Capability-Based Access Control

1 An Analogy: Bank Analogy

We would like to use an example to illustrate the need for capabilities. In the following bank example, we will discuss two access control mechanisms: access control list (ACL) and capability. We will compare the pros and cons of these two different mechanisms.

Example: Alice wishes to keep all of her valuables in three safe deposit boxes in the bank. Occasionally, she would like one or more trustworthy friends to make deposits or withdrawals for her. There are two ways that the bank can control access to the box.

- The bank maintains a list of people authorized to access each box.
- The bank issues Carla one or more keys to each of the safe deposit boxes.
- The ACL Approach
  - Authentication: The bank must authenticate.
  - Bank's involvement: The bank must (i) store the list, (ii) verify users.
  - Forging access right: The bank must safeguard the list.
  - Add a new person: The owner must visit the bank.
  - Delegation: A friend cannot extend his or her privilege to someone else.
  - Revocation: If a friend becomes untrustworthy, the owner can remove his/her name.
- Capability Approach
  - Authentication: The bank does not need to authenticate.
  - Bank's involvement: The bank need not be involved in any transactions
  - Forging access right: The key cannot be forged
  - Adding a new person: The owner can give the key to other people
  - Delegation: A friend can extend his or her privilege to someone else.
  - Revocation: The owner can ask for the key back, but it may not be possible to know whether or not the friend has made a copy.
- Alice in a hostile environment
  - Alice does have a social life, and she often go to bars with her friends, some of which might be evil. Therefore, Alice can get drunk; when people get drunk, they might do things or make mistakes that they regret to do. Which approach (ACL or Capability) is better to deal with this situation?
    A: With the capability approach, Alice can choose not to carry the keys with her when she goes to drink. This way, even if she get drunk, she cannot open the safe deposit box. In the ACL approach, there is no such kind of protection. This kind of protection by the capability approach exemplifies the least-privilege principle.

  - Alice often sends her employees to carry out tasks for her. These tasks involve going to the bank several times, opening several deposit boxes. However, the outside environment is quite hostile, the employees might be kidnapped at any point of time. Kidnaper can then force the employees to retrieve the valuables from the deposit boxes. Most employees will not resist if kidnapped. Which access control approach can better protect Alice's valuable properties?
    A: With the capability approach, employees can destroy the keys that will not be needed by the on-going tasks (Alice still has a copy of all the keys). This way, even if the employees are kidnapped, the damage can be reduced to the minimum. This kind of protection is difficult to achieve by the ACL approach.

2 Capability Concept

- The capability concept was introduced by Dennis and Van Horn in 1966.

  > ”A capability is a token, ticket, or key that gives the possessor permission to access an entity or object in a computer system”.

- Intuitive examples
  - A movie ticket is a capability to watch a movie.
  - A key is a capability to enter a house.
- A capability is implemented as a data structure that contains:
  - *Identifier:* addresses or names. e.g. a segment of memory, an array, a file, a printer, or a message port.
  - *Access right:* read, write, execute, access, etc.
- Using capabilities
  - **Explicit use:** you have to show your capabilities explicitly. This is what we do when we go to movie theaters: we show the doorkeeper our tickets. The following is another example that is quite common when a program tries to access a file:

    ```
    PUT (file_capability, "this is a record");
    ```

  - **Implicit use:** there is no need to show the capabilities, but the system will automatically check whether one has the proper capabilities. An analogy to this would be having the theater doorkeeper to search your pockets for the right tickets. Although the approach is awkward in this analogy, it is quite common in operating systems. The capability-list method basically uses this approach. Namely, each process carries a list of capabilities; when it tries to access an object, the access control system checks this list to see whether the process has the right capability. Unlike the explicit use approach, with this approach, processes (or the programmers who write the programs) do not need to figure out which capability should be presented to the system.
  - *Comparison:* The implicit approach is less efficient, especially, when the capability-list is long. However, it might be easier to use, because, unlike the explicit approach, capabilities are transparent to users; therefore, users do not need to be aware of the capabilities.

