Secure Automation: Achieving Least Privilege with SSH, Sudo and Setuid

Robert A. Napier – Cisco Systems

ABSTRACT

Automation tools commonly require some level of escalated privilege in order to perform their functions, often including escalated privileges on remote machines. To achieve this, developers may choose to provide their tools with wide-ranging privileges on many machines rather than providing just the privileges required. For example, tools may be made setuid root, granting them full root privileges for their entire run. Administrators may also be tempted to create unrestricted, null-password, root-access SSH keys for their tools, creating trust relationships that can be abused by attackers. Most of all, with the complexity of today's environments, it becomes harder for administrators to understand the far-reaching security implications of the privileges they grant their tools.

In this paper we will discuss the principle of least privilege and its importance to the overall security of an environment. We will cover simple attacks against SSH, sudo and setuid and how to reduce the need for root-setuid using other techniques such as non-root setuid, setgid scripts and directories, sudo and sticky bits. We will demonstrate how to properly limit sudo access both for administrators and tools. Finally we will introduce several SSH techniques to greatly limit the risk of abuse including non-root keys, command keys and other key restrictions.

Introduction

Since its introduction in 1995 by Tatu Ylonen, SSH has quickly spread as a secure way to login and run commands on remote hosts. Replacing the previous r-commands (rsh, rexec, rlogin), SSH provides much needed encryption and strong authentication features. Relying on public/private key techniques, SSH is very resistant to man-in-the-middle, IP spoofing and traffic sniffing attacks, all of which were significant problems with the r-commands. SSH was initially released under a free license, but has since split into commercial[1] and free versions. In this paper we will focus on the most popular free version, OpenSSH.

Sudo was developed in1980 to allow users to execute commands as root without using the root password. Today it provides per-host and per-command access control features and powerful logging facilities to track what is done by whom.

Setuid (also called ''suid'' or ''Set UID'') allows a UNIX program to run as a particular user. If the executable is owned by root for example, the program will run as the root user, giving it privileges that may be needed for its function. The passwd password-changing program is a good example of this, since it requires root privileges to write to /etc/shadow which holds user passwords. Setgid provides the same functionality for UNIX groups, giving the program access to files writable only by a particular group. For example, in FreeBSD programs that read system memory are setgid to a special kmem group.

These tools and features are available for all modern versions of UNIX, and are installed by default on most of them. All of them can be used to help enhance the principle of least privilege, which we will discuss at length here.

The Situation

Consider the following system administration environment and compare it with your own experience:

- SSH has universally replaced rsh, but null-password keys have been deployed to provide unrestricted root access to automation tools;
- Sudo has replaced the root password for most administration functions, but admins generally only use it to obtain root shells and almost never employ it in automation tools;
- Custom setuid scripts almost exclusively run as root and setgid is seldom used;
- Automation tools that require any root access, no matter how little, run as unrestricted root through root cron, root ssh and similar mechanisms;
- Automation tools receive little security review, even when granted wide-ranging privileges.

Such environments have been the norm in the author's experience. If you have a similar environment, this paper will introduce the ideas behind least privilege and how these tools can be used to enhance least privilege in your environment.

[1] The commercial version of SSH is owned by SSH Communications Security. The parts of this paper which refer to "commercial SSH" are based on SSH Secure Shell 3.2. Starting with version 4.0, this product is known as SSH Tectia.

The Risks

Some of the risks in the environment described above include:

- Null-password root SSH keys.[2] If an attacker can get to that key, she will have complete control over all machines that accept it. Even if your application is secure, any mechanism that an attacker can use to get to that file is fair game.
- Sudo passwords. Every account that has unrestricted root sudo access is another root-equivalent password for an attacker to guess or steal.
- Sudo hijacking. In sudo's default configuration, an attacker who can run commands as a sudo-enabled user can hijack that user's sudo privileges even without access to the user's password.
- Sudo escalation. It can be extremely challenging to limit sudo access to a few commands. Without great care, limited sudo can be trivially translated into full sudo access. While you may trust the user you granted access to, do you also trust the attacker who has stolen his identity?
- Script exploitation. Scripts that run as privileged users are obvious targets for attackers. Errors in the scripts are subject to exploitation. Setuid scripts are particularly susceptible because they are often written in scripting languages like Perl or Bourne Shell and can be read by an attacker searching for vulnerabilities.

We'll discuss how to mitigate all of these.

