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.

- **Part 1**: 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. Successful completion of this part is worth 2 lab points.
- **Part 2**: An exercise where you will configure permissions to match the security policy of the (fictional) company "SecuSoft & Consult AG". Successful completion of this exercise is worth 2 lab points.

Each part takes one week.

2 Part 1: Discretionary Access Control (DAC) in Linux

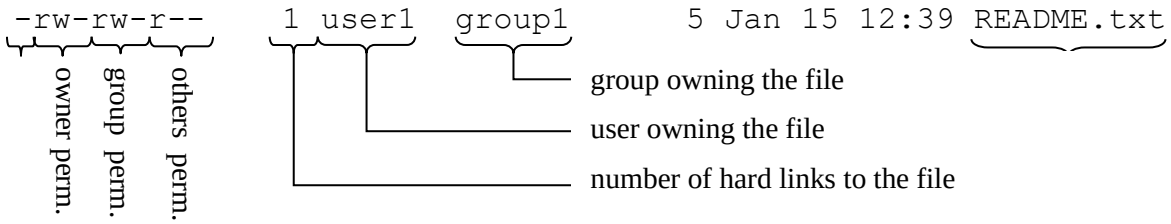
In today's general purpose operating systems (Unix/Linux, Windows, macOS,…), the predominant access control model is discretionary access control: The owner of an object (e.g. a file) controls which subject(s) (user, computer, group,…) can have access to it and to what degree. In the first part of this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

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 login to your Ubuntu vm (user *user*, pwd: securityzhaw) and open a terminal. Prepare the environment by running the following script (using *sudo* makes sure the script is executed as user *root*): For the git clone you need your ZHAW credentials.

- `git clone https://github.zhaw.ch/IS/teaching_public.git`
- `cd teaching_public/ITS/Lab7_authorization/authorization`
- `sudo chmod +x setSimpleDAC.sh`
- `sudo ./setSimpleDAC.sh`

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

- `cd 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.

[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.

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: 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.

1: Richtig
2: Falsch -> Read und Execute Permission
3: Richtig
4: Falsch -> Alle haben alle Permissions
5: Falsch -> Jeder hat Read / Execute Permission ausser Others
6: Richtig

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?

2: chmod <rwxr-x--x> <user1> : chmod 751 user1
4: chmod <r--r--r--><public> : chmod 777 public
5: chmod <rwxrwx---><management> : chmod 770 management

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`).

Only Read Permission und kein Output Permission! Beide Versuche sind fehlgeschlagen! Beim Zweiten Output Permission aber wegen Sticky bit nur user darf löschen!

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.

Alle User aus den beiden Gruppen sollten zusätzlich eine neue Gruppe zugewiesen werden! -> Für das Management ist die neue Gruppe nicht ersichtlich, warum sie existiert! Die Gruppe hat keine Funktion und ist nur ein Platzhalter!

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?

Man kann nur eine Permission geben, die spezifiziert ist!

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.

The support for POSIX ACLs is generally built into the file system, therefore no special configuration is needed to use it.

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 would get more permissions than you actually should have.

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:

- **`sudo su`**
- **`cd 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?

Admin hat alle Permissions.
Management hat nur Read / Execute Permissions.

Alle Anderen können nichts machen!

```
┌──(kali㉿kali)-[~/…/ITS/Lab7_authorization/authorization/DAC]
└─$ ls -dl meetings
drwxr-x--- 2 root management 4096 May 13 05:00 meetings

┌──(kali㉿kali)-[~/…/ITS/Lab7_authorization/authorization/DAC]
└─$ getfacl meetings
# file: meetings
# owner: root
# group: management
user::rwx
group::r-x
other::---
```

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[illegible]ree (!) new output lines:

1: group:management:r-x
2: Group:development:rwx
3: mask::rwx Wieso 2x?

```
└─$ getfacl meetings
# file: meetings
# owner: root
# group: management
user::rwx
group::r-x
group:management:r-x
group:development:rwx
mask::rwx
other::---
```

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.

Permission Denied

In earlier versions of the ACLs you could disable the acl option on the mounted file system. This is no longer possible, since it would lead to users gaining more access rights than they should have.

After enabling and disabling ACLs, the management group would have 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.

3 Part 2: 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.

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

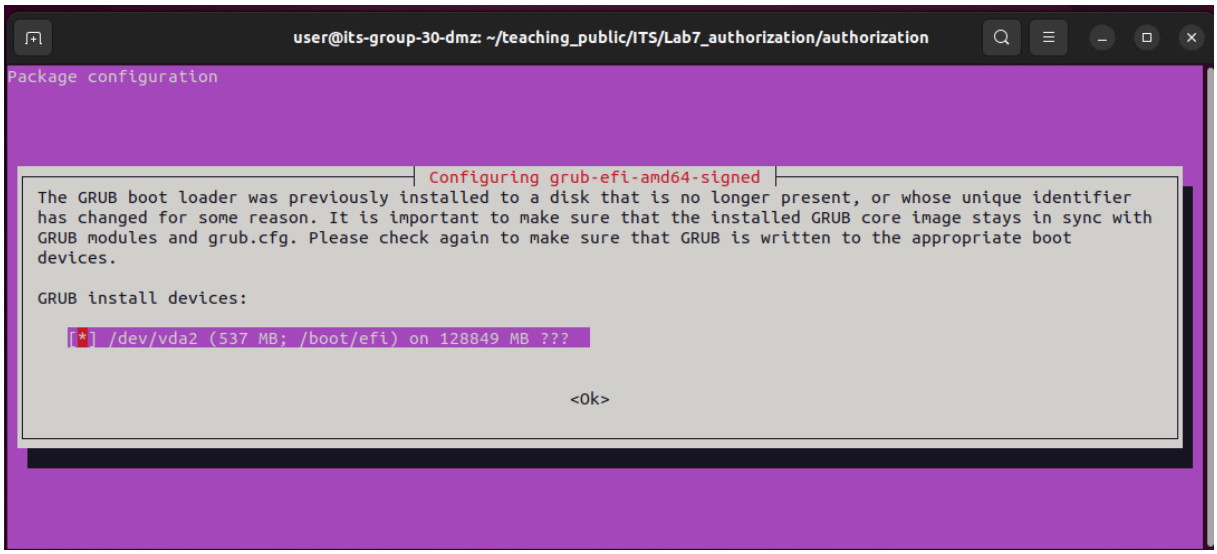
Your task is now to implement the above policy for the directory structure shown above.

Open a console and become *root*

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

- `cd ..`
- `sudo apt-get install libuser`

you will get an information message in the form of the following figure. Confirm it with <Enter>



- **chmod +x fileserver.sh**
- **sudo ./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.

```
./fileserver.sh clean
```

The file system of the file server is then "mounted" in *./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:

- **sudo apt-get install libswitch-perl**
- **chmod +x checkPolicy.pl**
- **./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 (see image below), 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
```