p medicine
personalized

Deliverable No. 3.4
Service Integration Guidelines

Grant Agreement No.:	270089
Deliverable No.:	D3.4
Deliverable Name:	Service Integration Guidelines
Contractual Submission Date:	31/07/2012
Actual Submission Date:	31/07/2012

Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

SEVENTH FRAMEWORK PROGRAMME

COVER AND CONTROL PAGE OF DOCUMENT	
Project Acronym:	***p-medicine***
Project Full Name:	From data sharing and integration via VPH models to personalized medicine
Deliverable No.:	D 3.4
Document name:	Service Integration Guidelines
Nature (R, P, D, O)[1]	R
Dissemination Level (PU, PP, RE, CO)[2]	PU
Version:	1
Actual Submission Date:	31/07/2012
Editor: Institution: E-Mail:	Elias Neri Custodix elias.neri@custodix.com

ABSTRACT:

This deliverable starts by elaborating on the technical details of the data security framework defined in D5.1. The main focus initially lies on the authentication components to enable SSO, brokered authentication for as well web sites as REST services and delegation. Later iterations will further define authorisation and auditing.

The second part of the deliverable further extends the privacy framework as defined in D5.1. A global high level overview of the import of sensitive patient data into p-medicine is given. Next to this the specific components CATS and PIMS are explained in more detail.

The last part explains how p-medicine services can integrate with the authentication components. For this the minimal requirements are given to which a service should adhere to be integrable. Some examples are given to explain in detail how integration can be achieved.

The integration of services with the privacy framework is described in D8.6.1 "Integration guidelines and monitoring of tools and services". D8.6.1 also covers more general non security related integration guidelines.

KEYWORD LIST: security, integration guidelines, SAML, XACML, delegation, REST, WS-Security, WS-TRUST, anonymisation, pseudonymisation, CATS, PIMS, Identity Provider

[1] **R**=Report, **P**=Prototype, **D**=Demonstrator, **O**=Other

[2] **PU**=Public, **PP**=Restricted to other programme participants (including the Commission Services), **RE**=Restricted to a group specified by the consortium (including the Commission Services), **CO**=Confidential, only for members of the consortium (including the Commission Services)

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n^{o} 270089.

The author is solely responsible for its content, it does not represent the opinion of the European Community and the Community is not responsible for any use that might be made of data appearing therein.

MODIFICATION CONTROL			
Version	Date	Status	Author
0.1	15/05/2012	Draft	Elias Neri, Wouter Dhaeze
0.2	13/07/12	Draft	Elias Neri, Wouter Dhaeze
1.0	31/07/12	Final	Elias Neri

List of contributors

– Benjamin Jefferys, UCL
– Dawid Szejnfeld, PSNC
– Elias Neri, Custodix
– Fatima Schera, Fraunhofer IBMT
– Giorgos Zacharioudakis, FORTH
– Jelle Van Den Driesche, Custodix
– Wouter Dhaeze, Custodix

Contents

1 Executive Summary

This document elaborates on the technical details of the data security framework defined in D5.1 and explains how the services within p-medicine can integrate with the security framework. The first step in the setup of the security architecture focuses on the authentication and de-identification components. Therefore this deliverable explains how services should integrate with the authentication components. As the de-identification components are part of the p-medicine workbench and tools, integration with the privacy framework is described in "D8.6.1 Integration guidelines and monitoring of tools and services".

The p-medicine security framework is designed around a lightweight dynamic architecture. It consists of modular re-usable components dealing with authentication, authorisation, auditing and de-identification based on widely used industry standards such as SAML and XACML.

Identity assertions (SAML tokens) are provided to the p-medicine web sites by the Identity Provider (IdP). Web sites integrate with the IdP by publishing SAML metadata that define which SAML profiles and bindings are supported by the site. SAML consumers, matching those profiles and bindings, are then responsible to consume and verify, on the web sites, the tokens issued by the IdP. A commonly used binding and profile for browser SSO is the HTTP redirect binding of the web browser SSO profile. A user agent is hereby redirected to the IdP, passing through an authentication request, when visiting a protected web site. If the user has no active SSO session open, the IdP will request the user to authenticate himself. Once authenticated the IdP issues an identity token and redirects the user agent to the web site passing through the issued token. Chapter 5 and chapter 6, through examples, further explain how web sites can integrate with the Identity Provider.

For non browser clients (e.g. clients calling web services) SAML provides the ECP (Enhanced Client or Proxy) profile. To not authenticate on each web request, the SAML profiles assume session-based security to keep the security context open. As web services are typically stateless and a different security context can be required for each message, message level security is preferred. For SOAP web services there is the whole set of WS-* specifications handling message level security and brokered authentication.

For REST web services there are no such specifications yet. As REST services are web services, security in REST can actually be based on WS-*. Therefore to do brokered authentication in REST the same SOAP Secure Token Service (STS) can be used as with SOAP web services. REST clients are then responsible for retrieving a SAML token from the STS and then passing it through the HTTP authorization header to the REST service.

The client needs security metadata from the REST service to know what kind of tokens the service requires and which STS to call. In WS-* a web service publicises its security policies through WS-Policy annotations on the service's WSDL metadata. Such a WSDL file, or a more REST oriented WADL file, annotated with security policies could actually also be used for REST services. For WADL no known specification exists that defines how to add security policies. Therefore, in the context of p-medicine, research should be done on whether WS-Policy metadata annotations can be used to add security annotations to a WADL metadata file.

Delegation allows an intermediate client to call a service in name of an end user. Delegation is supported by the STS through the ActAs element. A client can request a delegation token form the STS by embedding the end user's SAML identity token in the ActAs element of a token request. The returned delegation token is then a SAML token that identifies the end user as subject and the client as intermediate.

Next to authentication, the de-identification components are further defined. These components are an important part of p-medicine as all data that will be imported into p-

medicine, should be pseudonymised. Two rounds of pseudonymisation are mentioned. At the hospitals, the treatment domain, a first round of pseudonymisation is performed. The data is then passed to the Trusted Third Party (TTP) for a second round of pseudonymisation. This to ensure that there is no direct link between the p-medicine pseudonyms and the pseudonyms in the treatment domain. The data is then delivered to p-medicine. Any re-identification needs to pass through and be allowed by the TTP.

Two major de-identification components can be identified. CATS pseudonymises data files by using privacy profiles that define transformation rules such as pseudonymisation of identifiers, clearing of dates, encryption of sensitive information. PIMS stores the link between identifiable patient information and patient pseudonyms. PIMS also matches and links patient records that represent the same real-world person. PIMS hereby builds a so called Master Patient Index (MPI).

2 Introduction

The P-medicine security architecture is a lightweight dynamic architecture consisting of reusable components dealing with authentication, authorisation, auditing and de-identification.

The first step in the setup of the security architecture focuses on the authentication and de-identification components. Therefore this deliverable will elaborate on the authentication and de-identification components defined in D5.1. It then lays out how all services within p-medicine should integrate with the authentication components.

As the de-identification components are part of the p-medicine workbench and tools, the integration guidelines with the privacy framework are described in "D8.6.1 Integration guidelines and monitoring of tools and services".

Later iterations of this document will further define the authorisation and auditing components and the respective integration guidelines.

3 Security Framework Overview

3.1 Introduction

As described in deliverable 5.1 the p-medicine security framework is designed around a lightweight dynamic architecture, allowing it to evolve over time according to newly arising requirements. It consists of modular re-usable components dealing with e.g. authentication, authorisation, auditing, de-identification based on widely used industry standards such as SAML, XACML.

3.2 Security Architecture

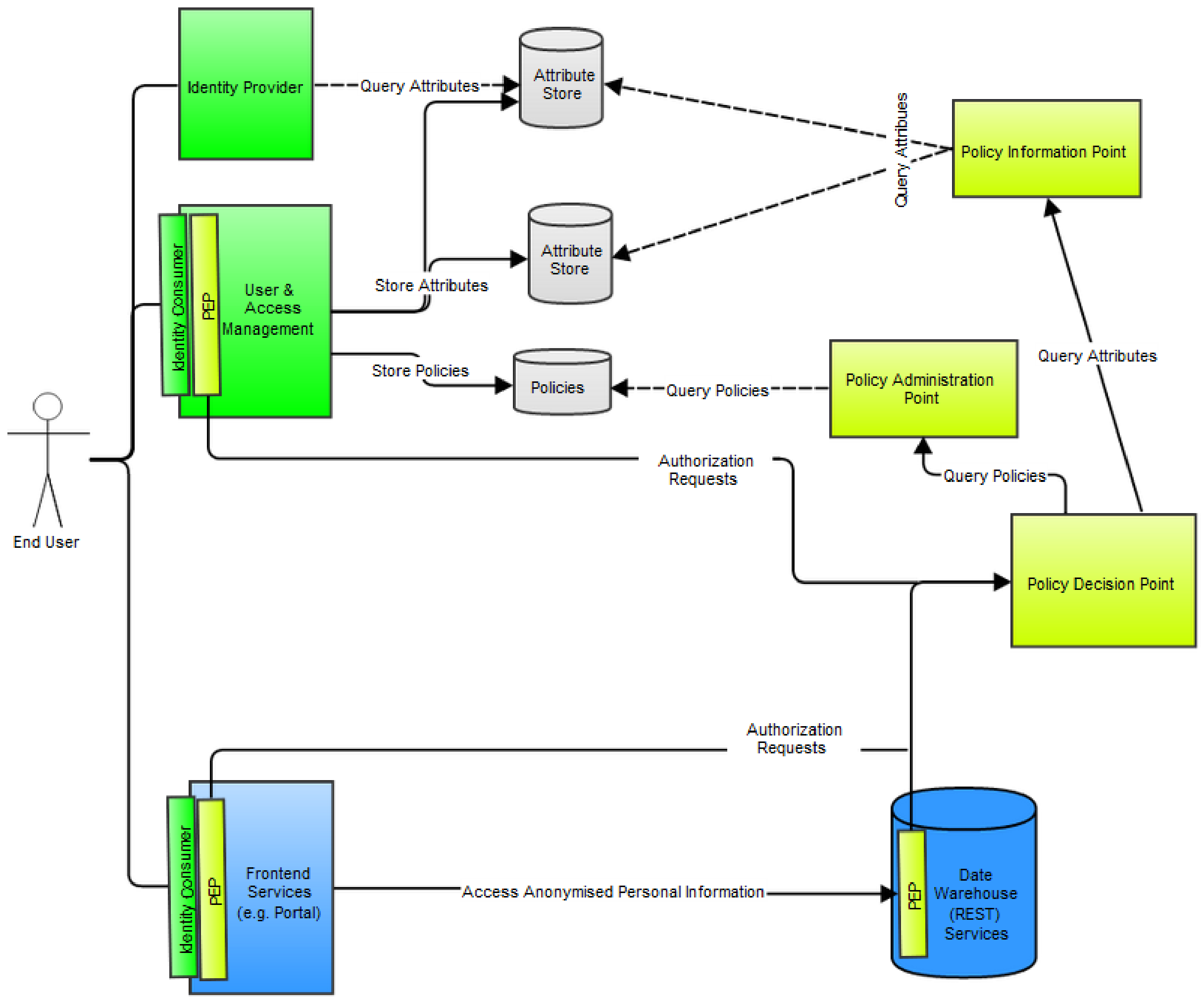

Figure 1 Security Architecture Overview

Major components in the architecture are:

- Authentication components:
  - The Identity Provider (IdP) is a service provider within a federation responsible for authentication. It provides identity assertions to other service providers.

