W3C Workshop on Privacy and data usage control Position Paper "Distributed Privacy Policy Enforcement by using Sticky Policies"

David Chadwick and Kaniz Fatema, University of Kent

Introduction

Today users have very little control over the privacy of their personal information (PII) once it has been submitted to a web site. The site may inform them about what it will do with their PII, and give them the option of agreeing to this or not, but once the PII has been submitted the user simply has to trust that the site will act in good faith.

As sites start to adopt policy based systems, it should become easier for them to allow their users to select their preferred privacy policies (probably from a small limited set of policies initially) and have these automatically processed by the site's PDP. However, if the site wishes to transfer a user's PII to another site with which it has a business relationship, this means that the remote site with either have to support the same policy language as the sending site, or have a policy language convertor, or we are back to square one with the sending site having to trust the recipient site to behave in good faith according to some written contractual policy. Even if the recipient site does support the same policy language, we could end up with conflicts between the user's policy and the site's policy.

In order to overcome these problems, in the EC TAS3 project we are constructing a policy enforcement infrastructure that will support the transfer between sites of PII together with its sticky policies, be capable of handling multiple policy languages (and their corresponding PDPs), and have a Master PDP that will resolve any policy decision conflicts. We use obligations to guarantee that sending sites will attach sticky policies to the outgoing PII and that receiving sites will store the sticky policies.

In our design we also had the following non-functional objectives:

- implement as much as possible of the functionality in an application independent infrastructure
- require the minimum of alterations to existing applications that already hold PII
- make the infrastructure as standards compliant as possible
- release the code as platform independent open source

The Architecture

The architecture is shown in Figure 1 below. We have introduced several new components to the traditional PEP and PDP, as described below.

Application Independent PEP

One can regard the AIPEP as a much richer functional version of the XACML context handler. The AIPEP is responsible for coordinating the actions of the various components of the application independent authorization infrastructure. It presents a single interface to the application PEP, in order to make integration easy. To the PEP it appears to be a single PDP as it responds to authorization decision queries in the standard way. When the AIPEP receives either an authorization decision query message or a credential validation message (step 1 in figure 1), it first calls the CVS

to validate any credentials that are contained in the message (step 2 in figure 1). If the message contains a sticky policy (see figure 2) then this will be stored in the policy store. The AIPEP retains a manifest which records which CVSs and PDPs are currently spawned and which policies each is configured with. If the AIPEP has policies for which there are currently no spawned CVSs or PDPs, then a PDP/CVS factory object is used to spawn new PDPs and/or CVSs as appropriate. The PDP/CVS factory object is configured to know each policy language that each PDP and CVS class supports. When it is passed a sticky policy it knows how to construct the appropriate PDP and/or CVS instances and to give them the correct components of the sticky policy. Once the correct CVS and PDPs have been spawned, the AIPEP calls the CVS to validate the credentials on the incoming message, and then tells the Master PDP which set of spawned PDPs to use for a particular authorization decision request. Finally the AIPEP calls the inbuilt Obligations Service to perform any returned obligations that the infrastructure knows about.

