Mastering SaltStack

This book will help you bring forward capabilities that will help you excel in the management of your servers.

You will be taken through the mind of the modern systems engineer, and discover how they use Salt to manage their infrastructures. The inner workings of Salt will be explored, so that as you advance your knowledge of Salt, you will be able to swim with the current, rather than against it.

Various subsystems of Salt such as Salt SSH, Salt Cloud, and job caches are explained in detail.

You will be taken through an in-depth discussion about how to effectively scale Salt to manage thousands of machines, and how to troubleshoot issues.

You will also be taken through an overview of RAET, Salt's new transport protocol, and given an insight into how this technology improves Salt.

Who this book is written for

This book is ideal for professionals who have been managing groups of servers, and want to learn how to expand their toolset. This book will also explain some of the more advanced features of Salt, and explore how to use them to bring additional power to the fundamentals that the professionals have already been using.

What you will learn from this book

- Learn how the pros are managing their infrastructures, and what techniques they use to keep everything running smoothly with Salt
- Understand what makes Salt tick, and how that affects the way you use it
- Take a look at familiar features in a new light, so that you have a better handle on how to approach tasks
- Use Salt SSH to manage the servers that don't fit Salt's traditional use cases
- Besides automating your configuration, you will be able to automate your servers, and make them behave more intelligently
- Make better use of cloud-based services, including compute clouds such as EC2, Azure and OpenStack
- Learn about the new RAET protocol, and how it changes the way automation works

Foreword by Thomas S. Hatch - Founder and CTO, SaltStack


Free Sample
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 3 'Exploring Salt SSH'
- A synopsis of the book’s content
- More information on Mastering SaltStack
About the Author

Joseph Hall has touched just about every area of the modern technology world from his beginnings as a support technician to being a web programmer, QA engineer, systems administrator, Linux instructor, and cloud engineer. He is currently a senior cloud and integrations engineer at SaltStack. Joseph enjoys working with some of the best minds in the business with his coworkers and SaltStack's partners. He is also a classically trained chef. Joseph's supreme pleasure lies in working with what he calls computational gastronomy.
Preface

I'm very excited to have been given the chance to put this book together. From an idea in the brain of Tom Hatch to an award-winning open source project to the flagship product of an award-winning open source company, I've been given the rare opportunity to watch Salt grow. Salt has become an incredibly powerful framework, which I wish I had access to years ago.

Everyday, I learn something new about Salt. This book is a collection of a number of these things, which is aimed at the advanced user. Don't see it as the last word on any of the topics it covers. Instead, see it as a guide to using this tool to its fullest potential on your journey.

As you read through this book, I hope that the ideas and examples in it inspire you to update and innovate your infrastructure.

What this book covers

Chapter 1, Reviewing a Few Essentials, talks about how to review a few fundamental concepts to get into the right frame of mind. While many of the concepts should be familiar to the experienced user, you are likely to find plenty of new information as well.

Chapter 2, Diving into Salt Internals, jumps into the deeper workings behind Salt. It discusses the internal configuration, the loader system, renderers, and the state compiler.

Chapter 3, Exploring Salt SSH, explores how Salt SSH is a powerful tool. It's been getting a lot of love from the core developers lately. This is possibly the most complete discussion of Salt SSH.

Chapter 4, Managing Tasks Asynchronously, discusses how one of the most important concepts behind Salt is asynchronicity. This chapter lays down the fundamentals that will be referenced throughout the rest of the book.
Preface

Chapter 5, *Taking Salt Cloud to the Next Level*, goes deeper, exposing parts of Salt Cloud, which turn casual users into experts. No matter how much you've used Salt Cloud, there's a good chance you've only scratched the surface.

Chapter 6, *Using Salt with REST*, talks about how it's almost impossible to work with technology these days without depending on REST services. It uses Salt to tie these services to your infrastructure with ease.

Chapter 7, *Understanding the RAET Protocol*, teaches you the concepts behind RAET and how they impact upon you. RAET is still new, but it's already found its way into large organizations.

Chapter 8, *Strategies for Scaling*, talks about how to never assume that your infrastructure will stay small. It makes you think about how to scale your infrastructure properly before it's too late.

Chapter 9, *Monitoring with Salt*, discovers how Salt is a powerful monitoring tool if you know how to use it. It tells you how to integrate it with existing tools or use Salt alone.

Chapter 10, *Exploring Best Practices*, discusses the good and bad ways to use any tool. It teaches you the good ways to use Salt.

