ssl and certificates

Contents

SSL and certificates

Puppet can use its built-in certificate authority (CA) and public key infrastructure (PKI) tools or use an existing external CA for all of its secure socket layer (SSL) communications.

Configuring external certificate authority

This information describes the supported and tested configurations for external CAs in this version of Puppet. If you have an external CA use case that isn't listed here, contact Puppet so we can learn more about it.

Supported external CA configurations

This version of Puppet supports some external CA configurations, however not every possible configuration is supported.

We fully support the following setup options:

- *Single self-signed CA* which directly issues SSL certificates.
- Puppet Server functioning as an intermediate CA of a *root self-signed CA*.

Fully supported by Puppet means:

- If issues arise that are considered bugs, we'll fix them as soon as possible.
- If issues arise in any other external CA setup that are considered feature requests, we'll consider whether to expand our support.

General notes and requirements

PEM encoding of credentials is mandatory

Puppet expects its SSL credentials to be in `.pem` format.

Normal Puppet certificate requirements still apply

Any Puppet Server certificate must contain the DNS name, either as the Subject Common Name (CN) or as a Subject Alternative Name (SAN), that agent nodes use to attempt contact with the server.

Option 1: Single CA

When Puppet uses its internal CA, it defaults to a single CA configuration. A single externally issued CA can also be used in a similar manner.

```
            +-----------------------+
            |                       |
            |  Root self-signed CA  |
            |                       |
            +------+---------+------+
                   |         |
        +----------+         +------------+
        |                                 |
        v                                 v
+-----------------+               +----------------+
|                 |               |                |
| Server SSL Cert |               | Agent SSL Cert |
|                 |               |                |
+-----------------+               +----------------+
```

This is an all or nothing configuration rather than a mix-and-match. When using an external CA, the built-in Puppet CA service must be disabled and cannot be used to issue SSL certificates.

Note: Puppet cannot automatically distribute certificates in this configuration. You must have your own complete system for issuing and distributing the certificate.

Puppet Server

Configure Puppet Server in three steps:

- Disable the internal CA service.
- Ensure that the certname will not change.
- Put certificates and keys in place on disk.

1. To disable the internal CA, edit the Puppet Server `/etc/puppetlabs/puppetserver/services.d/ca.cfg` file to comment and uncomment the appropriate settings:

   ```
   # puppetlabs.services.ca.certificate-authority-service/certificate-authority-service
   puppetlabs.services.ca.certificate-authority-disabled-service/certificate-authority-disabled-service
   ```

2. Set a static value for the `certname` setting in `puppet.conf`:

   ```
   [master]
   certname = puppetserver.example.com
   ```

   Setting a static value prevents any confusion if the machine's hostname changes. The value must match the certname you’ll use to issue the server's certificate, and it must not be blank.

3. Put the credentials from your external CA on disk in the correct locations. These locations must match what’s configured in your *webserver.conf file*.

   If you haven’t changed those settings, run the following commands to find the default locations.

Credential	File location
Server SSL certificate	`puppet config print hostcert --section master`
Server SSL certificate private key	`puppet config print hostprivkey --section master`
Root CA certificate	`puppet config print localcacert --section master`
Root certificate revocation list	`puppet config print hostcrl --section master`

If you've put the credentials in the correct locations, you shouldn't need to change any additional settings.

Puppet agent

You don't need to change any settings. Put the external credentials into the correct filesystem locations. You can run the following commands to find the appropriate locations.

Credential	File location
Agent SSL certificate	`puppet config print hostcert --section agent`
Agent SSL certificate private key	`puppet config print hostprivkey --section agent`
Root CA certificate	`puppet config print localcacert --section agent`
Root certificate revocation list	`puppet config print hostcrl --section agent`

Option 2: Puppet Server functioning as an intermediate CA

Puppet Server can operate as an intermediate CA to an external root CA.

For more information, see *Using Puppet Server as an intermediate certificate authority*.

