Automate backups on Linux

Secure Distributed Network Backups

Carlos Justiniano
cjus@chessbrain.net, cjus34@yahoo.com

July 8th 2004

Published in the IBM developerWorks Linux site
http://www-128.ibm.com/developerworks/linux/library/l-backup/index.html

The loss of critical data can prove devastating. Still, millions of professionals ignore backing up their data. While individual reasons vary, one of the most common explanations is that performing routine backups can be a real chore. Because machines excel at mundane and repetitive tasks, the key to reducing the inherent drudgery and the natural human tendency for procrastination is to automate the backup process.

If you use Linux, you already have access to extremely powerful tools for creating custom backup solutions. The solutions in this article can help you perform simple to more advanced and secure network backups using open source tools that are part of nearly every Linux distribution.

Simple backups

This article follows a step-by-step approach that is quite straightforward once you follow the basic steps.

Let's begin with a simple, yet powerful archive mechanism on our way to a more advanced distributed backup solution. Let's examine a handy script called arc, which will allow us to create backup snapshots from a Linux shell prompt.

Listing 1. The arc shell script

```
#!/bin/sh
tar czvf $1.$(date +%Y%m%d%-%H%M%S).tgz $1
exit $?
```

The arc script accepts a single file or directory name as a parameter and creates a compressed archive file with the current date embedded into the resulting archive file's name. For example, if you have a directory called beoserver, you can invoke the arc script, passing it the beoserver directory name to create a compressed archive such as: beoserver.20040321-014844.tgz

The use of the date command to embed a date and timestamp helps to organize your archived files. The date format is Year, Month, Day, Hour, Minutes, and Seconds -- although the use of the seconds field is perhaps a bit much. View the man page for the date command (man date) to learn about other options. Also, in Listing 1, we pass the -v (verbose) option to tar. This causes tar to display all of the files it's archiving. Remove the -v option if you'd like the backup to proceed silently.

Listing 2. Archiving the beoserver directory

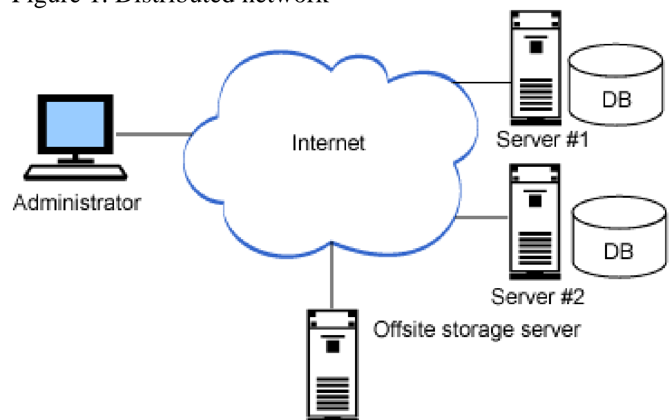
```
$ ls
arc  beoserver
$ ./arc beoserver
beoserver/
beoserver/book1.dat
beoserver/beoserver_ab_off
beoserver/beoserver_ab_on
$ ls
arc  beoserver  beoserver.20040321-014844.tgz
```

Advanced backups

This simple backup example is useful; however, it still includes a manual backup process. The industry's best practices recommend backing up often, onto multiple media, and to separate geographic locations. The central idea is to avoid relying entirely on any single storage media or single location.

We'll tackle this challenge in our next example, where we'll examine a fictitious distributed network, illustrated in Figure 1, which shows a system administrator with access to two remote servers and an offsite data storage server.

Figure 1. Distributed network



The backup files on Server #1 and #2 will be securely transmitted to the offsite storage server, and the entire distributed backup process will occur on a regular basis without human intervention. We'll use a set of standard tools that are part of the Open Secure Shell tool suite (OpenSSH), as well as the tape archiver (tar), and the cron task scheduling service. Our overall plan will be to use cron for scheduling, shell programming and the tar application during the backup process, OpenSSH secure shell (ssh) encryption for remote access, and authentication, and secure shell copy (scp) to automate file transfers. Be sure to review each tool's man page for additional information.