- The identifier of capability can be many things, including users, processes, procedures, and programs:
  - Capability on Users: Users are more persistent identifiers. Its capability can be stored in files.
  - Capability on Processes: Processes are not persistent identifiers, they usually obtain capabilities dynamically.
  - Capability on Procedures: (1) Caller and callee can have different capabilities (2) Most capability systems go a step further: allow each procedure to have a private capability list.
  - Capability on Programs.
    - Giving capabilities to programs can achieve privilege escalation and downgrading.
    - Example: Privileges in Trusted Solaris (see the case study on Trusted Solaris).
    - Set-UID programs are a special case: Set-UID programs have the root capability.
- Examples of capabilities implemented in LIDS (Linux Intrusion Detection System)
  - `CAP_CHOWN:` override the restriction of changing file ownership and group ownership.
  - `CAP_DAC_READ_SEARCH:` override all DAC restrictions regarding read and search on files and directories.
  - `CAP_KILL:` the capability to kill any process.
  - `CAP_NET_RAW:` the capability to use RAW sockets
  - `CAP_SYS_BOOT:` the capability to reboot.

3 Capability Implementation

- *Where should capabilities be stored?* Capabilities are critical to system security. Once a capability is issued to a user, the user should not be able to tamper with the capability.
  - **In a protected place:** Capabilities can be stored in a protected place. Users cannot touch the capability; they use capabilities in an implicit manner:
    - In kernel: this approach is adopted by the Capability-list approach, in which, the capability list is stored in the kernel (e.g. in the process data structure). Users cannot modify the contents of any capability, because they have no access to the kernel. Whenever users need their capabilities, the system will go to the kernel to the capability-list.
    - Tagged architecture: the capability can be saved in memories that are tagged as read-only and use-only.
  - **In an unprotected place:** In some applications, users may have to carry their capabilities with themselves. When they request an access, they simply present their capability to the system. This is an explicit use of capabilities.
    - How to prevent users from tampering with the capability? Because permissions are encoded in the capability, if users can tamper with the contents of a capability, they can gain unauthorized privileges.
    - The protection can be achieved using cryptographic checksum: the capability issuer can put a cryptographic checksum on the capability (e.g. digital signature). Any tampering of the capability will be detected.

    - ∗ This approach is widely used in distributed computing environments, where capabilities need to be carried from one computer to another; therefore, relying on kernel to protect capabilities is infeasible.
  - **Hybrid Approach:** users can use capabilities in an explicit manner, but the capabilities are stored in a safe place.
    - ∗ The real capabilities are stored in a table, which resides in a protected place (e.g. kernel).
    - ∗ Users are given the index to these capabilities. They can present the index to the system to explicitly use a capability.
    - ∗ Forging an index by users does not grant the users with any extra capability.

- Basic Operations on Capabilities:
  - **Create** capability: a capability is created for a user (or assign to a user).
  - **Delegate** capability: a subject delegates its capability to other subjects. There are many interesting features related to delegation:
    - ∗ Expiration time: specify the lifetime of a delegated capability.
    - ∗ Propagation control: specify whether the users who get a capability via delegation can further delegate the capability.
  - **Revoke** capability: a subject revokes the capabilities it has delegated to other subjects. The implementation of revocation in general is a difficult problem. The followings are two common revocation schemes:
    - ∗ Approach 1: Have each capability point to an indirect object. When revoking a capability, we can simply delete the indirect object.
    - ∗ Approach 2: Use a random number. The owner can change the number. A user must also present the number in addition to the capability. (used in Amoeba)
    - ∗ The above two approaches do not allow selective revocation.
    - ∗ Attach an expiration time to a delegated capability can achieve automatic revocation.
  - **Enable** capability: a subject enables a disabled capability.
  - **Disable** capability: a subject *temporarily* disables a capability.
  - **Delete** capability: a subject *permanently* deletes a capability.
    - ∗ It should be noted that *disabling* a capability is different from *deleting* a capability. They are both useful to achieve the least-privilege principle.
    - ∗ When a capability will not be needed anymore by a task (e.g. a process), this capability should be permanently *removed* from from the task. This way, even if the task is compromised to execute malicious code, the code cannot use the capability.
    - ∗ When a capability will still be needed later, but will not be needed by a subtask (e.g. a procedure within a process), the capability should be *disabled*. When the capability is needed, it can be enabled. It should be noted, if the task is compromised to execute malicious code, disabling capabilities does not help at all, because the malicious code can enable the capability. However, if the task is compromised through other ways (i.e., no malicious code is executed), disabling capabilities can reduce damage.

```
/* (in src/fs/fproc.h) */
struct fproc  {                    /* Process Table */
   ......
   struct filp *fp_filp[OPEN_MAX];  /* the file descriptor table */
}

struct filp {           /* Filp Table */
   mode_t file_mode;    /*RW bits, telling how file is opened */
   int filp_flags;
   int filp_count;   /* how many file descriptors share this slot? */
   struct inode *filp_ino   /* pointer to the inode table */
   off_t filp_pos;
}
```

