Global Information Assurance Certification Paper

Diskless Cluster Computing: Security Benefit of oneSIS and Git

GIAC (GSEC) Gold Certification

Author: Aron Warren, aronwarren@gmail.com
Advisor: Johannes B. Ulrich, Ph.D.

Accepted: April 12th 2012

Abstract

From desktops to servers to even the largest of supercomputers managing a constantly evolving diskless computing environment is usually quite challenging. Not only does the requirements for machines change over time but the need for timely security patching can make the maintenance unbearable. Creating a single diskless Linux image that is both easily customizable to different classes of machines, eg. clients versus servers, is easily solved with the use of oneSIS. Git, a version control system, provides a very easy method for upgrading, downgrading and verifying diskless images. The joining of these two software packages can easily save hundreds of hours for an administrator.

1. Introduction

This paper introduces the joining of two software packages, oneSIS and Git. Each package by itself is meant to tackle only a certain class of problem. What is amazing is when both are combined the resulting union offers an administrator of a diskless computing environment an easy to use and powerful toolset to defend against the efforts of hackers.

1.1. OneSIS

OneSIS is an open source project that was developed by Josh England while working for Sandia National Laboratories as a method for booting diskless compute nodes in a supercomputer from a single, or possibly multiple, boot servers. The flexibility gained by remotely sharing a root file system that can be custom tailored for each client or group of clients will be demonstrated in this paper. While oneSIS is primarily written for a handful of the most commonly used Linux distributions, namely RedHat, SuSE, Debian, Ubuntu and Gentoo (England, 2008) any Linux distribution can be ported in just a few minutes.

This software package can be beneficial for many types of deployments and its' use need not be limited to supercomputers. It can be used in many different diskless scenarios such as desktops in hotel business centers, airline kiosks, college computing terminals, and even business employee desktops. Another example class of machine could be static, diskless, redundant web servers designed to offer files from a central file server. A final example is the bind configuration files residing in a oneSIS disk image which allows offering of the same DNS configuration from multiple servers.

Deployment of diskless desktops and servers is easy to achieve with oneSIS though not a focus of this paper. This paper will focus on a more generic class of computers - the compute node class that reside in compute clusters, which are a form of

supercomputers. Compute nodes are one of the simplest types of computers to deploy in that the operating system (OS) stack tends to be a minimal set of features and functionality. The reason for this is the more rouge processes that perform OS scheduler detours away from the parallel application can cause significant problems (Beckman, Iskra, Yoshii, Coghlan, & Nataraj, 2008, pp. 4-5).

Differences between compute nodes tend to be limited to hostnames and network addresses. Thus with relative ease administrators can deploy an entire cluster in the matter of days to a few short weeks depending upon the variance from the standard OS installation. While oneSIS installation is a short and easy process, the debugging of boot problems and differences between nodes are usually the most time consuming part of the project.

OneSIS is not only for diskless machines, populating diskful machines is also a supported feature. Diskful installs are quite similar to the way ROCKS project handles building clusters. While very simple to deploy a ROCKS cluster, the benefits of having a single OS image being managed and the auditability provided by Git is lost. Local disks in compute nodes, as well as any of the above scenarios, mean another significant mechanical failure point in computers (Schroeder, 2007, p. 10). When dealing with a handful of hard drives the failure rate is probably rather small. For 10,000 node clusters, the mean time between failures can become a problem to simulations if the OS dies out from underneath the job. Hard drives also mean another point of accountability and liability let alone the resources needed to ensure proper destruction when the hard drive is decommissioned or replaced.

Before moving onto the introduction of Git, a conceptual framework needs to be established. When a reference to a oneSIS image is made, the reference is to set of files and directories that form the root partition. This *image* contains all the files necessary for the OS to run. It also contains all of the configuration files necessary for the image to give a personality to the diskless clients. For example, the oneSIS image contains files such as /etc/hosts, /etc/fstab, /bin/login, and so forth. Section 2.1 will show how this

image is created.

