GSA Valve Security Framework – Scenarios

Google Enterprise EMEA

This document explains in detail different technical scenarios that can be implemented using the GSA Valve Security Framework, an authentication and authorization solution designed for the Google Search Appliance (GSA).

This security framework, also known as The Valve, acts as an impersonation solution in which the search appliance delegates the user authentication and authorization, as it fully integrates with the content sources and check if every user has access to any document. It makes complex project deployments much simpler but keeping high security requirement levels, as each content repository is still responsible for authorizing users, so that every decision is taken in the same source the content really is.

You'll find here a technical explanation on each deployment scenario and how to configure the security framework for them. There are different possibilities including the option to mix heterogeneous credentials that are kept in desultory repositories. This open source solution brings out much more possibilities when deploying complex projects driven by security policies.

There are also some other technical documents about this framework that you may find useful to use in conjunction with this guide. They worth a read as they explain in detail some of the concepts managed here.

How does the Valve manage credentials?

The GSA Valve Security Framework is able to authenticate users and provide a Single Sign-On (SSO) experience. It has to collect user credentials and check them are valid before creating an internal user profile that drives the whole security process. A credential is something that uniquely identifies the user in every system, like for example a pair of username and password or a Kerberos ticket. This security framework is able to manage multiple credentials through a credential vector created in the authentication process based on the declarative configuration information.

If you set up authentication through a login form, the Valve always creates a credential store named "**root**" where the main username and password are kept. It drives the whole authentication process as they can be reused to access to any source repository. The definition of a standard Valve repository is done in the configuration file (`gsaValveConfig.xml`) with the *<repository>* tag, for example:

```
<repository id="SampleRepository" … />
   …
</repository>
```

The standard authentication/authorization processes defined in the Valve always try to find a credential container stored in that vector that matches the name of its repository "id" attribute (in this example is *SampleRepository*). This is done to support multiple credentials and that "SampleRepository" could be populated in any part of the

authentication process as for example in any other repository that adds new credentials to the vector. Imagine such a process is driven by a custom Valve authentication module (you can have a look at the other guides on how to create new AuthN/AuthZ modules) that accesses to a central credential repository like a database where all the user credentials are kept. We can feed the authentication process just recovering the application credentials each user has and then reuse them very easily to authenticate them with the specific application username and password. In order to reuse those credentials without hard coding the module you just have to populate the same credential id as the repository name.

In the case the credential *SampleRepository* does not exist, then the authentication process will reuse the "root" ones. That's why the credentials got in the forms based authentication login page drives the whole process. In the case you name a repository as "root", it means this is the central authentication step and it's usually associated to an LDAP authentication process. If the "root" authentication fails it means the overall process is stopped here.

There is also another special credential implemented in the GSA Valve that holds Kerberos tickets. Its keyword is "**krb5**" and works exactly the same way as the "root" one. You will find more detailed information in this document about how to implement different scenarios with these credentials. One important thing to mention is you can name your repository IDs whatever you want but you should not name it as "krb5".

Kerberos

Kerberos is a network authentication mechanism used to facilitate the access to the services, as the identities are sent through the network in a completely transparent way to the users. Kerberos makes life easy to the final users but it introduces complexity in the environment. That's why setting Kerberos up with an application like the Security Framework is not easy and sometimes you can find it challenging.

Since this technology was adopted by Microsoft Windows 2000 as a native authentication mechanism, this authentication protocol has been extensively used. Most of the existing applications, not necessarily only those running on the Microsoft platform, currently support Kerberos as an authentication mechanism. These Kerberized applications have to be integrated as well with the search infrastructure, so that the same user experience can be extended to the search engine offering an easily accessible and secure solution.

The GSA Valve Security Framework supports Kerberos silent authentication and is able to also impersonate users just reusing user's tickets, as it also authorizes them against the content sources based on their Kerberos identity, guaranteeing a high security level.

In this document you will find some scenarios with Kerberos in place from a complete Kerberized environment to a scenario where the Kerberos ticket is created in the Security Framework using corporate username and password per user without having a transparent negotiation in place.

One important thing about Kerberos tickets is they require a maximum age so that they are reused from time to time. This is implemented in the security framework using sessions.

You can find more information about how to configure the security framework to manage this authentication technology in the GSA Valve's Kerberos Configuration guide.

Sessions

The GSA Valve Security Framework natively supports sessions where the user authentication information is kept. This information is used then to authorize users when checking permissions in the source repositories. They are very useful for example in the following scenarios:

- *Kerberos is in place*: sessions are always used when Kerberos is set in the security framework. It has to impersonate and reuse user's tickets when authorizing. It can be extended to other authentication technologies that would require reusing credentials exactly the same way as Kerberos.
- *SAML interface is being used*: if you are using the Valve's SAML frontend, sessions are needed to be used as the authentication information like credentials or cookies has to be available during authorization. As the SAML authorization request does not send cookies, but just the user ID, it's important to keep that information to be reused during the authorization checking process.
- *Corporate security policy compliance*: if there are internal policies that either requires to be logged off from corporate applications from time to time or they avoid sending multiple cookies back to the browser.
- *Local application*: if you have a searchable application that locally manages sessions, then you can use Valve's sessions to guarantee the access to this application in the same conditions.

Sessions can be implemented just configuring the <Session> tag in the configuration file and internally implements authentication persistence information using Java technologies. In the configuration guide you can find the definition of the different Session parameters. A sample session configuration section in the Valve configuration file would look like as follows:

```
<sessions isSessionEnabled="true"
          sessionTimeout="-1"
          maxSessionAge="300"
          sessionCleanup="30"
          sendCookies="true"/>
```

This configuration supports sessions that means both the user credentials and cookies are persistent internally in the security framework. They can be reused in any other part of the process when authorizing. In the above session example we have configured sessions for an unlimited timeout (i.e. maximum user inactivity time), a maximum session age of 5 hours (300 minutes), the invalid sessions are cleaned up from the session array every 30 minutes and the cookies are sent back to the browser.

Here you are some session configuration rules:

- Use sessions just whenever they are needed but avoid using them in other cases as it adds some overhead. If you want to use Kerberos have a look at the Kerberos configuration document that explains the option to use sessions.
- You can have non-limited values in both the session timeout and the maximum session age if you set them to "-1". It means neither of them that have such value will be taken into account when managing sessions.
- Regarding `sessionTimeout` parameter this should be used very carefully and only if it's really needed in case you'd have a backend application that really requires it. The reason is this parameter adds some overhead and also can change the way the SSO feature provided by the Security Framework behaves. We strongly recommend you to set this parameter to "-1" whenever sessions are enabled and would be possible based on the business requirements.
- If you set `maxSessionAge` attribute, then the `authMaxAge` parameter has to match with that value. One important difference is `maxSessionAge` is

established in minutes whereas `authMaxAge` is done in seconds, so you should do the translation, for example:

```
…
<authMaxAge>24000</authMaxAge>
…
<sessions …
          maxSessionAge="400"
/>
```

- If "sendCookies" is set to "false", it means the authentication cookies apart from the security framework session one (in the above example this is name as gsaSSOCookie), are not sent back to the browser and only those kept in the session are used to authorize users. Set this attribute to "true" if your backend repositories manage cookies and you want to have a SSO experience when clicking on the result list.
- In the case you wouldn't like to use sessions in place, just set up isSessionEnabled attribute to "false" (you can delete the others are as immediate discarded –excepting maxSessionAge when using Kerberos-). You will see some examples later on.

