Chair for Network- and Data Security
Horst Görtz Institute for IT Security
Ruhr-University Bochum
Supervisors: Prof. Jörg Schwenk

Flexible authentication for stateless web services

Student:	DI Christian Müller; MSc (GIS)
Studies:	Applied IT Security
Semester:	4
Student ID:	none
Birth date:	18.11.1965
Address:	Draugasse 7/266
	1210 Vienna, Austria
Phone-No.:	+43 676 628 31 95
E-Mail:	christian.mueller@nvoe.at

10.6.2012

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich meine Masterarbeit mit dem Thema

Flexible authentication for stateless web services

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Wien, den 10. Juni 2012

Acknowledgements

The author has to thank his wife Henriette for her endless patience during the last years, especially for managing our 4 year old foster child Tscharto and accepting my working during the weekends.

My older son Lukas also respected my eager work and occupies himself with other topics. My stepdaughter Alex does already her own study with high grades and does not need any support fortunately.

Thanks to Google for sponsoring the start of the project during Google Summer of Code 2011. Andrea Aime, an open source developer with excellent skills was acting as my mentor during this Summer of Code. Thank you again.

Very special thanks for Justin Deoliveira for joining the practical work at a point in time where the author was powered out. Justin is an excellent open source developer and pushed to continue the work done during the Google Summer of Code.

In the meantime, the number of mails concerning security work between Justin and the author is about 400. Justin also prepares the way for getting a paid contract with OpenGeo[13] to finish the practical work.

Last not least, Justin as a native English speaker reviewed this master thesis and checked the technical correctness. Thanks again, Justin.

Abstract

Stateless Web Services have the advantage not to consume server resources between different requests. Another advantage is that clustering is working out of the box. On the contrary, for stateful services, server memory has to be reserved between requests from the same client.

If a client requests a protected resource, authentication is required. The authentication procedure may be costly regarding performance and it may be impossible to authenticate each individual request. Stateful services do not have this problem because the server can store the authentication result in the memory reserved for the client session.

This thesis presents a solution how to cache authentication results for different authentication mechanisms, enabling clustering of the authentication result and how to deal with SSO (Single Sign On).

Table of Contents

1 INTRODUCTION **10**

2 TECHNICAL BACKGROUND **12**

2.1 Cookies 12

2.2 HTTP Sessions 13

2.3 Types of HTTP Services 13

2.4 Principal and Credentials 14

2.5 RBAC 14

2.6 Clustering Scenarios 14

2.6.1 Clustering with stateful services *14*

2.6.2 Clustering with stateless services *15*

2.7 Passwords 15

2.7.1 Password Policy *15*

2.7.2 Password Generator *15*

2.7.3 Password Digesting *16*

2.7.3.1 The Salt 16

2.7.3.2 Iteration Count 17

2.7.4 Alternatives to Password Digesting *18*

2.7.5 PBE (Password Based Encryption) *18*

2.8 Needed Java Concepts 19

2.8.1 Passwords and Java heap dumps *19*

2.8.2 Java and unrestricted key length *19*

2.8.3 Key Store *20*

2.8.3.1 Concepts 20

2.8.3.2 Key Store Password 21

2.8.3.3 Protecting the key store password from foreign code 22

2.8.4 Java Servlets and Filters *22*

2.9 SAML 25

3 CONCEPT **29**

3.1 Encryption of URL Parameters 29

3.2 Caching the authentication token 29

3.3 Security Filter Chain 30

3.3.1 Channel Filter *33*

3.3.2 Security Context Persistence Filter *33*

3.3.3 Authentication Filters *33*

3.3.4 Remember me filter *34*

3.3.5 Anonymous authentication filter *34*

3.3.6 Exception Translation Filter *34*

3.3.7 Filter Security Interceptor *34*

3.4 Authentication Filters 35

3.4.1 Authentication Token as Thread Local Variable *35*

3.4.2 Basic Authentication Filter *35*

3.4.2.1 Entry Point 35

3.4.2.2 Description 36

3.4.2.3 Cache key derivation 37

3.4.3 Digest Authentication Filter *37*

3.4.3.1 Entry Point 38

3.4.3.1.1 Entry Point for RFC 2069 38

3.4.3.1.2 Entry Point for RFC 2617 38
3.4.3.2 Description 39
3.4.3.2.1 Description for RFC 2069 39
3.4.3.2.2 Description for RFC 2617 39
3.4.3.3 Cache Key derivation 40
3.4.4 Form Based Authentication 40
3.4.4.1 Entry Point 40
3.4.4.2 Description 41
3.4.4.3 Cache key derivation 41
3.4.4.4 Form Logout Filter 41
3.4.5 Proxy Authentication 41
3.4.5.1 Entry Point 42
3.4.5.2 Cache key derivation 42
3.4.5.3 J2EE proxy Authentication Filter 42
3.4.5.4 HTTP Request header proxy authentication filter 42
3.4.5.5 X509 Certificate Proxy Authentication Filter 43
3.4.6 Remember Me Authentication Filter 43
3.4.6.1 Entry Point 43
3.4.6.2 Description 43
3.4.6.3 Cache Key Derivation 44
3.4.7 Anonymous Authentication Filter 44
3.4.8 Cascaded Authentication Filters 44
3.5 CAS (Central Authentication Service) 46
3.5.1 SSO 46
3.5.1.1 Standard Login 46
3.5.1.2 Protocol Elements 49
3.5.1.3 Integration into Spring Security 51
3.5.1.4 Standard Login including a PGT (Proxy Granting Ticket) 52
3.5.2 Single Logout 54
3.5.2.1 Description 54
3.5.2.2 Protocol elements 55
3.5.3 Stateless services filter 55
3.5.3.1 Using Proxy Tickets 55
3.5.3.2 Protocol elements for proxy tickets 56
3.5.3.3 Entry Point 56
3.5.3.4 Cache key derivation 56
3.6 Stateless SAML Filter 58
4 IMPLEMENTATION 59
4.1 RBAC 59
4.2 Passwords 60
4.3 Encryption 60
4.4 Key Store 61
4.5 Caching the authentication token 62
4.6 Filters 62
4.7 CAS 63
5 CONCLUSION AND OUTLOOK 64
6 BIBLIOGRAPHY 65

List of Figures

Illustration 1: states of the USA....................11
Illustration 2: Calculating a password digest....................17
Illustration 3: J2EE Container....................23
Illustration 4: Filter chain 24
Illustration 5: Interrupted filter chain....................24
Illustration 6: Basic SAML concepts....................28
Illustration 7: Filter chain proxy....................32
Illustration 8: CAS login form....................48
Illustration 9: CAS collaboration....................51
Illustration 10: CAS Spring Security....................52

List of Tables

Table 1: Official key limitation table....................20
Table 2: Geoserver filter chains....................31

List of Abbreviations and Symbols

Abbreviation	Text
3DES	Triple DES
AES	Advanced Encryption Standard
API	Application Programming Interface
CAS	Central Authentication Service
DES	Data Encryption Standard
GIS	Geographic Information System
GML	Geographic Markup Language
GPL	GNU General Public License
GUI	Graphical User Interface
HTML	Hyper Text Markup Language
HTTP	Hyper Text Transfer Protocol
HTTPS	HTTP Secure
IdP	Identity Provider
IP	Internet Protocol
J2EE	Java 2 Enterprise Edition
JDBC	Java Database Connectivity
JKS	Java Key Store
JCEKS	Java Cryptography Extension Key Store
MD5	Message Digest Algorithm
OGC	Open Geospatial Consortium
PBE	Password Based Encryption
PKI	Public Key Infrastructure
PKCS#12	Public-Key Cryptography Standards #12 Key Store
PGT	Proxy Granting Ticket used by CAS
PT	Proxy Ticket used by CAS
RBAC	Role Based Access Control
RFC	Request for comment
SAML	Security Assertion Markup Language
SHA-?	Family of cryptographic hash functions, ? in range 0 to 3
SOAP	Simple Object Access Protocol
SP	Service Provider
SQL	Structured Query Language
SSL	Secure Socket Layer
SSO	Single sign on
ST	Service ticket used by CAS
TCP	Transmission control protocol
TGT	Ticket Granting Ticket used by CAS
TGC	Ticket Granting Cookie used by CAS

Abbreviation	Text
UA	User Agent
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WMS	Web Map Service
WFS	Web Feature Service
XML	Extensible Markup Language

1 Introduction

Stateless web services have some big advantages compared to web services that require a session with the client. An example for a stateless service is a search request using an search engine. The request transmits the keywords and the result is presented. Internet shopping is a good example for a stateful service needing a session as the server must track the ordered items between the requests.
Stateless services do not consume server resources after the response was sent. The concept is to have a simple request – response model. Additionally, techniques like clustering, load balancing and fail over are working out of the box. There is no problem to build server farms to satisfy many concurrent requests.

OGC [22] (Open Geospatial Consortium) offers standards for web requests in an GIS (Geographic Information System) infrastructure. These requests are stateless and the result of such a request is typically a rendered image or geographic data in GML (Geographic Mark up Language). The GML format is based on XML, the specification is located at [22] . This thesis uses OGC standards to illustrate the concepts without loss of generality, meaning that the concepts will apply to all kind of stateless services.

The above scenario works well if there is no need to protect the data provided by the server(s). In an GIS environment it is often needed to restrict some clients to certain regions or certain kinds of vector/raster data. As a consequence clients must authenticate themselves for each request.

Authentication mechanisms can be costly. An example is the usage of an authentication that requires a lot of complex calculations. Authentication for each request can reduce system performance dramatically therefore techniques like caching authentication results are needed.

Of course, it is possible to store authentication results in server sessions, but the unwanted consequence is the loss of the stateless request-response model.

SSO (Single sign on) allows a client to authenticate only once for using different sites, not forcing the client to re-authenticate for each site individually.

This thesis will focus on mechanisms for authenticating clients without needing a session between client and servers. Authentication mechanisms covered are standard HTTP authentications, some kinds of proxy authentication scenarios and SSO solutions.

Most of the concepts presented in this work will have a concrete implementation. Since the author of this work is a contributor to related open source GIS projects, the community offers the possibility to test the concepts on a concrete system.

The concrete project is called “Geoserver” and the home page is located at [12]. Work started during the Google summer of code 2011 initiative.

Geoserver supports OGC (Open Geospatial Consortium) standards and is based on the

Java programming language. The license of the software is GPL (GNU General Public License) and the copyright is hold by OpenGeo, the home page can be found at [13]..

The implementation uses the well known Spring framework[14] as a major building block. Spring itself is an umbrella for a number of sub projects, one of which is Spring Security that contains implementations of many standard authentication mechanisms. This work is based on the Spring Security framework and the concepts of this library will be described in the following chapters.

Geoserver employs a plug-in architecture that allows for new code and GUI (Graphical User Interface) elements to be injected without touching existing code. The new security architecture for Geoserver developed during this work follows this pattern, allowing injection of different kind of authentication services.

During the presentation of the theoretical concepts in this work the author will use the implementation experience to illustrate the used mechanisms and provide some arguments about design decisions.

Below an example how the system works, taken from the Geoserver demo data, running Geoserver on the local host. Sending the following URL to Geoserver results in a picture of the United States.

```
http://localhost:8080/geoserver/topp/wms?
service=WMS&version=1.1.0&request=GetMap&layers=topp:states&styles=&bbox
=-124.73142200000001,24.955967,-
66.969849,49.371735&width=780&height=330&srs=EPSG:4326&format=applicatio
n/openlayers
```

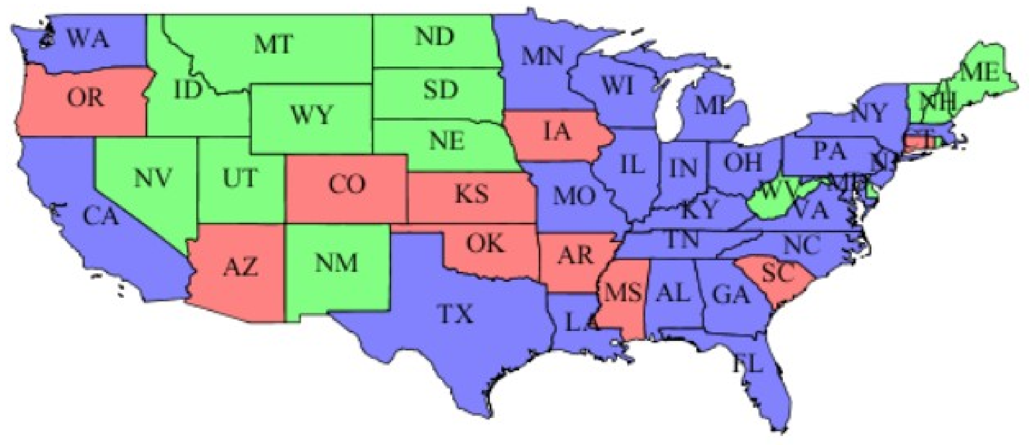


Illustration 1: states of the USA

taken from Geoserver demo installation

2 Technical Background

This section covers concepts not directly related to authentication, but needed as building blocks for later chapters. Most important are concepts related to HTTP (Hyper Text Transfer Protocol) and the Java programming language. A brief description of SAML (Security Assertion Markup Language) completes this chapter.

2.1 Cookies

A cookie is a formatted string consisting of a name/value pair and some additional attributes delimited by semi-colons. A cookie is created on the server and sent back to the client. The client sends back the cookie with all subsequent requests matching the cookie attributes.

A cookie can have the following attributes:

- **expires**
  The date and time when the cookie expires in GMT (Greenwich Mean Time). If this attribute is missing, the cookie becomes invalid after the client terminates. (Closing the browser as an example)

- **path**
  The path prefix to which the cookie applies

- **Domain**
  The domain in which the cookie is valid, starting with a dot

- **Secure**
  The cookie is only sent with a secured protocol like HTTPS (Hyper Text Transfer Protocol Secure)

- **HttpOnly** (intended for browsers)
  Instructs the browser software to send the cookie only with HTTP(S) requests originated from the browser itself.

A full specified example contained in an HTTP response

```
Set-Cookie: username=testuser; Domain=.mydomain.com ; path=/geoserver;
Expires=Wed, 13-Jan-2Related012 22:23:01 GMT; Secure; HttpOnly
```

“Set-Cookie” is an HTTP response header attribute set by the responding server. Clients should look for such header attributes (A response can contain more than one “Set-Cookie” attribute).

