Into The Salt Mine Documentation

Release 2015.09.02

Christer Edwards

July 06, 2016

Contents

CHAPTER 1

Preface

1.1 Who Should Read This Book

This book is for those people wanting to improve the way that they manage their systems. From top to bottom this book will guide you through the necesarry steps required to fundamentally change the way that you manage your entire system lifecycle. Sound too good to be true? It's really not. Every now and then someone among us challenges the status quo and forges new ground in the digital world. SaltStack is just that. A fundamental change in the way we will manage our digital real estate.

If you're already familiar with SaltStack and simply want to dive deeper Into The Salt Mine, this is the place. My goal here is to describe SaltStack in a modular, bottom-up approach, giving a better overall vision of what is possible. I guarantee you'll learn something new about SaltStack by reading this book.

1.2 Why I Wrote This Book

I have been thinking about writing this book for quite a long time. Based on my history with the Salt project, and my previous success as an author, I figured there was a prime opportunity for me to publish something in this space. I had kind of put the idea on the back burner until one day a co-worker asked me if there were any good Salt books available. To my knowledge there were none, and the idea resurfaced. I decided that day to try my hand at sharing the vision, and this is the result of that work.

In my preparation for writing this book I looked at a lot of the online documentation available for saltStack. The official docs and tutorials are detailed and generally well written. There is also a growing number of blog posts on the topic. The one thing that I found lacking though was the bottom-up approach. Instead of taking the time to describe how all of the pieces fit together, everyone seemed more interested in writing Quickstart guides. Something that could be easily consumed, liked, shared or retweeted. What was missing from all of these publications was the vision. The big picture. An exploration of the Salt Stack, from the foundation up. I decided that this was the way I wanted to address this book.

Lastly, I wrote this book because I'm very passionate about the topic. I sincerely believe that SaltStack is the way of the future. A fundamental change has occurred in the way we manage the Internet, and SaltStack is going to play a huge role. From small deployments to public and private clouds, SaltStack has the tools and the scaleability to allow us to manage a growing number of systems with ease. I truly believe that, once you get a glimpse of the big picture, you'll change the way you look at your systems and regain an excitement for the future of computing.

1.3 Navigating This Book

This book is designed to be read front to back. I specifically layout out the sections in a logical manner where one chapter builds off the previous. There are a few exceptions to this however.

The *Command & Control* section should be read in order. These sections specifically build one atop the other, and make logical sense in this order. To skip around within this section would likely cause some confusion.

The other sections could be used as reference sections, after the *Command & Control* section has been read. Once the foundation is laid, the other sections in this book will make more sense. Again, it is important that the *Command & Control* section be read in its proper order, and then the rest of the book can be taken as needed.

1.4 Online Resources

The official Salt documentation is a very good resource for detailed information about modules, states, general configuration and tutorials. Much of that information was used as a reference in writing this book. This documentation can be found at:

> docs.saltstack.com

The official SaltStack blog can be a good resource for upcoming events, including trainings, conferences and meetups. This can also be a good resource for enterprise use cases and links to other related publications. This blog can be found at:

> Salt Ink - The SaltStack Blog

The official website for this book may also be useful. This includes additional tutorials, release notes and errata for this book. The official site can be found here:

> Into The Salt Mine

1.5 Conventions Used In This Book

The following typographical conventions are used throughout this book:

Italic

- Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Fixed-width`

- Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

Code Snippets

```
#!/usr/bin/env python2

import salt.client

try:
    caller = salt.client.Caller()
    ret = caller.function('test.ping')
```

Note

Note: This signifies a tip, suggestion or general note.

Warning

Warning: This indicates a caution or warning.

See Also

See also:

This suggests related topics to be referenced for more detailed information.

1.6 Using Code Examples

1.7 How To Contact Me

1.8 Acknowledgments

As always, to my muse, Casandra.

1.9 Introduction

The digital world as we know it is constantly changing. New technologies, both in hardware and software, are released daily. The number of connected systems to manage grows exponentially day after day. The Internet itself is larger than all of us–it's the biggest thing mankind has ever built. If you think about the scale at which the technologies we work with are expanding it can be daunting. The unending amounts of data created, transferred, stored and analyzed is more than any one of us can keep up with. It's all pretty amazing if you really think about it.

Imagine for a minute the vastness of it all. The *entire* Internet. All the systems. Servers old and new. Clouds, public and private, constantly forming, shifting and expanding. Billions of websites around the world. Overwhelming really. At that scale there's not one of us that could manage it. Not a hundred of us or a thousand of us. The only reason it works is because of skilled admins dutifully maintaining their little sections individually. That's really the only way it can be done.. or is it?

Every now and then something changes the game. Some brilliant visionary among us comes up with an elegant solution to a problem. Something that, for those who see it, changes everything. These solutions are sometimes software. Sometimes they are hardware. Sometimes it's just a new way to think about a problem. I'm sure you can come up with a few examples. A piece of software you discovered that makes your life so much easier you wonder how you ever got by without it. Once something has improed your life that much, how could you ever go back? Once you've seen the potential it's really hard to not be excited.

This is the way I feel about SaltStack. I see a revolutionary tool unlike any other currently available. I see the collective contributions of brilliant developers around the world creating solutions to problems nearly as fast as they arrive. I see a modular, flexible and lightweight alternative to the legacy systems and methodologies that we've been using for years. What I see with SaltStack is the future of computing.

