Encrypting Hadoop Data with zNcrypt

*The purpose of this guide is to provide a (moderately) simple step-by-step overview for installing and configuring Gazzang/Cloudera's zNcrypt data-at-rest encryption utility for the Hadoop/CDH platform.*

**Encryption Overview**

Cloudera's zNcrypt (soon to be Navigator Encrypt) utility provides the following features:

* Transparent data-at-rest encryption for nearly any Linux file system object.
* Integrated key management through zTrustee Server (soon to be Navigator KeyTrustee Server).
* Process-based Access Control Lists (ACLs), which provide granular provisioning of access to encrypted data.

**Note!** zNcrypt does not offer data-in-motion encryption or full-disk encryption.

**The Argument for Encryption**

The typical use-case for data-at-rest encryption (beyond compliance requirements) is to protect against a physical threat vector. A physical threat vector would be a "malicious actor" (hacker, thief, etc.) attempting to gain access to a server or storage device. If the device is stopped, where the encryption key is not loaded, then all data is fully obfuscated and unreadable to anyone without access to the encryption key. However, if the device is running (as is usually the case), then the encryption key is loaded into memory and the data is accessible to anyone with read/write permissions to the appropriate directories.

**Where zNcrypt is Different**

zNcrypt aims to solve multiple issues beyond what generic encryption provides:

* With generic encryption, there would be nothing stopping an administrative user from accessing any of the encrypted data on the system. This means that a system administrator would have full read/write access to any sensitive HIPAA or PCI information that is on the system. This is typically a major violation in separation-of-duty requirements.
* System administrators typically have to maintain access to encryption keys in the case servers need to be restarted or brought down for maintenance.

Due to these issues, encryption can incur major operational overhead in order to remain effective and secure.

**zNcrypt Configuration Overview**

The typical workflow for installing/configuring zNcrypt usually looks similar to:

1. Install the zNcrypt utility
2. Register zNcrypt with a zTrustee/KeyTrustee Key Management Server
3. Create the encrypted partitions (where the data will be stored)
4. Create the necessary ACL rules to provision access to the (soon-to-be) encrypted data
5. Encrypt data!

Depending on the type of data that you would like to encrypt, these steps usually remain the same. Specifically for HDFS encryption, they can be modified slightly:

1. Install the zNcrypt utility on each Hadoop Datanode (or anywhere you are storing data that needs encrypting)
2. Register zNcrypt with a zTrustee/KeyTrustee Key Management Server
3. Create encrypted partitions on each HDFS drive.
4. As the Datanode process is running, create the necessary ACL rules
5. Stop the Datanode service on the specific node
6. Encrypt your HDFS data directories (usually anything listed under dfs\_datanode\_data\_dir configuration in the hdfs-site.xml file)
7. Start the Datanode service

**Installing zNcrypt**

In order to install the zNcrypt utility, you will need to install all of the necessary system and package dependencies. If you want to go ahead and jump in, you can run the following command from the Linux CLI on a test machine:

curl -sL https://archive.gazzang.com/deployment/zncrypt-install.sh | sudo bash

If that worked correctly, you can skip to the registration section below. To walk through the install manually, continue reading...

**System Requirements**

The system requirements differ between Linux distribution and version:

**Supported Linux Kernel Versions**

* RHEL/CentOS systems require 2.6.18-92 or above
* All other distributions require 2.6.19 or above

**Supported Linux Distributions and Versions**

* Red Hat Enterprise Linux 5.4+, 6.x
* CentOS 5.4+, 6.x
* Ubuntu 10.04, 12.04, 12.10, 13.04, 13.10
* Debian 6

And with limited support for Amazon Linux 2013.09.2 and 2014.03.1 (it'll work, but it might require some tweaking).

**Installing Dependencies**

The following dependencies are required and will be handled through package management (so you usually won't have to explicitly download/install them):

**Ubuntu**

* keyutils
* ecryptfs-utils
* cryptsetup
* openssl
* haveged
* libc6
* libcurl3
* libgpgme11
* libpcre3
* libssl1.0.0
* dkms
* make
* gawk

**RHEL/CentOS**

* cloog-ppl
* cpp
* ecryptfs-utils
* gcc
* glibc-devel
* glibc-headers
* keyutils
* libgomp
* lsof
* mpfr
* ppl
* trousers
* dkms

The following dependencies are required, and will **not** be resolved through package management (you will have to install them yourself):

**Ubuntu**

* linux-headers-$(uname -r)

**RHEL/CentOS**

* kernel-headers-$(uname -r)
* kernel-devel-$(uname -r)

Where $(uname -r) above signifies the version of the running Linux kernel.

