AGH UNIVERSITY OF SCIENCE AND TECHNOLOGY

IN CRACOW, POLAND

FACULTY OF ELECTRICAL ENGINEERING, AUTOMATICS, COMPUTER SCIENCE AND ELECTRONICS

INSTITUTE OF COMPUTER SCIENCE

Security in Component Grid Systems

Master of Science Thesis

Michał Dyrda
Matricula: 120525

Computer Science

Supervisor: Marian Bubak, PhD
Advice: Maciej Malawski, MSc

CRACOW, JUNE 2008

AKADEMIA GÓRNICZO-HUTNICZA

IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI, INFORMATYKI I ELEKTRONIKI

KATEDRA INFORMATYKI

Bezpieczeństwo w komponentowych systemach gridowych

Praca magisterska

Michał Dyrda
Nr albumu: 120525

Kierunek: Informatyka

Promotor: dr inż. Marian Bubak
Konsultacja: mgr inż. Maciej Malawski

KRAKÓW, CZERWIEC 2008

`

Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie i że nie korzystałem ze źródeł innych niż wymienione w pracy.

Abstract

The subject of this thesis is a detailed analysis and development of security in grid component systems on the example of MOCCA, a CCA-compliant framework build over H2O distributed computing platform.

The work is focused on providing H2O with an authentication mechanism that will be both secure and compliant with solutions commonly used in grid systems nowadays. The created authenticator is based on asymmetric cryptography with additional features provided by Globus Security Infrastructure. It reuses existing external libraries and architecture provided by H2O, employing required Public Key Infrastructure.

Within the scope of this work, existing authentication mechanisms of H2O with some related aspects of authorization as well as communication security were analyzed and described. A complete process of authenticator development was carried out. Finally, the created authenticator as well as the overall system were brought under tests, which proved their safeness and usability.

The thesis is organized as follows:

In Chapter 1 target environment and motivation for security, especially in the target systems, are introduced. Then the objective subset of security issues and goals of this thesis are stated. Chapter 2 presents background – it recalls key security concepts and describes current security architecture provided by the target system, emphasizing its missing features. Related work, with emphasis on GSI, is presented in Chapter 3. In Chapter 4 detailed requirements of the solution are specified and the general concept of the GSI Authenticator is outlined. Chapter 5 is devoted to implementation aspects of the solution. Chapter 6 provides an exhaustive description of example usage of the authenticator in target systems together with required configuration and Public Key Infrastructure. Moreover, it gives an answer on how the authenticator meets the usability requirements by performing execution and performance tests as well as a detailed threat analysis of the overall system. Chapter 7 concludes the work by enumerating achieved goals and providing some suggestions for future development.

Keywords:

Grid Computing, H2O, MOCCA, components, Common Component Architecture, security, authentication, Public Key Cryptography, PKI, certificates, Globus Toolkit, GSI, delegation, proxy certificates

Acknowledgements

I would like to express my gratitude to Marian Bubak – supervisor of this work, for his guidance and advices, Maciej Malawski, for valuable consultations, commitment and time, as well as Dawid Kurzyniec – main H2O author, for the support received during development of the system.

This work was made possible owing to the ViroLab and CoreGrid projects.

http://www.virolab.org/ http://www.coregrid.net/

Table of contents

List of Figures

Abbreviations

(in alphabetical order)

CA	- Certificate Authority
CCA	- Common Component Architecture
CRL	- Certificate Revocation List
CSR	- Certificate Signing Request
DN	- Distinguished Name
GSI	- Globus / Grid Security Infrastructure
GT	- Globus Toolkit
HPC	- High Performance Computing
HTTPS	- HyperText Transfer Protocol Secure
J2EE	- Java 2 Platform, Enterprise Edition
J2ME	- Java 2 Platform, Micro Edition
JAAS	- Java Authentication and Authorization Service
JCA	- Java Cryptography Architecture
JCE	- Java Cryptographic Extension
JRE	- Java Runtime Environment
JRMP	- Java Remote Method Protocol
JSSE	- Java Secure Socket Extension
LDAP	- Lightweight Directory Access Protocol
OCSP	- Online Certificate Status Protocol
PKI	- Public Key Infrastructure
PKIX	- Public-Key Infrastructure (X.509)
RA	- Registration Authority
RDN	- Relative Distinguished Name
RMIX	- RMI eXtended / RMI MIXture
RPC	- Remote Procedure Call
RSA	- Rivest, Shamir, Adleman
SOAP	- Simple Object Access Protocol
SSL	- Secure Socket Layer
TLS	- Transport Layer Security

Chapter 1. Introduction

The subject of this thesis is security in grid component systems on the example of H2O and MOCCA. At the beginning, some introduction will be performed in this chapter. First we will familiarize with the environment that will be targeted by this thesis. Afterwards aspects of security in grid component systems will be described and the importance of security at the example of target platforms will be shown. Finally the sub-goals of this thesis will be stated.

1.1 Target environment[1]

As the Grid system evolved, programming of mostly compute intensive, distributed, scientific applications that would utilize its growing resources, was becoming more and more complicated. A suitable programming model and a way of virtualization that would hide the complexity of heterogeneous environment, became two key challenges to compete. One of the proposed approaches was to use an extended component model with a virtualization layer applied. As one of the results, a CCA-standardized MOCCA framework together with an H2O platform, which it is based on, were designed. In this section they will be shortly introduced, with emphasis on the security aspects, which are important from the point of view of this thesis.

1.1.1 Component-based approach

In order to understand the architecture of target systems let us first briefly recall what the components are and see how the component-based approach can address Grid complexity issues.

Components are independent units of software of specified (reusable) functionality that can be dynamically composed and interact with other components using (and only through) well-defined interfaces (input and output ports). They are hosted by specific containers that are responsible for other services, such as communication, data storage or security.

[1] Based on [1] and [2]

Some features of component-based approach enable it to addresses Grid complexity issues:

- **Virtualization and scalability**

Component-based applications can be composed from (relatively) simple blocks hosted by containers running on multiple grid sites. The physical location of containers is not relevant for using them, they can even appear as a single logical resource. Both pool of containers and number of individual components can be managed dynamically, allowing to adjust the load according to number of owned resources. Also new components can be deployed when needed. Together with a lightweight platform it makes the approach scalable to different environments, from laptops to HPC clusters.

- **Communication**

Instead of Web Services, components can be directly connected without need to pass invocation data via central workflow engine. Parallel connections are allowed as well. Furthermore, they do not require SOAP as a protocol. In fact, containers allow for communication interoperability facilitating many communication protocols.

- **Adaptation to unreliable Grid environment**

Dynamic and interactive reconfiguration of connections, locations and bindings enables to adjust to the changing state of Grid network.

- **Ease of development**

Components' (relative) simplicity makes the development of large systems even more convenient. Moreover, developers can focus on components functionality itself, leaving cooperation with environment to the container. Together with a specified inter-object communication it all makes the approach to be considered as a step beyond object-oriented design, finally making cross-projects code reuse available in a practical way.

This part introduced the component-based approach and its usefulness in a Grid environment. Now we can look at some specific standards and solutions.

1.1.2 CCA as a standard for component-based approach

As a standard for component-based approach, the Common Component Architecture (CCA) was submitted. We describe it as the introduction to the MOCCA framework that is compliant with this standard.

CCA was designed by members of a Common Component Architecture Forum[2]. It defines 'uses' and 'provides' ports and a SIDL language to describe them (Figure 1).

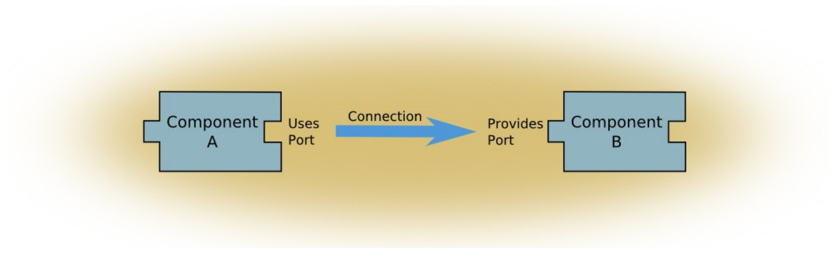


Figure 1. Common Component Architecture ports[3].
'Provides' ports are public interfaces that a component realizes whereas 'uses' ports specify dependency of other components' 'provides' ports, which it requires to use.

There exist multiple different frameworks for building applications compliant with CCA standard, i.a. loosely coupled, distributed XCAT or tightly coupled CCAFFEINE (with support for Babel and MPI). This thesis however focuses on the MOCCA framework based on the H2O platform. Both of them will be introduced in the subsequent sections of this chapter.

1.1.3 H2O as the underlying platform for MOCCA[4]

Now let us familiarize with H2O as the underlying platform of MOCCA. Following chapters of the thesis will focus on H2O security mechanisms, therefore this part is essential to understand them. By reading this part we will get to know the application and usage of H2O, learn its notation, actors and some interesting features.

As described by authors in [4]: *„H2O is a middleware platform for building and deploying distributed applications. H2O is Java-based, secure, scalable, stateless, and lightweight."* The main difference towards other component frameworks, such as J2EE, is that not only the container owner but any authorized third party (e.g. grid software developer) is able to deploy services and use them afterwards.

[2] http://www.cca-forum.org/
[3] Based on figure from [1]
[4] http://dcl.mathcs.emory.edu/h2o, based on [3]

Nomenclature:

H2O names for containers and components are respectively kernels and pluglets. The kernels are owned by Providers, who define their access control policy, start them up and terminate them. The pluglets are implemented by Developers and mostly placed as a digitally signed package (e.g. .jar file) in a repository. Deployers deploy them in the kernel and can do some initial aggregation. Finally, the pluglets are used by Clients by calling their remote methods and optionally by aggregating them if needed. Multiple roles can be assigned to one person.

All of the presented issues of H2O platform are used by the MOCCA framework, which will now be introduced.

1.1.4 MOCCA as an example of CCA-compliant Framework[5]

This part introduces application and features of MOCCA framework. It aims to show, how the security mechanisms of MOCCA and H2O are shared in order to specify the object of this thesis' interest.

MOCCA is a distributed component framework compliant with CCA standard, which is build on top of H2O platform. It allows building component applications on distributed resources available through H2O. Although MOCCA is supposed to support multiple programming languages, current version called MOCCA_Light is a pure-Java implementation. Its architecture is presented in Figure 2.

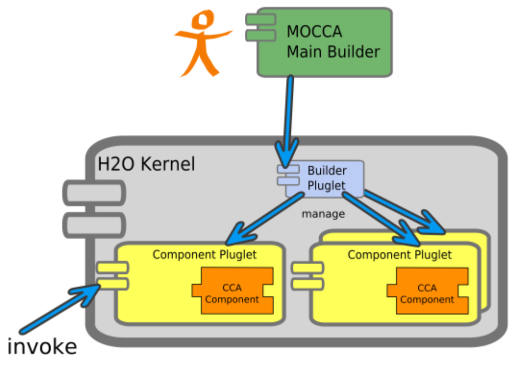


Figure 2. MOCCA = H2O + CCA [6].
Individual components, which are java classes implementing CCA interfaces (cca.Port, cca.Component), are mapped to separate pluglets. The deployed pluglets are managed and combined using MOCCA Builder and its Builder Pluglets.

[5] http://www.icsr.agh.edu.pl/mambo/mocca, http://mocca.icsr.agh.edu.pl/, based on [1]
[6] Based on figure from [1]

What's very important from the point of view of this thesis – **MOCCA uses the security mechanisms of H2O, sharing the same challenges and using the same solutions**. That means that competing the MOCCA challenges comes down to competing them in H2O, which in fact is done as the result of this work. **Therefore in subsequent chapters we will only refer to H2O platform remembering that the same issues apply to the MOCCA framework as well.**

1.1.5 Summary

This chapter gave an overview of a target environment, in which component-based approach and CCA standard led us to introduce the H2O platform and MOCCA framework, which security mechanisms are being focused by this thesis. This introduction will help us to understand the motivation and issues of security in Grid component systems, especially on the example of H2O and MOCCA.

1.2 Motivation for security

It is no unusual situation for Grid system that plenty of resources handling confidential data are shared on multiple sites by a large number of users from a variety of organizations. Security of such system is a critical issue, because not only data, but also hosts, resources and computations have to be secured from improper access. Based on such characteristic, several aspects of Grid security will be presented in this chapter. H2O, as a Grid-oriented software, has to meet them as well – this will be described afterwards.

1.2.1 Security concepts in Grid systems

The aim of this part is to introduce the main aspects of security in computer systems and analyze them taking into account the characteristic features of Grid environment.

- **Authentication**

The identity of every user has to be confirmed in order to enter the system. Additionally, authentication of the server can ensure that resources and data are not provided by an attacker.

- **Authorization**

Every authenticated user has to be authorized to access individual resources. It is difficult mostly because of the scalability and evolution of Grid systems - both number of hosts and users can be large (reaching thousands) and dynamic - because of joining and leaving all the time. Furthermore, resources are often owned by multiple administrative domains, which makes the administration difficult and demands complex, distributed authorization policies.

- **Single Sign-On and delegation**

The trade-off between user friendliness and system robustness is still an important issue. One of the aspects is to enable users to use multiple resources without the need to authenticate multiple times on their providers' hosts; moreover – to allow the system to work on user's behalf (e.g. to allow brokering services acquire resources on behalf of the user) in the same manner.

- **Communication security – message integrity and confidentiality**

Secure communication has to be encrypted and signed. Encryption is a way to ensure confidentiality - prevents the communication from being eavesdropped by an unauthorized third party. Digital signature ensures the communicating parties that the messages have not been modified on their way (e.g. the target account number of our transfer has not been changed).

- **Sandboxing**

Users are not always victims of security vulnerabilities – they can pose threats as well. On one hand, we have to protect the code running on shared computational resources from others; on the other, we have to ensure that no user's code will negatively affect the system, other computations or data.

- **Audit**

Violations can occur, indeed. In such situation it would be reasonable to have some logs and chains of accountability for actions that took place on the system, to find the responsible user.

- **Accounting**

In commercial systems the ability to limit or charge for consumption of resources is demanded.

Please notice that all of the 'A..' aspects (commonly called together as AAAA) need to be aware of resources distribution, which makes them more difficult than in casual systems.

1.2.2 Security concepts on the example of H2O

Now that we know the characteristics of security issues in Grid systems we can take a look at how it applies to the H2O platform. Finally we will select the aspects, which this thesis will be focused on.

Because the kernels are publicly accessible, it is important to authenticate users accessing them to verify their identity. The authentication is performed using some credentials, i.a. username and password. Authorization, in turn, needs to be introduced in order to distinguish between users with different permissions (e.g. for deploying pluglets or accessing them…). Also the code can be authenticated by signing the pluglets. Community policy management may need to be introduced in case of Virtual Organizations.

Moreover, one of the concepts is to facilitate direct links between pluglets. Since each pluglet can be kept in separate H2O kernel on multiple Grid sites, each connection between kernels has to be authenticated and authorized as well. That would demand a user to authenticate multiple times, which would be very inconvenient. To reduce the number of times the user must authenticate (e.g. enter his passphrase), credential delegation has been introduced (see Figure 3). The user authenticates only once upon connecting to first component; afterwards code running in the component can authenticate itself to another component on behalf of the user/client. That is called Single Sign-On.

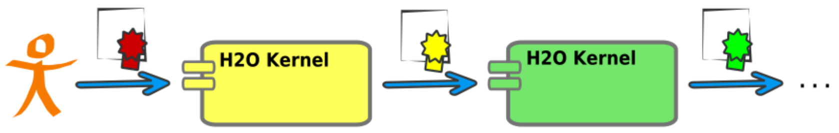


Figure 3. Credential delegation in H2O.
A user authenticates only once upon connecting to first component. Credential presented by the user is delegated for subsequent connections. Code running in the component can authenticate itself to another component on behalf of the user/client.

Furthermore, communication security can sometimes be a crucial aspect as well. Confidentiality and integrity can be enabled in order to be sure that no other peer will be able to understand or modify our communication.

Finally, users' code has to be separated from each other (run in sandbox) in order to prevent them from interfering and being threat to each other.

Audit and accounting have not been concerned yet

Using the example of H2O platform, this section has shown, how broad and complicated the aspects of security in grid component systems are. **As the topic of this thesis, the aspects of authentication, together with additional elements, as delegation, single sign-on and some part of authorization have been selected.** Their detailed analysis together with review of available solutions as well as implementation of selected one will be carried out in subsequent chapters.

1.2.3 Summary

This part introduced the aspects of security in grid component systems and related them to the example of H2O platform. At the end the subset of aspects, which are going to be considered by this thesis, were selected.

1.3 The MSc Thesis goals

The main goal of the thesis is to analyze, design and develop a solution that will be able to rise to the presented subset of security challenges in H2O. This chapter lists the sub-goals that will be realized by the thesis in order to achieve this goal.

- **Identification and analysis of security architecture and shortages in H2O**

At the beginning we need to analyze the current state of security mechanisms already implemented in H2O and find these aspects that are missing and have to be added. All requirements have to be identified in order to state a problem, this thesis will attempt to solve. In order to achieve this goal, it is necessary to deeply familiarize oneself with the system.

- **Overview of available solutions for H2O security enhancements**

In order to find the best solution, we need to conduct a research of modern security technologies used for authentication and examine if they are capable of answering the issues specified in the previous point.

- **Concept and development of new security system for H2O**

The analysis of possible solutions should lead to a concept of security system that will address all the identified challenges. The main goal of this thesis will be to design and implement a complete and comprehensive solution for the stated problem. Moreover, the system environment should be analyzed and required modifications should be identified and described.

- **Proving the correctness and usefulness of the created solution**

In order to prove the usefulness and robustness of the solution, an example usage description as well as several tests are to be created. The main point of this part, beside code and performance tests, will be the threat analysis. The goal will be to check the system against all known attempts of violation so as to prove that it is immune to them.

Since the complete H2O authentication architecture is very flexible and allows to choose between the performance and security level, the analysis of possible usages along with their pros and cons will be performed.

- **Build, configuration and usage description**

Because of a weak H2O documentation and encountered difficulties with building, configuring and using it, the experience gained during the development of the GSI Authenticator is going to be written down in form of a detailed description of the required actions and steps in order to simplify successive developers' work.

- **Identification of future work**

The subject of this thesis will be the next but not the last step of H2O development. At the end of the thesis available areas of future work will be identified, taking into account especially the newest trends in Grid computing and the scheduled development of the target systems.

1.4 Summary

The chapter introduced both the environment, to which the subject of this thesis is targeted, and the motivation for security in such an environment. The subset of security issues that the thesis will concentrate on, were selected – namely authentication together with delegation and single sign-on as well as some part of authorization. Finally, the goals of this thesis were stated.

Chapter 2. Background

In previous chapter an introduction of target environment was presented. This following chapter will provide a detailed description of existing H2O security architecture, together with a required theoretical background. First some key security concepts will be recalled that is Transport Layer Security and its foundations: Public Key Cryptography and Public Key Infrastructure. Their knowledge is essential for understanding the subsequent parts of the thesis. Furthermore, the existing H2O authentication and authorization architecture will be precisely described to provide us with the knowledge of current state of the art. Finally some missing features of the authentication mechanism will be discovered and stated.

2.1 Key concepts

As an introduction to further considerations a few security mechanisms are going to be reminded with emphasis on aspects, which are going to be particularly essential. The TLS protocol together with Public Key Cryptography and Public Key Infrastructure are going to be described.

2.1.1 Transport Layer Security

Transport Layer Security (TLS) [5], in its previous version known as Secure Sockets Layer (SSL) [7], is a protocol for establishing a secure channel across mistrusted networks. It ensures connection encryption, integrity (using digital signatures) and non-repudiation. Moreover, authentication is provided. Typically one side is authenticated only – the common example is using the HTTPS protocol for entering secured websites, which identity we need to be sure, e.g. bank services. On the other side some server may want to authenticate the user in order to share its resources in a secure manner. If the identity of both sides has to be confirmed – the mutual authentication takes place.

[7] The names TLS and SSL will be used replaceably in this thesis

Communication using TLS involves three basic phases:

- Peer negotiation for common algorithms (ciphers, authentication algorithms, message authentication codes)
- Key exchange and authentication (public key cryptography and certificates)
- Message encryption and message authentication (using symmetric cryptography)

TLS is based on Public Key Cryptography and Public Key Infrastructure, which are now going to be presented.

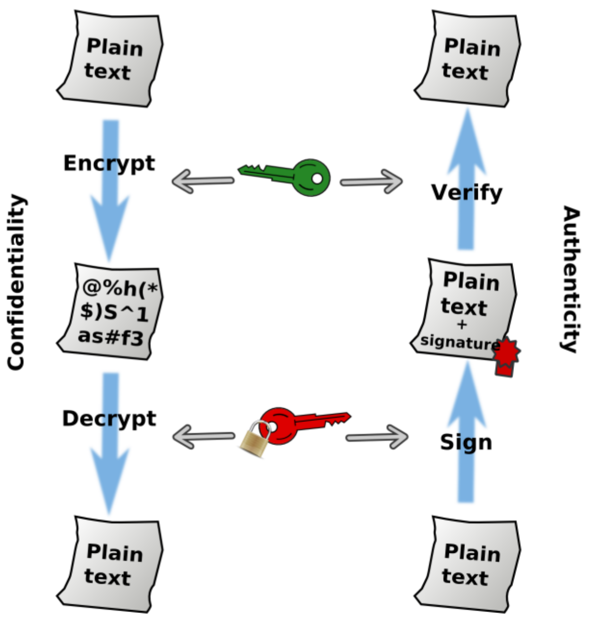
2.1.2 Public Key Cryptography

Instead of the symmetric-key cryptography, where a single key is used both for encryption and decryption, the public key (a.k.a. asymmetric) cryptography uses a pair of keys: a public key and a private key (Figure 4). The keys are mathematically related in a way that the private key cannot be practically derived from the public key. A message encrypted with one key can be decrypted only with the second - corresponding one.

Figure 4. Public and private key. The private key should be kept secret and be only known to the owner, whereas the public key may be widely distributed.

The two main applications of the Public Key Cryptography are to ensure confidentiality and authenticity. The applications are presented in Figure 5.

Figure 5. Confidentiality and Authenticity.
<u>Confidentiality</u> : a message encrypted with recipient's public key can only be decrypted by the recipient with his corresponding private key
<u>Authenticity</u> : a message signed with sender's private key can be verified using the wide-spread public key in order to prove the identity of the sender and the fact that the message hasn't been tampered with.



Because of the fact that public key cryptography is much more computationally intensive than symmetric one, their usage is commonly mixed. For example some protocols, like TLS, use the asymmetric keys during the handshake to authenticate the peers and establish a symmetric key for faster encryption of further communication.

The first and still most popular algorithm for public key cryptography is RSA [6], created in 1977 by Ron Rivest, Adi Shamir and Leonard Adleman at the Massachusetts Institute of Technology. The name of the algorithm is taken from the initials of authors' surnames.

But how do we know that the owner of the public key we possess is really the one she claims to be? In order to relate the public key encryption to the real world of people with names, addresses and organizations they belong to, the next mechanism: Public Key Infrastructure (PKI) was introduced.

2.1.3 Public Key Infrastructure

Public Key Infrastructure is based on certificates and Certificate Authorities. Certificates are electronic documents that associate public key with the owner's real identity. In order to assure this binding's integrity, it has to be digitally signed using some signer's private key. The certificate may be self-signed (using the owner's key) or signed by (issued by) some third-party. The signature unambiguously qualifies the signer. In order to trust the certificate, we have to trust the issuer of the certificate.

The trusted third parties, used for signing the certificates, are called Certificate Authorities. They are used to both sign and confirm the correctness and reliability of the certificates. The most publicly known CA is VeriSign[8]

Contents of the certificate:

The contents of the certificate typically includes:

- The public key being signed
- An associated name, which can refer to a person, a computer or an organization
- A validity period, after which the certificate won't be accepted
- A location of a revocation center
- A digital signature of the certificate, produced by the signer's private key

[8] http://www.verisign.com/

Certificate generation:

The process of generating a certificate contains the following steps:

1. Key pair is generated.
2. The Certificate Signing Requests (CSR) is created; this step requires user to enter some information that will become parts of the subject name, which are described below.
3. The request is sent to the signer (e.g. the Certificate Authority) to be signed.
4. Once signed, a certificate is returned.

It is important to notice that the private key is never sent anywhere.

Names in certificates[9]:

The subjects are identified with the Distinguished Names (DNs) that are commonly used (e.g. in LDAP directories) to uniquely identify users, systems and organizations. The format of DN, specified in one of X.500 series standard, has a hierarchical form that begins with a subject specific common name and proceeds with increasingly broader areas of identification until the country name is specified. Specific components are called Relative Distinguished Names (RDNs)

The typical RDNs are:

- CN – common name – that can be user's real name or hostname of the server
- OU – organizational unit
- O – organization
- L – locality / city
- ST – state / province
- C – country code

The example certificate DN used by the thesis' author is:

CN=Michal Dyrda, O=AGH, O=GRID, C=PL

In case of e.g. GRID, in order to enable secure communication all the entities, like users, hosts, servers, need to possess their certificates signed by the CA.

[9] The description is based on [7]

Credentials

In this thesis the word *credential* will be used for the pair of certificate and the corresponding private key, furthermore *permanent credential* for a long-term user's credential issued by the well known CA.

Certificate chains

The PKI architecture has a hierarchical structure of a tree. An example chain is presented in Figure 6.

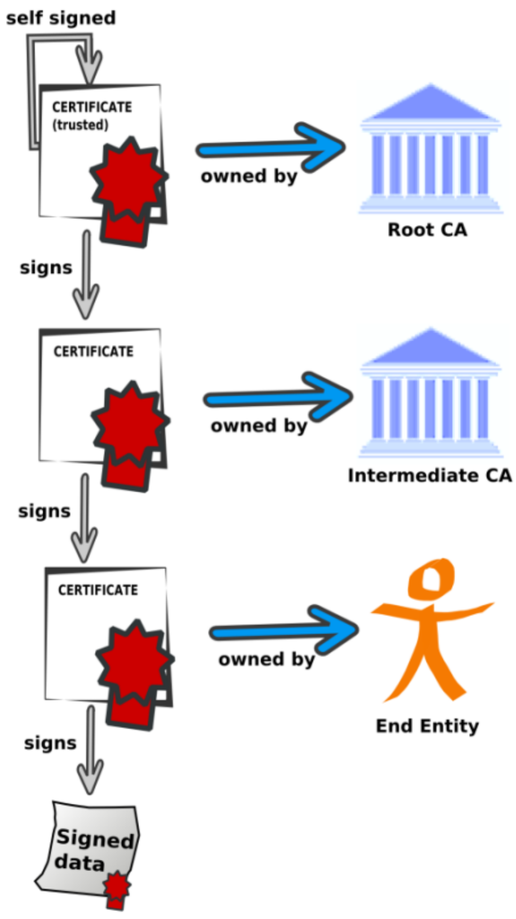


Figure 6. Certificate chain [10].
At the top there is a 'Root CA' with a self-signed root certificate. Other CAs can exist, whose certificates can be signed by the higher-level CA in the hierarchy. The leaves of the tree are end-entities certificates.

The set of certificates from an end-entity certificate up to a trusted Root CA certificate is called a *certificate chain*. All the descendants inherit the trustworthiness of their parents. That establishes a chain of trust. Trusting one of the CAs, we can trust any of the descendants. The verification of user's certificate requires verification of all the steps in the chain. The process is described in Appendix A.

[10] Based on figure from [8]

Revocation and CRL

Certificate may become invalid before the end of its validity time, e.g. owner's data may change or the private key may be compromised. In such cases, the CA revokes the certificate and puts it on a publicly accessible Certificate Revocation List (CRL), which should be updated regularly by the relying parties and checked during certificate validation. Avoiding certificate revocation verification may lead to serious security gaps.

Extensions

The recent, third version of certificates, available since 1996 supports certificate extensions, which may be defined and included in certificates. Some common extensions are defined by the standard, i.a.:

- Key Usage – limits the usage of the key to some particular applications, e.g. *signing only*
- Basic Constraints – identifies, whether the subject of the certificate is able to sign certificates (is CA) and may limit the number of certificates in the path following this CA certificate
- Alternative Names – allows to associate more identities with one public key, e.g. DNS names, email address or IP number

Each extension in a certificate can be marked as critical or noncritical. All the critical extensions must be recognized and processed by the certificate-using system, otherwise the certificate has to be rejected.

The knowledge of the extensions will be useful to understand problems that occurred during the development of the thesis target, which will be described in section 5.9.