1.9.1 History

The history of SaltStack goes back for years. It is the constantly improving culmination of many attempts at solving a common problem. The author and lead developer, Thomas S. Hatch, tried solving this problem many times for many companies. Each time improving on the last, and each time gaining valuable insight into the most efficient ways to manage systems. Let me take you back a few years to when my history with Salt began.

It was 2011 and I had just been hired by Thomas at an online music startup. He had recruited a few of us that had worked together in the past in what I thought of as a dream team. If I could work with anyone again, it would be these guys. We got along great. We worked hard. We solved complex problems that nobody had solved before. We were breaking new ground and growing fast.

Thomas started working on a tool to manage our private cloud infrastructure. The idea was, using some home-grown tools, we would be able to spin up machines on the fly, provision them and drop them into different environments. When we had updates to systems we would simply spin up a new machine with the latest code on it and transition it into place, recycling the old machine. The vision was there. It was, at the time, very exciting. The only problem was that the tools weren't there. The vision was there, but we didn't yet have the tools to take us from point A to point B.

Remember, we were a fledgling startup with grand ideas but not a lot of money. We had to come up with our own solutions and build our own tools. Everything from scratch, with a small team.

By the time I hired on work had begun on these tools. We had a tool called "Bacon", another called "Butter" and a third called "Salt". These were in-house, python-based tools for provisioning and managing our private cloud. At the time these three were very rudimentary, but for the most part they got the job done. They had little bugs here and there, but we also had a very talented development team that squashed those about as quickly as they came up.

I fondly remember times working in what couldn't even be called a conference room, shared with five other system engineers. I'd run into an issue with one of our tools and called it out. I'd promptly hear Thomas call back "give me one minute!", and like clockwork the bug would be fixed. Time and time again, improvements implemented at the speed only a startup can seem to manage. Day by day our tools become more mature, our infrastructure became larger, and the company seemed to be doing great.

Somewhere along the line here Thomas had the foresight to get the company to allow him to open source these tools. He had developed most of it on his own time at home in his basement (yes, the old cliche), and primarily fixed bugs with our feedback while at the office. He placed the code on github and work continued.. but not for long.

Before we knew it months had passed and it was the Holidays. Like any startup we had minimal time off during the season. There were constantly things to fix and systems to build. We were preparing for a big deal with a mobile telecom company and hoping to expand. All of our hard work was finally paying off! We were all very excited. We were supposed to be getting a dedicated NOC team, new hires, data center expansion–all the things we needed to support this upcoming deal. Unfortunately I don't think any of us really saw what was really coming.

I remember it clearly to this day. One of the perks of working at this startup was flexible working and office hours. We spent most of our time working from home and only met as a team in the office once or twice a week. This was a Wednesday, the week between Christmas and New Years, and I was working from home in an old beatup chair in my living room. Our lead network engineer and my roommate at the time was upstairs with the flu, taking the day off.

Then an email arrived. "Staff meeting conference call @5:00pm". I didn't think much of it, and dialed in at the suggested time. This, depending on how you look at it, is either where it all ended or where it all began.

Layoffs. Everyone. The investors had pulled out. There would be no severance. There would be no more work. Report to the office tomorrow to turn in any equipment you have and good luck to you. I was floored! I remember being so shocked and sitting up so abruptly in surprise that my laptop tumbled to the floor. My company laptop that I was expected to return in the morning. What had just happened? Was he for real? I–we–were all shocked. What were we going to do? I'd never been laid off before. The rest of that afternoon is a bit of a blur, but I'll always remember that moment as a crossroads for a lot of us.

As the days and the weeks went on some of us found jobs right away. Others took a bit longer. For some people it was a great move. They landed even better jobs with better companies. Some people found temporary consulting work. If

you've ever been part of a massive layoff you know how it goes. It's difficult, but people usually make it. During this time we all kept in touch pretty well. We'd share job leads and keep tabs on who was ending up where, and did that company need anyone else. During all of this there was one person that didn't seem worried, and didn't seem to be as concerned about finding a new job. Thomas.

Thomas had a plan. Something bigger than where we came from or where a lot of us ended up. He was going to strike out on his own with a new tool that had been slowly picking up popularity in the open source community. He had decided he was going to form SaltStack, the company behind the Salt tool we had begun with at the startup.

If you're reading this I think you have some idea how that turned out. SaltStack is now a growing, successful company filled with talented people. Some of the people from our original dream team now work for Thomas again, this time at a much more successful company.

To give you an idea of the usefulness of the Salt tool, every one of the systems engineers from that original startup have deployed Salt for our new employers. I'm currently architecting a Salt based solution across multiple products and nearly fourty-thousand servers. Salt is a game changer and I hope to be able to work with it long into the future.

CHAPTER 2

Chapters

2.1 Installation

2.1.1 Arch Linux

2.1.2 Debian

2.1.3 Fedora

2.1.4 FreeBSD

SaltStack is available for FreeBSD in both package and port form. Outlined below are instructions on installing and starting the Salt service(s) on FreeBSD.

Installation

On FreeBSD 10 and later Salt can be installed using the `pkgng` utility:

```
pkg install py27-salt
```

On older systems, or systems not using pre-compiled packages, compilation from ports is also available:

```
make -C /usr/ports/sysutils/py-salt install clean
```

Either of these methods will install the full set of Salt utilities including the Salt master, minion, syndic. Repeat the above instructions for any FreeBSD system you'd like to be part of your Salt infrastructure.

Post-Installation

The FreeBSD port for Salt lays down a sample config for both master and minion. While the service will technically run using only default values without a config file in place, you'll likely want to copy the sample config into use.

