4 Case Study: UNIX
Some basic features of UNIX

- UNIX supports participants in
  - using their own workstation for their specific application tasks
  - cooperating with colleagues in server-based local networks for joint projects

- a participant can manage his own computing resources at his discretion,
  - either keeping them private
  - or making them available to other particular participants or to everybody

- security mechanisms
  - enforce the virtual isolation of identified, previously registered users
  - enable the deliberate sharing of resources

- the mechanisms are closely intertwined with the basic functional concepts of files and processes, which are managed by the UNIX kernel

- the kernel acts as controller and monitor of all security-relevant accesses
Basic blocks of control and monitoring (and cryptography)

- identification of registered users as participants
- passwords for user authentication at login time
- a one-way hash function for storing password data
- discretionary access rights concerning files as basic objects and three fundamental operational modes, read, write and execute
- owners, as autonomous grantors of access rights
- owners, groups and the full community of all users, as kinds of grantees
- right amplification for temporarily increasing the operational options of a user
- a superuser, capable of overriding the specifications of owners
- access control concerning the commands and the corresponding system calls
- monitoring of the functionality
- kernel-based implementation of control and monitoring
Conceptual design of the operating system functionality

- UNIX provides a *virtual machine* that offers an external *command* interface with the following fundamental features:
  - identified *participants* can
  - *master processes* that
  - *execute* programs
  - stored in *files*

- the processes, in turn, can operate on files, in particular for *reading* and *writing*
ER model of fundamental functional features and security concepts
Participants, sessions and system calls

- A previously *registered participant* can start a *session* by means of the *login* command.

- Thereby the *system* assigns a *physical device* for input and output data to him.

- Starts a *command interpreter* as the first process mastered by that participant.

- Afterwards, the participant can issue *commands*, which may possibly generate additional processes that are also mastered by him.

- The commands invoke *system calls* that serve for:
  - Process management
  - Signaling
  - File management
  - Directory and file system management
  - Protection
  - Time management
Processes as active subjects

- *execute* (the program contained in) a file, and in doing so
- *read* or *write* in (usually other) files
- *create* new files and *remove* existing ones
- *generate* new (child) processes
- have a *lifespan*,
  starting with the generation by a father process and
  ending with a synchronization with the pertinent father process
- constitute a *process tree*:
  - when the UNIX system is started, an initial process *init* is generated
  - an already running *(father) process* can generate new *(child) processes*
Lifespan of a process

father process
fork
(child) process
fork

(child) process is generated by the father process
exec
(child) process exchanges process space
wait
exit
(child) process synchronizes with father process and is ended
Growing and shrinking of a process tree

Initial process

- Child process for each physical device
- Child process for command interpreter

- By login
- By a command that creates a new process
- By a further command that creates a new process
- By logout
Files as passive objects

- files are uniformly managed by the system using a file tree
- a file is identified by its *path name* within the file tree
- a file that constitutes a branching node in the file tree is a *directory* listing other files
- a file that constitutes a leaf in the file tree is a *plain file* containing data, which might be considered as an executable program
Conceptual design of the security concepts

• a participant acts as the owner of the files created by him

• the system administrator assigns participants as members of a group:
  – a group comprises those participants that are entitled to share files
  – an owner can make a file available for a group to share it

• for each file, the owner implicitly specifies three disjoint participant classes:
  – himself as owner
  – the members of the pertinent group, except the owner if applicable
  – all other participants

• the owner of a file discretionarily declares access privileges
  for each of these classes – for the processes mastered –
  by permitting or prohibiting the operations
  belonging to an operational mode:
  – read
  – write
  – execute
Some operations with commands and their operational mode