The Causes

It's tempting to simply blame ''coder laziness'' for this situation, but this isn't the case. There are several factors that we will need to address:

- Trust in ''instant security.'' Neither SSH nor sudo can be simply ''dropped in place'' and deliver an ideal security environment. While SSH is far better out of the box than rsh, it has its own security issues that have to be considered, and converting automation tools to use it can be difficult without tearing down some of its benefit. Similarly, sudo introduces several security concerns, some of which are worse than what it replaces (such as a greater number of root-equivalent username/password combinations). This is not to discourage the use of these tools, but they do not magically instill security on their own.
- Lack of best practices guides. There are limited resources available explaining the best way to set up SSH and sudo. Out of the box, sudo does not even have all of its security features turned on and is subject to hijacking (as we'll discuss below). SSH command keys are mentioned in the man pages, but there are few resources really explaining their use or the use of other SSH key restrictions.
- Added complexity. Many of the techniques in this paper increase the complexity of developing and deploying automation scripts. Automation is hard enough to just get working, let alone get working securely. If developers are only rewarded for functionality, then there is little incentive to take on the added support headaches of a more secure solution.

This paper will address the first two causes. Addressing the third is often a cultural and infrastructure challenge that can only be solved on a case-by-case basis.

The Goal: Least Privilege

Now that we've discussed what may be wrong with our environment, what do we want our environment to look like? In this paper we will mostly focus on least privilege, which is one piece of the bigger goal of layered security.

Layered security means that safeguards overlap such that if one fails, an attacker will still not have damaging access. Least privilege helps ensure that if a particular user's account is compromised, for whatever reason, the damage the attacker can do with it is limited as much as possible. This is why ''don't you trust me?'' should never be the argument for excessive privileges. Wherever possible, trust should be compartmentalized.

UNIX-like systems provide numerous ways to restrict privileged access. In this paper we will discuss the following techniques:

- Restricting SSH connections in what they can execute and where they can originate;
- Limiting privileged access through sudo by coupling it with non-root setuid;
- Replacing root-setuid with non-root setuid and setgid;
- Reducing the number of privileged processes with sticky bits and setgid directories.

Whenever a process or user needs elevated privileges, it should be second nature to ask precisely what privileges the process or user needs, and how to best limit the process or user to exactly those privileges.

When discussing the principle of least privilege, one might ask ''why would we have hired these people if we didn't trust them?'' Least privilege has little to do with the trust we have for our employees. Instead, it deals much more with the number of avenues an attacker has for exploiting the system. Of course an administrator should have every access she needs, but conversely she should have no access that she has no need for. How strictly ''need'' is defined is a serious trade-off to consider, but just requiring that an administrator explicitly request specific access, even if it is always granted, can go a long way towards controlling the number of avenues an attacker can use. If an attacker is successful, being able to enumerate the accounts with access is also a major benefit to investigators in determining possible further compromises.

[2] Throughout this paper, the term ''SSH key'' will be used to refer to both RSA and DSA keys.

When granting privileges to automation tools, one might assume that the security of a particular tool isn't very important if the data it deals with is non-sensitive. It is critical to always consider how a tool could be exploited to attack other parts of the network, not just the parts it's intended to control.

Moving towards least privilege, especially for automation tools, has other benefits. Establishing least privilege requires developers to understand the privileges actually used by their tools, which in turn forces them to understand what their tools are doing. Understanding software is a key step towards maintaining it. Moreover, simply enumerating the privileges that a tool requires can help a developer see how to reduce the number of privileges required. Does the tool really need the ability to "run an arbitrary command on any host in the system" or did it really just need the ability to "get a directory listing for a specific directory on three hosts?"

Least privilege is a philosophy, not a technology. By consistently employing it, an organization can better understand and control the security of the environment while still maintaining a strong culture of trust for the administrators.

Hardening the Environment

This paper focuses on automation techniques, but some basic environment hardening will set the stage for a secure automation environment.

Understanding the Environment

In a complex environment with many users and administrators, it is easy for trust relationships to grow throughout the system with little documentation or understanding. To combat this, it is helpful to create a directed trust graph of your network, indicating particularly how root can move through the system using SSH, rsh and other mechanisms (such as custom administration daemons and web scripts that are sometimes developed in large environments). There are few tools to automate this today, but even manually developing such a graph with tools like Microsoft's Visio or AT&T's Graphviz can provide significant insight into your environment.

Understanding what users and hosts are trusted with wide-ranging root access provides a road-map for improving enforcement of least privilege. There will always be a few places in any large system that require broad trust; understanding these will give a roadmap for hardening.

Similarly, administrators should maintain a catalog of known setuid and setgid programs and audit systems regularly for the creation of new ones.

Hardening and Managing SSH

Authorized Keys

By default, SSH relies on files in the user's home directory for certain authentication options. Chief among these is the authorized_keys[3] file. This file defines what keys will be accepted without a password and under what conditions, and will be the subject of several SSH techniques in this paper. Anyone who can write to this file for a particular user can log in as that user. This means that user home directories, particularly the ~/.ssh directory, are highly sensitive. Unfortunately if home directories are NFS mounted, there are a number of ways that attackers may be able to write to arbitrary user directories, and thereby update authorized_keys with keys the attacker controls.[4] The solution is to move authorized_keys out of the users' NFS-mounted home directories and onto local storage, generally under /var. For example, the following setting in sshd_config will read authorized_keys from /var/ssh/*user*/authorized_keys:

```
AuthorizedKeysFile /var/ssh/%u
```

Of course you will need to create directories for the users under /var/ssh which only they can write to. Users will also need to create separate authorized_keys files for every server. This differs from many users' behavior of setting up a single authorized_keys file for all servers (since it is mounted by NFS). While this has some overhead, it once again encourages the principle of least privilege in that only machines for which the user explicitly requests passwordless connections will accept them.

