

# GSA Valve Security Framework – Introduction

## Google Enterprise EMEA

Strategic and secure information sources are naturally becoming key repositories that customers want to make searchable. Since search is a platform and not just an application commodity, it is a pre-requisite to cope with heterogeneous security systems in order to seamlessly roll-out an Enterprise-wide secure search capability.

The search appliance can nicely integrate with popular single sign-on (SSO) systems, forms-based authentication systems and negotiate HTTP Basic and NTLM authentications/authorizations. However, it can't substitute the need for a unified authentication process across all secure sources. Heterogeneous sources may require different sets of credentials which make the search experience painful for the end-user, having to authenticate multiple times when querying.

Additionally, the appliance may not be able in certain cases to cope with complex and non-standard authentication and/or authorization processes which may put at risk the deployment of search technology.

The GSA Valve Security Framework was designed to answer both of these issues. It exposes a global authentication capability to the search user and then loads transparently the sets of credentials that are relevant to each indexed sources. It is a framework that can easily be extended to support the specifics of new repositories in terms of authentication and authorization processes. You can easily extend it creating new authentication and authorization modules that offer new capabilities making the Security Framework fully working with your custom content repositories.

This authentication and authorization framework acts as a content proxy and can be considered as a quick, simple and low-cost alternative to a single sign-on (SSO) system. It can be integrated as well with third-party SSO solutions in those situations where the corporate SSO doesn't secure all the searchable applications.

This framework supports two different interfaces for serving content in the search appliance:

- **Forms Based Authentication (Web Single Sign-On):** the primary frontend that was supported from the beginning was Forms Based Authentication as it's known in the GSA terminology. It really acts as a Single Sign-On, as the user has this experience and all the request are treated by this frontend.
- **SAML (Security Assertion Markup Language):** this is a security standard supported by the GSA. Since GSA Security Framework version 2, it's offered a SAML-enabled frontend.

The first decision you have to take when deploying this framework is which frontend is going to be used in your environment, as only one can be activated. We recommend you the use of the SAML interface as this is much better integrated with the search appliance so far. SAML will help on the support process as well due to the multiple interactions with the appliance that let the support engineers to have a better understanding on the status of the authentication/authorization process. Anyway, you can use any of these two interfaces and you will find in the documentation the differences between using Forms Based Authentication or SAML.

Another important thing to remark is this framework is not only valid for serving but also for crawling. In case you chose a Forms Based Authentication, it's fully integrated with the GSA (Google Search Appliance) using the Forms Based crawling feature that let the appliance to take the control during this stage. If you took the SAML approach though, the GSA native feature for crawling HTTP Basic and NTLM (Windows Native Authentication) can be extended using a crawling server created ad-hoc for those systems that don't support any of these technologies.

The Valve, a standard Java web application, resides with an application server, in this case Tomcat, and is configured as an application within it. When setting Forms Based Authentication approach up, a server filter as a Tomcat Valve, hence its original name, is configured the application server in order to manage HTTP request either coming from the appliance or the users. A **Valve** element within Tomcat represents a component that will be inserted into the request processing pipeline for the associated Catalina container ([Engine](#), [Host](#), or [Context](#)). Individual Valves have distinct processing capabilities, and in this case its main function is to evaluate a user's permission to access a piece of content.

The valve, as any other Java application running on an application server, is configured to respond in the processing pipeline of a specific URL, for example <http://valveServer:<PORT>/>

The way the content is accessed by the GSA is different in Forms Based and SAML approaches, and that's why the crawling methods generally described above are also different:

- **Forms Based Authentication:** the authorization decision has to be centralized and that's the reason why we have a central server (or set of multiple servers that process the same content), the Valve, that serves the content securely. The GSA is configured to then access all content via its URL. For example if the content to be indexed was at <http://server1/pageone.html> then it would be accessed via <http://valveServer/valve?returnPath=http://server1/pageone.html>. These URLs are used mainly for crawling and authorizing and it's a good practice not to expose these URLs to final users when serving. This can be done changing the User Interface as it's explained in the documentation.
- **SAML:** as the architecture of this well-known security standard is natively centralized, there is no need then to always access through the same URL that serves the secure content. The GSA can send SAML authentication and authorization request messages to a SAML server like the Valve provides, and the content URLs keep the same. In the example mentioned above, the URL will be then <http://server1/pageone.html>.

