fig. 3) is associated with



(a) Example data held by the services involved. Arrows between the services and the directory represent references.



(b) Protocol sequence with example parameters.

Figure 4: Authorization example.

a list of LDAP groups from our organization's user directory. When an authorization request for a particular user reaches the security service, the latter connects to the user directory and retrieves all groups this user is member of. The security service then compares these user directory groups with the list of group names associated with the requested role and area. If the user is a member of all necessary groups, access is granted. We decided to require the user to be member of *all* listed groups to facilitate the formation of intersections of groups. Thereby, we are able to bind access rights to user sets below the granularity of groups. This is not possible if membership in only *one* of the listed groups would suffice. For additional flexibility both schemes could be combined.

4.2 API

Our overall approach results in the following API for the security service:

issue_token This method is called by a client `POST`ing the credentials of a user to the security service. Credentials consist of user name and password and are sent as `POST` data. If the user is found in the user directory and the password matches, the security service responds with the HTTP status `200 OK` with the security token contained in the HTTP body. If the credentials are invalid, the security service responds with the HTTP code `401 Unauthorized`.

check_token This method is called by services checking the identity of a user who provided a security token along with her request. The method is called by an HTTP `GET`, with the security token passed as a parameter within the query string. If the token is found in the token table and has not expired, the security service responds with the HTTP code `200 OK`. The response body contains an XML document with the corresponding user name and the associated list of group memberships found in the user directory. If the token is invalid, the security service responds with the HTTP code `401 Unauthorized`.

check_authorization This method is called by services if a user requests access-controlled resources that may only be seen by users in a certain role. The method has three parameters: the security token of the requesting user, the area the requested resource belongs to, and the role matching the access. The method is called by an HTTP `GET` along with all three parameters. If the security token is invalid, the security service responds with the HTTP code `401 Unauthorized`. If role or area cannot be found, the security service responds with the HTTP code `404 Not Found`. This indicates that administrative action is necessary to create a authorization mapping for the service. If the mapping is found but the user is not a member of all required groups, the security service responds with the HTTP code `403 Forbidden`. If the user has all required memberships, the security service responds with `200 OK`.

Table 1: Protocol comparison of CAS and our framework. For an explanation of the CAS tickets, see [3].

Step	CAS	Our Framework
Login	2 calls (get login ticket, get ticket-granting ticket)	1 call (get token)
Service Access	2 calls (get service ticket, access with service ticket)	1 call (access with token)
Proxy Access	2 calls + 1 passive call (get proxy ticket, access with proxy ticket + receive proxy-granting ticket)	1 call (access with token)

The security service also provides an API for creating all necessary authorization mapping objects (see Fig. 3). This API follows a RESTful design, i.e., the individual objects are created, read, updated or deleted by using HTTP `POST`, `GET`, `PUT` or `DELETE`, in some cases with an XML representation of the resource in the HTTP body. Note that this API is not used by services or clients, but is accessed only by administrators. Writing one central, convenient interface should suffice. We have implemented a web interface for this purpose, using Ruby on Rails.

Table 1 compares the protocol API complexity of CAS and of our proposed framework in terms of HTTP calls to the central security service. It lists three use cases: A user gains access to the service system in the *Login* case. She sends a request to a service in the *Service Access* case, and a service sends a requests to another service in the *Proxy Access* case. Note that this comparison does not consider the authorization aspect, which is not supported by CAS. In our framework, a service can verify user authorization with one call (see *check_authorization*).

Compared with existing systems like CAS, our framework does not require complex request-response interactions. A user can perform any security-related action with one simple HTTP call. This makes it very easy to access our framework from any programming language with support for HTTP and XML.

