The following paper was originally published in the
Proceedings of the Twelfth Systems Administration Conference (LISA '98)
Boston, Massachusetts, December 6-11, 1998

SSU
Extending SSH for Secure Root Administration

Christopher Thorpe, Yahoo!, Inc.

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

SSU: Extending SSH for Secure Root Administration

Christopher Thorpe – Yahoo!, Inc.

ABSTRACT

SSU†, ‘‘Secure su,’’ is a mechanism that uses SSH [Ylonen] to provide the security for distributing access to privileged operations. Its features include both shell or per-command access, a password for each user that is distinct from the login password and easily changed, and high portability. By installing SSU, administrators build a solid infrastructure for using SSH for improving security in other areas, such as file distribution and revision control.

Introduction and Site Information

The EECS research computing environment at Harvard University is comprised of approximately two hundred workstations running eight variations of Unix. Users are primarily faculty, graduate students and researchers, most of whom need some level of root access to their workstations. Some students need only the ability to reboot their workstations or mount removable storage media, but many computer science researchers need full root access for their work. In addition, these Unix-savvy users ease the load on system administrators tremendously by performing administration tasks when possible.

Because the EECS environment is a research-oriented group, machines come and go on the network all the time, as do users who need privileged access. There are several independent research groups within EECS, and researchers in a group generally need privileged access only to that group’s machines. In addition, many researchers administer their own research machines while using our network and home directory NFS server. Since our environment is comprised of machines for which we provide various levels of administration, we built a system to give both administrators and researchers root access on various groups of machines. In addition, we made the system easy to install so that anyone setting up a new machine need only modify or create two files in addition to the ssh distribution (which everyone installs anyway.)

SSU sits on top of the widely-used secure shell protocol SSH. Because of its usefulness in providing security and ease of access to networked computing environments, ssh is now in use on all of the Unix-based machines within EECS. SSH can also be used for remote administration, where a trusted host uses the protocol to establish secure connections to other machines and execute privileged operations. SSH has been successfully used to support the secure exchange of data for programs such as rdist [Cooper] and CVS [Cyclic].

In any large installation, key management is an important task for the proper installation and maintenance of SSH. Since we already used SSH, it was natural to extend the key management system we developed for remote root operations into a more complex system supporting SSU. By doing so, researchers adding a new system need only install SSH and add the trusted public identity key in root’s authorized_keys file. After this is done, SSU is installed automatically from the trusted host and keys are periodically updated. Note that SSH uses two types of RSA keys: one for host identification and another for user authorization. SSU does not use the host identification keys outside their normal use when ssh establishes a connection between two hosts. All SSU authentication uses 1024-bit RSA key pairs, completely separate from a host’s identity keys.

The features we required:

- The ability to easily redefine users’ root access
- Definition of access in terms of groups of machines and automatic update of machines’ root access configurations when these groups change
- The ability to administer machines without possessing a local user id
- Easy installation for researchers installing new systems
- Seamless portability between all of our operating systems

Features we wanted:

- A password for privileged operations distinct from the user’s login password and consistent for all operations
- Authentication of the user within the EECS domain before allowing root access

Description of SSU

Table 1 shows the commands and files used by SSU. Examples 1, 2, and 3 show how to reboot the local machine, gain a remote shell, and grant access to

†The bulk of this work was completed while the author was employed as a student systems administrator while an undergraduate at Harvard University. SSU is currently used in Harvard’s EECS (Electrical Engineering and Computer Science) research environment.

a new user. Note that this program asks for multiple command/host pairs, so that it is possible to define a different set of commands for each group of hosts.

Overview of How SSU Works

When users are granted access they are given access to commands chosen from a list of SSU commands (checked in a configuration file for sanity).

A configuration file is created or updated by running ssu-user. This file may be modified by hand to change users' permissions in the future. The reason for using this file is that if a user is granted access to groups of servers (as defined in the netgroup file or the GROUPS directory) and those groups change the user's permissions are automatically adjusted to correspond to the changes in those groups the next time process-ssu is run.

An RSA key pair is generated for each command, and the passphrase is set to be the same for all commands for that particular user (in order to facilitate ease of remembering the passphrases.) The ssu-passwd command makes it easy for users to change all of their SSU passphrases at once.