1.2 Git

The next software package to be introduced is Git. Git is an open-sourced distributed version control system management system developed initially by Linus Torovolds for development of the Linux kernel (Google, 2007). Git is not the only versioning system that is available. Bitkeeper, CVS, Mercurial, Subversion and Perforce are just a handful of commercial or open source software packages available (Fish, 2011). Of the two models available: client-server or distributed, the distributed model gives us the most benefit for our application. Instead of requiring a centralized repository for commits and checkouts to be processed Git uses a distributed repository through the use of cloning. Each individual "has a complete copy of all historical revisions of every file" (Loeliger, 2009, "The Birth of Git," para 13). This ability to clone a complete repository lends itself to users ability to work on their own code changes or branches offline. It also allows them to later merge those changes into a shared repository or another individual's repository.

While Git is more thought of for software development the approach taken here is to take the revision control concept and apply Git management to an operating system installation, in particular the image produced by oneSIS. In taking the concept of revision control over an operating system the following gains are made:

1) Ability to detect any content change in the oneSIS image including ownership and file permissions changes. Git's native SHA1 file checksum usage lends itself to providing image integrity (Google, 2007). Also knowing the initial image commit's SHA1 lends itself to a thoroughly trusted commit tree as each commit is tied to a previous commit or commits.

2) Ability to move forward and backward along the oneSIS image's history for testing in a development environment to rolling back problems in a production environment.

3) Ease in upgradability of the oneSIS image when moving from a development to a production environment, or from one production version to another.

2. Server Setup

To begin our example oneSIS installation a server that offers the oneSIS image is needed. A server node needs to be installed with an operating system. In this paper CentOS will be used as the example for both the server node and the diskless client nodes. CentOS provides a simple installation process with minimal installation questions. The most important configuration option is the partition layout. One recommendation is if there are plans to utilize oneSIS and Git to the maximum advantages, have at least one very large partition that can contain many copies of the oneSIS image, referred to simply as images. The partition used here will be created at /var/lib/oneSIS/images. In the end there could be a production image, a development image, several backup copies created during upgrade time and a few for historical reasons. That could roughly be 8 images * 6GB = 48GB. Today's drive capacities are more than large enough to allow a lot of playing room but legacy systems could pose a challenge.

The server also needs two network interfaces, one to the outside world and one to a private network for the compute nodes. While not necessarily applicable to every deployment scenario, for this paper the compute cluster model will continue to be used. The other major upfront decision to make is the amount of packages to be installed. Best practice is to install the minimal number of packages necessary. Keep in mind that whatever packages you choose during the server installation as far as RPMs will also show up in your oneSIS image. Choosing a minimal software list lends your compute nodes to having a very nice, small, manageable footprint.

2.1 OneSIS Installation and Image Creation

Once the installation completes and the system has rebooted the oneSIS rpm needs to be installed.

```
[root@server]# rpm -Uvh oneSIS-2.0.2-1.noarch.rpm
```

The next step is to tell oneSIS to copy the root file system. This command will copy every local file system starting from / and place it in the location specified by the *-l* option.

```
[root@server]# copy-rootfs -l /var/lib/oneSIS/
images/image-prod
```

Next create the sysimage.conf file and place it in etc in your new image. The sysimage.conf file is the file used to give a compute node its' personality. Any files or directories listed here will be symlinked to a corresponding location in a ramdisk mounted under /ram. /ram is used to store dynamic files and personality specific configuration files. Sample sysimage.conf files are included with the oneSIS rpm but here is the file being used here as an example:

```
# distribution
DISTRO: RedHat EL-6 –sp
RAMSIZE:  500m

NODECLASS_REGEX         node\d+    cluster
NODECLASS_RANGE        node[1-5]  cluster.compute

# Writable system areas
LINKFILE: /etc/adjtime -d
LINKDIR: /root -d
LINKDIR: /tmp -d
LINKDIR: /var/tmp -d
LINKDIR: /var/cache -d
LINKDIR: /var/empty -d
LINKDIR: /var/lib/nfs -d
LINKDIR: /var/lock -d
LINKDIR: /var/log -d
LINKDIR: /var/run –d
LINKFILE: /etc/fstab
```

The administrator's guide goes into excellent description of each command and

configuration option but briefly here are the three options going to used:

1. NODECLASS_REGEX: oneSIS defines the class a node belongs to based solely on the node's hostname (England, 2011). In our example a hostname of *node* belongs to the *example* class and the *compute* subclass.

