EGEE

Global Security Architecture

FOR WEB AND LEGACY SERVICES

EU Deliverable DJRA3.3

Document identifier:	**EGEE-JRA3-TEC-602183-DJRA3.3-v-1-2**
Date:	**2005-09-08**
Activity:	**JRA3: Global Security Architecture**
Document status:	**FINAL**
Document link:	**https://edms.cern.ch/document/602183/**

Abstract: This document captures, at a high level, the global security architecture that is currently being implemented in the EGEE project. Components and services are modeled as orthogonally as possible, to allow for optional use depending on deployment and application requirements. We discuss the motivations, identified risks and security considerations associated with the the architectural choices we have made. The global security architecture inherits many of the thoughts from previous projects and parallel ongoing efforts. It is more of an evolutionary continuation rather than a clean start.

This document is an update of a previous EU deliverable, DJRA3.1, tracking the evolution and new ideas that emerged during the past 9 months in the EGEE project and Grid community at large.

Delivery Slip

	Name	Partner	Date	Signature
From	EGEE Security JRA3	KTH/PDC, NIKHEF, UH/HIP, UvA		
Reviewed by	Johan Montagnat	CNRS	2005-08-04	
	Romain Wartel	RAL	2005-08-04	
	Dave Kelsey	RAL	2005-09-05	
Approved by	PEB		2005-09-07	

Document Change Log

Issue	Date	Comment	Author
1.0	2005-06-17	Initial version for internal review	Olle Mulmo
1.1	2005-08-31	Updates after internal review	Olle Mulmo
1.2	2005-09-06	Third review iteration	Olle Mulmo

Document Change Record

Issue	Item	Reason for Change

CONTENTS

1. INTRODUCTION **5**
1.1. REFLECTIONS AFTER A YEAR OF DEVELOPMENT . . . 5
1.2. DEFINITION OF A SECURITY ARCHITECTURE . . . 5
1.3. A WORD ON INTEROPERABILITY . . . 6
1.4. ASSUMPTIONS ON THE TRUST MODEL . . . 7
1.5. A WORD ON VIRTUAL ORGANISATIONS . . . 7
1.6. SECURITY ARCHITECTURE OVERVIEW . . . 8
1.7. MECHANISMS TO BE SECURED AND OUR APPROACH . . . 9
1.8. REQUIREMENTS . . . 11

2. LOGGING AND AUDITING **14**
2.1. AUDITING . . . 14
2.2. SECURITY CONSIDERATIONS . . . 15

3. AUTHENTICATION **16**
3.1. IDENTITY CREDENTIAL FORMATS . . . 17
3.2. SITE-INTEGRATED CREDENTIAL SERVICES . . . 18
3.3. ENFORCING VALIDITY CONSTRAINTS . . . 18
3.4. REVOCATION . . . 19
3.5. CERTIFICATE RENEWAL . . . 19
3.6. ANONYMITY, PRIVACY, PSEUDONYMITY . . . 19
3.7. SECURITY CONSIDERATIONS . . . 21

4. USER KEY MANAGEMENT **22**
4.1. KEY HYGIENE AND REPUDIATION . . . 22
4.2. BOOTSTRAPPING AUTHENTICATION . . . 22
4.3. CREDENTIAL STORES . . . 23
4.4. SECURITY CONSIDERATIONS . . . 23

5. ENCRYPTED STORAGE **24**
5.1. SECURITY CONSIDERATIONS . . . 25

6. AUTHORIZATION **26**
6.1. DELEGATION . . . 27
6.2. AUTHORIZATION POLICY OVERLAY . . . 27
6.3. MUTUAL AUTHORIZATION . . . 28
6.4. AUTHORIZATION INTERFACES TO EXISTING SYSTEMS . . . 28
6.5. AUTHORIZATION FRAMEWORK . . . 29
6.6. SECURITY CONSIDERATIONS . . . 30

7. ISOLATION AND SANDBOXING **32**
7.1. SECURING THE HOSTED TO NATIVE INTERFACE 32
7.2. NETWORK ISOLATION 32
7.3. SECURITY CONSIDERATIONS 33

8. USE-CASES **34**
8.1. A GENERIC SERVICE REQUEST FLOW 34

1. INTRODUCTION

This document captures, at a high level, the global security architecture that is currently implemented in the EGEE project. We assume that the reader is familiar with Web Services and Grid security concepts and terminologies. We will continously refer to the EGEE middleware (gLite) throughout the document and therefore recommend the gLite architecture document, EU deliverable DJRA1.4 [40], as additional background reading.

The security architecture inherits many of the thoughts from previous projects and parallel ongoing efforts. It is more of an evolutionary continuation rather than a clean start, but with some new components and features due to the transition to a Services Oriented Architecture [40] and the inclusion of additional user requirements from e.g. the biomedical community.

Security is a continuous arms race: new threats trigger new countermeasures, and the bar is continuously raised. As such, our global security architecture, and consequently this document, is also subject to constant change and evolution. This deliverable is the first update of the architecture as we envision it for the duration of the EGEE project and its follow-on project(s). Although we are planning for realising and deploying most of the components described in this architecture, some of them will not be realised until the very end of the project.

This document does not specify what cryptographic algorithms or key sizes to use when and where, partially due to the continuous arms race issue, but also because there is often a trade-off between security and performance. We leave it as an operational and continuously reevaluated issue to define and set a current acceptable level. Today, these discussions take place in any or all of the EGEE Middleware Security Group and the EGEE/LCG/OSG Joint Security Policy Group and Operational Security Coordination Team, depending on the issue.

1.1. REFLECTIONS AFTER A YEAR OF DEVELOPMENT

This is an update of the security architecture document, DJRA3.1[22]. The security architecture as such has evolved marginally; the additions to this document is only at clarification or additional details level. We take this as a token of confirmation that the initial document contained solid and future-proof work.

After only a single year of work, it should be noted that the current prototype middleware and the security architecture is not always aligned. For instance, the security architecture make use of artifacts from the Service-Oriented Architecture (SOA) conceptual model, such as a notion of a service container; however, this has to date rarely been utilised in the development, which has caused some side effects in that the current middleware design contains a slightly different security model. For instance, the security architecture specifies a single container, governed by the local administrator, that in turn re-routes requests to different service implementations that may run in different local (unix) accounts, for the sake of privilege separation. However, in current practice, the services are shipped with their own built-in container that provides a direct interface to the outer world. This causes security side effects at the network layer in that additional firewall ports may need to opened (one for each service), at the transport layer in that the service implementations need access to the credentials authenticating the host, in the case of logging in that each service has to log incoming requests (and does so in slightly different formats), and so on. These additional problems have forced us to invent additional add-on, workaround solutions of temporary nature, which are not covered by this document, as it has the goal and ambition of providing the security architecture that we will (eventually) use.

1.2. DEFINITION OF A SECURITY ARCHITECTURE

We define a security architecture as *A set of features and services that tackles a set of security requirements and can handle a set of use cases*. We note that the term *service* is here used in a broader sense

than *(web) service*, which is the common jargon in other documents, and also includes infrastructure and user-driven services such as certification, auditing and incident response procedures.

In addition to providing the components that enable secure remote access to computing and storage resources, the security architecture also considers the non-negligible probability of malicious use or compromised systems. Therefore additional resource protection and monitoring mechanisms, such as application sandboxing and logging and auditing of security related events are an integral part of our security architecture.

Furthermore, operational components are crucial to realise the security architecture as well. These include incident response procedures and the recognition among users and resources of a common Acceptable Use Policy (AUP). Such issues are considered out of scope for this document.

1.3. A WORD ON INTEROPERABILITY

It is important that the security architecture used by EGEE allows for basic interoperability with other Grid deployments or middleware projects such as the Open Science Grid [1], OMII [2], Naregi[3], NorduGrid [4] and LCG [5]. At a minimum, it should be possible to use the authentication and authorization architecture with systems that are already in operation.

Currently, we do not know which other Grid deployment projects we will interact with or what technologies they will be using as are they are evolving, sometimes rapidly. Therefore, we do not make assumptions on any particular technology except in the case of authentication (Section 3.). Instead, we aim at adopting a modular software development approach and make interfaces as technology-agnostic as possible.

Our philosophy can be summarised as:

Be modular The system can be updated by distributing additional modules and plugging them into existing (deployed) frameworks with a simple change of configuration.

Be agnostic Here, a good example is authorization. We have seen a rapid evolution in the past years of different authorization technologies that are all non-interoperable (grid-map files used by early versions of Globus [18], VOMS [1], CAS [32], PERMIS [2], and Shibboleth[6], and so on.) In addition, different deployment projects have hitherto adopted or supported a single version of these technologies.

Our end goal is a system that enables an application and an associated governing policy to remain indifferent from what kind of attribute authorities or authorization information systems that are currently in use: those can change over time thanks to the modularity in the authorization and policy reasoning frameworks.

Be standard By adhering to standards, we will eliminate many of the interoperability problems beforehand. Sometimes there are many standards or no standards at all, often because no particular technology is widespread enough or mature enough. In those cases, we keep an open dialogue with the rest of the Grid community and try to settle for common practices. Considering the unique size and manpower of the EGEE project, it is imperative that we take a leading role in the Global Grid Forum [17] and at other Grid-related venues.

[1] Open Science Grid. http://www.opensciencegrid.org/
[2] Open Middleware Infrastructure Institute. http://www.omii.ac.uk/
[3] National Research Grid Initiative of Japan. http://www.naregi.org
[4] http://www.nordugrid.org
[5] LHC Computing Grid Project. http://http://lcg.web.cern.ch/LCG/
[6] http://shibboleth.internet2.edu/

1.4. ASSUMPTIONS ON THE TRUST MODEL

Grid computing is, in its essence, about bridging across organisational boundaries. In order to do so in a trustworthy manner, we identify two common solutions to this problem: *virtual organisations* (VOs) and *federated trust*. There has been much discussion around these concepts, and investigations have been made to clear which is the most suitable for large-scale, multi-purpose Grids [28]. Here we try to briefly summarise the differences, as we understand them, and motivate the choice of trust model.

In the Virtual Organisation model [21], the VOs are dependent on a common, often organisationally independent, trust fabric that typically materialises in an authentication infrastructure. This infrastructure can be shared by many different VOs that in turn operate under different policies: in essence, all parties (users and resources) are given authenticated unique identifiers, to which a VO or a site can tie policy governing e.g. access control permissions to a shared pool of resource available to that particular VO. We discuss this further in Sections 3. and 6.

The trust anchors in the VO model are the certification authorities (which govern the authentication infrastructure) and the VOs themselves, who self-govern the use of the resources that have been made available to them.

The federated trust model typically materialises as a more formal collaboration than that of virtual organisations, and is often used in business contexts. Here, an enumerable set of organisations join and agree on common policies and processes. When a user attempts to access a resource at another organisation, the user's home organisation assesses the validity of the request to the external resource: in this scenario, the trust anchors are the organisations themselves. Two examples of a federated trust model are the Liberty Alliance[7] and Shibboleth.

We have chosen the VO trust model for our global security architecture: legacy expertise from earlier projects such as EDG [11], application experiences from LCG [25], less need for organisational support and buy-in from the participating resource owners and service providers are some of the reasons that have led to this decision. As a side effect, this implies that it is hard(er) to fulfill anonymity requirements: we discuss this further in Section 3.6.