The client sends back cookies as a comma delimited list of key/value pairs, the response attribute is named “Cookie:”

```
Cookie: username=testuser,otherkey=othervalue
```

The client sends back the cookie only if the following properties hold:

1) The cookie has not expired
2) The server is in the correct domain (e. g. mydomain.com)
3) The URL path matches, (e. g. starts with /geoserver)
4) If the **Secure** attribute was set the request uses a secure channel
5) If the **HttpOnly** attribute was set the request must be originated from the browser directly and not by a script running in the browser

The server may place a new value for an existing cookie into a response. In this case the client should use the new value for future requests. This makes it possible to create a cancel cookie with an empty value.

2.2 HTTP Sessions

HTTP (Hyper Text Transfer Protocol) is stateless. For each HTTP request (POST or GET as an example) a new TCP (Transmission control protocol) connection is established. Performance my be increased by using keep-alive TCP connections (Since HTTP version 1.1 this is the default). If a web service needs to keep specific information for the client between a series of requests from this client, an HTTP session is created.

When a new session is created the server generates an unique session id and a cookie holding the value of this id. Furthermore the server reserves memory for the session to store client specific data. The server relies on the client to include this cookie in each of its subsequent requests.

An example for a session cookie

```
Set-Cookie: JSESSIONID=ABE341FFAADD342;.......
```

In the above example the generated server session id is " ABE341FFAADD342". At receipt of a client request the server looks for a cookie called "JSESSIONID", extracts the value, and associates the reserved memory for this value to the request.

There are other techniques to handle session ids like URL rewriting but that is beyond the scope of this work, most sessions today are cookie based.

2.3 Types of HTTP Services

Seen from the security perspective, there are two types of services:

1) Stateful services that establish a session with the client and store information needed during a series of requests sent form this particular client.

2) Stateless services that communicate without establishing a server session and use a simple request-response model.

A web browser is a typical client but many different types of clients exist. In almost any computer language it is possible to connect to a server using HTTP. A code snippet using Java

```
URL url = new URL("http://www.myserver.org/index.html");
HttpURLConnection connection = (HttpURLConnection)url.openConnection();
InputStream is = connection.getInputStream();
// InputStream is contains the server repsonse
```

2.4 Principal and Credentials

Principal denotes the subject that wants to authenticate. This may be human being or a software/hardware module in a trusted computing environment.

Credentials denote the secret that the principal presents to authenticate. This may be a password, a preshared key, a pin code, a finger print, a ticket and so on.

A user id is a kind of a principal, a password is a kind of a credential. A principal may have a set of credentials.

2.5 RBAC

Authentication of a principal may be required for a variety purposes. Billing and auditing are examples.

Often it is required to have an active access control system for the resources offered by the server. RBAC (Role Based Access Control) is a concept that describes how to authorize access for a principal to a specific set of server resources. The interested reader may find in depth information about RBAC at [1].

Resources are protected by roles. The system needs a role authority that assigns roles to an authenticated principal. Access to a resource is allowed only if the principal has one of the required roles.

This work uses the term "authentication procedure" for authenticating a principal and upon success calculating the roles of this principal. The result is referred to as an "authentication token".

2.6 Clustering Scenarios

A cluster is a set of servers sharing the load created by concurrent client requests. A cluster can grow and shrink dynamically. Additionally fail over mechanisms keep the web site online in the event of one or more individual server crashes. All clients use a single "cluster address" and the cluster software dispatches requests to individual cluster members.

2.6.1 Clustering with stateful services

The HTTP session itself is established with one individual cluster member. If the next client request is directed to another cluster member the problem is obvious.

The first solution to this problem is to have cluster members share sessions. Most Java server applications are run within an J2EE (Java 2 Enterprise Edition) container. Some container implementations offer the feature of session sharing. In this case it is usually the responsibility of the developers is to store only serializable (transformable into a byte sequence and creatable from this sequence) objects within the session.

The second solution is to use a cluster software providing session affinity. The cluster software itself routes each request containing session information to the individual cluster member for which the session is established with.

2.6.2 Clustering with stateless services

In the case of stateless services the cluster runs out of the box since each HTTP request performs the authentication procedure. But as already discussed the performance penalty may be a problem and a cache is a possible solution.

A precondition for caching is that the authentication token is serializable as a sequence of bytes suitable for transport over a communication channel and de-serialized by the receiver to get an identical clone of the authentication token.

Caching in a clustered environment requires a distributed or replicated cache.

2.7 Passwords

User name / password authentication schemes are widely used but care must be taken to avoid design mistakes. The best authentication scheme helps nothing if foreign passwords are easy to guess. Furthermore care must always be taken to avoid storing passwords unprotected in clear text.

2.7.1 Password Policy

Users tend to choose passwords that are easy to remember, especially because an average internet user has multiple user name / password pairs for different web sites. Unfortunately passwords that are easy to remember are also easy to guess. Passwords that are easy to guess are also known as weak passwords.

A password policy is a set of rules that force users to choose stronger passwords. Examples of such rules are

- force a minimum password length
- force a mixture of lower case and upper case letters
- force the inclusion of special characters , e.g. @, ?, !,#
- prohibit passwords found in a dictionary
- force a password change after a configurable number of days
- keep a password history for the user and prohibit reusing a password

The above rules are not exhaustive.

2.7.2 Password Generator

Password generators require an alphabet and a random generator. Random generators are initialized with an arbitrarily chosen number called a seed and produce a sequence of random numbers. Unfortunately knowing the seed makes it possible to reproduce the random sequence.

A cryptographically secure random number generator is needed for password purposes. This kind of generator does not exhibit the behaviour described above.

2.7.3 Password Digesting

There exist many systems that store passwords in plain text which is a very bad practice that can turn into disaster if a site is compromised.

The best way to handle passwords is digesting using hash functions. Examples of hash functions are MD5 or functions of the SHA family. These kinds of functions are one way functions and in practice it is not possible to calculate the plain text password from the digest.

The system itself does not store user passwords in plain text but instead the calculated digest is stored. During the authentication process the user provides the plain text password to the system which is then digested and compared with the stored digest of the user. If the comparison is successful the authentication is successful.

2.7.3.1 The Salt

If an attacker gets access to the password database and knows which digest algorithm is used he can attempt a dictionary attack. This is actually quite simple as the attacker only needs a dictionary with pre calculated digest values. A countermeasure is to use a second string known as a salt. This idea is illustrated in the following example:

Given the plain text password “mypass” , a salt “4711”, and the MD5 hash function, the calculation of the digest is

MD5(mypass:4711)

The result in hexadecimal is:

ad8f6c55ae3f3a414bda1f469f0c8f90

The password database stores the hexadecimal string and the digest. A simple dictionary attack using pre calculated MD5 digits will not work. It is not recommended to use one fixed salt for all passwords since a user with a weak password could compromise the whole password database. For a fixed salt, the attacker needs a precomputed dictionary for exactly this salt.

A good practice is to use a random salt for each user. In this case, the attacker needs a precomputed dictionary for each salt, making the pre-computing almost futile.

Another type of attack is based on the birthday paradox. This attack is performed on an a user database as a whole if the user database uses no salt or a fixed salt.

An attacker is not interested in the exact password, he is only interested in a password producing the needed digest (This is also true for all kind of attacks against digesting). Many passwords may produce the same digest and the probability of finding two passwords resulting in the same digest is described by the birthday paradox.

Short description:
How many people are required to find two people with the same birth day with a

probability greater 50 %. The answer is that only 23 persons are needed.

An attacker having access to a password database knows all digests stored in this database. The next step is to generate random passwords, calculate the digests and try to produce a collision with the digests in the user database. In caseof acollision, the attacker can authenticate with the user name of the victim by using his random password. The question is: How many passwords must be digested to get a digest equal to the digest of a random victim,

The mathematical calculations are beyond the scope of this work. If the bit length of the digest is n, roughly the square root of n calculations are needed to find 2 passwords producing the same digest with a probability of 50 %.

As an example, for n = 128 bit digest size, the attacker needs to digest 2^{64} password pairs to find a pair producing the same digest with a probability of roughly 50 %.

Since the attacker has access to the full user database, each generated password digest can be compared to all existing digests and the probability of a collision increases. Large user databases improve the chance of a successful attack.

The random salt is a countermeasure for this kind of the attack since the attacker must recalculate 2^{64} passwords for each user in the password database and cannot reuse calculation results.

2.7.3.2 Iteration Count

A further countermeasure is to apply the digest algorithm more than once. An iteration count greater than 1000 is recommended. Calculating 1000 MD5 sums iteratively for one password is not a big challenge for modern day computer systems. But for the attacker creating the dictionary containing the digests it could be a computational problem. The iterations are also a countermeasure to simple brute force attacks (trying all possible passwords). The following figure illustrates the concept.

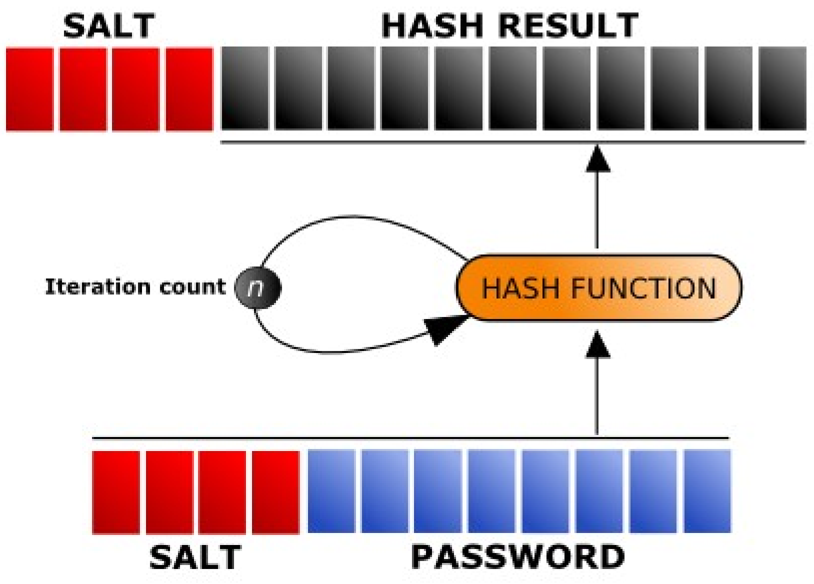


Illustration 2: Calculating a password digest taken from [4]

2.7.4 Alternatives to Password Digesting

alle überschriften mit Großbuchstaben anfangen
Since some authentication mechanisms require the user password in plain text (HTTP Digest as an example, covered in a later chapter), it is not always possible to use digest functions for storing password information. In such scenarios a cryptographic algorithm should be chosen, AES (Advanced Encryption Standard) is an example of one such algorithm.

2.7.5 PBE (Password Based Encryption)

Password based encryption derives the encryption key from a password. Input parameters are

- The password
- An iteration count
- A salt

The salt is added to the password and the concatenated string is hashed using a hash function such as MD5. The iteration count determines how often the hash function is executed. The result of this calculation is the encryption key used to encrypt/decrypt plain text. The ciphertext and the salt must be stored to be able to decrypt the encrypted password.

The Achilles tendon of this concept is that it uses passwords generated from humans that tend to be too simple. Assuming a password with the length of 8 bytes allows for 2^{64} different passwords.

Assuming 8 characters and a password alphabet consisting of 26 lower case letters, 26 upper case letters and 10 digits, $(26+26+10)^8=62^8$ different passwords can be created. Calculating

$$\ln(62^8)/\ln(2) = 47.6335$$

shows that the effective password length is about 6 bytes, since 6* 8 = 48. PBE encoding can simply be attacked by brute force.

2.8 Needed Java Concepts

2.8.1 Passwords and Java heap dumps

Java uses a memory heap to store objects. Strings are objects as well and stored on the heap. It is possible, mainly for fixing memory leaks, to create heap dumps from a running system that show the contents of the heap. There are a number of ways to generate a heap dump, one of them is

```
jmap -dump: format:b pid
```

where pid is the process id of the running Java virtual machine. If an attacker can trigger a heap dump and analyse it using appropriate tools, passwords could be compromised by looking for string objects containing user passwords.

The situation is even more worse for an Open Source project since the source code is readily available. This makes it is easier to navigate the heap dump and find password strings. Many Java applications store passwords as a String. Strings in Java are immutable and pooled in memory for performance reasons.

The following code snippet has not the desired effect:

```
String passwd = getPasswordForUser(“myuser”);
// do authentication
passwd=””;
```

The statement passwd=”” does not remove the password from the String pool and it can be found in the heap dump. A better approach is using character arrays:

```
char[] passwd = getPasswordForUser(“myuser”);
// do authentication
RandomPasswordGenerator.generate(passwd);
passwd = null;
```

The password is a character array and not stored in the String pool. After authentication, a random password generator is used to overwrite all the characters with new random characters. The statement passwd=null is a hint for the garbage collector to free the unused memory. It is not a good practice to overwrite all the password characters with a constant value because an attacker could look for this pattern in the heap dump.

It is important to minimize the time a password is stored in memory and overwrite it after usage. Digested passwords are resistant against such attacks, plain text and encrypted passwords are not.

2.8.2 Java and unrestricted key length

Java security supports the most commonly used security algorithms, mechanisms and protocols. Included are cryptography, public key infrastructure, secure communication, authentication and access control. An overview can be found here [2]

Due to US export restrictions encryption keys are limited for specific algorithms.
(* meaning no restriction)

Algortihm	***Maximum key size***
DES	64
DESede	*
RC2	128
RC4	128
RC5	128
RSA	*
All others	128

Table 1: Official key limitation table

Taken from [19], Appendix C

As a consequence, taking AES as an example, key size is limited to 128 bits. Java code trying to use a cipher with a key length greater than the allowed key size results in an “InvalidKeyLengthException”.

These limitations are configured by policy files shipped with the Java runtime environment. To remove these restrictions Java runtime providers offer alternative policy files that can be downloaded and installed in place of the original policy files. These policy files provide unlimited key length, are cryptographically signed by the Java runtime provider, and cannot be intermixed with other runtime environments.