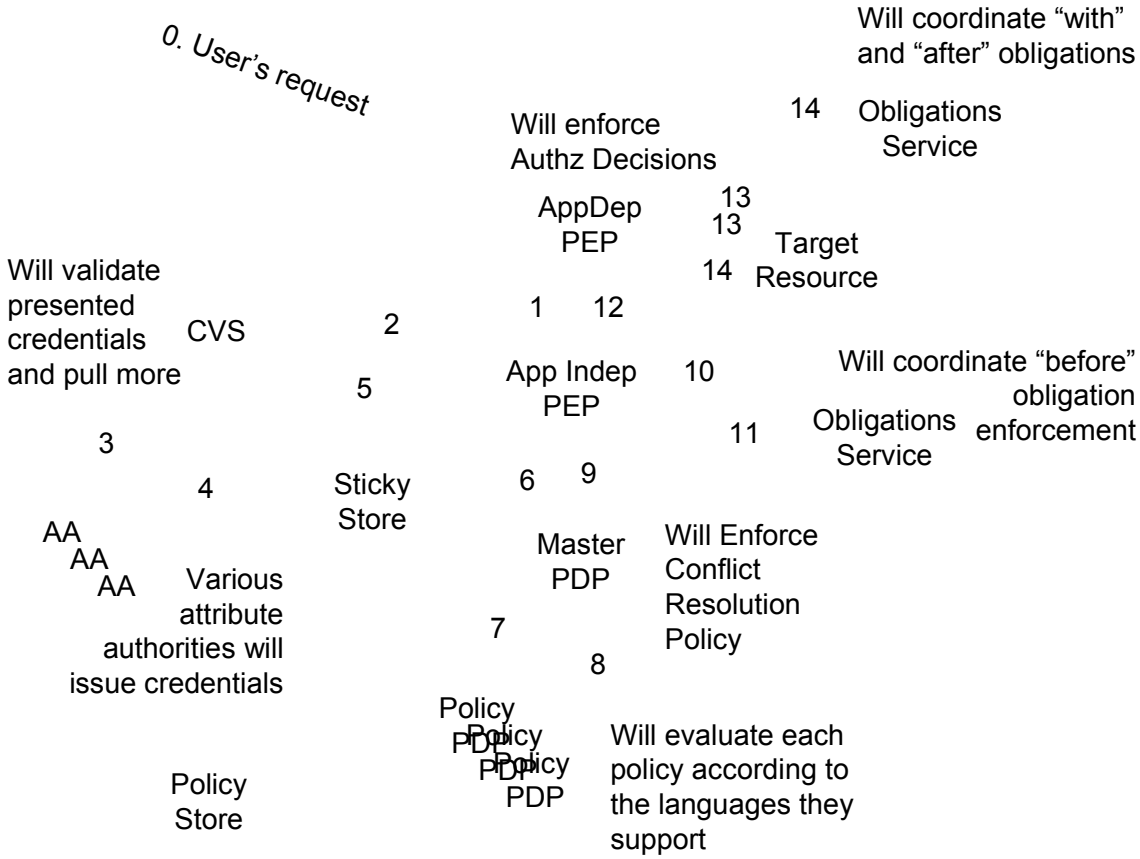

Figure 1. A Multi-Policy Application Independent Authorization Infrastructure

Credential Validation Service

The CVS is a specialized Policy Information Point (PIP) that is configured with a credential validation policy and cryptographic validation functions. The latter validate the signatures on the credentials, whilst the former tells it which credentials are valid, in terms of who the trusted authorities (IdPs) are and which attributes each is trusted to issue to which groups of users. Once the CVS has finished validating the subject's credentials, these are returned to the AIPEP as standard XACML formatted attributes (step 5). The CVS can work in either pull mode, push mode or pull and push mode. Pull mode means that the requester did not present any credentials and requires the

CVS to pull them itself from its configured trusted authorities, or a subset of them. In push mode the PEP pushes the credentials to the AIPEP, and in pull push mode some credentials are pushed and the remainder have to be pulled. The CVS is described in detail in [1].

Master PDP

The Master PDP is responsible for calling multiple subordinate PDPs (step 7) as directed by the AIPEP, obtaining their authorization decisions (step 8), and then resolving any conflicts between these decisions, before returning the overall authorization decision and any resulting obligations to the AIPEP (in step 9). The Master PDP has a conflict resolution policy which tells it, for each request context, which conflict resolution rule to use (e.g. Deny overrides, Grant overrides, First applicable etc.) Each of the policy PDPs supports the same interface, which is the XACML request-response context. This allows the Master PDP to call any number of subordinate PDPs, each configured with its own policy in its own language. This design isolates the policy language from the rest of the authorization infrastructure, and the Master PDP will not be affected by any changes to any policy language as it evolves or by the introduction of any new policy language. Of course, new policy languages will require new PDPs to be written to interpret them, and these new PDPs will require new code in the PDP/CVS factory object so that it knows how to spawn them on demand. But this is a one-off occurrence for each new policy language and PDP that needs to be supported by the infrastructure. Our first implementation will have built-in support for three different PDPs/policy languages, namely: XACMLv2, PERMIS [2] and SWI-Prolog (used in a trust based PDP built by TU-Eindhoven).