We note that the trust models as explained here are quite theoretical – in practice it is often hard to distinguish the boundaries between the VO model and the federated trust model. Thus, our choice was more of a paradigm choice or a way in which to think of the underlying problem. In fact, we don't rule out the possibility that some VOs or organisations may well operate under a federated trust model in the real world. From "our" perspective, it will look like another trust anchor upon which we will base authentication as well as authorization trust. An example of such a scenario is a VO making use of an existing Shibboleth infrastructure for managing access control and membership policy.

Besides the trust model, Grid computing has traditionally honored a golden rule of thumb to which we adhere strictly: *Always retain local control*. For example, any locally defined access control policy takes precedence over any "external" or centralised policy. This is further discussed in Section 6.2.

1.5. A WORD ON VIRTUAL ORGANISATIONS

During discussions over the past years, it has become clear that we all have slightly different perceptions on what a virtual organisation fundamentally is, and what generic properties you can attribute to it. A simplified view on how we envision a VO is therefore explained here.

The VO-based Grid computing paradigm does not mean that additional resources magically appear in the computing world for a user. A set of users, collaborating on a common goal, get together and throw whatever local resources they have access to in a common pool. Or differently put, the VO is empowered by the user with the permission to grant use of the resources to any member in the VO. The pool of

[7] http://www.projectliberty.org

resources is shared by the users, with the goal of obtaining an efficient and maximum total usage. We call the resource pool and the users a VO.

Note that the internal structure and management of a VO is not our concern. The VO can take the form of a hierarchial, well structured collaboration, or it can be some loosely-coupled peer-to-peer relationship.

That said, we still have to provide users with tools that enable them to manage their VO. One such tool that is provided out-of-the-box is VOMS [1]: others, such as CAS, PERMIS and Shibboleth are envisaged as well.

1.6. SECURITY ARCHITECTURE OVERVIEW

Our security architecture consists of the *services* listed in Table 1. As discussed in Section 1.2., we use the term service in a slightly wider perspective than "just" web services.

1.7. MECHANISMS TO BE SECURED AND OUR APPROACH

We have identified the following mechanisms that our security architecture must consider:

Web Services hosted in containers and application servers such as Apache Axis [15] and Tomcat [16]. In particular, this covers large parts of the gLite middleware architecture as described in [40]. Here we propose generic modules that integrate and extend the container environment, filtering incoming and outgoing requests and passing specific tokens to the front-ended resource and application logic.

We acknowledge that message or XML-based security fits much better than transport-level security (e.g., SOAP over HTTPS) for Web Services architectures, where intermediaries route requests to target actors at the back-end. In particular, real end-to-end security requirements are met. However, due to the overhead and limited interoperability in existing software and standards [8] , we conclude that transport-level security is an option that performs and scales better.

That said, we will make sure that we are ready to adopt the full suite of WS-Security message-level security when improvements in standards, interoperability and tooling performance make those technologies ready for production use.

Access to other protocols that are not Web Services based, such as GridFTP or common web servers. Here we propose modifications and/or plugins to a small set of commonly used software components on a per-need basis.

We note that the functionalities described in our security architecture are in most cases *embedded* in the service container or in the application itself, for performance reasons – they are not rendered as separate Web Services with which the applications or other applications interact.

Figure 1 depicts an overview on how the components in the security architecture interact in the following typical request flow:

1. The user [9] obtains Grid credentials from a credential store, and the necessary tokens that assert the user's rights to access the resource [10]. The credentials are short-lived and often derived from longer-term credentials, such as X.509 identity certificates issued by a Certification Authority (CA).

2. The user and the service container authenticate identities to each other and establish a secure communication channel across the (open) network with integrity, authenticity and confidentiality protection, and over which a SOAP message payload is conveyed. By default, this is accomplished by use of HTTP over TLS. The established connection event is logged.

3. The authentication layer validates the user's identity with the trust anchors and credential revocation information, if such exists. The result of the validation is logged. **Note:** While only depicted at the right-hand side in the picture, this check is mirrored at the client side: the client validates the target computer to which it is sending its message by performing the same set of checks.
The service container absorbs the payload and routes it to the correct service endpoint. In the case of message-level security, the authentication and integrity checks happen here (i.e., after the message has been absorbed from the network).

[8] For an amusing essay on this topic, see Peter Gutmann's *Why XML Security is Bad*, http://www.cs.auckland.ac.nz/~pgut001/pubs/xmlsec.txt

[9] We use the word "user" in wide terms: for instance, it also encompasses the software agents that act on the user's behalf

[10] A "resource" in Web Services terminology is practically anything that is managed/front-ended by a service: it can be a compute element, a file transfer client, an information index etc.

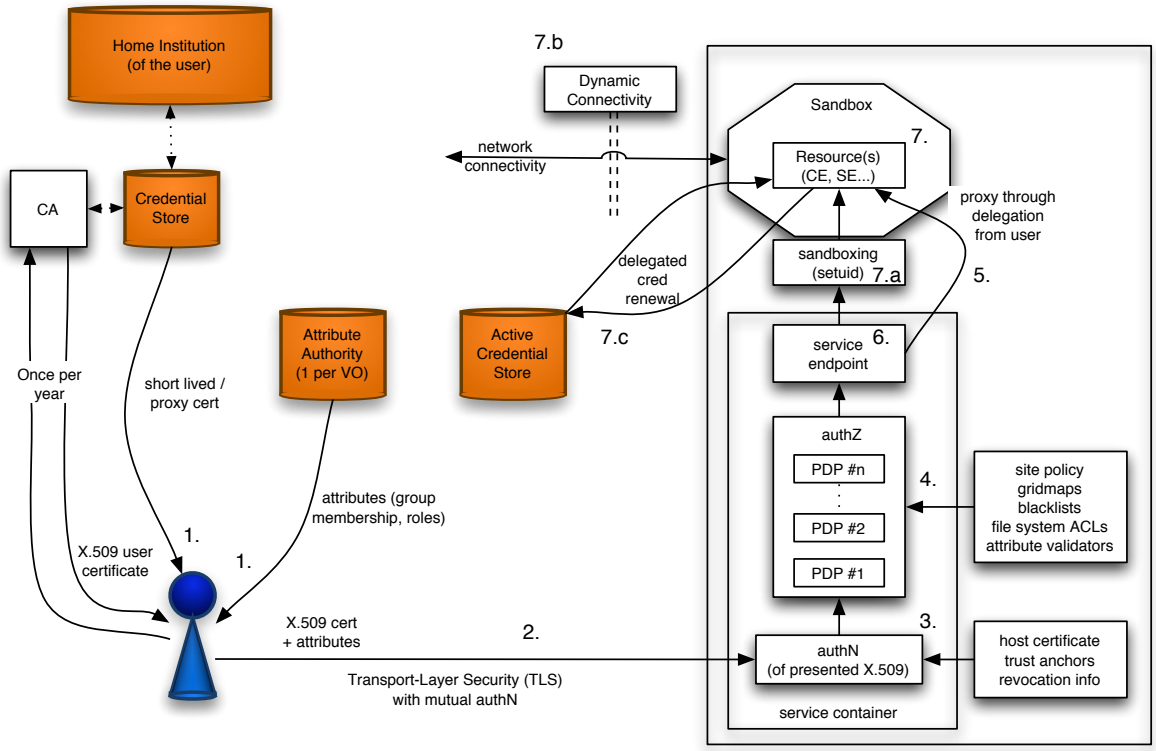

Figure 1: Overview of the components in the security architecture and a typical end-to-end interaction of a user (agent) accessing a resource. (Non-complete)

4. The authorization routines ensure that the user has permission to access the resource, by combining asserted attributes and the VO policy (sent with the request) with the local site policy and other sources of access control.

5. In the case that delegated credentials are used, the user delegates rights to the delegating resource to act on the user's behalf. Note however that delegation typically happens as a separate end-point invocation, and is part of the application-level message flow between the user and the service.

6. The service implementation gets invoked. The authorization routines may be used for additional evaluation and consultation.

7. The service interacts with the resource, which in turn may have delegated credentials at its disposal. Sandboxing and isolation techniques limit the user's influence on the resource to within the expected boundaries, avoiding malicious or unintended usage or in the worst scenario a security breach. These include

   a. Operating the resource in a different user space than that of the service container (Section 7.1.).

   b. Consulting the Dynamic Connectivity Service (Section 7.2.) in order to temporarily enable direct inbound and/or outbound network connectivity to the resource.

   c. Provide additional protection of the delegated credentials by use of an Active Credential Store (Section 4.3.). This is also useful in the case of long-term use of a resource, where a renewal of the delegated credentials may be necessary. mak

1.8. REQUIREMENTS

Numerous sources of requirements have been considered in the discussions that led up to the security architecture presented in this document. These sources include, but are not limited, to:

The JRA3 gap analysis Conducted in the beginning of the EGEE project to assess current support and possible performance issues in existing toolkits and software libraries for different security solutions [38],

The gLite architecture document [40],

The JRA3 User requirements survey , A summary of "old" but security related application and/or operational requirements, originating in previous projects or elsewhere [39],

The EGEE NA4 application requirements database [41].

It is not within the scope of this document to explicitly list each of the requirements and describe how they are fulfilled (or not) by the architecture. A brief overview is listed in Table 2 with references to relevant sections in this document where the issue is discussed in more detail.

Service	Description	Time frame	Section
Logging and Auditing	Ensures monitoring of system activities, and accountability in case of a security event	Now	2.
Authentication	**Credential storage** ensures proper security of (user-held) credentials	Now	4.3., 3.2.
	Proxy certificates enable single sign-on	Now	6.1.
	TLS, **GSI**, **WS-Security** and possibly other X.509 based transport or message-level security protocols ensure integrity, authenticity and (optionally) confidentiality	Now	3.
	EU GridPMA establishes a common set of trust anchors for the authentication infrastructure	Now	3.
	Pseudonymity services addresses anonymity and privacy concerns	Mid-term	3.6.
Authorization	**Attribute authorities** enable VO managed access control	Now	6.
	Policy assertion services enable the consolidation and central administration of common policy	Future	6.
	Authorization framework enables for local collection, arbitration, customisation and reasoning of policies from different administrative domains, as well as integration with service containers and legacy services	Now	6.5.
Delegation	Allows for an entity (user or resource) to empower another entity (local or remote) with the necessary permissions to act on its behalf	Now	6.1.
Encrypted storage	Enables long-term distributed storage of data for applications with privacy and/or confidentiality requirements	Mid-term	5.
Sandboxing	Isolates a resource from the local site infrastructure hosting the resource, mitigating attacks and malicious/wrongful use	Future	7.
Dynamic connectivity	Enables applications to communicate on-demand across heterogenous and non-transparent networks	Later	7.2.

Table 1: Overview of the security architecture services. With *Mid-term* we mean that solutions are to be included before end of the (first) EGEE project, *Future* means things identified for the continuation of the project.

Requirement	Fulfilled	Solution/Technology/Service	Time frame	Section
Single sign-on	Yes	Proxy certificates and a global authentication infrastructure	Now	3.
User Privacy	Partially	Pseudonymity services	Mid-term	3.6.
Data Privacy	Partially	Encrypted storage	Mid-term	5.
Audit ability	Partially	Meaningful log information	Now	2.1.
Accountability	Yes	All system interactions can be traced back to a user	Now	2.1., 3.
Combining policy from different administrative domains	Partially	Authorization framework	Now	6.5.
VO managed access control	Yes	VOMS	Now	6.
Support for legacy and non-WS based software components	Yes	Modular authentication and authorization software suitable for integration	Now	1.7.
Timely revocation delays	Yes	Gradual transition from CRL based revocation to OCSP based revocation	Mid-term	3.4.
Non-homogenous network access	Yes	Dynamic Connectivity service	Future	7.2.

