Operating System Security

Hans-Wolfgang Loidl
http://www.macs.hw.ac.uk/~hwloidl

School of Mathematical and Computer Sciences
Heriot-Watt University, Edinburgh

The Role of an Operating System

- An Operating System (OS) provides the interface between the user and the hardware.
- The OS **abstracts** over low-level system-specifics (e.g. physical memory size, by providing virtual memory).
- The OS manages the available resources and shares them among several users (e.g. multi-tasking).
- The OS can be seen as the **glue** linking application programs and the system.
- One current trend in OSs is **virtualisation** and more generally **Cloud computing**.

Operating Systems and Security

- Operating Systems are very complex.
- Dealing with complexity is one of the main challenges in computer security.
- Therefore, OS security is not only of interest in itself, but can also be seen as a case study of how to design security principles and mechanisms for complex systems.
- Many of these principles also apply to bespoke applications, that need to manage several classes of users, share resources among the users etc.

Scope of this Part of the Course

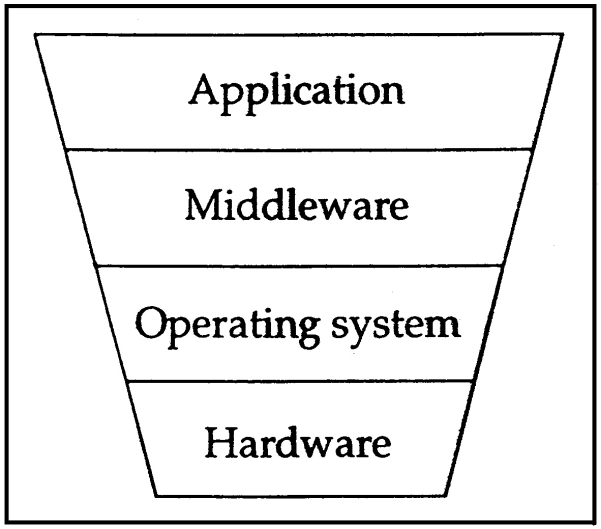
We will study,

- how operating systems work, in principle;
- how they can be attacked;
- how they can be protected.

Multi-user Systems

- Early operating systems, such as DOS, were **single-user systems**
- All modern operating systems support multiple-users
- In this setup confidentially is highly desirable, preventing the unauthorized reading of data
- To ensure **confidentially**, a mechanism for securing file access is needed
- The most common approach to ensure confidentiality is to classify users by “roles” and implement role-based access control
- More generally, confidentiality presumes a security (access) policy saying who or what can access our data.
- The **security policy** is used for access control.

Access controls at different levels in a system



[0]Figure from “Security Engineering” by Ross Anderson, Chapters 4, 6

- The OS connects the application-layer with the hardware-layer.
- Additionally, a middleware-layer might provide a common interface to frequently used abstractions outside the OS (e.g. database access).
- The OS itself is structured into
  - the **OS kernel**, which has direct, exclusive access to low-level resources (e.g. device drivers);
  - the **OS applications**, which perform non-essential operations inside the kernel.
- Because kernel code has direct access to hardware, its security requirements are highest.
- An indication of this is the fact that program verification techniques are often applied to device driver code.
- Another important security principle is to minimise the code in the OS kernel.

Interacting with the Operating System

- An application interacts with the OS by performing **system calls**.
- The kernel provides a library of functions that directly interact with the hardware.
- An example of such a library is `libc` with functions for opening/closing files etc.
- Often a system call is implemented by an interrupt, which stops the normal flow of computation, and starts code in kernel mode.
- The usage of such system calls in an application is crucial for the security of the entire application.

Processes

- A **process** is an execution environment of a program, with its own set of resources.
- Such resources are memory area, file handles etc.
- Several processes may share available resources, e.g. writing to the same file.
- The kernel is in charge of assigning physical resources to processes, e.g. which memory area to use for each process.
- The kernel uses time-slicing to seemingly run several processes at the same time: each process is executed for a short period of time (time slice); then a context switch is performed and another process is executed.
- Details on how to perform time-slicing and resource allocation depend on the concrete usage of the system: a server system may behave quite differently from a desktop system.

Processes and Security

- The OS must ensure that different **processes do not interfere** in a harmful way.
- Each process is owned by one user (user ID), determining its access to shared resources.
- Additionally, the process may temporarily assume the identity of another user (effective user ID) for example to gain access to restricted resources.
- The handling of such change in effective user IDs is one common pitfall and multi-user systems.
- A process can **fork** a sub-process, which inherits the permissions from its parent.
- This generates a process tree, which can be shown using the command `pstree`.

