Security Lab – Authorization in Linux

VMware

This lab must be done with the **Ubuntu image**.

1 Introduction

In this lab, you are going to broaden and deepen your knowledge on authorization in several ways. First, you are going to play around with the default way of doing discretionary access control (DAC) in Linux to see its merits and limits. Next, you'll get in touch with POSIX ACLs. POSIX ACLs are one way to overcome (some) of the limits of the default DAC approach in Linux. This topic is concluded with an exercise where you have to configure permissions to match the security policy of the (fictional) company "SecuSoft & Consult AG". Successful completion of this exercise is worth 4 lab points.

We then switch to mandatory access control (MAC). You are going to solve some tasks related to MAC with *AppArmor*. This topic is also concluded with an exercise where we ask you to secure an application with *AppArmor*. Successful completion of this exercise is worth another 2 lab points.

2 Discretionary Access Control (DAC) in Linux

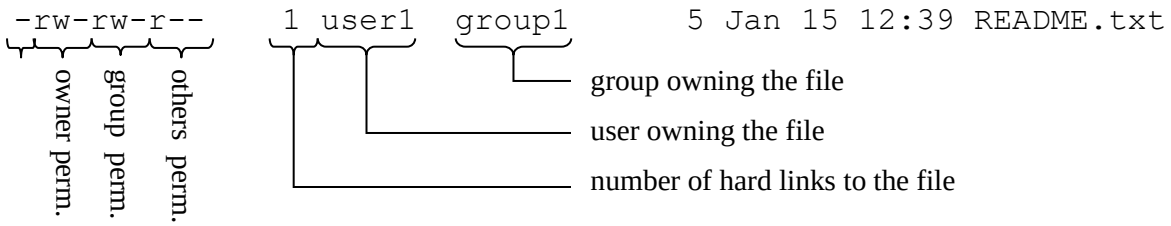
In today's general purpose operating systems (Unix/Linux, Windows, macOS,…), the predominant access control model is discretionary access control: The owner of an object (e.g. a file) controls which subject(s) (user, computer, group,…) can have access to it and to what degree. In the first part of this lab, we are looking at file system-related DAC in Linux.

If we say file system-related DAC in Linux, then we refer to more than just ordinary files. As the Linux expert knows, Linux adheres to the principle that "everything is a file": devices, sockets, pipes, and other things such as the list of file systems currently mounted, are accessible via (special) files.

2.1 The traditional UNIX-style permission model

The Linux security model is based on the one used in UNIX systems. On a Linux system, every file is owned by a user (the *owner*) and a group (the *group owner*). There is also a third category of users typically called *others* (or *world)*, since this category stands for all users that are neither the *owner* of the file nor belong to the group owning the file. For each *category* of users (owner, group, others), read, write and execute permissions can be granted or denied. Note that depending on the type of the file, the permissions have a different meaning – we come back to this later. First, let's have a look at the permissions. You do not need to touch the Ubuntu-Image yet, we use some sample outputs to discuss them.

If you use the `ls` command with option `-l`, you get a listing of the content of the current directory including the file permissions for the three user categories. The permissions are described by nine characters that follow the first character, which describes the file type.



The permissions are always in the same order: read, write, execute for the user, the group and others. The following tables show the meaning of the different permission and file type values:

Code	Meaning
- or 0	access right is not granted.
r or 4	read permission is granted
w or 2	write permission is granted
x or 1	execute permission is granted

Table 1 Meaning of permission codes

Code	Meaning
-	Regular file
b	Block special file
d	Directory
l	Symbolic link
n	Network file
p	FIFO
s	Socket

Table 2 Meaning of file type codes

Let's look at the following example:

```
alice@home-pc:~$ ls -l
drwxr-x---    2 alice   www-data      5 Jan 15 12:39 public_html
-rw-rw-r--    1 root    root          5 Jan 15 12:39 README.txt
```

The first file is a directory (code *d*). The user *alice* is the owner and has read, write and execute permissions. Users belonging to the group *www-data* have read and execute permissions. All other users have no permissions at all. The second entry is a regular file (code -). The user *root* and users in the group named *root* can read the file and write to it. Other users are only allowed to read it.

Note that read, write and execute permissions on directories have the following meaning:

- **Read:** You can look at the directory file which lists the files and directories it contains. You don't need read access on the files themselves to list them.
- **Write:** You can modify the content of the directory file. Hence, to delete, rename or move a file, you MUST have write permissions for the directory containing the file! The write permission on the file itself just allows you to alter its content but not to delete, rename or move it! Since deletion of a file is manipulating the contents of the directory file, you don't need write access on the file itself to delete it.
- **Execute:** You can think of *read* and *execute* on directories this way: directories are data files that hold two pieces of information for each file within: (1) the file's name and (2) its inode number[1]. To access a file, you need to know both pieces of information. *Read* permission is needed to access the names of files in a directory. *Execute* (or sometimes called *search*) permission is needed to access the inodes of files in a directory, if you already know the file's name. Hence, to access a file by its file name, as e.g., in the following command: `cat /home/alice/readme.txt` you need the appropriate rights on the file itself but also *execute* permission on the directory containing it and all its parent directories (`/`, `/home`, and `/home/alice`). If you know the inode, you could access it directly without the need for appropriate execute permissions on the directories. However, in Linux/UNIX systems, this can only be done with special tools requiring root privileges.

[1] In Linux/Unix systems, the inode number uniquely identifies a file in a file system. An inode number is basically an "address" that the operating system must know to access the file.