Table 2: High-level requirements and how the architecture address them.

2. LOGGING AND AUDITING

In the context of our security architecture, and in addition to any normal operational techniques for monitoring and intrusion detection, logged information will be used to:

- Trace and time-stamp security incidents.
- Provide evidence of incidents (to enable action to be taken).
- Derive incident reports.
- Conduct security audits.

In order to meet these requirements, the information logged must be "useful" (correct and complete information, uniform format, easy to digest, easy to correlate log entries related to the same event or user interaction). It is impossible to state a priori exactly what information is of interest to log and what is not, as we don't know under what circumstances we will consume the information. Still, general best-effort recommendations are provided in the EGEE developers guide [27].

In addition to the logging information being useful and relevant, the logging mechanism as such is a point of concern as well, as it is typically not reliable or integrity-protected. If components log directly to a local file, an attacker could modify or overwrite in order to not leave a trace. Methods such as logging to a local system service such as `syslog` provide better security, as the log information is then separated from the environment of the logging actor, and protected by the operating system. There are also logging systems that provide additional features such as making modification or backrolling of log entries hard to accomplish. All in all, we believe that there are plenty of ready-available techniques that can be used as mitigating factors to protect the logging of events, if that is deemed necessary from an operational perspective.

2.1. AUDITING

Auditing is a very general term: here, we primarily mean the system security aspects of auditing, such as monitoring and providing for post-mortem analysis of security-related events.

In computational Grids, auditing comes hand-in-hand with accounting, as they share the base requirements on the system's logging capabilities: *who did what where and when* (and in the case of accounting, *for how long or how much*) [19].

We note a clash between the requirements for auditing/accountability/accounting and the requirements for anonymity: the end-user identity should be clearly stated versus hidden respectively. This is discussed in more detail in Section 3.6.

Auditing is not only meant in case of an emergency (system breach), but is also useful for the validation of the continuous operations, verifying that components behave as expected and in accordance with specified metrics.

Just as in the case of logging, auditing implies a set of common principles and practices on all system components, e.g. to log correct, complete and relevant information. The audit information should itself be traceable and verifiable throughout its lifetime and throughout service invocations, so that any anomalies in the audit trail can be traced to identifiable service components.

For a service to provide correct and complete log is a "soft" goal that is hard to measure and evaluate in terms of compliance. We acknowledge this fact, and that we may have to resort to a best-effort approach, including provisioning of guidelines to developers, performing software component code reviews, and actively seek out ways in which a component, or collection of components, can be "broken".

We also acknowledge that the creation of a "real" audit service would clearly have benefits in terms of enforcing the uniformity of information, providing secure remote access to the information and ease the process of merging and combining the information from different services. However, the mere concept of having such a "Big Brother" capability, with deep insight into a site's internal affairs, has properties that are non-trivial from a political and social point of view in large parts of the world. This is one of reasons why, at this point in time, we have decided not to pursue this task, but leave the door open to a possible future inclusion of such a component in our architecture.

2.2. SECURITY CONSIDERATIONS

Ideally, for a system to have a "proper" means for audit ability according to recognised standards, we would need to invest heavily in detailed documentation of processes and methodologies used, as well as employ additional technology (things such as tamper-proof logs). These features are more than we can currently accomodate.

That being said, at least the audit information stored anywhere in the system should be self-contained and interpretable even if the original source of the audit information is later compromised, or if it is to be interpreted without access to the originating service [11].

[11] Or simply put, archive the log files and make sure the log messages are understandable and reflect actual events.

3. AUTHENTICATION

Authentication (AuthN) is concerned with identifying entities (users, agents, and services) when establishing a context for message exchange between actors. One of the key aims for Grid authentication is enabling single sign-on for the user – using a single identity credential with "universal" value across many different infrastructures, communities, virtual organisations and applications. As such, attention must be paid to the fact that the same identity is also to be used for other purposes than accessing Grid resources, such as networks and "traditional" web resources; or in interaction with government and administration.

The authentication model for EGEE is based on the concept of *trusted third parties* (TTPs): entities that are not related to any *relying party*[12] except through a trust relationship. Underpinning the trust relationship is the digital signature of the TTP, based on conventional asymmetric cryptography. The TTP will assert attributes or cryptographic key pairs to one or more identifiers that (uniquely) represent the entity.

Although theoretically a single TTP could service the entire community, in practice a mesh of TTPs exists for legal, cultural and administrative reasons. One can thus define a set of resources, users and services that agree to use a common set of TTPs for authentication. Such a common authentication domain does not imply common rights-of-access or constitute an "infrastructure" of sorts. In the context of the EGEE project, it is assumed that a common set of TTPs exists [13]. It is not assumed that all entities accept all TTPs: This introduces an additional failure mode that higher-level services should cover, anticipate and handle.

The strength of an authentication credential issued by a TTP is dependent on three things:

- The trust on the TTP, in particular its operations, procedures and general conduct.
- The quality of the original identity vetting. No amount of technology can overcome weak identity checking at the source. The use of appropriately qualified authorities in the credential issuing process is important. As such, the use of government-issued credentials (like PKI-enabled passports or photo-IDs) is encouraged.[15]
- The security of the private data needed to prove possession of the credential. This is further discussed in Section 4.

We are aware that alternative trust models exist (such as web-of-trust based on peer-to-peer authentication), but we have deemed it impossible to address these alternative authentication methods in the scope of the project, due to resources, timelines, and the big investment already made in the EUGridPMA.

Today, grid authentication is entirely based on the push model. As part of the connection establishment (when using a transport-level security mechanism), or as part of the message (when using message-level security), all relevant user data are sent. For example, the proxy certificate[10] contains the subject name of the originator, possibly a set of attributes signed by their respective attribute authorities, and a chain of signatures relating the subject to a trust anchor — a Certification Authority (CA) — known by the relying party.

This authentication model is ideally suited for building virtual organisations where independent users join in a common venture with independent resource providers. This is also the operating model that underlies most of the grids today. All major grid infrastructures in e-Science — worldwide — can rely on a coordinated set of trust providers that comply to a set of minimum requirements: the International Grid

[12] That is, neither the end user, the virtual organisation, nor the resource provider.

[13] For the EGEE project, management of the common authentication domain based on TTPs is delegated to the EUGridPMA[14] and the EGEE/LCG/OSG Joint Security Policy Group.

[15] That said, government-issued credentials can be of course be misused as well, once a malicious user has obtained them.

Trust Federation (IGTF)[16] where the EUGridPMA, together with the APGridPMA[17] and the TAGPMA[18] are responsible for maintaining these trust relations.[19] The Certification Authorities federated in the IGTF issue assertions to end-entities such as persons, networked entities, and attribute authorities.

The downside of this approach is that is is less suited in case larger collectives of users with common characteristics (e.g. a classroom of students) ventures on to a grid. In such cases, it is more convenient for both users and providers to be able to use the user's home organisation (UHO) ("the university") for bulk validation of the users. In that case not all students need to be issued with specific credentials or be re-registered. This of course can be done in both a push and a pull model, but in the interest of privacy preservation a pull model is often selected: the resource can query selectively for the required attributes (for which it is then up to the user to decide whether or not to give them), precluding the need for a negotiation phase. The best known example in this area is Shibboleth.

In this case, the client no longer has its own set of identity assertions and only the UHO has the capability to state the user's status (in this case "student"). The client itself cannot prove this status without help. Either before authentication is attempted, or as part of the authentication validation process by the service, these claims should be attested to by the UHO, and thus the UHO must be contacted as part of the process. In case of the assertion pull model, this will require additional "callouts" by the validation process that need to contact remote sources to obtain policy information: so-called Policy Information Points. It is expected that in this scenario, the authentication validation steps will be taken alongside other autorization decisions as part of the authorization chain: See Section 6.5. for additional details.

It should be carefully noted, however, that all the models described above imply that each domain of resource providers, users and identity providers (which is again a federation) will include many more trusted identity providers than what is customary today. For example in a case where large numbers of students join an infrastructure, one would have to trust each university to assert student affiliation. The level of trust placed in them, and the way they identify their subjects (students) is not uniform and difficult to audit effectively.

Thus there are important considerations regarding the risk assessment: the requirements on the trust level assured by the trust anhors (root authorities), and that these are intimately related to the value of the resources that they are protecting. For example, the EGEE operation today is based on the IGTF trust fabric and contains many thousands of machines, several petibytes of storage, but also expensive supercomputers, owned by organisations in many different countries world-wide. A weak authentication scheme, like one based on a valid email address only, is not adequate to protect these systems, and resource owners will not allow such weakly identified users to access the systems.

In a more dynamic federation, the quality of the (identity) assertion (information regarding its generation, the way the private data was stored, etc.) should be a part of the assertion itself, so that the authentication validation mechanism can implement constraints given the value of the resource to be protected. For example, an authority that issues certificates based on both smart-card based private keys embedded in a national passport, as well as certificates that are bound to a simple key stored in a file system, should indicate the quality of the private key storage in the certificates issued. That does imply that the source of the assertions itself is trusted, but it allows that source to issue assertions with a self-chosen level of validation.

3.1. IDENTITY CREDENTIAL FORMATS

We use X.509v3 public key certificates [9] to express identity assertions. These certificates are issued by Certification Authorities. Different types of CAs exist, as well as different means of delivering certificates

[16] http://www.gridpma.org
[17] http://www.apgridpma.org
[18] http://www.tagpma.org
[19] The various *nn*G(rid)PMA stand for: *EU*: Europe, *AP*: Asia–Pacific, *TA*: The Americas (north and south)

to end-entities. Whatever the delivery mechanism or operational mode of the CA, the authentication is based on at least the distinguished name (DN) of the subject contained therein.

While the distinguished name may contain information about the user such as name, organisational affiliation and email address, we do not make assumptions on these fields. The distinguished name is only considered as a unique identifier, nothing else.

In order to facilitate a single-sign on for Grid resources as part of the authorization process, our security architecture also allows *proxy certificates* as defined in [10]. Proxy certificates are normal X.509 identity certificates equipped with an extension that ensures that applications that do not support proxy certificates will not successfully validate them (the extension's criticality flag is set).

We note that other identity credential formats exist, most notably the SAML [31] format. While such credentials allow for transparent support for non-PKI technologies, they are neither widespread nor ubiquitously used in a variety of the protocols that are of interest to us: therefore, we do not consider SAML as a PKI replacement in the lifetime of this project. However, we see it as an interesting complement, and in regards to attribute assertions, support for SAML is paramount.

Shibboleth and Liberty Alliance are two, originally browser-centric, infrastructure technologies that also make use of SAML. Shibboleth has recently begun adoption attempts toward Grids [42]; We monitor this development closely for possible future inclusion.

3.2. SITE-INTEGRATED CREDENTIAL SERVICES

A site-integrated credential service (SICS) is an abstraction of an organisationally managed CA service that generate short-term identity credentials or user proxy credentials, without the user ever possessing any long-term credentials. This eliminates many of the key hygiene and exposure problems associated with user-held and long-term credentials, which are further discussed in Section 4. Several SICS implementations already exist today (kCA [33], Virtual Smart Card project [5]) and are described extensively elsewhere.

