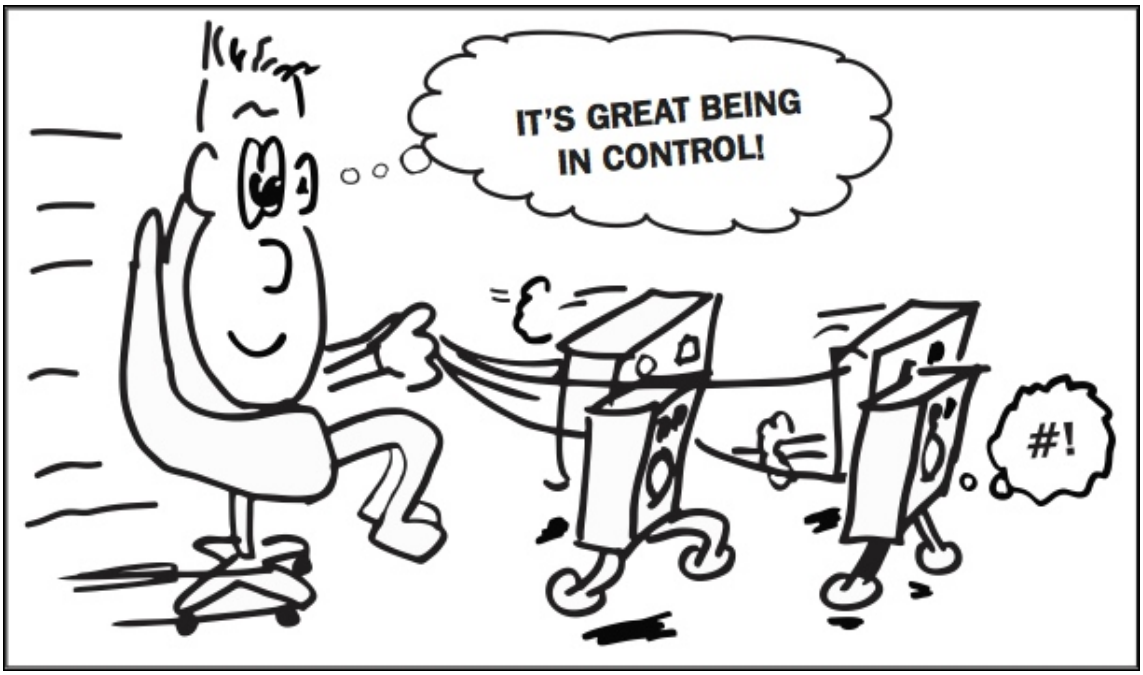
Lab 6. Managing data with Hiera

[*What you don't know can't hurt me.*]

--[[*Edward S. Marshall*]]{.attribution}

In this lab, you will learn why it's useful to separate your data and code. You will see how to set up Puppet's built-in Hiera mechanism, how to use it to store and query configuration data, including encrypted secrets such as passwords, and how to use Hiera data to create Puppet resources.



Why Hiera?

What do we mean by **configuration data**? There will be lots of pieces of information in your manifests which we can regard as configuration data: for example, the values of all your resource attributes. Look at the following example:

```
package { 'puppet-agent':
  ensure => '5.2.0-1xenial',
}
```

The preceding manifest declares that version `5.2.0-1xenial` of the `puppet-agent` package should be installed. But what happens when a new version of Puppet is released? When you want to upgrade to it, you'll have to find this code, possibly deep in multiple levels of directories, and edit it to change the desired version number.

Data needs to be maintained

Multiply this by all the packages managed throughout your manifest, and there is there's already a problem. But this is just one piece of data that needs to be maintained, and there are many more: the times of cron jobs, the email addresses for reports to be sent to, the URLs of files to fetch from the web, the parameters for monitoring checks,

the amount of memory to configure for the database server, and so on. If these values are embedded in code in hundreds of manifest files, you're setting up trouble for the future.

How can you make your config data easy to find and maintain?

Settings depend on nodes

Mixing data with code makes it harder to find and edit that data. But there's another problem. What if you have two nodes to manage with Puppet, and there's a config value which needs to be different on each of them? For example, they might both have a cron job to run the backup, but the job needs to run at a different time on each node.

How can you use different values for different nodes, without having lots of complicated logic in your manifest?

Operating systems differ

What if you have some nodes running Ubuntu 16, and some on Ubuntu 18? As you'll know if you've ever had to upgrade the operating system on a node, things change from one version to the next. For example, the name of the database server package might have changed from `mysql-server` to `mariadb-server`.

How can you find the right value to use in your manifest depending on what operating system the node is running?

The Hiera way

What we want is a kind of central database in Puppet where we can look up configuration settings. The data should be stored separately from Puppet code, and make it easy to find and edit values. It should be possible to look up values with a simple function call in Puppet code or templates. Further, we need to be able to specify different values depending on things like the hostname of the node, the operating system, or potentially anything else. We would also like to be able to enforce a particular data type for values, such as String or Boolean. The database should do all of this work for us, and just return the appropriate value to the manifest where it's needed.