Inter-Process Communication

- If several processes share resources, they often need to communicate with each other.
- This is usually done through **inter-process communication (IPC)**.
- IPC can be realised by reading from/writing to files. However,
  - this is usually not very efficient;
  - prone to attacks, if other users can delete files while the application is running.
- A more efficient way of IPC is through shared memory, managed by the kernel.
- Another mechanism that is commonly used are pipes and sockets: these act as tunnels between processes, using send and receive to exchange data.

Other OS concepts

- **Signals** are asynchronous messages used to communicate between processes.
- One specific signal (`INT` signal, number 9), kills one running processes.
- When receiving a signal, the OS interrupts the process and performs the associated action.
- **Remote Procedure Calls (RPCs)** provide a way to execute system code within another processes or even on a remote machine.
- Typically an OS uses **daemons** to provide services that should be continually available (e.g. print server).
- They are typically forked when booting the machine, and run with higher privileges than other processes.

Memory Management

- One important task of an OS is to ensure that processes do not interfere.
- One aspect of this task is to ensure that separate processes work on separate memory areas (**memory management**).
- During runtime the OS also has to decide which logical memory areas should be kept in the physical memory, and which areas can be kept on disk.
- The OS uses a specific **paging policy** to make these decisions.
- Typically, the most recently used (youngest) processes will be kept in memory, whereas older processes can be swapped to disk.

Memory Structure

In most Unix-based systems the memory area associated to one process, i.e. its **address space**, is structured into 5 segments:

- **Text**, containing the machine code of the program.
- **Data**, containing **static** program variables that have been initialised.
- **BSS**, containing **static** program variables that have **not** been initialised.
- **Heap**, containing **dynamically** allocated memory.
- **Stack**, managing the function/method call structure during the execution.

Each of these areas has its own set of permissions (readable, writable, executable) and is owned by the user id that started the process.

Virtual Memory

- Modern OSs abstract over the amount of physical memory, giving the illusion of a huge amount of available memory.
- In order to manage access to such **virtual memory**, the system partitions the memory into **pages**.
- If a process tries to access data on a page that is not in physical memory a **page fault** occurs.
- In this case the OS kicks in and transfers the page from disk into physical memory.
- Additionally, with virtual memory blocks of data can be treated contiguously, although they are physically split into chunks.
- A basic understanding of these mechanisms is needed to understand system-level attacks such as **buffer overflow** and **stack thrashing** attacks.

Virtual Machines

A current trend in operating systems is to use **virtual machines** in order to encapsulate a certain set of services.

- A **virtual machine (VM)** is a container that behaves like an independent OS.
- VMs are a generalisation of the separation of the address space between processes: VMs separate all resources.
- Therefore, one VM behaves like a completely separate machine, running its own OS.
- To implement such separation a **hypervisor or virtual machine monitor** has to be implemented, which handles all resource and hardware access.
- The hypervisor can implement hardware access in two different ways:
  - Using **emulation** it **converts** access to hardware of different types.
  - Using **virtualisation** it **transfers** access from the virtual to the physical hardware layer.
- A common use of VMs is to run a different OS, e.g. to run Windows on a Linux system.

Advantages of Virtualisation

- **Hardware efficiency:** Several VMs can be run on the same (powerful) server machine.
- **Portability:** The entire state of the guest operating system can be saved an run on the concrete host machine.
- **Security:** The VM acts as a sandbox, isolating potentially harmful code from the host systems and the actual hardware.
- **Management Convenience:** It is easy to take system snapshot and to restore a system to a previous, save state.

Virtualisation Technologies

- **VMware** `http://www.vmware.com/products/desktop_virtualization/`
  Commercial. Market leader on all kinds of virtualisation. Basic package (vmplayer) is free. Supports Windows, Linux (not sure about Macs). Probably highest quality of services.
- **Parallels:** `http://www.parallels.com/`
  Commercial. Fairly expensive (reduced rates might apply for students) Supports Macs, Windows, Linux. Requires hardware virtualization support. Very good performance especially on Macs. Supported Linux version seem quite dated.
- **VirtualBox:** `https://www.virtualbox.org/`
  OpenSource, GPL. Community effort in building a virtualisation platform. VirtualBox runs on Windows, Linux, Macintosh, and Solaris hosts VirtualBox 4.1.6 was released today Not sure how it compares with above two in terms of services and perf.