While a SICS is in all respects acting as any other CA from the view of an external party, we mention them separately as the elimination of long-term credentials somewhat alter the trust models. In addition, the trust is not anchored on a trusted third party anymore, but on the organisation itself which administer and manages the service: this does not scale well and it also hard(er) to ensure that a proper level of security and ensurance is imposed by all parties. This is one of the areas in the Shibboleth community that is still under development. See also Sections 3.4. and 4.4.

For the remainder of this document, we make a distinction between Traditional CAs and SICS CAs.

3.3. ENFORCING VALIDITY CONSTRAINTS

While proxy certificates enable single sign-on, they are typically stored with a weaker protection level (stored in clear text and safeguarded only by local file system privileges). This is in direct analogy with common handling of e.g. Kerberos [24] tickets.

As a consequence, security policy often declares that proxy certificates should not be trusted if issued with a long(er) validity: the GGF site-AAA recommendations [19] suggest a maximum lifetime of such proxy credentials to ≈ 24 hours. We enforce such common policies by extending normal proxy certificate validation by computing the proxy certificate's life time (which may be further limited by other certificates in the certificate chain) and comparing this to the configurable maximum life time: Proxies with a longer life time will be refused.

3.4. REVOCATION

There are good reasons why the binding between the public part of a key pair and an identifier or a set of identifiers should be revoked. These include the compromise of the associated private key or invalidation of one or more identifiers in the binding. The longer the key-to-identifier binding is considered valid, the higher the probability that such a binding will become invalid.

The TTP is responsible for revoking credentials it has issued, but it is up to the relying parties (both services and requesters in case of mutual authentication) to ensure that this revocation information is consulted.

The basic revocation information is typically distributed as a Certificate Revocation List (CRL) [9]. These CRLs act upon identity certificates issued by Traditional CAs only and are not applicable to SICS CAs or other short-term certificate issuers.

Timely identity revocation is needed to prevent exploitation of credentials that have been compromised. The allowed response time as specified by resource providers is in the order of 10-60 minutes, a constraint that practical implementations have shown cannot be satisfied by the periodic distribution of CRLs to all parties at a pan-European (or bigger) scale.

Therefore, any software component that performs certificate validation must be able to check the validity of the credentials in real-time. The protocol for validating authentication credentials will be the Online Certificate Status Protocol [8] (OCSP). Such services are being deployed primarily by the Traditional CAs in the course of the next year(s). While the details on the OCSP status propagation and the deployment of a suitable mesh of caches and responders is left as an operational issue, it should be mentioned that we are leading the GGF effort on a requirements document for an OCSP service architecture suitable for Grid environments.

We consider the use of OCSP as the primary source for revocation information as a mid-term project goal. In the meantime, as a backup mechanism in case of network partitioning or temporary server outage, revocation lists will be distributed periodically [20] and retrieved by all relying parties (i.e., users and service providers).

3.5. CERTIFICATE RENEWAL

Certificates are equipped with a validity timestamp and, as such, they expire and need to be renewed. There are several techniques that a CA may use to facilitate a trusted remote credential renewal, which we will not cover in detail here. For instance, the CA may choose to: simply issue a new certificate to the existing user-held key pair; require the user to generate a new key pair; countersign the new key with the old and so on.

In the case of short-lived proxy certificates, user invoked renewal is often needed for long-running processes and/or service interactions. Naturally, we cannot assume such level of availability from the users that such manual renewals would require. Instead, solutions such as MyProxy [29] have been deemed adequate to address such problems: here longer-lived credentials are consolidated in a (more trusted) online repository that can in turn issue short-lived credentials automatically upon request from the user or a running application. The user can then in turn update the credential repository at convenient intervals. We discuss the online availability of users' credentials in more detail in Section 4.3.

3.6. ANONYMITY, PRIVACY, PSEUDONYMITY

Application and user requirements call for anonymous use of the system: an outsider should not be able to deduce a particular user's activities, such as how much of the resources the user consumes or what

[20] Exactly how often CRLs are to be refreshed is an operational issue, but on the order of 4-6 times a day.

applications are run. Such *information creep* is of serious concern to applications in areas of highly competitive research, such as biomedicine.

One simplistic solution to the anonymity problem is by way of an anonymized access point (typically a web portal) through which limited access to a set of services can be provided. The portal in turn has an agent certificate which is traceable back to the person responsible and accountable for the portal and it's collective use of the Grid resources.

In order to enable true anonymous support throughout the system, we note that Grids are open-ended distributed systems. As such, protecting ourselves against information creep is prohibitive and infeasible. Even if all the information and message exchanges were made on encrypted and authenticated connections, analysing the message exchange patterns would still allow an adversary to deduce information on how the system is being used. In fact, true anonymity can only be achieved by all parties sending continuous streams of (possibly bogus) data to all other interacting parties at all time: This is a common tactic in the military and intelligence world.

Likewise, the only way in which you can obtain true privacy is by not sharing information or data at all – which is contradictory to the very nature and basis for Grid computing.

Thus, for these requirements, we provide best-effort solutions by the addition of a pseudonymity service [21]. The pseudonymity service swaps the user's real identity for a pseudonym, thus hiding it from immediate exposure in logs and on the network. The pseudonymity service acts in all regards as another TTP, with the addition that it is also trusted to keep the relationship between the pseudonym and real identity secret, unless law enforcement or a similar legitimate body requires the true identity to be revealed as part of e.g. an investigation on malicious use.

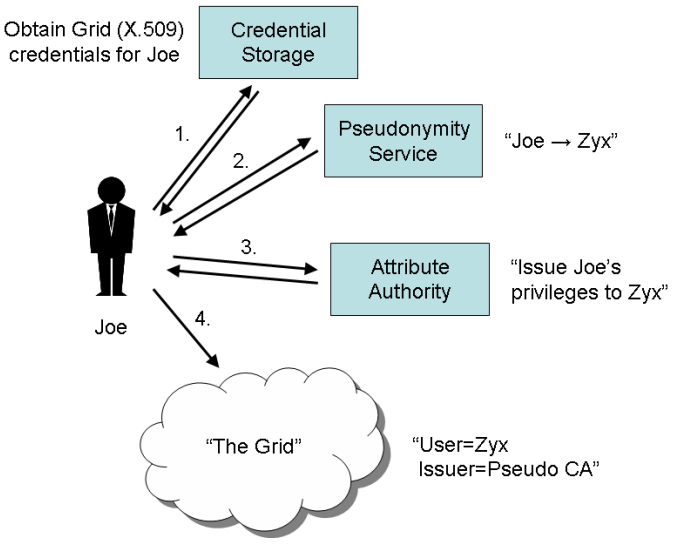

Figure 2: Using the pseudonymity service.

To acquire a pseudonym identity, a user performs the following steps (see Figure 2):

1. The user obtains their normal authentication credential, for example: *Joe*.
2. The user authenticates using *Joe* to the pseudonymity service which issues a short-lived, one-time identity credential, *Zyx*, to the user.
3. The user authenticates twice to the attribute authority using *both* the *Joe* and *Zyx* credentials, thus proving possession of them both. The attribute authority can then bind authorization attributes, originally issued to *Joe*, to the one-time identity.

[21] Pseudonymity is often used in the security literature to denote almost-anonymity.

Alternatively, an identity assertion, issued by the pseudonymity service that confirms the binding between the *Joe* and *Zyx* credentials, can be presented to the attribute authority. This is an approach used by e.g. Shibboleth, but is not directly applicable here as the identity and attribute authorities are different entities operating in different domains under different policies: the disclosure ramifications of a pseudonymity service issuing identity assertions must be thoroughly analysed first.

4. The user is now bootstrapped with a pseudonym credential *Zyx* and the necessary access privileges bound to that credential to make use of Grid services.

In the Grid VO model, the user's real identity is revealed to the pseudonymity service (the authentication trust anchor) and the VO service (the authorization trust anchor). We work under the assumption that we do not need to hide the true identity of a user from the other members of the same VO.

The pseudonymity service as proposed has many ramifications that must be seriously analysed: for instance, we note that access policy is constrained to work with VO attributes, such as role and group memberships. Certificate renewal is another that needs detailed investigation as well: ideally, a renewal should result in a new identity credential *Yxz* that cannot be easily associated with *Zyx*.

By using the above design approach, the pseudonym service becomes an optional and orthogonal service. It will be further investigated and for possible mid-term introduction.

3.7. SECURITY CONSIDERATIONS

Our authentication infrastructure depends on proper and trustworthy CA operations. The EUGridPMA is tasked to perform a continuous (re-)evaluation of the TTPs, asserting their compliance to minimum requirements and established best practices.

The delay of propagating revocation information throughout the whole system is a crude measure of the system's vulnerability, but it is still only a small part of the problem; We must be able to identify the need for revocation to begin with – a very hard operational problem.

SICS services make it easier for propagation of an attack from a compromised site to other organisations, since local site credentials can be used to obtain "global" authentication credentials. Thus, in order to trust certificates issued by a SICS service, the security of the organisation as a whole must be trusted. In fact, this is an example of the federated trust model explained in Section 1.4..

These are very tough requirements, and therefore we do not anticipate the existence of more than a handful SICS implementations among the TTPs used by the EGEE production service.

4. USER KEY MANAGEMENT

Experiences from the European DataGrid (EDG), the UK e-Science programme, and other projects have shown the complexity and problems associated with decentralised key management, and allowing the users to manage their own Grid credentials in so-called "soft credentials": that is, credentials issued and stored password-encrypted in normal files.

It is common knowledge within the security community that user-managed security results in sloppy security. How many people do not use `abc123` as their password, or have their ssh keys unencrypted in their home directory for convenience? By decentralising the management of the user keys to the users themselves, we lose the ability to assess the proper protection (strong password) and management (file access control) of the credentials[22].

Our security architecture does not prohibit the use of user-managed soft credentials, but we strongly recommend against this practice for general use in a large end-user population. Instead, we propose to invest in and make use of managed key solutions such as Site Integrated Credential Services (SICS) driven CA services, hardware tokens and active certificate stores. These are all discussed below.

4.1. KEY HYGIENE AND REPUDIATION

A private key accessible only by a single user is a very powerful tool from the point of view of auditing and accountability. This key makes it hard for a user to *repudiate* any proofs of their actions: this is, in essence, the non-repudiation aspect of PKI[23].

The handling and protection of credentials is often referred to as the problem with "key hygiene". The less a user protects their private key, the weaker the non-repudation argument becomes and the higher the risk of a compromise. As noted in the introduction to this section, it has been shown that we cannot delegate the key management responsibilities of soft credentials to the average end-user. Such responsibilities include choosing strong passwords and ensuring proper file protection.

4.2. BOOTSTRAPPING AUTHENTICATION

However we approach the key management problem, we will always end up with an end-user and a mechanism for the end-user to obtain Grid credentials. This bootstrap mechanism has historically been password based, where the (static) password is used to decrypt the scrambled contents of a soft credential file stored locally or in a MyProxy [29] server.

There have been many discussions, both within and outside the Grid community, to find better methods and technologies to bootstrap the authentication process. Such technologies include one-time passwords (OTP) based on communicating a one-time shared secret between two parties in an out-of-band fashion, which is then used in addition to an ordinary password. The out-of-band methods vary, from Crypto-Card or SecureID-based *fobs* [3][35] (small tamper resistant units equipped with a microprocessor and a display) to SMS via a mobile phone. Additional technologies include codes on paper similar to scratch lottery tickets, the use of biometric data such as fingerprints, retinal scans and voice recognition.

