Integrity Management:
Protecting Against and Detecting Change

License

Contents

[What to show your tutor for XP rewards]

Preparation

These tasks can be completed on the LinuxZ IMS image.

This lab could be completed on most RPM-based Linux systems with these tools installed: rsync, md5sum, md5deep, Perl ssh server (if working together).

Some of these exercises can be completed with a classmate, and assumes root access is available to each others systems via an ssh server.

Integrity

Security is often described in terms of confidentiality, integrity, and availability. Protecting the integrity of information involves preventing and detecting unauthorised changes. In many commercial organisations integrity of information is the highest priority security goal. Managing who is authorised to make changes to databases or files, and monitoring the integrity of resources for unauthorised changes is an important task in managing information security.

Protecting integrity

Protecting the integrity of resources, such as the files on a system, involves successfully managing a variety of security mechanisms, such as authentication, access controls and file permissions, firewalls, and so on.

> On Linux systems this can include managing passwords, packet filtering IPTables rules, standard Unix file permissions (rwx), Linux extended attributes (including ACLs for detailed authentication, labels for mandatory access control (MAC), and Linux Capabilities). Linux (like other Unix-like and Unix-based systems) has a long history of adding new security features as and when they are required.
>
> Note that many security controls such as those listed above are very important for protecting the integrity of files, but are beyond the scope of this lab. Here the focus is on techniques that are *entirely* focussed on integrity rather than confidentiality or availability.

There are precautions that can be taken to reduce the chances of unauthorised changes.

Protecting integrity with file attributes

Unix systems (such as Linux or FreeBSD) include file attributes that, amongst other features, can make files immutable or append only. Setting these file attributes can provide an effective layer of security, and yet could be considered one of the more obscure Unix security features. Note that this feature is dependent on the use of a compatible filesystem (most Unix filesystems, such as ext, are compatible with file attributes). Once configured, file attributes can even prevent root (the all-powerful Unix superuser) from making changes to certain files.

Open a terminal console (such as Konsole from KDEMenu → System → Terminal → Konsole).

Start by creating a file with some content. Run:

```
sudo bash -c 'cat > /var/log/mylogfile'
```

(Type some content, then press Ctrl-D to finish and return to the prompt.)

Look at the details of the file:

```
ls -la /var/log/mylogfile
```

```
-rw-r--r-- 1 root root 20 Feb  7 17:38 /var/log/mylogfile
```

As we can see above, the file is owned by root, who has read-write access – exploring Unix file permissions further is outside the scope of this lab, but will be covered elsewhere.

Run:

```
lsattr /var/log/mylogfile
```

```
-------------e- /var/log/mylogfile
```

The ‘e’ flag is common on ext filesystems, may or may not be present when you run the above, and does not really concern us. From a security perspective the ‘a’ and ‘i’ flags are the most interesting. Read the man page for chattr to find out more about these flags and what they do:

```
man chattr
```

(Press Q to leave the manual page)

Set the ‘i’ flag using the chattr command:

```
sudo chattr +i /var/log/mylogfile
```

Now try to delete the file and see what happens:

```
sudo rm /var/log/mylogfile
```

Denied! Opps, that's right, root owns the file (since you created it with sudo)! Use root to try to delete the file:

```
sudo rm /var/log/mylogfile
```

It still didn't work! That's right, even root can't delete the file, without changing the file's attributes back first.

Use some commands to remove the 'i' flag (hint: "-i", instead of "+i").

Now run a command to set the 'a' flag on /var/log/mylogfile.

If you have done so correctly, attempting to overwrite the file with a test message should fail:

```
sudo bash -c 'echo "test message" > /var/log/mylogfile'
```

This should produce an error, since '>' causes the output of the program to be written to the specified log file, which is not allowed due to the chattr command you have run.

Yet you should be able to append messages to the end of the file:

```
sudo bash -c 'echo "your-name: test message" >> /var/log/
mylogfile'
```

This should succeed, since '>>' causes the output of the program to be appended (added to the end of) to the specified log file, which is allowed. Use your name above, for example "echo "Cliffe: test message" >> /var/log/ mylogfile".

View your changes at the end of the file:

```
tail /var/log/mylogfile
```

This has obvious security benefits, this feature can be used to allow files to be written to without altering existing content. For example, for ensuring that log files can be written to, but avoiding giving everyone who can write to the file the ability to alter its contents.