Autosigning certificate requests

Before Puppet agent nodes can retrieve their configuration catalogs, they require a signed certificate from the local Puppet certificate authority (CA). When using Puppet's built-in CA instead of an external CA, agents submit a certificate signing request (CSR) to the CA Puppet master to retrieve a signed certificate once it's available.

By default, these CSRs must be manually signed by an admin user, using either the `puppet cert` command or the **Node requests** page in the Puppet Enterprise console.

Alternatively, to speed up the process of bringing new agent nodes into the deployment, you can configure the CA Puppet master to automatically sign certain CSRs.

CAUTION: Autosigning CSRs changes the nature of your deployment's security, and you should understand the implications before configuring it. Each type of autosigning has its own security impact.

Disabling autosigning

By default, the `autosign` setting in the `[master]` section of the CA Puppet master's `puppet.conf` file is set to `$confdir/autosign.conf`. This which that the basic autosigning functionality is enabled upon installation.

Depending on your installation method, there might not be a whitelist at that location once the Puppet master is running:

- Open source Puppet: `autosign.conf` doesn't exist by default.
- Monolithic Puppet Enterprise (PE) installations: All required services run on one server, and `autosign.conf` exists on the master, but by default it's empty because the master doesn't need to whitelist other servers.
- Split PE installations: Services like PuppetDB can run on different servers, the `autosign.conf` exists on the CA master and contains a whitelist of other required hosts.

If the `autosign.conf` file is empty or doesn't exist, the whitelist is effectively empty. The CA Puppet master doesn't autosign any certificates until the the autosign setting's path is configured, or until the default `autosign.conf` file is a non-executable whitelist file. This file must contain correctly formatted content or a custom policy executable that the Puppet user has permission to run.

To explicitly disable autosigning, set `autosign = false` in the `[master]` section of the CA Puppet master's `puppet.conf`. This disables CA autosigning even if the `autosign.conf` file or a custom policy executable exists.

For more information about the `autosign` setting in `puppet.conf`, see the *configuration reference*.

Naïve autosigning

Naïve autosigning causes the CA to autosign all CSRs.

To enable naïve autosigning, set `autosign = true` in the `[master]` section of the CA Puppet master's `puppet.conf`.

CAUTION: For security reasons, never use naïve autosigning in a production deployment. Naïve autosigning is suitable only for temporary test deployments that are incapable of serving catalogs containing sensitive information.

Basic autosigning (`autosign.conf`)

In basic autosigning, the CA uses a config file containing a whitelist of certificate names and domain name globs. When a CSR arrives, the requested certificate name is checked against the whitelist file. If the name is present, or covered by one of the domain name globs, the certificate is autosigned. If not, it's left for a manual review.

Enabling basic autosigning

The `autosign.conf` whitelist file's location and contents are described in its *documentation*.

Puppet looks for `autosign.conf` at the path configured in the `[autosign setting]` within the `[master]` section of `puppet.conf`. The default path is `$confdir/autosign.conf`, and the default `confdir` path depends on your operating system. For more information, see the *confdir documentation.*

If the `autosign.conf` file pointed to by the `autosign` setting is a file that the Puppet user can execute, Puppet instead attempts to run it as a custom policy executable, even if it contains a valid `autosign.conf` whitelist.

Note: In open source Puppet, no `autosign.conf` file exists by default. In Puppet Enterprise, the file exists by default but might be empty. In both cases, the basic autosigning feature is technically enabled by default but doesn't autosign any certificates because the whitelist is effectively empty.

The CA Puppet master therefore doesn't autosign any certificates until the `autosign.conf` file contains a properly formatted whitelist or is a custom policy executable that the Puppet user has permission to run, or until the `autosign` setting is pointed at a whitelist file with properly formatted content or a custom policy executable that the Puppet user has permission to run.

Security implications of basic autosigning

Basic autosigning can be considered insecure because any host can provide any certname when requesting a certificate. Only use it when you fully trust any computer capable of connecting to the Puppet master.