Chapter 11, *Troubleshooting Problems*, tells you where to look and how to find help when things go wrong.
Exploring Salt SSH

Salt introduced the powerful concept of using message queues as a communication transport mechanism. There are times when the old tools just make sense. However, there's no reason not to give them a kick in the seat of their pants when necessary. This is why Salt SSH was created. In this chapter, we'll cover the following topics:

- Using rosters
- Building dynamic rosters
- Understanding the salt-thin agent
- Using the raw SSH mode

Grappling with SSH

SSH is in fact based on very different concepts than the primary architecture of Salt. Salt was designed to communicate with large numbers of remote machines at once; SSH was designed to interact with only one at a time. Let's take a few minutes to examine some of the differences between Salt and SSH.

Remote shells

Let's take a step back in time to when the Internet wasn't around yet and the ARPANET was brand new. To accompany this new concept to nationally and globally-interconnected networks, a series of new protocols were introduced. Telnet, a communication mechanism to take advantage of them, was also introduced. Internet protocols were based on telnet, including a remote shell.
As security needs grew, so did the need to secure telnet. SSH was born; eventually, the OpenSSH project was broadly shipped and supported by a number of Unix-based platforms. While SSH means Secure Shell, it was in fact designed to secure tunnel applications that had traditionally communicated with telnet. The default application was a shell, replacing traditional telnet and its kin, but many, many more possibilities also existed.

With all of this in mind, it makes sense that developers and administrators would be so used to their shell-based remote administration tools. SSH offers security, a reasonably (but not completely) consistent platform between remote systems, and a familiar environment to work in. It was never designed to handle communications between multiple machines.

A number of solutions were available to address this situation. SSH password agents and passwordless SSH keys, along with the power of shell scripts, comprised the bulk of the solutions for years. One particular tool called ClusterSSH allowed multiple login windows to accept input from a single location and relay it across all connections. Also, before long, the entire remote execution platforms built-in SSH will be introduced.

Salt SSH was not the first of these. It was released by SaltStack to accommodate the needs of some of their users, who enjoyed the principles behind the Salt framework but had a need to use SSH in an automated fashion to manage some of their systems.

### Using rosters

Salt was originally designed to operate without the traditional database that many of its forefathers used to store remote system configuration. As its message bus could retrieve information directly from remote machines, often faster than a database lookup, the need for a database was minimalized.

As Minions connect to the Master, and not the other way around, in a traditional Salt infrastructure, the Master did not even have a need to store the network and host configuration for the Minions. The game changes when dealing with SSH-based connections because the Master necessarily connects to its minions via SSH.

Rosters were introduced as a means for Salt SSH to keep track of the host information. The default roster, which uses flat text files, is enough to get the job done. More dynamic rosters add vast depths of power.
The flat roster

As its name suggests, this roster uses a flat file. This is normally stored as `/etc/salt/roster`, but can be changed with the `-roster-file` option:

```
# salt-ssh --roster-file=/etc/salt/altroster myminion test.ping
```

At its most basic level, this file needs to contain only two pieces of information: the name of a Minion and a network address (IP or hostname) through which this Minion is reached:

(In `/etc/salt/roster`)

```
dufresne: 10.0.0.50
adria: 10.0.19.80
achatz: 10.0.200.5
blumenthal: 10.0.19.95
```

However, more information can be added as required:

(In `/etc/salt/roster`)

```
dufresne:
    host: 10.0.200.3
    user: wd50
adria:
    host: 10.0.19.80
    passwd: bulli
achatz:
    host: 10.0.200.5
    priv: /root/alinea.pem
blumenthal:
    host: 10.0.19.95
    sudo: True
```

As of version 2015.5, the options supported in a flat roster file are:

**host**

This can be an IP address or a hostname address. It should contain only the address and no other information, such as protocol, port, and so on.

**port**

This is normally Port 22, the SSH default. In nonstandard installations, this value can be changed as appropriate.
Exploring Salt SSH

**user**
This will default to the user running the `salt-ssh` client, which is normally `root`. As system administration tasks are often carried out using the `root` user, this is generally okay. If the username differs, add this field.

If there is a need to run different tasks as different users on the same machine, it may be useful to create separate roster entries for each user.

**passwd**
If using password authentication, then this is the password to use. This is normally discouraged because this file is plain text and viewable by anyone with appropriate permissions. If passwords are unavoidable in a roster file, then the read permissions on the file should be restricted to the very least.

This option can be avoided by specifying the password from the command line with the `--passwd` option:

```
# salt-ssh --passwd=verybadpass myminion test.ping
```

Alternately, Salt SSH can prompt the user for the password, eliminating the need for a plain text password to ever appear on screen:

```
# salt-ssh --askpass myminion test.ping
```

This should only be required on the first execution. Salt will ask whether or not it should set up its own access key for future commands (see the priv option). If allowed, subsequent commands will just use Salt's own SSH key instead of the user password.

