Slave To Master Access Control

- What is this?
- My Jenkins is broken, help!
  - Check if the updates to plugins are available
    - Credentials Plugin
    - SSH Credentials Plugin
    - Git Client Plugin
  - Enable the access control mechanism
  - Disable the access control mechanism altogether
- Whitelisting
  - Command whitelisting
  - File access rules
    - Path matching
  - Whitelisting from GUI
- I'm a plugin developer. What should I do?
  - Vetting SlaveToMasterCallable/SlaveToMasterFileCallable
  - Fixing plugins without requiring newer Jenkins
  - File access from agent to master
  - I'm not sure how to adapt my plugin, I need help

What is this?

Historically, Jenkins master and agents behaved as if they all together form a single distributed process. This means an agent can ask a master to do just about anything within the confinement of the operating system, such as accessing files on the master or trigger other jobs on Jenkins.

This has increasingly become problematic, as larger enterprise deployments have developed more sophisticated trust separation model, where the administrators of a master might take agents owned by other teams. In such an environment, agents are less trusted than the master.

Starting 1.587 (and 1.580.1), Jenkins added a subsystem to put a wall between a master and an agent to safely allow less trusted agents to be connected to a master. Since Jenkins 2.0, this subsystem is enabled for all new installations of Jenkins. If you fit the "larger enterprise deployment" description above, then we highly recommend you turn this mode on. The following are examples of deployments that fit this profile:

- You took agents that are managed by another person who is not Jenkins administrator because they have special requirement for their build jobs
- You have some jobs that are configured to run on a specific agent because it is sensitive

To turn this switch on, go to "Manage Jenkins" > "Configure Global Security", and check "Enable Agent Master Access Control"

On the other hand, if all your agents are trusted to the same degree as your master, then it is safe to leave this subsystem off. The following are examples of deployments that fit this profile:

- You have set up your Jenkins master and all the agents by yourself
- All the users of your Jenkins have the same access to all the jobs

For this subsystem to function, it needs cooperation from plugins. And until plugins get updated to work in this new environment, this may cause seemingly legitimate operations to fail, resulting in broken builds and other failures.

This Wiki page is intended to be the one-stop shop for those who are experiencing problems due to this subsystem.

My Jenkins is broken, help!

If you see an exception like the following in console output, logs, etc., then you are affected by this access control mechanism:

```
java.lang.SecurityException: agent may not create /var/lib/jenkins/foo/bar/...
See http://jenkins-ci.org/security-144 for more details

java.lang.SecurityException: Sending org.jenkinsci.plugins.gitclient.CliGitAPIImpl$GetPrivateKeys from agent to
master is prohibited.
See http://jenkins-ci.org/security-144 for more details
```

The first message indicates that an agent tried to access a file on the master, and Jenkins has blocked that. It is possible that this indicates an active attack, but it is also possible that plugins that haven't yet upgraded to work with SECURITY-144 is doing something legitimate.

The second message indicates that an agent has asked the master to execute a command, and Jenkins has blocked that because the command isn't classified by plugin developers whether it is safe to be executed from the agent.

To resolve this problem, you have the following options:

- Check if the updates to plugins are available
- Disable the access control mechanism altogether
- Whitelist the access

Check if the updates to plugins are available

During the testing of this feature, the following plugins and their versions have been known to interfere with this subsystem. If you are using one of those, upgrading those plugins will resolve the problem. This section will be updated as we discover affected plugins.

Credentials Plugin

This plugin should be upgraded to 1.17. Earlier versions have command implementations that are vulnerable to malicious agents.

SSH Credentials Plugin

This plugin should be upgraded to 1.10. Earlier versions have command implementations that are vulnerable to malicious agents.

Git Client Plugin

Versions up to 1.10.2 expose vulnerabilities in credentials plugin and SSH credentials plugin (see above). Upgrading those plugins will fix the problem, but there's also a separate fix in git-client plugin 1.11.0 itself that addresses this problem.