Master

Copy the sample config file:

```
cp /usr/local/etc/salt/master.sample /usr/local/etc/salt/master
```

rc.conf

Activate the Salt master in `/etc/rc.conf`:

```
sysrc salt_master_enable="YES"
```

Start the Master

Start the Salt master:

```
service salt_master start
```

Minion

Copy the sample config file:

```
cp /usr/local/etc/salt/minion.sample /usr/local/etc/salt/minion
```

rc.conf

Activate the Salt minion in `/etc/rc.conf`:

```
sysrc salt_minion_enable="YES"
```

Start the Minion

Start the Salt minion:

```
service salt_minion start
```

2.1.5 Gentoo

2.1.6 OS X

2.1.7 Red Hat Enterprise

Salt can be installed on Red Hat Enterprise (and variants) using the EPEL repository. This additional repository is maintained primarily by Red Hat employees and Fedora contributors. It contains additional enterprise packages for use with Red Hat and its variants.

Enable EPEL

To enable the EPEL repository install the appropriate package listed below based on your version of Red Hat.

RHEL 5

```
rpm -Uvh http://mirror.pnl.gov/epel/5/i386/epel-release-5-4.noarch.rpm
```

RHEL 6

```
rpm -Uvh http://ftp.linux.ncsu.edu/pub/epel/6/i386/epel-release-6-8.noarch.rpm
```

Installation

On Red Hat based systems the Salt master, minion and syndic packages are built seperately. It is necessary to install the appropriate package for the system role. Typically this means you'll have one Salt master and many Salt minions.

salt-master

```
yum install salt-master
```

salt-minion

```
yum install salt-minion
```

Post Installation

Master

Configure the service to start at boot:

```
chkconfig salt-master on
```

Start the service:

```
service salt-master start
```

Minion

Configure the service to start at boot:

```
chkconfig salt-minion on
```

Start the service:

```
service salt-minion start
```

2.1.8 Solaris

2.1.9 SuSE

2.1.10 Ubuntu

2.1.11 Windows

2.2 Command & Control

2.2.1 ZeroMQ

I'd be remiss in my duties as an author if I did not start a book about SaltStack by discussing ZeroMQ. This, I think, is the fundamental cornerstone of what makes SaltStack so powerful. Without ZeroMQ SaltStack would not be capable of what it does, and many of the performance-specific benefits would be gone. In order to share with you the big picture vision of what SaltStack is capable of, we'll first need to talk a little bit about ZeroMQ.

What is ZeroMQ?

What is ZeroMQ and why should I care? Let's just dive right in. First, ZeroMQ is a socket-based high speed networking library that is leveraged by SaltStack to allow for real-time communication between systems. SaltStack leverages ZeroMQ to perform the high-speed communication between systems, while not requiring a managed service to be running.

Technically ZeroMQ is a message queue, similar to those that you might be familiar with. AMQP, rabbitmq, kafka, etc. These are all popular message queue services that you can set up in your infrastructure. The primary difference

here is that ZeroMQ is not another service to setup and maintain. It is simply a programming library that application developers can leverage in order to achieve high speed network communication within their application. SaltStack's requirement on ZeroMQ requires no effort or configuration on your behalf, but provides all the benefits of a full-fledged messaging service.

So why ZeroMQ? ZeroMQ was developed initially for high-speed banking transactions. A large banking institution determined that the network transactions themselves were too expensive, costing the company millions of dollars. They specifically set out to design a message queue interface that would allow for high speed, low cost transactions on the network, saving them money.

We're all familiar with the word 'latency'. We generally understand that the lower the latency the better. This was essentially the goal here. Achieve the lowest latency possible, in the simplest manner possible. The people behind ZeroMQ initially created the AMQP system. This was designed for JP Morgan Chase, specific to their network latency needs. AMQP has since been abandoned by the initial developers (although still maintained by adopters), in preference of what they suggest is an improved system with ZeroMQ.

ZeroMQ and Salt

How does Salt leverage ZeroMQ? To be honest it's all pretty transparent to the end user. You could never learn anything more about ZeroMQ than what is included here and still use SaltStack in an enterprise setting. While it is a very critical and important piece, the way that it has been implemented is very transparent to the user. All you have to worry about is starting the `salt-master` and `salt-minion` services (which we'll cover later), and the ZeroMQ based network layer is automatically maintained. I think this fact is both a credit to the ZeroMQ developers as well as the SaltStack developers. Such an integral component is as transparent and maintenance-free as it is.

The reason that I wanted to cover ZeroMQ first is because it is a key reason why SaltStack is different than any of its competitors. It is what sets it apart from other remote execution and configuration management tools. Many of these other tools leverage existing communication protocols, such as `ssh` or a traditional serial `TCP` socket connection. While these solutions excel in some areas, they simply weren't designed for the scale at which SaltStack operates.

Let me see if I can describe this in a simple way.

SaltStack excels at high-speed, asynchronous communication between connected systems. This communication–this underlying message bus–can be leveraged for many different uses, only one of which is parallel remote execution. Let's imagine for a moment how this all fits together.

Imagine your infrastructure the way it stands today. Dozens, hundreds or thousands of servers tucked away nicely in data centers around the world. These systems are connected to power and networked via switches and routers. it isn't terribly complicated if you think about it. You can currently connect to any of your systems from your management network and do any amount of administration needed. That administration is generally done over `ssh`, which is usually a one-to-one connection. You connect securely to one system, do your maintenance, and move on to the next. That's the way it's been done for years. That's the way it's been done for as long as I can remember anyway.