**sudo**
In a situation where a privileged command must be performed by an unprivileged user, the sudo option should be set to True. The default is False. As of version 2015.5, the user specified must be set to not require a password. This can be accomplished by editing the sudoers file (normally, `/etc/sudoers` is editable with the visudo command) and adding the NOPASSWD flag to a user's entry:

```
heston ALL=(ALL) NOPASSWD: ALL
```

**priv**
In a situation where a private key is required to access a Minion, this will specify the path to a user-defined private key. If no such key is defined, then Salt will create one. Usually, the default location is `/etc/salt/pki/ssh/salt-ssh.rsa`. 
timeout
This is the number of seconds to wait for an SSH connection to be established. The default is 60.

thin_dir
This is the directory on the target Minion in which Salt's thin agent will be installed. This agent is discussed in more detail later in the chapter, in the Understanding the salt-thin agent section.

Other built-in rosters
A number of other rosters ship with Salt, which allows a much more dynamic means of identifying hosts and their connection parameters. To use a different roster than the standard flat file, add the --roster option to salt-ssh:

```
# salt-ssh --roster=cloud myminion test.ping
```

As of version 2015.5, the following rosters also ship with Salt.

scan
This was the first dynamic roster to ship with Salt SSH. It directs the client to attempt to log in to a range of IP addresses in order to issue the requested command. This roster is unique in that it does not make use of a Minion ID; the IP address that is generated in the scan is used, instead.

The following command will scan a small subnet and return True for each IP address that is able to answer:

```
# salt-ssh --roster=scan 10.0.0.0/24 test.ping
```

There are some considerations that should be kept in mind when working with the scan roster. First of all, all the connection information (aside from the IP address) needs to be the same across all hosts. The exception to this is SSH keys that have already been established and stored in an SSH key agent, if applicable.

However, there are security concerns with using preexisting SSH keys. Consider a scenario where you have deployed your public key across your entire infrastructure. Believing your network to be secure, you assume that any machine that is accessible via your key belongs to you and can be trusted. However, your key, public as it is, is acquired by an attacker in your network, who proceeds to set up a bogus Minion with it. As you issue what you believe to be secure commands using the scan roster, which may include sensitive data, their Minion is busy collecting this data.
Exploring Salt SSH

This is not an attack vector unique to Salt SSH. This attack was in use long before automated SSH tools hit the market; users have been falling prey to it for years. Rather than using the scan roster to issue sensitive commands, it should be used only for network discovery.

There are two specific options that can be used with the scan roster. The \texttt{--scan-ports} option accepts a comma-separated list of ports to attempt to log in to the target minion. The default is 22. As this may be seen as a form of port scanning in some organizations, be sure to check your security policy before using this option. The \texttt{--scan-timeout} option can also specify a time-out for the scanning process. The default is 0.01, which translates to 10ms.

\textbf{cache}

While Salt was not initially designed to use a database, some optimizations have since been added that accomplish many of the same goals. In particular, the Grains for each Minion are cached by default on the Master. As IPv4 addresses are among the data stored with the Grains, the cache roster takes advantage of it to obtain IP addresses for Minions that have already been accessed via another transport mechanism, such as Salt's ZMQ-based transport.

This can be useful when troubleshooting a machine on which the \texttt{salt-minion} client was previously installed, but is now no longer responding. So long as the IP address has not changed, the cache roster can look it up with the following command:

\begin{verbatim}
# salt-ssh --roster cache myminion service.start salt-minion
\end{verbatim}

As of version 2015.5, the limitations for this roster are similar to the scan roster. The user will default to whichever user is issuing the \texttt{salt-ssh} command. Also, if SSH keys are not established or specified with \texttt{--priv}, passwords must be supplied with either \texttt{--passwd} or \texttt{--askpass}.

The cache roster only supports IPv4 addresses. However, it can differentiate between local (127.0.0.0/8), private (10.0.0.0/8, 172.16.0.0/12, and 192.168.0.0/16), and public (everything else) addresses. It can also be configured to prefer one type over another. This is done with the \texttt{roster_order} option in the master configuration. The default is:

\begin{verbatim}
roster_order:
 - public
 - private
 - local
\end{verbatim}
cloud

The cloud roster is similar to the cache roster, but there are some differences. As with Salt, Salt Cloud caches information by default about Minions that it creates. The difference is that minions created with Salt Cloud don’t need to have a connection established with the Salt Master in order to have their IP address cached; since the host name was required to run the deploy process in the first place, Salt Cloud already knows what it is and caches it.

