

Encrypted NFS with OpenSSH and Linux

James Strandboge

NFS is a widely deployed, mature, and understood protocol that allows computers to share files over a network. The main problems with NFS are that it relies on the inherently insecure UDP protocol, transactions are not encrypted, hosts and users cannot be easily authenticated, and its difficulty in firewalling. This article provides a solution to most of these problems for Linux clients and servers. These principles may also be applied to any UNIX server with ssh installed. This article assumes basic knowledge of NFS and firewalling for Linux.



Server Configuration

First, make sure ssh is installed. Although any version of ssh should work, testing for this article was done using OpenSSH 2.9p2-4 from Debian woody. Because ssh is installed by default in most major Linux distributions, and easily obtainable in other UNIX systems, installation will not be covered here. Ssh protocol version 2 will be used in this article, however version 1 can be used instead. To use ssh to encrypt communications, the NFS server must be able to handle TCP requests, since ssh cannot do anything with UDP packets at present. This is easily tested with **rpcinfo -p** on the server.

FreeBSD and Solaris natively support a TCP NFS server. As of this writing, the 2.4 Linux kernel NFS server does not, but progress is being made toward that end. However, the Linux user space NFS server does handle TCP, and this article was tested using nfs-user-server 2.2beta47 from Debian woody. It is also best, though not strictly necessary, if portmap or rpc.bind have support for TCP Wrappers compiled in and have proxy forwarding disabled. Wietse Venema's portmap, the one shipped with all major Linux distributions, should be set up properly.

Once OpenSSH and NFS are installed, the NFS server must be configured. Because of the way ssh handles forwarded connections (tunneling), the **/etc/exports** file must be created or modified to export the NFS shares to itself, but not to the localhost. For example, if the NFS server's IP address is 192.168.2.4, the following should be added to **/etc/exports**:

```
/opt/export/users          192.168.2.4(rw,insecure,root_squash)
```

Under normal circumstances, exporting to oneself is not desired; however, with the firewalling rules described below, potential problems with this process are avoided. The **insecure** option simply means that the client is allowed to connect from a port greater than 1024. For security, no other hosts should be allowed to connect, and the volume should be exported read only with **root-squash** on. Refer to the man page for more options.

When the server is configured, care must be taken in the starting order of sshd and nfsd. Sshd must be started after the NFS server. Otherwise, if the NFS server is restarted, ssh may have bound the NFS ports, and the server won't start. An easy workaround is to modify the NFS init script to stop sshd before nfsd starts, start nfsd, and then start sshd. It may also be necessary to run **killall sshd** from the init script, just to make sure all the sshd servers are down. Worst case, it may take a minute or two for the ports to free up once sshd is stopped. This inconvenience should be alleviated once OpenSSH 3 is in widespread use because it includes the ClearAllForwardings option.

If the above steps are taken, the server can have the nfsd and mountd ports (UDP and TCP) completely firewalled out, since all NFS transactions will happen through the ssh port. The client will need access to TCP port 111 (portmap or sunrpc) to use rpcinfo (see below), but the UDP port can be blocked. Because TCP is not easily spoofed, at least in Linux, specifying which clients can connect to portmap via TCP does not present a problem. Use of TCP Wrappers can protect portmap even further.

Since firewall design is beyond the scope of this article, only the firewall rules to make nfs over ssh work are shown. For iptables, the commands are:

```
iptables -A INPUT -i eth0 -p tcp -s client -dport ssh \
-j ACCEPT
iptables -A OUTPUT -o eth0 -p tcp --sport ssh -d client \
-j ACCEPT
iptables -A INPUT -i eth0 -p tcp -s client -dport 111 \
-j ACCEPT
iptables -A OUTPUT -o eth0 -p tcp --sport 111 -d client \
-j ACCEPT
```

and for ipchains:

```
ipchains -A input -i eth0 -p tcp -s client -dport ssh \
-j ACCEPT
ipchains -A output -i eth0 -p tcp --sport ssh -d client \
-j ACCEPT
ipchains -A input -i eth0 -p tcp -s client -dport 111 \
-j ACCEPT
ipchains -A output -i eth0 -p tcp --sport 111 -d client \
-j ACCEPT
```

client is the client IP or network that is to be allowed in. The rules shown allow **client** to connect to the server on the ssh and sunrpc ports, and allow the server to respond. These rules assume that the firewall has a default policy of **deny**. If the default policy is **accept**, rules would have to be added to explicitly deny access to the nfsd and mountd ports. The interface **eth0** may be different for your hardware.

Note that there is a bug in util-linux prior to version 2.11k that prevents umount from using TCP, so blocking the UDP port causes an error message when unmounting the volume. However, the message is harmless because the volume will be unmounted.

Client Configuration

The client must have ssh as well as the NFS client software installed. It also needs to have the NFS code either compiled into the kernel or compiled as a module. This should not be a problem with any kernel included with a modern distribution.

