Chapter 4
Authentication and Access Control

Access control is the means by which access to system resources is restricted to authorized subjects. Access control has a wide scope and can be found in hardware and software, at all levels of the software stack. This includes memory management, operating systems, middleware application servers, databases and applications.

In this chapter we study access control, focussing on access to remote computers, as well as access to files and other resources stored on computers.

4.1 Objectives

After this chapter you will know different mechanisms for remote system access and the threats they incur. You will learn how to use Secure Shell and you will be able to configure it according to your needs.

You will learn the concept of file system permissions in a Linux-based environment and how to apply this. You will also learn to autonomously configure access restrictions at the level of operating systems. Finally you will be able to use this knowledge on your own personal computers.

4.2 Authentication

Authentication denotes the process by which the identity of a user (or any subject) is verified. In this process, the user provides his claimed identity together with evidence in the form of credentials. If the authenticating system accepts the evidence, the user is successfully authenticated. Authentication is usually a prerequisite for authorization to use system resources, i.e., one is authorized to use resources based on their identity.

The most common authentication mechanisms are based on user names and passwords. The system verifies the credentials presented by the subject against information stored in a database. There are numerous alternatives for authentication, such as one-time passwords (e.g., TAN lists) or certificates (e.g., in SSH), etc. There are also various options for storing authentication information. Examples are local files, such as `/etc/passwd` under Linux, or a central directory service, such as LDAP, RADIUS or Windows Active Directory.

4.2.1 Telnet and Remote Shell

The network protocol Telnet offers bidirectional communication and is used to remotely administer computer systems. Since Telnet does not use any encryption mechanisms, the entire session including the exchange of authentication credentials can easily be intercepted.

On **`mallet`** start the password sniffer dsniff. Using the option `-m` enables automatic detection of protocols and `-i` defines the interface the sniffer should listen to.

```
mallet@mallet:$ sudo dsniff -m -i eth0
dsniff: listening on eth0
```

On **`alice`** open a Telnet session to **`bob`**.

```
alice@alice:$ telnet bob
Trying 192.168.1.1...
Connected to bob.
Escape character is '^]'.
Debian GNU/Linux 5.0
bob login: bob
Password: ***

Last login: Mon Jul 26 14:35:56 CEST 2011 from alice on pts/0
Linux bob 2.6.26-2-686 #1 SMP Mon Jun 9 05:58:44 UTC 2010 i686
...
You have mail.
bob@bob:$ exit
```

The program dsniff displays the credentials sent by **`alice`** while connecting to **`bob`**.

```
-----------------
07/26/11 14:41:57 tcp alice.54943 -> bob.23 (telnet)
bob
bob
exit
```

In contrast to Telnet, the program *Remote Shell* (rsh) provides an option to log in without a password. In a `.rhosts` file the IP addresses are given for which access is granted without the system asking for a password. This circumvents the risk incurred by entering a password, which could be intercepted by an adversary. Authentication is only based on the user name and the corresponding IP address.

Since the rsh client chooses a well-known port number (a privileged port) as the source port, it must have the setuid bit set and must be owned by `root`.

Rsh also suffers from security vulnerabilities. Any password transmitted is sent in the clear. Moreover, its IP address-based authentication can be exploited by IP address spoofing where the adversary fabricates IP datagrams containing a fake source IP address. Finally, after login all subsequent information is also sent in the clear and is not authenticated.

Problem 4.1 Is it possible to increase security by using an authentication mechanism based on the MAC (Ethernet) address instead of the IP address? What is a fundamental argument against authentication based on MAC addresses in most network settings?

4.2.2 Secure Shell

The program *Secure Shell* (SSH) is a successor of rsh and is designed for use in untrusted networks like the Internet. It solves many of the security-related problems of rsh and Telnet. Since SSH encrypts all communication, it provides secure connections that are resistant against interception and it offers protection against message manipulation as well. Furthermore, SSH provides a secure alternative to rsh's user authentication by using public key cryptography to authenticate users. In the following we will use *OpenSSH*, a free version of SSH on Linux systems.

▷ Stop the SSH service (sshd) running on **`bob`** and start it again in debug mode, listening on the standard port 22 to observe the individual steps taken during authentication. Note that in debug mode only one connection can be established and sshd quits immediately after the connection is closed. By default, the debug information is written to the standard error output. In order to analyze the specific steps, we redirect the error output to the standard output first and then pipe the standard output to the command `less`.

```
bob:~# /etc/init.d/ssh stop
bob:~# /usr/sbin/sshd -d 2>&1 | less
```

▷ Now connect from **`alice`** to **`bob`** using SSH. You can use tcpdump or Wireshark on **`mallet`** to convince yourself that no plaintext information is sent.

```
alice@alice:$ ssh -v bob@bob
```

▷ Log in to **bob** with user *bob*'s password and close the connection afterwards.

```
bob@bob:$ exit
logout
```

Now analyze the output on **bob**, **alice** and **mallet** and answer the following question.