Fortunately, Hiera does exactly this. Hiera lets you store your config data in simple text files (actually, YAML, JSON, or HOCON files, which use popular structured text formats), and it looks like the following example:

```
---
  test: 'This is a test'
  consul_node: true
  apache_worker_factor: 100
  apparmor_enabled: true
  ...
```

In your manifest, you query the database using the `lookup()` function, as in the following example (`lookup.pp`):

```
file { lookup('backup_path', String):
  ensure => directory,
}
```

The arguments to `lookup` are the name of the Hiera key you want to retrieve (for example `backup_path`), and the expected data type (for example `String`).

Setting up Hiera

Hiera needs to know one or two things before you can start using it, which are specified in the Hiera configuration file, named `hiera.yaml` (not to be confused this with Hiera data files, which are also YAML files, and we'll find about those later in this lab.) Each Puppet environment has its own local Hiera config file, located at the root of the

environment directory (for example, for the `production` environment, the local Hiera config file would be `/etc/puppetlabs/code/environments/production/hiera.yaml`).

Note

Hiera can also use a global config file located at `/etc/puppetlabs/puppet/hiera.yaml`, which takes precedence over the per-environment file, but the Puppet documentation recommends you only use this config layer for certain exceptional purposes, such as temporary overrides; all your normal Hiera data and configuration should live at the environment layer.

The following example shows a minimal `hiera.yaml` file (`hiera_minimal.config.yaml`):

```
---
version: 5

defaults:
  datadir: data
  data_hash: yaml_data

hierarchy:
  - name: "Common defaults"
    path: "common.yaml"
```

YAML files begin with three dashes and a newline (`---`). This is part of the YAML format, not a Hiera feature; it's the syntax indicating the start of a new YAML document.

The most important setting in the `defaults` section is `datadir`. This tells Hiera in which directory to look for its data files. Conventionally, this is in a `data/` subdirectory of the Puppet manifest directory, but you can change this if you need to.

Note

Large organizations may find it useful to manage Hiera data files separately to Puppet code, perhaps in a separate Git repo (for example, you might want to give certain people permission to edit Hiera data, but not Puppet manifests).

The `hierarchy` section is also interesting. This tells Hiera which files to read for its data and in which order. In the example only `Common defaults` is defined, telling Hiera to look for data in a file called `common.yaml`. We'll see later in this lab what else you can do with the `hierarchy` section.

Adding Hiera data to your Puppet repo

Your Vagrant VM is already set up with a suitable Hiera config and the sample data file, in the `/etc/puppetlabs/code/environments/pbg` directory. Try it now:

Run the following commands:

```
sudo puppet lookup --environment pbg test
--- This is a test
```

Note

We haven't seen the `--environment` switch before, so it's time to briefly introduce Puppet environments. A Puppet **environment** is a directory containing a Hiera config file, Hiera data, a set of Puppet manifests---in other

words, a complete, self-contained Puppet setup. Each environment lives in a named directory under `/etc/puppetlabs/code/environments`. The default environment is `production`, but you can use any environment you like by giving the `--environment` switch to the `puppet lookup` command. In the example, we are telling Puppet to use the `/etc/puppetlabs/code/environments/pbg` directory.

When you come to add Hiera data to your own Puppet environment, you can use the example `hiera.yaml` and data files as a starting point.

Troubleshooting Hiera

If you don't get the result `This is a test`, your Hiera setup is not working properly. If you see the warning `Config file not found, using Hiera defaults`, check that your Vagrant box has an `/etc/puppetlabs/code/environments/pbg` directory. If not, destroy and re-provision your Vagrant box with:

```
vagrant destroy
scripts/start_vagrant.sh
```

If you see an error like the following, it generally indicates a problem with the Hiera data file syntax:

```
Error: Evaluation Error: Error while evaluating a Function Call,
(/etc/puppetlabs/code/environments/pbg/hiera.yaml): did not find expected key while
parsing a block mapping at line 11 column 5  at line 1:8 on node ubuntu-xenial
```

If this is the case, check the syntax of your Hiera data files.

Querying Hiera

In Puppet manifests, you can use the `lookup()` function to query Hiera for the specified key (you can think of Hiera as a key-value database, where the keys are strings, and values can be any type).

In general, you can use a call to `lookup()` anywhere in your Puppet manifests you might otherwise use a literal value. The following code shows some examples of this (`lookup2.pp`):

```
notice("Apache is set to use ${lookup('apache_worker_factor', Integer)} workers")

unless lookup('apparmor_enabled', Boolean) {
  exec { 'apt-get -y remove apparmor': }
}

notice('dns_allow_query enabled: ', lookup('dns_allow_query', Boolean))
```

To apply this manifest in the example environment, run the following command:

```
sudo puppet apply --environment pbg /examples/lookup2.pp
Notice: Scope(Class[main]): Apache is set to use 100 workers
Notice: Scope(Class[main]): dns_allow_query enabled:  true
```