**Adding the Repo**

Once the Linux kernel headers are installed, the rest of the dependencies can be resolved through package management:

**Ubuntu**

From a terminal, execute:

source /etc/lsb-release && echo "deb https://archive.gazzang.com/ubuntu/stable $DISTRIB\_CODENAME main" | sudo tee -a /etc/apt/sources.list

**RHEL/CentOS**

From a terminal, execute:

sudo cat << EOF > /etc/yum.repos.d/gazzang.repo [gazzang] name=RHEL \$releasever - gazzang.com - base baseurl=https://archive.gazzang.com/redhat/stable/\$releasever enabled=1 gpgcheck=1 gpgkey=https://archive.gazzang.com/gpg\_gazzang.asc EOF

**Installing the Client**

Once the repositories are appropriately configured, the next step is to simply install:

**Ubuntu**

sudo apt-get update && sudo apt-get install zncrypt haveged -y sudo service haveged start # haveged is used to keep the entropy pool on the system high (faster key generation)

**RHEL/CentOS**

sudo yum install zncrypt -y sudo chkconfig --level 2345 zncrypt-mount on

***Note!*** On some RHEL/CentOS systems, it might be necessary to also configure yum to use the EPEL repositories. For more information on EPEL (and how to configure yum), please see [here](https://fedoraproject.org/wiki/EPEL/FAQ#How_can_I_install_the_packages_from_the_EPEL_software_repository.3F).

**selinux Support**

At this time, selinux is **not supported** when used with zNcrypt. Our official recommendation is to deactivate selinux when deploying zNcrypt (the bash installation script will do this automatically).

To deactivate selinux manually, you will need to run the following commands:

# Disable selinux immediately (will be disabled until reboot) setenforce 0   # To disable across a reboot (recommended) sed -i.old s/SELINUX=enforcing/SELINUX=disabled/ /etc/selinux/config

If it is a customer requirement to use selinux with zNcrypt, please send all relevant information to [support@gazzang.com](mailto:support@gazzang.com).

**Firewall Configuration**

If a firewall is configured locally on the machine (typically iptables or ufw), you will need to make rules for the following exceptions:

* Outbound traffic on port 80 (HTTP)
* Outbound traffic on port 443 (HTTPS)

When installing in a customer environment, please make sure to make the necessary network changes **prior** to installing zNcrypt.

**Registering with a Key Server**

Once zNcrypt is fully installed, it is time to register with a zTrustee Server to store encryption keys. During this process, you will need to specify a "master passphrase" that will be used to lock the zNcrypt configuration. Ideally the master passphrase will be unique to the system, and separate from the Linux system administrator's password.

***Note!*** If you are going to be using the bash scripts to aid in the configuration process, a master passphrase will be auto-generated for you. Once the configuration is completed, make sure you change it (or at least remove it from the file system).

To get started, run the following in your terminal:

sudo zncrypt register -s <hostname of key server> -o <organization name> --auth= <authorization code>

Where:

* *<hostname of the key server>* is the fqdn of the zTrustee Key Server you want to register against. Ex: ztdemo.gazzang.net
* *<organization name>* is the name of the organization you would like to register the client against. This is a server-side setting, so, if you do not know this, you will need to contact a zTrustee Server admin. Ex: cloudera-testing
* *<authorization code>* is the authorization code/secret associated with the organization above. This is also a server-side setting, so contact your administrator if you're not sure what to put here. This is typically going to be a long string of garbled text.

**Testing Credentials**

The following command can be used for testing and**internal use only**. This server is cycled regularly, so **only use on short-lived or test machines**. Consider yourself warned.

sudo zncrypt register -s ztdemo.gazzang.net -o cloudera --auth=/Ey5NSA0YnLycApTIcejQQ==

And then follow the prompt to create a master passphrase and register with the key server. Once completed, your client is fully registered and operational.