Problem 4.2 What are the individual steps involved in establishing an SSH connection?

Most users apply SSH as just described, using a user name and password. But SSH also offers the option to remotely log in without entering a password. Instead of the IP address-based authentication of rsh, public key cryptography may be used.

When carrying out the following experiments, it may help to also read the corresponding manual pages of `sshd`, `ssh` and `ssh-keygen`.

▷ First, a key pair consisting of a public and a private key must be created. In order to preserve the comfort of rsh, we abandon the option of using a passphrase (`-N ""`), i.e., the private key is stored unencrypted.

```
alice@alice:$ ssh-keygen -f alice-key -t rsa -N ""
Generating public/private rsa key pair.
Your identification has been saved in alice-key.
Your public key has been saved in alice-key.pub.
The key fingerprint is:
d5:e3:9d:82:89:df:a6:fa:9e:07:46:6e:37:c8:da:4f alice@alice
The key's randomart image is:
+--[ RSA 2048]----+
|                 |
|           .     |
|          . o    |
|         o.+ o . |
|        S+o.o o  |
|         .B.o.   |
|         =.oE.   |
|        . .=.    |
|        .+=o.    |
+-----------------+
```

Note that the randomart image provides a visual representation of the generated key's fingerprint and allows easy validation of keys. Instead of comparing string representations of fingerprints, images can be compared.

▷ By entering the public key into the file `authorized_keys` of another host, the owner of the corresponding private key can log in without entering a password. Before you start, make sure that the hidden directory `/home/bob/.ssh` exists.

```
bob@bob:$ ssh alice@alice cat alice-key.pub >> \
> ~/.ssh/authorized_keys
alice@alice's password: *****
```

To prevent other system users from entering arbitrary other public keys into the `authorized_keys` file, you must appropriately set the file access permissions.

```
bob@bob:$ chmod og-rwx ~/.ssh/authorized_keys
```

Now it is possible to log in to **`bob`** (as user *bob*) using the private key `alice-key` without being asked for an additional password.

```
alice@alice:$ ssh -i alice-key bob@bob
```

In contrast to rsh, the user ID on the client machine does not play a role in this method. The authentication is based on the knowledge of the private key.

Problem 4.3 With regard to the steps you gave in your answer to Problem 4.2, what are the differences between authentication based on public keys and password-based authentication?

Problem 4.4 How is **`bob`**'s SSH daemon able to verify that Alice possesses the correct private key without sending the key over the network?

SSH has numerous options. For example, you can restrict clients to executing predefined commands only, and you can deactivate unnecessary SSH features such as port forwarding or agent forwarding. As an example, the following restricts Alice to executing the command `ls -al` on **`bob`**.

```
bob@bob:$ echo from=\"192.168.1.2\",command=\"ls -al\",\
> no-port-forwarding `ssh alice@alice cat alice-key.pub` > \
> ~/.ssh/authorized_keys
```

Alice can now execute this command on **`bob`** without giving a password, but only this command:

```
alice@alice:$ ssh -i alice-key bob@bob
```

This technique can be used in a cron job to copy files from one system to another or to restart a process on a remote machine. For these kinds of tasks, the `no-pty` option is recommended, which prevents allocating a pseudo terminal. The execution of interactive full-screen programs is also prevented (e.g., editors like vi or Emacs).

Problem 4.5 What are the security-relevant advantages of authentication using public key cryptography compared to the IP address-based authentication of rsh? What is the main threat that still remains?

In order to improve the security of a system, private keys should only be stored in encrypted form. However, it can be annoying to re-enter the passphrase repeatedly. For such situations, OpenSSH provides a program named *ssh-agent*. This program can be seen as a kind of "cache" for passphrases of private keys. When using it, the passphrase must be entered only once per login session and key, i.e., when the key is added to the ssh-agent.

To add an identity, you simply use the `ssh-add` command. You can repeat the above steps and generate a new key pair (e.g., `alice-key-enc`), but this time protecting the private key with a passphrase (ignore the option `-N ""`). Every time you use the newly generated identity to access **bob**, you are requested to enter the passphrase. Try it!

▷ Now add the identity to the ssh-agent:

```
alice@alice:$ ssh-add alice-key-enc
Enter passphrase for alice-key-enc:
```

From now on, for the entire login session, you will not have to enter the password for this identity again. Try it out and afterwards log out *alice* from **alice**. Log in again and try to connect to **bob** using Alice's identity `alice-key-enc`. Now you will be asked to enter the passphrase for the key again every time you access the key.

