Design and Implementation of User-Managed Access Framework for Web 2.0 Applications

Maciej P. Machulak
m.p.machulak@ncl.ac.uk

Łukasz Moreń
lukasz.moren@ncl.ac.uk

Aad van Moorsel
aad.vanmoorsel@ncl.ac.uk

School of Computing Science
Newcastle University
Newcastle upon Tyne, UK

ABSTRACT

Web 2.0 applications allow individuals to manage their content online and to share it with other users and services on the Web. Such sharing requires access control to be put in place. Existing access control solutions, however, are unsatisfactory as they do not offer the functionality that users need in the open and user-driven Web environment. Additionally, such solutions are often custom-built and require substantial development effort, or use existing frameworks that provide benefits to developers only.

New proposals such as User-Managed Access (UMA) show a promising solution to authorization for Web 2.0 applications. UMA puts the end user in charge of assigning access rights to Web resources. It allows users to share data more selectively using centralized authorization systems which make access decisions based on user instructions. In this paper, we present the UMA/j framework which implements the UMA protocol and allows users of Web applications to use their preferred authorization mechanisms. It also supports developers in building access control for their Web 2.0 applications by providing ready-to-use components that can be integrated with minimum effort.

Categories and Subject Descriptors

D.2.11 [**Software**]: Software Architectures—*Service-oriented Architecture (SOA)*

General Terms

Security, Design

Keywords

access control, security, Web applications, middleware

1. INTRODUCTION

Web 2.0 technologies have enabled applications to provide a rich and dynamic user experience and expanded functionality. Individuals use such applications as Facebook and Flickr to manage their data online and to share it with other users and services on the Web. To maintain privacy and ensure that data is shared in a secure fashion, access control and authorization mechanisms need to be put in place and integrated with Web applications.

Most existing authorization solutions, however, are ill-suited to the user-driven Web 2.0 environment, as they are tightly bound to applications and have limited flexibility in terms of their configuration or adaptation to a particular user's security requirements [15] [17]. For example, existing applications have different and non-reusable sharing options or do not even allow to share data selectively outside of the application itself.

Moreover, access control mechanisms are often custom-built by software developers for each application separately. This requires significant effort, tends to be error prone, and results in business logic being mixed with access control functionality. Existing authorization frameworks alleviate this problem only partially since they require developers to define static policies in local configuration files without providing flexible and usable access control for the end user.

Following the highly collaborative Web 2.0 paradigm, there is a clear need for new approaches to access management where the user plays a central role in the model. Such approaches would allow a user to be in full control of assigning access rights to their Web resources while retaining all the benefits of social interactions and data sharing that Web 2.0 provides. Moreover, these mechanisms would not require significant effort from software developers but would rely on ready-to-use and loosely coupled components.

The recently proposed User-Managed Access (UMA) protocol [6] is a new proposal to access control for the open Web. UMA puts the user in charge of assigning access rights to resources that may be hosted at various Web applications. UMA facilitates the ability of users to share data more selectively using centralized systems, called authorization managers, which make access decisions based on user instructions. Web applications do not have to implement access control, but they delegate it to these systems, concerning themselves only with enforcing access control decisions.

In this paper we discuss UMA/j, the first known framework to provide Web applications with support for the UMA protocol. Our solution allows Web applications to delegate access control to users' preferred authorization managers. Therefore, these users can protect their online resources with their preferred access control mechanisms. By putting the user in charge of managing the delegation process, UMA/j

MW4SOC '10, November 29, 2010, Bangalore, India

fits precisely into the user-driven Web 2.0 environment.

UMA/j differs significantly from existing proposals to access control for the Web. It supports establishing relationships between parties of the UMA protocol dynamically. It also does not rely on access control policies defined by developers in local configuration files at Web applications. Rather, policies are managed by the users of these applications whom the policies directly benefit. Moreover, UMA/j supports dynamic resource registration allowing users to define which of their resources are meant to be protected in due course. As such, it supports access control delegation in a flexible and entirely user-driven manner.

2. PROBLEM STATEMENT

Users create and manage their data using Web 2.0 applications and disseminate and share this data with other users and services on the Web. Despite the fact that users now play a pivotal role in various online interactions, they are not sufficiently empowered to control access to their Web resources. This is mostly due to applications implementing simple isolated authorization models that are ill-suited for the open Web and fail to provide the required level of flexibility and simplicity, such as that discussed in [18].