Virtualisation Technologies (cont'd)

- **Xen:** `http://xen.org/`
  Supports Linux, Solaris, BSD-variants and Windows. Probably the most stable of the Linux-based packages
- **KVM:** `http://www.linux-kvm.org/page/Main_Page`
  OpenSource Runs on Linux, using its kernel-support for VMs. Performs hardware emulation, so likely to be slower than other packages.
- **LinuX Containers (LXC):** `http://lxc.sourceforge.net/`
  Light-weight virtualisation support by the Linux kernel with a fairly minimal set of tools, shipped with most recent distros. Good for wrapping up separated services in an VM. Uses command-line tools for configuration (no shiny GUI interface): lxc-execute, lxc-create and the like.
  There are good HOWTOs online, some are distro specific.
  `http://www.ibm.com/developerworks/linux/library/l-lxc-containers/`

File System

- Modern OSs organise files into a tree of **folders**.
- **File permissions** are used to control access to these files.
- This is one example of using role-based access control to shared resources.
- We now look into this concept in more general terms.

User Authentication

- As a prerequisite for securing resources, the system needs to **authenticate users**
- This tackles the question, *How does the operating system securely identify its users?*
- The most common technique are passwords, i.e. secrets only known to that user.
- Alternatives are biometric data (e.g. fingerprints), or physical devices (e.g. smartcards).

Password-based Authentication

When using passwords to authenticate users, several security-relevant issues need to be taken care of:

- The password must be stored securely, e.g. in an encrypted form.
- Typically systems use a one-way hash function to store the passwords.
- To prevent dictionary attacks on weak passwords, the input is often **salted** before applying the hash function, i.e. a random bit pattern is added to the user id, and both, the salted user id and the password are hashed.
- The system has to store the salt in an encrypted way. Note that this encryption is completely under system control and therefore not susceptible to weak passwords.
- Most Unix systems use either salted MD5 or a DES variant as hash function. Often the sysadmin can choose a different algorithm from a list of safe choices (e.g. blowfish).

Access Control

Once a user has been authenticated,

- the next question is *How does the operating system determine what users have permission to do?*
- This is called the **Access Control Problem**.
- *The purpose of access control is to control which principals (persons, processes, machines, etc) have access to which resources in the system, e.g. which files they can read, which programs they can execute, how they share data with other principals, and so on.*
- This involves
  - **Authenticate** principals, e.g. through passwords,
  - **mediate** access to system resources, e.g. by using roles.
- We now look into the mediation part.

Access Control Lists

Access to system resources is usually managed by an access control matrix, e.g.

	Operating System	Accounts Program	Accounting Data	Audit Trail
Sam	rwx	rwx	rw	r
Alice	x	x	rw	–
Bob	rx	r	r	r

Note that this matrix needs an entry for each user and therefore doesn't scale well.

[0] Figure from "Security Engineering" by Ross Anderson, Chapters 4, 6

Roles

- "Roles" classify the potentially huge set of users, into a small set of groups.
- All members in one group use the same protection mechanism for the available resources.
- Therefore the access control matrix only needs one row per role.

Access Control Lists

- Access control lists (ACLs) are the columns in the access control matrix, stored separately with the resource they are controlling.
- ACLs are widely used in environments where users manage their own file security, such as the Unix systems
- Where the access control policy is set centrally, they are suited to environments where protection is data- oriented
- ACLs are very flexible (control access for each user) but not very efficient (the OS has to check the entire list on resource access).

Unix Operating System Security

- Attributes: read, write, execute
- Roles: owner, group, others
- Example ACL: `-rw-r--- Alice Accounts`
- This records that the file is not a directory; the file **owner** can read and write it; **group** members can read it but not write it; nongroup members (**others**) have no access at all;
- The file owner is Alice; and the group is Accounts.
- Note that the user trying to read the file, also needs execute permissions on all directories on the path to this file.
- Note that `root` always has full privileges
- Thus, an intruder, who gains root access, can delete all traces of his changes afterwards.
- FreeBSD addresses this problem: Files can be set to be append-only, immutable or undeletable for user, system or both

Unix Operating System Security

- Additionally, the owner can define extended attributes for his/her files.
- Some examples are to make a file append-only or immutable.
- The Unix commands `lsattr` and `chattr` are used to show and modify such settings.
- A complete list of possible attributes:
  - append only (a),
  - compressed (c),
  - no dump (d),
  - immutable (i),
  - data journalling (j),
  - secure deletion (s),
  - no tail-merging (t),
  - undeletable (u),
  - no atime updates (A),
  - synchronous directory updates (D),
  - synchronous updates (S),
  - top of directory hierarchy (T).