2.2 Architecture provided by H2O

The existing version of H2O already offers some solutions to the security issues presented in previous chapter. They are based both on proprietary solutions (the RMIX framework with connection sessions, authentication mechanism) and public standards (i.a. Java Authentication and Authorization Service), which we can get familiar with now.

2.2.1 Communication mechanisms

The H2O communication interoperability is managed by RMIX[11], "*a communication framework for Java, based on the Remote Method Invocation (RMI) paradigm*" [9]. It combines simplicity, flexibility and performance. Depending on application requirements, it enables communication over several remote method invocation protocols, such as SOAP, RPC and JRMP. Extensible, pluggable architecture provides possibility to add new protocols, even on runtime. Some interesting features include possibility to use many protocols within a single application, dynamic protocols negotiation upon connection and asynchronous calls as well as one-way invocations.

RMIX provides communication security as well. All of the currently supported RMIX protocols can be used together with SSL/TLS layer (e.g. SOAP/HTTPS or RPC/SSL). **What is most important from the point of view of this thesis – the same H2O authentication algorithms can be applied irrespective of the communication protocol used.**

Figure 7 presents the overview of RMIX framework in H2O.

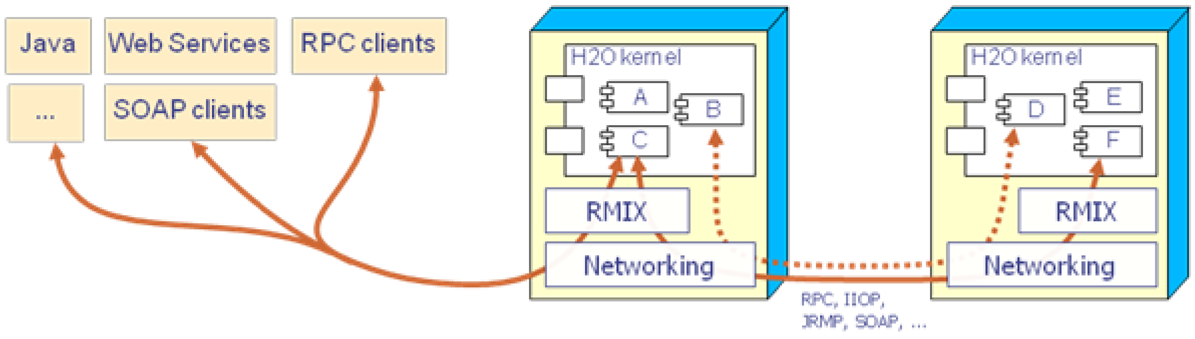


Figure 7. RMIX communication framework in H2O [12]
Depending on requirements, it enables communication between H2O kernels over several remote methods invocation protocols, such as SOAP, RPC and JRMP.

2.2.2 Connection sessions and transport layer parameters

H2O provides the concept of a session that allows user to use multiple connections and change their parameters in a flexible manner without the need of multiple authentication.

Upon connection a session certificate for client is created and based on the certificate, a connection session is being established. After user's authentication, user ID and roles are saved in session context on the server side. From that moment every

[11] http://dcl.mathcs.emory.edu/rmix
[12] The figure comes from [10]

client's connection to the server (which may use different sockets) may be established without the need of next authentication: client's certificate is used to identify the session which, in turn, provides client's roles required for specifying its permissions.

The initial connection is made using either plain (http) or TLS-secured (https) endpoint. After the session is being established, because of using the RMIX framework, client can modify the properties of both message (JRPM, SOAP, RPC…) and transport (TCP, TLS/TCP, …) layers used for subsequent communication. The connection features may be configured in order to trade-off between the performance and the security level of connection, depending on the requirements.

One of the examples is to disable the encryption of the communication channel, while retaining other features, such as integrity check. That allows to obtain better communication performance by getting rid of the encryption overhead while still preserving protection against tampering with the communication channel or man-in-the-middle attacks.

Secure connection parameters may only be achieved while using TLS endpoint for the initial connection. Using plain connection for the session establishment may allow to intercept the communication and impersonate the user, therefore assuring subsequent security is useless. Still the plain connection may be used for example in secured internal LANs.

2.2.3 Tunneled authentication

Creation of session keys for communication is usually performed by the authentication mechanism itself (e.g. in case of TLS). H2O mechanism is an example of tunneled authentication, where an 'outer' secure channel is used for creating a session over the message exchange of an 'inner' authentication algorithm.

The outer protocol is mostly responsible for the generation of a session key. However it may also enable encryption and integrity during the authentication, in order to prevent the authentication messages from being modified or understood. In that way even simple legacy authentication methods can be used securely.

2.2.4 Client authentication

Authentication is checking the identity, the client pretends to have. It consists both of getting the identity and verifying its correctness. The identity, together with the groups (roles), which are assigned to the user, are in H2O the basis of subsequent authorization.

To allow Single Sign-On, H2O authentication mechanism allows delegation of users' credentials together with login information (e.g. user id and groups).

Chain of authenticators structure

H2O kernel contains an extensible pluggable authentication architecture with support for multiple protocols, developed as separate modules that simplifies adding new authentication schemes. Similar solution is used i.a. in Pluggable Authentication Modules (PAM) [13] as well as for authorization in Globus Toolkit[14] [11]

The authentication engine evaluates a chain of configured authenticators that implement some authentication schemes or can call out to some external authentication services. That allows H2O kernel owner to provide multiple authentication methods, thus enabling users from different environments (e.g. organizations), with different credentials to use the resources provided by H2O.

Each authenticator has to be registered in H2O kernel (described in point 5.7)

Chain evaluation

The evaluation proceeds as shown in Figure 8:

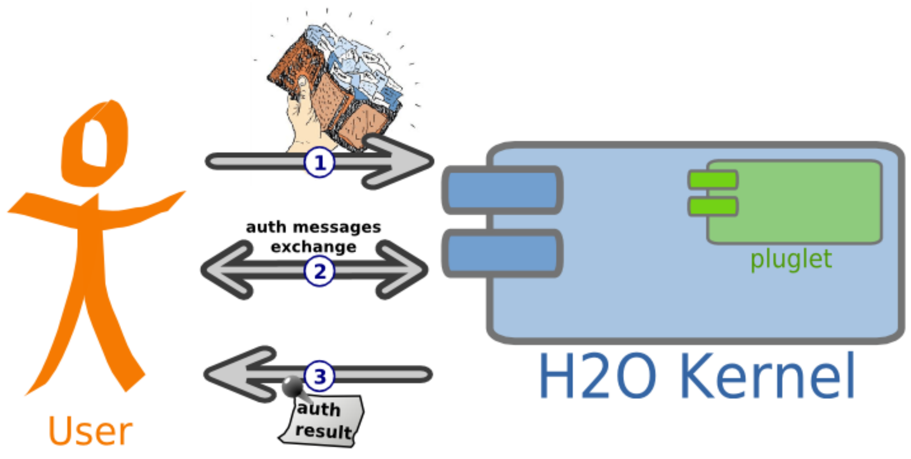


Figure 8. Authentication process in H2O – high-level diagram

1. **Submitting credentials in virtual Wallet**
2. **Proceeding the authentication messages exchange**
3. **Getting the authentication result**

- A user provides his credentials putting them into the virtual Wallet:

```
    clientCxt = H2OClient.newInstance(wallet, H2O.TRUST_ALWAYS);
```

[13] Described in RFC 86.0 published in October 1995
[14] www.globus.org/toolkit/

    - User doesn't have to supply credentials for all of the authenticators…
    - …but all of the supplied credentials have to be served by the engine (otherwise – authentication fails)
    - Each credential can be marked as 'required' by the user
- Upon connection, the wallet is submitted to the Kernel. [1]
- Authentication engine tries to map each credential to one of its authenticators
    - It finds the first authenticator that
        - supports the requested protocol
        - does not immediately fail after looking at the request – then the authenticator assumes that it is able to handle the credential
- The authentication messages exchange proceeds [2]
- The authentication result is returned [3]
    - If authentication of required credential fails, the chain evaluation breaks and access is denied
    - If authentication of non-required credential fails, the evaluation proceeds
    - If the end of the chain is reached without a 'required' authenticator returning false, the access is granted (as a 'guest' role)

Authenticators on the example of password authenticator

The H2O platform offers a simple password-based authenticator by default and allows custom authenticators to be written and added to the chain. This part presents details of the auth messages exchange from Figure 8 on the example of the password authenticator.

There exist two versions of the authenticator. In the first one only one message is sent from user to server, which contains both his public credential (id, realm) and the password. In the second one that is going to be described here, the password is sent as a second message.

The password authenticator specifies the following policies for credential delegation:

- DELEGATE_ALWAYS
  The delegation is performed even if the user is unknown (but not if the password is incorrect)

- DELEGATE_IF_AUTHENTICATED
  The delegation is done only if the user is known and correctly authenticated
- DELEGATE_NOT
  The delegation is never done

Figure 9 presents the sequence diagram of the second version of password authenticator. It assumes that the authentication successes and that the credential is to be delegated.

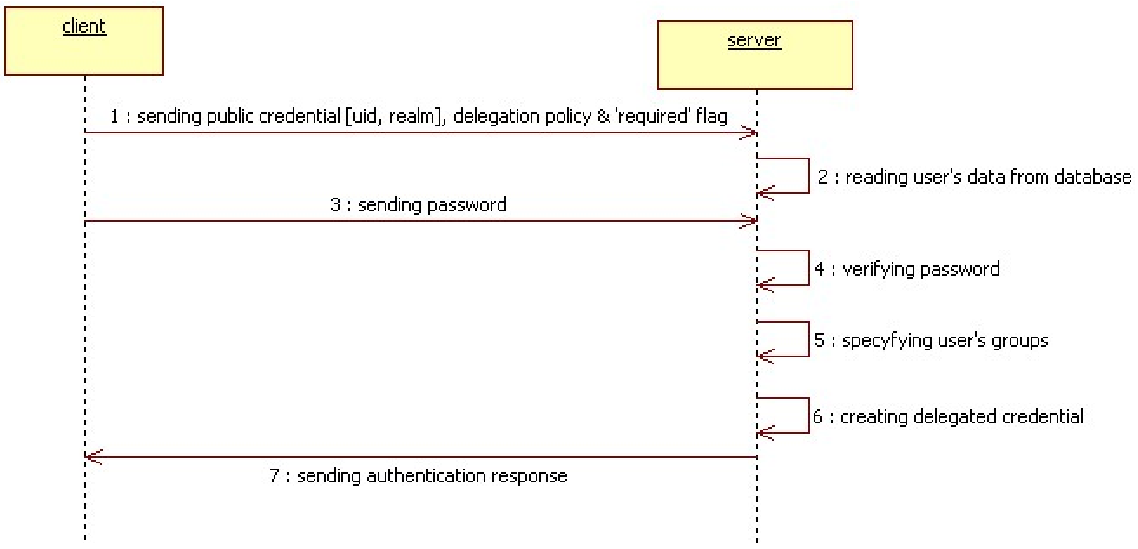

client
server
1 : sending public credential [uid, realm], delegation policy & 'required' flag
2 : reading user's data from database
3 : sending password
4 : verifying password
5 : specyfying user's groups
6 : creating delegated credential
7 : sending authentication response

Figure 9. Sequence diagram of password authenticator (v.2)
The following steps of the diagram are described in the use case in text below.

The following use case scenario describes the steps of the diagram:

Goal: user wants to authenticate himself to the kernel in order to perform some operations on it
Actors:
  client – user application connecting to the kernel
  server – the authenticator implementation on the server side
Preconditions: the credential from the Wallet is mapped to the proper authenticator
Triggers: authenticator engine starts the authentication process
Success guarantee : server responses with a success message and saves the client's principals (id, group membership) in login context

Basic flow:

1. Client sends the public credential (id and realm), the selected delegation policy and the 'required' flag
2. Server gets the credential and reads user from the database
3. Client sends password to the server. User may be prompted to type in the password at this time. It depends on client side implementation - current one uses a credential with prefilled password.
4. Server verifies the password
5. Server reads user's group membership information and saves it in the login context.
6. Server creates the delegated credentials and saves them in user's context
7. Server sends the authentication success response to the client

Alternative scenarios:

2a : The user was not found in the database
 2a1 : No groups are saved to user's login context
 2a2 : If the delegation flag is DELEGATE_ALWAYS, the delegation is performed
 2a3 : Server sends the authentication failure response to the client
 2a4 : The scenario is finished

4a : The password is incorrect
 4a1 : No groups are saved to user's login context
 4a2 : Server sends the authentication failure response to the client
 4a3 : The scenario is finished

6a : The delegation flag was DELEGATE_NOT
 6a1 : The delegation is not performed
 6a2 : The scenario goes to step 7

Users database

The password authenticator uses the file *Users.xml* placed in the security configuration directory (`{h2o-dist}/config/security`) as a database. It contains the entries of users both and their passwords (may be its digest) grouped in the four roles that will be described by presenting a *Policy*.xml file in section 2.2.6.

2.2.5 Server authentication

Server-side authentication can be a crucial thing in order to prevent some attacks on the system, as it is going to be presented in the threat analysis in point 6.5. Now the H2O mechanisms for enabling this feature will be described.

Upon connection, along with credentials wallet, client specifies so called trust manager to disallow access to distrusted kernels:

```
    clientCxt = H2OClient.newInstance(wallet, H2O.TRUST_CERTIFIED);
```

In order to require server authentication, instead of using the **H2O.TRUST_ALWAYS** trust manager, which allows connection to any kernel regardless of its certificate, client has to select the **H2O.TRUST_CERTIFIED** manager. It allows connection only to kernels, which identify themselves with valid certificates issued by CA that is included either in the default JSSE trust anchors, or the H2O trust anchors specified in `${h2o-dist}/config/security/cacerts`. Additionally, TrustCertified manager allows only secure connections and verifies the hostname of the kernel with the Common Name of kernel's certificate.

When specifying **no (null) trust manager**, the application will use default JSSE trust manager if the connection is done using secure endpoint.

Using server authentication requires some changes in kernel configuration. By default, the H2O kernel generates a self-signed certificate to identify itself to clients. It is difficult to manage those certificates and add them to clients' truststores. Instead of that, kernel can be forced to use other certificate, issued by a well-known CA. The configuration is presented in the appendix C.

2.2.6 Authorization

Authorization specifies users' permissions to connect to the kernel as well as to carry out given operations on it (deploy pluglets, activate sessions).

The H2O kernel security model is based on a customizable sandbox. Each pluglet uses separate classloader, so that they can run safely without the risk of affecting other pluglets or kernel itself. The authorization model of H2O is based on fine-grained permissions and handled by java JAAS. The default set of permissions is bounded to minimum (i.a. no file system access) and the permissions for specific operations have to be explicitly granted. Privileges are granted to groups (roles), which are assigned to each user.

JAAS

Java Authentication and Authorization Service [12] is a framework that extends the code-centric architecture, introduced in the Java 2 platform, with the new user-centric access control. In the hitherto architecture, the permissions are granted based on code characteristics: where it is coming from and whether it is digitally signed – and if so, by whom. The structure of the entries looks like:

```
       grant [signedBy <signer> [,codeBase <code source>] {
           permission <class> [<name> [, <action list>]];
       };
```

An example of the configuration is:

```
       grant codebase "file:./JaasAcn.jar" {
          permission javax.security.auth.AuthPermission
                            "createLoginContext.JaasSample";
       };
```

JAAS allows the permissions to be granted based not just on what code is running, but also who is running it. As a result of authentication, the **Subject** object is created. It represents the authenticated user, containing his credentials as well as a set of **Principals**, representing the identities of that user. Those principals may be assigned specific permissions in the policy. The entries are supplemented with new fields - the fully qualified name of a principal class and a principal name:

```
       grant [signedBy <signer> [,codeBase <code source>]
       [,principal <principal class> <principal name>] {
           permission <class> [<name> [, <action list>]];
       };
```

This time the example may look like:

```
       grant codebase "file:./SampleAction.jar",
               Principal sample.principal.SamplePrincipal "testUser" {

          permission java.util.PropertyPermission "java.home", "read";
       };
```

In order to take advantage of the permissions, the Subject, which is the result of authentication, must be associated with current access control context. For each subsequent security-sensitive operation, Java runtime will automatically determine, whether the required permission is granted to the specific principal and if so, the operation will be allowed only if the currently active subject contains it.

JAAS in H2O

The JAAS authorization component used in H2O requires the kernel to authenticate the user first. Instead of using JAAS authentication mechanisms, H2O expects its own authenticators to populate the Principals sets, which are then used by the authentication engine to create a Subject object and associate it with the context.

Policy.xml

The H2O authentication policy is by default specified in the *Policy.xml* file in `{h2o-dist}/config/security`. Since the authentication is both user-centric and code-centric, the file is divided into:

- Principal-based permissions – the permissions granted to the specific roles. There are four roles of increasing importance in the presented order; each role inherits the permissions of its antecessor and adds its own ones. The default permissions are presented here.

  Unauthorized user is assigned the **Guests** role whose permissions can be broaden for testing purposes but should be limited in production environment.

  Assigning other roles requires user authentication:
  - permissions given to **Users**
    - allow users to login (activate the session)
    - allow users to access and bind pluglets
  - permissions given to **Deployers**
    - allow deployers to deploy pluglets
    - the pluglets are allowed to accept connections
  - permissions given to **Administrators**
    - all permissions for kernel, pluglets and session

- Code-based permissions – special permissions (i.a. accessing the file system) can be granted to some code that is placed in specific location or that has been digitally signed. The default sections of standard H2O policy file are the following two:
  - H2O distribution pluglet permissions
  - permissions granted to pluglets signed by the Emory DCL[15]

[15] Distributed Computing Laboratory, Dept. of Math and Computer Science, Emory University

2.3 Missing features

Analysis of the system, its target usage and users' opinions as well as the observation of trends in the domain of security led to specification of missing features that need to be added to H2O in order to increase its usability and usage safety. This section is going to present and describe the features.

- **Credential security**

The provided password authenticator does not assure sufficient level of security. It is very unsecure without the tunneling. Moreover, it cannot be used for single sign-on and delegation (which feature is described below). And it is hardly possible to manage the credentials' validity lifetime.

The another common weak point of this solution are users as well, which are known to be careless with the passphrases or use very 'easy' secrets that are simple to break.

- **(Single sign-on and) delegation with short-term permissions**

The password authenticator enables credential delegation for single sign-on, though it requires user to pass over his password to the services, which she can't control – at the same time losing the control over the password.

- **Compatibility with well-known standards**

The current 'de facto' standard for security in grid computing is the GSI, used by Globus Toolkit as well as in some scientific projects, like EGEE[16]. It uses well-known technologies with readily available, well-tested open source implementations, which are standards in their fields as well. The flexibility of trust model for X.509 certificates assures good scalability and perspectives for broader field of application.

This solution may not only challenge all the other features presented above, but may also be a cornerstone for further enhancements, like centralized management of authentication and authorization policies or enabling the cooperation of multiple Virtual Organizations.

[16] http://public.eu-egee.org/

2.4 Summary

This chapter recalled us some key security concepts and gave us an overview of the current state of security architecture provided by H2O. First the Transport Layer Security both with Public Key Cryptography and Public Key Infrastructure were reminded. Furthermore the issues of H2O communication layer security, client- and server-side authentication and JAAS, an authorization mechanism used in H2O, were presented. Together with missing features, which were identified, the presented background will be the starting point for a design of a solution that will extend the existing architecture to answer the presented shortcomings.

Chapter 3. Related work

Prior to the design of this thesis' subject, let us look at how the presented security issues were answered by hitherto researches of other people. The result of their works - GSI, providing proxy credentials and delegation, MyProxy for credential storage and Needham-Schroeder protocol for authentication - had an essential influence on the designed solution and therefore are going to be described now.

3.1 GSI [17]

The analysis of the grid security issues mentioned earlier was the reason to create an official specification for safe communication in a grid environment. The Grid Security Infrastructure (GSI), formerly called the Globus Security Infrastructure, consists of protocols, libraries and tools that allow users and applications to make the communication in grid computing environment in a secret, tamper-proof and delegateable way.

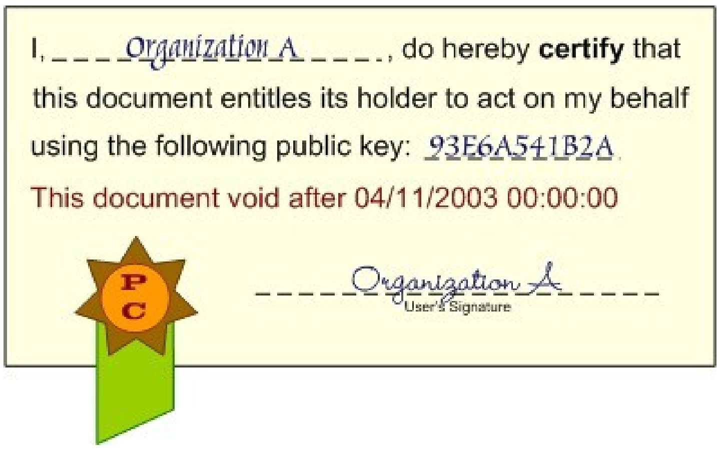
GSI is in principle based on existing mechanisms like public key encryption, X.509 certificates and TLS communication protocol with mutual authentication. Still it provides some extended functionality, like single sign-on and delegation, together with mechanisms that enable them.

3.1.1 Proxy certificate

The 'idea' is called proxy credential and is a short-term credential that is created in a base of user's permanent credential (private key with associated certificate obtained from the CA) and can be placed instead of it to authenticate that user.

A proxy certificate is made on basis of a new key pair and it's digitally signed by the owner of the original certificate using her private key. It contains the owner's identity, slightly modified to indicate its being a proxy, and a lifetime usually limited to some days or hours only. In a real world it could look like in Figure 10.

[17] Description of GSI is based on [13] [14] [15] [16]


I, ____Organization A_____, do hereby **certify** that this document entitles its holder to act on my behalf using the following public key: 93E6A541B2A

This document void after 04/11/2003 00:00:00

_____Organization A_____
User's Signature

Figure 10. Proxy certificate [18]
A short-term credential created in base on user's permanent credential that can be used instead of it to authenticate the user.

The credential also has to be kept secret but the limited lifetime makes it enough to protect it only by file system permission, which gives the possibility to use them by the user without inconvenience.

Proxy certificates mechanism allows certificates to be created dynamically without the need of standard heavy-weight vetting process associated with obtaining it from a CA.

Some technical information about proxy files - file format, extensions and Globus' proxy types as well as proxy validation - are described in Appendix A.

3.1.2 Single sign-on

While using multiple grid services, mutual authentication is demanded for each connection. In practice this requires user to access her private key each time the authentication is needed. And since private keys are protected with passwords, user would may have to sign on (type in the password) multiple times in a short period of time.

Using proxy certificates, it is enough to sign on only once to create a proxy certificate. The certificate is then used for all subsequent authentications. In practice this means the Proxy Certificate private key is stored on a local file system and is protected by only local file system permissions, which allow user's applications to access it without any manual intervention by the user herself. Moreover, proxy creation is normally done by a single application run by the user.

[18] The figure comes from [15]

Figure 11 presents the process of creating a proxy for single sign-on.

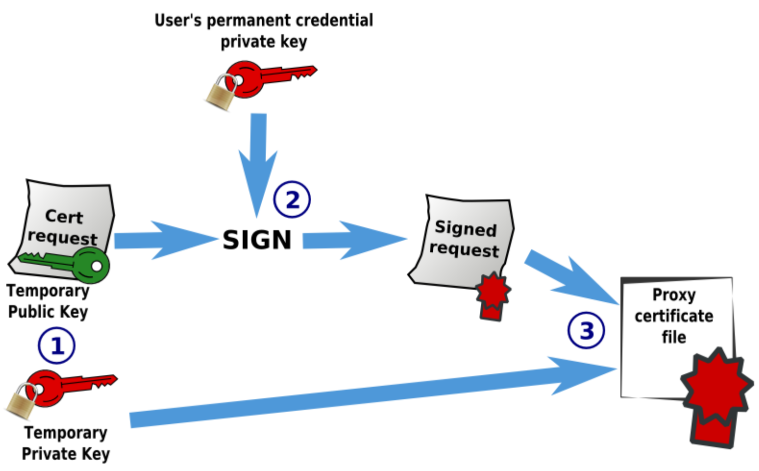


Figure 11. Creating proxy for Single Sign-On [19]

1. **New key pair for using with proxy certificate is generated on user's storage space. The certificate request is created.**
2. **The request is used to create the proxy certificate using user's permanent credentials. That will usually require the user to enter a pass phrase for accessing the credential. After signing the proxy certificate, the permanent credentials don't have to be accessed until the proxy expires.**
3. **The proxy certificate and its associated private key are placed in a file. The file is protected only by local file system permissions to allow for easy access by the user or software.**

3.1.3 Delegation over network

Other problem is using some grid services as agents that act on user's behalf. In order to allow it, they need to have access to user's credentials to use them for authentication to the services, they want to connect. The standard GSI software expects the user's private key to be stored locally on the machine, encrypted by a password to prevent other users from stealing it. The brute-force approach would be to send each one of the services our key pair over the network and type in the pass phrase each time they want to use it. The former is very dangerous, the letter – very inconvenient. And what about invoking 100 jobs on 1000 computers?!

This issue can be solved by using proxy certificates delegated over the network. The process is very similar to the process of creating proxy for single sign-on. It does not include exchange of any secret information. It only requires the connection to be

[19] Based on figure from [16]

tamper-proof to prevent the messages from modifications, no encryption is required though. That can be accomplished by using the TLS protocol.

After delegation, the access to user's private credentials is not needed. On the other hand the risk of losing control of the created proxy is avoided by its limited lifetime.

The process of proxy delegation (after establishing the integrity-protected channel) can be described using the chart from Figure 12.

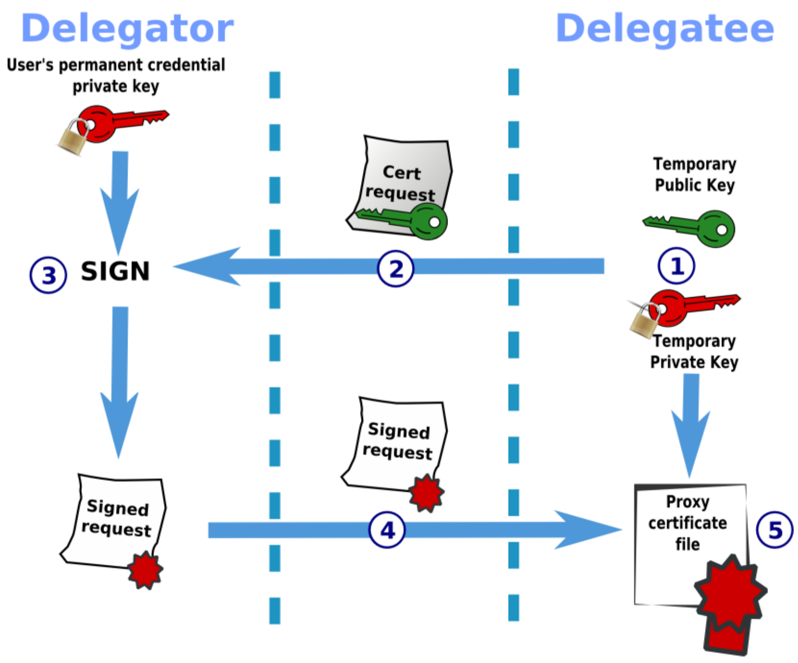


Figure 12. Delegation over network [20]

1. **The delegatee (let's say a grid service) generates a new key pair on its storage space**
2. **It creates the certificate request and sends it to the delegator (owner of the permanent credential)**
3. **The delegator signs the request...**
4. **... and sends it back to the delegatee.**
5. **The signed certificate and its associated private key are placed in a proxy certificate file. The new proxy certificate is used by the delegatee to authenticate itself on other grid services.**

The process of delegation can be chained. The new proxy can be used by the delegatee to create another proxy for a third peer and so on. What we get is a chain of certificates – similar to described in 2.1.3. Each proxy certificate contains all the certificates of its ancestors including CA's certificate. In this case the mutual

[20] Based on figure from [17]

authentication process runs slightly different. To verify the proxy at the end of the chain the previous proxy's public key (got from its certificate) is used to check the signature. Then the verification of the previous proxy goes analogously and so on. To check the first generated proxy, the user's public key is used. At the end the public key of CA is used to validate the signature on the user's certificate. This establishes a chain of trust.