Policy Store

The policy store is the location where policies can be safely stored and retrieved. If the store is trusted then policies can be placed there in an unsecured manner. If the store is not trustworthy then policies will need to be protected e.g. digitally signed and/or encrypted, to ensure that they remain confidential and are not tampered with. Each policy has an id, the PID, which the policy store returns to the AIPEP when asked to store a policy. The AIPEP can subsequently use the PID when asking either the PDP/CVS factory to spawn a new PDP/CVS or the sticky store to stick this policy to some PII. This design cleanly separates the implementation details of the policy store from the rest of the infrastructure, and allows different types of policy store to be constructed e.g. built on an LDAP directory or RDBMS.

Sticky Store

The sticky store holds the mapping between sticky policies and the resources to which they are stuck. This is a many to many mapping so that one policy can apply to many resources and one resource can have many sticky policies applied to it. The design requires that each policy id (PID) is globally unique so that when a sticky policy is moved from system to system, the receiver can determine if it needs to analyse each received policy or not. Already known PIDs don't need to be analysed, whereas unknown PIDs will need to be evaluated to ensure that they can be supported, otherwise the incoming data and sticky policy will need to be rejected. The RID is locally unique and may be constructed by applying a one way hash function such as SHA1 to the resource. We currently do not have any requirement to pass the RID from system to system so each system can compute its own.

Obligations Service

Obligations may be required *before* the user's action is performed, *after* the user's action has been performed, or simultaneously *with* the performance of the user's action. We call this the *temporal type* of the obligation. According to the XACML model, each obligation has a unique ID (a URI). We follow this scheme in our infrastructure. Each obligations service is configured at construction time with the obligation IDs it can enforce and the obligation handling services that are responsible for enacting them. It is also configured with the temporal type(s) of the obligations it is to enforce. When passed a set of obligations by the AIPEP, the internal obligations service will walk through this set, ignore any obligations of the wrong temporal type or unknown ID, and call the appropriate obligation handling service for the others. If any single obligation handling service returns an error, then the obligations service stops further processing and returns an error to the AIPEP. If all obligations are processed successfully, a success result is returned. Each of the obligations enforced by the AIPEP must be of temporal type *before*.

Walkthrough of user input of PII and sticky privacy policy

The user is presented with an application dependent GUI and is asked to enter their PII. Existing GUIs will need to be enhanced to invite the user to enter their privacy policy. We do not specify how this is done, but it is most likely that organisations will have a limited number of options that a user can choose from, with a default policy for users who don't really mind. When the application server receives the user's input, it must extract the subject's privacy policy from the application layer message and pass this to the AIPEP along with an authorisation decision request "can this user submit this PII (with this unique RID) to the data store, using this policy in conjunction with the existing policies". The AIPEP takes the policy, stores it in the policy store and is returned the PID of the policy. It then constructs a new PDP that can process this policy. The AIPEP has a manifest that records the number of PDPs that are currently active, along with their PIDs. These include the PDPs that were initialised when the authorisation infrastructure was started, plus any additional PDPs that have been dynamically constructed since then. Once the PDP has been constructed the AIPEP passes the authorisation decision request to the Master PDP along with the set of PDPs that should be used to process this request.

The Master PDP consults its conflict resolution policy for this request context and obtains the conflict resolution rule (CRR) to use. It then calls the subordinate PDPs, either sequentially (first applicable CRR) or in parallel (all other rules) and analyses the returned decisions according to the rule. If the CRR is deny or grant overrides, and there is more than one such response of the same type, then the obligations from all such similar responses are combined together in the response that is returned to the AIPEP by the Master PDP. If a grant is returned this will contain at least one "before" obligation which instructs the authorisation system to store the subject's sticky policy in the sticky store.