The above example can be extended to any of the scenarios shown in this document as the use of sessions is optional for the majority of the cases except those explained in this documentation. So consider the use of session in the deployment scenario best suits for you and take into account its advantages and disadvantages like the overhead and complexity it adds.

Crawling

This guide describes the scenarios mainly for serving content but it's also important to consider how the crawling process is tackled in each scenario as the Security Framework also supports crawling in some scenarios. You can discover the available documents in your environment in different ways like for example feeding XML messages that contain the information where the document is located; include some metadata on it or even injecting the content itself. This process can be also completely managed by the GSA through a Forms Based Authentication application like a corporate Single Sign-On (SSO) server. This is the case as well when you are using the Forms Based interface in the Valve Security Framework.

You can find more information on Crawling in the security framework's installation document when **SAML interface** is in place. Using this SAML frontend, there are multiple choices as the URLs are not rewritten and that's why this guide does not explain the options the content could be crawled. You have to take a decision in here based on the GSA features and your application security. All the several scenarios shown here in action that uses the SAML frontend are explained from the serving point of view.

In the case you were using the Valve's **Forms Based interface**, you can crawl using this Security Framework. This is done declaring a Forms Based Authentication rule for crawling (you can also define a similar rule for serving) in the search appliance web configuration console as you can see in the GSA Valve Security Framework configuration guide. You will see in this document how to define the crawling rules for each scenario when using this Forms based approach.

If you are using Kerberos credentials and want to make use of the GSA's Forms Based crawling services as well, you'd explicitly have to configure it in the config file. The reason behind it is the appliance can not create Kerberos tickets so we have to generate them using crawler's username and password credentials. In the Kerberos

configuration tab you can make use of the following bolded parameters when Kerberos is enabled:

```
<kerberos isKerberos="true"
        isNegotiate="true"
        krbini="c:\\config\\krb5.ini"
        krbconfig="c:\\config\\bcsLogin.conf"
        krbAdditionalAuthN="false"
        krbLoginUrl="http://securityframework.google.com:8080/valve/loginkrb.jsp"
        krbUsrPwdCrawler="true"
        krbUsrPwdCrawlerUrl="http://securityframework.google.com:8080/valve/logincrawlerkrb.jsp"/>
```

The parameters `krbUsrPwdCrawler` and `kerbUsrPwdCrawlerUrl` do not apply when using the SAML interface.

You will see in detail how to define crawling rules in every scenario explained in this document when Forms Based crawling is chosen.

Integration Frontends

As it is being mentioned in several parts of the Valve Security Framework documentation, you have the choice to use either a **Forms Based interface** or a **SAML interface**. The first frontend is the default one, so the scenarios shown below will be just using this interface. If you want to use the SAML interface instead, you just have to modify the following configuration to include the SAML attributes and the sessions. For example:

```
<!-- Session config -->
<sessions   isSessionEnabled="true"
            sessionTimeout="-1"
            maxSessionAge="500"
            sessionCleanup="90"
            sendCookies="true"/>

<!-- Saml config -->
<saml       isSAML="true"
            maxArtifactAge="60"
            samlTimeout="480"/>
```

The above SAML interface configuration establishes a maximum validity of the security lifecycle of 480 minutes, which means that right after that time the appliance will invalidate the internal GSA session and will force the user to be authenticated again. This rule also sets a maximum artifact age of 60 seconds, which means that during the authentication process if the artifact takes more than that time to be consumed, the process cannot continue. An artifact is an ID to let both the appliance and the SAML identity provider to know the sequence of request associated to an authentication process.

Whenever SAML interface is in place, sessions have to be configured in the security framework the following way:

- Configure the session parameters with the values that best fit with your environment.
- “sendCookies” parameter can be either set to “true” or “false”. Use the first value if you want to send the authentication cookies back to the user’s browser and have a SSO experience for those repositories that recognize those authentication cookies.
- “maxSessionAge” parameter has to be set in accordance with the samlTimeout value. The maximum session age should not be lower than the SAML timeout as it could drive into problems.

- It's highly recommended not to use the "sessionTimeout" attribute when the backend content sources invalids access from time to time, and use the "samlTimeout" attribute instead to implement that in the security framework, setting a "-1" as a sessionTimeout.

In the installation and configuration guide you can have more information on which integration frontend can fit better in your environment.

As the configuration examples shown below (see the configuration for scenario 1.1 and 2.1) associated to the different available deployment scenarios are represented for Forms Based authentication, just substitute that configuration with the one above in order to set up the SAML interface properly.

Another important thing to consider is the way the content is going to be crawled by the appliance. In the case you'd like to use the SAML interface, take into account the crawling process is out of the security framework scope as URLs are not rewritten and GSA's standard crawling features can be used. That's why it's not mentioned in this guide how to crawl when using the SAML frontend. You have more information about crawling and SAML at the Valve's Installation guide. This document includes information though on how to crawl thru the security framework when Forms Based interface is being used.

Deployment Scenarios

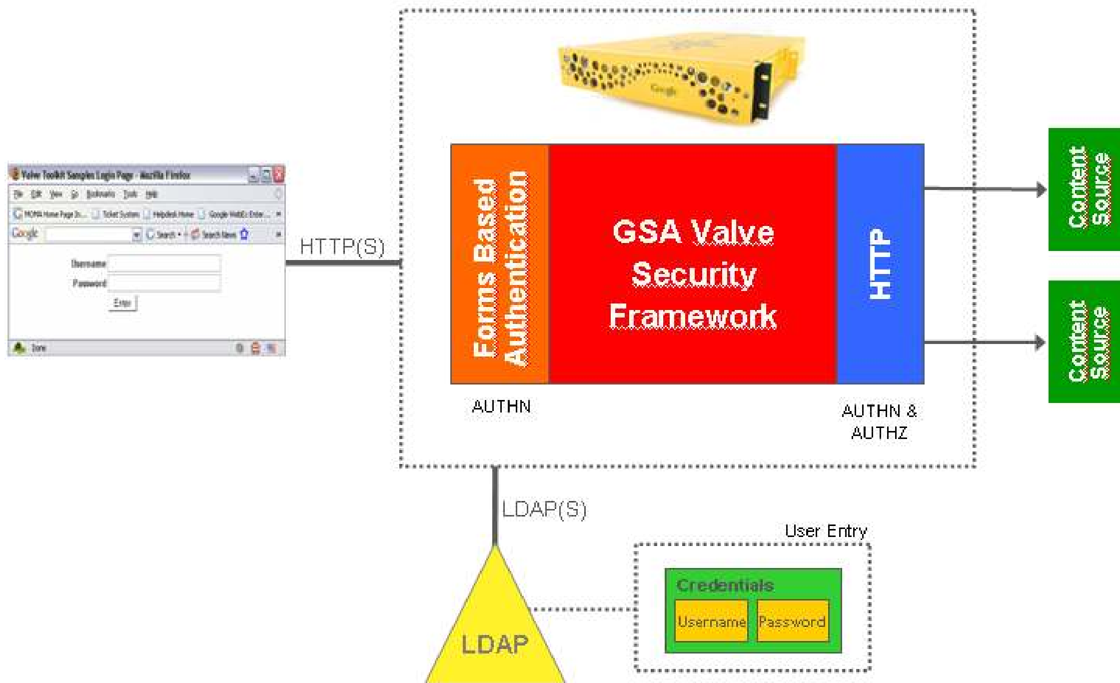
You can find below the different scenarios you can implement using the GSA Valve Security Framework, starting from the simple login form based authentication to more complex ones.