Enable the access control mechanism

There are essentially three ways of enabling the access control mechanism:

1. Through web UI, go to http://jenkins/configureSecurityand check "Enable Agent Master Access Control" option.
2. Through file system, create or edit the file `$JENKINS_HOME/secrets/slave-to-master-security-kill-switch` so that it contains `false`
3. Using a Groovy Hook Script and doing something like this:

```
import jenkins.security.s2m.AdminWhitelistRule
import jenkins.model.Jenkins
Jenkins.instance.getInjector().getInstance(AdminWhitelistRule.class)
.setMasterKillSwitch(false)
```

Disable the access control mechanism altogether

If all your agents are trusted to the same degree as your master, then it is safe to disable this subsystem completely. See the top of this page for a discussion of who can safely do this vs who should avoid doing this.

If you convinced yourself that this is the right thing to do, you can go to http://jenkins/configureSecurity and uncheck "Enable Agent Master Access Control" option. This setting is remembered by `$JENKINS_HOME/secrets/slave-to-master-security-kill-switch`. The file should either contain `true` or `false` as the content (when the *kill-switch* is set to `false`, it means that *Agent Master Access Control* is enabled). If you do not want this be configurable by an administrator, you can make this file read-only for Jenkins.

For instances that have no agents and instances that do not enable security, we do not expose this configuration switch as it is pointless.

Your Jenkins deployment may grow over time and start accepting less trusted agents. It's too easy for that to happen without you remembering this flag. So be careful when you do this. Please revisit this later to see if you should enable this subsystem.

Whitelisting

Jenkins allows administrators to whitelist specific commands and file accesses on their Jenkins instances. This configuration consists of two pieces.

Command whitelisting

Commands in Jenkins and its plugins are identified by their fully-qualified class names. Majority of those commands are intended to be executed on agents by a request of a master, but some of them are intended to be executed on a master by a request of an agent. Plugins not yet updated for this subsystem does not classify which category each command falls into. So when an agent requests a master to execute a command and if it is not classified explicitly as intended for agent master, Jenkins will err on the side of caution and refuses to execute the command.

Until all such plugins are properly updated, administrators can mark specific commands as intended to be executed on a master. We call this "whitelisting".

Administrators can whitelist classes by writing `$JENKINS_HOME/secrets/whitelisted-callables.d/*.conf` and listing command names in separate lines. All such files are read and the result gets combined. Jenkins by itself generates `default.conf` in this directory, which lists known safe commands. This file gets always overwritten by Jenkins every time it starts, but if you do not want to whitelist these classes for some reasons, you can do so by placing a file that's not writable by Jenkins.

Jenkins also manages `gui.conf` in this directory, which is editable through GUI as discussed later. If you do not want to allow Jenkins admins to whitelist anything, create an empty file that's not writable by Jenkins.

Whitelisting has to be done carefully

Whitelisting a command requires not only verifying that the command is intended to be used in this direction, but also that the command implementation is not exploitable by malicious agents. This requires careful analysis of the source code, taking such things into account as all possible serializable fields. As a user, you should just report those commands, and wait for project developers to perform this vetting process. Once we verified that they are safe, you can whitelist them by using this mechanism.

File access rules

File access request from agents is tested against the rules you specify. Each rule is a tuple that consists of:

- **allow/deny**: if the following two parameters match the current request being considered, an "allow" entry would allow the request to be carried out and a "deny" entry would deny the request to be rejected, regardless of what later rules might say.
- **operation**: the type of the operation requested. The following 6 values exist. You can also list them separating with ',' or use "all" to indicate a match for all operations:
  - read: read file content or list directory entries
  - write: write file content
  - mkdirs: create a new directory
  - create: create a file in an existing directory
  - delete: delete a file or directory
  - stat: read metadata of a file/directory, such as timestamp, length, file access modes.
- **file path**: regular expression that specifies file paths that match this rule. In addition to the base regexp syntax, it supports the following tokens:
  - `<JENKINS_HOME>` can be used as a prefix to match your $JENKINS_HOME directory
  - `<BUILDDIR>` can be used as a prefix to match your build record directory, such as `/var/lib/jenkins/job/foo/builds/2014-10-17_12-34-56`
  - `<BUILDID>` matches the timestamp-formatted build IDs, like `2014-10-17_12-34-56`.