3.1.4 Proxy Certificate Format

The format of Proxy Certificate is compliant with the X.509 public key certificate standard, thus allows it to be used in already existing protocols and libraries, making the implementation significantly easier.

In order to achieve the uniqueness of the proxy certificate, the subject name of the proxy and the value of the proxy serial number should be unique (at least statistically) to the issuer. The former one is accomplished by appending an additional CommonName component, to the subject's DN. The example DN of author's proxy certificate is:

CN=40253375, CN=Michal Dyrda, O=AGH, O=GRID, C=PL

Both RDN and serial number uniqueness may be achieved by using the hash of the public key as the value. The motivation for use the unique subject names and serial numbers is to enable the proxy certificates to be used together with attribute certificates, which are used by some mechanisms for authorization purposes that are not going to be described here.

3.2 MyProxy [21]

GSI allows us to use confidential, integral communication that is premised by a mutual authentication using PKI. Moreover it provides proxy certificates, which enable single sign-on and delegation. The disadvantage is that we get stuck with our permanent credentials and their location, which makes using the Grid from different terminals complicated and unsecure. As a solution the MyProxy was introduced. In this chapter it is going to be described and some usage examples will be provided. At the end, one more solution will be presented that is MyProxy CA, which allows us to completely get rid of users' permanent credentials in our PKI.

[21] Description of MyProxy is based on [14] [18] [19]

3.2.1 Overview

MyProxy is an open source software for managing security credentials. Its abilities will be best described using an example from the Figure 13.

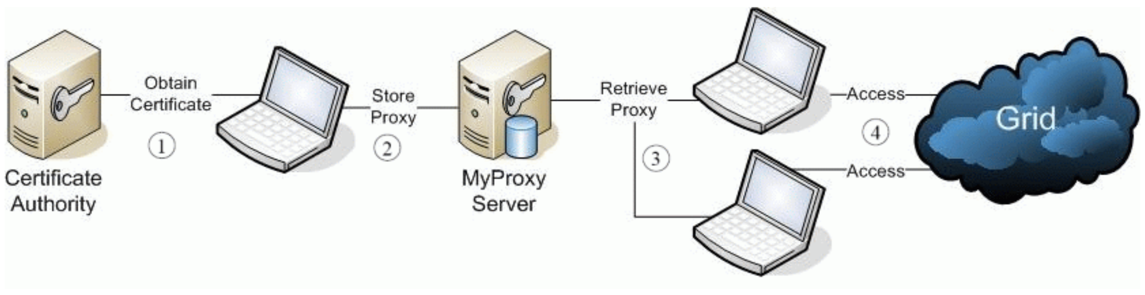


Figure 13. MyProxy overview [22]

1. **Instead of storing the credentials on each machine that we use to use to access the grid, we can take the certificate obtained from the CA...**
2. **... and delegate it to MyProxy repository in form of a long-term proxy, valid by default 7 days**
3. **Than we can use the proxy to create a short-time proxy credential (by the chain rule) upon request – by using any MyProxy client software.**
4. **Any portal / service / host can get our short-time proxy credential and work on our behalf just by providing them with the name and password, with which the credentials are protected on MyProxy server.**

MyProxy is developed at the National Center for Supercomputing Applications with contribution from the worldwide Globus community. Since 2000 it was an independent Globus Toolkit add-on and was integrated as an internal component in version 4.0 of the GT.

3.2.2 Usage Scenarios

Let us consider a detailed view of using MyProxy to manage our credentials by analyzing GRID usage with a grid portal. A grid portal is a simple graphical interface to grid applications. It is a web server that gives a possibility to use grid services from any standard web browser. After logging into the portal one can use a wide range of grid resources, submit jobs, transfer files or query information services. And the user doesn't want to type in a pass phrase any time she invokes a command.

[22] The figure and following ones concerning MyProxy come from [19]

Standard Web Security protocols don't fulfill the requirements of the grid portals. On the other side, the portals architecture doesn't work with existing grid security solutions, like GSI, which would allow us to use delegation and single sign-on.

Fortunately MyProxy can be used as a bridge between grid portals and GSI to provide secure interaction with grid resources. Let us analyze the Figure 14:

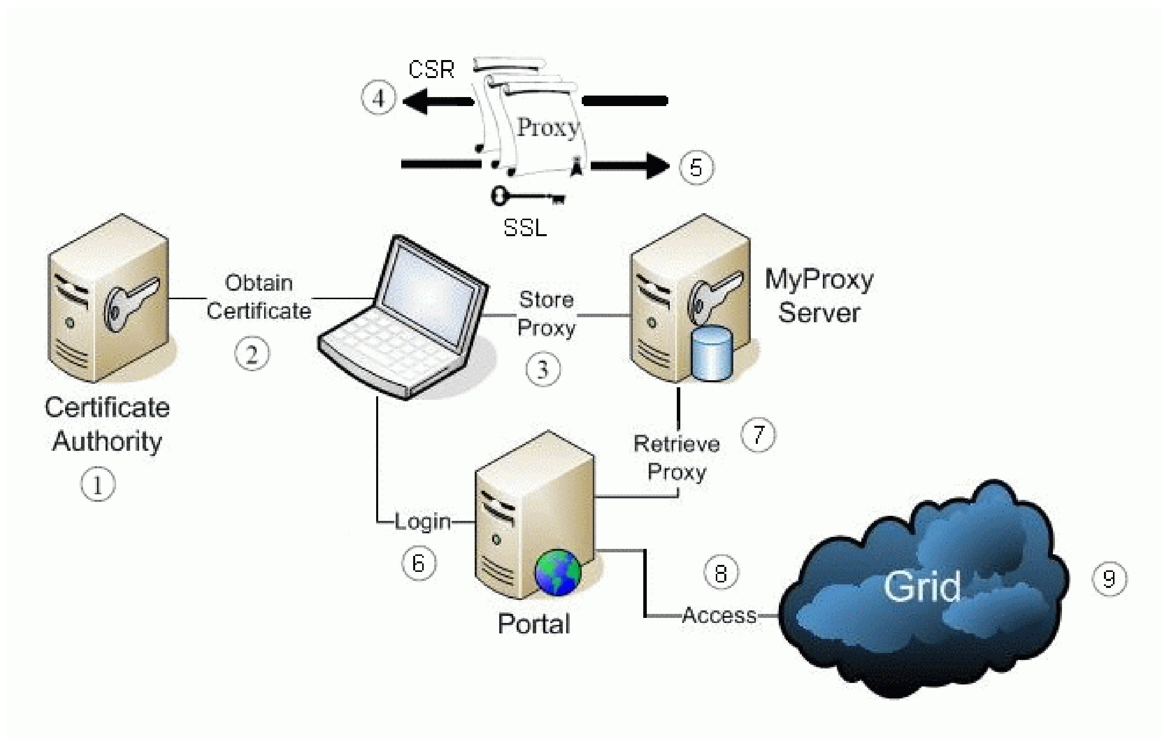


Figure 14. MyProxy usage example
The following steps of the example are described in text below.

I. Prerequisites

First we have to possess our own main pair of private key and certificate, which will be called a permanent credential. The private key is stored safely on our personal computer.

1. A CA well-known to all the services in the network exists…

2. … which is going to be used to sign the certificate request that is based on our public key.

II. Permanent credential delegation

3. Before we start using the GRID (e.g. GRID portal) we have to delegate the permanent credential stored on our PC to the MyProxy server using my_proxy_init command.

To do that we safely connect to the server using TLS and login using any authentication mechanism acceptable by the server (pass-phrase, certificate, Kerberos, Pubcookie etc.). The MyProxy server generates a long-term proxy certificate. What is exactly done is:

4. the server creates its own asymmetric key pair and certificate and responses with a Certificate Signing Request.

5. the certificate is signed by our private key (that's the only place we access it and give the password that protects it) with a default lifetime of 7 days and sent back to MyProxy server where it is safely saved together with the generated key pair on the server space using Unix username and the password given by the user.

III. Short-time credential generation

6. When we want to use the grid services, maybe at some later point of time, we login to the grid portal using the username and password which were used to store the proxy credential on the MyProxy server. We can connect from any place and using any terminal supplied with web browser, because we don't need the access to our permanent credential at this time.

7. To enable the portal to work on our behalf it has to create next, short-time proxy credential using the one stored on MyProxy server. Again key pair is created. At this time it is stored on the portal's storage space. The Certificate Signing Request along with the username and password given by portal login is send to the MyProxy server, which creates and signs the certificate using this private key and the certificate signed by ourselves that have been generated by initialization. The new certificate has a very short lifetime usually of some hours or maybe minutes only.

8. When the grid portal connects any service using the newly created short-time proxy, the certificate chain, which has arisen, is used to verify the correctness of the certificate. It also uses the Online Certificate Status Protocol (OCSP) to check, whether the certificate hasn't been revoked.

9. Successful verification can enable the service to work on our behalf. Any portal / service / host can get our proxy credential and work on our behalf just by providing them with the name and password, with which they are stored on MyProxy server.

The operation of logging out of the portal deletes the short-time credential on the portal. If we do not logout – the certificate short lifetime makes it expire soon.

3.2.3 My Proxy as a CA

MyProxy can also include the ability to act as a Certificate Authority. In that way users don't have to manage long-term proxies anymore. Upon the request of a short-time proxy credential (with myproxy-logon) the MyProxy CA issues a new original certificate and signs it using its own private key. To accept that, the grid services must be able to trust the MyProxy server's internal CA (either trusting the CA itself or any of the CAs higher in the hierarchy that have signed MyProxy CA certificate). An overview of example usage is presented in Figure 15

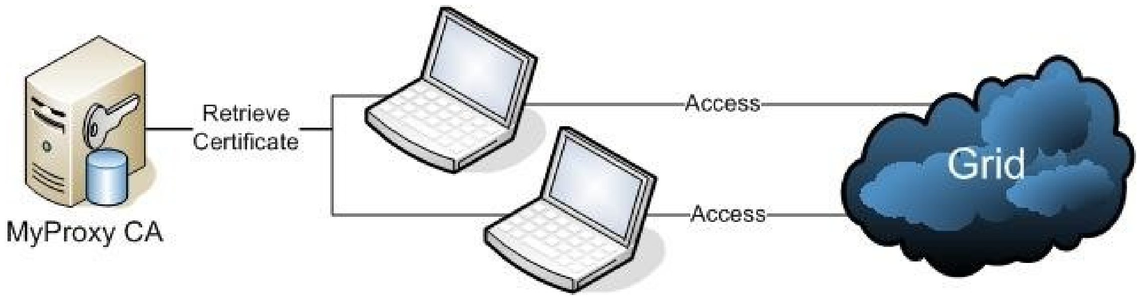


Figure 15. MyProxy CA usage
The proxy certificate retrieved from MyProxy CA are used by the users to access other grid resources

3.3 Needham-Schroeder protocol

This term refers to two protocols, which were proposed by Roger Needham and Michael Schroeder.

The first one, based on symmetric encryption algorithms, called **The Needham-Schroeder Symmetric Key Protocol**, was aimed to establish a session key between two parties. This protocol is not going to be considered in this thesis.

The second one, **The Needham-Schroeder Public-Key Protocol** [20], was intended to provide mutual authentication using asymmetric cryptography. It also establishes the session key. The protocol, its vulnerabilities and fixed versions are going to be analyzed here.

Assumptions

- Both Alice (A) and Bob (B) possesses valid certificates together with corresponding private keys.
  Alice's keys are named : K_{PA} and K_{SA} (respectively public and private-secret key)
  Bob's keys are named : K_{PB} and K_{SB}
- The certificates are signed by a trusted server (S), which is used to distribute public keys on request
  Server's keys are named: K_{PS} and K_{SS}

Basic protocol

The primary version of the protocol is presented below in a security protocol notation[23]. The brackets { … } indicate signing or encrypting their contents by the following private or public key respectively:

$A \rightarrow S: A, B$	A requests B's public keys from S
$S \rightarrow A: \{K_{PB}, B\}_{K_{SS}}$	S responds with B's identity placed alongside K_{PB} for confirmation.
$A \rightarrow B: \{N_A, A\}_{K_{PB}}$	A invents a random number, N_A, and sends it to B
$B \rightarrow S: B, A$	B requests A's public keys.
$S \rightarrow B: \{K_{PA}, A\}_{K_{SS}}$	Server responds.
$B \rightarrow A: \{N_A, N_B\}_{K_{PA}}$	B invents N_B, and sends it to A along with N_A to prove ability to decrypt with K_{SB}.
$A \rightarrow B: \{N_B\}_{K_{PB}}$	A confirms N_B to B, to prove ability to decrypt with K_{SA}

Figure 16 presents the message exchange between A and B (omitting the communication with S). All the keys used are public keys:

[23] en.wikipedia.org/wiki/Security_protocol_notation

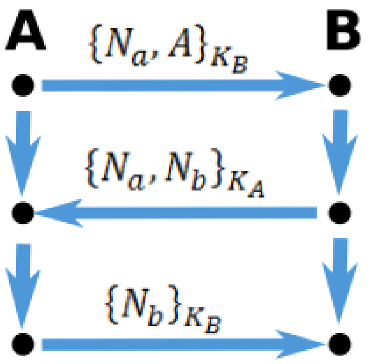


Figure 16. Needham-Schroeder protocol - message exchange
N_a and N_b – random numbers of Alice and Bob
A – identity of Alice
{ … } K_A – message encrypted with Alice's public key
{ … } K_B – message encrypted with Bob's public key

At the end both A and B know each other's identities and the N_A and N_B numbers. They are not known to eavesdroppers, though.

Attack

Unfortunately, this protocol was found to be vulnerable to a man-in-the-middle attack. Both the description and a fixed version of the scheme was described in 1995 by Gavin Lowe [21]. If an impostor M can impersonate B and persuade A to initiate a session with him, he can relay the messages to B and convince B that he is communicating with A. M keys are again K_{PM} and K_{SM}

Ignoring the traffic to and from S, which is unchanged, the attack runs as follows:

$A \rightarrow M: \{N_A, A\}_{K_{PM}}$	A sends N_A to M, who decrypts the message with K_{SM}
$M \rightarrow B: \{N_A, A\}_{K_{PB}}$	M relays the message to B, pretending that A is communicating
$B \rightarrow M: \{N_A, N_B\}_{K_{PA}}$	B sends N_B
$M \rightarrow A: \{N_A, N_B\}_{K_{PA}}$	M relays it to A
$A \rightarrow M: \{N_B\}_{K_{PM}}$	A decrypts N_B and confirms it to M, who learns it
$M \rightarrow B: \{N_B\}_{K_{PB}}$	M re-encrypts N_B, and convinces B that he's decrypted it

Again message exchange is presented in Figure 17

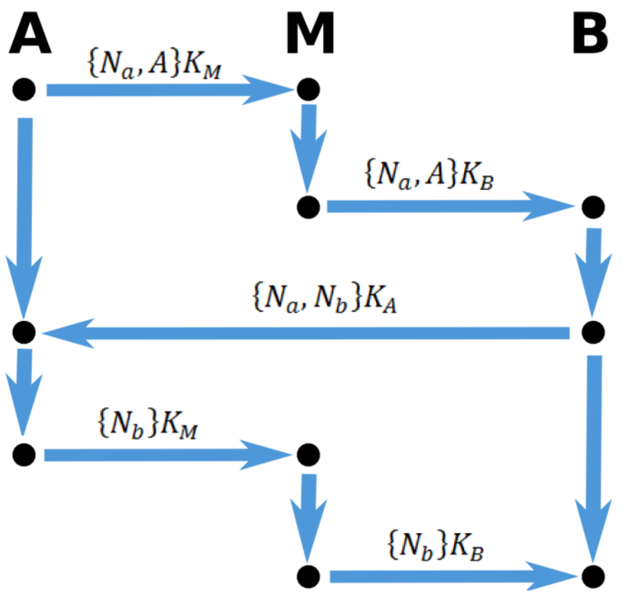


Figure 17. Attack on Needham-Schroeder protocol - message exchange
$\{ \dots \} K_M$ – message encrypted with public key of man-in-the-middle

At the end of the attack B falsely believes that A is communicating with him. Instead of that B is communicating with M, which also knows the N_A and N_B numbers needed for communication.

The fixed version

In order to fix the algorithm, the traffic to from B to A should be extended with the identity of B. We replace the step:

$$B \rightarrow A : \{N_A, N_B\}_{K_{PA}}$$

with

$$B \rightarrow A : \{N_A, N_B, B\}_{K_{PA}}$$

M doesn't know the value of N_B yet so it cannot exchange this message with one containing his own identity. After receiving the message, A can verify that she is really communicating with B or not.

3.4 Summary

This chapter presented some examples of related work. Others' experience, which in some cases became a 'de facto' standard, will be used in the subsequent part of the thesis during the development of the H2O security architecture.

Chapter 4. Concept and Design

This chapter focuses on the main target of the thesis – creation of an authenticator, which detailed requirements as well as additional demands towards its overall creation process will be presented first in the chapter. H2O connection use cases will allow us to state, in what extent its existing architecture can be used in the process of new authenticator creation – and which aspects are to be designed from the scratch. Furthermore the concept of the work will be presented. Detailed design will be supported with a data flow diagram as well as sequence diagram described as usage scenario.

4.1 Detailed requirements

The aim of the thesis will be to create the authenticator in a form of H2O Policy Decision Point that will target the presented challenges basing on standardized and verified technologies described above. This section will present the detailed requirements towards the authenticator.

- **Authentication based on PKI and X.509 certificates**

An authentication scheme based on Public Key Infrastructure should be implemented. Users identity should be confirmed using certificates issued by a trusted CA. Simple challenge-response algorithm should be used to verify that the user is really the owner of the certificate (possesses the corresponding private key). For the solution to be complete, support of certificate revocation should be added to the H2O.

- **Delegation based on proxy certificates**

The motivation for delegation using proxy certificates was already presented. This solution is secure and integrates very well with the suggested authentication algorithm and the Public Key Infrastructure. Moreover, it provides abilities to be extended for broader fields of application, i.a. for advanced authentication based on attributes. Some interesting solutions are going to be presented at the end of the thesis.

- **Compliance with GSI**

While using some well-known solutions, an implementation that is going to be compliant with existing and widely used systems should be used. Because of its popularity in other grid-related software, the compliance with GSI is expected. Some differences in used standards may cause some incompatibilities, which is going to be presented.

4.2 The name of the authenticator

The suggested solutions is named the GSI Authenticator to underline its compliance with the Globus mechanisms. Such name will be used in the following part of this thesis while relating to the authenticator.

4.3 Use Cases

On the basis of the H2O architecture and the features that are required from the GSI Authenticator, the expected use cases presented in Figure 18 can be identified:

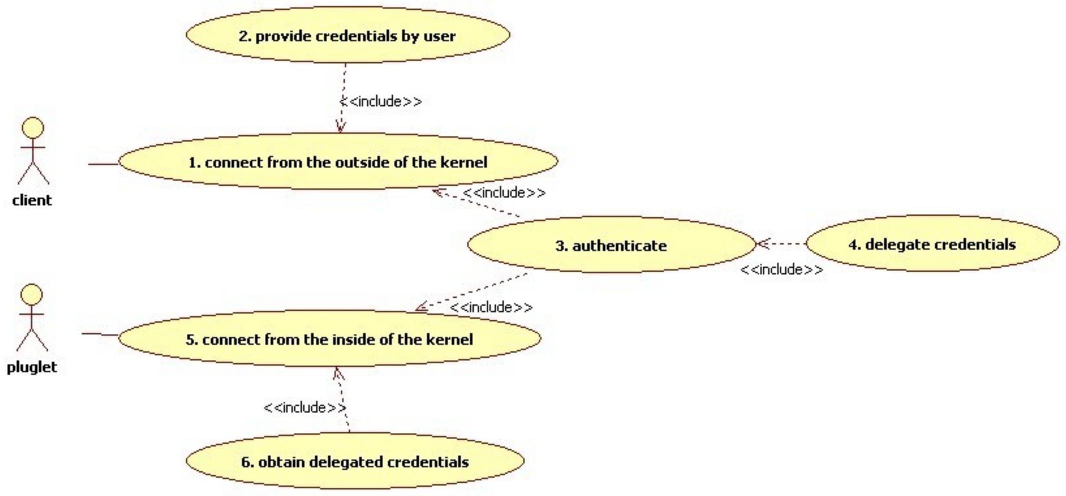

2. provide credentials by user
<<include>>
client
1. connect from the outside of the kernel
<<include>>
3. authenticate
4. delegate credentials
<<include>>
<<include>>
pluglet
5. connect from the inside of the kernel
<<include>>
6. obtain delegated credentials

Figure 18. H2O Use Cases
Client and pluglet connecting respectively from the outside and inside of the kernel together with required included operations

User connects to the kernel in order to deploy and/or utilize pluglets (1). For connection, user has to provide his credentials - certificate and corresponding private key or a proxy (2). The credentials are used for authentication (3). During the authentication process, credential delegation can be performed (4). The deployed pluglet can then act as an actor that wants to act on behalf of the client. Again, being itself in a kernel, it wants to connect to some other one (5). The delegated credentials should be provided by the kernel (6), then used for authentication (3) and maybe delegated (4).

The mechanisms for performing connection (1,5) and obtaining delegated credentials (6) are already provided by the H2O. Obtaining credentials from file system (2) will be achieved using some external libraries. The tool will be described in the implementation chapter (section 5.2.5). Now the concept of authentication and delegation (3,4) will be precisely described.

4.4 Concept of GSI Authenticator

Figure 19, which is an evolved diagram of authentication process from Figure 8, presents the high-level overview of GSI Authenticator usage.

User may use own permanent credentials to create a proxy for single-sign on. The first proxy or the credentials themselves are put into credentials wallet and used for authentication. During the authentication process delegation is performed and the delegated credential may be used by the pluglets to work on a user's behalf.

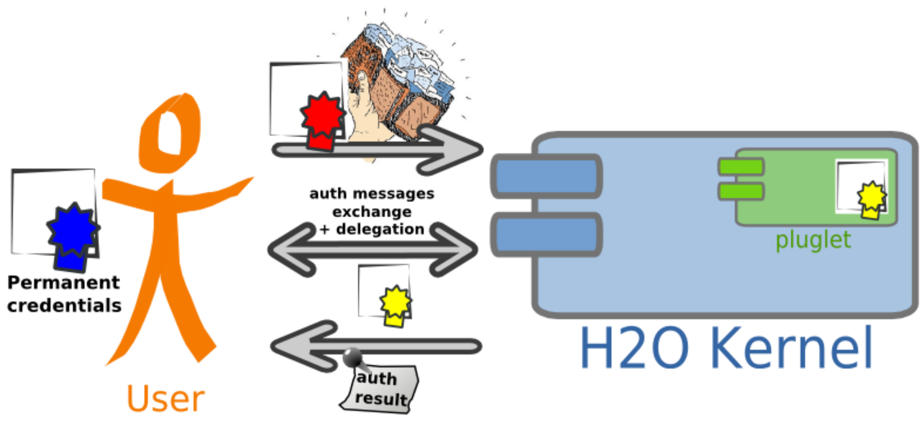


Figure 19. GSI Authenticator overview

1. Submitting GSI credentials in virtual Wallet
2. Proceeding the authentication messages exchange and credential delegation
3. Getting the authentication result

The delegated credential may be used by the pluglet for subsequent communications

Now the details of the authentication itself are going to be presented.

The concept of the GSI Authenticator is to use a simple challenge-response protocol with the requirement of tunneling the authentication process by a secure outer protocol. The single sign-on and delegation abilities will be provided using proxy certificates.

The successive steps are:

0. **Creating a proxy**
   This step is optional. In order to authenticate, a client has to present his certificate. Depending on the client's application implementation, permanent credentials as well as proxy may be used. The former requires the application to possess or ask user for the password while accessing the private key. The latter requires the valid proxy certificate to be available on the file system. The details of creating and accessing a proxy are going to be described in chapter 5.

1. **Establishment of a tunnel**
   A client establishes a secured connection with the kernel that will be used to prevent tampering and provide a session for securing subsequent communication. During the handshake, the authentication of kernel is performed by the client in order to prevent a man-in-the-middle attack.

2. **Identity introduction**
   A client introduces himself with the public certificate or a chain of certificates. Kernel verifies the validity of the certificate and checks, if the issuing CA is trusted. The existence of a user's entry in kernel's database is confirmed for the purpose of subsequent authorization.

3. **Identity confirmation**
   A simple challenge-response algorithm is used for identity confirmation. The kernel encrypts some randomly generated challenge number with a client's public key. A client uses his private key to decrypt it, then it signs it and sends back to the kernel as a response, to confirm the possession of the corresponding private key. Kernel verifies the response with the user's public key. If the response is verified, the identity of the client is trusted and saved in the session.

4. **Delegation**
   This step is optional as well. The delegation may be performed in order to allow deployed pluglets to work on the user's behalf. Kernel creates temporary

asymmetric keys for the proxy and a proxy request which is sent to the user. The user signs the request using his credentials (e.g. proxy for single sign-on) and sends it back to the kernel. The signed proxy together with the corresponding temporary private key are saved in the Kernel.

The step no.1 is achieved using secure communication layer with server authentication enabled. The client authentication and delegation process are going to be presented in subsequent sections.

4.5 Data Flow Diagram of H2O credentials

The diagram from Figure 20 gives some other view on the GSI Authenticator by illustrating the flow of the credentials and the way they are processed in the course of delegation.

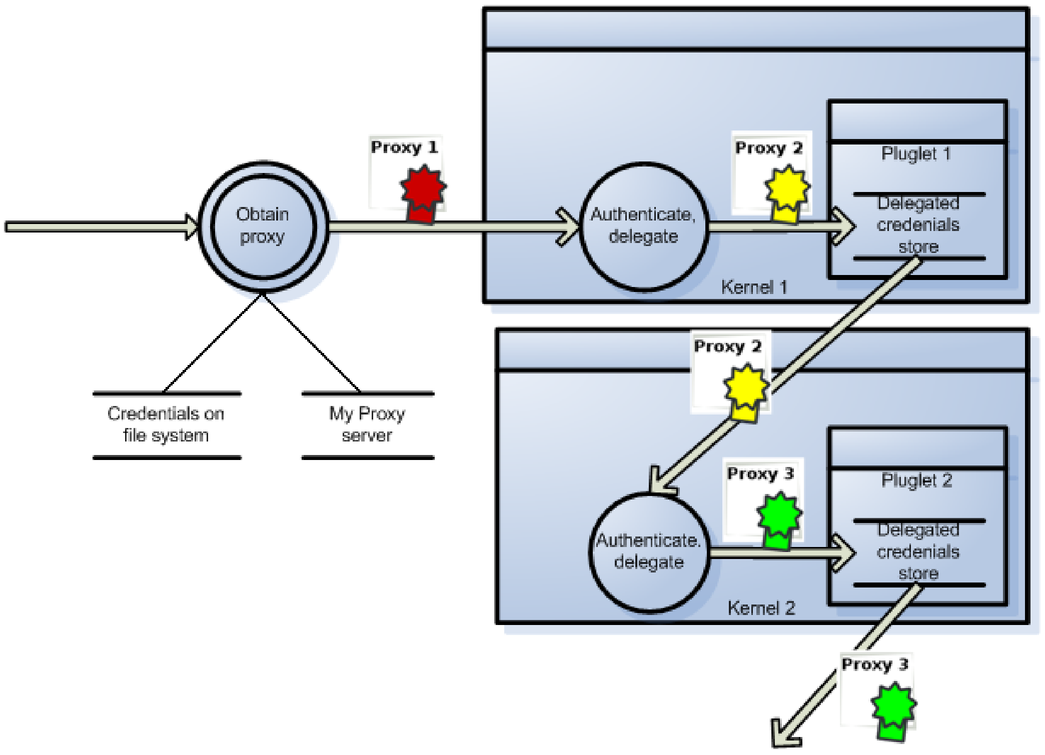


Figure 20. Data Flow Diagram of GSI Authenticator
The flow starts from obtaining a first proxy – either from the file system or from MyProxy server. The proxy is then used for authentication and delegated. The subsequent second proxy is saved for usage by pluglets on behalf of the user. After successive authentication and delegation next (third) proxy is received… and the process may go over.

4.6 Authentication sequence diagram and usage scenario

The details of message exchange and steps taken by the H2O client and kernel are going to be presented using the sequence diagram together with a corresponding usage scenario.