2. LINKFILE: Creates a file in /ram and changes the corresponding file in the image into a link pointing to the file in /ram (England, 2011). The *-d* option says to copy all of the directory's contents into /ram.

3. LINKDIR: Creates the directory in /ram and changes the corresponding directory in the image into a link pointing to the directory in /ram (England, 2011). The *-d* option says to copy all of the directory's contents into /ram.

Upon a client node booting oneSIS will take the hostname and create a symlink from /ram to the appropriate classed file or directory in the image. Thus the personality is made complete.

```
[root@node1]# ls -la /etc/fstab
/etc/fstab -> /ram/etc/fstab
[root@node1]# ls –la /ram/etc/fstab
/ram/etc/fstab -> /etc/fstab.cluster.compute
```

Now that the sysimage.conf file has been created oneSIS needs to be told to oneSIS'ify the image. The production image will be called *image-prod*. The *-c* option is where our configuration file is located. Best practice is to have /etc/sysimage.conf symlinked to the same location inside the image. The last argument is the location of the image.

```
[root@server]# mk-sysimage -c /var/lib/oneSIS/
images/image-prod/etc/sysimage.conf /var/lib/
oneSIS/images/image-prod
```

2.2 Service Installation and Configuration

Next prepare the server to serve the image in which includes installing dhcpd,

tftpd, and tftp services.

```
[root@server]# yum install dhcpd tftp tftp-server
```

The tftpd service needs to be edited to allow it to start, so set *disable=no* as well as add some verbosity, the *-v -v -v*, in order to assist in troubleshooting compute node booting. The verbose messages will be logged to syslog.

```
[root@server image-prod]# cat /etc/xinetd.d/tftp
service tftp
{
        socket_type             = dgram
        protocol                = udp
        wait                    = yes
        user                    = root
        server                  = /usr/sbin/in.tftpd
        server_args             = -s /tftpboot -v -v -v
        disable                 = no
        per_source              = 11
        cps                     = 100 2
        flags                   = IPv4
}
```

```
[root@server]# cat /etc/dhcp/dhcpd.conf
ddns-update-style none;
use-host-decl-names on;
subnet 192.168.200.0 netmask 255.255.255.0 {
   option routers 192.168.200.2;
   group {
        filename "/pxelinux.0";
        next-server 192.168.200.2;
        host node1 {
          hardware ethernet 00:50:56:25:6D:C8;
          fixed-address 192.168.200.100;
        }
   }
}
```

To diskless boot the image, the kernel needs to be copied to /tftpboot. The kernel doesn't have to be the server's current running kernel but in this example it is.

```
[root@server]# cd /boot; cp vmlinuz-2.6.32-
71.29.1.el6.x86_64 /tftpboot/
```

The next step is to build the bootstrapping image in which is done with the mk-initramfs-oneSIS command. The *-w* option is to ensure the e1000 ethernet driver isspecifically included. The *-b* option defines the base image location. The *-f* option means to overwrite any existing initramfs file. The final two options are the initramfs name and the kernel version to be used.

```
[root@server]# mk-initramfs-oneSIS -w e1000 -b
/var/lib/oneSIS/images/image-prod/ -f initramfs-
2.6.32-71.29.1.el6.x86_64 2.6.32-71.29.1.el6.x86_64
```

Be sure that you know all of the necessary drivers needed in order to boot the compute node, the most often problematic area is the network card. This completes the steps necessary to, with some good luck, get an image to boot on the compute node.