4.3 Deployment

We used our framework to implement a set of services for an internationally distributed engineering department developing high-performance processor chips. The services are accessed via a web front-end or by command-line scripts written in Python. The web server redirects an unauthenticated web client to a log-in page. For shell scripts, we wrote a tool that asks the user for credentials, trades them for a token with the security service, and stores the token in a shell environment variable from where service clients can pick it up. This mechanism provides single sign-on from a shell and enables automatically scheduled scripts that cannot ask any user for credentials: They can run with a token a user fetched beforehand.

We also made available a Ruby library for service developers who build upon the Ruby on Rails framework. This library implements the communication with the security service and supports queries to other services within the

framework. It can also help in the creation of Ruby client scripts. A separate Python library facilitates the creation of Python client scripts.

5 Conclusion

In this paper we have proposed a security framework for RESTful services deployed in the protected domain of an intranet. A central security service handles authentication and authorization for the services in the system, relieving services of the burden of managing users, groups and authorization rules. Because the services are trusted and run within the confines of a secure environment, we have been able to simplify the API for authentication and authorization.

Users authenticate by trading their credentials for a security token which identifies a user until it expires. The token is attached to each service call. Upon receiving a token, a service validates it with the security service. It can also check whether the user has been granted a particular role for the requested resource. The security service contains a mapping of user groups to roles, and dedicated resources called *areas*. Services grant access if the user has been granted the necessary role for the requested area. As all services are trusted, the framework enables them to use the user token to access other services on behalf of the user, greatly simplifying service composition.

We have presented an implementation of our security service using available software libraries. The prototype enables engineers to consume and create access-controlled services for improved collaboration in support of their work. As the framework is composed of simple HTTP-based RESTful services and the presented protocols cut the number of required API calls in half, we have lowered the barrier for engineers to include security features into their custom-built scripts and services, requiring only little training.

6 Acknowledgements

The authors wish to thank Chris Giblin, Olaf Zimmermann and Charlotte Bolliger of IBM Research for their significant help in improving the original manuscript.

References

[1] MIT. Kerberos: The network authentication protocol. `http://web.mit.edu/kerberos/`.

[2] Roland Schemers and Russ Allbery. WebAuth technical specification. `http://webauth.stanford.edu/protocol.html`.

[3] Drew Mazurek. CAS protocol. `http://www.jasig.org/cas/protocol`, May 2005.

[4] Vipin Samar. Single sign-on using cookies for web applications. In *Proceedings of the 8th Intl. Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE '99)*, pages 158–163, Stanford, CA, USA, June 1999. IEEE.

[5] OASIS. SAML specifications. `http://saml.xml.org/saml-specifications`.

[6] OASIS. OASIS eXtensible Access Control Markup Language (XACML) TC. `http://www.oasis-open.org/committees/xacml/`.

[7] Qublai Khan Ali Mirza. Restful implementation of authorization mechanisms. In *Proceedings of the International Conference on Technology and Business Management (ICTBM-11)*, pages 1001–1010, Dubai, UAE, March 2011. INFOMS.

[8] Kurt Gutzmann. Access control and session management in the HTTP environment. *IEEE Internet Computing*, 5:26–35, 2001.

[9] Sebastian Graf, Vyacheslav Zholudev, Lukas Lewandowski, and Marcel Waldvogel. Hecate, managing authorization with RESTful XML. In *Proceedings of the 2nd International Workshop on RESTful Design (WS-REST '11)*, pages 51–58, Hyderabad, India, March 2011. ACM.

[10] David Recordon and Drummond Reed. OpenID 2.0: A platform for user-centric identity management. In *Proceedings of the 2nd Workshop on Digital Identity Management (DIM '06)*, pages 11–16, Fairfax, Virginia, USA, November 2006. ACM.

[11] Eran Hammer-Lahav. The OAuth 1.0 protocol. RFC 5849, IETF, April 2010.

[12] Jazz community. Jazz. `https://jazz.net/`.

[13] American national standard for information technology – Role based access control. ANSI INCITS 359-2004, ANSI, February 2004.

[14] Matt Zukowski. RubyCAS-Server. `http://code.google.com/p/rubycas-server/`.