In addition to these basic permissions, there are three additional bits of information. If any of these bits is set, different characters than the *x* are used in the permissions (*s* for SUID/SGID and *t* for the sticky bit):

- **Setuid bit (SUID):** If this bit is set and the file is executed by an arbitrary user, the process will have the same rights as the owner of the file being executed.
- **Setgid bit (SGID):** Same as above, but inherits rights of the group owning the file. For directories, it also may mean (system dependent) that when a new file is created in the directory it will inherit the group of the directory (and not the group of the user who created the file).
- **Sticky bit:** This bit was used to trigger processes to "stick" in memory after finishing execution. Today, this usage is obsolete and currently, its use is system-dependent. Linux ignores the sticky bit on files. On directories, it means that files may only be unlinked (deleted) or renamed by root or the owner of the file.

Ok, that's enough of theory for the moment. Now, power on your Ubuntu image (if you have not yet done so), login as user *user* with password *user* and open a terminal. Prepare the environment by running the following script (using *sudo* makes sure the script is executed as user *root*):

```
sudo /securitylab/authorization/setSimpleDAC.sh
```

Change to the directory */securitylab/authorization/DAC* and list its contents by entering:

```
cd /securitylab/authorization/DAC
ls -l
```

The administrator who set this up wanted to achieve the following goals:

1. The contents of the *userX* folders should be under full control of user *userX*.
2. Group *groupX* should be able to list the contents of the *userX* folders and access any file or folder if the permissions on them allow it.
3. Other users (not *userX* and not in the group *groupX)* should not be able to extract any information about the contents of the folder *userX*. However, if it contains, e.g., a file with read permissions for everyone, they should be able to access it (if they know the name and path).
4. The *public* folder is open to everyone. Any user should be able to create files or directories in the public folder. It is then up to the creator/owner to set appropriate access permissions.
5. The *management* folder is owned by the user *root* and the group *management* which both have full control. All other users should not be able to access this folder, even if a user in the *management* group accidentally grants read permissions to everyone on one of his files.
6. The *development* folder is owned by the user *root* and the group *developers* which both have full control. All other users should be able to browse through this folder and list its contents.

Question: Why is it the administrator (root) who had to set this up by executing the script above? *Hint*: Try to create a file in */securitylab/authorization* by entering `touch test.txt`. and ask yourself why this fails.

Question: Compare the goals with the information extracted from the directory listing and note down which goals are not met and what must be changed so that they are met.

Now, with the help of the command *chmod*, make the above changes so that the goals are met.

Usage: **chmod <**`permissions`**> <**`file or directory`**>**

While <permissions> can be specified in different ways, the shortest form is the numeric form (see Table 1) where you add up the permissions on a per category basis resulting in three numbers -one for each category.

Example: **chmod** `700 example.txt` // is equal to permissions *rwx------*

Question: What are the commands you need to execute to make these changes?

Question: Now that the goals are met, is it possible to protect all of your files located in the *public* folder from being renamed or deleted by other users? And how about files in a folder you create within the *public* folder? Justify your answer.
Hint: Recall the description of the Sticky bit and look at the permissions of the files in the public folder (`ls -l public`). Try to delete e.g., the file *file_user2.txt* owned by *user2* (`rm public/file_user2.txt`) where you have read permissions only (the user *user* falls into the category *others* for this file). Try to do the same with the file in the *dir_user2* subfolder (`rm public/dir_user2/file_user2.txt`).

Let's now develop this scenario further and look at how we can meet the following additional requirement:

- The structure should include a folder called *meetings* where both, the *management* group and the *development* group should have full access.

Multiple groups requiring access to the same directory is actually quite a common requirement: Companies typically make use of groups to reflect a company's structure (teams, departments, country-offices…) and to group, e.g., people with similar tasks or functions and it is reasonable to assume that a company manages a lot of information which should be accessible to people in more than just one group. The technical specifications of a product should, e.g., be accessible to the project team responsible for this product (to keep it up-to-date) but also to product managers and sales personnel. In many

cases, it may also be required that different groups are assigned different permissions, unlike our requirement from above.

Question: How would you fulfill the above requirement with Unix-style permissions and groups? You don't have to implement the solution, just write down how you'd basically solve this. Why is the solution not optimal from a management/maintenance point of view?
Hint: Linux offers the following commands that you would use in practice to fulfill this task – check the manpages of these commands for details: `addgroup <group>` creates a new group. `adduser <user> <group>` adds a user to a group. And `groups <user>` lists the groups of a user.

Now let's specify another constraint for the *meetings* folder: The permissions for users in the management group should be different from those in the development group.

Question: Why can't you implement this constraint with the simple default Linux DAC mechanism?

This clearly shows you that the Linux DAC mechanism – while functional and simple – has its limitations if the requirements grow a bit more complex. To have more fine granular control and to mitigate maintenance issues, it is necessary to switch to a more powerful ACL mechanism.

2.2 POSIX ACLs

Most of the Unix- and Unix-like operating systems (e.g. Linux, BSD, or Solaris) support POSIX.1e ACLs, based on an earlier POSIX draft that was abandoned. We won't go into all the details of POSIX ACLs, but will focus on one feature that allows to solve the problem of assigning different permissions to the *development* and *management* groups for the *meetings* folder. This should demonstrate that POSIX ACLs are more powerful than the Linux DAC mechanism.

In general, POSIX ACLs can be used for situations where the traditional file permission concept does not suffice. They allow the assignment of permissions to individual users or groups even if these do not correspond to the owner or the owning group of a file. With POSIX ACLs, complex scenarios can be realized without implementing complex permission models on the application level. The advantages of POSIX ACLs are for instance clearly evident in situations such as the replacement of a Windows file server by a Linux file server. Since Windows supports fine granular control of permissions, the Linux file server would be unable to implement them with simple Linux permissions.