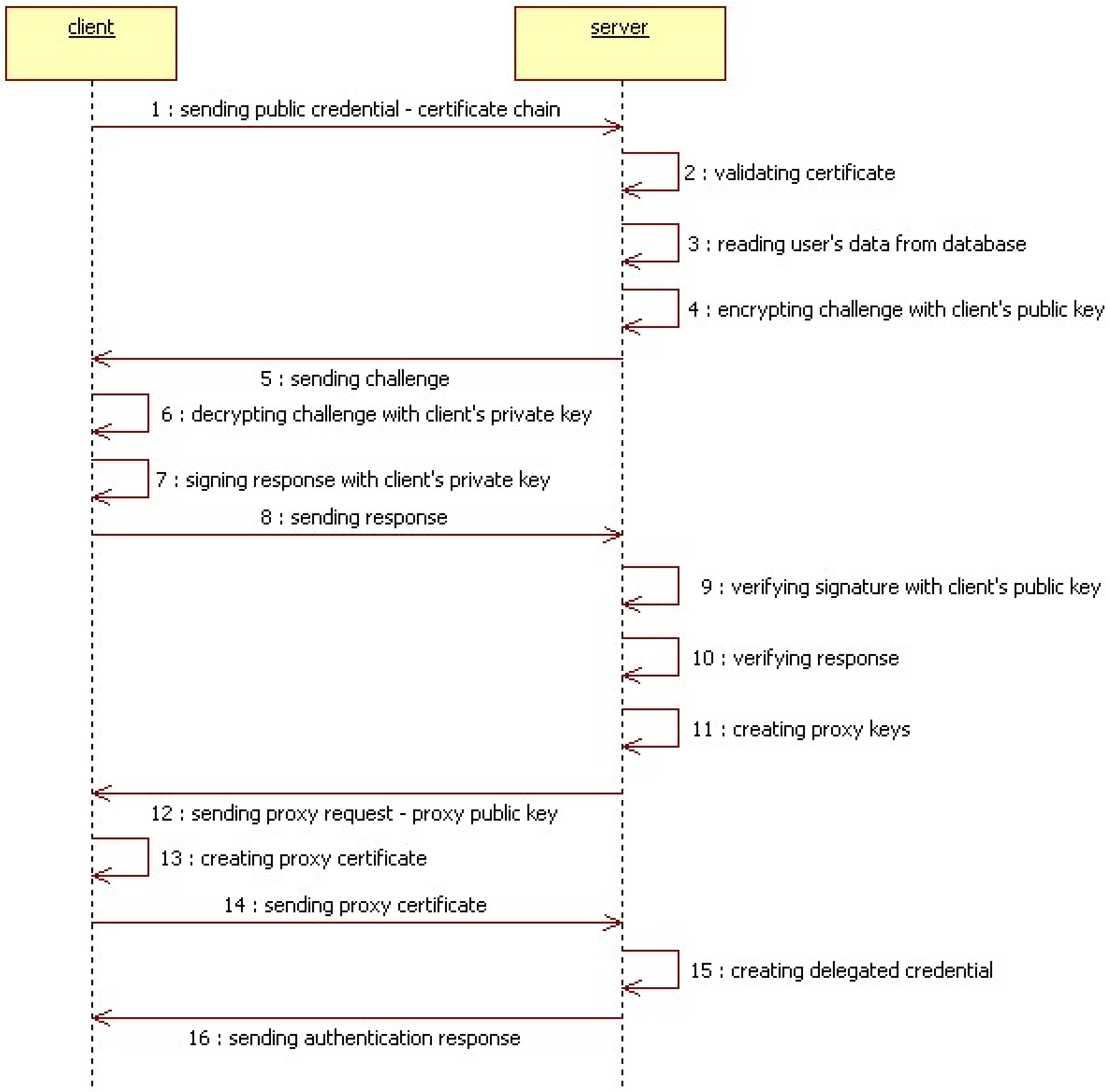

client
server
1 : sending public credential - certificate chain
2 : validating certificate
3 : reading user's data from database
4 : encrypting challenge with client's public key
5 : sending challenge
6 : decrypting challenge with client's private key
7 : signing response with client's private key
8 : sending response
9 : verifying signature with client's public key
10 : verifying response
11 : creating proxy keys
12 : sending proxy request - proxy public key
13 : creating proxy certificate
14 : sending proxy certificate
15 : creating delegated credential
16 : sending authentication response

Figure 21. Sequence diagram of GSI Authenticator
The following steps of the diagram are described in the use case in text below.

Figure 21 presents the sequence diagram of the GSI authenticator under assumption that the authentication successes and that the credential is to be delegated. The following use case scenario describes the steps of the diagram together with alternative paths.

Goal: user wants to authenticate himself to the kernel in order to perform some operations on it

Actors:

client – user application connecting to the kernel

server – the authenticator implementation on the server side

Preconditions:

- the credential from the Wallet is mapped to the proper authenticator
- an access to valid permanent credentials or proxy is provided

Triggers: authenticator engine starts the authentication process

Success guarantee : server responses with a success message and saves the client's principals (id, group membership) in login context

Basic flow:

<u>Introduction</u>

1. Client sends the chain of public certificates, the selected delegation policy and the 'required' flag
2. Server validates the chain
   The validation process for both standard and proxy certificates will be described in Appendix A.
3. Server reads user's data from the database

<u>Verification</u>

1. Server prepares a random challenge and encrypts it with client's public key
2. Server sends the challenge to the client
3. Client decrypts the challenge with his private key
   While using a proxy for single sign-on there is no need for prompting for a password.
4. Client signs the response with his private key
5. Client sends the response to the server.
6. Server verifies the signature of the response with client's public key
7. Server verifies the response

<u>Creating proxy certificate for delegation</u>

1. Server generates a proxy key pair
2. Server sends a proxy certificate request to the client
3. Client signs the certificate request using his private key
4. Client sends proxy certificate back to the server
5. Server creates the delegated credentials and saves them in user's context

6. Server sends the authentication success response to the client

Alternative scenarios:

2a : The validation fails
  2a1 : No groups are saved to user's login context
  2a2 : Server sends the authentication failure response to the client
  2a3 : The scenario is finished

3a : The user was not found in the database
  3a1 : No groups are saved to user's login context
  3a2 : If the delegation flag is DELEGATE_ALWAYS, the delegation is performed
  3a3 : Server sends the authentication failure response to the client
  3a4 : The scenario is finished

10a : The response is invalid
  10a1 : No groups are saved to user's login context
  10a2 : Server sends the authentication failure response to the client
  10a3 : The scenario is finished
11a : The delegation flag was DELEGATE_NOT
  11a1 : The delegation is not performed
  11a2 : The scenario goes to step 16

4.7 Summary

This chapter presented the concept and detailed design of GSI Authenticator, together with the successive steps of using it in the H2O authentication process: proxy creation, tunnel establishment, identity introduction and confirmation as well as delegation. The last three, which are 'core' of the development process, were described with details.

Chapter 5. Implementation

The chapter describes the implementation details of the GSI Authenticator. At the beginning, the implementation scope will be stated. Furthermore the external tools used will be described, especially the CoG JGlobus package, which contains the complete implementation of GSI. Next, a detailed description of implemented classes supported with UML diagrams as well as code examples of particular elements (e.g. validation and revocation) will be provided. Finally, the encountered problems with initial implementation without some external tools will be described in order to prevent further developers from possible difficulties.

5.1 Implementation scope

The main part of implementation concerns the new authenticator that will be plugged into the authenticators chain. An important issue is creating and using proxy certificates. It also requires some changes in the existing mechanism for serving the authenticators chain, in order to allow the verification of the proxies. Moreover, some new libraries are to be used and configuration system configuration has to be extended. Finally, some steps need to be taken to verify the ability of proxy and rights delegation. All the following steps will be described in this and subsequent chapters and in the Appendix:

- Preparations for authentication
    - Creating a proxy for single sign-on

- Client side implementation
    - Reading certificates / proxies from file system
    - Supplying credentials to the authenticator
    - Usage tutorial

- Authenticator in H2O
    - Server and client authenticator classes in H2O
    - Auxiliary classes

    - Validator (with revocation)
    - Encryption and decryption

- Authenticator in MOCCA
    - Supplying credentials in MOCCA
    - Usage tutorial

- Delegation issues:
    - Creating proxy for delegation
    - Verifying the delegation abilities of H2O mechanisms

- Build issues:
    - Adding new libraries (with security providers)
    - Updating build files

- Configuration
    - Adding new authenticator to chain
    - PKI configuration
    - CoG package configuration
    - Server authentication configuration
    - Setting permissions for security providers

5.2 Tools used

Java Cryptography Architecture, Java Cryptography Extension:

Cryptography-related part of Java is encompassed by the Java Cryptography Architecture [22] framework. Some of the elements that are provided by JCA API, are: digital signatures, message digests (hashes), certificates and certificate validation, encryption (symmetric/asymmetric block/stream ciphers), key generation and management, and secure random number generation.

The most of them are actually provided by the framework called Java Cryptography Extension (JCE) that once was a distinct package but now is bundled into the JDK distribution and therefore should be thought as a part of the JCA.

JCA (and at the same time JCE) allows for usage of multiple and interoperable cryptography implementations by using the architecture of so called *Cryptographic Service Provider*s.

Security providers:

The term refers to a package or set of packages, shortly called providers that supply an implementation of a subset of the Security API cryptography features. Depending on implementation, the provider may have different characteristics, thus allowing both developers and end-users to decide, which to use according to their needs.

In order to use the provider, it has to be registered in the system. There are two possibilities of registering the provider – statically, by editing a security properties configuration file of the JRE before running the application or dynamically, by calling a method at runtime.

The configuration file for static registration is located in `<java-home>/lib/security/java.security` and contains the entries, which declare providers together with their preference order n, like:

```
        security.provider.n=masterClassName
```

The default provider that comes with standard Sun's JDK is the SUN provider with main class Sun located in sun.security.provider package. The corresponding entry in the security file might be:

```
        security.provider.1=sun.security.provider.Sun
```

Dynamic registration may be performed by calling either the *addProvider()* or *insertProviderAt()* static methods from the Security class. It is not persistent across VM instances. Appropriate privileges are required by the java programs in order to register the provider dynamically[24].

Bouncy Castle[25]:

The Bouncy Castle provider is one of the products, which offer a collection of Java and C# API of multiple cryptographic algorithms, developed by the Legion of Bouncy Castle. It is based on the top of the other product, which is a low-level (also called light-weight) API suitable to use in any environment: i.a. memory constrained devices (available for J2ME) or with no easy access to the JCE libraries (applets). The provider is compatible with the JCE architecture and is publicly released under the terms of MIT License[26].

[24] For permissions configuration in H2O see Appendix C
[25] http://www.bouncycastle.org/
[26] www.opensource.org/licenses/mit-license.php

Cryptix JCE [27]:

The Cryptix JCE provider was created to address some US export restrictions problems of Sun JavaSoft implementations. It is fully compatible with Sun's implementation and released on the liberal, BSD-alike license. Implementations for both Java and Perl exist.

Both BouncyCastle and Cryptix are used by JGlobus, a part of Java CoG Kit.

The CoG (Commodity Grid) Kit:

The Commodity Grid (CoG) Kits were created to answer the developers desire to program the Grid in frameworks familiar to them, in order to enable rapid Gid application development. They allow Grid users, Grid application developers and Grid administrators to use, program, and administer Grids from a higher-level framework. Currently Java and python CoG Kits exist. They combine the advantages of the framework and grids in order to facilitate the development of advanced Grid services.

Java CoG Kit[28] and JGlobus[29]:

Because of the growth of the Java CoG Kit distribution, the contents of package was divided into separate modules. One of them is JGlobus, which is also an integral part of Globus Toolkit itself.

It provides client and limited server side capabilities. The most important from the point of view of this thesis is the complete implementation of GSI. Starting from the 1.4 version of JGlobus, the GSI library is complaint with standard described in RFC 3820 [13] (still providing backward compatibility with older versions of certificates). Some of other capabilities include MyProxy for certificate storage GridFTP for Remote Data Access or GRAM for remote job submission and monitoring.

CoG Kit Java Command Line tools:

There are several command-line tools that enter the Java CoG Kit. The relevant from the point of view of this thesis are the ones connected with certificates :

- grid-proxy-init – creates proxy certificate
- grid-cert-info – gives information about the permanent certificate
- grid-proxy-info – gives information about the proxy
- grid-proxy-destroy – delete the created proxy

[27] http://www.ntua.gr/cryptix/products/jce/
[28] http://wiki.cogkit.org, CoG Kits described also in [23]
[29] http://dev.globus.org/wiki/CoG_jglobus

All the tools use the CoG configuration file, described in the Appendix C, to find the location of specific files. Also COG_INSTALL_PATH and PATH variables are need to be set in order to run the tools.

Creating a proxy for single sign-on:

Using CoG JGlobus is a recommended way to create the proxy. Upon prior configuration (described in Appendix C), the *grid-proxy-init* command-line tool invocation creates a proxy from credentials and in the file specified in the configuration file. Using this tool all types of Globus proxies[30] can be generated (the default behavior depends on the version of tool).

5.3 GSI Authenticator classes

Following figures present UML class diagrams of the GSI Authenticator. The structure is mapped from the existing Password Authenticator and is compliant with the interfaces used for H2O authenticators chain processing.

First is the high-level class diagram (Figure 22). Main classes are GSIRemoteCredential and GSIRemoteAuthenticator, which contain AuthDialog client and server classes respectively that are used for message processing and exchange by the authentication protocol. The processing and exchange were presented on the sequence diagram in point 4.6 and are based on the *doPhase()* and *getNextToken()* methods of the AuthDialog classes.

The GSIRemoteCredential stores GSIPublicCredenial and GSIPrivateKey classes, which correspond to the credentials provided by user for authentication.

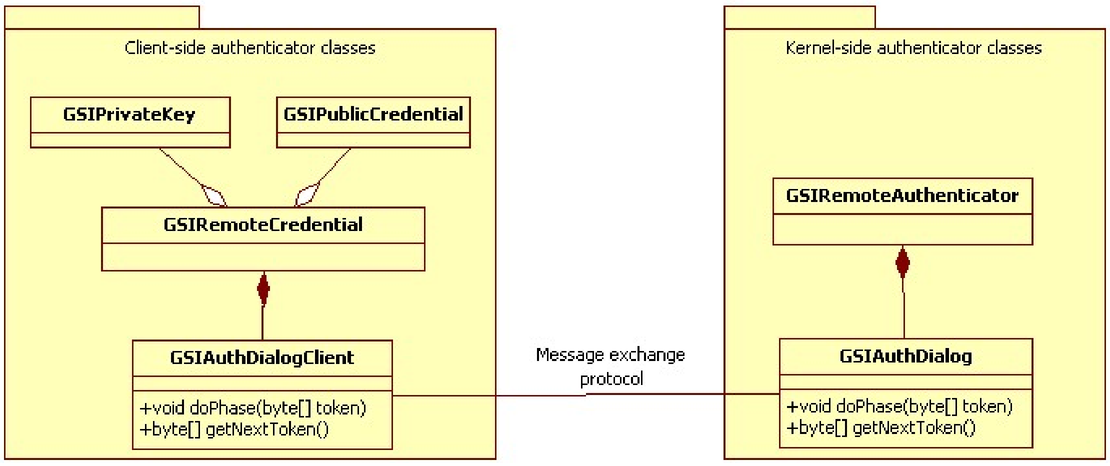


Figure 22. High-level class diagram

[30] See Appendix A for the types of Globus proxies

GSIPublicCredential

The class stores the certificate chain and the type of used proxy (if applicable) both with several methods that simplify using the credential as well as accessing some of the credential features:

- *readFrom()* and *writeTo()*
  allow for sending and receiving the credential through the ObjectStream in the authentication exchange protocol.

- *getPathLength()*
  returns the length of the stored certificate chain

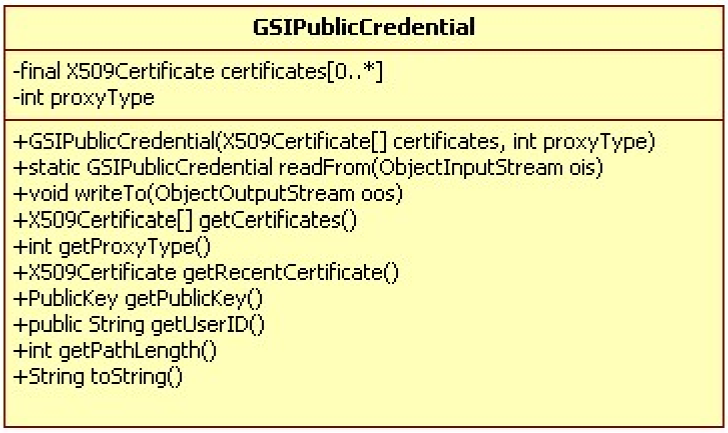
Very often the most recent certificate from the chain (which should be kept as the first one in the table) is used. Therefore a few methods for accessing the certificate are provided:

- *getRecentCertificate()*
  returns the certificate

- *getPublicKey()*
  returns the public key of the certificate

- *getUserID()*
  returns the Subject DN of the certificate

The class diagram is presented in Figure 23:



```
GSIPublicCredential
-final X509Certificate certificates[0..*]
-int proxyType
+GSIPublicCredential(X509Certificate[] certificates, int proxyType)
+static GSIPublicCredential readFrom(ObjectInputStream ois)
+void writeTo(ObjectOutputStream oos)
+X509Certificate[] getCertificates()
+int getProxyType()
+X509Certificate getRecentCertificate()
+PublicKey getPublicKey()
+public String getUserID()
+int getPathLength()
+String toString()
```

Figure 23. GSI PublicCredential class

GSIRemoteCredential

GSIRemoteCredential is a realization of the RemoteCredential interface, which indicates its ability to be used as a H2O Credential and put into credentials wallet. It stores the credentials in form of GSIPrivateKey and GSIPublicCredenial classes. The credentials are used by the GSIAuthDialogClient class for the authentication process itself.

GSIPrivateKey keeps the private key corresponding to the most recent certificate in the chain. It provides both cryptographic and proxy request signing methods:

- *decrypt()*, *encrypt()*
  use the private key for decryption and encryption of the provided byte array

- *createProxy()*
  signs the provided public key with own private key in order to create a proxy; used CoG method is described in subchapter 5.6.

The other attributes of the GSIRemoteCredential class define the way it has to be processed by the authenticator[31] :

- the 'required' flag
- the delegation policy of the credential

In order to create an instance of the class, three constructors are provided:

-
  ```
  public GSIRemoteCredential(GSIPublicCredential publicCred,
  GSIPrivateKey privCred, int delegationPolicy,
  boolean required);
  ```

  User is required to explicitely provide all the information:
  - public and private credential
  - delegation policy
  - 'required' flag

-
  ```
  public GSIRemoteCredential(GSIPublicCredential publicCred,
  PrivateKey passwd, boolean delegate);
  ```

  The 'required' flag is by default set to false
  The delegation policy is set to DELEGATE_IF_AUTHENTICATED or DELEGATE_NOT, depending on the boolean value provided

[31] See chain evaluation in Chapter 2.2.4

- `public GSIRemoteCredential(GlobusCredential cred, boolean delegate)`

  As above, but the public and private credentials are provided in form of GlobusCredential object.

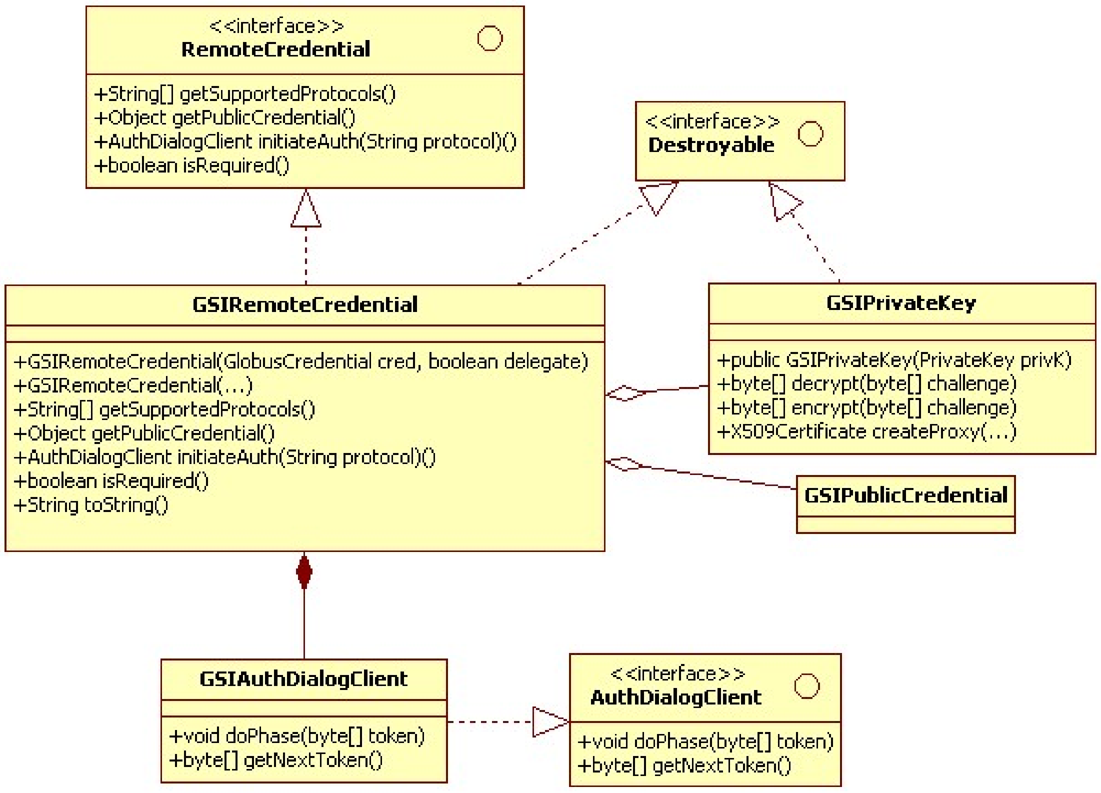
The methods of RemoteCredential interface provided by the GSIRemoteCredential class are:

- *getSupportedProtocols()*
  the class provides names of implemented authentication protocols, which are used for mapping credentials to authenticators afterwards; the method returns the protocols, which can utilize the credential

- *initiateAuth()*
  prepares the credential for the authentication message processing and exchange and returns the prepared AuthDialogClient

- *getPublicCredential()* and *isRequired()*
  are getters of the stored attributes

Figure 24 presentes the diagram for GSIRemoteCredential and related classes.



<<interface>>
RemoteCredential
+String[] getSupportedProtocols()
+Object getPublicCredential()
+AuthDialogClient initiateAuth(String protocol)()
+boolean isRequired()
<<interface>>
Destroyable
GSIRemoteCredential
+GSIRemoteCredential(GlobusCredential cred, boolean delegate)
+GSIRemoteCredential(...)
+String[] getSupportedProtocols()
+Object getPublicCredential()
+AuthDialogClient initiateAuth(String protocol)()
+boolean isRequired()
+String toString()
GSIPrivateKey
+public GSIPrivateKey(PrivateKey privK)
+byte[] decrypt(byte[] challenge)
+byte[] encrypt(byte[] challenge)
+X509Certificate createProxy(...)
GSIPublicCredential
GSIAuthDialogClient
+void doPhase(byte[] token)
+byte[] getNextToken()
<<interface>>
AuthDialogClient
+void doPhase(byte[] token)
+byte[] getNextToken()

Figure 24. GSIRemoteCredential class diagram

GSIRemoteAuthenticator

GSIRemoteAuthenticator is a realization of the RemoteAuthenticator interface, thus enabling it to be plugged into H2O authenticators chain.

The constructor gets the UserDatabase that is used by the kernel to read the groups of authenticated users. The database parse and access methods are provided in edu.emory.mathcs.h2o.security.auth.gsi package.

The two methods of the RemoteAuthenticator interface are similar to the ones of RemoteCredential:

- *getSupportedProtocols()*
  returns the names of protocols, which are provided by the authenticator

- *initiateAuth()*
  prepares the authenticator for the authentication message processing and exchange and returns the prepared AuthDialogClient

GSIAuthDialog implements the AuthDialog interface which is used for authentication message exchange as well as for retreiving some authentication results by the kernel. The latter are:

- getStatus
  returns the authentication result

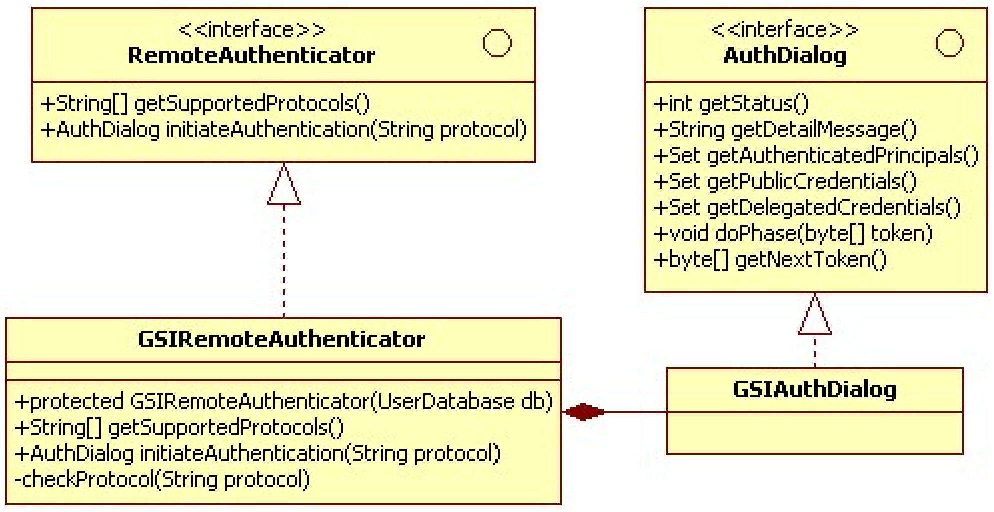
- *getDetailedMessage()*
  returns the message that describes the authentication result (e.g. failure reason)

- *getAuthenticatedPrincipals()*
  returns set of names and groups that were assigned to the authenticated user

- *getPublicCredentials()*
- returns the credentials that were used by the user for atuhentication

- *getDelegatedCredentials()*
  returns the credentials that were created during the delegation process

Figure 25 presentes the diagram for GSIRemoteAuthenticator and related classes.



<<interface>>
RemoteAuthenticator
+String[] getSupportedProtocols()
+AuthDialog initiateAuthentication(String protocol)
<<interface>>
AuthDialog
+int getStatus()
+String getDetailMessage()
+Set getAuthenticatedPrincipals()
+Set getPublicCredentials()
+Set getDelegatedCredentials()
+void doPhase(byte[] token)
+byte[] getNextToken()
GSIRemoteAuthenticator
+protected GSIRemoteAuthenticator(UserDatabase db)
+String[] getSupportedProtocols()
+AuthDialog initiateAuthentication(String protocol)
-checkProtocol(String protocol)
GSIAuthDialog

Figure 25. GSIRemoteAuthenticator class diagram

All the presented classes are placed in the edu.emory.mathcs.util.security.auth.spi.gsi package.

5.4 Other implemented classes

edu.emory.mathcs.util.security.MyCertUtils

The class provides several auxiliary static methods for the authenticator:

- *decrypt()* and *encrypt()*
  execute the cryptographic operations of GSIPrivateKey class

- *stringToCert()* and *certToString()*
  use Base64 encoder and decoder in order to map between X509Certificate object and string representation; this methods are provided because of problems with certificate representation while sending the X509Certificate object through ObjectStream; now their string representations are exchanged instead

- *changeCNFormat()*
  a method that is used in MyProxyPathValidator (described in point 5.5) in order to provide the validator with CNs of CRL issuers in form that corresponds to the format of CN notation in validated certificates. It performs mapping between two ways of CN notation with different order and white spaces usage, e.g. from

  - O=TestCA, ST=Some-State, C=PL – issuer's DN of validated certificate

to

  - C=PL,ST=Some-State,O=TestCA – issuer of CRL used for validation

- *readUserCredentials()*
  this method is provided in order to enable user to use some locations of credentials different from those served by CoG methods (which are described in subchapter 5.6). Two versions of the method exist. Both return objects of GlobusCredential class but can parse different locations of credentials:

  - `readUserCredentials(String certfile, String keyfile, H2OPasswordFinder passwd)`

    parses a certificate file and an encrypted RSA key file (GlobusCredential can parse only unencrypted files)

  - `readUserCredentials(String keystorePath, H2OPasswordFinder keystorePass, String alias, H2OPasswordFinder certPass)`

    gets the certificate and the key from a keystore; both keystore and the certificate can be secured with a password

In order to provide passwords for keyfile, keystore and certificate that are required by the presented methods, the subclasses of **H2OPasswordFinder** abstract class from the **edu.emory.mathcs.util.security.passwd** package are used (Figure 26).