With basic autosigning enabled, an attacker who guesses an unused certname allowed by `autosign.conf` can obtain a signed agent certificate from the Puppet master. The attacker could then obtain a configuration catalog, which can contain sensitive information depending on your deployment's Puppet code and node classification.

Policy-based autosigning

In policy-based autosigning, the CA will run an external policy executable every time it receives a CSR. This executable will examine the CSR and tell the CA whether the certificate is approved for autosigning. If the executable approves, the certificate is autosigned; if not, it's left for manual review.

Enabling policy-based autosigning

To enable policy-based autosigning, set `autosign = <policy executable file>` in the `[master]` section of the CA Puppet master's `puppet.conf`.

The policy executable file must be executable by the same user as the Puppet master. If not, it will be treated as a certname whitelist file.

Custom policy executables

A custom policy executable can be written in any programming language; it just has to be executable in a *nix-like environment. The Puppet master will pass it the certname of the request (as a command line argument) and the PEM-encoded CSR (on stdin), and will expect a `0` (approved) or non-zero (rejected) exit code.

Once it has the CSR, a policy executable can extract information from it and decide whether to approve the certificate for autosigning. This is useful when you are provisioning your nodes and are *embedding additional information in the CSR*.

If you aren't embedding additional data, the CSR will contain only the node's certname and public key. This can still provide more flexibility and security than `autosign.conf`, as the executable can do things like query your provisioning system, CMDB, or cloud provider to make sure a node with that name was recently added.

Security implications of policy-based autosigning

Depending on how you manage the information the policy executable is using, policy-based autosigning can be fast and extremely secure.

For example:

- If you embed a unique pre-shared key on each node you provision, and provide your policy executable with a database of these keys, your autosigning security will be as good as your handling of the keys. As long as it's impractical for an attacker to acquire a PSK, it will be impractical for them to acquire a signed certificate.
- If nodes running on a cloud service embed their instance UUIDs in their CSRs, and your executable queries the cloud provider's API to check that a node's UUID exists in your account, your autosigning security will be as good as the security of the cloud provider's API. If an attacker can impersonate a legit user to the API and get a list of node UUIDs, or if they can create a rogue node in your account, they can acquire a signed certificate.

When designing your CSR data and signing policy, you must think things through carefully. As long as you can arrange reasonable end-to-end security for secret data on your nodes, you should be able to configure a secure autosigning system.

Policy executable API

The API for policy executables is as follows.

Run environment	• The executable will run once for each incoming CSR. • It will be executed by the Puppet master process and will run as the same user as the Puppet master. • The Puppet master process will block until the executable finishes running. We expect policy

	executables to finish in a timely fashion; if they do not, it's possible for them to tie up all available Puppet master threads and deny service to other agents. If an executable needs to perform network requests or other potentially expensive operations, the author is in charge of implementing any necessary timeouts, possibly bailing and exiting non-zero in the event of failure. Alternatively, signing requests consume JRubies on a Puppet Server master but might not block all requests while the pool contains available JRubies, and won't block non-Ruby requests.
Arguments	• The executable must allow a single command line argument. This argument will be the Subject CN (certname) of the incoming CSR. • No other command line arguments should be provided. • The Puppet master should never fail to provide this argument.
Stdin	• The executable will receive the entirety of the incoming CSR on its stdin stream. The CSR will be encoded in `pem` format. • The stdin stream will contain nothing but the complete CSR. • The Puppet master should never fail to provide the CSR on stdin.
Exit status	• The executable must exit with a status of `0` if the certificate should be autosigned; it must exit with a non-zero status if it should not be autosigned. • The Puppet master will treat all non-zero exit statuses as equivalent.
Stdout and stderr	• Anything the executable emits on stdout or stderr will be copied to the Puppet master's log output at the `debug` log level. Puppet will otherwise ignore the executable's output; only the exit code is considered significant.