  - An Identity Consumer is a software component that is part of a service provider. It consumes the assertions provided by the Identity Provider. It will verify the received assertion and pass it to the service provider's application layer.
  - A User Enrolment & Management Service where users can be enrolled, revoked, edited, etc. (part of the user and access management in the above figure).
- Authorisation components:
  - A Policy Enforcement Point (PEP) is a software component which requests and enforces authorisation decisions.
  - A Policy Decision Point (PDP) is an entity that makes authorisation decisions. A PDP accepts authorisation requests and will make a decision based on policies fetched from a Policy Administration Point (PAP).
  - A Policy Information Point is an endpoint which provides missing information to a PDP i.e. attribute information. For example if a policy requires information on a specific attribute which has not been provided with the authorisation request, a PDP might request a PIP for information on that attribute.
  - A Policy Administration Point (PAP) is an endpoint which manages policies. The PAP provides a PDP with all policies required to produce an authorisation decision.
  - An Authorisation Rule (Policy) Management Service where authorisation rules can be configured generating authorisation policies (part of the user and access management in the above figure).

3.3 Brokered Authentication and SSO

Direct authentication of the user on each service is not practical or advisable (e.g. authentication would have to be implemented on each service or a user would have to authenticate on each service). In case of front-end services calling back-end services in name of the end user, direct authentication is even impossible. P-medicine will therefore provide brokered authentication in which a central identity provider (or authentication broker) is responsible for authenticating the users and issuing identity tokens. The user can then use such identity tokens to access the project's services. To avoid that the user needs to provide his credentials each time he accesses a different service, the identity provider can keep the authenticated session open. This results in new identity tokens being issued automatically (Single Sign-on) for each service the user accesses as long as the IdP's authenticated (SSO) session is still active.

Therefore when a client accesses a p-medicine service provider (SP) (e.g. the p-medicine portal) where he has no local active authenticated session, the client will not directly authenticate on the portal, but instead he will be requested to pass a p-medicine identity assertion (token). The client should then request such an assertion from the p-medicine Identity Provider (IdP). If he is already authenticated, the Idp will provide the identity assertion (SSO), if not, the client will first have to authenticate himself. The client will then pass that assertion to the SP he originally wished to access, which will then verify the assertion and give the client access if it is valid.

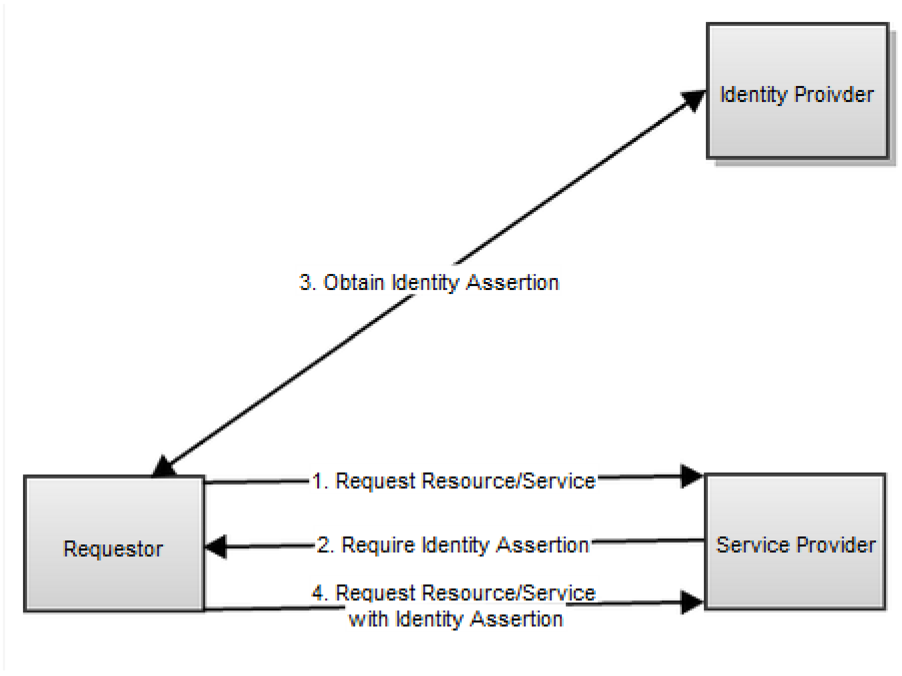
Identity Proivder
3. Obtain Identity Assertion
Requestor
1. Request Resource/Service
2. Require Identity Assertion
4. Request Resource/Service with Identity Assertion
Service Provider

Figure 2 Brokered authentication

Within p-medicine this identity information (assertions) is exchanged through SAML 2.0 tokens. The Security Assertion Markup Language defines an XML-based protocol, making it possible to exchange authorisation and authentication data between one or more security domains. It is a commonly used, well implemented, stable OASIS standard.

3.3.1 Web Sites

Web single sign-on is supported by SAML 2.0 through the Web Browsers SSO Profile.

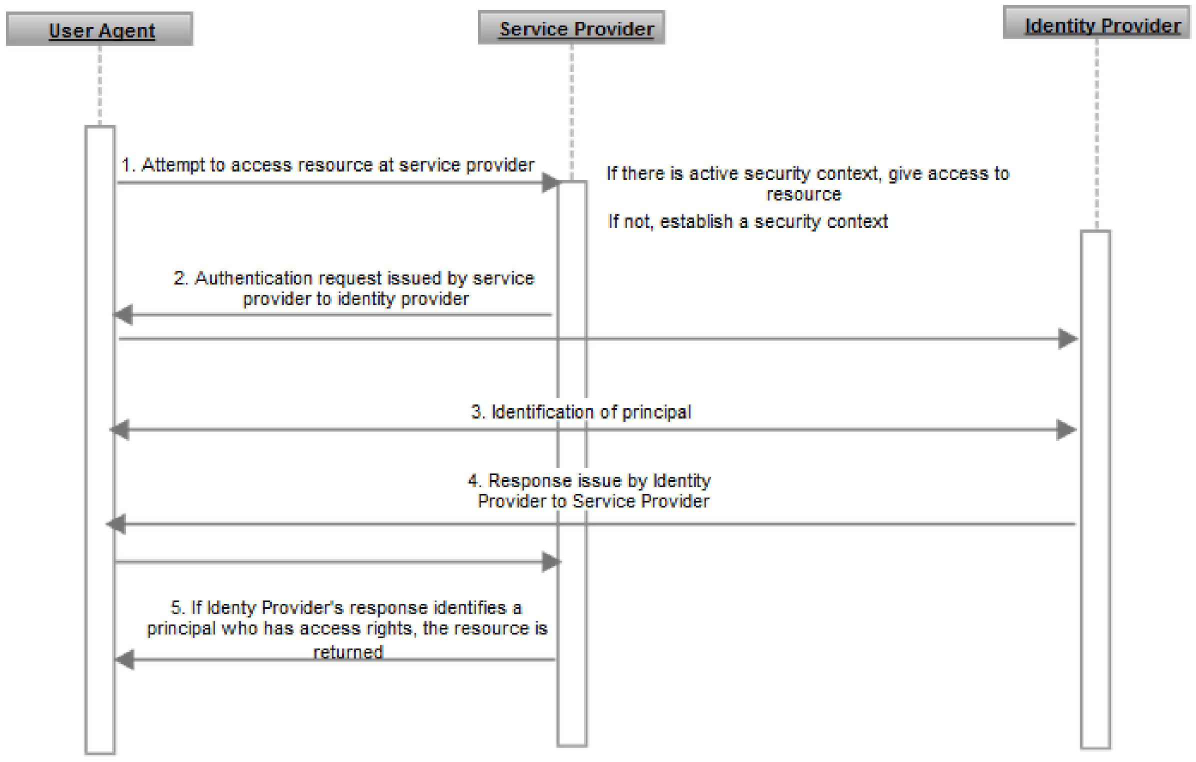
User Agent
Service Provider
Identity Provider
1. Attempt to access resource at service provider
If there is active security context, give access to resource
If not, establish a security context
2. Authentication request issued by service provider to identity provider
3. Identification of principal
4. Response issue by Identity Provider to Service Provider
5. If Identy Provider's response identifies a principal who has access rights, the resource is returned

Figure 3 Web SSO flow

1. The profile is initiated by a HTTP user agent that attempts to access a secured resource at the service provider (SP) without active security context.
2. The service provider will then attempt to determine what identity provider (IdP) to use. Initially there will be only one central IdP within p-medicine.
3. The SP issues an authentication request which should be delivered by the user agent to the IdP. Several bindings are defined to accomplish this, e.g. HTTP Redirect.
4. The IdP then needs to identify the principal (end user or service) by reusing an existing authenticated session or by requiring some form of authentication. How this is done is outside the scope of the SAML specification.
5. Once the principal is identified, the IdP issues a response through the user agent by using e.g. the HTTP POST binding. This response will contain an authentication assertion if no errors occurred.
6. The service provider then verifies the authentication assertion. When verification succeeds, the SP might establish a security context and return the requested resource.

3.3.2 Web Services

For clients that are not browsers, e.g. clients that are calling a web service, SAML provides the ECP (Enhanced Client or Proxy) profile. Similarly to the Web Browser SSO Profile, ECP does not authenticate on each HTTP request. Instead it assumes session-based security where the SAML login establishes a security context between the client and the web server. Web services though are typically stateless. Therefore message level security is preferred. Especially for clients (e.g. proxies) that act in name of several users through delegation, a session based security context is not really useful as each message can typically be sent in name of a different user and therefore requires a different security context.

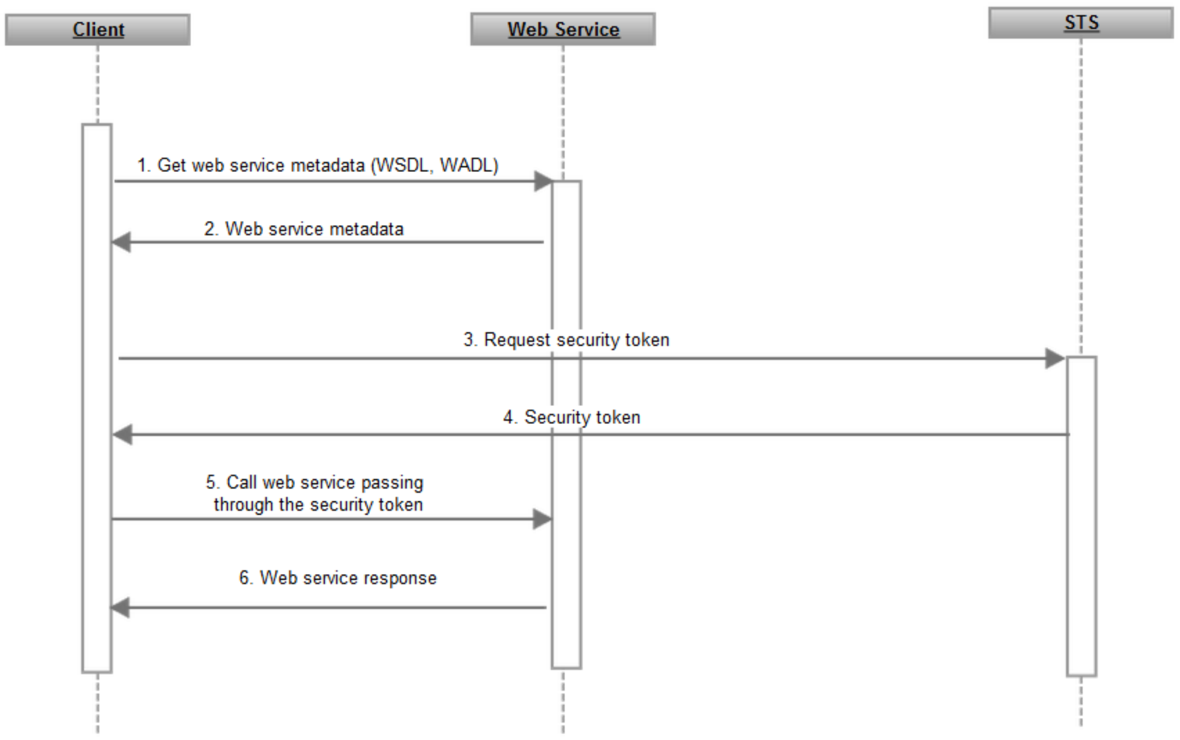
Client
Web Service
STS
1. Get web service metadata (WSDL, WADL)
2. Web service metadata
3. Request security token
4. Security token
5. Call web service passing through the security token
6. Web service response

Figure 4 Flow of secured web service calls

There are two major types of web services: SOAP web services and REST services. These types are described in following sections.