One of the goals of our security architecture is not to rely solely on static passwords in the authentication bootstrap process: or, using proper terminology, we want to move towards the use of *two-factor authentication*.[24] A future, separate JRA3 document will investigate and assess suitable complementary

[22] Based on empirical evidence, we make the assumption that it is virtually impossible to train all end-users to handle their credentials correctly.

[23] True experts say of course that there is no such thing as non-repudiation, but we will leave that contentious debate aside.

[24] *n*-factor authentication: using *n* factors (properties) when establishing the identity. Factors that are normally used in these types of schemes include 1) what you know (password), 2) what you possess (smartcard, fob, mobile phone), 3) who you are (biometrics).

technologies to static passwords. We will also closely monitor other related activities in this area, such as the large-site OTP-based integration effort currently underway in the U.S.

4.3. CREDENTIAL STORES

The private key(s) pertaining to an end-entity certificate is usually held by the subscriber. The traditional CA that signs the certificate has no means to enforce key hygiene on the side of the subscriber. Thus a key compromise will usually go undetected for a significant amount of time.

This key hygiene problem can be addressed by the introduction of *managed* credential stores in our architecture. However, one should keep in mind that such a credential store, when containing a sizeable amount of private keys, becomes a high-value attack target. In an operational setting, adequate attention should be given to securing the credential store and limiting the amount of possible damage done whenever possible. That said, it has been proven over the years that such a system is less prone to compromise if one manages one computer containing 1000 secrets instead of 1000 computers containing one secret each [4]: The point made here is that poorly managed user credentials have resulted in hundreds, if not more, of real-world compromises in past years, while, to the best of our knowledge, not a single centrally maintained authentication server has been compromised in the past decade.

The most promising credential store technologies are outlined below.

Smart Cards This type of credential store is most secure from non-authorized access. The private data is kept internally on the tamper-resistant device and cannot be released or used by anyone (including the user) unless they are in possession of both the smart card and the relevant activation data, such as pin-code or biometrics. However, technological barriers and lack of infrastructure support in general prohibit large-scale smart card deployments today (especially in the case of mobile users). Furthermore, renewal of short-term proxy certificates created by a smart card based credential is prohibitive. However, smart cards can be used in the bootstrapping phase: for instance, by registering a long-lived proxy certificate to a MyProxy server.

Active Certificate Stores An ”active” certificate store (ACS) can act on the user’s behalf with any (traditional) CA and request conventional certificates on the user’s behalf. The private data is not released to the user – only proxy credentials will be provided to the user. An ACS can be implemented by MyProxy or similar credential repository technology, such as a SICS CA.

4.4. SECURITY CONSIDERATIONS

The fact that the users’ credentials are held in an ACS necessitates that operations are run under a strict regime. For instance, work was done at NERSC on creating a security policy that does not allow any other process but the MyProxy process to run on a Linux server with the NSA SecureLinux kernel.

Nevertheless, by storing each private key individually encrypted with a password known only to the individual user, the implications of a security breach can be considered moderate. In the ACS model neither the system administrator nor an attacker has immediate access to any raw key material.

Although the SICS does not have the issues on user key hygiene associated with traditional CAs, it does not support the non-repudiation aspect to the same extent. It is potentially “easier” for a user to deny having performed an action, if that action has been authenticated with a SICS-issued credential. The user can claim that the SICS itself was compromised, or an entity capable of manipulating the SICS (such as the administrator) has abused their rights.

5. ENCRYPTED STORAGE

User groups are concerned with additional data protection mechanisms in addition to normal access control. In particular, they are concerned with plaintext[25], long-term storage of data.

As noted in Section 3.6., we can not provide a sound and complete privacy solution due to the distributed and open-ended nature of Grids. For example, we cannot enforce that data not be copied outside of "the Grid" and stored as copies since we do not have full control over the resources, network connectivity and the running applications.

We introduce a *Encrypted Storage key management service*, and stress the fact that this service solves the long-term plaintext storage problem and that problem only. It does not aim to evolve into a complete solution for data privacy – in fact, in that regard we have taken a dead-end approach. We envision the introduction of the key management service as a mid-term project goal. An early prototype has been developed together with NA4/BioMed and JRA1/DM for evaluation.

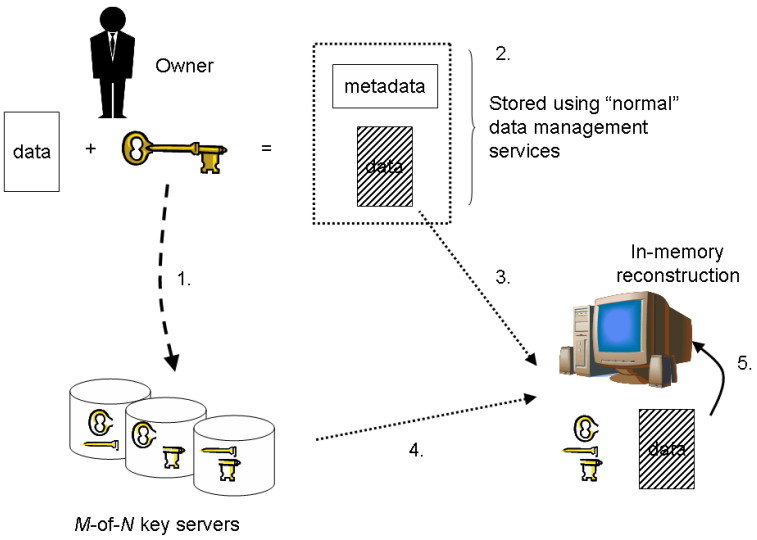

Figure 3: Data key management.

The overall architecture is as follows (see Figure 3):

1. The data owner (i.e., creator) generates a key with which the data is encrypted using conventional encryption algorithms. However, care must be taken in regards to what cipher mode is used so that we keep transparency: in particular, eliminate data size increases due to encryption, providing for random access mode, and reducing the risk of bit failure propagations. For additional details on different cipher modes see [13]. We leave the evaluation and actual choice of cipher mode as an implementation exercise.

   The encryption key is split into parts and registered to a set of key servers using off-the-shelf *M*-of-*N* techniques[26]. This way, no single server contains enough information to retrieve a plaintext copy of the data. Furthermore, *M*-of-*N* introduces redundancy and reduces the damage of a single server becoming compromised. In fact, there are cryptographic techniques, such as Shamir's secret sharing algorithm [36], that ensures that no information about the original encryption key is revelead at all if you are in possession of up to $N-1$ parts. Determining suitable numbers of *M* and

[25] We use the word *plaintext* when we mean *non-encrypted*.

[26] *M*-of-*N*: you need a subset of *M* parts of the possible *N* in order to reconstruct the original message (in this case, the encryption key).

N is left as an operational issue: we envision something like 2-of-3, but leave it as a configurable option.

2. The location of the key servers, the encryption algorithm with which the data was encrypted, and the algorithm used to split the encryption key can be left as a deployment configuration option, or modeled as additional metadata which then must be associated with the data itself. As it is not considered sensitive information, it can be registered and managed as any other "ordinary" data management metadata.

3. When reconstructing, a copy of the encrypted long-term storage data is retrieved using the standard data management services. The key server locations are determined, possibly from the metadata.

4. A process authenticates (using delegated credentials with the proper authorization attributes) to the key servers and reconstructs the original encryption key in runtime memory. Whether the process is an actual application or some wrapper component that performs this operation transparently to the application is left as an implementation choice.

5. The data is decrypted into plaintext in local memory, or possibly onto a temporary file on local disk (if allowed by the application).

5.1. Security Considerations

Our security model is far from fool-proof: it is merely a first attempt at increasing the security level that can be applied to a plaintext copy of some data.

There are currently no revocation or automatic replacement mechanisms for the encryption key. We offer no protection against attacks on the transient in-memory/local-disk copies of the plaintext data. We also do not attempt to contain the damage due to the malicious use or distribution of a stolen encryption key.

6. AUTHORIZATION

Authorization (AuthZ) is concerned with allowing or denying access to services based on policies. The core problem with authorization in a Grid setting is how to handle the overlay of policies from multiple administrative domains (VO specific policy, operational procedures, site-local policy) and how to combine them.

There are three basic authorization models [34], classified as *agent*, *push* and *pull*. In the *push* model, an authorization service issues tokens. The user collects the tokens and presents these to the resource where access is requested. Although this puts an additional burden on the client, the resource does not need to know ahead of time about the user's privileges.

In the *agent* model the user only interacts with the AuthZ server, which in turn forwards service-specific parts of the request to the underlying resources. While being a centric approach, network bandwidth or connectivity provisioning is best done in this mode since access to the network in the end must be transparent.

In the *pull* model, the resource asks the AuthZ service on a need-to-know basis. This puts the burden on the resource, as it needs to know up-front all the authoritative parties in the system and how to contact them. Cellular phone roaming, and various RADIUS -based [6] network access services use this model.

All in all, we have determined that the *push* model is the model that best suits dynamic, distributed and loosely coupled systems such as Grids. This is also the model that have been most widely used in other Grid projects.

There are two kinds of AuthZ services: attribute authorities (AA) and policy assertion services.

Attribute Authority services associate a user with a set of attributes in a trusted manner to a relying party, by way of digitally signed assertions. The relying party (the resource) evaluates the attribute assertions (e.g. role and group membership information) and includes them as "evidence" added to a *context*, which in turn is used when evaluating an access request against local policy. Examples of such services include the VO Membership Service [1] (VOMS) and Shibboleth, which issue attribute assertions using X.509 attribute certificates [12] or SAML [31], respectively. The attributes normally used are the VO membership for a particular user itself, and the user's role and group in that VO: this has been shown to provide an adequate, fine-enough access control granularity for most Grid communities to date.

In the case of policy assertion services, the resources consolidate some of the policy definitions to a trusted third party. The policy assertion service will issue claims that gives a user (or set of users) the explicit privilege to perform an action (or a set of actions) on a certain resource (or set of resources). The resource owners may or may not decide to comply with these claims, pending the evaluation of conflicting local policy. Examples of such services include the Community Authorization Service, CAS [32] and PERMIS [2], both of which can issue statements encoded in SAML [31].

In our architecture, we foresee the need for both the AA and policy assertion services mentioned above and the use of primarily the push model. Initially, an AA service such as VOMS will provide coarse-grained division of users into different groups, which can then be handled by the resources through local configuration. However, with an increasing number of VOs using the EGEE resources, such configuration becomes unmanageable and we will therefore need a consolidation of VO policy that can be provisioned to the resources, e.g. on a need-to-know basis via the client requests.

We do not consider the VO policy server component to be a crucial piece of the core architecture from the start of the project, as the initial number of VOs to handle and their operational requirements are fairly uniform and can be handled manually. However, there are several efforts underway on this front in which we are actively involved, and while additional research is still required at this point in time, we foresee a future inclusion and deployment of such a component in our architecture.

6.1. DELEGATION

We foresee that future Grid systems will be rendered as Web Services and therefore many security components and solutions (authentication, authorization, message integrity, etc) can be taken from the Web Services communities. Currently, the cross-organisational and dynamic properties of Grids have not been adequately dealt with by the Web Services community.