Typed lookups

As we've seen, `lookup()` takes a second parameter which specifies the expected type of the value to be retrieved. Although this is optional, you should always specify it, to help catch errors. If you accidentally look up the wrong key, or mistype the value in the data file, you'll get an error like this:

```
Error: Evaluation Error: Error while evaluating a Function Call, Found value has wrong type, expects a Boolean value, got String at /examples/lookup_type.pp:1:8 on node ubuntu-xenial
```

Types of Hiera data

As we've seen, Hiera data is stored in text files, structured using the format called **YAML Ain't Markup Language**, which is a common way of organizing data. Here's another snippet from our sample Hiera data file, which you'll find at `/etc/puppetlabs/code/environments/pbg/data/common.yaml` on the VM:

```
  syslog_server: '10.170.81.32'
  monitor_ips:
    - '10.179.203.46'
    - '212.100.235.160'
    - '10.181.120.77'
    - '94.236.56.148'
  cobbler_config:
    manage_dhcp: true
    pxe_just_once: true
```

There are actually three different kinds of Hiera data structures present: **single values**, **arrays**, and **hashes**. We'll examine these in detail in a moment.

Single values

Most Hiera data consists of a key associated with a single value, as in the previous example:

```
syslog_server: '10.170.81.32'
```

The value can be any legal Puppet value, such as a String, as in this case, or it can be an Integer:

```
apache_worker_factor: 100
```

Boolean values

You should specify Boolean values in Hiera as either `true` or `false`, without surrounding quotes. However, Hiera is fairly liberal in what it interprets as Boolean values: any of `true`, `on`, or `yes` (with or without quotes) are interpreted as a true value, and `false`, `off`, or `no` are interpreted as a false value. For clarity, though, stick to the following format:

```
consul_node: true
```

When you use `lookup()` to return a Boolean value in your Puppet code, you can use it as the conditional expression in, for example, an `if` statement:

```
if lookup('is_production', Boolean) {
  ...
```

Arrays

Usefully, Hiera can also store an array of values associated with a single key:

```
monitor_ips:
  - '10.179.203.46'
  - '212.100.235.160'
  - '10.181.120.77'
  - '94.236.56.148'
```

The key (`monitor_ips`) is followed by a list of values, each on its own line and preceded by a hyphen (`-`). When you call `lookup('monitor_ips', Array)` in your code, the values will be returned as a Puppet array.

Hashes

As we saw in [Lab 5], [*Variables, expressions, and facts], a hash (also called a **dictionary** in some programming languages) is like an array where each value has an identifying name (called the **key*), as in the following example:

```
cobbler_config:
  manage_dhcp: true
  pxe_just_once: true
```

Each key-value pair in the hash is listed, indented on its own line. The `cobbler_config` hash has two keys, `manage_dhcp` and `pxe_just_once` . The value associated with each of those keys is `true` .

When you call `lookup('cobbler_config', Hash)` in a manifest, the data will be returned as a Puppet hash, and you can reference individual values in it using the normal Puppet hash syntax, as we saw in [Lab 5], [*Variables, expressions, and facts*] (`lookup_hash.pp`):

```
$cobbler_config = lookup('cobbler_config', Hash)
$manage_dhcp = $cobbler_config['manage_dhcp']
$pxe_just_once = $cobbler_config['pxe_just_once']
if $pxe_just_once {
  notice('pxe_just_once is enabled')
} else {
  notice('pxe_just_once is disabled')
}
```

Since it's very common for Hiera data to be a hash of hashes, you can retrieve values from several levels down in a hash by using the following "dot notation" (`lookup_hash_dot.pp`):

```
$web_root = lookup('cms_parameters.static.web_root', String)
notice("web_root is ${web_root}")
```

Interpolation in Hiera data

Hiera data is not restricted to literal values; it can also include the value of Facter facts or Puppet variables, as in the following example:

```
 backup_path: "/backup/%{facts.hostname}"
```

Anything within the `%{}` delimiters inside a quoted string is evaluated and interpolated by Hiera. Here, we're using the dot notation to reference a value inside the `$facts` hash.

Using lookup()

Helpfully, you can also interpolate Hiera data in Hiera data, by using the `lookup()` function as part of the value. This can save you repeating the same value many times, and can make your data more readable, as in the following example (also from `hiera_sample.yaml`):

```
ips:
  home: '130.190.0.1'
  office1: '74.12.203.14'
  office2: '95.170.0.75'
firewall_allow_list:
  - "%{lookup('ips.home')}"
  - "%{lookup('ips.office1')}"
  - "%{lookup('ips.office2')}"
```

This is much more readable than simply listing a set of IP addresses with no indication of what they represent, and it prevents you accidentally introducing errors by updating a value in one place but not another. Use Hiera interpolation to make your data self-documenting.