4.3 User IDs and Permissions

4.3.1 File Access Permissions

The file system permissions in Linux are based on *access control lists* (ACL). For every file, the permissions to *read*, *write*, *execute* or any combination thereof, are individually defined for the three classes of users: *user*, *group* and *others*. We will see additional concepts later, but even this simple model suffices for most practical problems and it is both easy to understand and administrate.

▷ Read the following manual pages: `chmod`, `chown`, `chgrp`, `umask` and `chattr`.

Problem 4.6 The meaning of the basic access rights, read (r), write (w) and execute (x), is obvious for conventional files. What is their meaning for directories? Find the answers through experimentation!

In general you can find all world writable files in a given directory without symbolic links using the following command:

```
alice@alice:$ find <Directory> -perm -o=w -a ! -type l
```

Note that you may need *root* permissions to search some directories. Just prepend sudo to the above command. Also replace <Directory> with the path to the actual directory you want searched.

Problem 4.7 Use the command above to find all world writable files on **alice**. Also, find all world readable files in /etc and /var/log. Which files or directories might not be configured properly? Which permissions could be more restrictive?

When creating new objects in the file system, the default permissions are important. Although default file permissions depend on the program that creates the new object, the permissions are commonly set to 0666 for files and to 0777 for directories. The user can influence the default file permissions for newly created files by defining a corresponding *user mask* (umask) value. This value can be set with the umask command. Incorrectly set user masks may result in newly created files to be readable or even writable for other users.

Problem 4.8 How is the default file permission computed from the creating program's default permission and the user-defined umask value?

We now consider some examples to see the effect of different umask values:

```
alice@alice:$ umask 0022
alice@alice:$ touch test
alice@alice:$ ls -l test
-rw-r--r-- 1 alice alice 0 2011-07-27 22:26 test

alice@alice:$ umask 0020
alice@alice:$ touch test1
alice@alice:$ ls -l test*
-rw-r--r-- 1 alice alice 0 2011-07-27 22:26 test
-rw-r--rw- 1 alice alice 0 2011-07-27 22:27 test1

alice@alice:$ umask 0
alice@alice:$ touch test2
alice@alice:$ ls -l test*
-rw-r--r-- 1 alice alice 0 2011-07-27 22:26 test
-rw-r--rw- 1 alice alice 0 2011-07-27 22:27 test1
-rw-rw-rw- 1 alice alice 0 2011-07-27 22:28 test2
```

```
alice@alice:$ umask 0022
```

Problem 4.9 Determine the default umask value, which is set in `/etc/profile` on **`alice`**. Why is this a good default?

You have probably noticed that the newly created files are set to the owner's primary group. Note that not all Unix-like systems behave in this way. Some systems inherit the group from the parent directory.

Problem 4.10 In which file could you (as an ordinary system user) alter your own default umask? Hint: The invocation order is described in the manual page of `bash`.

On Linux systems, the permissions to create, delete or rename a file are not bound to the file's permissions, but rather to the permissions defined for the parent directory. This is because these operations do not change the file itself but only entries in directories. This is not a problem in most cases. However in a shared temporary directory, this can be a serious problem since every user is allowed to manipulate arbitrary files in the directory. The problem can be solved using the *sticky bit*, which ensures that only the owner of a file or *`root`* can rename or delete a file.

Even more fine-grained permissions and restrictions are possible using file attributes.

Problem 4.11 Read the manual page of `chattr` and determine which file attributes could be used to increase security in special situations. Provide an example where it would make sense to use each of these attributes.

Entire sections of a file system can be locked from the "public" by creating a group for the designated users (`chmod g+w,o-rw`). Note that after removing the access bit of a directory (e.g., `chmod o-x`), access to all files in this directory is prohibited, thereby overruling the attributes of these files. By just removing the read bit of a directory (e.g., `chmod o-r`), access to the files remains granted, but the content of the directory can no longer be listed.

Note that permissions are read from left to right and the first matching permission is applied:

```
alice@alice:$ echo TEST > /tmp/test
alice@alice:$ chmod u-r /tmp/test
alice@alice:$ ls -l /tmp/test
--w-r--r-- 1 alice alice 5 2011-07-27 23:25 /tmp/test
alice@alice:$ cat /tmp/test
cat: /tmp/test: Permission denied
```

As you can see, *`alice`* is no longer able to read the new file. Now change the user to *`bob`* (`su bob`) and try to read the file:

```
bob@alice:$ cat /tmp/test
TEST
```

In this way, operations can be blocked for specific groups.

Problem 4.12 Why is not even `root` allowed to execute a file if the execute bit is not set, although `root` has all permissions by definition? How is this related to the environment variable `PATH`, and why should “.” not be added to the path?