2.8.3 Key Store

2.8.3.1 Concepts

It is a bad practice to use the same secret key for different tasks. For a system offering web services there numerous tasks needing a secret key. Examples are

- Encryption of passwords for back end systems
- Encryption of URL parameters
- Encryption of user passwords if password digesting is not possible (HTTP digest access authentication as an example, covered in a later chapter)

A solution for this requirement is the use of a key store. The key store itself is protected by a key store password and contains the needed key material.

Examples of key store formats are JCEKS (Java Cryptography Extension key store), JKS (Java key store), and PKCS12 (Public-Key Cryptography Standards #12 key store)

JKS and JCEKS are proprietary formats developed for Java. A key store should be able to store

- Private keys (asymmetric key)
- X 509 certificates containing the public key
- Shared secrets (symmetric key)
- Certificates from Certificate Authorities

An JCEKS/JKS key store contains key store entries. Each entry may be protected by an individual password. Protection is achieved by encrypting the entry with the individual password. The whole key store is protected by a key store password, again by encrypting the whole key store with the key store password.

The “double” encryption additionally protects against heap dump attacks. At runtime the key store is loaded into memory and must be stored decrypted but the key entries are still encrypted. If a key is needed, the key is fetched from the key store, decrypted, used and finally scrambled.

2.8.3.2 Key Store Password

The plain text key store password is needed by the running system to open the key store. The question is how to provide the system with this password. The possibilities are discussed assuming the software (Java) is running in a Linux environment without loss of generality.

1) Prompt for the password during the bootstrap sequence. This is a good solution but requires human intervention on start up and is not suitable if an automatic start is a requirement.

2) Use a Java system property. The service is started passing the password in the command line.

```
/user/bin/java -DKEYSTOREPASSWORD=mypassword ….
```

The developer retrieves the password using

```
String password=System.getProperty(KEYSTOREPASSWORD);
```

This is a bad solution as opening a Linux terminal and entering

```
ps -ef | grep java
```

shows the running processes containing the string “java” and all its command line parameters, including “KEYSTOREPASSWORD=mypassword”

3) Add an environment variable to the start up script

```
export KEYSTOREPASSWORD=mypassword
```

This also a bad solution since start up scripts are world readable. Although a solution to this would be to protect this file using file access control.

4) Create a system user who starts the service. It is always a good practice to run

server software owned by a system user not having administration rights. The export statement used in 3) is added to the users profile. Again, the profile must be protected by file access control.

5) Like 4), but create a hidden file within the users home directory, only the user has read/write access. Write access is needed if the password must be changed.

2.8.3.3 Protecting the key store password from foreign code

Since Java employs a plug in architecture, additional code can be injected into a running system in a variety of ways. It is beyond the scope of this work to go into detail but some important concepts should be mentioned.

A Java class name should be unique world wide. Most class names start with a domain name, for example “org.geoserver.security.KeyStoreProvider”. The domain is “org.geoserver” and “org.geoserver.security” is the package name. Java classes and interfaces are grouped in packages.

Methods and instance variables may be protected by access modifiers.

- private , only members of the class can use such methods
- package, only members of the same package have access
- protected, members of the same package and sub classes have access
- public, any code has access

Additionally, classes can be declared with a “final” modifier to disallow sub classing.

Finally classes are deployed in “*.jar” files. A running Java application loads many such “*.jar” files. It is possible to seal such files. Sealing prevents another jar file from injecting classes with the same package name as already contained in the sealed jar file.

2.8.4 Java Servlets and Filters

Java server applications are typically run in an J2EE (Java 2 Enterprise Edition) Container. The container acts like a framework and provides much functionality useful for Java developers. One of these functions is converting HTTP requests to Java objects (class javax.servlet.http.HttpServletRequest) and preparing Java response objects (javax.servlet.http.HttpServletResponse) for transmission to the client.

The container routes each HTTP request to an object of type java.servlet.http.HttpServlet. For each HTTP method (POST , GET, PUT, DELETE, HEAD, OPTIONS,TRACE) exits a corresponding Java method

```
void doPost( HttpServletRequest req, HttpServletResponse res);
void doGet( HttpServletRequest req, HttpServletResponse res);
void doPut( HttpServletRequest req, HttpServletResponse res);
void doDelete( HttpServletRequest req, HttpServletResponse res);
void doHead( HttpServletRequest req, HttpServletResponse res);
void doOptions( HttpServletRequest req, HttpServletResponse res);
void doTrace( HttpServletRequest req, HttpServletResponse res);
```

The following illustration shows the overall architecture.

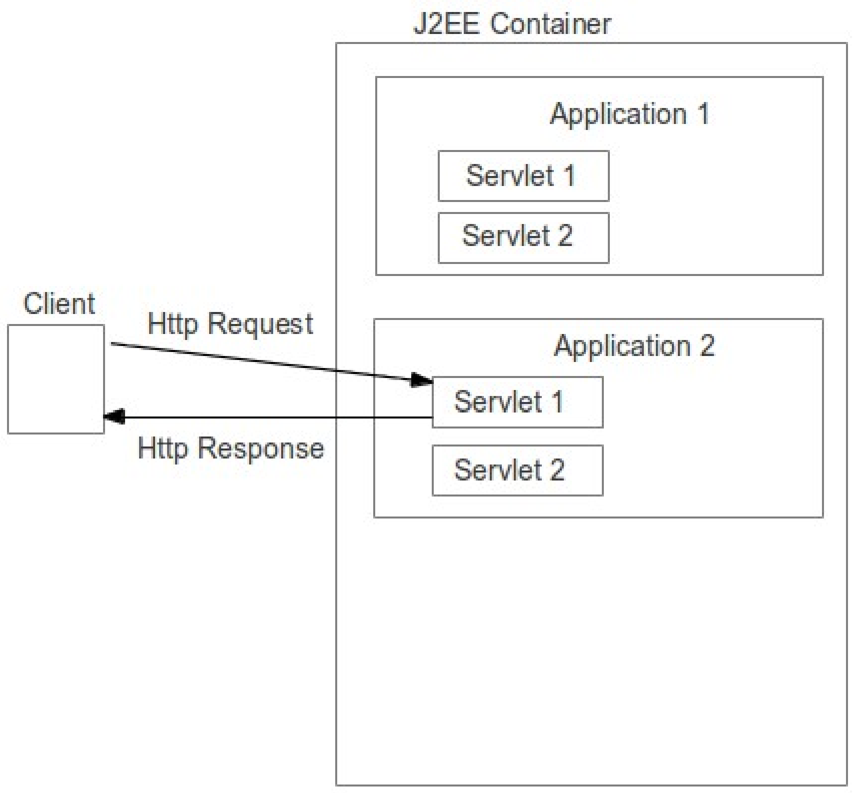


Illustration 3: J2EE Container

own creation

The URL of the client request determines which Servlet should to be used. The correct web application is found in the context root part of the URL.

Example:

```
http://localhost/geoserver/login.jsp
```

“geoserver” is the context root and the web application itself knows which servlet is responsible for /login.jsp.

It is possible to add Java servlet filters between the HTTP client and the servlet. A filter can modify the request before it is received by the next filter or the servlet at the end of the chain. The response generated by the servlet may be modified as well when travelling back to the client.

A filter can also interrupt the chain becoming itself the end point of the request. The code in the filters and in the servlet is executed within one Java thread. For each http request a Java thread is created and terminated after the response is sent to the client. (Most J2EE containers use thread pooling to minimize the effort to create and destroy threads).

The next illustration depicts a filter chain.

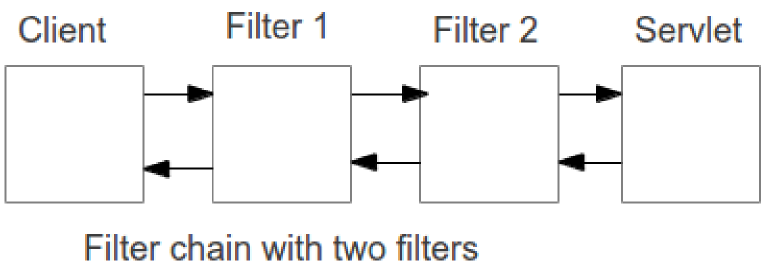


Illustration 4: Filter chain

own creation

A filter chain interrupted by "Filter 2"

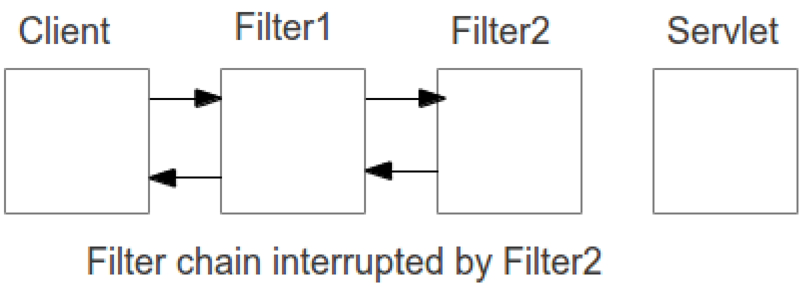


Illustration 5: Interrupted filter chain

own creation

2.9 SAML

SAML (Security Assertion Markup Language) is by far the most complex security architecture mentioned in this work. An overview of the version 2.0 is given and some investigations concerning stateless services are described in a later chapter.

SAML uses standard technologies such as XML and omits proprietary mechanisms. As a consequence this technology works well in heterogeneous environments.

Another benefit is no reliance on cookies. Cookies are not transmitted between domains, so a multi domain SSO cannot be realized using cookies. Most SSO products relying on cookies provide proprietary mechanisms for crossing domain boundaries.

The next advantage is the concept of a “Federated identity”. Most principals have different identifiers on different sites. SAML supports partners agreeing on sharing a federated identity and the mapping to the local identity.

The key concept of SAML is the idea of assertions. An assertion contains one or more statements. There are tree types of statements:

1) Authentication statements are created by a party responsible for authenticating a principal. At a minimum the statements contain the time of authentication and how the principal was authenticated. (e.g. the authentication mechanism).

2) Attribute statements contain attributes about the principals, comparable to a user profile.

3) Authorization statements describe what the principal is allowed to do, comparable to the role concept described previously in this work.

An example taken from [15]

```
<saml:Assertion xmlns:saml=”urn:oasis:names:tc:SAML:2.0:assertion”
 Version="2.0"
  IssueInstant="2005-01-31T12:00:00Z">
         <saml:Issuer Format=urn:oasis:names:SAML:2.0:nameid-format:entity>
              http://idp.example.org
        </saml:Issuer>
        <saml:Subject>
                <saml:NameID
                        Format="urn:oasis:names:tc:SAML:1.1:nameid-
                                        format:emailAddress">
                 j.doe@example.com
                </saml:NameID>
        </saml:Subject>
        <saml:Conditions
                NotBefore="2005-01-31T12:00:00Z"
                NotOnOrAfter="2005-01-31T12:10:00Z">
        </saml:Conditions>
        <saml:AuthnStatement
                AuthnInstant="2005-01-31T12:00:00Z" SessionIndex="67775277772">
                <saml:AuthnContext>
                        <saml:AuthnContextClassRef>
                urn:oasis:names:tc:SAML:2.0:ac:classes:PasswordProtectedTransport
                        </saml:AuthnContextClassRef>
                </saml:AuthnContext>
        </saml:AuthnStatement>
</saml:Assertion>
```

The subject identified is “j.doe@example.com”, expressed in email notation. The user was authenticated by a password transported using a secure channel. The validity of the assertion is included as a condition.

The content of SAML assertions are described using XML Schema. Sender and receivers may validate the content using an XML Schema processor.

The next building block are protocols describing requests and responses. Involved parties are the “asserting” party and the “relying” party. The asserting party is often referred to as the identity provider and the the relying party is often referred to as the service provider. There are six protocols defined.

1) Authentication Request Protocol:
   Describes how create an authentication request for an unauthenticated principal (SSO as an example).

2) Single Logout Protocol:
   Describes a logout informing all parties that the user has logged out.

3) Assertion Query and Request Protocol:
   Describes how a relying party can retrieve an assertion (by the assertion id) from an asserting party. The query part for queries about assertions (e.g. by the name of the principal).

4) Artifact Resolution protocol:
   Artifacts are references to protocol messages. Rather than sending an entire message itself, a small fixed-length value is sent. On demand the full protocol message is retrieved.

5) Name Identifier Management Protocol:
   Manages names and formats of principals.

6) Name Identifier Mapping Protocol:
   Manages mapping of SAML identifiers.

All SAML protocols need some kind of a transport protocol. The rules that define how to map SAML protocols are called bindings. The defined bindings are

1) HTTP Redirect Binding using HTTP response code 302

2) HTTP POST Binding using HTML from controls (base64 encoded)

3) HTTP Artifact Binding using an HTML form control or the query string to transport the artifact.

4) SAML SOAP Binding using Simple Object Access Protocol for transporting messages.

5) Reverse SOAP binding used in the Enhanced Client and Proxy profile described later.

6) SAML URI Binding used for retrieving SAML assertions by resolving an URI.

The last building block is the notion of a profile. Profiles combine assertions, protocol, and protocol binding for a special use case.

1) Web Browser SSO Profile:
Describes SSO from a standard web browser using HTTP Redirect, HTTP POST and HTTP artifact bindings.

2) Enhanced Client and Proxy (ECP) Profile.
SSO for clients using Reverse-SOAP bindings.

3) Identity Provider Discovery Profile:
Mechanisms for the service provider to detect the identity providers a principal has visited.

4) Single Logout Profile:
Logout with SOAP , HTTP Redirect, HTTP POST or HTTP Artifact bindings.

5) Assertion Query/Request Profile:
Defines how to use the SAML Query and Request profile to retrieve assertions.

6) Artifact Resolution Profile:
Describes how resolve artifacts referencing a protocol message.

7) Name Identifier Management Profile:
Shows how to use the Name Identifier Management Protocol with SOAP, HTTP Redirect, HTTP POST, HTTP Artifact bindings.

8) Name Identifier Mapping Profile:
Describes using the Name Identifier Mapping Protocol with SOAP.

In addition to the four described concepts (assertions, protocols, protocol bindings and profiles) two other concepts are needed.

“Metadata” for sharing configuration (e. g. key material) between SAML parties and “Authentication Context” for detailed information about an authentication of a principal. The authentication context, if needed, is referenced by the assertion.

The following illustration shows the relationship.