<table>
<thead>
<tr>
<th>Operation with command on plain file</th>
<th>Operation with command on directory</th>
<th>Operational mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>open file for reading:</td>
<td>open directory for scanning:</td>
<td>read</td>
</tr>
<tr>
<td>open( ,o_rdonly)</td>
<td>opendir</td>
<td></td>
</tr>
<tr>
<td>read content:</td>
<td>read next entry:</td>
<td></td>
</tr>
<tr>
<td>read</td>
<td>readdir</td>
<td></td>
</tr>
<tr>
<td>open file for writing:</td>
<td></td>
<td>write</td>
</tr>
<tr>
<td>open( ,o_wronly)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>modify content:</td>
<td>insert entry:</td>
<td></td>
</tr>
<tr>
<td>write</td>
<td>add</td>
<td></td>
</tr>
<tr>
<td>delete content:</td>
<td>delete entry:</td>
<td></td>
</tr>
<tr>
<td>truncate</td>
<td>remove</td>
<td></td>
</tr>
<tr>
<td>execute content as program:</td>
<td>rename entry:</td>
<td></td>
</tr>
<tr>
<td>execute</td>
<td>rename</td>
<td></td>
</tr>
<tr>
<td></td>
<td>select as current directory:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>cd</td>
<td>execute</td>
</tr>
</tbody>
</table>
Mastership and group mastership

• normally, a user is the *master* of the command interpreter process that he has started, and of all its descendants

• additionally, the (primary) group of that user is said to be the *group master* of all those processes

• if a process requests an operation `op` on a file `file`, then the access privileges `file.access_privileges` are inspected according to the masterships of the process in order to take an access decision

• for each file, the owner can additionally set two *execution flags*, `suid` and `sgid`, that direct its usage as a program, or as a directory, respectively:
  – for a plain file containing an executable program, the flag impacts on the *mastership* of an executing process
  – for a directory, the flag impacts on the *ownership* of inserted files
Refined ER model of the functional features and security concepts
Refined ER model: users

- User
  - Username
  - Password
  - Full name
  - Home directory
  - Shell file
  - Surrogate
  - Primary member
  - UID
    - (User identification)
  - GID
    - (Group identification)

- Human individual
- Physical device
- Connected
- Member

/etc/passwd
/etc/shadow
/etc/groups
Refined ER model: files

uid (user identification)

- superuser_id

gid (group identification)

access privileges

- owner rwx
- group rwx
- other rwx

filename

- suid
- owned by
- available for

- group share

file

i_nodes
Refined ER model: processes
Different notions of a participant

• a human individual

• the physical device
  from which the individual issued his last login command

• an abstract user:
  – representing the previously registered human individual within the system:
    as a result of a successful login command,
    the abstract user is connected to the
    physical device from which the command was received
  – uniquely identified by a username
  – associated with further administrative data, e.g.:
    – password data
    – full name,
    – (the path name of) home directory in the overall file tree
    – (the path name of the file containing) command interpreter (shell file)

• a user identification, i.e., a cardinal number uid,
  which serves as a (not necessarily unique) surrogate for an abstract user
System administrator

- is a *human individual*, typically registered as a distinguished *abstract user* whose username is *root* and whose surrogate is *superuser_id* (in general, represented by 0)

- enjoys nearly unrestricted operational options (consequently, so does any human individual who succeeds in being related to *root*)
Groups

- a group is represented by a *group identification, gid*

- each abstract user is a *primary member* of one group, and can be a *member* of any further groups
Mastership and group mastership refined

* all relationships of files/processes with participants/groups are interpreted as relationships with *user identification/group identifications*

* the *master* and the *group master* relationships are further differentiated in order to enable dynamic modifications

* a user identification *uid* (the surrogate of a user connected to a physical device from which a human individual has issued a *login* command) is seen as the *original master* of the *command interpreter process* generated during the login procedure *and of all its descendants*

* these processes are also said to have this *uid* as their *real uid*

* correspondingly, a group identification *gid* is seen as the *original group master* of these processes, which are also said to have this *gid* as their *real gid*
Current masterships

- normally, the *original* masterships are intended to determine the access decision when a process requests an operation on a file

- to distinguish between normal and *exceptional* cases,
  - an additional *current mastership* (an *effective uid*) and
  - an additional *current group mastership* (an *effective gid*)
    are maintained and actually employed for access decisions

- the current mastership and the current group mastership of a process
  are automatically manipulated according
  to the execution flags *suid* and *sgid* of the executed file:
  - normally, if the respective flag is *not* set,
    then the *current mastership* is assigned the *original mastership*, and
    the *current group mastership* is assigned the *original group mastership*
  - exceptionally, if the respective flag is set,
    then the *current mastership* is assigned
    the *user identification of the owner of the file to be executed*, and
    the *current group mastership* is assigned
    the *group identification for which that file has been made available*
Right amplification