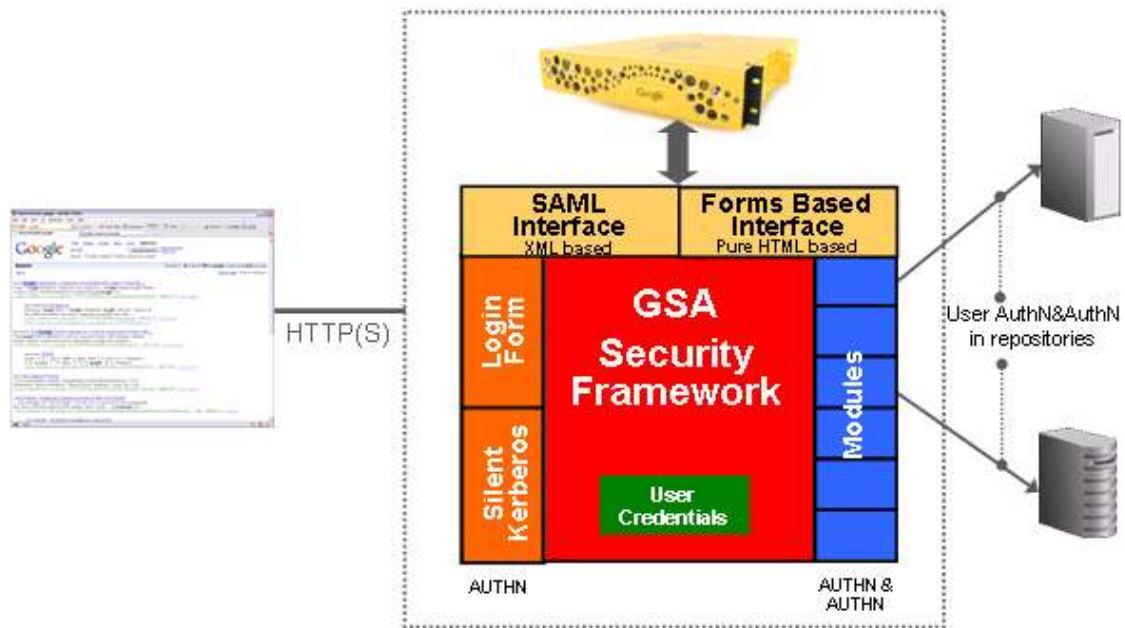
The valve provides a framework that allows classes to be defined which are responsible for providing authentication (AuthN) and authorisation (AuthZ) for a specific URL pattern. In the above example classes would be required to support both authentication and authorisation for <http://server1>. When the valve receives a request it matches the referrer URL against a list of configured patterns and then uses the defined authentication and authorisation classes for that pattern. This allows the valve to support authentication and authorisation against multiple systems, each using different authentication and authorisation techniques.

This solution already offers some AuthN/AuthZ modules that can be easily configured to protect applications thanks to its XML-based configuration file. If the Valve does not include a module to integrate a specific application or protocol, you can easily extend the framework to support it. That's the framework philosophy behind it so that it does not offer an out-of-the-box solution for all cases.

The overall framework architecture is represented in the following picture. These

extendable integration modules are shown here on the right hand side. They do the authentication and authorization processes and manages the integration with the content sources. The URL pattern will tell the framework which security module has to be thrown as it was explained above.

On the left hand side of the GSA Security Framework picture you can see the authentication frontends. A frontend could be either a login form, i.e. a username/password driven authentication page, or silent Kerberos. Don't get confuse here as these authentication mechanisms are independent of using either SAML or Forms Based Authentication as a GSA security integration technology.