Using alias()

When you use the `lookup()` function in a Hiera string value, the result is always a string. This is fine if you're working with string data, or if you want to interpolate a Hiera value into a string containing other text. However, if you're working with arrays, hashes, or Boolean values, you need to use the `alias()` function instead. This lets you re-use any Hiera data structure within Hiera, just by referencing its name:

```
firewall_allow_list:
  - "%{lookup('ips.home')}"
  - "%{lookup('ips.office1')}"
  - "%{lookup('ips.office2')}"
vpn_allow_list: "%{alias('firewall_allow_list')}"
```

Don't be fooled by the surrounding quotes: it may look as though `vpn_allow_list` will be a string value, but because we are using `alias()`, it will actually be an array, just like the value it is aliasing (`firewall_allow_list`).

Using literal()

Because the percent character (`%`) tells Hiera to interpolate a value, you might be wondering how to specify a literal percent sign in data. For example, Apache uses the percent sign in its configuration to refer to variable names like `%{HTTP_HOST}`. To write values like these in Hiera data, we need to use the `literal()` function, which exists only to refer to a literal percent character. For example, to write the value `%{HTTP_HOST}` as Hiera data, we would need to write:

```
%{literal('%')}{HTTP_HOST}
```

You can see a more complicated example in the sample Hiera data file:

```
force_www_rewrite:
  comment: "Force WWW"
  rewrite_cond: "%{literal('%')}{HTTP_HOST} !^www\\. [NC]"
  rewrite_rule: "^(.*)$ https://www.%{literal('%')}{HTTP_HOST}%{literal('%')}
{REQUEST_URI} [R=301,L]"
```

The hierarchy

So far, we've only used a single Hiera data source (`common.yaml`). Actually, you can have as many data sources as you like. Each usually corresponds to a YAML file, and they are listed in the `hierarchy` section of the `hiera.yaml` file, with the highest-priority source first and the lowest last:

```
hierarchy:
  ...
  - name: "Host-specific data"
    path: "nodes/%{facts.hostname}.yaml"
  - name: "OS release-specific data"
    path: "os/%{facts.os.release.major}.yaml"
  - name: "OS distro-specific data"
    path: "os/%{facts.os.distro.codename}.yaml"
  - name: "Common defaults"
    path: "common.yaml"
```

In general, though, you should keep as much data as possible in the `common.yaml` file, simply because it's easier to find and maintain data if it's in one place, rather than scattered through several files.

For example, if you have some Hiera data which is only used on the `monitor` node, you might be tempted to put it in a `nodes/monitor.yaml` file. But, unless it has to override some settings in `common.yaml`, you'll just be making it harder to find and update. Put everything in `common.yaml` that you can, and reserve other data sources only for overrides to common values.

Dealing with multiple values

You may be wondering what happens if the same key is listed in more than one Hiera data source. For example, imagine the first source contains the following:

```
consul_node: false
```

Also, assume that `common.yaml` contains:

```
consul_node: true
```

What happens when you call `lookup('consul_node', Boolean)` with this data? There are two different values for `consul_node` in two different files, so which one does Hiera return?

The answer is that Hiera searches data sources in the order they are listed in the `hierarchy` section; that is to say, in priority order. It returns the first value found, so if there are multiple values, only the value from the first---that is, highest-priority--- data source will be returned (that's the "hierarchy" part).

Merge behaviors

We said in the previous section that if there is more than one value matching the specified key, the first matching data source takes priority over the others. This is the default behavior, and this is what you'll usually want. However, sometimes you may want `lookup()` to return the union of all the matching values found, throughout the hierarchy. Hiera allows you to specify which of these strategies it should use when multiple values match your lookup.

This is called a **merge behavior**, and you can specify which merge behavior you want as the third argument to `lookup()`, after the key and data type (`lookup_merge.pp`):

```
notice(lookup('firewall_allow_list', Array, 'unique'))
```

The default merge behavior is called `first`, and it returns only one value, the first found value. By contrast, the `unique` merge behavior returns all the values found, as a flattened array, with duplicates removed (hence `unique`).

If you are looking up hash data, you can use the `hash` merge behavior to return a merged hash containing all the keys and values from all matching hashes found. If Hiera finds two hash keys with the same name, only the value of the first will be returned. This is known as a **shallow merge**. If you want a deep merge (that is, one where matching hashes will be merged at all levels, instead of just the top level) use the `deep` merge behavior.

If this all sounds a bit complicated, don't worry. The default merge behavior is probably what you want most of the time, and if you should happen to need one of the other behaviors instead, you can read more about it in the Puppet documentation.

Data sources based on facts

The hierarchy mechanism lets you set common default values for all situations (usually in `common.yaml`), but override them in specific circumstances. For example, you can set a data source in the hierarchy based on the value of a Puppet fact, such as the hostname:

```
  - name: "Host-specific data"
    path: "nodes/%{facts.hostname}.yaml"
```

Hiera will look up the value of the specified fact and search for a data file with that name in the `nodes/` directory. In the previous example, if the node's hostname is `web1`, Hiera will look for the data file `nodes/web1.yaml` in the Hiera data directory. If this file exists and contains the specified Hiera key, the `web1` node will receive that value for its lookup, while other nodes will get the default value from `common`.

Note

Note that you can organize your Hiera data files in subdirectories under the main `data/` directory if you like, such as `data/nodes/`.