Root Keys

Unrestricted SSH keys accepted by root are extremely powerful and should be avoided. Administrative users should generally use their own credentials to log into a server and then use sudo to gain root access there. Automation scripts that require remote root should use command-keys, which will be discussed further in "Command Keys." To enforce this, the PermitRootLogin option in sshd_config should be set to forced-commands-only.

Known Hosts

SSH provides powerful features to prevent server spoofing and man-in-the-middle attacks. Most notable is the use of public keys to strongly identify servers. This technique is not fool-proof however. SSH keys cannot be signed as X.509 certificates are, so unless you've received the server key from a trusted source, you have no way to know that the key is legitimate. There are three primary ways to get server keys: LDAP, centrally managed ssh_known_hosts and user-managed known_hosts. We will also briefly discuss using X.509 server certificates with commercial SSH.

Commercial SSH allows server keys to be centrally stored in LDAP, which is generally easiest to

[3] This paper uses the OpenSSH filenames and formats for configuration files. Commercial SSH uses slightly different file names and in some cases formats.

[4] Computer Incident Advisory Capability, CIAC Notes 95-07, "NFS export to unprivileged programs." See http://ciac.llnl.gov/ciac/notes/Notes07.shtml .

manage. OpenSSH and most free Microsoft Windows clients (such as Putty) cannot retrieve server keys from a central LDAP server, but for installations using commercial SSH, managing the server keys centrally is highly recommended. Whenever a server key is generated, it should be added to LDAP. For environments with multiple networks supported by different organizations, or for dealing with servers outside of your environment, commercial SSH supports multiple LDAP servers.

All UNIX SSH clients support a file called ssh_known_hosts, generally stored in /etc or /etc/ssh, which contains the official list of server keys. This file must somehow be distributed to all clients.[5] This file could also be NFS mounted, but this reintroduces the NFS security problems discussed above. Even so, NFS mounting this file may be better than not managing ssh_known_hosts at all. Centrally managing ssh_known_hosts is generally only effective within an organization. Since there can be only one file and it needs to be read from disk, there is no good way to include other organizations' host keys. Furthermore, since the users must trust the provider of the central ssh_known_hosts file to provide legitimate keys, this file can only be accepted as far as trust extends within the environment (generally as far as the central support organization).

If a server is not listed in the central ssh_known_hosts, SSH will by default prompt the user to add the key to the user's known_hosts, stored in ~/.ssh. This is the least secure option, since the user has no good way to determine the authenticity of the key. Once a key has been added to the user's known_hosts, however, SSH will warn the user if a server ever responds with a different key. This could indicate that a machine is being spoofed. Unfortunately it could also mean that the machine has been legitimately replaced. In environments where this is common, users have no good way to determine whether the warning is legitimate. To avoid these problems, it is highly recommended that ssh_known_hosts be centrally managed rather than rely on users' known_hosts.

Failing to centrally manage ssh_known_hosts creates special problems for automation scripts. Since scripts have no way to respond to the new key, they will fail if the key changes. This is a good thing in that it protects scripts from machine-spoofing, but it does create administrative headaches when scripts start failing due to a key change. Once again, the best solution to this problem is central management of ssh_known_hosts.

Commercial SSH improves this situation by allowing servers to use signed X.509 certificates rather than SSH keys. Since these keys are signed by a Certificate Authority, clients can rely on their authenticity without having all the keys in advance, greatly simplifying the administrative overhead of key management. Since most free clients (including OpenSSH) do not support these certificates, they are most useful in a completely commercial SSH environment, but in such an environment they are highly recommended as an alternative to ssh_known_hosts or LDAP.

[5] This works for managed UNIX clients, but has no good parallel for Windows clients.

Hardening Sudo

Unrestricted sudo effectively creates additional root-equivalent passwords for an attacker to guess or steal. Each administrator's password must now be protected with the same care as the root password. There are two approaches to mitigating this risk. Sudo-enabled administrative accounts can be separated from the administrator's regular account. Doing so will greatly reduce the opportunities for an attacker to steal the sensitive password. Alternately, sudo can be compiled to work with several one-time password systems such as OPIE, S/Key and SecurID. Deploying such systems is non-trivial, can be expensive and is beyond the scope of this paper.

