Jenkins 7

Jenkins Security

Jenkins security encompasses various measures and features aimed at safeguarding the Jenkins instance, its resources, and the data processed within it.

Here's an overview of key aspects of Jenkins security:

1. **Authentication**:
    - Jenkins supports various authentication mechanisms, including:
        - **Jenkins' Own User Database**: Users are managed within Jenkins, with passwords stored securely.
        - **LDAP (Lightweight Directory Access Protocol)**: Integration with external LDAP servers for centralized user authentication.
        - **SSO (Single Sign-On)**: Integration with identity providers supporting SAML or OAuth for seamless authentication.
    - Strong authentication helps ensure that only authorized users can access Jenkins and its resources.
2. **Authorization**:
    - Jenkins provides role-based access control (RBAC), allowing administrators to define granular permissions based on roles.
    - Users can be assigned roles with specific permissions to perform actions such as configuring jobs, running builds, or managing Jenkins settings.
    - Fine-grained access control ensures that users have access only to the resources and operations they need to perform their tasks.
3. **Project-Based Matrix Authorization Strategy**:
    - Jenkins allows administrators to define permissions at the project level using a matrix-based authorization strategy.

- This strategy enables specifying which users or groups have specific permissions (e.g., build, configure, delete) for each project or job.

4. **Security Realms**:
   - Jenkins supports multiple security realms for authentication, including Jenkins' own user database, LDAP, Unix user/group database, and external SSO providers.
   - Administrators can configure the preferred security realm based on their organization's requirements and infrastructure.
5. **Securing Jenkins Configuration**:
   - Jenkins administrators should ensure that the Jenkins configuration, including security settings, is appropriately secured.
   - This includes enforcing HTTPS for secure communication, restricting access to Jenkins' administrative interfaces, and securing sensitive configuration files.
6. **Plugin Security**:
   - Administrators should regularly update Jenkins and its plugins to mitigate security vulnerabilities.
   - Jenkins provides security advisories and notifications to alert administrators about security updates and patches for plugins.
7. **Monitoring and Auditing**:
   - Jenkins administrators should monitor system logs, access logs, and audit trails to detect and investigate security incidents.
   - Regular audits help ensure compliance with security policies and identify potential security weaknesses or unauthorized activities.
8. **Securing Build Environments**:
   - Jenkins administrators should implement security best practices for build agents and environments, including regularly updating software, restricting network access, and isolating sensitive data.

Install Security-Related Plugins

Install and configure plugins that enhance Jenkins security, such as:

- **Role-based Authorization Strategy:** Provides more advanced role-based access control capabilities.
- **Credentials Plugin:** Securely manage credentials used by Jenkins jobs and plugins.
- **LDAP Plugin:** Integrate Jenkins with LDAP for user authentication and authorization.

Configure Global Security

- Navigate to "Manage Jenkins" > "Configure Global Security".
- Check the "Enable security" checkbox to enable security in Jenkins.

Choose Authorization Strategy:

Under "Access Control", choose the desired authorization strategy:

- **Matrix-based security:** Allows configuring permissions for individual users and groups using a matrix grid.
- **Project-based Matrix Authorization Strategy:** Allows configuring permissions at the project level using a matrix grid.
- **Role-based Authorization Strategy:** Allows defining roles with specific permissions and assigning users or groups to those roles.

Matrix-based Security:

- If you choose Matrix-based security, you'll see a grid where you can specify permissions for each user or group.
- Check the checkboxes corresponding to the permissions you want to grant.

Project-based Matrix Authorization Strategy:

- If you choose Project-based Matrix Authorization Strategy, you can configure permissions at the project level.
- For each project, specify the users or groups and their corresponding permissions.

Role-based Authorization Strategy:

If you choose Role-based Authorization Strategy, you'll need to define roles and permissions:

- Click on "Add Role" to create a new role.
- Specify a name for the role and define the permissions associated with it.
- Click on "Add Permission" to add permissions to the role.
- Assign users or groups to the role by entering their usernames or group names.

Save Configuration:

Once you've configured the authorization strategy and defined permissions, click on "Save" or "Apply" to save the changes.

Test Authorization:

- Log in to Jenkins with different user accounts to test the authorization settings.
- Verify that users can access only the resources and perform only the actions allowed by their permissions.

Jenkins Shared Library

Jenkins Shared Library is a powerful feature that enables you to define reusable code and functionality that can be shared across multiple Jenkins Pipeline jobs.

It plays a crucial role in enhancing the efficiency, maintainability, and scalability of your Jenkins pipelines.