Another useful fact to reference in the hierarchy is the operating system major version or codename. This is very useful when you need your manifest to work on more than one release of the operating system. If you have more than a handful of nodes, migrating to the latest OS release is usually a gradual process, upgrading one node at a time. If something has changed from one version to the next that affects your Puppet manifest, you can use the `os.distro.codename` fact to select the appropriate Hiera data, as in the following example:

```
  - name: "OS-specific data"
    path: "os/%{facts.os.distro.codename}.yaml"
```

Alternatively, you can use the `os.release.major` fact:

```
  - name: "OS-specific data"
    path: "os/%{facts.os.release.major}.yaml"
```

For example, if your node is running Ubuntu 16.04 Xenial, Hiera will look for a data file named `os/xenial.yaml` (if you're using `os.distro.codename`) or `os/16.04.yaml` (if you're using `os.release.major`) in the Hiera data directory.

For more information about facts in Puppet, see [Lab 5], [*Variables, expressions, and facts*].

What belongs in Hiera?

What data should you put in Hiera, and what should be in your Puppet manifests? A good rule of thumb about when to separate data and code is to ask yourself what might **change** in the future. For example, the exact version of a package is a good candidate for Hiera data, because it's quite likely you'll need to update it in the future.

Another characteristic of data that belongs in Hiera is that it's **specific** to your site or company. If you take your Puppet manifest and give it to someone else in another company or organization, and she has to modify any values in the code to make it work at her site, then those values should probably be in Hiera. This makes it much easier to share and re-use code; all you have to do is edit some values in Hiera.

If the same data is needed in **more than one place** in your manifests, it's also a good idea for that data to be stored in Hiera. Otherwise, you have to either repeat the data, which makes it harder to maintain, or use a global variable, which is bad style in any programming language, and especially so in Puppet.

If you have to change a data value when you apply your manifests on a different **operating system**, that's also a candidate for Hiera data. As we've seen in this lab, you can use the hierarchy to select the correct value based on facts, such as the operating system or version.

One other kind of data that belongs in Hiera is parameter values for classes and modules; we'll see more about that in [Lab 7], [*Mastering modules*].

Creating resources with Hiera data

When we started working with Puppet, we created resources directly in the manifest using literal attribute values. In this lab, we've seen how to use Hiera data to fill in the title and attributes of resources in the manifest. We can now take this idea one step further and create resources **directly from Hiera** queries. The advantage of this method is that we can create any number of resources of any type, based purely on data.

Building resources from Hiera arrays

In [Lab 5], [*Variables, expressions, and facts*], we learned how to use Puppet's `each` function to iterate over an array or hash, creating resources as we go. Let's apply this technique to some Hiera data. In our first example, we'll create some user resources from a Hiera array.

Run the following command:

```
sudo puppet apply --environment pbg /examples/hiera_users.pp
Notice: /Stage[main]/Main/User[katy]/ensure: created
Notice: /Stage[main]/Main/User[lark]/ensure: created
Notice: /Stage[main]/Main/User[bridget]/ensure: created
Notice: /Stage[main]/Main/User[hsing-hui]/ensure: created
Notice: /Stage[main]/Main/User[charles]/ensure: created
```

Here's the data we're using (from the `/etc/puppetlabs/code/environments/pbg/data/common.yaml` file):

```
users:
  - 'katy'
  - 'lark'
  - 'bridget'
  - 'hsing-hui'
  - 'charles'
```

And here's the code which reads it and creates the corresponding user instances (`hiera_users.pp`):

```
lookup('users', Array[String]).each | String $username | {
  user { $username:
    ensure => present,
  }
}
```

Combining Hiera data with resource iteration is a powerful idea. This short manifest could manage all the users in your infrastructure, without you ever having to edit the Puppet code to make changes. To add new users, you need only edit the Hiera data.

Building resources from Hiera hashes

Of course, real life is never quite as simple as a programming language example. If you were really managing users with Hiera data in this way, you'd need to include more data than just their names: you'd need to be able to manage shells, UIDs, and so on, and you'd also need to be able to remove the users if necessary. To do that, we will need to add some structure to the Hiera data.

Run the following command:

```
sudo puppet apply --environment pbg /examples/hiera_users2.pp
Notice: Compiled catalog for ubuntu-xenial in environment pbg in 0.05 seconds
Notice: /Stage[main]/Main/User[katy]/uid: uid changed 1001 to 1900
Notice: /Stage[main]/Main/User[katy]/shell: shell changed '' to '/bin/bash'
Notice: /Stage[main]/Main/User[lark]/uid: uid changed 1002 to 1901
Notice: /Stage[main]/Main/User[lark]/shell: shell changed '' to '/bin/sh'
Notice: /Stage[main]/Main/User[bridget]/uid: uid changed 1003 to 1902
Notice: /Stage[main]/Main/User[bridget]/shell: shell changed '' to '/bin/bash'
Notice: /Stage[main]/Main/User[hsing-hui]/uid: uid changed 1004 to 1903
Notice: /Stage[main]/Main/User[hsing-hui]/shell: shell changed '' to '/bin/sh'
Notice: /Stage[main]/Main/User[charles]/uid: uid changed 1005 to 1904
Notice: /Stage[main]/Main/User[charles]/shell: shell changed '' to '/bin/bash'
Notice: Applied catalog in 0.17 seconds
```