3. Git repository setup

This section is where Git will be used to track changes in our newly created image. Git in itself does have a steep learning curve. While Git proficiency is not needed to understand the following work, additional resources are listed in the references section. Traverse to the directory to be tracked, /var/lib/oneSIS/images/image-prod in this instance, and set up a Git repository.

```
[root@server]# cd /var/lib/oneSIS/images/image-prod
[root@server image-prod]# git init
Initialized empty Git repository in
   /var/lib/oneSIS/images/image-prod/.git/
```

First step is to create a root level .gitignore of the files Git will not be tracking. Git will not track devices so directories that contain those need to be excluded and contained in that file.

```
[root@server image-prod]# cat .gitignore
dev
proc
```

Git does not track empty directories. Find any directory that is empty and populate it. An easy cheat is to create an empty .gitignore in that directory.

```
[root@server image-prod]# find . -name .git -prune
-o -type d -empty -exec touch {}/.gitignore \;
```

Since Git doesn't track devices in /dev there is need to create a backup of what is currently residing there and then Git track the tarball.

```
[root@server image-prod]# tar cvf dev.tar dev/*
```

Git by itself will not track ownership or permissions so the next step is to add in the setgitperms.perl hook to allow tracking of them.

```
[root@localhost image-prod]# cd .git/hooks/

[root@localhost hooks]# cp /usr/share/git-core
/contrib/hooks/* .
```

The next step, which is bypassed for brevity's sake, is to create post-commit, post-merge and post-checkout as per the instructions in setgitperms.perl. Next to make easy work do a "git add *". This is a very time consuming process as it must inspect each file in the file system and calculate the SHA1 hashes.

```
[root@server image-prod]# git add -f *
```

Git has now added every directory and file in the entire image. This process, in Git speak, is called staging files. There definitely are files that you will not wanting to be Git tracked. Here I walk through an example of finding a file and then unstaging that file.

```
[root@server image-prod]# git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
...
```

```
#       new file:   dev/.udev/db/block:ram11

[root@server image-prod]# git rm -r --cached dev/
```

Now that an initial image is staged and are ready to commit, perform the commit step.

```
[root@server image-prod]# git commit -m "Initial
Commit" --author="Aron Warren
<aronwarren@gmail.com>"

[master (root-commit) 3311f7e] Initial Commit
 Author: Aron Warren <aronwarren@gmail.com>
 Committer: root <root@localhost.localdomain>

 310406 files changed, 41830189 insertions(+), 0
deletions(-)
 create mode 100644 .gitmeta
 create mode 100755 bin/alsaunmute
 create mode 100755 bin/arch
...
...
...
```

6. Booting the Compute Node

In this section it is time to boot the client compute node's image. This is typically done by applying power to the compute node. It is helpful to have a window open with a tail of syslog and another window with a tcpdump inspecting the client-server traffic. If a remote console is available via console redirection then that is helpful to monitor what kernel messages or other messages are being shown. As a last resort a crash cart connected to the node can offer the same information. While the image is booting there is an opportunity to see what is broken and needs to be repaired. Assume that everything during boot went perfectly proceed onto the next step. At this point start fixing problems one-by-one. With each issues resolved use Git to commit the change and use that as documentation for how the machine running ended up successfully running. As an example let's modify the sshd_config.

```
[root@localhost image-prod]# vi /etc/ssh/sshd_config
...
We set PermitRootLogin to "no"
...
[root@localhost image-prod]# git status
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be
committed)
#   (use "git checkout -- <file>..." to discard
changes in working directory)
#
#       modified:   etc/ssh/sshd_config
[root@localhost image-prod]# git diff
etc/ssh/sshd_config
diff --git a/etc/ssh/sshd_config
b/etc/ssh/sshd_config
index 16c7adf..32b7547 100644
--- a/etc/ssh/sshd_config
+++ b/etc/ssh/sshd_config
@@ -39,7 +39,7 @@ SyslogFacility AUTHPRIV
 # Authentication:

 #LoginGraceTime 2m
-#PermitRootLogin yes
+PermitRootLogin no
 #StrictModes yes
 #MaxAuthTries 6
 #MaxSessions 10
```

"git status" told us sshd_config was modified. "git diff" told us what was modified between the current file and the previous commit containing that file. For our example the original commit was the initial commit and in this case was the originally installed sshd_config.

As there are usually many patches offered for either the OS or for the software stack during its lifetime, keeping all commits in a single branch, like the default master branch, can become unmanageable. Branches can be quite helpful to indicate major upgrades or where major functionality was installed. External documentation that references that particular branch ID is useful but it may even be better to reference the particular commit ID associated with the change.