Existing authorization systems based on such models require end users to manage their access control using different management tools provided by Web applications which results in an inconsistent user experience (UX). Moreover, underlying policies often differ and are incompatible with each other, thus they cannot be easily reused for distributed Web resources. For example, a user cannot create a single policy to share data with the same set of family members and apply this policy to photos at Facebook and Flickr.

Moreover, access control is implemented individually at each Web application, thus users need to manage authorization settings separately and are required to traverse Web applications to configure these settings. This problem also applies to auditing and monitoring sharing and limits the view of the applied security over distributed resources [12].

Access control solutions tend to be problematic from the perspective of Web application development as well. Existing solutions are either custom-built or are based on such frameworks as ESAPI [4], Spring Security [5] or OpenSSO [3]. Custom-built solutions require tight coupling of access control with the application itself, resulting in business logic being mixed with authorization functionality. Systems based on authorization frameworks, despite allowing developers to reuse provided components, require substantial effort to expose access control (in order to provide sharing functionality) to end users. Moreover, developed user interfaces (UI) often satisfy requirements of a particular group of users only. For example, a UI with various sharing options may be suitable for advanced users but too complex for others.

Proposals for delegated authorization, such as those based on XACML [8], which is well-supported by various implementations, allow to use a centralized system for access control to distributed resources. Such proposals, however, do not fully alleviate the problem of access control for the open and user-driven Web 2.0 environment. These proposals are mainly targeted at internally deployed Web systems with well-defined points of control. They require relationships between applications of the system to be configured by developers and do not grant the end user full control over assigning access rights to their resources. For example, in OpenSSO, applications have to be integrated with a central authorization server during their deployment and it is the policy administrator and not the end user who is responsible for defining access rights to Web resources.

3. USER-MANAGED ACCESS

User-Managed Access proposes a solution to authorization for Web resources that responds to security and privacy challenges in the Web 2.0 environment. It provides a method for users to control third-party application access to their protected resources, residing on any number of host sites, through a centralized authorization manager that makes access decisions based on user instructions. This allows a user to choose the mechanisms that provide the exact functionality or UX that a user requires.

3.1 Architecture

UMA has four main entities: Authorizing User, Authorization Manager, Host, and Requester (Fig. 1). An *Authorizing User* delegates access control from their chosen set of Hosts to an Authorization Manager (AM) and composes access control policies at the AM. An *Authorization Manager* acts on behalf of that user and evaluates access requests made by Requesters against applicable policies, issuing Access Tokens necessary to make authorized access requests to Protected Resources. A *Host* is a Web application that is used by an Authorizing User to store and manage *Protected Resources*. It delegates access control to AM. A *Requester* is an application, controlled by a person or a company (Requesting Party), that interacts with a Host to get access to a Protected Resource, which can be accomplished after it interacts with the AM to obtain an Access Token.

For example, a Web user (Authorizing User) can authorize a Web application (Requester) to gain one-time or ongoing access to photos stored at Facebook and Flickr (Hosts), by telling these hosts to act on access decisions made by his authorization decision-making service (Authorization Manager). The requesting party might be an e-commerce company whose site is acting on behalf of the user himself to print selected photos, or it might be his friend who is using an online photo editing service to retouch photos. We refer the reader to [7] for other UMA use case scenarios.

3.2 Delegation Protocol

The UMA protocol describes interactions between the previously defined entities and consists of the following steps: (1) User introduces host to AM, (2) Requester gets access token from AM, (3) Requester wields access token at host to gain access (Fig. 1). For the sake of completeness, we also describe the dynamic binding step which allows UMA interactions to occur between Web applications that do not have any pre-established relationships. We also discuss the dynamic resource registration step which currently constitutes an optional step of the UMA protocol. A more detailed description of the protocol can be found in [6] and [16].