Take a screenshot of the three commands showing a denied write, accepted append, and tail of your log file containing your name, as evidence that you have completed this part of the task.

Label it or save it as "Integrity-1".

Protecting integrity with read-only filesystems

On Unix, a filesystem is mounted to a particular point in the directory structure; for example, a USB thumb drive may be mounted to /media/myUSB/. Some filesystems will automatically mount read-only; for example, if you insert a CD-ROM, since those disks are physically read-only. It is possible to optionally mount almost any filesystem, such as a USB, in read-only mode, which will make it practically impossible to write changes to it (without remounting or accessing the drive in other ways, which normally only root can do).

Assuming you have one available (if you don't, just read through this section), insert a USB thumb drive.

Browse to the contents of your USB in a graphical program such as Dophin (the KDE file explorer). This ensures that the operating system has automatically mounted the drive.

In a command prompt, run:

```
mount
```

This will list all of the currently mounted filesystems, and should include your USB drive towards the bottom of the list.

The line that we are interested in will look something like this:

```
/dev/sdc1 on /media/CLIFFE type vfat
(rw,nosuid,nodev,relatime,uid=1000,gid=100,fmask=
0022,dmask=0077,codepage=cp437,iocharset=iso8859-
1,shortname=mixed,showexec,utf8,flush,errors=remount-
ro,uhelper=udisks)
```

Note that in this case the device "/dev/sdc1" is mounted to "/media/CLIFFE" and is mounted read-write (**rw**), meaning we can access the drive at /media/CLIFFE and we

can change the contents of it. For security reasons, it can be safer to mount things as read-only, when we don't need to be able to make changes to the contents.

Make a note of the output from when you run mount (similar to the example above), and close all programs displaying the contents of your USB, such as Dolphin.

Unmount your USB drive (similar to what happens when you safely remove a USB drive):

```
sudo umount /media/CLIFFE
```

(where /media/CLIFFE is what you noted earlier, where your own drive mounts to)

Now mount your drive read-only:

```
sudo mkdir /mnt/read-only-usb
```

```
sudo mount /dev/sdc1 -t auto /mnt/read-only-usb -o ro
```

(where /dev/sdc1 is what you noted earlier, your drives device file)

Now open Dolphin (or other graphical file explorer) and browse to /mnt/read-only-usb (if you are using Dolphin, start by first clicking "Root", or type the location into the address bar).

You should be able to open files that are there. Try to create a new file or make a change to an existing file. Your attempts should fail, since the filesystem has been mounted read only!

When you are finished, close the programs displaying the USB drive, and unmount:

```
sudo umount /mnt/read-only-usb
```

Mounting read-only can be an effective way of protecting resources that you don't need to make any changes to. Read-only mounting is particularly effective when the actual disk resides externally, and can be enforced remotely. For example, when sharing files over the network.

Note that mounting read-only may be circumvented by root (or a user with enough privilege) via direct access to the device files (/dev/sdc1 in the example above), or by re-mounting as read-write.

Aside: in new versions of Linux, it is also possible to have a directory (one part of what is on a disk) present in the directory structure twice with different mount options

(for example, /home/cliffe and /home/cliffe-read-only). This can be achieved by bind mounting, and then remounting to set the bind mount to read only. More information: http://lwn.net/Articles/281157/

Detecting changes to resources

Although we can aim to protect integrity, eventually even the strongest defenses can fail, and when they do we want to know about it! In order to respond to a security incident we need to detect that one has occurred. One way we do so, is to detect changes to files on our system.

Detecting changes to resources using backups

One technique is to compare files to a backup known to represent the the system or resources in a clean state. One advantage of this approach is that we can not only detect that files have changed, but also see exactly how they differ.

You can (and, if possible, should) **conduct this exercise with a classmate**.

Make a backup of your /etc/passwd file:

```
cp /etc/passwd /tmp/passwd_backup
```

This file (/etc/passwd) is an important file on Unix systems, which lists the user accounts on the system. Although historically the hashes of passwords were once stored here, they are now typically stored in /etc/shadow. Changes to the /etc/passwd file are usually infrequent – such as when new user accounts are created – and changes should only be made for authorised purposes.