**Preparing Drives for Encryption**

Once the client is registered, you can start encrypting data. Before we can encrypt any data, though, we'll need somewhere to store it. This is where the `zncrypt-prepare` operation comes in, and what we like to call "preparing drives." Before we can prepare any drives, though, we need to go over the two different types of encryption: **file** and **block**.

**The Different Types of Encryption**

**File-level Encryption**

Data is stored inside of a virtualized, encrypted file system that encrypts each individual file with a different symmetric encryption key. This key is then embedded into the header of the file, and encrypted with the 'mount key' for the file system.

**Pros**

* Very simple and flexible to deploy (requires no pre-allocation of space or forethought).
* Easy to move around and backup in the case of an emergency.

**Cons**

* Can be slower when the file system is under heavy IO load (ie Hadoop).

File-level encryption is best paired with testing environments, relational databases (MySQL, PostgreSQL, etc.), or systems with smaller datasets.

**Block-level Encryption**

Data is stored inside of an encrypted block device, where one symmetric encryption key is used to encrypt all of the data in the device. This key is then embedded into the header of the device, and encrypted using a mount key.

**Pros**

* Very performant when the file system will be under heavy IO load.

**Cons**

* Harder to deploy, as it requires a 'clean' device to store encrypted data on. This is the main pain-point when installing into pre-existing customer environments.
* Very difficult to backup and restore in physical or otherwise restrictive environments.

Block-level encryption is best paired with systems undergoing performance testing, systems that have very large datasets (1TB+ per drive), or "NoSQL" environments (Hadoop, Cassandra, MongoDB, etc.).

**Creating the Encrypted File System**

**File-Level Encryption**

To get started, we will need to create the directories where we will want to store (and access) our data. A good default is:

* /var/lib/zncrypt/encrypted as the mount location for the encrypted file system (this is where the data will be read/written from/to)
* /var/lib/zncrypt/.private as the storage location for the encrypted file system (this is where the raw encrypted data will be stored)

***Note!*** The encrypted file system will share the same size/space requirements as the partition where the **storage**location is placed, so please keep that in mind if you are tight on available space.

Once we know where we want to store the encrypted data, let's prepare the volume:

sudo mkdir -p /var/lib/zncrypt/encrypted /var/lib/zncrypt/.private sudo zncrypt-prepare /var/lib/zncrypt/.private /var/lib/zncrypt/encrypted

Where the 'zncrypt-prepare' command takes the storage, then mount directory. Alternatively, running zncrypt-prepare by itself will open an interactive prompt to configure the encrypted file system.

Once the volume is fully prepared, we are ready to start encrypting data.

**Block-level Encryption**

In order to utilize block-level encryption, you will need to have an available storage device to act as our "vault" for encrypted data.

***Note!****This device will be formatted/cleaned during the encryption process, so it is important to make sure that all relevant data has been backed-up prior to proceeding.*

Similar to file-level encryption, we will need to create a directory that will act as a mount-point for our encrypted file system. A good default (for single-device deployments) is:

* /var/lib/zncrypt/encrypted as the mount location for the encrypted file system (this is where the data will be read/written from/to)

Now we will need to prepare the device for encryption:

sudo mkdir -p /var/lib/zncrypt/encrypted sudo zncrypt-prepare /dev/sdc1 /var/lib/zncrypt/encrypted

Where the zncrypt-prepare command takes the device/partition location ('/dev/sdc1' in the example above), then the mount directory.

***Note!***By default, the 'zncrypt-prepare' command will create an EXT4 file system on the storage device, but this can be configured to another valid file system by specifying the '-t' option ('-t xfs' for XFS, etc).

Once the device is prepared, we are ready to start storing encrypted data.

**Provisioning ACL's**

This step can be performed either pre or post encryption. It is typically easier to do this before the encryption is in place (assuming you know what ACL's you want to use), so that once the data is encrypted it is ready to be accessed immediately. One of zNcrypt's main features is the process-based access provisioning through the use of Access Control Lists, or ACL's for short.