3.3.2.1 SOAP Web Services

For SOAP Web services there is the whole set of WS-* (WS-Security and WS-Trust especially), specifications handling message level security and brokered authentication. WS-* is a collective noun for a variety of specifications associated with web services. Together these specifications form the basic framework for web services build on the first-generation standards of SOAP and WSDL.

WS-Security[3] defines how web service messages can be exchanged in a secure way by guarding the integrity, confidentiality and the sender's identity of the messages. To enforce this, WS-Security uses XML signature (for integrity), XML encryption (for confidentiality) and various security token formats, such as SAML, Kerberos, X.509 (for sender authentication), to provide end-to-end security.

WS-Trust[4] is an extension of WS-Security providing methods for issuing, renewing and validating security tokens and providing ways to establish, assess the presence of, and broker trust relationships. Using the extensions defined in WS-Trust, applications can participate in secure communication designed to work within the web service framework. A main concept in WS-Trust is the Security Token Service (STS). This is a special web service that issues security tokens conforming to the WS-Security specification.

[3] OASIS, WS-Security specification, version 1.1, 2004, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss

[4] OASIS, WS-Trust specification, version 1.4, 2009, http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/os/ws-trust-1.4-spec-os.pdf

WS-SecureConversation[5] defines extensions that include session key derivation and security context establishment/sharing.

WS-SecurePolicy[6] defines XML based policies that are called security policy assertions. These policies allow web services to express their constraints and requirements. Policies can be used to drive development tools to generate code with certain capabilities, or may be used at runtime to negotiate the security aspects of web service communication. The intent is to provide enough information for compatibility and interoperability to be determined by web service participants.

WS-Federation[7] aims to simplify the development of federated services through cross-realm communication and management of federation services by re-using the WS-Trust Security Token Service model and protocol.

P-medicine does not plan to have any SOAP but rather REST web services. Therefore this document will not go into specific details of the WS-* specification. As brokered authentication in REST though, laid down in the next chapter, will be based on WS-Trust, the next paragraphs will go deeper in the interfaces defined in WS-Trust.

When doing brokered authentication through WS-* a service will typically publicize through policies defined in its WSDL that it requires a token from a given Secure Token Service (STS). The web service client should then request a token from that specific STS and then pass it together with the service call to the web service.

An STS is a SOAP Web Service with the following operations (non exclusive):

1) **Issue**: this operation issues a new security token based on the credential provided or proven in the request. The operation accepts a request security token (RST) and returns a request security token response (RSTR).

```
<wst:RequestSecurityToken Context="..." xmlns:wst="...">
    <wst:TokenType>...</wst:TokenType>
    <wst:RequestType>...</wst:RequestType>
    <wst:SecondaryParameters>...</wst:SecondaryParameters>
    ...
    <wsp:AppliesTo>...</wsp:AppliesTo>
    <wst:Claims Dialect="...">...</wst:Claims>
    <wst:Entropy>
        <wst:BinarySecret>...</wst:BinarySecret>
    </wst:Entropy>
    <wst:Lifetime>
        <wsu:Created>...</wsu:Created>
        <wsu:Expires>...</wsu:Expires>
    </wst:Lifetime>
</wst:RequestSecurityToken>
```

```
<wst:RequestSecurityTokenResponse Context="..." xmlns:wst="...">
    <wst:TokenType>...</wst:TokenType>
    <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>
</wst:RequestSecurityTokenResponse>
```

2) **Cancel**: this operation cancels a token so that it cannot be used anymore when it is no longer needed. After cancellation the STS will not renew or validate the token anymore.

[5] OASIS, WS-SecureConversation specification, version 1.4, 2009, http://docs.oasis-open.org/ws-sx/ws-secureconversation/v1.4/os/ws-secureconversation-1.4-spec-os.pdf

[6] OASIS, WS-SecurePolicy, version 1.3, 2009, http://docs.oasis-open.org/ws-sx/ws-securitypolicy/v1.3/os/ws-securitypolicy-1.3-spec-os.pdf

[7] OASIS, WS-Federation specification, version 1.2 , 2009, http://docs.oasis-open.org/wsfed/federation/v1.2/os/ws-federation-1.2-spec-os.pdf

```
<wst:RequestSecurityToken>
    <wst:RequestType>
        http://docs.oasis-open.org/ws-sx/ws-trust/200512/Cancel
    </wst:RequestType>
    <wst:CancelTarget>
        ...
    </wst:CancelTarget>
</wst:RequestSecurityToken>
```

```
<wst:RequestSecurityTokenResponse>
    <wst:RequestedTokenCancelled/>
</wst:RequestSecurityTokenResponse>
```

3) **Renew**: a previously issued possible expired token is presented and the same token is returned with new expiration semantics. The requestor must either prove authorized use of the token or be trusted by STS to issue third-party renewal requests.

```
<wst:RequestSecurityToken xmlns:wst="...">
    <wst:TokenType>...</wst:TokenType>
    <wst:RequestType>
        http://docs.oasis-open.org/ws-sx/ws-trust/200512/Renew
    </wst:RequestType>
    <wst:RenewTarget>
        ... reference to previously issued token ...
    </wst:RenewTarget>
    <wst:AllowPostdating/>
    <wst:Renewing Allow=”...” OK=”...”/>
</wst:RequestSecurityToken>
```

```
<wst:RequestSecurityTokenResponse xmlns:wst="...">
    <wst:TokenType> ... </wst:TokenType>
    <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>
    <wst:Lifetime>...</wst:Lifetime>
</wst:RequestSecurityTokenResponse>
```

4) **Validate**: this operation evaluates the specified token. The result can be a status, a new token or both.

```
<wst:RequestSecurityToken xmlns:wst="...">
    <wst:TokenType>...</wst:TokenType>
    <wst:RequestType>...</wst:RequestType>
    <wst:ValidateTarget>... </wst:ValidateTarget>
</wst:RequestSecurityToken>
```

```
<wst:RequestSecurityTokenResponse xmlns:wst="..." >
    <wst:TokenType>...</wst:TokenType>
    <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>
    <wst:Status>
        <wst:Code>...</wst:Code>
        <wst:Reason>...</wst:Reason>
    </wst:Status>
</wst:RequestSecurityTokenResponse>
```

WS-Trust does not specify the format of the token issued. Within p-medicine SAML V2.0 tokens should be requested and issued. A SAML profile for WS-Security exists: SAML Token Profile 1.1. This profile defines, amongst other things, how a SAML token can be passed to a web service by using SOAP headers.

```
<--SAML token issued by STS for "End User" targetted to "Service A"-->
```

```
<saml2:Assertion xmlns:saml2="..."
         ID="uuid-ec80226b-0a2e-43bd-a98b-54b071407edd"
         IssueInstant="2012-05-09T09:43:00.292Z" Version="2.0">
    <saml2:Issuer>STS</saml2:Issuer>
    <saml2:Subject>
         <saml2:NameID>End User</saml2:NameID>
    </saml2:Subject>
    <saml2:Conditions NotBefore="..." NotOnOrAfter="...">
         <saml2:AudienceRestriction>
              <saml2:Audience>http://serviceA.com</saml2:Audience>
         </saml2:AudienceRestriction>
    </saml2:Conditions>
    <saml2:AttributeStatement>
         ...
    </saml2:AttributeStatement>
</saml2:Assertion>
```

3.3.2.1 REST Web Services

For REST Web Services there is no standard specifying how to do message level security and brokered authentication yet. As described in 3.3.2.1 the SAML ECP profile can be used for authentication in REST. REST services though are typically stateless while the ECP profile requires a session based security context.

As REST Web Services are actually also web services, a more natural, practical solution would therefore be to look at the WS-* specifications. The Secure Token Service (STS) as defined in WS-Trust can actually also be used with REST. Before calling the REST service a client would then send a SOAP request containing an RST (RequestSecurityToken) to the STS. The STS then returns the identity assertion as a SAML token, embedded in a RSTR (RequestSecurityTokenResponse). In SOAP this SAML token is passed to the calling service through a SOAP header. This is not possible in REST though as there are no SOAP headers. In REST, the HTTP headers and request line are part of the message. An HTTP header is therefore the REST equivalent of a SOAP header. A SAML token can then be passed to a REST service through the HTTP authorization header. The authorization header value should be formatted as follows: "SAML <Base 64 encoded compressed SAML token>". The SAML token is hereby compressed with the zlib compression algorithm. This to ensure that the token fits in the typical 4kb header size limit.

So, to secure a REST service, basically the same SOAP STS service is used as with SOAP services as defined in WS-Trust. REST clients are responsible for retrieving a SAML token from the STS and then passing it through a HTTP authorization header to the REST service. The client somehow needs to find out what the REST service's security policies are (i.e what STS to call). In WS-* a web service publicises its security policies through WS-Policy annotations on the service's WSDL metadata. Such a WSDL file annotated with security policies (by using the WSDL HTTP binding) could actually also be used for REST services. As REST is resource based while SOAP and the WSDL metadata format are action based, it is not an exact match. There is a metadata format, called WADL, which is specifically designed to describe a REST service. No known specification that defines how to add security policies to a WADL metadata file, is available yet. Therefore, in the context of p-medicine, research should be done on whether WS-Policy metadata annotations can be used to add security annotations to a WADL metadata file.

A drawback of using a SOAP STS is that clients are forced to use SOAP to request security tokens. Therefore the secure token service could be better integrated into REST services by restifying it[8]. Such an STS will be specified in a next iteration of this deliverable.

The next step is to guarantee the integrity and confidentiality of REST message. Signing and possibly encrypting the message body is not sufficient though as the HTTP request line and headers are also part of a REST message. There are also no standards yet that specify how to sign a body, the HTTP headers and request line. This will therefore also be an interesting research topic within p-medicine.

3.4 Authorisation

The data protection in p-medicine is based on two pillars.

1. All sensitive patient information imported within p-medicine is de facto anonymised so that patients cannot be re-identified.
2. Access to the de facto anonymised patient data stored in p-medicine is restricted to authorised persons.