Sudo has a significant security flaw in its default configuration that permits hijacking in which an attacker can make use of the victim's sudo privileges without the victim's password. Sudo uses tickets, files that are created to only require a user to enter her password at certain intervals. By default these tickets are created on a per-user basis, so if the user is logged on multiple TTYs on the same host, her ticket is valid for all of them. While modestly convenient, this is a significant security hole. If an attacker is able to run an arbitrary process as the victim user, then the attacker can piggyback on the victim's sudo privileges even without the victim's password. When the victim uses sudo, the attacker then has a five minute (by default) window to use sudo without a password. Coupled with the NFS authorized_keys attack discussed above, this is a very significant attack against administrative users.[6]

There is a complete but inconvenient solution to this, and an incomplete but fairly easy solution. The complete solution is to turn off password caching entirely, either by compiling with --with-timeout=0 or by setting passwd_timeout to 0 in the central sudo configuration file, /etc/sudoers. Doing so completely closes this particular attack, but strongly encourages system administrators to use a root shell to avoid retyping their passwords repeatedly. Since root shells cannot be easily logged, this is a significant auditing trade-off.

The less drastic solution is to compile sudo with --with-tty-tickets or set tty_tickets to ''on'' in sudoers. This will create a separate ticket to each user/TTY combination, stopping an attacker from piggybacking on the ticket in many cases. This is not a complete solution, however. The attacker can still attack the victim's login scripts to have the attack happen within the

[6] ssh-agent [OSSH] can be similarly attacked in order to make use of another user's SSH key. This seldom impacts automation tools because they are less likely to use ssh-agent, but it is worth keeping in mind for administrators.

victim's TTY. The attacker can also attempt to login to the server immediately after the victim logs off. On many operating systems (including Solaris, *BSD, and Linux) the attacker will often be allocated the same TTY as the victim had, and the ticket may still be valid. This latter attack can be mitigated with logout scripts that run ''sudo -k'' to destroy tickets, but it can be challenging to ensure that all administrators run this logout script. So turning on TTY tickets is better, but to completely close this hole, password caching has to be turned off.

It is very difficult to manage sudo such that users cannot escalate their privileges. This will be discussed further in ''Controlling Sudo.''

Limiting Privilege with SSH Command Keys

The most significant way to limit the power of an SSH key is to apply a command restriction. When a user connects using an SSH key with a command restriction, or a command key, a pre-defined command runs rather than providing the user with a shell. Applying this to root access, along with setting PermitRootLogin to forced-commands-only,[7] provides a powerful way to control automation tools. If the automation tool runs as a non-privileged user and only has access to a particular root command key, then that tool can get the root access it needs while reducing the ability to subvert it into performing arbitrary actions as root.

For most of the examples in this paper, we will consider the same simple task. We will change Apache's ErrorLog entry on a remote host to include the current month and restart Apache. This is a somewhat contrived example, since this would generally be done in simpler ways, but it demonstrates some of the main issues. The script we wish to run, update_errorlog, is shown in Figure 1.

To create a command key that runs update_errorlog, first create a keypair on the source machine:

```
$ ssh-keygen -t dsa -f errorlog_key
```

You now have a public key called errorlog_key.pub and a private key called errorlog_key. Prepend this with your command restriction and append it to ~root/.ssh/authorized_keys on the target machine. The format is as follows:

```
command="/usr/bin/update_errorlog"
                    [public_key]
```

[7] This only allows root to accept command keys, so there cannot be root-level SSH login keys.

Now login using the new key and the script will run:

```
$ ssh -i errorlog_key \
              root@target.example.com
```

Non-root Keys

Many remote functions do not require root access at all. By creating special users for these functions and providing them distinct SSH command keys, attackers who are able to steal the key will have extremely limited access.

This can be combined with sudo to provide functionality very similar to root command keys. By granting the special user specific sudo privileges, it is possible to create scripts that use root precisely when they need it and no more. As an example, we'll run update_errorlog (Figure 1) using a non-root SSH key.

On the target machine, create a new group apacheconf that can write to httpd.conf. We don't want to use the apache group itself, because httpd should not be allowed to write to its own configuration files (otherwise a security flaw in Apache could be used to reconfigure Apache). Use a low-numbered GID to help distinguish it from user accounts. Put httpd.conf into the apacheconf group so that our new group can manage it without root access.

Now create a new user, updatelog, to run update_errorlog. Put it into the apacheconf group and give it a low-numbered UID to help distinguish it from user accounts.

Our update_errorlog (Figure 1) script now needs a small modification, adding ''sudo'':

```
[...]
sudo /usr/sbin/apachectl graceful
```

Edit sudoers to grant the errorlog account permission to run ''/usr/sbin/apachectl graceful''.

Finally, set up a command key as we did in the ''Command Keys'' section, but instead of making it a root SSH key, make it an SSH key for errorlog.