Optional ACL-based Permissions

- Recent Linux distributions extend this scheme of base permissions.
- **Named users** and **named groups** can be created.
- This lifts the restriction of associating only 1 user (owner) and 1 group to a file or folder.
- For each named user and named group, an own set of permissions can be defined.
- A **mask** defines the maximal permissions allowed for owner, named users and named groups
- The respective commands are: `getfacl` and `setfacl`

Optional ACL-based Permissions

- The output of `getfacl` for a directory looks like this:

```
1:  # file: somedir/
2:  # owner: lisa
3:  # group: staff
4:  user::rwx
5:  user:joe:rwx                #effective:r-x
6:  group::rwx                  #effective:r-x
7:  group:cool:r-x
8:  mask:r-x
9:  other:r-x
10: default:user::rwx
11: default:user:joe:rwx        #effective:r-x
12: default:group::r-x
13: default:mask:r-x
14: default:other:---
```


Security-enhanced Linux (SELinux)

SELinux develops the ACL principle even further:

- It uses **mandatory access control**, as opposed the discretionary access control discussed so far.
- Users are not allowed to change permission; this is delegated to the security policy administrator.
- Almost every operation is checked against a rule-base, defining the permissions.
- Each rule defines a subject (the process trying to perform the operation) and an object (the resource attempted to be accessed).
- In principle, a sequence of rules has to be checked on every resource access.
- SELinux operates on the principle of **least privileges:** each process receives the minimal set of permissions needed to perform its job.
- SELinux achieves the most refined resource control, but has a considerable performance impact.

Capabilities

- **Capabilities** are the **rows** in an access control matrix.
- Advantages: Runtime security checking is more efficient, and we can do delegation without much difficulty
- Disadvantages: changing a file's status can suddenly become more tricky, as it can be difficult to find out which users have access
- Recently capabilities are making a comeback in the form of public key certificates.

Granlularity of Access Control

- The smallest unit, for which access can be controlled (**granularity**), is a file.
- Thus, access control cannot be used to hide parts of file.
- Such functionality needs to be coded in the application.
- If a database is used as the data-storage back-end, it needs to use its own access control mechanism.
- This generates layers of access control and complicated security management.

Hardware Protection

- Hardware protection prevents processes from interfering with each other.
- This requires a mechanism that will stop one program from overwriting another's code or data.
- Such **hardware access control** must be integrated with the processor's memory management functions.
- A typical mechanism is **segment addressing:**
  - an address consist of a segment address and an offset into the segment
  - the segment address is controlled by the operating system
  - only the owning process of a segment can write to it

Dynamically Changing Attributes: `setuid`

- Sometimes we want to specify that a file can only be modified by a certain program.
- Thus, we want to control access on a per-program, rather than a per-user basis.
- We can achieve this by creating a new user, representing the role of a modifier for these files.
- Mark the program, as `setuid` to this user.
- This means, no matter who started the program, it will run under the user id of this new user.
- Example:

User	Operating System	Accounts Program	Accounting Data	Audit Trail
Sam	rwx	rwx	r	r
Alice	rx	x	–	–
Accounts program	rx	r	rw	w
Bob	rx	r	r	r

- **Beware:** `setuid` programs are a major security pitfall!

[0] Figure from "Security Engineering" by Ross Anderson, Chapters 4, 6

Example code for setuid

```
static uid_t euid, uid;
int main(int argc, char * argvp[]) {
  FILE *file;
  /* Store real and effective user IDs */
  uid = getuid();  euid = geteuid();
  /* Drop privileges  */
  seteuid(uid);
  /* Do something useful ...  */
  /* Raise privileges, in order to access the file */
  seteuid(euid);
  /* Open the file; NB: this is owned and readable only by a different user *
  file = fopen("/tmp/logfile", "a");
  /* Drop privileges again */
  seteuid(uid);
  /* Write to the file */
  if (file) {
    fprintf(file, "Someone used this program: UID=%d, EUID=%d\n", getuid(), g
  } else {
    fprintf(stderr, "Could not open file /tmp/logfile; aborting ...\n");
    return 1;
  }
  fclose(file); return 0; }
```


Testing this prgram

As normal user do the following:

```
# do everything in an open directory
> cd /tmp
# download the source code
>  wget http://www.macs.hw.ac.uk/~hwloidl/Courses/F21CN/Labs/OSsec/setuid1.c
# compile the program
> gcc -o s1 setuid1.c
# change permissions so that everyone can execute it
> chmod a+x s1
# check the permissions
> ls -lad s1
-rwxrwxr-x 1 hwloidl hwloidl 10046 2011-11-11 22:06 s1
# generate an empty logfile
> touch /tmp/logfile
# change permissions to make it read/writeable only by the owner!
> chmod go-rwx /tmp/logfile
# check the permissions
> ls -lad /tmp/logfile
-rw------- 1 hwloidl hwloidl 0 2011-11-11 22:06 /tmp/logfile
```