Problem 4.13 Following up on the last problem, note that some people append “.” as the last entry in the path and argue that only the first appearance of the executable file is used when typing a command. Is this variant also prone to potential security problems? If so, what are these problems?

4.3.2 Setuid and Setgid

In order to understand the set user ID and set group ID mechanisms, we take a closer look at how IDs are handled in Linux. Within the kernel, user names and group names are represented by unique non-negative numbers. These numbers are called the *user ID* (uid) and the *group ID* (gid), and they are mapped to the corresponding users and groups by the files `/etc/passwd` and `/etc/group`, respectively. By convention, the user ID `0` and the group ID `0` are associated with the superuser `root` and his user private group (UPG). User private groups is a concept where every system user is assigned to a dedicated group with the user’s name. Note that some Linux distributions do not use UPGs but instead assign new users to a system-wide default group such as `staff` or `users`.

The Linux kernel assigns to every process a set of IDs. We distinguish between the *effective*, *real* and *saved* user ID (the situation is analogous for group IDs and we omit their discussion). The real user ID coincides with the user ID of the creator of the respective process. Instead of the real user ID, the effective user ID is used to verify the permissions of the process when executing system calls such as accessing files. Usually, real and effective IDs are identical, but in some situations they differ. This is used to raise the privileges of a process temporarily. To do this it is possible to set the *setuid* bit of an executable binary using the command `chmod u+s <executableBinary>`. If the setuid bit is set, the saved user ID of the process is set to the owner of the executable binary, otherwise to the real user ID. If a user (or more specifically a user’s process) executes the program, the effective user ID of the corresponding process can be set to the saved user ID. The real user ID still refers to the user who invoked the program.

Example 4.1. The command `passwd` allows a user to change his password and therefore to modify the protected system file `/etc/passwd`. In order to allow ordinary users to access and modify this file without making it world accessible, the setuid bit of the binary `/usr/bin/passwd` is set. Since this binary is owned by *`root`*, the saved user ID of the respective process is set to `0` and the effective user ID is therefore also set to `0` whenever a user executes `passwd`. Thus, access to the `/etc/passwd` file is permitted. Since the real user ID still refers to the user, the program can determine which password in the file the user may alter.

For example, if user *bob* with user ID `17` executes the command `passwd`, the new process is first assigned the following user IDs:

real user ID:	`17`
saved user ID:	`0`
effective user ID:	`17`

The program passwd then invokes the system call `seteuid(0)` to set the effective user ID to `0`. Because the saved user ID is indeed `0`, the kernel accepts the call and sets the effective user ID accordingly:

real user ID:	`17`
saved user ID:	`0`
effective user ID:	`0`

Commands like `su`, `sudo` and many others also employ the setuid concept and allow an ordinary user to execute specific commands as *`root`* or any other user.

Setuid programs are often classified as potential security risks because they enable *privilege escalation*. If the setuid bit of a program owned by *`root`* is set, then an exploit can be used by an ordinary user to run commands as *`root`*. This might be done by exploiting a vulnerability in the implementation, such as a buffer overflow. However, when used properly, setuid programs can actually reduce security risks.

Note that for security reasons, the setuid bit of shell scripts is ignored by the kernel on most Linux systems. One reason is a race condition inherent to the way *shebang* (`#!`) is usually implemented. When a shell script is executed, the kernel opens the executable script that starts with a shebang. Next, after reading the shebang, the kernel closes the script and executes the corresponding interpreter defined after the shebang, with the path to the script added to the argument list. Now imagine that setuid scripts were allowed. Then an adversary could first create a symbolic link to an existing setuid script, execute it, and change the link right after the kernel opened the setuid script but before the interpreter opens it.

Example 4.2. Assume an existing setuid script `/bin/mySuidScript.sh` that is owned by *`root`* and starts with `#!/bin/bash`. An adversary could now proceed as follows to run his own `evilScript.sh` with *`root`* privileges:

```
mallet@alice:$ cd /tmp
mallet@alice:$ ln /bin/mySuidScript.sh temp
mallet@alice:$ nice -20 temp &
mallet@alice:$ mv evilScript.sh temp
```

The kernel interprets the third command as `nice -20 /bin/bash temp`. Since `nice -20` alters the scheduling priority to the lowest possible, the fourth command is likely to be executed before the interpreter opens `temp`. Hence, `evilScript.sh` gets executed with *`root`* permissions.

Problem 4.14 In what respect does the `passwd` command improve a system's security? How could an ordinary user be given the ability to change his own password without a setuid program?

Problem 4.15 Search for four setuid programs on **`alice`** and explain why they are setuid.