Granted there are some tools that expand this functionality a bit. Tools like `clusterssh` allow you to mirror commands to multiple systems at once over multiple `ssh` connections. This works well to an extent, but there is a point at which this method doesn't scale. Unfortunately there are simply not a lot of tools that allow you to control large pieces of your infrastructure in a timely manner.

Now imagine that all of your systems were connected over a high speed, asynchronous messaging system. Imagine being able to send commands to one system or a thousand systems using the same method. Imagine being able to leverage this high speed message bus to query real-time information about all of your systems, all at once. This is what ZeroMQ provides for us in the SaltStack world. A high-speed way to connect to not one-at-a-time, not two-at-a-time but all-at-a-time with little overhead.

Just imagine for a second a high-speed network connecting all of your systems. A network that is fully encrypted and sits on top of your existing physical network infrastructure. This network can easily connect hundreds, thousands and even tens of thousands of systems without issue.This network can provide you with a low latency method by which you can query for or send information to your entire network of servers. I really hope you're paying attention here,

because this is the foundational piece of the entire Salt stack. This is the piece that allows the rest of the magic to happen. This is the piece that facilitates the high speed communication between systems, and not just minion and master. This high speed network can be leveraged for one minion to query or communicate with another minion. While it is traditionally a pub-sub communication pattern it can be thought of more broadly in the sense that it allows near instant communication between your entire infrastructure.

I want to again reiterate that this high speed network requires no additional services to install and maintain, and little to know knowledge of the underlying connectivity to use. It is truly high speed, low latency connectivity at low cost. Low cost to the network and low cost to the maintainers of the infrastructure. It is all behind the scenes. The magic behind the curtain.

As you get deeper into this book I want you to keep in mind that everything rides on top of this high speed encrypted network. Every command you post to the message queue is instantly available to all of your systems. Every time you want to update configuration on your servers the instructions are instantly available everywhere. No more iterating through lists and sending commands over ssh. No more sitting and waiting for networking protocols that weren't designed for the scale of todays internet try and keep up. You will be leveraging a modern, encrypted, asynchronous communication network actually designed just for this.The Internet is growing. Your company is growing. Grow with it.

2.2.2 Key Management

Before we can begin any communication on top of our ZeroMQ network we need to accept encryption keys. The underlying ZeroMQ network is not encrypted, but SaltStack adds a layer of AES public key encryption to all communications. This adds very little overhead while ensuring that all communications are securely encrypted between all hosts. Before these encryption keys are accepted on the master, no communication will take place.

The Salt Master provides a utility called `salt-key` to allow you to manage these encryption keys. Each minion will automatically generate their respective keys and submit them to the master for acceptance. There are a number of ways to manage keys at scale, but here we'll just look at the basic options of the `salt-key` utility.

`salt-key` executes simple management of Salt public keys used for authentication and encryption.

Listing keys

```
-l ARG, --list=ARG
```

The args `pre`, `un`, and `unaccepted` will list unaccepted/unsigned keys. The args `acc` or `accepted` will list accepted/signed keys. The args `rej` or `rejected` will list rejected keys. Finally, `all` will list all keys.

```
-L, --list-all
```

List all public keys. (DEPRECATED: use `--list-all`)

Accepting keys

```
-a key_name, --accept=key_name
```

Accept the specified public key(s). Globs are supported.

```
-A, --accept-all
```

Accept all pending keys.

```
--include-all
```

Include non-pending keys when accepting or rejecting keys.

Rejecting keys

```
-r key_name, --reject=key_name
```

Reject the specified public key. Globs are supported.

```
-R, --reject-all
```

Reject all pending keys.

```
--include-all
```

Include non-pending keys when accepting or rejecting keys.

Printing keys

```
-p key_name, --print=key_name
```

Print the specified public key.

```
-P, --print-all
```

Print all public keys.

Deleting keys

```
-d key_name, --delete=key_name
```

Delete the specified key(s). Globs are supported.

```
-D, --delete-all
```

Delete ALL keys.

Key fingerprints

```
-f key_name, --finger=key_name
```

Print the specified key fingerprint.

```
-F, --finger-all
```

Print all keys fingerprints.

Key Generation

```
--gen-keys=key_name
```

Generate a keypair for use with Salt.

```
--gen-keys-dir=/path/
```

Define the path to save the generated keypair. Only works with the `--gen-keys` option; default is the current directory.

```
--keysize=key_size
```

Set the keysize for the generated key. Only works with the `--gen-keys` option. Keysize must be 2048 or higher; the default is 2048.

2.2.3 Grains

Introduction

Salt includes a built-in mechanism for determining static information about a system. These bits of information are referred to as *grains* within the Salt vocabulary. You might think of it as "little *grains* of information" about a machine. These *grains* of information include hardware and networking information, operating system details, and much more. These *grains* are also expandable to include other bits of static information that you'd like to have assigned to a machine. In this chapter we'll explore the *grains* subsystem and learn how to leverage this data within our Salt infrastructure.

Goals

Once you've completed this chapter you should have an improved understanding of what grains are, how grains are queried, defined and synced. You should also understand the order of grain definition precedence.

Note: If you're not yet familiar with the *grains* system it is important to study this entire chapter. Upcoming chapters will make extensive use of the information outlined here.

Standard Grains

Salt includes a set of "core grains" that should be available on any system. These grains should be detected properly on every supported operating system and distribution. In this section I'll outline many of these core grains.

The list below defines the full set of core grains found on a CentOS Linux system. As you can see, these are just the keys without values, but it gives you an idea of what type of information is stored in grains.