A user executes a privileged command by typing "ssu command" or "ssu command@host." The ssu script then determines the appropriate identity to use, connects to the SSU port on the remote machine or loopback interface as root, and ssh executes the command as root. See Appendix D for a description of how SSH authenticates a user. Note that SSU does not allow .rhosts, .shosts or password authentication, and disables TCP/IP, X, and agent forwarding by default.

Key Management

Key management was probably the most difficult problem in implementing this solution. Since the system's security is based on RSA public/private key cryptography, it is vital to correctly administer the keys. First, we introduce the idea of a "trusted master." This machine holds an RSA private key for the root user that is trusted by the root users on all other machines on our network. (It is possible, and probably wise, to have multiple trusted masters.)

By configuring all the machines in the network to trust the root identity on this host, it allows all authentication for user root access to take place on the trusted machine. Connections with root access can be established from there. This also has the useful side effect of creating a machine that can establish secure, privileged connections to all other machines for any purpose – e.g., rdist, network monitoring and backups.

Command/File	Description
ssu	Used to invoke a privileged operation locally or remotely.
ssu-passwd	Used to modify a user's RSA passphrases for all SSU commands.
ssu-user	Administrators' tool for creating or modifying SSU privileges.
process-ssu	Processes the configuration files, generates the authorized_keys files, and pushes the files to the hosts.
SSU.pm	A Perl 5 module that contains local configuration settings and library functions used by ssu commands.
ssu_cmdgroups	Definitions of convenient groups of SSU commands.
ssu_usergroups	Definitions of convenient groups of SSU users. These groups may work with or be instead of the system group file.
ssu_hostgroups	Definitions of convenient groups of hosts for SSU. These groups may work with or be instead of the system netgroup file.

Table 1: User Interface.

```
% ssu reboot
Enter passphrase for RSA key 'cat:tcsh@eecs': [passphrase]
Shutdown at Sun Sep 20 14:59:19 1998.
shutdown: [pid 19268]
Connection to localhost closed.
```

Example 1: Rebooting the local machine.

```
% ssu tcsh@herbert
Enter passphrase for RSA key 'cat:reboot@eecs': [passphrase]
herbert#
```

Example 2: Gaining a shell on a remote machine.

Alternatively, private keys may be pushed to all of the machines on the network or placed in an NFS-mounted directory (see Appendix C for security concerns). With the keys available everywhere, users can gain root access through the loopback port. This allows restricted login access to the trusted master as well as the ability to gain root access even if the trusted master is inaccessible.

The master's trusted private key should be carefully guarded. We do not protect it with a passphrase so automatic privileged operations can execute without administrator intervention at boot time. In Appendix B we describe a method of passphrase protection that requires minimal administrator intervention for automatic operations. Using this method, however, it is required to enter the passphrase for each manually executed remote operation, e.g., updating the remote keys.

It is also possible to use SSU without allowing remote root access; to do this the keys must be distributed as described and connections as root allowed only through the loopback port. Pushing keys to remote hosts is difficult without remote root access. One solution is to make the clients periodically query the host where keys are stored to obtain the most recent authorized_keys file and the appropriate private keys.

If remote root access is desired, the trusted master is used to distribute lists of authorized keys to each of the machines for which privileged access is desired. These lists are constructed for each machine separately from the configuration files described below. Each entry in the list contains a command, any special options with which ssh will execute the command (e.g., environment="SHELL=/bin/false" to prevent shell escapes from vi or less), and the public key for the user-command pair generated by ssu-user or process-ssu.

Since the private keys of these user-command pairs are protected by passphrases, even if they are captured via NFS sniffing or user negligence they are still secure. Some administrators may wish to place them on the trusted server's local disk if root access is to be limited to connections coming from the trusted server. Since we allow trusted connections from the trusted server and from the loopback port, we place the keys in the users' NFS-mounted home directories for convenience. (See Appendix C for further discussion of the security issues here.) We also store the keys on the trusted server so that if our NFS server fails it doesn't disable all privileged access.