The Google Search Appliance documentation is available online as well as some other very interesting technical articles at <http://code.google.com/enterprise/> (Google Search Appliance section). They are definitely worth reading documents and we do recommend having them a look for a better understanding on the product. It will help you on learning the most of this security framework documentation.

### How it Works

The Valve receives the authentication and authorization request coming from the appliance and performs a number of different actions depending on the URL requested. The valve allows classes to be defined for AuthN and AuthZ using patterns to match the URL in the request. A pattern can be associated to one or more sites, so that it means they are protected by the same security modules.

The way the framework behaves is completely different if we are using the Forms Based or the SAML interface. The first one is driven by a session cookie, so when the appliance detects the absence of this cookie, the authentication process is triggered. The SAML interface is completely independent of the existence of that SSO authentication cookie though, as the appliance directly authenticates users sending XML messages against the SAML server. In this guide you will find an explanation on how each interface works.

## A- Forms Based Interface

The Forms Based interface makes use of cookies as part of the process flow and the default root authentication cookie is `gsaSSOCookie`, although its name can be configured. If the `gsaSSOCookie` does not exist then the valve will send a redirect to start the authentication process. This process could be for example a login page, in the case the user has to be prompted for a username/password or another server component that silently authenticates the user using for example Kerberos. Each user can have multiple credentials of whatever kind.

Using this security interface, the search appliance just relies on this external session cookie and don't have the information who the user really is. The only thing the appliance is aware of is the multiple cookies that are coming from the user browser and are reused for the authorization.

The valve defines two root classes, one for AuthN (`RootAuthenticationProcess`) and AuthZ (`RootAuthorizationProcess`). These two classes implement a rule based switch statement. A rule would be defined for each external site and the appropriate AuthN and AuthZ classes to deal with the specific site.

You can find below the situation when the user is accessing the Valve security environment either with the authentication cookie or without it.

### Without `gsaSSOCookie`

If `gsaSSOCookie` does not exist the request is redirected to the login page or whatever any other authentication process, as defined in the configuration. The login page is typically a JSP page and can be change to suit the needs of the environment. Typically as a minimum it would ask a user for a username and password.

The `gsaSSOCookie` is created by the authentication servlet that is called by the login page if it exists, and then processes any authentication class. This is automatically done by this `RootAuthenticationProcess` class based on the repository definition in the configuration file (by default its name is `gsaValveConfig.xml`) that is located at `Tomcat_HOME/common/classes`. Each repository has its own entry in the configuration file the following way (this is an HTTP Basic protected repository sample):

```
<repository id="HTTPBasicAuth" pattern="http://server.google.com:90"
  authN="com.google.gsa.valve.modules.httbasic.HTTPBasicAuthenticationProcess"
  authZ="com.google.gsa.valve.modules.httbasic.HTTPBasicAuthorizationProcess"
  failureAllow="true" checkAuthN="true">
  <P N="HTTPAuthPage" V="http://server.google.com:90"/>
</repository>
```

Each of these AuthN classes defined in the "authN" attribute would Authenticate against the external site and generate any additional cookies that might be required. These cookies can be sent back to the browser and be reused during the authorization process as well.

### With `gsaSSOCookie`

If the `gsaSSOCookie` exists, which means a user has logged in and been authenticated, the Valve processes the request based upon the URL pattern that is defined in the

returnPath. Rules defined in the configuration file and treated by the RootAuthorizationProcess class defines which classes should be loaded for each match URL in the return path.

The authorization classes are specified in the repository “authZ” attribute, as you can see in the previous sample.

### **B- SAML Interface**

The SAML interface shipped with the security framework works slightly different than the Forms Based one. The main reason is the whole security process in this case is not driven by an external authentication cookie, although a session cookie is finally created at the end of the authentication process by the GSA to take control over the client communications. In this case, the GSA does know who the user is it keeps the user ID during the whole security cycle that is reused for every authorization process.