- SSDs
- biosreleasedate
- biosversion
- cpu_flags
- cpu_model
- cpuarch
- domain
- fqdn
- fqdn_ip4
- fqdn_ip6
- gpus
- host
- hwaddr_interfaces

- id
- ip4_interfaces
- ip6_interfaces
- ip_interfaces
- ipv4
- ipv6
- kernel
- kernelrelease
- locale_info
- localhost
- lsb_distrib_codename
- lsb_distrib_id
- lsb_distrib_release
- machine_id
- manufacturer
- master
- mem_total
- nodename
- num_cpus
- num_gpus
- os
- os_family
- osarch
- oscodename
- osfinger
- osfullname
- osmajorrelease
- osrelease
- osrelease_info
- path
- productname
- ps
- pythonexecutable
- pythonpath
- pythonversion
- saltpath

- saltversion
- saltversioninfo
- selinux
- serialnumber
- server_id
- shell
- virtual
- zmqversion

As you can see, there are over fifty items defined for each system within your Salt infrastructure. These items will be used in upcoming chapters to demonstrate the flexibility of making your minion management more dynamic.

Let's look at what some of these values store.

Listing Grains

As you saw in the previous examples, there are a few different ways to query for grains. Primarily you'll use the `grains` module and query for one, all or specific grain values. If you know the name of the grain you're looking for you can query for that directly:

```
[root@minion ~]# salt '*' grains.item fqdn
alpha:
----------
fqdn:
    alpha.domain.tld
```

```
[root@minion ~]# salt '*' grains.item kernelrelease
alpha:
----------
kernelrelease:
    2.6.32-431.29.2.el6.x86_64
```

```
[root@minion ~]# salt '*' grains.item mem_total
alpha:
----------
mem_total:
    7872
```

```
[root@minion ~]# salt '*' grains.item cpuarch
alpha:
----------
cpuarch:
    x86_64
```

Defining Grains

Beyond the "core grains" that are defined on every system, it is possible to define custom grains. These custom grains can be used to define additional attributes about your systems. Examples of this might be *datacenter*, *rack*, *cabinet*, or other internal or deployment-specific information. Grains are defined on a per-minion basis and append to the existing grains.

See also:

For more information about custom grains replacing existing grains, see the next section **Precedence**.

Custom grains can be defined in a couple of places. Again, because grains are unique per minion, custom grains are defined on a per-minion basis in one of two places:

- /etc/salt/minion
- /etc/salt/grains

There is a slight difference in the way custom grains are imported depending on the location. First we'll outline the `/etc/salt/minion` method, followed by `/etc/salt/grains`.

/etc/salt/minion

Custom grains can be added directly to the minion config file, or included as a new file in the `/etc/salt/minion.d/` directory. If custom grains are added to either of these locations the whole structure needs to be prefixed with the `grains` configuration option. See the example below:

```
grains:
  datacenter: va5
  rack: 17
  cabinet: 3
  role: webserver
```

As you can see, custom grains are the same simple key-value pairs that the core grains are. These can be any arbitrary key-value pair that you want to define for your systems. In addition to the key-value pairs, you can define other more complex data structures such as lists or dictionaries. See the example below for more complex custom grains.

```
grains:
  role:
    - webserver
    - memcache
  owners:
    - tuttle
    - ewoolley
```

/etc/salt/grains

```
role:
  - webserver
  - memcache
```

will have the ability to limit which systems have access to data, as well as limit access on a per-user basis.

The components outlined in this chapter require Salt version 2014.7.0 or later.

2.9.2 Requirements

The set of requirements defined for this project are outlined below.

Encryption

- Secrets not stored in plain text on disk (client or server)
- Secrets always encrypted in transit across network

Reliability

- Fault tolerant enough to have the server go down and the application keeps functioning
- Server uptime equivalent to the most critical server or component in the data center

Scalability

- Can support thousands of concurrent servers
- Can support storage of thousands of secrets

Secret types

- Support standard secret types (PKI key pairs, hashes, etc)
- Custom secrets with custom key/value meta data
- Support for automated changing of passwords on schedules and on demand

Administration

- Searching across all fields
- APIs for administration of secrets from the command line
- Bulk edit of secrets
- SDKs for integrating with code and cli
- CLI utility

Access Control

- Secrets can be permissioned to individual servers
- Read, write, and delete as separate permissions
- Group permissioning for both secrets and applications/clients

Required Packages

In order to build this secret storage solution you'll need:

- Salt Master
- Salt Minion(s)
- Public and private GPG keys
- GPG Agent (optional)

Required Configuration

In order for Salt master to parse the GPG cipher data it needs the GPG renderer enabled. This is done by updating the `/etc/salt/master` config file and applying the below change.

```
- #renderer: yaml_jinja
+ renderer: jinja | yaml | gpg
```

2.9.3 Architecture

This secret storage solution fits natively into any Salt infrastructure. This means SaltStack provides the functionality for this service out of the box. It simply requires some configuration and initial setup. The following sections will outline each stop in the process.

Overview

For a really high level overview here's how each of these components fit together:

The Salt Master will be the central storage server. All secrets will be stored within the pillar data on the master (usually `/srv/pillar/`). Each secret is stored on the master in a GPG encrypted cipher. The public and private GPG keys are only ever stored on the master. GPG keys never need to be shared to minions. The `python-gnupg` library needs to be installed on any minion wanting to request secure keys. Lastly, assuming you'll want to secure your encrypted secrets with a GPG key passphrase, you'll need a configured GPG agent to unlock these secrets upon request, without the requirement of entering a passphrase each time.