Note that instead of setuid, setgid can be used in many cases. Setgid is generally less dangerous than setuid. The reason is that groups typically have fewer permissions than owners. For example, a group cannot be assigned the permission to change access permissions. You can set a program's setgid bit using the command `chmod g+s <executableBinary>`.

Problem 4.16 Find some setgid programs on **`alice`** and explain why they are setgid. Could the setuid programs in the last problem be setgid instead of setuid?

4.4 Shell Script Security

Shell scripts offer a fast and convenient way to write simple programs and to perform repetitive tasks. However, as with programs written in other programming languages, shell scripts may contain security vulnerabilities. Therefore shell scripts must be designed, implemented and documented carefully, taking into account secure programming techniques as well as proper error handling.

We already discussed security flaws related to file system permissions. Furthermore, we pointed out that on modern Linux systems the kernel ignores the setuid and setgid bit of shell scripts by default. Therefore, some administrators run their shell scripts with *`root`* permissions to ensure that all commands used in the script have the permissions they require. In doing so, they give the script too many permissions, thereby contradicting the principle of least privilege. Whenever creating a shell script, know what permissions are really needed and restrict your script's permissions to this minimum.

In this section we introduce some common pitfalls, possible attacks and ways to prevent them. Background on shells is given in Appendix C.

4.4.1 Symbolic Links

When accessing files you should make sure that you are actually accessing the intended file rather than one it is linked to. The following example shows how an adversary may abuse symbolic links to undermine system integrity.

▷ On **`alice`**, create the shell script `mylog.sh` that creates a temporary log file and writes information into the file. Because of other operations, designated in the script by `...`, you decide to run the script as *`root`*.

```
#!/bin/bash
touch /tmp/logfile
echo "This is the log entry" > /tmp/logfile
...
rm -f /tmp/logfile
```

Since the `/tmp` directory is world writable, an adversary with ordinary user permissions can create a link to a security-critical system file and name it `logfile` before the script is executed. When the script is executed, the content of the linked system file is replaced by "This is the log entry".

▷ Take now a snapshot of virtual machine **`alice`**, since after the attack the system will no longer work properly. Afterwards symbolically link `/tmp/logfile` to the `/etc/passwd` file, which contains information on all system users and their passwords. Finally, execute the above shell script with *`root`* permissions.

```
alice@alice:$ ln -s /etc/passwd /tmp/logfile
alice@alice:$ sudo ./mylog.sh
alice@alice:$ cat /etc/passwd
This is the log entry.
alice@alice:$
```

As you can see, a simple symbolic link to the `passwd` file has led the script to overwrite a write-protected file owned by *`root`*. To prevent such attacks, a script should check for symbolic links before using a file and take appropriate actions such as exiting the script.

```
...
TEST=$(file /tmp/logfile | grep -i "symbolic link")
if [ -n "$TEST" ]; then
  exit 1
fi
...
```

Problem 4.17 The script fragment is still prone to a race condition. Explain what a race condition is and how this may affect this script's behavior when run as *root*.

4.4.2 Temporary Files

When developing shell scripts there is sometimes the need to store information temporarily in a file. In the attack in Sect. 4.4.1 the script used the command `touch` to create a new file and the adversary created a file with the same name. Hence, the attack exploits the adversary's ability to know or guess the name used to create a temporary file.

Problem 4.18 Read the manual page of `touch`. What is this command primarily used for?

Many modern Linux systems provide the command `mktemp` for safely creating temporary files. This command allows one to create a temporary file with a random name, thereby making it harder for an adversary to guess the file's name in advance.

▷ Read the manual page of `mktemp` and create some temporary files in `/tmp` using its different parameters.

Problem 4.19 Use `mktemp` to prevent the symbolic-link attack in Sect. 4.4.1. Implement what you have learned so far to create the temporary file. The script now looks as follows:

```
#!/bin/bash
LOGFILE=$(mktemp /tmp/myTempFile.XXX)
echo "This is the log entry." > $LOGFILE
...
rm -f $LOGFILE
```

Is this script now secure against the symbolic-link attack? What can the adversary still do?

4.4.3 Environment

A shell script's behavior often depends on the environment the script is executed in. Variables such as `$HOME`, `$PWD` and `$PATH` store information on the shell's current state and influence its behavior. In general you should not trust externally provided inputs, and this is also the case when users can make their own assignments to these variables. Users may maliciously choose inputs that change the behavior of your script. We saw an example of this in Problem 4.12.

Problem 4.20 Why should you prefer full paths over relative paths? Explain what an adversary could do when using relative paths.

To make your scripts robust against maliciously configured environment variables you may overwrite them at the start of the script. For example, you can define the paths for system binaries as follows:

```
#!/bin/bash
PATH=<pathToBinary1>:<pathToBinary2>
...
```

Note that the newly defined `$PATH` variable's scope is just the script where it is defined.