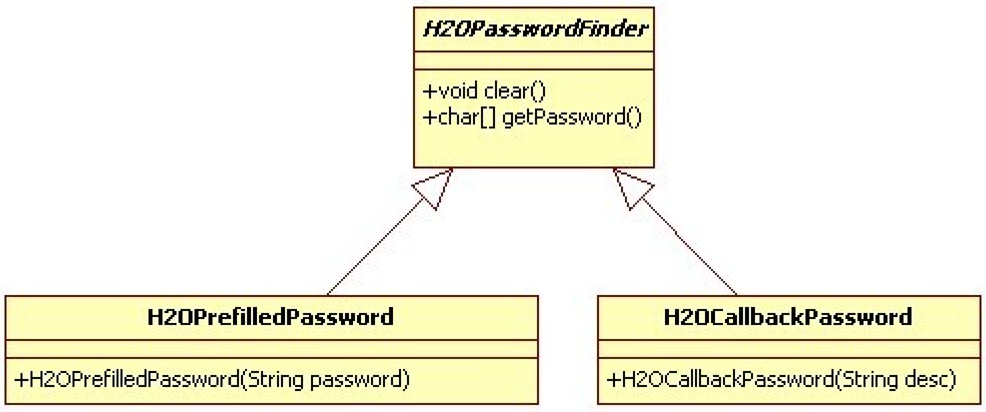


Figure 26. H2OPasswordFinder class diagram

The abstract class declares methods for obtaining a password and for clearing it from memory. The H2OPrefilledPassword class is used when the passphrase is known a priori and can be provided to the constructor. The H2OCallbackPassword class is used in order to ask user for the passphrase on demand and requires the user to type in the password during program execution. The 'desc' parameter is used for password request printout to describe the request to the user.

5.5 Revocation mechanism

The process of verifying if the validated certificate is not revoked, is often neglected by creation of systems based on X.509 certificates and may lead to serious security gaps. During development of GSI Authenticator, the possibility to use Certificate Revocation Lists was added in H2O. The changed files and the principle of operation will now be described.

KernelConfig.xml changes

The URLs of the Certificate Revocation Lists provided by CAs are to be provided in the *KernelConfig.xml* configuration file in `{h2o-dist}/config` directory. Its structure was developed in order to support new entries in the <Security> section:

```
<CRLLocations>
     <CRLLocationEntry location="<crl_url>"/>
</CRLLocations>
```

kernelConfig-1.0.dtd changes

The extension required some changes in *kernelConfig-1.0.dtd* - XML Document Type Definition file of the *KernelConfig.xml*, which is placed in edu.emory.mathcs.h2o.server.impl package. The Security element of the file was extended with CRLLocations element:

```
<!ELEMENT Security (KeyStores?, Identity?, Authenticators,
       TrustedCodeCerts?, AuthorizationPolicy, CRLLocations?)>
```

and the CRLLocations element itself was added as well:

```
<!ELEMENT CRLLocations (CRLLocationEntry*)>
     <!ELEMENT CRLLocationEntry EMPTY>
           <!ATTLIST CRLLocationEntry location CDATA #REQUIRED>
```

Main.java changes

The *KernelConfig.xml* file is parsed with the edu.emory.mathcs.h2o.server.impl.Main class. To use the new entries, Security subclass of the Main class was extended with the crlLocations list and proper methods.

ProxyPathValidator changes

In order to enable validation, some changes in JGlobus' ProxyPathValidator (org.globus.gsi.proxy) had to be performed because of the inconsistence in CN elements order. The modified class is placed in the util subproject of H2O as edu.emory.mathcs.util.security.MyProxyPathValidator.

Principle of operation

Globus Validator uses only CRLs from the file system, therefore H2O uses the locations saved in the config file to download the CRL files to the local file system during kernel startup. The files are subsequently used by the Validator during validation of certificates.

5.6 CoG usage code examples

In this chapter the most interesting mechanisms of the CoG JGlobus package, which were used in the GSI Authenticator, are going to be presented.

Reading credentials from file system

For reading certificates, keys and proxies from files, the org.globus.gsi.GlobusCredential class from JGlobus is used. It constructors allow to parse credentials from different sources:

```
GlobusCredential(InputStream input)
```

Creates a GlobusCredential from an input stream.

```
GlobusCredential(String proxyFile)
```

Creates a GlobusCredential from a proxy file.

```
GlobusCredential(String certFile, String unencryptedKeyFile)
```

Creates a GlobusCredential from certificate file and an unencrypted key file

The example usage is:

```
        GlobusCredential credd =
            new GlobusCredential("/tmp/x509up_u1000");
```

Creating proxy certificate

To create a proxy certificate on-the-fly, i.a. for delegation purposes, the org.globus.gsi.bc.BouncyCastleCertProcessingFactory class is used.

```
        BouncyCastleCertProcessingFactory factory =
            BouncyCastleCertProcessingFactory.getDefault()
```

The new certificate is created using the following method:

```
createProxyCertificate(X509Certificate issuerCert,
PrivateKey issuerKey, PublicKey publicKey, int lifetime, int proxyType,
X509ExtensionSet extSet, String cnValue)
```

The parameters are:

- issuerCert - the issuing certificate
- issuerKey - private key, corresponding to the public key of issuer certificate that will be used to sign the proxy
- publicKey - the public key of the new certificate
- lifetime - lifetime of the new certificate in seconds. If 0 (or less then) the new certificate will have the same lifetime as the issuing certificate.
- proxyType - can be one of
    - GSIConstants.DELEGATION_LIMITED,
    - GSIConstants.DELEGATION_FULL,
    - GSIConstants.GSI_2_LIMITED_PROXY,
    - GSIConstants.GSI_2_PROXY,
    - GSIConstants.GSI_3_IMPERSONATION_PROXY,
    - GSIConstants.GSI_3_LIMITED_PROXY,
    - GSIConstants.GSI_3_INDEPENDENT_PROXY,
    - GSIConstants.GSI_3_RESTRICTED_PROXY.
- extSet - a set of X.509 extensions to be included in the new proxy certificate.
- cnValue - the value of the CN component of the subject of the new certificate. If null, the defaults will be used depending on the proxy certificate type created.

For detailed description of the parameters as well as the attributes of created proxies (which depend on their type) please refer to javadoc[32].

The parameters used by H2O authenticator produce proxies of the same lifetime and type as the issuer's one. The type of the source proxy is recognized during authenticator initialization. If it is a plain certificate, the type GSIConstants.DELEGATION_FULL is used for creating proxies.

The delegation method uses the subject name of the issuing certificate to create the subject name of the proxy by appending a random number CN component.

Validator

The validation framework of H2O is placed in the edu.emory.mathcs.h2o.impl.TrustManagers class. In order to correctly validate the proxy certificates, the org.globus.gsi.proxy.ProxyPathValidator is used. The validation method takes three parameters:

```
validate(certPath, trustedCerts)

validate(certPath, trustedCerts, crlsList)
```

- certPath is a table of X509Certificate objects, starting with the most recent certificate
- trustedCerts is a table of trusted CAs' certificates
- crlsList is a CertificateRevocationLists object that specifies the locations of CRLs to use with the validator

The TrustManager class provides a method for getting the certificates of trusted CAs from both JSSE and kernel truststores.

Since the Globus validator is used, the authenticator is capable of serving all proxy types defined for GT, which were described above.

Revocation is available in Globus Validator although it is not described in the RFC document. The revocation check in H2O was already described.

Encryption and decryption

For encryption and decryption the JCE methods of Bouncy Castle provider are used:

```
        Cipher cipher = Cipher.getInstance(key.getAlgorithm(), "BC");
        cipher.init(mode, key);
        cipher.doFinal(bytes);
```

[32] www-unix.globus.org/cog/distribution/1.4/api/

The provided key is a public or private key, depending on usage. 'bytes' is the byte array to decode / encode. The used mode values are:

- `Cipher.ENCRYPT_MODE` - Constant used to initialize cipher to encryption mode.
- `Cipher.DECRYPT_MODE` - Constant used to initialize cipher to decryption mode.

5.7 Adding new authenticator to chain

In order to use the new authenticator, it has to be appended to the authenticators chain in the *KernelConfig.xml* configuration file. In the <Authenticators> section the following entry is to be added:

```
<Authenticator class="edu.emory.mathcs.h2o.security.auth.gsi.GSIAuthenticator"
trustdb="security/Users.xml"/>
```

The parser uses the XML Document Type Definition files to specify the database structure. The entries of files used by GSI Authenticator have to be included in the `h2o-server\build-jbexport.xml` file of the distribution:

```
<include name="edu/emory/mathcs/h2o/security/auth/gsi/XMLFileUserDB-0.8.dtd"/>
<include name="edu/emory/mathcs/h2o/security/auth/gsi/XMLFileUserDB-0.9.dtd"/>
```

5.8 Adding GSI Authenticator handling in MOCCA

The modified H2O authentication mechanisms are almost ready to use in the MOCCA framework. In order to enable it, the framework has to be extended with handling of H2O credentials and using them for performing connections to kernels. The required changes in MOCCA source code as well as build and running configuration are now going to be presented. As a source, version 0.10 of mocca-light was used [33]

Changes in source code:

In order to use H2O credentials in MOCCA, a singleton class MoccaAuthClient was created in mocca.client package. It provides methods for supplying the credentials as well as setting the trust manager and allows to obtain the H2OClient instance used for performing connection to H2O Kernel.

The class diagram for MoccaAuthClient is presented in Figure 27.

[33] Available at https://gforge.cyfronet.pl/projects/mocca

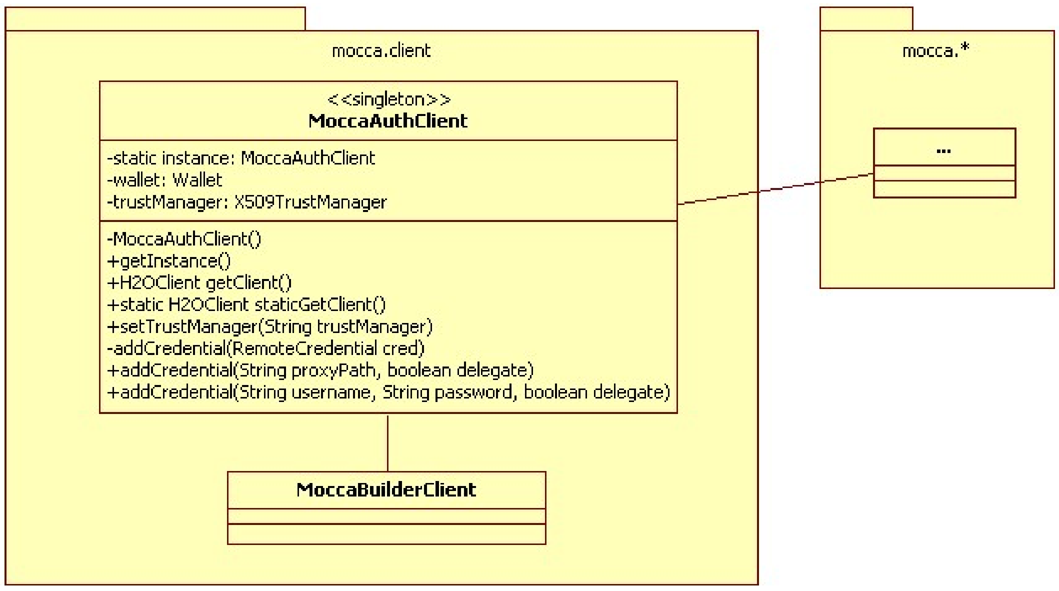

mocca.client
<<singleton>>
MoccaAuthClient
-static instance: MoccaAuthClient
-wallet: Wallet
-trustManager: X509TrustManager
-MoccaAuthClient()
+getInstance()
+H2OClient getClient()
+static H2OClient staticGetClient()
+setTrustManager(String trustManager)
-addCredential(RemoteCredential cred)
+addCredential(String proxyPath, boolean delegate)
+addCredential(String username, String password, boolean delegate)
MoccaBuilderClient
mocca.*
...

Figure 27. MoccaAuthClientSingleton class diagram

In order to prepare an instance of H2OClient, the following methods are used:

- *addCredential()* methods are used to add a H2O credential to the wallet of MoccaAuthClient; credentials for both GSIAuthenticator (proxy) and PasswdAuthenticator (username and password) may be provided

- the default trust manager used by MoccaAuthClient is TRUST_ALWAYS; it can be changed by using the *setTrustManager()* method; the valid parameters are: “always”, “certified” and “null” for TRUST_ALWAYS, TRUST_CERTIFIED and null trust managers respectively[34] ; using other value will throw an InvalidArgumentException

getClient() and *getStaticClient()* methods are used in order to get the instance of H2OClient, supplied with added credentials and selected trust manager; the existing source code of MOCCA was modified in order to provide the H2OClient for performing connections with kernels (in *getKernelContext()* and *getPlugletContext()* methods) :

- MoccaBuilderClient class:
    - in constructor, used for non-static methods of the class
    - in static methods of the class:
        - *invokeMethodOnComponent()*
        - *invokeMethodBySignature()*

[34] The trust managers of H2O are described in section 2.2.5

        - *invokeMethodByName()*
- MoccaClientConnection
    - in *getProviderPluglet()* method

Changes in build:

- Required CoG libraries were added to `{mocca_src}/lib/cog`
- changes in `{mocca_src}/build.xml` :
    - `/lib/cog/cog-jglobus.jar` added to the classpath
    - added copying of CoG libraries to `{mocca_dist}/lib/cog`
- changes in `{mocca_examples}/bin/runscript.sh`:
    - `$MOCCA_DIST/lib/cog/cog-jglobus.jar` added to classpath

It is important to build MOCCA with current H2O distribution provided in `h2o-dist` directory of MOCCA package.

5.9 Encountered problems with initial implementation without using CoG JGlobus package

The first idea of creating the proxy, still before using JGlobus, was to create own proxy-alike certificates using available providers, namely BouncyCastle. They were ordinary certificates and standard PKIX validator was used for validation. Several methods were implemented that allowed for parsing proxy files, creating a proxy and saving the proxy in a file. Combined together they created a grid-proxy-init-alike tool. However, the compatibility with Globus, which was required, could not be achieved. There were problems in both ways (for description of standards, which are mentioned here, please refer to the Appendix A)

- The end entity certificates, which are issued by Certificate Authorities, usually do not allow for signing other certificates – which is specified by the keyUsage extension. The same situation applies to the RFC 3820 Globus proxies. Thus creating the proxy-alike certificates from standard permanent credentials or RFC-compliant proxies was impossible – the PKIX Validator returned an java.security.cert.CertPathValidatorException: Not a CA certificate

- On the other hand, the certificates created with the grid-proxy-init-alike tool were not acceptable by Globus. The problem concerned the incompatibility of encoding standards of the private key:

- Globus uses the PKCS#1 standard for storing the private key in a proxy file (ASN.1-encoded in clear-text). However, using BouncyCastle there was no possibility of storing the key in this standard

- Instead, PKCS#8 (not encrypted, used by Apache) was used, which was not parsed by Globus.

Some code examples of the initial implementation are provided in Appendix B.

5.10 Implementation summary

During the implementation of the project the available source code and existing libraries were tried to be reused. Therefore the implementation of the authentication algorithm itself was not extensive. The H2O project was extended with 28 new source files, including usage examples and tests. Much more code were written during the phase of technology recognition as well as initial implementation described above. Several original H2O classes were changed as well, correcting a few existing bugs by the way (i.a. related to obtaining delegated credentials). 12 files of external libraries were used. All the source code was written in Java.

Several ant build files as well as execution scripts (both windows and unix) end configuration files were edited. Multiple asymmetric keys and certificates were generated with the CoG tools and signed using test Certificate Authorities that were created using the OpenSSL package.

The created source code and distribution together with example configuration files and credentials are placed in the SVN repositories:

- H2O:
  http://dcl.mathcs.emory.edu/bin/viewvc/software/harness2/trunk
  (Developer Access tab on H2O page [35])

- MOCCA
  https://gforge.cyfronet.pl/svn/mocca/branches/mocca-gsi

[35] http://dcl.mathcs.emory.edu/h2o

Chapter 6. Usage and validation of GSI Authenticator

The aim of this section is to verify the usefulness and robustness of the implemented authenticator. An entire description of H2O usage will be presented, paying special attention to the GSI Authenticator. Moreover, a description of providing GSI credentials to MOCCA will be presented. Additionally, the environment configuration in form of the required Public Key Infrastructure will be introduced.

Finally, the propriety and quality of the solution will be confirmed by presenting implemented test suites, performance tests with measurements as well as threat analysis with the description of proper configuration that will protect from known attacks.

6.1 Example usage in H2O

An example client application was created in order to present the abilities of the GSI Authenticator. The aim of this example is to present the following aspects of the authenticator:

- Supplying credentials to the authenticator
- Authentication with GSI Authenticator
- Deployment of pluglets
- Usage of delegated credentials on deployed pluglets

The subsequent steps of the example are:

1. Reading proxy from file and supplying it to the Wallet
2. Connecting to kernel with authentication and delegation
3. Example usage scheme:
    a. Deployment of the first pluglet
    b. Subsequent recursive deployments of pluglets
    c. Invocation of the *hello()* method

The implemented example usage scheme is presented on the diagram in Figure 28 and described below:

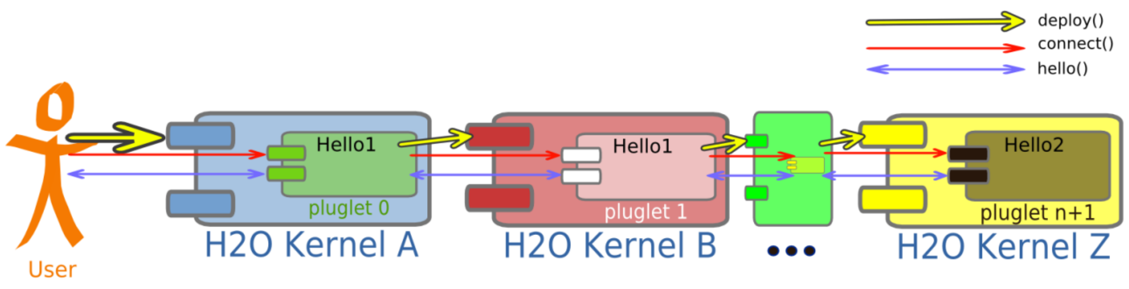


Figure 28. Implemented example usage scheme in H2O
Client deploys a pluglet, authenticating himself to the Kernel A with delegation enabled. After deployment, the *connect()* method of the 'Hello1' pluglet (no. 0) running on kernel side is invoked. In that method the pluglet connects to (may be another) kernel B (with subsequent authentication and delegation) and recursively deploys itself on it. This process repeats, until the specified amount of pluglets (n) is deployed. Finally, the last instance deploys another pluglet, 'Hello2' (no. n+1).
After deployment, client invokes a *hello()* method of the deployed pluglet 0. The invocation is recursively processed in the chain of pluglets. The last pluglet answers with the common "Hello world" greeting, which is returned back along the chain to the client.

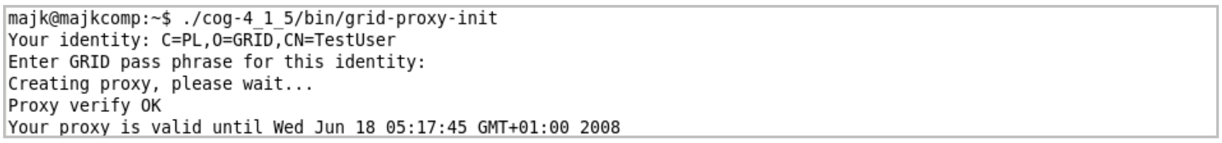
In order to run the example it is necessary to:

- Build the distribution
- Configure the CoG distribution e.g. with provided keys
    - In case of using own keys instead of those provided with distribution:
        - Truststore has to be supplied with the issuer's certificate
        - *Users.xml* has to be supplied with subject's CN
- Configure file permissions in *Policy.xml* file
- Optionally configure CRL locations and server authentication
- Provide a valid path to Proxy file on tutorial execution

The steps are respectively described in Appendix C. The remaining commands are:

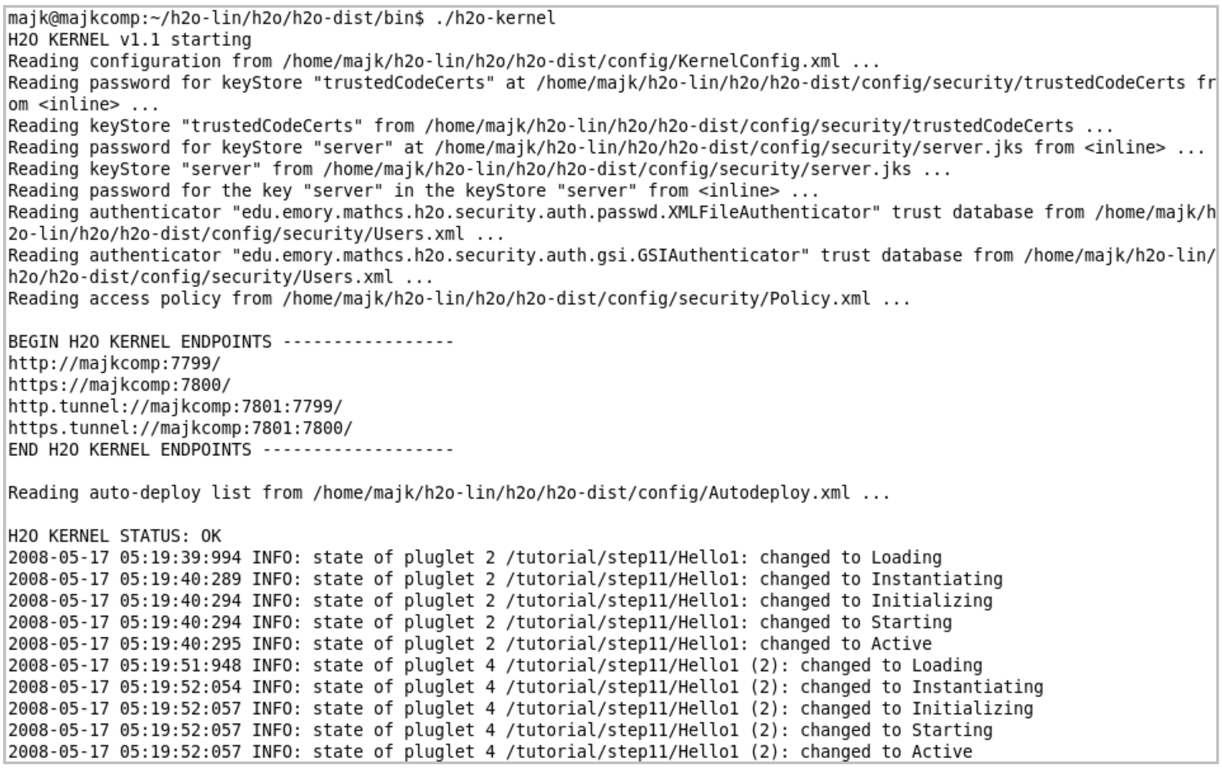
- Creating a proxy (Figure 29):
    - `{cog-home}/bin/grid-proxy-init`
- Running a kernel (Figure 30):
    - `{h2o-dist}/bin/h2o-kernel`
- Running an example (Figure 31):
    - `{h2o-dist}/tutorial/bin/step11 <endpoint> <path_to_proxy> <n>`

where available endpoints will be listed by kernel upon startup



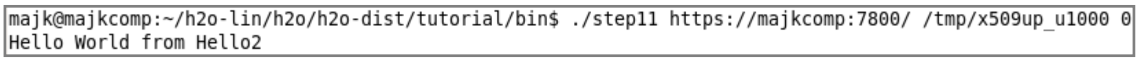
```
majk@majkcomp:~$ ./cog-4_1_5/bin/grid-proxy-init
Your identity: C=PL,O=GRID,CN=TestUser
Enter GRID pass phrase for this identity:
Creating proxy, please wait...
Proxy verify OK
Your proxy is valid until Wed Jun 18 05:17:45 GMT+01:00 2008
```

Figure 29. Creating a proxy – console log



```
majk@majkcomp:~/h2o-lin/h2o/h2o-dist/bin$ ./h2o-kernel
H2O KERNEL v1.1 starting
Reading configuration from /home/majk/h2o-lin/h2o/h2o-dist/config/KernelConfig.xml ...
Reading password for keyStore "trustedCodeCerts" at /home/majk/h2o-lin/h2o/h2o-dist/config/security/trustedCodeCerts fr
om <inline> ...
Reading keyStore "trustedCodeCerts" from /home/majk/h2o-lin/h2o/h2o-dist/config/security/trustedCodeCerts ...
Reading password for keyStore "server" at /home/majk/h2o-lin/h2o/h2o-dist/config/security/server.jks from <inline> ...
Reading keyStore "server" from /home/majk/h2o-lin/h2o/h2o-dist/config/security/server.jks ...
Reading password for the key "server" in the keyStore "server" from <inline> ...
Reading authenticator "edu.emory.mathcs.h2o.security.auth.passwd.XMLFileAuthenticator" trust database from /home/majk/h
2o-lin/h2o/h2o-dist/config/security/Users.xml ...
Reading authenticator "edu.emory.mathcs.h2o.security.auth.gsi.GSIAuthenticator" trust database from /home/majk/h2o-lin/
h2o/h2o-dist/config/security/Users.xml ...
Reading access policy from /home/majk/h2o-lin/h2o/h2o-dist/config/security/Policy.xml ...

BEGIN H2O KERNEL ENDPOINTS -----------------
http://majkcomp:7799/
https://majkcomp:7800/
http.tunnel://majkcomp:7801:7799/
https.tunnel://majkcomp:7801:7800/
END H2O KERNEL ENDPOINTS -------------------

Reading auto-deploy list from /home/majk/h2o-lin/h2o/h2o-dist/config/Autodeploy.xml ...

H2O KERNEL STATUS: OK
2008-05-17 05:19:39:994 INFO: state of pluglet 2 /tutorial/step11/Hello1: changed to Loading
2008-05-17 05:19:40:289 INFO: state of pluglet 2 /tutorial/step11/Hello1: changed to Instantiating
2008-05-17 05:19:40:294 INFO: state of pluglet 2 /tutorial/step11/Hello1: changed to Initializing
2008-05-17 05:19:40:294 INFO: state of pluglet 2 /tutorial/step11/Hello1: changed to Starting
2008-05-17 05:19:40:295 INFO: state of pluglet 2 /tutorial/step11/Hello1: changed to Active
2008-05-17 05:19:51:948 INFO: state of pluglet 4 /tutorial/step11/Hello1 (2): changed to Loading
2008-05-17 05:19:52:054 INFO: state of pluglet 4 /tutorial/step11/Hello1 (2): changed to Instantiating
2008-05-17 05:19:52:057 INFO: state of pluglet 4 /tutorial/step11/Hello1 (2): changed to Initializing
2008-05-17 05:19:52:057 INFO: state of pluglet 4 /tutorial/step11/Hello1 (2): changed to Starting
2008-05-17 05:19:52:057 INFO: state of pluglet 4 /tutorial/step11/Hello1 (2): changed to Active
```

Figure 30. Running a kernel – console log
The kernel prints out i.a. used authenticators and available endpoints. Furthermore two instances of Hello1 pluglet are deployed and started.



```
majk@majkcomp:~/h2o-lin/h2o/h2o-dist/tutorial/bin$ ./step11 https://majkcomp:7800/ /tmp/x509up_u1000 0
Hello World from Hello2
```

Figure 31. Running an example – console log
The answer is returned from Hello2 pluglet

6.2 Example usage in MOCCA

Verification of changes performed in MOCCA was done using the examples provided with distribution. Here the changes and execution of `{mocca_dist}/bin/moccaping.py` will be shown.