Secure remote access using public/private keys

In the context of digital security, a key is a piece of data which is used to encrypt or decrypt other pieces of data. The public and private key scheme is interesting because data encrypted with a public key can only be decrypted with the associated private key. You may freely distribute a public key so that others can encrypt the messages they send you. One of the reasons that public/private key schemes have revolutionized digital security is because the sender and receiver don't have to share a common password. Among other things, public/private key cryptography has made e-commerce and other secure transactions possible. In this article, we'll create and use public and private keys to create a highly secure distributed backup solution.

Each machine involved in the backup process must be running the OpenSSH secure shell service (sshd) with port 22 accessible through any intermediate firewall. If you access remote servers, then there is a good chance you're already using secure shell.

Our goal will be to provide machines with secure access without requiring the need to manually provide passwords. Some people think that the easiest way to do this is to set up password-less access: do not do this. It is not secure. Instead, the approach we'll use in this article will take perhaps an hour of your time, set up a system which gives all the convenience of "passphraseless" accounts -- but is recognized as being highly secure.

Let's begin by ensuring that OpenSSH is installed and proceed to check its version number. At the time this article was written, the latest OpenSSH release was version 3.8, released on February 24, 2004. You should consider using a recent and stable release, and at the very least use a release which is newer than version 2.x. Visit the OpenSSH Security page for details regarding older version-specific vulnerabilities (see the link in Resources later in this article). At this point in time, OpenSSH is quite stable and has proven to be immune to many of the vulnerabilities which have been reported for other SSH tools.

At a shell prompt, type ssh with the capital V option to check the version number:

```
$ ssh -V
OpenSSH_3.5p1, SSH protocols 1.5/2.0, OpenSSL 0x0090701f
```

If ssh returns a version number greater than 2.x, the machine is in relatively good shape. However, it is recommended that you use the latest stable releases of all software, and this is especially important for security-related software.

Our first step is to log in to the offsite storage server machine using the account, which will have the privilege of being able to access servers 1 and 2 (see Figure 1).

```
$ ssh accountname@somedomain.com
```

Once logged on to the offsite storage machine, use the ssh-keygen program to create a public/private key pair using the -t dsa option. The -t option is required, and is used to specify the type of encryption key we're interested in generating. We'll use the Digital Signature Algorithm (DSA), which will enable us to use the newer SSH2 protocol. See the ssh-keygen man page for more details.

During the execution of ssh-keygen, you'll be prompted for the location where the ssh keys will be stored before you're asked for a passphrase. Simply press enter when asked where to save the key and the ssh-keygen program will create a hidden directory called .ssh (if one doesn't already exist) along with two files, a public and private key file.

An interesting feature of ssh-keygen is that it will allow you to simply press enter when prompted for a passphrase. If you don't supply a passphrase, then ssh-keygen will generate keys which are not encrypted! As you can imagine, this isn't a good idea. When asked for a passphrase, make sure to enter a reasonably long string message which contains alphanumeric characters rather than a simple password string.

Listing 3. Always choose a good passphrase

```
[offsite]:$ ssh-keygen -t dsa
Generating public/private dsa key pair.
Enter file in which to save the key
(/home/accountname/.ssh/id_dsa):
Enter passphrase (empty for no passphrase):
(enter passphrase)
Enter same passphrase again: (enter passphrase)
Your identification has been saved in /
home/accountname/.ssh/id_dsa.
Your public key has been saved in /
home/accountname/.ssh/id_dsa.pub.
The key fingerprint is:
7e:5e:b2:f2:d4:54:58:6a:fa:6b:52:9c:da:a8:53:1b
accountname@offsite
```

Because the .ssh directory which ssh-keygen creates is a hidden "dot" directory, pass the -a option to the ls command to view the newly created directory:

```
[offsite]$ ls -a
. .. .bash_logout .bash_profile .bashrc .emacs .
gtkrc .ssh
```

Enter the hidden .ssh directory and list the contents:

```
[offsite]$ cd .ssh
[offsite]$ ls -lrt
id_dsa id_dsa.pub
```

We now have a private key (id_dsa) and a public key (id_dsa.pub) in the hidden .ssh directory. You can examine the contents of each key file using a text editor such as vi or emacs, or simply by using the less or cat commands. You'll notice that the contents consist of alphanumeric characters encoded in base64.