Figure 1: File Descriptor Table Data Structure

4 Case Study: Using Capabilities for File Access

It is widely known that most `Unix` operating systems use Access Control List (ACL) as their basic access control mechanism; however it is less well known that the capability concept has also been used in most `Unix` operating systems for a long time. If you do not believe this, look at the following program (the program is executed by a normal user):

```
1:   f = open("/etc/passwd", "r");
2:   read(f, buf, 10);
3:   write(f, buf, 10);

     /* Before the following statement is executed, the root modifies
        the permission on /etc/passwd to 600, i.e., normal users cannot
        read this file any more.  */
4:   read(f, buf, 10);
```

Because the `/etc/passwd` file has a permission 644, normal users can open the file for read. So the statement in Line 1 is successful. This access control decision is based on ACL. Now look at Line 2 and 3. We know that Line 2 will succeed, but Line 3 will fail; obviously, there is an access control on both `read` and `write`. Is the access control based on ACL? If your answer is yes, then answer the following question: *will Line 4 succeed or fail?* Line 4 is carried out after the access control list on the `passwd` file is modified. If the access control in Line 3 is based on ACL, the `read` operation should fail. However, interestingly, the program can still read from the `passwd` file. There is one logic conclusion we can make: the access control decision for `read` are not based on ACL. Then what is it based on? It is actually based on capability.

- File descriptor is an application of capability. When a file is open, a file descriptor is created and stored in the `filp` table (Figure 1). Each process has a `filp` table, which is stored in the kernel space (protected). The user-space application is given the index of the file descriptor (we often call this index the file descriptor, but actually, it is just an index to the real descriptor).

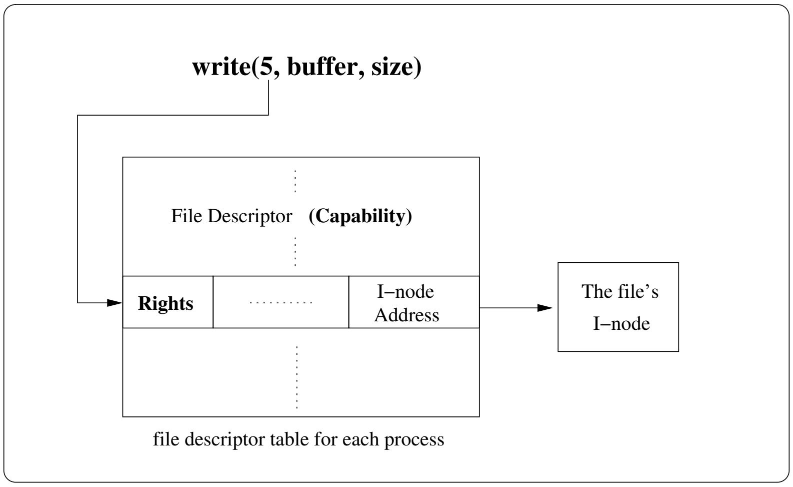

write(5, buffer, size)
File Descriptor (Capability)
Rights
I–node Address
The file's I–node
file descriptor table for each process