Part of the modified script file is presented in the code snippet:

```
 1 import sys
 2
 3 from mocca.client import MoccaMainBuilder
 4 from mocca.client import MoccaBuilderClient
 5 from mocca.client import MoccaAuthClient
 6 from java.net import URI
 7 from mocca.srv.impl import MoccaTypeMap
 8
 9 authClient = MoccaAuthClient.getInstance()
10 authClient.addCredential("/tmp/x509up_u1000", 1)
11 #authClient.addCredential("username","password",1)
12 authClient.setTrustManager("always")
13
14 builder = MoccaMainBuilder()
15
16 uriKernel = URI.create("https://majkcomp:7800/")
17 uriKernel2 = URI.create("https://jano:7800/")
18
19 userBuilderID = builder.addNewBuilder(uriKernel, "MyBuilderPlugletA")
20 providerBuilderID = builder.addNewBuilder(uriKernel2,
"MyBuilderPlugletB")
21
22 properties = MoccaTypeMap()
   …
```

For providing credentials to MOCCA, the MoccaAuthClient singleton class is used. The following entries were added to the script:

Line 5 : importing the MoccaAuthClient class
Line 9 : getting the instance of the singleton
Line 10 : adding credential to MOCCA; here proxy file path is provided that will be used for GSI Authenticator; the second parameter (boolean value written as integer) is used to select, whether the delegation should be performed
Line 11 : an example entry of data provided for PasswdAuthenticator
Line 12 : setting the trust manager for the connection with kernel
For detailed description of the methods please refer to section 5.8.

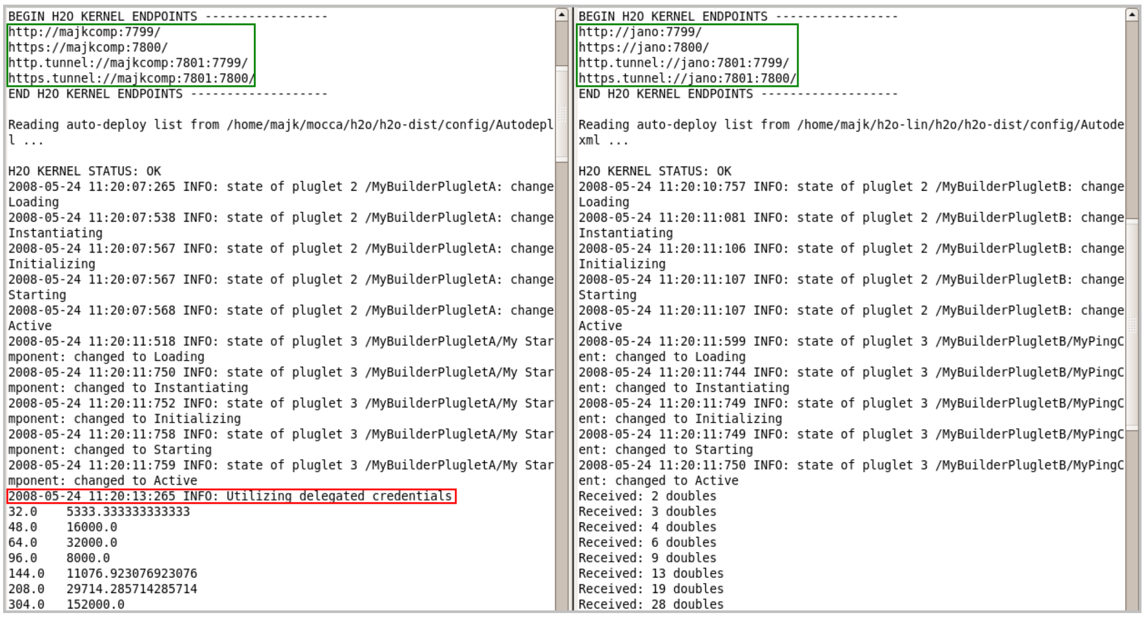
In the example two kernels were used, running on two separate hosts: *majkcomp* and *jano*. Connections to both of them were performed using secure endpoints (see lines 16 and 17)

In order to run the example, the following permissions have to be set in H2O kernel:

```
<permission type="java.net.SocketPermission" target="*"
actions="connect,resolve"/>

<permission type="java.lang.RuntimePermission"
target="accessDeclaredMembers"/>
```

Upon proper configuration, the H2O kernels and MOCCA builder can be run with the following commands:

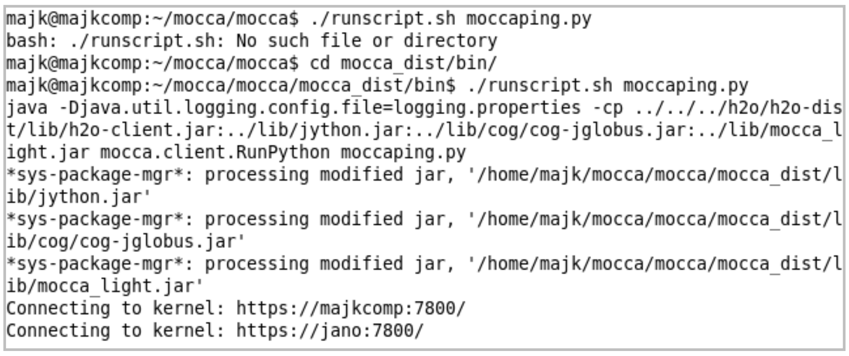
- Running kernels on both hosts (Figure 32):
  - `{h2o-dist}/bin/h2o-kernel`
- Running MOCCA builder (Figure 33):
  - `{mocca_dist}/bin/runscript.sh moccaping.py`



```
BEGIN H2O KERNEL ENDPOINTS -----------------
http://majkcomp:7799/
https://majkcomp:7800/
http.tunnel://majkcomp:7801:7799/
https.tunnel://majkcomp:7801:7800/
END H2O KERNEL ENDPOINTS -------------------

Reading auto-deploy list from /home/majk/mocca/h2o/h2o-dist/config/Autodepl
l ...

H2O KERNEL STATUS: OK
2008-05-24 11:20:07:265 INFO: state of pluglet 2 /MyBuilderPlugletA: change
Loading
2008-05-24 11:20:07:538 INFO: state of pluglet 2 /MyBuilderPlugletA: change
Instantiating
2008-05-24 11:20:07:567 INFO: state of pluglet 2 /MyBuilderPlugletA: change
Initializing
2008-05-24 11:20:07:567 INFO: state of pluglet 2 /MyBuilderPlugletA: change
Starting
2008-05-24 11:20:07:568 INFO: state of pluglet 2 /MyBuilderPlugletA: change
Active
2008-05-24 11:20:11:518 INFO: state of pluglet 3 /MyBuilderPlugletA/My Star
mponent: changed to Loading
2008-05-24 11:20:11:750 INFO: state of pluglet 3 /MyBuilderPlugletA/My Star
mponent: changed to Instantiating
2008-05-24 11:20:11:752 INFO: state of pluglet 3 /MyBuilderPlugletA/My Star
mponent: changed to Initializing
2008-05-24 11:20:11:758 INFO: state of pluglet 3 /MyBuilderPlugletA/My Star
mponent: changed to Starting
2008-05-24 11:20:11:759 INFO: state of pluglet 3 /MyBuilderPlugletA/My Star
mponent: changed to Active
2008-05-24 11:20:13:265 INFO: Utilizing delegated credentials
32.0    5333.333333333333
48.0    16000.0
64.0    32000.0
96.0    8000.0
144.0   11076.923076923076
208.0   29714.285714285714
304.0   152000.0
```

```
BEGIN H2O KERNEL ENDPOINTS -----------------
http://jano:7799/
https://jano:7800/
http.tunnel://jano:7801:7799/
https.tunnel://jano:7801:7800/
END H2O KERNEL ENDPOINTS -------------------

Reading auto-deploy list from /home/majk/h2o-lin/h2o/h2o-dist/config/Autode
xml ...

H2O KERNEL STATUS: OK
2008-05-24 11:20:10:757 INFO: state of pluglet 2 /MyBuilderPlugletB: change
Loading
2008-05-24 11:20:11:081 INFO: state of pluglet 2 /MyBuilderPlugletB: change
Instantiating
2008-05-24 11:20:11:106 INFO: state of pluglet 2 /MyBuilderPlugletB: change
Initializing
2008-05-24 11:20:11:107 INFO: state of pluglet 2 /MyBuilderPlugletB: change
Starting
2008-05-24 11:20:11:107 INFO: state of pluglet 2 /MyBuilderPlugletB: change
Active
2008-05-24 11:20:11:599 INFO: state of pluglet 3 /MyBuilderPlugletB/MyPingC
ent: changed to Loading
2008-05-24 11:20:11:744 INFO: state of pluglet 3 /MyBuilderPlugletB/MyPingC
ent: changed to Instantiating
2008-05-24 11:20:11:749 INFO: state of pluglet 3 /MyBuilderPlugletB/MyPingC
ent: changed to Initializing
2008-05-24 11:20:11:749 INFO: state of pluglet 3 /MyBuilderPlugletB/MyPingC
ent: changed to Starting
2008-05-24 11:20:11:750 INFO: state of pluglet 3 /MyBuilderPlugletB/MyPingC
ent: changed to Active
Received: 2 doubles
Received: 3 doubles
Received: 4 doubles
Received: 6 doubles
Received: 9 doubles
Received: 13 doubles
Received: 19 doubles
Received: 28 doubles
```

Figure 32. Running H2O kernel for MOCCA example – console log
The kernels are running on two separate hosts. After deployment, pluglets exchange messages. An additional information can be seen that was printed out in order to present, when the delegated credentials are used by the pluglet in kernel on *majkcomp* to perform connection to kernel on *jano*.



```
majk@majkcomp:~/mocca/mocca$ ./runscript.sh moccaping.py
bash: ./runscript.sh: No such file or directory
majk@majkcomp:~/mocca/mocca$ cd mocca_dist/bin/
majk@majkcomp:~/mocca/mocca/mocca_dist/bin$ ./runscript.sh moccaping.py
java -Djava.util.logging.config.file=logging.properties -cp ../../../h2o/h2o-dis
t/lib/h2o-client.jar:../lib/jython.jar:../lib/cog/cog-jglobus.jar:../lib/mocca_l
ight.jar mocca.client.RunPython moccaping.py
*sys-package-mgr*: processing modified jar, '/home/majk/mocca/mocca/mocca_dist/l
ib/jython.jar'
*sys-package-mgr*: processing modified jar, '/home/majk/mocca/mocca/mocca_dist/l
ib/cog/cog-jglobus.jar'
*sys-package-mgr*: processing modified jar, '/home/majk/mocca/mocca/mocca_dist/l
ib/mocca_light.jar'
Connecting to kernel: https://majkcomp:7800/
Connecting to kernel: https://jano:7800/
```

Figure 33. Running MOCCA example – console log
Moccaping example uses secure endpoints of *majkcomp* and *jano* hosts. Using the connections, the specified in script pluglets are deployed and connected and their methods are invoked.

6.3 PKI configuration [36]

Ten chapter presents the elements of the Public Key Infrastructure that has to be implemented in the organization in order to take advantage of the H2O together with GSI Authenticator and use it in a secure manner. The required entities will be presented, together with procedures that should be provided by the PKI.

Entities:

- **Registration Authority (RA)** – collects certificate requests; verifies users' identities – e.g. by checking the identity card; therefore usually requires human to human interaction

- **Certification Authority (CA)** – issues certificates, CRL lists, certifies subsequent CAs, provides the repository of issued certificates; the Root CA of the PKI is the root of trust and most often implemented by using a self-issued certificate, therefore the strength of the key must be high and the private key must be protected in the best possible way

- **Clients and Kernels**
  Both communicating parties have to possess certificates issued by CAs. Different CAs might be used but they have to be trusted by the peer.

[36] The description is based on [24]

Procedures:

The three main procedures in the PKI are presented on the chart in Figure 34 and described below:

1. Key generation

A key pair may be generated either by the user herself or by the CA. The former is more secure, because the private key is never sent. While using the latter option, the generated keys have to be transported in a secure way afterwards (e.g. using secured smart cards).

2. Registration

The process of certificate issuance has to be premised by the identity verification.
Upon certificate signing request user provides some personal information, required by the Certification Practices Statement (CPS) of the Registration Authority. Before issuing the certificate, RA has to confirm the provided information.

3. Certification

After identity confirmation, the request is signed by the CA and the public certificate is returned back to the user. Moreover, it can be placed on some public repository managed by the CA to allow users to fetch other users' certificates for the purpose of secure message exchange.

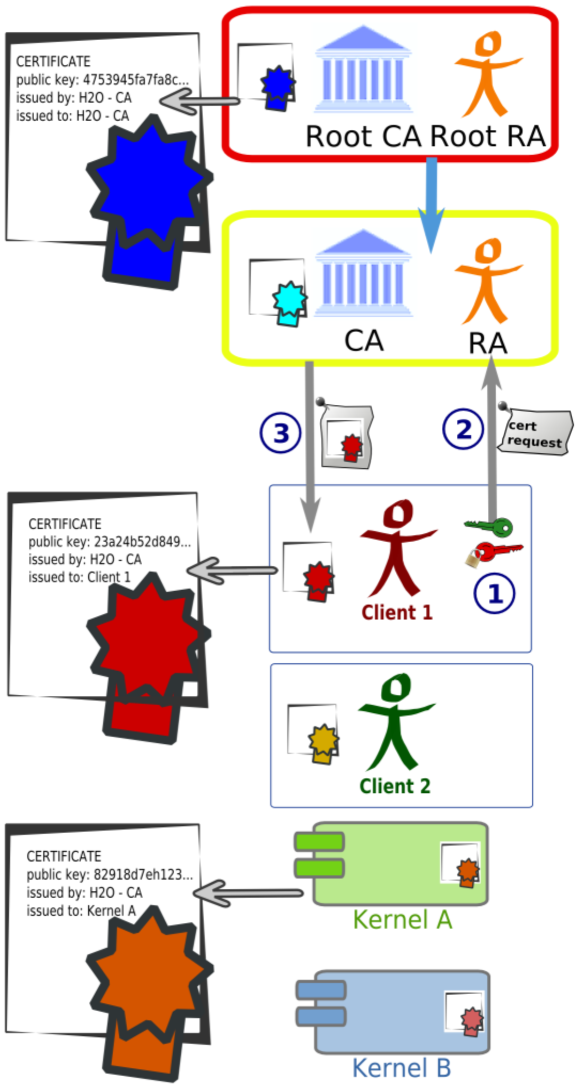


Figure 34. Suggested PKI for H2O
Each entity possess its own certificate, signed by H2O-CA (with a self-signed certificate) and has to trust the CA. The process of issuing a certificate consists of key generation (1), registration (2) and certification (3).

The key generation procedure is simple while using CoG JGlobus package. The grid-cert-request command line tool generates RSA keys together with certificate request, which is created on basis of some information, provided by the user (Common Name, Locality etc.). The file should be sent to the respective CA. The received certificate (usercert.pem file) should be placed in the directory defined by Globus configuration[37]

Other procedures:

- **Key and Certificate Update**
  Required when:
  - The lifetime of a certificate is expired – a normal situation
  - The certificate is compromised – an unusual satiation; the old certificate is revoked and placed on the CRL list; a new certificate has to be issued

- **Certificate Revocation**
  - Performed when the information contained by a certificate are not valid anymore, e.g. the employee has left the company or the personal information of the certificate owner had changed

- **Key recovery**
  - Performed when the owner lost her keys. Possible only when the PKI allows for keeping a safe backup of the keys. It is still important to allow no one but the owner to access them.

6.4 Test suites

The existing H2O tests were extended with a new suite, testing the correctness of authentication using GSI. It verifies the H2O kernel responses on connection attempts using both valid and invalid GSI credentials. The valid credential is a proxy issued by a trusted party to a subject that is known to the kernel (specified in the *Users.xml* file). The analyzed invalid cases are:

- The subject is unknown to the kernel
- The CA is not trusted by the kernel
- The lifetime of the proxy is over

[37] See Appendix C for configuration

- The certificate is revoked

In order to perform those tests some example Certificate Authorities, keys and certificates had to be prepared:

- An example Certificate Authority (CN=TestCA) with self-signed certificate valid for 10000 days was created using OpenSSL. The certificate was added to the trusted certificates of the kernel. The same credentials are provided with the distribution for the purpose of running the usage examples.

- Two key pairs along with certificate requests were created using CoG's grid-certificate-request tool: one for valid user (CN=TestUser) and one for invalid (CN=FalseTestUser)

- The TestCA was used to sign the certificate requests – a certificate (valid 10000 days) was created both for TestUser and FalseTestUser

- Another certificate (valid 10000 days) was created for TestUser. The certificate has been revoked by the CA.

- The CoG's grid-proxy-init tool was used to create a proxy for both users:
    - One proxy valid for 10000 days for both TestUser and FalseUser
    - Additional proxy valid for 1 hour for TestUser

- Another CA (CN=FalseTestCA) was created and used to sign the TestUser certificate request; another long-life proxy was created for the user. The CA certificate was not supplied to the truststore of the kernel.

The details of the credentials are presented in Table 1.

Table 1. Properties of credentials used for tests
The credential that should be accepted by the kernel is highlighted with green background. In case of other credentials, reasons of their being incorrect are emphasized.

	Correct	***Incorrect***	***Incorrect***	***Incorrect***	***Incorrect***
Issuer	*TestCA*	*TestCA*	*TestCA*	*FalseTestCA*	*TestCA*
Subject	*TestUser*	*TestUser*	*FalseTestUser*	*TestUser*	*TestUser*
Cert validity time	*~2035*	*~2035*	*~2035*	*~2035*	*~2035*
Proxy validity time	*~2035*	*Not valid anymore*	*~2035*	*~2035*	*~2035*
Revoked	*No*	*No*	*No*	*No*	*Yes*

In order to perform the tests, some changes in suite configuration had to be performed:

- The GSI Authenticator was added to the authenticators chain (entry in *GSITestKernelConfig.xml* configuration file of the test suite). The PasswordAuthenticator couldn't be removed from the chain for it is used by the test framework to enter the kernel

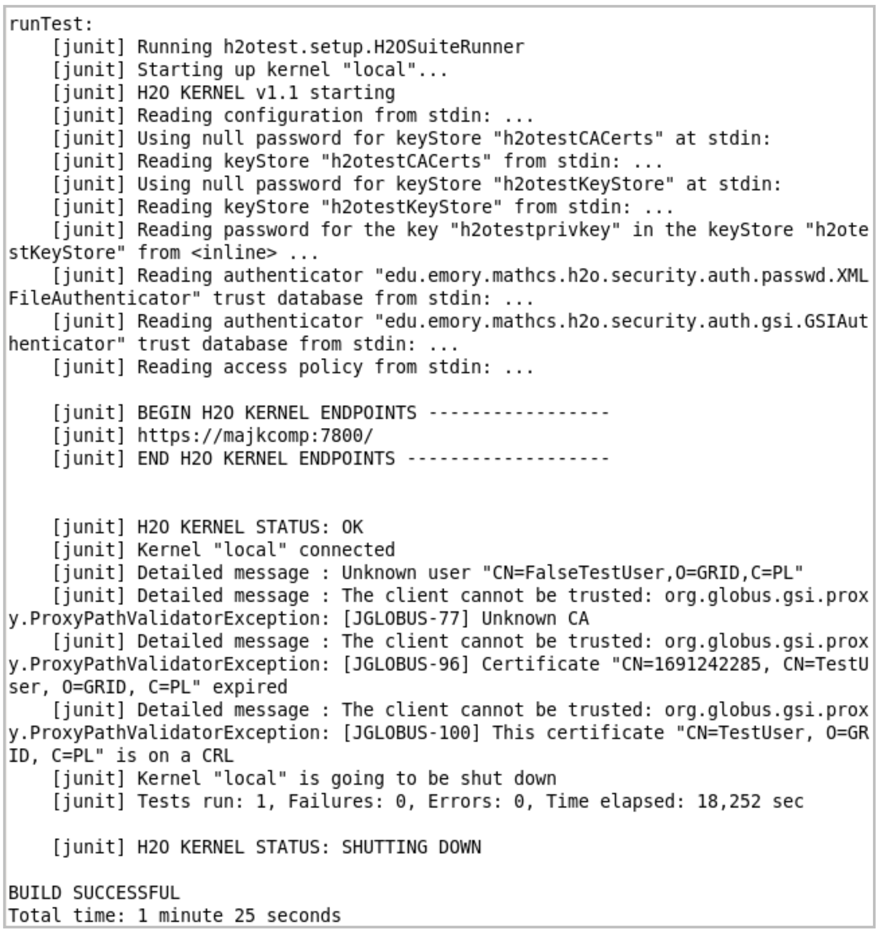
- The CN of TestUser was supplied to the *Users.xml* configuration file of the test suite

- The location of TestCA CRL list was added into the *GSITestKernelConfig.xml* configuration file of the test suite

The test named GSIKernelAccessControlTest and test proxies are placed in h2otest.cases.gsi package. The test suite class together with kernel configuration files (*Policy.xml* and *Users.xml*) are placed in h2otest.suites.gsi package and its subpackages. Additionally, the GSITestCase class was created in the h2otest.cases package

Before running the test, it is necessary to be customize the paths of file permissions in *Policy.xml* file in h2otest.suites.gsi.config.security package in order to reflect the true location of the files. Afterwards the test can be run with the Ant tool by running the following command in the `{h2o-dist}/h2o-test` directory (Figure 35) :

ant runGSITests



```
runTest:
    [junit] Running h2otest.setup.H2OSuiteRunner
    [junit] Starting up kernel "local"...
    [junit] H2O KERNEL v1.1 starting
    [junit] Reading configuration from stdin: ...
    [junit] Using null password for keyStore "h2otestCACerts" at stdin:
    [junit] Reading keyStore "h2otestCACerts" from stdin: ...
    [junit] Using null password for keyStore "h2otestKeyStore" at stdin:
    [junit] Reading keyStore "h2otestKeyStore" from stdin: ...
    [junit] Reading password for the key "h2otestprivkey" in the keyStore "h2ote
stKeyStore" from <inline> ...
    [junit] Reading authenticator "edu.emory.mathcs.h2o.security.auth.passwd.XML
FileAuthenticator" trust database from stdin: ...
    [junit] Reading authenticator "edu.emory.mathcs.h2o.security.auth.gsi.GSIAut
henticator" trust database from stdin: ...
    [junit] Reading access policy from stdin: ...

    [junit] BEGIN H2O KERNEL ENDPOINTS ----------------
    [junit] https://majkcomp:7800/
    [junit] END H2O KERNEL ENDPOINTS ------------------


    [junit] H2O KERNEL STATUS: OK
    [junit] Kernel "local" connected
    [junit] Detailed message : Unknown user "CN=FalseTestUser,O=GRID,C=PL"
    [junit] Detailed message : The client cannot be trusted: org.globus.gsi.prox
y.ProxyPathValidatorException: [JGLOBUS-77] Unknown CA
    [junit] Detailed message : The client cannot be trusted: org.globus.gsi.prox
y.ProxyPathValidatorException: [JGLOBUS-96] Certificate "CN=1691242285, CN=TestU
ser, O=GRID, C=PL" expired
    [junit] Detailed message : The client cannot be trusted: org.globus.gsi.prox
y.ProxyPathValidatorException: [JGLOBUS-100] This certificate "CN=TestUser, O=GR
ID, C=PL" is on a CRL
    [junit] Kernel "local" is going to be shut down
    [junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 18,252 sec

    [junit] H2O KERNEL STATUS: SHUTTING DOWN

BUILD SUCCESSFUL
Total time: 1 minute 25 seconds
```

Figure 35. Running test suite – console log
Noticeable are the printouts of detailed messages for invalid credentials: unknown user, unknown CA, expired proxy and revoked certificate

6.5 Threat analysis

During the development of security elements it is essential to perform a threat analysis and try to look at as many aspects that can contain security gaps, as possible. Even the most sophisticated and laboriously made mechanisms are useless, if they contain a single weakpoint.

The H2O authentication uses both cryptography and network protocols, which are a source of many threats. Moreover the broad possibilities of H2O transport layer configuration provides large flexibility, but vulnerabilities as well.

While performing the threat analysis, the Microsoft Threat Analysis and Modeling Tool[38] was used. Several possible attacks and the way of preventing them using proper H2O configuration and GSI Authenticator will now be presented.

[38] Search on http://msdn.microsoft.com

6.5.1 Analyzed threats

The analysis concerns the authentication process together with subsequent communication session and encompasses the following threats :

- Confidentiality threats – refer to unauthorized disclosure of the executing identity and data

- Integrity Threats – concerning violation of access control and business role (gaining access as a different user) as well as integrity of transferred data

- Availability Threats - affect mostly the quality of offered services. The primary factors are unavailability and performance degradation, which in commercial application may lead to business loss. The Availability Threats will only be shortly mentioned as they are not outside the interest of the thesis.

6.5.2 Analyzed attacks on the system

The following points will present the description and countermeasures of several attacks that may affect H2O, which were discovered by the Microsoft Threat Analysis and Modeling Tool. Afterwards the detailed analyze is presented and applied H2O mechanisms are described:

a. Cryptanalysis Attacks

Cryptanalysis is the science of cracking codes, decoding secrets, violating authentication schemes and breaking cryptographic protocols. It is also the science devoted to finding and correcting weaknesses in cryptographic algorithms. It is understood within the field of Cryptology that an algorithm should not rely on its secrecy. An algorithm should always be made available for public scrutiny. It is this scrutiny that will make it a well trusted algorithm. Inevitably, vulnerability in the algorithm will be exploited.

Countermeasures:

- ✓ Use well-known implementations of well-known cryptographic algorithms
- ✓ Utilize SSL or IPSec w/ Encryption to establish a secure communication channel
- ✓ Use cryptographically generated random keys

H2O uses well-known opensource implementations for both GSI Authenticator elements (JCE providers and JGlobus package) and for securing the connection (SSL-secured connection in RMIX framework). The GSI Authenticator instead of using plain passwords, which are known to provide many vulnerabilities (related to their complexity as well as the way of using them by the users), is based on asymmetric cryptography, which provides much better cryptographic properties.

b. Network Eavesdropping

Network Eavesdropping is the act of monitoring network traffic for data, such as clear-text passwords or configuration information. With a simple packet sniffer, all plaintext traffic can be read easily. Also, lightweight hashing algorithms can be cracked and the payload that was thought to be safe can be deciphered.

Countermeasures:

- ✓ Utilize SSL or IPSEC w/ Encryption to establish a secure communication channel

Unencrypted connection allows for eavesdropping the authentication credentials. It can be dangerous while using some trivial authentication mechanism, like H2O default password authenticator. Therefore the secure endpoint has to be used for tunneling the authentication process. GSI Authenticator does not require the connection to be encrypted – neither during authentication nor during delegation any secret data are transferred. Still it has to be tamper-proof, in order to prevent credentials manipulation.

c. Session Hijacking

Session hijacking is the act of taking control of a user session after successfully obtaining or generating an authentication session ID. In session hijacking an attacker using a captured, brute forced or reverse-engineered session ID seizes control of a legitimate user's Web application session while that session is still in progress. The severity of the damage incurred depends on what's stored in the session state.

Countermeasures:

- ✓ Use strong random numbers for session IDs

The authentication process is usually combined with the procedure of establishing a shared session keys only known to the authenticated parties. The keys are then used for subsequent communication i.a. to ensure them that the peer is really the one that took part in the authentication.

Absence of this mechanism may provide for kind of session hijacking with a simple replay attack. Let us consider the following situation from Figure 36:

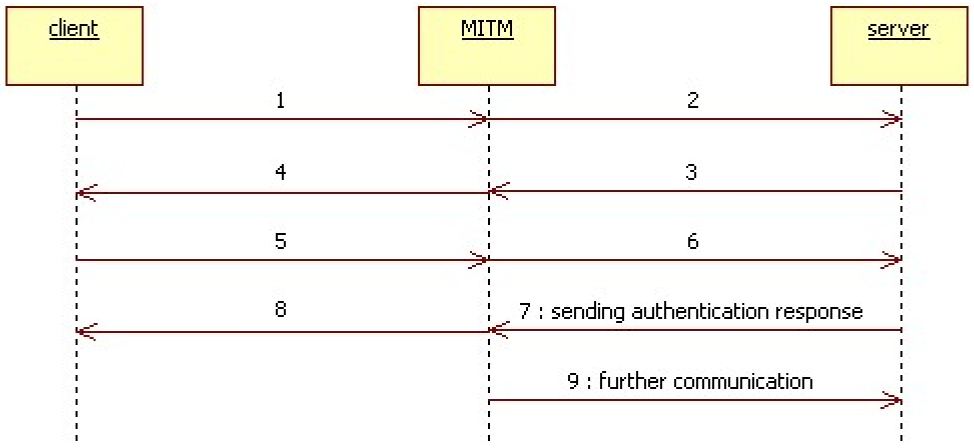

client
MITM
server
1
2
4
3
5
6
8
7 : sending authentication response
9 : further communication

Figure 36. Sequence diagram of man-in-the-middle attack without shared communication key.
The authentication messages exchange between the authorized user and server may be captured and replayed by an attacker. Even though the messages may be encrypted, and the attacker may not know what the actual keys and passwords are, the retransmission of valid logon messages is sufficient to gain access to the server. After succeeding, it can start its own communication with server as an authorized user.