The way the GSA is integrated with a SAML server is perfectly described in the following guide (GSA 5.0 release):

<[http://code.google.com/apis/searchappliance/documentation/50/authn\\_authz\\_spi.html](http://code.google.com/apis/searchappliance/documentation/50/authn_authz_spi.html)>

The same integration scenario as the one described in the above GSA guide applies for this framework. Take into account that the SAML Authentication and Authorization processes can be treated separately, for instance we can use a non SAML authentication mechanism using the HTTP Basic feature natively implemented in the Google search platform and a SAML-based Authorization SPI at the same time. This security framework provides a SAML solution that has to be used both for authentication and authorization.

When a user makes a secure search and the appliance does not have the GSA session cookie (or the session is timed out), the user is redirected then to the login process defined in the console at Serving -> Access Control. During that process, this framework authenticates the user, getting the credentials and launching the rootAuthenticationProcess that reads all the repositories declared in the configuration file the same way as in the Forms Based interface, and once the user is successfully authenticated, redirects him/her back to the appliance. Finally the appliance asks the framework what the user ID is so that it can be reused during the authorization process.

The Valve framework provides a SAML Authorization SPI as well that drives the authorization process, handling the authorization request coming from the appliance that contains both the user ID and the content resources that match with the original query, so that all the search results are served securely.

### ***Application structure***

GSA Valve Application Framework is using different classes that implement the whole processing. In this section we'll see how the application is organized and how these classes are structured.

#### **Valve**

Valve is the component that filters the request and controls the whole application flow when

using the Forms Based interface. It's defined in the application server as a "Valve" component that intercepts every request made to the application and applies the right patterns for authenticating and authorizing users.

### Authentication Servlet

This is the component that interacts with the user authentication process. It is responsible for authenticating users based on their main credentials (username/password, client certificate, Kerberos ticket, etc.) There is currently a couple of authentication servlet implemented in the Valve, but you can even extend it and create your own one:

- *Authenticate*: username and password credential servlet that is invoked by the login page.
- *Kerberos*: this component implements the Kerberos scenarios supported by the Valve.

This component, located in the same place as the Valve class at `com.google.gsa`, finally invokes the next one to follow with the authentication process.

### Root Authentication Process

The class `RootAuthenticationProcess` governs the whole authentication mechanism in the Valve and it's fed by the repository definition in the configuration file. Once this component has the user main credentials provided by the authentication servlet, it controls the authentication process, calling each individual AuthN mechanism defined in each repository. This process can also manage more than one user credential, as for example one of the modules can read new user credentials from a database or LDAP, and use them to authenticate to one of the sources. For example, the user credentials (a pair of username and password) for Documentum can be completely different than those to access to Windows resources, but the user only provides the last ones to the Valve.

This class is located in the code at `com.google.gsa.valve.rootAuth`, at the same place where the root authentication process is.