P-medicine will technically enforce and govern access control to the de facto anonymised patient data by relying on policy-based authorisation services. These authorisation services are build upon the eXtensible Access Control Markup (XACML) Language. XACML is an OASIS XML-based standard for authorisation and access control for which multiple, both open source and commercial, implementations are available. XACML implements the attribute-based access control (ABAC) model. Attributes associated with a user, action or resource serve in ABAC as input to the decision of whether a given user is allowed to perform a specific action on a given resource. ABAC is capable of meeting "modern" access control demands such as data dependant or environment dependant access policies.

In XACML, an access request is modelled as a “subject” who wants to perform an “action” on a “resource” (subject/action/resource triplet). A policy enforcement point (PEP) intercepts this request and queries the policy decision point (PDP) on whether the user is allowed to perform this action on the resource. The PDP makes a decision based on the request parameters and the available policies. These policies (in the form of XML files) are created and managed by the policy administration point (PAP). The PDP decision goes back to the PEP which is responsible for enforcing it (allowing or denying access).

[8] http://weblogs.asp.net/cibrax/archive/2009/03/06/brokered-authentication-for-rest-active-clients-with-saml.aspx

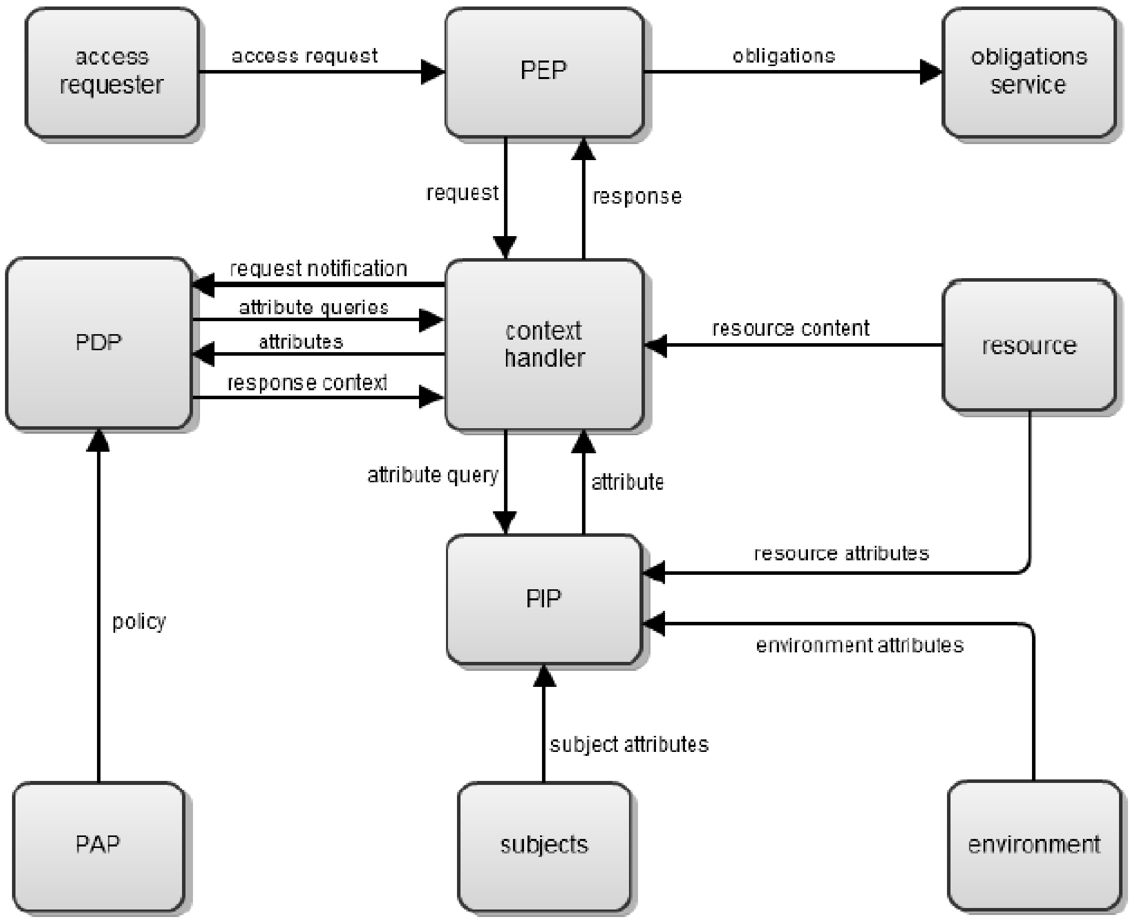

Figure 5 XACML Access Control Model

Although XACML is able to meet many of the "modern" access control demands, XACML also has its share of limitations. There are for example no mechanisms for expressing links (e.g. hierarchy) between different attributes in a convenient way in a XACML policy. Approaches for solving such problems include e.g. use of a strict structural profile in authored policies (e.g. RBAC profile for XACML) or the use of semantic reasoners as policy decision engines with linked to that the use of a security ontology as policy language[9].

Solutions for p-medicine will be based on work done in the INTEGRATE[10] project. The INTEGRATE project will provide the base authorisation security fabric compatible with the P-medicine identity management structure. INTEGRATE tries to solve the XACML limitations, not by proposing language alternatives to XACML, but rather by introducing supporting components complementary to a standard XACML engine. The advantages of this solution are that:

1. the final solution is mainly standard based allowing drop-in replacement of different implementations of the core XACML components.
2. it separates the concern of maintaining the policy decision logic from the development of more advances features.

[9] Rodolfo Ferrini and Elisa Bertino. 2009. Supporting RBAC with XACML+OWL. In Proceedings of the 14th ACM symposium on Access control models and technologies (SACMAT '09). ACM, New York, NY, USA, 145-154.
[10] http://www.fp7-integrate.eu/

This approach has been previously[11] tested with success in the context of attribute translation between XACML policy decision points (PDP's) in different security domains (with a different attribute vocabulary).

3.5 User Management

The user management component in the p-medicine security framework is responsible for user enrolment, user identity and credential management. This component consists of

1. **administration pages** where user administrators can register new users, manage existing users, manage and create organisations, manage organisation membership and enable, disable and remove users.
2. **a user profile** page where users can manage their own identity attributes such as first name, last name, email address.
3. **a public registration page** where users can register themselves. A user who registers himself through the public registration pages first needs to be accepted by a user administrator before he can create credentials.
4. **an activation page** where users can choose a username and password. After a user is registered and, in case of public registration, accepted by a user administrator, the user receives an activation mail. Through a link in this mail the user is guided to the activation page where he can chose his username and password.
5. **credential management pages** where users can request username or password recovery.

3.6 Delegation

Through delegation an end user U can allow a service A to access a service B on his/her behalf in a limited context (i.e. limited in time, limited in action, etc.). To support this a service A can request a delegation assertion from the STS or IdP. This delegation assertion will then state the identity of the current client (service A) and the identity of the user on whose behalf the client is acting (end user U).

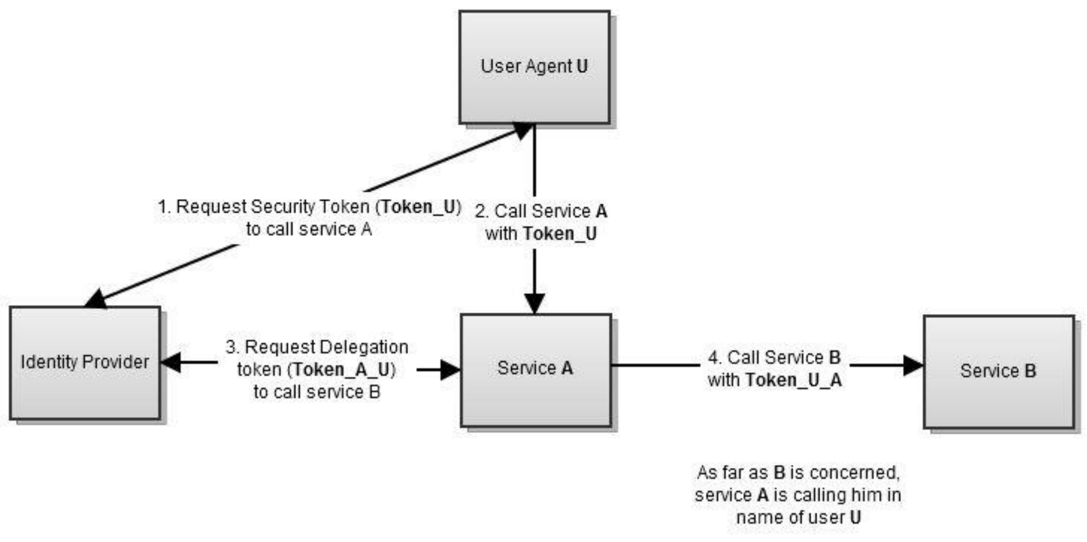
User Agent U
1. Request Security Token (Token_U) to call service A
2. Call Service A with Token_U
Identity Provider
3. Request Delegation token (Token_A_U) to call service B
Service A
4. Call Service B with Token_U_A
Service B
As far as B is concerned, service A is calling him in name of user U

Figure 6 Credential Delegation

[11] I. Ciuci, B. Claerhout, L. Schilders, R. Meersman. 2011. Ontology-Based Matching of Security Attributes for Personal Data Access in e-Health

3.6.1 Delegation in REST

Normally when a REST client calls a REST service, it will present a SAML token identifying the client. Before calling a service, a client will therefore request such an identity token from the STS by sending a WS-Trust RequestSecurityToken request, as described in 3.3.2. The STS will respond with a RequestSecurityTokenResponse containing the SAML identity token.