Sometimes the attacker may try to replay the messages later, even on next days. This threat may easily be avoided by using timestamp or 'nonces' (one-time random numbers) in the authentication procedure. But the presented real-time replay cannot be avoided in that way.

H2O however provides the mechanism of sessions, which was described in section 2.2.2, thus being resistant to the presented situation. The suggested countermeasure is obtained by using asymmetric cryptography in form of session certificates and keys, which make it difficult to impersonate the session.

d. Man-in-the-middle attack

A man in the middle attack occurs when the attacker intercepts messages sent between the sender and receiver. The attacker then changes message and sends it to the original recipient.

Countermeasures:

- ✓ Utilize SSL or IPSec w/ Encryption to establish a secure communication channel
- ✓ Utilize a well-known authentication protocol to authenticate the server

In order to assure a secure connection, the crucial thing is the authentication of the server. Ignoring it may lead to another man-in-the-middle attack, even while using the secure tunneled connection. Let us analyze the diagram from Figure 37:

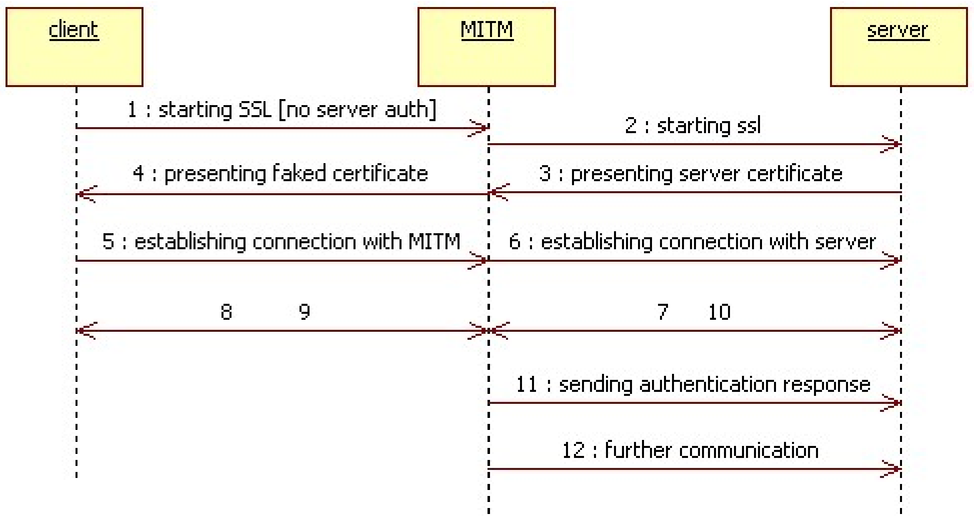

client
MITM
server
1 : starting SSL [no server auth]
2 : starting ssl
4 : presenting faked certificate
3 : presenting server certificate
5 : establishing connection with MITM
6 : establishing connection with server
8
9
7
10
11 : sending authentication response
12 : further communication

Figure 37. Sequence diagram of man-in-the-middle attack without server authentication The client starts an SSL connection. The MITM impersonates the server and presents a faked certificate. The client establishes a secured connection with MITM. At the same time a secured connection is being established between the MITM and the server. Afterwards, the MITM replies the messages between client and server, encrypting them and decrypting accordingly to the peer it communicates with. Upon successful authentication, it starts its own further communication with the server.

In order to prevent it, server's identity at the TLS handshake has to be verified by the client e.g. with a certificate signed by a known CA that it accepts. That will prevent client from establish a faked connection and frustrate the attack.

6.5.3 GSI Authenticator threat analysis

After considering the available attacks on the H2O system, let us focus on the features of the GSI Authenticator. At this point the following steps of the scheme will be analyzed in order to see, if they can be misused for impersonation.

Assumptions:

- the authentication is tunneled in a secure (at least tamper-proof) connection; otherwise the subsequent connection session won't be provided and the authentication process is useless
- server must always be authenticated; otherwise it can be used for the man-in-the-middle attack as described above

Let us analyze the authentication procedure step by step. We will consider a legitimate user and an attacker, both performing a connection to a kernel. In order to authenticate, the user has to:

1. send his certificate

The attacker may use the user's certificate for introduction in this step, because it is public. It won't be useful though, as we well see soon.

2. receive and decrypt the challenge

Only the owner of the corresponding private key (the legitimate user) can decrypt the challenge. It will not be possible for the attacker, thus the challenge obtained by attacker during authentication with the user's certificate cannot be used for subsequent steps.

3. send signed response

Again - only the owner of the corresponding private key (the legitimate user) can sign the response. For the attacker's connection with the server, the attacker needs to obtain a valid response signed by the user. Let us consider the following possibilities:

- an attacker may try to modify a connection between a legitimate user and a kernel and alter a challenge sent by the kernel to the user (exchange it with the challenge which he got) in order to use the user's response for own connection afterwards; this is prevented by the secure outer channel which is at least tamper-proof

- an attacker may try to replay an old response of legitimate user; in order to prevent that the challenge contains the actual time (in milliseconds) – ensuring that it won't repeat

- an attacker may try to start the connection exactly at the same time as another legitimate user to get the same challenge and use the user's response; in order to prevent that, the challenge contains also a random integer value, which makes it almost impossible to get the same challenge by different users in the same millisecond

In none of the possibilities the attacker is possible to get valid response and authenticate to the kernel.

6.5.4 Conclusions

This chapter presented possible threats and recommended countermeasures for H2O security. The security of H2O authentication and subsequent communication is provided by H2O mechanisms, however requires proper configuration, in order to prevent form known attacks, which were described. The configuration includes

- Using secure credentials with GSI Authenticator
- Using secured endpoint for tunneling the authentication process and providing a communication session
- Always authenticating the server in order to prevent main-in-the-middle attack

6.6 Performance tests

Some performance tests were carried out in order to measure the overhead of the GSI Authenticator as well as to compare it with the existing Password Authenticator and with the configuration with no authentication enabled. Furthermore, the usage of SSL tunneling on the authentication time was verified.

Additional tests were performed to estimate the percentage usage of particular parts in the authentication process.

Communication overhead, which is equal to all the authentication schemes, was not considered in the tests.

Test configuration:

OS : Ubuntu 8.04 Hardy Heron
Processor: Centrino Duo T2300 1,66 GHz (one core enabled)
Memory: 2 GB

Key length : 1024 bits

The results of the tests together with risk analysis from the security point of view will now be presented:

6.6.1 Authenticators comparison

In order to perform the comparison tests of the authenticators, the example usage classes presented in chapter 6.1 were made use of, differently configured. Three factors were adjusted:

- The length of the pluglets chain (namely the amount of recursively deployed instances of first pluglet after deployment by the client, before deploying the second one) : 0, 5, 10, 20

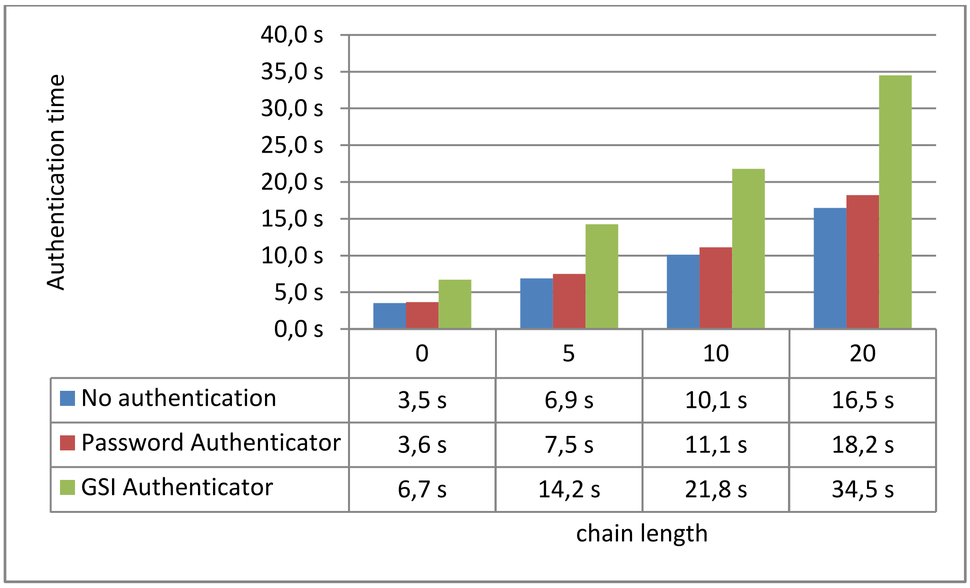
- Used authenticator : GSI-based, password-based or none (in this case guest' permissions have to be enabled)
- The existence of an 'outer' tunneling protocol for the authentication: using plain or SSL-secured endpoint

A short script was written to perform the tests, which executed it 5 times for each configuration. Extreme results were omitted and the average of the remaining was taken.

Following tables (Table 2-4) and charts (Figure 38-40) present the results divided into those obtained for plain and SSL socket. Authentication time for particular authentication schemes and chain lengths are presented together with the overhead of the GSI Authenticator over other schemes.

It is important to notice that the achieved time results include the time of pluglets deployment – so the overhead of GSI Authenticator over no authentication scheme is a real result of how the authentication affects the execution time.

Results for plain socket:



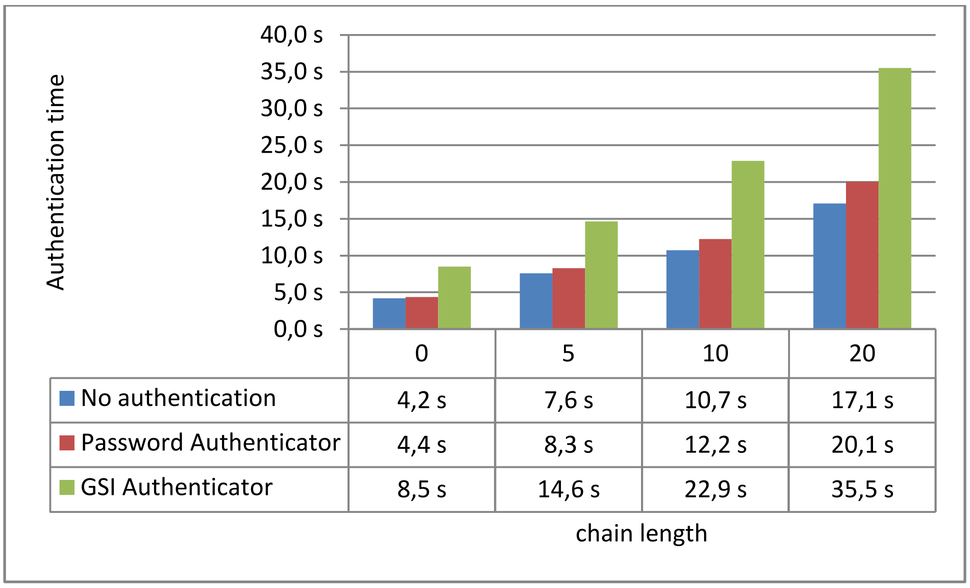
	0	5	10	20
🟦 No authentication	3,5 s	6,9 s	10,1 s	16,5 s
🟥 Password Authenticator	3,6 s	7,5 s	11,1 s	18,2 s
🟩 GSI Authenticator	6,7 s	14,2 s	21,8 s	34,5 s

Figure 38. Authentication time depending on authentication scheme and chain length for plain socket
Noticeable is the high overhead of GSI Authenticator comparing to other schemes

Table 2. The overhead of GSI Authenticator over other authentication schemes for plain socket

	0	5	10	20	average
GSI vs password	84,3 %	89,7 %	96,4 %	89,3 %	**89,9 %**
GSI vs no auth	90,6 %	106,5 %	115,6 %	109,2 %	**105,5 %**

Results for SSL socket:



	0	5	10	20
🟦 No authentication	4,2 s	7,6 s	10,7 s	17,1 s
🟥 Password Authenticator	4,4 s	8,3 s	12,2 s	20,1 s
🟩 GSI Authenticator	8,5 s	14,6 s	22,9 s	35,5 s

Figure 39. Authentication time depending on authentication scheme and chain length for SSL socket

Table 3. The overhead of GSI Authenticator over other authentication schemes for SSL socket

	0	5	10	20	average:
GSI vs password	95,4 %	77,0 %	86,9 %	77,0 %	**84,1 %**
GSI vs no auth	102,4 %	93,6 %	113,0 %	107,7 %	**104,2 %**

The obtained results were used to estimate the overhead of the connection using SSL socket over plain connection. The differences are graphically presented on the charts and the overhead is listed in the table. Most important is the average overhead in the last column.

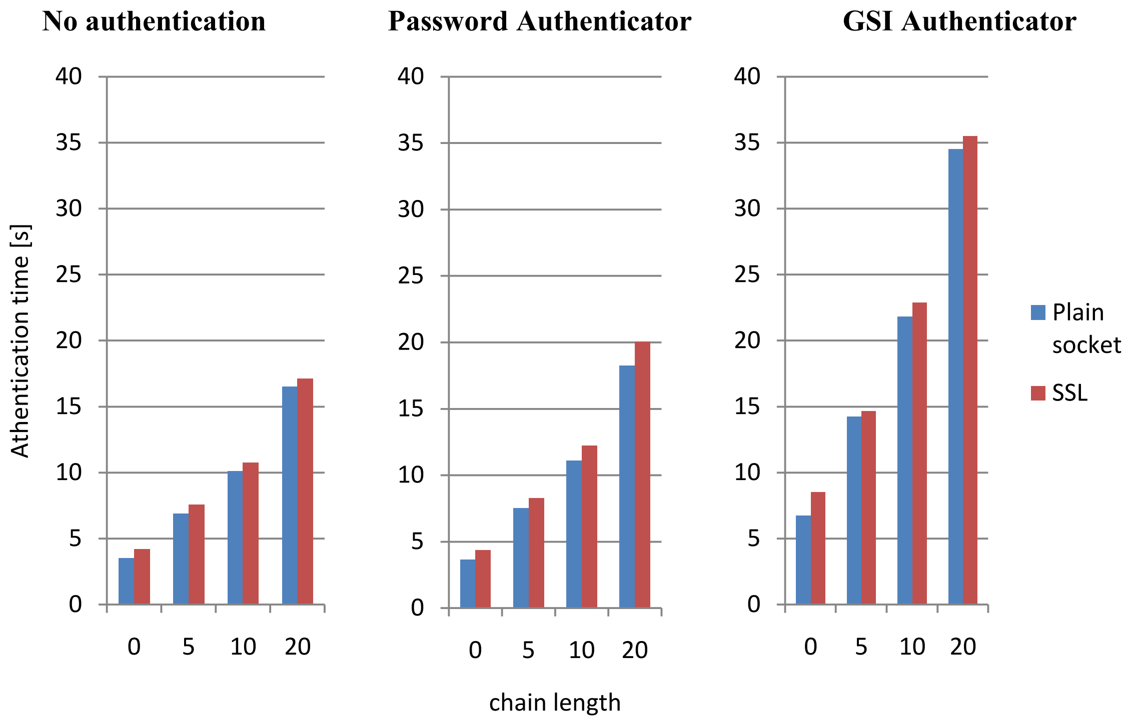


Figure 40. Compared authentication time for plain and SSL socket for particular authentication schemes

Table 4. The overhead of SSL-secured authentication over plain connection for particular authentication schemes

	0	**5**	**10**	**20**	**average:**
No authentication	19,3 %	9,7 %	6,1 %	3,6 %	**9,7 %**
PasswdAuthenticator	19,5 %	10,3 %	10,2 %	10,1 %	**12,5 %**
GSIAuthenticator	26,7 %	2,9 %	4,9 %	2,9 %	**9,3 %**

6.6.2 Server authentication

A short test was performed in order to check the server authentication overhead. The test was performed for a chain length 10 for SSL authentication. The authentication with server authentication enabled was about 5,7% longer than without it (19,5 s vs 18,45 s).

Conclusions:

What's obvious, the execution without authentication is the fastest one. The authentication mechanism doesn't provide much overhead though – connecting using the simple password authenticator scheme is only a bit slower than without any authentication at all. The GSI Authenticator is much slower than other two authentication schemes – using it will take us twice as much time as using no authentication. It is no large problem when not many authentications are needed but for a longer authentication time it may become inconvenient.

The usage of SSL-secured connection doesn't change the relative times of authentication, still provides about 10% overhead. Another 5% is added while using server authentication. Since not much data are sent, the time is mostly affected by the SSL handshake.

Noticeable is a very small overhead when GSI Authenticator is often use that can be seen in Table 4. It may be a result of the fact that similar cryptographic operations are required for both establishing the SSL connection and for the GSI Authenticator – which can be optimized by CPU.

As a conclusion, Figure 41 may be used in order to summarize, how the execution time depends on the authentication scheme and used endpoint:

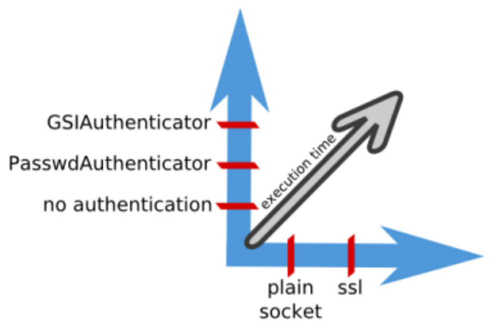


Figure 41. The change of execution time depending on the authentication scheme and used endpoint
Using GSI Authenticator over secured endpoint provides the highest security at the expense of longer execution time

6.6.3 Risk analysis

While considering the performance of security-related issues it is always important to analyze, how much time can be gained – and at the same time how much security can be lost – by using different possible configurations. Some suggestions can be provided by taking into account the performed threat analysis and gained performance results.

It is by no means reasonable to abandon the SSL for securing the connection, especially in a public network. The overhead is not so high – but the gained security features are significant.

The decision whether to use the GSI Authentication should depend on the company policy. It offers much higher security level than e.g. the password authenticator, together with a reasonable delegation. The decision, if we can resign from those features in support of half execution time is considerable.

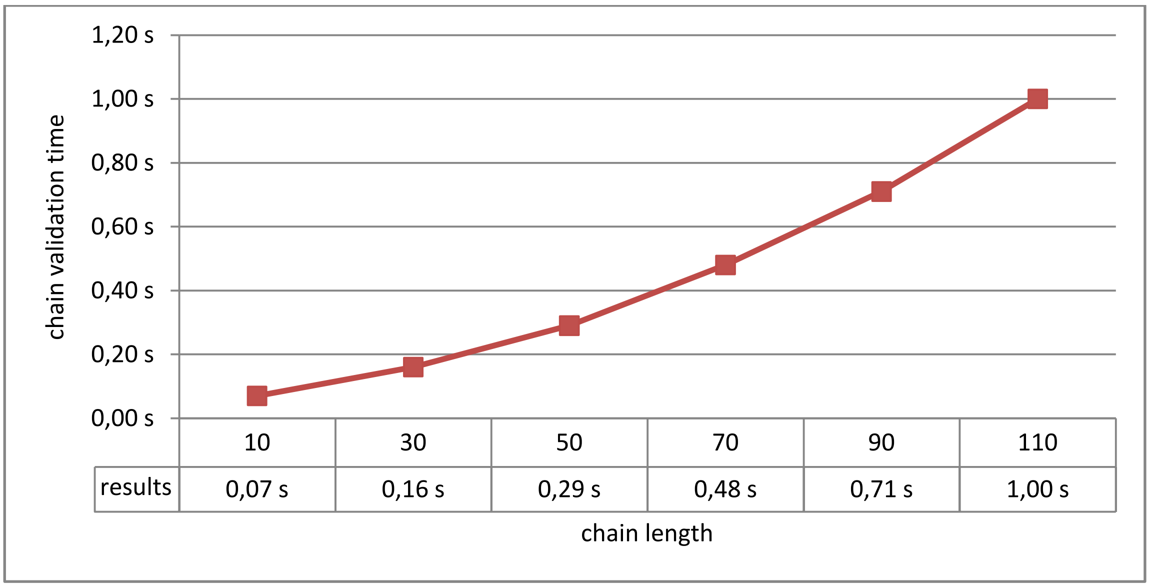
Finally, the cost of implementing the Public Key Infrastructure needs to be taken into account. It is not a case if the required PKI is already present and all the users are provided with their permanent credentials. If not, the value of exchanged data, existing threats and potential losses have to be considered and compared with the required investment.

6.6.4 GSI Authenticator analysis

In order to check, which parts of GSI Authenticator have the highest influence on the authentication time, two additional tests were performed. The tests won't be useful in taking security-related decisions. They are rather performed from the curiosity – to check the suppositions related with the complexity of cryptographic algorithms. Some interesting information can be seen by the way.

First, the key creation and chain validation time were estimated. The test was performed 10 times for chain length from 10 to 110 (step 20). For each length the two extreme results were omitted and the average of the remaining was taken. Finally, the average time of one key creation was taken. The average time is 0,52 s

The chain validation time is presented on the chart in Figure 42:



	10	30	50	70	90	110
results	0,07 s	0,16 s	0,29 s	0,48 s	0,71 s	1,00 s

Figure 42. Chain validation time depending on chain length

Afterwards, a detailed test of particular GSI Authenticator elements was performed. The analyzed operations together with the results are presented in Table 5 and in Figure 43. The test was performed five times. Presented operations were performed iteratively multiple times in each test; the amount of iterations was increased and is presented in columns headers. The obtained results are normalized (summarized times of each operation's multiple execution divided by the amount of iterations) and presented in milliseconds. In last columns, the average time of all test executions are provided together with the percentage usage of each operation in the overall execution process.

Table 5. Times of execution [milis] of particular GSI Authenticator elements together with average time and percentage usage in overall authentication process

	25	50	75	100	200	average	%
Total time	572,8	614,7	671,6	622,4	629,6	622,2	
Credential read	2,8	0,9	1,0	0,9	0,9	1,3	**0,2 %**
Key generation	518,4	560,0	617,2	568,7	577,2	568,3	**91,3 %**
Proxy creation	4,2	4,0	3,9	3,9	3,7	3,9	**0,6 %**
Challenge decryption (twice)	0,8	0,8	0,8	0,9	0,8	0,8	**0,1 %**
Challenge encryption (twice)	13,7	13,8	13,6	13,8	13,7	13,7	**2,2 %**
Validation	32,9	35,2	35,0	34,1	33,1	34,1	**5,5 %**

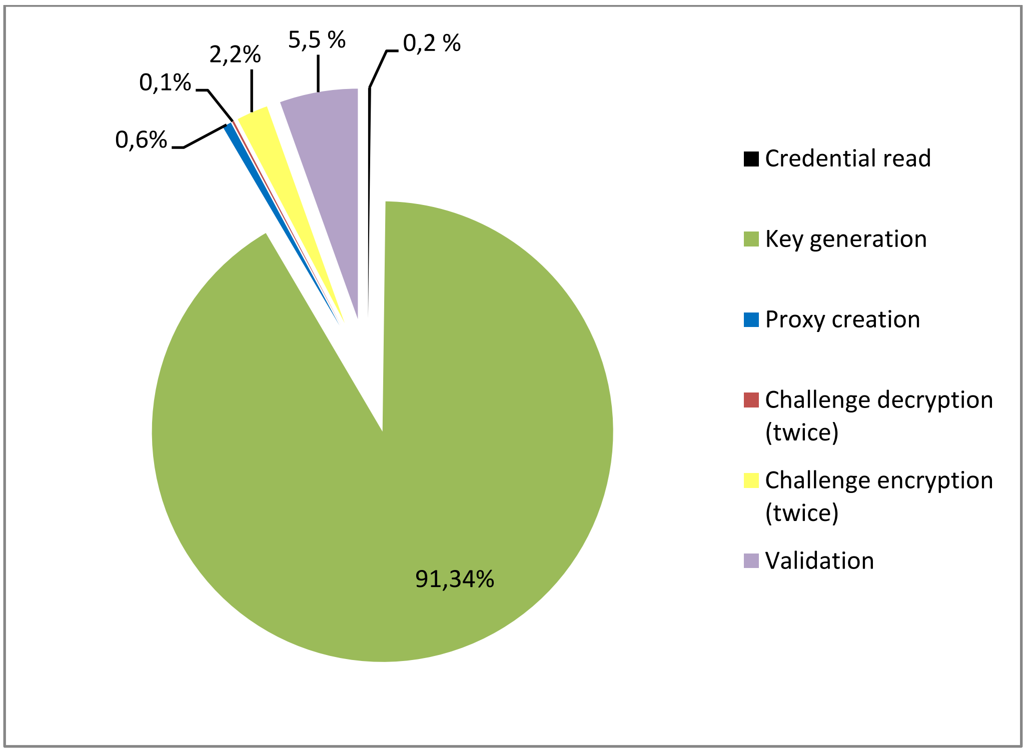


Figure 43. Percentage usage of particular GSI Authenticator elements in overall authentication process

Conclusions:

The chain validation time is rather insignificant for short chains. It grows almost linearly with the enlargement of the chain.

Relatively time-consuming is the key generation procedure, which needs about half second for each key. In comparison to other elements that are used in the GSI Authenticator, it dominates the authentication time.

Noticeable is the difference between the encryption and decryption time. It comes from the characteristic of asymmetric cryptography – data should be easily decrypted while using the proper key and practically not decryptable without it. To achieve it, time-consuming key creation and data encryption algorithms have to be used.

6.7 Summary

The chapter presented the tests that were performed in order to verify the usefulness and quality of the generated authenticator. Its knowledge should also allow readers to easily build the distribution, prepare the environment and run the provided examples.

Chapter 7. Conclusions and future work

In this chapter the accomplished work will be summarized by a short description of achieved goals and by suggestion of the future steps of H2O development.

7.1 Achieved goals

The main goal was to create an H2O-applicable authenticator based on PKI and X.509 certificates that will be compliant with GSI and provide delegation based on proxy certificates. This goal has been successfully achieved. The authenticator was verified and added to the distribution for further development. All the sub-goals identified in chapter 1 were accomplished:

- **Identification and analysis of security architecture and shortages in H2O**

First two chapters provide a comprehensive analysis of current state of security mechanisms in H2O with identification of missing features that were to be challenged by the thesis. The results of the analysis were described in section 2.2.

- **Overview of available solutions for H2O security enhancements**

In chapter 3 some modern security technologies used for authentication were presented. These are GSI, My Proxy and Needham-Schroeder protocol. They became the cornerstone of a future concept of the developed authenticator.

- **Concept and development of new security system for H2O**

The analysis of possible solutions led to a concept of created authenticator. Together with detailed design and implementation it was described in chapters 4 and 5. The added feature significantly increased the usability of the system, which becomes ready to cooperate with current standards. A few existing bugs were corrected as well.

- **Proving the correctness and usefulness of the created solution**

Chapter 6 provides a comprehensive validation and usage description of the GSI Authenticator. First usage examples for both H2O and MOCCA were created. The usefulness and robustness of the solution were proved by several tests as well as detailed threat analysis – of the overall system and of the authenticator itself. The trade-off between performance and security level was described and possible usages were identified.

- **Build, configuration and usage description**

Together with the usage examples presented in chapter 6, an additional description of actions and steps required to build, configure and use H2O and MOCCA is presented in sections 1 and 2 of the chapter and in Appendix C. and will hopefully help further developers of the system.

- **Identification of future work**

This issue is presented in the subsequent section.

Based on the thesis, a publication is being prepared by Bubak M., Dyrda M., Malawski M. and Naqvi S. The Table of Contents of the publication is presented in Appendix D.

7.2 Future work

GSI Authenticator is the next but not the last step in the process of developing H2O in order to provide an architecture that would be both secure and convenient from the point of view of large distributed grid systems. Therefore some future work was identified that may be considered by future developers:

1. **Delegation of trust anchors**

During the work it was identified that the H2O server authentication mechanism of trust managers does not provide delegation of client's trust anchors, thus disabling the pluglets to fully exploit the power of credentials delegation.

2. CRL update and the Online Certificate Status Protocol (OCSP) for certificate revocation verification

The current architecture provides revocation verification based on CRL lists, which are downloaded from specified locations during a start of the kernel. Automatic updates, based on the dates of next publications, provided in the CRL files, should be added.