To use POSIX ACLs, you have to configure the file system to use it. This is simple and only requires you to add the option *acl* to the line for the root file system in */etc/fstab*. You need to do this as root (by using *sudo*). Do this now in the editor of your choice (e.g., vim or nano). The modified line should look like this:

```
#<filesystem> <mount-point> <type> <options>              <dump> <pass>
UID=<number>  /             ext4   acl,errors=remount-ro  0      1
```

To make Linux aware of it, you have to remount the root files system by entering:

```
sudo mount -o remount /
```

This points us at a VERY IMPORTANT aspect of discretionary access control using ACLs (permissions stored with the objects): Whether or not these ACLs are taken into account depends on whether they are respected by the operating system! If you are e.g., able to mount a POSIX ACL protected file system without the *acl* flag, you might get more permissions than you actually have (you'll try this out later). That's certainly one good reason why mount operations typically require root privileges. Of course, if you have access to the hardware, you could also just take the hard drive and mount it in a system of your choice with a file system driver of your choice.

Let's now experiment a bit with POSIX ACLs and finally create the *meetings* folder with appropriate rights for the *developer* and *manager* groups. Let's start by creating the *meetings* directory and checking its permissions. Enter the following commands:

```
su -
```
(followed by password root, as *umask* is user-specific)

```
cd /securitylab/authorization/DAC
umask 027
mkdir meetings
chown root:management meetings
exit
```
(to make sure you are working as *user* again)

The *umask* determines which permissions will be masked off when the directory (or file) is created. A umask of 027 (octal) disables write access for the owning group and read, write, and execute access for others. It basically inverts the meaning of the codes in Table 1 (e.g., 0=no permissions → 0=all permissions 2=write access → 2=disables write access etc.).

Now, execute the following command:

```
ls -dl meetings
```

Question: Who is currently allowed to do what with files in this directory?

Note that so far, we haven't explicitly configured the (POSIX) ACL of the *meetings* directory. However, the basic Linux permissions we configured above become part of the ACL, which can be verified using the *getfacl* command, which is used to display the ACL. Enter the following:

```
getfacl meetings
```

You should get the following output:

```
# file: meetings/
# owner: root
# group: management
user::rwx
group::r-x
other::---
```

This is nothing else than the base permissions configured above, but displayed in "ACL syntax":

- The *owner* (root) of the directory translated to the *base user* in the ACL (user::), indicated by the missing name between the two colons (:). Note that the rights (rwx) correspond to the rights defined above.
- Likewise, the *group owner* (management) is translated to the *base group* in the ACL (group::), again indicated by the missing text between the colons (:). The permissions correspond to the permissions defined above.
- The right of *other* is also directly inherited from the configured base permission.

If a POSIX ACL contains only an entry for the *owner*, the *owning group* and *other class* – as it does so far – it is a so-called **minimal ACL** and the permissions shown by the `ls -l` command reflect exactly the permissions according to the POSIX ACL.

Once ACLs have been enabled, you can now add additional entries to the ACL, so-called **named users or named groups**. Named users or named groups are nothing else than additional users or groups that can access the file or directory, whereas each user/group can have separate permission. We will use this to add ACL entries for the *management* and *development* group. Note that it doesn't matter that we add a named group that is the same as the owning group (management) of the directory – the effective access permissions of the management group will be determined by combining (adding) the rights of "both" groups.

Use the *setfacl* command to add the two named groups:

- grant read, write and execute permissions to the *development* group by entering:
  `sudo setfacl -m group:development:rwx meetings`
- grant read and execute permissions to the *management* group by entering:
  `sudo setfacl -m group:management:rx meetings`

To check whether it worked, look at the output of the *getfacl* command for the *meetings* directory.

Question: Do the ACL entries reflect the permissions as specified above? Note down the three (!) new output lines:

The third new line is the *mask* entry. It is automatically created and its permissions are set to the union of the permissions of the group owner and all named users and named groups.

Now, check again the output of `ls -dl meetings`. Interestingly, the permissions for the owning group are now `rwx`. The explanation is that when ACLs are used, these permissions are not the one of the owning group, but represent the mask. The mask represents the upper bound the owning group or any named user/group can have: if the mask is `r-w` und a named group has the permissions `rwx`, the effective permissions are `r-w`. Usually, the mask has no notable effect because whenever a named user/group is added that extends the permissions of the mask, the new permissions are added to the mask.

Note that the "old" owning group permission is actually replaced (in the file system) with the mask value, which is why the *ls* command prints it. The reason for this has mainly to do with backward compatibility for tools that are not aware of POSIX ACLs and that therefore interpret the standard permissions: they now simply get the "combined permissions" of any named user/group and the owning group. Note also that the administrator can manually reduce the mask permissions (with *chmod* in just the same way as you would adapt the owning group permissions) to reduce the permissions of all named users/groups and the group owner.

Question: Aside from the permissions of the mask, there is something else that changed: An additional character not present before indicates that for this directory an extended ACL is in effect. What character is this?

As explained above, according to the *ls* command, the owning group (management) now has permissions `rwx`. Is this a problem? They should only have permissions `r-x`. Test it by trying to create a file as *user1*, who is in the management group:

```
sudo -u user1 touch meetings/test.txt
```

Question: What is the result of this test? Explain the result.

Now, disable *acl* support by modifying */etc/fstab* again: alter the *acl* option into the *noacl* option[2] and remount the root file system. Do the same test again:

```
sudo -u user1 touch meetings/test.txt
```

Question: What is the result of this test? Explain the result. You can also use *ls* again to check the permissions of the management group

This seems strange: After enabling and disabling ACLs, the management group has more permissions than in the beginning (remember that in the beginning, we gave the management group read and execute permissions on the *meetings* directory, which does not allow members of that group to create a file within the directory). But recalling that the mask replaces the original owning group permissions, this makes perfectly sense. You can argue this is a security limitation and in fact it is, as the permissions of the owning group are increased in this case although the administrator never actually granted the write permission to the management group. The morale of the story is that as an administrator, you simply have to be aware of this: When switching to ACLs, you are likely to stick to it so it's not a problem in practice. However, keep in mind that deactivating ACLs does not imply that the permissions are the same as they were before activating them.