We can now restart update httpd.conf from source. example.com:

```
source$ ssh -i errorlog_key \
          errorlog@target.example.com
```

Originator Restrictions

Keys (both regular keys and command keys) can be further restricted to specific originating hosts using the ''from'' option in authorized_keys. For example:

```
from="*.example.com,*.example.net"
     ssh-rsa AAA.oeTp0=rnapier@adminhost
```

```
#!/bin/sh
PATH=/bin:/usr/bin:/usr/sbin
date=`date +%F`
# Rewrite httpd.conf
perl -eip "s!^ErrorLog(.*)!ErrorLog /var/log/error_log.$date!" \
   /etc/httpd/conf/httpd.conf
# Restart Apache
apachectl graceful
```

Figure 1: update_errorlog.

This key will only permit connections from machines within the example.com or example.net domains. The "from" option accepts a comma separated list of canonical names or IP address including wildcards. Note that rnapier@adminhost is only a comment to give a hint where this key was created and has no impact.

Originator restrictions based on DNS names are reliant on trustworthy name services, but this still greatly increases the complexity of attack. The attacker must already have stolen and possibly cracked the private key, and then will still have to poison or compromise DNS in order to make use of that key.[8]

Other Restrictions

Most extended SSH features can be turned off on a per-key basis. This includes X-forwarding, port-forwarding, PTY-generation[9] and similar features. It is generally a good idea to turn off any features you don't need. For example:

```
no-port-forwarding, no-X11-forwarding,
    no-agent-forwarding,no-pty ssh-rsa
    AAA...oeTp0=rnapier@adminhost
```

Controlling Sudo

The Pitfalls of Limited Sudo

Using sudo to give limited access to root is a very tricky proposition, since the most obvious sudo configurations can be easily escalated to unlimited root access. As we discussed in the Introduction, even if you trust the user not to do this, you also have to trust the attacker who gains access to the user's password (or subverts sudo in some other way).

Some exploitable situations include:

- Permission to run commands in a user-writable directory.
- Access to chmod (even more easily exploitable with access to chown or cp)
- Access to any command with shell-outs (vi, emacs, ed, edit, more, less, find), though version 1.6.8 promises to help here
- Access to any command that can write (especially append) to an arbitrary file (vi, emacs, ed, edit, tee, less)
- Access to root's crontab or atjobs (crontab, batch, at)
- Any command that honors PAGER, EDITOR, or VISUAL (man, less, more)
- In some cases, any command that can read an arbitrary file (cat, less, more, tail). These can be used to get /etc/shadow for offline cracking, or can be used to read other protected files
- Access to sudo itself as root. This allows attacks like "sudo sudo /bin/sh". There are options to prevent this (--disable-root-sudo at compile time, or unsetting root_sudo in sudoers), but these are fairly weak protections meant to stop administrators from circumventing a !SHELLS entry. If you need these options, then you're probably allowing so many other commands (like those above) that an attacker can easily gain a root shell anyway.

With the release of sudo 1.6.8, two new features have been added that make limited sudo somewhat easier to implement. A common sudo need is to allow the editing of a protected file. This has historically been very difficult to provide in a controlled way without writing wrapper scripts. A user who is allowed to run an editor as root can almost certainly modify arbitrary files and trivially gain shell access. The new "-e" option to sudo, also accessible by running sudoedit, fixes this. It makes a temporary copy of the target file that is owned by the user. The user is then provided the editor of their choice, but since they are still running under their own userid there is no security issue. When the editor exits, sudo will replace the original file with the temporary copy. In the past, some administrators have written scripts to do just this, but moving this functionality into sudo itself should make things much easier. To allow a user to use sudoedit, treat it like any other command, but don't give a full path to it. The alias "sudoedit" represents either sudoedit, or "sudo -e". By appending a filename, you can restrict the user to editing particular files. For example:

```
rnapier host=(root) sudoedit /etc/httpd.conf
```

Another major improvement in 1.6.8 is the addition of a NOEXEC option.[10] On operating systems that support it,[11] the NOEXEC option will prevent a command run under sudo from calling exec() itself. This will prevent the shell-outs that provide trivial root shells from so many commands from editors to pagers. Given the newness of this technique, only time will tell how effective it is in practice.

The solution to providing limited sudo is single-purpose wrappers, small scripts written to do exactly what is required. By providing sudo access to just these wrappers, least privilege can be much better achieved.

For example, let's consider a script mysqllog, which prompts the user for her password, validates it against /etc/shadow, and if successful, displays /var/log/mysqld.log. This log file is owned by the mysql user and group-owned by the mysql group. It is only readable by user and group.

[8] SSH protects clients from connecting to the wrong server through host keys, but it doesn't protect servers from hostile clients. If a user shows up with the correct user key, no client host key checking is done. Even with the "from" restriction, only the DNS name is checked, not a host key, since there often will be no host key for a client.

[9] Many UNIX commands, most notably ls, have different newline handling if there isn't a PTY. If your tool can't handle this, you may need to allow PTY creation.

[10] [SUDO], sudoers man page, "NOEXEC and EXEC."