OCSP was created as an alternative to Certificate Revocation Lists and is described in RFC 2560 [25]. Because of getting more popularity, its usage could be added to the Validator. It can be very straightforward because of the Open GRid Ocsp (ORGO) project [39]

3. MyProxy for credentials storage

The next step which emerges from the trends in the domain of security may be the addition of MyProxy server attendance to the suggested Public Key Infrastructure, and at the same time to the GSI Authenticator. Instead of performing the delegation between the communicating peers, it would allow to get the delegated credentials from MyProxy server, making users independent of their permanent credentials location. Also the usage of MyProxy CA may be considered.

4. More sophisticated authentication mechanisms

The thesis was focused on authentication, bringing up the issues of H2O authorization in a superficial way. However, the next step towards the advancement of the system should be creation of more sophisticated authorization mechanisms that would allow to get rid of local configuration of all the kernels using *Users.xml* files in favour of some centralized management of the overall distributed system. Some research in this field has already been started.

[39] http://dev.globus.org/wiki/Incubator/OGRO

References

1. **M. Malawski, D. Kurzyniec, V. Sunderam.** MOCCA – towards a distributed CCA framework for metacomputing. *In Proceedings of 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Joint Workshop on High-Performance Grid Computing & High-Level Parallel Programming Models - HIPS-HPGC, April 4-8, 2005, Denver, Colorado, USA.* [Online] http://mathcs.emory.edu/dcl/h2o/papers/h2o_hips05.pdf.

2. **Malawski, M.** Component-based Grid Environment for Programming Scientific Applications. *Presentation at Cracow Grid Seminar.* [Online] www.eu-crossgrid.org/Seminars-INP/MOCCA-Seminarium-Nov06.ppt.

3. **V. Sunderam, D. Kurzyniec.** Lightweight self-organizing frameworks for metacomputing. *In The 11th International Symposium on High Performance Distributed Computing (HPDC-11 '02), Edinburgh, Scotland, July 2002.* [Online] http://www.dcl.mathcs.emory.edu/downloads/h2o/papers/h2o_hpdc02.pdf.

4. **Distributed Computing Laboratory, Emory University.** H2O. *Java-based, secure, scalable, stateless, lightweight, flexible, component-hosting distributed application platform.* [Online] January 10, 2008. http://dcl.mathcs.emory.edu/h2o.

5. **IETF.** RFC 2246. *The TLS Protocol.* [Online] http://tools.ietf.org/html/rfc2246.

6. **RSA Laboratories.** RSA Algorithm. [Online] http://www.rsa.com/rsalabs/node.asp?id=2146.

7. Red Hat Certificate System. *Administrator's Guide.* [Online] http://www.redhat.com/docs/manuals/cert-system/admin/7.1/app_dn.html.

8. UIQ developer portal. *Certificate Validation in PKIX.* [Online] http://developer.uiq.com/devlib/uiq_30/SDKDocumentation/sdl/guide/N1010A/CertMan/CertValidation.html.

9. **D. Kurzyniec, T. Wrzosek, V. Sunderam, A. Slominski.** RMIX: A multiprotocol RMI framework for java. *In Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03), Nice, France, April 2003.* [Online] http://www.dcl.mathcs.emory.edu/downloads/rmix/papers/rmix.pdf.

10. **T.Ampula, D. Drzewiecki, et al.** Harness and H2O. *Alternative approaches to metacomputing.* [Online] www.cyfronet.krakow.pl/cgw03/presentations/1.ppt.

11. The Globus Authorization Framework with PDPs. [Online] http://globus.org/toolkit/docs/4.0/security/authzframe/developer-index.html.

12. JavaTM Authentication and Authorization Service (JAAS). *Reference Guide.* [Online] http://java.sun.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html.

13. **IETF.** RFC 3820. *Internet X.509 Public Key Infrastructure (PKI) Proxy Certificate Profile.* [Online] http://www.ietf.org/rfc/rfc3820.

14. **T. Bujok, M. Dyrda.** Grid Security Infrastructure Management on the basis of Shibboleth and MyProxy Systems. [Online] http://virolab1.cyfronet.pl/student/doku.php?id=wiki:0:grid_security_infrastructure_management_on_the_basis_of_shibboleth_and_myproxy_systems.

15. GT 4.0: Security. [Online] http://globus.org/toolkit/docs/4.0/security/.

16. **V. Welch, I. Foster et al.** X.509 Proxy Certificates for Dynamic Delegation. [Online] middleware.internet2.edu/pki04/proceedings/proxy_certs.pdf.

17. **J. Basney.** MyProxy. *A Multi-Purpose Grid Authentication Service.* [Online] www.ncsa.uiuc.edu/~jbasney/MyProxy-WCGA06.ppt.

18. **J. Novotny, S. Tuecke, V. Welch.** An Online Credential Repository for the Grid: MyProxy. [Online] www.globus.org/alliance/publications/papers/myproxy.pdf.

19. **National Center for Supercomputing Applications (NCSA), University of Illinois.** MyProxy. *Credential Management Service.* [Online] http://grid.ncsa.uiuc.edu/myproxy/.

20. **R. Needham, M. Schroeder.** Needham-Schroeder Public Key protocol. [Online] December 4, 1978. http://www.lsv.ens-cachan.fr/spore/nspk.pdf.

21. **G. Lowe.** Lowe's fixed version of Needham-Schroder Public Key. [Online] 1995/2002. http://www.lsv.ens-cachan.fr/spore/nspkLowe.pdf.

22. Java ™ Cryptography Architecture (JCA). *Reference Guide.* [Online] http://java.sun.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html.

23. **G. von Laszewski, I. Foster, J. Gawor, P. Lane.** A Java Commodity Grid Toolkit. *ACM 2000 Java Grande Conference.* [Online] June 28, 2001. http://www.globus.org/alliance/publications/papers/vonLaszewski--cog-cpe-final.pdf.

24. **signet : Centrum Certyfikacji.** Wprowadzenie do PKI. [Online] http://www.signet.pl/pomoc/pki.html.

25. **IETF.** RFC 2560. *X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP.* [Online] http://tools.ietf.org/html/rfc2560.

26. —. RFC 3280. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile.* [Online] http://tools.ietf.org/html/rfc3280.

27. **D. Hook.** X.509 Public Key Certificate and Certification Request Generation. [Online] http://www.bouncycastle.org/wiki/display/JA1/X.509+Public+Key+Certificate+and+Certification+Request+Generation.

28. **The Globus Security Team.** Globus Toolkit Version 4 Grid Security Infrastructure. *A Standards Perspective.* [Online] September 12, 2005. http://www.globus.org/toolkit/docs/4.0/security/GT4-GSI-Overview.pdf.

29. **IETF.** Public-Key Infrastructure (X.509) (pkix). [Online] http://www.ietf.org/html.charters/pkix-charter.html.

30. **ITU-T.** X Series recommendations. *Data networks, open system communications and security.* [Online] http://www.itu.int/rec/T-REC-X/en.

31. Proxy Certificates Types. [Online] http://dev.globus.org/wiki/Security/ProxyCertTypes.

32. **RSA Laboratories.** Public-Key Cryptography Standards (PKCS). [Online] www.rsa.com/rsalabs/pkcs/.

33. Wikipedia. *the free encyclopedia.* [Online] http://www.wikipedia.org/.

Appendix A : Standards and formats related to cryptography

A. 1. What you should know about PEM, DER, PKCS, …[40]

The motivation for providing successive encoding standards was the difficulty in transferring data between different systems. Several relevant standards are now going to be presented:

- Notation: **ASN.1**

Abstract Syntax Notation One (ASN.1) is an ISO / ITU-T standard and notation that allows to define data messages, which can be exchanged between communicating systems regardless of the underlying machine-specific encoding. It describes data structures for representing, encoding, transmitting and decoding data.

- Standards **: PKCS**

The **Public Key Cryptography Standards** (PKCS) were created by RSA Security in order to standardize the format of objects used during public key operations . They became part of many formal and de facto standards, including PKIX and SSL.

Some of the most important standards from the point of view of this thesis are:

  - **PKCS #1**

PKCS #1 is a RSA Cryptography Standard that specifies the mathematical properties and format (ASN.1 syntax) of RSA public and private keys together with basic algorithms for performing cryptographic operations using those keys, like encryption and decryption or producing and verifying signatures.

  - **PKCS #8**

[40] The description is based on [30] [29] [32]

This standard describes syntax for private-key information: the private keys themselves and additional attributes for public key algorithms. The standard also describes an abstract syntax for encrypted private keys.

- **Transfer syntax** (encoding rules) :

Transfer syntax are rules for encoding abstract information (e.g. data structures described in ASN.1) info a concrete data stream. The most commonly used formats will now be described.

The first were the **Basic Encoding Rules** (BER), defined already as a part of the ASN.1 standard. This is an example of TLV (type-length-value) encoding, where the data elements are encoded as a type identifier, a length description, the actual data elements and the end marker if needed. One of the advantage of this format is a possibility to decode some information from an incomplete stream. BER however does not provide an unique representation of data that means the same information can be presented in multiple serialized ways. For example, there are 255 ways of saving the boolean value of true. The unique representation is however required while using digital signatures of data, e.g. for X.509 certificates.

Because of that, the subset of the BER was selected, which restricts the possibilities of representing any ASN.1 value to a single option. The obtained transfer syntax is called the **Distinguished Encoding Rules (DER). Still, DER encoding is a valid BER encoding.**

Another version of BER subset are the **Canonical Encoding Rules** (CER). The basic difference between DER and CER is that DER uses definitive length form, by always providing a leading length information, whereas CER may use the end-of-contents octet , providing the indefinite length in some cases.

CER, DER and BER are binary formats that encode data in octet (groups of eight bits) sequences. In order to get a 'printable' encoding, the **Privacy-Enhanced Mail** (PEM) format can be used. It is the Base64 encoding of the DER format. Base64 is an encoding that allows to encode a sequence of octets as a sequence of printable ASCII characters.

PEM files may contains certificate(s) and/or private key(s) enclosed between appropriate header and footer lines. For certificates, they have the following form:

```
-----BEGIN CERTIFICATE-----
-----END CERTIFICATE-----
```

The format of header and footer for private key depends on the standard of the key. The PEM header of PKCS#1 private key is:

```
-----BEGIN RSA PRIVATE KEY-----
-----END RSA PRIVATE KEY-----
```

The PEM header of a PKCS#8 encrypted private key is :

```
-----BEGIN ENCRYPTED PRIVATE KEY-----
 -----END ENCRYPTED PRIVATE KEY-----
```

whereas the unencrypted form uses:

```
 -----BEGIN PRIVATE KEY-----
 -----END PRIVATE KEY-----
```

- **PKCS #12**

Another commonly used transfer syntax is described in the PKCS#12 standard. It describes encoding of personal identity information, i.a. certificates and private keys. It is considered as one of the most complex cryptographic protocols, still it is the only standard that enables storing private keys together with certificates in a single encrypted file.

- **File extensions**

In general, the .PEM files are mostly used in the Unix world, the .DER files in the Java world and the .P12 (PKCS12) files in the Microsoft world.

There may be some ambiguity with the .CER extension, which should point to CER-encoded certificates but is used by Microsoft both for both DER and Base64 certificate files.

A. 2. Proxy credential file format

The proxy file acceptable by Globus has a specific format: it contains the most recent proxy certificate, followed by the corresponding private key, followed by the chain of certificates, starting from the previous proxy up to the end entity certificate. Individual blocks are enclosed by specific header and footer lines.

The example (truncated) file looks as follows:

```
-----BEGIN CERTIFICATE-----
MIICcDCCAhqgAwIBAgIEU6SeXjANBgkqhkiG9w0BAQQFADCBojENMAsGA1UEChME
[…]
-----END CERTIFICATE-----
-----BEGIN RSA PRIVATE KEY-----
MIIBOQIBAAJBAMc0n9W1E1KjK6saavXXZ/QhLjJ/TK40uW29l/wduSrHWCu1e5Kr
[…]
-----END RSA PRIVATE KEY-----
-----BEGIN CERTIFICATE-----
[…]
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
[…]
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
[…]
-----END CERTIFICATE-----
```

Both certificates and private key are PEM-encoded.

A. 3. Proxy certificate extensions

The RFC requires all the proxy certificates to include the newly introduced Proxy Certificate Information (ProxyCertInfo) extension and the extension must be critical. It indicates that a certificate is a proxy and may specify some restrictions, placed on its usage.

The fields contained by the extension are:

- pCPathLenConstraint – specifies the maximum number of proxy certificates in the chain that may follow the one being checked.

  It is distinct from keyUsage, because it concerns only proxy certificates, whereas keyUsage concerns only non-proxy certificates

  Value 0 means that the certificate must not be used to sign a proxy. If the field is missing, unlimited path length is allowed. End entity certificates have unlimited maximum proxy path length.

- proxyPolicy – specifies a policy to use for the purpose of authorization. It consists of two parts, one indicating the policy language and other expressing the policy itself.

A. 4. Proxy certificate types[41] :

During the advancement of the proxy conception, different standards emerged, until finally the official standard in form of RFC 3820 appeared. However, the older versions can still be encountered, and what is more, Globus provides a backward compatibility, to serve them. Therefore it is good to familiarize with the existing formats:

- **Legacy Proxy Certificate**

First of them, introduced already in GT 2.0, was specified before the publication of the RFC. It is recognized by the lack of ProxyCertInfo extension and the use of *CN=proxy* or *CN=limited proxy* DN components.

- **Proxy Draft Proxy Certificates**

This format, also called the GSI3 Proxy Certificates (because of its first appearance in GT3) is very similar to the RFC 3820 Proxy Certificates. The only difference is the non-standard OID used to identify the ProxyCertInfo extension.

- **RFC 3820 Proxy Certificates**

RFC 3820 Proxy Certificates are proxy certificates that fully conform to RFC 3820 [13].

Proxy Certificate Compatibility

- In GT 4.2.x, it is expected that RFC 3820 Proxies will be generated by *grid-proxy-init* by default.

- GT 4.x accepts all three types of proxy certificates listed above and generates Proxy Draft Proxy Certificates by default.
  The RFC 3820 Proxies can be generated using *grid-proxy-init -rfc*

- GT 3.x accepts Proxy Draft and Legacy proxy certificates.

- GT 2.x accepts Legacy proxy certificates.

[41] The description is based on [31]

A. 5. Certificate chain validation [42]

In order to validate a certificate, its issuer has to be verified. As it was presented in the chapter 2.1.3, in PKI a certificate path starts with the End Entity certificate and proceeds through a number of intermediate certificates up to a root certificate, which is typically a self-signed CA certificate. At least one of them has to be trusted by the verifier.

The validation is recursive as well. At the beginning the End Entity certificate is being checked. In order to verify the signature, the signer's certificate has to be used. This one may need to be verified as well – and this process is repeated until some trusted certificate is reached.

A standardized path validation algorithm for X.509 certificates, given a certificate path, is defined in RFC 3280 [26].

Validator input

- The certificate path to verify (End Entity and intermediate certificates)
- Trusted roots certificates
- Current date/time
- Acceptable policies – i.a. to specify the application, for which the verified certificate will be used

Chain construction

A user may not always possess a complete path from a trusted CA to the End Entity. Some solutions, commonly described as Path Discovery processes, are provided but they are not going to be covered in this thesis.

Validation algorithm

The following steps are performed for each certificate in the path, starting from the trust anchor. If any check fails on any certificate, the algorithm terminates and path validation fails:

- Signature Verification & Name Chaining
  Each certificate, except the self-signed root certificate, must be signed by the certificate above in the chain. The issuer name of signed certificate must match the subject name of the signer. The self-signed certificate subject and issuer names must be equal.

[42] The description is based on [8]

- Validity Period Checking
  The time of validation must lie within the validity period of (each) certificate

- Extensions Processing
  The validation algorithm must process at least the critical extensions contained by the certificate. The restrictions, provided by the extensions (i.a. KeyUsage), have to be taken into account.

- Policies verification
  Policy constraints are checked, to ensure that any explicit policy requirements are not violated

- Revocation Checking
  If CRL locations are provided, certificates are checked against being revoked.

A. 6. Proxy chain validation [43]

Validation of a certificate chain has two distinct phases. First validation of the certificate chain from the End User non-proxy certificate up to the trusted anchor based on RFC 3280 occurs, as described above. Afterwards the validation of the proxy certificates, from the already verified end-user certificate down to the most recent proxy takes place. This process is, in turn, based on RFC 3820. Some of the steps are similar; the main steps are:

- Signature Verification & Name Chaining
  As in case of plain certificates, but each Proxy Certificate must have a subject name derived from the subject name.

- Validity Period Checking

- Extensions Processing
  Proxy certificates may contain additional required proxy extension, which has to be processed by the Validator. The extension was described in point 3 of this Appendix

Current RFC contains no description of proxy revocation, some mechanisms are already provided though.

[43] The description is based on [16]

Appendix B: Encountered problems with initial implementation without CoG JGlobus package – code snippets

For parsing and writing proxy files in the initial configuration, which was performed without the usage of Globus libraries, as described in section 5.9, the BouncyCastle and Sun security tools were used. This appendix contains some code examples that might be useful for future developers of proxy-related elements.

For reading certificate and key files the org.bouncycastle.openssl.PEMReader can be used:

```
        PEMReader r = new PEMReader(new FileReader(certfile));
        X509Certificate cert = (X509Certificate)r.readObject();
```

The encrypted key file might be used by using the PasswordFinder parameter in the PEMReader (`org.bouncycastle.openssl)` constructor:

```
        PEMReader r = new PEMReader(new FileReader(keyfile), pwdFinder);
```

This parameter has to be a class implementing the org.bouncycastle.openssl.PasswordFinder interface, which contains only one method:

```
        public char[] getPassword();
```

In order to read the proxy file (not standard version with PKCS8 encoded key only), it was first parsed by using the header and footer lines of certificates and keys. Then the string representations were decoded, using sun.misc.BASE64Decoder

```
        byte[] bytes = new BASE64Decoder().decodeBuffer(string);
```

Afterwards, java.security.cert.CertificateFactory and java.security.KeyFactory were accordingly used.

```
        CertificateFactory cf = CertificateFactory.getInstance("X.509");
        DataInputStream dis = new DataInputStream(new
ByteArrayInputStream(bytes));
        X509Certificate cert = (X509Certificate) cf.generateCertificate(dis);
```

The key still had to be translated into so called ‘key material’, in form of a java.security.spec.PKCS8EncodedKeySpec object:

```
        PKCS8EncodedKeySpec privSpec = new PKCS8EncodedKeySpec(bytes);
        KeyFactory fact = KeyFactory.getInstance("RSA");
        RSAPrivateKey key = (RSAPrivateKey) fact.generatePrivate(privSpec);
```

In order to write the proxy file, the sun.misc.Base64Encoder is used to default encoded versions of certificate and key respectively:

```
                return new BASE64Encoder().encode(cert.getEncoded());
                return new BASE64Encoder().encode(privKey.getEncoded());
```

For creating proxy certificate the BouncyCastle API [27] was used.

Appendix C : Configuration

C. 1. Building the distribution

The following tools were used to build the H2O distribution (using other versions may cause difficulties):

- Java 1.5
- ant 1.6.5
- junit 4.4 in classpath

A keystore with credentials that will be used in order to sign the distribution jar files, has to be provided for the build to complete successfully. The keystore file should be placed in *`${user.home}/priv/dclPrivKeyStore`* along with the *`${user.home}/priv/dclStorePass`* file containing password to the keystore.

An example keystore that can be used is provided in the *misc* directory of the H2O distribution.

For the GSI Authenticator to work, the CoG libraries had to be added to the distribution. The description of the required changes that were made, is placed in the section 8 of this Appendix.

C. 2. CoG package configuration

In order to use the CoG Kit package, it has to be properly configured, by writing a cog.properties file in `{user.home}/.globus` directory. The file contains the following information:

- The location of user's certificate and private key, which are usually stored in the mentioned .globus directory and called usercert.pem and userkey.pem respectively , e.g.:

```
usercert=/home/majk/.globus/usercert.pem
userkey=/home/majk/.globus/userkey.pem
```

The certificate file is public and can be published without any risk, however the private key should be kept secure from unauthorized access, so it is important to set their access rights properly:

```
> ls -al .globus/user*.pem
-r--------   1 majk majk  951 2008-05-07 15:06 userkey.pem
-rw-r--r--   1 majk majk 2030 2008-05-07 15:06 userrequest.pem
```

- The location of the trusted CA certificate files

```
cacert=/etc/grid-security/certificates, /home/majk/.globus/cacert.pem
```

- The location of the proxy file (also the place where to put it upon generation by the tool). The file is usually placed in a temporary directory with a name x509up_u{ uid}:

```
proxy=/tmp/x509up_u1000
```

The file is protected by only local file system permissions, which allow the user's applications to access it without any manual intervention by the user, therefore the maximum allowable permissions for the proxy are restricted (usually to 600):

```
> ls -al /tmp/x509up_u1000
-rw-------  1 majk majk 4729 2008-04-26 09:50 /tmp/x509up_u1000
```

- The system IP address

```
ip=192.168.120.51
```

The cog.properties file can be modified by hand or using the GUI wizard run by executing the cog-jglobus.jar file.

The GSI Authenticator can accept GlobusCredential objects as well as separate certificate chain and private key object. In order to use the authenticator, we need to possess valid credentials that are accepted by the kernel. Example credentials and configuration file are provided in the *misc* directory of the H2O distribution. The

Issuer's certificate is added into the distribution truststore[44] and the Subject CN is added to the *Users.xml* configuration file[45]

The details of the credentials are:

Issuer: C=PL,ST=Some-State,O=TestCA
Subject: C=PL,O=GRID,CN=TestUser
Validity end time (both for CA and user certificate) : ~2035

The password for the private key is : testUserPass

C. 3. H2O truststore configuration

The truststore of the H2O distribution is placed in the `${h2o-dist}/config/security/cacerts` file and secured with password h2o-CA

The truststore may be used both by client to specify trusted kernels (see server authentication) and by kernel to specify clients that are allowed to connect using GSIAuthenticator.

In order to add a new certificate to the truststore, the java keytool utility can be used:

```
{java_home}/bin/keytool -import -keystore cacerts -file <certificate file> -
alias <certificate alias>
```

The TestCA certificate, which is used to issue the TestUser certificate, is already contained by the distribution cacerts file.

C. 4. Users.xml configuration

Users.xml, placed in *`{h2o-dist}/config/security`*, contains the entries required for user authorization, as described in chapter 2.2.5.

In order to give rights to the users authenticated with the certificates, their CN have to be supplied in the file. The default file provided with the distribution contains the entries for the TestUser that give him the rights of Deployer.

[44] See point 3 of this Appendix
[45] See point 4 of this Appendix

The example entry for the TestUser is:

```
        <member type="user" id="CN=TestUser,O=GRID,C=PL"/>
```

in the Deployers group

```
        <user uid="CN=TestUser,O=GRID,C=PL" password=""/>
```

at the end of the configuration file

It is important that the order of the CN elements used by the Validator of GSI Authenticator is reversed comparing to the one presented by the common certificate tools.

C. 5. Permissions in Policy.xml configuration file

Because of the security restraints, several permissions have to be set for using the CoG JGlobus distribution and GSI Authenticator. The permissions are placed in *Policy.xml* file placed in *{h2o-dist}/config/security* directory. A few entries concern the location of configuration and certificate files that are used by the authenticator. While using unusual configuration, they have to be adjusted properly. The permissions entry and the target are:

```
<permission type="java.io.FilePermission" target="…" actions="read"/>
```

`crl${/}*`	The location of downloaded crl files
`${h2o.home}${/}lib${/}cog${/}cryptix32.jar`	The location of the cryptix provider in CoG lib directory
`${user.home}${/}.globus`	The location of the user’s .globus directory
`${user.home}${/}.globus${/}*`	This entry specifies all the files in user’s .globus directory, e.g. cog.properties file or trusted ca certificates

The last entry is most important. GSI Authenticator requires permissions to access all the cacert files that are specified in the cog.properties file. If the files are not placed in user’s .globus directory, additional permissions have to be added in the Policy file.

Some other permissions that were added are:

- The permissions for using security providers used by CoG:

```
<permission type="java.security.SecurityPermission" target="…"
actions="accept"/>
```

Where targets are:

```
putProviderProperty.ClaymoreProvider
insertProvider.ClaymoreProvider
removeProvider.BC
putProviderProperty.BC
insertProvider.BC
putProviderProperty.Cryptix
```

Other permissions:

```
<permission type="java.lang.RuntimePermission"
      target="accessClassInPackage.sun.misc"/>
<permission type="edu.emory.mathcs.rmix.RmixRuntimePermission"
      target="accessClientSocketFactory"/>
<permission type="java.io.FilePermission" target="/dev/urandom"
      actions="read"/>
```

C. 6. Revocation configuration

The URLs of the Certificate Revocation Lists provided by CAs are to be provided in the *KernelConfig.xml* file. The entries should be added in the <Security> section:

```
   <CRLLocations>
      <CRLLocationEntry location="<crl_url>"/>
   </CRLLocations>
```

C. 7. Server authentication configuration

As it was said, by default H2O kernel generates a self-signed certificate to identify itself to clients. This behavior can be overridden by specifying custom keystore with X509 credentials to use in the *KernelConfig.xml* file:

- in the <KeyStores> section a new KeyStore is to be added, e.g.:

```
       <KeyStore id="server" location="security/server.jks"
            passwordSource="here:server"/>
```

- The keystore id is used to identify the keystore in the Identity entry in the <Security> section:

```
       <Identity keyStore="server" alias="server"
passwordSource="here:server"/>
```

Obviously valid keystore location and certificate alias have to be provided, as well as the password – in form of a direct passphrase entry (here:<passphrase>) or a path to a file that contains it.

- After configuring the kernel, it can be authenticated by using the H2O TRUST_CERTIFIED trust manager for client context:

```
clientCxt = H2OClient.newInstance(wallet, H2O.TRUST_CERTIFIED);
```

For the purpose of testing, example credentials were created. However in order for the certificate to be valid, the CN of the certificate must be consistent with the hostname of the server – therefore providing them in the distribution is not reasonable. For creating such credentials, the Portecle[46] application might be useful.

C. 8. H2O build files configuration

The required libraries for creating the GSI Authenticator are taken from the cog-jglobus package[47]. The following files are used:

cog-jglobus.jar	cryptix.jar	jgss.jar
cog-jobmanager.jar	cryptix32.jar	junit.jar
cog-url.jar	cryptix-asn1.jar	log4j-1.2.13.jar
commons-logging-1.1.jar	jce-jdk13-131.jar	puretls.jar

The package contains the following providers:

- Cryptix
- BouncyCastleProvider
- ClaymoreProvider

Since the files are used by different H2O subprojects, the following changes in H2O configuration were performed:

- util subproject:
  - cog-globus.jar added in the lib/cog directory and proper entry added into classpath in build-jbexport.xml file
  - copying of lib/cog directory into h2o directory added in build.xml
- h2o subproject:
  - copying of lib/cog directory into h2o-dist directory added in build.xml
- h2o-client subproject:
  - cog-jglobus.jar added into classpath in build-jbexport.xml file
- h2o-server subproject:

[46] http://portecle.sourceforge.net/
[47] http://dev.globus.org/wiki/CoG_jglobus

    - cog-jglobus.jar added into classpath h2o-kernel and h2o-kernel.bat execution scripts
    - setting permissions in Policy file[48]
- h2o-example-tutorial subproject:
    - cog-jglobus.jar added into classpath in build/build.xml file and in the execution scripts of the GSI example (step11 and step11.bat)
- h2o-test subproject:
    - cog-jglobus.jar added into classpath in build-jbexport.xml file and in the build/build.xml file

[48] Described in point 5 of this Appendix

Integration of GSI with MOCCA component environment

M. Dyrda[1], M. Malawski[1], S. Naqvi[3], M. Bubak[1,2]

[1] Institute of Computer Science, AGH, al. Mickiewicza 30, 30-059, Kraków, Poland
[2] Academic Computer Center CYFRONET, ul. Nawojki 11, 30-950 Kraków, Poland
[3] CETIC, Rue des Freres Wright 29/3, B-6041 Charleroi, Belgium
email: `[bubak,malawski]@agh.edu.pl`
phone: (+48 12) 617 3964, *fax:* (+48 12) 338 054

Abstract

The subject of this paper is a detailed analysis and development of security in grid component systems on the example of MOCCA, a CCA-compliant framework build over H2O distributed computing platform. The goal of this work is to provide H2O with an authentication mechanism that will be both secure and compliant with solutions commonly used in grid systems nowadays. The proposed authenticator is based on asymmetric cryptography with additional features provided by Globus Security Infrastructure - proxy certificates that are used for Single Sign-On and delegation. The created authenticator as well as the overall system were brought under tests and threat analysis, which proved their safeness and usability.

Table of Contents