Each user requires a separate key for each command. Fortunately, most users need only execute a few commands. For example, using "adminmenu," each user can be given a specific list of privileges from a selection of common administrative operations. The list of allowed operations is set in the configuration for each user. This cuts down on the administration of individual commands.

Because these keys are normal SSH public/private key pairs, the ssh-agent can be used to store the passphrases for these keys. We discourage this use of ssh-agent, as it creates two minor security holes. First, if a user adds the key to the agent, anyone can sit down at the computer later and execute the privileged commands without a passphrase. Second, if the login

```
% ssu-user newbie
SSU New User Configuration for newbie (Nathan Ewbie)

Enter the SSU commands newbie should have access to, space separated:
[Blank exits.]
bash reboot

Enter the hosts on which these commands should be accessible:
[Blank returns to previous prompt.]
bach handel mozart

Enter the SSU commands newbie should have access to, space separated:
[Blank exits.]
vi-aliases

Enter the hosts on which these commands should be accessible:
[Blank returns to previous prompt.]
mailhost
Enter the SSU commands newbie should have access to, space separated:
[Blank exits.]
[return]

Please remember to run process-ssu after you are finished adding new users!
```

Example 3: Granting access to a new user.

is shared, compromised, or a login is left open on another machine, a malicious user can set environment variables to use the running ssh-agent to gain access without a passphrase.

Configuration

Within the /usr/local/etc/SSU directory, there are five important configuration files: SSU.pm, ssu_hostgroups, ssu_usergroups, ssu_cmdgroups, and ssu_config. In addition, a COMMANDS directory contains a file containing definitions for each command SSU will be used to execute as root and a HOSTS directory contains the authorized_keys file for each host in the installation. The HOST_DEFAULTS directory contains files that are prepended to the host definitions created by process-ssu so that special defaults can be created on a host-by-host basis. SSU.pm is a Perl 5 module that contains important local configuration details such as the location of files and how to obtain the domain name correctly.

The ssu_hostgroups file contains a list of keywords that map to groups of hosts or other groups. For example, the lines:

```
lab-linux: bach handel mozart brahms
linux: lab-linux alfie betty
```

define groups lab-linux (bach, handel, mozart and brahms) and linux (bach, handel, mozart, brahms, alfie and betty). A group must be defined before use as a subgroup. Obviously groups cannot be subgroups of each other, so this "before" rule does not limit groups' definitions. Such group definitions allow the administrator to grant a user permission to execute commands on one or more groups of hosts, e.g., mailhosts. When mailhosts changes, users with privileges for mailhosts automatically have their keys distributed to the new mailhosts map on the next key distribution update. Included with the SSU distribution is a utility that will generate ssu_hostgroups lines from an NIS [Sun] netgroup format.

The ssu_cmdgroups file contains keywords that map to lists of commands that a user in that command group may execute. The file behaves exactly like ssu_hostgroups, and command groups may contain other groups already defined. For example, if helpdesk staff needed a certain group of commands, an administrator might add the line:

```
helpdesk: passwd, lprm, reboot
```

The ssu_usergroups file contains keywords that map to lists of users. Again, groups may consist of usernames or other groups. Note that SSU reads the /etc/group file before processing this file, so groups defined there need not be redefined, and groups in ssu_usergroups may use groups defined in /etc/group. If a group is redefined in ssu_usergroups after being initially defined in /etc/group, a warning is printed and the old definition is lost.

The COMMANDS directory contains several short files that function as SSU command aliases. These aliases are used in the ssu-user script so that the full pathname and environment need not be specified for commands. This file contains part of line that will go into the ~root/.ssh/authorized_keys file. Example 4 shows how one might configure files for a root tcsh shell and a command to modify the mail aliases. In Example 5 we set the SHELL environment variable to /bin/false to prevent the user from performing shell operations through vi. We set the path explicitly as well for security, and because kill is not in the same location on all of our hosts.

```
command="/usr/local/bin/tcsh"
```

Example 4: Configuring a root tcsh; file *tcsh*.

```
no-port-forwarding,no-X11-forwarding,
o-agent-forwarding,
environment="SHELL=/bin/false",
PATH="/bin:/usr/local/sbin:/usr/bin"
command"vi /etc/sendmail.cf;
    kill -HUP `cat /var/run/sendmail.pid`"
```

Example 5: Preventing shell operations from vi, file *vi-aliases*.