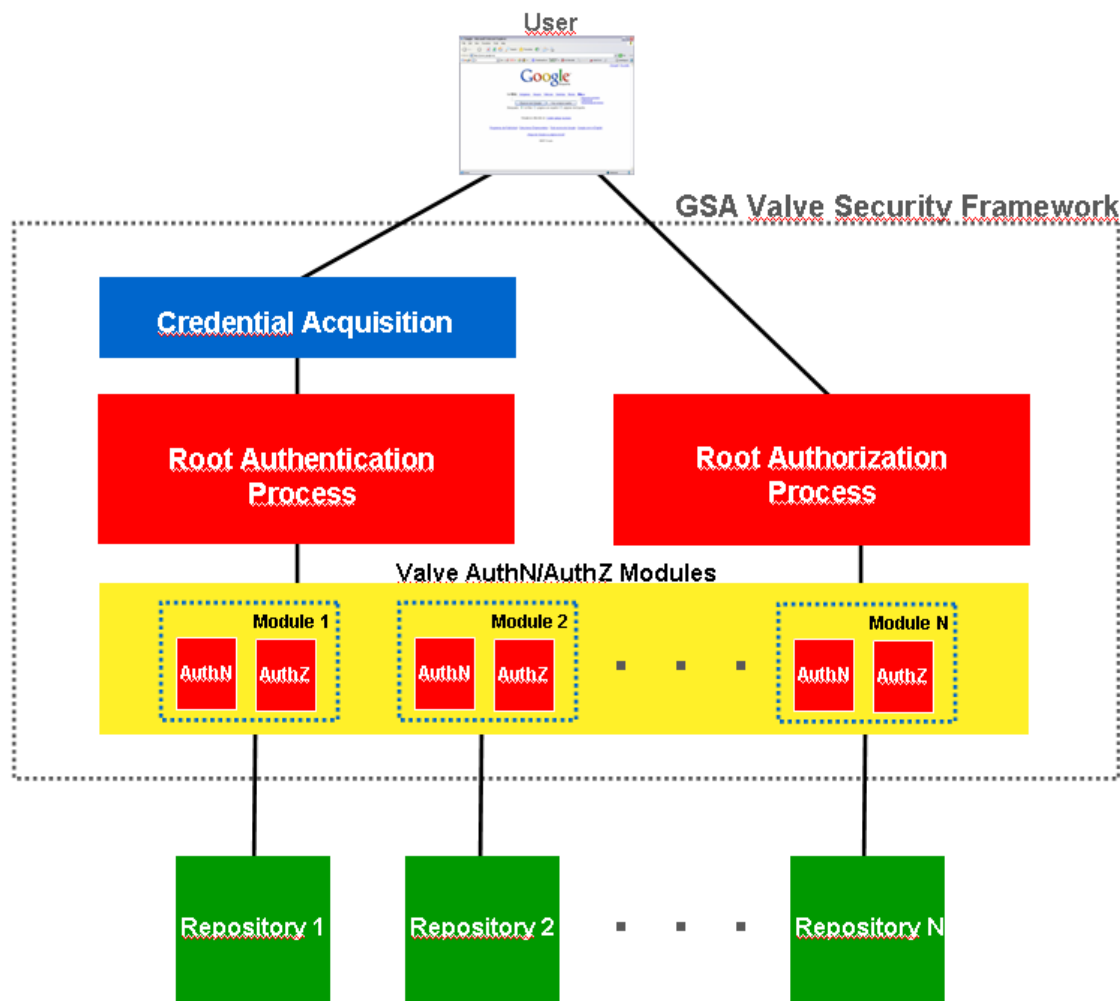
### Root Authorization Process

The authorization process starts once the user is already authenticated and is accessing a document. The general authorization processing is managed by the Valve's `RootAuthorizationProcess` class that validates the pattern associated to the URL and launches its authorization process. Finally it sends back the response to the user. Whereas the root authentication process only happens once when the user is not authenticated, the authorization will be processed as many times as the GSA/user have to check permissions in the remote repository.

Both the authentication and the authorization processes are very generic that support multiple configuration options, but you are able to create your own classes that manages the security cycle in the Valve framework.

Next picture represents the main internal functional components in the GSA Valve and how they do interact with each other. You can see that the authentication and authorization process contact the repository modules, but they just interact either with the module's AuthN or AuthZ component respectively.

The root authorization process is called from the Valve filter when using the Forms Based interface or from the Authorization servlet (SAMLAuthZ) when SAML interface is in place.



### ***AuthN/AuthZ Modules***

The integration with the content sources is done through Valve modules that include the authentication and authorization processing. The security framework already provides some technical modules like HTTP Basic or Kerberos that can be easily used in your configuration to secure the access to sites protected to these technologies. You can see how to configure most of them in the scenario guide.

You are also able to define your own modules that implement the integration with your repositories. This can be easily done using the same development rules as they were used for the existing modules, in fact you can use them as a model to create new ones. The main difference resides on the specific coding complexity to interact with your content source. We recommend you to use one of the existing modules (a good one could be the HTTP Basic module) as a reference to create your own modules.