As guest user do the following

```
> cd /tmp
# try to run the program
> ./s1
Could not open file /tmp/logfile; aborting ...
# this failed, because guest doesn't have permission to write to logfile
```

As normal user do the following

```
# set the setuid bit
> chmod +s s1
> ls -lad s1
-rwsrwsr-x 1 hwloidl hwloidl 10046 2011-11-11 22:06 s1
```

Now, as guest you can run the program:

```
> ./s1
# now this succeeds, although the user still cannot read the file
> cat /tmp/logfile
cat: /tmp/logfile: Permission denied
```

But the normal user can read the file, eg:

```
> cat /tmp/logfile
Someone used this program: UID=1701, EUID=1701
Someone used this program: UID=12386, EUID=12386
```


Buffer Overflow Attacks

- Often low-level programs use fixed-size arrays (buffers) to store data.
- When copying into such buffers, the program has to check that it doesn't exceed the size of the buffer.
- There are no automatic bounds checks in low-level languages such as C.
- If no check is performed, the program would just overwrite the following data block.
- If the data beyond the bound is chosen to be malign, executable machine code, an attacker can gain control of the system in this way.

Example 1: Rsyslog

The following vulnerability in the `rsyslog` program was reported in Linux Magazin 12/11:

```
[...]
int i;  /* general index for parsing */
uchar bufParseTAG[CONF_TAG_MAXSIZE];
uchar bufParseHOSTNAME[CONF_HOSTNAME_MAXSIZE];
[...]
while(lenMsg > 0 && *p2parse != ':' && *p2parse != ' ' &&
      i < CONF_TAG_MAXSIZE) {
 bufParseTAG[i++] = *p2parse++;
 --lenMsg;
}
if(lenMsg > 0 && *p2parse == ':') {
 ++p2parse;
 --lenMsg;
 bufParseTAG[i++] = ':';
}
[...]
bufParseTAG[i] = '\0';  /* terminate string */
```


Example 2:

The following vulnerability in the `rsyslog` program was reported in Linux Magazin 12/11:

```
[...]
int i;  /* general index for parsing */
uchar bufParseTAG[CONF_TAG_MAXSIZE];
uchar bufParseHOSTNAME[CONF_HOSTNAME_MAXSIZE];
[...]
while(lenMsg > 0 && *p2parse != ':' && *p2parse != ' ' &&
      i < CONF_TAG_MAXSIZE) {
 bufParseTAG[i++] = *p2parse++;
 --lenMsg;
}
if(lenMsg > 0 && *p2parse == ':') {
 ++p2parse;
 --lenMsg;
 bufParseTAG[i++] = ':';
}
[...]
bufParseTAG[i] = '\0';  /* terminate string */
```


Discussion

- The goal of this code is to read tags and store them in a buffer.
- The program reads from a memory location `p2parse` and writes into the buffer `bufParseTAG`.
- The fixed size of the buffer is `CONF_TAG_MAXSIZE`
- The while-loop iterates over the input text, and also checks whether the index `i` is still within bounds.
- **BUT:** after the while loop, 1 or 2 characters are added to the buffer as termination characters; this can cause a buffer overflow!
- The impact of the overflow is system-specific. It can lead to overwriting the variable `i` on the stack.

Smashing the Stack

- One common form of exploiting a buffer overflow is to manipulate the stack.
- This can happen through unchecked copy operations into a local function variable or argument.
- This is dangerous, because local variables are kept on the stack, together with the return address for the function.
- Therefore, a buffer-overflow can directly **modify the control-flow** in the program.

Example of Smashing the Stack

Assume, we call this function:

```
int function() {
 int a;
 char b[5];
 char c[4];
 ...
}
```

The stack-layout for this function is:

```
c
b
a
...
return address
```

A buffer overflow of b can overwrite the contents of a, or maybe even the return address, which would change the control flow of the program.
Stack Guard and other security programs re-order the variables on the stack, and add variables at the end to detect overwrites.

Difficulties in exploiting the vulnerability

- The attacker needs to locate the position of the return address, and write the address of its own, malign code there.
- Several techniques can be used to achieve this.
- In a return-to-libc attack, the attacker overwrites the return address with a call to a known libc library function (eg. `system`).
- After this, the return address to the malign code and data for the arguments to the libc function is placed.
- This will cause a call to the libc function, followed by executing the malign code itself.

