JasperReports® Server

Security Guide

Version 9.0.0 | January 2024

Contents

Overview of JasperReports Server Security

JasperReports® Server ensures that people can access only the data they are allowed to see. The settings that define organizations, users, roles, and repository resources work together to provide complete access control that includes:

- Authentication – Restricts access to identified users and protects that access with passwords. Defines roles for grouping users and assigning permissions.
- Authorization – Controls access to repository objects, pages, and menus based on users and roles.
- Data level security (commercial version only) – Defines row and column level permissions to access your data. Row and column level permissions can be defined and enforced in Domains.

Administrators must keep security in mind at all times when managing organizations, users, roles, and resources, because the security settings behind each of these rely on the others.

Caution: The bundled installer is not meant for use in either production environments or security testing. It is only intended for evaluation purposes. The application server provided in that package has been configured with minimal security. We recommend that production environments use the WAR package deployed to an application server configured to your security standards.

Caution: This guide focuses on security concerns specific to JasperReports Server. However, you should consider other security precautions in your environment. For example, an end user can potentially exploit JasperReports Server's Test Connection option when scheduling reports to an FTP server. If this is a concern, you can secure the port (by default, port 21) at the operating system level.

Note: The chapter on data-level security for Domains has been moved from this guide to the new document *JasperReports Server Data Management Using Domains*. That guide covers all aspects of Domains, including creating the security file.

This chapter contains the following sections:

Authentication

The first part of security is to define user accounts and secure them with passwords to give each user an identity within the JasperReports® Server. The server stores user definitions, including encrypted passwords, in a private database. Administrators create, modify, and delete user accounts through the administrator pages, as described in the *JasperReports Server Administrator Guide*.

JasperReports® Server also implements roles for creating groups or classes of users with similar permissions. A user can belong to any number of roles and have the privileges of each. The server stores the role definition in its private database, and administrators create, modify, and delete roles through the administrator pages, as described in the *JasperReports Server Administrator Guide*.

JasperReports® Server relies on the open source Spring security framework. It has many configurable options for:

- External authentication services such as LDAP (used by Microsoft Active Directory and Novell eDirectory)
- Single Sign-on using JA-SIG's Central Authentication Service (CAS)
- Java Authentication and Authorization Service (JAAS)
- Container security (Tomcat, Jetty)
- SiteMinder
- Anonymous user access (disabled by default)

JasperReports® Server also supports these encryption and authentication standards:

- HTTPS, including requiring HTTPS

- HTTP Basic
- HTTP Digest
- X509

The Spring framework is readily extensible to integrate with custom and commercial authentication services and transports.

Authentication occurs by default through the web user interface, forcing login, and/or through HTTP Basic authentication for web services, such as Jaspersoft® Studio and for XML/A traffic. The server can automatically synchronize with an external authentication service. External users don't need to be created manually in the server first. Both users and roles are created automatically in the server from their definitions in an external authentication service. For an overview of the authentication system and details about external authentication, see the JasperReports® Server External Authentication Cookbook.

Authorization Overview

With a user's identity and roles established, JasperReports® Server controls the user's access in these ways:

Menu options and pages	The menus appear in the JasperReports® Server UI depending on the user's roles. For example, only users with the administrator role can see the Manage menu and access the administrator pages. By modifying the server's configuration, you can modify access to menus, menu items, and individual pages. Refer to the JasperReports® Server Source Build Guide and JasperReports Server Ultimate Guide for more information.
Organization scope	Users belong to organizations and are restricted to resources within their organizations. Organizations have their own administrators who each see only the users, roles, and resources of their own organization. When the JasperReports® Server is configured with multiple organizations, those organizations are effectively isolated from each other, although the system admin can share resources through the Public folder. For more information, see the *JasperReports Server Administrator Guide*.

Resource permissions	Administrators can define access permissions on every folder and resource in the repository. You can define permissions for every role and every user, or leave them undefined to be inherited from the parent folder. For example, a user may have read-write access to a folder where they create reports, but the administrator can also create shared reports in the same folder that are set to read-only. The possible permissions are: no access, execute only, read-only, read-delete, read-write-delete, and administer (see "Repository Administration" in the *JasperReports Server Administrator Guide*). Permissions are enforced when accessing any resource whether directly through the repository interface, indirectly when called from a report, or programmatically through the web services. A user's access to resources is limited by the permissions defined in the user's roles.
Administrator privileges	JasperReports® Server distinguishes between reading or writing a resource in the repository and viewing or editing the internal definition of a resource. For security purposes, granting a user read or write permission on a resource does not allow viewing or editing the resource definition. For example, users need execute or read permission on a data source to run the reports that use it, but they cannot view the data source’s definition, which includes a database password. Also, only administrators can interact with theme folders to upload, download, and activate CSS files that control the UI's appearance.
Data-level security	Data-level security determines the data that can be retrieved and viewed in a report, based on the username and roles of the user running the report. For example, a management report could allow any user to see the management hierarchy, managers would see the salary information for their direct employees, and only human resource managers would see all salary values. Data-level security in Domains is explained in the new *JasperReports Server Data Management Using Domains*. Data-level security through OLAP views is covered in the Jaspersoft OLAP User Guide. Note: This type of security is available only in the commercial edition of JasperReports® Server.

User attributes	User attributes are name-value pairs associated with a user, organization, or server. User attributes provide additional information about the user and can also be used to restrict a user's access to data through Domain security files and OLAP schemas. For information on defining user attributes, see "Editing User Attributes" in the *JasperReports Server Administrator Guide*. User, organization, and server attributes can be used to customize the definition of a data source or as parameters of a report. See "Attributes in Data Source Definitions" and "Attribute-Based Parameters for Queries and Reports" in the *JasperReports Server Administrator Guide*

Key and Keystore Management

JasperReports Server uses cryptographic keys internally to secure sensitive content such as database passwords in the configuration and user passwords in the database and export catalogs. The keys are used to encrypt information before storage and decrypt it on retrieval.

The keys themselves are sensitive security items that must be carefully stored and safeguarded. A keystore is a standard file that holds keys and protects them with passwords. The Java Cryptography Architecture (JCA) provides the ciphers and the protocols that protect the keys and the keystore. Administrators use the command-line keytool to manage keys in the keystore, and the server accesses keys as permitted through Java APIs.

As of JasperReports Server 7.5, key and keystore management has been updated to improve consistency and secure all sensitive server and user data inside and outside the server application. Administrators should become familiar with the new procedures and how to upgrade keys and the keystore from previous versions if necessary.

Because the keystore and keys are created during installation, the user account that performs the installation is the owner of the keystore file and holder of the keystore passwords. If either the keystore or its passwords are lost, the server can no longer function and the data it contains may become inaccessible, so be sure to keep backup copies.

This chapter contains the following sections:

Managing Keys During Installation

As of JasperReports Server 7.5, the use of keys in a single keystore has been standardized, and all the necessary files and configuration settings are created and initialized during the

installation.

The following files are created during installation, where $USER is the user who installed the server:

Filename	Default Location	Description
`.jrsks`	$USER/home	The encrypted keystore file containing the actual keys. Only the user who performs the installation can access and modify this file using the keytool utility.
`.jrsksp`	$USER/home	The keystore properties file that defines the keys in the keystore. This file is encoded so that it does not appear in plain text, and permissions are set so that only the user who performs the installation can modify it.
`keystore.init.properties`	buildomatic and WEB-INF/classes	Contains the path to the keystore files above, so that JasperReports® Server and its app server can use them. This file should always point to the same keystore that was created at installation. This file is copied in two locations so that when other system users (for example tomcatuser) run the buildomatic commands, they can detect the existing keystore and not create a new one. If this file is missing and the buildomatic scripts do not detect the keystore, they prompt the user to create a new one. If a new keystore is created twice for a server, the scripts may overwrite database passwords and the server will no longer be able to access its internal database. Be sure to never create more than one keystore for the server.

The server uses different cryptographic keys for the following tasks:

- Encrypting user passwords and secure files in the internal database.
- Encrypting and decrypting passwords in import and export catalogs. The server may also import keys to decrypt catalogs from other servers.

- Encrypting passwords and sensitive data that appear in configuration files.
- Encrypting log contents in log collector output and diagnostic data.
- Encrypting HTTP parameters with a static key (now deprecated)

Keys During Upgrade

Because key management was introduced recently in JasperReports Server, upgrade procedures must also deal with upgrading keys so they are unified in the keystore. For more details, see the JasperReports Server Upgrade Guide.

One important detail is that the keys and keystore are associated with the user that originally installed the server. Therefore, you must do the following for the upgrade to recover your encrypted repository contents:

- Back up your original keystore by copying the .jrsks and .jrsksp files to a safe location. Remember that these files contain sensitive keys for your data, so they must always be transmitted and stored securely.
- Run the upgrade script as the same user that was originally used to install the server. Then the keystore is available to the script in the user's home directory.
- Alternatively, copy the .jrsks and .jrsksp files to the home directory of the user that runs the upgrade script.

When the server's original keystore is available to the user running the upgrade script, the keys it contains are copied and preserved in the new keystore (.jrsks) with the aliases deprecatedPasswordEncSecret and deprecatedImportExportEncSecret.

When the upgrade script does not detect any keystore in the user's home directory, the script prompts you to create a new keystore. DO NOT create a new keystore if you wish to recover the contents of your old server through the upgrade process. If you create a new keystore during the upgrade procedure, you need to recover your server's repository data (all users and all reports) from a backup in a separate import. You will also need a backup of your old keystore to import the old keys. You need both backups to have been created and saved previously, they are not created by the upgrade process. In general, if the upgrade script prompts you to create a new keystore, it is recommended to quit the script and rerun it as a user with access to the original keystore, as describe above.

Making Backups

During installation, the keys in the keystore are used to protect sensitive data by encrypting configuration files and the server's internal repository database. Once the server is installed, the keys are used during normal operation to encrypt or decrypt information as needed. For example, when anyone logs into the server, their password is encrypted with the corresponding key and compared to the encrypted password stored in their user profile. Or when importing a report from an export catalog, the catalog must be decrypted to access the contents.

Without the keystore files, the specific files created with random keys during the installation, your instance of the server cannot function and all the information it contains becomes inaccessible. This is why having backups of the keystore files must be a part of your larger backup and recovery plans for your data. Businesses usually have IT policies for making backups, and the keystore files for your JasperReports® Server instance should be included in your policies and procedures.

Backups of the keystore files are digital copies of the files stored in a secure location, usually determined by your IT policies. Use the following guidelines when creating and implementing your keystore backup policies:

- Copy both the .jrsks and .jrsksp files together, keeping the .jrsksp file encoded as it is.
- The keystore files should be copied only by the system user who installed the server.
- Restrict access to the backup keystore files as you would the originals on production servers. This includes digital access security for online backups and physical security for offline backups. The files are literally the keys to the application and should be guarded as such.
- If you need to restore from the backups, the system user who installed the server should copy the files to their home directory ($USER/$HOME). This is the location where the server expects to find them at runtime.

Managing Keys for Import and Export

As of JasperReports Server 7.5, the management of the encryption keys used during import and export has been automated. These encryption keys are used by the server to encrypt user passwords so they are not revealed in export catalogs.

In previous versions of the server, the import-export key had to be configured manually outside the server. Beginning with version 7.5, the server creates and manages its import-export key internally, and also includes UI and REST options for specifying keys during import and export operations. However, there are still cases when you may need to manage keys on the server and specify keys during import-export operations.

The following table summarizes the use of keys when importing catalogs into the current version of the server:

Origin of Export Catalog	Guidelines for Importing into 9.0.0
Before version 7.5 With default key	The current version of the server handles previous export catalogs created with legacy keys, and even older catalogs that did not use encryption. When importing through the UI, specify the Legacy key, and when importing through the command line, specify the deprecatedImportExportEncSecret.
Before version 7.5 With custom import-export keys	The custom keys are not known to your server, so to import these catalogs, you will need to share the custom key with the importing server. There are two ways to specify custom keys: 1. For one-time or occasional imports, you can enter the key's hex value into the UI or the import command line, or store it in the repository for repeated use. 2. For continual use, you should import the key to server's keystore with the command-line import command. Then the key is available by it is alias either through the UI or the command line when importing a catalog.
Version 7.5 and later From the same server	When you import a catalog back into the same server, the server uses the same key for export, and import and thus can read its own export catalog. When importing through the UI, specify the Server key, and when importing through the command line, the key is detected automatically. This is also true after upgrading the server, as long as the catalog was exported from version 7.5 or later and the server's keystore was preserved during the upgrade. For more information about the keystore and catalogs during upgrade, see the JasperReports Server Upgrade Guide.

Origin of Export Catalog	Guidelines for Importing into 9.0.0
Version 7.5 and later From a different server	Because each server has unique randomly generate keys, the keys of the export server are not known to your server. However, unlike the custom key scenario above, you do not have direct access to the server's keys. Before you can move export catalogs between servers, such as test and production servers, you need to generate and share a key between the servers: 1. Generate a key during the export operation on the first server. 2. Import the new key into the second server's keystore with the command-line import command. 3. Import the catalog into the second server and specify the custom key either through the UI or the command line. If you also import the key back into the originating server, both servers share the same key. Then if you specify this custom key during import and export, catalogs can be exported from either server and imported into the other one.

This guide documents the following operations to set up and manipulate keys:

- Specifying custom keys during import and export operations.
- Importing keys used by other servers.
- Exporting keys for use in other servers.
- Sharing custom keys between multiple servers.

For the default import and export operations, including specifying import-export keys through the UI, see the JasperReports Server Administrator Guide:

- Importing catalogs from older servers with legacy or custom keys.
- Exporting catalogs with a specific key.

The following sections describe the three ways to import a catalog that is encrypted with a custom key:

- Use the import UI, and enter the key’s hexadecimal bytes in the Key Value field. This method is simple, but the key will not be stored in the server for multiple imports.

- Store the custom key as a secure file resource in the repository, so it can be reused. Then use the import UI with the Stored Key field.
- Use the import command line to import the key into the keystore so that it is available for any import operation in the future.

Entering a Key Value in the Import UI

The simplest way to import a catalog with a custom key is to use the Settings UI for import and enter the key value. The key value is its representation in hexadecimal, for example:

0x1c 0x40 0xb9 0xf6 0xe2 0xd3 0xf9 0xd0 0x5a 0xab 0x84 0xe6 0xd4 0xe8 0x5f 0xed

1. Log in as system administrator (superuser by default).
2. Select Manage > Server Settings, then click Import in the left-hand panel.
3. In the right-hand panel, browse the file system to enter the catalog file you want to import.

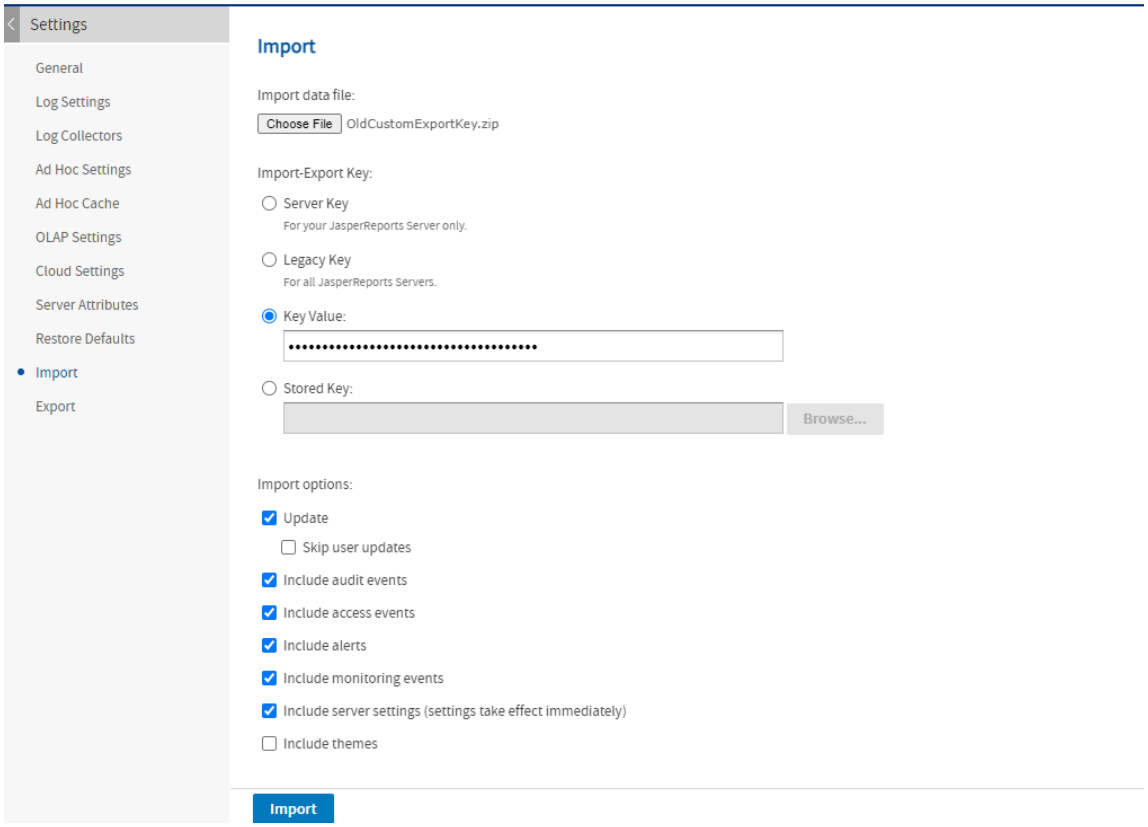

Figure 1: Import UI with Key Value

4. Choose the Key Value radio button and paste the entire key value in the designated field. The characters of the key value are hidden to keep them secret.
5. Select your import options and click Import.

   If the key does not decrypt the catalog file, you get an error message, otherwise the import proceeds.

Using a Stored Key in the Repository

If you have multiple files to import, you can store the custom key in a secure file resource in the repository. The contents of secure file resources are considered sensitive and protected internally in the same way as user passwords, that is, they are encrypted with a key in the internal database.