Scenario 1: Login Form Authentication

As it was previously mentioned, this framework can act as a standard cookie-based Single Sign-On (SSO) system. This kind of solution can be easily integrated with the Google appliance using its Forms Based Authentication feature that is able to be integrated with external authentication servers. The other option here is to use Access Control policies using the SAML frontend provided by this framework. The appliance offers an integration interface as well for this security standard.

The GSA Valve Security Framework can be configured to authenticate users through a login form that prompts for user credentials, offering a tight integration with the GSA. These credentials can be reused to authenticate users to whatever content repository is defined, so that the framework is more a kind of a secure proxy solution as it always takes authN/authZ decisions based on the source. The scenarios shown in this chapter are those that are driven by a username/password authentication thru a login form.

The security framework, as it's very flexible solution, offers multiple ways to treat those username/password credentials collected from the login page. The most common one is checking them against a standard-based identity repository like an LDAP, so that we are sure the credentials are valid and the corporate security policies are respected. This step can be fully customized and implemented in the security framework. The following picture represents a usual scenario integrated with a central LDAP directory:



This integration can be done with a Valve's authentication module that is completely based on LDAP standards and does not require a special configuration, apart from if you want to use secure LDAP protocol (LDAPS) that has to be configured accordingly. The standard Valve's LDAP module that offers such functionality is LDAPUniqueCreds that works with any LDAPv3 compliant directory like Active Directory or Oracle Internet Directory (OID) as it is implemented using Java standard JNDI libraries. Following the framework philosophy behind it, you can also build your own integration module for example to extend the LDAPUniqueCreds functionality or create a new one that interacts with a database.

This module is defined in the framework's main configuration file in a declarative way. This guide is intended to be a reference when configuring your environment, so you'll find here lots of examples on how you can set different deployment scenarios up, just fitting the configuration parameters to your own environment. In the first example you will see the entire configuration file but in the next samples, as most of the parameters are common, you'll just see those parameters strictly related to the concrete configuration.

Scenario 1.1: Active Directory Configuration

You can find below a pretty common Security Framework configuration when checking main user credentials against Active Directory (look further for other directories). As you can see in the next example, the LDAP integration has been implemented as any other standard repository that just manages the authentication. In this example we have defined as a sample corporate domain "google.com" and the URLs shown here do not correspond to any Google online service but they are sample URLs that point to a local Search infrastructure, so just substitute this configuration with the appropriate information from your own local infrastructure.

```
<?xml version="1.0"?>
<GSAValveConfiguration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="file:///c:/ValveFiles/gsaValveConfiguration.xsd">

<!--  URl to the login page that is part of the valve -->
<loginUrl>http://securityframework.google.com:8080/valve/login.jsp</loginUrl>
<authCookieDomain>.google.com</authCookieDomain>
<authenticationProcessImpl>com.google.gsa.valve.rootAuth.RootAuthenticationProcess</authenticationProcessImpl>
<authenticateServletPath>/Authenticate</authenticateServletPath>
<authorizationProcessImpl>com.google.gsa.valve.rootAuth.RootAuthorizationProcess</authorizationProcessImpl>
<authCookiePath>/</authCookiePath>
<authMaxAge>-1</authMaxAge>
<authCookieName>gsaSSOCookie</authCookieName>
<refererCookieName>gsaRefererCookie</refererCookieName>
<searchHost>http://gsainstance.google.com</searchHost>
<maxConnectionsPerHost>80</maxConnectionsPerHost>
<maxTotalConnections>1000</maxTotalConnections>
<testFormsCrawlUrl>http://securityframework.google.com:8080/valve/test.html</testFormsCrawlUrl>
<errorLocation>C:\\Tomcat\\webapps\\valve\\WEB-INF\\error</errorLocation>

<kerberos isKerberos="false"
        isNegotiate="false"/>

<sessions isSessionEnabled="false"/>

<saml       isSAML="false"/>

<!-- LDAP Unique creds with Active Directory →
<repository id="root" pattern="securityframework.google.com"
    authN="com.google.gsa.valve.modules.ldap.LDAPUniqueCreds" authZ="" failureAllow="true" checkAuthN="true">

            <P N="ldapBaseuser" V="dc=enterprise,dc=google,dc=com"/>
            <P N="ldapHost" V="ldap://activedirectory.google.com:389"/>
            <P N="ldapDomain" V="@enterprise.google.com"/>
            <P N="rdnAttr" V="cn"/>
</repository>

<!-- Set up here additional repositories that will be proxied -->

</GSAValveConfiguration>
```

As it was mentioned, change the saml and sessions parameters if you want to use the SAML interface instead of the Forms based one. The default login form page when using SAML is *loginSAML.jsp* that has to set in *loginUrl* parameter.

The **LDAP parameters** that have to be configured are the following:

- ldapBaseuser: it has to point to the LDAP base entry where the users are located behind. If you have different user branches configure such location that can access all of them. It's recommendable to point to the deeper entry that addresses all the users in order to avoid performance issues.
- ldapHost: this is the Active Directory (AD) hostname using the ldap (or ldaps) format.
- ldapDomain: it's the Windows corporate domain name (don't forget to include the "@").
- rdnAttr: this is the LDAP attribute used as the one to uniquely identify users in the LDAP and it's part of their Distinguished Name (DN). In AD this should be "cn".

Scenario 1.2: Other Standard LDAP Configuration

In the case you are using any other directory from other software vendors like Oracle, Sun or IBM, the configuration is slightly different and more standard. If this is the case, the Valve configuration file looks like (in this sample you can see just the LDAP repository configuration):

```
<repository id="root" pattern="securityframework.google.com"
      authN="com.google.gsa.valve.modules.ldap.LDAPUniqueCreds" authZ="" failureAllow="true" " checkAuthN="true">
            <P N="ldapBaseuser" V="cn=users,dc=google,dc=com"/>
            <P N="ldapHost" V="ldap://ldapserver.google.com:389"/>
            <P N="rdnAttr" V="cn"/>
</repository>
```

As you can see the parameters are roughly the same excepting the domain is not necessary.

If you don't want to use such AD or non-AD LDAP integration (or any other central authentication repository) it means credentials are not going to be checked against a central credential container but will be checked anyway against the additional content repositories. As a reminder, if one of your repository identifier (id) is "root" it means this is the central authentication repository so that this drives the whole authentication process. If this "root" authentication repository fails then the user will be automatically unauthorized to access to any content and the authentication process stops here.