▷ On **`alice`**, create a shell script `path.sh` that first prints the current path and afterwards sets the path to `/bin:/usr/bin` and again prints it.

```
#!/bin/bash
echo $PATH
PATH=/bin:/usr/bin
echo $PATH
```

Now make your script executable and run it.

As expected, the script first prints the value of the globally defined path variable followed by `/bin:/usr/bin`. Now, after the script has executed convince yourself that the path variable of the current shell has not been affected by the shell script:

```
alice@alice:$ ./path.sh
/usr/local/sbin/:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
/bin:/usr/bin
alice@alice:$ echo $PATH
/usr/local/sbin/:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

4.4.4 Data Validation

As with all programs, input and output data in shell scripts must be validated. This prevents the adversary from injecting unintended input. Think carefully about what your script will consume and what you want it to produce.

▷ Create another shell script `treasure.sh` on **`alice`**. This time, the script reads a password provided by the user. If the user enters the correct password, the script prints a secret to the command line.

```
#!/bin/bash
echo "Enter the password:"
read INPUT
if [ $INPUT == "opensesame" ];then
  echo "This is the secret!"
else
  echo "Invalid Password!"
fi
```

The script is vulnerable to a classical injection attack. An adversary simply enters something like `alibaba == alibaba -o alibaba`, where `-o` represents the logical `OR` in shell scripts. The script thus evaluates `if [ alibaba == alibaba -o alibaba == "opensesame" ]`, which is true.

```
alice@alice:$ ./treasure.sh
Enter the password: alibaba == alibaba -o alibaba
This is the secret!
alice@alice:$
```

This injection vulnerability can be prevented by quoting variables (e.g., `"$INPUT"`) so that they are tested as strings.

```
...
if [ "$INPUT" == "opensesame"];then
...
```

Another good practice is to sanitize the input by removing undesired characters. In the following example, only alphanumeric characters are accepted and all other characters are removed. See the manual page of the command `tr` for more information.

```
...
INPUT=$(echo "$INPUT" | tr -cd '[:alnum:]')
...
```

4.5 Quotas

Whereas in the past partitioning was done because of hardware restrictions, the trend now is towards using a single huge root partition. However, there are still good reasons to subdivide disk space into multiple smaller partitions.

Often quotas are defined for system users in order to prevent them from filling a partition or even an entire disk. Bob's account on **`alice`** is restricted to 100 MB. You can verify this by using the following command:

```
alice@alice:$ sudo edquota -u bob
```

Problem 4.21 Read the manual page of `edquota`. What kinds of limits can you set? What is the difference between a soft limit and a hard limit?

In most settings, users are restricted in how much disk space they can write. Such a setting nevertheless allows an adversary to exhaust space in a world writable directory without debiting his block quota at all. Under some circumstances, the adversary would be able to fill an entire partition despite existing quotas. The following example shows you how this could work. For the example to work, switch the user on **`alice`** to *bob* (`su bob`).

```
bob@alice:$ dd if=/dev/zero of=/var/tmp/testfile1
dd: writing to '/var/tmp/testfile1': Disk quota exceeded
198497+0 records in
198496+0 records out
101629952 bytes (102 MB) copied, 1.73983 s, 58.4 MB/s

bob@alice:$ quota
Disk quotas for user bob (uid 1001):
Filesystem blocks quota limit grace files quota limit grace
/dev/sda1  99996 100000 100000    128    0    0

bob@alice:$ echo "This is a test text" > \
> /var/tmp/testfile2
bash: echo: write error: Disk quota exceeded

bob@alice:$ df -k /var
Filesystem        1K-blocks   Used Available Use% Mounted on
/dev/sda1           9833300   2648504   6685292  29% /

bob@alice:$ count=1
bob@alice:$ while true; do
bob@alice:$ ln /var/tmp/testfile1 \
> /var/tmp/XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX.$count
bob@alice:$ ((count=count + 1))
bob@alice:$ done
```

▷ After a while, quit the process by pressing `ctrl-c`:

```
^C
bob@alice:$ df -k /var
Filesystem        1K-blocks   Used Available Use% Mounted on
/dev/sda1           9833300   2648816   6684980  29% /

bob@alice:$ quota
Disk quotas for user bob (uid 1001):
Filesystem blocks  quota  limit grace files quota limit grace
/dev/sda1  99996   100000 100000      128   0     0
```

Problem 4.22 Which resources are exhausted in this attack? Why is the user's quota not debited when the links are created? What can the adversary accomplish with such an attack? How can this problem be solved?

Whenever possible, world writable directories should be moved to a partition separate from the operating system. Otherwise, depending on the system's implementation, an adversary who is able to fill the entire file system as described above could successfully mount a denial-of-service attack. For a system-wide temporary directory, a RAM disk could be used.

4.6 Change Root

A further possibility to restrict permissions is *Change Root* (chroot), in which a process is "jailed" in a subdirectory of the file system.

▷ Read the manual page for `chroot`.

In order to jail a process in a new root directory, we will have to provide all the necessary commands we want the process to be able to execute within this chroot environment. As an example, we will execute the simple script `chroot-test.sh`, which only returns a listing of the root directory of the system.

Note that in this section the term root does not refer to the superuser *root* but to the root node of the file system.

▷ Create a simple script `/home/alice/chroot-test.sh`, which contains only the command `ls /`. Then make it executable and execute it.

```
alice@alice:$ /home/alice/chroot-test.sh
```

Now, we will jail the script in the `/home` directory. Hence, we will have to provide the binaries and necessary libraries for `bash` and `ls` in the new environment. We therefore first create a `bin` and a `lib` directory and then copy all the necessary files. Note that by using the command `lld` a binary's dependencies can be examined.

▷ Build a minimal chroot environment by performing the following commands.

```
alice@alice:$ sudo mkdir /home/bin /home/lib
alice@alice:$ sudo cp /bin/bash /bin/ls /home/bin

alice@alice:$ ldd /bin/bash
linux-gate.so.1 =>  (0x00695000)
libncurses.so.5 => /lib/libncurses.so.5 (0x00947000)
libdl.so.2 => /lib/tls/i686/cmov/libdl.so.2 (0x0082d000)
libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0x00a8c000)
/lib/ld-linux.so.2 (0x00535000)

alice@alice:$ sudo cp /lib/libncurses.so.5 /lib/libdl.so.2 \
> /lib/libc.so.6 /lib/ld-linux.so.2 /home/lib

alice@alice:$ ldd /bin/ls
linux-gate.so.1 =>  (0x0055c000)
librt.so.1 => /lib/tls/i686/cmov/librt.so.1 (0x00901000)
libselinux.so.1 => /lib/libselinux.so.1 (0x00715000)
libacl.so.1 => /lib/libacl.so.1 (0x00819000)
libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0x009a2000)
libpthread.so.0 => /lib/tls/i686/cmov/libpthread.so.0
(0x007f8000)
/lib/ld-linux.so.2 (0x00f93000)
libdl.so.2 => /lib/tls/i686/cmov/libdl.so.2 (0x00115000)
libattr.so.1 => /lib/libattr.so.1 (0x00989000)

alice@alice:$ sudo cp /lib/librt.so.1 /lib/libselinux.so.1 \
> /lib/libacl.so.1 /lib/libpthread.so.0 /lib/libattr.so.1 \
> /home/lib
```

You can similarly define additional commands that you want the jailed process to be able to execute.

▷ Execute the `chroot-test.sh` script again, but this time in the jail. Be aware of the fact that the path to the file differs, since we set the root to `/home`.

```
alice@alice:$ sudo chroot /home /alice/chroot-test.sh
alice  bin  bob  lib
```

As you can see, the output reflects that the script was executed in the defined chroot environment. Since we also copied bash and its dependencies, you can also execute an interactive shell in the jail and try to escape.

```
alice@alice:$ sudo chroot /home bash
bash-4.1#
```

Problem 4.23 What are the security-relevant advantages of chroot environments, for example, for a server program?

We will now apply this mechanism to the remote access on **alice**. OpenSSH offers the opportunity to jail the users that are logging in remotely by restricting the files and directories that they may access.

▷ Read the manual page for `sshd_config`.

In what follows, we will restrict SSH access on **alice** in such a way that any authorized user is jailed in the directory `/home`. Thus, a user accessing **alice** remotely will not be able to leave the `/home` directory and is provided only with a minimal set of commands. We will use the change root environment built above.

▷ Append `ChrootDirectory /home` to the `sshd_config` file. Note that you need to be *root* to execute the next command.

```
root@alice:# echo "ChrootDirectory /home" >> \
> /etc/ssh/sshd_config
```

▷ Restart sshd now and log in to **alice** from another machine, for example, from **bob**.