The first difference from the previous example is that instead of the data being a simple array, it's a hash of hashes:

```
users2:
  'katy':
    ensure: present
    uid: 1900
    shell: '/bin/bash'
  'lark':
    ensure: present
    uid: 1901
    shell: '/bin/sh'
  'bridget':
    ensure: present
    uid: 1902
    shell: '/bin/bash'
  'hsing-hui':
    ensure: present
    uid: 1903
    shell: '/bin/sh'
  'charles':
```

```
    ensure: present
    uid: 1904
    shell: '/bin/bash'
```

Here's the code which processes that data (`hiera_users2.pp`):

```
lookup('users2', Hash, 'hash').each | String $username, Hash $attrs | {
  user { $username:
    * => $attrs,
  }
}
```

Each of the keys in the `users2` hash is a username, and each value is a hash of user attributes such as `uid` and `shell` .

When we call `each` on this hash, we specify two parameters to the loop instead of one:

```
| String $username, Hash $attrs |
```

As we saw in [Lab 5], [*Variables, expressions, and facts*], when iterating over a hash, these two parameters receive the hash key and its value, respectively.

Inside the loop, we create a user resource for each element of the hash:

```
user { $username:
  * => $attrs,
}
```

You may recall from the previous lab that the `*` operator (the attribute splat operator) tells Puppet to treat `$attrs` as a hash of attribute-value pairs. So the first time round the loop, with user `katy` , Puppet will create a user resource equivalent to the following manifest:

```
user { 'katy':
  ensure => present,
  uid    => 1900,
  shell  => '/bin/bash',
}
```

Every time we go round the loop with the next element of `users` , Puppet will create another user resource with the specified attributes.

The advantages of managing resources with Hiera data

The previous example makes it easy to manage users across your network without having to edit Puppet code: if you want to remove a user, for example, you would simply change her `ensure` attribute in the Hiera data to `absent` . Although each of the users happens to have the same set of attributes specified, this isn't essential; you could add any attribute supported by the Puppet `user` resource to any user in the data. Also, if there's an attribute whose value is always the same for all users, you need not list it in the Hiera data for every user. You can add it as a literal attribute value of the `user` resource inside the loop, and thus every user will have it.

This makes it easier to add and update users on a routine basis, but there are other advantages too: for example, you could write a simple web application which allowed HR staff to add or edit users using a browser interface, and it would only need to output a YAML file with the required data. This is much easier and more robust than trying to

generate Puppet code automatically. Even better, you could pull user data from an LDAP or **Active Directory** (**AD**) server and put it into Hiera YAML format for input into this manifest.

This is a very powerful and flexible technique, and of course you can use it to manage any kind of Puppet resource: files, packages, Apache virtual hosts, MySQL databases---anything you can do with a resource you can do with Hiera data and `each`. You can also use Hiera's override mechanism to create different sets of resources for different nodes, roles, or operating systems.

However, you shouldn't over-use this technique. Creating resources from Hiera data adds a layer of abstraction which makes it harder to understand the code for anyone trying to read or maintain it. With Hiera, it can also be difficult to work out from inspection exactly what data the node will get in a given set of circumstances. Keep your hierarchy as simple as possible, and reserve the data-driven resources trick for situations where you have a large and variable number of resources which you need to update frequently. In [Lab 11], [*Orchestrating cloud resources*], we'll see how to use the same technique to manage cloud instances, for example.

Managing secret data

Puppet often needs to know your secrets; for example, passwords, private keys, and other credentials need to be configured on the node, and Puppet must have access to this information. The problem is how to make sure that no-one else does. If you are checking this data into a Git repo, it will be available to anybody who has access to the repo, and if it's a public GitHub repo, everybody in the world can see it.

Clearly, it's essential to be able to encrypt secret data in such a way that Puppet can decrypt it on individual nodes where it's needed, but it's indecipherable to anybody who does not have the key. The popular GnuPG encryption tool is a good choice for this. It lets you encrypt data using a public key which can be distributed widely, but only someone with the corresponding private key can decrypt the information.

Hiera has a pluggable **backend** system which allows it to support various different ways of storing data. One such backend is called `hiera-eyaml-gpg`, which allows Hiera to use a GnuPG-encrypted data store. Rather than encrypting a whole data file, `hiera-eyaml-gpg` lets you mix encrypted and plaintext data in the same YAML file. That way, even someone who doesn't have the private key can still edit and update the plaintext values in Hiera data files, although the encrypted data values will be unreadable to them.