The HOSTS directory contains a file for each host on which SSU is to be run. It is initialized from the HOST_DEFAULTS directory each time process-ssu runs. process-ssu then appends to or creates files in this directory to complete its list of authorized_keys files. To do so, it examines the list of commands for each user in the configuration file, and creates a line in the appropriate host file for each user-command pair. These lines are constructed by concatenating the command definitions with the user's public key for that command and eliminating all internal newlines.

Once the user/command pairs have been processed, process-ssu then goes through each host in the HOSTS directory and sends the new authorized_keys file to the host via scp (part of ssh). If a passphrase is needed, the administrator should run ssh-agent before process-ssu.

Host Configuration

For security reasons, we want to limit privileged connections to the loopback port and trusted hosts. Since ssh does not allow limiting connections on a per-user basis, we run two ssh daemons – one on the standard port 22 for general access, and another on another privileged port (we use 122). The only recommendation we make with regard to the standard ssh configuration is that it contain the line `PermitRootLogin` no so that root access is only available through the special ssh daemon.

We then configure another ssh daemon to run for supplying privileged access, using a file including the following lines:

```
Port 122
```

```
RandomSeed /etc/ssh_root_random_seed
PidFile /var/run/sshd_root.pid
PermitRootLogin nopwd
RhostsAuthentication no
RhostsRSAAuthentication no
RSAAuthentication yes
PasswordAuthentication no
AllowHosts 127.0.0.1
     140.247.60.30 140.247.60.20
```

(See the source for a complete listing of the file we use.)

The Port, RandomSeed and PidFile lines prevent conflict with the standard ssh daemon. We turn password authentication completely off so that users will never be prompted for root's password – even if they know it. We allow only RSA authentication using the trusted keys and those generated from SSU-user. Finally, we restrict the hosts that may connect as root to the two trusted servers and the loopback port. This makes certain that even if intruders break into a non-trusted machine and gain a passphrase and private key, they cannot gain root access elsewhere on the network.

Related Solutions

Others have implemented similar solutions to this problem. These include priv [Hill], sudo [Courtesan], and op [Christiansen] which are very similar programs. These allow users to execute certain commands with root privileges. Each of these uses a configuration file describing privileged commands, users allowed to execute those commands, and the hosts on which and arguments with which the commands may be executed. Users invoke privileged commands by using a prefix (''priv,'' ''sudo'' or ''op''), then the command which is to be executed. While these systems are useful, we found them to be inadequate for our needs.

By extending our installation of SSH to SSU, we did not introduce any new binaries. All of the features SSU provides are written in Perl. SSH and Perl are already ported and thoroughly tested on all of the operating systems we use, and are utilities that we already use and maintain for other purposes. By using only existing binaries, the system is completely portable, and we avoid potential security holes introduced by writing and porting setuid C code. Given the highly diverse nature of our environment, portability and ease of installation is a very high priority.

None of the above solutions allows for a password distinct from login passwords in the password database. Op allows for specifying the password of another user; priv allows for that as well as ''safe deposit box'' authentication where two distinct users must type their password. These solutions are better, but login passwords are notorious for being sniffed and cracked, even in shadow password systems. Priv also provides mechanisms for password challenges and single-use passwords, but we concluded that these solutions were too cumbersome for our users. SSU has four levels of security: the standard login password, the hosts' RSA key pairs, and the user's RSA keys and the passphrase protecting the private key.

SSU also is unique in that it allows privileged operations on a host to which the user does not have login access. If a researcher wishes to restrict user access or not to install NIS on a system, staff members might not have logins on systems (particularly those that are new or unstable). While one might argue that machines should always have staff logins, it is unrealistic to expect all researchers to repeatedly update their systems with the most up-to-date staff information. An SSU public key in root's authorized_keys file gives users gives users the ability to perform operations on remote systems through authentication on a trusted host.