```
alice@alice:$ sudo /etc/init.d/ssh restart

bob@bob:$ ssh alice@alice
alice@alice's password: *****
```

You are now logged in on **alice**, but restricted to `/home` as the root directory. You could use this mechanism to build more complicated environments. For example, in `/etc/ssh/sshd_config` you could chroot to `%h`, which refers to the current user's home directory. Then, analogously to the above, you could create different environments for every user and thus restrict them to specific sets of allowed commands within their own home directories.

Problem 4.24 What problem arises when building sophisticated chroot environments? What difficulties do you expect in terms of the administration and the operation of such environments?

Note that whenever an adversary gains superuser access within a chroot environment, there are different possibilities to escape from the jail. One is based on the fact that not all file descriptors are closed when calling `chroot()`. A simple C program could be written that exploits this fact. More restrictive solutions than chroot exist, such as virtualization. However, chroot offers a simple and straightforward way to restrict processes and their abilities.

4.7 Exercises

Question 4.1 It is mentioned at the start of this chapter that access control is used in memory management, operating systems, middleware application servers, databases and applications. In each of these cases, what are the respective subjects and objects (i.e., resources)? What kinds of authorization policies are typically enforced by the access control mechanism?

Question 4.2 Explain what `chroot` does and what kinds of security problems can be tackled using this command.

Question 4.3 Give two examples of how the application of `chroot` is helpful and explain the benefits.

Question 4.4 Unfortunately, `chroot` has some security weaknesses. Explain two of these.

Question 4.5 You have learned about file attributes such as `r`, `w`, `x` and `t`. Briefly explain these attributes.

Question 4.6 There are file additional attributes beyond those of the last question. Name two such attributes and a security application for each of them.

Question 4.7 Explain the concept behind the setuid (and setgid) bits of Unix files.

Question 4.8 Explain why a buffer-overflow vulnerability in an executable program where the setuid bit is set is typically more serious than a similar vulnerability where the setuid bit is not set.

Question 4.9 Recently there have been various attacks based on so-called DNS-rebinding. A description of such an attack is given below. The setting is depicted in Fig. 4.1.

1. Explain how the adversary can use the technique described in the attack to access a server behind the firewall. That is, describe the chain of events that leads to information being transmitted from the internal (possibly secret) server to the adversary.
2. Describe two countermeasures that could prevent such an attack. These countermeasures should not be overly restrictive. For example, closing the firewall for all connections to the Internet would not be an option.

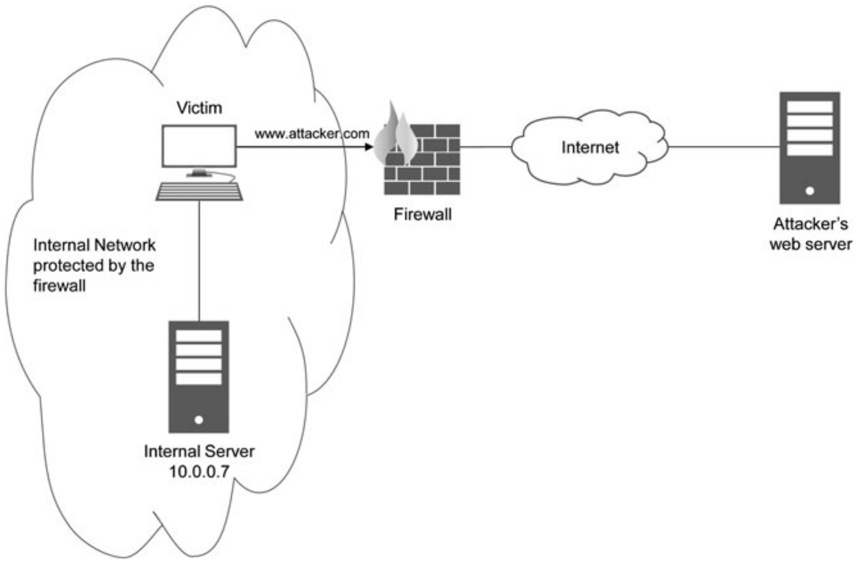

Fig. 4.1 Overview of the DNS-rebinding attack

Attack: The simple attack, where a JavaScript would directly access information from the internal server and forward it to the adversary, is prevented by the so-called same-origin policy. Most web browsers implement this policy in the way that content from one origin (domain) can perform HTTP requests to servers from another origin, but cannot read responses since access is restricted to "send-only".

Therefore, adversary Charlie registers a domain name, like `attacker.com`. As the authoritative name server for this domain, he registers his own name server. If somebody wants to access the domain `attacker.com`, the DNS resolution is done by the adversary-controlled DNS server. On his DNS server the adversary sets the time to live (TTL) field associated with the domain `attacker.com` to a low value such as 0 seconds. This value determines how long a domain name IP address pair is cached before the authoritative name server is contacted again to resolve the name.

The adversary now provides visiting clients a malicious JavaScript (e.g., reading content from a directory on `attacker.com` and sending the data back to `attacker.com`). Since the TTL of `attacker.com` has been set to 0, the client that wants to read data from `attacker.com` again contacts the adversary's name server to resolve the domain name. This time the adversary's DNS server does not respond with the correct IP address.