Scenario 1.3: Multiple Credentials

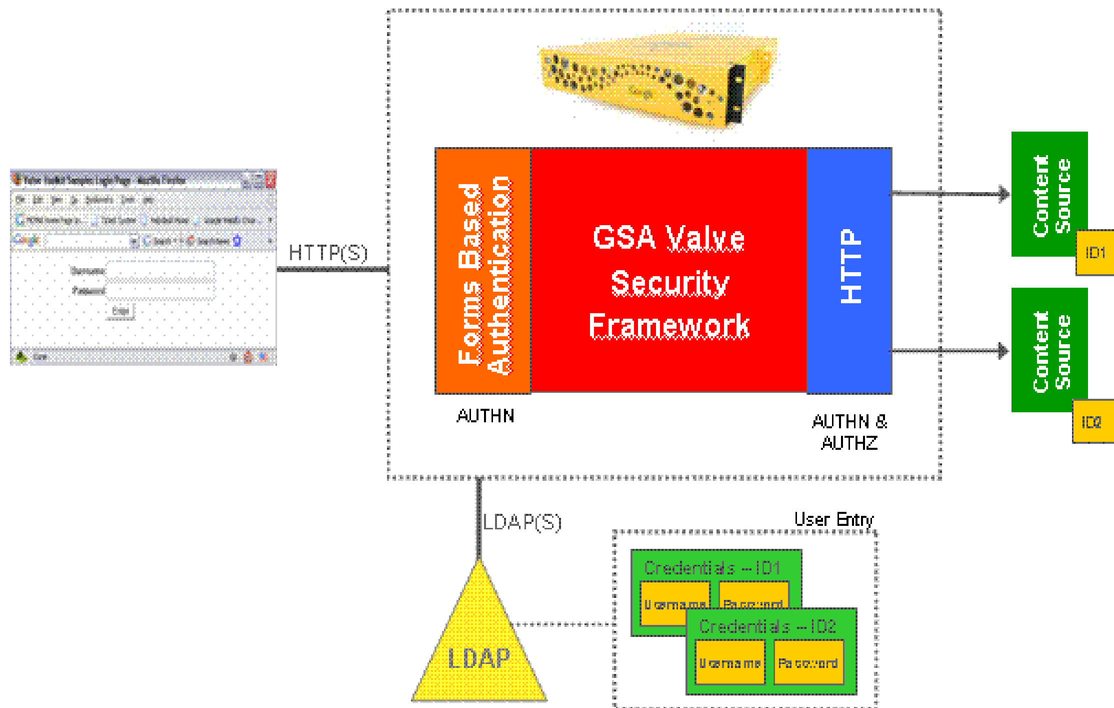
Until now we have seen how we can integrate the GSA Valve Security Framework with a central LDAP directory so that we can authenticate the user based on his/her main credentials and it's possible to reuse those credentials to authenticate users and create the proper sessions in the content repositories. In this section we'll see how we can manage a scenario where some repositories are using different credentials per user. This is a very common situation as usually those credentials, either the username or the password or even both, are not synchronized in all the backend applications. It clearly affects usability as users would be prompted several times with different credentials if this situation is not solved.

The Security Framework is able to manage more than one credential per user. One possible way to address it is getting all the different user credentials straight from the user in just one login form, i.e. in the same login page we can get those credentials for the different application, but it affects search experience as well specially if the number of credential pair combinations is huge. So, as we don't want the user to be affected by this issue, authentication process has to be fed somehow with multiple credentials.

The way this has been addressed in some projects is having a main credential per user that is kept in a central identity repository, for example in a LDAP directory server, and linking there the multiple credentials associated to that main user identity. Therefore we only physically authenticate the user once but we collect all the authentication credentials for all the secure backend applications at the same time so that the user experiences a Single Sign-On environment. This implementation permits one identity to be mapped with multiple ones in each way.

There is an already implemented Security Framework module that manages multiple credentials in a LDAP server. This has been implemented using some attributes extended in the LDAP that hold the credential information for accessing multiple sources. It works very similar to the previous explained module, as it also checks main authentication credentials provided by the user against the LDAP, but once the user is

authenticated it populates the multiple credentials to be available during the whole security process. In the next picture represents the architecture of this scenario:



This module is named *LDAPSSO* and it's available at the same location as the previous one. You can configure the LDAP attributes that are taken from the directory in a declarative way in the main Security Framework's configuration file. You can find below just an example on how this module is set up:

```
<repository id="root" pattern="securityframework.google.com"
   authN="com.google.gsa.valve.modules.ldap.LDAPSSO" authZ="" failureAllow="true" " checkAuthN="true">

            <P N="ldapBaseuser" V="dc=enterprise,dcdc=google,dc=com"/>
            <P N="ldapHost" V="ldap://activedirectory.google.com:389"/>
            <P N="ldapDomain" V="@enterprise.google.com"/>
            <P N="rdnAttr" V="cn"/>


            <P N="id1" V="documentum"/>
            <P N="username1" V="usernameDCTM"/>
            <P N="password1" V="passwordDCTM"/>
            <P N="id2" V="livelink"/>
            <P N="username2" V="usernameLL"/>
            <P N="password2" V="passwordLL"/>

</repository>
```

As you can see the LDAP connection configuration is common to the module previously seen that just checks unique user credentials against a directory server. In the above example we are accessing an Active Directory instance and collect up there a couple of additional credentials in this case implemented to be reused by a couple of repositories: Documentum and LiveLink. Here you are the main points to consider when configuring such module:

- This is a root module (id="root"), so it's the main authentication module.
- It should be the first one, so that the rest of the repositories can access to the populated credentials.
- The first configuration parameters are for the LDAP connection. You should take into consideration exactly the same points seen for the unique credential LDAP solution. Bear in mind this configuration is different depending on whether

Active Directory is in place or not.

- You can declare as many LDAP authentication attributes as you want simply defining them in the configuration file. Follow the following rules:
    - Each credential is built using three parameters:
        - *id*: this is the credential identification defined in its value attribute (“V”). This is important as well to then reuse it.
        - *username*: the value of this parameter is the LDAP attribute that holds the username for this credential.
        - *password*: the same as the above parameter for the password.
    - You have to sequentially identify the credentials starting by 1. So, in the case you want to have only one additional credential, then its id, username and password parameters have to be named with a 1 as a suffix of each parameter name (i.e: id1, username1 and password1). If you want to have a second credential its parameters should be declare as id2, username2 and password2. And so on and so forth.
- You can associate each new credential pair taken during this process to a specific repository, so that users can be authenticated against such repository using his/her proper credentials, assigning it the same “id” as the credential is named. In the case of the above example we can reuse those user credentials (“documentum” and “livelink”) for a couple of repositories for example the following way:

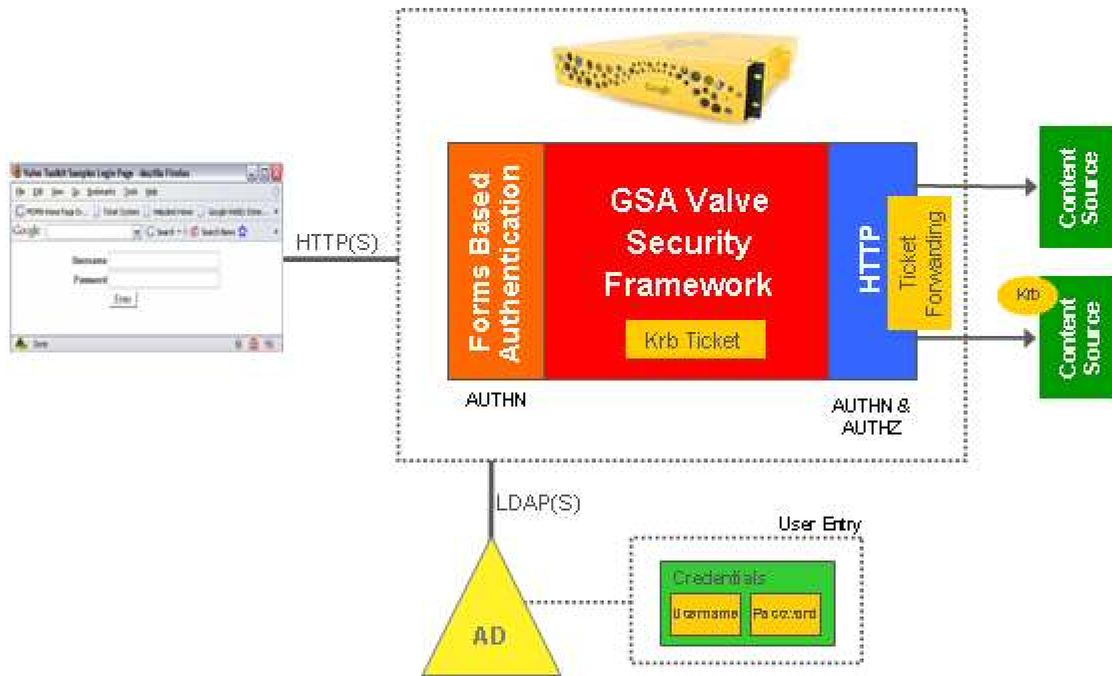
```
<!--right after the root LDAPSSO definition -->

<repository id="documentum" pattern="dctm.google.com"
   authN="com.google.gsa.valve.yourmodule.YourDCTMAuthN"
   authZ=" com.google.gsa.valve.yourmodule.YourDCTMAuthZ" failureAllow="true" " checkAuthN=”true”>
</repository>

<repository id="livelink" pattern="livelink.google.com"
   authN="com.google.gsa.valve.yourmodule.YourLiveLinkAuthN"
   authZ=" com.google.gsa.valve.yourmodule.YourLiveLinkAuthZ" failureAllow="true" " checkAuthN=”true”>
</repository>
```

This is just an example using fictitious AuthN/AuthZ classes that do not actually exist in the framework. As you can see each repository “id” matches with an “id” defined in the LDAPSSO repository.

Scenario 1.4: Forms Authentication with Kerberos (non-silently)

Apart from whether you want to use any of the previous scenarios, you could have some backend repositories protected by Kerberos like for example some Microsoft IIS sites. This scenario that is going to be explained here permits to create the Kerberos ticket for the users during the login form based authentication and create the ticket during the authentication request. We’ll create it using the username and password passed by the user so that the user accounts should be defined in the Kerberos repository (in Windows environment it is the Active Directory). Don’t get confused with the other Kerberos scenarios where we silently authenticate users and further explained in this document as this one does not establish any negotiation and the ticket is just created from the prompted login form.

The below picture represents this scenario:



This scenario is useful to be implemented when you have several content repositories some of them protected by Kerberos and others just by username and password. The main requirement here is the credentials provided by the user through the login form have to be the ones store in the Kerberos database so that the Kerberos ticket can be created on the fly using them. As we mentioned before, in the case you have a Windows domain, this database is Active Directory, so if you collect the Active Directory user credentials during the authentication process then you can create the user Kerberos ticket as well that would be valid to authenticate and impersonate the user against Kerberized content sources.

You should have to configure appropriately the Security Framework to use Kerberos tickets. Some specific configuration tasks have to be done in the environment as it is described in the Kerberos guide.

In the scenarios we'll see later on you will find how they get the Kerberos tickets silently based on a negotiation between the browser and the server in a completely transparent way to the user, whereas in this one we are creating such a ticket using the username and password provided by the user.

You can configure this scenario in the config file using the following parameters (pay attention as only those that really change from previous configuration are shown):

```
<?xml version="1.0"?>
<GSAValveConfiguration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="file:///c:/ValveFiles/gsaValveConfiguration.xsd">

<!--Use the same global configuration parameters as before, changing the following ones -->

<kerberos isKerberos="true"
            isNegotiate="false"
            krbini="c:\\config\\krb5.ini"
            krbconfig="c:\\config\\bcsLogin.conf"
            krbAdditionalAuthN="false"
            krbLoginUrl="http://securityframework.google.com:8080/valve/loginkrb.jsp"
            krbUsrPwdCrawler="true"
            krbUsrPwdCrawlerUrl="http://securityframework.google.com:8080/valve/logincrawlerkrb.jsp"/>

<!-- Adjust the session values to your environment -->
<sessions isSessionEnabled="true"
            sessionTimeout="-1"
            maxSessionAge="300"
            sessionCleanup="30"
            sendCookies="true"/>

<!--Set you own Kerberized repositories in the Repository section. For example -->

<repository id="krbsamplerepository" pattern="krbrepository.google.com"
            authN="com.google.gsa.valve.modules.krb.KerberosAuthenticationProcess"
            authZ=" com.google.gsa.valve.rmodules.krb.KerberosAuthorizationProcess" failureAllow="true" "
            checkAuthN="true">
</repository>
</GSAValveConfiguration>
```

The Kerberos credentials are created as a consequence of executing a Kerberized repository like the one used above. This process will create the user's Kerberos ticket that can be reused as many times as needed by the other modules. It means that at least, the first Kerberized authentication module has to create the Kerberos ticket like the KerberosAuthenticationProcess does, and that's why the checkAuthN parameter has to be set to "true" but the next ones.

If the SAML interface is used here, then the krbUsrPawdCrawler and krbUsrPwdCrawlerUrl attributes are not needed as the crawling process resides out of the security frameworks. The default additional Kerberos login URL when using SAML is *loginkrbSAML.jsp* that has to be set in *krbLoginUrl*.

GSA Serving Configuration (just for scenarios 1.x)

Here is the serving configuration in the appliance when using either SAML or Forms Based interface:

Serving Configuration with SAML Interface

The serving configuration when SAML interface is in place is done in the GSA console at Serving → Access Control. Set there the following parameters adapting them to your environment:

- User Login URL: *http://<security_framework_host>:<port>/valve/loginSAML.jsp*
- Artifact Service URL: h*ttp://<security_framework_host>:<port>/valve/samlauthnresolve*
- Authorization SPI: *http://<security_framework_host>:<port>/valve/samlauthz*
- Check "Disable prompt for Basic authentication or NTLM authentication" if you have some content sources that were crawled using these authentication methods.

Just change these values to the one that fits with your environment just in case you were using HTTPS, the name of the deployed application were different to “valve” or you changed in the framework the name of the above JSP or servlets.