Salt Master

The secret store is housed on the SaltStack master within the Pillar system. This allows for the secrets to be available to any connected system, with access limitations defined by the Pillar top file. In some of my early discussions about this architecture there was some concern about the privacy and general access to the secret store server. It should be noted that the SaltStack master should restrict general access just as any "vault" would. If you can't trust your administrators that have root access to your systems with the secrets stored on your server, well, I'd say you have bigger problems. This configuration assumes access to the SaltStack master is restricted to those with root access to the general infrastructure.

Note: Anyone with root access on the SaltStack master will be able to retrieve any encrypted secret.

Pillar Data

The pillar system sits at the heart of this secret storage solution. This is where the data is stored and where limitations on access to the data is configured. In this example I assume the default path for pillar data, `/srv/pillar`.

First, if it doesn't yet exist, create the pillar path:

```
mkdir -p /srv/pillar
```

Second, populate a default `top.sls` file. This file is where you map data to minions.

```
base:
  '*':
    - cache
  'alpha':
    - vault1
  'bravo':
    - vault2
```

In the example above I've made accessible anything in the `cache.sls` file to all machines. The `vault1.sls` is accessible only to the host alpha, and `vault2.sls` only available to bravo. You'll of course want to update these values to represent your own system names.

Salt Minion

The SaltStack minions will generally be the source for queries against the secret store. By defining restrictions within the Pillar configuration you can limit which machines have access to specific secrets or groups of secrets. Secrets are queried from the minions using the `pillar` module. I've included a few examples below:

The command below will query for the value of the `secret1` key. This key represents a key-value store within one of the "vaults" available to this system.

```
salt-call pillar.item secret1
```

If you would like to request all secrets available to a specific system you can use a similiar command:

```
salt-call pillar.items
```

Note: Different outputters can be used when retrieving these secrets. Outputters include `--json`, `--raw`, `--txt`, `yaml_out`, and more.

2.9.4 Key Generation

This section outlines the steps required to generate the required public and private keys.

```
mkdir /etc/salt/gpgkeys
chmod 0700 /etc/salt/gpgkeys

gpg --homedir /etc/salt/gpgkeys --gen-key
```

At this point you'll be prompted with key generation options.

- Select `(1) RSA and RSA (default).`
- Select `2048` for the keysize (default).
- Select `0` for the key expiration (key does not expire).

- Enter your project name for `Real Name`.
- Enter an email address for `Email address`.
- Enter a blank line for `Comment`.

You'll be given one last chance to overview what you've entered and make any changes. When you're ready to generate your keys, select `(O)kay`.

At this point you'll be prompted for a passphrase. It should be noted that the inclusion of a passphrase makes the overall configuration a bit more complicated, but retains the highest amount of security. It will require a step of manually unlocking the key with a passphrase anytime the server is rebooted.

Once the key is generated (it may take some time depending on your hardware) you'll need to export the public key and import it into the systems keyring. To export the public key you'll need to specify which key (as systems can have many keys). This can be done using any unique information about the key, including `Real Name`, `Email address` or `Comment` as defined during the key generation. The example below exports the key using the email address.

```
gpg --homedir /etc/salt/gpgkeys --armor --export email@address.org > pubkey.gpg
```

```
gpg --import pubkey.gpg
```

You are now ready to encrypt data using this GPG key-pair. This can be done using a simple shell one-liner:

```
echo -n "top secret data" | gpg --homedir /etc/salt/gpgkeys --armor --encrypt -r email@address.org
```

The resulting output of this can then be used within the SaltStack pillar system in the following format:

```
secret: |
  -----BEGIN PGP MESSAGE-----
  Version: GnuPG v1

  hQEMAweRHKaPCfNeAQf9GLTN16hCfXAbPwU6BbBK0unOc7i9/etGuVc5CyU9Q6um
  QuetdvQVLFO/HkrC4lgeNQdM6D9E8PKonMlgJPyUvC8ggxhj0/IPFEKmrsnv2k6+
  cnEfmVexS7o/U1VOVjoyUeliMCJlAz/30RXaME49Cpi6No2+vKD8a4q4nZN1UZcG
  RhkhC0S22zNxOXQ38TBkmtJcqxnqT6YWKTUsjVubW3bVC+u2HGqJHu79wmwuN8tz
  m4wBkfCAd8Eyo2jEnWQcM4TcXiF01XPL4z4g1/9AAxh+Q4d8RIRP4fbw7ct4nCJv
  Gr9v2DTF7HNigIMl4ivMIn9fp+EZurJNiQskLgNbktJGAeEKYkqX5iCuB1b693hJ
  FKlwHiJt5yA8X2dDtfk8/Ph1Jx2TwGS+lGjlZaNqp3R1xuAZzXzZMLyZDe5+i3RJ
  skqmFTbOiA==
  =Eqsm
  -----END PGP MESSAGE-----
```

Note: Please note the pipe character ("|") after the key name as well as the yaml-style indentation for the entire GPG cipher value.

2.9.5 GPG Agent Requirements

This section outlines the requirements for the `gpg.conf` and `gpg-agent.conf`.

- gpg.conf updated to use-agent
- gpg-agent.conf updated to specify pinentry-program
- gpg-agent.conf updated to specify extended cache-ttl

2.9.6 The GPG Agent

In order to publish GPG encrypted secrets using a passphrase-enabled key you'll need to run a GPG agent. This agent will allow you to authenticate once to the encryption key and not require a passphrase be entered anytime someone requests a key. This provides the added security of a passphrase on the encryption key, but the usability of not requesting a passphrase on every request.