One such area that is lacking is credential delegation. It is often the case that Grid users need to delegate some subset of their privileges to another (dynamically created) entity on relatively short notice and for a brief amount of time. For example, a user needing to move a dataset in order to use it in a computation may want to grant to a file transfer service the necessary rights to access the dataset and storage so that the service can perform a set of file transfers on the user's behalf. Since such actions may be difficult to predict, the need to arrange delegation ahead of time is prohibitive.

An important security aspect in regards to delegation is the principle of least privilege: you only want to delegate as much privileges as is necessary. This is hard to accomplish in reality, but our security system software must support some simple provisioning and enforcement for constrained delegation. To a first order of approxmation, such constraints would include a limitation against delegating a privilege further, and to not delegate the privilege to execute or start up new processes on behalf of the user. In addition, it is often very useful to separate the privilege to delegate from the privilege itself.

A number of existing mechanisms could satisfy the delegation use case mentioned above. The use of proxy certificates has been the most widely mechanism adopted by the Grid community [43]. However, the principle of least privilege has historically *not* been honored, except for signaling whether or not the delegated credential can be used to start up new processes.

Historically, credential delegation has been closely tied to authentication, e.g. as an extra optional step performed after a TLS handshake as was done in the first incarnations of GSI [14]. However, this breaks compatibility with the TLS protocol and subsequently any protocol on top of TLS, such as HTTPS. In our architecture, we strive to separate delegation from authentication for the non-legacy software components. We define and implement a stand-alone Web Services delegation portType[27], for which support can be embedded in the service container/application server (as a separate service), or in the application itself (by inclusion of the the delegation's portType in the service description and helper libraries that implement the operations exposed by that portType).

6.2. AUTHORIZATION POLICY OVERLAY

When making an AuthZ decision, we must be able to combine information from a number of distinct sources (Figure 4). Besides Grid-wide or VO-wide attribute and policy assertions that are potentially provided to a service as part of the client request, we also need to include domain specific policy such as file system ACLs for file access or a (sub-)set of VOs that are allowed access to a particular computational resource. Finally, and *most* importantly, we must also take any locally defined site policy into consideration when making a unified context-specific AuthZ decision on an individual request basis.

In order to be able to combine information from multiple sources in this manner, we must have a way to convert any domain-specific access control language into a common language with strong support for combining policies, such as the eXtensible Access Control Markup Language [30] (XACML). While ideal from an architecture and interoperability point of view, this XACML conversion comes with considerable performance penalty. Another possible solution is to combine the evaluation of domain-specific policy engines in a common framework. We discuss this further in Section 6.5., where we present the architecture of an authorization framework that can handle both use cases.

[27] `portType` is Web Services speak for the interface definition of the "exposed" parts of a service. We refer to the Web Services literature for details.

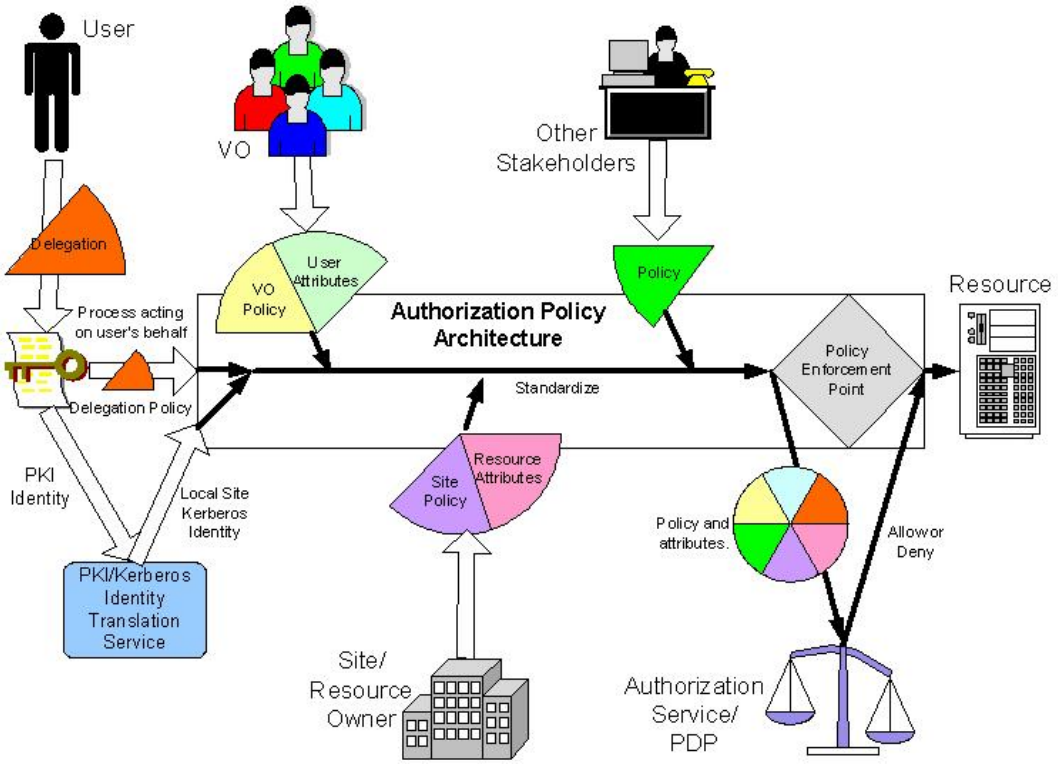

Figure 4: Conceptual view of a framework for collecting authorization policy from different stakeholders and combining them into a final authorization decision. Graphics from Globus Alliance and the security design team in the GGF OGSA working group.

6.3. MUTUAL AUTHORIZATION

There are use cases where a client wants to authorize a service as well as the service authorizing the client access; we call this *mutual authorization*. For instance, a client wanting to store sensitive data may want to first ensure that the SE has been approved to service such storage requests. Mutual authorization only succeeds if both parties can successfully authorize each other.

If we were to build this feature tightly into every service (for instance, by having services authenticate by using proxy certificates with embedded authorization information) this would result in additional complexity of all client software.

There is a strong interest in keeping clients as simple as possible, ideally one should be able to interact with a service using simple command-line tools such as curl or perl+openssl, whichever the version that comes shipped natively with the operating system. Therefore, we have chosen to make mutual authorization an *optional* operation, triggered by a (more complex) client. A client can obtain a mutual authorization by interrogating a WSDL portType that a service can include as part of its WSDL specification. The client can obtain from the service, for example, VOMS tokens tied to the service authentication credential.

6.4. AUTHORIZATION INTERFACES TO EXISTING SYSTEMS

Some legacy services will continue to be used despite the paradigm shift to Web Services. Such services will be updated on a per-need basis: for instance, to add VOMS and/or pool account support to a GridFTP server. In this case, we note the usefulness of third-party applications implementing common pluggable interfaces, such as e.g. the Globus authorization callout plugin interface, but also that such interfaces are complete in their structure and not only cover the most basic integration use case.

Unfortunately, such modifications to legacy code will in many cases mean that we have to apply patches to the source code, but thanks to the systems that we provide, such as the Java AuthZ framework and LCMAPS[28], such patchwork should be reasonably easy.

6.5. AUTHORIZATION FRAMEWORK

When controlling access to services and resources, both local and distributed policies must be taken into consideration. In this section, we show a model and framework for enforcing, retrieving, evaluating and combining policies locally at the individual resource sites, e.g. inside service containers.

It must be easy for administrators to configure and extend the authorization system to implement local policies. A resource in a dynamic Grid often does not have the luxury of knowing all its users a-priori, and can thus not rely solely on local access control lists (ACLs). Furthermore, simple ACLs are typically too coarse-grained for more advanced policy-based applications.

The goal of the Authorization framework is to provide a light-weight, configurable, and easily deployable policy-engine-chaining infrastructure that is agnostic to back-end enforcers and evaluators, as well as the run-time container infrastructure and the state model that hosts them. The framework allows for a combined and flexible decision making process, taking into account information, assertions and policies from a variety of authorities.

XACML may very well be used as a cornerstone in this architecture. On the other hand, many production quality authorization systems already exist today, and should be leveraged with a minimal effort and intrusion, e.g. file-level ACLs, GACL, VOMS, LCMAPS, LCAS, and CAS. Security Assertion Markup Language [31] (SAML) may also be used to transport capability and privilege assertions (see previous discussion in this chapter). Therefore, the core Authorization framework does not make any assumptions about policy languages or network protocols. The only consideration is that if policies should be exposed to external parties. This would imply the policies should be representable in a language such as XML. If external policy modifications are allowed, the same consideration applies for the policy update language.

CORE DESIGN

While we reuse the nomenclature and overall architecture from XACML, some details in the design are somewhat different. Additional details of this design and future extension can be found in [37].

The core part of the framework provides interfaces to a virtual Policy Enforcement Point (PEP) and Policy Administration Point (PAP). The PEP invokes a chain of policy engines and gets an authorization decision result (permit, deny, not applicable) in return. A policy engine may be either a Policy Information Point (PIP) or a Policy Decision Point (PDP). PIPs are responsible for interfacing to Attribute Authority-based services to collect and verify assertions and capabilities associated with the authenticated identity of the user[29]. These attributes can be used by PDPs configured at subsequent positions in the chain. A PDP has the additional capability of affecting the outcome of a policy decision. It can decide whether the user is allowed to perform the requested action, whether further evaluation is needed, or whether the evaluation should be interrupted and the user denied access. Both custom policy engines and policy combining chains may be plugged into the framework through configuration. Chains may be ordered in tree-based hierarchies where the lower-level junior chains inherit the policies of the higher-level senior chains. There is always just a single root chain, and thus there is always a guarantee that the decision that is derived is authoritative.

[28] http://www.nikhef.nl/grid/lcaslcmaps/

[29] Note that the attributes may be part of the incoming request, e.g. VOMS tokens embedded in a user's proxy certificate. In such cases, the PIP in question will do introspection on said certificate – there will not be a direct interaction with any Attribute Authority.

We deemed converting domain specific policy into a common lingua franca, such as XACML, impractical and too performance sensitive. The tree-based hierarchy of authorization chains allow for defining precedence and combination of multiple policies, each evaluated by its own (domain specific) decision point. In essence, the chains make the internal workings of a PDP – as defined by XACML – explicit.

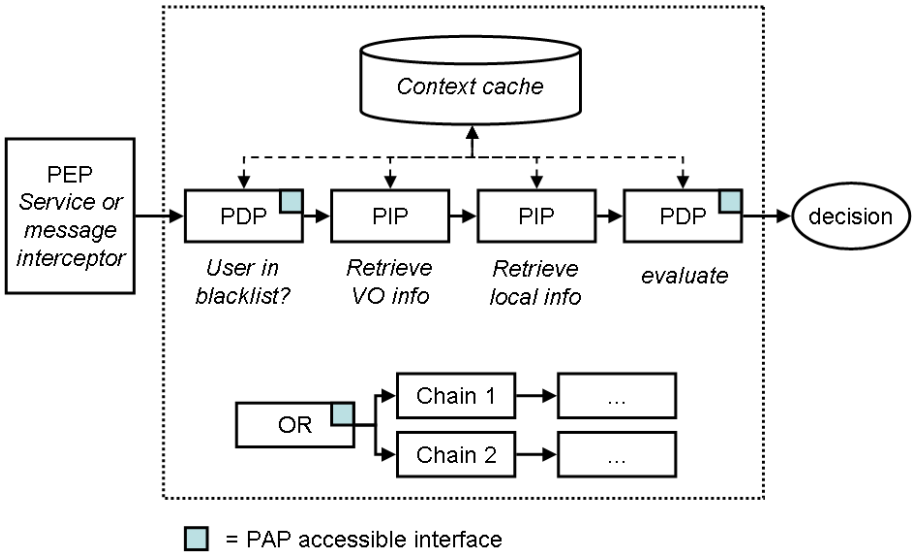

Figure 5: Conceptual model of the authorization framework, invoked from a evaluation point (PEP) in the application or service container. The authorization request and associated metadata is subsequently routed through a chain of information collectors (PIP), policy-driven decisions (PDP), or modules that perform logical operations on results from an underlying collection of chains (in the picture depicted by an OR).