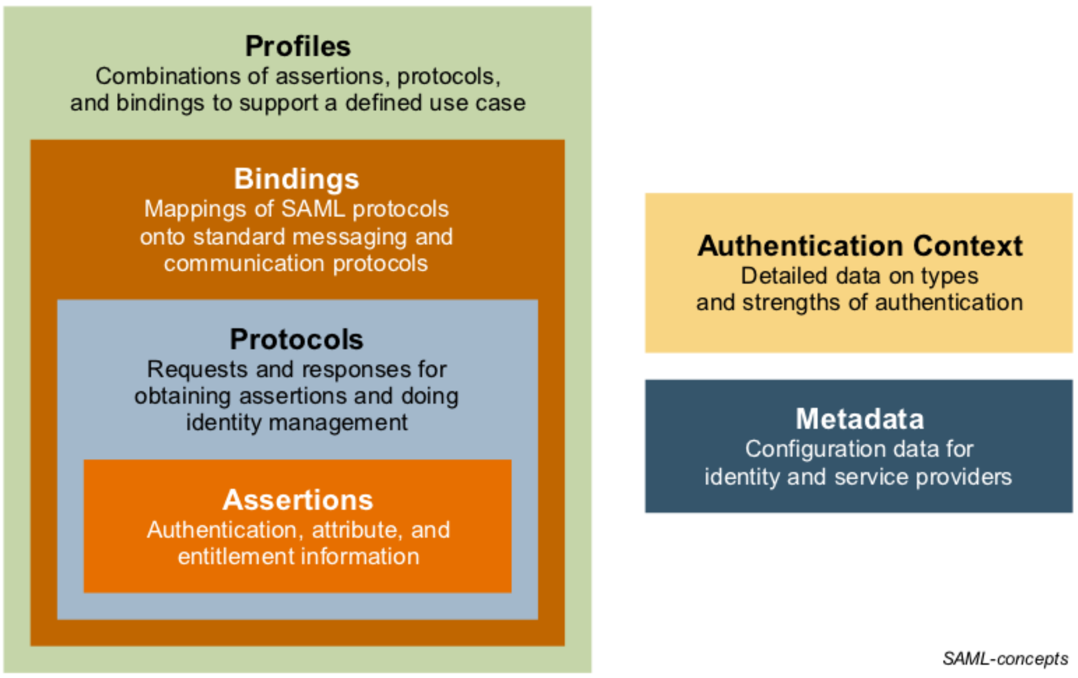

Profiles
Combinations of assertions, protocols, and bindings to support a defined use case
Bindings
Mappings of SAML protocols onto standard messaging and communication protocols
Protocols
Requests and responses for obtaining assertions and doing identity management
Assertions
Authentication, attribute, and entitlement information
Authentication Context
Detailed data on types and strengths of authentication
Metadata
Configuration data for identity and service providers
SAML-concepts

Illustration 6: Basic SAML concepts

Taken from [15]

3 Concept

This chapter introduces concepts increasing the security level of a software participating in the world wide web.

3.1 Encryption of URL Parameters

In some environments it may be necessary to encrypt URL parameters, especially if the parameters contain sensitive information. Most web browsers allow users to see the page source and to inspect the "raw" HTML (Hyper Text Markup Language) code returned by a server. As an example, without encryption:

```
http://GEOSERVER/web/?
wicket:bookmarkablePage=:org.geoserver.security.web.SecuritySettingsPage
```

After encryption:

```
http://GEOSERVER/web/?
x=hrTNYMcF3OY7u4NdyYnRanL6a1PxMdLxTZcY5xK5ZXyi617EFEFCagMwH
BWhrlg*ujTOyd17DLSn0NO2JKO1Dw
```

3.2 Caching the authentication token

The authentication token contains information about the principal (for example the user name) as well as the assigned roles including role parameters. Optional information about the credentials may also be included, such as the IP address of the client for example.

The authentication procedure may be costly, especially if connections to back end systems like relational databases are needed.

For stateful services the authentication procedure is executed only once and the authentication token is stored in the http session object created on the server side. All subsequent requests from the same principal are assigned to the session of this principal and the authentication token can be retrieved from the session itself.

For stateless services the situation is more complex. Since there is no http session the procedure to get the authenticated token has to be executed for each request. In most concrete deployments the overall server performance will decrease since significant server resources are needed to execute the authentication procedure for each request. A solution to this problem is to cache the authentication token. A precondition for caching is to derive a unique key for the authentication token from the incoming HTTP request. Each of the authentication mechanisms described in later chapters includes a proposal for key derivation.

The caching component has to meet the following requirements.

- A maximum number of token entries must be configurable

- Time to live (seconds from creation time) must be configurable. This is the maximum time a token may be cached for, after which period the token is removed and the authentication procedure has to be re-executed.

- Time to idle (seconds from last access) must be configurable. If a token is not used by an http request during this amount of time the token is evicted from the cache.

- Time to idle and time to live may be overridden for an individual cache entry

- It must be possible to remove a single cache entry (logout scenario)

- It must be possible to empty the cache (if the security system is compromised)

Some additional remarks:

Time to live is a hard limit, it does not make sense to have time to idle larger than time to live.

The individual time to idle and time to live parameters are necessary since some authentication mechanisms have their individual time out values. For example, consider HTTP digest authentication (covered in a later chapter) that generates a nonce (number used only once) and uses a validity in seconds for this nonce. After the period given by the validity expires the nonce is not accepted any longer. Having the time to live of the cache greater than the nonce results in caching unusable authentication tokens. The possibility of customizing time periods for individual cache entries helps to avoid such situations.

As long as the authentication token is in the cache, modifications of any kind (principal, credentials, role assignments) do not have any effect. The authentication token must be removed explicitly to force a re-execution of the authentication procedure. Therefore cache time outs should be chosen with care.

3.3 Security Filter Chain

The architecture used by Spring Security injects one special filter in the filter chain. The filter is called “FilterChainProxy” and it manages an ordered list of security filter chains. Each security filter chain has a pattern that is matched against the URL path of the incoming request. (Regular Expressions and Ant patterns[23] are supported). The first successful match selects the chain for the incoming http request.

As an example, Geoserver uses the following filter chains at the time of this writing:

Url Pattern	***Filter chain description***
/web/**	Allow anonymous login, use form login for administrative tasks
/j_spring_security_check	Log in a user, user name and password are found in HTTP parameters
/j_spring_security_logout	Log out a user explicitly
/rest/**	Login in a user using basic authentication
/gwc/rest/web/**	Allow anonymous login, use form login for administrative tasks
/gwc/rest/**	Login in a user using basic authentication
/**	Login in a user using basic authentication

Table 2: Geoserver filter chains

The following illustration shows this concept:

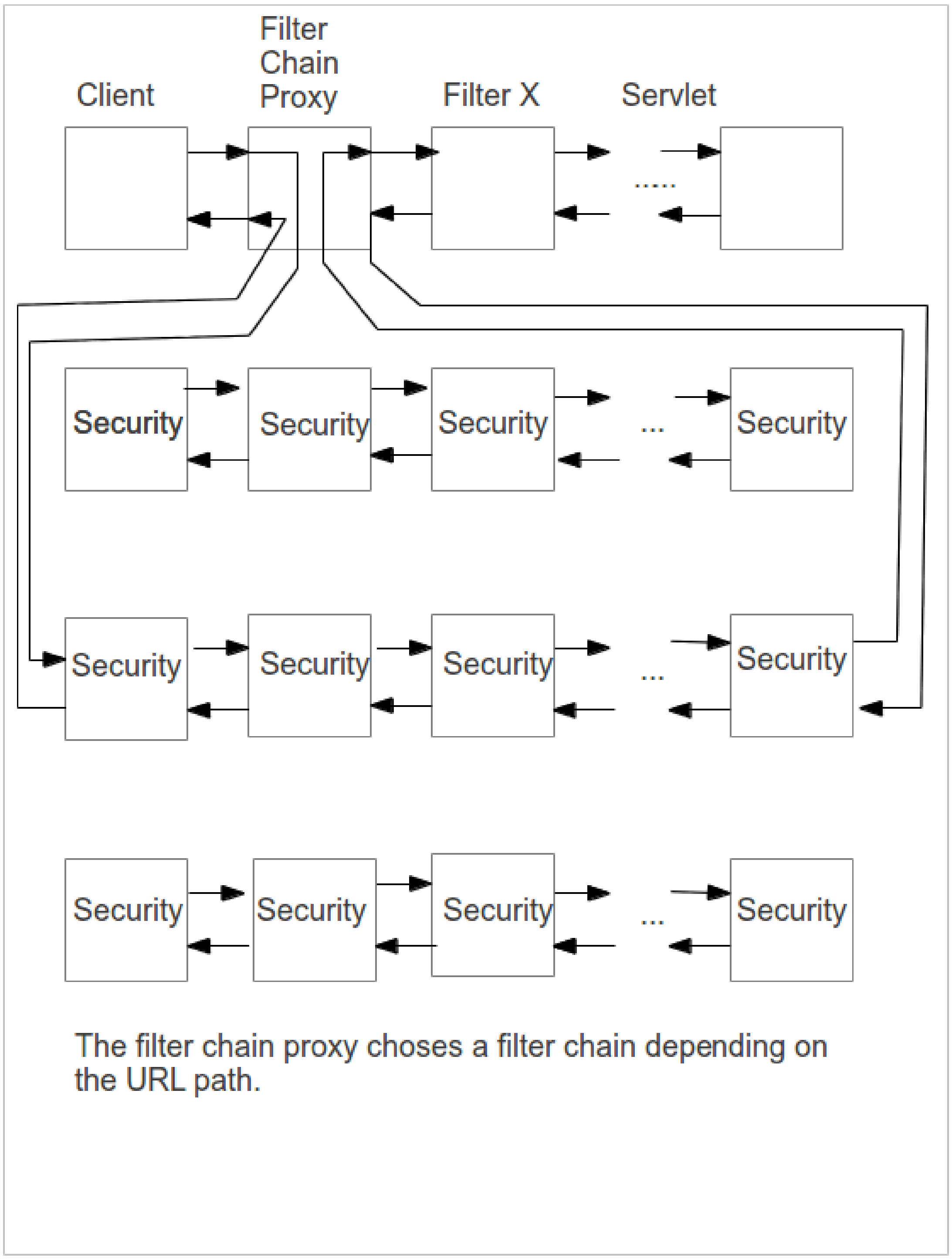


Illustration 7: Filter chain proxy

own creation

This chapter covers the architecture of a typical security filter chain. The necessary filters are described except filters doing authentication. Authentication filters are covered in a separate chapter.

The order of the filters within the chain is important. This work describes the filters in the required order. Some filters are optional while others are mandatory.

An empty filter chain is valid. Such a chain can be used for public resources. As an example, an empty filter chain with the ant pattern "/public/**" would route all http requests like http://localhost:8080/geoserver/public/..... through the empty chain. (In this example, "geoserver" is the context root and its used to find the correct web application running in the J2EE container).

The term "authentication entry point" is used to describe how the server challenges the client to provide a principal and credentials. A well known example is presenting a login form to the user and challenging for user name and password.

3.3.1 Channel Filter

This filter is optional. The most common use case is to force an HTTPS connection. The channel filter checks if the incoming request is transported over a secure channel. If this is not the case the chain is interrupted and an HTTP redirect is sent.

As an example, http://localhost/geoserver/login could be redirected to https://localhost/geoserver/login.

This filter uses the standard port mappings, port 80 for HTTP and port 443 for HTTPS. Other mappings are configurable, mapping port 9080 to port 9443 is also common.

Since some authentication mechanisms transport credentials in plain text over the network (Base64 encoding as used by Basic authentication is considered as plain text as well), forcing a secure channel is highly recommended.

3.3.2 Security Context Persistence Filter

This filter is mandatory. The authentication token is stored in a thread local variable by the authentication filters. A thread local variable behaves like a global variable but each thread has its own copy. The most important function of this filter is to nullify the thread local authentication variable at the moment the response object passes this filter. This no problem because except the optional channel filter (which does nothing with the response object) all other filters and the Servlet have completed their work.

Each HTTP request is executed within a single thread. Most J2EE containers work with a thread pool to avoid repeated thread creation and termination. Not nullifying the thread local variable could result in requests having an authentication token that was created by another request, resulting in a security disaster.

This filter is the first or second filter receiving the request object. It checks if there is an HTTP session established between the client and the application. The J2EE container does the job of assigning the correct session for the concrete client. If a session exists the filter determines if there is an authentication token stored in the session. If this is the case it takes the authentication token and puts the token in the tread local variable. All authentication filters in the chain will not try to authenticate since the thread local variable is already set.

The filter has a configuration property named "allowSessionCreation" that controls how sessions are created. If the property is set to true the filter creates an HTTP session and stores the authentication token in the session before nullifying the thread local variable. This makes sense for stateful services but not for stateless services.

3.3.3 Authentication Filters

Next in the chain are authentication filters covered in the next chapter. On successful authentication, the authentication token is put into the thread local variable.

3.3.4 Remember me filter

Remember me authentication is optional and may be used for "convenience" authentication. If the filter finds an empty thread local variable and a special cookie in the request it attempts to authenticate using the cookie and populates the thread local variable.

3.3.5 Anonymous authentication filter

This filter, if present, authenticates an anonymous user.

3.3.6 Exception Translation Filter

The exception translation filter is mandatory and is responsible for catching exceptions thrown by the "Filter Security Interceptor" covered in the next chapter.

There are two kinds of exceptions:

- Authentication exceptions, used for unauthenticated requests
- Access Denied exceptions, used if the request is authenticated but the principal does not have the required privileges (roles) to access the resource. This exception is not of interest for this work.

This filter needs a configured entry point that is used in case of an authentication exception.

A second responsibility of the filter is to save the original http request for redirecting after a successful authentication. This behaviour is configurable and it is also possible to redirect to a constant URL on successful login.

3.3.7 Filter Security Interceptor

This filter is the last in the chain and it is mandatory. It checks the following:

- Check if an authentication token present in the thread local variable, if not throw an authentication exception
- Does the authentication token have the required roles to grant access to the protected resource, if not throw an access denied exception

3.4 Authentication Filters

This chapter covers different authentication mechanisms, discussing advantages and disadvantages and practical implementation hints.