There are some useful features in these systems that ssu lacks. For example, priv allows the administrator to specify time of day and terminals at which users may execute privileged operations. SSH has no mechanism for supporting this, and hence, neither does SSU. With sudo, users need not retype their passwords each time sudo is executed – it ''remembers'' authentication for a few minutes. (As described above, we discourage SSU users from using ssh-add to add an SSU key to an ssh-agent for this purpose.)

Another advantage of sudo, priv and op is that they act more like ''su'' than SSU. Commands such as sendmail and shutdown ''remember'' the user id of the original user and report it, where SSU creates a login as the root user. While it might be possible to modify SSH or the login shell to solve this problem, that defeats the portability of our system. To do what we can, we do set the USER and LOGNAME variables of the new shell's environment to the root-invoking user when creating the authorized_keys files.

ksu

Before installing SSU, we used Kerberos [Neuman] ksu to give root shells to trusted users. While ksu requires a Kerberos root instance password distinct from the Kerberos password and the login password, it does not allow execution of limited privileged commands. The most significant reason we discarded ksu was that a full installation required compiling and maintaining Kerberos binaries on multiple platforms. Using Kerberos would have inconvenienced researchers, as they would have to install and configure it on each machine they added to the network. We found that SSH provided the security features we used Kerberos for; the administration cost of Kerberos for ksu alone outweighs its benefit.

s/key

s/key [Haller] and other one-time password schemes for granting root access do not provide much

more administrative flexibility than giving users the root password. Revoking access from a user who has a list of valid s/key passphrases involves distributing new passphrases to the other users who need them. These systems may provide extra security, but they do not solve the problem of managing access to privileged operations.

login

Sharing the root password with everyone is probably the simplest solution, but is also the least secure. When the root password changes, everyone who needs to know it must be notified, and whenever anyone no longer should have root access, it must be changed. This solution is generally only viable in small operations with a small number of capable administrators.

Comparisons

Table 2 compares the various solutions.

Feature	root	s/key	priv	sudo	op	ksu	SSU
Same root-invoking password everywhere	•	-	1	•	1	•	•
Allows only specific commands	-	-	•	•	•	-	•
Each user has a unique password	-	•	•	•	•	•	•
Root passwords are independent from login passwords	-	•	2	-	-	•	•
Can gain root access locally if trusted master is unreachable	•	•	•	•	•	-	3
Requires authentication as a known user before gaining root access	4	-	•	•	•	•	•
The root password itself is never requested	-	-	•	•	•	•	•
Does not require a local user login	•	•	-	-	-	-	•
Simple to configure command limitations	•	•	•	-	-		
Can specify arguments to operations on the root-invoking command line	•	•	•	-	-		
Secure if user's login password is compromised	•	•	-	-	-	•	•
Secure if someone captures the local network traffic when run	5	•	•	•	•	•	•
Secure if root access is obtained through an unencrypted link	-	•	-	-	-	-	-

Key	
•	yes
-	no
space	not applicable
1	priv and op allow specified passwords for each command
2	priv allows for challenges and one-time passwords
3	SSU requires keys to be distributed or NFS-mounted for this to work. See Appendix C for a discussion of the security ramifications of this.
4	some operating systems restrict terminals where root can login and the users who may su
5	If a user telnets as root this is a risk; otherwise there is no problem.

Table 2: Comparison of various solutions.

Drawbacks

The most serious drawback of SSU is its reliance on the RSA keys. If the keys are kept on a master server or NFS server and that machine goes down, root access is disabled. The only way to completely remove this problem is by distributing the private keys to every machine's local disk. This (as does sharing them via NFS) increases the risk that an intruder could gain a private key. The solution with two master servers, each with local copies of the passphrase-encrypted keys, is probably solid enough for most installations.

Any SSU operation runs under a root login, whether executing a shell or a single command. Accountability is not provided on the local host, though the system logs from the ssh daemon provide ample information for this purpose. Broadcast messages and mail sent appear as from ''root'' rather than the invoking user.