CSR attributes and certificate extensions

When Puppet agent nodes request their certificates, the certificate signing request (CSR) usually contains only their certname and the necessary cryptographic information. Agents can also embed additional data in their CSR, useful for policy-based autosigning and for adding new trusted facts.

Embedding additional data into CSRs is useful when:

- Large numbers of nodes are regularly created and destroyed as part of an elastic scaling system.
- You are willing to build custom tooling to make certificate autosigning more secure and useful.

It might also be useful in deployments where Puppet is used to deploy private keys or other sensitive information, and you want extra control over nodes that receive this data.

If your deployment doesn't match one of these descriptions, you might not need this feature.

Timing: When data can be added to CSRs and certificates

When Puppet agent starts the process of requesting a catalog, it checks whether it has a valid signed certificate. If it does not, it generates a key pair, crafts a CSR, and submits it to the certificate authority (CA) Puppet master. For detailed information, see *agent/master HTTPS traffic*.

For practical purposes, a certificate is locked and immutable as soon as it is signed. For data to persist in the certificate, it has to be added to the CSR before the CA signs the certificate.

This means any desired extra data must be present before Puppet agent attempts to request its catalog for the first time.

You should populate any extra data when provisioning the node. If you make an error, see *Recovering From Failed Data Embedding* below.

Data location and format

Extra data for the CSR is read from the `csr_attributes.yaml` file in Puppet's `confdir`. The location of this file can be changed with the `csr_attributes` configuration setting.

The `csr_attributes.yaml` file must contain a YAML hash with one or both of the following keys:

- `custom_attributes`
- `extension_requests`

The value of each key must also be a hash, where:

- Each key is a valid *object identifier (OID)* — *Puppet-specific OIDs* can optionally be referenced by short name instead of by numeric ID.
- Each value is an object that can be cast to a string — numbers are allowed but arrays are not.

For information about how each hash is used and recommended OIDs for each hash, see the sections below.

Custom attributes (transient CSR data)

Custom attributes are pieces of data that are only embedded in the CSR. The CA can use them when deciding whether to sign the certificate, but they are discarded after that and aren't transferred to the final certificate.

Default behavior

The `puppet cert list` command doesn't display custom attributes for pending CSRs, and *basic autosigning (autosign.conf)* doesn't check them before signing.

Configurable behavior

If you use *policy-based autosigning* your policy executable receives the complete CSR in PEM format. The executable can extract and inspect the custom attributes, and use them to decide whether to sign the certificate.

The simplest method is to embed a pre-shared key of some kind in the custom attributes. A policy executable can compare it to a list of known keys and autosign certificates for any pre-authorized nodes.

A more complex use might be to embed an instance-specific ID and write a policy executable that can check it against a list of your recently requested instances on a public cloud, like EC2 or GCE.

If you use Puppet Server 2.5.0 or higher, you can also sign requests using authorization extensions and the `--allow-authorization-extensions` flag for `puppet cert sign`.

Manually checking for custom attributes in CSRs

You can check for custom attributes by using OpenSSL to dump a CSR in `pem` format to text format, by running this command:

```
openssl req -noout -text -in <name>.pem
```

In the output, look for the `Attributes` section which appears below the `Subject Public Key Info` block:

```
Attributes:
    challengePassword        :342thbjkt82094y0uthhor289jnqthpc2290
```

Recommended OIDs for attributes

Custom attributes can use any public or site-specific OID, with the exception of the OIDs used for core X.509 functionality. This means you can't re-use existing OIDs for things like subject alternative names.

One useful OID is the `challengePassword` attribute — `1.2.840.113549.1.9.7`. This is a rarely-used corner of X.509 that can easily be repurposed to hold a pre-shared key. The benefit of using this instead of an arbitrary OID is that it appears by name when using OpenSSL to dump the CSR to text; OIDs that `openssl req` can't recognize are displayed as numerical strings.

You can also use the *Puppet-specific OIDs*.