3.4.1 Authentication Token as Thread Local Variable

Thread local variables behave like global variables except that a difference instance exits for each thread. Spring Security uses a thread local variable to hold the authentication token. Again for each HTTP request the J2EE container creates a thread. All authentication filters respect the following rules:

- First the filter determines if the thread local authentication token is already set. If this is the case the request is passed to the next member in the chain and no further processing is done.
- If the thread local authentication token is not set the filter examines the http request to see if the information that it needs to try an authentication is present. If no such information exists in the request the request is passed to the next member.
- If the thread local authentication token is not set and proper information is found in the HTTP request, try an authentication. On success set the thread local authentication token and pass the request to the next member.
- If the authentication was not successful it is configurable if the filter should stop the chain and challenge the client for credentials or should pass the request to the next member.

For each of the following authentication mechanisms a short description is given and the entry point is described. Cache key derivations for stateless services are also discussed. To reiterate, all of the following filters become active if both the following conditions hold

1) The thread local variable does not contain an authentication token
2) The filter recognizes that it is responsible for this HTTP request

For filters that use the authentication token cache the following responsibilities also exist

3) Calculate the cache key, perform a look up for a token in the cache, if found put the token in the thread local variable and call the next filter
4) If the thread local variable was empty and a successful authentication procedure was executed put the authentication token in the cache before calling the next filter.

3.4.2 Basic Authentication Filter

3.4.2.1 Entry Point

The filter challenges the client for providing a principal and credentials by sending an HTTP response containing a header attribute “WWW-Authenticate”.

```
HTTP/1.1 401 Authorization Required
WWW-Authenticate: Basic realm="Geoserver Realm"
```

```
(additional attributes are not of interest)
```

The value of the HTTP response header attribute named “WWW-Authentication” contains the authentication scheme and the authentication realm. In the above example the scheme of “Basic” indicates HTTP basic authentication.

3.4.2.2 Description

This is the simplest form of authentication and it is supported by all modern browsers. The client begins by sending a request without authentication information

```
GET /protected/index.html
Host: my.geoserver.org
```

The “Filter Security Interceptor” (last filter in the chain) recognizes an unauthenticated access request and throws an authentication exception caught by the “Exception Translation Filter” that in turn activates the entry point.

After the client retrieves the user name and the password (by presenting the user with a login dialog box) the user name and password are base64 encoded and the original request is re-sent including an HTTP request header attribute called “Authorization”

```
GET /protected/index.html
Host: my.geoserver.org
Authorization: Basic QAxhzGRpbjpvcLVuIHNlc2FtZr==
```

The filter decodes the value of the “Authorization” header and checks the information against its user database. If the user can be authenticated successfully the filter proceeds with the chain and the the final response to the client is

```
HTTP/1.1 200 OK
(additional attributes are not of interest)
```

If it is known in advance that the client has to authenticate the procedure can be shortened. The client includes the “Authorization” header in the first request and the authentication filter tries to authenticate. On success, the request is passed to the next filter. On failure (user name unknown or wrong password) the entry point is activated.

The main disadvantage is that the password is NOT encrypted. The value of the “Authorization” header is created in the following manner: (the function base64 does a base64 encoding)

Basic base64(user:password)

The user “myuser” with password “mypassword” results in base64 encoding of “myuser:mypassword” and the final value is:

Base bXl1c2VyOm15cGFzc3dvcmQ=

Base64 encoding is invertible and seen from a security perspective it is the same as sending the password in plain text. As a consequence basic access authentication is only secure when used in combination with SSL (Secure Socket Layer).

For developers, using basic authentication is quite easy. A Java example:

```
HttpConnection conn = (HttpConnection)   URL.getConnection(url);
conn.setRequestProperty(
     "Authorization",
     "basic " +  Base64.encode("myuser:mypassword".getBytes(),
                      Base64.DEFAULT)
    );
```

Further details can be found in RFC 2617 [5]

3.4.2.3 Cache key derivation

Since HTTP is a stateless protocol the "Authorization" header must be present for each request. For stateless services the server must run the authentication procedure for each request. As described in section 3.5 it is desirable to use a cache. The question is how to derive a unique cache key from the "Authorization" header attribute.

As a precondition the password must contribute to the key generation, otherwise an attacker could authenticate without knowing the password. (If the authentication token is in the cache).

An idea would be to use the value of the "Authorization" attribute, e. g.

Base bXl1c2VyOm15cGFzc3dvcmQ

Unfortunately this is a bad practice because passwords should not remain in memory longer than needed and Base64 encoding does not really hide the password. A better solution is to extract the password, create a digest and use the digest as part of the key like

    user:MD5(password:filter name)

Example with "myuser", "mypassword" and a filter name "basicAuthFilter" results in

    myuser:a38ae2204c7aebb66c51da1ba80d7a4b

3.4.3 Digest Authentication Filter

There are two specifications describing digest authentication, the first being RFC 2069 [6] which was later replaced by RFC 2617 [5]. A server supporting digest authentication should support both specifications.

The important fact about digest authentication is that the password is NEVER sent over the network.

3.4.3.1 Entry Point

Digest authentication is a challenge-response protocol meaning the filter creates a challenge and verifies if the client can solve the challenge. The solution of the challenge is called the "response" and sent by the client. The filter solves the challenge for itself and compares the values. Knowledge of the password is needed for solving the challenge.

First the filter generates a nonce (number used only once) that has an expiration time. The time period is configurable, 300 seconds is a typical example. The expiration time is calculated by adding the time period to the current time of the system.

It is not specified how to generate the nonce. Spring Security uses a configurable fixed key and the following formulas:

```
temp = hex(md5(expiration time:key)
nonce=hex(expiration time:temp)
```

The entry point is used in case of an unauthenticated request or if the nonce has timed out.

3.4.3.1.1 Entry Point for RFC 2069

For an unauthenticated request the filter generates a nonce and sends an HTTP response

```
HTTP/1.1 401 Authorization Required
WWW-Authenticate: Digest realm="Geoserver Realm" nonce=”af097bde889”
(additional attributes are not of interest)
```

If the nonce has expired a new nonce is generated and a parameter named "stale" is added. The answer is

```
HTTP/1.1 401 Authorization Required
WWW-Authenticate: Digest realm="Geoserver Realm" nonce=”af097bde889”
stale=”true”
(additional attributes are not of interest)
```

3.4.3.1.2 Entry Point for RFC 2617

For an unauthenticated request the filter generates a nonce and sends an HTTP response

```
HTTP/1.1 401 Authorization Required
WWW-Authenticate: Digest realm="Geoserver Realm" nonce=”af097bde889”
qop=”auth,auth-int”
(additional attributes are not of interest)
```

The qop (quality of protection) attribute informs the client which kind of calculation for the response can be used. If the nonce is expired a new nonce is generated, the "stale" parameter is added.

Remark: There should be an additional parameter called “opaque” but Spring Security does not use this parameter because Internet Explorer 6 does not handle it correctly.

3.4.3.2 Description

3.4.3.2.1 Description for RFC 2069

The client parses the “WWW-Authenticate” response header attribute to retrieve the realm and the nonce.

```
HashValue1 = hex(MD5(user name:realm:password))
HashValue2 = hex(MD5(method:uri))
response=hex(MD5(HashValue1:nonce:HashValue2))
```

“method” is the used HTTP method (GET, POST, PUT, DELETE, HEAD, TRACE, OPTIONS).

The “uri” is chosen by the client.

The client sends

```
GET /protected/index.html
Host: my.geoserver.org
Authorization: Digest username=”myuser” realm=”Geoserver Realm”,
nonce=”af097bde889” uri=”/index.html” response= “9140abe89f”
```

The filter extracts the parameter values from the “Authorization” header and performs exactly the same calculation, comparing the result with the value of the “response” parameter.

If the values match and the nonce has not expired the user is authenticated and the request is processed as intended. Otherwise the entry point is activated.

3.4.3.2.2 Description for RFC 2617

The client parses the “WWW-Authenticate” response header attribute and retrieves the realm, the nonce, and the value of the “qop” parameter.

The value of the “qop” parameter may be “auth”, auth-int” or “auth,auth-int”. The last value leaves it up to client to decide between “auth” or “auth-int”.

The calculation for “auth”:

```
HashValue1 = hex(MD5(user name:realm:password))
HashValue2 = hex(MD5(method:uri))
response=hex(MD5(
        HashValue1:nonce:nonceCount:clientNonce:qop:HashValue2))
```

“clientNonce” is a nonce chosen by the client, “nonceCount” should be incremented for each request.

The calculation for “auth-int”:

```
HashValue1 = hex(MD5(user name:realm:password))
HashValue2 = hex(MD5(method:uri:MD5(entityBody))
response=hex(MD5(
        HashValue1:nonce:nonceCount:clientNonce:qop:HashValue2))
```

The client sends

```
GET /protected/index.html
Host: my.geoserver.org
Authorization: Digest username=”myuser” realm=”Geoserver Realm”,
nonce=”af097bde889” uri=”/index.html” qop=”auth” nc=”000001”,
cnonce=”ab67cd45” response= “9140abe89f”
```

Further processing as described for RFC 2069.

3.4.3.3 Cache Key derivation

The cache key can be derived by creating the following string:

```
username:realm:nonce:response
```

The response is important because the password has to contribute to the calculation of the key.

3.4.4 Form Based Authentication

3.4.4.1 Entry Point

The entry point is a combination of HTML and HTTP. The entry point answers with an HTTP redirect (response code 302) and the client is redirected to the login page.

Html Snippet showing a login form:

```
<form method="post" action="/j_spring_security_check">
  <input type="text" name="username" required>
  <input type="password" name="password" required>
  <input type="submit" value="Login">
</form>
```

Due to the fact that this entry point is a mixture of HTML and HTTP a special security filter chain is needed. Looking at the “action” attribute in the above HTML snippet the pattern for the chain must be the constant “/j_spring_security_check”.

The first filter in this chain must be a “Security Context Persistence” filter allowed to create http sessions. The second filter is a user name / password filter.

After the user submits the form the request is routed to the described filter chain and the security persistence filter passes the request object to the user name / password filter. This filter extracts the parameters “username” and “password” and executes the authentication procedure. Upon success an authentication token is set in the thread local variable and finally the “Security Context Persistence” filter stores the authentication token in the HTTP session object.

3.4.4.2 Description

Form based authentication is not a standardized HTTP authentication technique. The following steps are required

- An unauthenticated client wants to access a protected resource
- The exception translation filter activates the entry point.
- The browser renders the login form, user name and password must be entered.
- The client submits this information to the special user name / password filter, that tries to authenticate.
- On failure the entry point is reactivated
- On success the client is redirected to the original URL being accessed (or to a fixed URL if configured)

User name and password are sent in plain text so the use of SSL is recommended. The filter should accept the login request only if the HTTP request method is POST. Using the POST method implies transporting the user name and the password in the HTTP body. The format of which is:

```
username=myusername&passwod=mypassword
```

Using the HTTP Get method implies encoding of user name and password in the URL:

```
http://geoserver/web/login?username=myusername&password=mypassword
```

URLs are stored/cached in browsers for future use and for showing the browsing history. An attacker merely must to get access to the browser in order to compromise the system.

3.4.4.3 Cache key derivation

Form based login requires an HTTP session and it does not make sense to show a login form for each request. Therefore this authentication scheme is not usable for stateless services.

3.4.4.4 Form Logout Filter

A dedicated logout makes sense only for stateful services. A special security filter chain for a logout pattern like “j_spring_security_logout” is required. The first filter is a security persistence context filter and the second one is a logout filter. The logout filter invalidates the session and activates the entry point.

3.4.5 Proxy Authentication

Proxy authentication schemes have some common properties concerning authentication entry point and cache key derivation.

3.4.5.1 Entry Point

There is none. Since the user is already authenticated there is no need to challenge the client for principal and credentials.

To cover situations where the user name is invalid (e. g. the user has been disabled by the administrator) the entry point could send back an HTTP 403 error code (FORBIDDEN). This is the same behaviour like having a valid authentication token but the token does not have an associated role to get access to the server resource.

3.4.5.2 Cache key derivation

Since the principal is already authenticated no credentials are available. The cache key is the name of the principal.

3.4.5.3 J2EE proxy Authentication Filter

Most Java server applications are run inside of an J2EE container. An J2EE container provides an authentication infrastructure and a security API for developer. The interface "javax.servlet.http.HttpServletRequest" has a method named getUserPrincipal() that returns the principal and a method isUserInRole(String role) returning true or false.

In the J2EE concept, roles are created by developers and designers of an application. At deployment time, roles are associated to users and groups. There is no API to obtain all roles of a principal.

Unfortunately this is a problem for building an authentication token since the token should have associated all roles of the principal. If the set of roles is accessible by another mechanism, the solution is to iterate over all roles and assign the role to the token if isUserInRole(role) returns true.

3.4.5.4 HTTP Request header proxy authentication filter

The authentication (or the whole authentication procedure) takes place in a proxy server. The proxy server builds the authentication token and puts this information into HTTP request header attributes.

For example, consider a user "myuser" with two roles called "employee" and "supervisor". The "employee" role has a role parameter "nr" (employee number) with the value 4711:

```
principal: myuser
roles: employee(nr=4711);supervisor
```

The filter parses the value of these two attributes and builds the authentication token. This scenario does not require to cache the token since parsing should be fast enough.

The disadvantage is that an attacker could build such an HTTP request easily. As a countermeasure the authentication procedure should only accept these headers if the IP address of the sender is identical to the IP address of the proxy. However still an attacker using ip-spoofing (faking the sender IP address) could compromise the system.

A better solution is to use SSL with server and client certificates. This requires building up a PKI (Public Key Infrastructure).

3.4.5.5 X509 Certificate Proxy Authentication Filter

For SSL connections that use a client certificate the certificate is used to obtain the name of the principal. The SSL protocol is handled completely by the J2EE container and if the client presents a valid X509 certificate during the SSL handshake the container stores a X509Certificate object as an HTTPServletRequest attribute with the name “javax.servlet.request.X509Certificate“.

The filter retrieves the X509Certificate object from the HTTPServletRequest object and uses the method getSubjectDN() to get the distinguishable name.

Distinguishable names have a format like

CN=myname, OU=my organisational unit, …..

Using a regular expression “CN=(.*?),” makes it easy to extract “myname” from the distinguishable name.

3.4.6 Remember Me Authentication Filter

Remember me authentication is for user convenience and is not in itself a full featured authentication mechanism. The idea is to remember a user name and password for a certain amount of time. Within this period a user is logged in automatically without challenging for credentials.