This step of the process requires an update to a configuration file as well as manually unlocking the GPG key. I will again mention that this process is currently manual and will need to be repeated anytime the system is restarted and the GPG agent restarted.

There are a few settings that need to be defined in order for this to work properly. The next two sections tell the system that you want to use an agent, and how that agent should be used to prompt you for a passphrase.

It is important to note that the gpg-agent has a default cache ttl value. If the key is not unlocked or requested within that cache time the passphrase will be forgotten and you'll need to request it again.

I have solved this by increasing the default max-cache-ttl value to one day as well as configured a salt scheduler to request a "cache" token from the secret store on a regular interval. On each successful request the max-cache-ttl is reset and the countdown starts over. The combination of a one-time unlocking and regular queries for an encrypted value will allow the cache to remain effective until the system or the services is restarted.

gpg.conf

You need to tell the gpg utility that it should use the agent. This is done by updating the `gpg.conf` file, which you'll likely need to create inside the `/etc/salt/gpgkeys` directory. This will tell any instance of `gpg` specifying this path as a `--homedir` that it should use a gpg-agent.

```
+ use-agent
```

gpg-agent.conf

In order for Salt to prompt you for the passphrase it needs to know how to do so. This can be defined within the `gpg-agent.conf` file, which you'll likely need to create inside the `/etc/salt/gpgkeys` directory. This file simply holds configuration on how the agent should run. In our basic setup you'll only need to add a single line to this new file. The example below shows a unified diff of the file. Add the line(s) as defined by the + character, but do not add the + character itself.

```
+ pinentry-program /usr/bin/pinentry-curses
+ default-cache-ttl 86400 # one day
+ max-cache-ttl 31536000  # one year
```

gpg-agent

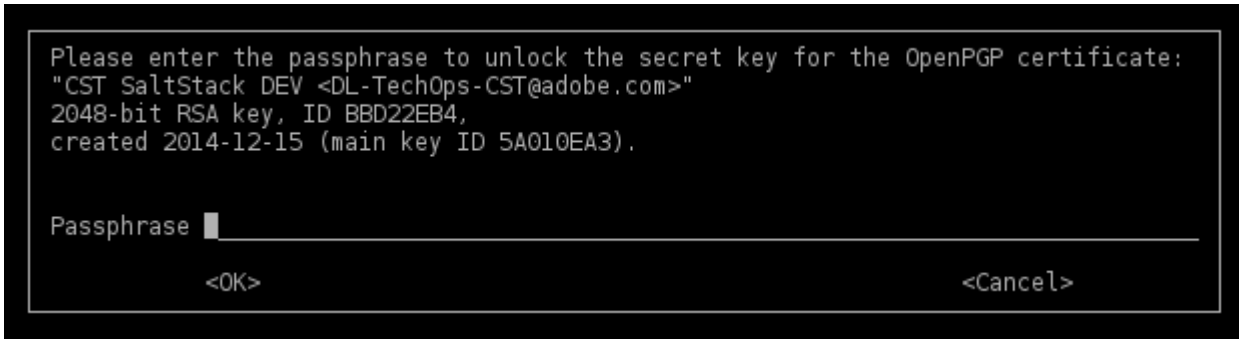
You'll need to manually launch the gpg-agent and then tell SaltStack where it can find the running agent. This

unlock the key

Once you've started the gpg-agent and provided SaltStack with the information required to access this agent you're ready to unlock the keyring. All you should need to do for this to happen is request a key from the Salt master. Make sure that your pillar top file has been configured to allow the Salt master access to a key (any key will do). Request this key using the command below:

```
salt-call pillar.item secret
```

When you run this command Salt will try to decipher the encrypted value stored within your pillar data. Now that it knows about your gpg agent information it'll request access through that socket. The first time it runs it'll determine that no access has been granted and prompt you for a passphrase. You should see a curses-based prompt appear in your terminal asking you for the encryption password. Enter this password once and your key will be unlocked. This should last for the duration of your session or your GPG max-cache-ttl.



2.9.7 Conclusion

2.10 Development

2.10.1 Writing Modules

While not always required, sometimes it is necessary to write and distribute custom Salt modules for added functionality or integration with other products. This chapter will outline the basic structure and best practices for creating custom execution modules.

Anatomy of a Salt module

When developing custom Salt execution modules there are a few basic rules that need to be followed. This chapter aims to outline the basic structure of a module, its key components and general best practices. Let's dive right in.

header

```
# -*- coding: utf-8 -*-
'''
:maintainer: Christer Edwards (christer.edwards@gmail.com)
:maturity: 20150910
:requires: none
:platform: all
'''
```

If you've written Python scripts in the past you might wonder where the "shebang" is; the *#!/usr/bin/env python* declaration. In the case of a Salt module (or technically a Python module) this is not required as it is not called directly from the shell. Because this file will be imported by the Salt loader, as long as it parses properly, it will become available without a "shebang".

In this case we simply define the character encoding used within this module. Unless you need to or have a good reason to use an alternate encoding, *utf-8* is probably preferred.

For more information on this tag see PEP-0263.

Next we document the module. Due to the way that the Salt documentation is automatically generated, whatever documentation you define within this top-level docstring will be used in the documentation page. While this is not *required* it is preferred, especially if you ever hope to have your module merged in upstream Salt.