At this point, also make sure you have a backup of any work.

If you are not working with someone else, then skip the following step.

If you are working with a classmate, once you (and your classmate) have saved a backup copy of your own passwd file using the above command, then connect to your classmate's computer using ssh:

> Find your IP address using ifconfig, and tell your classmate.
>
> ```
> ssh root@their-ip-address
> ```
>
> (Where *their-ip-address* is as noted earlier.) If you do not know each other's root password, then feel free to log each other in on an ssh session as root.

Now that you have root access to their system, add a new user to their computer... Your aim is to make the new account hard to notice. If you are working alone, just do this on your own system:

```
useradd new-username
```

Where new-username, is some new name. Don't tell your classmate the name of the account you have created. You may want to create a username that looks innocent.

To make things even more interesting, edit the /etc/passwd file and move the new user account line somewhere other than right at the bottom, so that it is less obvious:

```
vi /etc/passwd
```

Move the cursor onto the line representing your new account (probably at the bottom).

In vi type:

```
:m -number
```

Where number is the number of lines to move up, for example: ":m -20" will move the currently selected line up 20 lines, "hiding" the new user account amongst the others.

Save your changes and exit vi by typing:

```
wq
```

Now exit ssh:

```
exit
```

If you are working together, on your own computer, look at the changes your classmate made, and try to spot the new user account:

```
less /etc/passwd
```

(Q to exit)

Its not as easy as it sounds, especially if your system has lots of user accounts, like the LinuxZ system does.

Since you have a backup of your passwd file, you can compare the backup with the current passwd file to determine it has been modified. One such tool for determining changes is diff. Diff is a standard Unix command. Run:

```
diff -q /tmp/passwd_backup /etc/passwd
```

Diff should report that the two files differ. Diff can also produce an easy to read description of exactly how the file has changed. This is a popular format used by programmers for sharing changes to source code:

```
diff -u /tmp/passwd_backup /etc/passwd
```

There are many advantages to the comparison to backups approach to detecting changes, but it also has its limitations. To apply this approach to an entire system, you will need a fairly large amount of either local or network shared storage, and writes need to be controlled to protect the backups, yet written to whenever authorised changes are made to keep the backup up-to-date. Also, when the comparisons are made substantial disk/network access is involved, since both both sources need to be read at the same time in order to do the comparison.

In the example above, the backup was stored on the same computer. Did you think of editing your classmates backup passwd file? This is related to a major issue when checking for changes to the system: if your system has been compromised, then you can't necessarily trust any of the local software or files, since they may have been replaced or modified by an attacker. For that reason, it can be safer to run software (such as diff) from a separate read-only storage. Yet that still may not be enough, the entire operating system could be infected by a rootkit.

Aside: Filesystems, such as btrfs, that support history and snapshots can also be helpful for investigating breaches in integrity.

Take a screenshot of the previous output from diff, identifying the new user account that was created, as evidence that you have completed this part of the task.

Label it or save it as "Integrity-2".

Detecting changes to resources using hashes and file integrity checkers

Another technique for detecting modifications to files is to use hashes of files in their known good state. Rather than storing and comparing complete copies, a one way hash function can be used to produce a fixed length hash (or "digest"), which can be used for later comparisons. Hashes have security properties that enable this use:

- Each hash is unique to the input
- It is extremely difficult (practically impossible) to find another input that produces the same hash output
- Any change to the input (no matter how minor) changes the output hash dramatically

We can store a hash and later recompute the hash, to determine whether the file has changed (if the hash is different), or it is exactly the same (if the hash is the same). If you have studied digital forensics, many of these concepts will be familiar to you, since hashes are also commonly used for verifying the integrity of digital evidence.

Generate an MD5 hash of your backup password file, which you copied above:

```
md5sum /tmp/passwd_backup
```

Now calculate a hash of your current passwd file:

```
md5sum /etc/passwd
```

If the generated hashes are different, you know the files do not have exactly the same content.

Note that using hashes, there is no need to have the backup on-hand in order to check the integrity of files, you can just compare a newly generated hash to a previous one.