3 Exercise: Implementing the security policy of "SecuSoft & Consult AG" using POSIX ACLs

"SecuSoft & Consult AG" is a (fictional) IT security company. More precisely, it is both a software producer and a consulting firm. The company produces *WebTables*, a web application firewall and *NetMonitorIT*. Both products as well as the company's consulting services enjoy a good reputation.

[2] Simply removing the previously set *acl* option does not work because permissions have been added. Therefore, the *noacl* option is needed to explicitly ignore them.

To facilitate the management and sharing of data, the company set up a file server accessible to all employees. Clearly, there is a lot of information that not everyone should be able to see or modify: There is, e.g., no a-priori need for a developer to access information from consulting projects or for consultants to access the data of all consulting projects. Furthermore, since in some projects, the company cooperates with freelancers or other companies, not only employees but also external people need access to (selected areas of) the file server.

After a thorough analysis of the processes, the chief security officer (CSO) released a new security policy for the file server. As the overall policy is quite complex, we consider only parts of it.

Let's now have a look at the structure of the file server, the users and their group and project memberships and finally the security policy before we discuss your task in more detail.

3.1 Directory structure of the file server

The directory structure of the file server is as follows (entries not preceded by dashes are files):

```
|-applications                      # contains applications
     erp_client
     timesheet_client
|-projects                          # contains projects
|---consulting
|-----Julius_Baer_I_EB_200908       # banking project
|-------report
|-----Xilinx_TrustDB_200704         # industry project
|-------report
|-staff                             # Contains directories of employees
|---A
|-----ahas                          # Directory of employee "ahas"
|-------public
|-------public_html
 (...)
```

3.2 Users, groups and project memberships

We consider the following *users*:

- Employees: ahas, ator, dbau, fgla, wmei
- Externals: krol

The following table describes the *groups* and *group memberships*:

Groups	Description	Members
staff	All employees of the company are in this group.	ahas, ator, dbau, fgla, wmei
ext	External people (freelancers, sub-contractors…)	krol
consulting	All employees working as consultants are in this group (members are also in staff)	ahas, ator, dbau, wmei
management	CEO, CSO, CIO, CTO and members of the board of directors (members are also in staff)	fgla
pm_banking	Project manager(s) for consulting projects in the banking sector (members are either in staff or ext)	ator
pm_industry	Project manager(s) for consulting projects in the industry sector (members are either in staff or ext)	dbau

These users, groups and memberships will be automatically created by the script *fileserver.sh* (see below).

In addition, there are two project teams for the two projects defined in the directory structure:

- Julius Baer project: ahas, wmei
- Xilinx project: wmei, krol

This is not created by the script; you have to do this yourself (details see below).

3.3 Security policy

General:

- **Use groups, not individual users:** Whenever possible, avoid assigning permissions on a per user basis. It complicates the management of permissions and requires inspecting and updating many objects when the user switches e.g. from the development to the consulting staff or when he leaves the company. It is far easier to just modify his group membership(s).
- **Default deny:** If something is not explicitly allowed by the policy, it is denied.

Resources and Access policy:

- The following table specifies who needs access to what. It contains nearly all necessary information you'll need to solve the task:

Resource	Policy
ERP client	*management* must be able to start this application.
Time Sheet client	*staff* and *ext* must be able to start this application.
projects/	*staff* and *ext* can traverse the directory and all subdirectories.
consulting/	*consulting* can browse it (just this directory). Since all consultants are also in the *staff* group, it's not necessary to give the *consulting* group the rights to traverse the *projects* directory.
consulting/<project>/	Project team members and the appropriate project manager(s) have full control. *management* can access files in the *report* subdirectory. Since all of these people are either in the staff or ext group no additional traversal rights must be given for them to reach the *consulting/<project>* directory.
staff, staff/<A-Z>	Employees (*staff* group) can browse the contents of these folders.
Personal folder	Each employee owns his staff folder (staff/<A-Z>/<employee id>) and everything it contains. Owners have full control on directories (rwx) and read/write permissions on files (rw). Employees of the company (*staff* group) can access files located in the subfolder *public*. The *public_html* subfolder is reserved for the personal web page. The user *www-data* needs access to its content. To help with this task, there exists a group *www-data*, and you may assume that the user *www-data* is the only member of this group.

Note that "can access files in a directory" means that a file in this folder can be read / modified / executed if its file permissions are set accordingly! Deleting and renaming files in this folder should NOT be possible.

3.4 Task

Your task is now to implement the above policy for the directory structure shown above.

Open a console and become *root* by entering `su -` and providing the password *root* when asked. Re-enable *acl* support by modifying */etc/fstab* and remounting the root file system as described above.

To generate the users, groups and the directory structure needed for this task, execute the following command:

```
/securitylab/authorization/fileserver.sh
```

If you need to reset the configuration for this task, run the following command and start over.

WARNING: This command DELETES the entire fileserver/ directory and recreates it from scratch! Therefore, do not store any of your files inside the fileserver/ subtree.

```
/securitylab/authorization/fileserver.sh clean
```

The file system of the file server is then "mounted" in */securitylab/authorization/fileserver/*. All files and directories are currently owned by the user and group *root* and the permissions of all files and directories are set to *000* (no permissions!).

Take into account the following hints when solving the task:

- To "implement" the two project teams, you have to define two additional groups yourself. Name them *banking* (Julius Baer project) and *industry* (Xilinx project).
- Use a structured approach to solve the problem by, e.g., applying the configurations in sequence according the table above. Furthermore, it is advisable to put all necessary commands into a file so you can execute all of them together by executing the file.
- As you may have noticed, with the exception of the two applications and the personal folder, the policy is just about ACLs on directories. Use the tips and tricks in the box below to apply a modification to a single file or directory or to multiple files or directories or both.