- the exceptional case is used for *right amplification*, to dynamically increase the operational options of a process while it is executing a file with a flag set

- the owner of that file allows all “participants” that are permitted to execute the file at all to act thereby as if they were the owner himself

- if the owner is more powerful than such a participant (e.g., if the owner is the nearly omnipotent abstract user *root*), then the operational options of the participant are temporarily increased

- the current masterships and current group masterships can also be manipulated by special, suitably protected commands

- for this option, the additional *saved mastership* and *saved group mastership* are used to restore the original situation
Identification and authentication

• a human individual can act as a participant of a UNIX installation only if the system administrator has *registered* him in advance as *user*, thereby assigning a *username* to him

• this assignment and further user-related data are stored in the files `/etc/passwd` and `/etc/shadow`

• the usernames serve for *identification* and for *accountability* of all actions

• whenever an individual submits a *login* command, the system
  – checks whether the username is *known* from a registration by inspecting the file `/etc/passwd`:
    if the username is found, it is considered as known, otherwise as unknown
  – evaluates whether the actual command is *authentic*, relying on:
    – appropriate registrations
    – the integrity of the underlying files
Proof of authenticity by a password procedure

- if the individual can input the agreed password, then the command is seen as authentic

- the system relies on
  - appropriate password agreements
  - the individual’s care in keeping his password secret
  - the integrity and confidentiality of the file /etc/shadow

- the confidentiality of this file is supported by several mechanisms:
  - passwords are not stored directly, but only their images under a one-way hash function
  - on any input of the password, the system immediately computes its hash value and compares that hash value with the stored value

- the hash values are stored in a specially protected file /etc/shadow:
  - a write access to an entry (password modification) is allowed only if the request stems from root or from the pertinent user
  - a read access to an entry is allowed only for authenticity evaluations
Access decisions

• the kernel has to take *access decisions* concerning
  – a *process* as an active subject
  – a *file* as a controlled passive object
  – a requested *operation*

• given a triple (process, file, operation),
  the kernel has to decide whether
  – the process identified by `process.current_master` is allowed
  – to actually execute the operation denoted by `operation`
  – on the file named `file`

• two cases according to the *effective user identification* of the process,
  `process.current_master`:
  – if `process.current_master = superuser_uid`,
    then nearly everything is considered to be allowed
  – otherwise, a decision procedure is called
Access decisions regarding normal users

function decide(process, file, operation): Boolean;

if process.current_master = file.owner
then return file.access_privileges.owner.mode(operation)

else
    if process.current_groupmaster = file.group
    OR EXISTS process.supplementary_groupmaster:
        process.supplementary_groupmaster = file.group
    then return file.access_privileges.group.mode(operation)

else return file.access_privileges.other.mode(operation)
Knowledge base on permitted operational options

- implemented by means of the fundamental functional features of UNIX

- data about *users* and *groups* is stored in the special files
  - `/etc/passwd`
  - `/etc/shadow`
  - `/etc/group`

- these files are owned by the system administrator (under `superuser_id`)

- the access privileges for these files are given by
  - `r--|r--|r--`
  - `rw-|---|---`
  - `r--|---|---`

- additionally, modifications of the files `/etc/passwd` and `/etc/group` are specially restricted to processes with the effective `uid` `superuser_id`

- security relevant data about *files* is managed in *i-nodes*

- security-relevant data about *processes* is maintained in the *process table*
Main entries of the administration files for users and groups

<table>
<thead>
<tr>
<th>/etc/passwd</th>
<th>/etc/shadow</th>
<th>/etc/group</th>
</tr>
</thead>
<tbody>
<tr>
<td>username</td>
<td>username</td>
<td>groupname</td>
</tr>
<tr>
<td>reference to /etc/shadow</td>
<td>hash value of password</td>
<td>group password</td>
</tr>
<tr>
<td>user identification (uid)</td>
<td>date of last modification</td>
<td>group identification (gid)</td>
</tr>
<tr>
<td>gid of primary group</td>
<td>maximum lifetime</td>
<td>usernames of members</td>
</tr>
<tr>
<td>full name, comment</td>
<td>date of expiration</td>
<td></td>
</tr>
<tr>
<td>path name of home directory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>path name of shell file</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Modifications of the knowledge base: user and group administration