Repeat the above two commands using shasum rather than md5sum. SHA1 and SHA2 are considered to be more secure than the "cryptographically broken" MD5 algorithm. Although MD5 is still in use today, it is safer to use a stronger hash algorithm, since MD5 is not collision-resistant, meaning it is possible to find multiple files that result in the same hash. SHA1 is considered partially broken, so a new algorithm such as SHA2 is currently a good option. There are a number of related commands for generating hashes, named md5sum, shasum, sha224sum, sha256sum, and so on. These commands (as well as those in the next section) are readily available on most Unix systems, and are also available for Windows.

File integrity checkers

A file integrity checker is a program that compares files to previously generated hashes. A number of these kinds of tools exist, and these can be considered a form of host-based intrusion detection system (HIDS), particularly if the checking happens automatically. One of the most well known integrity checkers is Tripwire, which was previously released open source; although, new versions are closed source and maintained by Tripwire, Inc, with a more holistic enterprise ICT change management focus. There are other tools similar to Tripwire, such as AIDE (Advanced Intrusion Detection Environment), and OSSEC (Open Source Host-based Intrusion Detection System).

The above md5sum, shasum (and so on) programs can also be used to check a list of file hashes.

Create an empty file, where *your-name*, is your actual name:

```
touch your-name
```

Run the following to generate a file containing hashes of files we can later check against:

```
shasum your-name >> /tmp/hash.sha

shasum /etc/passwd >> /tmp/hash.sha

sudo shasum /etc/shadow >> /tmp/hash.sha

shasum /bin/bash >> /tmp/hash.sha

shasum /bin/ls >> /tmp/hash.sha
```

Look at the contents of our new hashes file:

```
less /tmp/hash.sha
```

Now use your new hash list to check that nothing has changed since we generated the hashes:

```
shasum -c /tmp/hash.sha
```

Make a change to our empty "*your-name*" file:

```
echo "hello" > your-name
```

Check whether anything has changed since we generated hashes:

```
shasum -c /tmp/hash.sha
```

You should see a nice explanation of the files that have changed since generating the hashes.

Take a screenshot of the previous output from shasum, identifying that a file named after you has changed, as evidence that you have completed this part of the task.

Label it or save it as “Integrity-3”.

Scripted integrity checking

The above can also be accomplished via a simple script (in this case a Perl script):

```
#!/usr/bin/perl
# Copyleft 2012, Z. Cliffe Schreuders
# Licenced under the terms of the GPLv3
use warnings;
use strict;

my %files_hashes = (
        "/bin/ls"=>"9304c5cba4e2a7dc25c2d56a6da6522e929eb848",
        "/bin/bash"=>"54d0d9610e49a654843497c19f6211b3ae41b7c0",
        "/etc/passwd"=>"69773dcef97bca8f689c5bc00e9335f7dd3d9e08"
);

foreach my $file_entry (keys %files_hashes) {
        my $hash = `sha1sum $file_entry|awk '{print \$1}'|head -n1`;
        chomp($hash);
        if($hash ne $files_hashes{$file_entry}){
                        warn "FILE CHANGED: $file_entry (hash was $hash, expected
$files_hashes{$file_entry})\n";
        } else {
                        print "File unmodified: $file_entry (hash was $hash, as expected)\n";
        }
}
```

This script simply iterates over a list of file paths with SHA1 hashes (stored in an associative array), and runs sha1sum for each one to check whether the files are still the same.

Save the script as checker.pl, and run with:

```
perl checker.pl
```

Are the files reported as unmodified, or have they changed? Why might they be different to when I wrote the script?

Recursive file integrity checkers

The md5deep program (also known as sha1deep, sha256deep, and so on for different hash algorithms) can recursively walk through directories (and into all contained subdirectories) to generate and check lists of hashes.

Run:

```
sudo sha1deep -r /etc
```

If the md5deep command is not available, install it:

On openSUSE this can be done by first running "cnf sha1deep", to find the name of the package containing the program, then run the install command it gives you, such as "sudo zypper install md5deep".

If you get "PackageKit is blocking zypper", then select "no", and kill PackageKit, by running "kill -9 *pid*", where *pid* is the number reported by the previous command. Now run the above again.

If the zypper command is stuck on refreshing a repository, then press "Ctrl-C", "a" (for abort), then proceed with the installation as per normal.

Once the required software is installed, try the sha1deep command again.