The rules are ordered and applied in that order. The earliest match wins. So for example, the following rules allow access to `$JENKINS_HOME` except its `secrets` folders:

```
# To avoid hassle of escaping every '\' on Windows, you can use / everywhere, even on Windows.
deny all <JENKINS_HOME>/secrets/.*
allow all <JENKINS_HOME>/.*
```

The following rules are incorrectly written because the 2nd rule will never match:

```
allow all <JENKINS_HOME>/.*
deny all <JENKINS_HOME>/secrets/.*
```

Rules are read from `$JENKINS_HOME/secrets/filepath-filters.d/*.conf` after sorting these files in alphabetical order.

Jenkins by itself generates `30-default.conf` in this directory, which lists rules that the Jenkins core developers currently think are the best balance between compatibility and security. This file gets overwritten by Jenkins every time it starts, but if you do not want to whitelist these classes for some reasons, you can do so by placing a file with that name that's not writable by Jenkins.

Jenkins also manages `50-gui.conf` in this directory, which is editable through GUI as discussed later. If you do not want to allow Jenkins admins to whitelist anything, create an empty file that's not writable by Jenkins.

Unlike command whitelisting, file access rule decisions can be made individually based on common sense.

Path matching

When a file access is checked, the path of a file being considered is absolutized (i.e., can be `/foo/bar/zot` but not `./zot`). It is also normalized to remove all intermediate "." and "..". So a regular expression `/foo/bar/zot.*` will never match `/foo/bar/zot/../../../etc/passwd`, and likewise a regular expression `/foo/bar/../zot/.+` will never match `/foo/zot/bar`.

A path is not always canonicalized. So if you have a symlink in `/var/lib/jenkins/passwd` that points to `/etc/passwd`, and if you allow read access to `/var/lib/jenkins/.*`, then `/etc/passwd` can be read.

The following Groovy script can be used from http://jenkins/script to test the rules:

```
import jenkins.security.admin.*;
import jenkins.security.s2m.AdminWhitelistRule;

String op = "write"; // or any other operation like "read"
File f = new File("/userContent/some-path");
Jenkins.instance.injector.getInstance(AdminWhitelistRule.class).checkFileAccess(op,f)
// true means whitelisted. false or SecurityException means rejected
```

More precisely, `FilePath` always internally normalize paths, and while it allows relative paths, no legitimate code will ever use it, so it shouldn't have to be factored in when writing rules. It isn't that the access checking subsystem does normalization/absolutization.

Whitelisting from GUI

On Jenkins, you can go to http://jenkins/administrativeMonitor/slaveToMasterAccessControl/ to edit whitelist rules from GUI and have them reflected right away in the running instance. It consists of the following three sections:

- **Currently whitelisted commands**: See above for what this field means.
- **Currently rejected commands**: This section lists unclassified commands that Jenkins has actually rejected. You can check boxes and submit them to have Jenkins write them into the "currently whitelisted commands" section. Be careful when you do this, though. See the command whitelisting discussion above for the implications.
- **File access rules**: See above for what this field means.

When submitted, these changes are written back to disk and then re-read right away into Jenkins, including all `whitelisted-callables.d/*.conf` and `filepath-filters.d/*.conf` files.

I'm a plugin developer. What should I do?

For the access control to work without requiring manual intervention by users, plugins need to classify their `Callable` and `FileCallable` objects whether they are meant to be run on a master or on an agent.

For this purpose, the `remoting` library has added the `RoleSensitive` interface with a `checkRoles()` method. `Callable`, `FileCallable`, and other similar interfaces extend from this interface. So if you are directly implementing `Callable` you will get an error saying that you have unimplemented abstract methods.