When you think you are done, you can execute the *checkPolicy.pl* script to check whether your configuration correctly implements the policy. Execute the script with the following command:

```
/securitylab/authorization/checkPolicy.pl
```

If any problems are detected, you'll get a description, which will be helpful to correct the configuration. If the script finishes without errors, then you have successfully completed this task and are ready to get the corresponding lab points.

Tips & Tricks

```
chown -R nobody:root /home/user        #sets the owner of the folder /home/user and everything
                                       #it contains (-R means recursive) to nobody and the
                                       #owning group to root
setfacl -R –m group:mygroup:rx /usr    #sets read and execute permissions for mygroup for the
                                       #whole /usr subtree (files and folders)
```

The following commands affect all files/directories in `<list>`:

```
setfacl -m group:mygroup:rx <list>     #sets read and execute permissions for mygroup
getfacl <list>                         #shows the ACL(s)
chmod 700 <list>                       #set permissions to full control for the owner and to no
                                       #permissions for owning group/others.
chmod u+rwx <list>                     #set permissions to full control for the owner while
                                       #keeping all other.permissions.
```

`<list>` can be a single directory or file such as /usr/ or /data/myfile.txt. But it can also be replaced by an expression returning a list of files and|or directories or both. Examples of such expressions:

```
/usr/*/*/data/          #All 4rd level directories with name data located in /usr
/usr/*/data/            #All 3rd level directories with name data located in /usr
/usr/data/*             #All files and directories in the /usr/data/ directory
/usr/data/*/            #All directories in the /usr/data/ directory
`find /usr/data/`       #List of all files and directories in the /usr/data subtree
`find /usr/data/ -type f`   #List of all files in the /usr/data subtree
`find /usr/images/ -type d` #List of all directories in the /usr/images subtree
```

4 MAC in Linux with AppArmor

As briefly introduced in the lecture, AppArmor is a security tool designed to provide an easy-to-use security framework to confine individual applications. By enforcing good behavior, AppArmor proactively protects the operating system and applications from external or internal threats.

AppArmor security policies, called ***profiles***, completely define what system resources individual applications can access, and with what privileges. Any restrictions made in a profile have precedence over restrictions from DAC. To enforce this precedence, AppArmor interacts with the Linux Security Modules (LSM) kernel interface.

A number of default profiles for well-known applications are included with AppArmor. But even applications for which there is no such profile can be deployed successfully in a matter of hours. To do so, AppArmor can be configured to log (to */var/log/syslog* in Ubuntu) and not prevent access operations of such an application. If violations are logged but not prevented, the profile is said to operate in *complain* mode. The information in the log can then be used to create a profile. To actually prevent violations, the profile must then be set to *enforce* mode. However, there are limitations with this approach since it is difficult to cover all possible use cases necessary to compile a profile that is self-contained.

In this lab, you are going to develop a profile for the Firefox browser to get familiar with AppArmor configuration and usage. Clearly, Firefox is an application that is constantly exposed to a hostile environment. If Firefox is hijacked, e.g., due to some flaw in its code, an attacker may use Firefox to access your home directory or system files or to execute arbitrary code. With access to so many things, the chance is high that an attacker can exploit another (local) vulnerability to gain root access.

The traditional approach to handle vulnerabilities in an application such as Firefox is to fix the flaws in the code or the configuration (with a patch) and possibly creating rules for an IDS (remember Snort?) to detect attacks targeting these vulnerabilities. The approach taken by AppArmor is different. Instead of a reactive approach fixing things when they are discovered, AppArmor tries to limit damage proactively: AppArmor confines Firefox by enforcing that only the objects specified in the profile can be accessed.

The profiles of the applications are stored in separate files in */etc/apparmor.d*. The file name of a profile corresponds to the full path to the application it confines (dropping the first "/" character and converting the others to a "." character. Hence, the filename of the profile for Firefox is *usr.lib.firefox.firefox.sh* because the full path to Firefox is */usr/lib/firefox/firefox.sh*.

Once a profile is defined, it is automatically activated when the application is started.

4.1 Anatomy of a profile

Per default any profile consists of four sections:

- **#include:** AppArmor provides an easy abstraction mechanism to group common file access requirements. This makes writing new AppArmor profiles very simple by assembling the needed building blocks for any given program. These building blocks can be included into a profile with the help of the #include statement. The *#include* statement allows embedding the content of an arbitrary text file into the profile: the statement is replaced with the specified file's contents.

Format:	*#include /absolute/path*	Specifies that /absolute/path should be used
	#include relative/path	Specifies that relative/path should be used (is relative to the current working directory)
	#include <magic/path>	Most common usage; it will load magic/path relative to */etc/apparmor.d/* (default).
Example:	*#include <abstractions/fonts>*	Rules to access fonts and the font libraries

- **capability:** This section specifies the POSIX capabilities to which the application is restricted. See the man pages (`man capabilities`) or one of the many resources on the web.

Format: *capability <capability name>,*
Example: *capability setuid,* Allow arbitrary manipulations of process UIDs.

- **hats:** While an AppArmor profile is applied to an application, there are times in which a sub process of the program may need access differing from the main program. In this event, the sub process may "change hats" or use an alternate sub-profile. Therefore, a profile may have more than one sub-profile. Right now very few applications use hats (one of them is Apache).
- **rules:** The rules section specifies what objects the application can access with what permissions. There are two types of rules: file rules and network rules. Rules are discussed in more detail in the next section.