1. Start by saving your custom key value as a hexadecimal number in a plain text file, for example:

   0x1c 0x40 0xb9 0xf6 0xe2 0xd3 0xf9 0xd0 0x5a 0xab 0x84 0xe6 0xd4 0xe8 0x5f 0xed
2. Log in as system administrator (superuser by default).
3. Select View > Repository, then browse the repository tree to find an appropriate folder.
4. Right-click the folder and select Add Resource > File > Secure File.
5. In the Add File dialog, browse the file system to enter your text file with the key.

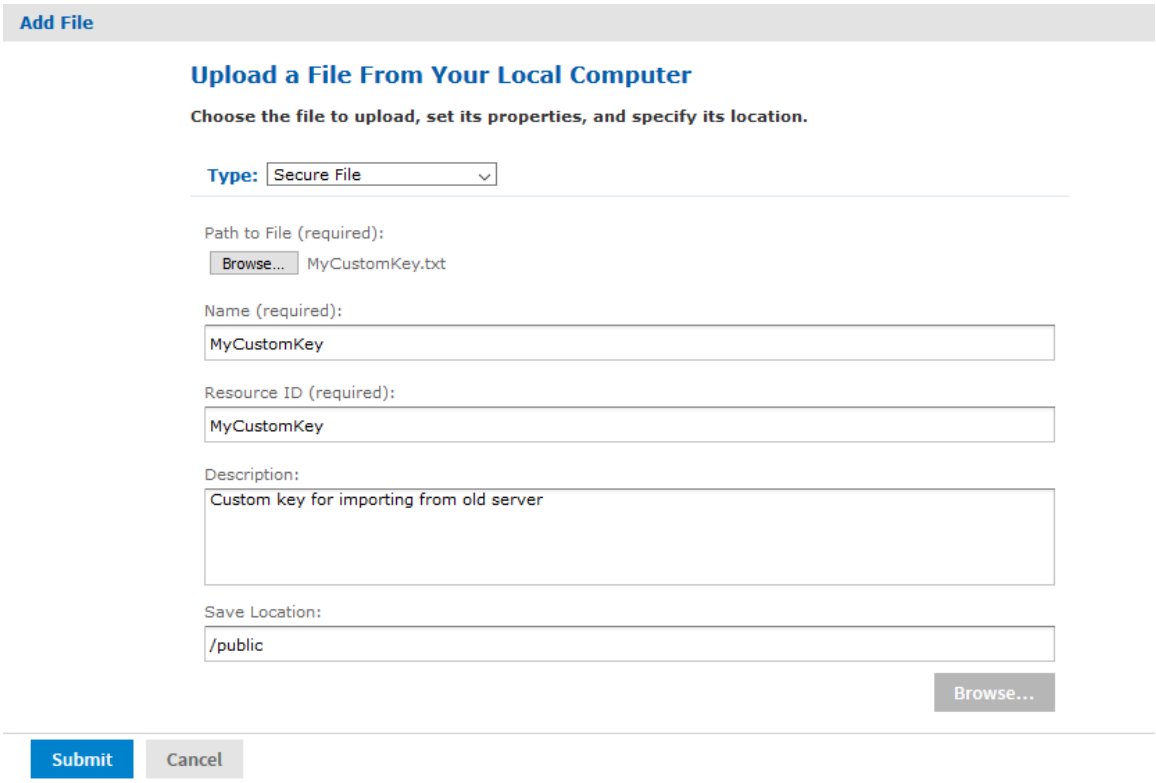

Figure 2: Add Secure File Dialog

6. Fill in the other fields and click Submit. The File appears in the repository.
7. Select Manage > Server Settings, then click Import in the left-hand panel.
8. In the right-hand panel, browse the file system to enter the catalog file you want to import.

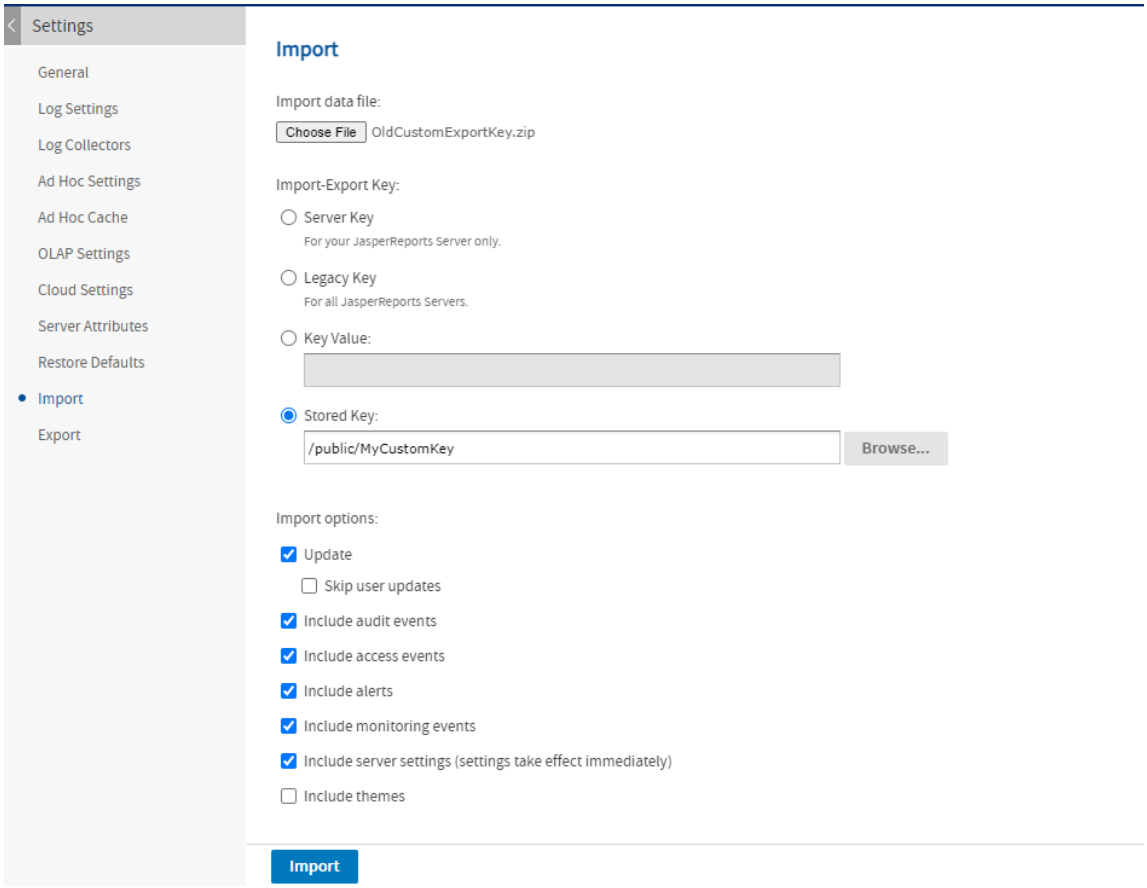

Figure 3: Import UI with Key File

9. Choose the Stored Key radio button and browse the repository to find your secure file.
10. Select your import options and click Import.

    If the key does not decrypt the catalog file, you will get an error message, otherwise the import proceeds.

Specifying an Import Key on the Command Line

Similar to the import UI, the js-import command-line utility has new options to specify the key needed to decrypt passwords in the catalog, for example, catalogs from older servers with different keys. There are two ways to specify the import key:

- Give the hexadecimal bytes of the key.
- Give the alias of a key in the server's keystore (.jrsks).

js-import Options to Specify an Import Key	
Option	**Explanation**
`--input-zip`	Specifies the file path to a zipped input catalog from an older JasperReports Server instance that was configured to use custom keys.
`import options`	The standard import options specifying the import behavior, for example `--update --skip-user-update`. These options are documented in the JasperReports Server Administrator Guide.
`--keyalias`	When used with input-zip, this option specifies a key in the server's keystore (.jrsks) to use when decrypting passwords in the import catalog.
`--secret-key`	Lets you specify the hexadecimal representation of a key to be used as a one-time import key to decrypt any passwords in the input-zip.
`--keyalg`	When used with `--secret-key`, this option defines the algorithm, either AES or DES, for the hexidecimal key. The default is AES.

The following example shows how to import a catalog with a custom key.

```
js-import.sh --secret-key "0x1c 0x40 0xb9 0xf6 0xe2 0xd3 0xf9 0xd0 0x5a
0xab 0x84 0xe6 0xd4
0xe8 0x5f 0xed" --input-zip myExport.zip
```

The following example shows how to import a catalog using a key already saved in the keystore.

```
js-import.sh --keyalias productionServerKey --input-zip myExport.zip
```

Importing a Key from the Command Line

If you have many catalogs to import from a server with a custom key, the js-import script has different options to import the key and add it to the local keystore (.jrsks by default). You can then use the example in the previous section to specify the new key by its alias when importing. There are three ways to define the key to import:

- Provide the hexadecimal bytes of the key.
- Provide a keystore and the alias (and password) of a key that it contains.
- Request a random key to be generated and associated with an alias (and password) you provide.

js-import Options to Import a Key

Option	Explanation
`--input-key`	This option specifies a key to be added to the server's keystore. This option should be followed by the hexadecimal representation of the key, or by the `--keystore` or `--genkey` options (see below). Use the `--keyalias`, `--keyalg`, and `--keypass` options to add properties to the definition of the key in the keystore.
`--keystore`	This option specifies the path and filename of a keystore file from which to read and copy the key designated by the `--keyalias` option. Also specify the `--storepass` option to access the source keystore and the `--keypass` option to give the key's password in the keystore file.
`--storepass`	This option specifies the password for the keystore file from which to read and copy the key designated by the `--keyalias` option.
`--genkey`	This option triggers the import utility to generate a random key to be added to the server's keystore with the alias and password you specify (in the other options below) so that you can later access and use it. This option is a shortcut for creating a random key in an external keystore and then importing it with the `--keystore` option.
`--keyalias`	When used with a hexidecimal input-key, it specifies the alias of the new key to be imported. When used with the `--genkey` option, this specifies the alias of the new key to create. When used with the `--keystore` option, it specifies the alias of the key to be copied, and the copy of the key will have the same alias.
`--keypass`	When used with a hexidecimal input-key, it specifies the password of the new key to be imported. When used with the `--genkey` option, this specifies the password of the new key to create. When used with

js-import Options to Import a Key	
Option	**Explanation**
	the `--keystore` option, it specifies the password of the key to be copied, and the copy of the key will have the same password.
`--keyalg`	When used with a hexidecimal input-key, this option defines the algorithm, either AES or DES, for the key being imported in the keystore. When used with the `--genkey` option, this specifies the algorithm to use when creating the new key.
`--keysize`	When used with a hexidecimal input-key, this option defines the key length in bits, usually 128 or 256, for the key being imported in the keystore. When used with the --genkey option, this specifies the length of the new key to create.
`--visible`	Specify this flag to make the imported key displayed in the list of Custom Keys as shown in Specifying a Custom Key in the Import UI. When omitted, the imported key is not available for UI import operations, only through command-line import operations (using `--keyalias`).
`--keylabel`	When --visible is specified, this option lets you specify the name of the key to display in the Custom Key selection interface. If --visible is specified without this option, the key alias is displayed in the UI. This option has no effect when --visible is not specified.

The following example shows how to add a key to the keystore, so it can be used for other import operations:

```
js-import.sh --input-key "0x59 0xe3 0xd9 0xce 0x7f 0x34 0xab 0x27 0xb8
0xdf 0xc3 0x7e 0x01 0xab
             0x4d 0x6c" --keyalias productionKey --keyalg AES --keypass
productionKeyPass
             --visible --keylabel ProductionServerKey
```

The following example shows how to copy a key from an external keystore file into the default keystore.

```
js-import.sh --input-key --keystore ./mystore --storepass password --
keyalias productionKey2
             --keypass productionKeyPass2 --visible --keylabel
ProductionServerKey2
```

Specifying a Custom Key in the Import UI

After adding custom keys to the keystore from the command line using the --visible option, you can also select the keys in the UI during import operations. The keys are identified by their alias or label if given.

1. Log in as system administrator (superuser by default).
2. Select Manage > Server Settings, then click Import in the left-hand panel.
3. In the right-hand panel, browse the file system to enter the catalog file you want to import.

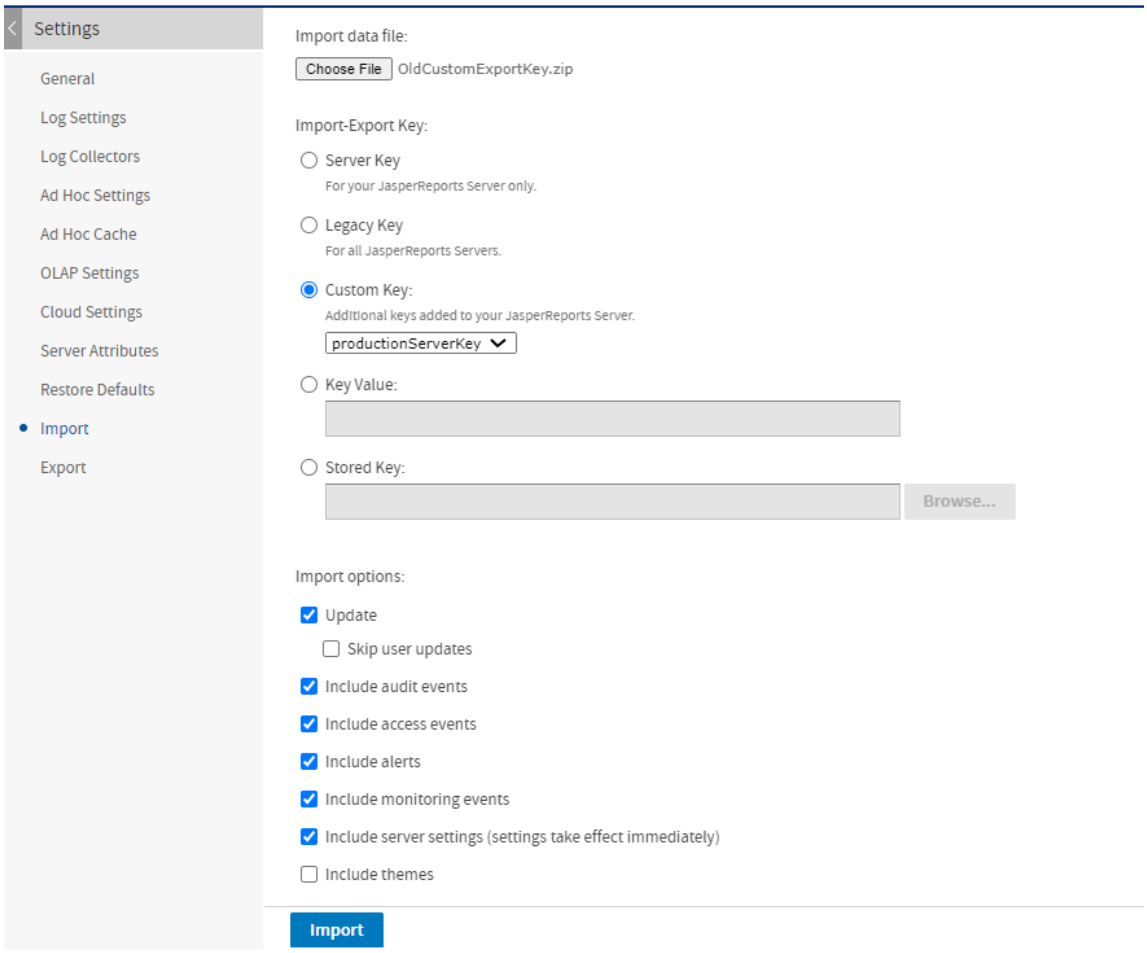

Figure 4: Import UI with Key Value

4. When the server's keystore contains custom keys, the list of keys appears as the third bullet. Note that key files in the repository do not appear in this list, only custom keys in the keystore. Each key in the list is identified by its label if it was defined on import, otherwise by its alias. Choose this bullet and select your key from the dropdown list.
5. Select your import options and click Import.

If the key does not decrypt the catalog file, you get an error message, otherwise the import proceeds.

Specifying an Export Key on the Command Line

As with the export UI, you can specify custom keys when exporting from the command line. For example, you can create an export catalog that can be imported into another server instance that has different keys. There are three ways to specify the export key:

- Provide the hexadecimal bytes of the key.
- Give the alias of a key in the server's keystore (.jrsks).
- Request a random key be generated and displayed on the console.

js-export Options to Specify the Export Encryption Key

Option	Explanation
`--output-zip`	Specifies the name of a zipped output catalog that will use the custom keys specified by the other options in this table. The index.xml file in the catalog contains new attributes to handle keys.
`export options`	The standard export options specifying the resources to export and export behavior, for example `--uris ... --skip-dependent-resources`. These options are documented in the JasperReports Server Administrator Guide.
`--secret-key`	This option lets you specify the hexadecimal representation of a key to be used as a one-time export key to encrypt any passwords in the output-zip.
`--keyalg`	This option may be specified only with the `secret-key` option above. It specifies the cryptography algorithm of the given key, either AES or DES (RSA is not supported by default). When not specified, the secret-key is assumed to use the same algorithm as the server's default import-export key (AES).
`--keysize`	This option may be specified only with the `secret-key` option above. It specifies the key length in bits (usually 128 or 256) to apply to the given key. When not specified, the size is the same as the server's default import-export key (128).
`--keyalias`	This option specifies a key in the keystore to use when encrypting

js-export Options to Specify the Export Encryption Key	
Option	**Explanation**
	passwords in the export catalog, instead of the default import-export key.
`--keypass`	This option may be specified only with the `keyalias` option above. It is required in the unlikely situation where the key with the given alias is held in the keystore, but the .jrsksp file is not configured with the password to access the key.
`--genkey`	This option generates a random key using the same algorithm and key size as the server's default import-export key (AES-128), and uses it to encrypt passwords in the export catalog. If the export is successful, the js-export script prints the key's hexadecimal representation and a unique alias for it on the console where it is running. You need to specify the same key when importing the catalog, for example with the `js-import --input-key` option or as explained in Entering a Key Value in the Import UI.