Here are some key points highlighting the importance of Jenkins Shared Library:

1. **Code Reusability**: Shared libraries allow you to encapsulate common tasks, functions, and logic into reusable components. This promotes code reuse across different pipelines, reducing duplication and ensuring consistency in your CI/CD workflows.
2. **Standardization**: Shared libraries enable you to establish coding standards, best practices, and conventions that can be applied uniformly across all pipelines within your organization. This ensures consistency in the way pipelines are developed and maintained.
3. **Centralized Management**: By centralizing shared code in a dedicated library, you can manage and maintain it more effectively. Updates, bug fixes, and enhancements can be applied to the shared library, and all pipelines referencing it will automatically benefit from these changes.
4. **Improved Maintainability**: Shared libraries make it easier to maintain and evolve your Jenkins pipelines over time. Instead of modifying the same code in multiple pipeline scripts, you can make changes in one central location within the shared library, reducing the risk of errors and ensuring that all pipelines are updated simultaneously.
5. **Encapsulation of Complexity**: Complex or repetitive tasks can be abstracted into reusable functions and methods within the shared library. This simplifies pipeline scripts, making them more readable and easier to understand, while also promoting modular design principles.
6. **Facilitates Collaboration**: Shared libraries facilitate collaboration among teams by providing a common foundation for building pipelines. Teams can share common functionalities,

workflows, and integrations, fostering collaboration and knowledge sharing across the organization.

Jenkins shared library can be created and used from the Jenkins master server or from a SCM like Git and configure Jenkins to retrieve the shared library from the SCM repository.

On the Jenkins Master Server:

- Navigate to the directory where Jenkins is installed.
- Inside the Jenkins home directory (often located at **/var/lib/jenkins** on Linux systems), you'll find a directory named **vars** where shared library files are stored.
- Create a subdirectory within **vars** for your shared library, and then place your Groovy files inside it
- Example path on Linux: **/var/lib/jenkins/vars/my_shared_library**

In the SCM Repository:

- The directory path in the SCM repository where the shared library files are stored depends on the structure of your SCM repository.
- If you're using Git, for example, the shared library files might be stored in a subdirectory within the repository.
- Example path in Git repository: **repo_name/vars**
- In this example, **repo_name** is the name of the repository, and **vars** is the directory within the repository where the shared library files are stored.
- Example of shared library path in Git repository: **repo_name/vars/my_shared_library**

If the shared library is located in a Git repository, then it need to be explicitly configured in Jenkins

Configure Jenkins to Use the Shared Library in a Git repository

- In the Jenkins dashboard, go to **"Manage Jenkins"** > "**Configure System**".
- Scroll down to the **"Global Pipeline Libraries"** section.
- Add a new library configuration:
    - Name: **my-shared-library**
    - Default version: **master** (or specify a specific branch/tag)
    - Retrieval method: **Modern SCM** (select Git and provide the repository URL)
- Save the configuration.

NB: No explicit configurations are needed for a shared library located in the Jenkins master server.

Implementing Jenkins shared library using the previously used declarative pipeline codes.

/var/lib/Jenkins/vars/

```
└── my_shared_library /
    ├── getCode.groovy
    ├── testAndBuild.groovy
    ├── codeQuality.groovy
    └── uploadToNexus.groovy
```

Define functions for each stage in the respective Groovy files.

// File: my-shared-library/getCode.groovy

```
def call() {
    stage('Get Code') {
        steps {
            sh "echo 'cloning the latest application version' "
            git branch: 'feature', credentialsId: 'gitHubCredentials',
url: https://github.com/fewaitconsulting/maven-web-app'
        }
    }
}

// File: my-shared-library/testAndBuild.groovy

def call() {
    stage('Test and Build') {
        steps {
            sh "echo 'running JUnit-test-cases' "
            sh "echo 'testing must passed to create artifacts ' "
            sh "mvn clean package"
        }
    }
}
```

```
// File: my-shared-library/vars/codeQuality.groovy


def call() {
    stage('Code Quality') {
        steps {
            sh "echo 'Performing Code Quality Analysis' "
            sh "mvn sonar:sonar"
        }
    }
}
// File: my-shared-library/vars/uploadToNexus.groovy


def call() {
    stage('Upload to Nexus') {
        steps {
            sh "mvn deploy"
        }
    }
}
// File: my-shared-library/vars/deployToTomcat.groovy


def call() {
    stage('Deploy to Tomcat') {
        steps {
            deploy adapters: [tomcat8(credentialsId: 'tomcat-credentials', path: '', url:
'http://35.170.249.131:8080/')], contextPath: null, war: 'target/*.war'
        }
    }
}
```

Using the Shared Library in a Jenkinsfile:

In your Jenkins Pipeline script (Jenkinsfile), import and use the shared library functions.

```
@Library('my-shared-library') // Import the shared library
pipeline {
    agent any
    tools {
        maven "maven3.6.0"
    }
    stages {
        getCode()
        testAndBuild()
        codeQuality()
        uploadToNexus()
        deployToTomcat()
    }
    post {
        always {
            emailext body: '''Hey guys
Please check build status.

Thanks
Fewa
+237650661631''', recipientProviders: [buildUser(), developers()], subject: 'success', to: 'paypal-team@gmail.com'
        }
        success {
            emailext body: '''Hey guys
Good job build and deployment is successful.

Thanks
Fewa
+237650661631''', recipientProviders: [buildUser(), developers()], subject: 'success', to: 'paypal-team@gmail.com'

        }

        failure {

            emailext body: '''Hey guys

Build failed. Please resolve issues.


Thanks

Fewa

+237650661631''', recipientProviders: [buildUser(), developers()], subject: 'success', to: 'paypal-team@gmail.com'

        }

    }

}
```

The above single Jenkins Shared Library now contains functions for each stage of the pipeline, allowing you to easily reuse and maintain your Jenkins Pipeline scripts across multiple jobs.