Below you find a sample profile. Just have a look at it for now, explanations will follow:

```
#include <tunables/global>              #include statement (mainly
                                        #includes some variables)
@{HOME}=/home/*/ /root/                 #variable

/usr/bin/foo {                          #profile for /usr/bin/foo

   #include <abstractions/base>         #include statement
   network inet tcp,                    #network rule
   capability setgid,                   #capability
   link /etc/sysconfig/foo -> /etc/foo.conf,     #link rule

   /dev/{,u}random     r,               #file rule
   /lib/lib*.so*       mr,              #file rule
   /proc/[0-9]**       r,               #file rule
   /tmp/               r,               #file rule
   /tmp/foo.pid        wr,              #file rule
   @{HOME}/.foo_file   rw,              #file rule
   @{HOME}/.foo_lock   kw,              #file rule
   deny /etc/shadow    w,               #file rule (deny)
   owner /home/*/**    rw,              #file rule: grant access to
                                        #files owned by the user as
                                        #which the application runs

   /bin/**             px -> bin_generic    #apply proile bin_generic
                                        #when starting an application
                                        #in /bin. Profile must exist.
}
```

4.1.1 Rules

Rules are basically a set of permissions applied to files and directories or network access. Note that the next two sections just describe the basic syntax and give some examples. For detailed syntax information, see the following two sections (4.1.2 and 4.1.3) or – for a more detailed explanation – see the appendix (Section 6).

4.1.2 File rules

The syntax for file rules is as follows:

FILE RULE = ('"' *FILEGLOB* '"' | *FILEGLOB*) *ACCESS* ','

Examples:			
	`/tmp/`	`r,`	allow read on directory */tmp* (to list its content)
	`/tmp/*`	`rw,`	allow read/write on files directly in */tmp*
	`/tmp/*/`	`r,`	allow read on all directories directly in */tmp*
	`/tmp/**`	`r,`	allow read on anything anywhere underneath */tmp*

`/tmp/**/ r,`	allow read on directories anywhere underneath */tmp*

Furthermore, these rules can be preceded by the keywords *deny* or *owner*. *deny* is typically used to exclude selected files included by a coarse grained allow rule. If a rule allows, e.g., read access to everything in */etc*, read access to */etc/shadow* can be prevented by adding an appropriate *deny* rule. The *owner* keyword restricts access to files owned by the user who requests access. Hence, if a user starts an application with the rule *owner /home/*/** *rw*, in its profile, the application is typically restricted to read/write on files in this user's home directory.

What is a bit special is the distinction between reading/writing files and reading a directory:

- */tmp/* rw*, allows to write (create, delete, modify) files in the *tmp* directory. But you don't need write access to the directory itself (as is needed with DAC), this is implicitly granted.
- To read the directory content (list the files and directories it contains), however, you explicitly need read rights on the *tmp* directory. The rule above only grants the right to read individual files in the directory (provided you know the path), but not to list them. As a result, */tmp/ r*, is also needed to grant the right to list the files.

Question: What rules are required to create, read and write files directly in the */etc* directory and to also list the content of the directory? As mentioned above, you need TWO rules to accomplish this.

4.1.3 Network rules

Network rules specify whether and with what restrictions networking is allowed. The rule syntax is:

NETWORK RULE = 'network' [[*DOMAIN*] [*TYPE*] [I <*PROTOCOL*>]] ','

Examples:	`network,`	allows all networking
	`network tcp,`	allows IPv4 and IPv6 TCP networking
	`network inet tcp,`	allows IPv4 TCP networking

4.1.4 Fixing a broken profile

Before you are going to write a profile (almost) from scratch, we guide you step-by-step through the process of fixing a profile. For this purpose, we deliberately "broke" the profile for the Firefox browser.

Open a terminal and become *root* by entering `su -` (password: *root*). Use this terminal for all subsequent commands except for starting *Firefox*. Enter the following command to load the broken profile:

```
/securitylab/authorization/setAppArmor.sh
```

The script installs the broken profile and restarts AppArmor to activate it.

Next, start *Firefox* (as user *user*, not as *root*!) and (1) try to open a web page and (2) try to save a page to */home/user/Downloads* by right-clicking and selecting *Save Page As…* Try different ways to access a web page, e.g. by using the IP address instead of the host name (you can get the IP address of a host name with the *nslookup* command in a terminal). Also, check the DAC permissions when trying to save the page.

Question: What is the result of these tests and what do you think are the reasons for the observed results?

Question: What changes to the profile do you think could solve the first problem (open a web page)? The changes should not open up the profile more than necessary. Don't modify the profile yet!
Hint: This problem can be solved by using an *#include* statement. Check the files in */etc/apparmor.d/abstractions* for candidates.

Question: What changes to the profile do you think could solve the second problem (save a page)? The changes should not open up the profile more than necessary. Don't modify the profile yet!

Question: What result do you expect if you started Firefox as root (don't do it, just think about it)? Justify your answer.

Now let's see whether your two modifications from above are correct by checking what the automated process for solving these problems would suggest. To do so, you'll run the Firefox profile in complain mode so that violations are logged to */var/log/syslog* but not blocked. You then use the *aa-logprof* tool to process the logged data and to modify the profile accordingly:

1. Put the Firefox profile in complain mode (only log violations, don't prevent them):

   ```
   aa-complain firefox
   ```

2. Add a mark to the */var/log/syslog* file so that we can tell the *aa-logprof* tool where it should start looking for information. You can do this with the following command (use e.g. a number for `<identifier>` and use a different number when repeating this procedure).

   ```
   echo "MARK-<identifier>" >> /var/log/syslog
   ```

3. Start Firefox (as user *user*, not as *root*) and redo the two tests from above. Then close Firefox.

4. Start the *aa-logprof* tool to review the complain mode output found in */var/log/syslog* and to generate entries to fix the *Firefox* profile to allow denied operations:

   ```
   aa-logprof -m "MARK-<identifier>"
   ```

5. *aa-logprof* should now propose changes to the profile. Check these proposals carefully, they should be quite self-explanatory. With file rules, don't just check the path but also the permissions (r/w etc.).
   Note 1: If *aa-logprof* proposes more than one fix, you can select which one should be used by pressing the corresponding number and then press A to allow the related operations.