Serving Configuration with Forms Based Interface

You have to define the serving configuration in the appliance. The next configuration step applies to all the above Forms Based Authentication related scenarios. A new Forms Based Authentication serving rule has to be defined in the appliance’s admin console at Serving → Forms Authentication with the following parameters:

- Choose “Always redirect to external login server”
- URL: *http://<security_framework_host>:<port>/valve/login.jsp*
- Cookie name: the same name as the one defined in the *authCookieName* configuration tag (by default: *gsaSSOCookie*)

GSA Crawling Configuration (just for scenarios 1.x) – Only Forms Based

If you want to crawl using the Forms Authentication crawling approach you just have to define such a rule in the GSA admin console exactly the same way it’s explained in the Valve Security Framework configuration guide. You should create a Crawling Forms Authentication rule with the following values:

- **Sample Forms Authentication protected URL**: an internal Valve Security Framework URL has to be configured for example:

  ```
  http:// <valve-loc>/valve/test.html
  ```

  Where <valve-loc> is your own GSA Valve Security Framework URL, for example: securityframework.google.com:8080.

- **URL pattern for this URL**: just point to the Security Framework URL pattern the following way:

  ```
  http:// <valve-loc>/valve/
  ```

In order to crawl your own content documents you can configure them in the admin console at Crawl and Index → Crawl URLs. If you want the Search Appliance to automatically crawl those documents served by the Security Framework you have to set up them including the proxy URL for example:

```
http://<security_framework_host>:<port>/valve/login.jsp?returnPath=http://server/doc
```

Scenario 2: Silent Kerberos Authentication

Another completely different feasible deployment scenario is authenticating the users using their Kerberos tickets in a completely transparent way to them, having a very good user experience as they are not bother with a login form. The Security Framework is able to manage those tickets and reuse them to impersonate users to the remote content repositories as it’s explained as well in both the Configuration and Kerberos guides.

We can support as well different scenarios when Kerberos authentication is in place. Apart from the scenario seen before identified as 1.4, where Kerberos tickets are just

created in the Security Framework on the fly using corporate username and password, we can support also a couple of additional scenarios that are using Kerberos authentication too when tickets are obtained based on a negotiation between the client (browser) and the server (Security Framework) in a transparent way to the users.

Scenario 2.1: Silent Authentication using Kerberos tickets

This scenario implements a fully silent authentication process through Kerberos tickets that means users are never prompted for credentials and the whole process is transparent for them. This implements an extremely productive environment as users are never bothered for authentication. It has some important requirements though:

- Any user has to be authenticated through Kerberos that have to be set up in the environment.
- Both the backend repositories and the Valve Security Framework AuthN/AuthZ modules used to access to the associated content sources have to be Kerberized.

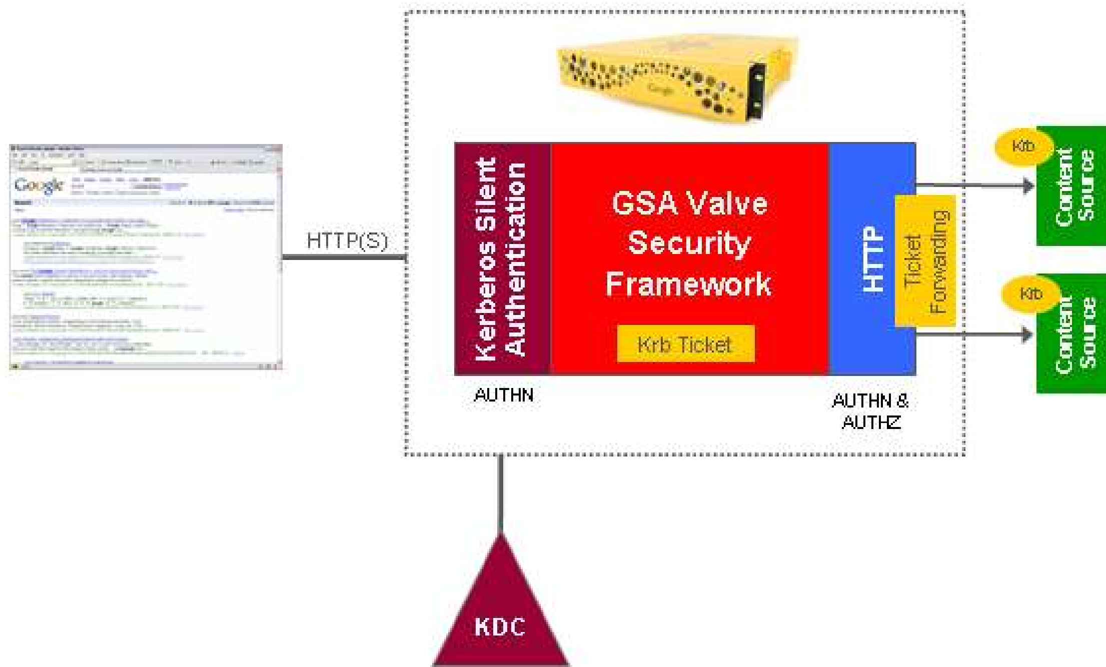
In the GSA Valve Security Framework you can find some Authentication and Authorization modules that are already Kerberized as for example:

- KerberosAuthenticationProcess: this is the main component that drives the Kerberos authentication and gets the user tickets. It's used by default by this scenario to collect up silently those tickets.
- KerberosAuthorizationProcess: it's the HTTP authorization process that authorizes users using Kerberos credentials against purely standard HTTP servers that secure the access using this technology. Examples of these web servers are Microsoft IIS or Apache when using Kerberos modules.
- Other modules: some other AuthN/AuthZ classes are Kerberized as for example Documentum or LiveLink.

In the case you want to implement your own Kerberized module you should consider the following important points:

- The Kerberos tickets are obtained during authentication natively by the Security Framework. That's why module's authentication processes are never processed by the Security Framework in this scenario. Kerberos tickets will be passed through the credential named as "krb5" to authorization modules. You can create your authorization module in such way that reads that credential and does whatever processes you'd need in order to authenticate users in your content repository. You can read the Kerberos user ID from such ticket.
- If your repository you want to create your module for is HTTP-enabled and is able to manage Kerberos authentication exactly the same way as a standard web server does, then you can just use the KerberosAuthorizationProcess. If not you should have to create a new one or Kerberized an existing module.

In the following picture you can find an architectural representation of this silent Kerberized scenario:



As you can see in this illustration the backend repositories are able to both authenticate and authorize users using Kerberos credentials that the Security Framework can forward to. This impersonation process is represented here in yellow. The KDC is the Kerberos Key Distribution Center that in Windows environment is actually the Domain Controller.

You should follow the steps defined in the Kerberos guide to let the Security Framework work with this authentication technology in your environment. The Security Framework configuration file should have the following important changes compared to the previous scenarios:

- The loginURL is different as it has to be as follows:

```
<loginUrl>http://<gsa_security_framework_host>:<port>/valve/kerberos</loginUrl>
```

- Kerberos has to be enabled and its “isNegotiate” parameter should be set to “true”.

Regarding session configuration you should follow exactly the same rules explained in the Kerberos guide.