PROVIDERS AND PLUGINS

Several plug-ins will be available both as a guide on how to use the AuthZ framework, and as a sufficient base for setting up a simple, but complete authorization system. These include

- A plugin that verifies any VOMS tokens associated with the authenticated user, checking VO membership as well as role and group attributes.
- A PDP/PIP that authorize an authenticated credential subject using a Globus gridmap-file
- A "BlackList" PDP that checks the authenticated user against a local list of banned users
- A PDP that can integrate with external authorization sources using the GGF OGSA AuthZ callout protocol
- A policy enforcement module that performs additional checks on the user's credentials, for instance checking that the validity of a proxy certificate is within the established operative guidelines

A more elaborate listing is given in the Site Access Control document, EGEE deliverable DJRA3.2 [23], but some of the PIP and PDP plugins that are planned or already supported include

6.6. SECURITY CONSIDERATIONS

It is vital that any components of the AuthZ infrastructure are managed and operated as securely as the corresponding parts of the authentication infrastructure. For instance, a repository containing the trusted

signature keys of the AAs needs to be protected as securely against infiltration as a repository of the authentication trust anchors (CAs).

7. ISOLATION AND SANDBOXING

Access to remote computing or storage resources should ideally be completely transparent to the user. Furthermore, the user's actions should be *sandboxed* in order to minimise the impact on the local system. Within the sandbox the user is free to do whatever they want and the removal of the sandbox as soon as they are finished minimises the effect on the local system. The generic creation of sandboxes, only based on an entity's Grid credentials such as VO membership, relieves the local system administrator from the burden of administering individual users.

For the creation, access control and life-time management of sandboxed environments, we use Workspace management service (WSS) [7]. Further details on this solution can be found in project deliverables DJRA1.4 and DJRA3.4.

By allowing a process to only access its home directory and a standard location for its temporary files (on Linux/Unix systems defined by the `$TMPDIR` environment variable) the process is reasonably sandboxed, but it is very difficult to enforce this. In particular, once local access is obtained, the more advanced so-called root kits will be able to compromise almost any Linux installation (patched or not) and provide the adversary with root access. We don't foresee that this situation will change to the better anytime soon, and therefore work under the assumption that an adversary's access to a normal account can and will result in root compromise, given enough time.

A more elegant way to solve this problem is to run the Grid user processes inside a virtual machine, which provides the user job with a complete operating system. The entire system of the virtual machine is contained in a single file and can be discarded after the Grid user processes are finished. The long-term negative effects of an adversary gaining root privileges in a virtual machine are negligible and the network connections to/from a virtual machine can be easily controlled.

While several successful proof-of-concept experiments with virtual machines such as VMWare[30] and Xen[31] have been reported to date, work is still underway: deployment, packaging, management, performance, installation invasiveness (modifications to the hosting operating system are often needed) and overhead reduction, in particular in resource consumption and memory footprint, are some of the problems that are still being investigated.

7.1. SECURING THE HOSTED TO NATIVE INTERFACE

Hosted services (Web Services in containers) and application servers need to carry out actions on the native system, for which a change of 'user space' is needed. This requires a trusted 'setuid' functionality that can provide a correct mapping to a local account, based on the user's Grid credentials, and put restrictions on the resulting process driven by local policy (for instance, only allowing executable). Implementations providing this setuid functionality exists, such as the LCMAPS plugin framework and the GridSite [26] extensions to the Apache web server. Of interest is also support for industry standard system interfaces such as PAM/ NSS.

The detailed design of the setuid components is addressed in the updated version of deliverable DJRA3.2.

7.2. NETWORK ISOLATION

By reducing connectivity using firewall or NAT[32] techniques, a site can reduce its overall vulnerability to outside attacks. Connectivity goes both ways: a site may choose to cut off inbound as well as outbound connectivity for various, often well motivated, reasons.

[30] http://www.vmware.com
[31] http://www.xensource.com
[32] NAT: Network Address Translation.

In principle, only the services that are essential for the site to operate in a Grid environment need inbound connectivity by default. They also need outbound connectivity to be able to contact other principal services at other sites.

All other services and resources that will handle the requested operations may have connectivity constraints by default. For instance, this would include machines that are designated as worker nodes in a compute element. The site externally publishes its policy on possible connectivity constraints.

For current internet-capable applications to work in such a setting, a user must be able to trigger a dynamic change in the site policy regarding its inbound and outbound connectivity. For instance, an online database may be consulted over an HTTPS connection, or a real-time UDP data feed needs to be propagated to an application component running elsewhere.

This dynamic connectivity problem can be solved with a Dynamic Connectivity service that temporarily grants the ability to connect to a requested location with a specified protocol. This service would be able to uphold a site policy that describes the network capabilities and restrictions with regards to IP-address/dns-name, port numbers, protocols and bandwidth. It would be capable of applying changes to network restricting units, such as the iptables/ipchains configuration on a worker node, or by directly reconfiguring a router through a management interface, and could be implemented as a thin wrapper around the authorization framework for policy decision and enforcement.

The Dynamic Connectivity service could also be used to create a central manageable point for a site to keep track of its current connectivity needs. It is an optional component that is considered outside of the core security architecture for the time being, but we mention it for completeness and work with collaborators to conduct research and develop experience thorough experiments with prototypes of this service.

7.3. Security Considerations

Sandboxing and isolation techniques often give a false sense of security. Improper system management and operational procedures can never be made up for by additional (site) perimeter defense.

8. USE-CASES

In this section, we should ideally present a use-case for each and every component that is listed in Table 1 and/or presented as part of our security architecture, to assess its necessity and showcase how it will interact with other services and components. However, as stated in Section 1., our security architecture is an evolutionary continuation where we have leveraged ideas and experiences gained in previous projects. This includes sharing the underlying motivation and some of the use-cases documented elsewhere: instead of replicating them here, we will simply refer to them.

The EDG's security design document [20] showcases several use cases in the form of UML sequence diagrams, of which some are still valid: in particular how to obtain a certificate and VO user registration. Those use-cases will not be replicated here.

The gLite architecture document[40] showcases how some selected services make use of credential stores, authentication, attribute authorities, authorization framework and logging (for auditing purposes). Those use-cases will not be replicated here.

Figure 2 and 3 are described in Sections 3.6. and 5., respectively. While not UML renderings, we feel that they provide an adequate description of scenarios where pseudonymity and Encryted Storage key management services are used, and will not be replicated here.

For clarity, the slightly complex Figure 1 described in Section 1.7. is redrawn as UML sequence diagrams and explained in more detail below.

8.1. A GENERIC SERVICE REQUEST FLOW

Figure 1 depicts an overview on how the components in the security architecture interact in a request, showcasing most of the security architecture components coming into play. This sequence is quite long and therefore divided into two parts, Figures 6 and 7.

Please note that in a normal deployment scenario, many steps are optional. For instance, the deployment of and need for interaction with a Site Proxy service depends on local network policy of the site hosting the resource.

The sequence in Figure 6 depicts a user obtaining credentials, establishing a secure communication channel with a service, over which credentials are being delegated. This is described in more detail below. References to sections in this document are made throughout for further information. While logging is an important detail of the security architecture, it has not been thoroughly documented but only mentioned in the text below, to reduce complexity.

1. The user identifies to a credential store by way of some bootstrap mechanism, such as password, one-time-token or kerberos ticket. (Sections 3.2.,4.3.,4.2.)
2. The credential store validates the user's request and provides the user with authentication credentials. If the credential store is active, it logs the event. (Sections 3.2.,4.3.)
3. The user authenticates to the necessary attribute authorities (or policy assertion services) to obtain authorization assertion tokens that assert a user's privilege to access/make use of the resource. (Section 6.)
4. The attribute authority validates the user's request and issues the request authorization assertions. The event is logged. (Section 6.)
5. The authorization assertions are returned to the user and included as part of the prepared service request, by inclusion in a proxy certificate. (Section 3.1.)

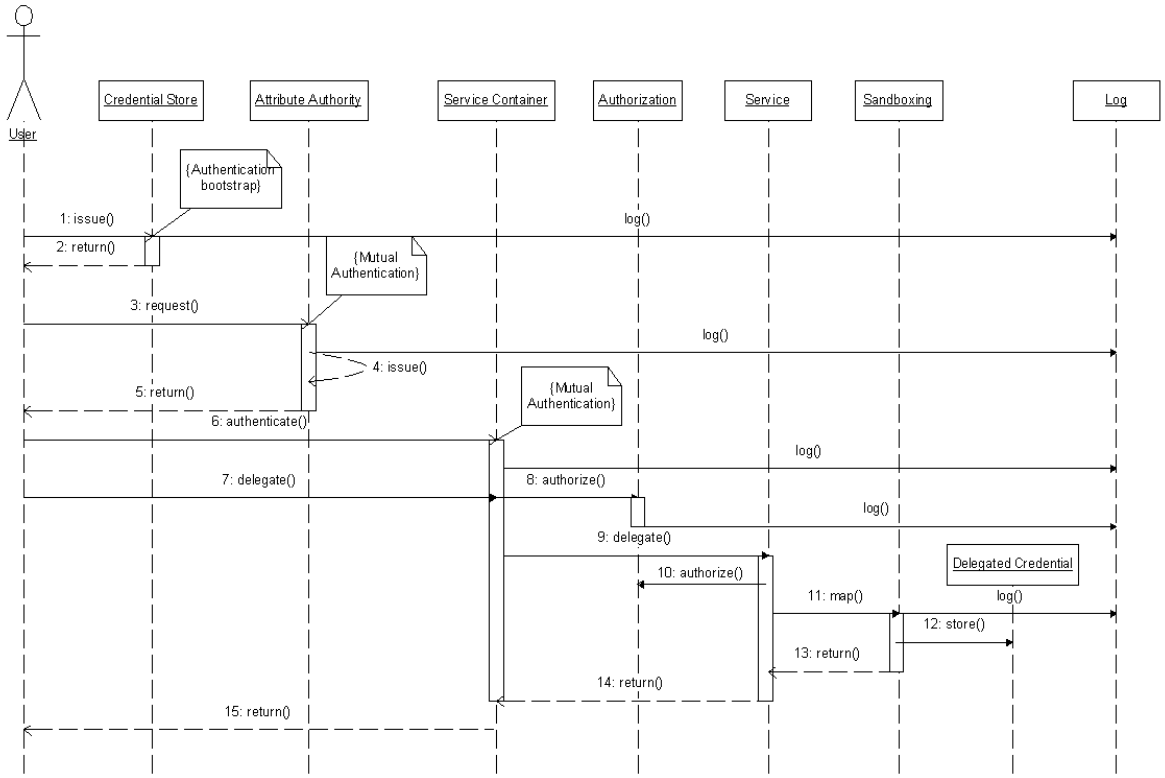

Figure 6: Sequence diagram over security architecture components and their interactions (1 of 2).