Extension requests (permanent certificate data)

Extension requests are pieces of data that are transferred as extensions to the final certificate, when the CA signs the CSR. They persist as trusted, immutable data, that cannot be altered after the certificate is signed.

They can also be used by the CA when deciding whether or not to sign the certificate.

Default behavior

When signing a certificate, Puppet's CA tools transfer any extension requests into the final certificate.

You can access certificate extensions in manifests as `$trusted[extensions][<EXTENSION OID>]`.

Any OIDs in the *ppRegCertExt* range appear using their short names. By default, any other OIDs appear as plain dotted numbers, but you can use the *`custom_trusted_oid_mapping.yaml`* file to assign short names to any other OIDs you use at your site. If you do, those OIDs will appear in `$trusted` as their short names, instead of their full numerical OID.

For more information about `$trusted`, see *facts and special variables*.

The visibility of extensions is limited:

- The `puppet cert list` command does not display custom attributes for any pending CSRs, and *basic autosigning (autosign.conf)* doesn't check them before signing. Either use *policy-based autosigning* or inspect CSRs manually with the `openssl` command (see below).
- The `puppet cert print` command does display any extensions in a signed certificate, under the `X509v3 extensions` section.

Puppet's authorization system (`auth.conf`) does not use certificate extensions, but *Puppet Server's authorization system*, which is based on `trapperkeeper-authorization`, can use extensions in the `ppAuthCertExt` OID range, and requires them for requests to write access rules.

Configurable behavior

If you use *policy-based autosigning*, your policy executable receives the complete CSR in `pem` format. The executable can extract and inspect the extension requests, and use them when deciding whether to sign the certificate.

Manually checking for extensions in CSRs and certificates

You can check for extension requests in a CSR by running the OpenSSL command to dump a CSR in `pem` format to text format:

```
openssl req -noout -text -in <name>.pem
```

In the output, look for a section called `Requested Extensions`, which appears below the `Subject Public Key Info` and `Attributes` blocks:

```
Requested Extensions:
    pp_uuid:
    .$ED803750-E3C7-44F5-BB08-41A04433FE2E
    1.3.6.1.4.1.34380.1.1.3:
    ..my_ami_image
    1.3.6.1.4.1.34380.1.1.4:
    .$342thbjkt82094y0uthhor289jnqthpc2290
```

Note: Every extension is preceded by any combination of two characters (`.$` and `..` in the example above) that contain ASN.1 encoding information. Because OpenSSL is unaware of Puppet's custom extensions OIDs, it's unable to properly display the values.

Any Puppet-specific OIDs (see below) appear as numeric strings when using OpenSSL.

You can check for extensions in a signed certificate by running `puppet cert print <name>`. In the output, look for the `X509v3 extensions` section. Any of the Puppet-specific *registered OIDs* appear as their descriptive names:

```
X509v3 extensions:
    Netscape Comment:
        Puppet Ruby/OpenSSL Internal Certificate
    X509v3 Subject Key Identifier:
        47:BC:D5:14:33:F2:ED:85:B9:52:FD:A2:EA:E4:CC:00:7F:7F:19:7E
    Puppet Node UUID:
        ED803750-E3C7-44F5-BB08-41A04433FE2E
    X509v3 Extended Key Usage: critical
        TLS Web Server Authentication, TLS Web Client Authentication
    X509v3 Basic Constraints: critical
        CA:FALSE
    Puppet Node Preshared Key:
        342thbjkt82094y0uthhor289jnqthpc2290
    X509v3 Key Usage: critical
        Digital Signature, Key Encipherment
    Puppet Node Image Name:
        my_ami_image
```

Recommended OIDs for extensions

Extension request OIDs must be under the `ppRegCertExt` (`1.3.6.1.4.1.34380.1.1`), `ppPrivCertExt` (`1.3.6.1.4.1.34380.1.2`), or `ppAuthCertExt` (`1.3.6.1.4.1.34380.1.3`) OID arcs.