A Worst Case Scenario

A particularly dangerous combination of weaknesses is the following:

- A setuid function, raising privileges temporarily,
- which contains a buffer overflow vulnerability,
- and an attacker that plants shellcode as malign code onto the stack.
- If successful, the shellcode will give the attacker access to a full shell with the privileges used in that part of the application.
- If these are root privileges, the attacker can do anything he wants!

Prevention Mechanisms

- Canary variables, eg. on the stack, can detect overflows.
- Re-ordering variables on the stack can help to reduce the impact of a buffer overflow.
- Compiler modifications can change the pointer semantics, eg. never store a pointer directly, but only a version that needs to be XORed to get to the real address.
- Some operating systems allow to mark address blocks as non-executable.
- Address randomisation (re-arranging data at random in the address space) is frequently in modern operating systems to make it more difficult to predict where to find a return address or similar, attackable control-flow data.

Listing 2: imap/nntpd.c

Another attack mentioned in Linux Magazin 12/11 is this one:

```
do {
    if ((c = strrchr(str, ',')))
     *c++ = '\0';
 else
     c = str;

 if (!(n % 10)) /* alloc some more */
     wild = xrealloc(wild, (n + 11) * sizeof(struct wildmat));

 if (*c == '!') wild[n].not = 1;  /* not */
 else if (*c == '@') wild[n].not = -1; /* absolute not (feeding) */
 else wild[n].not = 0;

 strcpy(p, wild[n].not ? c + 1 : c);
 wild[n++].pat = xstrdup(pattern);
} while (c != str);
```


Listing 2: imap/nntpd.c

Another attack mentioned in Linux Magazin 12/11 is this one:

```
do {
    if ((c = strrchr(str, ',')))
     *c++ = '\0';
 else
     c = str;

 if (!(n % 10)) /* alloc some more */
     wild = xrealloc(wild, (n + 11) * sizeof(struct wildmat));

 if (*c == '!') wild[n].not = 1;  /* not */
 else if (*c == '@') wild[n].not = -1; /* absolute not (feeding) */
 else wild[n].not = 0;

 strcpy(p, wild[n].not ? c + 1 : c);
 wild[n++].pat = xstrdup(pattern);
} while (c != str);
```


Discussion

- This example is part of an IMAP server for emails.
- This code segment handles wildcards to perform operations.
- Its weakness is that it uses `strcpy` to copy a block of characters, which copies an **unbounded** 0-terminated block of memory.
- Instead, the function `strncpy` should be used, which takes the size of the block to copy as additional argument.

Race Conditions

- Another common bug are race conditions.
- Here, the behaviour of the application depends on the relative speed of 2 or more concurrent processes.
- These bugs are very difficult to find, because they depend on the relative speed.
- Therefore, program behaviour may be non-deterministic, i.e. for the same input the program behaves differently.

User Interface Failures

- Trojan Horses: programs that pretend to perform a well-known functionality, but gather sensitive information in the background (login, ls etc)
- Mandatory root privileges for installation (tailor your program to just the privileges that are needed **principle of least privilege**)
- Instead, create groups with limited power, and give appropriate permissions to the group.
- Changing privileges inside a program (`setuid`) should only be used as a last resort.

Design Failures

- Monolithic OS structure:
  - Operating systems tend to be huge (Linux kernel: 13,320,934 LoC as of May 2010; was: 176,250 LoC as of March 1994)
  - Programmers write bugs.
  - OS bugs might be catastrophic, because they may give an intruder un-privileged access.
  - **Solution:** micro-kernel structured OS, which minimises the trusted code base and implements all non-critical code in user space
- Unrevisited design decisions
  - Unix evolved out of a single-user, non-networked operating system (Multics)
  - Extensions for multiple users and networks were added on
  - But, assumptions made in the initial design often don't hold any more, and sources for security bugs

Remedies

- Buffer overflow: several online tools check for buffer overflow in C (Cyclone, CCured)
- Enforce the principle of least privilege
- The default configuration of an application should be safe, and not make implicit assumption on the OS configuration.

Secure Storage

The goal of secure storage is to prevent unauthorised access to data even in a situation where the adversary has physical access to a device, e.g. USB stick.

The concept that needs to be ensured is **confidentiality**.

The most common form of ensuring secure storage is to use **encrypted filesystems.**

File vs Filesystem Encryption

With tools such as `gpg` it is easy to encrypt individual files that contain confidential information.

An application such as an office suite, file browser or editor might transparently en/decrypt files to minimise the intrusion encryption has on the usual workflow.