Keep in mind, that if a minion that was created with Salt Cloud reboots, then the IP address will likely change, unless it is specifically configured not to. However, issuing a full query (covered in Chapter 5, Taking Salt Cloud to the Next Level) will refresh this data. Therefore, the cloud roster may be able to obtain the IP information that is otherwise not accessible to the cache roster.

# salt-ssh --roster cloud myminion service.start salt-minion

The cloud roster also has the ability to borrow authentication information (SSH keys, passwords, and so on) from the provider or profile configuration that was used to create the Minion. So long as this information has not changed, it should not be necessary to specify that information again.

Like the cache roster, the cloud roster supports the roster_order option in the Master configuration with the same defaults.

ansible

A notable SSH automation platform is the Ansible product. This program has seen widespread adoption (especially among developers) because of its ease of use and its abundant suite of tools. Many users of Ansible have found a need to use both Salt and SSH to manage their machines. Other users have decided to switch altogether.

Rather than rosters, Ansible uses inventories to maintain the host information. The ansible roster allows Salt SSH to natively use Ansible inventories instead of roster files to obtain the host information.

As a path to an Ansible inventory must be specified, the --roster-file option is used in conjunction with this roster:

# salt-ssh --roster ansible --roster-file /etc/salt/hosts myminion test.ping
Building dynamic rosters

There is no reason to restrict yourself to the rosters that ship with Salt. As with Pillars, any external source that can provide the appropriate data can be used. In the case of rosters, the appropriate data should look like the data that is stored in flat roster files.

Consider the following entry from a flat roster file:

```yaml
myminion:
  host: 10.0.11.38
  user: larry
  password: 5700g3z43v4r
```

If you have a data source that provides this data, then you can plug the data in it. In fact, if you have a command that outputs this data, for instance in YAML, then you can easily write a roster that wraps this command.

```python
import yaml
def targets(tgt, tgt_type='glob', **kwargs):
    return yaml.safe_load(__salt__['cmd.run']('cat /etc/salt/roster'))
```

This is almost identical to the code used in the cmd_yaml external Pillar, but it can be adapted for use with rosters. Even if you don't know Python, the preceding code can easily be changed to wrap your own custom commands, even those written in a different language.

Using Salt SSH

We've spent some time discussing how to configure Minions using rosters. Let's take a few minutes to discuss some basic usage.

The `salt-ssh` command is very similar to the `salt` command in usage. A target is provided, followed by a module and function, which is optionally followed by any arguments for the function. Targets can be specified in the same way as with the `salt` command, although not as many target types are supported. As of version 2015.5, the following target types are supported by Salt SSH:

- Glob (the default)
- Perl Compatible Regular Expression (-E or --pcre)
- List (-L or --list)
- Grain (-G or --grain)
- Nodegroup (-N or --nodegroup)
- Range (-R or --range)
Using a target type is the same as with the salt command:

```
# salt-ssh -G 'os:Ubuntu' test.ping
```

All the outputters available in the salt command are also available and accessed the same way. The command to access the outputters is as follows:

```
# salt-ssh myminion grains.items --out json
```

Other options are unique to Salt SSH. For instance, in order to target by Grain, the Master needs to have a copy of the Minion's Grain data in its cache. This can be done using the --refresh flag:

```
# salt-ssh --refresh myminion test.ping
```

### Using a Saltfile

If a number of options are commonly used with Salt SSH, it can become cumbersome to type all of them. Fortunately, a Saltfile can be used to automate adding these options. This is a YAML file that can contain any number of options that are normally passed on the command line. The following is a snippet from the Saltfile:

```
salt-ssh:
  max_procs: 15
  wipe_ssh: True
```

This file is normally called Saltfile. If it is present in the current working directory when the salt-ssh command is issued, it will be used. The salt-ssh command can also point directly to a Saltfile in another directory, as shown in the following code:

```
# salt-ssh --saltfile=/etc/salt/Saltfile myminion test.ping
```

If you have a global Saltfile that you want to use everywhere, you can create a shortcut that as well with an alias (if your shell supports it):

```
# alias salt-ssh='salt-ssh --saltfile=/etc/salt/Saltfile'
```

You can also set an environment variable called SALT_SALTFILE:

```
# export SALT_SALTFILE=/etc/salt/Saltfile
```

The following options and their command-line equivalents are available in a Salt SSH Saltfile:

- `raw_shell (-r, --raw, --raw_shell)`
- `roster (-roster)`
- `roster_file (-roster-file)`
Exploring Salt SSH

- refresh_cache (--refresh, --refresh-cache)
- max_procs (--max-procs)
- extra_fileres --extra-fileres
- wipe_ssh (--w, --wipe)
- ssh_priv (--priv)
- ignore_host_keys (-i, --ignore-host-keys)
- ssh_user (--user)
- ssh_passwd (--passwd)
- ssh_askpass (--askpass)
- ssh_key_deploy (--key-deploy)
- ssh_scan_ports (--scan-ports)
- ssh_scan_timeout (--scan-timeout)

Salt versus Salt SSH

In its default mode, Salt SSH is designed to behave (as far as the user is concerned) exactly like the salt command. Minions can be targeted just like the salt command, modules are made available exactly the same way across Minions, arguments are specified the same way, and output is displayed in exactly the same way. However, there are some differences.

Architecture

The primary difference between the standard salt command and the salt-ssh command is how the communication is architected based on the transport mechanism. The salt command uses a message queue, whereas the salt-ssh command (of course) uses SSH.

Think of the message queue like a television station. All of your Minions are watching the television and waiting for instruction. As tasks are issued, they will be broadcast to the Minions along with information on who should perform them. When a Minion finishes a task, it will return the result to the master along a similar queue. The transmission from master to Minions is a one-to-many communication, whereas the transmission back is a many-to-one communication. In fact, since the Master makes use of its own group of local workers to receive responses, the transmission back really is more of a many-to-"not-quite as-many" connection.
SSH is more like a telephone line, in that it is designed for a one-to-one communication. Individual Minions listen for their phone to ring, and when a call comes in with a task, they can execute it and return a result immediately. The more Minions required to perform tasks, the more phone calls must be made. The Master can use local workers to set up multiple concurrent connections, much like a call center, but each task must still be relayed individually.

**Performance**

Another difference is performance. Salt SSH is very fast, but it has some overhead, some of which is consistent with SSH in general. The following actions are performed in addition to what Salt already does:

- Building and deploying the salt-thin agent
- Building and deploying the States tarball
- Establishing an SSH connection to the target

The last of these will happen with any program that uses SSH under the hood. The others may or may not happen with other frameworks. In a small infrastructure, this may be unnoticeable, but in a larger setup it may be problematic.

The salt-thin agent (covered in the next section) is not a problem because it will be generated in the first connection. Then it will be cached until the version of Salt changes on the master.

The State tarball (also covered in the next section) will be generated each time a State run is issued, which does cause some slowdown. However, it will not affect other execution modules.

Establishing SSH connections may be the biggest overhead. One system can only maintain so many connections at once. In fact, with a large enough job, Salt SSH will limit the number of active connections to 25 by default. This can be changed with the `--max-procs` flag:

```
# salt-ssh --max-procs 100 '*' test.ping
```

Exercise caution here. Increasing the maximum number of connections to a number that is not supported by the available resources can cause other issues outside Salt.
Understanding the salt-thin agent

As it turns out, automating SSH commands is not as simple as it may look at first. In an environment where every server runs exactly the same version of exactly the same operating system and exactly the same pieces of software, executing remote commands can be greatly simplified. However, very few environments meet this requirement, and Salt was designed to handle multiple environments.

In order to accommodate these disparate configurations, the code that performs the tasks needs to be able to autodetect pieces of its environment and then execute the tasks required by the user. In short, a piece of software that behaves exactly like Salt is necessary. Fortunately, in the Salt environment, that software is already available.

The salt-thin agent was designed to be a lightweight version of Salt that could be quickly copied to a remote system by Salt SSH in order to perform tasks. It doesn't ship with Salt (not as such, at least). It is built as needed by Salt SSH, using the Salt version and modules already installed on the master.

Once salt-thin has been packaged, it can be copied to the target system, where it is unpacked and then executed. Let's look at the specifics.

Building the thin package

In its default mode, Salt SSH requires the thin package. Raw mode doesn't require the thin package, but we'll cover this in a bit. However, since the package doesn't ship with Salt (not as such, at least). It is built as needed by Salt SSH, using the Salt version and modules already installed on the master.

The thin package contains just a little more than the bare essentials to run Salt. All the files for this package are collected from various locations on the Master. Most of these files exist inside the installation directories for Salt itself, but some belong to other packages that Salt depends on.