The AIPEP calls the obligations service passing it the set of received obligations. This obligations service is only configured to process "before" type obligations, one of which will be instructions to the sticky store obligation service to store the subject's sticky policy. Other "before" type obligation handling services may be configured into the internal obligations service, such as "audit the authz decision" etc. Only if all "before" type obligations that are known about are successfully enacted will the AIPEP return granted to the PEP. If any of the known obligations fail to be enacted, then the AIPEP will return a deny response and will rollback its actions i.e.

remove the subject's privacy policy from the policy store, terminate the appropriate subordinate PDP and remove it from its manifest. When the PEP receives the granted response it will store the user's PII in its application dependent storage.

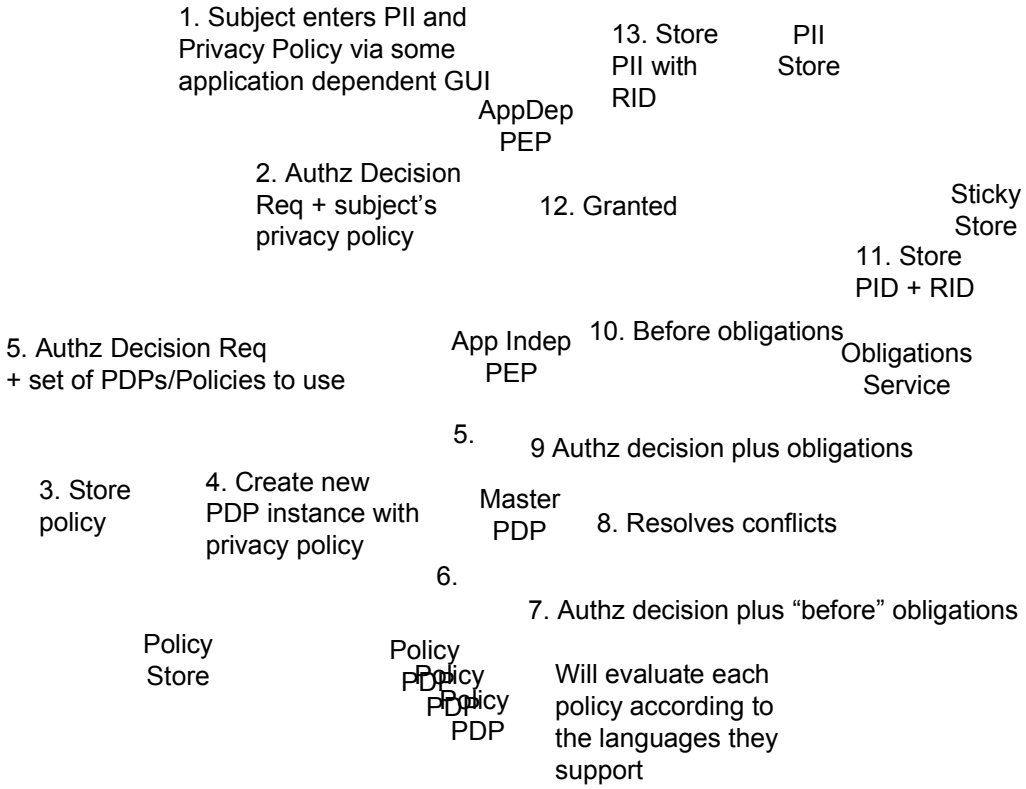

Figure 2. User input of PII plus privacy policy

References

[1] Chadwick DW, Otenko S and Nguyen A T. "Adding Support to XACML for Multi-Domain User to User Dynamic Delegation of Authority". International Journal of Information Security. April 2009; Volume 8: Number 2:pp 137-152.

[2] Chadwick DW, Zhao G, Otenko S, Laborde R, Su L and Nguyen A T. "PERMIS: a modular authorization infrastructure". Concurrency And Computation: Practice And Experience. 10 August 2008; Volume 20: Issue 11: Pages 1341-1357.

Acknowledgements

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 216287 (TAS³ - Trusted Architecture for Securely SharedServices)[1].

[1] The information in this document is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The above referenced consortium members shall have no liability for damages of any kind including without limitation direct, special, indirect, or consequential damages that may result from the use of these materials subject to any liability which is mandatory due to applicable law.