- the commands `useradd`, `usermod` and `userdel` manipulate the entries for *users* in the files `/etc/passwd`, `/etc/shadow` and `/etc/group`:
  - only executed for a process whose effective user identification is `superuser_uid`

- the commands `groupadd`, `groupmod` and `groupdel` manipulate the entries for *groups* in the file `/etc/group`:
  - only executed for a process whose effective user identification is `superuser_uid`
Modifications of the knowledge base: password management

• the command `passwd` modifies an entry of a user in the file `/etc/shadow`:
  
  only executed for a process whose effective user identification is
  
  – `superuser_uid`
  or
  
  – equal to the user identification of the user whose password is requested to be changed
Modifications of the knowledge base: login procedure

- the command `login` tries to identify and authenticate the issuer

- on success, the issuer is recognized as a known registered user

- by a system call `fork`, a new process is generated for that user

- that process, by use of a system call `exec`, starts executing the shell file of the user as a command interpreter

- the masterships and group masterships are determined as follows:
  - the real `uid`, effective `uid` and `saved uid` are all assigned the user identification of the user, i.e., `user.surrogate`
  - the real `gid`, effective `gid` and `saved gid` are all assigned the primary group of the user, i.e., `user.primary_member`
  - the supplementary `gid` is assigned the set of elements of `user.member`

- subsequently, this process is treated as the original ancestor of all processes that are generated during the session started by the `login` command
Modifications of the knowledge base: mastership assignments

- normally,
  a process inherits its masterships and group masterships from its immediate ancestor

- exceptionally,
  masterships and group masterships are determined differently, namely if
  - the file executed has an execution flag `suid` or `sgid` set,
  or
  - some explicit command modifies the implicit assignment
Modifications of the knowledge base: file management

- the system call
  `create(filename, access_privileges, suid, sgid)`
  creates a new file

- the owner and the group share of the file are assigned
  the effective uid and the effective gid, respectively,
  of the creating process

- the access privileges and
  the execution flags suid and sgid are assigned
  according to the respective parameters of the call,
  possibly modified according to the mask umask
Modifications of the knowledge base: masking access privileges

- the mask `umask` specifies nine truth values, one for each value contained in the parameter for the access privileges:
  - each mask value is complemented
  - the conjunction with the corresponding parameter value is taken

- a mask value `true` (or `1`) is complemented into `false` (or `0`) and thus always results in the corresponding access privilege being set to `false` (or `0`), thereby expressing a prohibition

- in general, individuals are strongly recommended to prohibit write access to files with an execution flag `suid` or `sgid` set: avoids unintended/malicious modification of the program contained, resulting in unwanted effects of right amplification

- the system call `umask(new_umask)` modifies the current nine truth values of the mask `umask` into the values specified by the parameter `new_umask`
Modifications of the knowledge base: process management

- the system call `fork` generates a new process

- a subsequent system call `exec(command_file)` exchanges the content of its address space, thereby loading the program that is contained in the file specified as the parameter `command_file`, whose instructions are then executed

- masterships, group masterships and the mask `umask` of that process:
  - if the flags `suid` and `sgid` of the file `command_file` are *not* set, then the new process inherits all masterships and group masterships from its father process
  - if the flag `suid` is set, then the *effective* `uid` and the *saved* `uid` are assigned to `command_file.owner`
  - if the flag `sgid` is set, then the *effective* `gid` and the *saved* `gid` are assigned to `command_file.group` share
  - the mask `umask` is inherited from the father process
Modifications of the knowledge base: execution flags

- The system call `setuid(uid)` assigns the masterships `real uid`, `effective uid` and `saved uid` the parameter value `uid`:
  - Only executed for a process that satisfies the following precondition:
    - The `effective uid` equals `superuser_uid`,
    - Or the `real uid` equals the parameter value `uid`
  (i.e., in the latter case, the original situation is restored)