3.2.1 User introduces host to AM

First, a user establishes a trust relationship between a host and AM. A user provides the location of a preferred AM to a host. A host then uses the site-meta [20] and host-meta [14] discovery mechanisms to obtain a metadata document from the AM defining the location of various endpoints that this

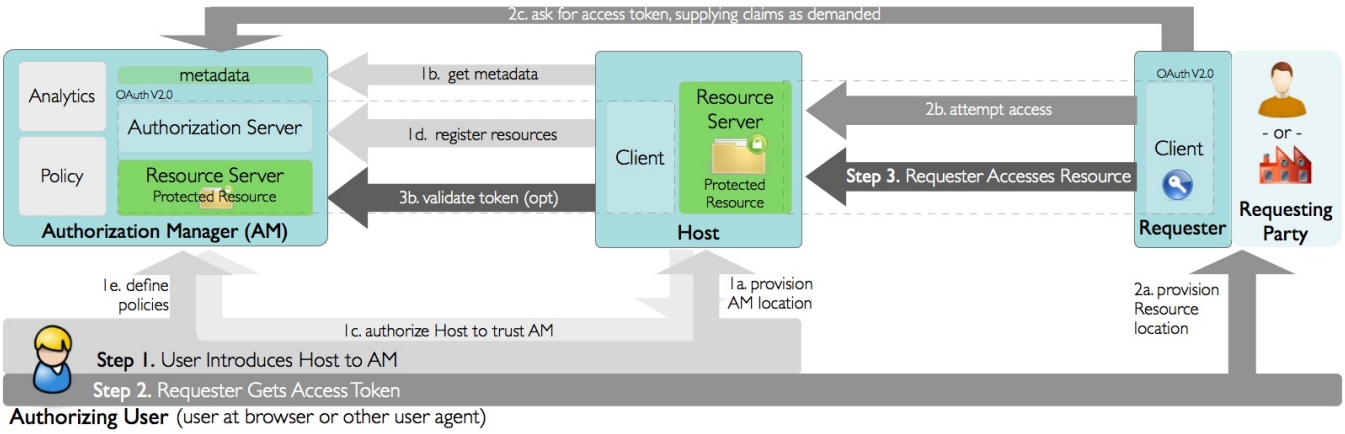

Figure 1: High-level overview of the User-Managed Access protocol [16].

AM exposes, among other information such as supported access token types [6].

If the user's chosen AM is unknown to the host, then this host registers itself dynamically with the AM by providing necessary information about itself, using the push or pull host registration model [21]. The host is provisioned in this fashion with OAuth client credentials $\{cid, csecret\}$ that are later used to interact with this AM, regardless of the user in question.

The host, acting as the OAuth client, then uses the OAuth V2.0 Web Server Flow to obtain the user's authorization to use this AM. This step results in the host being provisioned with the access token AT_{host} and an optional refresh token RT_{host}. AT_{host} allows the host to use the AM's host-specific endpoints described later in this section. If this token expires, then the RT_{host} is used to obtain a new AT_{host}.

User registers resources at AM

A host may delegate access control to the AM for all resources of a particular user, for groups of resources or for individual resources only. The protocol itself allows for arbitrary granularity levels of access control delegation.

A user on a host can select resources and groups of resources that are meant to be policy-protected at the AM. A host then sends information about these resources to the AM's resource registration URL wielding its own AT_{host}.

3.2.2 *Requester gets access token from AM*

A requester can access a protected resource on a host only if the access request is accompanied by an access token AT_{req}. If AT_{req} is missing then a host responds with "HTTP 401 Unauthorized" response that signals how to obtain such a token from the AM protecting this resource.

A requester can ask for AT_{req} by providing information to the AM's access token URL about the required type of access (method and scope) being sought at a host. The AM evaluates such a request and responds with a successful access response, an unsuccessful access response, or a claims-requested document. The first one contains the AT_{req} and an optional refresh token (RT_{req}) while the second one denies authorization. The claims-requested document contains a list of properties that a requester must prove to posses before access to a resource can be granted [6].

3.2.3 *Requester wields access token at host to gain access*

Once the requester has AT_{req} it can issue an access request to a protected resource wielding this token. The host evaluates AT_{req} locally or may use the registered AM for this resource at run time for the evaluation process. In the latter case, the host sends AT_{req} to the AM's token validation URL wielding its own AT_{host} and awaits decision whether access to the resource should be granted or not.