6. Transport-level security: The user and the computer on which the service container runs perform a handshake operation. This handshake authenticates and validates each other's identities and establishes a secure communication channel across the (open) network with integrity, authenticity and confidentiality protection. SOAP messages can later be conveyed over this connection. The established connection event is logged.

   In the case of message-level security, the user's SOAP message is routed to the service, which in turn takes care of the authentication, integrity and confidentiality, for instance by validating the signature on a user's request, and/or en/decrypting the content of the SOAP message body and selected headers in accordance with conventional XML and Web Services standards.[33] (Section 1.7.)

7. The user initiates a delegation of privileges, to enable to resource act on its behalf by way of a proxy certificate. (Section 6.1.)

8. (Optional) The service container authorizes incoming messages using the authorization framework, to assert whether the user has the necessary priviliges to interact with the service. The result is logged. (Section 6.5.)

9. The service container routes the SOAP message to the correct service implementation and invokes the request service operation.

10. The service implementation in turn makes additional authorization (and in the case of message-level security, authentication) checks, possibly reusing the same authorization framework as the service container. The result is logged. (Section 1.7.,6.5.)

[33] It is a common implementation approach that these operations are handled by the service container, but we ignore that to clarify an ideal system architecture enabling end-to-end message-level security.

11. (Optional) The service uses sandboxing techniques to access the local resource in a confined and regulated manner to reduce the risk for vulnerabilities, for instance by mapping a into different local user account using a setuid component. The mapping event is logged. (Section 7.1.)

12. The delegated credential is stored.

13. The service prepares a message with the result of the operation. In case message-level security is used, the message is signed.

14. The service container adds necessary enveloping to the message. In case transport-level security is used, the message is encrypted as it is being sent.

15. The user retrieves, authenticates and unwraps the result message of the operation.

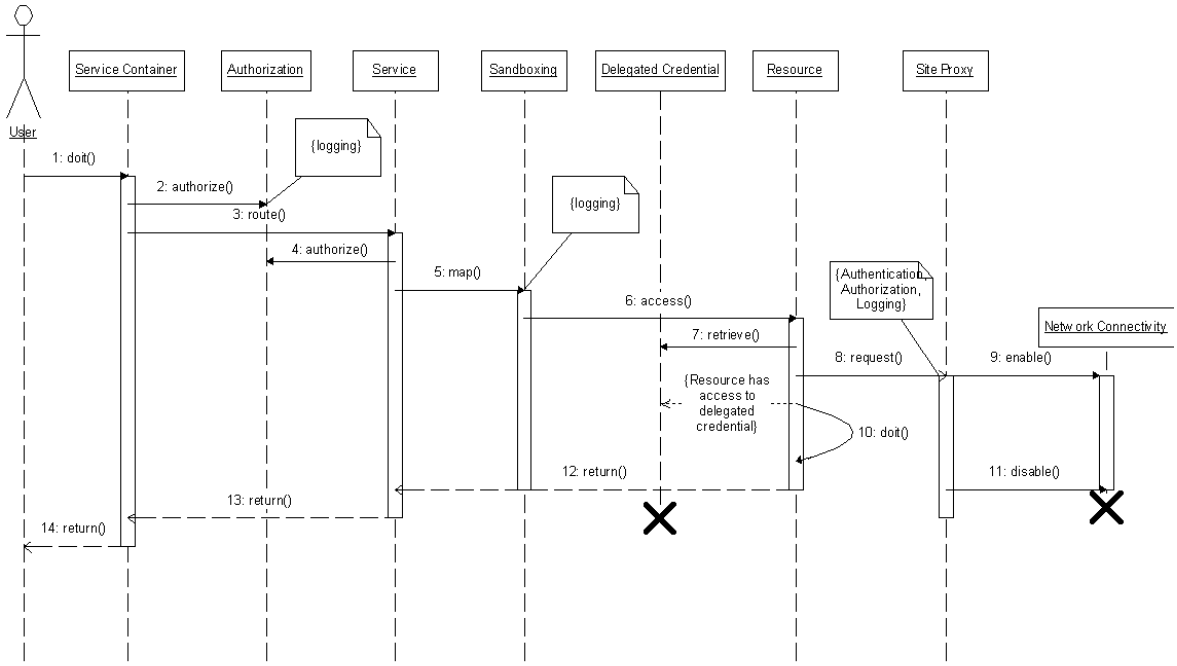

Figure 7: Sequence diagram over security architecture components and their interactions (2 of 2).

The sequence in Figure 7 is a continuation of Figure 6. Here, the user makes use of a resource managed by the service. The resource (for instance, an application) in turn interacts with other services and resources distributed across the Grid. In order to obtain network connectivity, the local Site Proxy is used. In this sequence the logging component is not depicted at all for clarity, but its use is mentioned in the text below.

1. Steps 1 through 6 are analogous to steps 6 through 11 in Figure 6.

7. The resource obtains the delegated credentials, if such were provided. In case the delegated credential was stored in an active certificate store, the event is logged. (Section 4.3.)

8. The resource make use of the delegated credentials to contact the Site Proxy. The Site Proxy in turn authenticates and authorizes the request. The event is logged. (Section 7.2.)

9. The Site Proxy enables network connectivity, for instance by modifying the local firewall configuration. (Section 7.2.)

10. The resource is used for the requested use by the user (for instance, to run an application).
11. The Site Proxy closes down network access after some period of time, or by request from the resource.
12. through 14 are analogous to steps 13 through 15 in Figure 6.

REFERENCES

[1] Alfieri R. et al. VOMS, an Authorization System for Virtual Organizations. In *Grid Computing, First European Across Grids Conference*, 2004.

[2] D. W. Chadwick, A. Otenko, and E. Ball. Implementing role based access controls using x.509 attribute certificates. *IEEE Internet Computing*, 7(2):62–69, 2003.

[3] CryptoCard Corp. http://www.cryptocard.com.

[4] D. Skow, Private communication. http://www.ppdg.net/docs/Papers/PPDG_25.pdf.

[5] Andy Hanushevsky et al. The SLAC Virtual Smart Card project.

[6] C. Rigney et al. RFC2865: Remote Authentication Dial In User Service (RADIUS). http://www.ietf.org/rfc/rfc2865.txt.

[7] Kate Keahey et al. Workspace Management Service. http://www.mcs.anl.gov/workspace.

[8] M. Myers et al. RFC2560: X.509 Internet Public Key Infrastructure Online Certificate Status Protocol (OCSP). http://www.ietf.org/rfc/rfc2560.txt.

[9] R. Housley et al. RFC3280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. http://www.ietf.org/rfc/rfc3280.txt.

[10] S. Tuecke et al. RFC3820: Internet X.509 Public Key Infrastructure (PKI) Proxy Certificate Profile. http://www.ietf.org/rfc/rfc3820.txt.

[11] European DataGrid (EDG). http://www.edg.org/.

[12] S. Farrell and R. Housley. RFC3281: An Internet Attribute Certificate Profile for Authorization. http://www.ietf.org/rfc/rfc3281.txt.

[13] Niels Ferguson and Bruce Schneier. *Practical Cryptography*. John Wiley & Sons, 2003.

[14] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. In *ACM Conference on Computers and Security*, pages 83–91. ACM Press, 1998.

[15] Apache Software Foundation. Axis SOAP container. http://ws.apache.org/axis.

[16] Apache Software Foundation. Jakarta Tomcat servlet container. http://jakarta.apache.org/tomcat.

[17] Global Grid Forum. http://www.ggf.org/.

[18] Globus Alliance. http://www.globus.org/.

[19] GGF Site AAA Research Group. Grid Authentication Authorization and Accounting Requirements. http://forge.gridforum.org/projects/saaa-rg/document/.

[20] Security Coordination Group. EDG Deliverable 7.6: Security System Design. https://edms.cern.ch/file/344562/2.0/.

[21] I. Foster and C. Kesselman and S. Tuecke. The Anatomy of the Grid. *The International Journal of High Performance Computing Applications*, 15(3):200–222, Fall 2001.

[22] EGEE JRA3. EGEE Deliverable 3.1: Global Security Architecture. https://edms.cern.ch/document/487004/.

[23] EGEE JRA3. EGEE Deliverable 3.2: Site Access Control. https://edms.cern.ch/document/523948/.

[24] J. Kohl and C. Neuman. RFC1510: The Kerberos Network Authentication Service (V5), 1993. http://www.ietf.org/rfc/rfc1510.txt.

[25] LHC Computing Grid Project (LCG). http://lcg.web.cern.ch/LCG/.

[26] Andrew McNab. GridSite. http://www.gridsite.org.

[27] Alberto Di Meglio and Joachim Flammer. A Proposal To Standardize Configuration, Logging and Error Handling. https://edms.cern.ch/document/486630.

[28] O. Mulmo. Comparison of different security infrastructure implementations. Presented at the GlobusWORLD 2004 Security workshop. http://grid.ncsa.uiuc.edu/gw04-security/GW04-SecWkshp-trust-comparison.ppt.

[29] J. Novotny, S. Tuecke, and V. Welch. An online credential repository for the grid: Myproxy. In *Proceedings of the Tenth IEEE International Symposium on High Performance Distributed Computing (HPDC-10)*, August 2001.

[30] OASIS. eXtensible Access Control Markup Language (XACML). http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml.

[31] OASIS. Security Assertion Markup Language (SAML). http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security.

[32] L. Pearlman, V. Welch, I. Foster, C. Kesselman, and S. Tuecke. A Community Authorization Service for Group Collaboration. In *Proceedings of the IEEE 3rd International Workshop on Policies for Distributed Systems and Networks*, 2002.

[33] UMich Kerberos PKI Project. http://www.citi.umich.edu/projects/kerb_pki/.

[34] IRTF AAAarch RG. RFC2904: AAA Authorization Framework. http://www.ietf.org/rfc/rfc2904.txt.

[35] SecureID. http://www.ctrl-key.co.uk/index.htm.

[36] A. Shamir. How to share a secret. *Communications of the ACM*, 22:612–613, 1979.

[37] F. Siebenlist, T. Mori, R. Ananthakrishnan, L. Fang, T. Freeman, K. Keahey, S. Meder, O. Mulmo, and T. Sandholm. The Globus authorization processing framework, April 2005. Presented at the New Challenges for Access Control workshop, Ottawa, CA. http://lotos.site.uottawa.ca/ncac05/.

[38] EGEE JRA3 Team. JRA3 Gap Analysis, v2. https://edms.cern.ch/document/473230.

[39] EGEE JRA3 Team. User requirements survey. https://edms.cern.ch/document/485295.

[40] EGEE Middleware Design Team. EGEE Deliverable 1.4: EGEE Middleware Architecture. https://edms.cern.ch/document/594698/.

[41] EGEE NA4 Team. Application requirements database. http://egee-na4.ct.infn.it/requirements/.

[42] V. Welch, T. Barton, K. Keahey, and F. Siebenlist. Attributes, Anonymity, and Access: Shibboleth and Globus Integration to Facilitate Grid Collaboration. In *4th Annual PKI R&D Workshop*, 2005.

[43] V. Welch, I. Foster, C. Kesselman, O. Mulmo, L. Pearlman, S. Tuecke, J. Gawor, S. Meder, and F. Siebenlist. X.509 Proxy Certificates for Dynamic Delegation. In *3rd Annual PKI R&D Workshop*, 2004.