Note 2: Pressing G gives you additional fix options.
Note 3: As soon as you think you have included everything, you can select F.

6. After stepping through all of the proposals (or selecting F), you are asked whether you want to save the changes. If you think you made a mistake, do not save the changes and start over.
7. Put the Firefox profile back to enforce mode:

   ```
   aa-enforce firefox
   ```

8. Start Firefox and redo the tests. If the problems are solved now, you are done. Otherwise, go back to 1 (as sometimes, more than one iteration is needed) or try to fix the profile (*usr.lib.firefox.firefox.sh*) manually.

Question: Which additions to the Firefox profile does *aa-logprof* propose to address the problems encountered with the above tests? Are the proposed rules restrictive enough or could they be improved? Check whether they match the changes you proposed in the answers to the two questions regarding this issue.

5 Exercise: Protecting an Application using AppArmor

You should now be ready to protect a custom-made sample application called "ShareIt 2.0". While this application does not do anything useful, it accesses resources a file-sharing application would probably also need to have access to. Your task is to write a profile allowing execution of the "ShareIt 2.0" application without giving it access to anything it does not need. The access control-relevant specification of "ShareIt 2.0" is as follows:

- "ShareIt 2.0" can get the identity of the user which started the program, can do DNS lookups, and can talk to other "ShareIt 2.0" clients using the UDP protocol. Note that all of this can be enabled by including <*abstractions/nameservice*>.
- "ShareIt 2.0" reads configuration data from */etc/shareit.conf* when it is started.
- "ShareIt 2.0" uses */tmp/shareit* to store (write and read) temporary files
- "ShareIt 2.0" must be able to execute the shell commands `rm`, `touch`, and `mkdir` (all located in */bin*). The commands must be run with the same restrictions as "ShareIt 2.0". Note that this requires granting *rix* rights to the executables (*r* for read and *ix* for starting the application).
- The user executing the application should be able to store downloaded files in his *home* and read data (including the home directory content) from his *home*.
- Access to the hidden files and directories (they always start with a .) located in the *home* directories (/home/<userid>/.<name>, e.g.: /home/user/.keystore) and anything underneath these directories must not be possible. Refer to the appendix for hints about how these files and directories can be identified.

You can try to start the application (as *user*, not as *root*) by entering:

```
/securitylab/authorization/shareit
```

Since the profile in */etc/apparmor.d/securitylab.authorization.shareit* is set to enforcing mode and does not yet meet the specification, the application should fail with an error.

Now, modify the profile – we recommend doing this manually – to meet the specification and test whether the application runs and terminates without errors. Try to write the profile according to the

specification and then test and modify it if necessary. Ignore the line *#include <abstractions/shareit>* that is already included in the profile as this contains additional rules needed for the "ShareIt 2.0" application to run.

Note that when you modify the profile, you have to **reload it** for the changes to take effect. If you switch mode (e.g., *enforcing => complain* or vice versa) this is done automatically. If your current directory is the directory where the application is located (*/securitylab/authorization/*), you can enter:

```
(aa-enforce|aa-complain) shareit
```

Otherwise, you have to provide the full path to the profile:

```
aa-enforce /etc/apparmor.d/securitylab.authorization.shareit
```

If you don't want to switch mode (which you are likely doing here as you should create the profile manually), you can reload it by entering:

```
apparmor_parser -r /etc/apparmor.d/securitylab.authorization.shareit
```

If running the application terminates without errors, you can run it with additional tests by using the option `--checkref`. Once this works without errors as well, you have successfully completed this task and are ready to get the corresponding lab points.

5.1 Cleanup

To reset the system to its original state, execute the command:

```
/securitylab/authorization/unsetAppArmor.sh
```

Lab Points

In this lab, you can get **6 Lab Points**. To get them, you must demonstrate to the instructor that you have successfully solved the tasks in Sections 3 and 5:

- You get 4 points for successfully solving the exercise in Section 3. Demonstrate this by running the script *checkPolicy.pl*.
- You get 2 points for successfully solving the exercise in Section 5. Demonstrate this by running the *shareit* program with the *--checkref* option and by showing the profile you created for the "ShareIt 2.0" application.

6 Appendix: AppArmor reference

This appendix contains detailed information about the syntax of file- and network rules and a listing of useful AppArmor commands.

6.1 File rules

The syntax for file rules is as follows:

FILE RULE = ('"' *FILEGLOB* '"' | *FILEGLOB*) *ACCESS* ','

FILEGLOB = Must start with '/'. **?*[]{}^** have special meanings; see below. It may include a *VARIABLE* whose content is expanded before the rule is evaluated. Rules with embedded spaces or tabs must be quoted. Rules must end with '/' to apply to directories.

Examples:	*/tmp/**	All files directly in */tmp*
	/tmp//*	All directories directly in */tmp*
	*/tmp/***	Files and directories anywhere underneath */tmp*
	*/tmp/**/*	Directories anywhere underneath */tmp*
	*/tmp/.**	All files starting with . directly in */tmp*
	*/tmp/.***	Files and directories starting with . in */tmp* and anything underneath these directories

Character	Meaning
*	Substitutes for any number of characters, except /
**	Substitutes for any number of characters, including /
?	Substitutes for any single character, except /
[abc]	Substitutes for the single character a, b or c
[a-c]	Substitutes for the single character a, b or c
{ ab,cd }	Expand to one rule to match ab and another to match cd
[^a]	Substitutes for any character except a

Table 3 Meaning of special characters in FILEGLOB

ACCESS = ('a' | 'r' | 'w' | 'l' | 'ix' | 'ux' | 'Ux' | 'px' | 'Px' | 'm') [*ACCESS* ...].

The following table contains short descriptions of the access modes relevant to all files and to applications.