I have written some custom modules that include more content within the top-level docstring than actual code in the module. This amount of documentation never hurts, and you'll never be accused of not properly documenting your code!

imports

```
from __future__ import absolute_import

import logging
```

At this point you can begin importing the Python modules you require. At minimum you should use the lines in the example above. You'll likely need more, but this should always be your baseline.

The *absolute_import* function from the *__future__* provides compatibility between Python2 and Python3. At the time of this writing Salt is still not fully Python3 compatible, but using the "future" import standard ensures that custom modules are at least up to par in that regard.

The second component that you want to import is the logging system. This allows you to easily add debugging output to your module. This can be extremely helpful during development and testing, and allows end-users to configure log levels during runtime.

We'll explore examples of implementing logging later, but for now you should make sure you import the logging module.

GLOBALS

```
LOG = logging.getLogger(__name__)
```

The above *GLOBAL* activates the logger and makes it available throughout your module. In order to leverage the logging *GLOBAL* use the following syntax:

LOG Example:

```
LOG.info('info output')
LOG.debug('debug output')
LOG.error('error output')
LOG.warning('warning output')
LOG.critical('critical output')
```

__virtualname__

```
__virtualname__ = 'custom_module'
```

The *__virtualname__* variable definition defines a custom name for your module. If this definition is missing it will default to the name of the module file itself (minus the *.py*). While not required, this variable definition is common to most modules, and often simply matches the Python module name itself.

This definition also allows the module layer to be abstracted, and is what allows a standard command across multiple platforms. For example, the *pkg* module supports a wide range of binary package providers. From *yum* to *apt-get* and everywhere in-between. Each of these defines *__virtualname__* as *pkg*, meaning based on the conditional statements within the *__virtual__* function only the appropriate *pkg* provider is loaded.

__virtual__()

```
def __virtual__():
    '''
    Determine whether or not to load this module
    '''
    if __salt__['grains.get']('kernel:Linux'):
        return __virtualname__
```

The *__virtual__()* function is a critical component of any Salt execution module. This function allows you to enter logic to determine whether or not your module should load on the given platform. You have full access to Salt components, including *grains*, *pillar*, testing on the availability of other Salt execution modules, and more.

2.10.2 Functions

A Salt execution module generally consists of "private" and "public" functions. These functions are either callable from within the module itself (private), or callable directly through Salt (public). The way Salt treats functions within these custom modules very much follows the Pythonic way of handling modules and methods.

In this section I provide examples of both types of functions:

"private"

```
def _private():
    '''
    "Private" function; only callable within this module
    '''
    LOG.debug('Executing the _private function')

    ret = {}
    return ret
```

A "private" function works the same way that any other function does. The only difference here is that the function name is preceded with an underscore (_). Any function prefixed with an underscore character will only be available within the module, and will not be directly callable through Salt.

"public"

```
def public(*args, **kwargs):
    '''
    "Public" function; available to Salt, ie; module.public

    CLI Example:

        salt '*' module.public
    '''
    LOG.debug('Executing the public function')

    ret = _private()
    return ret
```

"public" functions within an execution module are mapped and made available to the Salt administrators. Any "public" function becomes available to be called from Salt, prefixed by the module name. For example, if our custom module was called "module", and our function name was "public", we'd call this through Salt by targeting *module.public*.

Full Example

```
# -*- coding: utf-8 -*-
'''
:maintainer: Christer Edwards (christer.edwards@gmail.com)
:maturity: 20150910
:requires: none
:platform: all
'''
from __future__ import absolute_import

import logging

LOG = logging.getLogger(__name__)

__virtualname__ = 'module_name'


def __virtual__():
    '''
    Determine whether or not to load this module
    '''
    if __salt__['grains.get']('kernel:Linux'):
        return __virtualname__


def _private():
    '''
    "Private" function; only callable within this module
    '''
    LOG.debug('Executing the _private function')

    ret = {}
    return ret


def public(*args, **kwargs):
    '''
    "Public" function; available to Salt, ie; module.public

    CLI Example:

        salt '*' module.public
    '''
    LOG.debug('Executing the public function')

    ret = _private()
    return ret
```

2.10.3 Running Commands & Executing Modules

Often times a custom execution module is simply a wrapper around a command line utility. This means that "under the hood" Salt is simply executing an existing command with certain options. When you realize how this works your first thought in regards to development might be "Perfect. So I'll use *subprocess* and call the binary..." While that may be the right approach in other cases, Salt makes this simpler. Salt makes all other loaded modules available to your custom module. This means you can call any other available Salt module through your Salt module, including *cmd* to

run arbitrary commands. Please do not use *subprocess* in your custom module unless you have a very good reason to do so. Use the existing *cmd* module to execute arbitrary commands. An example might be as follows:

```
cmd = '{0} {1} {2}'.format('egrep', string, filename)
ret = salt['cmd.run'](cmd)
```

This function does not process commands through a shell unless the *python_shell* flag is set to *True*. This means that any shell-specific functionality such as 'echo' or the use of pipes, redirection or &&, should either be migrated to *cmd.shell* or have the *python_shell=True* flag set here.

Note: The use of *python_shell=True* means that the shell will accept *_any_* input including potentially malicious commands such as 'good_command; rm -rf /'. Be absolutely certain that you have sanitized your input prior to using *python_shell=True*

2.10.4 Writing States

2.10.5 Writing Returners

2.10.6 Writing Renderers

2.10.7 Writing Grains

2.11 Configuration Options

2.11.1 Master Config

2.11.2 Minion Config

2.11.3 Syndic Config

CHAPTER 3

About The Author

CHAPTER 4

Copyright