The basic steps for a client to encrypt and mount an NFS share are:

1. Find which ports mountd and nfsd are running on.
2. Set up the local forwarding rules with ssh.
3. Mount the NFS share from the localhost.

Assuming that the NFS server is named "nfs1", running **rpcinfo -p nfs1** on the client yields something like:

```
program vers  proto  port
100000      2    tcp    111  portmapper
100000      2    udp    111  portmapper
100024      1    udp    946  status
100024      1    tcp    946  status
100003      2    udp    2049 nfs
100003      2    tcp    2049 nfs
100005      1    udp    945  mountd
```

```
100005 2 udp 945 mountd
100005 1 tcp 945 mountd
100005 2 tcp 945 mountd
```

This indicates that the computer named **nfs1** has NFS listening on port 2049, and mountd listening on port 945. The NFS server almost always listens on port 2049, but mountd will listen on different ports, unless the server is configured to listen on a specific port.

Now that the NFS server ports are known, the ssh tunnel can be created. Port forwarding with ssh can be a little confusing for those who are unfamiliar with it. The goal is to establish a tunnel from the local computer to the server. The basic syntax is:

```
ssh -f -L localport:server:serverport -l username server command
```

The **-L** says it is a local forward; **localport** is the port to which our local process can connect; **server** is the server to connect to; **serverport** is what was found using rcpinfo; **-l username** specifies a valid user on the server; **command** is what should be run on the server; and the **-f** says to launch that command in the background. To continue with this example, run:

```
ssh -f -L 2818:nfs1:2049 -l james nfs1 sleep 300
```

This is creating a tunnel from the local computer on port 2818 to port 2049 on nfs1 with username **james**. Ssh runs sleep for 300 seconds (5 minutes), so there is time to connect to the tunnel. Once connected, ssh will not close the tunnel until the client disconnects from the localport, even though it was only initially set up for 300 seconds. Another tunnel has to be set up for mountd:

```
ssh -f -L 3045:nfs1:945 -l james nfs1 sleep 300
```

OpenSSH provides some extra functionality, so the actual command would be something like:

```
ssh -f -c blowfish -L 2818:nfs1:2049 -L 3045:nfs1:945 \
-l james nfs1 /bin/sleep 86400
```

This sets up both tunnels at the same time, sets sleep up for one day, and adds Blowfish encryption, which is much faster than the default. Notice also that the localports used are arbitrary, and can be any free ports on the client system.

Now that the ssh tunnels are set up, mount the NFS volume with:

```
mount -t nfs -o tcp,port=2818,mountport=3045 \
localhost:/opt/export/users /mnt/nfs/sshmount
```

This specifies to mount with TCP and the command appears to mount a filesystem on the localhost via NFS. However, port 2818 is an ssh tunnel to port 2049 on nfs1, and port 3045 is an ssh tunnel to port 945 on **nfs1**. The directory to mount is the same one specified in **/etc/exports** on the server. Test out the encrypted NFS session with the **ls** command and a packet sniffer (Ethereal is nice). Everything is encrypted!

To make mounting the above volume easier, add this to the client's **/etc/fstab**:

```
localhost:/opt/export/users /mnt/nfs/sshmount nfs \
tcp,rsize=8192,wsize=8192,intr,rw,bg,nosuid, \
port=2818,mountport=3045,noauto
```

Now **mount /mnt/nfs/sshmount** can be run instead, as long as the tunnel is already established. This entry is probably a good configuration for mounting user directories. The options can vary, but be sure to keep the ones already mentioned (**tcp**, **port**, and **mountport**), as well as **intr** and **bg**. This makes mount do its work in the background, and allows it to be interrupted in case something goes wrong.

As mentioned before, only Linux clients are supported, and the only reason is because the **umount** command in other unices, like *BSD and Solaris, does not support the **mountport** option. If it did, then any UNIX client would work. Modifying the source code for umount should not be too difficult, otherwise talk to your vendor.

Firewalling the client is similar to the server, except the client can be setup to only allow already established connections from the server on the ssh and sunrpc ports. The commands for iptables are:

```
iptables -A INPUT -i eth0 -p tcp ! -syn -s nfs1 --sport ssh \
-j ACCEPT
iptables -A OUTPUT -o eth0 -p tcp -d nfs1 --dport ssh \
-j ACCEPT
iptables -A INPUT -i eth0 -p tcp ! -syn -s nfs1 --sport 111 \
-j ACCEPT
iptables -A OUTPUT -o eth0 -p tcp -d nfs1 --dport 111 \
-j ACCEPT
```

and for ipchains:

```
ipchains -A input -i eth0 -p tcp ! -y -s nfs1 --sport ssh \
-j ACCEPT
ipchains -A output -i eth0 -p tcp -d nfs1 --dport ssh \
-j ACCEPT
ipchains -A input -i eth0 -p tcp ! -y -s nfs1 --sport 111 \
-j ACCEPT
ipchains -A output -i eth0 -p tcp -d nfs1 --dport 111 \
-j ACCEPT
```

Making It Easier

To simplify this for the client, I wrote a Perl script that will maintain the ssh tunnel on the client indefinitely (see [Listing 1](#)). It can be started from **rc.local** on bootup. To configure, just set up the configuration variables at the beginning of the script as directed by the script.