When doing delegation though, the client should not represent a token identifying itself but instead a token identifying the end user with the client as intermediate. To get such a token the client (in this example called 'Service A') sends to the STS a RequestSecurityToken for 'Service B' with a SAML token identifying the end user embedded in an ActAs element as defined in WS-Trust.

```
<trust:RequestSecurityToken xmlns:trust="http://docs.oasis-open.org/ws-
sx/ws-trust/200512" xmlns:sc="http://docs.oasis-open.org/ws-sx/ws-
secureconversation/200512"
xmlns:wsa="http://www.w3.org/2005/08/addressing"
xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
wssecurity-secext-1.0.xsd"
xmlns:wst="http://schemas.xmlsoap.org/ws/2005/02/trust"
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
wssecurity-utility-1.0.xsd">
      <trust:RequestType>
            http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
      </trust:RequestType>
      <wsp:AppliesTo>
            <wsa:EndpointReference>
                  <wsa:Address>http://serviceB.com</wsa:Address>
            </wsa:EndpointReference>
      </wsp:AppliesTo>
      <wst14:ActAs xmlns:wst14="http://docs.oasis-open.org/ws-sx/ws-
trust/200802">
            <saml2:Assertion
xmlns:saml2="urn:oasis:names:tc:SAML:2.0:assertion"
xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
xmlns:exc14n="http://www.w3.org/2001/10/xml-exc-c14n#"
xmlns:wsse11="http://docs.oasis-open.org/wss/oasis-wss-wssecurity-secext-
1.1.xsd" xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
xmlns:xs="http://www.w3.org/2001/XMLSchema" ID="uuid-ec80226b-0a2e-43bd-
a98b-54b071407edd" IssueInstant="2012-05-09T09:43:00.292Z" Version="2.0">
                  <saml2:Issuer>IdP</saml2:Issuer>
                  <saml2:Subject>
                        <saml2:NameID>End User</saml2:NameID>
                  </saml2:Subject>
                  <saml2:Conditions NotBefore="..." NotOnOrAfter="...">
                        <saml2:AudienceRestriction>
      <saml2:Audience> http://serviceA.com</</saml2:Audience>
                        </saml2:AudienceRestriction>
                  </saml2:Conditions>
                  <saml2:AttributeStatement>
                        ...
                  </saml2:AttributeStatement>
            </saml2:Assertion>
      </wst14:ActAs>
      ....
</trust:RequestSecurityToken>
```

The STS responds with a RequestSecurityTokenResponse containing the SAML token which identifies the end user with the REST client "Service A" as intermediate. As the newly issued token is targeted to "Service B" the audience is restricted to that "service B".

```
<saml2:Assertion xmlns:saml2="urn:oasis:names:tc:SAML:2.0:assertion"
xmlns:del="urn:oasis:names:tc:SAML:2.0:conditions:delegation" ID="..."
IssueInstant="..." Version="2.0">
      <saml2:Issuer>STS</saml2:Issuer>
      <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
            ...
      </ds:Signature>
      <saml2:Subject>
            <saml2:NameID>End User</saml2:NameID>
      </saml2:Subject>
      <saml2:Conditions NotBefore="..." NotOnOrAfter="...">
            <saml2:AudienceRestriction>
      <saml2:Audience>ServiceB</saml2:Audience>
            </saml2:AudienceRestriction>
            <saml2:Condition xmlns:ns7="http://www.w3.org/2001/XMLSchema-
instance" ns7:type="del:DelegationRestrictionType">
                  <del:Delegate>
                        <del:NameID>ServiceA</ns1:NameID>
                  </del:Delegate>
            </saml2:Condition>
      </saml2:Conditions>
      <saml2:AttributeStatement>
            ...
      </saml2:AttributeStatement>

</saml2:Assertion>
```

The specification "SAML V2.0 Condition for Delegation"[12] extends SAML with a condition statement through which an intermediate service which acts in name of the end user can be defined.

[12] http://docs.oasis-open.org/security/saml/Post2.0/sstc-saml-delegation-cs-01.pdf

4 Privacy Framework

4.1 Introduction

The goal of the privacy framework or pseudonymisation platform is to pseudonymise the data[13], before delivering it to the p-medicine research domain. All data that is passed to the p-medicine platform must therefore pass through a state of the art pseudonymisation platform. This implies that all tools and software components that import data to the research platform must integrate with the pseudonymisation platform.

There are two rounds of pseudonymisation. A first round is performed within the boundaries of the treatment domain. Afterwards the data is passed to a Trusted Third Party (TTP) that performs a second round of pseudonymisation before delivering it to the p-medicine research domain. CATS (Custodix Anonymisation Tool) will be used as default pseudonymisation tool; however hospitals are free to choose whatever pseudonymisation tool they want to use, as long as it has been approved by the Center for Data Protection (CDP)[14].

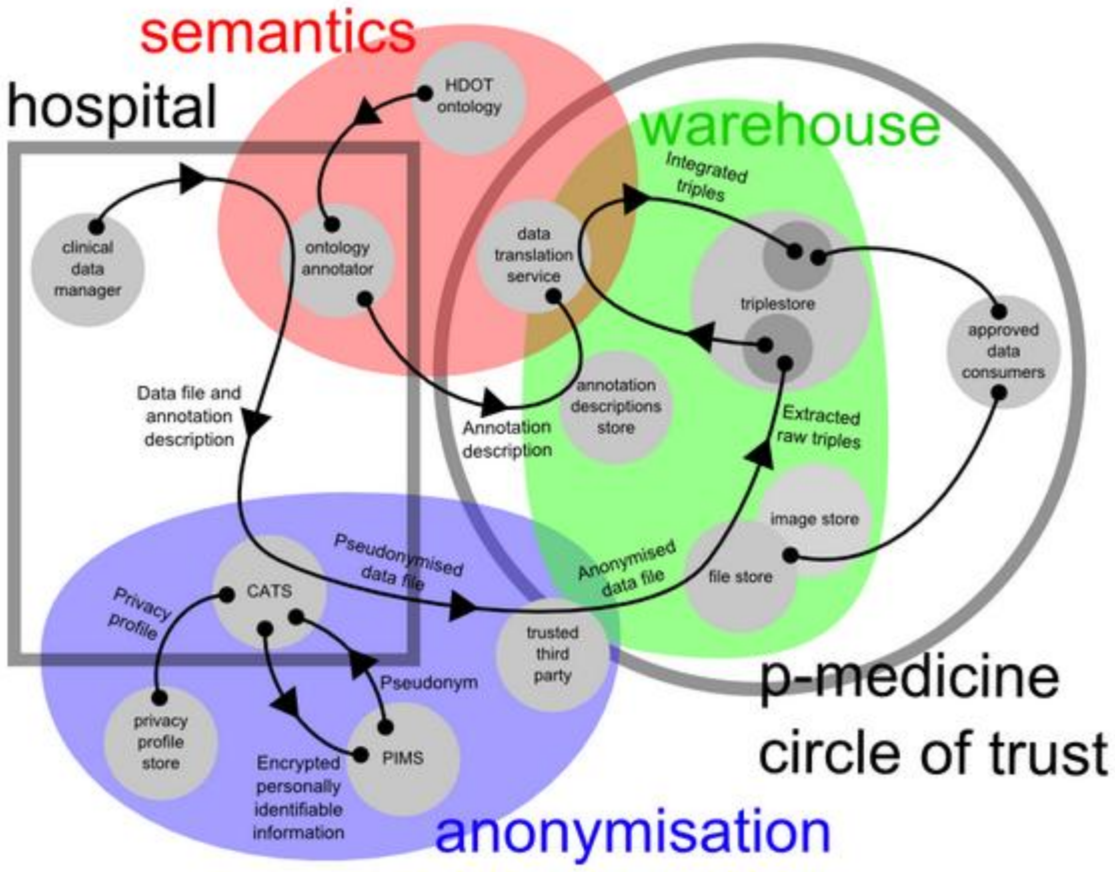

Figure 7 p-medicine data warehouse architecture

[13] de facto anonymous as defined in deliverable 5.1

[14] see deliverable 5.1

The figure above depicts the conceptual architecture of the p-medicine data warehouse and its key components. Data (clinical, molecular, DICOM, etc) from the hospitals flows to the data warehouse, passing through the anonymisation domain. Data files are uploaded to CATS which pseudonymise the files. CATS closely interacts with PIMS, the patient information management system. In short PIMS' responsibility is twofold:

- to store a link between the personally identifiable information and the patient pseudonym
- to match and link patient records representing the same real-world person, coming from different sources, to one index. This is called building an Master Patient Index (MPI)

After CATS processed a data file it will be send to the TTP for a second round of pseudonymisation to complete the de facto anonymisation. The TTP will transform all pseudonyms in a given data file to a p-medicine specific pseudonym. Re-identification of this pseudonym is only possible by passing through the TTP again.

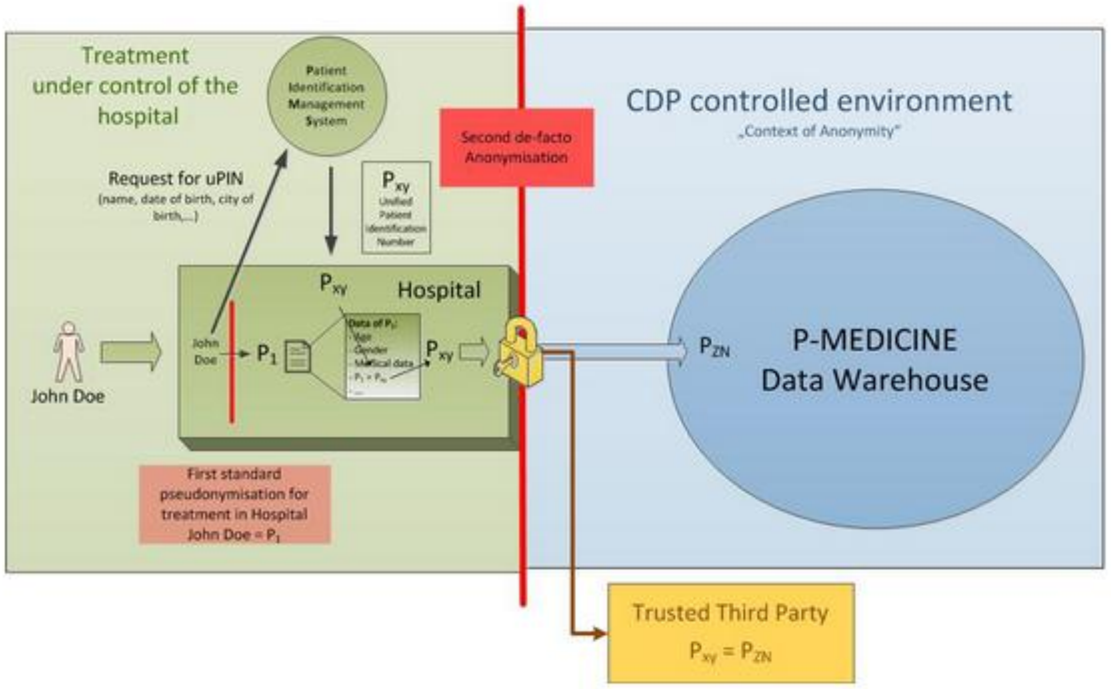
Treatment under control of the hospital
Patient Identification Management System
Second de-facto Anonymisation
CDP controlled environment
„Context of Anonymity"
Request for uPIN
(name, date of birth, city of birth,...)
P_{xy} Unified Patient Identification Number
P_{xy}
Hospital
John Doe
John Doe
P_1
Data of P_1:
-Age
-Gender
-Medical data
-$P_1 = P_{xy}$
P_{xy}
P_{ZN}
P-MEDICINE Data Warehouse
First standard pseudonymisation for treatment in Hospital John Doe = P_1
Trusted Third Party
$P_{xy} = P_{ZN}$

Figure 8 Schematic representation of the two pseudonymisation rounds.

4.2 CATS

CATS is a tool, developed by Custodix, responsible for the de-identification or anonymisation of (clinical) data files. Based on the mime type of the data files and a pre-configured set of transformation rules (privacy profiles), data files are anonymised or pseudonymised. Key transformation rules are:

- scan for patient identifying data and replace with a pseudonym. The reference between the patient data and the pseudonym can be stored at PIMS or locally.
- clear patient identifying data.
- encrypt (parts of) the result file.

4.3 PIMS

PIMS core task is twofold:

- issuing pseudonyms and storing the link between the issued pseudonyms and the patient identifying data.
- patient record linkage, meaning that records that represent the same real-world person are linked to the same index (Master Patient Index or MPI).

While CATS is processing a data file, it will request pseudonyms from PIMS, providing PIMS with patient identifying data. The patient data is stored as a patient record in a secured, PIMS-dedicated database. The issued pseudonym is valid within the domain of the source from which the file originate, a so called "source pseudonym". Afterwards PIMS will try to match the new patient record with previously stored records possibly originating from other sources (hospitals). If a match is found patient records are linked and a global pseudonym valid over multiple sources may be issued, a so called treatment pseudonym.