The following example shows how to export a catalog with passwords encrypted with a custom key:

```
js-export.sh --uris /public/samples/AccountList --output-zip
myExport.zip
--secret-key="0x6f 0x00 0xf1 0xbd 0x46 0x1f 0x62 0xa1 0x03 0x56 0x13
0xda 0x07 0x00 0x7c 0x10"
```

The following example shows how to export a catalog with passwords encrypted by one of the keys in the keystore:

```
./js-export.sh --uris /public/samples/AccountList --output-zip
myExport.zip
--keyalias productionServerKey --keypass "mykeypw2"
```

Exporting a Key from the Command Line

The js-export utility can also be used to export one of the keys from the server's keystore (.jrsks). In addition to exporting repository resources in an export catalog, the following options generate a Java keystore file containing the specified key. The keystore is a secure file protected by the given password that can be used with the keytool utility or with the keystore options of the js-import tool. For more information, see Importing a Key from the Command Line.

js-export Options to Export a Key

Option	Explanation
`--keyalias`	When used with the `--destkeystore` option below, this specifies the alias of the key to be exported from the server's keystore, and the copy of the key will have the same alias. If the export includes resources in an export catalog, any passwords it contains will be encrypted with this key.
`--keypass`	This option may be specified only with the `keyalias` option above. It is required in the unlikely situation where the key with the given alias is held in the keystore, but the .jrsksp file is not configured with the password to access the key.
`--genkey`	This option generates a random key using the same algoritm and key size as the server's default import-export key (AES-128), and exports it in the specified keystore. If the export includes resources in an export catalog, any passwords it contains will be encrypted with this key.
`--destkeystore`	This option specifies the filename of a keystore file to create, in order to export the key designated by the `--keyalias` option. You must also specify the `--deststorepass` option.
`--deststorepass`	This option specifies the password for the keystore file to be created to hold the exported key.
`--destkeypass`	Specifies a new password for the key in the newly created keystore.

You may specify both an output zip catalog and list of resources to export, as well as a key alias and keystore filename. In the following example of this, the export creates two output

files, the zip catalog and the keystore, and any passwords in the catalog are encrypted with the same key that was exported.

```
js-export.sh --everything --output-zip myExport.zip --destkeystore
mystore --deststorepass storepw --genkey
```

The server that generates this key stores a copy of it in its keystore, and if you import the key to another server, they share the key. If you examine the key with the keytool utility, it has a unique alias name:

```
keytool -list -v -keystore ./mystore -storetype jceks

Enter keystore password: *******
Keystore type: JCEKS
Keystore provider: SunJCE
Your keystore contains 1 entry
Alias name: ced6b744-033d-4516-b293-c4776035a6f1
Creation date: Dec 12, 2019
Entry type: SecretKeyEntry

*****************************************
*****************************************
```

Now you can specify this unique alias name whenever importing or exporting from your two servers, the encryption is mutually compatible, and you will not need to export or import keys anymore.

Sharing Custom Keys

The import and export functionality can be used to share export catalogs between servers that have different keys, for example an old server with custom keys. If you wish to share catalogs between two servers that are both on the same release, you can add the keys directly to the server's keystore.

For example, if you have a test server for developing reports and dashboards, and a production server where users need them, you can transfer them by exporting from one and importing into the other. To do so, both servers need the same import-export key, but after installation, each will have a different and random key. The recommended solution is to generate the new key in a new keystore file, and then imports it to both servers.

The following procedure assumes you are familiar with the command-line keytool utility. For more information, see the Java keytool reference.

To create and import a custom key to multiple servers:

1. Generate your custom keys in a keystore. In this example, we generate two keys, which overwrite the default import-export key and the diagnostic key.

```
keytool -genseckey -keystore ./mystore -storetype jceks -storepass
storepw
-keyalg AES -keysize 128 -alias importExportEncSecret -keypass
myimportexportpw

keytool -genseckey -keystore ./mystore -storetype jceks -storepass
storepw
-keyalg AES -keysize 128 -alias diagnosticDataEncSecret -keypass
mydiagnosticpw
```

   Use the keytool utility again to verify your new keys:

```
keytool -list -v -keystore ./mystore -storetype jceks

Enter keystore password: *******
Keystore type: JCEKS
Keystore provider: SunJCE
Your keystore contains 2 entries
Alias name: diagnosticdataencsecret
Creation date: Dec 12, 2019
Entry type: SecretKeyEntry

*****************************************
*****************************************

Alias name: importexportencsecret
Creation date: Dec 12, 2019
Entry type: SecretKeyEntry

*****************************************
*****************************************
```

2. Copy the keystore file to both servers using a secure method such as scp, sftp, or rsync.

```
scp ./mystore jrsusr@bi-test.example.com:/opt/jasperreports-
```

```
server/jasperreports-server-x.x.x/buildomatic/

scp ./mystore jrsusr@bi-production.example.com:/opt/jasperreports-
server/jasperreports-server-x.x.x/buildomatic/
```

3. Log in to the first server (bi-test) as the system user who installed JasperReports Server (jrsusr) and stop the app server. Then import the keys with the following commands:

```
cd /opt/jasperreports-server/jasperreports-server-
x.x.x/buildomatic/

./js-import.sh --input-key --keystore ./mystore --storepass storepw
--keyalias importExportEncSecret --keypass myimportexportpw

./js-import.sh --input-key --keystore ./mystore --storepass storepw
--keyalias diagnosticDataEncSecret --keypass mydiagnosticpw
```

4. Log in to the second server (bi-production) as the system user who installed JasperReports Server (jrsusr) and stop the app server. Then import the keys with the same commands as above.
5. Restart both app servers, and now they will use your custom keys.

In this example, the two custom keys were given the same alias as the keys that are created by default in the server's own keystore (/users/jrsuser/.jrsks). As a result, the custom keys overwrite the server's default keys, which will be used in any operation where the default keys are used. This has the following consequences:

- Export catalogs can be shared between the two servers. Any passwords in the export catalog will be encrypted with the new importExportEncSecret on one server and decrypted with the same key on the other server. Export catalogs can be moved from the test server to the production server for deployment and vice versa for debugging, without exchanging keys or even specifying key aliases.
- Log collectors are encrypted with a known key. For security, the diagnostic information in the log collector is encrypted with the diagnosticDataEncSecret key. Now when you download the log collector zip file, you need a copy of the mystore keystore file with your new diagnosticDataEncSecret key to decrypt it.

The keystore you created in this procdure contains the same keys as your production server, and could thus be used to access sensitive data. Be sure to delete the copies of the keystore you no longer need, and safeguard the passwords you used in these commands.

Configuring Encryption

In a normal server installation for evaluation or production environments, once the server is installed or upgraded, the use of the keystore is transparent and requires no further configuration. If you need to handle keys for old servers, you can use the import and export tool so that servers have the keys they need.

For special situations, it is possible to customize the server's use of encryption, such as configuring specific ciphers or cipher length. For example, the server only supports the AES (Advanced Encryption Standard) and DES (Data Encryption Standard) algorithms for encryption by default. If you wish to use a different algorithm such as RSA (Rivest–Shamir–Adleman), you need to change the cipher implementation to one that supports RSA. Configuring encryption is best done before installation, but can also be done after.

However, the specifics of configuring encryption in JasperReports Server are beyond the scope of this document. This section is intended only to introduce the concepts and guidelines for advanced use cases. Administrators wishing to customize encryption settings must be proficient in the cryptography libraries of the Java Cryptography Architecture (JCA) and know the risks to avoid.

Before you consider modifying the encryption configuration, keep in mind the following:

- Always make a backup of the server and original keystore files before configuring encryption. The keystore files are unique to every installation and the server is inaccessible without them.
- If you need to modify the encryption settings, do it before provisioning your server with production data.
- Even if the server has no production data, you should export the entire repository along with the export key before proceeding. Default accounts such as superuser and jasperadmin must be reimported later to work with your new encryption settings.
- You must be familiar with the cryptographic concepts and details of the keystore APIs. For more information, see the Java Cryptography Architecture (JCA) Reference Guide and its section on key management.
- Encryption may be configured in the .jrsksp file and also in beans and properties in other configuration files. This may create a complex configuration where values override other settings, and what appears in the keystore properties file may not be the final configuration at runtime. While this may be desirable or necessary for your configuration, it creates complex dependencies and risk.

- Be sure to document your new configuration, including any secondary configuration file dependencies.
- Incorrect configuration of the keystore or importing with the wrong keys may corrupt your data or make it impossible to access the server. Therefore, it is critical to know and test your encryption configuration and import procedure. Testing on an isolated and empty evaluation server instance is recommended.
- Be sure to securely delete any draft copies of your encryption configuration and wipe any test servers to ensure the security of your production server.
- After configuring the encryption, importing your export catalog, and testing your server, remember to back up your new keystore files, including any other configuration files that may contribute to the encryption settings.

For security, the .jrsksp file is Base64 encoded so that it is not a plain text file. To read and modify the file, the system user who installed the server must decode the file, for example:

Windows: `certutil -decode .jrsksp myconfig.txt`
Linux: `cat ~/.jrsksp | openssl base64 -d > myconfig.txt`

Inside the keystore is the configuration for each of the following keys:

Key Alias	Description
buildSecret	Key for encrypting passwords and sensitive values in configuration files in the file system.
importExportEncSecret	Key for all import and export operations of the new server.
deprecatedImportExportEncSecret	Key for importing from previous versions of the server.
passwordEncSecret	Key for encrypting user passwords and other sensitive content in the server's internal database (the repository).
deprecatedPasswordEncSecret	Key for upgrading to 7.5 without exporting everything. However, the best practice is to export everything, modify the configuration, and reimport, as described in the upgrade section.

Key Alias	Description
diagnosticDataEncSecret	Key for encryption of log collector output. Use js-export to export this key and decrypt the output file.
httpParameterEncSecret	The key used for HTTP parameter encryption in releases prior to 7.5, now deprecated. If upgrading from a previous release, this key needs to be exported from the old keystore, and imported into the server.
deprecatedHttpParameterEncSecret	This key is not used.

If you have added the key to the keystore with the js-import command, then they have their alias and password defined here as well.

Configuration properties are typical Java properties (name=value), one per line. Special precaution needs to be taken while working with the .jrsksp properties because certain symbols must be escaped with a backslash. For example, #, :, \, and = are represented as \#, \:, \\, and \= to be interpreted correctly.

Configuration values with the same name may be set in other files and take precedence. This may be necessary for certain configurations, but it is more complicated and may lead to errors. If possible, keep all the encryption settings in the .jrsksp file.

The values of the keys themselves are encrypted and stored in the .jrsks file. Only the Java keytool utility in the JDK (Java Development Kit) can read, write, or modify keys in the keystore file.

The procedure for configuring encryption depends on whether you can do it before installation, which is easier, or after.

Configuring Encryption Before Installation

The easiest way to customize the encryption on your server is to modify configuration files before doing a WAR file installation. That way, the installation scripts use your settings when generating keys and the keystore, and all encryption is performed once with the properties you want.

You may need to install an evaluation server to access it's .jrsksp file and determine which settings you want to modify. Of course, you should also test your custom encryption configuration on test installations before installing your production servers.

The default values of properties may be modified by defining them in default_master.properties, after copying the appropriate <name>_master.properties file and before running the installation scripts. For example, you could specify predefined passwords for each key instead of randomly generated ones. When the installation runs, it performs the keystore creation and all initial encryption using your configuration.

After the installation is successful, you should be sure to back up and then delete any files that contain sensitive encryption configuration values such as passwords. Also document your custom installation for ease of maintenance and support.

Configuring Encryption After Installation

If possible, you should customize your encryption configuration before you install the server. In case if it is not possible, you can configure encryption after the fact, but the procedure is much longer, depending on the settings you need to change. For example, changing the password of a key does not impact contents that are already encrypted, but changing the strength of the password cipher means you need to re-encrypt all user passwords.

The following procedure gives the general steps for changing the encryption configuration of a server after it has been installed and provisioned. This assumes that your changes require the server's contents to be re-encrypted.

Procedure

1. Export the entire contents of the server including the import-export cipher.
2. Stop the server.
3. Decode the .jrsksp file as described above, and make changes to its settings. It is also possible to add encryption configuration settings in the applicationContext-security.xml file if necessary.
4. Depending on what you modify in the configuration, you may need to generate or modify keys using the keytool utility. For example, if you want a stronger cipher, you need to generate the longer key to replace the existing one. If you change a password in the properties file, you must also set the password in the keystore with keytool. Ensure the keystore is updated in the same way as the .jrsksp properties file.

5. After all, the modifications .jrsksp file must be Base64 encoded and replaced in the user's home directory with the updated keystore (.jrsks) file.
6. Restart the server.
7. Import your server's export catalog with its old export key (if the export key has changed). If the configuration is coherent and the keys are correct, you should be able to log in.

As mentioned previously, the details and complexity of these procedures are beyond the scope of this document. You must have the knowledge and experience with the Java Cryptography Architecture to successfully modify the encryption configuration.

Legacy Encryption Configurations

In previous releases of the server, encryption was often defined in configuration files and could be modified. As of release 7.5, all encryption keys are stored in the server's keystore (.jrsks) with the matching configuration in the keystore properties file (.jrsksp). However, in certain cases where you wish to customize how the encryption works, you could use the legacy configuration.

The following sections describe legacy encryption configurations that have been replaced by the keystore functionality, but could be used as documentation for advanced encryption configuration. In general, if you configure a key through a configuration file, it is used instead of the key from the keystore:

- Static Key Encryption for HTTP parameters.
- Encrypting User Passwords in the internal database.
- Encryption Options for encrypting passwords in configuration files.

Application Security

This chapter describes the configuration settings that protect JasperReports Server and its users from unauthorized access. The configuration properties appear in two locations:

- Some properties must be configured during the installation and deployment phase, before users access the server. These settings are configured through files used by the installation scripts. These settings are available only when performing a WAR file installation.
- Properties you can configure after installation are located in files in various folders. Configuration file paths are relative to the <js-install> directory, which is the root of your JasperReports Server installation. To change the configuration, edit these files then restart the server.

Because the locations of files described in this chapter vary with your application server, the paths specified in this chapter are relative to the deployed WAR file for the application. For example, the applicationContext.xml file is shown as residing in the WEB-INF folder. If you use the Tomcat application server bundled with the installer, the default path to this location is:

C:\Program Files\jasperreports-server-9.0.0\apache-tomcat\webapps\jasperserver-pro\WEB-INF

Use caution when editing the properties described in this chapter. Inadvertent changes may cause unexpected errors throughout JasperReports® Server that may be difficult to troubleshoot. Before changing any files, back them up to a location outside of your JasperReports® Server installation.

Do not modify settings not described in the documentation. Even though some settings may appear straightforward, values other than the default may not work properly and may cause errors.

This chapter contains the following sections:

- Restricting File Uploads
- Restricting Groovy Access
- Hiding Stack Trace Messages
- Defining a Cross-Domain Policy for Flash
- Enabling SSL in Tomcat
- Disabling Unused HTTP Verbs
- Configuring HTTP Header Options
- Setting the Secure Flag on Cookies
- Setting httpOnly for Cookies
- Protection Domain Infrastructure in Tomcat
- Encrypting Passwords in URLs
- Host Header Injection Protection

Encrypting Passwords in Configuration Files

In JasperReports Server version 5.5 or later, administrators can obfuscate passwords that appear in the configuration files. This satisfies security audit requirement and prevents the passwords from being seen by unauthorized individuals. Typically, the following are encrypted:

- The password to JasperReports Server's internal database (jasperserver).
- The passwords to the sample databases (foodmart and sugarcrm).
- On Tomcat, passwords in JNDI resource definitions.

You can change the configuration to encrypt:

- The password for the mail server used by the scheduler (quartz.mail.sender.password).
- The password for LDAP external authentication.

Passwords in configuration files are encrypted during JasperReports Server installation. If the installation deploys to the Tomcat application server, the database password is also automatically encrypted in the JNDI configuration (in the file context.xml).

Full password security cannot be guaranteed from within JasperReports Server. A user with sufficient privileges and knowledge of JasperReports Server can gain access to the encryption keys and the configuration passwords. While you could require a password on every server restart, this is impractical for most users. The only practical way to guarantee password security is through backup and restriction of access to the keystore property file.

Encrypting Configuration Passwords on Tomcat

To encrypt passwords in a Tomcat installation, modify the installation procedure:

1. Depending on the database you use, copy the installation configuration file as usual:

   from: <js-install>/buildomatic/sample_conf/<database>_master.propertiesto: <js-install>/buildomatic/default_master.properties

2. Edit the default_master.properties file:
   - Enter values specific to your installation.
   - Enter your passwords in plain text.
   - Turn on configuration file encryption by uncommenting the encrypt=true property. You don't have to uncomment any other encryption properties because they all have the default values shown.
   - Unless you are using Oracle, uncomment propsToEncrypt and set it to dbPassword,sysPassword.
   - Optionally, specify additional properties to encrypt as described in Encrypting Additional Properties in default_master.properties.
   - Optionally, change the settings for configuration file encryption as described in Encryption Options.
3. Run the buildomatic installation script (js-install) and all other installation steps according to the JasperReports® Server Installation Guide. This has the following effects:
   a. The plain text passwords in default_master.properties are overwritten with their encrypted equivalents. There is no warning when you run js-install with encrypt=true.
   b. The encrypted passwords are propagated to all configuration files.

c. The installation proceeds and copies the files to their final locations.

4. After installation, passwords are encrypted in the following locations:

- In all server configuration files in .../WEB-INF/applicationContext*.xml.
- In JNDI definitions in .../META-INF/context.xml.
- In the default_master.properties files that remain after installation.

