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). Then we have a look at how secrecy and integrity can be ensured in a system, and finally have a look at shared access control in multi-agent systems. But before we start, let us motivate access control systems by the kind of attacks we have seen in the last lecture.

There are two further general approaches for countering buffer overflow attacks (and other similar attacks). One are Unix-like access controls, which enable a particular architecture for network applications, for example web-servers. This architecture minimises the attack surface that is visible from, for example, the Internet. And if an attack occurs the architecture attempts to limit the damage. The other approach is to *radically* minimise the attack surface by running only the bare essentials on the web-server. In this approach, even the operating system is eliminated. This approach is called *unikernel*.

A *unikernel* is essentially a single, fixed purpose program running on a server. Nothing else is running on the server, except potentially many instances of this single program are run concurrently with the help of a hypervisor.[1] This single program implements the functionality the server offers (for example serving web-pages). The main point is that all the services the operating system normally provides (network stack, file system, ssh and so on) are not used by default in unikernels. Instead, the single program uses libraries (the unikernel) whenever some essential functionality is needed. The developer only needs to select a minimal set of these libraries in order to implement a server for web-pages, for example. In this way, ssh, say, is only provided, when it is absolutely necessary.

Unikernels are a rather recent idea for hardening servers. I have not seen any production use of this idea, but there are plenty of examples from academia. The advantage of unikernels is the rather small footprint in terms of memory, booting times and so on (no big operating system is needed). This allows unikernels to run on low-coast hardware such as Raspberry Pi's or Cubieboards, where they can replace much more expensive hardware for the same purpose. The low booting times of unikernels are also an advantage when your server needs to scale up to higher user-demands. Then it is often possible to just run another instance of the single program, which can be started almost instantly without the user seeing any delay (unlike if you have to start, say, Windows and then on top of that start your network application). One of the most well-known examples of a unikernel is MirageOS available from

© Christian Urban, King's College London, 2014, 2015

[1]Xen is a popular hypervisor; it provides the mechanism of several virtual machines on a single computer.

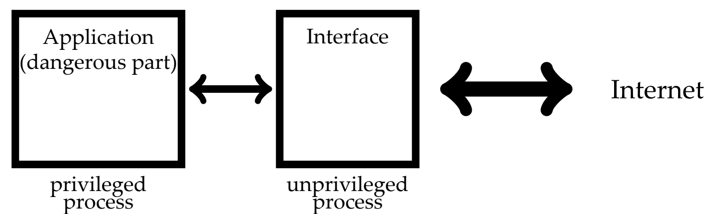
https://mirage.io

This unikernel is based on the functional programming language Ocaml, which provides added security (Ocaml does not allow buffer overflow attacks, for example). If you want to test the security of MirageOS, the developers issued a Bitcoin challenge: if you can break into their system at

http://ownme.ipredator.se

you can get 10 Bitcoins. This is approximately £41,000.

However, sometimes you cannot, or do not want to, get rid of the operating system. In such cases it is still a good idea to minimise the attack surface. For this it helps if the network application can be split into two parts—an application and an interface:



The idea is that all heavy-duty lifting, or dangerous operations, in the application (for example database access or writing a file) is done by a privileged process. All user input from the internet is received by an *un*privileged process, which is restricted to only receive user input from the Internet and communicates with the privileged process. This communication, however, needs to be sanitised, meaning any unexpected user-input needs to be rejected. The idea behind this split is that if an attacker can take control of the *un*privileged process, then he or she cannot do much damage. However, the split into such privileged and unprivileged process requires an operating system that supports Unix-style access controls, which we will look at next.

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. A typical permission of a file owned by `bob` being in the group `staff` might look as follows:

```
- r-- rw- rwx bob staff
```

directory user group other

For the moment let us ignore the directory bit. The Unix access rules imply that Bob will only have read access to this file, even if he is in the group `staff` and this group's access permissions allow read and write. Similarly every member in the `staff` group who is not `bob`, will only have read-write access permissions, not read-write-execute.

This relatively fine granularity of owner, group, everybody else 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 (provided 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 sys-

tem, which only root should have access 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`). 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.

If you want to play more with access rights in Unix, you can use the program in Figure 1. It explicitly checks for readability and writability of files. The `main` function is organised into two parts: the first checks readability and writability with the permissions according to a potential setuid bit, and the second (starting in Line 34) when the permissions are lowered to the caller. Note that this program has one problem as well: it only gives a reliable answer in cases a file is **not** readable or **not** writable when it returns an error code 13 (permission denied). It sometimes claims a file is not writable, say, but with an error code 26 (text file busy). This is unrelated to the permissions of the file.

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 from the "outside world". An example would be to keep insiders from leaking information to competitors. The secrecy levels used in the military are an instance of such an access control system. There you distinguish usually 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

```c
#include <stdio.h>
#include <unistd.h>
#include <errno.h>

FILE *f; //file pointer

//tests return errno = 13 for permission denied
void read_test(char *name)
{
    if ((f = fopen(name, "r")) == NULL) {
      printf("%s is not readable, errno = %d\n", name, errno);
    } else {
      printf("%s is readable\n", name); fclose(f);
    }
}

void write_test(char *name)
{
    if ((f = fopen(name, "r+")) == NULL) {
      printf("%s is not writable, errno = %d\n", name, errno);
    } else {
      printf("%s is writable\n", name); fclose(f);
    }
}

int main(int argc, char *argv[])
{
    printf("Real UID = %d\n", getuid());
    printf("Effective UID = %d\n", geteuid());

    read_test(argv[1]);
    write_test(argv[1]);

    //lowering the access rights to the caller
    if (setuid(getuid())) {
      printf("could not reset setuid, errno = %d\n", errno);
      return 1;
    }

    printf("Real UID = %d\n", getuid());
    printf("Effective UID = %d\n", geteuid());

    read_test(argv[1]);
    write_test(argv[1]);

    return 0;
}
```