So to conclude:

- A source pseudonym uniquely identifies a patient record within its originating source. A real life patient may have multiple source pseudonyms in the same source. Source pseudonyms are linked to one patient record only.
- A treatment pseudonym uniquely identifies a patient record within the whole treatment domain. A real life patient has only one treatment pseudonym in the treatment domain. Treatment pseudonyms are linked to multiple patient records originating from multiple sources.

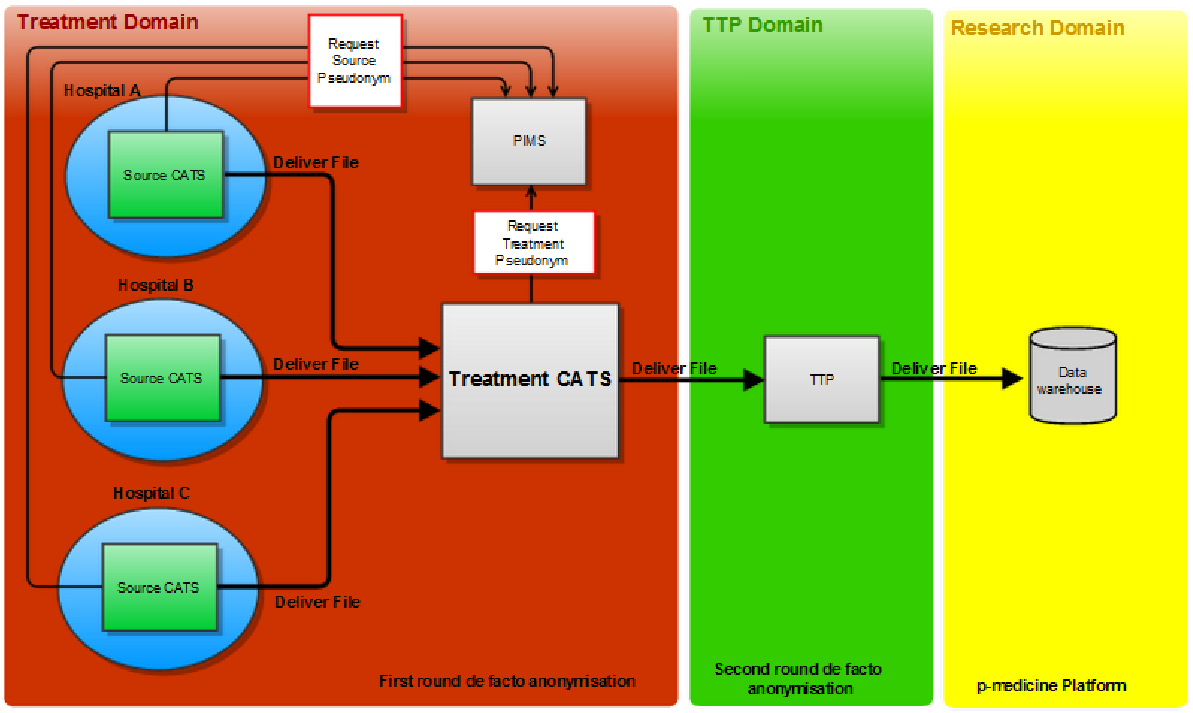

Figure 9 diagram which clarifies the concept of source and treatment pseudonyms

In the diagram above the CATS platform is split in a "source CATS" and a treatment CATS. Each hospital (or source) has its own instance of source CATS installed. Within the treatment domain there is only one instance of the treatment CATS. There is only one instance of PIMS as well, which makes sense because PIMS must issue source pseudonyms as well as treatment pseudonyms and needs to link patient records originating from different hospitals. There are three hospitals, each with their own source CATS instance. The source CATS instances will de-identify data files, feeding patient identifying data to and receiving source pseudonyms from PIMS. PIMS will build an internal MPI by matching all patient records coming from the three sources. After file processing, the source CATS will upload its files to

the treatment CATS. This CATS server will reprocess all files and replace the source pseudonyms with treatment pseudonyms for the whole treatment domain. As such, the same real life patient present in all three hospitals receives the same global pseudonym. The treatment CATS will then upload its processed files to the TTP.
Types of pseudonyms:

- A **source pseudonym:** is issued by a source CATS and is only valid within the source (or hospital).
- A **treatment pseudonym**: is always issued by the treatment CATS and is valid over the whole treatment domain and its participating sources.
- A **p-medicine pseudonym**: is issued by the TTP. In fact this is an encrypted treatment pseudonym. It is only valid within the research domain of p-medicine.

5 Integration with Authentication Framework

Any service which is full SAML 2.0 compliant can seamlessly integrate itself into the p-medicine authentication framework by registering itself to the IdP. Through this registration process the service will provide the IdP its SAML metadata. The exact registration protocol and flow has not yet been defined. Therefore initially a service can register itself by mailing its SAML metadata to pmedicine-support@custodix.com. Custodix will then configure the IdP to accept this service.

A website can be integrated as a service provider (SP) in p-medicine if it supports the following SAML V2.0 Profiles:

1) The website must support the following bindings of the Web Browser SSO Profile:
   a) for <AuthnRequest> from SP to IdP:
      i) HTTP redirect
      ii) HTTP POST
   b) for IdP <Response> to SP:
      i) HTTP POST
      ii) HTTP artifact
2) The website must support at least the following bindings of the Single Logout Profile:
   a) for IdP-initiated single logout the HTTP redirect and SOAP binding must be supported.
   b) for SP-initiated single logout the HTTP redirect and SOAP binding must be supported.
3) The website must publish its SAML metadata.

```
<!-- Sample metadata of a compatible service provider -->
<?xml version="1.0" encoding="UTF-8"?>
<md:EntityDescriptor entityID="..."
    xmlns:md="urn:oasis:names:tc:SAML:2.0:metadata">
    <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        ...
    </ds:Signature>
    <md:SPSSODescriptor AuthnRequestsSigned="true"
        WantAssertionsSigned="true"
protocolSupportEnumeration="urn:oasis:names:tc:SAML:2.0:protocol">
        <md:KeyDescriptor use="signing">
            <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
                <ds:X509Data>
                    <ds:X509Certificate>...</ds:X509Certificate>
                </ds:X509Data>
            </ds:KeyInfo>
        </md:KeyDescriptor>
        <md:KeyDescriptor use="encryption">
            <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
                <ds:X509Data>
                    <ds:X509Certificate>...</ds:X509Certificate>
                </ds:X509Data>
            </ds:KeyInfo>
        </md:KeyDescriptor>
        <md:SingleLogoutService
            Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Redirect"
            Location="..." />
        <md:SingleLogoutService
Binding="urn:oasis:names:tc:SAML:2.0:bindings:SOAP"
            Location="..." />
        <md:NameIDFormat>urn:oasis:names:tc:SAML:2.0:nameid-
format:persistent</md:NameIDFormat>
        <md:AssertionConsumerService
            Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"
            Location="..."
            index="0" isDefault="true" />
        <md:AssertionConsumerService
            Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Artifact"
            Location="http..."
            index="1" isDefault="false" />
    </md:SPSSODescriptor>
</md:EntityDescriptor>
```

To integrate a REST Web Service as a service provider (SP) in p-medicine:

1) the REST service should be able to process a SAML v2.0 identity token passed to it with each REST call through the HTTP authorization header.
2) the REST service should initially publish its security policies in a WSDL metadata file. Later on in the project the REST services might be able to publish those security policies through a WADL metadata file.

6 Service Provider Implementation

6.1 Example: Java Spring Security

A website secured by Spring Security can be made SAML complaint by using and configuring the Spring Security SAML extension[15].

6.1.1 Configure your project to use Spring and Spring Security

1) Add the required Spring and Spring Security dependencies to your project.
   a) bouncy castle provider 145
   b) commons codec 1.3
   c) commons collections 3.1
   d) commons httpclient 3.1
   e) commons lang 2.4
   f) commons logging 1.1
   g) joda time 1.6.2
   h) jstl 1.2
   i) not yet commons ssl 0.3.9
   j) opensaml 2 2.4.1
   k) openws 1.4.1
   l) spring framework 3.0.5
   m) slf4j api 1.6.1
   n) spring security 3.0.7
   o) spring security saml extension (trunk or Custodix Modified Extension)
   p) velocity 1.5
   q) xalan 2.7.0
   r) xerces impl 2.9.0
   s) xml-apis 2.0.2
   t) xml security 1.4.4
   u) xml tooling 1.3.1
2) Configure your web project to use spring by adding:
   a) a context parameter to WEB-INF/web.xml defined the location of the spring configuration files.

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        WEB-INF/applicationContext.xml
        WEB-INF/applicationContext-security.xml
        WEB-INF/applicationContext-security-saml.xml
    </param-value>
```

[15] http://static.springsource.org/spring-security/site/extensions/saml/index.html

```
</context-param>
```

b) the spring security filter to WEB-INF/web.xml.

```
<filter>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-
class>org.springframework.web.filter.DelegatingFilterProxy</filter-
class>
</filter>
<filter-mapping>
    <filter-name>springSecurityFilterChain</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

c) the spring context listener

```
<listener>
    <listener-
class>org.springframework.web.context.ContextLoaderListener</listener
-class>
</listener>
```

3) Create the following configuration files in the project's WEB-INF folder:

a) the spring global configuration file 'applicationContext.xml'.

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
            http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

    <!-- Enable autowiring -->
    <context:annotation-config/>

    <!-- Load in the security properties with SAML Service Provider
Configuration -->
    <bean id="daoPropertyConfigurer"

class="org.springframework.web.context.support.ServletContextProperty
PlaceholderConfigurer">
        <property name="locations">
            <list>
                    <value>/WEB-INF/security.properties</value>
            </list>
        </property>
        <property name="contextOverride" value="true"/>
        <property name="fileEncoding" value="UTF-8"/>
    </bean>
</beans>
```

b) the spring security configuration file 'applicationContext-security.xml'.

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:security="http://www.springframework.org/schema/security"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/security
http://www.springframework.org/schema/security/spring-security-
3.0.xsd
        http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

    <security:http entry-point-ref="samlEntryPoint">
        <!-- Spring security interceptors -->
        <security:intercept-url pattern="/**"
                                access="IS_AUTHENTICATED_FULLY"/>
        <security:intercept-url pattern="/favicon.ico"
                                filters="none"/>
        <!-- Custom filters for SAML -->
        <security:custom-filter before="PRE_AUTH_FILTER"
                                ref="metadataFilter"/>
        <security:custom-filter position="PRE_AUTH_FILTER"
                                ref="samlEntryPoint"/>
        <security:custom-filter after="BASIC_AUTH_FILTER"
                                ref="samlProcessingFilter"/>
        <security:custom-filter after="LOGOUT_FILTER"
                                ref="samlLogoutFilter"/>
        <security:custom-filter before="LOGOUT_FILTER"
                                ref="samlLogoutProcessingFilter"/>
    </security:http>

    <!-- Handler deciding where to redirect user after successful
login -->
    <bean id="successRedirectHandler"

class="org.springframework.security.web.authentication.SavedRequestAw
areAuthenticationSuccessHandler">
        <property name="defaultTargetUrl"
value="${security.successRedirectHandler.defaultTargetUrl}"/>
    </bean>

    <!-- Handler for successful logout -->
    <bean id="successLogoutHandler"

class="org.springframework.security.web.authentication.logout.SimpleU
rlLogoutSuccessHandler">
        <property name="defaultTargetUrl"
value="${security.successLogoutHandler.defaultTargetUrl}"/>
    </bean>

    <!-- Register authentication manager with SAML provider -->
    <security:authentication-manager alias="authenticationManager">
        <security:authentication-provider
                        ref="samlAuthenticationProvider"/>
    </security:authentication-manager>

    <!-- Logout handler terminating local session -->
    <bean id="logoutHandler"

class="org.springframework.security.web.authentication.logout.Securit
yContextLogoutHandler">
        <property name="invalidateHttpSession" value="false"/>
    </bean>
</beans>
```

c) the SAML security configuration file 'applicationContext-security-saml.xml'.