The output of the above command will include hashes of every file in /etc, which is where system-wide configuration files are stored on Unix.

Read the sha1deep manual to understand the above command:

```
man sha1deep
```

Figure out what the -r flag does.

(Q to quit)

We can save (redirect) this output to a file so that we have a record of the current state of our system's configuration:

```
sudo sha1deep -r /etc > /tmp/etc_hashes
```

This may take a minute or so, while the program calculates the hashes and sends them to standard out (known as stdout), which is then redirected to the etc_hashes file.

Next, lets compare the size of our list of hashes, with the actual content that we have hashed...

See how big our list of hashes is:

```
ls -hs /tmp/etc_hashes
```

(-h = human readable, -s = size)

This is likely in the Kilobytes.

And for the size of all of the files in /etc:

```
sudo du -hs /etc
```

(-h = human readable, -s summarise)

This is likely in the Megabytes (or maybe even Gigabytes).

Clearly, the list of hashes is much smaller.

If you are **working with a classmate**, log into their system using ssh (as done previously). If you are working alone, simply run all the commands on your own system.

Create a new file somewhere in /etc/, containing your name. Name the file whatever you like (for example /etc/test), although the more inconspicuous the better.

> Hint: “vi /etc/test”, “i” to enter insert mode, and after typing your name, “Esc”, “:wq”.

Also, change an existing file in /etc on their system, but please do be careful to only make a minor change that will **not cause damage to their system**. For example, you could use vi to edit /etc/rsyslog.conf (“sudo vi /etc/rsyslog.conf”), and add a comment to the file such as “#your-name: bet you can’t find this comment!”

You can now “exit” the ssh session.

On your own system, lets try to identify what the “attacker” has done to our system...

Now that we have a list of hashes of our files, use shasum to check if anything has changed using our newly generated list of hashes (/tmp/etc_hashes).

Hint: look at the previous command using shasum to check hashes.

Does this detect our the changed file AND the new file? Why not?

Md5deep/sha1deep takes a different approach to checking integrity, by checking all of the files it is told to check (possibly recursing over all files in a directory) against a list of hashes, and reporting whether any files it checked did not (or did, depending on the flags used) have its hash somewhere in the hash list.

Run sha1deep to check whether any files in /etc/ do not match a hash previously generated:

```
sudo sha1deep -X /tmp/etc_hashes -r /etc
```

This should detect both modified files, both new and modified.

Take a screenshot of the previous output from sha1deep, identifying the two files (new and modified) that have changed, as evidence that you have completed this part of the task.

Label it or save it as "Integrity-4".

But would sha1deep detect a copy of an existing file, to a new location?

Try it:

```
sudo cp /etc/passwd /etc/passwd.backup
```

Now rerun the previous sha1deep command. Was the copy detected? Why not?

What about copying one file over another? Which out of shasum or sha1deep would detect that?

Another tool, hashdeep, which is included with md5deep, provides more coverage when it comes to detecting files that have moved, changed, or created.

Generate a hash list for /etc using hashdeep:

```
sudo hashdeep -r /etc > /tmp/etc_hashdeep_hashes
```

Hashdeep stores hashes in a different format than the previous tools. Have a look:

```
less /tmp/etc_hashdeep_hashes
```

(Q to quit)

Note that the output includes some more information, such as the file size for each file.

Delete the new file that your “attacker” (the person who sshed into your system) created earlier:

```
sudo rm /etc/whatever-the-filename-was
```

Conduct a hashdeep audit to detect any changes:

```
sudo hashdeep -r -a -k /tmp/etc_hashdeep_hashes /etc
```

Note, that this can take a while, so feel free to start working through the next section in another terminal, if you like.

After, run it again, this time asking for more details, since the default message does not provide any information as to why an audit has failed:

```
sudo hashdeep -ravv -k /tmp/etc_hashdeep_hashes /etc
```

Consult the man page for information about what each of the above flags do.

Detecting changes to resources using package management

On Linux systems, package management systems are used to organise, install, and update software. The package management system has a database that keeps track of all the files for each program or software package. Depending on the package management system used, the database may maintain hashes in order to detect changes to files since install. RPM-based systems (such as Red Hat, Fedora, and OpenSUSE), store hashes of each file that is included in software packages. There are commands that can be used to detect changes to files that have occurred since being installed by the package management software.