The Security Framework should look like as the following example. As mentioned before this is just an example configured for the Google corporate domain “google.com” that represents a sample enterprise domain and then the following URLs do not correspond to any Google online service, so just substitute the Google corporate domain with your own enterprise domain and set up URLs properly:

```
<?xml version="1.0"?>
<GSAValveConfiguration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="file:///c:/ValveFiles/gsaValveConfiguration.xsd">

<!-- URl to the login page that is part of the valve -->
<loginUrl>http://securityframework.google.com:8080/valve/kerberos</loginUrl>
<authCookieDomain>.google.com</authCookieDomain>
<authenticationProcessImpl>com.google.gsa.valve.rootAuth.RootAuthenticationProcess</authenticationProcessImpl>
<authenticateServletPath>/Authenticate</authenticateServletPath>
<authorizationProcessImpl>com.google.gsa.valve.rootAuth.RootAuthorizationProcess</authorizationProcessImpl>
<authCookiePath>/</authCookiePath>
<authMaxAge>-1</authMaxAge>
<authCookieName>gsaSSOCookie</authCookieName>
<refererCookieName>gsaRefererCookie</refererCookieName>
<searchHost>http://gsainstance.google.com</searchHost>
<maxConnectionsPerHost>80</maxConnectionsPerHost>
<maxTotalConnections>1000</maxTotalConnections>
<testFormsCrawlUrl>http://securityframework.google.com:8080/valve/test.html</testFormsCrawlUrl>
<errorLocation>C:\\Tomcat\\webapps\\valve\\WEB-INF\\error</errorLocation>

<kerberos isKerberos="true"
            isNegotiate="true"
            krbini="c:\\config\\krb5.ini"
            krbconfig="c:\\config\\bcsLogin.conf"
            krbAdditionalAuthN="false"
            krbLoginUrl="http://securityframework.google.com:8080/valve/loginkrb.jsp"
            krbUsrPwdCrawler="true"
            krbUsrPwdCrawlerUrl="http://securityframework.google.com:8080/valve/logincrawlerkrb.jsp"/>

<!-- Adjust the session values to your environment -->
<sessions isSessionEnabled="true"
            sessionTimeout="-1"
            maxSessionAge="300"
            sessionCleanup="30"
            sendCookies="true"/>

<saml      isSAML="false"/>

<!-- Set up here Kerberzed repositories that will be proxied. For example: -->
<repository id="KrbServer1" pattern="http://krbserver1.google.com:83"
           authN=""
           authZ="com.google.gsa.valve.modules.krb.KerberosAuthorizationProcess" failureAllow="true" "
           checkAuthN="false">
</repository>

</GSAValveConfiguration>
```

In this case, the Kerberos ticket was obtained during the negotiation process and that's why in the case of using the Kerberos modules shipped with the security framework, it's not needed to process the Kerberos authentication again as it happened when the Kerberos ticket was created by using the username and password (scenario number 1.4). That's why in this example checkAuthN attribute for that repository was set to "false".

As it was already mentioned, the above configuration example is meant just for Forms Based authentication frontend. If you want to use the SAML one instead, follow the rules explained in this document on how to configure it setting the "saml" tag. The krbUsrPwdCrawler has to be set to "false" as well. The default additional Kerberos login URL when using SAML is *loginkrbSAML.jsp* that has to be set in *krbLoginUrl*.

Regarding the integration with Google Search Appliance you can find later on the instructions on how to configure both the crawling and the serving.

Scenario 2.2: Kerberos Silent and Login Form Authentication

This is a special scenario where we can have both Kerberos silent and Forms Based authentication at the same time as a unique authentication process would be happening. This scenario has been implemented for those situations where we have

two separated environments, one where Kerberos authentication is in place and another separated one securely driven by username and password. This is a further description of these two environments:

- **Kerberized environment**: in this environment the applications are protected by Kerberos and the KDC stores these users. This is usually a Windows environment where the users are part of the Domain Controller's Active Directory.
- **Username/Password driven environment**: some other applications in your environment are secured by username and password credentials that are stored in a completely different place than the Kerberos KDC database. This is usually represented by a corporate LDAP that holds the user credentials for the applications.

This scenario is common in some corporations where they have separated the Operating System network from the application infrastructure. This scenario only makes sense if the Kerberos user database is completely different than the other corporate identity repository. If they are the same user database/LDAP (for example the same Active Directory instance) we'll be in the scenario 1.4, where Kerberos tickets are generated on the fly from username and password credentials.

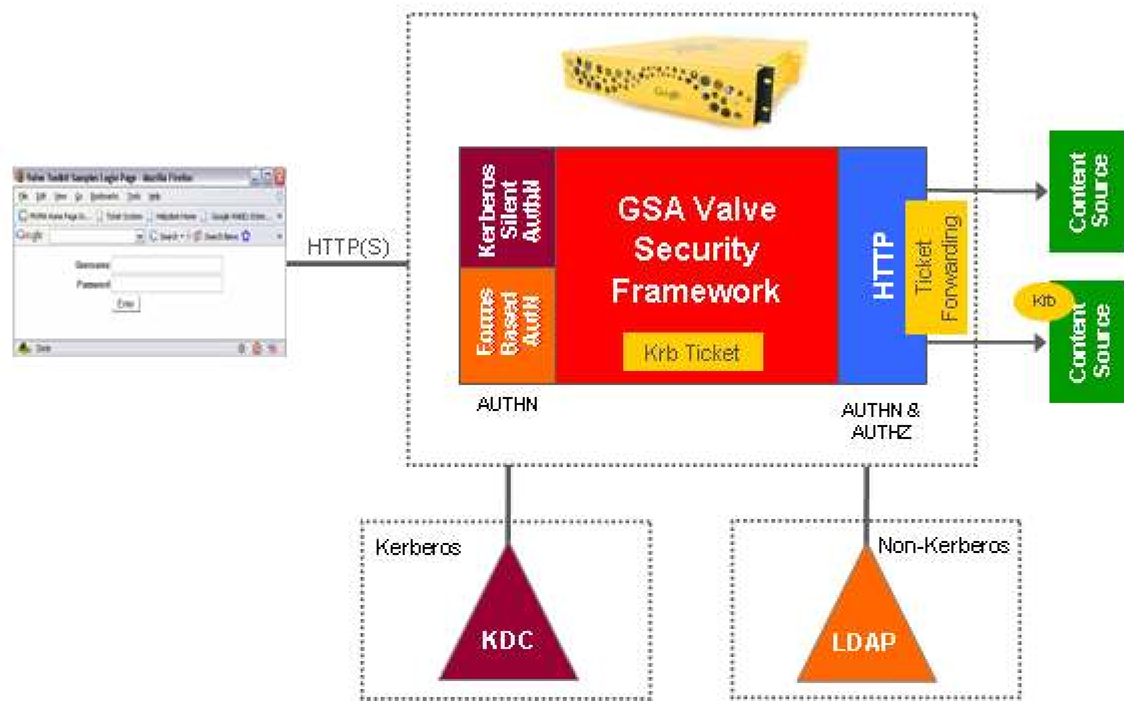
Therefore we'll have two completely separated authentication mechanisms that are processed the following way:

- The user is initially silently authenticated using Kerberos credentials. This is the principal authentication approach as if this step fails the whole user authentication process is not valid and the next process is not processed.
- Once the user is successfully authenticated thru Kerberos, then the user is challenged with a forms based authentication. This mechanism works exactly the same way as any other Forms based scenario described in the first part of this document, so that we can check the credentials against a central LDAP.