SSU does not allow arguments to commands on the root-invoking command line. This forces the use of scripts or shell access for simple operations like renicing or killing processes, changing the ownership of files or setting the time for reboot of a machine. In addition, since the commands are executed in a remote manner, interactivity is limited. With priv, sudo and op a shutdown command is easily undone; SSU requires another connection to stop the shutdown process. Limiting command parameters is complicated in SSU, as it must be done using the backend script. However, there is less room for error in a commonly used language like perl than in a unique, single-purpose language.

Conclusion

SSU is a very useful tool for distributing root access among a large group of skilled researchers and faculty. Its most important features are that it allows distinct root passwords and configurable commands for each individual, uses a secure infrastructure for authorization throughout an installation, and is extremely portable. The security is solid and well-tested. It logs all commands or shells started for accountability. We have several examples of commands that can be used for common system administration tasks, and hope that others who use this system will share their work.

Availability

The SSU distribution is available via anonymous ftp at ftp://eecs.harvard.edu/pub/cat/ssu and contains the most recent version of this document, all of the programs referenced herein, and additional utilities and documentation as ssu is improved.

Acknowledgements

Special thanks to Michael Barrientos, who supplied Appendix D and provided moral support and excellent feedback during the preparation of this work. Also special thanks to Peg Schafer, who encouraged me (''strongly'' would be an understatement) to complete this work and submit it for publication at LISA. Phil Cox, for his help preparing the work for LISA.

Author Information

Christopher Thorpe graduated in June 1998 with an A. B. in Computer Science and Music from Harvard University. While at Harvard, Chris worked as a system administrator for Harvard's EECS (Electrical Engineering and Computer Science) research group. In addition to working as a system administrator, Chris was a head teaching fellow for introductory computer science courses at the university. Christopher is now employed at Yahoo! in Santa Clara, California as a member of the Yahoo! Store team. He lives in Sunnyvale, California and spends his negligible free time playing the piano and french horn, participating in community theatre, scuba diving, and playing computer games.

References

[Christiansen] Christiansen, T. ''Op: A Flexible Tool for Restricted Superuse Access,'' *Proceedings of the Workshop on Large Installation System Administration (LISA 88)*. Monterey, CA, USA, 1988. pp. 89-94.

[Cooper] Cooper, M. ''Overhauling Rdist for the 90's,'' *Proceedings of the Sixth Conference on Systems Administration (LISA 92)*. Long Beach, CA, USA, 1992. pp. 175-188.

[Cyclic] *CVS*, Cyclic Software http://www.cyclic.com .

[Courtesan] *Sudo*, Courtesan Consulting http://www.courtesan.com/courtesan/products/sudo .

[Haller] Haller, N. M. ''The S/KEY One-time Password System,'' *Proceedings of The Internet Society Symposium on Network and Distributed System Security*, 1994, pp. vi+173, 151-57.

[Hill] Hill, B. University of California, Davis ''Priv: Secure and Flexible Privileged Access Dissemination,'' *Proceedings of the Tenth USENIX System Administration Conference (LISA 96)*. Chicago, IL, USA, Sept. 29-Oct. 4, 1996. pp. 1-8.

[Neuman] Neuman, B. C. and Ts'o, T. ''Kerberos: An Authentication Service for Computer Networks,'' *IEEE Communications Magazine*, vol. 32, no. 9, pp. 33-38.

[Sun] *NIS+*, Sun Microsystems Incorporated.

[Ylonen] Ylonen, T. ''SSH – Secure Login Connections over the Internet,'' *Proceedings of the Sixth USENIX UNIX Security Symposium*, San Jose CA, USA, July 22-25, 1996. pp. 214, 37-42.

Appendix A: United States Export Restrictions

The United States government restricts export of certain strong encryption algorithms. Since SSH is freely available from sites outside the US, we assume that anyone who wishes to install SSU can obtain and install a working version of SSH. The SSU distribution contains no encryption code; this is taken care of entirely by SSH. Those users who need information on installing SSH should consult the SSH FAQ (http://www.uni-karlsruhe.de/~ig25/ssh-faq/) and the SSH Distribution (ftp://ftp.cs.hut.fi/pub/ssh/).