If you get an error like the following when restarting the server:

```
javax.naming.NamingException: KeystoreManager.init was never called or there are errors instantiating an instance
```

You may need to add the following to your Tomcat service start properties:

```
-Duser.home=c:\Users\<TomcatUser>
```

Encrypting Configuration Passwords on Enterprise Servers

Most enterprise servers, like JBoss, Glassfish, WebSphere, and WebLogic, have proprietary ways to set up password encryption. You should use these encryption methods. JasperReports® Server doesn't automatically set up encrypted passwords for these servers during deployment. In this case, you can encrypt the passwords in the buildomatic file after deployment:

1. Deploy JasperReports Server to your enterprise server as specified in the JasperReports Server Installation Guide. The resulting JasperReports Server instance will have unencrypted JNDI data source passwords. If you want to encrypt these passwords, refer to your application server's documentation.
2. After the server has been successfully configured, encrypt the JasperReports Server configuration files as follows:
    a. In default_master.properties, turn on encryption by uncommenting encrypt=true.
    b. Run the target js-ant refresh-config. This will remove and recreate all the configuration files without deploying them to the application server. Now the buildomatic files will have the database passwords encrypted. You should still be able to execute import/export or other scripts.

3. After running js-ant refresh-config, you will need to manually copy the encrypted password to the application server configuration file. Copy the encrypted password from the updated default_master.properties file to the corresponding database connection files on the server, such as the /META-INF/context.xml file for Tomcat.

Do not run js-install or js-ant deploy-webapp-pro. These commands overwrite the WAR file created in step 1 and render the server data sources inaccessible. If you need to redeploy the WAR file, reset the database passwords to plain text in your default_master.properties file and start again with step 1.

Encrypting Additional Properties in default_master.properties

You can encrypt additional properties in the default_master.properties file. To work correctly, these properties need to be decrypted when used. Currently decryption is supported for properties loaded into the Spring application context via the propertyConfigurer bean in applicationContext-webapp.xml.

If a property is defined via JNDI, we recommend pointing there instead of encrypting:

```
<property name="password">
    <jee:jndi-lookup jndi-name="java:comp/env/emailPassword" />
</property>
```

The following code sample shows the propertyConfigurer bean in applicationContext-webapp.xml:

```
<bean id="propertyConfigurer"
class="com.jaspersoft.jasperserver.api.common.properties.DecryptingPrope
rtyPlaceholderConfigurer">
    <property name="locations">
      <list>
        <value>/WEB-INF/classes/hibernate.properties</value>
        <value>/WEB-INF/js.quartz.properties</value>
        <value>/WEB-INF/js.spring.properties</value>
        <value>/WEB-INF/js.scheduling.properties</value>
        <value>/WEB-INF/mondrian.connect.string.properties</value>
        <value>/WEB-INF/js.diagnostic.properties</value>
        <value>/WEB-INF/js.aws.datasource.properties</value>
        <value>/WEB-INF/js.config.properties</value>
        <value>/WEB-INF/js.externalAuth.properties</value>
```

```
      </list>
    </property>
    ...
</bean>
</pre>
```

Because we extended Spring's PropertyPlaceholderConfigurer class as DecryptingPropertyPlaceholderConfigurer, all the loaded properties are scanned for the special marker ENC-<value>-. If that marker is found around the property value, that property is decrypted before it is loaded into the Spring context.To determine if your property is scanned by propertyConfigurer, search the files in propertyConfigurer's location to see if it is defined in one of these files. For example, suppose you want to encrypt the password property of the reportSchedulerMailSender bean in applicationContext-report-scheduling.xml:

```
<bean id="reportSchedulerMailSender"
class="org.springframework.mail.javamail.JavaMailSenderImpl">
  <property name="host" value="${report.scheduler.mail.sender.host}"/>
  <property name="username"
value="${report.scheduler.mail.sender.username}"/>
  <property name="password"
value="${report.scheduler.mail.sender.password}"/>
  <property name="protocol"
value="${report.scheduler.mail.sender.protocol}"/>
  <property name="port" value="${report.scheduler.mail.sender.port}"/>
  <property name="javaMailProperties">
    <props>
      <prop key="mail.smtp.auth">false</prop>
    </props>
  </property>
</bean>
```

The use of the ${...} syntax tells you that report.scheduler.mail.sender.password is most likely defined via the propertyConfigurer bean. Search through the propertyConfigurer locations to verify. This property is defined in /WEB-INF/js.quartz.properties as follows: report.scheduler.mail.sender.password=${quartz.mail.sender.password}. Once you've verified that the quartz.mail.sender.password property can be encrypted using default-master.properties, you set up encryption before installation as follows:

1. Set the password for quartz.mail.sender.password in default-master.properties:

   ```
   quartz.mail.sender.password=cleartextpassword
   ```

2. Uncomment the `encrypt=true` property in the same file.
3. Uncomment `propsToEncrypt=dbPassword` in default-master.properties.
4. Add `quartz.mail.sender.password` to `propsToEncrypt`:

```
quartz.mail.sender.password=cleartextpassword
...
encrypt=true
propsToEncrypt=dbPassword,quartz.mail.sender.password
```

5. Configure and install your JasperReports® Server WAR installation as described in the JasperReports Server Installation Guide.
6. Verify that report.scheduler.mail.sender.password was encrypted in both default-master.properties and in /WEB-INF/js.quartz.properties.

Password Encryption for External Authentication

As of JasperReports Server 5.6, you can encrypt the passwords in the external authentication configuration files for LDAP and external database authentication. Here we cover only the encryption of these passwords; for details about configuring external authentication, see the JasperReports Server External Authentication Cookbook. To enable encryption during installation, property values in the external authentication sample configuration are referenced from other configuration files. For example, if you're using LDAP to authenticate, the sample configuration file contains the following reference to the LDAP password:

```
<bean id="ldapContextSource"

class="com.jaspersoft.jasperserver.api.security.externalAuth.ldap.JSLdap
ContextSource">
  <constructor-arg value="${external.ldap.url}" />
  <property name="userDn" value="${external.ldap.username}" />
  <property name="password" value="${external.ldap.password}"/>
</bean>
```

The values referenced by the ${...} format are defined in the js.externalAuth.properties file and imported into the Spring context via the propertyConfigurer. For example, the LDAP properties are defined in js.externalAuth.properties as follows:

```
external.ldap.url=${external.ldapUrl}
external.ldap.username=${external.ldapDn}
external.ldap.password=${external.ldapPassword}
```

The ${...} syntax again references other configuration properties that must be set in default_master.properties before installation or upgrade. The following example shows the syntax of the properties in the default_master.properties file:

```
external.ldapUrl=ldap://hostname:389/dc=example,dc=com
external.ldapDn=cn=Administrator,dc=example,dc=com
external.ldapPassword=password
```

To encrypt the password property, set the following values in default_master.properties before installation or upgrade:

```
external.ldapPassword=cleartextpassword
...
encrypt=true
propsToEncrypt=dbPassword, external.ldapPassword
```

During the installation process, the password value in default_master.properties and its reference in js.externalAuth.properties are overwritten with the encrypted value.If your external authentication is configured to create organizations for external users, and you're using JasperReports Server 6.0, or later, there is another password to encrypt. When external authentication creates an organization, it uses the information in ExternalTenantSetupUser of the externalTenantSetupProcessor bean to create the organization administrator.

```
<bean
class="com.jaspersoft.jasperserver.multipleTenancy.security.externalAuth
.processors.
            MTAbstractExternalProcessor.ExternalTenantSetupUser">
  <property name="username" value="${new.tenant.user.name.1}"/>
  <property name="fullName" value="${new.tenant.user.fullname.1}"/>
  <property name="password" value="${new.tenant.user.password.1}"/>
  <property name="emailAddress" value="${new.tenant.user.email.1}"/>
  <property name="roleSet">
    <set>
      <value>ROLE_ADMINISTRATOR</value>
      <value>ROLE_USER</value>
    </set>
```

```
  </property>
</bean>
```

The values referenced by the ${...} format are defined in the js.config.properties file as follows:

```
## New tenant creation: user config
new.tenant.user.name.1=jasperadmin
new.tenant.user.fullname.1=jasperadmin
...
new.tenant.user.password.1=jasperadmin
new.tenant.user.email.1=
```

The default values for new tenant (organization) administrators in js.config.properties apply only to external authentication. They do not apply to organizations created by administrators through the UI or REST interface.

To encrypt this password, modify the js.config.properties file as follows:

```
new.tenant.user.password.1=${tenant.user.password}
```

Then add the following lines to default_master.properties before installation or upgrade:

```
tenant.user.password=cleartextpassword
...
encrypt=true
propsToEncrypt=dbPassword, external.ldapPassword, tenant.user.password
```

During the installation process, the password value in default_master.properties and its reference in js.config.properties are overwritten with the encrypted value.

Encryption Options

As of JasperReports Server 7.5, all encryption in the server relies on cryptoghaphic keys stored in the server's keystore. For more information, see Key and Keystore Management.The configuration files and properties described in this section are no longer used by this feature. They are documented here only for legacy purposes.

In buildomatic installation scripts, the passwords are symmetrically encrypted: the same secret key is used for both encryption and decryption. The key and its containing keystore file are randomly generated on each machine during the first JasperReports® Server

installation. All subsequent JasperReports® Server installations on the same server rely on the same keystore; they do not regenerate the key.The keystore is an encrypted file used to securely store secret keys. JasperReports® Server uses keystore properties to access the keystore. Both the keystore and keystore properties files are created by default in the user home directory. Alternatively, before running js-install, you can specify different locations for the keystore and keystore properties files via the environmental variables ks and ksp.By default, database passwords are encrypted with the AES-128 algorithm in Cipher Block Chaining mode with PKCS5 padding. The AES algorithm is the current industry encryption standard. You can choose to modify the encryption strength by choosing either a different algorithm, a longer secret key size (for example AES-256), or a different encryption mode.Edit the following properties in your default_master.properties and set these options. If a property is commented out, the default is used:

Property	Description	Default
build.key.algo	The algorithm is used to encrypt the properties in configuration files.	AES
build.key.size	Size of the encryption key as in AES-128.To increase the key size, if it has not been done before, you might have to install "Unlimited Strength Jurisdiction Policy Files" from the Oracle site for your Java version. To install the files, download US_ export_policy.jar and local_policy.jar. AFTER backing up the old files, extract the jars into %JAVA_ HOME%/jre/lib/security directory.Alternatively, you may download one of the reputable providers such as Bouncy Castle (ships with JasperReports® Server). You would need to add the Bouncy Castle provider to the list in %JAVA_HOME%/jre/lib/security/java.security file: security.provider.<seq number>= org.bouncycastle.jce.provider.BouncyCastleProvider	128 (bits)
enc.transformation	So-called encryption mode. See Java's javax.crypto documentation to understand the modes and padding better.	AES/CBC /PKCS5 Padding
enc.block.size	The size of the block that is encrypted. Encrypted	16 (bytes)

Property	Description	Default
	text can contain many blocks. Usually the block is changed together with the encryption algorithm.	
propsToEncrypt	A comma separated list of the properties to encrypt.	dbPassword

Configuring CSRF Protection

Cross-Site Request Forgery (CSRF) enables an attacker to either gain information or perform actions while a user is logged into JasperReports Server. The user may be logged in another window or tab of the same browser. This is called session riding. For example, a server administrator logged into JasperReports Server is tricked into opening a malicious website that invisibly uses the browser session to create a user with administrator permissions. The attacker can then use it to access the system later.

JasperReports Server uses the latest release of CSRFGuard from OWASP (Open Web Application Security Project). CSRFGuard verifies that every POST, PUT, and DELETE request submits a valid token previously obtained from the server. This includes every request submitted via forms or AJAX. When a malicious request arrives without the proper token, the server does not reply and logs an error for administrators to analyze later.

Tokens are sent in HTTP headers or parameters, and the entire exchange is invisible to users. Tokens have the following syntax:

OWASP_CSRFTOKEN: K8E9-L4NZ-58H6-Z4P2-ZG75-KKBW-U53Z-ZL6X

In the default configuration of the server, CSRF protection is active. We recommend leaving this setting unchanged.

However, to fully implement CSRF and secure your server, you must configure the domain whitelist as explained in the next section.

CSRF Protection
Configuration File
.../WEB-INF/csrf/jrs.csrfguard.properties

CSRF Protection		
Property	Value	Description
`org.owasp.csrfguard.Enabled`	`true` <default> `false`	Turns CSRF protection on or off. By default, CSRF protection is enabled. Setting this value to false disables the CSRF filter and allows any request regardless of tokens.

This configuration file contains many settings that are preconfigured for JasperReports Server. We do not recommend changing any other settings. In particular, the two `configOverlay` properties are unreliable and not supported.

After updating the `jrs.csrfguard.properties` file, you must restart JasperReports Server for the new values to take effect.

Setting the Cross-Domain Whitelist

In all cases, even if you do not use Visualize.js, you must configure the whitelist. Never use a server in production with the default whitelist.

Applications that use the embedded Visualize.js library typically access JasperReports Server from a different domain. For this reason, CSRF protection includes a whitelist of domains that you specifically allow to access the server. Initially, all your Visualize.js applications can access the server, but you should configure the whitelist so that only your domains have access. Then, any Visualize.js request from an unknown domain fails with HTTP error 401, and the server logs a CSRF warning.

The domain whitelist is implemented through attributes named `domainWhitelist` at the user, organization, or server-level. You may specify different values at each level. The values are defined according to the attribute hierarchy. In addition, the `domainWhitelist` attribute is defined with administrator permissions, implying that organization admins can set their own values. The attributes are set through the server UI or through the REST API. For more information on how to define attributes and how their values are determined by hierarchy, refer to the JasperReports Server Administrator Guide.

There are four cases listed in the table below. Choose the one suited to your use of Visualize.js.

Cross-Domain Whitelist		
Configuration Location		
The attribute `domainWhitelist` is defined at the server level. In addition to setting any alternate values at the organization or user levels, for security, always set the server level as described below: • Server level: as system admin (`superuser`), select **Manage** > **Server Settings** then **Server Attributes**. • Organization or user level: as any administrator, select **Manage** > **Organizations** or **Manage** > **Users**, then select the organization or user, click **Edit** in the right-hand panel, and select the **Attributes** tab.		
Attribute	Value	Description
`domainWhitelist` at server level	<blank>	Explicitly set the whitelist to blank (attribute defined with an empty value) if: • you do not have any Visualize.js-enabled web applications, OR • you have Visualize.js-enabled web applications that access your server from the *same* domain as the server
`domainWhitelist` at server level	`example.com` (See below)	If you have Visualize.js-enabled web applications that access your server from a *different* domain, then specify an expression that matches the domain name. For the syntax of this expression, see below.

`domainWhitelist` at server level
`domainWhitelist` at org1 level
`domainWhitelist` at user2 level
	<blank>
	`example1.com`
	`example2.com`
	...
		If your organizations or users have Visualize.js applications on specific domains, you could use the hierarchy of attributes to set the whitelist according to each organization's or each user's

Cross-Domain Whitelist		
...	(See below)	individual domain. In this case, make sure that the whitelist at the server level is defined as blank. For the syntax of this expression, see below.
`domainWhitelist1` `domainWhitelist2`	<regexp> <regexp>	If you want to add more than one regular expression to the whitelist, define these additional attributes at the same level as `domainWhitelist`. If you need further attributes, you can specify them in the `additionalWhitelistAttributes` property of the `crossDomainFilter` bean in the file `.../WEB-INF/applicationContext.xml`.

The actual value of the attribute is a simplified expression that the server converts into the full regular expression. The value must include the protocol (http), any sub-domains that you use, and the port as well. The value can contain `*` and `.` which the server translates into the proper form as `.*` and `\.`. The server also adds `^` and `$` to the ends of the expression. For example, a typical value for this attribute would be:

> `http://*.myexample.com:80\d0` which is translated to
> `^http://.*\.myexample\.com:80\d0$`

This matches the following domains that you might use:

> http://bi3.myexample.com:8080 and http://bi3.myexample.com:8090
>
> http://bi4.myexample.com:8080 and http://bi4.myexample.com:8090

But does not match the following:

> http://myexample.com:8080 or http://bi3.myexample.com:8081

If you wish to write your own complete regular expression, surround it with `^` and `$`, and it will be used as-is by the server.

Remember that if you add Visualize.js applications that run on different domains, or change the domains where they run, then you must update the whitelist attributes accordingly. Visualize.js applications on domains that are not whitelisted do not work.

Do not delete the `domainWhitelist` property from the server level. That removes the whitelist, but on upgrading the server, the attribute is restored with a less secure default value. When the attribute is defined, even with an empty value, it remains during any server upgrade.

Sending REST Requests from a Browser

If you use the REST API to access JasperReports Server from within an application, this does not trigger a CSRF warning because the application is separate from any access through the browser. However, some browser plug-ins can be used to send REST API requests. Using these to send POST, PUT, or DELETE requests trigger a CSRF warning and fail. GET requests from a browser REST client are safe and do not fail the CSRF check.

To allow REST API requests through a browser, configure your browser REST client to include the following header in every request:

```
X-REMOTE-DOMAIN: 1
```

CSRF Browser Compatibility

Only browsers are susceptible to CSRF. Hence, the CSRF protection mechanism detects browsers based on the user-agent string embedded in the request. For performance reasons, the current configuration only filters for Mozilla and Opera user-agents. They cover more than 99% of the browsers in use, such as Chrome, Firefox, Internet Explorer, and Safari.

If your users have browsers with user-agents other than Mozilla, they will not be protected against CSRF by default.

All browsers officially supported by JasperReports Server are protected against CSRF. The following instructions are provided for testing purposes only.

To enable CSRF protection for these browsers, you can add the corresponding user-agent to the CSRF filter:

1. Find the name of the user-agent for the given browser. If you cannot find the user-agent, many are listed on the following website:

   http://www.useragentstring.com/pages/Browserlist/

2. Open the file `.../WEB-INF/applicationContext.xml` for editing.

3. Locate the `csrfGuardFilter` bean and its `protectedUserAgentRegexs` property. Each list value is a regular expression that is matched against every request's user-agent value in its entirety.
4. Add a regular expression to the `protectedUserAgentRegexs` property list that matches the user-agent string from your desired browser.
5. Restart JasperReports Server.

Configuring XSS Protection

Cross-site scripting (XSS) is a security threat where attackers inject malicious data into the server so that the data is run as JavaScript when it is displayed in the UI. The Open Web Application Security Project (OWASP) lists cross-site scripting in Top 10 Most Critical Web Application Security Risks.

