Handout 4 (Access Control)

Access control is essentially about deciding whether to grant access to a resource or deny it. Sounds easy. No? Well it turns out that things are not as simple as they seem at first glance. Let us first look, as a case-study, at how access control is organised in Unix-like systems (Windows systems have similar access controls, although the details might be quite different).

Unix-Style Access Control

Following the Unix-philosophy that everything is considered as a file, even memory, ports and so on, access control in Unix is organised around 11 Bits that specify how a file can be accessed. These Bits are sometimes called the *permission attributes* of a file. There are typically three modes for access: **r**ead, **w**rite and e**x**ecute. Moreover there are three user groups to which the modes apply: the owner of the file, the group the file is associated with and everybody else. This relatively fine granularity seems to cover many useful scenarios of access control. A typical example of some files with permission attributes is as follows:

```
1 $ ls -ld . * */*
2 drwxr-xr-x ping staff  32768 Apr  2 2010 .
3 -rw----r-- ping students  31359 Jul 24 2011 manual.txt
4 -r--rw--w- bob  students   4359 Jul 24 2011 report.txt
5 -rwsr--r-x bob  students 141359 Jun  1 2013 microedit
6 dr--r-xr-x bob  staff     32768 Jul 23 2011 src
7 -rw-r--r-- bob  staff     81359 Feb 28 2012 src/code.c
8 -r--rw---- emma students    959 Jan 23 2012 src/code.h
```

The leading `d` in Lines 2 and 6 indicate that the file is a directory, whereby in the Unix-tradition the `.` points to the directory itself. The `..` points at the directory "above", or parent directory. The second to fourth letter specify how the owner of the file can access the file. For example Line 3 states that `ping` can read and write `manual.txt`, but cannot execute it. The next three letters specify how the group members of the file can access the file. In Line 4, for example, all students can read and write the file `report.txt`. Finally the last three letters specify how everybody else can access a file. This should all be relatively familiar and straightforward. No?

There are already some special rules for directories and links. If the execute attribute of a directory is *not* set, then one cannot change into the directory and one cannot access any file inside it. If the write attribute is *not* set, then one can change existing files (provide they are changeable), but one cannot create new files. If the read attribute is *not* set, one cannot search inside the directory (`ls -la` does not work) but one can access an existing file, provided one knows its name. Links to files never depend on the permission of the link, but the file they are pointing to. Otherwise one could easily change access rights to files.

While the above might sound already moderately complicated, the real complications with Unix-style file permissions involve the setuid and setgid attributes. For example the file `microedit` in Line 5 has the setuid attribute set (indicated by the s in place of the usual x). The purpose of setuid and setgid is to solve the following puzzle: The program passwd allows users to change their passwords. Therefore passwd needs to have write access to the file /etc/passwd. But this file cannot be writable for every user, otherwise anyone can set anyone else's password. So changing securely passwords cannot be achieved with the simple Unix access rights discussed so far. While this situation might look like an anomaly, it is in fact an often occurring problem. For example looking at current active processes with /bin/ps requires access to internal data structures of the operating system, which only root should be allowed to. In fact any of the following actions cannot be configured for single users, but need privileged root access

- changing system databases (users, groups, routing tables and so on)
- opening a network port below 1024
- interacting with peripheral hardware, such as printers, harddisk etc
- overwriting operating system facilities, like process scheduling and memory management

This will typically involve quite a lot of programs on a Unix system. I counted 90 programs with the setuid attribute set on my bog-standard Mac OSX system (including the program /usr/bin/login for example). The problem is that if there is a security problem with only one of them, be it a buffer overflow for example, then malicious users can gain root access (and for outside attackers it is much easier to take over a system). Unfortunately it is rather easy to cause a security problem since the handling of elevating and dropping access rights in such programs rests entirely with the programmer.

The fundamental idea behind the setuid attribute is that a file will be able to run not with the callers access rights, but with the rights of the owner of the file. So /usr/bin/login will always be running with root access rights, no matter who invokes this program. The problem is that this entails a rather complicated semantics of what the identity of a process (that runs the program) is. One would hope there is only one such ID, but in fact Unix distinguishes three(!):

- *real identity*
  This is the ID of the user who creates the process; can only be changed to something else by root.
- *effective identity*
  This is the ID that is used to grant or deny access to a resource; can be changed to either the real identity or saved identity by users, can be changed to anything by root.
- *saved identity*
  If the setuid bit set in a file then the process is started with the real identity of the user who started the program, and the identity of the owner of the

program as effective and saved identity. If the setuid bit is not set, then the saved identity will be the real identity.

As an example consider again the `passwd` program. When started by, say the user `foo`, it has at the beginning the identities:

- *real identity*: `foo`
  effective identity: `foo`
  saved identity: `root`

It is then allowed to change the effective identity to the saved identity to have

- *real identity*: `foo`
  effective identity: `root`
  saved identity: `root`

It can now read and write the file `/etc/passwd`. After finishing the job it is supposed to drop the effective identity back to `foo`. This is the responsibility of the programmers who wrote `passwd`. Notice that the effective identity is not automatically elevated to `root`, but the program itself must make this change. After it has done the work, the effective identity should go back to the real identity.

Despite this complicated semantics, Unix-style access control is of no use in a number of situations. For example it cannot be used to exclude some subset of people, but otherwise have files readable by everybody else (say you want to restrict access to a file such that your office mates cannot access a file). You could try setting the group of the file to this subset and then restrict access accordingly. But this does not help, because users can drop membership in groups. If one needs such fine-grained control over who can access a file, one needs more powerful *mandatory access controls* as described next.

Secrecy and Integrity

Often you need to keep information secret within a system or organisation, or secret to the "outside world". An example would be to keep information secret such that insiders cannot leak information to competitors. A very good instance of such an access control system is the secrecy levels used in the military. There you distinguish four secrecy levels:

- top secret
- secret
- confidential
- unclassified

The idea is that the secrets classified as top-secret are most closely guarded and only accessible to people who have a special clearance. The unclassified category is the lowest level not needing any clearance. While the idea behind these security levels is quite straightforward, there are some interesting implications for when you want to realise such a system. To begin the access control needs to be *mandatory* as opposed to *discretionary*. With discretionary access control, the users can decide how to restrict or grant access to resources. With mandatory access control, the access to resources is enforced "system-wide" and cannot be controlled by the user. There are also some interesting rules for reading and writing an object that need to be enforced:

- **Read Rule**: a principal *P* can read an object *O* provided *P*'s security level is at least as high as *O*'s
- **Write Rule**: a principal *P* can write an object *O* provided *O*'s security level is at least as high as *P*'s

The first rule says that a principal with secret clearance can read secret documents or lower, but not documents classified top-secret. The second rule for writing needs to be the other way around: someone with secret clearance can write secret or top-secret documents—no information is leaked. In contrast it cannot write confidential documents, because then information can be leaked to lower levels. These rules about enforcing secrecy with mult-level clearances is often called *Bell/LaPudela* model, named after two people who studied such systems.

A problem with this access control system is when two people want to talk to each other but having different security clearances, say secret and confidential.

While secrecy is one property you often want to enforce, integrity is another. This property ensures that no

Further Information

If you want to know more about the intricacies of the "simple" Unix access control system you might find the relatively readable paper about "Setuid Demystified" useful.

http://www.cs.berkeley.edu/~daw/papers/setuid-usenix02.pdf