[11] This includes at least SunOS, Solaris, *BSD, Linux, IRIX, Tru64 UNIX, MacOS X, and HP-UX 11.x. It does not work on AIX and UnixWare. [SUDO]

```
#!/usr/bin/perl -w
use strict;

sub error { print @_; exit 2 };

if( $> != 0 ) { error "Must run as root\n"; }

my $good_pwd = (getpwuid($<))[1] or error($!);
chomp( my $test_pwd = <> );
if (crypt($test_pwd, $good_pwd) ne $good_pwd) {
   exit 1;
} else {
   exit 0;
}
```

Figure 2: checkpass.

```
#!/usr/bin/perl -w
use strict;
my $user = getpwuid($<);
print "Password: ";
system( '/bin/stty', '-echo' ); # Don't echo
my $password = <>;
system( '/bin/stty', 'echo' ); # Do echo
print "\n";

open( CHECKPASS, '|-', '/usr/bin/sudo',
          '/home/rnapier/checkpass' )
   or die $!;
print CHECKPASS $password;
close CHECKPASS;
if ($? == 0) {
   system( '/usr/bin/sudo /bin/cat'.
            ' /var/log/mysqld.log' );
}
else {
   print "Bad password.\n"
}
```

Figure 3: mysqllog.

Since mysqld.log is group-owned by mysql, the script will need to have access to that group. It also seems to need root privileges in order to access /etc/shadow. The obvious solution is to simply make it a setuid root script, but this would give it far more access than it needs. In fact, this script doesn't actually need to be able to read /etc/shadow; it only needs to be able to verify that a given username/password combination is valid. Carefully stating your privilege requirements is the first step towards achieving least-privilege.

As before, we'll create a user, mysqllog, to run this script and edit sudoers to give it permission to run "checkpass" and "/bin/cat /var/log/mysqld.log".

The checkpass script is listed in Figure 2. It reads a password for the current user from STDIN. It then exits with a 0 to indicate a good password, a 1 to indicate a bad password, or a 2 to indicate an error. We pass the password in on STDIN because command line parameters can be seen in the process table by all users. Note that this script can only validate the current user, not an arbitrary user. Once again we keep to least privilege.

The code to perform our task is shown in Figure 3. We make it setuid to the mysqllog user we created earlier. Now arbitrary users can run this script, enter their password, and get the contents of mysqld.log. Even if an attacker can find a bug in the script, the privileges that can be exploited are very limited.

Setuid/setgid vs. Sudo

Setuid scripts can check the calling user to determine whether they have rights to run this script (generally by checking group membership). This is convenient for the authorized user, because she is saved the trouble of typing "sudo." An example of such a check in Perl is given in Figure 4.

```
#!/usr/bin/perl -w

my $group_wheel = getgrnam( 'wheel' )
                                 or die;
if( $( !~ /\b$group_wheel\b/ ) {
   die( "Must be part of wheel".
    " group to run this script.\n" );
}
```

Figure 4: Checking group membership in Perl.

Alternately, privilege-requiring scripts can be executed using sudo. This has the advantage of providing centralized accounting of all privileged users and reducing the complexity of the scripts.

Non-root Sudo

One should always consider when using sudo whether the user needs root access or whether access to a non-privileged user like apache or jabber might be sufficient. Sometimes changing the ownership or group of configuration or log files is enough to allow less-privileged accounts to manage them. Be careful with this, however. Many services like Apache should not be run under a UID that can write to their configuration files. Doing so could allow a minor compromise to be escalated into a larger compromise by allowing the server to be reconfigured by an attacker. That said, there is no reason that Apache's configuration files can't be owned by an apacheconf user and administrators given appropriate sudoedit privileges to that user.

Setuid/setgid with Sudo

Setuid/setgid can be combined very effectively with sudo. For example, the script can be setgid to a special group. This group can then be given root sudo privileges to run specific single-purpose wrappers. The script can then use sudo to execute these wrappers to escalate to root privileges precisely when needed, and only for precisely what is needed. Furthermore, since the single-purpose wrappers are not themselves setuid, they can only be called indirectly, by already-privileged processes. This helps prevent an attacker from passing them unusual parameters, making them less susceptible to security coding flaws.

As an example, we can achieve the same functionality as in update_errorlog (Figure 1) on our local machine using a setuid script (Figure 5). As in the "Non-root keys" section, we'll set up an errorlog user, including its sudo privileges, and make the script setuid to errorlog. When you run update_errorlog it will then update httpd.conf and restart Apache as long as you are in the wheel group.

Similar techniques can be used with SSH command-keys or CGI scripts that are run under specific user IDs.

Setuid/Setgid Best Practices

Non-root Setuid

"Setuid" is sometimes confused with "run as root," but this need not be the case. Setuid can be used to run a command as any particular user by chowning the file to that user.

As with sudo, always consider whether your setuid script really needs root access or just access to a special user. For example, if a script needs access to a file containing a password, there's no reason that file needs to be owned by root. It could be owned by any non-user account.