This scenario implements the strength of having two different authentication processes in parallel but the user is only prompted once, so the user experience is not affected at all. During authentication we'll get both Kerberos ticket and username/password credentials for each user that can be used to authenticate him/her in each content repository. We'll have then two different sorts of AuthN/AuthZ modules depending on whether they are using Kerberos credentials or username and password:

- **Kerberized modules**: these are working the same way as the Kerberized repositories explained in the previous scenario. An example of this implementation is KerberosAuthorizationProcess that natively uses the Kerberos credential (internally named as "krb5").
- **Non-Kerberized modules**: they are those that natively are using username and password to authenticate users in the content sources. As an example of this could be the HTTPBasicAuthentication class.

This scenario is represented in the following picture:



As you can see in the above illustration we have two separated identity LDAP/databases that stores the credentials needed to access to those content repositories either Kerberized or username/password secure. The access to the Kerberized ones is represented in yellow.

You have to configure Kerberos to let the GSA Valve Security Framework makes use of it exactly the same way as the previous scenarios where Kerberos was in place. Review the Kerberos configuration guide on how to do it.

Then the Valve configuration file has to be properly parameterized. This configuration is quite similar to the previous scenario as the general and session parameters have to be configured the same. The only aspects that change here are the following:

- In the Kerberos tab, the krbAdditionalAuthN parameter has to be set to “true”
- The repository definition area contains both Kerberized and non-Kerberized content sources:
  - You can define a “root” authentication (look for what it really means in the configuration guide) for the Forms Based main repository. You can define as well any kind of scenario described at the beginning of this document.
  - You would have some repositories that natively support Kerberos and others that don’t. These repositories should take those credentials that really need in order to authenticate users.

The main Security Framework configuration file (without the common part with the previous scenario) would look like the following example:

```
<?xml version="1.0"?>
<GSAValveConfiguration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="file:///c:/ValveFiles/gsaValveConfiguration.xsd">

<!--  Same global and session parameters as the previous scenario -->

<kerberos isKerberos="true"
              isNegotiate="true"
              krbini="c:\\config\\krb5.ini"
              krbconfig="c:\\config\\bcsLogin.conf"
              krbAdditionalAuthN="true"
              krbLoginUrl="http://securityframework.google.com:8080/valve/loginkrb.jsp"
              krbUsrPwdCrawler="true"
              krbUsrPwdCrawlerUrl="http://securityframework.google.com:8080/valve/logincrawlerkrb.jsp"/>

…
<!--You can have here a root repository for the Forms Based Authn. For example: -->
<repository id="root" pattern="securityframework.google.com"
    authN="com.google.gsa.valve.modules.ldap.LDAPUniqueCreds" authZ="" failureAllow="true">

             <P N="ldapBaseuser" V="dc=enterprise,dc=google,dc=com"/>
             <P N="ldapHost" V="ldap://activedirectory.google.com:389"/>
             <P N="ldapDomain" V="@enterprise.google.com"/>
             <P N="rdnAttr" V="cn"/>
</repository>

<--You would have some Kerberized repositories.For example the following one -->
<repository id="KrbServer1" pattern="http://krbserver1.google.com:83"
             authN=""
             authZ="com.google.gsa.valve.modules.krb.KerberosAuthorizationProcess" failureAllow="true"
             checkAuthN="false">
</repository>

<--You would have some NON Kerberized repositories. For example the following one -->
<repository id="HTTPBasicAuth" pattern="http://server1.google.com"
              authN="com.google.gsa.valve.modules.httpbasic.HTTPBasicAuthenticationProcess"
              authZ="com.google.gsa.valve.modules.httpbasic.HTTPBasicAuthorizationProcess" failureAllow="true"
              checkAuthN="true">
             <P N="HTTPAuthPage" V="http://server1.google.com/"/>
</repository>

</GSAValveConfiguration>
```

Adapt this configuration in case the SAML interface is going to be used as it's been previously mentioned setting the usrKrbPwdCrawler attribute to "false" and making use of the saml tag.

GSA Serving Configuration (just for scenarios 2.x)

Here is the serving configuration in the appliance when using either SAML or Forms Based interface with these 2.x scenarios:

Serving Configuration with SAML Interface

The serving configuration when SAML interface is in place is done in the GSA console at Serving → Access Control. Set there the following parameters, in a similar way as for the 1.x scenarios, adapting them to your environment:

- User Login URL: *http://<security_framework_host>:<port>/valve/kerberos*
- Artifact Service URL: h*ttp://<security_framework_host>:<port>/valve/samlauthnresolve*
- Authorization SPI: *http://<security_framework_host>:<port>/valve/samlauthz*
- Check "Disable prompt for Basic authentication or NTLM authentication" if you have some content sources that were crawled using these authentication methods.

Just change these values to the one that fits with your environment just in case you

were using HTTPS, the name of the deployed application were different to “valve” or you changed in the framework the name of the above JSP or servlets.

Serving Configuration with Forms Based Interface

You have to define the serving configuration in the appliance that is very similar to the configuration in the Forms Based Authentication scenarios (1.x) as just the external login server URL slightly changes. We have to still use a GSA’s Forms Based Authentication serving rule as we have to redirect the user to the Security Framework as this is the way to interact between the user (browser) and the search infrastructure. This rule has to be defined in the admin console at Serving → Forms Authentication with the following parameters:

- Choose “Always redirect to external login server”
- URL: *http://<security_framework_host>:<port>/valve/kerberos*
- Cookie name: the same name as the one defined in the *authCookieName* configuration tag (by default: *gsaSSOCookie*)

GSA Crawling Configuration (just for scenarios 2.x) – Only Forms Based

If you want to crawl your content using the Forms Authentication crawling approach as it’s described in the Crawling section at the beginning of this document, you can configured it following up the instructions that are explained in the GSA Valve Security Framework configuration guide, taking into account that just for these two last Kerberos scenarios, the GSA’s Crawling Forms Authentication rules have to be as follows:

- **Sample Forms Authentication protected URL**: an internal Valve Security Framework redirected URL has to be configured for example:

  http://<valve-loc>/valve/kerberos?returnPath=http://<valve-loc>:<port>/valve/test.html

  Where <valve-loc> is your own GSA Valve Security Framework URL, for example: securityframework.google.com:8080. The content of the returnPath parameter should be a local URL to the application server the Security Framework is running on.

- **URL pattern for this URL**: just point to the Security Framework URL pattern the following way:

  http://<valve-loc>/valve/

In order to crawl your own content documents you can configure them in the admin console at Crawl and Index → Crawl URLs. If you want the Search Appliance to automatically crawl those documents served by the Security Framework you have to set up them including the proxy URL for example:

http://<valve-loc>/valve/kerberos?returnPath=http://server/doc

The server here can be either Kerberized or not, so just use this format for any kind of content you’d like crawling using Forms Based approach and it’d be protected by the Security Framework.