The script simply calls rcpinfo, and determines which ports the server is listening on, based on the version specified. Then it calls ssh with the appropriate options. Putting passwords into scripts should be avoided for security reasons, though options could be added to the script to do so. As written, the script will require the user to enter a password, but this can be remedied by using a password-less DSA (or RSA) key with ssh. You can also specify that the key will only allow the user to execute certain commands to limit the effects if the key is compromised. Since ssh only needs to call the sleep command if the key is compromised, the effect is negligible. To get this to work, first create a DSA key on the client:

```
ssh-keygen -t dsa
```

Assuming the username on the client is "jdstrand", when prompted, store the key as **/home/jdstrand/.ssh/id_dsa_nfs**. When prompted for a password, you can just hit "Enter".

This creates two keys — **id_dsa_nfs** and **id_dsa_nfs.pub**. Then the contents of the public key should be added to the server's **authorized_keys2** file. If the username to log in to the server is "james", copy **id_dsa_nfs.pub** to **/home/james/.ssh/id_dsa_nfs.pub** on the server and run on the server:

```
cd /home/james/.ssh
cat ./id_dsa_nfs.pub >> ./authorized_keys2
```

Now for the client to log in to the server without a password, run:

```
ssh -2 -i /home/jdstrand/.ssh/id_dsa_nfs -l james nfs1
```

This calls ssh with protocol version 2, and specifies which id key to use for authentication. This should be added to the **\$ssh_opts** variable in **nfs_ssh_fw.pl** if the

script is being used. Before using the script, you should log in at least once using this key id to make sure that everything works.

To tighten this up, add this to the beginning of the entry for the key just added to `/home/james/.ssh/authorized_keys2`:

```
from="client",command="/bin/sleep 86400" ssh-dss ... rest of key ...
```

client needs to either be an ip address or the hosts fully qualified domain name. Now whenever the client is authenticated with this key, only that command can be executed. If another command is specified, it will be ignored in favor of the one specified in **authorized_keys2**.

Once everything is verified to work properly, add this to `/etc/init.d/rc.local` (or similar startup script on the client) to make the encrypted filesystem mount on boot:

```
echo "Mounting the encrypted filesystem"  
/usr/local/sbin/nfs_ssh_fw.pl &  
sleep 10  
mount /mnt/nfs/sshmount
```

The **sleep** command is to give ssh a chance to set up the tunnel. If the recommended **bg** option is specified in `/etc/fstab`, the mount would eventually happen without the **sleep** command, but errors would be displayed on the console. Keep in mind, this is a not a robust method because it does not check whether the NFS server is up. A more robust method should be developed in a production environment.

Caveats and Warnings

Users are still not authenticated. However, hosts are when using the ssh keys for validation.

Encrypting every NFS client connection comes at a cost — mainly CPU overhead. However, depending on the cipher used for encryption, this issue may not be as big as one might think. For example, I had a 300% increase in throughput using Blowfish over the default 3des.

As mentioned above, only Linux clients are supported.

There is a small window where the mount will fail when the ssh forwarding is being established. The easy fix is to simply to mount again.

Autofs (on Linux) does not work.

Summary

File sharing is an indispensable service in almost any network. NFS is a mature and easy to implement protocol for sharing files; unfortunately, it has a bad reputation for being insecure. Utilizing ssh can significantly increase the security of NFS by using TCP to prevent spoof attacks, by encrypting all data moving over the network, and by using strong host authentication. Other benefits are that a new protocol does not have to be learned, as is the case for AFS and IPsec. NFS over ssh also works with NAT, unlike IPsec.

The methods in this script have been used on Linux and FreeBSD 4.3 servers, and various Linux clients. The performance of NFS with TCP and encryption is, of course, lower than straight NFS, but in practice this has proved to be a good tradeoff with user's home directories. Although my network is small, I have been using NFS over ssh for months with no problems or complaints from my users.

James Strandboge first started using GNU/Linux and UNIX in 1997 while attending graduate school at the University of North Carolina in Charlotte. He gets people to use Free and Open Source software whenever they give him half a chance. He is very interested in networking and security for GNU/Linux. While away from his computers, he enjoys spending time with his son Dane and wife Tish. He can be reached at jstrand1@rochester.rr.com.

Copyright © 2001 Sys Admin, [Sys Admin's Privacy Policy](#). Comments about the Web site: webmaster@sysadminmag.com