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:security="http://www.springframework.org/schema/security"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
              http://www.springframework.org/schema/security
http://www.springframework.org/schema/security/spring-security-
3.0.xsd
              http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">
    <!-- Logger for SAML messages and events -->
    <bean id="samlLogger"
class="org.springframework.security.saml.log.SAMLDefaultLogger"/>
    <!-- Central storage of cryptographic keys -->
    <bean id="keyManager"
class="org.springframework.security.saml.key.JKSKeyManager">
        <constructor-arg value="${security.saml.keystore}"/>
        <constructor-arg type="java.lang.String"
value="${security.saml.keystore.password}"/>
        <constructor-arg>
            <map>
                <entry key="${security.saml.keystore.key.alias}"
value="${security.saml.keystore.key.password}"/>
            </map>
        </constructor-arg>
        <constructor-arg type="java.lang.String"
value="${security.saml.keystore.defaultKey}"/>
    </bean>
    <!-- Entry point to initialize authentication, default values
taken from properties file -->
    <bean id="samlEntryPoint"
class="org.springframework.security.saml.SAMLEntryPoint">
        <property name="filterSuffix"
value="${security.saml.entryPoint.filterSuffix}"/>
        <!-- OPTIONAL property: In case idpSelectionPath property is
not set the user will be redirected to the default IDP -->
        <!--property name="idpSelectionPath" value="/WEB-
INF/security/idpSelection.jsp"/-->
        <property name="defaultProfileOptions">
            <bean
class="org.springframework.security.saml.websso.WebSSOProfileOptions"
>
                <property name="includeScoping" value="false"/>
                <property name="binding"
value="${security.saml.entryPoint.binding}"/>
            </bean>
        </property>
    </bean>
    <!-- OPTIONAL bean: The filter is waiting for connections on URL
suffixed with filterSuffix and presents SP metatdata there -->
    <bean id="metadataFilter"
class="org.springframework.security.saml.metadata.MetadataDisplayFilt
er">
        <property name="filterSuffix"
value="${security.saml.metadata.filterSuffix}"/>
```

```
    </bean>
    <!-- Class is capable of generating SP metadata describing the
currently running environnment -->
    <bean id="metadataGenerator"
class="org.springframework.security.saml.metadata.MetadataGenerator">
        <property name="entityId" value="${security.saml.sp.id}"/>
        <property name="entityBaseURL"
value="${security.saml.sp.url}"/>
        <property name="entityAlias"
value="${security.saml.sp.alias}"/>
        <property name="signingKey"
value="${security.saml.sp.signingKeyAlias}"/>
        <property name="encryptionKey"
value="${security.saml.sp.encryptionKeyAlias}"/>
        <property name="tlsKey" value="${security.saml.sp.tlsKey}"/>
        <property name="requestSigned" value="true"/>
        <property name="signMetadata" value="true"/>
        <property name="wantAssertionSigned" value="true"/>
        <property name="includeDiscovery" value="true"/>
    </bean>
    <!-- SAML Authentication Provider responsible for validating of
received SAML messages -->
    <bean id="samlAuthenticationProvider"
class="org.springframework.security.saml.SAMLAuthenticationProvider">
        <!-- OPTIONAL property: can be used to store/load user data
after login -->
        <!--
        <property name="userDetails" ref="bean" />
        -->
    </bean>
    <!-- Provider of default SAML Context -->
    <bean id="contextProvider"
class="org.springframework.security.saml.context.SAMLContextProviderI
mpl"/>
    <!-- Override default authentication processing filter with the
one processing SAML messages -->
    <bean id="samlProcessingFilter"
class="org.springframework.security.saml.SAMLProcessingFilter">
        <property name="authenticationManager"
ref="authenticationManager"/>
        <property name="authenticationSuccessHandler"
ref="successRedirectHandler"/>
    </bean>
    <!-- Override default logout processing filter with the one
processing SAML messages -->
    <bean id="samlLogoutFilter"
class="org.springframework.security.saml.SAMLLogoutFilter">
        <constructor-arg ref="successLogoutHandler"/>
        <constructor-arg ref="logoutHandler"/>
        <constructor-arg ref="logoutHandler"/>
    </bean>
    <!-- Filter processing incoming logout messages -->
    <!-- First argument determines URL user will be redirected to
after successful global logout -->
    <bean id="samlLogoutProcessingFilter"
class="org.springframework.security.saml.SAMLLogoutProcessingFilter">
        <constructor-arg ref="successLogoutHandler"/>
        <constructor-arg ref="logoutHandler"/>
    </bean>
    <!-- Class loading incoming SAML messages from httpRequest stream
-->
```

```
    <bean id="processor"
class="org.springframework.security.saml.processor.SAMLProcessorImpl"
>
        <constructor-arg>
            <list>
                <ref bean="redirectBinding"/>
                <ref bean="postBinding"/>
                <ref bean="artifactBinding"/>
                <ref bean="soapBinding"/>
                <ref bean="paosBinding"/>
            </list>
        </constructor-arg>
    </bean>
    <!-- SAML 2.0 Assertion Consumer -->
    <bean id="webSSOprofileConsumer"
class="org.springframework.security.saml.websso.WebSSOProfileConsumer
Impl"/>
    <!-- SAML 2.0 Web SSO profile -->
    <bean id="webSSOprofile"
class="org.springframework.security.saml.websso.WebSSOProfileImpl"/>
    <!-- SAML 2.0 ECP profile -->
    <bean id="ecpprofile"
class="org.springframework.security.saml.websso.WebSSOProfileECPImpl"
/>
    <!-- SAML 2.0 Logout Profile -->
    <bean id="logoutprofile"
class="org.springframework.security.saml.websso.SingleLogoutProfileIm
pl"/>
    <!-- Bindings, encoders and decoders used for creating and
parsing messages -->
    <bean id="postBinding"
class="org.springframework.security.saml.processor.HTTPPostBinding">
        <constructor-arg ref="parserPool"/>
        <constructor-arg ref="velocityEngine"/>
    </bean>
    <bean id="redirectBinding"
class="org.springframework.security.saml.processor.HTTPRedirectDeflat
eBinding">
        <constructor-arg ref="parserPool"/>
    </bean>
    <bean id="artifactBinding"
class="org.springframework.security.saml.processor.HTTPArtifactBindin
g">
        <constructor-arg ref="parserPool"/>
        <constructor-arg ref="velocityEngine"/>
        <constructor-arg>
            <bean
class="org.springframework.security.saml.websso.ArtifactResolutionPro
fileImpl">
                <constructor-arg>
                    <bean
class="org.apache.commons.httpclient.HttpClient"/>
                </constructor-arg>
                <property name="processor">
                    <bean id="soapProcessor"
class="org.springframework.security.saml.processor.SAMLProcessorImpl"
>
                        <constructor-arg ref="soapBinding"/>
                    </bean>
                </property>
            </bean>
        </constructor-arg>
    </bean>
    <bean id="soapBinding"
class="org.springframework.security.saml.processor.HTTPSOAP11Binding"
>
        <constructor-arg ref="parserPool"/>
    </bean>
    <bean id="paosBinding"
class="org.springframework.security.saml.processor.HTTPPAOS11Binding"
>
        <constructor-arg ref="parserPool"/>
    </bean>
    <!-- Initialization of OpenSAML library-->
    <bean class="org.springframework.security.saml.SAMLBootstrap"/>
    <!-- Initialization of the velocity engine -->
    <bean id="velocityEngine"
class="org.springframework.security.saml.util.VelocityFactory"
          factory-method="getEngine"/>
    <!-- XML parser pool needed for OpenSAML parsing -->
    <bean id="parserPool"
class="org.opensaml.xml.parse.BasicParserPool" scope="singleton"/>
    <!-- IDP Metadata configuration - paths to metadata of IDPs in
circle of trust is here -->
    <!-- Do no forget to call iniitalize method on providers -->
    <bean id="metadata"
class="org.springframework.security.saml.metadata.CachingMetadataMana
ger">
        <constructor-arg>
            <null/>
        </constructor-arg>
        <property name="providers">
            <list>
                <bean
class="org.opensaml.saml2.metadata.provider.HTTPMetadataProvider">
                    <!-- URL containing the metadata -->
                    <constructor-arg>
                        <value
type="java.lang.String">${security.saml.idp.metadataURL}</value>
                    </constructor-arg>
                    <!-- Timeout for metadata loading in ms -->
                    <constructor-arg>
                        <value type="int">5000</value>
                    </constructor-arg>
                    <property name="parserPool" ref="parserPool"/>
                </bean>
                <bean
class="org.springframework.security.saml.metadata.ExtendedMetadataDel
egate">
                    <constructor-arg>
                        <bean
class="org.springframework.security.saml.metadata.MetadataMemoryProvi
der">
                            <constructor-arg>
                                <bean factory-
bean="metadataGenerator" factory-method="generateMetadata"/>
                            </constructor-arg>
                        </bean>
                    </constructor-arg>
                    <constructor-arg>
                        <bean
class="org.springframework.security.saml.metadata.ExtendedMetadata">
```

```
                            <property name="local" value="true"/>
                            <property name="alias"
value="${security.saml.sp.alias}"/>
                            <property name="securityProfile"
value="metaiop"/>
                            <property name="signingKey"
value="${security.saml.sp.signingKeyAlias}"/>
                            <property name="encryptionKey"
value="${security.saml.sp.encryptionKeyAlias}"/>
                            <property name="tlsKey"
value="${security.saml.sp.tlsKey}"/>
                            <property
name="requireArtifactResolveSigned" value="true"/>
                            <property
name="requireLogoutRequestSigned" value="true"/>
                            <property
name="requireLogoutResponseSigned" value="true"/>
                        </bean>
                    </constructor-arg>
                </bean>
            </list>
        </property>
        <!-- OPTIONAL used when one of the metadata files contains
information about this service provider -->
        <!-- <property name="hostedSPName" value=""/> -->
        <!-- OPTIONAL property: can tell the system which IDP should
be used for authenticating user by default. -->
        <!-- <property name="defaultIDP"
value="http://localhost:8080/opensso"/> -->
    </bean>
</beans>
```

d) the property configuration files 'security.properties' in which all property templates used in above configuration files are configured.