Interestingly, not all of Salt's dependencies will be needed. Many of the packages that Salt normally depends on are not necessary. As the communication will happen using SSH instead of ZeroMQ, this will not be included. The encryption libraries that are used to secure this communication transport are also not needed because the connection is secured by SSH itself.
Python will also not be packed in the thin package because a minimum version of Python must already be installed on the target system. Why not include Python in the thin package? There are a number of answers, but the most prominent one is binary compatibility. Python itself is compiled differently across various platforms with different versions of the gcc compiler. A master running Enterprise Linux would not be able to control a target running Ubuntu because the version of Python on the master would not meet the environment requirements on the target. Likewise, a 64-bit master would not be able to control any 32-bit targets.

Once the files that are necessary are collected, they are bound to a tarball called salt-thin.tgz. This package will contain only files that do not depend on the binary compatibility between the Master and Minion. This limits the tarball not only to scripts, mostly written in Python, but also to shell scripts (specifically the Bourne shell, also known as sh).

The actual construction of the thin package is performed by the thin runner. If necessary, for purposes such as testing, the thin package can be generated manually using this runner:

```
# salt-run thin.generate
```

The tarball will be saved in the cache directory on the master, usually in the /var/cache/salt/master/thin/ directory. If the file already exists, you will want to make sure to tell the runner to overwrite it. The command to overwrite the file is as follows:

```
# salt-run thin.generate overwrite=True
```

If you were to unpack the tarball after it was built, you'd find a small file structure with a handful of files in it. Libraries such as Jinja2 and PyYAML will be there along with a directory for Salt.

### Including extra modules

By default, the thin package will include all the modules that ship with Salt and the dependencies for the core Salt code. However, it will not include the Python modules that are required for noncore modules. For instance, if you are using the etcd execution module, which requires the etcd Python module, you need to be sure to include it in your thin package, by adding it to the end of the thin.generate command. The command to carry out this action is as follows:

```
# salt-run thin.generate etcd
```

Multiple modules can be specified and separated by commas with the following code:

```
# salt-run thin.generate etcd,MySQLdb
```
Deploying the thin package

After Salt SSH packages this file, it will be copied in the remote system. By default, it will be placed in the /tmp/ directory in a hidden directory with the name of the user that will be logged in on the target and a unique ID seeded with the hostname of the target system.

For instance, on a system whose FQDN is simply dufresne, the directory may be called /tmp/.root_0338d8__salt/. This directory will be owned by the user that Salt SSH logged in as (usually root), and the permissions will be set to 0700 so that no other users can read it.

If you would like to see this directory in action, including the unpacked thin package, you can do so by executing some introspective Salt commands:

```
# salt-ssh myminion cmd.run 'ls -ls /tmp'
myminion:
    drwxrwxrwt 16 root   root    420 Apr  3 16:51 .
    drwxr-xr-x 20 root   root   4096 Jul 29  2014 ..
    drwx------  8 root   root    260 Apr  3 16:50 .root_0338d8__salt

# salt-ssh myminion cmd.run 'ls -ls /tmp/.root*''
myminion:
    drwx------  8 root   root     280 Apr  3 17:43 .
    drwxrwxrwt 17 root   root     440 Apr  3 17:46 ..
    drwxr-xr-x  2 root   root     160 Apr  3 16:50 certifi
    drwxr-xr-x  3 root   root     880 Apr  3 16:50 jinja2
    drwxr-xr-x  2 root   root     220 Apr  3 16:50 markupsafe
    drwxr-xr-x  4 root   root     80 Apr  3 16:50 running_data
    drwxr-xr-x 31 root   root    1300 Apr  3 16:50 salt
    -rw-r--r--  1 root   root      79 Apr  3 16:50 salt-call
    -rw-r--r--  1 root   root     27591 Dec 13 21:18 six.py
    -rw-r--r--  1 root   root       8 Apr  3 16:50 version
    drwxr-xr-x  2 root   root     720 Apr  3 16:50 yaml
```

Executing the thin package

Now that the thin package is installed on the target, it must be executed. However, there is more work to be done before Salt is actually executed. Python can live in different locations depending on the environment, and Salt SSH needs to find where it is before it can call it.
The Salt SSH shim

A shim is a very tiny shell script whose job is to find the Python interpreter on the target system and then use it to start Salt. It is encoded to a base64 string on the master, sent to the Minion, decoded, and executed.

There are certain conditions that will affect how the shim is executed. If `sudo` is required on the target system, then the necessary commands will be embedded to the shim. If debug logging is turned on on the Master, then the shim will be executed in a debug mode, the output of which will be displayed on the master.

The way the shim is run can also vary. If the target system requires a connection with a `tty`, then the shim will be copied to the remote system using `scp` and then piped to `/bin/sh`; otherwise, it will be executed directly as one large command over SSH.