Puppet provides several registered OIDs (under `ppRegCertExt`) for the most common kinds of extension information, a private OID range (`ppPrivCertExt`) for site-specific extension information, and an OID range for safe authorization to Puppet Server (`ppAuthCertExt`).

There are several benefits to using the registered OIDs:

- You can reference them in the `csr_attributes.yaml` file with their short names instead of their numeric IDs.
- You can access them in `$trusted[extensions]` with their short names instead of their numeric IDs.

- When using Puppet tools to print certificate info, they will appear using their descriptive names instead of their numeric IDs.

The private range is available for any information you want to embed into a certificate that isn't widely used already. It is completely unregulated, and its contents are expected to be different in every Puppet deployment.

You can use the *custom_trusted_oid_mapping.yaml* file to set short names for any private extension OIDs you use. Note that this only enables the short names in the `$trusted[extensions]` hash.

Puppet-specific registered IDs

`ppRegCertExt`

The `ppRegCertExt` OID range contains the following OIDs:

Numeric ID	Short name	Descriptive name
1.3.6.1.4.1.34380.1.1.1	`pp_uuid`	Puppet node UUID
1.3.6.1.4.1.34380.1.1.2	`pp_instance_id`	Puppet node instance ID
1.3.6.1.4.1.34380.1.1.3	`pp_image_name`	Puppet node image name
1.3.6.1.4.1.34380.1.1.4	`pp_preshared_key`	Puppet node preshared key
1.3.6.1.4.1.34380.1.1.5	`pp_cost_center`	Puppet node cost center name
1.3.6.1.4.1.34380.1.1.6	`pp_product`	Puppet node product name
1.3.6.1.4.1.34380.1.1.7	`pp_project`	Puppet node project name
1.3.6.1.4.1.34380.1.1.8	`pp_application`	Puppet node application name
1.3.6.1.4.1.34380.1.1.9	`pp_service`	Puppet node service name
1.3.6.1.4.1.34380.1.1.10	`pp_employee`	Puppet node employee name
1.3.6.1.4.1.34380.1.1.11	`pp_created_by`	Puppet node `created_by` tag
1.3.6.1.4.1.34380.1.1.12	`pp_environment`	Puppet node environment name
1.3.6.1.4.1.34380.1.1.13	`pp_role`	Puppet node role name
1.3.6.1.4.1.34380.1.1.14	`pp_software_version`	Puppet node software version
1.3.6.1.4.1.34380.1.1.15	`pp_department`	Puppet node department name
1.3.6.1.4.1.34380.1.1.16	`pp_cluster`	Puppet node cluster name
1.3.6.1.4.1.34380.1.1.17	`pp_provisioner`	Puppet node provisioner name
1.3.6.1.4.1.34380.1.1.18	`pp_region`	Puppet node region name
1.3.6.1.4.1.34380.1.1.19	`pp_datacenter`	Puppet node datacenter name
1.3.6.1.4.1.34380.1.1.20	`pp_zone`	Puppet node zone name
1.3.6.1.4.1.34380.1.1.21	`pp_network`	Puppet node network name
1.3.6.1.4.1.34380.1.1.22	`pp_securitypolicy`	Puppet node security policy name
1.3.6.1.4.1.34380.1.1.23	`pp_cloudplatform`	Puppet node cloud platform name
1.3.6.1.4.1.34380.1.1.24	`pp_apptier`	Puppet node application tier
1.3.6.1.4.1.34380.1.1.25	`pp_hostname`	Puppet node hostname

`ppAuthCertExt`

The `ppAuthCertExt` OID range contains the following OIDs:

Numeric ID	Short name	Descriptive name
1.3.6.1.4.1.34380.1.3.1	`pp_authorization`	Certificate extension authorization
1.3.6.1.4.1.34380.1.3.13	`pp_auth_role`	Puppet node role name for authorization

AWS attributes and extensions population example

To populate the `csr_attributes.yaml` file when you provision a node, use an automated script such as cloud-init.