Next, we need to copy and install the public key on servers 1 and 2. Do not use ftp. Rather, use the secure copy program to transmit the public keys onto each of the remote machines:

Listing 4. Installing the public keys on the remote servers

```
[offsite]$ scp .ssh/id_dsa.pub
accountname@server1.com:offsite.pub
accountname@server1.com's password: (enter
password, not new passphrase!)
id_dsa.pub 100% |*****************************|
614 00:00

[offsite]$ scp .ssh/id_dsa.pub
accountname@server2.com:offsite.pub
accountname@server2.com's password: (enter
password, not new passphrase!)
id_dsa.pub 100% |*****************************|
614 00:00
```

After we install the new public keys, we'll be able to sign on to each machine using the passphrase we specified when creating the private and public keys. For now, log in to each machine and append the contents of the offsite.pub file to a file called authorized_keys, which is stored in each remote machine's .ssh directory. We can use a text editor or simply use the cat command to append the offsite.pub file's contents onto the authorized_keys file:

Listing 5. Add offsite.pub to your list of authorized keys

```
[offsite]$ ssh accountname@server1.com
accountname@server1.com's password: (enter
password, not new passphrase!)
[server1]$ cat offsite.pub >> ./
ssh/authorized_keys
```

The next step involves employing a bit of extra security. First, we change the access rights for the .ssh directory so that only the owner has read, write, and execute privileges. Next, we'll make sure that the authorized_keys file can only be accessed by the owner. And finally, we'll remove the previously uploaded offsite.pub key file, since it's no longer required. It's important to ensure that access permissions are properly set because the OpenSSH server may refuse to use keys which have non-secure access rights.

Listing 6. Changing permissions with chmod

```
[server1]$ chmod 700 .ssh
[server1]$ chmod 600 ./ssh/authorized_keys
[server1]$ rm offsite.pub
[server1]$ exit
```

After completing the same process on server2, we are ready to return to the offsite storage machine to test the new passphrase type access. >From the offsite server you could type the following:

```
[offsite]$ ssh -v accountname@server1.com
```

Use the -v, or verbose flag option, to display debugging information while verifying that your account is now able to access the remote server using the new passphrase rather than the original password. The debug output displays important information which you might not otherwise see, in addition to offering a high level view of how the authentication process works. You won't need to specify the -v flag on subsequent connections; but it is quite useful to do so while testing a connection.

Automating machine access using ssh-agent

The ssh-agent program acts like a gatekeeper, securely providing access to security keys as needed. Once ssh-agent is started, it sits in the background and makes itself available to other OpenSSH applications such as ssh and scp programs. This allows the ssh program to request an already decrypted key, rather than asking you for the private key's secret passphrase each time it's required.

Let's take a closer look at ssh-agent. When ssh-agent runs it outputs shell commands:

Listing 7. ssh-agent in action

```
[offsite]$ ssh-agent
SSH_AUTH_SOCK=/tmp/ssh-XX1O24LS/agent.14179;
export SSH_AUTH_SOCK;
SSH_AGENT_PID=14180; export SSH_AGENT_PID;
echo Agent pid 14180;
```

We can instruct the shell to execute the output commands which ssh-agent displays using the shell's eval command:

```
[offsite]$ eval `ssh-agent`
Agent pid 14198
```

The eval command tells the shell to evaluate (execute) the commands generated by the ssh-agent program. Make sure that you specify the back-quote character (`) and not a single quote! Once executed, the eval `ssh-agent` statement will return the agent's process identifier. Behind the scenes, the SSH_AUTH_SOCK and SSH_AGENT_PID shell variables have been exported and are now available. You can view their values by displaying them to the shell console:

```
[offsite]$ echo $SSH_AUTH_SOCK
/tmp/ssh-XX7bhIwq/agent.14197
```

The $SSH_AUTH_SOCK (short for SSH Authentication Socket) is the location of a local socket which applications can use to speak to ssh-agent. To ensure that the SSH_AUTH_SOCK and SSH_AGENT_PID variables are always registered, enter the eval `ssh-agent` statement into your ~/.bash_profile.

ssh-agent has now become a background process which is visible using the top and ps commands.