**Command Overview**

To get started, let's take a look at a very simple 'acl --add' command:

sudo zncrypt acl --add -r "ALLOW @mycatrule \* /bin/cat"

As you can see above, there's not much to it:

* zncrypt acl --add
  + This just signifies that we will be adding a zNcrypt ACL rule. The opposite is --del.
* -r
  + Signifies the rule we will be adding.
* "ALLOW @mycatrule \* /bin/cat"
  + This is the ACL rule. Lets take a closer look:
    - ALLOW  
      * We will be ALLOW-ing access to the encrypted data. DENY could have also been used here (all actions are denied by default, but this could be useful when layering on multiple rules simultaneously).
    - @mycatrule  
      * This is the category name, where the @ signifies a category and mycatrule is the category name. Categories are user-defined groups used to assign rules to different sets of data.
    - \*  
      * We can specify to which directories this ACL applies here. Usually it is best to leave this as simply \*.
    - /bin/cat  
      * This is the binary or process that will have access to the encrypted data. In this rule, we want the /bin/cat binary to have sole access to the encrypted data. This can be any binary on the file system.

***Note!*** When an ACL rule is added, the binary ( /bin/cat in the example above) is fingerprinted with a SHA256 digest. If the fingerprint of the binary changes (for example, a system update occurs) then access to the encrypted data will be broken. This is important to remember and is the most common issue that we see customers run into. If access is ever broken, you can restore access by running:

sudo zncrypt acl --update

You might also have to restart whatever lost access as well, but that varies by platform (for HDFS/DataNode processes, you typically don't need to, but for MySQL you do).

**ACL's in Hadoop**

For Hadoop, setting ACL rules becomes slightly more complicated. The reason for this is because, since Hadoop is a java-based application, all Hadoop processes on a system (and all java processes, for that matter) will share the same java binary. This means that simply adding an ACL rule java will only keep out non-java processes, but all other java processes will still have access to the encrypted data (typically an undesirable feature).

To bypass this issue, zNcrypt has a secondary ACL attribute called a 'profile.' An ACL profile will allow you to further granulize access provisioning to the encrypted data even further by specifying UID, GID, and other CLI parameters. It is the zNcrypt profile feature that allows us to protect Hadoop and any HDFS data that we want protected.

A typical profile will look something like this:

{ "uid":"496", "comm":"java", "cmdline":"/usr/lib/jvm/jre-openjdk/bin/java -Dproc\_datanode -Xmx1000m -Dhadoop.log.dir=/var/log/hadoop-hdfs -Dhadoop.log.file=hadoop-hdfs-datanode-vagrant-centos64.vagrantup.com.log -Dhadoop.home.dir=/usr/lib/hadoop -Dhadoop.id.str=hdfs -Dhadoop.root.logger=INFO,RFA -Djava.library.path=/usr/lib/hadoop/lib/native -Dhadoop.policy.file=hadoop-policy.xml -Djava.net.preferIPv4Stack=true -server -Dhadoop.security.logger=INFO,RFAS org.apache.hadoop.hdfs.server.datanode.DataNode" }

To get started, you will need to capture the profile of a running Hadoop system. If the datanode process is stopped on your machine, then you will need to start it now.

To collect the profile, we need to know the PID of the process we want to profile. In this case, let's use the datanode process:

sudo -s # Capture PID(s) of datanode process. Replace [d]atanode below with whatever you want to capture ([h]adoop, etc.) PIDS="$(ps -ef | grep -i [d]atanode | awk '{ print $2}')"   # Generate profile(s), where we use a for loop just in case we get more than one result for pid in ${PIDS[@]}; do zncrypt-profile --pid=$pid > /tmp/znc\_profile\_$pid; done   # Attach the profiles to the ACL's for profile in "$(ls /tmp/znc\_profile\_\*)"; do zncrypt acl --add -r "ALLOW @hadoop \* $(grep "cmdline" $profile | tr [=\"=],[=:=] " " | awk '{ print $2 }' | xargs readlink -f)" --profile-file=$profile; done   # If you added ACLs prior to attaching profiles, make sure you remove them once completed. exit

Once all of the profiles have been added, you can list out the full set of ACL's by executing:

sudo zncrypt acl --list --all

Which should successfully list out the full profile for each ACL.

**Encrypting Data (and, more importantly, Hadoop)**

Once the ACL's have been added, we'll want to encrypt some data.

***Note!***Before encrypting ***ANYTHING***, you will want to make sure and **stop** all services that might need to access the target data. In the case of Hadoop, you will need to stop (at least) the DataNode process on each machine prior to encryption. Accessing data while it is being encrypted can lead to data corruption or loss of data.