For example, when provisioning a new node from the AWS EC2 dashboard, enter the following script into the **Configure Instance Details —> Advanced Details** section:

```
#!/bin/sh
if [ ! -d /etc/puppetlabs/puppet ]; then
   mkdir /etc/puppetlabs/puppet
fi
cat > /etc/puppetlabs/puppet/csr_attributes.yaml << YAML
custom_attributes:
    1.2.840.113549.1.9.7: mySuperAwesomePassword
extension_requests:
    pp_instance_id: $(curl -s http://169.254.169.254/latest/meta-data/
instance-id)
    pp_image_name:  $(curl -s http://169.254.169.254/latest/meta-data/ami-
id)
YAML
```

Assuming your image has the `erb` binary available, this populates the attributes file with the AWS instance ID, image name, and a pre-shared key to use with policy-based autosigning.

Troubleshooting

Recovering from failed data embedding

When testing this feature for the first time, you might not embed the right information in a CSR, or certificate, and might want to start over for your test nodes. This is not really a problem once your provisioning system is changed to populate the data, but it can easily happen when doing things manually.

To start over, do the following.

On the test node:

- Turn off Puppet agent, if it's running.
- Check whether a CSR is present in `$ssldir/certificate_requests/<name>.pem`. If it exists, delete it.
- Check whether a certificate is present in `$ssldir/certs/<name>.pem`. If it exists, delete it.

On the CA Puppet master:

- Check whether a signed certificate exists. Use `puppet cert list --all` to see the complete list. If it exists, revoke and delete it with `puppet cert clean <name>`.
- Check whether a CSR for the node exists in `$ssldir/ca/requests/<name>.pem`. If it exists, delete it.

After you've done that, you can start over.

Regenerating all certificates in a Puppet deployment

In some cases, you might need to regenerate the certificates and security credentials (private and public keys) that are generated by Puppet's built-in certificate authority (CA).

For example, you might have a Puppet master you need to move to a different network in your infrastructure, or you might have experienced a security vulnerability that makes existing credentials untrustworthy.

Note: If you're visiting this page to remediate your Puppet Enterprise deployment due to *CVE-2014-0160*, also known as Heartbleed, see this *announcement* for additional information and links to more resources. Before applying these instructions, please note that this is a non-trivial operation that contains some manual steps and will require you to replace certificates on every agent node managed by your Puppet master.

Important: The information on this page describes the steps for regenerating certs in an open source Puppet deployment. If you use Puppet Enterprise do not use the information on this page, as it will leave you with an incomplete replacement and non-functional deployment. Instead, PE customers must refer to one of the following pages:

- *Regenerating certificates in split PE deployments*
- *Regenerating certificates in monolithic PE deployments*

Regardless of your situation, regenerating your certs involves the following three steps, described in detail in the sections below:

1. On your master, you'll clear the certs and security credentials, regenerate the CA, and then regenerate the certs and security credentials.
2. You'll clear and regenerate certs and security credentials for any extensions.
3. You'll clear and regenerate certs and security credentials for all agent nodes.

CAUTION: This process destroys the certificate authority and all other certificates. It is meant for use in the event of a total compromise of your site, or some other unusual circumstance. If you just need to replace a few agent certificates, use the `puppet cert clean` command on your Puppet master and then follow step 3 for any agents that need to be replaced.

Step 1: Clear and regenerate certs on your Puppet master

On the Puppet master hosting the CA:

1. Back up the *SSL directory*, which should be `/etc/puppetlabs/puppet/ssl/`. If something goes wrong, you can restore this directory so your deployment can stay functional. However**,** if you needed to regenerate your certs for security reasons and couldn't, you should get some assistance as soon as possible so you can keep your site secure.
2. Stop the agent service:

   ```
   sudo puppet resource service puppet ensure=stopped
   ```
3. Stop the master service.