Appendix B: Security Issues with Automatic Administration from a Trusted Host

The ssh-agent utility allows ssh users to enter the passphrases for a private key and saves the decrypted private key in memory. Further ssh connections use sockets administered by the ssh-agent. When a system is brought up, an ssh-agent running as root should be started with something like (assuming the trusted key is /root/.ssh/trusted)

```
csh% ssh-agent > /var/run/trusted-agent
csh% source /var/run/trusted-agent
Agent pid 1234;
csh% ssh-add -p
Need passphrase for /root/.ssh/trusted
    (root@foo.com).
Enter passphrase: [passphrase]
Identity added: /root/.ssh/trusted
    (root@foo.com).
```

(The -p option prevents ssh-add from starting an X11 window to read the passphrase if X is running.)

In scripts, one need only add the line

```
source /var/run/trusted-agent
```

before using ssh to make privileged connections to remote hosts. This will connect the process with the agent that already has the decrypted trusted key in memory. If any user other than root attempts to use this agent, they will fail.

This only increases security completely against someone obtaining the private key. The identity file is useless without the passphrase. If someone obtains a root shell on the trusted host while this is running, it is more difficult but still possible to gain remote access. In this scenario, they need only to examine /var/run/trusted-agent or use the agent-socket attack described below. Even if there is no trusted-agent file and a single ssh-agent is started for a master script running all remote operations, intruders could check the process table and the /tmp/ssh-root directories to find an ssh-agent process and agent-socket file with the same timestamp. They could then set the appropriate environment variables and "piggyback" on that agent to other hosts. Obviously this takes some knowledge of ssh and the local configuration, but is not particularly challenging. (Perhaps someone should add an option to ssh-agent to only allow connections from processes that are children of the agent itself.)

The other option, using the ~root/.shosts file, has its own weaknesses. (For some reason, SSH 1.2 does not like /etc/{s,}hosts.equiv for root.) The .shosts file essentially uses the hosts' private keys, which are never encrypted with a passphrase, as the sole source of authentication. With .shosts, if intruders obtain the trusted host's private key then they can spoof a connection. This may or may not be more difficult than gaining a shell or the root user's private key on the trusted host. Clearly, if they gain root access to the trusted host then they can ssh over through .shosts anyway. Since security-conscious configurations will limit trusted connections to trusted hosts, .shosts is no more secure than an RSA trusted key pair without passphrase protection.

With this in mind, if the trusted host is used to perform automated administration, configurations should ideally allow only trusted logins, and accept only ssh connections from other machines within the network. Such a trusted host should not run any other daemons that might compromise the security of the system, e.g., mail or telnet. In general, using any automated administration that trusts root on another host is a security risk, because when intruders obtain access to the trusted host, they can gain access remotely by modifying the automation scripts themselves.

Appendix C: Security Issues with Distributing Keys via NFS

While NFS is riddled with security holes, distributing the private RSA keys used for user authentication is not the security risk that it might seem to be, provided that these keys are encrypted with passphrases. (All of the scripts supplied with SSU disallow empty passphrases.) For this reason, even if an intruder were to obtain both pieces of a private/public key pair, it would still be necessary to obtain the passphrase. An intruder with both the public and private keys of an RSA key pair might be able to crack the passphrase if it were poorly chosen. Even if an intruder were able to crack the passphrase, however, it would then be necessary to gain a shell on a machine in order to use the passphrase to gain local root access. Malicious users with login access can often gain local root access and certainly degrade performance. For this reason, we believe that the barriers of obtaining the private key (which is never transmitted by ssh) and the public key, discovering the passphrase, and gaining login access to a machine are sufficient to prevent NFS-mounted SSU keys from becoming a security hole.

Another problem with distributing keys via NFS is that if the NFS server goes down, the keys are inaccessible. If the trusted master holds all of the SSU keys, then it is possible to obtain access to any machine through the trusted master.

Without using NFS, it is advisable to keep a local copy of the private keys on each host. The private

keys used for root access to a host should be pushed to it by the trusted master, or obtained by querying the trusted master periodically. In this way, an intruder with access must gain physical access to the files on the host in order to gain access to the private keys. This method allows local root access even if the trusted master and NFS server are inaccessible.

Appendix D: Establishing a Secure Connection via SSH

by Michael Barrientos and Christopher Thorpe