Figure 2: File Descriptor Table

- The `filp` table is actually a capability list. It contains a list of file descriptors. Each file descriptor contains a permission part that describes what the process can do to this file; the file descriptor also contains an identifier, which is the address of the file's `I-node` (Figure 2).
- Basic capability operations:
  - *Create capability:* a capability is created via the `open()` system call. Whether a process is allowed to create a capability depends on another access control mechanism, the Access Control List (ACL). Namely, the ACL of the file will be checked to decide whether the process can open this file. If yes, a capability will be created. This interesting example demonstrates one type of coordination between ACL and capability.
  - *Delete capability:* a capability is deleted via the `close()`. This system call will remove the corresponding capability from the `filp` table.
  - The other operations, such as delegation, revocation, enabling, and disabling, are not supported.
- Questions and Answers:
  - **Q**: Can one forge a capability? i.e., can one access a file without the legitimate capability?
    A: No. The corresponding `I-node` entry must be in the table.
  - **Q**: Can we directly use `fp_filp[10]`?
    A: There is no use if `fp_filp[10]` is empty. Users cannot modify `filp` table, because the table is stored in the kernel space.
  - **Q**: After a process opens a file (permission is 744, it is owned by root), the file's permission is changed to 700 by the root; can this process still be able to read the file?
    A: Yes, as long as the process uses the file descriptor to access the file. The file descriptor is a capability that allows the process to access the file, even after the ACL of the file changes.

5 Case Study: Using Capabilities for Memory Access

- A process should only be able to access its own memory, and the access must be authorized. This access control is implemented mostly using capability (See Figure 3).

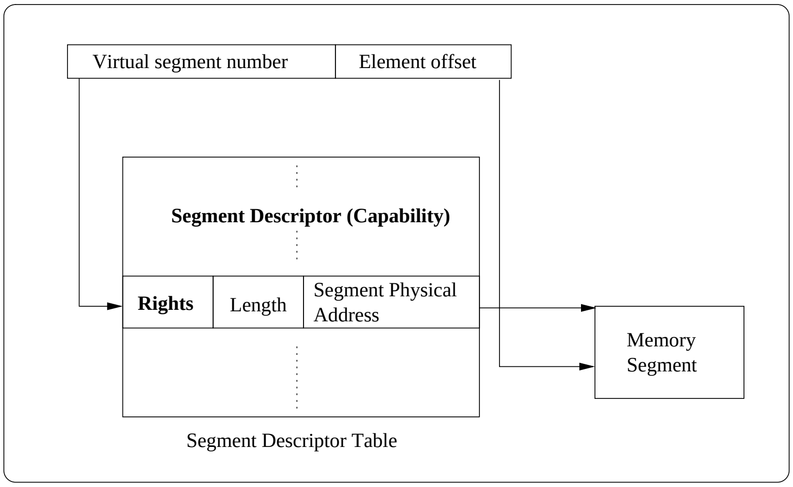


Figure 3: Conventional Segment Address Translation

- The segment descriptor table is a capability list, which contains all the memory segments that the process can access. The segment table can only be set by the system, not by users. Each descriptor specifies a block of memory that can be accessed using this capability; it also specifies the process's permissions on this block memory (read, write, and executable).
- When a process tries to access a memory, the address provided by the process is treated as a virtual address. A virtual address contains an index that points to the capability in the segment descriptor table. Using the capability, the system will first make sure that the process has a right to access the memory; it then computes the real physical address of the memory.
- This entire process is carried out by hardware support. Otherwise, you can imagine how many CPU cycles it will cost for a single memory access. The 80x86 protection mode also uses this approach. We will discuss this protection mode in more details in the later part of this course.

- Some capability-based system consists of a set of capability registers.
  - Programs can execute hardware instructions to transfer capabilities between the capability list and the capability registers.
  - Only the capabilities contained in the capability registers are effective. This way, a process can restrict its own capabilities to achieve the least privilege principle.
  - Another benefit is the performance. Capabilities in registers can be processed faster than those stored in memory.

6 Case Study: Privileged Programs using Capabilities

Oftentimes, users need a special privilege to carry out a task (e.g. changing their passwords). In a capability-based system, privileges are often represented by capabilities. Namely, to carry out the task, the users need some special capabilities. However, it is insecure to grant the users such capabilities, because they might use the privileges on some other tasks. It is desirable if we can ensure that the users only get the capabilities while carrying out the intended task; the capabilities will be taken back from the users once the task is finished.

This objective is similar to what the `Set-UID` programs are trying to achieve. The basic idea is to assign the privileges to programs, not to users. Users gain the privileges if they run this program; they will lose the privileges if the program finishes. We will study how such mechanism can be implemented in a capability-based system.