4. UMA/J FRAMEWORK

UMA/j is the first known framework to provide Web applications with support for the User-Managed Access protocol, with a few extensions proposed by the authors. This framework can be integrated with new and existing Web applications to allow these applications to become UMA-compliant hosts capable of delegating access control to users' chosen authorization managers.

4.1 Design

UMA/j consists of the following modules: Discovery, Dynamic Registration, Host Introduction, Resource Registration, Token Validation, Request Filter and Core modules (Fig. 2). The Web application can use the framework through its *UMA API* exposed by the Core module which hides complexities of lower-level APIs of individual components. The application, however, can use low-level APIs as well in case more fine-grained control over authorization delegation is required.

4.1.1 *Discovery*

The discovery module implements the site-meta [20] and host-meta [14] discovery, which allows the application to obtain information about services provided by a user's chosen AM. It consumes the XRD document [10] that describes the locations of these services (endpoints) and the AM's supported access token formats, among other information. Acquired information is later consumed by the Core module and is available to other UMA/j components.

4.1.2 *Dynamic Registration*

The dynamic registration module allows the application to receive required OAuth client credentials $\{cid, csecret\}$

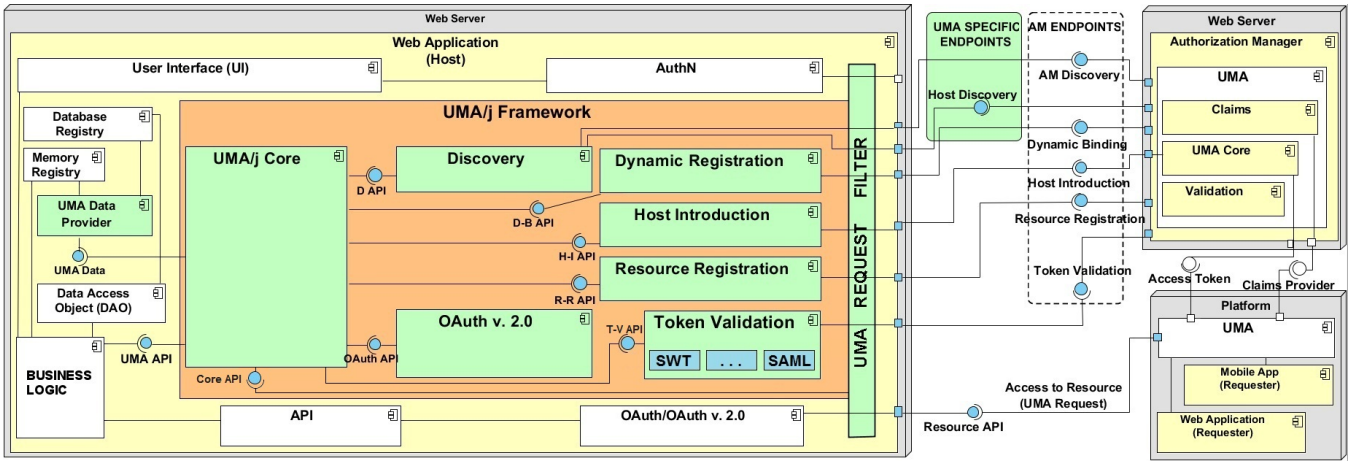

Figure 2: Design of the UMA/j framework and its integration with a Web application.

that the application must use when establishing a trust relationship with the AM (recall step (1) of the UMA protocol). This involves providing the AM with information about the application, i.e. its name, description, address, redirect URL, and icon. The AM displays such information to the user when asking for authorization of this trust relationship.

Dynamic registration is performed according to models defined in [21], where the host pushes the information to the AM (push model) or where the AM discovers information directly from the host (pull model). We extend the pull model by allowing the AM to respond with a nonce which the host must place into its description file (e.g. XRD). The AM then retrieves this file from the host's discovery URL and verifies its validity.

In the pull registration model, the AM has more flexibility in determining the trustworthiness of a Web application because its description is retrieved from an authoritative source. For example, the AM may use HTTPS to retrieve information and may check if the domain in the obtained SSL certificate matches that from the descriptor file. The nonce is included in the file to ensure that it's the actual application that provisions this description.

This module then retrieves {*cid,csecret*}. These credentials are retrieved only once for each host-AM pair and are reused among different users of the same application (a user of the application never learns the *csecret*).