	Mode	Meaning
All Files	r	Allows the application to read a file with this name
	w	Allows the application to write to a file with this name
	l	Allow the application to create a link to a file with this name
	k	Allows the application to lock a file with this name
	a	Allows the application to append data to a file of this name
Applications	ix	If the application starts an application with this name, it inherits the parent's profile
	px, Px	If the application starts an application with this name, a separate profile must exist
	Cx,Cx	If the application starts an application with this name, a local profile must exist
	ux, Ux	If the application starts an application with this name, no profile is applied.
	m	Allows a file with this name to be mapped into memory using the PROT_EXEC flag.

Table 4 Access modes for files

6.2 Network rules

Network rules specify whether and with what restrictions networking is allowed. The rule syntax is:

NETWORK RULE = 'network' [[*DOMAIN*] [*TYPE*] [I <PROTOCOL>]] ','

DOMAIN = ('inet' | 'ax25' | 'ipx' | 'appletalk' | 'netrom' | 'bridge' | 'atmpvc' | 'x25' | 'inet6' | 'rose' | 'netbeui' | 'security' | 'key' | 'packet' | 'ash' | 'econet' | 'atmsvc' | 'sna' | 'irda' | 'pppox' | 'wanpipe' | 'bluetooth') ','

TYPE = ('stream' | 'dgram' | 'seqpacket' | 'rdm' | 'raw' | 'packet')

PROTOCOL = ('tcp' | 'udp' | 'icmp')

Examples:

```
network                 #allows all networking
network tcp             #allows IPv4 and IPv6 TCP networking
network inet tcp        #allows IPv4 TCP networking
```

6.3 Commands

There are several useful AppArmor commands:

/etc/init.d/apparmor ***{ start|stop|restart|try-restart|reload|force-reload|status|kill }***
This command is used to start, stop, etc AppArmor service.
aa-complain ***<application name|full path and name of the profile>***
Set an AppArmor profile to complain mode. Reloads the profile. Working with the application name as parameter only works when the application is in the current directory or is in one of the directories referenced by the PATH environment variable.
aa-enforce ***<application name|full path and name of the profile>***
Set an AppArmor profile to enforce mode. Reloads the profile. Working with the application name as parameter only works when the application is in the current directory or is in one of the directories referenced by the PATH environment variable.
aa-unconfined
Outputs a list of processes with open TCP or UDP ports that do not have AppArmor profiles loaded.
aa-logprof ***[-m <mark>] [-d <profile directory>] [-f <logfile to scan>]***
aa-logprof is an interactive tool used to review the complain mode output found in the log-facilities to which AppArmor logs (on Ubuntu typically /var/log/syslog) and to generate new entries in AppArmor profiles.
apparmor_parser ***[-r] /etc/apparmor.d/<profile>***
This command is used to load (or reload with -r option) a profile into the kernel. So after modifying a profile you can use this command to make AppArmor aware of them by reloading it..

```
addgroup banking
addgroup industry
addgroup ahas banking
addgroup wmei banking
addgroup wmei industry
addgroup krol industry

# Grant access to run the applications
setfacl -m group:management:x applications
setfacl -m group:staff:x applications
setfacl -m group:ext:x applications
```

```
setfacl -m group:management:rx applications/erp_client
setfacl -m group:ext:rx applications/timesheet_client
setfacl -m group:staff:rx applications/timesheet_client

# Grant access to projects
setfacl -m group:ext:x `find projects/ -type d`
setfacl -m group:staff:x `find projects/ -type d`
setfacl -m group:consulting:rx projects/consulting
setfacl -m group:banking:rwx -R projects/consulting/Julius_Baer_I_EB_200908/
setfacl -m group:pm_banking:rwx -R projects/consulting/Julius_Baer_I_EB_200908/
setfacl -m group:industry:rwx -R projects/consulting/Xilinx_TrustDB_200704/
setfacl -m group:pm_industry:rwx -R projects/consulting/Xilinx_TrustDB_200704/
setfacl -m group:management:rx projects/consulting/*/report/

# Employees can browse the staff and staff/* folder
setfacl -m group:staff:rx staff/
setfacl -m group:staff:rx staff/*/

# Employees own their folder and have full control
chown ator:root -R staff/A/ator
chown ahas:root -R staff/A/ahas
chown dbau:root -R staff/D/dbau
chown fgla:root -R staff/F/fgla
chown wmei:root -R staff/W/wmei
chmod u+rwx `find staff/*/* -type d`
chmod u+rw `find staff/*/* -type f`

# Employees can access files in all users' public folders
setfacl -m group:staff:x staff/*/*/
setfacl -m group:staff:rx staff/*/*/public

# www-data needs access to public_html
setfacl -m group:www-data:x staff/
setfacl -m group:www-data:x staff/*/
setfacl -m group:www-data:x staff/*/*/
setfacl -m group:www-data:rx staff/*/*/public_html
```

Checking for correctness:

md5sum /securitylab/authorization/checkPolicy.pl

```
11b3880723bbfce4c029b8042f7da917  /securitylab/authorization/checkPolicy.pl
```

/securitylab/authorization/checkPolicy.pl

```
Checking permissions for user with groups "staff"...
Checking permissions for user with groups "consulting:staff"...
Checking permissions for user with groups "management:staff"...
Checking permissions for user with groups "pm_banking:staff"...
Checking permissions for user with groups "pm_banking:ext"...
Checking permissions for user with groups "pm_industry:staff"...
Checking permissions for user with groups "pm_industry:ext"...
Checking permissions for user with groups "ext"...
Checking permissions for user "ator"...
Checking permissions for user "ahas"…
```

Checking permissions for user "dbau"...
Checking permissions for user "fgla"...
Checking permissions for user "krol"...
Checking permissions for user "wmei"...
Checking permissions for user "www-data"...
Group memberships of ator : staff consulting pm_banking
Group memberships of krol : ext industry
Group memberships of wmei : staff consulting banking industry