Seen from the security perspective each automatic login is not a good concept. In any case there must be a possibility to stop logging in automatically for a specific user.

This feature is useful for developers and in non critical environments. It is surely not for production systems.

3.4.6.1 Entry Point

There is none. Authentication mechanisms requiring user name and password (basic authentication, form based, etc...) may communicate a successful login to the remember me service. Additionally a user group service is needed to retrieve the password of a user.

3.4.6.2 Description

The remember me service is called after a successful login at which time it knows the user name and the password. Additionally the expire time in seconds must be configured (e.g. 1209600 seconds for two weeks). The remember me service then creates a cookie with a predefined name. Spring Security uses the cookie name “SPRING_SECURITY_REMEMBER_ME_COOKIE_KEY”.

The calculations for the cookie value:

Signature = MD(user name:expire:password)
Cookie value = base64 (user expire:Signature)

The cookie attributes are set according to the description in section 3.1.

The cookie is sent on subsequent HTTP requests. The remember me filter checks if this cookie is included in the request and if the request is still unauthenticated. If this is the case the following authentication takes place:

- Base64 decoding of the cookie value
- Extract user name, expiry, and password
- Retrieve the password for this user name from the user group service
- Calculate the signature to verify user name and expire time
- If signature verification fails send a cancel cookie and stop
- If the expire time is exhausted send a cancel cookie and stop
- Authenticate the user

Sending a cancel cookie means sending a cookie with the fixed name, an empty value, and an expire attribute set to 0. This makes the cookie unusable.

The danger in this scheme is hijacking of the cookie. If the user recognizes that his cookie is compromised he must alter his password making the cookie signature invalid. If the user does not recognize the cookie theft the attacker can log in as this user until the cookie expires.

3.4.6.3 Cache Key Derivation

Caching does not make sense in this case since the remember me authentication can be seen as its own caching mechanism.

3.4.7 Anonymous Authentication Filter

The anonymous authentication filter looks into the thread local variable and if it is empty it puts a fixed anonymous authentication token into the variable. This token has exactly one role called something like "ROLE_ANONYMOUS".

No entry point and no cache key derivation are needed for this filter.

3.4.8 Cascaded Authentication Filters

At least one authentication filter is needed in a single chain but it is possible to put any combination of these filters in the chain. This is a very powerful feature. Consider the following filter sequence as an example:

1) J2EE proxy authentication
2) X509 Certificate authentication
3) Basic Authentication
4) Digest Authentication

This is possible since each filter checks the request to see if it can extract a principal and credentials (for filters needing credentials). If the filter does not find a principal it passes the request to the next filter.

If a filter is able to extract a principal it tries to authenticate. On failure the filter activates its entry point and interrupts the chain. On success the authentication token is created by the authentication procedure and store in the thread local variable.

As an example, the following “Authorization” attribute is contained in an HTTP request header and an empty thread local variable is given. Additionally no J2EE authentication has happened and no X509 certificate was sent.:

```
GET /protected/index.html
Host: my.geoserver.org
Authorization: Basic QAxhzGRpbjpvcLVuIHNlc2FtZr==
```

1) The J2EE filter finds an empty thread local variable and tries to get a principal from the J22E container. Since there is no principal the filter passes the request to the next filter.

2) The X509 filter finds an empty thread local variable and looks if it can obtain a X509 certificate from the container. There is no such certificate so the filter passes the request to the next filter.

3) The Basic authentication filter finds an empty thread local variable and looks for an HTTP header attribute named “Authorization”. If finds one and the attribute value starts with “Basic “ indicating that this filter is responsible for authentication.
On success the authentication token is stored in the thread local variable.

4) The Digest authentication filter finds the thread local variable already set and passes the request to the next filter.

If the authentication is not successful in step 3 the filter interrupts the chain and activates its entry point, sending back

```
HTTP/1.1 401 Authorization Required
WWW-Authenticate: Basic realm="Geoserver Realm"
```

3.5 CAS (Central Authentication Service)

CAS is an SSO (Single sign on) protocol based on HTTP. The actual protocol version is 2.0 and will be covered here. Stateful and stateless services can use CAS for authentication. Single sign on support is only available if all connections use SSL.
The protocol is based on HTTP 302 redirect messages and the CAS server itself never initiates a new connection. (There is one exception to this rule discussed later).

CAS uses cookies to manage Single sign on. After a successful login a TGT (Ticket Granting Ticket) is created and a TGC (Ticket Granting Cookie) is sent to the client.

```
CASTGC=TGT-10-PjKxupOEQ6fNk4H9gRbpvGT-cas; Path=/cas/; Secure
```

Remark: For all examples showing tickets the ticket is shortened for readability purposes.

The name of the cookie is “CASTGC” and the value always starts with “TGT-”. Since the cookie has the attribute **Secure** the cookie will be sent by the client only over SSL connections.

The home page of the project can be found here[7].

On the project home page an open source implementation called CAS Server can be downloaded. The implementation is written in Java and packaged as Java web archive. Installation of the software is quite simple. The Spring framework is a major building block and configuration is done by modifying Spring XML files. There is no administrative GUI. The user manual is hosted at [8].

Spring Security itself supports CAS with a special entry point, an CAS filter, and some other required classes.

3.5.1 SSO

3.5.1.1 Standard Login

CAS uses tickets for authentication. The principal must log into the CAS Server and on success a TGT is created and a TGC is sent back to the client. The validity time of the ticket is configurable. The TGT may be reused as often as needed until it expires.

There are two different scenarios concerning the work flow for a login depending on whether the user contacts the CAS server first, or contacts the web application first.

Contacting the CAS server first only works with SSL and requires the following steps.

1) The user logs in using the CAS server login form

2) If not successful the user stays in the loop and the login form is presented again.

3) If successful the user receives the TGC and an information screen about successful login is presented.

4) The user switches to the web application and tries to access a protected resource.

5) The web applications sends back a redirect response to the CAS server. (The entry point is activated). The redirect is executed transparently from the user point of view. The TGC and service URL of the web application are included in the redirect request.

6) The CAS server checks the TGC and that the user is already logged in. Next a check is made that the user is allowed to access the service. On success a ST (Service Ticket) is generated and a redirect (including the ST) to the service URL is sent.

7) The CAS filter in the web application extracts the ST and sends a validation request (including the ST) to the CAS server.

8) The CAS server validates the ST and on success returns the user name in the response.

9) The CAS filter executes the authentication procedure and redirects the user to the original request or to a constant login success page.

If the user contacts the web application first the following steps are carried out.

1) The unauthenticated client wants to access a protected resource so an authentication exception is triggered.

2) The exception translation filter uses the CAS entry point that sends an HTTP redirect back to the client. The client is redirected to the CAS login form. The redirect URL has a “service” parameter containing the properly encoded URL where the application itself expects the response of the CAS Server.

3) On login failure the user stays in the loop.

4) On successful login the CAS server generates the TGC and sends a redirect to the client pointing to the URL passed in the service parameter. Additionally the CAS server adds an URL parameter named “ticket” that contains the ST (Service Ticket).

5) The CAS filter extracts the ticket and sends it to CAS server for validation.

6) If the ticket validation is successful the name of the principal is found in the response from the CAS server.

7) The CAS filter executes the authentication procedure and redirects the user to the original request or to a constant login success page.

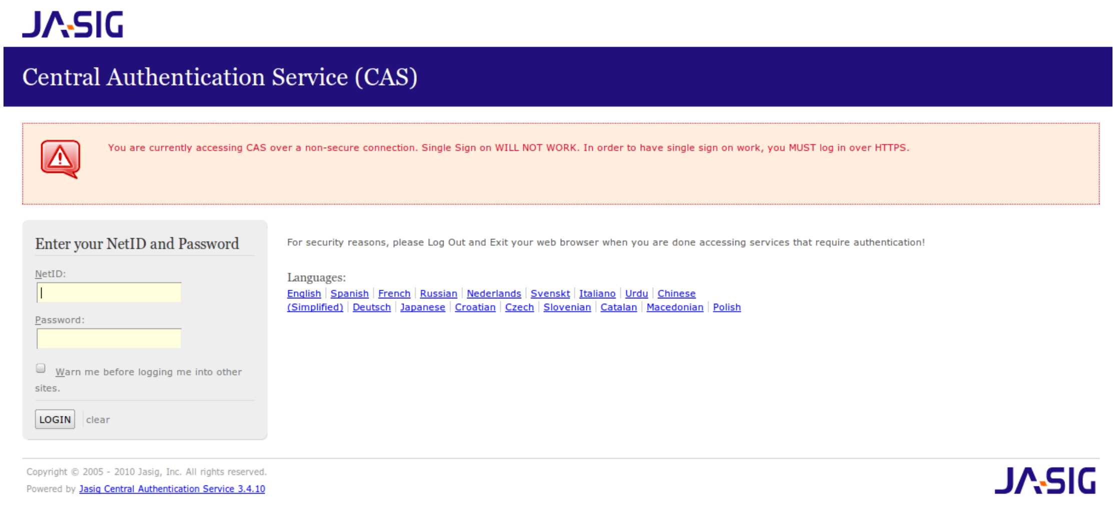
With a running CAS server it is possible to illustrate this procedure using a browser. In this concrete example the server is reachable under “http://ux-server02:8080/cas” and the service URL is “http://ux-desktop03/geoserver/j_cas_spring_security_check”.

The CAS standard installation comes with a test authenticator, using the same characters for user name and password results in a successful log in. The standard installation also allows to use any value as service as long as the the value is a well formed URL. In practice each user is actually restricted to a set of existing services.

Entering the following URL in a browser

```
http://ux-server02:8080/cas/login?service=http://ux-desktop03/geoserver/j_cas_spring_security_check
```

results in



Central Authentication Service (CAS)

You are currently accessing CAS over a non-secure connection. Single Sign on WILL NOT WORK. In order to have single sign on work, you MUST log in over HTTPS.

Enter your NetID and Password

NetID:

Password:

Warn me before logging me into other sites.

LOGIN | clear

For security reasons, please Log Out and Exit your web browser when you are done accessing services that require authentication!

Languages:
English | Spanish | French | Russian | Nederlands | Svenskt | Italiano | Urdu | Chinese (Simplified) | Deutsch | Japanese | Croatian | Czech | Slovenian | Catalan | Macedonian | Polish

Copyright © 2005 - 2010 Jasig, Inc. All rights reserved.
Powered by Jasig Central Authentication Service 3.4.10

Illustration 8: CAS login form

own creation

A warning is shown since the URL uses HTTP and makes the TGC obsolete.

After a successful login the CAS server sends an HTTP redirect. Since there is no running web application the browser will complain about not finding the URL but the URL itself is the interesting object.

```
http://ux-desktop03/geoserver/j_cas_spring_security_check?ticket=ST-6-3WsfnsTJ41qi4Y95cjre-cas
```

A service ticket is included. Now the web application is simulated by validating the ticket and entering

```
http://ux-server02:8080/cas/serviceValidate?service=http://ux-desktop03/geoserver/j_cas_spring_security_check&ticket=ST-6-3WsfnsTJ41qi4Y95cjre-cas
```

results in the browser showing

```
<cas:serviceResponse xmlns:cas='http://www.yale.edu/tp/cas'>
   <cas:authenticationSuccess>
      <cas:user>myuser</cas:user>
      </cas:proxyGrantingTicket>
   </cas:authenticationSuccess>
</cas:serviceResponse>
```

If the ticket validation fails (e.g. ticket time out) , the answer is something like

```
<cas:serviceResponse xmlns:cas='http://www.yale.edu/tp/cas'>
   <cas:authenticationFailure code="INVALID_TICKET">
      Ticket ST-6-3WsfnsTJ41qi4Y95cjre-cas not recognized
   </cas:authenticationFailure>
</cas:serviceResponse>
```

During this process a TGT and a ST have been created. The ST in only valid for a few seconds (configurable) and must be validated within this time period. Such a ticket can be used only once. A second validation would result in an error.

3.5.1.2 Protocol Elements

Terminology used for protocol description:

- “Client” refers to the user and/or the web browser.
- “Server” refers to the CAS server.
- “Service” refers to the application the clients wants to access.
- “Back-end” or “target service” is an application the service is trying to access on behalf of the client. (Proxy capabilities)

As seen in the above description some predefined URLs were used like “http://ux-server02:8080/cas/serviceValidate” and “http://ux-server02:8080/cas/login”. In these URIs “cas” is the context root of the web application and may differ. “/serviceValidate” and “/login” are predefined URIs (Uniform Resource Identifier) defined by the CAS protocol.

The “/login” URI can act as credential requester and as credential acceptor.

As a requester the following parameters can be passed:

- **service** (optional)
  The URL the client is redirected to after successful SSO authentication. This URL contains the “ticket” parameter and the value is a ST. If the URL is missing no redirection takes place.

  If the user is not actually signed in he is challenged for credentials. After successful SSO login a TGT is created and TGC is sent back. Clients having a valid TGC already are not challenged for credentials. (Only for SSL connections)

- **renew** (optional)
  If this parameter is set to "true" SSO will be bypassed. The client must present its credentials regardless of the existence of an existing SSO session.

- **gateway** (optional)
  The client is not challenged for credentials. If an SSO session exists the redirect takes place with a new ST. If no session exists the redirect takes place without a ST. This parameter has an exclusive or relationship with **renew.**

Valid parameters for "/login" acting as credential acceptor include:

- **service** (optional)
  As described above

- **warn** (optional)
  If this parameter is set to true Single sign on must not be transparent. The client must be prompted before being authenticated to another service.

Additional parameters are dependent on the authentication mechanism. For form based authentication the following parameters are needed.

- **username** (required)

- **password** (required)

- **lt** (required)
  "lt" is an abbreviation for "login ticket" and it is generated each time the login form is presented. The login ticket is generated to protect against replay attacks. The CAS server will accept a login ticket only once.

A special type of authentication is called "trust authentication" and must be configured individually. The protocol does not define parameters for this scenario. Trust authentication instructs the CAS server to trust that the client has already authenticated the user. No credentials are necessary.

The "serviceValidate" URI checks the validity of a ST and returns an XML fragment as response. On success this fragment contains the name of the principal.

- **service** (required)
  As described above the **service** must match the **service** from the "/login"

- **ticket** (required)
  The ticket to validate.