   For Puppet Server, run:

   ```
   sudo puppet resource service puppetserver ensure=stopped
   ```

   For a Rack-based master, stop whatever web server you're using.
4. Delete the SSL directory:

   ```
   sudo rm -r /etc/puppetlabs/puppet/ssl
   ```
5. Regenerate the CA:

   ```
   sudo puppetserver ca list -a
   ```

   You should see this message: `Notice: Signed certificate request for ca.`
6. Generate the Puppet master's new certs:

   ```
   sudo puppet master --no-daemonize --verbose
   ```

7. When you see the message `Notice: Starting Puppet master <VERSION>`, type **CTRL** + **C**.
8. Start the Puppet master service either by restarting the web server you stopped in step 3, if a Rack-based master, or by running:

    ```
    sudo puppet resource service puppetserver ensure=running
    ```

9. Start the Puppet agent service by running this command:

    ```
    sudo puppet resource service puppet ensure=running
    ```

At this point:

- You have a new CA certificate and key.
- Your Puppet master has a certificate from the new CA, and it can field new certificate requests.
- The Puppet master will reject any requests for configuration catalogs from nodes that haven't replaced their certificates. At this point, it will be all of them except itself.
- If you are using any extensions that rely on Puppet certificates, like PuppetDB or MCollective, the Puppet master won't be able to communicate with them. Consequently, it might not be able to serve catalogs, even to agents that do have new certificates.

Step 2: Clear and regenerate certs for any extension

You might be using an extension, like PuppetDB or MCollective, to enhance Puppet. These extensions probably use certificates from Puppet's CA in order to communicate securely with the Puppet master. For each extension like this, you'll need to regenerate the certificates it uses.

Many tools have scripts or documentation to help you set up SSL, and you can often just re-run the setup instructions.

PuppetDB

PuppetDB users should first follow the instructions in Step 3: Clear and regenerate certs for agents, below, since PuppetDB re-uses Puppet agents' certificates. After that, restart the PuppetDB service.

MCollective

MCollective often uses SSL certificates from Puppet's CA. If you are replacing your Puppet CA and are using the same certs for MCollective, refer to the standard *deployment guide* and re-do any steps involving security credentials. You'll generally need to replace client certificates, your server keypair, and the ActiveMQ server's keystore and truststore.

Important: As of Puppet agent 5.5.4, MCollective is deprecated and will be removed in a future version of Puppet agent. If you use Puppet Enterprise, consider migrating from *MCollective to Puppet orchestrator*. If you use open source Puppet, migrate MCollective agents and filters using tools like *Bolt* and PuppetDB's *Puppet Query Language*.

Step 3: Clear and regenerate certs for Puppet agents

To replace the certs on agents, you'll need to log into each agent node and do the following steps.

1. Stop the agent service. On *nix:

    ```
    sudo puppet resource service puppet ensure=stopped
    ```

    On Windows, with Administrator privileges:

    ```
    puppet resource service puppet ensure=stopped
    ```

2. Locate Puppet's *SSL directory* and delete its contents.

    The SSL directory should be at `/etc/puppetlabs/puppet/ssl` or `C:\ProgramData\PuppetLabs\puppet\etc\ssl`.

3. **Restart the agent service. On *nix:**

   ```
   sudo puppet resource service puppet ensure=running
   ```

   On Windows, with Administrator privileges:

   ```
   puppet resource service puppet ensure=running
   ```

   When the agent starts, it generates keys and requests a new certificate from the CA master.
4. **If you are not using autosigning, log in to the CA master server and sign each agent node’s certificate request.**

   To view pending requests, run:

   ```
   sudo puppet cert list
   ```

   To sign requests, run:

   ```
   sudo puppet cert sign <NAME>
   ```

   Once an agent node’s new certificate is signed, it's retrieved within a few minutes and a Puppet run starts.

Once you have regenerated all agents’ certificates, everything should be fully functional under the new CA.