Note that there are times where it is perfectly normal for a number of files to not match the “fresh” versions that were installed: for example, configuring a system for use will involve editing configuration files that were distributed with software packages.

The “rpm” command has a -V flag for verifying the integrity of packages.

Choose any system file on the computer, such as /etc/securetty. To determine which package the file belongs to:

```
rpm -q --whatprovides any-file-you-chose
```

Where any-file-you-chose is any file such as /etc/securetty.

The output of that command the package-name, and is required in the next step.

Check the integrity of the file:

```
rpm -V package-name
```

Where package-name is the output from the previous command.

An example from the output would be:

```
5S.T.....  c /etc/securetty
```

Which means, it is a config file (c), and:

- S – file Size differs
- M – Mode differs (includes permissions and file type)
- 5 – MD5 sum differs
- D – Device major/minor number mismatch
- L – readLink(2) path mismatch
- U – User ownership differs
- G – Group ownership differs
- T – mTime differs
- P – caPabilities differ

Use the above information to understand the output from your above rpm -V command.

Next verify the integrity of all of the packages on the entire system (this may take a while):

```
rpm -Va
```

Try to understand the cause of any files failing the integrity checks.

Consider the limitations of this approach: what files will (and won't) this approach to integrity management cover?

Choosing files for integrity checking

Fill in the table below with a number of files on a Unix or Windows system whose integrity should be monitored for security purposes:

Filename	**Description**	**Security reason for monitoring integrity**
Unix: /etc/passwd	*List of user accounts*	*Adding or modifying users is security sensitive, since rouge user accounts could present a backdoor to the system*

Complete the table above, with a number of Unix/Windows files that should be monitored for integrity, as evidence that you have completed this part of the task.

Label it or save it as "Integrity-5".

Limitations of integrity checking

Perhaps the greatest limitation to all of these approaches, is that if a system is compromised, you may not be able to trust any of the tools on the system, or even the operating system itself to behave as expected. In the case of a security compromise, your configuration files may have been altered, including any hashes you have stored locally, and tools may have been replaced by Trojan horses. For this reason it is safer to run tools over the network or from a removable drive, with read-only access to protect your backups and hashes. Even then, the OS/kernel/shell may not be telling you the truth about what is happening, since a rootkit could be concealing the truth from other programs.

Problem-based tasks

Add an integrity monitoring solution (md5sum, md5deep, or hashdeep) to a cron job, so that every hour the integrity of some important files are checked, and any errors are emailed to root.

Hints: Any output on standard error (stderr) on a cron job results in a local email to root. As root, type "mail" to read the local emails. Run "crontab -e" to add scheduled tasks. Google will certainly help here.

Take screenshots of an hourly cronjob rule, and email with an integrity report from md5sum/deep or hashdeep, as evidence that you have completed this part of the task.

Label it or save it as "Integrity-A1".

Add to your above solution, by considering and implementing some protection against modifications to your hash file/database.

Take a screenshot of your configuration for protection of the hash file (and include a one sentence description), as evidence that you have completed this part of the task.

Label it or save it as "Integrity-A2".

Install either OSSEC (Open Source Host-based Intrusion Detection System), AIDE (Advanced Intrusion Detection Environment), or Tripwire (if you can find a copy), and use it to monitor the integrity of your files. Modify a file named *your-name* (your actual name) and view a report or alert that the integrity of the file has been compromised.

Take a screenshot of a report from OSSEC or AIDE that a file named after you has been altered, as evidence that you have completed this part of the task.

Label it or save it as "Integrity-A3".

Add to the integrity monitoring script given earlier, to store and retrieve the hashes from a file.

For extra marks, protect the hashes using a HMAC, with user interaction to enter a password.

Save your modifications of the script, as evidence that you have completed this part of the task.

Label it or save it as "Integrity-A4".

Resources

An excellent resource on the subject of integrity management is Chapter 20 of the excellent book *Practical Unix & Internet Security, 3rd Ed*, by Garfinkel et al (2003).

What to show your tutor for XP rewards

Show your tutor each of the above (in red) evidences. You may be asked to justify your decisions. This will be used to allocate XP for the module. Further details of the XP rewards and requirements are available on the *My XP* site.