- When a privileged program is executed, the capabilities will be effective. For example, if a program has a file-reading capability, it can read all files even if the user who runs the program is not superuser.
- Where do we store capabilities for programs?
  - Store the capabilities in a configuration file, such as `/etc/cap.conf`. When system bootup, configuration in this file will be read in kernel and saved in a capability array (an approach used by `LIDS` of `Linux`).
  - Store the capabilities in the program's `I-node`. When a process is created to run the program, the process will be initialized with the program's capabilities.
- We should be very careful when writing these privileged programs. The privileged program must contain the capability within the program, so users can only use the capability for the actions intended by the program. If there is a flaw in the program, users might be able to escape the containment with the capability. Without the containment, the users can use the capability for actions that are not intended by the program. Consequently, security breaches can happen.

7 Case Study: Capabilities in Linux

Starting with kernel 2.2, Linux divides the root privileges into smaller privileges, known as capabilities. Capabilities are a per-thread attribute, and they can be independently enabled and disabled.

- **Capabilities.** The number of capabilities has been changing from version to version. We list some examples here.
  - `CAP_CHOWN:` Make arbitrary changes to file UIDs and GIDs.
  - `CAP_DAC_OVERRIDE:` Bypass file read, write, and execute permission checks. (DAC is an abbreviation of ”discretionary access control”.)
  - `CAP_DAC_READ_SEARCH:` Bypass file read permission checks and directory read and execute permission checks.
  - `CAP_NET_ADMIN:` Perform various network-related operations (e.g., setting privileged socket options, enabling multicasting, interface configuration, modifying routing tables).
  - `CAP_NET_RAW:` Use RAW and PACKET sockets.
  - `CAP_SYS_PTRACE:` Trace arbitrary processes using `ptrace(2)`.
- **Thread Capability Sets.** Each thread has three capability sets containing zero or more of the above capabilities.
  - *Permitted Set:* This is the set of capability that a thread have.
  - *Effective Set:* This is the set of capabilities that are currently effective in the process, i.e. the access control will use this set of capabilities.

  - *Inheritable Set:* This is a set of capabilities preserved across an `execve(2)`. It provides a mechanism for a process to assign capabilities to the permitted set of the new program during an `execve(2)`.

- **File Capabilities.** Since kernel 2.6.24, the kernel supports associating capability sets with an executable file using `setcap(8)`. To do so, one needs the `CAP_SETFCAP` capability. The file capability sets, in conjunction with the capability sets of the thread, determine the capabilities of a thread after an `execve(2)`.

  - When a thread executes a program with file capabilities, the thread can get extra privileges. Therefore, programs with file capabilities are privileged programs.
  - We can replace `Set-UID` programs using file capabilities, i.e., instead of giving a privilege program the root privilege, we can assign a set of required capabilities to the program. This way, we can enforce the *principle of least privilege*.

- **Capability Bounding.** In Linux kernels 2.2.11 and later, system administrators can bound the capabilities allowed on the system. They can remove capabilities from a running system. Once a capability has been removed, it cannot be added back again, until the system reboots. This can limit the damage for some systems even if the root has been compromised.

- The `libcap` Library. There are several ways for user-level programs to interact with the capability features in `Linux`, such as setting/getting thread capabilities, setting/getting file capabilities, etc. The most convenient way is to use the `libcap` library, which is now the standard library for the capability-related programming.

- Capability Operations: using the `libcap` library, we can implement the following functionalities related to capabilities:

  - *Disabling capabilities:* only temporarily disable certain capabilities, i.e., remove the capabilities from the *effective set*; the capabilities are still in the *permitted set*, and can be enabled later.
  - *Enabling capabilities:* if a capability is in the *permitted set*, it can be enabled and thus becomes effective.
  - *Deleting capabilities:* if a capability is no longer needed, it can be permanently removed from the `permitted set`.

8 Case Study: Capabilities in Trusted Solaris

- Privilege: use capability to escalating an applications privilege.