Our framework supports all aspects of dynamic registration based on the application's configuration provided by the developer. In case of the pull model, for example, UMA/j provides an endpoint that handles host discovery by the AM.

4.1.3 Host Introduction

The host introduction process in UMA is based on the OAuth V2.0 Web Server Flow [11]. The UMA/j framework supports this step of the protocol with a custom-built OAuth V2.0 library [2] that integrates with other modules of the framework and can be also used as a standalone library.

This module allows the application to communicate with the AM using {*cid,csecret*} in order to receive the AT_{host}. This token allows the application to use the AM to register resources which should be protected and to validate AT_{req} tokens received from requesters.

Our OAuth V2.0 library is capable of receiving both access tokens (AT_{host}) and refresh tokens (RT_{host}) as defined in [11]. When a particular AT_{host} is no longer valid for the AM, then a corresponding RT_{host} is used to obtain a new AT_{host}. UMA/j supports this process in a transparent way to the application.

4.1.4 Resource Registration

As discussed earlier, a host may delegate access control to the AM for all resources of a particular user, for groups of resources or for individual resources only. UMA/j supports all three granularity levels of access control delegation.

Typically, an application would protect all resources of a particular user at this user's preferred AM. In more advanced settings, once a user introduces an application to the AM they are able to select which resources must be protected in due course. For example, a user registers his online photo service with his chosen AM and then selects individual albums or photos that require protection, among others that require none. UMA/j supports this by allowing the application to communicate information about resources to the AM, including information about resource location and associated metadata (e.g. resource's groups). Such information can then be used when a user defines access control policies at the AM.

Web applications can register arbitrary resources that can be identified by URIs as required by UMA. Our framework extends this by allowing arbitrary grouping of resources and protecting such groups by AMs. It is the responsibility of a developer, however, to define these groups for our framework. For example, an online gallery service could group photos in albums or allow grouping of individual photos based on user-supplied tags.

4.1.5 Request Filter

UMA/j provides a generic mechanism for filtering requests to Web resources that is based on Servlet Filters [1]. The request filter checks HTTP requests and detects those that are issued to UMA-protected resources. Information about whether a resource is UMA-protected and regarding the resource's group is provided by the Core module.

When the resource is protected but the required AT_{req} is missing then the filter responds with a standard "HTTP 401 Unauthorized" response and includes information about

the AM that protects this resource. However, if AT_{req} is detected, then the filter passes the request to the core module which uses the token validation component for further processing.

4.1.6 Token Validation

This component supports both local and remote validation of AT_{req} received by a host from a requester, which is first processed by the Request Filter. The UMA protocol itself does not restrict the token format to be used, so this module can work with different token types.

Local validation is performed by plugins responsible for different token types. By default, the framework supports validation of SWT tokens [9] by checking the token's issuer (i.e. if it's the AM registered by a user) and matching its properties against requested access type. UMA/j also supports the inclusion of custom plugins for other token types.

Remote token validation is performed by sending a JSON-formatted access decision request to the registered AM. This request describes the access request issued by a requester (i.e. AT_{req}, scope, HTTP method) and has a host's AT_{host} attached to it. This request may also include additional information. Our framework, for example, includes information about token usage in access decision requests to provide support for one-time tokens.

4.1.7 Core

This module aims to (1) allow the application to use UMA functionality, (2) provide services for all UMA/j components, and (3) expose an abstraction of persistent storage of UMA-related data to these components. As such, it glues all the modules together and ensures that their functionalities comply with the UMA protocol.

The Core module hides the complexity of using separate UMA/j components by exposing a high-level *UMA API*. Firstly, this API allows the application to delegate access control to users' preferred AMs. Secondly, it allows to register resources that are meant to be protected. The framework then takes care of underlying steps (e.g. discovery and dynamic registration in the host introduction process).

This module provides services for other UMA/j components by providing them with the required data and ensuring that such data is passed correctly as necessary. For example, it passes discovered information about endpoints to all modules or provisions the correct AT_{host} to resource registration and token validation modules (in case remote validation is necessary). Moreover, it ensures that all UMA-related data remains in a consistent state for separate components.