Setting up GnuPG

First, we'll need to install GnuPG and create a key pair for use with Hiera. The following instructions will help you do this:

1. Run the following command:

```
sudo apt-get install gnupg rng-tools
```

2. Once GnuPG is installed, run the following command to generate a new key pair:

```
gpg --gen-key
```

3. When prompted, select the RSA and RSA key type:

```
Please select what kind of key you want:
   (1) RSA and RSA (default)
   (2) DSA and Elgamal
   (3) DSA (sign only)
   (4) RSA (sign only)
Your selection? 1
```

4. Select a 2,048 bit key size:

```
RSA keys may be between 1024 and 4096 bits long.
What keysize do you want? (2048) 2048
```

5. Enter `0` for the key expiry time:

```
Key is valid for? (0) 0
Key does not expire at all
Is this correct? (y/N) y
```

6. When prompted for a real name, email address, and comment for the key, enter whatever is appropriate for your site:

```
Real name: Puppet
Email address: puppet@cat-pictures.com
Comment:
You selected this USER-ID:
    "Puppet <puppet@cat-pictures.com>"

Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit? o
```

7. When prompted for a passphrase, just hit [*Enter*] (the key can't have a passphrase, because Puppet won't be able to supply it).

It may take a few moments to generate the key, but once this is complete, GnuPG will print out the key fingerprint and details (yours will look different):

```
pub   2048R/40486112 2016-09-30
      Key fingerprint = 6758 6CEE D221 7AA0 8369  FF3A FEC1 0055 4048 6112
uid                  Puppet <puppet@cat-pictures.com>
sub   2048R/472954EB 2016-09-30
```

This key is now stored in your GnuPG keyring, and Hiera will be able to use it to encrypt and decrypt your secret data on this node. We'll see later in the lab how to distribute this key to other nodes managed by Puppet.

Adding an encrypted Hiera source

A Hiera source using GPG-encrypted data needs a couple of extra parameters. Here's the relevant section from the example `hiera.yaml` file:

```
  - name: "Secret data (encrypted)"
    lookup_key: eyaml_lookup_key
    path: "secret.eyaml"
    options:
      gpg_gnupghome: '/home/ubuntu/.gnupg'
```

As with normal data sources, we a have `name` and a `path` to the data file, but we also need to specify the `lookup_key` function, which in this case is `eyaml_lookup_key`, and set `options['gpg_gnupghome']` to point to the GnuPG directory, where the decryption key lives.

Creating an encrypted secret

You're now ready to add some secret data to your Hiera store.

1. Create a new empty Hiera data file with the following commands:

```
cd /etc/puppetlabs/code/environments/pbg
sudo touch data/secret.eyaml
```

2. Run the following command to edit the data file using the `eyaml` editor (which automatically encrypts the data for you when you save it). Instead of `puppet@cat-pictures.com`, use the email address that you entered when you created your GPG key.

```
sudo /opt/puppetlabs/puppet/bin/eyaml edit --gpg-always-trust --gpg-
recipients=puppet@cat-pictures.com data/secret.eyaml
```

3. If the system prompts you to select your default editor, choose the editor you prefer. If you're familiar with Vim, I recommend you choose that, but otherwise, you will probably find `nano` the easiest option. (You should learn Vim, but that's a subject for another course.)

4. Your selected editor will be started with the following text already inserted in the file:

```
#| This is eyaml edit mode. This text (lines starting with #| at the top of the
#| file) will be removed when you save and exit.
#|  - To edit encrypted values, change the content of the DEC(<num>)::PKCS7[]!
#|    block (or DEC(<num>)::GPG[]!).
#|    WARNING: DO NOT change the number in the parentheses.
#|  - To add a new encrypted value copy and paste a new block from the
#|    appropriate example below. Note that:
#|     * the text to encrypt goes in the square brackets
#|     * ensure you include the exclamation mark when you copy and paste
#|     * you must not include a number when adding a new block
#|    e.g. DEC::PKCS7[]! -or- DEC::GPG[]!
```

5. Enter the following text below the commented message, exactly as shown, including the beginning three hyphens:

```
---
  test_secret: DEC::GPG[This is a test secret]!
```

6. Save the file and exit the editor.

7. Run the following command to test that Puppet can read and decrypt your secret:

```
sudo puppet lookup --environment pbg test_secret
--- This is a test secret
```

How Hiera decrypts secrets

To prove to yourself that the secret data is actually encrypted, run the following command to see what it looks like in the data file on disk:

```
cat data/secret.eyaml
---
  test_secret:
ENC[GPG,hQEMA4+8DyxHKVTrAQf/QQPL4zD2kkU7T+FhaEdptu68RAw2m2KAXGujjnQPXoONrbh1QjtzZiJBlhq(
```

Of course, the actual ciphertext will be different for you, since you're using a different encryption key. The point is, though, the message is completely scrambled. GnuPG's encryption algorithms are extremely strong; even using every computer on Earth simultaneously, it would take (on average) many times the current age of the Universe to unscramble data encrypted with a 2,048-bit key. (Or, to put it a different way, the chances of decrypting the data within a reasonable amount of time are many billions to one.)