**Command Overview**

To encrypt any data, we will need to use the zncrypt-move command. This command has 4 parameters:

zncrypt-move [action] @[category] [target] [mount]

* The **action** to complete (encrypt or decrypt)
* The **category** to associate for the data (in the example above, this was @hadoop
* The **target** data that we would like to encrypt (in the case of Hadoop, typically a HDFS data directory)
* The **mount** location where we would like to store the encrypted data (the encrypted mount-point from the zncrypt-prepare command above). ***Note!*** Be sure to keep size and space requirements in mind. For example, make sure the '/data/N/dfs' data is being stored on the '/data/N/encrypted/mnt' partition (for all data directories).

As an example:

sudo zncrypt-move encrypt @hadoop /dfs/dn /var/lib/zncrypt/encrypted

Where we are encrypting the '/data/dfs' directory against the '@hadoop' ACL set, and the data will be stored inside of the '/data/encrypted/mnt' encrypted partition. Once this command is executed, all of the data inside of the '/data/dfs' directory will be moved into the '/data/encrypted/mnt' partition (with a symbolic link pointing to the new location).

***Note!***Typically the deciding factor for how long an install will take depends on the amount of data that you will need to encrypt. As a rule of thumb, we like to say that the encryption process will take around the same amount of time as it would take to copy the data. Make sure you keep this in mind, especially when encrypting large datasets from a remote terminal (try to use screen or nohup when possible).

**Encrypting an Entire Cluster**

When encrypting an entire Hadoop cluster, it is important to consider three things:

* The number of data nodes in the cluster  – This will be the number of zNcrypt clients that you will need to install.
* The number of HDFS drives per data node – This will be the number of encrypted mount-points per zNcrypt client.
* The amount of raw data that is on each HDFS drive – This will be the determining factor in how long the encryption process will take.

By taking into account the three factors listed above, we can start to plan:

* How we should install the zNcrypt clients, and whether or not to use a deployment tool (Parcels, Chef, Ansible, Puppet, etc).  
  + If this is a small test/demo cluster, then it might not be necessary that we spend the time and bandwidth configuring Chef/Puppet, for the install.
  + If this is a large prod/QA cluster, then we will definitely want to spend the extra time setting up a repeatable process to configure each data node.
* What form of encryption we should use (file-level or block-level).
  + For small test clusters, file-level encryption should be sufficient.
  + For large production clusters (or anything that will be used to provide benchmarks), block-level encryption should be used over file-level encryption.
    - ***IMPORTANT!*** When using block-level encryption, you will need to consider the amount of data *currently* on the HDFS drives that will need to be encrypted. In order to use block-level encryption, you will need to reformat each device that will be used to store encrypted data. This means that you will either need to:
      * Migrate the data off of the drives you will want to encrypt (which will incur its own time and maintenance overhead for copying the data off then back on).
      * Reformat the drive without migrating the data, and then let HDFS replicate the data back through the cluster (can incur a time overhead to wait for replication).
* How long the entire process (installation through encryption) will take, and whether or not we will need to allocate downtime to a specific set of nodes (or the entire cluster).
  + For small clusters (or clusters that are not being used), this might not be an issue. You can take each node down one at a time to encrypt, or you can take the entire cluster down.
  + For large production clusters (or clusters that are in active use), you will need to plan how/when to take each node down for encryption. Depending on how in-use the cluster is, it might be more efficient to request downtime for a subset of nodes, and work your way through the cluster.

Now that we have a "plan of attack" on encrypting the cluster, it's time to execute our plan.

**Bash Scripting**

If you would like to use Bash scripts to deploy and configure zNcrypt, we have a Github repository filled with scripts to help [here](https://github.com/gazzang/zncrypt-deployment). With the important ones being:

* [zncrypt-install.sh](https://github.com/gazzang/zncrypt-deployment/blob/master/zncrypt-install.sh)– This script can be used to install zNcrypt on almost every platform zNcrypt is supported (requires Internet access).
* [zncrypt-autoconfigure.sh](https://github.com/gazzang/zncrypt-deployment/blob/master/zncrypt-autoconfigure.sh)– This script can be used to configure zNcrypt for every step of the process cited above. You specify your environmental attribute at the top of the script, and then execute after stopping the data node process. A sample configuration to accomplish the example steps above (using file-level encryption) might look similar to:

**zncrypt-autoconfigure.sh**

###################################### # global configuration variables (change these to fit your environment) ######################################   # zTrustee Server to register against. This will default to [ztdemo.gazzang.net] if not set. keyserver="ztdemo.gazzang.net" # Registration credentials for authenticating with the keyserver org="cloudera\_wiki" auth="/Ey5NSA0YnLApTIcejQQ==" # Storage and mount locations (can be either an array or string) storage=( "/data/1/encrypted/.private" "/data/2/encrypted/.private" ) mount=( "/data/1/encrypted/mount" "/data/2/encrypted/mount" ) # Data to encrypt/protect to\_encrypt=( "/data/1/dfs/dn" "/data/2/dfs/dn" ) # ACL categories to set category=( "hadoop" "hadoop" ) acl\_binary=( "/usr/lib/jvm/java-1.7.0-openjdk-1.7.0.55.x86\_64/jre/bin/java" "/usr/lib/jvm/java-1.7.0-openjdk-1.7.0.55.x86\_64/jre/bin/java" ) # Please note, the last 5 variables listed above correspond 1:1 with eachother. # Every array must be of equal length. For example, if you would like to encrypt # more than one target against the same mount, you must list the mount/storage # location twice to match both targets.

***WARNING!*** When using the 'zncrypt-autoconfigure.sh' script, you will want to stop the 'hadoop-hdfs-datanode' process on all target machines **before** executing the script. The datanode directories might still be in use, and could potentially corrupt the file system if left active during encryption.

**Creating Your Own Scripts**

If you would like to script out any of the above steps, you can "pipe" the master password into any zNcrypt command:

# Cat from a password file (be sure to remove the password file when configuration is completed) cat mypassword-file | zncrypt ...   # Or echo directly (will show up in logs, so use only for testing) echo "mypassword" | zncrypt...

**Utilizing Configuration Management Tools**

For large deployments, you will definitely want to use a standardized method to install and configure zNcrypt. As of this writing, Gazzang has deployment samples written for the following configuration tools:

* [Chef](https://github.com/gazzang/cookbooks)
* [Puppet](https://github.com/gazzang/puppet)
* [Ansible](https://github.com/gazzang/ansible)

**Utilizing Cloudera Manager**

For integrations with Cloudera Manager, Gazzang has provided a parcel repository for easy deployment of the zNcrypt binaries. To configure Cloudera Manager to use the zNcrypt parcel, you can add the following repo to your parcel repository configuration:

**zNcrypt Parcel Repository**

http://archive.gazzang.com/parcels/latest/

Once added to your Cloudera Manager instance, you can then distribute and activate the parcel throughout the cluster. The parcel repo is slightly outdated since the acquisition, so you might be a version behind until the Cloudera engineering team is able to integrate it into their existing parcel distribution model.

***Note!****Once the parcel is activated, zNcrypt will only be****installed****. You will have to use a separate tool to configure zNcrypt (until a CSD is created).*

**Pitfalls**

Below are some common pitfalls that we see customers run into the most.

**Forgetting to Update the ACL Fingerprint**

When creating an ACL rule, zNcrypt will automatically take a SHA-256 digest of the target binary to prevent malicious tampering. This safety precaution would help prevent a rogue administrator from attempting to hot-swap a trusted process with malicious one. While this feature does prevent against malicious tampering, it also can add in operational complexity when the fingerprint changes. The most common reasons why a fingerprint will change:

* A system update occurred (apt-get upgrade, yum update, etc.), whether it was intentional or not
* The prelink subsystem is active, which can (at random times) rearrange binary data in order to speed up application startup

To see if a fingerprint has changed, you can execute:

zncrypt acl --list

Where a '!!' symbol denotes a fingerprint mismatch. Once a fingerprint changes, you will need to execute an ACL update by running:

zncrypt acl --update

***Note!*** You might also need to restart the service or application that was using the encrypted data in order to restore access to the encrypted data.