- **pgtUrl** (optional)
  URL for proxy callback, discussed in the section about proxy techniques

- **renew** (optional)
  If "true" the principal must have created a new SSO during "/login", existing SSO sessions are not accepted

Possible response codes:

- INVALID_REQUEST
  Not all request parameters where found
- INVALID_TICKET
  The ticket was not valid
- INVALID_SERVICE
  Service mismatch
- INTERNAL:_ERROR
  Internal error occurred during ticket validation.

3.5.1.3 Integration into Spring Security

The Java CAS client has its own security filters. Spring Security uses a lot of code from CAS and adds its own CAS filter to better integrate in to the Spring Security architecture. First the high level collaboration diagram:

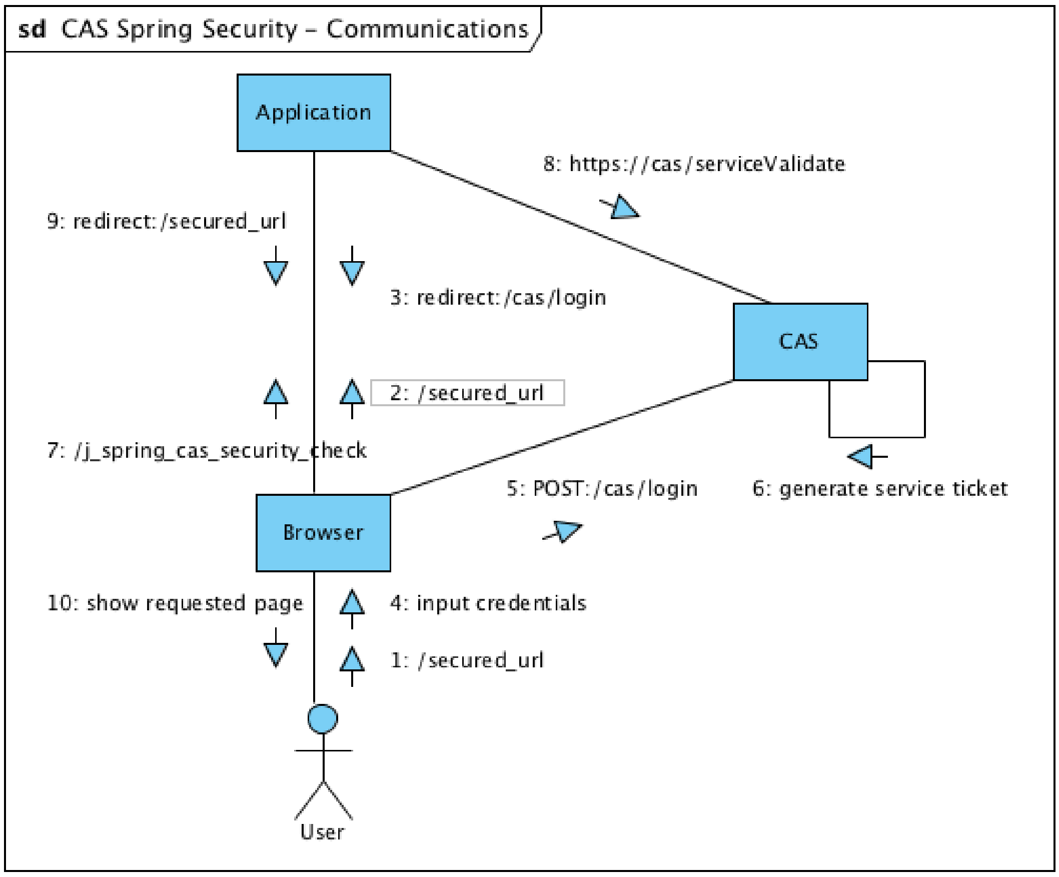


Illustration 9: CAS collaboration

Taken from [9]

There are two important facts to point out. The first is that the CAS server NEVER initiates a connection, the whole procedure works with HTTP redirects. The second is that the whole procedure is transparent to the user and he only fills out the CAS login form comparable to a local log in form.

Spring Security requires three types of objects for integration:

- CasProcessingFilterEntryPoint
- CasProcessingFilter
- CasAuthenticationProvider

Next the detailed sequence diagram:

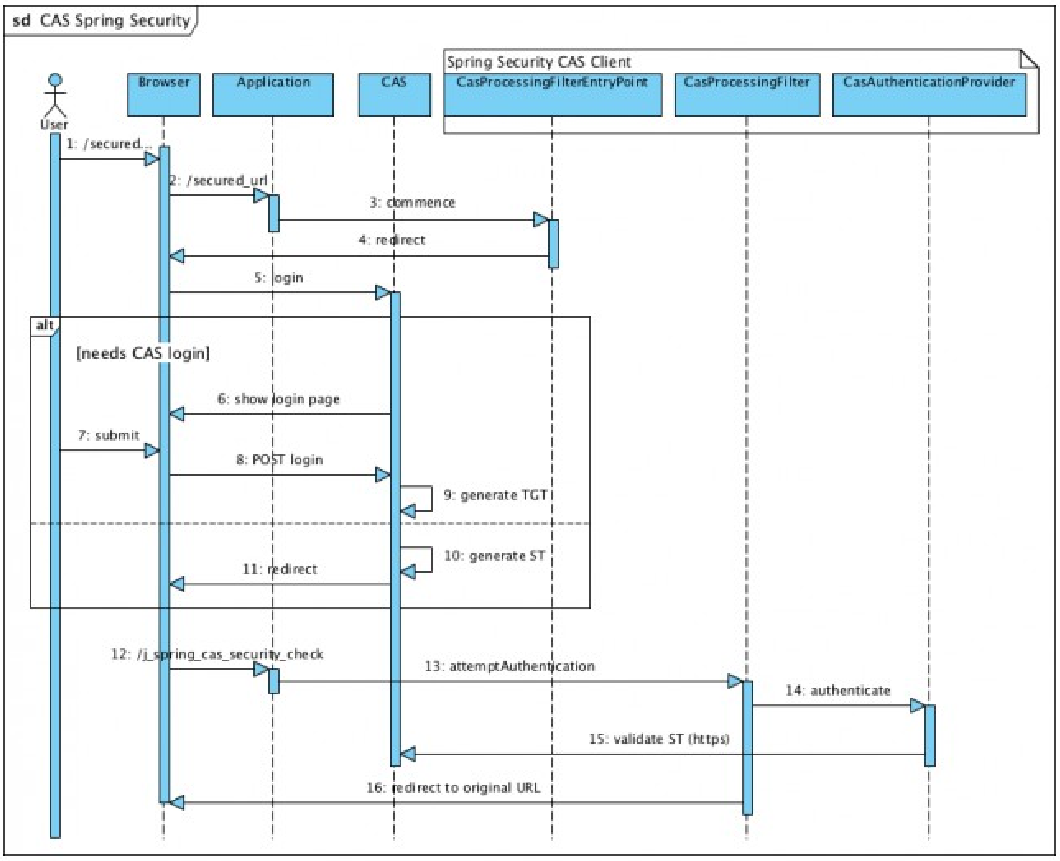


Illustration 10: CAS Spring Security

Taken from [9]

3.5.1.4 Standard Login including a PGT (Proxy Granting Ticket)

In this scenario three parties are involved.

1) The CAS server creating a TGT and issuing an ST and a PGT
2) The web application acting as a proxy, requesting the ST and the PGT
3) The proxied application (see chapter about stateless services)

This chapter covers steps 1) and 2). The procedure is more complicated and a simulation with the browser is shown in [10].

The first request is the same as for standard login

```
http://ux-server02:8080/cas/login?service=http://ux-
desktop03/geoserver/j_cas_spring_security_check
```

Again the browser will complain about a non existing URL but the ST is visible and can be used.

```
http://ux-desktop03/geoserver/j_cas_spring_security_check?ticket=ST-6-
3WsfnsTJ41qi4Y95cjre-cas
```

The simulation of the validation URL takes an additional parameter named **pgtURL.**

```
http://ux-server02:8080/cas/serviceValidate?service=http://ux-
desktop03/geoserver/j_cas_spring_security_check&ticket=ST-6-
3WsfnsTJ41qi4Y95cjre-cas&pgtURL=https://ux-
desktop03/geoserver/j_spring_cas_security_proxyreceptor
```

The CAS server executes the following steps on receipt of this request.

1) Check if proxy granting ticket URL is HTTPS. If not return successful authentication with no PGT

2) Validate the ST and on failure end the procedure and return error information in XML format.

3) Try a simple HTTPS GET without any parameters with the pgtURL and verify if the response is empty. On failure return as described in 2).

4) Check the certificate of the requesting web application. The certificate must be a server certificate and the cn (common name) must be the name of the server in the pgtURL. Otherwise return as described in 2)

5) Execute an HTTPS GET with the pgtURL containing two parameters: the first is **pgtId** holding the PGT and the second **pgtIow** with an opaque String. On error return as described in 2)

6) Return the successful result in XML format. The result contains the user name and the **pgtIow** (Iow stands for “I owe you”)

The result looks like:

```
<cas:serviceResponse                          xmlns:cas='http://www.yale.edu/tp/cas'>
   <cas:authenticationSuccess>
      <cas:user>myuser</cas:user>
        <cas:proxyGrantingTicket>PGTIOU-85-u4yeb9WJIRdngg7fzl523Eti2td
        </cas:proxyGrantingTicket>
   </cas:authenticationSuccess>
</cas:serviceResponse>
```

Before this answer arrives the callback has been processed. Seen form the web application the following steps must be taken.

1) Send the validation request

2) An incoming request on the call back URL https://ux-desktop03/j_spring_cas_security_proxyreceptoor is received. An empty response must be sent.

3) An incoming request on the call back URL with two parameters is received: pgtIou=**PGTIOU-85-u4yeb9WJIRdngg7fzl523Eti2td** and pgtId = **PGT-330-CSdUc5fCBz3g8KDDiSgO5osXf**. The second parameter is the PGT. The application must store both values.

4) The XML result described above is received. This result contains the pgtIou value that is must be used to obtain the PGT received from 3).

A PGT allows for generating (proxy) tickets outside the CAS server. The SSL callback should avoid misuse of these kind of tickets.

3.5.2 *Single Logout*

3.5.2.1 Description

Single logout logs a principal out of all services registered with the CAS Server as participating in the CAS SSO. Again two scenarios are possible.

If the web application triggers the logout the following steps are necessary.

1) The principal logs out locally and is redirected to a logout success page. On this page a button for logging out from CAS is presented (There are different possibilities how to design the GUI).

2) If the user clicks the button a logout request is sent to the CAS server which in turn invalidates the TGT and TGC and sends a logout request to all registered services.

3) All web applications receiving the request from 2) execute a local logout.

If the principal logs out directly from the CAS server the following steps are required.

1) The principals logs out at the CAS server, the TGT and TGC are invalidated, and the logout request is sent to all registered services.

2) All web applications receiving the request from 2) execute a local logout.

3.5.2.2 Protocol elements

The “/logout” URI has only one parameter and is needed for triggering the single logout:

- **url** (optional)
  After successful logout the CAS server presents a logout page. If the “url” parameter is specified the logout page should contain this URL.

The CAS server notifies all services by sending an HTTP POST request to the URL of the service and including a parameter named **logoutRequest**. CAS uses XML elements from SAML (Security Assertion Markup Language) which is covered in another chapter. The HTTP body contains:

```
<samlp:LogoutRequest ID="[RANDOM ID]" Version="2.0"
IssueInstant="[CURRENT DATE/TIME]">
   <saml:NameID>@NOT_USED@</saml:NameID>
   <samlp:SessionIndex>[SESSION IDENTIFIER]</samlp:SessionIndex>
</samlp:LogoutRequest>
```

The [SESSION IDENTIFIER] is identical to the ST (service ticket).

3.5.3 Stateless services filter

3.5.3.1 Using Proxy Tickets

Stateless services in CAS can be realized using the proxy ticket architecture. A proxy with a valid PGT is needed. The proxy may be played by the same web application or by a different one. For simplicity this work assumes a proxy application running under “http://ux-desktop03/geoserver“ and the proxied application running under “http://ux-desktop03/geoserver/stateless_service”.

The proxy application has a PGT as described above. To get a PT (proxy ticket), the following request to the CAS server must be issued.

```
http://ux-server02:8080/cas/proxy
targetService=http://ux-desktop03/geoserver/stateless_service&
pgt=PGT-330-CSdUc5fCBz3g8KDDiSgO5osXf
```

The CAS server responds with

```
<cas:serviceResponse>
   <cas:proxySuccess>
      <cas:proxyTicket>PT-957-ZuucXqTZ1YcJw81T3dxf</cas:proxyTicket>
   </cas:proxySuccess>
</cas:serviceResponse>
```

The proxy application sends the following request to the proxied application, adding a ticket parameter.

```
http://ux-desktop03/geoserver/stateless_service?ticket=PT-957-
ZuucXqTZ1YcJw81T3dxf
```

The proxied application has an CAS filter in its chain looking for requests containing a “ticket” parameter. The value should start with “PT-”.

If such a request is received the filter validates the proxy ticket with

```
http://ux-server02:8080/cas/proxyValidate/?
targetService=http://ux-desktop03/geoserver/stateless_service&
ticket=PT-957-ZuucXqTZ1YcJw81T3dxf
```

On success the answer contains the user name and the URL of the proxy callback.

```
<cas:serviceResponse xmlns:cas='http://www.yale.edu/tp/cas'>
   <cas:authenticationSuccess>
      <cas:user>myuser</cas:user>
      <cas:proxies>
        <cas:proxy>https://uxdesktop03/geoserver/j_spring_cas_security_proxyreceptor
         /cas:proxy>
      </cas:proxies>
   </cas:authenticationSuccess>
</cas:serviceResponse>
```

3.5.3.2 Protocol elements for proxy tickets

The “/proxy” URI requests a proxy ticket. Parameters are

- **targetService** (required)
  The target service. This does not necessarily have to be an URL, the target may be any type of software providing services. This parameter must match the service parameter used in “proxyValidate”.

- **pgt** (required)
  A valid PGT.

The “/proxyValidate” URI validates proxy tickets and service tickets. Its parameters are equal to “/serviceValidate”.

3.5.3.3 Entry Point

If the authentication token has expired from the cache or the proxy ticket does not validate successfully an HTTP 401 (UNAUTHORIZED) is sent back.

