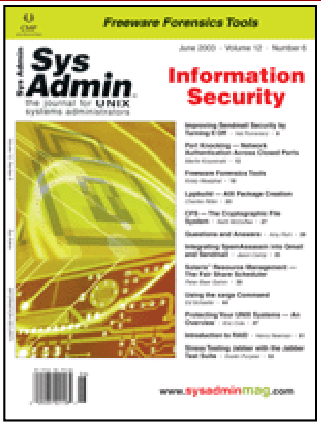
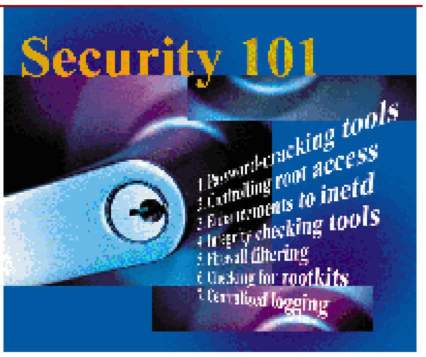


Article

jun2003.tar

Protecting Your UNIX Systems -- An Overview



Eric Cole

Before you read this article, I'd like you to take a quiz. Pick one of the UNIX servers at your organization and just from memory, list all of the ports that are open and the corresponding services that are running on your system. Don't get upset if you can't list them all -- most people can't. However, if you don't know what the gateways of access into your system are, you can't know whether your systems are secure or whether they have been attacked.

Every company possesses intellectual property that makes it unique, and in order to protect that property, you must identify exactly what it is for all levels within your entire organization. Once you have identified that information, you must, of course, find out where it resides. In some organizations the information is spread over a large number of systems. In these situations, it might be useful to look into whether consolidation of data is a worthwhile effort. Consolidating sensitive data onto fewer servers reduces your exposure and makes the data easier to secure. However, you should also make sure you don't put all of your eggs into one basket. Ensuring that there is enough reliability across those systems is also important.

Tools for Protecting Your Critical Data

Once you've identified which servers the data resides on, you must make sure those servers are as secure as possible. In this article, I will provide a general introduction to some basic but critical tools used to protect your data:

1. Password-cracking tools

2. Controlling root access

3. Enhancements to inetd

4. Integrity checking tools

5. Firewall filtering

6. Checking for rootkits

7. Centralized logging

Password Cracking Tools

Password controls on UNIX systems have traditionally been weak. Efforts have been developed to produce password controls that ensure strong passwords, however, users still find ways around this. Through the use of PAM (Password Authentication Module) a dictionary attack can be performed on a password at the time the user enters it. This prevents users from entering extremely weak passwords, but is not completely fail-safe. The best method for ensuring strong passwords is to use password-cracking tools on a regular basis and force users to change weak passwords. To avoid potential legal liability associated with this type of scanning, I recommend running your password-cracking tool in such a way that you only crack weak passwords. For example, if your password policy requires that all passwords contain letters, numbers, and special characters, you could run a brute force attack looking for all possible combinations of the following: letters; numbers; and special; letters and numbers; letters and special; and numbers and special.

Thus, if a password contains letters, numbers, and special characters, it will not be cracked. If you then require users with weak passwords to change them, you have greatly reduced your liability issues.

Controlling Root Access

On UNIX systems, the root account, or more specifically any account with a UID of 0, has full access to the system. Because this account has unrestricted access to the entire system, attackers try to acquire the account. Root access is dangerous because the root account has no accountability -- if three people have the root password and someone logs in as root and does damage, you have no idea which one did it. Thus, individual root accounts should be given to each user to log into the system. Another common method is to use the **su** command. With this method, administrators log in under their normal user accounts, then type **su** and enter their password to elevate their access to root. This provides more accountability.

Another practice involves instituting a principle of least privilege to provide users with the minimal amount of access they need to do their job and nothing else. Thus, the use of **sudo** is recommended instead of **su**. **sudo** provides more detailed logging as well as a finer level of access control.