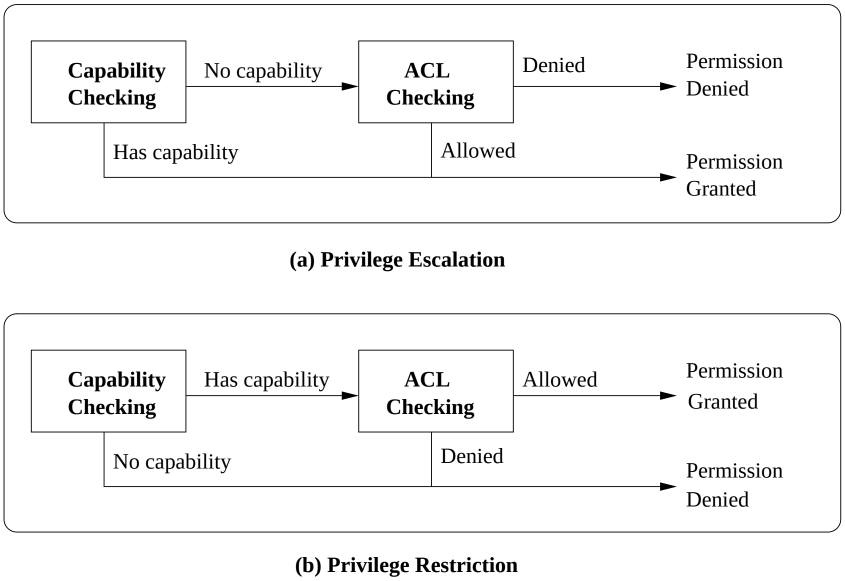
  - Privilege (i.e. capability): a discrete right granted to an application to perform an operation that would otherwise be prohibited.
    - Overriding other security policies, such as ACL.
    - Difference between capability and this privilege concept: capability is usually granted to a subject (e.g. user and process), while the privilege in Trusted Solaris is granted to an application (e.g. program).
  - Trusted Solaris 8 provides more than 70 privileges.
    - File System security: overriding ACLs, etc.

    - Network security: overriding restrictions on ports, etc.
    - Process security: overriding restrictions on auditing, ownership, user IDS, etc.
  - These privileges avoid granting an application the root privileges or the Set-UID bit.
  - "Privilege" is a more general way to escalate an application's privilege.
    - Set-UID always escalates the privilege to "root".
    - "Privilege" divide "root" to 70 sub-privileges, and escalate the privilege of an application to a sub-set of these privileges.

- Authorizations:
  - An authorization is a discrete right granted to a user or role.
  - Authorizations are capabilities.
  - Examples:
    - `solaris.admin.usermgr.read:` read but not write access to user configuration files.
    - `solaris.admin.usermgr.pswd:` change a user's password.
    - `solaris.admin.printer.delete:` delete a printer.
    - `solaris.admin.usermgr.grant:` delegate any of the authorizations with the same prefix to other users.
  - Authorizations are stored at `/etc/security/auth_attr`. Currently, it is impossible to add new authorizations.

9 Comparison of Capability and ACL

- Naming objects:
  - ACL: can attempt to name any object in the system as the target of an operation.
    - Pros: The set of the accessible objects is not bounded.
    - Cons: Worm, virus, backdoor, stack buffer overflow.
  - Capability: a user can only name those objects for which a capability is held.
    - Pros: the least privilege principle
    - Cons: the set of the accessible objects is bounded.
- Granularity:
  - ACL is based on users.
  - Capabilities can be based on process, procedure, programs, and users.
  - Finer granularity $\Longrightarrow$ the principle of least privilege.



(a) Privilege Escalation

(b) Privilege Restriction

Figure 4: Privilege Restriction and Escalation

10 Combining Capability with ACL

- Privilege Escalation (see Figure 4(a)):
  After a program gets certain capabilities, a user's privilege is escalated when he runs the program. This is like `Set-UID`, which only has one capability: the root capability. A general capability framework can define multiple capabilities. A program is only granted the privileges that are necessary. Therefore, no program will have a "superpower" like `Set-UID` programs. This is what Trusted Solaris 8 does.

- Privilege Restriction (see Figure 4(b)):
  User can further restrict a program's privilege. For example, if a program does not need to write to any file, the user can remove the file-writing capability from this program. Therefore, if the program is compromised, and tries to write to the user's files, the access will be denied even though it is allowed by ACL.