- The system call `seteuid(euid)` assigns the current mastership `effective uid` the parameter value `euid`, which might be the `real uid` or the `saved uid`

- Thereby, while executing a file with the execution flag `suid` set, a process can repeatedly change its `effective uid`:
  - The process can select the `uid` of that user who has generated the original ancestor, or the `uid` of the owner of the file executed
Modifications of the knowledge base: some further manipulations

- the system calls `setgid(gid)` and `setegid(egid)` manipulate the group masterships

- the command `/bin/su -` changes the effective uid of the currently executed process into `superuser_uid` (thus the system administrator can acquire the mastership of any process): only executed if the issuer is successfully authenticated with the agreed password for the system administrator with username `root`

- the command `chown` changes the owner of a file: only executed for a process that satisfies the following precondition: the effective uid equals `superuser_uid` or equals the current owner of the file

- the command `chmod` changes the access privileges of a file: only executed for a process that satisfies the following precondition: the effective uid equals `superuser_uid` or equals the current owner of the file
Knowledge base on usage history

- basically, UNIX does not maintain an explicit *knowledge base* on the *usage history* for taking *access decisions*, except for keeping track of process generations

- most UNIX versions offer log services for *monitoring* that
  - produce *log data* about issued commands and executed system calls
  - store that data in special *log files*
Examples of UNIX log files

- the file `lastlog` contains the date of the last issuing of a `login` command for each of the registered users, whether successful or failed

- the file `loginlog` contains entries about all failed issuings of a `login` command, comprising the username employed, the physical device used and the date

- the file `pacct` contains entries about all issued commands, including their date
Examples of UNIX log files, continued

- **the file sulog contains**
  entries about all successful or failed attempts to issue the critical `su` command;
  for each attempt, the following is recorded:
  - success or failure
  - the username employed
  - the physical device used
  - the date

- **the files utmp or wtmp contain**
  entries about the currently active participants;
  in particular, the following is recorded:
  - the username employed
  - the physical device used
  - the process identification of the original ancestor process
    that was started by the `login` command
    to execute the user’s command interpreter
Audit services

• log services send their log data as *audit messages* to an audit service that unifies and prepares that data for persistent storage or further monitoring

• the audit service *syslog* works on audit messages that are sent
  – by the kernel, exploiting `/dev/klog`
  – by user processes, exploiting `/dev/log`
  – by network services, exploiting the UDP port 514

• the audit messages consist of four entries:
  – the name of the *program* whose execution generated the message
  – a *classification* of the executing process into one of a restricted number of event sources, called *facilities*, which are known as *kern*, *user*, *mail*, *lpr*, *auth*, *daemon*, *news*, *uucp*, *local0*, …, *local7*, *mark*
  – a *priority level*, which is one of *emerg*ency, *alert*, *crit*ical, *err*or, *warning*, *notice*, *info*rmational, *(from) debug*ging, *none*
  – the actual notification of the *action* that has occurred
Configuration of an audit service: example

- The system administrator can configure the audit service `syslog` using the file `/etc/syslog.conf`, which contains expressions of the form `facility.priority destination`.

- Such an expression determines how an audit message
  - that stems from an event source classified as `facility` and
  - has the level `priority` should be treated, i.e.,
  - to which `destination` it has to be forwarded

- `destination` might denote
  - the path name of a file
  - a username,
  - a remote address,
  - a pipe
  - the wildcard `*` (standing for all possible receivers)
Overall architecture

- control and monitoring are part of the operating system kernel
- the *kernel* realizes the system calls offered by UNIX
- a *system call* is treated roughly as follows:
  - the kernel checks the operator and the parameters of the call and then deposits these items in dedicated registers or storage cells
  - a software interrupt or trap dispenses the calling process
  - the program determined by the specified operator is executed with the specified parameters
  - if applicable, return values for the calling process are deposited
  - subsequently, the calling process can be resumed
- this procedure needs special hardware support for security: *storage protection, processor states, privileged instructions, process space separation, ...*
- most UNIX installations are part of a *network*, and thus employ various features for *securing the connections* to remote participants and the interactions with them