Enhancements to inetd

When someone connects to a port, instead of directly giving them access to the service daemon, you can have a program sit between inetd and the service, providing firewall-like functionality and additional logging. TCP Wrappers, one of the traditional programs for doing this, uses a host.allow and hosts.deny file. Any entry that is in the hosts.allow file is allowed, and any entry that is in the hosts.deny file is denied access. However, if there is not an entry in either file, then the specific traffic is allowed into the network. This is not good; thus, your hosts.deny file should always contain the entry ALL:ALL. With that entry, the security stance becomes a default deny, which states anything that is not explicitly allowed is denied. The following is a default entry for inetd:

```
tftp  dgram udp wait root  /usr/sbin/in.tftpd  in.tftpd -s /tftpboot
```

And this shows the same entry with TCP Wrappers installed:

```
tftp  dgram udp wait root  /usr/sbin/tcpd  /usr/sbin/in.tftpd -s /tftpboot
```

You can see that when you install TCP Wrappers, inetd would call tcpd, which is the TCP Wrappers daemon instead of the actual program.

Newer versions of Linux have a xinetd file, which provides similar functionality to TCP Wrappers, but it's built into the OS. Setting up these rules for each service forces you to better understand your system, which will help make it more secure. With xinetd, the system creates a file for each service, which controls the behavior. This can be seen in the following:

```
# default: on
# description: The telnet server serves telnet sessions; it uses \
#    unencrypted username/password pairs for authentication.
service telnet
{
flags               = REUSE
socket_type         = stream
wait                = no
user                = root
server              = /usr/sbin/in.telnetd
log_on_failure     += USERID
disable             = no
}
```

Integrity Checking Tools

Attackers commonly modify key files in order to cover their tracks and create backdoors into the system. For example, if they created a directory called "eric" that they did not want the administrator to find, they would put a trojanized version of **ls** on the system. Then, when the administrator types **ls**, it will list every directory except eric. Tripwire is a common tool for performing integrity checks across key files. Tripwire performs a cryptographic hash against the file and then re-runs those hashes at certain intervals. Only if the files are exactly the same will the hashes match.

Firewall Filtering

Under Linux, firewalls like ipchains and iptables can be used to protect an entire network, but they can also be used to protect a single system much like a personal firewall. There are several tools available, such as netmon, which provide GUI-based interfaces for setting up these firewalls. Even if you have a corporate firewall, defense in depth is still a good stance to take. Depending on the version of UNIX, there could be a firewall built in. With the latest version of Linux kernel 2.4, Net Filter (iptables) support is built in. Here is a sample firewall ruleset:

```
iptables -A FORWARD -i eth1 -p tcp -s 127.0.0.1 -d 0/0  -j DROP
iptables -A FORWARD -i eth1 -p tcp -s 10.0.0.0/8 -d 0/0  -j DROP
iptables -A FORWARD -i eth1 -p tcp -s 192.168.0.0/16 -d 0/0  -j DROP
```

The above ruleset could be used to set up basic ingress or egress filtering for a site or system.

Checking for Rootkits

As I mentioned previously, attackers commonly use rootkits and trojan files when they compromise a system. Integrity checking tools like Tripwire will protect against file-level rootkits, such as lrk5 and t0rnkit. However, they are ineffective against kernel-level rootkits like knark, adore, and KIS. Since these types of rootkits sit below the files as a LKM or loadable kernel module, tools like Tripwire cannot detect them. Tools like lsof, however, can sometimes be used to identify hidden processes that are running. Since kernel-level rootkits are actually adding on to the kernel with Linux, most kernel-level rootkits can be detected by looking in /boot/System.map. This file contains a mapping of all kernel symbols each time the kernel is compiled or modified. The following shows part of the output of looking at the System.map:

```
c0100000 A _text
c0100000 T _stext
c0100000 T stext
c0100000 t startup_32
c0100139 t is486
c0100148 t is386
c0100191 t L6
c0100193 t check_x87
c01001ba t setup_idt
c01001d7 t rp_sidt
c01001e4 T stack_start
c01001ec t int_msg
c0100200 t ignore_int
c0100222 t idt_descr
c0100224 T idt
c010022a t gdt_descr
```