Security modules are located behind `com.google.gsa.valve.modules` package and each of them has to provide one class for authentication and another one for authorization for the following purposes:

- *Authentication*: it is the responsible for authenticating the user against a specific application or environment. This process can take the credentials from the root authentication process and use them, or even get new ones, to start the authentication process. At the end it creates a cookie that helps the module to know if the user was already authenticated.
- *Authorization*: this process is pretty common for all the modules as it just checks if the user is authorized to access to documents based on his/her credentials. It really depends on the application but usually just sending the cookies created during the authentication it's more than enough to authorize as well.

Regarding the modules already created in the Valve you can find, among others, the following:

- *LDAP processing*: they are just authentication modules and they are basically:
  - *LDAP Unique credentials*: it checks user credentials with the LDAP users.
  - *LDAP SSO process*: this class is being designed to get additional user credentials from the directory. They are stored in some LDAP attributes and this class just reads and includes them in the user credentials. You can define the repositories that are going use them. You can also encrypt/decrypt these attributes using your own algorithms.

They can be used both with Active Directory and third-party LDAP servers as well.

- *HttpBasic*: it authenticates and authorizes using http basic credentials.
- *Kerberos*: it's able to authenticate the user using Kerberos credentials completely transparent for users. It also does http authorization using the Kerberos ticket.

You can have more than one Authentication/Authorization module associated to a URL pattern, this way we can have multiple security checking against the same content source. The way it works during authorization is as follows:

- The framework looks for those patterns defined in the Valve modules that matches with the URL that we are actually checking authorization.
- Those authorization modules (Java classes) that match with the URL are executed in order.
- If any of these modules returns an unauthorized response, then the final result is the user is not able to see that document and the next authorization modules are not checked at all.
- If all the modules return an OK, then the user will be able to access to the content.

The kind of Authentication/Authorization module you can build up could be for instance one the following types:

- *ACLs*: if you have internal corporate rules that can be used to infer the security policies in some of the content sources, you can implement a kind of ACL (Access Control List). This is usually implemented in a central LDAP or database where the users are included in corporate groups that can be used to permit or deny the access to a concrete document or system based on that group membership.
- *Custom security implementation*: if any of your backend content sources is using a custom security implementation that resides in the application, you can create an Authentication/Authorization module to let the search infrastructure access securely to that application. It also includes if your application is using a specific protocol that is not managed by the search infrastructure by default.
- *Adapt authorization responses*: it's a common issue in some corporate applications and portals, when the user is not authorized to access to a

particular document, the backend system sends back no HTTP error message as it returns an OK (200), but it contains an unauthorized error report in the HTML. You can create a custom module that deals with this scenario sending the appliance back an unauthorized error when this problem happens.

- *SSO integration*: multiple enterprise applications provide their own security module that is tightly coupled with these applications. It means that they have their own SSO authentication cookie that has to be presented when authorizing. As a consequence of this situation, we can have multiple SSO systems that complicate the integration solution. These applications have usually the option to be integrated with a corporate SSO that acts as a meta security solution that avoids this issue. If this is not your case, you can implement a security framework module that deals with this situation and would be able to manage the authentication cookie for every application.

## Conclusion

The GSA Valve Security Framework comes as a Java framework and is ready to use. It provides the end-user with a single-sign on experience and as such implements a customizable and global authentication process.

Acting as an umbrella on top of secure sources, it handles heterogeneous security systems by encapsulating the authentication and the authorization processes in two separate Java classes loaded dynamically.

This framework is easily extensible. Adding a new secure source consists in declaring a new URL pattern and in replicating the matching authentication process to validate the end-user credentials and the matching authorization process to validate the access to the content. Facilities for reproducing the HTTP sequence of a Web front-end and introspecting the returned HTML pages are provided so that integrating a new system may be a question of hours to a few days.

This controlled framework ensures the compatibility with the search appliance security model and a quick response-time in implementing/deploying a scalable and unified secure search solution.