Now we're ready to share our passphrase with ssh-agent. To do so, we must use a program called ssh-add, which adds (sends) our passphrase to the running ssh-agent program.

Listing 8. ssh-add for hassle-free login

```
[offsite]$ ssh-add
Enter passphrase for /
home/accountname/.ssh/id_dsa: (enter passphrase)
Identity added: /home/accountname/.ssh/id_dsa
(/home/accountname/.ssh/id_dsa)
```

Now when we access server1, we're not prompted for a passphrase:

```
[offsite]$ ssh accountname@server1.com
[server1]$ exit
```

If you're not convinced, try removing (kill -9) the ssh-agent process and reconnecting to server1. This time, you'll notice that server1 will request the passphrase for the private key stored in the id_dsa file in the .ssh directory:

```
[offsite]$ kill -9 $SSH_AGENT_PID
[offsite]$ ssh accountname@server1.com
Enter passphrase for key
'/home/accountname/.ssh/id_dsa':
```

Simplifying key access using keychain

So far, we've learned about several OpenSSH programs (ssh, scp, ssh-agent and ssh-add), and we've created and installed private and public keys to enable a secure and automated login process. You may have realized that most of our setup work only has to be done once. For example, the process of creating the keys, installing them, and getting ssh-agent to execute via a .bash_profile only has to be done once per machine. That's the really good news.

The less than ideal news is that ssh-add must be invoked each time we sign on to the offsite machine and ssh-agent isn't immediately compatible with the cron scheduling process which we'll need to automate our backups. The reason that cron processes can't communicate with ssh-agent is that cron jobs are executed as child processes by cron and thus do not inherit the $SSH_AUTH_SOCK shell variable.

Fortunately, there is a solution which not only eliminates limitations associated with ssh-agent and ssh-add, but also allows us to use cron to automate all sorts of processes requiring secure passwordless access to other machines. In his 2001 three-part developerWorks series, OpenSSH key management (see Resources for a link), Daniel Robbins presented a shell script called keychain, which is a front-end to ssh-add and ssh-agent and which simplifies the entire passwordless process. Over time, the keychain script has undergone a number of improvements and is now maintained by Aron Griffis, with a recent 2.3.2-1 release posted on June 17, 2004.

The keychain shell script is a bit too large to list in this article because the well-written script includes lots of error checking, ample documentation, and a generous serving of cross-platform code. However, keychain can be quickly downloaded from the project's Web site (see Resources for a link).

Once you download and install keychain, using it is remarkably easy. Simply log in to each machine and add the following two lines to each .bash_profile:

```
keychain id_dsa
. ~/.keychain/$HOSTNAME-sh
```

The first time you log back in to each machine, keychain will prompt you for the passphrase. However, keychain won't ask you to reenter the passphrase on subsequent login attempts unless the machine has been restarted. Best of all, cron tasks are now able to use OpenSSH commands to securely access remote machines without requiring the interactive use of passphrases. Now we have the best of both worlds, added security and ease of use.

Listing 9. Initializing keychain on each machine

```
KeyChain 2.3.2;
http://www.gentoo.org/projects/keychain
Copyright 2002-2004 Gentoo Technologies, Inc.;
Distributed under the
GPL

* Initializing /
home/accountname/.keychain/localhost.localdomain-
sh file...
* Initializing /
home/accountname/.keychain/localhost.localdomain-
csh file...
```

```
* Starting ssh-agent
* Adding 1 key(s)...
Enter passphrase for /
home/accountname/.ssh/id_dsa: (enter passphrase)
```

Scripting a backup process

Our next task is to create the shell scripts, which will perform the necessary backup operations. The goal is to perform a complete database backup of servers 1 and 2. In our example, each server is running the MySQL database server and we'll use the mysqldump command-line utility to export a few database tables to an SQL import file.