**zNcrypt Registration Never Completing**

If the 'zncrypt register' command is hanging or not returning in a reasonable amount of time, there are a few steps you can take to identify the issue. During the register command, there are a couple things that can go wrong:

* zNcrypt is reaching out to the Navigator KeyTrustee Server to enable storage of encryption keys. In order for this to complete successfully, make sure that the necessary network changes have been made, and that the zNcrypt server can initiate traffic over ports 80 and 443 to the hostname or IP of the specified key server.
* zNcrypt is generating an 2048-bit RSA GPG public/private key pair during registration. This key pair will be used for encryption of all data stored inside the key server. This process, in order to be as secure as possible, harvests entropy from the system entropy pool (typically '/dev/random'). On a system that is not very active, the entropy pool might become depleted very quickly. In order to combat low system entropy, make sure the following services are enabled:
  + On Debian/Ubuntu, haveged.
  + On RHEL/CentOS, rngd or haveged (requires EPEL).
* When all else fails, it might be an issue with the key server. To make sure the key server is up and functioning properly, you can run a:

curl -k https://ztdemo.gazzang.net/?a=fingerprint

To make sure that the services are up and functioning properly.

If you continue to see issues, send a note to [support@gazzang.com](mailto:support@gazzang.com) for troubleshooting help.

**Checking Connectivity to the Key Server**

If you are unsure whether or not you can reach the key server, there are a few commands you can run to verify connectivity.

To check connectivity over port 80:

curl http://ztdemo.gazzang.net

Which should result in a 404 error (verifying Apache is servicing requests and that traffic is allowed):

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN"> <title>404 Not Found</title> <h1>Not Found</h1> <p>The requested URL was not found on the server. If you entered the URL manually please check your spelling and try again.</p>

To check connectivity over port 443:

curl -k https://ztdemo.gazzang.net/?a=fingerprint

Which should result with the web application returning the GPG fingerprint of the server:

4096R/D180C781731952E005EEA7992EC94D6C21FB518D

**Skipping SSL Validation During Registration**

If you would like to stop zNcrypt from checking for valid SSL certificates, you can disable this option during registration with:

zncrypt register ... --skip-ssl-check

***Note!*** that this command should **never** be used for production or non-testing environments. This makes clients much more susceptible to man-in-the-middle attacks, which could compromise security.

**What Happens When You Lose Connection to the Key Server**

If, for any reason, your client loses connection to the key server, you can still manually mount the encrypted drives locally by using the master password. To start zNcrypt manually, run:

mount.zncrypt /path/to/storage /path/to/mount

For example (using the same storage/mount directories from above):

mount.zncrypt /var/lib/zncrypt/.private /var/lib/zncrypt/encrypted

**Backup Best-Practices**

To backup or migrate zNcrypt encrypted data, you will need three items:

* A copy of the zNcrypt configuration directory (/etc/zncrypt), which contains zTrustee configuration data as well as encrypted copies of the Key Encryption Keys (KEKs).
* The master password used to encrypt the data **when the backup was taken**. If you end up changing the master password down the road, the old backup will not be able to be accessed with the new master password.
* A backup of the zNcrypt **storage** directory for the target encrypted file system (/var/lib/zncrypt/.private in the example above). The mount location is not important.

The only requirement for backing up an encrypted volume is to stop the zncrypt-mount service, which will unmount the encrypted partitions and free up any open file descriptors. You can stop the service by issuing:

service zncrypt-mount stop

When restoring encrypted data to a server, it is usually best to restore the objects above to the same locations they were on the original server. If you are not able to do that, then you will have to manually modify the zNcrypt configuration so that zNcrypt knows what encryption keys should be used for which directories. The key items that you will need to modify are the following:

* /etc/zncrypt/control
* /etc/zncrypt/ztab
* /etc/zncrypt/ztrustee/deposits

**Unregistering Encrypted Devices**

If you would like to "unregister" or "un-prepare" an encrypted volume, the zncrypt-prepare command has a built-in utility to handle this. Using the same mount/storage directories from before, we could remove them from the zNcrypt configuration by issuing a:

zncrypt-prepare -u /var/lib/zncrypt/.private

**Compatibility with Sophos Anti-Virus**

If running zNcrypt on a system that is also running Sophos anti-virus software, you will need to add eCryptFS to the Sophos 'skip' list. To do this, you can run:

# Add the exception to talpa echo "skip\_list=+ecryptfs" >/opt/sophos-av/talpa/override/talpa\_vfshook.options   # Restart the service service sav-protect restart