Manual examination of this file is not practical; it has thousand of entries and very few people would be able to detect something unusual just by looking at it. If you have a clean copy of your System.map, you can compare it to a newer version and any differences could indicate an infection, however, there is still a chance of false positives. A more automated tool for checking for rootkits is chrootkit, which is available from:

```
http://www.chrootkit.org
```

This program is very robust and does a good job of detecting kernel-level rootkits. The following shows part of the output of running chrootkit on a system. In this case, the system has been infected with adore:

```
Searching for LOC rootkit ... nothing found
Searching for Romanian rootkit ... nothing found
Searching for anomalies in shell history files... nothing found
Checking 'asp'... not infected
Checking 'bindshell'... not infected
Checking 'lkm'... Warning: Adore LKM installed
SIGINVISIBLE Adore found
Warning: Possible LKM Trojan installed
```

Chrootkit will detect most of the common kernel-level rootkits available today, but the fact that it does not detect a rootkit does not mean that your system is not infected.

Centralized Logging

When an incident occurs, it is critical to contain the problem and figure out what is happening as quickly as possible by analyzing the events that occurred on that system. The most basic way to look at an event is by an entry in a log file, thus you should keep log files for several months or longer. Even if you do not have the resources to review the logs on a regular basis, you should still store them, because you never know when you will need them.

Being able to validate events from multiple sources is critical, especially with log files. If you only keep log files on the local machine and that machine becomes compromised, then the integrity of your log files is lost. One of the first things an attacker will try to do is delete your log files. Always store your logs on the local machine and send them to a remote syslog server. This provides an automated method to check for possible intruders. If you send log files to both the local system and remote system, the logs should be the same. You should have a program that compares the two files on a regular basis and alerts you if it detects differences. The configuration for syslog is normally stored in syslog.conf, and since attackers typically use pre-compiled scripts to break into systems, they will check there to see whether remote logging is enabled. Here is part of a default syslog.conf file:

```
# Log all kernel messages to the console.
kern.*                             /dev/console

# Log anything (except mail) of level info or higher.
# Don't log private authentication messages!
*.info;mail.none;authpriv.none     /var/log/messages

# The authpriv file has restricted access.
authpriv.*                         /var/log/secure

# Everybody gets emergency messages
*.emerg
```

If the attacker looked at the standard syslog.conf file in this case, he would see that the system is only performing local logging. I keep a copy of syslog.conf with only local logging entries. Then I keep my real syslog configuration file in another location and recompile my syslog daemon to point to that location. The real configuration file is shown here:

```
kern.*        /var/log/kern
kern.*        @loghost

auth.*        /var/log/auth
auth.*        @loghost

mail.*        /var/log/mail
mail.*        @loghost
```

In this case, every entry is being sent to both the local files and a remote system, but the novice attacker might think they were only being saved locally.

Conclusion

The steps outlined in this paper will by no means make your systems completely secure, but they are good first steps toward securing your organization. Security is about protecting access to proprietary information. Therefore you must determine where that information is located and then take steps to protect those servers. Ideally, systems administrators must implement comprehensive security policies covering all aspects of the network, but this article is meant simply to describe a few basic steps you can take today to protect your information.

Eric Cole is a speaker at national and international conferences on network security, including SANS. An information security expert for more than 10 years, Eric holds several professional certifications. Eric is currently chief scientist for The Sytex Group's Information Warfare Center, where he heads cutting edge research in technology and various areas of network security. He is the author of Hackers Beware, Hiding in Plain Site, *co-author of* SANS GIAC: Security Essentials Toolkit *and co-author of* Security Essentials.