When you reference a Hiera key such as `test_secret` in your manifest, what happens next? Hiera consults its list of data sources configured in `hiera.yaml`. The first source in the hierarchy is `secret.eyaml`, which contains the key we're interested in (`test_secret`). Here's the value:

```
ENC[GPG,hQEMA4 … EEU4cw==]
```

The `ENC` tells Hiera that this is an encrypted value, and the `GPG` identifies which type of encryption is being used (`hiera-eyaml` supports several encryption methods, of which GPG is one). Hiera calls the GPG subsystem to process the encrypted data, and GPG searches the keyring to find the appropriate decryption key. Assuming it finds the key, GPG decrypts the data and passes the result back to Hiera, which returns it to Puppet, and the result is the plaintext:

```
This is a test secret
```

The beauty of the system is that all of this complexity is hidden from you; all you have to do is call the function `lookup('test_secret', String)` in your manifest, and you get the answer.

Editing or adding encrypted secrets

If the secret data is stored in encrypted form, you might be wondering how to edit it when you want to change the secret value. Fortunately, there's a way to do this. Recall that when you first entered the secret data, you used the following command:

```
sudo /opt/puppetlabs/puppet/bin/eyaml edit --gpg-always-trust --gpg-
recipients=puppet@cat-pictures.com data/secret.eyaml
```

If you run the same command again, you'll find that you're looking at your original plaintext (along with some explanatory comments):

```
---
  test_secret: DEC(1)::GPG[This is a test secret]!
```

You can edit the `This is a test secret` string (make sure to leave everything else exactly as it is, including the `DEC::GPG[]!` delimiters). When you save the file and close the editor, the data will be re-encrypted using your key, if it has changed.

Don't remove the `(1)` in parentheses after `DEC`; it tells Hiera that this is an existing secret, not a new one. As you add more secrets to this file, they will be identified with increasing numbers.

For convenience of editing, I suggest you make a shell script, called something like `/usr/local/bin/eyaml_edit`, which runs the `eyaml edit` command. There's an example on your Vagrant

box, at `/examples/eyaml_edit.sh`, which you can copy to `/usr/local/bin` and edit (as before, substitute the `gpg-recipients` email address with the one associated with your GPG key):

```
#!/bin/bash
/opt/puppetlabs/puppet/bin/eyaml edit --gpg-always-trust --gpg-recipients=puppet@cat-pictures.com /etc/puppetlabs/code/environments/pbg/data/secret.eyaml
```

Now, whenever you need to edit your secret data, you can simply run the following command:

```
sudo eyaml_edit
```

To add a new secret, add a line like this:

```
  new_secret: DEC::GPG[Somebody wake up Hicks]!
```

When you save and quit the editor, the newly-encrypted secret will be stored in the data file.

Distributing the decryption key

Now that your Puppet manifests use encrypted Hiera data, you'll need to make sure that each node running Puppet has a copy of the decryption key. Export the key to a text file using the following command (use your key's email address, of course):

```
sudo sh -c 'gpg --export-secret-key -a puppet@cat-pictures.com >key.txt'
```

Copy the `key.txt` file to any nodes which need the key, and run the following command to import it:

```
sudo gpg --import key.txt
sudo rm key.txt
```

Make sure that you delete all copies of the text file once you have imported the key.

Note

Important note

Because all Puppet nodes have a copy of the decryption key, this method only protects your secret data from someone who does not have access to the nodes. It is still considerably better than putting secret data in your manifests in plaintext, but it has the disadvantage that someone with access to a node can decrypt, modify, and re-encrypt the secret data. For improved security you should use a secrets management system where the node does not have the key, and Puppet has read-only access to secrets. Some options here include Vault, from Hashicorp, and Summon, from Conjur.

Summary

In this lab we've outlined some of the problems with maintaining configuration data in Puppet manifests, and introduced Hiera as a powerful solution. We've seen how to configure Puppet to use the Hiera data store, and how to query Hiera keys in Puppet manifests using `lookup()`.

We've looked at how to write Hiera data sources, including string, array, and hash data structures, and how to interpolate values into Hiera strings using `lookup()`, including Puppet facts and other Hiera data, and how to duplicate Hiera data structures using `alias()`. We've learned how Hiera's hierarchy works, and how to configure it using the `hiera.yaml` file.

We've seen how our example Puppet infrastructure is configured to use Hiera data, and demonstrated the process by looking up a data value in a Puppet manifest. In case of problems, we also looked at some common Hiera errors, and we've discussed rules of thumb about when to put data into Hiera.

We've explored using Hiera data to create resources, using an `each` loop over an array or hash. Finally, we've covered using encrypted data with Hiera, using the `hiera-eyaml-gpg` backend, and we've seen how to create a GnuPG key and use it to encrypt a secret value, and retrieve it again via Puppet. We've explored the process Hiera uses to find and decrypt secret data, developed a simple script to make it easy to edit encrypted data files, and outlined a basic way to distribute the decryption key to multiple nodes.

In the next lab, we'll look at how to find and use public modules from Puppet Forge; how to use public modules to manage software including Apache, MySQL, and archive files; how to use the `r10k` tool to deploy and manage third-party modules; and how to write and structure your own modules.