As of JasperReports Server 6.1, all output in the UI is escaped so that no malicious scripts can run. For example, if an attacker inserts the `<script ...>` tag into the text of a resource description, the HTML generated by the server contains `&lt;script ...&gt;` that is displayed but will not run as code. If you see `<script ... >` in the data shown in the UI it means someone is trying to inject a cross-site script on the server.

Before output escaping, the security framework implemented an input validation mechanism to block cross-site scripting. Input validation is now deprecated in JasperReports Server and no longer supported.

Like many modern web apps, JasperReports Server consists of interactive pages that use JavaScript to modify and update the page dynamically in the browser. To display this dynamic content, JavaScript inserts HTML snippets or raw data from the server into the page's static HTML. The static page is generated by JavaServer Pages (JSP) and HTML templates, which have mechanisms for output escaping to prevent XSS. JasperReports Server has additional mechanisms to escape the output in the dynamic content. Otherwise, it would be vulnerable to XSS. The dynamic output escaping blocks dangerous tags such as `<script ...>` and it removes dangerous attributes such as `onmouseover`.

The default configuration of JasperReports Server provides output escaping of both static and dynamic content, and thus protects the server from XSS. The output escaping mechanism for static content cannot be configured. However, for advanced uses, the output escaping mechanism for dynamic content can be configured to allow different HTML tags and block new attributes. The output escaping mechanism is implemented in .../scripts/runtime_dependencies/js-sdk/src/commom/util/xssUtil.js. It defines the tags that

are allowed, called the tag whitelist, and HTML attributes that are blocked, called the attribute map. The following configuration properties can supplement or replace these defaults.

The default configuration of the server provides secure XSS protection. Modifying the following configuration is for advanced use cases only and must be done correctly. When configured improperly, these settings may inadvertently break the server UI, or silently disable XSS protection.

Output Escaping	
Configuration File	
.../WEB-INF/classes/esapi/security-config.properties	
Property	Description
`xss.soft.html.escape.tag.whitelist`	The whitelist is the list of HTML tags that will not be escaped when the server renders dynamic content to the UI. This property expands or replaces the default list in xssUtil.js. Specify comma-separated tag names without <> brackets. Use + as the first character to append to the default whitelist. If this property is not specified or the list is empty, the default whitelist applies. For details, refer to Configuring the Tag Whitelist.
`xss.soft.html.escape.attrib.map`	The attribute map determines which HTML attributes create vulnerabilities in dynamic content and how to replace them. This property defines a map of case-insensitive regular expressions (regex syntax) and replacements. When specified, it overrides the default map defined in xssUtil.js. If this property is absent or not set, the default map is used. For details, refer to Configuring the Attribute Map.

These configurations only apply to XSS protection of dynamic content. They do not affect how static pages or static content are escaped when generated by the server.

Configuring the Tag Whitelist

The tag whitelist specifies all HTML tags (elements) that are allowed in dynamic content sent to a user's browser, sometimes called asynchronous data. Tags not in the whitelist are

escaped, meaning their < and > brackets are replaced with < and > so they are displayed as < and > but not interpreted as HTML. The default whitelist is defined in the xssUtil.js file. It allows the tags needed for the UI to be displayed and escapes tags such as `<script ...>` that create XSS vulnerabilities.

The `xss.soft.html.escape.tag.whitelist` property expands or replaces the default whitelist. It contains comma-separated tag names without < > brackets. If this property is not specified or the list is empty, the default whitelist in xssUtil.js applies.

In normal usage, the first character is + so that the specified tags are added to the default whitelist. For example, if you want to add blink and marquee to the list of allowed HTML tags, specify the following value:

```
xss.soft.html.escape.tag.whitelist=+blink,marquee
```

When + is omitted, this list replaces the entire default whitelist. For example, if you wish to block a tag that is specified in the default whitelist, copy all the default tags from xssUtil.js, and delete the ones you wish to block. Be very careful with this usage, because whitelisting the wrong tags can create vulnerabilities. Also, some parts of the UI depend on the default whitelist, and they may appear broken if they are removed from the whitelist.

Never add the `script` tag to the whitelist because it disables output escaping of dynamic content.

Configuring the Attribute Map

Certain HTML attributes create XSS vulnerabilities because they switch to JavaScript context, for example `onmouseover` and the like. The attribute map defines which attributes are dangerous and how to replace them when performing output escaping of dynamic content, also called asynchronous data. It uses a map of case-insensitive regular expressions (regex syntax) and replacements to detect and neutralize such malicious HTML. The default map that is coded in xssUtil.js is equivalent to the following expression:

```
xss.soft.html.escape.attrib.map= {'\\\\bjavascript:': '', '\\\\bon(\\\\w+?)\\\\s*=': 'on$1=',
'\\\\(':'(', \ '\\\\bsrcdoc\\\\s*=': 'srcdoc='}
```

When regex syntax appears in properties files, `\` characters must be escaped. For example, `\s` appears as `\\\\s`.

For advanced use cases, you can modify this property by adding more pairs to the map. Copy the default map above and add the new regex and its safe replacement at the end. For example, to escape the string `data:text/html` by replacing it with nothing, use the following map:

```
xss.soft.html.escape.attrib.map= {'\\\\bjavascript:': '', '\\\\bon(\\\\w+?)\\\\s*=': 'on$1=',
'\\\\(':'(', \ '\\\\bsrcdoc\\\\s*=': 'srcdoc=', '\\\\bdata:\\\\s*text/html\\\\b': ''}
```

Modify this property at your own risk. To work properly, the regex keys in the map must be very specific. Also, the replacement values in the map should never be the same as any regex keys, otherwise multiple replacements will happen, and the output will be corrupted in unpredictable ways.

Never set the map to {} because this will disable HTML attribute escaping in dynamic content.

Protecting Against SQL Injection

SQL injection is an attack that uses malicious SQL queries in reports to gain access or do damage to your databases. By default, JasperReports Server validates query strings to protect against SQL injection.

Whenever the server runs an SQL query, the server validates the query string with the following rules:

- SQL queries must start with SELECT.
- Queries may not contain INTO clauses.
- Queries may call stored procedures (CALL command used by JDBC drivers).
- Multiple queries separated by semi-colons (;) are also prohibited.
- SQL comments are allowed, but will be removed before being transmitted.

If your reports or Domains use such queries, you need to either change your queries or update the security configuration to allow them.

Users who run a report with a query that does not meet the rules see an error. Administrators can monitor the server logs to search for evidence of attempted security breaches.

SQL query validation is enabled by default when installing JasperReports Server. To turn off this protection, edit the following file:

SQL Query Validation		
Configuration File		
.../WEB-INF/classes/esapi/security-config.properties		
Property	Default Value	Description
`security.validation.sql.on`	`true`	Turns SQL query validation on or off in the server. Any other value besides "case-insensitive=false" is equivalent to true.

SQL query validation rules were added to comply with security guidelines for web applications. Turning off query validation or modifying the validation rules may make the server more vulnerable to web attacks.

Customizing the Error Message

When query validation blocks a query that violates a security rule, the server displays an error in the UI. By default, security messages are intentionally generic to avoid alerting potential attackers to security errors.

We highly recommend that external deployments customize the security error message to be unique, yet still generic. You can change both the message and the error number. Choose any combination of numbers or letters so administrators can easily search the logs to detect security violations.

Query Validation Messages
Configuration File
.../WEB-INF/bundles/security.properties

Query Validation Messages	
Property	Default Value
`message.validation.sql`	`An error has occurred. Please contact your system administrator. (6632)`

If you translate your application into other languages, be sure to create a locale-specific copy of this file and translate these messages as well.

Understanding Query Validation

Query validation uses a mechanism to validate every SQL query before running it. The validation process is defined by a validation rule that references a validator expression. The rule and the expression are defined in separate files.

The security.properties and validation.properties files contain many validation rules and expressions. These were used for general input validation in the server, but this mechanism is deprecated and no longer used. Only the expressions for SQL validation are still applicable.

Query Validation Rule	
Configuration File	
.../WEB-INF/classes/esapi/security.properties	
Property	Default Value
`sqlQueryExecutor`	`Alpha,ValidSQL,500000,true,SQL_Query_Executor_context`

The validation rule contains five comma-separated values:

- `Alpha` – Not used for query validation.
- `ValidSQL` – The name of the SQL validator expression in the other file.
- `500000` – The maximum length allowed for the query.
- `true` – Whether the query can be blank.

- `SQL_Query_Executor_context` – Context string for log messages.

SQL Validator Expression	
Configuration File	
.../WEB-INF/classes/esapi/validation.properties	
Property	Default Value
`Validator.ValidSQL`	`(?is)^\\s*(select|call)\\b((?!\\b (into|delete|update|drop)\\b)[^;]);?s$`

Note: The default value for the *Validator.ValidSQL* property is a single-line string:

```
(?is)^\\s*(select|call)\\b((?!\\b(into|delete|update|drop)\\b)[^;]);?s$
```

The validator expression is a regular expression that must match the query string. The default expression enforces the following:

- Queries may only use the SELECT statement, which is read-only. The following write statements are forbidden: `DROP, INSERT, UPDATE, DELETE`
- SELECT statements may not use the INTO clause that could copy data.
- CALL statements for stored procedures are allowed.
- Multiple queries separated by semi-colons (`;`) will be rejected. The following example causes a security error: `SELECT f1,f2 FROM tbl_1; SELECT f3 from tbl_2;`

The rule and validator expression are commented by default because the server implements the same SQL validation with an internal mechanism. If you wish to customize the SQL validation, uncomment the rule and create a validator expression as described below.

Customizing Query Validation

If you wish to use a different validator expression for queries, always create a validator expression with a new name in validation.properties. Then substitute that name in the validation rule in security.properties. For example, if you wish to forbid queries from running stored procedures in your database, add the following validator expression in validation.properties:

```
#Validator.ValidSQL=(?is)^\\s*(select|call)\\b((?!\\binto\\b)[^;])*;?\\s*$
Validator.ValidSQLnoProc=(?is)^\\s*(select)\\b((?!\\binto\\b)[^;])*;?\\s*$
```

Then you would uncomment and modify the validation rule in security.properties as follows:

```
# Main SQL execution point
sqlQueryExecutor=Alpha,ValidSQLnoProc,500000,true,SQL_Query_Executor_context
```

It is also possible to have two or more validation rules that are applied sequentially (logical AND) until one fails. The rules must have the same name but with a numerical suffix, for example:

```
# Main SQL execution point
sqlQueryExecutor=Alpha,ValidSQL,500000,true,SQL_Query_Executor_context
sqlQueryExecutor2=Alpha,ValidSQLCustom,500000,true,SQL_Custom_Executor_context
```

With multiple rules for query validation, each rule is applied in the order listed until one fails. When one rule fails, the whole validation fails.

Performance Issues

By default, the internal SQL validation mechanism accesses the query metadata to allow semicolons (;) in the data part of the query, for example in table names. This access can cause a performance issues with certain JDBC drivers, in which case you can disable it as follows:

Advanced Input Validation		
Configuration File		
.../WEB-INF/classes/esapi/security-config.properties		
Property	Default Value	Description
`validate.sql.via.metadata.query.execution`	true	Set this value to false to

Advanced Input Validation		
		disable semicolon checking in query metadata if SQL validation causes performance issues with your JDBC driver.

Further Security Configuration

The security configuration file contains other default security settings. In particular, they can warn you when a security file has a syntax error and could not be loaded. Changing these defaults is possible but not recommended:

Advanced Input Validation		
Configuration File		
.../WEB-INF/classes/esapi/security-config.properties		
Property	Default Value	Description
`log.msg.security.off`	SECURITY for [%s] is OFF	If security is turned OFF, this message is logged. This message in the logs can alert administrators if the security configuration has been tampered with.
`msg.cannot.load`	Security configuration [%s] cannot be loaded.	If there is an error in the security configuration files, this message is logged. This is a severe error and should be resolved by the administrator.

Protecting Against XML External Entity Attacks

XML files are vulnerable to XML External Entity (XXE) attacks when they include a Document Type Definition (DTD) with a DOCTYPE declaration. Because of this risk, JasperReports Server checks for DOCTYPE declarations. By default, this protection is disabled as the setting causes errors if the XML files are vulnerable to the attack. Consider enabling this setting if XXE attacks are a concern. For more information on this security issue, see Wikipedia's article on XML External Entity Attack.

Before enabling the check, ensure that the XML files in your repository do not include DOCTYPE declarations.

To enable XXE protection:

1. Identify and edit any XML file in the JasperReports Server repository that include a DOCTYPE declaration. Delete the declaration and update the JasperReport on the server. Since JasperReports Server does not support DTDs themselves, we recommend removing them entirely.
2. Use a text editor to open the `.../WEB-INF/applicationContext.xml` file.
3. Locate the `skipXXECheck` property and set it to `false`.
4. Restart JasperReports Server.

Protecting Against Clickjacking Attacks

JasperReports Server implements a mechanism to protect against clickjacking attacks. To enable this mechanism, edit the following configuration file: applicationContext-security-web.xml.

1. Using a text editor, open the applicationContext-security-web.xml file (found in <js-install>\apache-tomcat\webapps\jasperserver-pro\WEB-INF).
2. Locate the `antiClickJackingEnabled` property in the `webAppSecurityFilter` bean, and set it to true. Setting this property to true instructs JasperReports Server to include an X-Frame-Options header in every response.
3. You can also set the `antiClickJackingOption` property to control the header value. Valid values are:

- DENY - JasperReports Server does not load into any iframe.

- SAMEORIGIN - JasperReports Server only loads into an iframe on a page in the same domain as JasperReports Server.
- ALLOW-FROM - JasperReports Server only loads in a frame on a page specified in the `antiClickJackingUri` property.

4. If you set the `antiClickJackingOption` property to ALLOW-FROM, also set the `antiClickJackingUri` property to a valid URI.
5. Save the file and restart the server.

If you use iframes to embed JasperReports Server (including use of Visualize.js), set `antiClickJackingOption` to either:

- SAMEORIGIN (if the embedding host is on the same domain as JasperReports Server) or
- ALLOW-FROM (if the embedding host is on a different domain than JasperReports Server).

If you use ALLOW-FROM, set the `antiClickJackingUri` property too.

Clickjack protection does not support cases in which multiple domains embed JasperReports Server.

Restricting File Uploads

Several dialogs in JasperReports Server prompt the user to upload a file to the server. For performance and security reasons, you may want to restrict file uploads by name and size.

The following setting is the global file upload limit for the entire server. Any single upload that exceeds this limit triggers an error and a stack trace message. It is intended to be an absolute maximum to prevent a worse out-of-memory error that affects the entire server.

Global File Size Upload Limit		
Configuration File		
…/WEB-INF/js.config.properties		
Property	**Value**	**Description**
`file.upload.max.size`	`-1` <default>	Maximum size in bytes allowed for any file

Global File Size Upload Limit		
		upload. The default value, -1, means that there is no limit to the file size, and a large enough file could cause an out-of-memory error in the JVM. Some file uploads such as importing through the UI are necessarily large and must be considered. Set this value larger than your largest expected import and smaller than your available memory.

The following settings apply to most file upload dialogs in the UI, such as uploading a JRXML or a JAR file to create a JasperReport in the repository. These settings in the `fileResourceValidator` bean restrict the file size and the filename pattern.

File Upload Restrictions		
Configuration File		
.../WEB-INF/flows/fileResourceBeans.xml		
Property	**Value**	**Description**
`maxFileSize`	`-1` <default>	The maximum size in bytes allowed for a file uploaded through most UI dialogs. If an upload exceeds this limit, the server displays a helpful error message. The default value, -1, means that there is no limit to the file size, and an upload could reach the global limit if set, or an out-of-memory error. Usually, the files required in resources are smaller, and a limit of 10 MB is reasonable.
`fileNameRegexp`	`^.+$` <default>	A regular expression that matches allowed file names. The default expression matches all filenames of one or more characters. A more restrictive expression such as [a-zA-Z0-9]{1,200}\.[a-zA-Z0-9]{1,10} would limit uploads to alpha-numeric names with an extension.

File Upload Restrictions		
`fileName ValidationMessageKey`	`<null/>` <default>	The name of a Java property key whose value is a custom message to display when the uploaded filename does not match `fileNameRegexp`. For example, you could add the following line to WEB-INF/js.config.properties: `my.filename.validation=The name of the uploaded filename must contain only alphanumeric characters and have a valid extension.`

The following setting restricts the extension of the uploaded file for the sub flows, when adding files to a composite resource like reports, for example, Add Resource > JasperReport. The upload dialog searches for files with the given extensions only.

File Upload Extensions	
Configuration File	
<jasperserver-pro-war>/scripts/resource.locate.js	
Property	**Value**
`ALLOWED_FILE_ RESOURCE_EXTENSIONS`	By default, the following extensions are allowed: `"css", "ttf", "jpg", "jpeg", "gif", "bmp", "png", "jar", "jrxml", "properties", "jrtx", "xml", "agxml", "docx", "doc", "ppt", "pptx", "xls", "xlsx", "ods", "odt", "odp", "pdf", "rtf", "html"` Add or remove extensions to change the file type restrictions.

The following setting restricts the extension of the uploaded file for adding individual files to the repository (for example, Add Resource > File > JRXML). The upload dialog browses only for files with the extensions that are mapped to resource types.

File Upload Extensions	
Configuration File	
<jasperserver-pro-war>/scripts/resource.add.files.js	
Property	**Value**
`typeToExtMap`	This property specifies the mapping of resource types to the file extensions. For example: `'img': ['jpg', 'jpeg', 'gif', 'bmp', 'png']` Add or remove extensions to change the file type restrictions.

Restricting Groovy Access

This section describes functionality that can be restricted by the software license for JasperReports Server. If you do not see some of the options described in this section, your license may prohibit you from using them. To find out what you are licensed to use, or to upgrade your license, contact Jaspersoft.

JasperReports Server relies on Apache Groovy in a number of contexts, including:

- When a Domain definition includes a security file that determines which users or roles have access to various data.
- When a calculated field in an Ad Hoc view or Domain relies on a Groovy expression.