3.5.3.4 Cache key derivation

The proxy ticket PT is a one time ticket comparable to the service ticket ST. The proxy application is responsible for acquiring a new proxy ticket before a cache time out occurs. The cache key is the ticket value. If the ticket value is not in the cache for the first request the CAS server is contacted for validation. For all subsequent requests the ticket is retrieved from the cache.

3.6 Stateless SAML Filter

The Spring Security SAML module is still under development. Some documentation can be found at[11].

All login / logout profiles would work for a stateless service but the authentication procedure must be executed for each request. Caching is required.

A SAML assertion has optional condition elements to limit the validity of an assertion. (NotBefore and NotOnOrAfter). If present, the caching time of an authentication token could be calculated, if not, a default value must be used.

The cache key could be the session index contained in the authentication statement, or in case of an artifact, the artifact itself.

Finally, the ideas described above have yet to be evaluated in practice. The implementation is deferred until the SAML Spring Security module gets release status and is added to the Spring Security framework officially.

4 Implementation

This chapter describes the practical implementation of the concepts introduced in Chapter 2 and 3. Geoserver has to support both kinds of services. OGC Web services are stateless, but there is an administration GUI needing HTTP sessions.

Looking at the big picture, a secure authentication mechanism alone does not result in a secure system. There are other major issues like programming guidelines, rules for password creation, password protection and more.

4.1 RBAC

Geoserver offers a user/group service for managing users, groups and group membership. The default implementation uses XML for persistent storage. An JDBC implementation is also available, storing data into a SQL database. The service is designed as a plug in, open for future implementations.

Additionally, a role service implementation exists for managing roles and for assigning roles to user names and group names. Again, the default implementation is based on XML, an JDBC alternative may also be chosen, the plug in design offers the possibility of alternative implementations.

Role hierarchies are supported (Single inheritance) . A user having a specific role assigned has also all ancestor roles assigned.

Role calculation for an authenticated user:

1) Find all groups where the user is a member
2) Look up the roles for a user and the roles for his groups, build the union set
3) Search all ancestor roles and add them to the set
4) Add a special role "ROLE_AUTHENTICATED", indicating that the user has authenticated successfully

Maps and vector data (streets, points of interest, boundaries) are protected by roles.

An additional feature are role parameters. A role called "employee" may have a role parameter "employee number". During role calculation, the role parameter gets its concrete value, the employee number.

Some authentication procedures do not need one (or both) of these services. A counterexample is a proxy server doing the authentication and calculating the roles, putting the result as header attribute in the HTTP request.

A special role ROLE_ADMINSTRATOR exits, a principal having this role has no restrictions.

4.2 Passwords

Geoserver implements password policies as plug ins. There is a standard implementation covering

- force a minimum password length
- force a mixture of lower case and upper case letters
- force the inclusion of special characters , e.g. @, ?, !,#

Adding more sophisticated policies is possible.

The Java runtime environment provides a secure random generator. Geoserver uses a password generator with an alphabet of 92 human readable characters. If the system needs a generated password, a password with the length of 40 characters is generated. The effective password strength is

$\ln(92^{40}) / \ln(2) = 260.942478242$

This is a little bit more than the key strength of an AES-256

Geoserver provides digesting of passwords. A random salt of 16 bytes is used, the iteration count is 100000. It is also possible to use encrypted passwords, but not recommended.

Most Java runtime environments use native libraries (mostly written in the C programming language for such classical algorithms)

The security code always tries to use Java character arrays instead of Strings and to scramble these arrays after a password is not used any more.

4.3 Encryption

The jasypt library [21] is used for encryption. During the bootstrap process the software tries to encrypt some dummy data with AES and a key length 256. If successful, unlimited key size is given. If not, a warning is written in the log file.

On the administrator GUI, an information message about the key length is displayed.

For developers, test installations and evaluation purposes, restricted key length is not a problem. For production systems, the opposite holds true.

The system uses PBE with a random salt. Out of the box, two encryption implementations are supported. If unrestricted key length is not available, DES as algorithm and MD5 as hash function is used. If unrestricted key length is available, the algorithm is AES (256 Bit) with Cipher-block chaining, the hash function is SHA (256 Bit) . The software architecture makes it possible to inject custom implementations.

The standard installation uses restricted key length, otherwise Geoserver is not guaranteed to start. The user documentation describes the steps for production deployment in more detail.

The system makes heavy use of PBE encryption (for all purposes), but uses only passwords created by the password generator.

For each different task needing a key, a new password is generated. One task is encrypting user passwords. For each password store using encryption instead of digesting, a new key is generated and used.

Another task is URL encryption (optional). Geoserver uses Apache Wicket[20] for developing the administration GUI. Apache Wicket supports URL encryption in conjunction with Jayspt[21].

4.4 Key Store

At the time of this writing, Oracle suggests to use the JKS or the JCEKS format because the Java binding to other formats is not fully specified. JKS is not able to store symmetric keys. As a consequence, JCEKS was chosen as key store format.

For the sake of simplicity, Geoserver uses the same password for the key store and the key store entries. It must be possible to alter the key store password which implies a re-encryption of each key store entry and the whole key store.

This password is the Achilles tendon of the system because it is needed in plain text at runtime. Geoserver uses the term “master password” as alias for key store password.

Out of the box, the master password is stored encrypted in a file within Geoserver's data directory. The system uses a character array of 40 characters and a permutation for positions 1 to 40. The permutation has a cycle of 169 and is applied 32 times to the character array. The result is the key to encrypt/decrypt the key store password. This is a suboptimal solution since anybody reading the source code is able to derive the key.

A solution to this problem is to give users/ developers the chance to inject their own Java code to provide the key store password. The new class must subclass the existing class “AbstractMasterPasswordProvider” and implement a method “doGetMasterPassword()” returning the key as a character array. The code is injected using the Spring framework.

Geoserver has to protect the “doGetMasterPassword” method described above. The access modifier is “protected”, allowing access only to package members and subclasses. The jar file is sealed not allowing another jar file to inject a class in this package. Custom code sub classing “AbstractMasterPasswordProvider” must be declared final, otherwise the custom code is rejected. Otherwise an attacker could again subclass from the custom code and retrieve the master password.

The whole situation is comparable to alarm systems in cars. Professional thefts have a good knowledge about the systems of the manufacturer. A better solution is to build in an individual system.

Work flow to change the password;

1) Enter old password
2) Enter reference to the new master password provider
3) Enter new password
4) Check old password
5) Verify the new password against the password policy
6) Check if new password equals the password provided by 2)
7) Re-encrypt the key store with the new password
8) Store a digest of the key store password in the global configuration

There is a special user called “root”. This user has administrative privileges and authenticates itself by knowing the master password. If the “root” user logs in, its password is validated against the stored digest from the global configuration, no need to fetch the master password into memory.

During the start up sequence, the system fetches the master password, opens the key store and overwrites the plain text password with random bytes. The plain text master password should not be found in main memory.

4.5 Caching the authentication token

At the time of this writing, the choice of the caching component is still under discussion. At the moment, an own implementation is used. A well know possibility is the use of EHCache[3] which meets the requirements described in the previous sections. Additionally, Geoserver uses the standard Java mechanism for serializing and de-serializing authentication tokens to be prepared for a clustering scenario.

EHCache offers a distributed cache based on Terracotta libraries[3]. A replicated cache is available by using a group ware component or a messaging service. Another alternative would be a database back end. As mentioned earlier, the caching component is still under discussion.

Cache key derivation is done as described in the previous chapters. Additionally, the name of the authentication filter is used as prefix.

4.6 Filters

Geoserver supports some predefined filter chains, the chains itself are configurable.

The system uses two instances of the Security Context Persistence filter, one that is allowed to crate sessions and one that is not allowed. The first one is used for chains assigned to the GUI, the second one for all stateless services.

A special implementation for the exception translation filter exits. If the entry point is not specified explicitly, the filter uses the entry point of the last authentication filter in the chain. All Geoserver authentication filters put their entry point in the request object passing through the filter chain. The entry point saved in the request object by the predecessor is overwritten by the current authentication filter. In this scenario, the order of the authentication filters is important.

All standard authentication filters described in Chapter 3.4 are supported and it is possible to inject custom filters for special deployments.

The remember me services is not supported for stateless services. For stateful services like the administrator GUI, the remember me service can be used if two preconditions are fulfilled. First, the global remember me service must be explicitly enabled, second , the user must click on a check box on the login page to explicitly enable this feature for himself. In this case, a parameter named

“_spring_security_remember_me”

is added to the HTTP request. The remember me service only creates a cookie if this parameter is present.

Since the system supports more than one user group service, the user name in the cookie is suffixed by the name of the user group service.

user name@name of user group service

This is necessary to know which user group service is providing the password for this individual user (altering the password makes the cookie invalid).

4.7 CAS

Geoserver supports standard CAS login for stateful services by providing its own filter chain for “j_cas_spring_security_check” and an entry point redirecting to the CAS login form.

For retrieving a PGT another specific filter chain called j_spring_cas_security_proxyreceptor is used.

For single logout it is necessary to have a global data structure for mapping tickets to sessions. In the case of a logout request the session can be found by the ticket and invalidated. The Java CAS client library has built in support that is used.

For triggering a single logout by Geoserver itself a special filter chain “j_spring_cas_security_logout” is required.

Concerning stateless services, the system is capable of handling proxy tickets.

5 Conclusion and outlook

This work is a summary of the past 12 months of work studying specifications, designing and implementing the new security architecture of Geoserver. The important fact is to see the big picture, an authentication mechanism alone is not enough to secure a system. The introduced concepts and the implementation of these concepts will make Geoserver more resistant against attacks.

The plan is to have these features in version 2.2.x. Until versions 2.1.x Geoserver supports form based login for stateful web services and basic authentication for stateless web services. No password digesting /encryption is supported and roles are assigned to individual users using a flat file.

A lot of work has still to be done. Finishing and integrating all these new features, enhancing the administrator GUI to configure these features, and finishing the online documentation. The test coverage of the core modules should be greater than 90 percent.

It is also possible to configure Geoserver using REST (Representational state transfer). Work on this module has yet to be started but the developers had it in mind and the design of the security architecture has taken a future REST API into account.

There are two other interesting security architectures not covered in this work. OpenId[16] and OAuth[17]. A validation and a possible integration into Geoserver is planed for the future.

The most impressive security concept presented in this work is SAML, for which this work has only scratched the surface of. An open source implementation (called “opensaml”) is available at [18] which is used by Spring developers to integrate SAML into Spring Security. Since Geoserver is using Spring Security the team must wait until the SAML module for Spring is available.

6 Bibliography

[1] National Institute of Standards and Technology, Role engineering and RBAC standards,Specification,2008,http://csrc.nist.gov/groups/SNS/rbac/,Accessed 20.01.2012]

[2] Sun Microsystems, Java SE Security,Overview, 2012 http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136007.html#overview [Accessed 23.01.2012]

[3] Terracotta , EHCache , Overview, 2012, http://ehcache.org[Accessed 19.03.2012]

[4] Danial Fernandez,"How to encrypt user passwords", article, 2011, http://www.-jasypt.org/howtoencryptuserpasswords.html[Accessed 19.03.2012]

[5] Network working group, "HTTP Authentication: Basic and Digest Access Authentication", request for comment, 1999,http://www.ietf.org/rfc/rfc2617.txt [Accessed 19.03.2012]

[6] Network working group, "An Extension to HTTP : Digest Access Authentication ", request for comment, 1997 http://www.ietf.org/rfc/rfc2069.txt[Accessed 19.03.2012]

[7] Jasig, "CAS", overview, 2004, http://www.jasig.org/cas/[Accessed 22.03.2012]

[8] Scott Battaglia, Kim Cary, CAS User Manual, 2007, online wikipedia, https://wiki.jasig.org/display/CASUM/Home[Accessed 22.03.2012]

[9] Matt Flemming, "How Spring Security hooks to Central Authentication Service (CAS) ", online article, 2010 http://mattfleming.com/node/269[Accessed 22.03.2012]

[10] Andrew Petro, "Proxy CAS Walkthrough",wiki article, 2004, https://wiki.jasig.org/display/CAS/Proxy+CAS+Walkthrough[Accessed 22.03.2012]

[11] Vladimir Schäfer, "Spring Security SAML Module", technical article, 2009, https://jira.springsource.org/secure/attachment/15148/SpringSecurity+SAML+-+documentation.pdf [Accessed 26.03.2012]

[12] Open Source Project, "Geoserver", home page, 2005, http://www.geoserver.org[Accessed 28.0.3.2012]

[13] OpenGeo, “OpenGeo”, home page, 2001,URL: http://www.opengeo.org[Accessed 28.0.3.2012]

[14] Open Source Project, “SpringSource”, home page, 2003, http://www.spring-source.org[Accessed 28.0.3.2012]

[15] OASIS Security Services TC, “Security Assertion Markup Language (SAML) V2.0 Technical Overview ”, Committee Draft, 2008, http://www.oasis-open.org/committees/download.php/27819/sstc-saml-tech-overview-2.0-cd-02.pdf[Accessed 30.0.3.2012]

[16] OpenID Foundation, “OpenID”, home page, 2007, “ http://openid.net/[Accessed 30.0.3.2012]

[17] OAuth Community, ”OAuth”, home page, 2007, http://oauth.net/[Accessed 30.0.3.2012]

[18] Open Source Project, “OpenSAML”,online wikipedia, 2008, https://wiki.shibboleth-.net/confluence/display/OpenSAML/Home[Accessed 30.0.3.2012]

[19] Sun Microsystems, “Java Cryptography Architecture”,online documentation, 2006, http://docs.oracle.com/javase/6/docs/technotes/guides/security/crypto/Crypto-Spec.html[Accessed 01.0.6.2012]

[20] Apache Foundation, “Apache Wicket”,home page, 2007, http://wicket.apache.org/[Accessed 01.0.6.2012]

[21] Open Source Project, “jasypt”, home page, 2007,http://www.jasypt.org/[Accessed 01.0.6.2012]

[22] Open Geospatial Consortium , “OGC”, home page, 1994,http://www.opengeospatial.org/ogc/history [Accessed 01.0.6.2012]

[23] Apache Foundation, “PatternSet”, online manual, 2002,http://ant.apache.org/manual/Types/patternset.html [Accessed 01.0.6.2012]