Reducing the number of setuid-root scripts reduces the number of ways attackers can exploit coding errors to obtain root.

Setgid

In some cases, setgid can be even more useful than non-root setuid. As we saw in the "Non-root keys" section, creating a group to manage configuration files such as for Apache can help isolate access to these files from root access. Setgid scripts can grant users access to these protected files, while still preserving the user's own privileges (such as access to their home directory) without any special handling of effective UID.

Setuid Script Obfuscation

Setuid scripts in languages like Perl and Python present a special problem. They have to be readable by the user, giving an attacker an opportunity to study them looking for security flaws to exploit. The attacker may even be able to copy the script to another machine to test possible exploits offline.

Compiled programs do not generally have to be readable by the user; they only require that the executable bit be set. So when writing setuid and setgid scripts in interpreted languages such as perl or python, there is some value to creating a small wrapper in C, as shown in Figure 6.

```
#include <stdio.h>
#define CMD "/usr/local/protected/myscript"
main(ac, av)
  char **av;
{
  char error[80];
  execv(CMD, av);
  snprintf( error, sizeof( error ),
                "Unable to run %s",CMD );
  perror( error );
  exit( 1 );
}
```

Figure 6: myscript.c setuid wrapper.

In the above example, /usr/local/protected should only be readable by the setuid user (often root), and myscript should be replaced with script filename.

Keep in mind that this is an obfuscation technique, not a security technique. If your script had no security flaws in it, then this technique wouldn't be needed and using this technique doesn't prevent an attacker from exploiting your script's security flaws. It just makes finding the flaws harder.

While languages like Perl and Python have special handling to make setuid scripts "safe" (though readable by the user), it is not trivial (or even possible on some older platforms) to make Bourne and similar shell scripts setuid safely. Most operating systems don't even allow this anymore.[12] These will absolutely require a wrapper script, though it would be wise to

[12]Shell scripts are subject to environment attacks including manipulation of PATH or IFS, and timing-based attacks based on moving links around between the time that the script is started and the script is read. Some of these have been addressed in modern versions of UNIX-based operating systems, but because of Bourne shell's reliance on exter-

```
#!/usr/bin/perl -w
use POSIX qw(strftime);
my $group_wheel = getgrnam( 'wheel' ) or die;
if( $( !~ /\b$group_wheel\b/ ) {
   die( "Must be part of wheel group to run this script." );
}
my $file = '/etc/httpd/conf/httpd.conf';
my $date=strftime("%F", localtime);
if( -e "${file}.bak") { unlink( "${file}.bak" ) or die "$!" }
rename( $file, "$file.bak" ) or die "$!";
open( INFILE, "$file.bak" )  or die "$!";
open( OUTFILE, ">$file" )    or die "$!";
while( <INFILE> ) {
   s!^ErrorLog(.*)!ErrorLog /var/log/error_log.$date!;
   print OUTFILE;
}
close INFILE  or die $!;
close OUTFILE or die $!;
system( qw(/usr/bin/sudo /usr/sbin/apachectl restart) ) == 0 or die;
```

Figure 5: update_errorlog in Perl.

write setuid scripts in a language like Perl or Python in any case. Bourne shell and its cousins rely very heavily on the operating environment and external programs and so are much harder to adequately secure in a setuid context.

Perl in particular provides taint checking, which helps programmers keep track of what data could have been influenced by the user. Setuid Perl scripts automatically turn on taint checking. If you use a C-wrapper as above, Perl will no longer automatically turn on taint checking, so you should do so by passing ''-T'' in the perl invocation.

Finally, whenever possible, make setuid and setgid programs unreadable by anyone but the owner.

Best Practices in Handling Privileged UID/GIDs

Ideally all ''special'' UIDs and GIDs should have a consistent numbering convention. Generally, numbers under 100 (or 1000 for larger systems) should be reserved for these special IDs.

Special UIDs should not generally permit direct login. They should not have a valid password or shell.

It is often convenient for administrative staff to belong to special GIDs so that they can manage configuration or data files directly without needing further access (such as sudo). This is particularly useful for allowing non-root users to administer particular parts of the system.

Odds and Ends

Sticky Bits

Setting the sticky bit on a directory allows users to write files that other users cannot remove, even though the directory is world writable. In some cases this can get rid of the need for setuid user scripts to write protected files. /tmp is a good example of where this is used. Setting the sticky bit is done as follows:

```
chmod o+t directory
```

Setgid Directories

Setting a directory setgid will cause files created there to belong to the same group as the directory rather than the user's primary group. In some cases this can get rid of the need for privileged daemons that need to read things created by users.

Combining this with the sticky bit is an effective way to create a drop-box location for a non-privileged daemon. Users can put things into this directory, but they can't list the entries in the directory (since we won't add the directory read privilege), and they can only remove their own files (because we'll set the sticky bit).