By default, Groovy is given broad access within your application server, which is a good approach to certain design, testing, and evaluation tasks. However, some production systems should be configured to restrict Groovy to more limited access by creating a whitelist that only includes the classes Groovy should access. Once configured, the server returns an error when the Groovy compiler encounters code that does not conform to the whitelist.

Groovy's access is set at the server level; configure it by editing properties files as well as a Groovy source file:

1. Configure the `groovyRunner` to enable the restriction in general.
2. Configure the preprocessor to enable the restriction for Groovy expressions in DomEL.

3. Optionally configure the whitelist to allow Groovy access to additional classes.

First, enable the Groovy restriction:

Groovy Restriction		
Configuration File		
.../WEB-INF/applicationContext-semanticLayer.xml		
Property	Bean	Description
`groovyCustomizerFactory`	`groovyRunner`	Uncomment this property to enable the restriction.

In addition to enabling the Groovy restriction, configure the DomEL preprocessor:

DomEL Restriction		
Configuration File		
.../WEB-INF/applicationContext-datarator.xml		
Attribute	Bean	Description
`preprocessGroovy`	`defaultPreprocessor`	Set this value to true to apply the Groovy restriction to all DomEL expressions that rely on the `groovy()` function.

Optionally, you can extend the whitelist by adding additional classes that you want Groovy to access:

Groovy Whitelist	
Groovy Source File	
.../groovy/com/jaspersoft/commons/groovy/GroovyCustomizerFactoryImpl.groovy	
Class	Description

`GroovyCustomizerFactoryImpl`	List of classes that Groovy can access. Enclose each classname in quotes and delimit each entry with a comma. For example:
	`def receiversWhiteList = [`
	`'java.lang.Byte',`
	`'java.lang.Character',`
	`...`
	`]`
	The last entry should not be followed by a comma.

Which classes you might restrict Groovy from accessing depends largely on your usage patterns, environment, and security concerns. Because of this, we cannot provide specific advice about what you should whitelist. However, we have some general recommendations of classes you would or would not want to whitelist.

For example, Groovy can be used to run commands in the server host's operating system using a string literal such as `rm -rf /".execute()`. Therefore, `java.lang.String` should not be added to the whitelist.

However, some classes, like those in the default list, are considered much safer. For example, the class `org.apache.commons.lang3.StringUtils` consists solely of static utility string methods, so if it is in the whitelist, you can call `StringUtils.isEmpty()` to check for an empty string, instead of calling `isEmpty()` on a string directly.

When you enable and configure the whitelist, be sure to test your JasperReports Server environment thoroughly.

If you have been running your server without this restriction, and then enable and configure it, some functionality may fail. For example, Domains that include a security file may return errors, since they rely on Groovy to evaluate the `principalExpression`. The failure is likely because the Groovy expression calls classes that are not in your whitelist. However, your best course of action is not necessarily to add those classes to the whitelist, as it may be difficult to debug. It is better to create a method in `BaseGroovyScript` and call it from the Domain security expression. For more information, please see our article on the Jaspersoft community site (http://community.jaspersoft.com).

For more information about Groovy, see Apache's Groovy web site.

Enabling JNDI Security

When you have read-write access to huge volumes of data at your disposal, you can retrieve, modify, copy, or move data anytime. This increases the risk of data corruption and reduces data security. It is imperative to restrict access to check the sanity and quality of data. On enabling JNDI security, read-only access is provided to data sources.

You can enable JNDI security for data sources that are already deployed on JasperReports Server. However, you must manually migrate the data sources to a secure setup. You can also enable JNDI security when deploying JasperReports Server on Tomcat or JBoss EAP or Wildfly. For details, see the *JasperReports® Server Installation Guide*.

You can enable JNDI restricted security to JasperReports Serverdata sources with the following setting:

JNDI Restricted Access	
Configuration File	
.../WEB-INF/classes/hibernate.properties	
Property	**Description**
`metadata.hibernate.jndi.restrictedAccess.enabled`	Can be set to: • `true` • `false`

JNDI Restricted Access	
	By default, it is set to `false`, implying that the connection to all JNDI data sources is successful. However, if set to `true`, connection to only the JNDI restricted data sources is successful, and, connection to JNDI non restricted data sources fails.

When installing JasperReports Server manually or when setting `metadata.hibernate.jndi.restrictedAccess.enabled` to `true`, the user must update the data sources using the `jdbc/jasperserver` or the `jdbc/jasperserverAudit` connections to access older resources. The following list offers the sample data sources:

- Audit data source: `jdbc/jasperserverAudit`
- Jasperserver repository SQL data source: `jdbc/jasperserver`
- Jasperserver data source: `jdbc/jasperserver`
- Profile data source JNDI: `jdbc/jasperserver`
- Report monitoring data source: `jdbc/jasperserverAudit`
- Jasperserver SQL data source: `jdbc/jasperserver`

The sample JNDI data sources must be edited to use JNDI restricted data sources.

Current name	New name
jdbc/jasperserver	jdbc/jasperserverSystemAnalytics
jdbc/jasperserverAudit	jdbc/jasperserverAuditAnalytics

Impact on Datasource, Domains, and Reports

If metadata.hibernate.jndi.restrictedAccess.enabled=false

JNDI name	Permission	Create Datasource, Domains, and Reports	View Reports
jasperserver	Read-write	Yes	Yes
jasperserverAudit	Read-write	Yes	Yes
jasperserverSystemAnalytics	Read-only	Yes	Yes
jasperserverAuditAnalytics	Read-only	Yes	Yes

If metadata.hibernate.jndi.restrictedAccess.enabled=true

JNDI name	Permission	Create Datasource, Domains, and Reports	View Reports
jasperserver	Read-write	No	No
jasperserverAudit	Read-write	No	No
jasperserverSystemAnalytics	Read-only	Yes	Yes
jasperserverAuditAnalytics	Read-only	Yes	Yes

Hiding Stack Trace Messages

By default, JasperReports Server displays stack traces in certain error messages. Stack traces reveal some information about the application, and security experts recommend that an application not displays them.

The following setting determines what error messages are displayed:

Hiding Stack Trace Messages
Configuration File

Hiding Stack Trace Messages		
.../WEB-INF/applicationContext-security.xml		
Property	Bean	Description
`outputControlMap`	`exceptionOutputManager`	Set the roles in the list for each of the three levels of error details. Only users who have a given role sees that level of detail. See sample below.
`outputControlMapForContexts`	`exceptionOutputManager`	This property overrides the existing *outputControlMap* property of *exceptionOutputManager* in */jasperserver-pro/rest_ v2/contexts* flow. See sample below.

Error messages contain three parts: an ID, the stack trace, and a message. You can control which of these error message parts are displayed to users based on roles.

For example, for regular users not to see stack traces, but to see error messages, remove `ROLE_USER` from the `ERROR_UID` list and add it into `MESSAGE` list, resulting in the following configuration:

```
<bean name="exceptionOutputManager"
class="com.jaspersoft.jasperserver.api.common.error.handling.ExceptionOu
tputManagerImpl">
            <property name="outputControlMap">
                <map>
                    <entry key="ERROR_UID">
                        <list>
                            <!--<value>ROLE_USER</value>-->
                        </list>
                    </entry>
                    <entry key="STACKTRACE">
                        <list>
                            <value>ROLE_SUPERUSER</value>
                        </list>
                    </entry>
                    <entry key="MESSAGE">
                        <list>
                                                    <value>ROLE_USER</value>
```

```
                            <value>ROLE_SUPERUSER</value>
                        </list>
                    </entry>
                </map>
            </property>
        </bean>
    </beans>
<beans profile="engine">
        <util:map id="outputControlMapForContexts">
            <entry key="ERROR_UID">
                <list>
                    <value>ROLE_USER</value>
                </list>
            </entry>
            <entry key="STACKTRACE">
                 <list>
                 <value>ROLE_SUPERUSER</value>
              </list>
          </entry>
          <entry key="MESSAGE">
              <list>
                  <value>ROLE_SUPERUSER</value>
                  <value>ROLE_ADMINISTRATOR</value>
              </list>
          </entry>
      </util:map>
</beans>
```

Access to the error messages shown in Domain Designer when executing SQL Queries can be configured separately from error messages that user can get in other places. As another example removing `ROLE_ADMINISTRATOR` from `MESSAGE` list and adding into `ERROR_UID` list will hide error messages returned by SQL in Domains, resulting in the following configuration:

```
<beans profile="engine">
        <util:map id="outputControlMapForContexts">
            <entry key="ERROR_UID">
                <list>
                    <value>ROLE_USER</value>
                                        <value>ROLE_ADMINISTRATOR</value>
                </list>
            </entry>
            <entry key="STACKTRACE">
                 <list>
                 <value>ROLE_SUPERUSER</value>
              </list>
```

```
            </entry>
            <entry key="MESSAGE">
                <list>
                    <value>ROLE_SUPERUSER</value>
                    <!--<value>ROLE_ADMINISTRATOR</value>-->
                </list>
            </entry>
        </util:map>
</beans>
```

When configuring error messages, keep in mind the following:

- We recommend the configuration shown above, so that users see a descriptive error message.
- You can turn off any or all error message parts, however, when both `STACKTRACE` and `MESSAGE` are not displayed to a user, a generic message is output instead. The generic message text is defined as follows:

Generic Error Message	
Configuration File	
.../WEB-INF/bundles/jasperserver_messages*.properties	
Property	Value
`generic.error.message`	By default, `There was an error on the server. Try again or contact site administrators.` is displayed. If you modify this message, you must update the translation in all language files of the bundle.

- If you remove both `STACKTRACE` and `MESSAGE` for a given role, we recommend adding back `ERROR_UID` for that role. That way, the user will see the generic message and an ID that can be sent to administrators and correlated with events in the log file.

If you update any of the error message configuration or bundles, restart your application server or redeploy the JasperReports Server web app.

Defining a Cross-Domain Policy for Flash

JasperReports Server can be configured to use Flash for advanced Fusion-based charts such as gauges and maps. For security reasons, a Flash animation playing in a web browser is not allowed to access data that resides outside the exact web domain where the SWF originated.

As a result, even servers in subdomains cannot share data with a server in the parent domain unless they define a cross-domain policy that explicitly allows it. The file crossdomain.xml, located at the root of the server containing the data, determines which domains can access the data without prompting the user to grant access in a security dialog. Therefore, the server containing the data determines which other servers may access the data.

The following crossdomain.xml sample allows access from only the example domain or any of its subdomains. This example says the server with this file trusts only example.com to use its data.

```
<?xml version="1.0" ?>
    <!DOCTYPE cross-domain-policy SYSTEM
        "http://www.macromedia.com/xml/dtds/cross-domain-policy.dtd">

    <cross-domain-policy>
        <allow-access-from domain="example.com" />
        <allow-access-from domain="*.example.com" />
    </cross-domain-policy>
```

Behind a firewall, servers and users often refer to other computers in the same domain without using the domain name. Flash considers this a different domain and blocks access to data unless the computer name is given in the policy.

```
    <cross-domain-policy>
        <allow-access-from domain="myserver.example.com" />
        <allow-access-from domain="myserver" />
    </cross-domain-policy>
```

When using web services, use the `allow-http-request-headers-from` element so that actions encoded in the request header are allowed. The following example allows standard requests and web service requests from any subdomain of example.com.

```
    <cross-domain-policy>
        <site-control permitted-cross-domain-policies="master-only"/>
        <allow-access-from domain="*.example.com"/>
```

```
        <allow-http-request-headers-from domain="*.example.com" headers="*"
            secure="true"/>
    </cross-domain-policy>
```

For a description of all possible properties, see the cross-domain policy file specification.

To define a cross-domain policy for Flash-based reports, create a file such as the one above on the server containing the data being accessed. Be sure to place the crossdomain.xml file at the root of the filespace being served. For example, if you use Apache Tomcat, place your files in the following locations:

File	Location
crossdomain.xml	<website-B-tomcat-dir>/webapps/ROOT/crossdomain.xml
XML data (*.xml)	<website-B-tomcat-dir>/webapps/ROOT/<any-dir>/*.xml
Flash component (*.swf)	<website-A-tomcat-dir>/webapps/<appname>/<any-dir>

For more information o configuring the server to use Flash to render advanced charts, see the JasperReports Server Administrator Guide.

Enabling SSL in Tomcat

Secure Sockets Layer (SSL) is a widely-used protocol for secure network communications. It encrypts network connections at the Transport Layer and is used with HTTPS, the secure version of the HTTP protocol. This section shows how to install SSL on Tomcat 9 and to configure JasperReports Server to use only SSL in Tomcat.

Setting Up an SSL Certificate

To use SSL, you need a valid certificate in the Tomcat keystore. In the Java Virtual Machine (JVM), certificates and private keys are saved in a keystore. This is the repository for your keys and certificates. By default, it is implemented as a password-protected file (public keys and certificates are stored elsewhere).

If you already have a suitable certificate, you can import it into the keystore, using the import switch on the JVM keytool utility. If you do not have a certificate, you can use the keytool utility to generate a self-signed certificate (one signed by your own certificate authority). Self-signed certificates are acceptable in most cases, although certificates issued by certificate authorities are even more secure. And they do not require your users to respond to a security warning every time they login, as self-signed certificates do.

The following command is an example of how to import a certificate. In this case a self-signed certificate imported into a PKCS12 keystore using OpenSSL:

```
openssl pkcs12 \-export \-in mycert.crt \-inkey mykey.key \-out mycert.p12
               \-name tomcat \-CAfile myCA.crt \-caname root \-chain
```

Next in this example, you create key.bin, the keystore file, in the Tomcat home folder. Use one of these commands.

For Windows:

```
%JAVA_HOME%\bin\keytool -genkey -alias tomcat -keyalg RSA -keystore %CATALINA_HOME%\conf\key.bin
```

For Unix:

```
$JAVA_HOME/bin/keytool -genkey -alias tomcat -keyalg RSA -keystore $CATALINA_HOME/conf/key.bin
```

The basic install requires certain data. With the above commands, you are prompted for the data:

- Enter two passwords twice. The default for both is “changeit”. If you use the default, be sure to set better, stronger passwords later.
- Specify information about your organization, including your first and last name, your organization unit, and organization. The normal response for the first and last name is the domain of your server, such as jasperserver.mycompany.com. This identifies the organization that the certificate is issued *to*. For organization unit, enter your department or similar-sized unit; for organization, enter the company or corporation. These identify the organization that the certificate is issued *by*.
- Keytool has numerous switches. For more information about it, see the Java documentation.

Enabling SSL in the Web Server

Once the certificate and key are saved in the Tomcat keystore, you need to configure your secure socket in the $CATALINA_BASE/conf/server.xml file, where $CATALINA_BASE represents the base directory for the Tomcat instance. For your convenience, sample `<Connector>` elements for two common SSL connectors (blocking and non-blocking) are included in the default server.xml file that is installed with Tomcat. They are similar to the code below, with the connector elements commented out, as shown.

```
<!-- Define a SSL HTTP/1.1 Connector on port 8443
     This connector uses the JSSE configuration, when using APR, the
     connector should be using the OpenSSL style configuration
     described in the APR documentation -->
<!--
<Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true"
           maxThreads="150" scheme="https" secure="true"
           clientAuth="false" sslProtocol="TLS" />
-->
```

To implement a connector, you need to remove the comment tags around its code. Then you can customize the specified options as necessary. For detailed information about the common options, consult the Tomcat 9.0 SSL Configuration HOW-TO. For detailed information about all possible options, consult the Server Configuration Reference.

The default protocol is HTTP 1.1. The default port is 8443. The port is the TCP/IP port number on which Tomcat listens for secure connections. You can change it to any port number (such as the default port for HTTPS communications, which is 443). However, note that if you run Tomcat on port numbers lower than 1024, a special setup outside the scope of this document is necessary on many operating systems.

Configuring JasperReports Server to Use Only SSL

At this point, the JasperReports Server web application runs on either protocol (HTTP and HTTPS). You can test the protocols in your web browser.

HTTP: http://localhost:8080/jasperserver[-pro]/

HTTPS: https://localhost:<SSLport>./jasperserver[-pro]/

The next step then is to configure the web application to enforce SSL as the *only* protocol allowed. Otherwise, requests coming through HTTP are still serviced.

Edit the file <js-webapp>/WEB-INF/web.xml. Near the end of the file, make the following changes inside the first `<security-constraint>` tag:

- Comment out the line `<transport-guarantee>NONE</transport-guarantee>`.
- Uncomment the line `<transport-guarantee>CONFIDENTIAL</transport-guarantee>`.

Your final code should be like the following:

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>JasperServerWebApp</web-resource-name>
    <url-pattern>/*</url-pattern>
  </web-resource-collection>
  <user-data-constraint>
    <!-- SSL not enforced -->
    <!-- <transport-guarantee>NONE</transport-guarantee> -->
    <!-- SSL enforced -->
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

The term `CONFIDENTIAL` forces the server to accept only SSL connections through HTTPS. And because of the URL pattern `/*`, all web services must also use HTTPS. If you need to turn off SSL mode, you can set the transport guarantee back to `NONE` or delete the entire `<security-constraint>` tag.

Disabling Unused HTTP Verbs

It is a good idea to disable all unused HTTP verbs so they cannot be used by intruders.

In the default JasperReports Server installation, the following HTTP verbs are not used, but they are allowed. To make it easier to disable the verbs, they are listed in a single block of code in <js-webapp>/WEB-INF/web.xml. As in the code immediately above, the URL pattern `/*` applies the security constraint to all access to the server, including web service requests.

The list is commented out by default because it has not been exhaustively tested with all system configurations and platforms.

After uncommenting the security constraint, your final code should be like the following:

```
<!-- This constraint disables the listed HTTP methods, which are not used by JS -->
<security-constraint>
  <web-resource-collection>
    <web-resource-name>RestrictedMethods</web-resource-name>
```

```
        <url-pattern>/*</url-pattern>
        <http-method>HEAD</http-method>
        <http-method>CONNECT</http-method>
        <http-method>COPY</http-method>
        <http-method>LOCK</http-method>
        <http-method>MKCOL</http-method>
        <http-method>OPTIONS</http-method>
        <http-method>PATCH</http-method>
        <http-method>PROPFIND</http-method>
        <http-method>PROPPATCH</http-method>
        <http-method>SEARCH</http-method>
        <http-method>TRACE</http-method>
        <http-method>UNLOCK</http-method>
    </web-resource-collection>
</security-constraint>
```

Configuring HTTP Header Options

Application servers usually provide mechanisms to secure HTTP headers. For example:

- X-Content-Type-Options
- X-XSS-Protection

For Tomcat, both options are described in Apache's Tomcat documentation.