Aron Warren, aronwarren@gmail.com

Now that there is a working compute node, or at least a partially working compute node, following are some real world issues that may crop up.

4. Performing Image Upgrades

Upgrades to computers can either be a simple affair or a complete headache. The scenario outlined in this paper gives a very easy method for performing upgrades. The essential steps to an upgrade are:

1) Reconcile any uncommitted image changes indicated by "git status" (it will save headaches later).

2) Copy the currently running image from image-prod to image-update. One way is *rsync -a image-prod image-update*. Then in image-update create a new branch.

3) Boot a node into the image-update image directory and new branch by modifying the tftp boot image to reflect the new clone's image location.

4) On the compute node once it has booted mount the image read-write.

5) Apply the patches performing Git commits as frequently as desired.

6) Remount the image read-only.

7) Reboot the compute node and verify system functionality.

At this point if everything works as expected then all that is needed is to restart all of the compute nodes using the new image. If something is not working then all that is needed is to repeat steps 4-7 on the compute node.

5. Detecting an Unauthorized Change

Since the image is always mounted read-only in production mode, if someone suspects that a computer has been compromised a reboot is all that is needed to reset the machine to a good state. If there is suspicion that the image has been compromised, then

a Git status will alert the system administrator to any modified files.

6. Items to consider

One of the downsides of using Git to linearly track changes is the problem of the rpm database. Over time rpms will be added or removed from the image in which in itself is normal for the lifecycle of a computer but a problem will arise when needing to reset the image to a particular commit in the history of the image. Essentially all of the rpm modifications that have occurred from that point in history to the present will need to be replayed. Most of the time this will not be a major issue but it can cause an increase in workload if the situation arises.

While this model does assist software upgrades and deployment of machines that can use existing oneSIS images, a scalability issue can occur when a site manages several different oneSIS images. Take for example that a site can end up with 5 or 6 different images: two for clusters, one for desktops, one for servers, one for kiosks and a final one for an upcoming new machine deployment. The amount of work it takes to manage the numerous images with variances amongst image layout can pose a problem for the small shops.

9. Conclusion

The use of oneSIS and Git together provide a mechanism for improved computer security allowing a consistent application of change management practices. The verification of changes and the audit-ability of the current running system is achieved through the version control system and hashing. By having a single image deployment to multiple machine eliminates the need for patching individual systems. The ability to easily revert changes by checking out a previous commit means that broken deployment can easily be fixed.

Aron Warren, aronwarren@gmail.com

References:

Beckman, P., Iskra, K., Yoshii, K., Coghlan, S., Nataraj, A. (2008). Benchmarking the effects of operating system interference on extreme-scale parallel machines. *Cluster Computing,* 11, 3-16. doi:10.1007/s10586-007-0047-2

Chacon, S. (2009). *Pro Git* [Kindle Mac version]. Retrieved from http://www.amazon.com

England, J. (2008, July 10). *oneSIS -Introduction.* Retrieved February 25, 2012, from oneSIS: http://onesis.sourceforge.net/intro.php

England, J. (2011, September). *oneSIS v2.0.3: Administrator's Manual*. Retrieved February 25, 2012, from oneSIS: http://onesis.org/oneSIS-manual-onepage/oneSIS-manual.html

Fish, S. (2011, January 19). *Version Control System Comparison.* Retrieved February 25, 2012, from Better SCM: http://better-scm.shlomifish.org/comparison/comparison.html

Google. (2007, May 14). *Tech Talk: Linux Torvalds on git* . Retrieved February 25, 2012, from YouTube: http://www.youtube.com/watch?v=4XpnKHJAok8

Loeliger, J. (2009). *Version Control With Git* [Kindle Mac version]. Retrieved from http://www.amazon.com

Schroeder, B., Gibson, G. A. (2007). Understanding disk failure rates: What does an MTTF of 1,000,000 hours mean to you?. Trans. Storage, 3(3), 8:1-8:31. DOI=10.1145/1288783.1288785