Create a directory ''drop'' and set the sticky and setgid bits:

```
# chown myservice:myservice drop
# chmod u=rwx,g=rwxs,o=wxt drop
```

This means that anyone can drop files into the drop directory, users can't modify each other's files and the myservice user doesn't need any special privileges to manage files dropped into this directory.

Web Applications

By default, web applications run as the web user. On a system with multiple web applications, each application will have access to the others' data in this configuration. By creating separate accounts for each application, they can then be separated using suexec, an Apache feature that causes CGI programs to run under user accounts rather than as the web server. This can protect application data from exploits against other CGI programs as well as the web server itself.

Recommendations for the Future

There are several features that would help large installations manage sudo and SSH better. Some of these are currently possible with custom work, but they should be integrated better into the products.

- SSH should be able to get its authorized_keys information easily from LDAP or Active Directory rather than the user's .ssh directory. This would allow the list of authorized keys to be protected from NFS attacks without resorting to local configuration files on each host (as described in ''Hardening and Managing SSH''). To maintain least-privilege, LDAP keys should be assignable to specific servers (rather than being globally accepted), and would need to allow restrictions such as ''from'' and ''command''. The existing SSH LDAP solutions[13] allow X.509 certificate authentication of users, but do not fully replace the authorized_keys file.
- SSH needs better integration with one time passwords (OTP). In particular, it should be possible to configure keys that allow an interactive shell to require one-time passwords while not requiring this for command keys. Interactive shells are extremely powerful and should always have a human available to enter the OTP. Command-keys are very restricted and generally won't have a human available to enter the OTP. This is difficult to implement with the current SSH tools, which generally requires all-or-nothing use of OTP.
- Sudo needs to be able to get sudoers configuration from LDAP. This would make it much easier to integrate with a Role Based Access Control (RBAC) system or other centralized account and authorization systems. It is currently possible to generate a sudoers file out of LDAP with custom tools, but this is cumbersome and creates

nal programs for most handling, it is very difficult to protect yourself from all of them. Most modern UNIX-based operating systems do not permit setuid shell scripts. See the UNIX FAQ Question 4.7 at http://www.faqs.org/faqs/unix-faq/faq/part4 for more information.

[13] LDAP is managed by the ''Certificate Authentication'' feature of commercial SSH and the ''OpenSSH LDAP Public Key Patch'' (http://ldappubkey.gcu-squad.org/) for OpenSSH.

a delay between when LDAP is updated and when the change takes effect.
- A tool that would automatically determine trust relationships created by sudo and SSH and display this in a consolidated format (such as a directed graph) would be extremely valuable.
- Sudo should use TTY tickets by default and optionally clean up old tickets automatically whenever sudo is run.

Availability

OpenSSH is freely available under the BSD license from the OpenBSD project. Their website is available at http://www.openssh.org . At the time of this writing, the currently available version is 3.8.

SSH Secure Shell, discussed in this paper, has been replaced by SSH Tectia. Both are commercial products available from SSH Communications Security (http://www.ssh.com). Where this paper refers to the commercial product, it is written to SSH Secure Shell version 3.2.9. At the time of this writing, the most recent version is SSH Tectia 4.1.

Sudo is freely available and maintained by Todd Miller (Todd.Miller@courtesan.com) at http://www.courtesan.com/sudo. At the time of this writing, the most recent version is Sudo 1.6.7p5, though some features of the upcoming 1.6.8 are discussed in this paper.

Conclusion

In this paper we have established the importance of the principle of least privilege to the overall security of an environment, by reducing the avenues of attack and the extent that any particular attack can compromise the system as a whole. We have discussed problems with the techniques that may currently be used in many environments including unrestricted SSH keys for automation tools and setuid tools with excessive privileges. Finally we have provided techniques and examples of how to apply least privilege to real-world automation problems, including restrictions on sudo and SSH, wrapper scripts, setgid and sticky directories. While these techniques are useful and important, even more important is the philosophy behind least privilege. By constantly asking ourselves what the minimum set of privileges a particular operation needs, and challenging ourselves to reduce and compartmentalize those privileges, the security of our environments will not only improve, but become pervasive.

Author Information

Robert A. Napier is a founding member of the Corporate Information Asset Protection team for Cisco Systems where he trains internal groups on classifying and protecting sensitive information with special focus on technical safeguards. He is a member of the GCUX board and also holds IAM and CISSP certifications. He is a co-author of *Special Edition: Using Linux 6e* and a contributing author to *Special Edition: Using KDE* and *Red Hat Linux Installation & Configuration Handbook*. He can be reached electronically at rnapier@employees.org .

References

[SUDO] Miller, Todd, *Sudo Main Page*, http://www.courtesan.com/sudo , 2003.

[OSSH] OpenBSD, *OpenSSH Manual*, http://www.openssh.org/manual.html , 2004.

[SSH] SSH Communications Security, *SSH Secure Shell for Servers Version 3.2.9 Administrator's Guide*, http://ssh.com/support/documentation/online/ssh/adminguide/32 , 2003.