Setting the Secure Flag on Cookies

JasperReports Server uses cookies in several ways:

- `userTimezone` and `userLocale` to store user settings.
- Other UI settings such as "Recently Viewed Resources" and "Popular Resources" on the home page and data source page history. The cookie names for those resources are homePageRecentlyViewedResourcesExpandableListState, homePagePopularLinksExpandableListState, and DataSourceControllerHistory.

The JSESSIONID cookie is managed by the application server, so its security setting depends on your app server configuration.

Jaspersoft does not set the secure flag on these cookies because we do not want to force you to use secure connections. If you want all cookies to be secure, you must customize the source files that create the cookies. This requires the source code distribution and recompiling and building the server app, as described in the JasperReports Server Source Build Guide.

To customize JasperReports Server so cookies are sent only via secure connections:

1. For the time zone and locale cookies, open the following file to edit:

   jasperserver-war-jar\src\main\java\com\jaspersoft\jasperserver\war\UserPreferencesFilter.java

2. Locate the following code in two locations, one for each cookie, and add the middle line to both:

   ```
   cookie.setMaxAge(cookieAge);
   cookie.setSecure(true); /* requires HTTPS */
   ...
   httpOnlyResponseWrapper.addCookie(cookie);
   ```

   For more information, see the JavaDoc for the setSecure method on the `javax.servlet.http.Cookie` class.

3. For the cookies set in JavaScript (homePageRecentlyViewedResourcesExpandableListState and homePagePopularLinksExpandableListState), edit the following file:

   1. jasperserver-war\src\main\webapp\scripts\home\util\cookie.js

4. Locate the following line:

   ```
   document.cookie = updatedCookie;
   ```

   Modify the line as follows:

   ```
   document.cookie = updatedCookie + ";secure;";
   ```

5. Edit the following file:

   jasperserver-war\src\main\webapp\scripts\runtime_dependencies\jrs-ui\src\utils.common.js

6. Located the following line:

   ```
   return _.template('{{- name}}={{- value}}; expires={{- expires}};
   path=/;')
   ```

   Modify the line as follows:

   ```
   return _.template('{{- name}}={{- value}}; expires={{- expires}};
   path=/;secure;')
   ```

7. To redeploy the JavaScript files, you need to optimize and implement them as described in the section "Customizing JavaScript Files" in the JasperReports Server Ultimate Guide. The optimized scripts are the ones that are served by JasperReports

Server by default.

8. Recompile, rebuild, and redeploy the JasperReports Server application.

   This acts only on the cookies. Providing a secure connection is up to the client application, usually by configuring and establishing an HTTPS connection, as described in Enabling SSL in Tomcat. If no secure connection is established, the cookies with the secure flag will not be sent and user settings will not take effect.

Setting httpOnly for Cookies

The application server that hosts JasperReports Server handles the session cookie. To prevent malicious scripts on a client from accessing the user connection, you should set the application server to use httpOnly cookies. This tells the browser that only the server may access the cookie, not scripts running on the client. This setting safeguards against cross-site scripting (XSS) attacks. Consult the documentation for your application server on how to set httpOnly cookies.

Protection Domain Infrastructure in Tomcat

Legitimate code can be used to introduce harmful measures into the web application. For instance, calls for disk access and calls to `System.Exit` can be hidden in classpaths. An effective measure against such intrusions is to implement a protection domain. In Tomcat, you have to enable the Tomcat Security Manager then edit its parameters according to the requirements of your server environment.

The `ProtectionDomain` class encloses a group of classes whose instances have the same permissions, public keys, and URI. A given class can belong to only one ProtectionDomain. For more information on `ProtectionDomain`, see the Java documentation.

Enabling the JVM Security Manager

The Security Manager restricts permissions at the application server level. By default, no permissions are disallowed at that level, so legitimate permissions must be specifically added. Add permissions for JasperReports Server. Doing so does not interfere with server operations because JasperReports Server security restrictions occur on other levels.

Add the enabling code for the Security Manager in the file <apache-tomcat>/conf/catalina.policy. ProtectionDomains can be enabled, as defined in <js-webapp>/WEB-INF/applicationContext.xml, `reportsProtectionDomainProvider` bean.

To enable the Security Manager and give JasperReports Server full permissions there, add the following code fragment at the end of catalina.policy.

```
// These permissions apply to the JasperReports Server application
grant codeBase "file:${catalina.home}/webapps/jasperserver[-pro]/-" {
      permission java.security.AllPermission;
};
```

```
grant codeBase "file:/groovy/script" {
        permission java.io.FilePermission
"${catalina.home}${file.separator}webapps${file.separator}jasperserver[-pro]${file.separator}WEB-
INF${file.separator}classes${file.separator}-", "read";
```

```
  permission java.io.FilePermission
```

```
 "${catalina.home}${file.separator}webapps${file.separator}jasperserver[-pro]${file.separator}WEB-
INF${file.separator}lib${file.separator}*", "read";
```

```
         permission java.util.PropertyPermission "groovy.use.classvalue", "read";
};
```

After enabling the manager in catalina.policy, you should limit the packages that the JasperReports Library can access. To do so, edit <apache-tomcat>/conf/catalina.policy, locate the `package.access` property, and add the names of the packages that JasperReports Library should be prevented from accessing. We recommend that you block these packages:

- `com.jaspersoft.jasperserver`
- `org.springframework`

After editing, it should be similar to:

```
package.access=sun.,org.apache.catalina.,org.apache.coyote.,org.apache.jasper.,
org.apache.tomcat.,com.jaspersoft.jasperserver.,org.springframework.
```

After enabling the manager, you should add the security parameter to your Tomcat startup command. For example:

<apache-tomcat>`\bin\startup -security`

If you did not add the permissions properly, you receive errors like the following:

```
Feb 9, 2010 12:34:05 PM org.apache.catalina.core.StandardContext listenerStart
SEVERE: Exception sending context initialized event to listener instance of class
org.springframework.web.context.ContextLoaderListener
java.security.AccessControlException: access denied (java.lang.RuntimePermission
accessDeclaredMembers)
  at java.security.AccessControlContext.checkPermission(Unknown Source)
  at java.security.AccessController.checkPermission(Unknown Source)
  at java.lang.SecurityManager.checkPermission(Unknown Source)
  at java.lang.SecurityManager.checkMemberAccess(Unknown Source)
  at java.lang.Class.checkMemberAccess(Unknown Source)
  at java.lang.Class.getDeclaredMethods(Unknown Source)
...
```

Restoring Disallowed Permissions

The file <js-webapp>/WEB-INF/applicationContext.xml defines the permissions allowed for `java.security.Class`. You might have to use the file to add permissions disallowed by enabling the Security Manager. On the application level, only specified permissions are granted now, so any application-level permissions you were using have been disallowed. Write code that restores them.

Refer to this commented sample applicationContext.xml file when you restore necessary permissions.

For instance, to add permission for read/write access to the /temp and JasperReport resources folders, add the `java.io.FilePermission` beans to the `permissions` property of `reportsProtectionDomainProvider`:

```
<bean id="reportsProtectionDomainProvider"
class="com.jaspersoft.jasperserver.api.engine.jasperreports.util.
PermissionsListProtectionDomainProvider">
   <property name="permissions">
          <list>
             <bean class="java.io.FilePermission">
                <constructor-arg value="${java.io.tmpdir}${file.separator}*"/>
                <constructor-arg value="read,write"/>
             </bean>
```

```
             <bean class="java.io.FilePermission">
                <constructor-arg value="${catalina.home}${file.separator}webapps${file.separator}
                jasperserver[-pro]${file.separator}WEB-
INF${file.separator}classes${file.separator}-"/>
                <constructor-arg value="read"/>
             </bean>
```

```
            <bean class="java.io.FilePermission">
                <constructor-arg value="${catalina.home}${file.separator}webapps${file.separator}
                jasperserver[-pro]${file.separator}WEB-
INF${file.separator}lib${file.separator}*"/>
                <constructor-arg value="read"/>
             </bean>
          </list>
   </property>
</bean>
```

Encrypting Passwords in URLs

As of JasperReports Server 7.5, encryption of HTTP parameters is deprecated and this feature may be removed in future versions. Jaspersoft recommends using TLS (Transport Layer Security) in your app server to enable HTTPS when accessing your server.

One advantage of JasperReports Server is the ability to share reports with other users. You can easily share the URL to access a report, even with people who do not have a username. To embed the web app, it is often necessary to include a link to a page without logging in, for example:

http://example.com:8080/jasperserver/flow.html?_flowId=homeFlow&j_username=joeuser&j_password=joeuser

However, you must take special precautions to avoid revealing a password in plain text. The server provides a way to encrypt any password that appears in a URL:

1. Configure login encryption as described in Encrypting User Session Login. Specify static key encryption by setting encryption.dynamic.key to false and configure the keystore as described.
2. Once the server is restarted, log into the server to generate the static key.
3. Open the following URL: http://example.com:8080/jasperserver/encrypt.html.
4. Enter the password that you want to encrypt then click **Encrypt**. The script on this page uses the public key to encrypt the password.
5. Paste the encrypted password into the URL instead of the plain text password (log out of the server to test this):
6. http://example.com:8080/jasperserver/flow.html?_flowId=homeFlow&j_username=joeuser&j_password=<encrypted>
7. Use the URL with the encrypted password to share a report.

For complex web applications generating report URLs on the fly, you can also encrypt the password with JavaScript. Your JavaScript should perform the same operations as the encrypt.js script used by the encrypt.html page at the URL indicated above. Using the encryptData() function in encrypt.js, your JavaScript can generate the encrypted password and use it to create the URL.

Static key encryption is very insecure and is recommended only for intranet server installation where the network traffic is more protected. Anyone who sees the username and encrypted password can use them to log into JasperReports Server. Therefore, we recommend creating user IDs with very specific permissions to control access from URLs.

The only advantage of encrypting passwords in URLs is that passwords cannot be deciphered and used to attack other systems where users might have the same password.

Host Header Injection Protection

An HTTP Host header attack is a type of web vulnerability where an attacker can manipulate the HTTP Host header of a web request to trick a server into responding to a request that was not intended for that server. This can lead to a variety of security issues, including domain hijacking, cache poisoning, and server-side request forgery.

To enable JasperReports Server to filter requests matched by the Host header, edit the following configuration file: *applicationContext-security-web.xml*.

1. Using a text editor, open the applicationContext-security-web.xml file (found in <js-install>\apache-tomcat\webapps\jasperserver-pro\WEB-INF).
2. Locate `<bean id="authenticationAuthorizationFilterChainProxy" class="org.springframework.security.web.FilterChainProxy">`.
3. Add allowedHostnames into the firewall:

```
<property name="firewall">
<bean class="org.springframework.security.web.firewall.StrictHttpFirewall">
<property name="allowUrlEncodedSlash" value="true"/>
<property name="allowSemicolon" value="true"/>
<property name="allowUrlEncodedPercent" value="true"/>
<property name="allowBackSlash" value="true"/>
<property name="allowedHostnames">
<value>#{ T(java.util.function.Predicate).isEqual("allowed.hostname.com").or(T
(java.util.function.Predicate).isEqual("localhost")) }</value>
</property>
</bean>
</property>
```

4. Save the file and restart the server.

In this example, we allow requests coming with HOST header == allowed.hostname.com or == localhost. To add more hosts, you can add chained "or(T(java.util.function.Predicate).isEqual ("YOURHOST"))", if only one host is needed, "or...." should be removed.

User Security

JasperReports Server ensures that users access only the data they're allowed to see. The settings that define organizations, users, roles, and repository resources work together to provide complete access control.

This chapter contains the following sections:

Configuring the User Session Timeout

After a period of inactivity, JasperReports Server displays a pop-up notice that the user's session is about to timeout. This gives the user a chance to continue without being logged out.

User Session Timeout		
Configuration File		
.../WEB-INF/web.xml		
Property	Value	Description
`<session-config> <session-timeout>`	`20` <default>	Set the number of minutes that a user session can remain idle before an automatic log out. Setting to 0 (zero) prevents session timeouts.

Note that the session timeout also applies to how long a session remains in memory after a web service call finishes. If another web service call with the same credentials occurs within the timeout period, the server reuses the same session. If the timeout is too short for this case, you may have performance issues caused by a high load of web service calls.

If the timeout is too long, a session may stay active for a long time (even indefinitely with a timeout of 0). The risk of allowing long sessions is that the in-memory session is not updated with any role changes until the user logs out manually (ending the session) and logs in again (creating a session).

Configuring User Password Options

The user password options determine whether passwords can be remembered by the browser, whether users can change their own passwords, and whether password changes are mandatory or optional.

By default, passwords are stored in an encrypted format in the server's private database. For information about changing the way passwords are encrypted, see Encrypting User Passwords

Configuring Password Memory

As a general security policy, sensitive passwords should not be stored in browsers. Many browsers have a "remember passwords" feature that stores a user's passwords. Most browsers do not protect passwords with a master password by default. JasperReports Server can send the property `autocomplete="off"` to indicate that its users' passwords should not be stored or filled in automatically. This helps to ensure that your users do not store their passwords. Actual behavior depends on the browser settings and how the browser responds to the `autocomplete="off"` property.

Login encryption described in Encrypting User Session Login is not compatible with password memory in the browser. Independent of the auto-complete setting, the JavaScript that implements the login encryption clears the password field before submitting the page. As a result, most browsers will not prompt to remember the password when login encryption is enabled, even if the user has password memory enabled in the browser.

When `autoCompleteLoginForm= true`, as in the default installation, you should ensure that all of your users have a master password in their browser.

Password Memory in the Browser		
Configuration File		
.../WEB-INF/jasperserver-servlet.xml		
Property	Value	Description
`autoCompleteLoginForm`	`true` <default> `false`	When false, the server sets autocomplete="off" on the login page and browsers will not fill in or prompt to save Jaspersoft passwords. When true, the autocomplete property is not sent at all, and browser behavior depends on user settings.

Enabling Password Expiration

If your security policies require users to change their passwords at regular intervals, you can enable password expiration. This way JasperReports Server prompts users to change their passwords at your set interval. Users with expired passwords cannot log in without changing their passwords. This option is disabled by default, meaning passwords do not expire and users are never prompted.

When you enable this option, the server automatically enables the **Change Password** link on the Login page, even if `allowUserPasswordChange` is set to false.

If your users are externally authenticated, for example with LDAP, do not enable this option.

Password Administration Option		
Configuration File		
.../WEB-INF/jasperserver-servlet.xml (controls the Login page) .../WEB-INF/applicationContext-security-web.xml (controls web services)		
Property	Value	Description
`passwordExpirationInDays`	`0` <default> <any other value>	Set the value to any positive, non-zero value to specify the number of days after which a password expires.

Allowing Users to Change their Passwords

With this configuration the Change Password link on the Login page is enabled. By default, this option is turned off, and an administrator must define user passwords initially or reset a forgotten password. Enabling the password expiration option (described in the previous section) automatically enables users to change their passwords.

If your users are externally authenticated, for example with LDAP, do not enable this option.

Password Administration Option		
Configuration File		
.../WEB-INF/jasperserver-servlet.xml		
Property	Value	Description
`allowUserPasswordChange`	`false` <default> `true`	Set the value to `true` to enable the **Change Password** link. Any other value disables it.

Enforcing Password Patterns

If you allow or force users to change their passwords, you can enforce patterns for valid strong passwords, by requiring a minimum length and a mix of uppercase, lowercase, and numbers. The default pattern accepts any password of any length, including an empty password.

If your users are externally authenticated, for example with LDAP, do not enable this option.

Password Administration Option		
Configuration File		
.../WEB-INF/applicationContext.xml		
Property	Bean	Description
`allowedPasswordPattern`	`userAuthority Service`	A regular expression that matches valid passwords. The default pattern `^.*$` matches any password. Change the regular expression to enforce patterns such as: • Minimum and maximum password length • Both uppercase and lowercase characters • At least one number or special character Be sure that your pattern allows whitespace and international characters if needed by your users.

When you enforce a password pattern, you should set the following message to inform users why their password was rejected. Be sure to set the message in all your locales.

Password Administration Option	
Configuration File	
.../WEB-INF/bundles/jsexceptions_messages[_locale].properties	
Property	Description
`exception.remote.weak.password`	A message is displayed to users when password pattern matching fails.

Limiting Failed Login Attempts

To prevent brute-force attacks against user and administrator accounts, JasperReports Server locks an account after a configurable number of failed login attempts. JasperReports Server records failed login attempts and administrators may configure a set limit after which the account is disabled. An administrator must enable the account before it can be used again. The setting (enabled by default) is set for 10 attempts, but can be disabled, or configured to reduce or increase the number of attempts. Refer to the *JasperReports Server Administrator Guide* for more information on how to enable a locked user account.

The following bean definition is available in the *applicationContext-security.xml* for the class *LoginLockoutConfig*.

<bean id="loginLockoutConfig"
class="com.jaspersoft.jasperserver.api.common.configuration.LoginLockoutConfig">

<property name="allowedNumberOfLoginAttempts" value="10"></property>

</bean>

The property *allowedNumberOfLoginAttempts* is used to configure the value. The default value is set as *10*.

To disable the feature, set the value to 0. Afterwards, the user has no limit to the number of times they can attempt to log into an account.

JIUser and JExternalUserLoginEvents Tables

A new column has been added to the *JIUser* table named *numberOfFailedLoginAttempts*.

A new table *JIExternalUserLoginEvents* is added. This table is used to track valid or invalid login attempts by external and non-existing users. The property *userLoginAttemptsThreshold* is responsible for the maximum number of rows/records allowed in the table; exceeding this number removes the oldest records from that table to keep the records count under the threshold.