There are several forms of file-encryption:

- All the above techniques do **user-selected, file-based encryption**: the user decides for each file whether it should be encrypted (rare case)
- **Per-file filesystem encryption** (or filesystem stacked level encryption) integrates the encryption into the code for reading/writing from/to filesystem.
- **Full-disk filesystem encryption** (or block device level encryption) encrypts an entire partition (logical disk).

Application-integrated file encryption

Encryption of files can be integrated into another application:

- Examples of integrated file encryption are Microsoft Office and Adobe Acrobat.
- Both use AES as block cipher.
- Microsoft Office 2007 derives a secret key from the user's password, by iterating SHA-1 hashing on the password 50,000 times.
- This does **not** increase cryptographic security, but is designed to slow down a brute force attack of guessing passwords.
- Adobe Acrobat 9 uses SHA-256, which stronger than SHA-1, and hashes the password only once.
- It has been reported, that brute-force password attacks on Adobe Acrobat 9 can test 10,000 more passwords per seconds than on Microsoft Office 2007.

Filesystem Encryption: EFS

Encrypting File System (EFS) is a per-file file-system encryption scheme available for Windows.

- EFS transparently encrypts all files, performing automatic encryption and decryption when the user opens, saves, and closes a file.
- Files and folders must be explicitly tagged as being encrypted to enable this functionality.
- For such tagged files and folders, the data on disk is never represented in plain text.
- EFS uses a combination of symmetric and asymmetric encryption.
- The contents of files is encrypted with a **filesystem encryption key (FEK)**, using AES as symmetric block cipher.
- The FEK is encrypted with the users **public-key**, using asymmetric encryption, and stored in the files meta-data.
- Decrypting the file involves first decrypting the FEK with the user's private key, and then decrypting the file contents.

Filesystem Encryption: EFS (cont'd)

Encrypting File System (EFS) is a per-file file-system encryption scheme available for Windows.

- To enable sharing of files between users, several FEKs can be associated with a file.
- To retrieve data in case of lost passwords, data recovery agents can be identified by the admin. These are allowed to decrypt all data on an EFS filesystem, in effect giving them a master key for all data.
- Several short-comings have been identified with EFS:
  - Only contents of files, but not meta-data, is encrypted.
  - Encryption is only enabled on EFS filesystems, so copying data between filesystem may lead to inadvertently decrypting the data.
  - Contents may be exposed through temporary files, stored in an un-encrypted filesystem.
  - Since the private key is stored on disk in a hashed, salted form, gaining access to the user's password will also expose his private key.
  - Gaining access to the accounts of any data recovery agent will also expose all encrypted data.

Full-disk encryption

Rather than encoding individual files and folders, it is also possible to encrypt an entire disk or partition.

Three commonly used full-disk filesystem encryption tools are:

- BitLocker (Windows; closed source); supports AES etc
- CipherShed, formerly TrueCrypt (Windows, Linux, Mac; open source); supports AES etc
- LUKS (Linux; open source); supports AES etc

These tools encrypt either an entire disk, a partition (logical disk), or a container file, which can then be mounted as a logical disk and will appear as a separate partition (Linux) or as a separate volume with drive letter (Windows).

Full-disk encryption: CipherShed

CipherShed, formerly TrueCrypt, is an open-source utility for full-disk filesystem encryption:

- The password used to encrypt a partition must be entered when mounting the partition.
- TrueCrypt also supports **plausible deniability**, which means it can be configured to hide even the existence of encrypted data.
- This is supported by hidden, encrypted volumes, which are stored in the free space of another TrueCrypt encrypted volume.
- Since TrueCrypt initialises unused space on an encrypted partition with random data, a hidden, inner encrypted volume is indistinguishable from such random data.
- The weakness with hidden volumes is that opther applications may leave a trace of, e.g. recently opened files, which will point an adversary to the existence of a hidden volume.
- For details see: `https://ciphershed.org/`

Full-disk encrption: LUKS

Linux Unified Key Setup (LUKS) is the Linux standard for full-disk encrption.