```
#URL to redirect to after succesful authentication
security.successRedirectHandler.defaultTargetUrl=/
#URL to redirect to after succesful logout
security.successLogoutHandler.defaultTargetUrl=/
#Whether sessions should be invalidated after logout
security.logout.invalidateHttpSession=true
#Suffix of the login filter, saml authentication is initiated when
user browses to this url
security.saml.entryPoint.filterSuffix=/saml/login
#SAML Binding to be used for above entry point url.
security.saml.entryPoint.binding=urn\:oasis\:names\:tc\:SAML\:2.0\:bi
ndings\:HTTP-POST
#Suffix of the Service Provider's metadata, this url needs to be
configured on IDP
security.saml.metadata.filterSuffix=/saml/metadata
#Alias of the Service Provider
security.saml.sp.alias=SampleService
#ID of the Service Provider
security.saml.sp.id=SampleService
#Alias of the Service Provider's signing key
security.saml.sp.signingKeyAlias=apollo
#URL of the service provider
security.saml.sp.url=http\://127.0.0.1\:8080/SP
#URL to the IDP's metadata
security.saml.idp.metadataURL=http\://dev-pmed-
vm.custodix.com/idp/shibboleth
```

```
#Alias of the Service Provider's tls key
security.saml.sp.tlsKey=apollo
#Alias of the Service Provider's encryption key
security.saml.sp.encryptionKeyAlias=apollo
#path to keystore which contains keys used by the Service Provider
security.saml.keystore=classpath\:resources/samlKeystore.jks
#keystore's storepass
security.saml.keystore.password=nalle123
#keystore's default key
security.saml.keystore.defaultKey=apollo
#Alias of a key in the keystore
security.saml.keystore.key.alias=apollo
#Password of that the key with above alis in the keystore
security.saml.keystore.key.password=nalle123
#Note that it's only possible to configure one key alias through
property file
#If different keys are used for encryption, signing, ...
#the applicationContext-spring-security-saml needs to be updated.
```

6.1.2 Register website's metadata

The above configuration results in a website which publishes its metadata on 'http://localhost:8080/SP/saml/metadata.

```
<?xml version="1.0" encoding="UTF-8"?>
<md:EntityDescriptor entityID="http://127.0.0.1:8080/SP"
    xmlns:md="urn:oasis:names:tc:SAML:2.0:metadata">
    <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        <ds:SignedInfo>
            <ds:CanonicalizationMethod
Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
            <ds:SignatureMethod
Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
            <ds:Reference URI="">
                <ds:Transforms>
                    <ds:Transform

Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature" />
                    <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-
exc-c14n#" />
                </ds:Transforms>
                <ds:DigestMethod
Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
                <ds:DigestValue>...</ds:DigestValue>
            </ds:Reference>
        </ds:SignedInfo>
        <ds:SignatureValue>...</ds:SignatureValue>
        <ds:KeyInfo>
            <ds:X509Data>
                <ds:X509Certificate>...</ds:X509Certificate>
            </ds:X509Data>
        </ds:KeyInfo>
    </ds:Signature>
    <md:SPSSODescriptor AuthnRequestsSigned="true"
        WantAssertionsSigned="true"
protocolSupportEnumeration="urn:oasis:names:tc:SAML:2.0:protocol">
        <md:Extensions>
            <idpdisco:DiscoveryResponse
                Binding="urn:oasis:names:tc:SAML:profiles:SSO:idp-
discovery-protocol"
```

```
Location="http://127.0.0.1:8080/SP/saml/login/alias/SampleService_Localhost
?disco=true"
                xmlns:idpdisco="urn:oasis:names:tc:SAML:profiles:SSO:idp-
discovery-protocol" />
        </md:Extensions>
        <md:KeyDescriptor use="signing">
            <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
                <ds:X509Data>
                    <ds:X509Certificate>...</ds:X509Certificate>
                </ds:X509Data>
            </ds:KeyInfo>
        </md:KeyDescriptor>
        <md:KeyDescriptor use="encryption">
            <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
                <ds:X509Data>
                    <ds:X509Certificate>...</ds:X509Certificate>
                </ds:X509Data>
            </ds:KeyInfo>
        </md:KeyDescriptor>
        <md:SingleLogoutService
            Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"

Location="http://127.0.0.1:8080/SP/saml/SingleLogout/alias/SampleService_Lo
calhost" />
        <md:SingleLogoutService
            Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Redirect"

Location="http://127.0.0.1:8080/SP/saml/SingleLogout/alias/SampleService_Lo
calhost" />
        <md:SingleLogoutService
Binding="urn:oasis:names:tc:SAML:2.0:bindings:SOAP"

Location="http://127.0.0.1:8080/SP/saml/SingleLogout/alias/SampleService_Lo
calhost" />
        <md:NameIDFormat>urn:oasis:names:tc:SAML:1.1:nameid-
format:emailAddress</md:NameIDFormat>
        <md:NameIDFormat>urn:oasis:names:tc:SAML:2.0:nameid-
format:transient</md:NameIDFormat>
        <md:NameIDFormat>urn:oasis:names:tc:SAML:2.0:nameid-
format:persistent</md:NameIDFormat>
        <md:NameIDFormat>urn:oasis:names:tc:SAML:1.1:nameid-
format:unspecified</md:NameIDFormat>
        <md:NameIDFormat>urn:oasis:names:tc:SAML:1.1:nameid-
format:X509SubjectName</md:NameIDFormat>
        <md:AssertionConsumerService
            Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"

Location="http://127.0.0.1:8080/SP/saml/SSO/alias/SampleService_Localhost"
            index="0" isDefault="true" />
        <md:AssertionConsumerService
            Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Artifact"

Location="http://127.0.0.1:8080/SP/saml/SSO/alias/SampleService_Localhost"
            index="1" isDefault="false" />
        <md:AssertionConsumerService
            Binding="urn:oasis:names:tc:SAML:2.0:bindings:PAOS"

Location="http://127.0.0.1:8080/SP/saml/SSO/alias/SampleService_Localhost"
            index="2" isDefault="false" />
    </md:SPSSODescriptor>
```

```
</md:EntityDescriptor>
```

The next step is to register this service's metadata by mailing it to pmedicine-support@custodix.com. This medata defines:

- that the SP requires assertions to be signed.
- that the SP will issue signed authentication requests.
- the certificate used to verify the PS's signature.
- the key that the IdP can/should use to encrypt its responses.
- the SP's single logout end points for the HTTP-POST, HTTP-REDIRECT and SOAP binding.
- the name id formats supported by the SP.
- the assertion consumer endpoints for the HTTP_POST and HTTP-Artifact binding of the SSO Profile and the PAOS binding of the ECP Profile.

6.2 Example: Liferay

Liferay is the implementation choice for the portal within p-medicine[16]. As the portal is the entry point for p-medicine, integration of Liferay into the p-medicine security architecture is an import use case.

As Liferay uses Spring, using Spring Security as defined in 6.1, is the easiest way to integrate Liferay into the security architecture. Therefore the first step is to create a Liferay extension which adds a Spring Security SAML entry point (as explained in 6.1),. The result of this is that when someone goes to the URL "saml/login" SAML authentication is initiated. Once SAML authentication succeeds an authenticated Spring Security context will be active.

As Liferay itself does not use Spring Security an authenticated Spring Security context does not result in an authenticated Liferay session. For this Liferay provides an interface <AutoLogin> which should be implemented to add support for custom authentication mechanisms. All configured instances of the <AutoLogin> interface will run in consecutive order for all unauthenticated users. Once one of them returns a valid user id and password combination, the portal automatically authenticates that user.

Table 1 AutoLogin interface

```
package com.liferay.portal.security.auth;

import  javax.servlet.http.HttpServletRequest;
import  javax.servlet.http.HttpServletResponse;

/**
All configured instances of this interface will run in consecutive order
for all unauthenticated users until one of them returns a valid user id
and password combination. If no valid combination is returned, then the
request continues to process normally. If a valid combination is returned,
then the portal will automatically login that user with the returned user
id and password combination.
```

[16] D8.1.2 Design and prototype implementation of the p-medicine portal

```
*/
public interface AutoLogin {


      public String[] login(
                  HttpServletRequest request, HttpServletResponse response)
            throws AutoLoginException;


}
```

Therefore the interface <AutoLogin> needs to be implemented to extract the user's unique p-medicine identifier from the saml token available through the authenticated Spring Security context. Through this identifier the Liferay user can be looked up and its username and password can be queried and returned. Note that this requires the user's Liferay password to be available in unencrypted form.

If no user exists for the given identifier a new user could be created. All identifying information needed for automatic user creation (such as email address, first name, last name) can be extracted from the SAML token.

Table 2 Example implemenation of AutoLogin interface for SAML authentication with Spring Security

```
package com.custodix.liferay.ext.saml;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.security.core.context.SecurityContext;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.saml.SAMLCredential;

import com.liferay.portal.NoSuchUserException;
import com.liferay.portal.kernel.exception.PortalException;
import com.liferay.portal.kernel.exception.SystemException;
import com.liferay.portal.model.User;
import com.liferay.portal.security.auth.AutoLogin;
import com.liferay.portal.security.auth.AutoLoginException;
import com.liferay.portal.service.UserLocalServiceUtil;

public class SAMLAutoLogin implements AutoLogin {

  public String[] login(HttpServletRequest req, HttpServletResponse res)
              throws AutoLoginException {
    //Get the saml token from the authenticated spring security context.
    SecurityContext context =SecurityContextHolder.getContext();
    if(context!=null) {
      SAMLCredential credential = (SAMLCredential)SecurityContextHolder
             .getContext().getAuthentication().getCredentials();
      String uid = credential.getNameID().getValue();
      try {
        User user;
        try {
          user = UserLocalServiceUtil.getUserByUuid(uid);
        } catch(NoSuchUserException e) {
          //user doesn't exist in Liferay database
          user = null;
        }

        if(user==null) {
          //if the user doesn't exist, automatically create him.
        }
        if(user!=null) {
          //user =UserLocalServiceUtil.updateUser(user);
          //get the the user id and  password.
          long userId = user.getUserId();
          String userPwd = user.getPasswordUnencrypted();
          return new
String[]{String.valueOf(userId),userPwd,Boolean.TRUE.toString()};
        }
      } catch (PortalException e) {
        throw new AutoLoginException("Authn failed: "+e);
      } catch (SystemException e) {
        throw new AutoLoginException("Authn failed: "+e);
      }
    }
    throw new AutoLoginException("Invalid context");
  }

}
```

7 Conclusion

This deliverable started by elaborating on the technical details of the data security framework. The authentication components were the main focus. Brokered authentication and SSO can be implemented on web sites by using the standard SAML profiles and bindings. For REST services there are no existing standard specifications yet. Therefore the conclusion was that REST services can use a Secure Token Service as specified in the SOAP WS-Trust specification. Such a Secure Token Service can also issue a delegation token more specifically a SAML token that identifies the end user. This token contains a condition that states the intermediate delegating services as specified in "SAML V2.0 Condition for Delegation".

The second part of the deliverable further extended the privacy framework. A global high level overview of the import of sensitive patient data into p-medicine was given.

The final part of the deliverable explained through some examples how services can easily integrate with the authentication components by using Spring Security.

Appendix 1 - Abbreviations and acronyms

ABAC	Attribute-Based Access Control
CATS	Custodix Anonymisation Tool Services
IdP	Identity Provider
PAP	Policy Administration Point
PDP	Policy Decision Point
PEP	Policy Enforcement Point
PIMS	Patient Identity Management System
REST	REpresentational State Transfer
SAML	Security Assertion Markup Language
SOAP	Simple Object Access Protocol
SP	Service Provider
STS	Secure Token Service
WADL	Web Application Description Language
WSDL	Web Services Description Language
XACML	eXtensible Access Control Markup Language