The *JIUser* and *JIExternalUserLoginEvents* tables are used to disable a user if the Number of Failed Attempts to login is reached. If the feature is enabled, then when a user enters an invalid password, the user sees a message that states the number of login attempts still available. If the user enters a valid password, before the counter reaches 0, then the counter gets reset back to 10. If the user exceeds the 10 counter (or whatever the administrator has set this value to (10 is the default), then the user is locked out of their account and will need to have their administrator to unlock/enable the account.

You can leave the default cleanup of 1 hour or you can change it.

The following steps, as an example, describe how to change the threshold to 2 and cron time to 20 minutes:

1. Edit the applicationContext.xml file.
2. Search for '*<bean id="externalUserLoginAttemptsCleanUpService">*'
3. CHANGE "*<property name="userLoginAttemptsThreshold" value="100"/>*" to "*<property name="userLoginAttemptsThreshold" value="2"/>*".
4. Search for "<task:scheduled-tasks scheduler="externalUserCleanupScheduler">".
5. Change "*<task:scheduled ref="externalUserLoginAttemptsCleanUpService" method="clearAllData" cron="0 0 /1 * * *" />*" to "*<task:scheduled ref="externalUserLoginAttemptsCleanUpService" method="clearAllData" cron="0 */20 * * *" />*".

   Note: The cron job runs every 20 minutes. Adjust this value if needed.
6. Restart Tomcat.

Limitations

LDAP users can log into JasperReports Server with or without specifying an organization_id, for example, "*user_1|organization/password*" and "*user_1/password*". Internally, such users are treated as the same user, which means that locking out "*user_1|organization/password*" will also lock out "*user_1/password*". This can be a problem when

you have some internally defined users with the same name in the root level organization. To avoid such situations, stick to the following rules:

- If possible, avoid creating users in the root organization with a user id that can match an external user id.
- External LDAP users should enter their organization ids along with user id and password.

Encrypting User Passwords

> **Warning:**
> As of JasperReports Server 7.5, all encryption in the server relies on cryptographic keys stored in the server's keystore. For more information, see Key and Keystore Management.
>
> The configuration files and properties described in this section are no longer used by this feature. They are documented here only for legacy purposes.

User passwords are stored along with user profiles in JasperReports Server's private database. Password encryption is enabled and passwords are stored as cipher text in the database by default. The following procedure enables system administrators to turn user password encryption on or off. They can also change the encryption algorithm and specify the salt key used to initiate the encryption algorithm.

To Configure User Password Encryption:

1. As a precaution, back up the server's private `jasperserver` database. To back up the default PostgreSQL database, go to the <js-install> directory and run the following command:

   ```
   pg_dump -U postgres jasperserver > js-backup.sql
   ```

   To back up DB2, Oracle, Microsoft SQL Server, and MySQL databases, refer to your database product documentation.
2. Stop your application server. Leave your database running.
3. Export the entire contents of the repository, which includes user profiles and their passwords, with the following commands. Note that there are two dashes (--) in front of the command options:

Windows:
```
cd <js-install>\buildomatic
js-export.bat --everything --output-dir js-backup-catalog
```
Linux:
```
cd <js-install>/buildomatic
js-export.sh --everything --output-dir js-backup-catalog
```

In the export operation, passwords are decrypted using the existing user password ciphers and re-encrypted with the import-export encryption key. This is a separate encryption that ensures that passwords are never in plain text, even when exported. For more information, see "Import and Export" in the *JasperReports Server Administrator Guide*.

4. Edit the properties in the following table to configure different ciphers. Both the server and the import-export scripts access the user profiles and must be configured identically. Make the same changes in both files:

User Password Encryption Configuration

DEPRECATED User Password Encryption Configuration		
<jasperserver-pro-war>/WEB-INF/applicationContext-security.xml <js-install>/buildomatic/conf_source/iePro/applicationContext-security.xml		
Property	Bean	Description
`allowEncoding`	`passwordEncoder`	With the default setting of `true`, user passwords are encrypted when stored. When `false`, user passwords are stored in clear text in JasperReports Server's private database. We do not recommend changing this setting.
`keyInPlainText`	`passwordEncoder`	When `true`, the `secretKey` value is given as a plain text string. When `false`, the secretKey value is a numeric representation that can be parsed by Java's Integer.decode() method. By default, this setting is false, and the `secretKey` is in hexadecimal notation (0xAB).

secretKey	passwordEncoder	This value is the salt used by the encryption algorithm to make encrypted values unique. This value can be a text string or a numeric representation depending on the value of keyInPlainText.
secretKeyAlgorithm	passwordEncoder	The name of the algorithm used to process the key, by default DESede.
cipher Transformation	passwordEncoder	The name of the cipher transformation used to encrypt passwords, by default DESede/CBC/ PKCS5Padding.

Warning:
Change the secretKey value so it is different from the default.

The secretKey, secretKeyAlgorithm, and cipherTransformation properties must be consistent. For example, the secretKey must be 24 bytes long in hexadecimal notation or 24 characters in plain text for the default cipher (DESede/CBC/PKCS5Padding). Different algorithms expect different key lengths. For more information, see Java's javax.crypto documentation.

5. Next, drop your existing jasperserver database, where the passwords had the old encoding, and recreate an empty jasperserver database. Follow the instructions for your database server:
   - Dropping and Recreating the Database in PostgreSQL
   - Dropping and Recreating the Database in MySQL
   - Dropping and Recreating the Database in Oracle
   - Dropping and Recreating in the Database in Microsoft SQL Server
6. Import your exported repository contents with the following commands. The import operation restores the contents of JasperReports Server's private database, including user profiles. As the user profiles are imported, the passwords are encrypted using the new cipher settings.

   Note that there are two dashes (--) in front of the command options:

Windows:
```
cd <js-install>\buildomatic
js-import.bat --input-dir js-backup-catalog
```
Linux:
```
cd <js-install>/buildomatic
js-import.sh --input-dir js-backup-catalog
```

During the import operation, passwords are decrypted with the import-export encryption key and then re-encrypted in the database with the new user password encryption settings. For more information, see “Setting the Import-Export Encryption Key” in the *JasperReports Server Administrator Guide*.

7. Use a database like the SQuirreL tool to check the contents of the JIUser table in the jasperserver database and verify that the password column values are encrypted.
8. Restart your application server. Your database should already be running.
9. Log into JasperReports Server to verify that encryption is working properly during the log in process.

Dropping and Recreating the Database in PostgreSQL

1. Change the directory to `<js-install>/buildomatic/install_resources/sql/postgresql`.
2. Start psql using an administrator account such as PostgreSQL:`psql -U postgres`
3. Drop the `jasperserver` database, create a one, and load the `jasperserver` schema:

```
drop database jasperserver;
create database jasperserver encoding='utf8';
\c jasperserver
\i js-pro-create.ddl
\i quartz.ddl
```

Dropping and Recreating the Database in MySQL

1. Change the directory to `<js-install>/buildomatic/install_resources/sql/mysql`.
2. Log in to your MySQL client:`mysql -u root -p`
3. Drop the jasperserver database, create a one, and load the jasperserver schema:

```
mysql>drop database jasperserver;
mysql>create database jasperserver character set utf8;
mysql>use jasperserver;
mysql>source js-pro-create.ddl;
mysql>source quartz.ddl;
```

Dropping and Recreating the Database in Oracle

1. Change the directory to `<js-install>/buildomatic/install_resources/sql/oracle`.
2. Log in to your SQLPlus client, for example: `sqlplus sys/sys as sysdba`
3. Drop the `jasperserver` database, create a one, and load the `jasperserver` schema:

```
SQL> drop user jasperserver cascade;
SQL> create user jasperserver identified by password;
SQL> connect jasperserver/password
SQL> @js-pro-create.ddl
SQL> @quartz.ddl
```

Dropping and Recreating in the Database in Microsoft SQL Server

1. Change the directory to `<js-install>/buildomatic/install_resources/sql/sqlserver`.
2. Drop the `jasperserver` database, create a one, and load the `jasperserver` schema using the SQLCMD utility:

```
cd <js-install>\buildomatic\install_resources\sql\sqlserver
sqlcmd -S ServerName -Usa -Psa
1> DROP DATABASE [jasperserver]
2> GO
1> CREATE DATABASE [jasperserver]
2> GO
1> USE [jasperserver]
2> GO
```

```
1> :r js-pro-create.ddl
2> GO
1> :r quartz.ddl
2> GO
```

Encrypting User Session Login

As of JasperReports Server 7.5, encryption of HTTP parameters is deprecated and this feature may be removed in future versions. Jaspersoft recommends using TLS (Transport Layer Security) in your app server to enable HTTPS when accessing your server.

By default, JasperReports Server does *not* enable the Secure Socket Layer/Transport Layer Security (SSL/TLS) to encrypt all data between the browser and the server, also known as HTTPS. Enabling HTTPS requires a certificate and a careful configuration of your servers. We recommend implementing HTTPS but recognize that it is not always feasible. See Enabling SSL in Tomcat

Without HTTPS, all data sent by the user, including passwords, appear unencrypted in the network traffic. Because passwords should never be visible, JasperReports Server provides an independent method for encrypting the password values without using HTTPS. Passwords are encrypted in the following cases:

- Passwords sent from the login page.
- Passwords sent from the change password dialog. See Configuring User Password Options.
- Passwords sent from the user management pages by an administrator.

When a browser requests one of these pages, the server generates a private-public key pair and sends the public key along with the page. A JavaScript in the requested page encrypts the password when the user posts it to the server. Meanwhile, the server saves its private key and uses it to decrypt the password when it arrives. After decrypting the password, the server continues with the usual authentication methods.

Login encryption is not compatible with password memory in the browser. Independent of the autocomplete setting described in Configuring Password Memory, the JavaScript that implements login encryption clears the password field before submitting the page. As a result, most browsers will never prompt to remember the encrypted password.

The disadvantage of login encryption is the added processing and the added complexity of web services login. For backward compatibility, login encryption is disabled by default. To

enable login encryption, set the following properties. After making any changes, redeploy the JasperReports Server webapp or restart the application server.

When login encryption is enabled, web services and URL parameters must also send encrypted passwords. Your applications must first obtain the key from the server and then encrypt the password before sending it. See the JasperReports Server Web Services Guide.

Login Encryption		
Configuration File		
.../WEB-INF/classes/esapi/security-config.properties		
Property	Value	Description
`encryption.on`	`true` `false` <default>	Turns login encryption on or off. Encryption is off by default. Any other value besides case-insensitive “false” is equivalent to true.
`encryption.type`	`RSA` <default>	Encryption algorithm; currently, only RSA is supported.
`encryption.key.length`	integer power of 2 `1024` <default>	The length of the generated encryption keys. This affects the strength of encryption and the length of the encrypted string.
`encryption.dynamic.key`	`true` <default> `false`	When true, a key will be generated per every single request. When false, the key will be generated once per application installation. See descriptions in Dynamic Key Encryption and Static Key Encryption below.

Encryption has two modes, dynamic and static, as determined by the `encryption.dynamic.key` parameter. These modes provide different levels of security and are further described in the following sections.

Dynamic Key Encryption

The advantage of encrypting the password at login is to prevent it from being seen, but also to prevent it from being used. For password encryption to achieve this, the password must be encrypted differently every time it's sent. With dynamic key encryption, the server uses a new public-private key pair with every login request.

Every time someone logs in, the server generates a new key pair and sends the new public key to the JavaScript on the page that sends the password. This ensures that the encrypted password is different every time it's sent, and a potential attacker won't be able to steal the encrypted password to log in or send a different request.

Because it's more secure, dynamic key encryption is the default setting when encryption is enabled. The disadvantage is that it slows down each login, though users may not always notice. Another effect of dynamic key encryption is that it doesn't allow remembering passwords in the browser. While this may seem inconvenient, it's more secure to not store passwords in the browser. See Configuring Password Memory.

Static Key Encryption

As of JasperReports Server 7.5, all encryption in the server relies on cryptoghaphic keys stored in the server's keystore. For more information, see Key and Keystore Management.

The configuration files and properties described in this section are no longer used by this feature. They are documented here only for legacy purposes.

JasperReports Server also supports static key encryption. For every login, the server expects the client to encode parameters such as passwords with the httpParameterEncSecret key in the keystore. Because the key is always the same, the encrypted value of a user’s password is always the same. This means an attacker could steal the encrypted password and use it to access the server.

Static key encryption is very insecure and is recommended only for intranet server installation where the network traffic is more protected. The only advantage of static encryption over no encryption at all is that passwords can't be deciphered and used to attack other systems where users might have the same password.

Before setting `encryption.dynamic.key=false` to use static encryption, you must also configure the secure file called keystore where the key pair is kept. Be sure to customize the keystore parameters listed in the following table to make your keystore file unique and secure.

For security reasons, always change the default keystore passwords immediately after installing the server.

DEPRECATED Keystore Configuration (when encryption.dynamic.key=false)		
Configuration File		
.../WEB-INF/classes/esapi/security-config.properties		
Property	Value	Description
`keystore.location`	`keystore.jks` <default>	Path and filename of the keystore file. This parameter is either an absolute path or a file in the webapp classpath, for example <tomcat>/webapps/jasperserver-pro/WEB-INF/classes>. By default, the keystore.jks file is shipped with the server and doesn't contains any keys.
`keystore.password`	`jasper123` <default>	Password for the whole keystore file. This password is used to verify keystore's integrity.
`keystore.key.alias`	`jasper` <default>	Name by which the single key is retrieved from keystore. If a new alias is specified and does not correspond to an existing key, a new key will be generated and inserted into the keystore.
`keystore.key.password`	`jasper321` <default>	Password for the key whose alias is specified by keystore.key.alias.

When you change the key alias, the old key will not be deleted. You can use it again by resetting the key alias. Also, once the key has been created with a password, you can't change the password through the keystore configuration. To delete keys or change a keystore password, the server administrator must use the Java `keytool` utility in the bin directory of the JDK. If you change the keystore password or the key password, the keystore configuration above must reflect the new values or login will fail for all users.

Jaspersoft Documentation and Support Services

For information about this product, you can read the documentation, contact Support, and join Jaspersoft Community.

How to Access Jaspersoft Documentation

Documentation for Jaspersoft products is available on the Product Documentation website, mainly in HTML and PDF formats.

The Product Documentation website is updated frequently and is more current than any other documentation included with the product.

Product-Specific Documentation

The documentation for this product is available on the JasperReports® Server Product Documentation page.

How to Access Related Third-Party Documentation

When working with JasperReports® Server, you may find it useful to read the documentation of the following third-party products:

How to Contact Support for Jaspersoft Products

You can contact the Support team in the following ways:

- To access the Support Knowledge Base and getting personalized content about products you are interested in, visit our product Support website.
- To create a Support case, you must have a valid maintenance or support contract with a Cloud Software Group entity. You also need a username and password to log in to the product Support website. If you do not have a username, you can request one by clicking **Register** on the website.

How to Join Jaspersoft Community

Jaspersoft Community is the official channel for Jaspersoft customers, partners, and employee subject matter experts to share and access their collective experience. Jaspersoft Community offers access to Q&A forums, product wikis, and best practices. It also offers access to extensions, adapters, solution accelerators, and tools that extend and enable customers to gain full value from Jaspersoft products. In addition, users can submit and vote on feature requests from within the Jaspersoft Ideas Portal. For a free registration, go to Jaspersoft Community.

Legal and Third-Party Notices

SOME CLOUD SOFTWARE GROUP, INC. (“CLOUD SG”) SOFTWARE AND CLOUD SERVICES EMBED, BUNDLE, OR OTHERWISE INCLUDE OTHER SOFTWARE, INCLUDING OTHER CLOUD SG SOFTWARE (COLLECTIVELY, “INCLUDED SOFTWARE”). USE OF INCLUDED SOFTWARE IS SOLELY TO ENABLE THE FUNCTIONALITY (OR PROVIDE LIMITED ADD-ON FUNCTIONALITY) OF THE LICENSED CLOUD SG SOFTWARE AND/OR CLOUD SERVICES. THE INCLUDED SOFTWARE IS NOT LICENSED TO BE USED OR ACCESSED BY ANY OTHER CLOUD SG SOFTWARE AND/OR CLOUD SERVICES OR FOR ANY OTHER PURPOSE.

USE OF CLOUD SG SOFTWARE AND CLOUD SERVICES IS SUBJECT TO THE TERMS AND CONDITIONS OF AN AGREEMENT FOUND IN EITHER A SEPARATELY EXECUTED AGREEMENT, OR, IF THERE IS NO SUCH SEPARATE AGREEMENT, THE CLICKWRAP END USER AGREEMENT WHICH IS DISPLAYED WHEN ACCESSING, DOWNLOADING, OR INSTALLING THE SOFTWARE OR CLOUD SERVICES (AND WHICH IS DUPLICATED IN THE LICENSE FILE) OR IF THERE IS NO SUCH LICENSE AGREEMENT OR CLICKWRAP END USER AGREEMENT, THE LICENSE(S) LOCATED IN THE “LICENSE” FILE(S) OF THE SOFTWARE. USE OF THIS DOCUMENT IS SUBJECT TO THOSE SAME TERMS AND CONDITIONS, AND YOUR USE HEREOF SHALL CONSTITUTE ACCEPTANCE OF AND AN AGREEMENT TO BE BOUND BY THE SAME.

Jaspersoft, JasperReports, Visualize.js, and TIBCO are either registered trademarks or trademarks of Cloud Software Group, Inc. in the United States and/or other countries.

All other product and company names and marks mentioned in this document are the property of their respective owners and are mentioned for identification purposes only. You acknowledge that all rights to these third party marks are the exclusive property of their respective owners. Please refer to Cloud SG’s Third Party Trademark Notices (https://www.cloud.com/legal) for more information.

This document includes fonts that are licensed under the SIL Open Font License, Version 1.1, which is available at: https://scripts.sil.org/OFL

Copyright (c) Paul D. Hunt, with Reserved Font Name Source Sans Pro and Source Code Pro.

Cloud SG software may be available on multiple operating systems. However, not all operating system platforms for a specific software version are released at the same time. See the “readme” file for the availability of a specific version of Cloud SG software on a specific operating system platform.