Figure 1: A read/write test program in C. It returns errno = 13 in cases when permission is denied.

category is the lowest level not needing any clearance. While the idea behind these security levels is quite straightforward, there are some interesting phenomenons that you need to think about when realising such a system. First this kind of 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 would be no point to let users set the secrecy level, because if they want to leak information they would set it to the lowest. Even if there is no malicious intent, it could happen that somebody by accident sets the secrecy level too low for a document. Note also that the secrecy levels are in tension with the Unix-style access controls. There root is allowed to do everything, but in a system enforcing secrecy, you might not like to give root such powers.

There are also some interesting rules for reading and writing a resource that need to be enforced:

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

The first rule implies 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 these cases. In contrast the principal cannot write confidential documents, because then information can be leaked to lower levels. These rules about enforcing secrecy with multi-level clearances are often called *Bell/LaPadula* model, named after two people who studied such systems.

A problem with this kind of access control system is when two people want to talk to each other but are assigned different security clearances, say secret and confidential. In these situations, the people with the higher clearance have to lower their security level and are not allowed to take any document from the higher level with them to the lower level (otherwise information could be leaked). In actual systems, this might mean that people need to log out and log into the system again—this time with credentials for the lower level.

While secrecy is one property you often want to enforce, integrity is another. This property ensures that nobody without adequate clearance can change, or tamper with, systems. An example for this property is a *fire-wall*, which isolates a local system from threads from the Internet, for example. The rule for such a system is that somebody from inside the fire-wall can write resources outside the firewall, but you cannot write a resource inside the fire-wall from outside. Otherwise an outside can just tamper with a system in order to break in. In contrast we can read resources from inside the fire-wall, for example web-pages. But we cannot read anything from outside the fire-wall. Lest we might intro-

duce a virus into the system (behind the fire-wall). In effect in order to ensure integrity the read and write rules are reversed from the case of secrecy:

- **Read Rule**: a principal P can read a resource O provided P's security level is lower or equal than O's
- **Write Rule**: a principal P can write a resource O provided O's security level is lower or equal than P's

This kind of access control system is called *Biba* model, named after Kenneth Biba. Its purpose is to prevent data modification by unauthorised principals.

The somewhat paradoxical result of the different reading and writing rules in the *Bell/LaPadula* and *Biba* models is that we cannot have secrecy and integrity at the same time in a system, or they need to be enforced by different means.

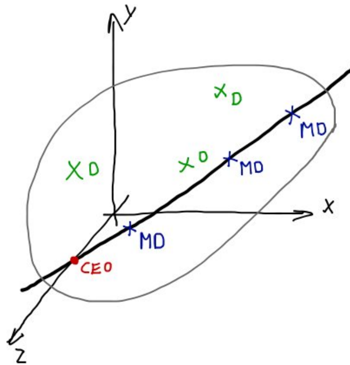
Multi-Agent Access Control

In military or banking, for example, very critical decisions need to be made using a *two-people rule*. This means such decisions need to be taken by two people together, so that no single person can defraud a bank or start a nuclear war (you will know what I mean if you have seen the classic movie "Dr Strangelove or: How I Learned to Stop Worrying and Love the Bomb"[2]). Translating the two-people rule into a software system seems not as straightforward as one might think.

Let us assume we want to implement a system where CEOs can make decisions on their own, for example whether or not to sell assets, but two managing directors (MDs) need to come together to make the same decision. If "lowly" directors (Ds) want to take this decision, three need to come together. Remember cryptographic keys are just sequences of bits. A naive solution to the problem above is to split the necessary key into n parts according to the "level" where the decision is taken. For example one complete key for a CEO, halves of the key for the MDs and thirds for the Ds. The problem with this kind of sharing a key is that there might be many hundreds MDs and Ds in your organisations. Simple-minded halving or division by three of the key just does not work.

A much more clever solution was proposed by Blakley and Shamir in 1979. This solution is inspired by some simple geometric laws. Suppose a three-dimensional axis system. We can, clearly, specify a point on the z-axis, say, by specifying its coordinates. But we could equally specify this point by a line that intersects the z-axis in this point. How can a line be specified? Well, by giving two points in space. But as you might remember from school days, we can specify the point also by a plane intersecting the z-axis and a plane can be specified by three points in space. This could be pictured as follows:

[2] http://en.wikipedia.org/wiki/Dr._Strangelove



The idea is to use the points as keys for each level of shared access. The CEO gets the point directly. The MDs get keys lying on a line and the Ds get keys lying on the plane. Clever, no? Scaling this idea to more dimensions allows for even more levels of access control and more interesting access rules, like one MD and 2 Ds can take a decision together.

Is such a shared access control used in practice? Well military command-chains are obviously organised like this. But in software systems often need to rely on data that might not be entirely accurate. So the CEO-level would correspond to the in-house data-source that you can trust completely. The MD-level would correspond to simple errors where you need three inputs and you decide on what to do next according to what at least two data-sources agree (the third source is then disregarded, because it is assumed it contains an error). If your data contains not just simple errors, you need levels corresponding to Ds.

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`

About secrecy and integrity, and shared access control I recommend to read the chapters on "Nuclear Command and Control" and "Multi-Level Security" in Ross Anderson's Security Engineering book (whose second edition is free).