The Core module provides an interface for persistent storage of all UMA-related data which includes endpoint descriptions, credentials, and access and refresh tokens for different AMs (possibly used by different users of the same Web application). This information is then used by the Web application and other modules of the framework. We depend, however, on the developer to provide a concrete implementation of this interface (e.g. a developer can decide to store UMA-related data in a file or using a database).

4.2 Implementation

UMA/j has been implemented in Java according to software engineering best practices with design patterns applied where necessary. Modularity of the framework allows for adjustment to different deployment environments and the use of different components independently. For example, the OAuth V2.0 module can be used as a standalone library. Moreover, UMA/j does not rely on any specific Web framework and can be easily integrated with applications that conform to the Java Servlet specification [1].

Unit tests have been provided for individual components and the framework's compliance with the UMA protocol has been verified using integration tests. UMA/j is planned to be released as an open-source project.

4.3 Evaluation

UMA/j has been evaluated empirically from the perspective of Web application developers as well as end users of these applications. At this stage of development, we have not yet conducted any performance or scalability tests of the proposed framework.

To test UMA/j empirically, we implemented two prototype Web applications, an online storage service and an online photo gallery service, that can be deployed to the Google App Engine (GAE) platform. Both applications provide user interfaces (UI) and RESTful APIs. The storage service is based on Java servlets while the gallery service is based on the Spring MVC framework. RESTful APIs were provided using Apache CXF. We integrated UMA/j with both applications and tested using our prototype AM and requester applications to verify our solution for User-Managed Access.

Integration of UMA/j required minimal programming effort and changes to configuration files only. Firstly, necessary dependencies were included in the classpath of both applications. Then, the UMA API was added, separately from the business logic. Host discovery, OAuth, and request filtering have been configured by adding these modules to deployment descriptors of Web applications. The first two modules were included as a single servlet while the third module was included as a filter. Configuration for all UMA/j modules was provided in a Java properties file and included in WAR files of both applications. UMA functionality was exposed to the users with a custom UI at each application.

End users of our Web applications were able to use the provided UI to delegate access control to our prototype AM. Such delegation had to be set up only once per user per application. The user would choose their preferred AM either from a list of pre-configured AMs or by providing the AM's URL. Then, users were able to select which resources should be UMA-protected and they would define access control policies for their resources using our prototype AM.

5. RELATED WORK

Delegating access control from Web applications to specialized components is usually provided in form of middleware or frameworks, with examples such as ESAPI, Spring Security or OpenSSO. These frameworks partially alleviate the problem of implementing custom authorization for applications by providing high-level access control APIs and allowing policies to be defined in separate configuration files (or in a central location as in OpenSSO).

These solutions, however, have not been primarily designed for the open and user-driven Web 2.0 environment. ESAPI and Spring Security provide benefits to developers only and they are based on access control policies configured for each application separately. OpenSSO, on the other hand, requires pre-established relationships between applications and their central access control system. It also lacks

support for discovery of services provided by such systems and does not allow for dynamic resource registration.

The OAuth V1.0 [13] proposal to access control for Web services attracted much attention from the Web community. It allows to share data stored at one service with another service without revealing user credentials. Each OAuth service provider, however, must independently serve an authorization management function and users cannot centralize their access control settings in a single location.

OAuth V2.0 [11] is a less complex attempt to solve the problem of authorization for Web resources. It allows applications to delegate authorization to a trusted authority (Authorization Server) which issues tokens to clients of these applications. OAuth V2.0, however, fails to provide certain features required in today's open Web environment. In particular, it does not support dynamic introductions by users and requires relationships between parties to be pre-established. Therefore, despite logically separating the authorization system from the service hosting data, these two need to be tightly coupled for the protocol to work.

Kerberos [19] is a well-established protocol that allows to delegate access control from servers to a centralized system. Clients first obtain a token from such a system and then use it to gain access to resources on a server. Kerberos allows to centralize authorization for various services but it requires a well-defined point of authority, relationships between services to be pre-established and relies heavily on symmetric cryptography. As such, it is ill-fitted for the open Web environment.

6. CONCLUSIONS

We have discussed UMA/j, the first known framework to provide Web applications with support for the User-Managed Access protocol. The framework allows users of Web applications to delegate access control to their preferred authorization managers. Such delegation can be done dynamically and does not require pre-established relationships between these applications and the users' chosen AMs. Moreover, users of Web applications can select in due course which of their resources are meant to be protected by their authorization managers. As such, UMA/j provides access control delegation in a highly flexible manner, allowing users of Web applications to use access control mechanisms that meet their exact requirements and expectations.