Listing 10. The dbbackup.sh shell script for server 1

```
#!/bin/sh

# change into the backup_agent directory where
data files are stored.
cd /home/backup_agent

# use mysqldump utility to export the sites
database tables
mysqldump -u sitedb -pG0oDP@sswrd --add-drop-
table sitedb --tables
tbl_ccode tbl_machine tbl_session tbl_stats >
userdb.sql

# compress and archive
tar czf userdb.tgz userdb.sql
```

On server 2, we'll place a similar script which backs up the unique tables present in the site's database. Each script is flagged as executable using:

```
[server1]:$ chmod +x dbbackup.sh
```

With a dbbackup.sh file on servers 1 and 2, we return to the offsite data server, where we'll create a shell script to invoke each remote dbbackup.sh script prior to initiating a transfer of the compressed (.tgz) data files.

Listing 11. backup_remote_servers.sh shell script for use on the offsite data server

```
#!/bin/sh

# use ssh to remotely execute the dbbackup.sh
script on server 1
/usr/bin/ssh backup_agent@server1.com
"/home/backup_agent/dbbackup.sh"

# use scp to securely copy the newly archived
userdb.tgz file
# from server 1.  Note the use of the date
command to timestamp
# the file on the offsite data server.
/usr/bin/scp
backup_agent@server1.com:/home/backup_agent/userd
b.tgz
/home/backups/userdb-$(date +%Y%m%d-%H%M%S).tgz

# execute dbbackup.sh on server 2
/usr/bin/ssh backup_agent@server2.com
"/home/backup_agent/dbbackup.sh"

# use scp to transfer transdb.tgz to offsite
server.
/usr/bin/scp
backup_agent@server2.com:/home/backup_agent/trans
db.tgz
/home/backups/transdb-$(date +%Y%m%d-%H%M%S).tgz
```

The backup_remote_servers.sh shell script uses the ssh command to execute a script on the remote servers. Because we've set up passwordless access, the ssh command is able to execute commands on servers 1 and 2 remotely from the offsite server. The entire authentication process is now handled automatically, thanks to keychain.

Scheduling

Our next and final task involves scheduling the execution of the backup_remote_servers.sh shell script on the offsite data storage server. We'll add two entries to the cron scheduling server to request execution of the backup script twice per day, at 3:34 am and again at 8:34 pm. On the offsite server invoke the crontab program with the edit (-e) option.

```
[offsite]:$ crontab -e
```

The crontab invokes the default editor, as specified using the VISUAL or EDITOR shell environment variables. Next, type two entries and save and close the file.

Listing 12. Crontab entries on the offsite server

```
34 3 * * * /home/backups/remote_db_backup.sh
34 20 * * * /home/backups/remote_db_backup.sh
```

A crontab line contains two main sections, a time schedule section followed by a command section. The time schedule is divided into fields for specifying when a command should be executed:

Listing 13. Crontab format

```
+---- minute
| +----- hour
| | +------ day of the month
| | | +------ month
| | | | +---- day of the week
| | | | | +-- command to execute
| | | | | |
34 3 * * * /home/backups/remote_db_backup.sh
```

Verifying your backups

You should routinely check your backups to ensure that the process is working correctly. Automating processes can remove unnecessary drudgery, but should never be a way of escaping due diligence. If your data is worth backing up, then it's also worth spot checking from time to time.

Consider adding a cron job to remind yourself to check your backups at least once per month. In addition, it's a good idea to change security keys every once in a while, and you can schedule a cron job to remind you of that as well.

Additional security precautions

For added security, consider installing and configuring an Intrusion Detection System (IDS), such as Snort, on each machine. Presumably, an IDS will notify you when an intrusion is underway or has recently occurred. With an IDS in place, you'll be able to add other levels of security such as digitally signing and encrypting your backups.

Popular open source tools such as GNU Privacy Guard (GnuPG), OpenSSL and ncrypt enable securing archive files via shell scripts, but doing so without the extra level of shielding that an IDS provides isn't recommended (see Resources for more information on Snort).

Conclusion

This article has shown you how to allow your scripts to execute on remote servers and how to perform secure and automated file transfers. I hope you'll feel inspired to start thinking about protecting your own valuable data and building new solutions using open source tools like OpenSSH and Snort.