Preparing for Salt States

To run an execution command, not much is needed. However, executing Salt States does require a little more work. This is because even a traditional Minion that runs `salt-call` in the local mode requires a local copy of all the necessary files in the State tree.

When a Salt SSH command is executed using the State system, another tarball called `salt_state.tgz` will also need to be created. This file will be placed in the same hidden thin directory on the target as the `salt-thin.tgz` package. This tarball contains a copy of the necessary files from the State tree on the master so that the `salt-call` command will have access to everything that it needs for a State run.

The State tarball will also contain a copy of the State data (this is converted to low chunks) and a copy of any pillar data from the master. These files can also be viewed with a couple of the following introspective Salt commands:

```
# salt-ssh myminion state.single cmd.run name='tar -tzvf /tmp/.root*/salt_state.tgz'
myminion:
--------
  ID: tar -tvf /tmp/.root*/salt_state.tgz
  Function: cmd.run
  Result: True
  Comment: Command "tar -tzvf /tmp/.root*/salt_state.tgz" run
  Started: 17:53:46.683337
  Duration: 7.335 ms
  Changes:
   --------
```

[ 63 ]
Exploring Salt SSH

pid:
   26843
retcode:
   0
stderr:
stdout:

json
-rw-r--r-- root/root 128 2015-04-03 17:53 lowstate.json

Summary
-------
Succeeded: 1 (changed=1)
Failed: 0
-------
Total states run: 1

# salt-ssh myminion state.single cmd.run name='tar -Ozxvf /tmp/.root*/salt_state.tgz lowstate.json'

myminion:
--------
ID: tar -Ozxvf /tmp/.root*/salt_state.tgz lowstate.json
Function: cmd.run
Result: True
Comment: Command "tar -Ozxvf /tmp/.root*/salt_state.tgz lowstate.json" run
Started: 17:58:35.972658
Duration: 10.14 ms
Changes:
--------
pid:
   29014
retcode:
   0
stderr:
stdout:

  lowstate.json

  [{"fun": "run", "state": "cmd", "__id__": "tar -Ozxvf /tmp/.root*/salt_state.tgz lowstate.json", "name": "tar -Ozxvf /tmp/.root*/salt_state.tgz lowstate.json"}]
Summary
--------
Succeeded: 1 (changed=1)
Failed: 0
--------
Total states run: 1

Running Salt

Once the shim has found the Python interpreter and once the salt_state.tgz tarball is deployed (if necessary), it is finally able to execute Salt. Unlike a traditional Salt setup, it will not be run as a daemon. Instead, the salt-call command will be executed in the local mode, just like it would on a minion. The output will then be collected by the Salt SSH client on the Master, parsed, and sent to the user. We can see this information by running with a trace log level.

```
# salt-ssh myminion test.ping --log-level trace
...SNIP...
SALT_ARGV: ['/usr/bin/python2.7', '/tmp/.root_0338d8__salt/salt-call', '--local', '--metadata', '--out', 'json', '-l', 'quiet', '-c', '/tmp/.root_0338d8__salt', '--', 'test.ping']
_edbc7885e4f9aac9b83b35999b68d015148caf467b78fa39c05f669c0ff89878
[DEBUG ] RETCODE localhost: 0
[DEBUG ] LazyLoaded nested.output
[TRACE ] data = {'myminion': True}
myminion: True
```

It is also possible to take a deeper look at the salt_state.tgz tarball, but it will require logging in to the Minion for the last command, as shown in the following code:

```
master# cp /etc/services /srv/salt/
master# salt-ssh myminion state.single file.managed /tmp/services
source=salt://services
myminion: ----------
ID: /tmp/services
Function: file.managed
Result: True
Comment: File /tmp/services is in the correct state
Started: 18:18:28.216961
Duration: 5.656 ms
```
Exploring Salt SSH

Changes:

Summary
----------
Succeeded: 1
Failed: 0
----------
Total states run: 1

minion# tar -tzvf /tmp/.root_0338d8__salt/salt_state.tgz
-rw-r--r-- root/root 15895 2015-04-03 18:18 pillar.json
-rw-r--r-- root/root 118 2015-04-03 18:18 lowstate.json
-rw------- root/root 289283 2015-04-03 18:18 base/services

It will not be possible to view using two sequential state.single commands over Salt SSH because the second command will generate a new salt_state.tgz tarball, which will not include the base/services file. In order to obtain a truly informative view of the target with a single salt-ssh command, a full SLS file with enough States to perform sufficient introspection on the target will be required.