7. ACKNOWLEDGMENTS

The authors are supported by the JISC-funded project "SMART". Additionaly, authors Machulak and van Moorsel are supported by UK Technology Strategy Board grant nr. P0007E ("Trust Economics"). The authors would like to acknowledge the work done by members of the UMA WG towards standardization of the UMA protocol and would like to thank Eve L. Maler and Dr. Carlos Molina-Jimenez for their valuable feedback on this paper.

8. REFERENCES

[1] JSR-154: Java Servlet 2.5 Specification. http://jcp.org/en/jsr/detail?id=154. Accessed 29/09/2010.

[2] OAuth leeloo. http://leeloo.smartam.net/. Accessed 29/09/2010.

[3] OpenSSO Project. https://opensso.dev.java.net/. Accessed 29/09/2010.

[4] OWASP Enterprise Security API. http://www.owasp.org/. Accessed 29/09/2010.

[5] Spring Security. http://static.springsource.org/spring-security. Accessed 29/09/2010.

[6] UMA 1.0 Core Protocol. http://kantarainitiative.org/confluence/display/uma/UMA+1.0+Core+Protocol. Accessed 29/09/2010.

[7] UMA Scenarios and Use Cases. http://kantarainitiative.org/confluence/display/uma/UMA+Scenarios+and+Use+Cases. Accessed 29/09/2010.

[8] OASIS eXtensible Access Control Markup Language (XACML). http://www.oasis-open.org/committees/xacml/, 2005. Version 2.0.

[9] Simple Web Token. http://oauth-wrap-wg.googlegroups.com/web/SWT-v0.9.5.1.pdf, November 2009. Version 0.9.5.1.

[10] Extensible Resource Descriptor (XRD) Version 1.0. http://docs.oasis-open.org/xri/xrd/v1.0/xrd-1.0.html July 2010. Committee Specification 01

[11] The OAuth 2.0 Protocol. http://tools.ietf.org/html/draft-ietf-oauth-v2, June 2010. (Work in Progress). Draft 09.

[12] A. Cavoukian. Privacy in the clouds. *Identity in the Information Society*, 1:89–108, December 2008.

[13] E. Hammer-Lahav. The OAuth 1.0 Protocol. RFC 5849 (Draft Standard), 2010.

[14] E. Hammer-Lahav. Web Host Metadata. http://tools.ietf.org/html/draft-hammer-hostmeta, June 2010. (Work in Progress). Draft 13.

[15] M. Hart, R. Johnson, and A. Stent. More content - less control: Access control in the web 2.0. In *WOSP '08: Proc. of the First Workshop on Online Social Networks*, New York, NY, USA, 2008.

[16] M. P. Machulak, D. Catalano, E. L. Maler, and A. van Moorsel. User-Managed Access to Web Resources. In *DIM '10: Proc. of the 6th ACM Workshop on Digital Identity Management*, New York, NY, USA, 2010.

[17] M. P. Machulak and A. van Moorsel. Architecture and Protocol for User-Controlled Access Management in Web 2.0 Applications. In *ICDCS-SPCC 2010: Proc. of the 1st ICDCS Workshop on Security and Privacy in Cloud Computing*, Genoa, Italy, June 2010.

[18] M. L. Mazurek et al. Access control for home data sharing: Attitudes, needs and practices. In *CHI '10: Proc. of the 28th Intl. Conf. on Human Factors in Computing Systems*, New York, NY, USA, 2010.

[19] C. Neuman, S. Hartman, and K. Raeburn. The Kerberos Network Authentication Service (V5). RFC 4120 (Draft Standard), 2005.

[20] M. Nottingham and E. Hammer-Lahav. Defining Well-Known Uniform Resource Identifiers (URIs). RFC 5785 (Draft Standard), 2010.

[21] Scholz, C. and Machulak, M. P. and Maler, E. L. OAuth Dynamic Client Registration Protocol. http://tools.ietf.org/html/draft-oauth-dyn-reg. Accessed 29/09/2010.