The easiest way to fix this is by extending from `MasterToSlaveCallable`, to indicate that your `Callable` is only meant to be sent from a master to an agent, or `SlaveToMasterCallable`, to indicate that your `Callable` is meant to be sent from an agent to a master. Note that `SlaveToMasterCallable` can still be executed on an agent, as agents do not perform this access control check. `FileCallable` similarly has `MasterToSlaveFileCallable` and `SlaveToMasterFileCallable`.

Vetting `SlaveToMasterCallable`/`SlaveToMasterFileCallable`

When marking `Callable` for agent master, care has to be taken to ensure that the implementation is not exploitable by malicious agents.

- A malicious agent controls the Java serialization payload, so when your `Callable` gets deserialized on the master, all the serialized fields are controlled by the agent.
- An agent does not control class definitions on the master, so you can trust all the classes and methods to behave as it is written. It is not possible for a malicious agent to change the code executed on the master.

For example, the following `SlaveToMasterCallable` is exploitable. Callable itself is not public, but a malicious agent can send in arbitrary `path`, so it can be used to read any file on the master:

```
// UNSAFE
class SomeCodeThatRunsOnAgent {
    void readBackSomeFileFromMaster() {
        final String path = "...";
        channel.call(new SlaveToMasterCallable<String,IOException>() {
            public String call() {
                return FileUtils.readFileToString(new File(path));
            }
        });
    }
}
```

`Callable` that delegates execution to a deserialized object is dangerous and needs to be carefully examined, because a malicious agent can designate unintended `Runnable` object:

```
// UNSAFE
class MyCallable extends SlaveToMasterCallable<Void> {
    Runnable r;
    public Void call() {
        r.run();
        return null;
    }
}
```

To avoid this hassle entirely, consider rewriting your code not to call back to a master from an agent. Instead, when a master first sends a command to an agent, you can carry all the data you'll need with you. This may not be always possible or practical, but it's a lot easier to secure.

Fixing plugins without requiring newer Jenkins

Classifying `Callable`/`FileCallable` requires new classes added to Jenkins 1.587/1.580.1. This poses a challenge if you want to retain backward compatibility with earlier versions of Jenkins.

To solve this problem, we've developed SECURITY-144-compat module. This module let you classify `Callable`, while still functioning correctly on earlier versions of Jenkins. See the documentation of `SECURITY-144-compat` for details.

Note

As of version 1.1, this library is deprecated, as its use caused some unresolved problems (JENKINS-25625).
Anyway 1.580.1 is now a fairly conservative choice of baseline: you will not exclude so many users by requiring it for new plugin releases.

File access from agent to master

To avoid getting affected by file access rules, have the master work on files of an agent, instead of the other way around.

The following code example shows how the code that used to write a file from an agent now avoids that:

```
// PROBLEMATIC
class MySCM extends SCM {
    ...
    public void checkout( ..., FilePath workspace, File _changelogFile ) {
        FilePath changelogFile = new FilePath(_changelogFile);
        workspace.act(new Callable<Void,IOException>() {
            public Void call() {
                // this results in an agent asking the master to open a file for write
                try (OutputStream os = changelogFile.write()) {
                    writeStuffTo(os);
                }
            }
        });
    }
}

// GOOD
class MySCM extends SCM {
    ...
    public void checkout( ..., FilePath workspace, File _changelogFile ) {
        try (final OutputStream out = new RemoteOutputStream(_changelogFile)) {
            workspace.act(new Callable<Void,IOException>() {
                public Void call() {
                    // agent is just writing to a pipe to the master. Quite safe
                    writeStuffTo(out);
                }
            });
        }
    }
}
```

See `RemoteInputStream`, `RemoteOutputStream`, `RemoteWriter`, and `Pipe` for ways to do this.

I'm not sure how to adapt my plugin, I need help

If you have questions, please write to jenkinsci-dev@googlegroups.com, or talk to us on IRC.