Salt’s running data

One more directory that you may have noticed in the temporary directory is the running_data/ directory. One design goal is to have Salt SSH remain as nonintrusive as possible. This means that the directory structure that Salt normally uses has to live someplace different: the temporary directory. We can take a look at this structure with another Salt SSH command:

# salt-ssh myminion cmd.run 'tree /tmp/.root*/running_data'

myminion:

```
/tmp/.root_0338d8__salt/running_data
|-- etc
 |  |-- salt
 |     |-- pki
 |     |   |-- minion
 |  `-- var
 |     |-- cache
 |        |-- salt
 |        |   |-- minion
 |        |     |-- proc
```
As you continue to issue commands against this minion, the directory structure will continue to grow and look more like a standard Minion directory structure. If you want Salt to completely remove all its traces when finished, including this directory, you can use the `--wipe` or `-w` flag:

```
# salt-ssh --wipe myminion test.ping
```

### Using the raw SSH mode

Salt SSH is very powerful in its default mode with salt-thin. However, there are some situations where it makes more sense to issue a raw SSH command. This can be accomplished using the `--raw` flag (referred to in its short form as `-r` from here on for brevity).

Using the raw mode will bypass all the overhead of creating and deploying the thin package. Just log in to the target, issue a command, and log out. The following two commands are functionally identical:

```
# salt-ssh myminion cmd.run date
myminion:
         Fri Apr  3 21:07:43 MDT 2015
# salt-ssh -r myminion date
myminion:
         ---------
         retcode:
                      0
         stderr:
         stdout:
         Fri Apr  3 21:07:43 MDT 2015
```

However, the raw command will execute faster because it has less overhead. It will also contain more information, such as `STDERR`, `STDOUT`, and the exit or return code from the command that was issued as well.
Exploring Salt SSH

This can be useful if you wrap Salt SSH with another program that depends on the output (especially, the return code) of the command on the remote machine. Make sure to run the command with an outputter that is consistent and easy to parse, such as JSON:

```
# salt-ssh -r myminion 'ping 256.0.0.0' --out json
{
    "myminion": {
        "retcode": 2,
        "stderr": "ping: unknown host 256.0.0.0",
        "stdout": ""
    }
}
```

In this example, there is no output to examine, but the error message can certainly be checked. Also, the return code will always be available.

Caching SSH connections

The raw SSH mode makes the execution model of Salt a little clearer. When a command is executed anywhere in Salt—be it the salt command, salt-call, or the salt-ssh mode—it will start a job, issue the command, and return the result. Depending on how it is called, Salt may or may not have a connection already established, but it will behave (so far as the user is concerned) as if it is creating a new connection, executing the job, and tearing down the connection.

This is fine in most instances, but there are some notable exceptions. For instance, configuring a network switch over SSH can be problematic. This is because a number of switches use the following configuration models:

- SSH into the switch
- Switch to a privileged user mode
- Execute commands that change configuration
- Review changes (if necessary)
- Commit changes
- Exit the privileged user mode
- Log out of the switch

Trying to use Salt SSH in the raw mode may make it as far as switching to the privileged user mode, but then it will log back out, forcing you to start over.
If you want to use OpenSSH on your master, you can take advantage of SSH, caching to maintain a connection to the switch as necessary. This is not something that is built-in to Salt SSH, but it can be used nonetheless. It is especially useful when scripting Salt SSH, for instance, in a bash script.

First, use the following command to set up the connection:

```
# ssh -fMN -o "ControlPath /tmp/salt_ssh_ctrl" myminion.com
```

This will tell SSH to set up a connection in myminion.com, but to do nothing with it. However, subsequent commands to that machine will automatically use the connection, which will be cached with a socket stored at /tmp/salt_ssh_ctrl on the Master.

This trick is useful outside Salt SSH as well, especially if you are regularly issuing one-off SSH commands against a machine. Even Salt SSH in its default and nonraw mode will see a slight performance increase because the overhead of establishing and tearing down each connection disappears.

When you are finished with this host, be sure to tear down the connection, as shown in the following code:

```
# ssh -O exit -o "ControlPath /tmp/salt_ssh_ctrl" myminion.com
```

This will disconnect from the target and remove the socket file from the Master.

**Summary**

Salt SSH is a powerful tool. It can be very comfortable for users in smaller infrastructures. This tool can also be useful for dealing with devices that allow SSH connections but are not able to have Python installed or cannot allow software (such as Salt) to be installed.

Next we will delve into the asynchronous nature of Salt and start to really explore how Salt can be used as an autonomous management platform.
Where to buy this book

You can buy Mastering SaltStack from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.