- It is supported by all recent Linux kernels.
- It is accessed by the `dm-crypt` package that supports tools for creating, mounting and unmounting encrypted partitions.
- The main tool to perform encryption is `cryptsetup`.
- A detailed discussion on LUKS usage is given on the Linux Arch Wiki: `https://wiki.archlinux.org/index.php/dm-crypt_with_LUKS`
- The main web page for the `dm-crypt` package is: `http://code.google.com/p/cryptsetup/`

Example LUKS Usage

Sample session of creating and opening a file as a LUKS partition

```
# testing installation
> rpm -qf `which cryptsetup`
cryptsetup-1.4.1-1.mga2
# generate an empty file, that can be used as a container
> dd if=/dev/zero of=verysecret.loop bs=52428800 count=1
# connect that file with loop-back device /dev/loop0 to use it like a p
> losetup /dev/loop0 verysecret.loop
# format this partition using a 256-bit AES key in CBC mode (this will
> cryptsetup -c aes-cbc-essiv:sha256 -y -s 256 luksFormat /dev/loop0
# open the partition (this will ask for the password)
> cryptsetup luksOpen /dev/loop0 verysecret
# now, create a file-system on that partition
> mkfs.xfs /dev/mapper/verysecret
# finally, mount the partition to dir /mnt/t
mount /dev/mapper/verysecret /mnt/t
```

Example LUKS Usage (cont'd)

Sample session of closing a LUKS partition

```
# to close all files, first unmount the partition
> umount /mnt/t
# then close the encrypted partition
> cryptsetup luksClose verysecret
# and dis-connect the loop-back device from the file
> losetup -d /dev/loop0
# check that there are no connected devices left
> losetup -a

# to add a key, for opening a device
> cryptsetup luksAddKey verysecret
# to delete a key
> cryptsetup luksDelKey /dev/loop0 0
```


Full-disk filesystem encrption: BitLocker

BitLocker is another full-disk filesystem encryption utility, specific to Windows with NTFS partitions (and closed source).

- Two partitions are used: one with operating system and user data, the other is an unecrypted boot volume.
- At boot time the user needs to authenticate, and at this time will unlock the volume master key.
- It supports AES and other symmetric encryption methods for encrypting the bulk of the data.
- Another way to unlock the data is to use a **trusted platform module (TPM)**, such as a USB stick.

Trusted Platform Module (TPM)

Trusted Platform Module (TPM) is a crypto-co-processor chip, with a unique RSA private key, that can generate and store cryptopgraphic keys.

- Platform configuration registers (PCRs) are used to store keys and ciphertexts for several cryptographic operations.
- It provides functionality for cryptographic hashes, for sealing (encrypting) and unsealing (decrypting).
- Sealing and unsealing are tied to the device and will not work on other devices with a different private key.
- This technology can be used to bind secret data to the TPM that can only be extracted if the state of the device is identical to the state when it was generated.
- A concrete application is to test the integrity of trusted OS components, before decrypting and enabling them.
- This technology can also be used for digital-rights management and software licencing.
- TPM is being pushed by Microsoft to strengthen software lock-in.

Steganography

Steganography is the science of writing hidden message within unsuspicious information.

- Intuitively, steganography "hides messages in plain sight".
- A concrete example is to use invisible ink to add text to, e.g. a letter containing non-confidential information.
- In computer science, there are several **covert channels** that can be used to hide information.
- For example, meta-data of files might contain unused byte, which will not normally be displayed.
- Another example is to embedd information within a picture, by using the lowest bit in each RGB value for the hidden text.
- In an 1024×768 picture, this allows to store ca 295kB of data, which is enough to store compressed versions of Hamlet, King Lear, Macbeth, The Merchant of Venice and Julius Caesar.
- For a practical example see: `http://www.cs.vu.nl/~ast/`.

Summary

- Access control mechanisms are used at different levels, to restrict access to resources
  - application level
  - operating system
  - hardware level
- Control on higher levels is more powerful but also more complex to manage and thus more vulnerable
- Effectively using the OS's mechanism can make all of your applications more secure
- Being aware of hardware protection mechanisms is important to understand potential attacks

Closing Remarks

- The main function of access control in computer operating systems is to limit the damage that can be done by particular groups, users, and programs whether through error or malice
- The general concepts of access control from read, write, and execute permissions to groups and roles will crop up again and again.
- An example comes from public key infrastructures, which are a reimplementation of an old access control concept, the capability.

Further Reading

- Michael T. Goodrich and Roberto Tamassia
  Introduction to Computer Security,
  Addison Wesley, 2011. ISBN-10: 0321512944
  Chapters 3 (O.S. Security) and Section 9.7 (Secure Storage).
- Ross Anderson, *"Security Engineering"*,
  John Wiley & Sons Ltd, 2001.
  On-line: `http://www.cl.cam.ac.uk/~rja14/book.html`.
  Chapter 4, 6.
- Dieter Gollmann. *Computer Security*.
  John Wiley & Sons, second edition, 2006.
- Andrew S. Tanenbaum. *Modern Operating Systems*
  Pearson Education; 3 edition; 2007. ISBN: 0138134596
  Chapter 9 (with the Steganography example in Section 9.3)